ModelArts

Model Training

Issue	01
Date	2026-06-04

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

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

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

Huawei Cloud Computing Technologies Co., Ltd.

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

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

Contents

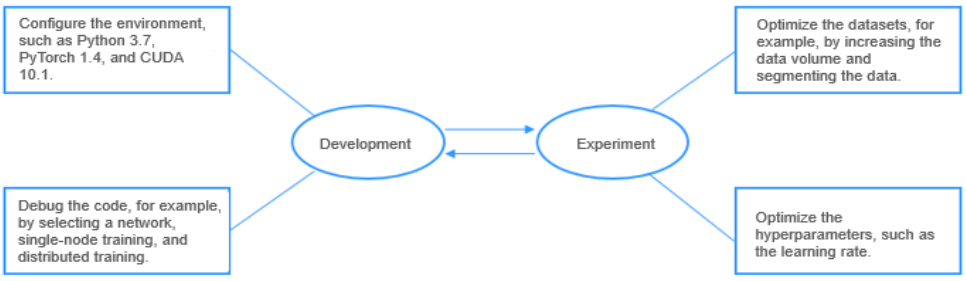
1 Model Training Workflow

The process of developing an AI model is referred to as modeling, which typically involves two stages:

- Development: This involves preparing and configuring the environment and debugging code to ensure it is ready for deep learning training. It is recommended that you debug code within the ModelArts development environment.
- Experimentation: This stage focuses on fine-tuning datasets and adjusting hyperparameters. Through multiple rounds of experimentation, you can train a model that meets your performance goals. It is recommended that you conduct these experiments using ModelArts training jobs.

These two processes are interchangeable. For example, once the code stabilizes in the development stage, the workflow enters the experimentation stage to iterate on the model by continuously tuning hyperparameters. Conversely, if you identify a potential optimization for training performance during the experimentation stage, you can return to the development stage to optimize your code.

Figure 1-1 Model development workflow



ModelArts provides model training capabilities, allowing you to monitor training progress and continuously tune model parameters. You can also select resource pools of different specifications based on your data requirements for model training.

Follow the guidance below to train models on ModelArts.

Table 1-1 Model training workflow

Task	Subtask	Description
Making preparations	Prepare training code.	The essential elements for model training include training code, a training framework (image), and training data. Training code consists of the boot file or command and training dependency packages. • When using a preset image to create a training job, develop training code by referring to **Developing Code for Training Using a Preset Image**. • When using a custom image to create a training job, develop training code by referring to **Developing Code for Training Using a Custom Image**.
	Prepare a training image.	Model training supports multiple image sources. For details, see **Preparing a Model Training Image**. • ModelArts offers mainstream preset images for model training, ready for immediate use. • If the preset images do not meet your needs, create a custom image.
	Prepare training data.	In addition to training datasets, training data can also include predictive models. Prepare your data before creating a training job. • If the data is ready for use without further processing, upload it directly to an OBS bucket. Specify the OBS path as the input parameter when creating the training job. • If your dataset is unlabeled or requires further preprocessing, import it into the ModelArts **Data Management** module. Select the dataset from this module as the input parameter when creating the training job.
Creating an algorithm	Create an algorithm.	Before creating a production training job, you must prepare your own algorithm or subscribe to an algorithm from AI Gallery.
Creating a production training job	Use basic training features.	• You can create a training job on the ModelArts console. Multiple creation methods are available depending on the algorithm type and training framework. For details, see **Table 1-2**. • ModelArts also allows you to create training jobs using APIs. For details, see **Using PyTorch to Create a Training Job (New-Version Training)**.

Task	Subtask	Description
	Use advanced training features.	ModelArts supports the following advanced training features: • Incremental learning • Distributed training • Training acceleration • High training reliability
Viewing training results and logs	View training job details.	During or after a training job, you can view parameter settings, job events, and other details on the training job details page.
	View training job logs.	Training logs record the execution process and exception information. You can use these logs to locate issues occurred during job execution.

Table 1-2 Training job creation methods

Creation Method	Use Case
Preset images	Use this method if you have developed your algorithm locally using a mainstream framework.
Custom images	Use this method if your algorithm relies on a non-mainstream framework. You can create an image using your algorithm and use the image to create training jobs.
Existing algorithms	Use this method if you want to use algorithms already managed in the **Algorithm Management** module, including those you created yourself or those subscribed to from AI Gallery.
AI Gallery algorithms	Use this method if you want to leverage ready-to-use algorithms. You can subscribe to algorithms from AI Gallery to quickly create training jobs.

2 Preparing Model Training Code

2.1 Starting Training Using a Preset Image's Boot File

ModelArts offers preset popular AI images tailored to their specific features. To train models efficiently, modify the boot command according to your chosen image's requirements for seamless execution.

ModelArts provides four preset images for creating training jobs.

Table 2-1 Preset images

Scenario	Preset Image	Description		Advantages and Suggestions	
NPU	**Ascend-Powered-Engine**	It is a set of AI images, runtime environments, and boot modes tailored for Huawei Cloud's AI accelerator chips. There are three boot modes:	**Method 1: Using an RTF file to start a training job**	Select this option if you use Huawei Cloud NPUs. This option can improve the model training efficiency.	RTF: general scenarios
			Method 2: Using the torchrun command to start a training job		torchrun: Use PyTorch +Ascend Extension for PyTorch.

Scenario	Preset Image	Description		Advantages and Suggestions	
			Method 3: Using the msrun command to start a training job		msrun: You can use this mode after MindSpore is installed. This option works independently of external libraries or configuration files, includes disaster recovery, and maintains high security.
GPU	**PyTorch-GPU**	PyTorch-GPU preset image.		This image features flexibility and ease of use, and works well for research and quick development.	
	TensorFlow-GPU	TensorFlow-GPU preset image.		This image excels in deploying models in production environments and optimizing performance, making it ideal for enterprise applications.	
	Horovod/MPI/MindSpore-GPU	ModelArts uses **mpirun** to run boot files for Horovod, MPI, or MindSpore-GPU.		This feature requires the OpenMPI library. Choose it only if you are familiar with OpenMPI.	

This section describes how to modify the boot file when creating a training job using different preset images.

Constraints

When you use the **torchrun** command to start a training job, the PyTorch version must be 1.11.0 or later.

When you use the **msrun** command to start a training job, the MindSpore version must be 2.3.0 or later.

Ascend-Powered-Engine

Ascend-Powered-Engine is a unique engine that combines an AI framework, runtime environment, and boot mode tailored to Ascend accelerator chips. Unlike

conventional AI frameworks like PyTorch or TensorFlow, or parallel execution frameworks such as MPI, it serves a distinct purpose.

Snt9 Ascend accelerators run on Arm CPU environments, which means their Docker images are Arm images. Ascend-Powered-Engine images include the Huawei's CANN (heterogeneous computing architecture) compute library instead of the CUDA (unified computing architecture) compute library used in GPU setups. CANN supports AI tasks and works with Ascend drivers.

After a training job is submitted, ModelArts automatically runs the boot file. When using the Ascend-Powered-Engine framework, both single-node and distributed jobs start with the same command.

The Ascend-Powered-Engine framework offers three boot modes. By default, the boot file is executed based on **RANK_TABLE_FILE**. You can also configure the **MA_RUN_METHOD** environment variable to run the boot file using alternative methods. The **MA_RUN_METHOD** environment variable offers two boot options: **torchrun** and **msrun**.

- **Method 1: Using an RTF file to start a training job**

  WARNING

  MindSpore 2.7.2 and newer versions no longer support the ranktable boot mode. You are advised to use the msrun boot mode instead.

  If the environment variable **MA_RUN_METHOD** is not configured, ModelArts uses a ranktable file to start the boot file of the training job by default.

  The number of times the boot file runs for each training job depends on the number of PUs used. When a job is running, the boot file is executed once for each PU. For example, in a single-node job with one PU, the boot file runs once. In a single-node job with eight PUs, it runs eight times. So, do not listen on ports in the boot file.

  The following environment variables are automatically configured in the boot file:

  - **RANK_TABLE_FILE**: ranktable file path.
  - **ASCEND_DEVICE_ID**: logical device ID. For example, for single-PU training, the value is always **0**.
  - **RANK_ID**: logical (sequential) number of a device in a training job.
  - **RANK_SIZE**: Set this parameter based on the number of devices in the ranktable file. For example, the value is **4** for 4 Snt9b devices.

  To ensure the boot file runs only once, check the **ASCEND_DEVICE_ID** value. If it is **0**, execute the logic; otherwise, exit directly.

  To enable ranktable dynamic routing for training network acceleration, add the environment variable **ROUTE_PLAN=true**. For details, see **Enabling Dynamic Route Acceleration for Training Jobs**.

  Sample code script **mindspore-verification.py** of the Ascend-Powered-Engine framework:

```
import os
import numpy as np
from mindspore import Tensor
```



```
import mindspore.ops as ops
import mindspore.context as context

print('Ascend Envs')
print('------')
print('JOB_ID: ', os.environ['JOB_ID'])
print('RANK_TABLE_FILE: ', os.environ['RANK_TABLE_FILE'])
print('RANK_SIZE: ', os.environ['RANK_SIZE'])
print('ASCEND_DEVICE_ID: ', os.environ['ASCEND_DEVICE_ID'])
print('DEVICE_ID: ', os.environ['DEVICE_ID'])
print('RANK_ID: ', os.environ['RANK_ID'])
print('------')

context.set_context(device_target="Ascend")
x = Tensor(np.ones([1,3,3,4]).astype(np.float32))
y = Tensor(np.ones([1,3,3,4]).astype(np.float32))

print(ops.add(x, y))
```

- **Method 2: Using the torchrun command to start a training job**

  If the environment variable **MA_RUN_METHOD** is set to **torchrun**, ModelArts uses the **torchrun** command to run the boot file.

  - For single-node jobs, ModelArts uses these commands to start the boot file:

    ```
    torchrun --standalone --nnodes=${MA_NUM_HOSTS} --nproc_per_node=${MA_NUM_GPUS} $
    {MA_EXTRA_TORCHRUN_PARAMS} "Boot file" {arg1} {arg2} ...
    ```

  - For multi-node jobs, ModelArts uses these commands to start the boot file:

    ```
    torchrun --nnodes=${MA_NUM_HOSTS} --nproc_per_node=${MA_NUM_GPUS} --node_rank=$
    {VC_TASK_INDEX} --master_addr={master_addr} --master_port=$
    {MA_TORCHRUN_MASTER_PORT} --rdzv_id={ma_job_name} --rdzv_backend=static $
    {MA_EXTRA_TORCHRUN_PARAMS} "Boot file" {arg1} {arg2} ...
    ```

  Parameters:

Table 2-2 Parameters for starting a training job using the **torchrun** command

Parameter	Description
standalone	Identifier of a single-node job.
nnodes	Number of task nodes.
nproc_per_node	Number of main processes started by each task node. Set this parameter to the number of NPUs allocated to the task.
node_rank	Task rank, which is used for multi-task distributed training.
master_addr	Address of the main task (rank 0). Set it to the communication domain name of **worker-0**.
master_port	Port used for communication during distributed training on the main task (rank 0). The default value is **18888**. When a **master_port** conflict occurs, you can modify the port configuration by configuring the **MA_TORCHRUN_MASTER_PORT** environment variable.
rdzv_id	Rendezvous ID. Set it to a value with the training job ID.

Parameter	Description
rdzv_backend	Rendezvous backend, which is fixed at **static**. That is, **master_addr** and **master_port** are used instead of Rendezvous.

  - Additionally, you can configure the **MA_EXTRA_TORCHRUN_PARAMS** environment variable to add additional **torchrun** command parameters or overwrite the preset **torchrun** command parameters. The following is an example of configuring the **rdzv_conf** parameter in the **torchrun** command:

    ```
    "environments": {
    "MA_RUN_METHOD": "torchrun",
    "MA_EXTRA_TORCHRUN_PARAMS": "--rdzv_conf=timeout=7200"
    }
    ```

    If the **RuntimeError: Socket Timeout** error occurs during the distributed process group initialization using **torchrun**, go to **1** and further locate the fault.

- **Method 3: Using the msrun command to start a training job**

  If the environment variable **MA_RUN_METHOD** is set to **msrun**, ModelArts uses the **msrun** command to run the boot file.

  This solution supports dynamic networking and ranktable file-based networking. If you set the environment variable **MS_RANKTABLE_ENABLE** to **True**, **msrun** reads the ranktable file for networking. Otherwise, dynamic networking is used by default.

  msrun uses these commands to start the boot file:

  ```
  msrun --worker_num=${msrun_worker_num} --local_worker_num=${MA_NUM_GPUS} --master_addr=${msrun_master_addr} --node_rank=${VC_TASK_INDEX} --master_port=${msrun_master_port} --log_dir=${msrun_log_dir} --join=True --cluster_time_out=${MSRUN_CLUSTER_TIME_OUT} --rank_table_file=${msrun_rank_table_file} "Boot file" {arg1} {arg2} ...
  ```

  Parameters:

  Table 2-3 Parameters for starting a training job using the **msrun** command

Parameter	Description
worker_num	Total number of processes, which is also equivalent to the number of PUs involved, as each PU initiates a process.
local_worker_num	Number of processes on the current node, which is also the number of PUs used by the current node.
master_addr	IP address of the node where the **msrun** scheduling process is located. This parameter does not need to be configured for single-node jobs.
master_port	Port number of the **msrun** scheduling process.
node_rank	Port number of the **msrun** scheduling process.
log_dir	Log output directory of **msrun** and all processes.

Parameter	Description
join	Specifies whether the **msrun** process still exists after the training process is started. The default value is **True**, indicating that the **msrun** process exits after all processes exit.
cluster_time_out	Timeout interval of the cluster networking. The default value is **600s**. The value can be controlled by the **MSRUN_CLUSTER_TIME_OUT** environment variable.
rank_table_file	Address of the ranktable file. If the environment variable **MS_RANKTABLE_ENABLE** is set to **True**, this parameter is added during startup.

PyTorch-GPU

For single-node multi-PU scenarios, the platform adds the **--init_method "tcp://<ip>:<port>"** parameter to the boot file.

For multi-node multi-PU scenarios, the platform adds the **--init_method "tcp://<ip>:<port>" --rank <rank_id> --world_size <node_num>** parameter to the boot file.

The preceding parameters must be parsed in the boot file.

For details about the code example of the PyTorch-GPU framework, see "Method 1" in **Example: Creating a DDP Distributed Training Job (PyTorch + GPU)**.

TensorFlow-GPU

For a single-node job, ModelArts starts a training container that exclusively uses the resources on the node.

For a multi-node job, ModelArts starts a parameter server and a worker on the same node. It allocates parameter server and worker tasks in a 1:1 ratio. For example, in a two-node job, two parameter servers and two workers are allocated. ModelArts also injects the following parameters into the boot file:

```
--task_index <VC_TASK_INDEX> --ps_hosts <TF_PS_HOSTS> --worker_hosts <TF_WORKER_HOSTS> --job_name <MA_TASK_NAME>
```

The following parameters must be parsed in the boot file.

Table 2-4 Parameters

Parameter	Description
VC_TASK_INDEX	Task serial number, for example, **0**, **1**, or **2**.
TF_PS_HOSTS	Addresses of parameter server nodes, for example, **[***xx***-ps-0.***xx***:***TCP_PORT***,***xx***-ps-1.***xx***:***TCP_PORT***]**. The value of *TCP_PORT* is a random port ranging from 5,000 to 10,000.

Parameter	Description
TF_WORKER_HOSTS	Addresses of worker nodes, for example, **[***xx*-**worker-0.***xx*:*TCP_PORT*,*xx*-**worker-1.***xx*:*TCP_PORT***]**. The value of *TCP_PORT* is a random port ranging from 5,000 to 10,000.
MA_TASK_NAME	Task name, which can be **ps** or **worker**.

Horovod/MPI/MindSpore-GPU

ModelArts uses **mpirun** to run training boot files for Horovod, MPI, or MindSpore-GPU. To use a preset engine in ModelArts, simply edit the boot file (training script). ModelArts Standard automatically builds the **mpirun** command and training job cluster. The platform does not add extra parameters to the boot file.

Example of **pytorch_synthetic_benchmark.py**:

```
import argparse
import torch.backends.cudnn as cudnn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data.distributed
from torchvision import models
import horovod.torch as hvd
import timeit
import numpy as np

# Benchmark settings
parser = argparse.ArgumentParser(description='PyTorch Synthetic Benchmark',
                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--fp16-allreduce', action='store_true', default=False,
                    help='use fp16 compression during allreduce')

parser.add_argument('--model', type=str, default='resnet50',
                    help='model to benchmark')
parser.add_argument('--batch-size', type=int, default=32,
                    help='input batch size')

parser.add_argument('--num-warmup-batches', type=int, default=10,
                    help='number of warm-up batches that don\'t count towards benchmark')
parser.add_argument('--num-batches-per-iter', type=int, default=10,
                    help='number of batches per benchmark iteration')
parser.add_argument('--num-iters', type=int, default=10,
                    help='number of benchmark iterations')

parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')

parser.add_argument('--use-adasum', action='store_true', default=False,
                    help='use adasum algorithm to do reduction')

args = parser.parse_args()
args.cuda = not args.no_cuda and torch.cuda.is_available()

hvd.init()

if args.cuda:
    # Horovod: pin GPU to local rank.
    torch.cuda.set_device(hvd.local_rank())

cudnn.benchmark = True
```



```
# Set up standard model.
model = getattr(models, args.model)()

# By default, Adasum doesn't need scaling up learning rate.
lr_scaler = hvd.size() if not args.use_adasum else 1

if args.cuda:
    # Move model to GPU.
    model.cuda()
    # If using GPU Adasum allreduce, scale learning rate by local_size.
    if args.use_adasum and hvd.nccl_built():
        lr_scaler = hvd.local_size()

optimizer = optim.SGD(model.parameters(), lr=0.01 * lr_scaler)

# Horovod: (optional) compression algorithm.
compression = hvd.Compression.fp16 if args.fp16_allreduce else hvd.Compression.none

# Horovod: wrap optimizer with DistributedOptimizer.
optimizer = hvd.DistributedOptimizer(optimizer,
                                     named_parameters=model.named_parameters(),
                                     compression=compression,
                                     op=hvd.Adasum if args.use_adasum else hvd.Average)

# Horovod: broadcast parameters & optimizer state.
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
hvd.broadcast_optimizer_state(optimizer, root_rank=0)

# Set up fixed fake data
data = torch.randn(args.batch_size, 3, 224, 224)
target = torch.LongTensor(args.batch_size).random_() % 1000
if args.cuda:
    data, target = data.cuda(), target.cuda()


def benchmark_step():
    optimizer.zero_grad()
    output = model(data)
    loss = F.cross_entropy(output, target)
    loss.backward()
    optimizer.step()


def log(s, nl=True):
    if hvd.rank() != 0:
        return
    print(s, end='\n' if nl else '')


log('Model: %s' % args.model)
log('Batch size: %d' % args.batch_size)
device = 'GPU' if args.cuda else 'CPU'
log('Number of %ss: %d' % (device, hvd.size()))

# Warm-up
log('Running warmup...')
timeit.timeit(benchmark_step, number=args.num_warmup_batches)

# Benchmark
log('Running benchmark...')
img_secs = []
for x in range(args.num_iters):
    time = timeit.timeit(benchmark_step, number=args.num_batches_per_iter)
    img_sec = args.batch_size * args.num_batches_per_iter / time
    log('Iter #%d: %.1f img/sec per %s' % (x, img_sec, device))
    img_secs.append(img_sec)

# Results
img_sec_mean = np.mean(img_secs)
```



```
img_sec_conf = 1.96 * np.std(img_secs)
log('Img/sec per %s: %.1f +-%.1f' % (device, img_sec_mean, img_sec_conf))
log('Total img/sec on %d %s(s): %.1f +-%.1f' %
    (hvd.size(), device, hvd.size() * img_sec_mean, hvd.size() * img_sec_conf))
```

run_mpi.sh is as follows:

```
#!/bin/bash
MY_HOME=/home/ma-user

MY_SSHD_PORT=${MY_SSHD_PORT:-"36666"}

MY_MPI_BTL_TCP_IF=${MY_MPI_BTL_TCP_IF:-"eth0,bond0"}

MY_TASK_INDEX=${MA_TASK_INDEX:-${VC_TASK_INDEX:-${VK_TASK_INDEX}}}

MY_MPI_SLOTS=${MY_MPI_SLOTS:-"${MA_NUM_GPUS}"}

MY_MPI_TUNE_FILE="${MY_HOME}/env_for_user_process"

if [ -z ${MY_MPI_SLOTS} ]; then
    echo "[run_mpi] MY_MPI_SLOTS is empty, set it be 1"
    MY_MPI_SLOTS="1"
fi

printf "MY_HOME: ${MY_HOME}\nMY_SSHD_PORT: ${MY_SSHD_PORT}\nMY_MPI_BTL_TCP_IF: $
{MY_MPI_BTL_TCP_IF}\nMY_TASK_INDEX: ${MY_TASK_INDEX}\nMY_MPI_SLOTS: ${MY_MPI_SLOTS}\n"

env | grep -E '^MA_|SHARED_|^S3_|^PATH|^VC_WORKER_|^SCC|^CRED' | grep -v '=$' > $
{MY_MPI_TUNE_FILE}
# add -x to each line
sed -i 's/^/-x /' ${MY_MPI_TUNE_FILE}

sed -i "s|{{MY_SSHD_PORT}}|${MY_SSHD_PORT}|g" ${MY_HOME}/etc/ssh/sshd_config

# start sshd service
bash -c "$(which sshd) -f ${MY_HOME}/etc/ssh/sshd_config"

# confirm the sshd is up
netstat -anp | grep LIS | grep ${MY_SSHD_PORT}

if [ $MY_TASK_INDEX -eq 0 ]; then
    # generate the hostfile of mpi
    for ((i=0; i<$MA_NUM_HOSTS; i++))
    do
        eval hostname=${MA_VJ_NAME}-${MA_TASK_NAME}-${i}.${MA_VJ_NAME}
        echo "[run_mpi] hostname: ${hostname}"

        ip=""
        while [ -z "$ip" ]; do
            ip=$(ping -c 1 ${hostname} | grep "PING" | sed -E 's/PING .* .([0-9.]+). .*/\1/g')
            sleep 1
        done
        echo "[run_mpi] resolved ip: ${ip}"

        # test the sshd is up
        while :
        do
            if [ cat < /dev/null >/dev/tcp/${ip}/${MY_SSHD_PORT} ]; then
                break
            fi
            sleep 1
        done

        echo "[run_mpi] the sshd of ip ${ip} is up"

        echo "${ip} slots=$MY_MPI_SLOTS" >> ${MY_HOME}/hostfile
    done

    printf "[run_mpi] hostfile:\n`cat ${MY_HOME}/hostfile`\n"
```



```
fi

RET_CODE=0

if [ $MY_TASK_INDEX -eq 0 ]; then

    echo "[run_mpi] start exec command time: "$(date +"%Y-%m-%d-%H:%M:%S")

    np=$(( ${MA_NUM_HOSTS} * ${MY_MPI_SLOTS} ))

    echo "[run_mpi] command: mpirun -np ${np} -hostfile ${MY_HOME}/hostfile -mca plm_rsh_args \"-p $
{MY_SSHD_PORT}\" -tune ${MY_MPI_TUNE_FILE} ... $@"

    # execute mpirun at worker-0
    # mpirun
    mpirun \
        -np ${np} \
        -hostfile ${MY_HOME}/hostfile \
        -mca plm_rsh_args "-p ${MY_SSHD_PORT}" \
        -tune ${MY_MPI_TUNE_FILE} \
        -bind-to none -map-by slot \
        -x NCCL_DEBUG=INFO -x NCCL_SOCKET_IFNAME=${MY_MPI_BTL_TCP_IF} -x
NCCL_SOCKET_FAMILY=AF_INET \
        -x HOROVOD_MPI_THREADS_DISABLE=1 \
        -x LD_LIBRARY_PATH \
        -mca pml ob1 -mca btl ^openib -mca plm_rsh_no_tree_spawn true \
        "$@"

    RET_CODE=$?

    if [ $RET_CODE -ne 0 ]; then
        echo "[run_mpi] exec command failed, exited with $RET_CODE"
    else
        echo "[run_mpi] exec command successfully, exited with $RET_CODE"
    fi

    # stop 1...N worker by killing the sleep proc
    sed -i '1d' ${MY_HOME}/hostfile
    if [ `cat ${MY_HOME}/hostfile | wc -l` -ne 0 ]; then
        echo "[run_mpi] stop 1 to (N - 1) worker by killing the sleep proc"

        sed -i 's/${MY_MPI_SLOTS}/1/g' ${MY_HOME}/hostfile
        printf "[run_mpi] hostfile:\n`cat ${MY_HOME}/hostfile`\n"

        mpirun \
        --hostfile ${MY_HOME}/hostfile \
        --mca btl_tcp_if_include ${MY_MPI_BTL_TCP_IF} \
        --mca plm_rsh_args "-p ${MY_SSHD_PORT}" \
        -x PATH -x LD_LIBRARY_PATH \
        pkill sleep \
        > /dev/null 2>&1
    fi

    echo "[run_mpi] exit time: "$(date +"%Y-%m-%d-%H:%M:%S")
else
    echo "[run_mpi] the training log is in worker-0"
    sleep 365d
    echo "[run_mpi] exit time: "$(date +"%Y-%m-%d-%H:%M:%S")
fi

exit $RET_CODE
```

FAQs

1. What Should I Do If RuntimeError: Socket Timeout Is Displayed During Distributed Process Group Initialization Using torchrun?

   If the **RuntimeError: Socket Timeout** error occurs during the distributed process group initialization using **torchrun**, you can add the following

environment variables to create a training job again to view initialization details and further locate the fault.

- LOGLEVEL=INFO
- TORCH_CPP_LOG_LEVEL=INFO
- TORCH_DISTRIBUTED_DEBUG=DETAIL

The **RuntimeError: Socket Timeout** error is caused by a significant time discrepancy between tasks when running the **torchrun** command. The time discrepancy is caused by initialization tasks, like downloading the training data and checkpoint read/write, which happen before the **torchrun** command is run. If the time taken to complete these initialization tasks varies significantly, a Socket Timeout error may occur. When this error happens, check the time difference between the **torchrun** execution points for each task. If the time difference is too large, optimize the initialization process before running the **torchrun** command to ensure a reasonable time gap.

2.2 Developing Code for Training Using a Preset Image

On ModelArts, you must use preset images for building and training models. If these images do not fit specific needs, customizing them becomes necessary. Before creating an algorithm with a preset image, writing algorithm code ensures it fully supports your requirements. This preparation improves efficiency in model training and optimization, streamlining the overall development process.

Configuration Path

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. For details about the boot process of the boot file of a preset image, see **Starting Training Using a Preset Image's Boot File**.

- Input path

  The training data must be uploaded to an OBS bucket or stored in the **dataset**. In the training code, the input path must be parsed. ModelArts

automatically downloads the data in the input path to the local container directory for training. Ensure that you have the read permission to the OBS bucket. After the training job is started, ModelArts mounts a disk to the **/cache** directory. You can use this directory to store temporary files. For details about the size of the **/cache** directory, see **What Are Sizes of the /cache Directories for Different Resource Specifications in the Training Environment?**

- Output path

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

In ModelArts, the training code must contain the steps in **(Optional) Introducing Dependencies** and **Parsing and Setting Input and Output Paths**.

(Optional) Introducing Dependencies

If your model references other dependencies, place the required file or installation package in **Code Directory** you set during algorithm creation.

- For details about how to install the Python dependency package, see **How Do I Create a Training Job When a Dependency Package Is Referenced by the Model to Be Trained?**
- For details about how to install a C++ dependency library, see **How Do I Install a Library That C++ Depends on?**
- For details about how to load parameters to a pre-trained model, see **How Do I Load Some Well Trained Parameters During Job Training?**

Parsing and Setting Input and Output Paths

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

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

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

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

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

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

2. When creating a training job, set the input and output paths.

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

Complete Training Code Example

The training code is 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.

The following training code example contains the code for saving the model.

```
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'))
```

2.3 Developing Code for Training Using a Custom Image

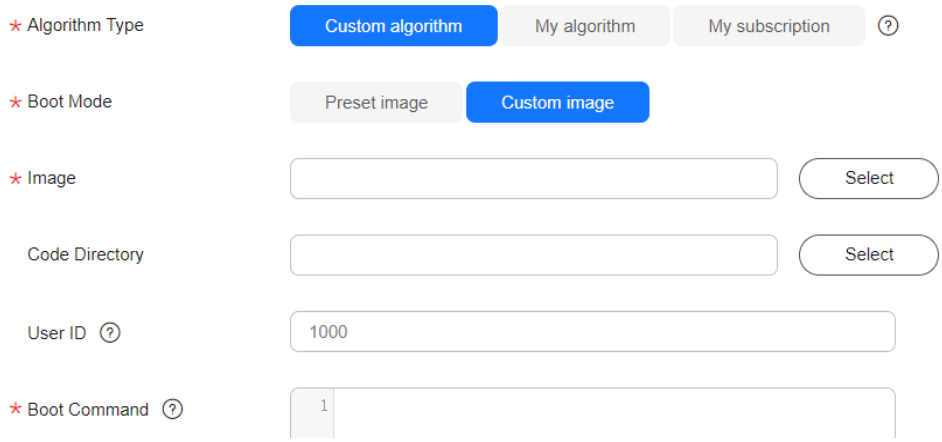
ModelArts offers various preset images and algorithms for most needs. If these do not suit you, its custom image training feature allows flexibility. You can create a tailored image to design the exact training environment required.

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).

Boot Command Specifications for Custom Images

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.

Figure 2-1 Selecting a custom image



WARNING

1. When you use a custom image to create a training job, the boot command must be executed in the **/home/ma-user** directory. Otherwise, the training job may run abnormally.
2. If there are input pipes, output pipes, or hyperparameters, ensure that the last command in the boot commands runs the training script. Otherwise, an error is reported.

1. You must use **conda env** to start training jobs created using custom images.

   Training jobs do not run in a shell. Therefore, you are not allowed to run the **conda activate** command to activate a specified Conda environment. In this case, use other methods to start training. For example, Conda in your custom image is installed in the **/home/ma-user/anaconda3** directory, the Conda environment is **python-3.7.10**, and the training script is stored in **/home/ma-user/modelarts/user-job-dir/code/train.py**. Use a specified Conda environment to start training in one of the following ways:

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

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

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

     ```
     python /home/ma-user/modelarts/user-job-dir/code/train.py
     ```

   - Method 2: Use the absolute path of Conda environment Python.

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

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

   - Method 3: Configure the **PATH** environment variable.

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

     ```
     export PATH=/home/ma-user/anaconda3/envs/python-3.7.10/bin:$PATH; python /home/ma-user/modelarts/user-job-dir/code/train.py
     ```


- Method 4: Run the **conda run -n** command.

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

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

2. The last command in the boot commands must run the training script.

   If there are input pipes, output pipes, or hyperparameters, ensure that the last command in the boot commands runs the training script.

   Reason: The system appends input pipes, output pipes, and hyperparameters to the end of the boot commands. If the last command is not the training script, an error will occur.

   Example: If the last line of the boot commands is **python train.py** and the **--data_url** hyperparameter exists, the system executes **python train.py --data_url=/input** when running properly. However, if the boot commands end with another command, such as:

   ```
   python train.py
   pwd    # The last command is pwd instead of the training script.
   ```

   The system will execute **python train.py pwd --data_url=/input**, leading to an error.

Training Code Adaptation Specifications for Training Using an Ascend-powered Custom Image

When creating a training job that uses NPU resources, the system automatically generates the **Ascend HCCL RANK_TABLE_FILE** file in the training container. When using a preset image, **Ascend HCCL RANK_TABLE_FILE** is automatically parsed during training. When using a custom image, the training code must be modified to read and parse **Ascend HCCL RANK_TABLE_FILE**.

Ascend HCCL RANK_TABLE_FILE file description

Ascend HCCL RANK_TABLE_FILE provides the cluster used by Ascend distributed training jobs. It is used for distributed communication between Ascend chips and can be parsed by Huawei Collective Communication Library (HCCL). The file has two format versions: template 1 and template 2.

- ModelArts provides the template 2 format. The **Ascend HCCL RANK_TABLE_FILE** file in the ModelArts training environment is named **jobstart_hccl.json**. You can access this file using the preset **RANK_TABLE_FILE** environment variable.

  Table 2-5 RANK_TABLE_FILE environment variables

Environment Variable	Description
RANK_TABLE_FILE	Directory of **Ascend HCCL RANK_TABLE_FILE**, which is **/user/config**. Obtain the file using **${RANK_TABLE_FILE}/jobstart_hccl.json**.

Example of the **jobstart_hccl.json** file content in the ModelArts training environment (template 2):

```
{
   "group_count": "1",
   "group_list": [{
      "device_count": "1",
      "group_name": "job-trainjob",
      "instance_count": "1",
      "instance_list": [{
         "devices": [{
            "device_id": "4",
            "device_ip": "192.1.10.254"
         }],
         "pod_name": "jobxxxxxxxx-job-trainjob-0",
         "server_id": "192.168.0.25"
      }]
   }],
   "status": "completed"
}
```

In **jobstart_hccl.json**, the **status** value may not be **completed** when the training script is started. In this case, wait until the **status** value changes to **completed** and read the remaining content of the file.

- After the **status** field is **completed**, use the training script to convert the **jobstart_hccl.json** file from template 2 to template 1 format.

  Format of the **jobstart_hccl.json** file after format conversion (template 1):

```
{
   "server_count": "1",
   "server_list": [{
      "device": [{
         "device_id": "4",
         "device_ip": "192.1.10.254",
         "rank_id": "0"
      }],
      "server_id": "192.168.0.25"
   }],
   "status": "completed",
   "version": "1.0"
}
```

Mount Points of a Training Job in a Container

When training a model with a custom image, the mount points in the container are shown in **Table 2-6**.

Table 2-6 Training job mount points

Mount Point	Read Only	Remarks
/xxx	No	Directory where a dedicated resource pool mounts an SFS disk. You can specify this directory.
/home/ma-user/modelarts	No	This folder is empty. You should use it as the main directory.
/cache	No	Used to mount the hard disk of the host NVMe (supported by bare metal specifications).
/dev/shm	No	Used for PyTorch engine acceleration

Mount Point	Read Only	Remarks
/usr/local/nvidia	Yes	NV library of the host machine.

FAQs

1. What Should I Do If an Error Is Reported Indicating that the .so File in the $ANACONDA_DIR/envs/$DEFAULT_CONDA_ENV_NAME/lib Directory Cannot Be Found During Training?

   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
   ```

2.4 Configuring Password-free SSH Mutual Trust Between Instances for a Training Job Created Using a Custom Image

For distributed training with custom images using MPI or Horovod, set up password-free SSH trust between instances to enable seamless communication. Otherwise, the training will fail.

This involves code adaptation and training job parameter configuration.

1. Create a custom image with OpenSSH pre-installed. The training framework should be MPI or Horovod.
2. Create a boot script file **start_sshd.sh**.

   ```
   MY_SSHD_PORT=${MY_SSHD_PORT:-"38888"}
   mkdir -p /home/ma-user/etc
   ssh-keygen -f /home/ma-user/etc/ssh_host_rsa_key0 -N '' -t rsa > /dev/null
   /usr/sbin/sshd -p $MY_SSHD_PORT -h /home/ma-user/etc/ssh_host_rsa_key0
   ```

3. Upload the sshd startup script file to the training code directory in OBS.
4. Create a training job using the custom image.
   - **Code Directory**: Select the OBS path where the sshd boot script file is stored.
   - **Boot Command**: Adapt the boot command to the sshd boot script.

     ```
     bash ${MA_JOB_DIR}/demo-code/start_sshd.sh && your custom command
     ```

     In the command, **your custom command** indicates custom commands you want to execute in the training job.
   - **Environment Variable**: Add **MY_SSHD_PORT = 38888**.
   - **Password-free SSH Between Nodes**: Enable it and set **Password-free SSH File Directory**. Use the default value in most cases. After a training job is delivered, the SSH key file and configuration file **authorized_keys**

config id_rsa id_rsa.pub are automatically generated in the **/home/ma-user/.ssh** directory of the training container.

5. After a training job is created, its instances can establish an SSH connection with each other by using the domain name and port number throughout the training process. The sample code is as follows:

```
ssh modelarts-job-a0978141-1712-4f9b-8a83-000000000000-worker-1 -p $MY_SSHD_PORT
```


3 Preparing a Model Training Image

ModelArts provides deep learning-powered base images such as TensorFlow, PyTorch, and MindSpore images. In these images, the software mandatory for running training jobs has been installed. If the software in the base images cannot meet your service requirements, create new images based on the base images and use the new images to create training jobs.

Preset Training Images

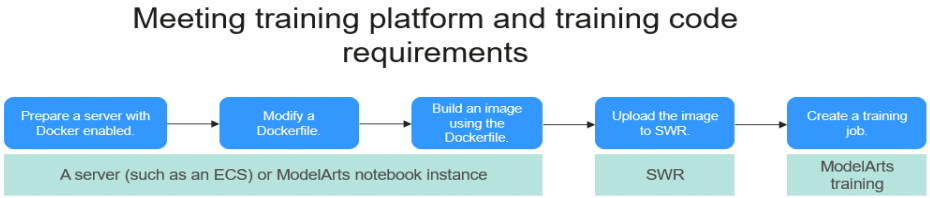
The following table lists the preset training base images of ModelArts.

Table 3-1 ModelArts training base images

Engine	Version
PyTorch	pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64
	pytorch_2.1.0-cuda_12.1-py_3.10.6-ubuntu_22.04-x86_64
TensorFlow	tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64
Horovod	horovod_0.20.0-tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64
	horovod_0.22.1-pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64
MPI	mindspore_1.3.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64
Ascend-Powered-Engine	tensorflow_1.15-cann_5.1.0-py_3.7-euler_2.8.3-aarch64
	mindspore_1.7.0-cann_5.1.0-py_3.7-euler_2.8.3-aarch64
	pytorch_2.1.0-cann_7.0.1.1-py_3.9-euler_2.10.7-aarch64-snt3p
	mindspore_2.2.12-cann_7.0.1.1-py_3.9-euler_2.10.7-aarch64-snt3p

Creating a Custom Training Image

Figure 3-1 Creating a custom image for a training job



Scenario 1: If the preset images meet ModelArts training constraints but lack necessary code dependencies, install additional software packages.

For details, see **Creating a Custom Training Image Using a Preset Image**.

Scenario 2: If the local images meet code dependency requirements but not ModelArts training constraints, adapt them to ModelArts.

For details, see **Migrating Existing Images to ModelArts**.

Scenario 3: If neither the preset nor local images meet your needs, create an image that meets both code dependency and ModelArts training constraints. For details, see the following cases:

Creating a Custom Training Image (PyTorch + Ascend)

Creating a Custom Training Image (PyTorch + CPU/GPU)

Creating a Custom Training Image (MPI + CPU/GPU)

Creating a Custom Training Image (Tensorflow + GPU)

Installing pip Dependencies in an Image

Before creating distributed training jobs, pre-install all required pip dependencies. If there are more than 10 nodes, the system automatically deletes the pip source configuration. Executing pip install commands during training may cause training failures.

Install all dependency packages before training. This stops failures from missing pip source configurations when using many nodes, making training more stable and efficient.

Install pip dependencies in either of the following ways:

- Method 1: Install dependencies in a notebook and save it as an image.
  a. Run the target image in the notebook environment.
  b. Run the **pip install** command to install all dependency packages.
  c. Save the image.
  d. Obtain the SWR address on the image details page for future training.
- Method 2: Install dependencies in a local container and export an image.

a. Run the container in the local environment and run the **pip install** command to install all dependencies.

b. Run the following command to save the container as an image:

```
docker commit <Container ID> <Image name>
```

Example:

```
docker commit my_container my_image:v1
```

c. Run the following command to export the image as a .tar file:

```
docker save -o <.tar file name>.tar <Image name>:<Tag>
```

Example:

```
docker save -o my_image_v1.tar my_image:v1
```

d. Upload the image to SWR for future training jobs.

4 Creating an Algorithm

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

ModelArts provides the following algorithm sources for model training:

- Using a subscribed algorithm

  You can directly subscribe to algorithms in ModelArts AI Gallery and use them to build models without writing code.

- Using a preset image

  To use a custom algorithm, use a framework built in ModelArts. ModelArts supports most mainstream AI engines. For details, see **Starting Training Using a Preset Image's Boot File**. 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 **Developing Code for Training Using a Preset Image**.

- Using a preset image with customization

  If you use a preset image to create an algorithm 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.

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

- 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. 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.

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

NOTE

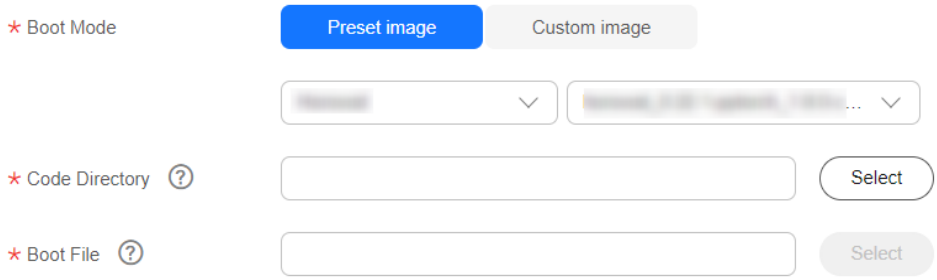
When you use a custom image to create a training job, the boot command must be executed in the **/home/ma-user** directory. Otherwise, the training job may run abnormally.

Creating an Algorithm

Your locally developed algorithms or algorithms developed using other tools can be uploaded to ModelArts for unified management.

1. Make preparations.
   - Create a dataset in ModelArts or upload a training dataset to an OBS directory.
   - Your training script has been uploaded to an OBS directory. For details about how to develop a training script, see **Developing Code for Training Using a Preset Image** or **Developing Code for Training Using a Custom Image**.
   - Create at least one empty folder in OBS for storing training outputs.
   - Make sure your OBS directory and ModelArts are in the same region.
2. Access the algorithm creation page.
   a. Log in to the **ModelArts console**. In the navigation pane, choose **Asset Management** > **Algorithm Management**.
   b. In the **My algorithm** tab, click **Create Algorithm**. Enter the basic algorithm information, including **Name** and **Description**.
3. Set the algorithm boot mode. The options are as follows:
   - Using a preset image

Figure 4-1 Using a preset image to create an algorithm



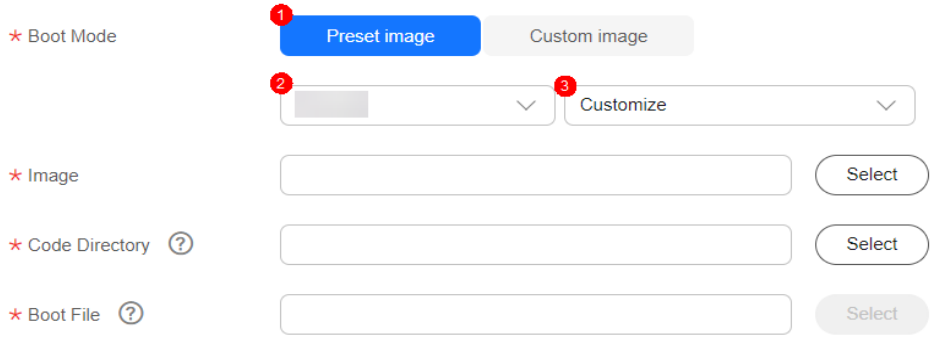
Set **Code Directory** and **Boot File** based on the algorithm code. Ensure that the preset image you select is the same as the one you use for editing algorithm code. For example, if TensorFlow is used for writing algorithm code, select TensorFlow when you create an algorithm.

Table 4-1 Parameters

Parameter	Description
Boot Mode	Select **Preset image**. Select a preset image and its version used by the algorithm.
Code Directory	Select an 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 training 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.

– Using a preset image with customization

Figure 4-2 Creating an algorithm using a preset image with customization



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

Table 4-2 Parameters

Parameter	Description
Boot Mode	Select **Preset image**. Select **Customize** for the engine version.
Image	Select your image uploaded to SWR. For details about how to create an image, see **Creating a Custom Training Image**.

Parameter	Description
Code Directory	Select an 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. When the training job starts, ModelArts downloads the training code directory and its subdirectories to the training 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 for training code. The training boot file must be a Python file. • The total number of both files and folders cannot exceed 1,000. • The total size of files cannot exceed 5 GB. • The file depth cannot exceed 32.
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.

Selecting a preset image with customization results in the same background behavior as running a training job directly with that 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 4-3 Creating an algorithm using a custom image

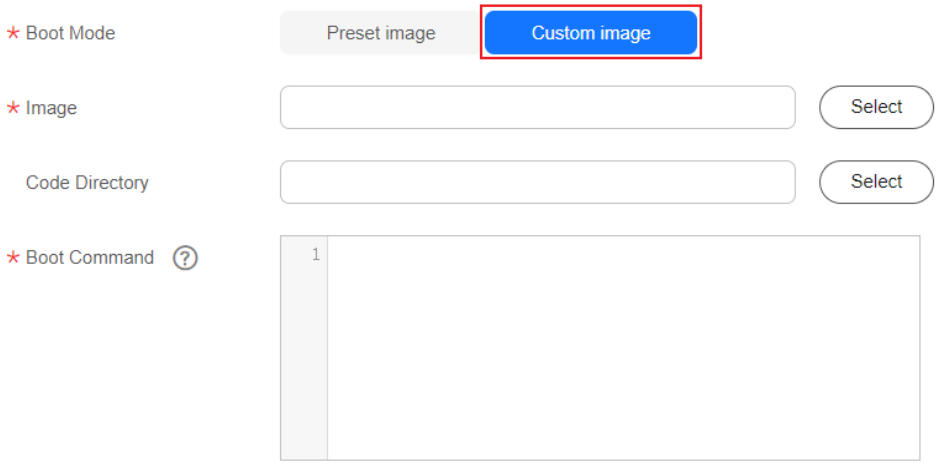


Table 4-3 Parameters

Parameter	Description
Boot Mode	Select **Custom image**.
Image	Select your image uploaded to SWR. For details about how to create an image, see **Creating a Custom Training Image**.

Parameter	Description
Code Directory	Select an 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. Configure this parameter only if your custom image does not contain training code. 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. When the training job starts, ModelArts downloads the training code directory and its subdirectories to the training 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 for training code. The training boot file must be a Python file. ● The total number of both files and folders cannot exceed 1,000. ● The total size of files cannot exceed 5 GB. ● The file depth cannot exceed 32.

Parameter	Description

Boot Command	Command for booting an image. This parameter is mandatory.
	When a training job is running, the boot command is automatically executed after the code directory is downloaded.
	• If the training boot script is a .py file, **train.py** for example, the boot command is as follows:
	`python ${MA_JOB_DIR}/demo-code/train.py`
	• If the training boot script is a .sh file, **main.sh** for example, the boot command is as follows:
	`bash ${MA_JOB_DIR}/demo-code/main.sh`
	You can use semicolons (;) and ampersands (&&) to combine multiple commands. **demo-code** in the command is the last-level OBS directory where the code is stored. Replace it with the actual one.
	If there are input pipes, output pipes, or hyperparameters, ensure that the last command of the boot command runs the training script.
	Reason: The system appends input pipes, output pipes, and hyperparameters to the end of the boot command. If the last command is not the training script, an error will occur.
	Example: If the last line of the boot command is **python train.py** and the **--data_url** hyperparameter exists, the system executes **python train.py --data_url=/input** when running properly. However, if the boot command ends with another command, such as:
	`python train.py`
	`pwd  # The last command is pwd instead of the training script.`
	The system will execute **python train.py pwd --data_url=/input**, leading to an error.

For details about how to use custom images supported by training, see **Boot Command Specifications for Custom Images**.

4. Configure pipelines.

   An 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 **Preparing Model Training Code**.

   – Input configurations

Table 4-4 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. You can select a storage path or ModelArts dataset. This parameter is optional. If you select a ModelArts dataset, set the following parameters: • **Labeling Type**: For details, see **Creating a Labeling Job**. • **Data Format**, which can be **Default**, **CarbonData**, or both. **Default** indicates the manifest format. • **Data Segmentation** is available only for image classification, object detection, text classification, and sound classification datasets. The options are **Segmented dataset**, **Dataset not segmented**, and **Unlimited**. For details, see **Publishing a Data Version**.
Add	Add multiple input data sources based on your algorithm.

– Output configurations

Table 4-5 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 data 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.

Parameter	Description
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.

5. Define hyperparameters.

   When you create an algorithm, ModelArts allows you to customize hyperparameters so you can view or modify them anytime. Defined hyperparameters are displayed in the boot command and passed to your boot file as CLI parameters.

   a. Click **Add hyperparameter** to manually add hyperparameters.

   b. Edit hyperparameters.

   NOTE

   To ensure data security, do not enter sensitive information, such as plaintext passwords.

   Table 4-6 Editing hyperparameters

Parameter	Description
Name	Enter the hyperparameter name. Enter 1 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Type	Select the 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**. Then, 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.

Parameter	Description
Description	Enter the description of the hyperparameter. Only letters, digits, spaces, hyphens (-), underscores (_), commas (,), and periods (.) are allowed.

6. Configure supported policies.

   Auto search on ModelArts automatically finds the optimal hyperparameters without any code modification. For details about parameter settings, see **Creating a Training Job for Automatic Model Tuning**.

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

7. Add 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-PU training.
   - **Distributed Training**: Choose whether to support distributed training.

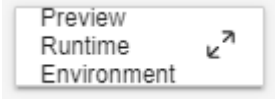
8. Click **Submit**.

   In the algorithm list, click the algorithm to access its details page and view the algorithm details.

   - In the **Basic Information** tab, you can view the algorithm information.

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

Previewing the Runtime Environment

When creating an algorithm, click the arrow on 

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.

Deleting an Algorithm

NOTICE

Deleted algorithm assets cannot be restored.

To delete your algorithm, choose **Asset Management** > **Algorithm Management**. Click **Delete** in the **Operation** column. In the displayed dialog box, enter **DELETE**, and click **OK** to confirm the deletion.

To delete a subscribed algorithm, go to AI Gallery, choose **My Assets** > **Algorithm**, click **My Subscription**, and click **Cancel Subscription** for the algorithm you want to delete. In the displayed dialog box, click **OK**.

5 Creating a Training Job

5.1 Creating a Training Job

Developing models involves optimizing their performance effectively. Traditional methods require repeatedly testing various model designs, datasets, and hyperparameters, which takes significant time and effort but may still fail to deliver good results. ModelArts simplifies this process by offering tools for creating training jobs, tracking progress in real time, and managing versions. With ModelArts, users can test different configurations easily and identify the best-performing setup faster.

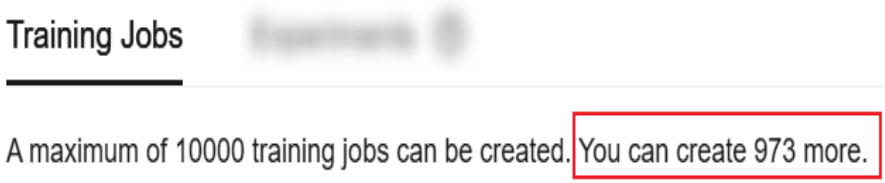
Create a production training job in either of the following ways:

- Create a production training job on the ModelArts console. This chapter provides the operation guide of the new page. For details about the operation guide of the old (default) page, see **Creating a Production Training Job (Old Version)**.
- Use the ModelArts API to create a production training job. For details, see **Using PyTorch to Create a Training Job (New-Version Training)**.

Notes and Constraints

By default, up to 10,000 training jobs can be created. You can view the remaining quota on the training job list page.

Figure 5-1 Viewing the remaining quota of a training job



Prerequisites

- Account not in arrears (paid resources required for training jobs).
- Data for training uploaded to an OBS directory.

- At least one empty folder in OBS for storing training output.

  NOTE

  ModelArts does not support encrypted OBS buckets. When creating an OBS bucket, do not enable bucket encryption.
- OBS directory and ModelArts in the same region.
- Access authorization configured. If you have not yet configured access, follow the instructions in **Configuring Agency Authorization for ModelArts with One Click**.

Billing

Model training in ModelArts uses compute and storage resources, which are billed. Compute resources are billed for running training jobs. Storage resources are billed for storing data in OBS or SFS. For details, see **Model Training Billing Items**.

Procedure

To create a training job, follow these steps:

Access the page for creating a training job. For details, see **Step 1: Accessing the Page for Creating a Training Job**.

Configure basic information, such as the runtime type, job name, description, and experiment. For details, see **Step 2: Configuring Basic Settings**.

Configure environment information, such as the algorithm type, boot mode, engine, and version. For details, see **Step 3: Configuring the Environment**.

Configure training parameters, including inputs, outputs, hyperparameters, and environment variables. For details, see **Step 4: (Optional) Configuring Training Settings**.

Configure the resource pool type, instance specifications, number of instances, and mounted storage. For details, see **Step 5: Configuring Resources**.

Configure auto restart (unconditional auto restart and restart upon suspension). For details, see **Step 6: Configuring HA**.

Set the job priority, preemption, and auto stop. For details, see **Step 7: Configuring Scheduling Parameters**.

Configure logs, event notifications, and tags. For details, see **Step 8: Configure Advanced Parameters**.

Submit a training job and view its status. For details, see **Step 9: Submitting a Training Job and Viewing Its Status**.

Step 1: Accessing the Page for Creating a Training Job

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs**.
3. Click **Create Training Job**. The new-version page is displayed by default. The following describes how to create a training job on the old-version page.

Step 2: Configuring Basic Settings

On the **Create Training Job** page, configure basic parameters.

Table 5-1 Basic parameters

Parameter	Description
Runtime Type	Select **Production**. Debug your training code either in the cloud or locally before using it to create production training jobs. • **Production**: Run the training job in the production environment. You are advised to start with the debug mode to save resources. • **Debug**: Modify and debug your training code in real time.
Name	Job name, which is mandatory. The system automatically generates a name, which you can then rename according to the following rules. • The name contains 1 to 64 characters. • Letters, digits, hyphens (-), and underscores (_) are allowed.
Description (Optional)	Job description, which helps you learn about the job information in the training job list. Enter 0 to 256 characters. Only letters, digits, spaces, hyphens (-), underscores (_), commas (,), and periods (.) are supported.
Experiment	Specifies whether to organize training jobs into experiments for better management. It helps manage multiple job versions efficiently. Experiments help manage and optimize training jobs. For example, after fine-tuning hyperparameters, you can sort and compare job results in an experiment to find the optimal training configuration. • **Use existing**: Select an existing experiment to add the training job to the experiment. • **Create new**: Enter an experiment name and description to add the training job to the new experiment. If you do not enable this feature, the job will not be managed in any experiment.

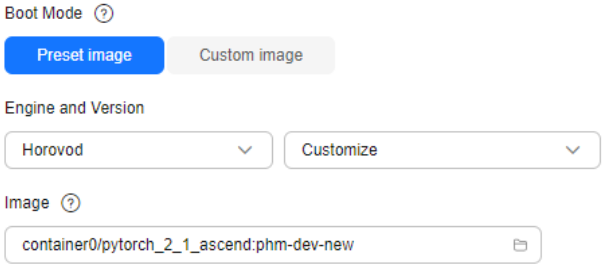
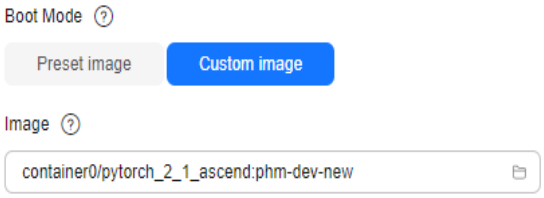
Step 3: Configuring the Environment

When creating a training job, configure the algorithm source, boot mode, image engine and version, and code directory. Training jobs can have various algorithm types.

- **Custom algorithm**: Create a training job using a preset image or a custom image.

Table 5-2 Environment settings (custom algorithm)

Parameter	Description
Algorithm Type	Select **Custom algorithm**. This parameter is mandatory.
Boot Mode	This parameter is mandatory when **Algorithm Type** is set to **Custom algorithm**. Options: • **Preset image**: Create a training job using a preset training framework and image. • **Custom image**: Create a training job using a custom image. If the software in the preset images cannot meet your needs, you can use a custom image for training. This image must be uploaded to SWR beforehand. For details about how to create an image, see **Preparing a Model Training Image**.
Engine and Version	If **Boot Mode** is set to **Preset image**, you need to select the required engine and version. Ensure that the framework of the AI engine you select is the same as the one you use for writing algorithm code. For example, if PyTorch is used for writing algorithm code, select PyTorch when you create a job.

Parameter	Description
Image	Select a container image for training. For details about the training image creation requirements, see **Preparing a Model Training Image**. • If **Boot Mode** is set to **Preset image** and the engine version is set to **Customize**, you need to select a proper image from the container images. **Figure 5-2** Boot Mode > Preset image  • If the **Boot Mode** is set to **Custom image**, you need to select a proper image from the container images. **Figure 5-3 Boot Mode** > **Custom image**  You can set the container image path in either of the following ways: • To select your image or an image shared by others, click **Select** on the right and select a container image for training. The required image must be uploaded to SWR beforehand. • To select a public image, enter the address of the public image in SWR. Enter the image path in the format of "Organization name/Image name:Version name". Do not contain the domain name (swr.<region>.myhuaweicloud.com) in the path because the system will automatically add the domain name to the path. For example, if the SWR address of a public image is **swr.<region>.myhuaweicloud.com/test-image/tensorflow2_1_1:1.1.1**, enter **test-images/tensorflow2_1_1:1.1.1**.

Parameter	Description
Code Source	Select the code source. The default value is **Object Storage Service – Bucket**. • **Object Storage Service – Bucket**: Select **Object Storage Service – Bucket** if the training code is stored in an OBS bucket.
Code Directory	This parameter is available only when **Code Source** is set to **OBS**. Select the OBS directory where the training code file is stored. This parameter is mandatory when **Boot Mode** is set to **Preset image**. This parameter is optional when **Boot Mode** is set to **Custom image**. • Upload code to the OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1,000, and the folder depth cannot exceed 32. If there is a pre-trained model, put it in the code directory. • The training code file is automatically downloaded to the **${MA_JOB_DIR}/demo-code** directory of the training container when the training job is started. **demo-code** is the last-level OBS directory for storing the code. For example, if **Code Directory** is set to **/test/code**, the training code file is downloaded to the **${MA_JOB_DIR}/code** directory of the training container. **NOTE** Encrypt sensitive data before saving it to your OBS bucket.
Boot File	Select or enter the Python boot script of the training job in the code directory. This parameter is mandatory when **Boot Mode** is set to **Preset image**. This parameter is not required when **Boot Mode** is set to **Custom image**. ModelArts supports only the boot file written in Python. Therefore, the boot file must end with .py.

Parameter	Description

Boot Command	Command for booting an image. This parameter is not required when **Boot Mode** is set to **Preset image**. This parameter is mandatory when **Boot Mode** is set to **Custom image**.
	When a training job is running, the boot command is automatically executed after the code directory is downloaded.
	● If the training boot script is a .py file, **train.py** for example, the boot command is as follows:
	`python ${MA_JOB_DIR}/demo-code/train.py`
	● If the training boot script is a .sh file, **main.sh** for example, the boot command is as follows:
	`bash ${MA_JOB_DIR}/demo-code/main.sh`
	You can use semicolons (;) and ampersands (&&) to combine multiple commands. **demo-code** in the command is the last-level OBS directory where the code is stored. Replace it with the actual one.
	If there are input pipes, output pipes, or hyperparameters, ensure that the last command of the boot command runs the training script.
	Reason: The system appends input pipes, output pipes, and hyperparameters to the end of the boot command. If the last command is not the training script, an error will occur.
	Example: If the last line of the boot command is **python train.py** and the **--data_url** hyperparameter exists, the system executes **python train.py --data_url=/input** when running properly. However, if the boot command ends with another command, such as:
	`python train.py`
	`pwd  # The last command is pwd instead of the training script.`
	The system will execute **python train.py pwd --data_url=/input**, leading to an error.
	NOTE
	To ensure data security, do not enter sensitive information, such as plaintext passwords.

User ID	ID of the user who runs the container. This parameter is not required when **Boot Mode** is set to **Preset image**. This parameter is optional when **Boot Mode** is set to **Custom image**. 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** The default value **1000** is recommended. If the user ID is set to **0**, the user in the training container is **root**.

Parameter	Description
Local Code Directory	This parameter is available in **More Configurations** only when **Code Source** is set to **OBS**. This parameter is optional. This parameter specifies the local directory of the training container. When training starts, the code directory is downloaded to this directory. The default local code directory is **/home/ma-user/modelarts/user-job-dir**. Cannot be under **/home/ma-user/modelarts/***, **/home/ma-user/modelarts-dev/***, **/home/ma-user/infer/***, or **/home/ma-user**. Click **Preview Runtime Environment** in the upper right corner of the page to view the work directory of the training job.
Container Execution Directory	Specify the local directory of the training container. During training, the system automatically runs the **cd** command to execute the boot file in this directory. It is the local directory where the boot command is executed during the training job. This directory can store generated temporary files. This directory must be the parent directory of the boot file's local directory.

- **My algorithm**: Use an algorithm in Algorithm Management to create a training job.

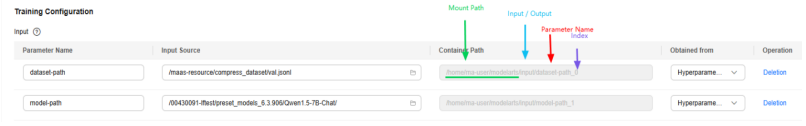
  Set **Algorithm Type** to **My algorithm** and select an algorithm from the algorithm list. If no algorithm meets the requirements, you can create an algorithm. For details, see **Creating an Algorithm**.

Step 4: (Optional) Configuring Training Settings

When creating a training job, you must configure inputs, outputs, hyperparameters, and environment variables of the training job.

Table 5-3 Training settings

Parameter	Description

Input	Click **Add** and configure training inputs.
	• Parameter name
	The algorithm code reads the training input data based on the input parameter name.
	The recommended value is **data_url**. The training input parameters must match the input parameters of the selected algorithm.
	• **Input Source**: The training input supports OBS storage and datasets.
	– Click next to **Input Source** and select the storage path to the training input data from an OBS bucket. Files must not exceed 10 GB in total size, 1,000 in number, or 1 GB per file.
	– Click **Dataset** and select the target dataset and its version in the ModelArts dataset list.
	When the training job is started, ModelArts automatically downloads the data in the input path to the training container.
	• **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", "")`
	• **Container Path**: local directory of the container to which the data input channel is mapped.
	– The local path of **Input** cannot be modified. It is automatically set by the system.
	– The value combines the parameter name, local code path, index, and input or output type using this format: Local code path + input/output + parameter name + "_" + index.
	Figure 5-4 Example
	 

Parameter	Description

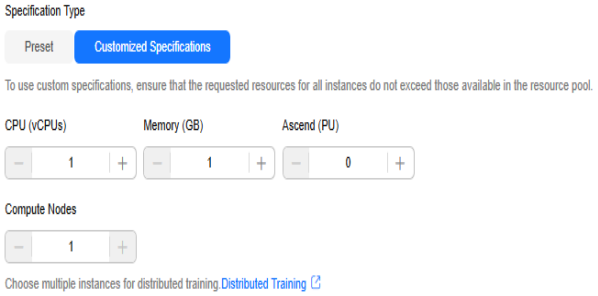
Output	Click **Add** and configure training outputs.
	• Parameter name
	The algorithm code reads the training output data based on the output parameter name.
	The recommended value is **train_url**. The training output parameters must match the output parameters of the selected algorithm.
	• **Data path**
	Click next to **Data path** and select the storage path to the training output data from an OBS bucket. Files must not exceed 1 GB in total size, 128 in number, or 128 MB per file.
	During training, the system automatically synchronizes files from the local code directory of the training container to the data path.
	Only OBS can be used to store output data. Choose an empty folder for your output files to prevent issues with stored training inputs.
	• **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", "")`
	• **Container Path**: local directory of the container to which the data output channel is mapped.
	– The local path of **Output** cannot be modified. It is automatically set by the system.
	– The value combines the parameter name, local code path, index, and input or output type using this format: Local code path + input/output + parameter name + "_" + index.
	• **Predownload to Container Directory**
	Choose whether to pre-download files in the output directory to the training container.
	– If you set this parameter to **No**, the system does not download the files in the training output path to the local directory of the training container when the training job is started.
	– If you set this parameter to **Yes**, the system automatically downloads the files in the training output path to the local directory of the training container when the training job is started. The larger the file size, the longer the download time. To avoid excessive training time, remove any unneeded files as soon

Parameter	Description
	as possible. Select **Yes** for **Incremental Model Training** or **Resumable Training**.
Hyperparameter	Used for tuning. This parameter is determined by the selected algorithm. If hyperparameters have been defined in the algorithm, all hyperparameters in the algorithm are displayed. Hyperparameters can be modified and deleted. The status depends on the hyperparameter constraint settings in the algorithm. For details, see **Table 4-6**. • Click **Add** to add hyperparameters. The total number of hyperparameters cannot exceed 100. • To import hyperparameters in batches, click **Upload**. You will need to fill in the hyperparameters based on the provided template. The total number of hyperparameters should not exceed 100, or the import will fail. **NOTE** To ensure data security, do not enter sensitive information, such as plaintext passwords.
Environment Variable	Add environment variables based on service requirements. For details about the environment variables preset in the training container, see **Managing Environment Variables of a Training Container**. • Click **Add** to add environment variables. The total number of environment variables cannot exceed 100. • To import environment variables in batches, click **Upload**. You will need to fill in the environment variables based on the provided template. The total number of environment variables should not exceed 100, or the import will fail. **NOTE** To ensure data security, do not enter sensitive information, such as plaintext passwords.
Automated Hyperparameter Search	If you select **My algorithm** for **Algorithm Type** and the selected algorithm supports the **autoSearch(S)** policy, you can click **More Configurations** to show **Automated Hyperparameter Search**. Selecting it enables automated hyperparameter search during training, potentially increasing the time needed. For details, see **Overview**.

Step 5: Configuring Resources

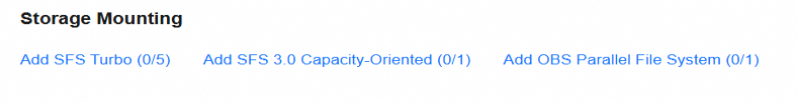
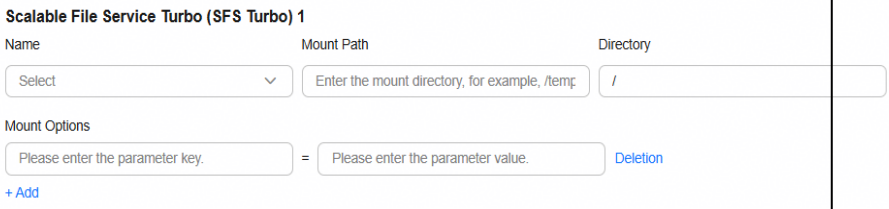
When creating a training job, you need to choose training resources. Select **Public resource pool** or **Dedicated resource pool**. Select a resource pool as required. A dedicated resource pool is recommended. For details about the differences between dedicated and public resource pools, see **Differences between dedicated resource pools and public resource pools**.

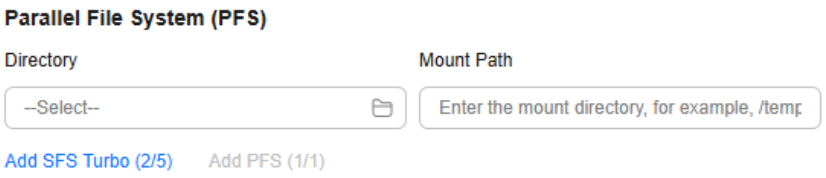
Table 5-4 Resource parameters

Parameter	Description
Source of resources	• **Public resource pool**: The public resource pool is available for all tenants and does not require user creation. • **Dedicated resource pool**: Dedicated resource pools are created separately and used exclusively. For details, see **Creating a Dedicated Resource Pool**.
Resource Pool	This parameter appears only for dedicated resource pools. In the **Resource Pool** section, click **Select Resource Pool** and choose your desired dedicated resource pool or logical subpool from the menu on the right. Click **OK**. You can view the dedicated resource pool name, node pool specifications, number of available nodes/maximum number of nodes, number of available NPU/GPUs, available CPUs (vCPUs), available memory (GiB), and resource fragments. Hover over **View** in the **Resource Fragment** column to check fragment details and check whether the resource pool meets the training requirements. Once you choose a resource pool, its details appear. To choose a different one, click **Reselect**.
Specification Type	This parameter is displayed when you select a dedicated resource pool. The following specifications types are supported: • **Preset**: Select preset instance specifications in the dedicated resource pool. Ensure that the selected flavor has sufficient disk space to download the input file. • **Customized Specifications**: Training jobs allow custom resource specifications using dedicated pools to enhance their utilization. You can set **CPU (vCPUs)**, **Memory (GB)**, **Ascend (PU)**, and **Compute Nodes** as required. Custom specifications must match or stay below the node specifications of the dedicated resource pool. • For CPU specifications, you can only customize the number of vCPUs and memory. For GPU and Ascend specifications, you can customize the number of vCPUs, memory, and PUs. **Figure 5-5** Specifications 

Parameter	Description
Specifications	For a dedicated resource pool, choose both a resource pool and its preset specifications. For a public resource pool, simply choose the specifications in the right pane and click **OK**. • The upper part displays the CPU, NPU, and GPU specifications. The lower part displays details such as the specification name, memory, and reference price. • The specification name contains the number of PUs, processor model, CPU and GPU information, and memory size. • If some resource types are invisible or unavailable for selection, they are not supported. • If **Input** is configured, click **Check Input Size** next to resource pool specifications to ensure the storage is larger than the input data size. **NOTE** • The instance specifications **GPU:*n**tnt004** (*n* indicates a specific number) do not support multi-process training. • For a public resource pool: Use a custom image with the same image and resource types as your instance when creating a job, for example, only GPU types. Otherwise, the training job will fail.
Compute Nodes	Select the number of instances as required. The default value is **1**. • If only one instance is used, a single-node training job is created. ModelArts starts one training container on this node. The training container exclusively uses the compute resources of the selected specifications. • If more than one instance is used, a distributed training job is created. For more information about distributed training configurations, see **Overview**. **If the HA node feature is enabled for the cluster, the total number of nodes available for training job scheduling is the sum of the actual available nodes and the HA nodes. HA nodes are reserved by the system and cannot be directly scheduled by your jobs; they are used exclusively for fault recovery when a node abnormality occurs.** Before creating a distributed training job, pre-install all required pip dependencies (see **Installing pip Dependencies in an Image**). If there are more than 10 nodes, the system automatically deletes the pip source configuration. Executing pip install commands during training may cause training failures.

Parameter	Description
Specify Affinity Nodes	This parameter is supported only for dedicated resource pools. It allows you to configure supernode and node affinity for training jobs. Select the checkbox to enable it. When enabled, it allows fine-grained control over pod deployment strategies, including: strict placement (strong affinity), preferred placement (weak affinity), prohibited placement (strong anti-affinity), and avoided placement (weak anti-affinity). **Affinity Type**: Node affinity: Requires all instances of a training job to be scheduled on selected nodes, either strictly or preferentially. Node anti-affinity: Requires all instances of a training job to be avoided or strictly excluded from selected nodes. **Strength**: The degree of affinity. Weak: The system will try to place the pod on the specified node, but it is not guaranteed. Strong: The pod must be scheduled onto the specified node; otherwise, scheduling will not proceed. **Supernode Affinity Method**: Supported only for supernode resource pools. At the supernode level, currently only scenarios where all instances of a training job belong to one affinity group are supported. This is suitable for training jobs where traffic must not cross supernodes. **Random child nodes**: The system randomly schedules tasks to child nodes within the target supernode. **Specify child nodes**: The system schedules tasks to the specified child nodes. **Select Supernode**: Choose the supernode(s) to be configured. Supported only for supernode resource pools. **Select Node**: Choose the node(s) to be configured.

Parameter	Description
Storage Mounting	• When you select a dedicated resource pool, you can mount multiple storage types to improve data access efficiency. **Figure 5-6** Storage Mounting  – **Add SFS Turbo** When ModelArts and SFS Turbo are directly connected, multiple SFS Turbo file systems can be mounted to a training job to store training data. You can mount a file system multiple times, but each mount path must be distinct. A maximum of five disks can be mounted to a training job. For details, see **Configuring Network Passthrough Between ModelArts and SFS Turbo**. **Figure 5-7** SFS Turbo  – **Name**: Select an SFS Turbo file system. – **Mount Path**: Enter the SFS Turbo mounting path in the training container. The path cannot be a **/** directory or a system-mounted directory like **/cache** or **/home/ma-user/modelarts**. – **Directory**: Specify the SFS Turbo storage location. If you have configured the folder control permission, select a storage location. If you have not configured the folder control permission, retain the default value **/** or customize a location. – **Mounting Mode**: Permission on the mounted SFS Turbo file system. This parameter is displayed as **Read/Write** or **Read-only** based on the permission of the SFS Turbo storage location. If you have not configured the folder control permission, this parameter is unavailable. For details about how to set permissions for SFS Turbo folders, see **Permissions Management**. – **Mount Options**: Configure SFS mount parameters to accelerate and optimize training. For details about the parameters, see **Configuring SFS Turbo Mount Options**. Alternatively, retain the default settings below:

Parameter	Description
	mountOptions: - vers=3 - timeo=600 - nolock - hard **NOTE** Configuring SFS Turbo allows the frontend page to fetch the latest storage details and settings directly, ensuring valid and accurate data. 1. **Querying Details About a File System** 2. **Listing File Systems** • **Add SFS 3.0 Capacity-Oriented**: A training job can mount an SFS 3.0 file system to store training data. Set the parameters below. – **Name**: Enter the name of the file system. The name must be the same as that in SFS. Otherwise, the mounting fails. – **Mount Path**: Enter the cloud mount path in the training container. **NOTE** To add SFS 3.0 Capacity-Oriented, submit a service ticket. • **Add OBS Parallel File System**: A training job can mount an OBS parallel file system to store training data. Set the parameters below. **Figure 5-8** PFS  – **Storage Configuration**: Select a parallel file system. – **Mount Path**: Enter the cloud mount path in the training container. **NOTE** **Add OBS Parallel File System** is restricted. To use this function, submit a service ticket.

Parameter	Description
Supernode Affinity Group Instances	• Selecting a dedicated resource pool and an Snt9b23 flavor allows you to set **Supernode Affinity Group Instances**. • You can set this parameter if you select a supernode resource pool. If **Supernode Affinity Group Instances** is set to N, every N pods are scheduled to the same supernode to schedule affinity jobs. In distributed training, affinity job scheduling ensures uniformity in the architecture of the allocated compute resources. • You must set the number of instances as an integral multiple of **Supernode Affinity Group Instances**. Otherwise, the training job cannot be created. • For more information about supernode affinity group instances, see **Configuring Supernode Affinity Group Instances**. • When using an Ascend Snt9b21 resource pool, you do not need to configure the number of supernode affinity group instances. Instead, you need to configure the environment variable **export HCCL_LOGIC_SUPERPOD_ID=${VC_TASK_INDEX}** in the training job startup command to use the parameter plane network.
Training Mode	ModelArts offers various training modes when using a MindSpore preset image with Ascend specifications. • Common mode: It is the default training scenario. • 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 mode: Certain O&M functions will be enabled or adjusted to collect more information for locating faults.

Step 6: Configuring HA

You can set auto restart for a training job when creating it.

Table 5-5 HA configuration

Parameter	Description
Auto Restart	Choose whether to enable automatic restart for a training job. • This function is disabled by default. If a training exception occurs, the job is directly stopped. • If this function is enabled, the system will handle any exceptions caused by environmental or suspension issues during a training job. The system automatically detects faults and processes them according to the corresponding policies, thereby increasing the training success rate. Training job recovery policies enable automatic restarts at the process, container, and job levels. These policies require no manual configuration as they are automatically applied and upgraded as needed. To avoid losing training progress, ensure your code can resume training from where it is interrupted, and then enable unconditional auto restart to optimize compute usage. For details, see **Resumable Training**. If auto restart is triggered during training, the system records the restart information. You can check the fault recovery details on the training job details page. For details, see **Training Job Fault Tolerance Check**.
Maximum Restarts	This parameter is available when **Auto Restart** is enabled. The training job will stop if it is still abnormal after maximum automatic restarts. • Default value: **3** • Value range: **1** to **128** The value cannot be changed once the training job is created. Set this parameter based on your needs.
Unconditional Auto Restart	This parameter is available when **Auto Restart** is enabled. If **Unconditional auto restart** is selected, the training job will be restarted unconditionally once the system detects a training exception. To prevent invalid restarts, the system limits unconditional restarts to three consecutive attempts.
Restart Upon Suspension	This parameter is available when **Auto Restart** is enabled. ModelArts continuously monitors job processes to detect suspension and optimize resource usage. When this feature is enabled, suspended jobs can be automatically restarted at the process level. CPU specifications do not support job restarts upon suspension. However, ModelArts does not verify code logic, and suspension detection is periodic, which may result in false reports. By enabling this feature, you acknowledge the possibility of false positives. To prevent unnecessary restarts, ModelArts limits consecutive restarts to three.

Step 7: Configuring Scheduling Parameters

When creating a training job, you can configure its scheduling policy. For example, you can increase its priority, enable preemption, or set it to stop automatically. These changes help boost scheduling efficiency.

Table 5-6 Scheduling settings

Parameter	Description
Job Priority	• When using a dedicated resource pool, you can set and change the scheduling priority of the training job. This parameter is not supported when a public resource pool is used. • The platform handles jobs by prioritizing them from highest to lowest. If multiple jobs share the same priority, they are scheduled in the order they were submitted. When resources are available, the earliest-submitted job gets processed first. • Changing the number changes the priority of the job in the queue. The priority can be set to **1**, **2**, or **3**. A larger number indicates a higher priority. The default priority is **1**, and the highest priority is **3**. • To set the priority to **3**, you will also need the permission. For details about how to set the permission, see **Assigning the Permission to Set the Highest Job Priority to an IAM User**. • If a training job is in the **Pending** state for a long time, you can change the job priority to reduce the queuing duration. For details, see **Priority of a Training Job**.
Preemption	• When using a dedicated resource pool, you can set this parameter. This parameter is not supported when a public resource pool is used. • When enabled, jobs that allow preemption may be terminated and re-queued if resource pool capacity is insufficient. To avoid losing training progress, configure resumable training before enabling this function. For details, see **Resumable Training**.
Auto Stop	Choose whether to enable **Auto Stop**. • This function is disabled by default, and the training job keeps running until the training is completed. • If you enable this function, set the auto stop time. The value can be **1 hour**, **2 hours**, **4 hours**, **6 hours**, or **Customize**. The customized time must range from 1 hour to 720 hours. When you enable this function, the training stops automatically when the time limit is reached. The time limit does not count down when the training is paused.

Step 8: Configure Advanced Parameters

Table 5-7 Advanced settings

Parameter	Description
Persistent Log Saving	This function is enabled by default when Ascend specifications are selected. This function is available when CPU or GPU specifications are selected. • If this function is enabled (default), configure **Log Path**. The platform permanently stores training logs to the specified OBS path. • If this function is disabled, ModelArts automatically stores the logs for 30 days. You can download all logs on the job details page to a local path.
Log Path	When **Persistent Log Saving** is enabled, you must configure a log path to store log files generated by the training job. Ensure that you have read and write permissions to the selected OBS directory. You can choose your own OBS bucket or enter a path. The path must start with **obs://** and end with a slash (/), like this: **obs://bucketname/path/**. For shared buckets from other users, you must enter the path.
Job Visibility	The options are **Workspace** and **Creator**. • **Workspace**: The created training job is visible to all users in the current workspace. • **Creator**: Only the creator can view the job by default. To access it, other users must request the **modelarts:trainJob:listAll** permission, which allows them to view all training jobs, including those limited to the creator.

Parameter	Description
Event Notification	Choose whether to enable event notification for the training job. • This function is deselected by default, which means SMN is disabled. • If this function is enabled, you will be notified of specific events, such as job status changes or suspected suspensions, via an SMS or email. Notifications will be billed based on SMN pricing. In this case, you must configure the topic name and events. – **Topic**: topic of event notifications. Click **Create Topic** to go to the SMN console to create a topic and add a subscription to the topic. You will receive event notifications only after the subscription status changes to **Confirmed**. For details, see **Adding a Subscription**. – **Event**: events you want to subscribe to. Examples: **JobStarted**, **JobCompleted**, **JobFailed**, **JobTerminated**, **JobHanged**, **JobRestarted**, and **JobPreempted**. **NOTE** • SMN charges you for the number of notification messages. For details, see **Billing**. • Only training jobs using GPUs or NPUs support **JobHanged** events.
Password-free SSH Between Instances	Choose whether to enable password-free SSH between instances. • This function is disabled by default, which means the password-free SSH file is not generated. • If this function is enabled, the password-free SSH file is generated. To enable distributed training with a custom MPI or Horovod-based image, set up password-free SSH trust between instances for seamless communication. Configure the password-free SSH file directory. This directory stores the auto-generated SSH key files in the training container. By default, it is set to **/home/ma-user/.ssh**. For details, see **Configuring Password-free SSH Mutual Trust Between Instances for a Training Job Created Using a Custom Image**.
Tags	TMS's predefined tags are recommended for adding the same tag to different cloud resources. For details about how to use tags, see **Using TMS Tags to Manage Resources by Group**. You can add up to 20 tags to a training job.

Step 9: Submitting a Training Job and Viewing Its Status

After setting the parameters, click **Submit**.

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

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

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

FAQs

1. **How Do I View the Resource Usage of a Training Job in ModelArts?**

6 Distributed Model Training

6.1 Overview

Distributed Training

Distributed training speeds up deep learning by running tasks simultaneously across several compute nodes like servers or GPUs, enabling faster model training or handling bigger datasets. Distributed training splits a training task across multiple nodes, where each node processes a portion of the model. These nodes share their computed results through a communication system, enabling the full model to be trained efficiently. This approach greatly enhances training speed, particularly for complex models and large datasets.

ModelArts enables distributed training by automatically managing node communications and resources for effective parallel processing.

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

It supports the following two approaches:

1. Single-node multi-PU data parallelism (DP): Multiple GPUs work together on one server to speed up training using data parallelism. This approach maximizes the use of all available GPU resources on a single server.
2. Distributed data parallelism (DDP): Multiple servers work together, with each server using several GPUs to increase training capacity. It works well for handling large datasets or complex models.

Constraints

- 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 PUs.
- 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.

Billing

Model training in ModelArts uses compute and storage resources, which are billed. Compute resources are billed for running training jobs. Storage resources are billed for storing data in OBS or SFS. For details, see **Model Training Billing Items**.

Advantages and Disadvantages of Single-Node Multi-PU Training Using DataParallel

- 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.

Advantages of Multi-Node Multi-PU Training Using DistributedDataParallel

- Fast communication
- Balanced load
- Fast running speed

Related Chapters

- **Creating a Single-Node Multi-PU Distributed Training Job (DataParallel)**: describes single-node multi-PU training using DataParallel, and corresponding code modifications.
- **Creating a Multiple-Node Multi-PU Distributed Training Job (DistributedDataParallel)**: describes multi-node multi-PU training using DistributedDataParallel, and corresponding code modifications.
- **Example: Creating a DDP Distributed Training Job (PyTorch + GPU)**: describes the procedure and code example of distributed debugging adaptation.
- **Example: Creating a DDP Distributed Training Job (PyTorch + NPU)**: provides a complete code sample of distributed parallel training for the classification task of ResNet18 on the CIFAR-10 dataset.
- **Debugging a Training Job**: describes how to use the SDK to debug a single-node or multi-node training job on the ModelArts development environment.

6.2 Creating a Single-Node Multi-PU Distributed Training Job (DataParallel)

As deep learning models grow larger, their training times increase. Efficient parallel computing methods are essential for faster training. A major challenge in single-node setups is maximizing the use of multiple GPUs. This section explains how to conduct single-node, multi-GPU data parallel training using PyTorch. Properly splitting data and syncing models across devices allows full utilization of GPU resources, significantly boosting training speed.

For details about distributed training with the MindSpore engine, visit the **MindSpore official website**. You can select the required version in the upper left corner.

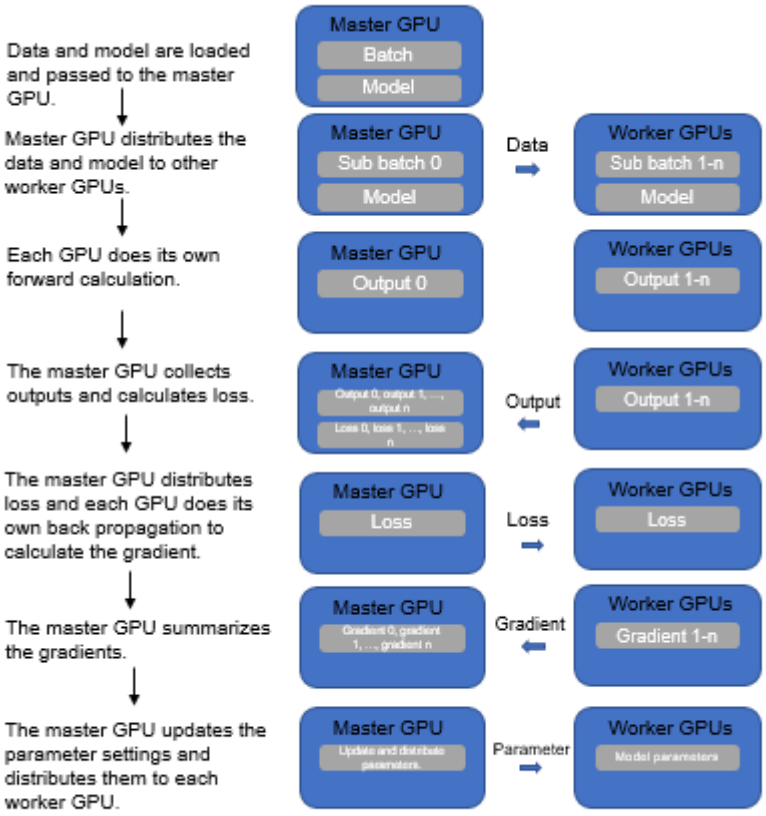
Training Process

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

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

The detailed flowchart is as follows.

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



Code Modifications

Model distribution: DataParallel(model)

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

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

model = Net().cuda()

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

6.3 Creating a Multiple-Node Multi-PU Distributed Training Job (DistributedDataParallel)

As models grow larger and datasets expand in deep learning, single-node training becomes insufficient. Implementing efficient multi-node, multi-PU training is now essential.

This section explains how to use PyTorch for multi-node, multi-PU data parallel training. It includes practical code adaptations and full examples to help you learn

and apply these methods. Using ResNet18 on the CIFAR-10 dataset for image classification, this section demonstrates Distributed Data Parallel (DDP) implementation with reproducible steps for easy reference.

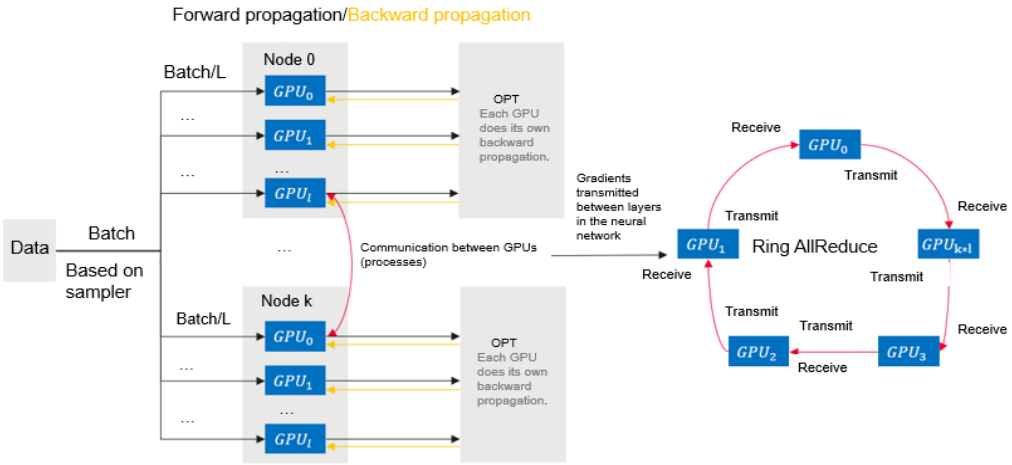
Training Process

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

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

The detailed flowchart is as follows.

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



Code Modifications

- Multi-process startup
- New variables such as rank ID and world_size are used along with the TCP protocol.
- Sampler for data distribution to avoid duplicate data between different processes

- Model distribution: DistributedDataParallel(model)
- Model saved in GPU 0

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

model = Net().cuda()

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

Multi-Node Distributed Debugging Adaptation and Code Example

DDP is widely used for distributed data parallel training. Each process loads batch data independently, computes gradients, and averages them across all processes to produce the final result. Since DDP processes more samples, its gradients are more accurate. This allows for higher learning rates, speeding up model convergence.

This section offers full code examples for both single-node and distributed training using DDP for ResNet18 on the CIFAR10 dataset. The code works for multi-node distributed training and supports both CPU and GPU environments. By commenting out specific sections, you can quickly switch between distributed and single-node training modes.

The training script includes three key parameter groups: general parameters, distributed parameters, and data parameters. Distributed parameters are auto-filled by the platform, requiring no manual setup. In data parameters, a **custom_data** toggle allows you to decide whether to train with randomly generated PyTorch data, offering flexibility for experimentation.

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 ... ###** 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-PU training can be executed. For details about the complete code, see **Code Example 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-PU and two-node 16-PU. The training duration and test set accuracy are as follows.

Table 6-1 Training result comparison

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

Code Example 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-PU 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 unzipped dataset folder (stored in **input_dir**).
   - To use user-defined random data, change the value of **custom_data** in the code example to **true**.

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

     Then, run **main.py**. The parameters for creating a training job are the same as those shown in the preceding figure.
2. **Why Can I Leave the IP Address of the Master Node Blank for DDP?**

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

6.4 Example: Creating a DDP Distributed Training Job (PyTorch + GPU)

PyTorch's DDP enables efficient distributed training. This section describes how to start DDP training with sample code provided.

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

  These methods offer adaptable solutions tailored to specific needs and setups.

  After reviewing this section, choose the right boot mode for your situation to start using PyTorch distributed training efficiently.

Creating a Training Job

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

  For details about parameters for creating a training job, see **Table 6-2**.

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

Parameter	Description
Algorithm Type	Select **Custom algorithm**.
Boot Mode	Select **Preset image** then **PyTorch**. Select a version as needed.
Code Directory	Select the training code path from your OBS bucket, for example, **obs://test-modelarts/code/**.
Boot File	Select the Python boot script of the training job in the code directory, for example, **obs://test-modelarts/code/main.py**.
Hyperparameter	To use a single-node multi-PU flavor, set the hyperparameters **world_size** and **rank**. If you choose a flavor with multiple instances, these hyperparameters are automatically set by ModelArts.

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

  For details about parameters for creating a training job, see **Table 6-3**.

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

Parameter	Description
Algorithm Type	Select **Custom algorithm**.
Boot Mode	Select **Custom image**.
Image	Select a PyTorch image for training.
Code Directory	Select the training code path from your OBS bucket, for example, **obs://test-modelarts/code/**.
Boot Command	Enter the Python boot command of the image, for example: `bash ${MA_JOB_DIR}/code/torchlaunch.sh`

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

  For details about parameters for creating a training job, see **Table 6-4**.

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

Parameter	Description
Algorithm Type	Select **Custom algorithm**.
Boot Mode	Select **Custom image**.
Image	Select a PyTorch image for training.
Code Directory	Select the training code path from your OBS bucket, for example, **obs://test-modelarts/code/**.
Boot Command	Enter the Python boot command of the image, for example: `bash ${MA_JOB_DIR}/code/torchrun.sh`

Code Example

Upload the following files to an OBS bucket:

```
code                    # Root directory of the code
 └─torch_ddp.py           # Code file for PyTorch DDP training
 └─main.py              # Boot file for starting training using the PyTorch preset image and the mp.spawn
command
 └─torchlaunch.sh         # Boot file for starting training using the custom image and the
```



```
torch.distributed.launch command
  └─torchrun.sh               # Boot file for starting training using the custom image and the
torch.distributed.run command
```

torch_ddp.py

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

# Start training by running mp.spawn.
def init_from_arg(local_rank, base_rank, world_size, init_method):
    rank = base_rank + local_rank
    dist.init_process_group("nccl", rank=rank, init_method=init_method, world_size=world_size)
    ddp_train(local_rank)

# Start training by running torch.distributed.launch or torch.distributed.run.
def init_from_env():
    dist.init_process_group(backend='nccl', init_method='env://')
    local_rank=int(os.environ["LOCAL_RANK"])
    ddp_train(local_rank)

def cleanup():
    dist.destroy_process_group()

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

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

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

main.py

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

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

if __name__ == "__main__":
    n_gpus = torch.cuda.device_count()
    world_size = n_gpus * args.world_size
    base_rank = n_gpus * args.rank
    # Call the start function in the DDP sample code.
```



```
    from torch_ddp import init_from_arg
    mp.spawn(init_from_arg,
        args=(base_rank, world_size, args.init_method),
        nprocs=n_gpus,
        join=True)
```

torchlaunch.sh

```
#!/bin/bash
# Default system environment variables. Do not modify them.
MASTER_HOST="$VC_WORKER_HOSTS"
MASTER_ADDR="${VC_WORKER_HOSTS%%,*}"
MASTER_PORT="6060"
JOB_ID="1234"
NNODES="$MA_NUM_HOSTS"
NODE_RANK="$VC_TASK_INDEX"
NGPUS_PER_NODE="$MA_NUM_GPUS"

# Custom environment variables to specify the Python script and parameters.
PYTHON_SCRIPT=${MA_JOB_DIR}/code/torch_ddp.py
PYTHON_ARGS=""

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

torchrun.sh

```
#!/bin/bash
# Default system environment variables. Do not modify them.
MASTER_HOST="$VC_WORKER_HOSTS"
MASTER_ADDR="${VC_WORKER_HOSTS%%,*}"
MASTER_PORT="6060"
JOB_ID="1234"
NNODES="$MA_NUM_HOSTS"
NODE_RANK="$VC_TASK_INDEX"
NGPUS_PER_NODE="$MA_NUM_GPUS"
# Custom environment variables to specify the Python script and parameters.
PYTHON_SCRIPT=${MA_JOB_DIR}/code/torch_ddp.py
PYTHON_ARGS=""
CMD="python -m torch.distributed.run \
    --nnodes=$NNODES \
    --node_rank=$NODE_RANK \
    $EXT_ARGS \
    --nproc_per_node=$NGPUS_PER_NODE \
    --rdzv_id=$JOB_ID \
    --rdzv_backend=static \  # In PyTorch 2.1, you must set rdzv_backend to static: --rdzv_backend=static.
    --rdzv_endpoint=$MASTER_ADDR:$MASTER_PORT \
    $PYTHON_SCRIPT \
    $PYTHON_ARGS
    "
echo $CMD
```

6.5 Example: Creating a DDP Distributed Training Job (PyTorch + NPU)

PyTorch's DDP enables efficient distributed training on Ascend accelerator cards. Implementing this with custom images and boot commands can pose challenges.

This guide offers a solution: configure custom images and boot commands to run PyTorch DDP training on Ascend accelerator cards. It supports user-specific needs and adapts to various training scenarios.

With this approach, users can streamline PyTorch DDP training for better performance on Ascend accelerator cards.

Prerequisites

An Ascend accelerator card resource pool is available.

Creating a Training Job

The following table describes the parameters you need to configure during training job creation.

Table 6-5 Parameters for creating a training job

Parameter	Description
Algorithm Type	Select **Custom algorithm**.
Boot Mode	Select **Custom image**.
Image	Select a custom image for training.
Code Directory	Select the code directory required for this training job, for example, **obs://test-modelarts/ascend/code/** in this case.
Boot Command	Python boot command of the image, for example, **bash ${MA_JOB_DIR}/code/run_torch_ddp_npu.sh** in this case. For details about the complete code of the boot script, see **Code Example**.

(Optional) Enabling Ranktable Dynamic Routing

To use ranktable dynamic routing for network acceleration, contact technical support to enable cabinet scheduling permission. For details, see **Enabling Dynamic Route Acceleration for Training Jobs**.

Code Example

The following shows an example boot script of a training job.

NOTE

To store the generated plog data, you need to specify the path in the startup script as **/home/ma-user/modelarts/log/modelarts-job-{*id*}/worker-{*index*}/**. The system will automatically upload the ***.log** file in the **/home/ma-user/modelarts/log/** directory to the OBS log directory of your training job. The system will only upload the log files (larger than 0 MB) in the local directory to the corresponding parent directory. Unlike MindSpore, PyTorch NPU plog logs are organized by worker instead of rank ID. PyTorch NPU does not rely on the ranktable file.

```
#!/bin/bash

# load env variables
source /usr/local/Ascend/ascend-toolkit/set_env.sh

# MA preset envs
MASTER_HOST="$VC_WORKER_HOSTS"
MASTER_ADDR="${VC_WORKER_HOSTS%%,*}"
NNODES="$MA_NUM_HOSTS"
NODE_RANK="$VC_TASK_INDEX"
# also indicates NPU per node
NGPUS_PER_NODE="$MA_NUM_GPUS"

# self-define, it can be changed to >=10000 port
MASTER_PORT="38888"

# replace ${MA_JOB_DIR}/code/torch_ddp.py to the actual training script
PYTHON_SCRIPT=${MA_JOB_DIR}/code/torch_ddp.py
PYTHON_ARGS=""

export HCCL_WHITELIST_DISABLE=1

# set npu plog env
ma_vj_name=`echo ${MA_VJ_NAME} | sed 's:ma-job:modelarts-job:g'`
task_name="worker-${VC_TASK_INDEX}"
task_plog_path=${MA_LOG_DIR}/${ma_vj_name}/${task_name}

mkdir -p ${task_plog_path}
export ASCEND_PROCESS_LOG_PATH=${task_plog_path}

echo "plog path: ${ASCEND_PROCESS_LOG_PATH}"

# set hccl timeout time in seconds
export HCCL_CONNECT_TIMEOUT=1800

# replace ${ANACONDA_DIR}/envs/${ENV_NAME}/bin/python to the actual python
CMD="${ANACONDA_DIR}/envs/${ENV_NAME}/bin/python -m torch.distributed.launch \
    --nnodes=$NNODES \
    --node_rank=$NODE_RANK \
    --nproc_per_node=$NGPUS_PER_NODE \
    --master_addr=$MASTER_ADDR \
    --master_port=$MASTER_PORT \
    --use_env \
    $PYTHON_SCRIPT \
    $PYTHON_ARGS
"
echo $CMD
$CMD
```

6.6 Example: Creating a Ray Cluster

Users performing reinforcement learning (RL) training tasks typically require a Ray cluster to ensure smooth task execution.

The ModelArts environment supports running Ray distributed jobs by starting the Ray cluster before the training begins. To start a Ray cluster, execute the **ray start**

--head command on the master node, while worker nodes join the cluster by executing **ray start --address="master_ip:6379"**. ModelArts containers provide environment variables to distinguish whether the current node is a master or a worker node, enabling the execution of the appropriate commands. The following is a script example for starting a Ray cluster across multiple nodes:

```
#!/bin/bash
pkill -9 python
ray stop --force

# Total number of nodes used for training
NNODES=${VC_WORKER_NUM:-1}
# Number of NPUs per node
NPUS_PER_NODE=$(lspci | grep d80 | wc -l)
MASTER_ADDR=$(python -c "import os; print(os.getenv('VC_WORKER_HOSTS','127.0.0.1').split(',')[0])")

if [ "$VC_TASK_INDEX" == "0" ]; then
# Start the master node
    ray start --head --resources='{"NPU": '$NPUS_PER_NODE'}'

    while true; do
        ray_status_output=$(ray status)
        npu_count=$(echo "$ray_status_output" | grep -oP '(?<=/)\d+\.\d+(?=\s*NPU)' | head -n 1)
        npu_count_int=$(echo "$npu_count" | awk '{print int($1)}')
        device_count=$((npu_count_int / $NPUS_PER_NODE))

                # Check if device_count matches NNODES
        if [ "$device_count" -eq "$NNODES" ]; then
            echo "Ray cluster is ready with $device_count devices (from $npu_count NPU resources), starting
Python script."
            ray status
            break
        else
            echo "Waiting for Ray to allocate $NNODES devices. Current device count: $device_count"
            sleep 5
        fi
    done
else
    # Worker nodes attempt to register ray with the master node until the registration is successful
    while true; do
        # Attempt to connect to the Ray cluster.
        ray start --address="$MASTER_ADDR:6379" --resources='{"NPU": '$NPUS_PER_NODE'}'
        # Check if the connection was successful.
        ray status
        if [ $? -eq 0 ]; then
            echo "Successfully connected to the Ray cluster!"
            break
        else
            echo "Failed to connect to the Ray cluster. Retrying in 5 seconds..."
            sleep 5
        fi
    done
fi
```

Save the script above as a file (for example, **start_ray_cluster.sh**) and upload it to OBS.

To create a Ray cluster and start a training task via a custom method in ModelArts, simply execute the command **bash start_ray_cluster.sh**. After execution, you can verify the cluster status using the **ray status** command.

7 Enabling Dynamic Route Acceleration for Training Jobs

Distributed training faces performance issues because networks struggle with slow data transfer and poor bandwidth usage when exchanging information across multiple nodes. To address these challenges, ModelArts offers dynamic routing acceleration. It smartly optimizes network paths for training jobs, boosting overall performance. This guide explains how to enable this feature on ModelArts, covering both preset frameworks and custom images. It also includes setup instructions and tips to maximize your distributed training results.

Notes and Constraints

- Dynamic routing acceleration can only be enabled in the following training scenarios:
  - **Scenario 1: Using the Ascend-Powered-Engine Preset Image, MindSpore, and NPUs for Training**
  - **Scenario 2: Using a Custom Image, PyTorch, and NPUs for Training**
- The training job must use Python 3.7, 3.8, 3.9, or 3.10.
- Before enabling dynamic routing acceleration, contact ModelArts technical support to ensure that the cabinet plug-in and scheduling permissions of the cluster are enabled.

Scenario 1: Using the Ascend-Powered-Engine Preset Image, MindSpore, and NPUs for Training

When using the Ascend-Powered-Engine preset image to create a training job, refer to **Table 7-1** to create a training job and enable dynamic routing acceleration. The table below describes only key parameters. Configure other parameters based on actual needs.

Table 7-1 Using a preset image to create a training job

Step	Parameter	Description
Environment settings	Algorithm Type	Select **Custom algorithm**. You can use a custom algorithm or an algorithm subscribed to in AI Gallery to create a training job.
	Boot Mode	Select **Preset image**. Boot mode of the environment set when you create a training job.
	Engine and Version	Select **Ascend-Powered-Engine** and a MindSpore-related engine version.
	Code Directory	Select the OBS directory where the training code file is stored. Dynamic routing acceleration improves network communication by adjusting the rank ID. To prevent communication issues, unify the rank usage in the code.
	Boot File	Select the Python boot script of the training job in the code directory,
Training settings	Environment Variable	Add the following environment variables: `ROUTE_PLAN = true` Do not configure the environment variable **MA_RUN_METHOD**. Ensure that the boot file of the training job is started using the rank table file.
Resource settings	Resource Pool	Select a dedicated resource pool.
	Specifications	Select instance specifications that meet the following requirements: • All processing units (PUs) on each node must be fully utilized. Otherwise, the effectiveness of dynamic routing acceleration may be impacted. For example, if a node has eight PUs, all eight must be used. • The resources must be Ascend Snt9b or Snt9b23. Instance specifications contain the server type and model.
	Compute Nodes	Select at least three compute nodes.

Scenario 2: Using a Custom Image, PyTorch, and NPUs for Training

When using a custom image and Ascend resource pool to create a training job, refer to **Table 7-2** to create a training job and enable dynamic routing acceleration. The table below describes only key parameters. Configure other parameters based on actual needs.

Table 7-2 Using a custom image to create a training job

Step	Parameter	Description
Environment settings	Algorithm Type	Select **Custom algorithm**.
	Boot Mode	Select **Custom image**.
	Image	Select a custom image for training. The training image must use the PyTorch framework.
	Code Directory (Optional)	Select the OBS directory where the training code file is stored. Dynamic routing acceleration improves network communication by adjusting the rank ID. To prevent communication issues, unify the rank usage in the code.

	Boot Command	Enter the Python boot command of the image.
		Modify the following code in the training boot script. The values vary according to the NPU hardware.
		• Snt9b scenario
		`MASTER_ADDR="${MA_VJ_NAME}-${MA_TASK_NAME}-${MA_MASTER_INDEX:-0}.${MA_VJ_NAME}"`
		`NODE_RANK="${RANK_AFTER_ACC:-$VC_TASK_INDEX}"`
		• Snt9b23 scenario
		`MASTER_ADDR="${VC_WORKER_HOSTS%%,*}"`
		`NODE_RANK="${RANK_AFTER_ACC:-$VC_TASK_INDEX}"`

Training settings	Environment Variable	Add the following environment variables: `ROUTE_PLAN = true`
Resource settings	Resource Pool	Select a dedicated resource pool.
	Specifications	Select instance specifications that meet the following requirements: • All processing units (PUs) on each node must be fully utilized. Otherwise, the effectiveness of dynamic routing acceleration may be impacted. For example, if a node has eight PUs, all eight must be used. • The resources must be Ascend Snt9b or Snt9b23.
	Compute Nodes	Select at least three compute nodes.

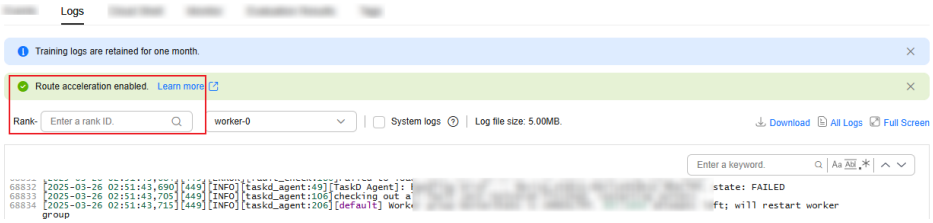
Viewing Training Logs of Dynamic Route Acceleration

When using an Ascend resource pool for training, you can check whether the route is enabled and query the log information of each rank in the **Logs** tab of the training job details page.

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Build** > **Training**. (On the old console, choose **Model Training** > **Training Jobs**.)
3. In the training job list, click the target job to access its details page.
4. Click the **Logs** tab.

   You can view that dynamic routing has been enabled for the training job and search for logs by rank ID.

Figure 7-1 Viewing dynamic route acceleration logs



8 Incremental Model Training

What Is Incremental Training?

Incremental learning is a machine learning method that enables AI models to learn from new data without restarting the training process. It builds on existing knowledge, allowing the model to expand its capabilities and improve its performance over time.

Incremental learning allows for training on data in smaller chunks, reducing storage needs and alleviating resource constraints. It also conserves computing power and time, and lowers retraining costs.

Incremental training is ideal for these scenarios:

- Continuous data updates: It allows models to adapt to new data without retraining.
- Resource constraints: It is a more economical choice when retraining a model is too costly.
- Avoiding knowledge loss: It retains old knowledge while learning new information, preventing the model from forgetting what it has learned.

Incremental training is used in various fields, including natural language processing, computer vision, and recommendation systems. It makes AI systems more flexible and adaptable, allowing them to handle changing data in real-world environments.

Implementing Incremental Training in ModelArts

The checkpoint mechanism enables incremental training.

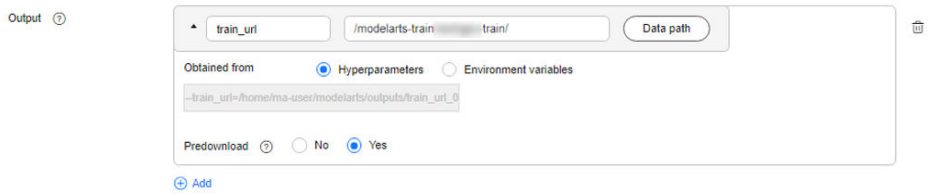
During model training, training results (including but not limited to epochs, model weights, optimizer status, and scheduler status) are continuously saved. To add data and resume a training job, load a checkpoint and use the checkpoint information to initialize the training status. To do so, add **reload ckpt** to the code.

To incrementally train a model in ModelArts, configure the training output.

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

the training output data path to a local directory of the training container before the training job is started.

Figure 8-1 Configuring training output



reload ckpt for PyTorch

- Use either of the following methods to save a PyTorch model.
  - Save model parameters only.
    ```
    state_dict = model.state_dict()
    torch.save(state_dict, path)
    ```
  - Save the entire model (not recommended).
    ```
    torch.save(model, path)
    ```
- Save the data generated during model training at regular intervals based on steps and time.

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

  ```
  checkpoint = {
          "net": model.state_dict(),
          "optimizer": optimizer.state_dict(),
          "epoch": epoch
  }
  if not os.path.isdir('model_save_dir'):
      os.makedirs('model_save_dir')
  torch.save(checkpoint,'model_save_dir/ckpt_{}.pth'.format(str(epoch)))
  ```
- Check the complete code example below.

  ```
  import os
  import argparse
  parser = argparse.ArgumentParser()
  parser.add_argument("--train_output", type=str)
  args, unparsed = parser.parse_known_args()
  args = parser.parse_known_args()
  # train_output is set to /home/ma-user/modelarts/outputs/train_output_0.
  train_output = args.train_output

  # Check whether there is a model file in the output path. If there is no file, the model will be trained from the beginning by default. If there is a model file, the CKPT file with the maximum epoch value will be loaded as the pre-trained model.
  if os.listdir(train_output):
      print('> load last ckpt and continue training!!')
      last_ckpt = sorted([file for file in os.listdir(train_output) if file.endswith(".pth")])[-1]
      local_ckpt_file = os.path.join(train_output, last_ckpt)
      print('last_ckpt:', last_ckpt)
      # Load the checkpoint.
      checkpoint = torch.load(local_ckpt_file)
      # Load the parameters that can be learned by the model.
      model.load_state_dict(checkpoint['net'])
      # Load optimizer parameters.
      optimizer.load_state_dict(checkpoint['optimizer'])
      # Obtain the saved epoch. The model will continue to be trained based on the epoch value.
      start_epoch = checkpoint['epoch']
  start = datetime.now()
  total_step = len(train_loader)
  for epoch in range(start_epoch + 1, args.epochs):
  ```



```
for i, (images, labels) in enumerate(train_loader):
    images = images.cuda(non_blocking=True)
    labels = labels.cuda(non_blocking=True)
    # Forward pass
    outputs = model(images)
    loss = criterion(outputs, labels)
    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    ...

# Save the network weight, optimizer weight, and epoch during model training.
checkpoint = {
      "net": model.state_dict(),
      "optimizer": optimizer.state_dict(),
      "epoch": epoch
    }
if not os.path.isdir(train_output):
    os.makedirs(train_output)
    torch.save(checkpoint, os.path.join(train_output, 'ckpt_best_{}.pth'.format(epoch)))
```


9 Automatic Model Tuning (AutoSearch)

9.1 Overview

ModelArts automatically searches for optimal hyperparameters for your models, saving time and effort.

During training, hyperparameters like **learning_rate** and **weight_decay** need to be adjusted. ModelArts hyperparameter search optimizes these settings automatically, outperforming manual tuning in speed and precision.

ModelArts supports the following hyperparameter search algorithms:

- Bayesian Optimization (SMAC)
- Tree-structured Parzen Estimator (TPE)
- Simulated Annealing

Bayesian Optimization (SMAC)

Bayesian optimization assumes a functional relationship between hyperparameters and the objective function. It estimates the mean and variance of objective function values at other search points using Gaussian process regression, based on the evaluation values of the searched hyperparameters. The mean and variance are then used to construct the acquisition function, which identifies the next search point as its maximum value. Bayesian optimization reduces the number of iterations and search time by leveraging previous evaluation results, but it can struggle to find the global optimal solution.

Table 9-1 Bayesian optimization parameters

Parameter	Description	Recommended Value
num_samples	Number of hyperparameter groups to search	This integer ranges from 10 to 20. Larger values increase search time but improve results.
kind	Acquisition function type	This string defaults to **ucb**. Other options are **ei** and **poi**, but it is best to stick with the default.

Parameter	Description	Recommended Value
kappa	Adjustment parameter for the **ucb** acquisition function, representing the upper confidence boundary	This float value should remain unchanged.
xi	Adjustment parameter for **poi** and **ei** acquisition functions.	This float value should remain unchanged.

Tree-structured Parzen Estimator (TPE)

The TPE algorithm uses a Gaussian mixture model to learn model hyperparameters. On each trial, TPE fits two Gaussian mixture models: one to the best objective values and another to the remaining values. It selects the hyperparameter value that maximizes the ratio of these two models.

Table 9-2 TPE parameters

Parameter	Description	Recommended Value
num_samples	Number of hyperparameter groups to search	This integer ranges from 10 to 20. Larger values increase search time but improve results.
n_initial_points	Number of random evaluations of the objective function before using tree-structured parzen estimators	This integer should remain unchanged.
gamma	Quantile used by the TPE algorithm to split l(x) and g(x)	This float value ranges from 0 to 1 and should remain unchanged.

Simulated Annealing

The simulated annealing algorithm is a simple and effective search method that uses the smoothness of the response surface. It starts with a previous trial point and samples each hyperparameter from a distribution similar to the prior, but with a higher concentration around the chosen point. Over time, the algorithm focuses on sampling points closer to the best ones. Occasionally, it may select a runner-up trial as the best to avoid local optima with a certain probability.

Table 9-3 Simulated annealing parameters

Parameter	Description	Recommended Value
num_samples	Number of hyperparameter groups to search	This integer ranges from 10 to 20. Larger values increase search time but improve results.
avg_best_idx	Mean of the geometric distribution used to select trials for exploration, based on their scores	This float value should remain unchanged.
shrink_coef	Rate at which the sampling neighborhood size decreases as more points are explored	This float value should remain unchanged.

9.2 Creating a Training Job for Automatic Model Tuning

Context

To use ModelArts hyperparameter search, the AI engine must be either **pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64** or **tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64**, and the hyperparameter to be optimized must be a float value.

To perform a hyperparameter search without any code modification, follow these steps:

1. **Preparations**
2. **Creating an Algorithm**
3. **Creating a Training Job**
4. **Viewing Details About a Hyperparameter Search Job**

Preparations

- Create a dataset in ModelArts or upload a training dataset to an OBS directory.
- Upload your training script to an OBS directory. For details about how to develop a training script, see **Developing Code for Training Using a Preset Image**.
- Print search indicator parameters in the training code.
- Create at least one empty folder in OBS for storing training outputs.
- Make sure your account is not in arrears, as training jobs consume resources.
- Make sure your OBS directory and ModelArts are in the same region.

Creating an Algorithm

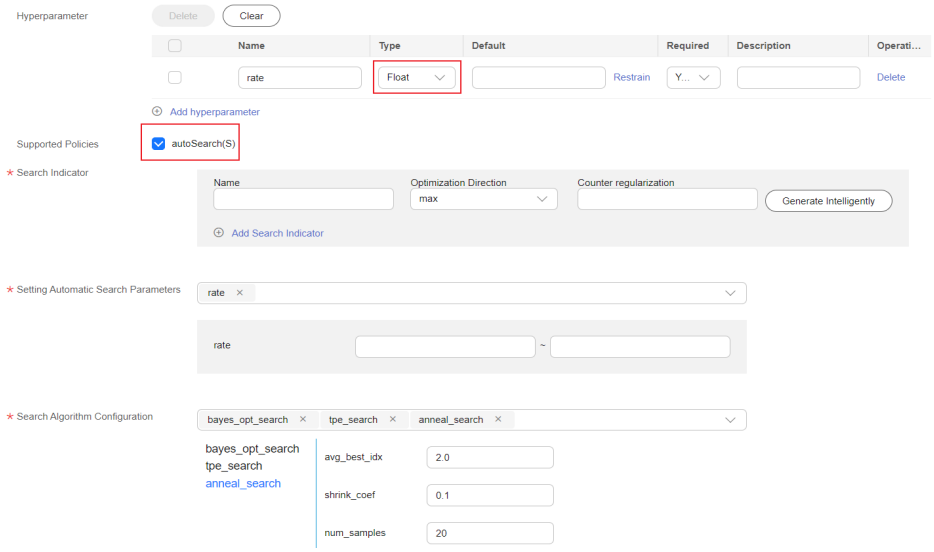
Log in to the **ModelArts console** and create a custom algorithm by referring to **Creating an Algorithm**. The image must use the **pytorch_1.8.0-cuda_10.2-**

py_3.7-ubuntu_18.04-x86_64 or **tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64** engine.

To define hyperparameters for optimization, specify the name, type, default value, and constraints in **Hyperparameter**. For details, see **Table 4-6**.

To enable auto search for the algorithm, select **autoSearch(S)**, print search parameters in the code, and configure the following parameters. ModelArts uses a regular expression to obtain search indicator parameters during an auto search, and then performs hyperparameter optimization based on the specified optimization direction.

Figure 9-1 Enabling auto search



- Search Indicator

  The search indicator represents the value of the objective function, such as loss or accuracy. By optimizing and converging this value in the desired direction, you can find the optimal hyperparameters to enhance model accuracy and convergence speed.

  Table 9-4 Search indicator parameters

Parameter	Description
Name	Enter a search indicator name. This value must be identical to the search indicator parameter in the code.
Optimization Direction	Select **max** or **min**.
Counter regularization	Enter a regular expression or click **Generate Intelligently** to generate a regular expression automatically.

- Setting Automatic Search Parameters

Select hyperparameters from the **Hyperparameters** configuration. Only float-type hyperparameters are supported. After selecting **autoSearch(S)**, set the value range.

- Search Algorithm Configuration

  ModelArts has three built-in algorithms for hyperparameter search. You can select one or more algorithms as needed. The algorithms and their parameter description are as follows:

  - bayes_opt_search: **Bayesian Optimization (SMAC)**
  - tpe_search: **Tree-structured Parzen Estimator (TPE)**
  - anneal_search: **Simulated Annealing**

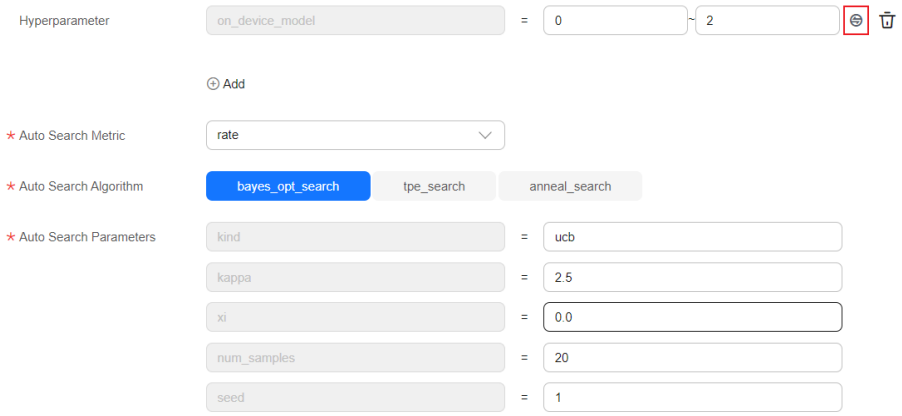
After creating the algorithm, use it to create a training job.

Creating a Training Job

Log in to the **ModelArts console** and create a training job by referring to **Creating a Training Job**.

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

Figure 9-2 Enabling hyperparameter search



After enabling the hyperparameter search, you can configure the search indicator, search algorithm, and search algorithm parameters. These parameters have the same values as the hyperparameters of the algorithm you created.

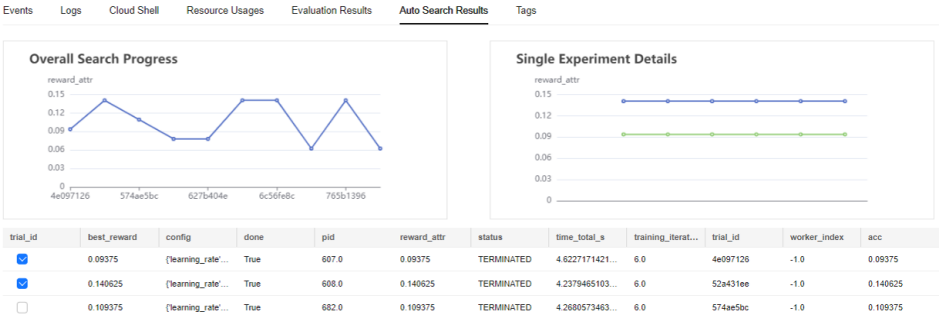
The hyperparameter search job will take some time to run after it is created.

Viewing Details About a Hyperparameter Search Job

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

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

Figure 9-3 Hyperparameter search results



10 High Model Training Reliability

10.1 Resumable 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.

The checkpoint mechanism enables resumable training.

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.

Implementing Resumable Training via Training Output in ModelArts

New version:

To implement resumable training or incremental training in ModelArts, you are advised to use storage mounts.

When creating a training job, you can save and load checkpoint files by mounting a storage path. The procedure is as follows:

1. In the training job settings, mount the storage directory (where checkpoints are stored) to a local directory within the training container.
2. During the training process, save checkpoint files to the mounted local directory. The data will automatically synchronize to the mounted path.
3. To resume from a breakpoint, ensure the mounted storage directory contains the previous checkpoint files. Your training script will then automatically load the latest checkpoint to continue the training.

Using storage mounts ensures persistent data storage and enables model reuse across different training jobs.

When you create a training job in ModelArts, you can choose any of the storage mount options below. The table below shows different storage choices for easy selection based on your needs.

Table 10-1 Comparison of storage mount options

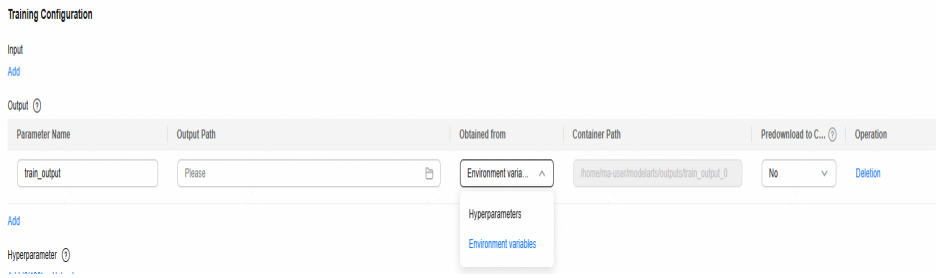
Storage Type	Performance	Capacity	Scenario	Price	Remarks
SFS Turbo	High	Large	SFS Turbo is suitable for AI training, AI generated content, autonomous driving, rendering, EDA simulation, and enterprise NAS applications.	Relatively high	General
OBS	Medium	Large	Using OBS to decouple storage from compute in big data scenarios.	Moderate	High-frequency read and low-frequency write

Old version:

To resume model training or incrementally train a model in ModelArts, configure training output.

When creating a training job, set the training **Output** parameter name to **train_output**. You can then retrieve this parameter via environment variables or hyperparameters. Once configured, checkpoints can be saved to the specified data storage location. Ensure that **Predownload** is set 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 10-1 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.

Reloading Checkpoints in the VeRL Framework

VeRL is a flexible, efficient, and widely used reinforcement learning training library, serving as the de facto standard framework for post-training. VeRL is an open-source implementation of the paper **HybridFlow: A Flexible and Efficient RLHF Framework**.

1. Configure **trainer.save_freq** and **trainer.default_local_dir** in the VeRL training YAML file.

   VeRL uses the **trainer.default_local_dir** parameter to specify the output directory. Within this directory, multiple weight subdirectories named **global_steps**_*xx* will be created. The **trainer.save_freq** parameter determines the frequency of weight saving, allowing checkpoints to be stored every set number of steps.

2. Configure **trainer.resume_mode** in the VeRL training YAML file.

   When **trainer.resume_mode** is set to **auto**, VeRL automatically scans the **trainer.default_local_dir** path to load the most recent and valid checkpoint. Taking the **train_output** parameter from **Implementing Resumable Training via Training Output in ModelArts** as an example, the parameter settings are as follows:

   ```
   trainer.default_local_dir="${train_output}"
   trainer.resume_mode=auto
   ```

Reloading Checkpoints in the MindSpeed-LLM Framework

MindSpeed LLM is a distributed training framework for large language models (LLMs) based on the Ascend ecosystem. It aims to provide an E2E LLM training solution for Huawei **Ascend chip** ecosystem partners, including distributed pre-training, distributed instruction fine-tuning, and the corresponding development toolchain, such as data preprocessing, weight transformation, online inference, and baseline evaluation. As the flagship training framework for Ascend computing, it is deeply optimized for performance, particularly for large-scale parameters, large clusters, and Mixture-of-Experts (MoE) models. It is also compatible with Megatron-LM, allowing Megatron users to migrate smoothly.

1. Configure the **--save** and **--save-interval** parameters in MindSpeed-LLM.

   In the MindSpeed-LLM training startup script, the **--save** parameter specifies the output directory. This directory will contain multiple weight subdirectories named **iter**_*xx* and a **latest_checkpointed_iteration.txt** file that records the step count of the most recent saved weights. The **latest_checkpointed_iteration.txt** file is updated after every save. The **--save-interval** parameter defines the frequency of weight saving, ensuring checkpoints are stored every set number of steps.

2. Configure the **--load** parameter to match the **--save** path in MindSpeed-LLM.

   The **--load** parameter in the training startup script specifies the input directory. When the **--load** path is set to be identical to the **--save** path, the training task will automatically load the latest weights upon each restart. Taking the **train_output** parameter from **Implementing Resumable Training via Training Output in ModelArts** as an example, the parameter configuration is as follows:

   ```
   --save-interval 1000
   --save ${train_output}
   --load ${train_output}
   ```


Reloading Checkpoints in the LLaMA-Factory Framework

LLaMA-Factory is a popular open-source framework for training foundation models. You can easily fine-tune hundreds of models, such as language and multimodal ones, using either the CLI or WebUI. Built on Transformers and DeepSpeed, it works well with various open-source models.

1. Configure **output_dir** and **save_steps** in the LLaMA-Factory training YAML file.

   LLaMA-Factory uses the **output_dir** parameter to specify the output directory. Within this directory, multiple weight subdirectories named **checkpoint-***xxx* will be created. The **save_steps** parameter configures the frequency of weight saving.

2. Configure **resume_from_checkpoint** to match the **output_dir** path in the LLaMA-Factory training YAML file.

   The **resume_from_checkpoint** parameter explicitly specifies the checkpoint to be used for the current training session. If a valid checkpoint is provided, training resumes from it. However, if **resume_from_checkpoint** is set to the same path as **output_dir**, and **output_dir** itself is not a valid checkpoint directory (but rather a parent directory containing multiple checkpoints), additional steps (3 and 4) are required. Taking the **train_output** parameter from **Implementing Resumable Training via Training Output in ModelArts** as an example, the parameter settings are as follows:

   ```
   ### output
   output_dir: ${train_output}
   save_steps: 500

   ### train
   resume_from_checkpoint: ${train_output}
   ```

3. Create a **resume.py** script. This script requires the absolute path of the training configuration YAML file as an input. The specific code is shown below:

   ```
   import os
   import re
   import sys


   def update_resume_config(config_file): # Receives the configuration file path.
       # Read the configuration content
       with open(config_file, 'r', encoding='utf-8') as f:
           lines = f.readlines()

       resume_line_num = None
       resume_path = None

       # Locate the resume_from_checkpoint line
       for i, line in enumerate(lines):
           if line.strip().startswith('resume_from_checkpoint:'):
               resume_line_num = i
               # Extract the value
               parts = line.split(':', 1)
               if len(parts) > 1:
                   resume_path = parts[1].strip().strip('"\'')  # Remove quotes
               break

       # If not found or value is null, do nothing
       if resume_line_num is None or resume_path in (None, 'null', ''):
           return
   ```



```
    # Check the directory and find the latest checkpoint
    new_resume_path = None
    if os.path.isdir(resume_path):
        # Find all checkpoint-number folders
        checkpoint_pattern = re.compile(r'^checkpoint-(\d+)$')
        checkpoints = []

        for item in os.listdir(resume_path):
            item_path = os.path.join(resume_path, item)
            if os.path.isdir(item_path):
                match = checkpoint_pattern.match(item)
                if match:
                    step = int(match.group(1))
                    checkpoints.append((step, item_path))

        # If checkpoints are found, use the latest one
        if checkpoints:
            checkpoints.sort(key=lambda x: x[0])
            new_resume_path = checkpoints[-1][1]

    # Modify the configuration line
    indent = len(line) - len(line.lstrip())  # Preserve original indentation
    if new_resume_path:
        lines[resume_line_num] = f'{" " * indent}resume_from_checkpoint: {new_resume_path}\n'
    else:
        lines[resume_line_num] = f'{" " * indent}resume_from_checkpoint: null\n'

    # Write back to the file
    with open(config_file, 'w', encoding='utf-8') as f:
        f.writelines(lines)


if __name__ == "__main__":
    # Get the configuration file path from command line arguments
    if len(sys.argv) < 2:
        print("Usage: python resume.py <config_file_path>")
        sys.exit(1)
    config_file = sys.argv[1]  # Receive the abc.yaml passed from the command line
    update_resume_config(config_file)  # Execute by passing to the function
```

4. Modify the training startup script. Before executing the **llamafactory-cli train** command, run the **resume.py** script to update the YAML. The script will scan the path provided in **resume_from_checkpoint**, find the **checkpoint-***xxx* directory with the largest step number, and update the parameter to that absolute path. Example modification for **train_lora/deepseek3_lora_sft_kt.yaml** (where **WORK_DIR** is the working directory):

```
#!/bin/bash
...
...

python $WORK_DIR/resume.py $WORK_DIR/LLaMA-Factory/examples/train_lora/
deepseek3_lora_sft_kt.yaml
llamafactory-cli train $WORK_DIR/LLaMA-Factory/examples/train_lora/deepseek3_lora_sft_kt.yaml
```

reload ckpt for PyTorch

- Use either of the following methods to save a PyTorch model.
  - Save model parameters only.
    ```
    state_dict = model.state_dict()
    torch.save(state_dict, path)
    ```
  - Save the entire model (not recommended).
    ```
    torch.save(model, path)
    ```
- Save the data generated during model training at regular intervals based on steps and time.

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

```
checkpoint = {
        "net": model.state_dict(),
        "optimizer": optimizer.state_dict(),
        "epoch": epoch
}
if not os.path.isdir('model_save_dir'):
    os.makedirs('model_save_dir')
torch.save(checkpoint,'model_save_dir/ckpt_{}.pth'.format(str(epoch)))
```

- Check the complete code example below.

```
import os
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--train_output", type=str)
args, unparsed = parser.parse_known_args()
args = parser.parse_known_args()
# train_output is set to /home/ma-user/modelarts/outputs/train_output_0.
train_output = args.train_output

# Check whether there is a model file in the output path. If there is no file, the model will be trained from the beginning by default. If there is a model file, the CKPT file with the maximum epoch value will be loaded as the pre-trained model.
if os.listdir(train_output):
    print('> load last ckpt and continue training!!')
    last_ckpt = sorted([file for file in os.listdir(train_output) if file.endswith(".pth")])[-1]
    local_ckpt_file = os.path.join(train_output, last_ckpt)
    print('last_ckpt:', last_ckpt)
    # Load the checkpoint.
    checkpoint = torch.load(local_ckpt_file)
    # Load the parameters that can be learned by the model.
    model.load_state_dict(checkpoint['net'])
    # Load optimizer parameters.
    optimizer.load_state_dict(checkpoint['optimizer'])
    # Obtain the saved epoch. The model will continue to be trained based on the epoch value.
    start_epoch = checkpoint['epoch']
start = datetime.now()
total_step = len(train_loader)
for epoch in range(start_epoch + 1, args.epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.cuda(non_blocking=True)
        labels = labels.cuda(non_blocking=True)
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        ...

    # Save the network weight, optimizer weight, and epoch during model training.
    checkpoint = {
          "net": model.state_dict(),
          "optimizer": optimizer.state_dict(),
          "epoch": epoch
        }
    if not os.path.isdir(train_output):
        os.makedirs(train_output)
        torch.save(checkpoint, os.path.join(train_output, 'ckpt_best_{}.pth'.format(epoch)))
```


10.2 Training Job Fault Tolerance Check

Description

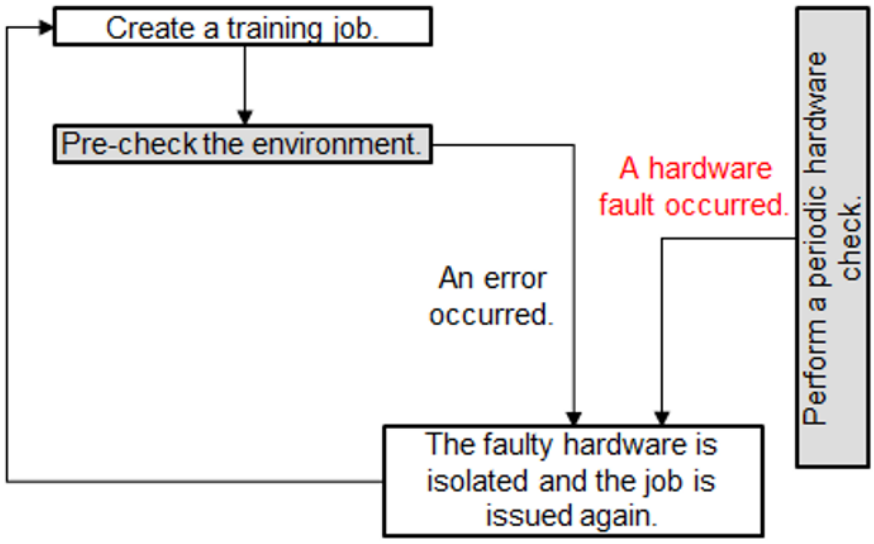
During model training, a training failure may occur due to a hardware fault. For hardware faults, ModelArts provides fault tolerance check to isolate faulty nodes to improve user experience in training.

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.

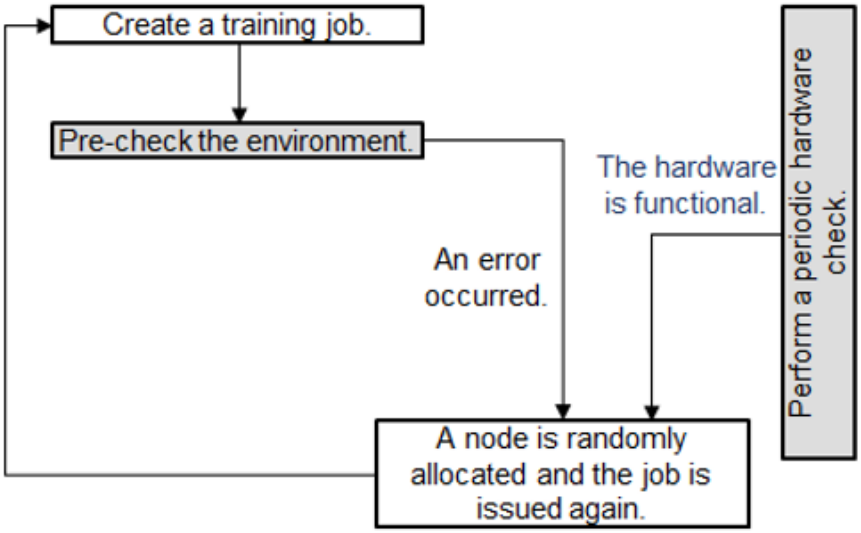
  Figure 10-2 Pre-check failure and hardware fault



Pre-check failure and hardware fault

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

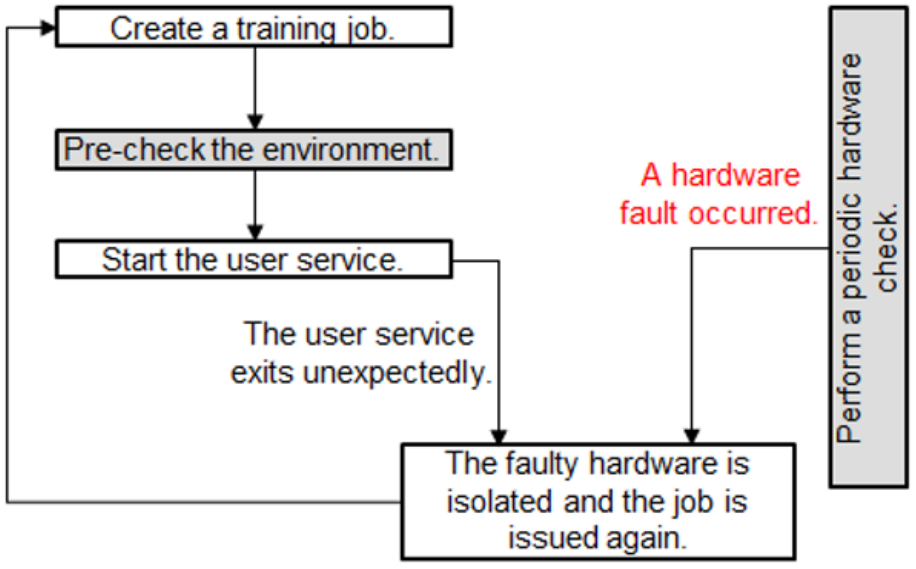
Figure 10-3 Pre-check failure but functional hardware



Pre-check failure and normal 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.

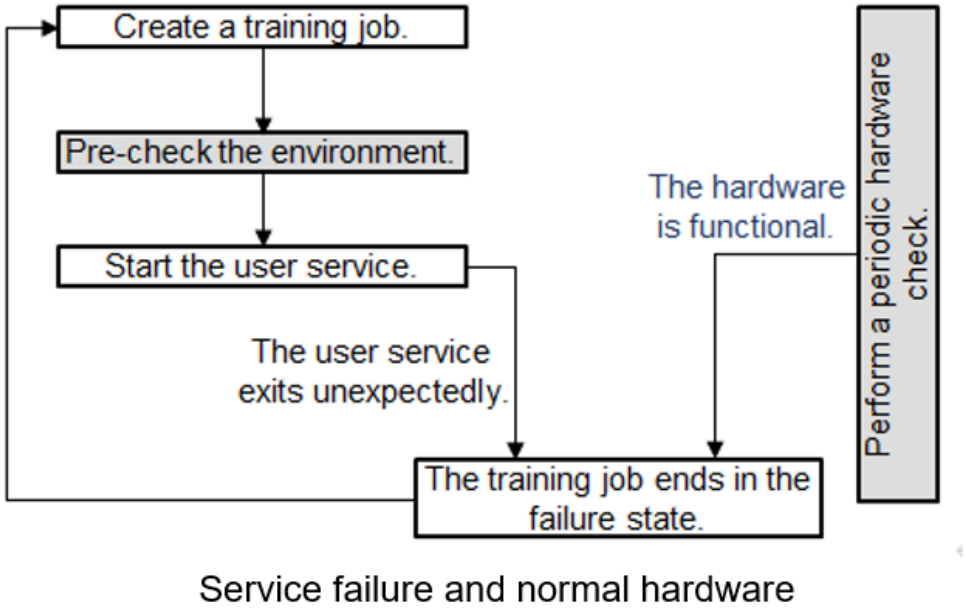
  Figure 10-4 Service failure and hardware fault



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, and the training job ends in the failure state.

Figure 10-5 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**.

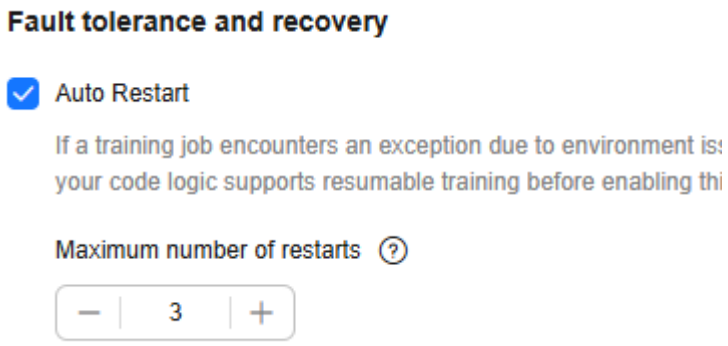
Enabling Fault Tolerance Check

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

- Configure fault tolerance check on the ModelArts management console:

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

Figure 10-6 Auto Restart



- Configure fault tolerance check using an API:

  Enable auto restart upon a fault using an API. When creating a training job, configure the **fault-tolerance/job-retry-num** field in **annotations** of the **metadata** field.

  If the **fault-tolerance/job-retry-num** field is added, auto restart is enabled. The value can be an integer ranging from **1** to **128**. specifying the maximum number of times that a job can be re-issued. If this hyperparameter is not specified, the default value **0** is used, indicating that the job will not be re-issued and the environment pre-check will not be enabled.

  Figure 10-7 Setting the API

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

Check Items and Conditions

Check Item	Item (Log Keyword)	Execution Condition	Requirements for a Check
Domain name detection	dns	None	The domain names of the volcano containers in the .host file in **/etc/volcano** are successfully resolved.
Disk size - Container root directory	disk-size root	None	The directory is greater than 32 GB.
Disk size - /dev/shm	disk-size shm	None	The directory is greater than 1 GB.

Check Item	Item (Log Keyword)	Execution Condition	Requirements for a Check
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.
- 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.

  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**.

Checking Fault Tolerance and Recovery Details

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

If you enable auto restart when creating a training job, you can view the number of restart times on the training job details page. **Restarts** displays the current number of restart times and the maximum number of restart times. In the **Fault tolerance and recovery** tab of the training job details page, you can view the restart details of the training job.

Figure 10-8 Restarts

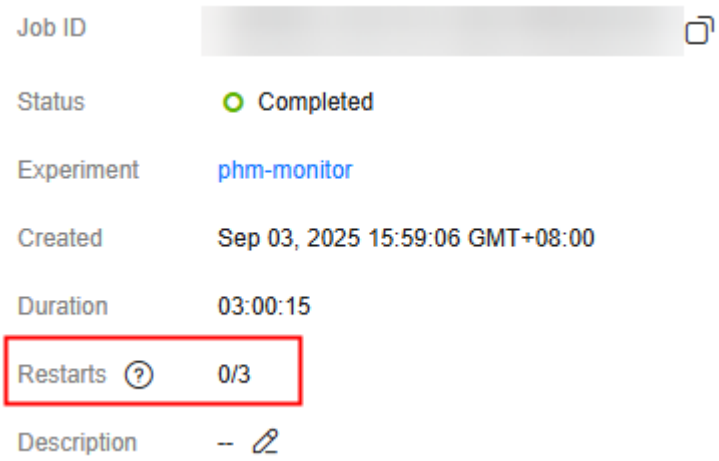
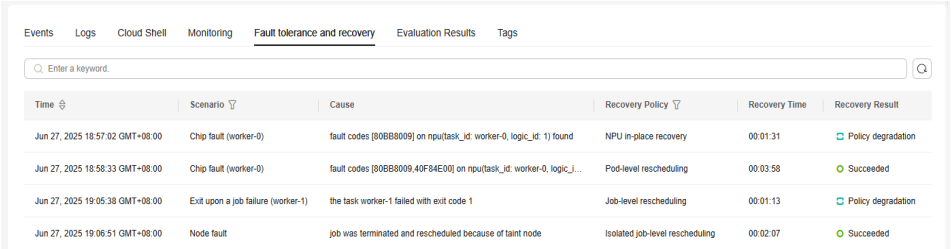


Figure 10-9 Fault tolerance and recovery



10.3 Detecting Training Job Suspension

Overview

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

Detection Rules

Suspension detection determines whether a job is suspended based on the monitored job process status and resource usage. A coroutine is started to periodically monitor the changes of the two metrics. There are two types of suspension detection rules: single-instance detection and all-instance detection. Both apply simultaneously.

- **Single-instance detection**
  - Process status: If the process I/O of a single instance of a training job changes, the next detection period starts. If the process I/O remains

unchanged in multiple detection periods, the resource usage detection starts.

- Resource usage: If the process I/O of a single instance of a training job remains unchanged, the system collects the GPU or NPU usage within a certain period of time and determines whether the resource usage changes based on the variance and median of the GPU or NPU usage within the period. If the GPU usage is not changed, the job is suspended.

- **All-instance detection**

  Resource usage: The system suspends a job if all its running instances' GPU or NPU usage remains unchanged for a while and each instance's CPU usage stays below one core.

The system has the environment variable **MA_HANG_DETECT_TIME** set to **30**. This means the job suspends if the system detects a metric issue for 30 minutes. To adjust this, update the value of the **MA_HANG_DETECT_TIME** variable. For details, see **Managing Environment Variables of a Training Container**.

CAUTION

- Due to the limitation of detection rules, there is a certain error probability in suspension detection. If the suspension is caused by the logic of job code (for example, long-time sleep), ignore it.

Constraints

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

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

1. **Data replication suspension**

   Symptoms

   The system stopped responding when **mox.file.copy_parallel** was called to copy data.

   Solution

   - Run the following commands to copy files or folders:

     ```
     import moxing as mox
     mox.file.set_auth(is_secure=False)
     ```

   - Run the following command to copy a single file that is greater than 5 GB:

     ```
     from moxing.framework.file import file_io
     ```

     Run **file_io._LARGE_FILE_METHOD** to check the version of the MoXing API. Output value **1** indicates V1 and **2** indicates V2.

Run **file_io._NUMBER_OF_PROCESSES=1** to resolve the issue for the V1 API.

To resolve the issue for the V2 API, use **file_io._LARGE_FILE_METHOD = 1** to switch to V1 and perform operations required in V1. Alternatively, use **file_io._LARGE_FILE_TASK_NUM=1** to resolve this issue.

- Run the following command to copy a folder:
  ```
  mox.file.copy_parallel(threads=0,is_processing=False)
  ```

2. **Suspension before training**

   If a job was trained on multiple nodes and suspension occurred before the job started, add **os.environ["NCCL_DEBUG"] = "INFO"** to the code to view the NCCL debugging information.

   - **Symptom 1**

     The job was suspended before the NCCL debugging information was printed in logs.

     Solution 1

     Check the code for parameters such as **master_ip** and **rank**. Ensure that these parameters are specified.

   - **Symptom 2**

     According to the distributed training logs, some nodes contain GDR information, but some nodes do not. The suspension may be caused by GDR.

     ```
     # Logs of node A
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-1:1136:1191 [2] NCCL INFO Channel 00 : 3[5f000] -> 10[5b000] [receive] via NET/IB/0/GDRDMA
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-1:1140:1196 [6] NCCL INFO Channel 00 : 14[e1000] -> 15[e9000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-1:1141:1187 [7] NCCL INFO Channel 00 : 15[e9000] -> 11[5f000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-1:1138:1189 [4] NCCL INFO Channel 00 : 12[b5000] -> 14[e1000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-1:1137:1197 [3] NCCL INFO Channel 00 : 11[5f000] -> 16[2d000] [send] via NET/IB/0/GDRDMA

     # Logs of node B
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-2:1139:1198 [2] NCCL INFO Channel 00 : 18[5b000] -> 19[5f000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-2:1144:1200 [7] NCCL INFO Channel 00 : 23[e9000] -> 20[b5000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-2:1142:1196 [5] NCCL INFO Channel 00 : 21[be000] -> 17[32000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-2:1143:1194 [6] NCCL INFO Channel 00 : 22[e1000] -> 21[be000] via P2P/IPC
     modelarts-job-a7305e27-d1cf-4c71-ae6e-a12da6761d5a-worker-2:1141:1191 [4] NCCL INFO Channel 00 : 20[b5000] -> 22[e1000] via P2P/IPC
     ```

     Solution 2

     Set **os.environ["NCCL_NET_GDR_LEVEL"] = '0'** at the beginning of the program to disable GDR or ask the O&M personnel to add the GDR information to the affected nodes.

   - **Symptom 3**

     Communication information such as "Got completion with error 12, opcode 1, len 32478, vendor err 129" was displayed. The current network was unstable.

     Solution 3

     Add the following environment variables:

- **NCCL_IB_GID_INDEX=3**: enables RoCEv2. RoCEv1 is enabled by default. However, RoCEv1 does not support congestion control on switches, which may lead to packet loss. In addition, later-version switches do not support RoCEv1, leading to a RoCEv1 failure.
- **NCCL_IB_TC=128**: enables data packets to be transmitted through the queue 4 of switches, which is RoCE-compliant.
- **NCCL_IB_TIMEOUT=22**: enables a longer timeout interval. Generally, there is a network interruption lasting about 5s if the network is unstable and then the timeout message is returned. Change the timeout interval to 22s, indicating that the timeout message will be returned in about 20s (4.096 μs x 2 ^ timeout).

3. **Suspension during training**
   - **Symptom 1**

     According to the logs of the nodes on which a training job ran, an error occurred on a node but the job did not exit, leading to the job suspension.

     Solution 1

     Check the error cause and rectify the fault.
   - **Symptom 2**

     The job was stuck in sync-batch-norm or the training speed was lowered down. If sync-batch-norm is enabled for PyTorch, the training speed is lowered down because all node data must be synchronized on each batch normalization layer in every iteration, which leads to heavy communication traffic.

     Solution 2

     Disable sync-batch-norm, or upgrade the PyTorch version to 1.10.
   - **Symptom 3**

     The job is stuck in TensorBoard, and the following error is reported:

     ```
     writer = SummaryWriter('./path/to/log')
     ```

     Solution 3

     Set a local path for storage, for example, **cache/tensorboard**. Do not store data in OBS.
   - **Symptom 4**

     When PyTorch DataLoader is used to read data, the job is stuck in data reading, and logs stop to update.

     Solution 4

     When the DataLoader is used to read data, reduce the value of **num_worker**.
4. **Suspension in the Last Training Epoch**

   Symptoms

   Logs showed that an error occurred in split data. As a result, processes are in different epochs, and uncompleted processes are suspended because they do not receive response from other processes. As shown in the following figure, some processes are in epoch 48 while others are in epoch 49 at the same time.


```
loss exit lane:0.12314446270465851
step loss is 0.29470521211624146
[2022-04-26 13:57:20,757][INFO][train_epoch]:Rank:2 Epoch:[48][20384/all] Data Time 0.000(0.000)
Net Time 0.705(0.890) Loss 0.3403(0.3792)LR 0.00021887
[2022-04-26 13:57:20,757][INFO][train_epoch]:Rank:1 Epoch:[48][20384/all] Data Time 0.000(0.000)
Net Time 0.705(0.891) Loss 0.3028(0.3466) LR 0.00021887
[2022-04-26 13:57:20,757][INFO][train_epoch]:Rank:4 Epoch:[49][20384/all] Data Time 0.000(0.147)
Net Time 0.705(0.709) Loss 0.3364(0.3414)LR 0.00021887
[2022-04-26 13:57:20,758][INFO][train_epoch]:Rank:3 Epoch:[49][20384/all] Data Time 0.000 (0.115)
Net Time 0.706(0.814) Loss 0.3345(0.3418) LR 0.00021887
[2022-04-26 13:57:20,758][INFO][train_epoch]:Rank:0 Epoch:[49][20384/all] Data Time 0.000(0.006)
Net Time 0.704(0.885) Loss 0.2947(0.3566) LR 0.00021887
[2022-04-26 13:57:20,758][INFO][train_epoch]:Rank:7 Epoch:[49][20384/all] Data Time 0.001 (0.000)
Net Time 0.706 (0.891) Loss 0.3782(0.3614) LR 0.00021887
[2022-04-26 13:57:20,759][INFO][train_epoch]:Rank:5 Epoch:[48][20384/all] Data Time 0.000(0.000)
Net Time 0.706(0.891) Loss 0.5471(0.3642) LR 0.00021887
[2022-04-26 13:57:20,763][INFO][train_epoch]:Rank:6 Epoch:[49][20384/all] Data Time 0.000(0.000)
Net Time 0.704(0.891) Loss 0.2643(0.3390)LR 0.00021887
stage 1 loss 0.4600560665130615 mul_cls_loss loss:0.01245919056236744 mul_offset_loss
0.44759687781333923 origin stage2_loss 0.048592399805784225
stage 1 loss:0.4600560665130615 stage 2 loss:0.048592399805784225 loss exit
lane:0.10233864188194275
```

Solution

Split tensors to align data.

10.4 Recovering a Training Job

Description

When creating a training job, you can **enable fault tolerance checks** by configuring automatic restarts. When a node failure occurs or the system detects an abnormality in the training job, the fault recovery mechanism is automatically triggered. This mechanism attempts to restore training services by restarting processes or rebuilding the job. The descriptions and differences of various recovery policies are shown in **Table 10-2**.

Constraints

1. Fault tolerance recovery recovers training services by restarting processes or rebuilding jobs. To avoid training progress loss and compute waste, ensure that the code has been adapted to resumable training before enabling this function. For details, see **Resumable Training**.
2. The constraints for different recovery policies are as follows:

Table 10-2 Comparison of recovery policies

Recovery Policy	Description	Failure Scenario	Constraints
In-place recovery	Restarts the training job within the original container without involving resource rescheduling.	• NPU chip failure self-healing (for example, HBM multi-bit ECC).	• Regular checkpoint (CKPT) saving is required. • Supports resumable training. • Retains the container environment from the time of failure; requires the job script to be re-entrant. • No standby nodes required.
Unconditional job-level rescheduling	Terminates all Pods associated with the job and completely rebuilds the job instance when a job exits with a non-zero error code.	• Occasional software failures or network fluctuations.	• Regular CKPT saving is required. • Supports resumable training. • Operations must support interruption and exit with a non-zero error code upon failure. • No standby nodes required.
Isolated job-level rescheduling	Based on unconditional job-level rescheduling, if a node failure is detected during a job anomaly, the faulty node is isolated before rescheduling.	• All NPU chip failures. • Node failures.	• Regular CKPT saving is required. • Supports resumable training. • Healthy standby nodes required.
Pod-level rescheduling	Retains the existing job instance, isolates the faulty node, and only recreates the faulty Pods.	• All NPU chip failures. • Node failures.	• Regular CKPT saving is required. • Supports resumable training. • Some instances retain the container environment from the time of failure; requires the job script to be re-entrant. • Healthy standby nodes required.

Recovery Policy	Description	Failure Scenario	Constraints
Restart upon suspension	If ModelArts detects a suspension state during job execution, it forcibly terminates user processes within the container. The container itself is not destroyed, and the training start command is re-executed. This does not involve resource rescheduling.	• Training job suspension detected by ModelArts.	• Regular CKPT saving is required. • Supports resumable training. • Retains the container environment from the time of failure; requires the job script to be re-entrant. • No standby nodes required.

Fault Recovery Trigger Modes

- Console settings

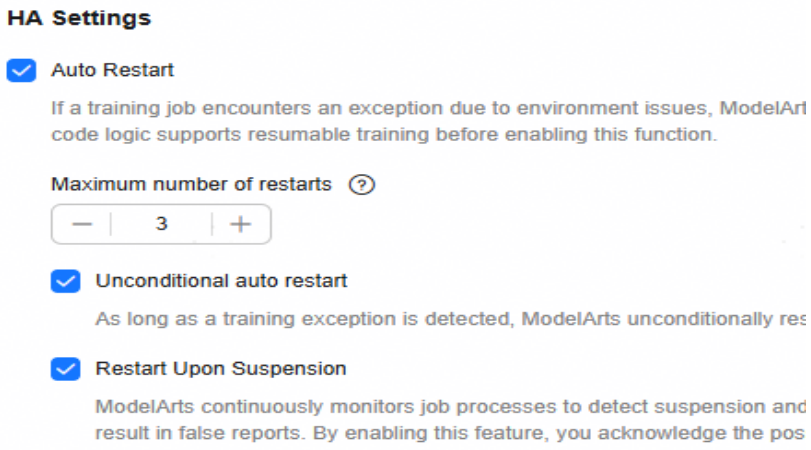
  When creating a training job, you can enable it based on **Step 6: Configuring HA**.

  Auto Restart: Includes in-place recovery and isolated job-level rescheduling. Note that in-place recovery does not consume **Maximum Restarts**.

  Unconditional Auto Restart: Includes unconditional job-level rescheduling.

  Restart Upon Suspension: Includes restarts triggered by a suspension state. Job suspension-triggered restarts do not consume **Maximum Restarts**.

  Figure 10-10 Fault tolerance and recovery

- API settings

  Auto restart: When creating a training job via the API, include the **fault-tolerance/job-retry-num** key within the annotations of the **metadata** field. Assign an integer between 1 and 128 to enable auto-restart and set the maximum retry attempts.

Unconditional auto restart: Assign a value to **fault-tolerance/job-retry-num** and set **fault-tolerance/job-unconditional-retry** to **true**.

Restart upon suspension: Assign a value to **fault-tolerance/job-retry-num** and set **fault-tolerance/hang-retry** to **true**.

Pod rescheduling: Assign values to both **fault-tolerance/job-retry-num** and **fault-tolerance/pod-retry-num**.

Parameter	Mandatory	Type	Description
annotations	No	Map<String,String>	**Description**: Advanced functions of a training job. **Constraints**: The options are as follows: • **fault-tolerance/job-retry-num**: **3** (number of retries upon a fault) • **fault-tolerance/job-unconditional-retry**: **true** (unconditional restart) • **fault-tolerance/hang-retry**: **true** (restart upon suspension) • "fault-tolerance/pod-retry-num": **3** (number of pod reschedules)

```
{
   "kind": "job",
   "metadata": {
      "annotations": {
         "fault-tolerance/job-retry-num": "8",
         "fault-tolerance/job-unconditional-retry": "true",
        "fault-tolerance/hang-retry": "true",
         "fault-tolerance/pod-retry-num": "3"
      }
   }
}
```

Fault Recovery Environment Variables

The following environment variables can be used to determine whether a job has undergone resumable, primarily ensuring the reentrancy of training scripts.

Table 10-3 Fault recovery environment variables – 1

Variable	Description
MA_SCHEDULE_CNT	Represents the number of times the Pod where the task resides has been scheduled. For a newly submitted job, the initial value is **1**. After each rescheduling recovery, **MA_SCHEDULE_CNT** increments by 1. Therefore, when **MA_SCHEDULE_CNT** > **1**, it indicates that the current container has undergone at least one rescheduling recovery.
MA_PROC_START_CNT	Represents the number of times your script has been executed within the current Pod. When a new job is submitted or a job script starts after rescheduling recovery, **MA_PROC_START_CNT** is reset to **1**. After each **in-place recovery** or **restart upon suspension**, this value increments by 1 when your script is executed again. Therefore, when **MA_PROC_START_CNT** > **1**, it indicates that your script has already been executed in the current container. If the process involves shared memory or data loading, this variable can be used to determine whether to reopen shared memory or skip data loading.

The following environment variable can be used to accelerate the speed of rescheduling.

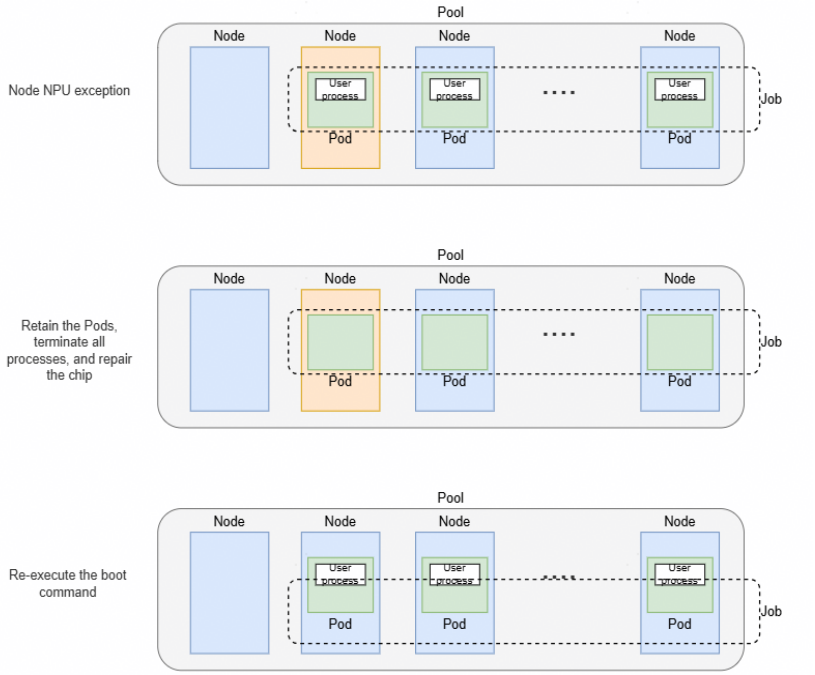
Table 10-4 Fault recovery environment variables – 2

Variable	Description
MA_FAILOVER_TERMINATION_GRACE_PERIOD_SECONDS	When set to a positive integer *N*, the volume unmounting step during the container termination phase will be set to asynchronous execution after *N* seconds. This effectively reduces the rescheduling recovery time for large-scale jobs. You are advised to set this to **10** in scenarios primarily using SFS Turbo storage. Default value: **-1**

In-Place Recovery

During the operation of NPU training jobs, chip failures may occur. Some of these failures can be self-healed through system repair or resetting. For such self-healable chip failures, the system forcibly terminates the user processes within the container. The container itself is not destroyed, thus preserving the runtime environment. After all processes are terminated, the system attempts NPU chip self-healing. If the fault is cleared and the chip returns to normal, all containers will re-execute the training job's startup command. In-place recovery does not involve resource rescheduling; it simply restarts the training job within the original containers. The process is illustrated in this figure.

Figure 10-11 In-place recovery



Trigger Scenarios

A self-healable fault occurs on the NPU chip.

Constraints

- The job must periodically save CKPTs.
- The job must support resumable training.
- In-place recovery preserves the container environment from the time of failure. This requires the training script to be reentrant. Typically, you need to skip **data downloading and preprocessing steps, and delete and rebuild shared memory with the same name**. You can use the **environment variable** **MA_PROC_START_CNT** to determine if an in-place recovery has occurred.

Degradation Policies

- If chip self-healing fails, the in-place recovery fails simultaneously. In this case, the node will be isolated, and the system will degrade to **isolated job-level rescheduling**.
- During a single training session, if the same fault code occurs 3 consecutive times within 24 hours on the same chip of the same node, the node will be isolated and the system will degrade to **isolated job-level rescheduling**.
- During a single training session, if the same fault code occurs 3 consecutive times within 24 hours on the same chip of the same node, and the fault is caused by your input (such as codes 80C98002 or 80CB8002), the node will not be isolated, and only rescheduling will be performed.

NOTE

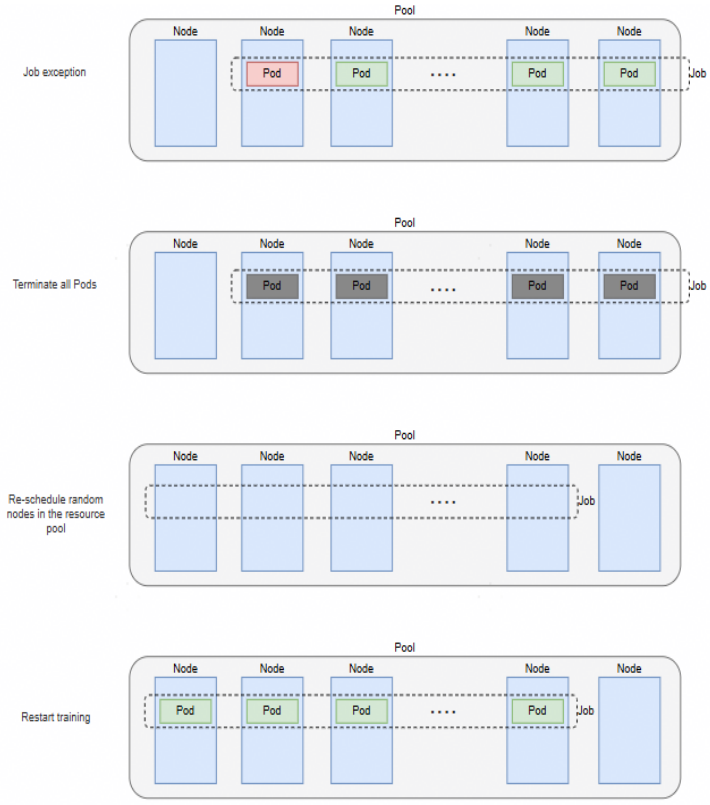
Self-healable Faults: Typically fault codes with Minor or Major severity levels. Warning levels do not require processing, while Critical levels cannot be self-healed.

User-Induced Faults: NPU failures caused by operator anomalies or input data. These usually require troubleshooting the CANN version, operator implementation, or data integrity.

Unconditional Job-Level Rescheduling

During the training process, unexpected situations may occur that lead to training failure and prevent the job from restarting in a timely manner, thereby extending the training cycle. Unconditional job-level rescheduling is designed to avoid such issues, improving the training success rate and job stability. When a job terminates abnormally with a non-zero exit code, Unconditional job-level rescheduling terminates all Pods associated with the job and completely rebuilds the job instance. The process is illustrated in this figure.

Figure 10-12 Unconditional job-level rescheduling

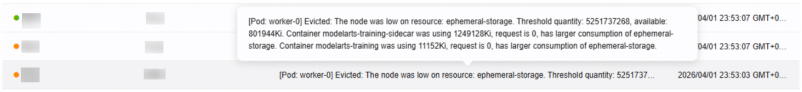


Trigger Scenarios

- Degradation caused by user-input-related faults during **in-place recovery**.
- Job failure or interruption with a non-zero exit code.
- Degradation resulting from a failed **pod-level rescheduling** attempt.

WARNING

Currently, unconditional job-level rescheduling cannot be triggered when a job is evicted. In this case, the job status changes to **Failed**. You are advised to first check for software code issues.



Constraints

- The job must periodically save CKPTs.
- The job must support resumable training.
- Operations must support interruption and exit with a non-zero error code upon failure. If the process cannot interrupt and remains in a running state during an anomaly, job-level rescheduling will not be triggered.

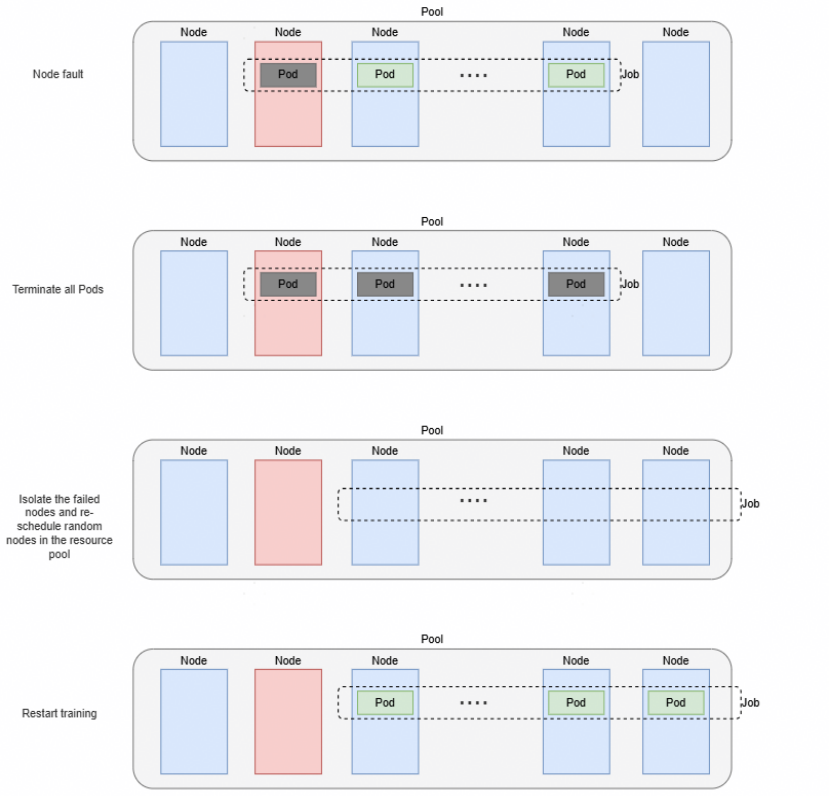
Degradation Policies

If the system triggers unconditional job rescheduling three times in a row and the issue persists on the fourth attempt with no clear node or chip faults, the system assumes the user's code has errors. Consequently, it stops the job and marks it as failed.

Isolated Job-Level Rescheduling

Isolated job-level rescheduling builds upon unconditional job-level rescheduling. If a node failure is detected when a job encounters an anomaly, the system will isolate the faulty node before performing rescheduling. The process is illustrated in this figure.

Figure 10-13 Isolated job-level rescheduling



Trigger Scenarios

- Degradation caused by non-user-input factors during **in-place recovery**.
- Occurrence of a node failure.

Constraints

- The job must periodically save CKPTs.
- The job must support resumable training.

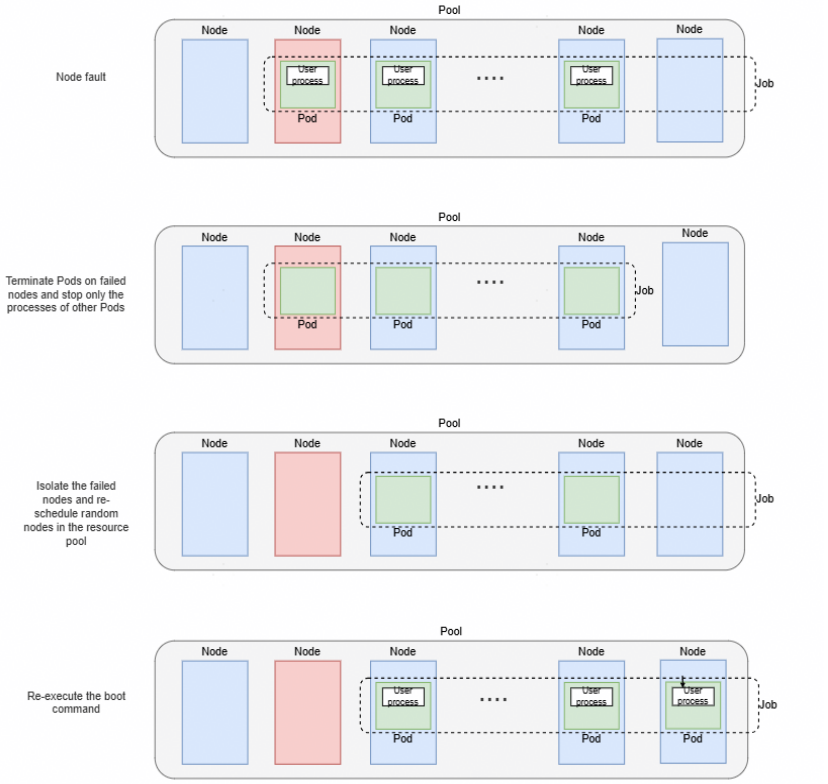
Degradation Policies

None.

Pod-Level Rescheduling

Compared to **isolated job-level rescheduling**, which deletes and rebuilds the entire job, pod-level rescheduling keeps the existing job instance. It isolates the faulty node first and only recreates the specific Pods that were affected by the failure.

Figure 10-14 Pod-level rescheduling



Trigger Scenarios

- Degradation caused by non-user-input factors during **in-place recovery**.
- Occurrence of a node failure.

Constraints

- The job must periodically save CKPTs.
- The job must support resumable training.
- Pod-level rescheduling preserves the container environment of the healthy instances. This requires the training script to be reentrant. Typically, you need to skip **data downloading or preprocessing steps, and delete and rebuild shared memory with the same name**.

Degradation Policies

If pod-level rescheduling fails, the system will degrade to **Unconditional Job-Level Rescheduling** (since the faulty node has already been isolated).

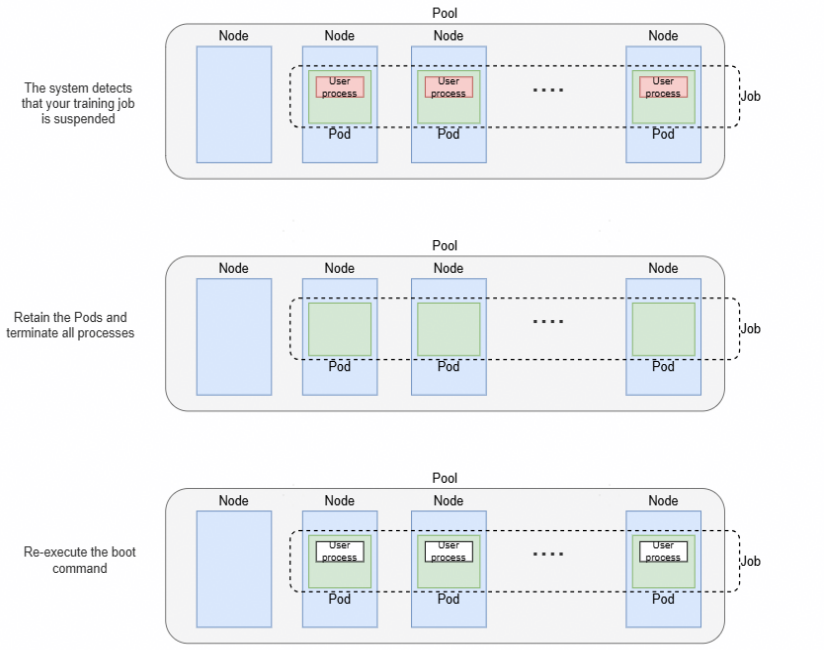
Restart upon Suspension

If a stable training job runs properly for a while and then gets suspended without a hardware fault, you can restart it to fix the issue. However, because a suspended training job cannot automatically terminate its container, it cannot directly trigger **job-level rescheduling**; instead, restart upon suspension must be configured. When restart upon suspension is enabled, ModelArts monitors the job's status

during runtime. Upon detecting a suspension, it forcibly terminates your processes within the container. The container itself is not destroyed, thus preserving the runtime environment. Once the processes are stopped, the system re-executes the training job's startup command. This does not involve resource rescheduling.

For details about the suspension detection rules, see **Detecting Training Job Suspension**. The process is illustrated in this figure.

Figure 10-15 Restart upon suspension



Trigger Scenarios

- The system detects a suspension in an NPU or GPU training job.

Constraints

- The job must periodically save CKPTs.
- The job must support resumable training.
- Restart upon suspension preserves the container environment from the time of the incident. This requires the training script to be reentrant, which typically involves handling operation logic such as **data downloading, data preprocessing, and the creation of shared memory with the same name**.

Degradation Policies

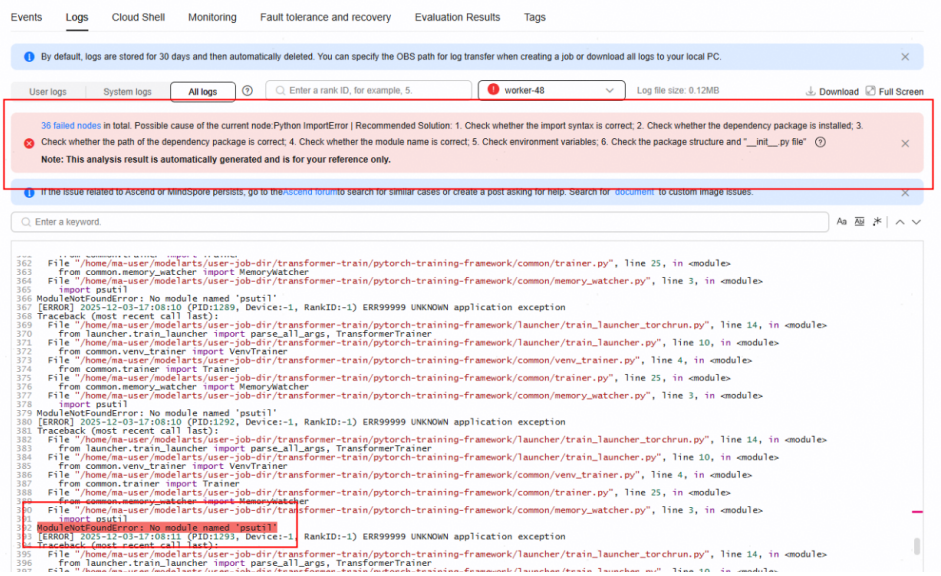
If restart upon suspension is triggered 3 consecutive times, the system will automatically terminate the job and set it to a failed status.

10.5 Training Log Failure Analysis

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 **All logs** page. The message consists of possible causes, recommended solutions, and error logs (marked in red).

Figure 10-16 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: Copy 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.

11 Configuring Supernode Affinity Group Instances

Description

In foundation model training, multiple parallel strategies boost efficiency and scalability using distributed computing. Model parallelism uses AllReduce communication, while MoE expert parallelism uses all-to-all communication. Both require high network bandwidth between processing units (PUs). These communication steps often become bottlenecks due to hardware limitations, limiting training performance.

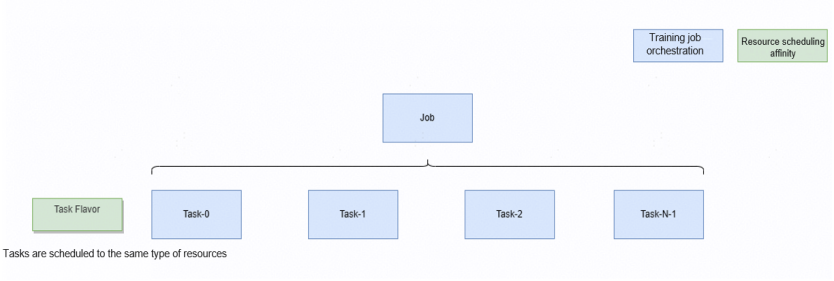
Snt9b23 uses the HCCS bus to connect NPUs across multiple compute nodes, creating supernodes. Within each supernode, a fully interconnected network known as a superplane enhances AI task communication. With supernode hardware, models can adjust parallel strategies like model parallelism or MoE expert parallelism to speed up training with higher bandwidth.

ModelArts provides a **supernode affinity feature** that acts as a scheduling policy. This feature organizes AI training jobs to align with the hardware network of compute resources, maximizing the high bandwidth and low latency of supernodes for better training efficiency. Algorithm engineers can effortlessly use supernode hardware through simple setups.

Principles

The figure shows that each task in a training job is referred to as an instance. ModelArts uses affinity groups to schedule training jobs on supernodes and maximize their ultra-high bandwidth. An affinity group is a group of instances that you want to connect through a hyperplane in a training job. Instances in the same affinity group run on the same supernode.

Figure 11-1 Training job orchestration



To boost training efficiency, place PUs that need high bandwidth in the same affinity group.

For Tensor Parallelism (TP), in older hardware, TP is often set to 8, linking eight PUs per node with HCCS. In supernodes, more PUs connect via HCCS, allowing for larger TP domains.

Prerequisite: In distributed training, each instance must use all PUs on the node.

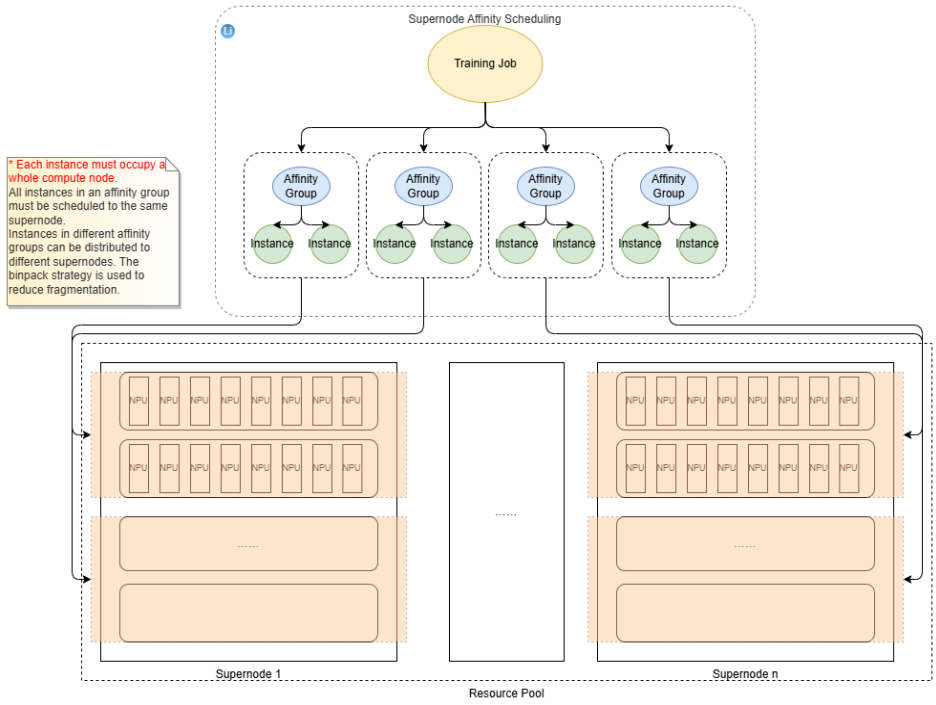
If TP is 32, then 32 PUs form one TP domain, and each node has 16 PUs. Here, two instances must share the same supernode, forming an affinity group.

You only need to set the new parameter **Supernode Affinity Group Instances** to **2** when creating a training job.

The training job runs only if the resource pool's available supernodes meet these conditions:

1. Available nodes ≥ Job instances
2. Allocatable affinity groups ≥ Required affinity groups

Figure 11-2 Supernode affinity scheduling for a training job



If the training job needs eight instances, with **Supernode Affinity Group Instances** set to **2**. Here are the available resources:

- Resource pool A

  Supernode 1: maximum nodes: 8; available nodes: 6

  Supernode 2: maximum nodes: 8; available nodes: 2
- Resource pool B

  Supernode 3: maximum nodes: 8; available nodes: 7

  Supernode 4: maximum nodes: 8; available nodes: 1

Both pools have eight available nodes, meeting the **first** condition. However, supernode 3 in pool B can allocate three affinity groups, while supernode 4 can allocate none. The **second** condition is not met. Thus, only pool A can run the training job. If the job goes to pool B, it will wait in the queue due to insufficient resources.

Constraints

- The affinity group feature only works with dedicated resource pools that have Ascend Snt9b23 supernodes.
- In distributed training, each instance must use all PUs on the node.
- The affinity group feature only supports resource pools with single-step specifications; it does not support resource pools with multi-step specifications.

Configuration Method

When you **create a training job** on the **ModelArts console**, selecting a dedicated resource pool and an Snt9b23 flavor allows you to set **Supernode Affinity Group Instances** during **Step 5: Configuring Resources**.

Notes for setting **Supernode Affinity Group Instances**:

- The default value is **1**, and the maximum value cannot exceed the maximum number of nodes on a single supernode.
- You must set the number of instances as an integral multiple of **Supernode Affinity Group Instances**. Otherwise, the training job cannot be created.
- After choosing a dedicated resource pool, you can check its supernode specifications, like total nodes, total PUs, available nodes, and available PUs. This helps you decide on **Supernode Affinity Group Instances**.

12 Managing Model Training Jobs

12.1 Training Dashboard Monitoring

Description

Use ModelArts to train models while monitoring their performance in real time for smooth operation. For deeper insights, the **ModelArts console** provides advanced monitoring and data export options. Go to **Model Training** > **Training Jobs** > **Monitoring Dashboard** (old console) to see an overview, health checks, and live updates on your training job. Click **Export** to save specific monitoring data for local analysis. These tools help you monitor progress effectively and improve training efficiency through better data management.

Constraints

You can view the monitoring data of the last year on the training dashboard.

Training Job Overview

The training job overview shows the total number of jobs, current resource requests, and job states, giving you a quick understanding of the training progress and resource usage.

Metric	Description
Total Jobs	The total number of all training jobs under the current workspace for the account, showing the overall scale of jobs.
Resource Usage (PUs)	The total number of accelerator cards requested by all currently running training jobs, indicating real-time resource demand.
Jobs in each state	The number of jobs in different statuses (such as **Queuing**, **Running**, **Completed**, **Abnormal/Failed**), used to monitor job health and distribution.

Health Monitoring

The health check module focuses on the stability and reliability management of training jobs. By quantitatively evaluating the job execution results and the system's fault tolerance capabilities, it provides critical evidence for O&M decision-making.

Metric	Description
Success Rate	The percentage of jobs successfully completed during the statistical period.
Fault Recovery Trigger Rate	The percentage of jobs with fault tolerance and recovery enabled during the statistical period.
Job Recovery Success Rate	The percentage of abnormal or interrupted jobs that were successfully restarted and used accelerator cards during the statistical period

Job Monitoring

Job monitoring includes tracking faults and resource usage. You can check this data for the past week, 30 days, or a custom time range.

Job Fault Monitoring

Table 12-1 Job fault monitoring

Metric	Description
Job Failure Rate	The percentage of jobs that failed during the statistical period.
Job Recovery Success Rate	The percentage of abnormal or interrupted jobs that were successfully restarted and used accelerator cards during the statistical period
Job Recovery Duration	The average time taken to restart a failed or interrupted job with accelerator cards used during the statistical period

Job Resource Consumption

Table 12-2 Job resource consumption

Metric	Description
Resource Consumption Trend	The compute resources requested by training jobs over time. The data can be filtered by NPU or GPU.

Metric	Description
Top Jobs by Resource Consumption	A list of training jobs that requested the most compute resources during the statistical period. The data can be filtered by NPU, GPU, or CPU.

12.2 Viewing Training Jobs and Details

Scenario

When using ModelArts model training, you may need to view the list of ongoing training jobs and their details to ensure the training process is proceeding smoothly.

Through the job list page, you can easily monitor the status of each job, customize the displayed content, filter by required attributes, quickly locate specific training jobs, and export the job list as an Excel file to your local device.

By clicking a specific job to enter the details page, you can view the job workflow, specific configuration details, as well as events, logs, monitoring data, evaluation results, and tags. Additionally, on the details page, you can view and export logs, log in to check the job via Cloud Shell, or use the intelligent O&M feature for job monitoring and troubleshooting.

How to View

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs** (old console).
   - In the job list, click **Export** to export training job details in a certain time range as an Excel file. A maximum of 200 rows of data can be exported.
   - Click  on the right of the job search box to set and adjust the content displayed in the job list.
   - In the search box above the training job list, you can filter training jobs by attribute type, such as name, ID, status, job mode, job type, algorithm, resource pool ID, job priority, description, and creator.
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, view basic job settings and algorithm parameters.
   - **Basic job information**

     Table 12-3 Basic job information

Parameter	Description
Job ID	Unique ID of the training job.

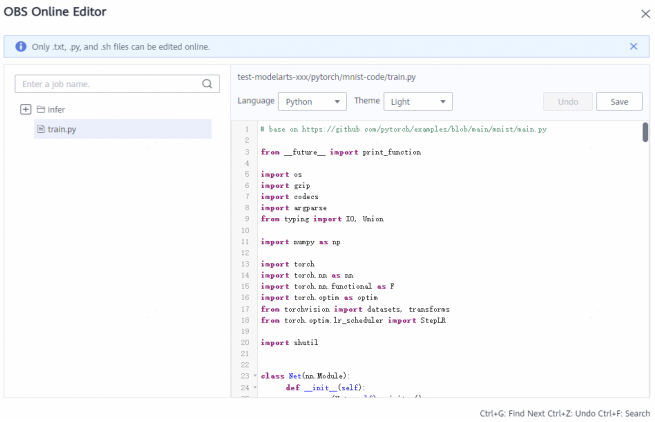
Parameter	Description
Status	• Status of the training job. • The value can be **Completed**, **Pending**, **Running**, **Creating**, **Terminating**, **Terminated**, **Failed**, **Abnormal**, or **Deleting**.
Experiment	• Name of the experiment to which the training job belongs. • Click to go to the experiment's job list.
Created	Time when the training job is created.
Duration	Duration of a training job, which is the total duration of Kubernetes resources in the entire lifecycle of a training job.
Retries	• Number of times that the training job automatically restarts upon a fault. This parameter is only available when **Auto Restart** is enabled during training job creation. • **Number of restarts/Maximum number of restarts** is displayed here.
Unconditional Auto Restart	It is displayed after auto restart is enabled. When **Unconditional Auto Restart** is enabled during job creation, **Open** is displayed. If it is not configured or is not enabled, **Disabled** is displayed.
Restart Upon Suspension	It is displayed after auto restart is enabled. When **Restart Upon Suspension** is enabled during job creation, **Open** is displayed. If it is not configured or is not enabled, **Disabled** is displayed.
Description	Description of the training job. When left unset, **--** appears. Click  to edit the training job's description.

Parameter	Description
Job Priority	• Priority of a training job created using a dedicated resource pool. If a training job is created using a public resource pool, this parameter is not displayed. • The platform handles jobs by prioritizing them from highest to lowest. If multiple jobs share the same priority, they are scheduled in the order they were submitted. When resources are available, the earliest-submitted job gets processed first. • The priority can be set to **1**, **2**, or **3**. A larger number indicates a higher priority. The default priority is **1**, and the highest priority is **3**. • If a training job is in the **Pending** state for a long time, you can change the job priority to reduce the queuing duration. For details, see **Priority of a Training Job**.
Preemption	• When using a dedicated resource pool, you can set this parameter. This parameter is not displayed when a public resource pool is used. • When enabled, jobs that allow preemption may be terminated and re-queued if resource pool capacity is insufficient. To avoid losing training progress, configure resumable training before enabling this function. For details, see **Resumable Training**. • **Disabled** is displayed when it is not set.
Job Visibility	The options are **Workspace** and **Creator**. • **Workspace**: The created training job is visible to all users in the current workspace. • **Creator**: Only the creator can view the job by default. To access it, other users must request the **modelarts:trainJob:listAll** permission, which allows them to view all training jobs, including those limited to the creator.

- **Training job parameters**

Table 12-4 Training job parameters

Parameter	Description
Runtime Type	Job mode, which can be **Debug** or **Production**.
Preset images	Preset image used by the training job. This parameter is available only for training jobs created using a preset image.

Parameter	Description
Custom image	Custom image used by the training job. This parameter is available only for training jobs created using a custom image.
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 a subscribed algorithm to create a training job.
Boot File	Location where the training boot file is stored. **NOTE** This parameter is not supported when you use a subscribed algorithm to create a training job.
Boot Command	Command for booting an image. This parameter appears only when **Boot Mode** is set to **Custom image**, not for **Preset image**. You can view both the parameter and its value on the training job details page.
User ID	ID of the user who runs the container.
Algorithm Name	• Algorithm used in the training job. You can click the algorithm name to go to the algorithm details page. • If this parameter is not configured, -- appears.
Local Code Directory	Path to the training code in the training container.
Work Directory	Path to the training boot file in the training container.
Compute Nodes	Number of instances for the training job.

Parameter	Description
Dedicated resource pool	Dedicated resource pool information. This parameter is available only when a training job uses a dedicated resource pool.
Compute Node	Names and IP addresses of the compute nodes used by the training job. This parameter is only displayed when the training job uses a dedicated resource pool.
Specifications	• Instance specifications used by the training job. This parameter is only displayed when the training job does not use custom specifications of a dedicated resource pool. • This parameter shows the instance specifications for the training job, both allocated to the training containers and chosen during job creation. The actual resources used are usually less than those chosen at job creation. This happens because the job's internal containers use some resources. These containers help run the training job smoothly.
Customized Specifications	• Instance specifications used by the training job. This parameter is only displayed when the training job uses custom specifications of a dedicated resource pool. • This parameter shows the custom resource and instance specifications selected for the training job.
Job Log Path	• This parameter is displayed if you select **Persistent Log Saving** and configure **Job Log Path** when creating the training job. This parameter is not displayed if **Persistent Log Saving** is not selected. • Click the path to go to the directory where the configured path is located.
Event Notification	• Topic and events set for event notification during training job creation. • **Disabled** is displayed when it is not configured.
Input > Input Path	OBS path where the input data is stored.
Input > Parameter Name	Input path parameter specified in the algorithm code.
Input > Obtained from	Method of obtaining the training job input.
Input > Container Path	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.

Parameter	Description
Output > Output Path	OBS path where the output data is stored.
Output > Parameter Name	Output path parameter specified in the algorithm code.
Output > Obtained from	Method of obtaining the training job output.
Output > Container Path	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.

5. On the training details page, manage event notifications of the training job.

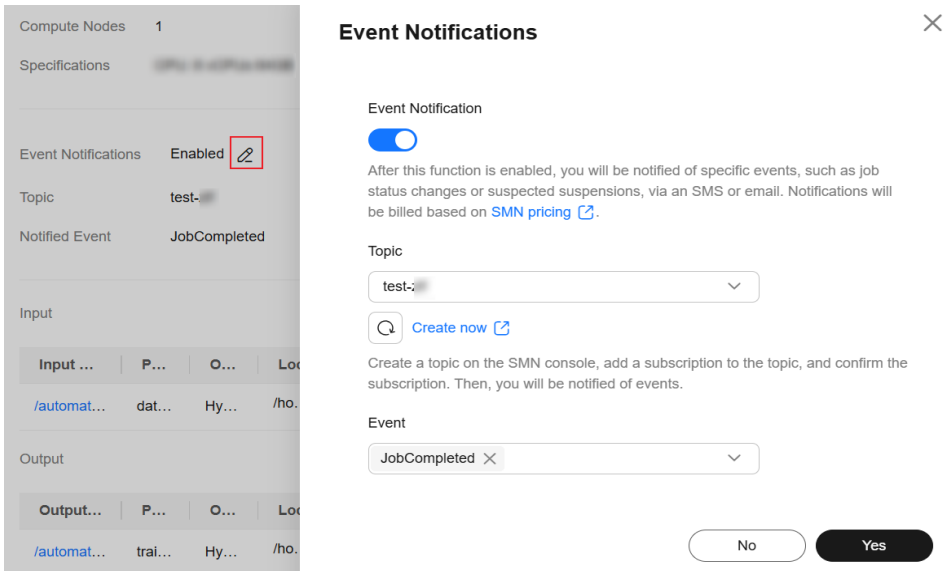
   NOTE

   - Event notifications cannot be configured for training jobs in the **Completed**, **Failed**, **Abnormal**, or **Terminated** state.
   - To set up event notifications, you need permission to view jobs.
   - Only the updated training status is notified for modification events.

   After event notification is enabled, you will be notified of a specific event, such as a job status change or suspected suspension, through an SMS message or email. Notifications will be billed based on SMN pricing. For details, see **Billing**.

   – If event notification has been enabled for a training job, you can click  next to **Enabled** to modify or disable event notification.

Figure 12-1 Modifying event notification



– If event notification has not been enabled for a training job, you can click next to **Disabled** to enable event notification.

Figure 12-2 Configuring event notification

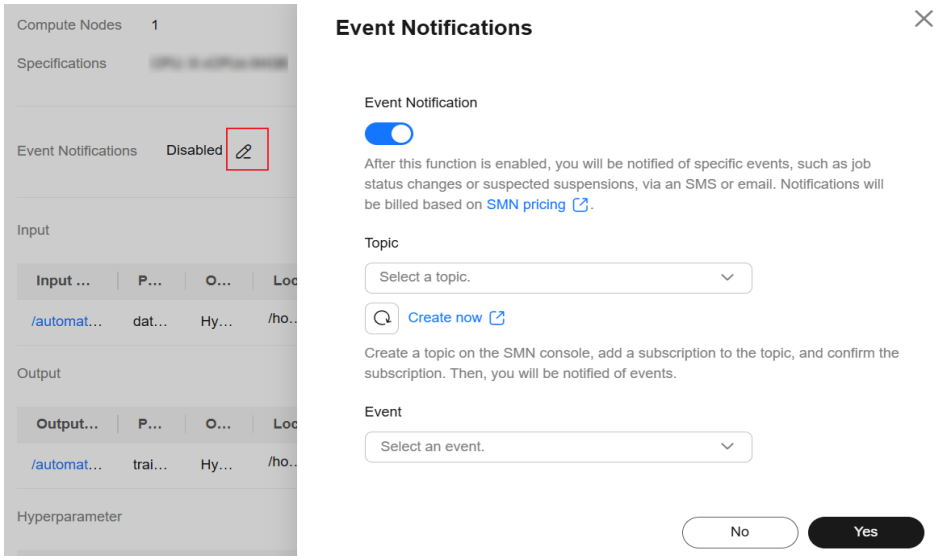


Table 12-5 Event notification parameters

Parameter	Description
Topic	Topic name of event notification. You can select a topic from the drop-down list or click **Create now** to create a topic on the SMN console. **NOTE** You can create a topic on the SMN console, add a subscription to it, and confirm the subscription status. Once these steps are completed, you will be notified of the event.

Parameter	Description
Event	Select events you want to subscribe to. Examples: **JobStarted**, **JobCompleted**, **JobFailed**, **JobTerminated**, and **JobHanged**. **NOTE** Only training jobs using GPUs or NPUs support **JobHanged** events.

12.3 Visualizing the Training Job Process

Description

When training machine learning models, you must track the status of your training jobs. Traditional methods often fail to show all necessary details, making it hard to follow progress accurately. With this feature, you can check the real-time status of a training job across multiple dimensions like job scheduling, environment setup, and execution progress. The visual interface allows you to check the entire lifecycle of a training job, helping you monitor progress, adjust parameters, and improve both efficiency and user experience.

Constraints

The main phases of the job process remain constant, while the subphases can vary. For example, if a training job lacks input data, the "input is downloading" sub-phase under "Environment Preparing" will be absent. Similarly, if the "job initializing environment is checking" event is missing, its corresponding sub-phase within the "Job Running" phase will also be omitted.

Checking Training Phases

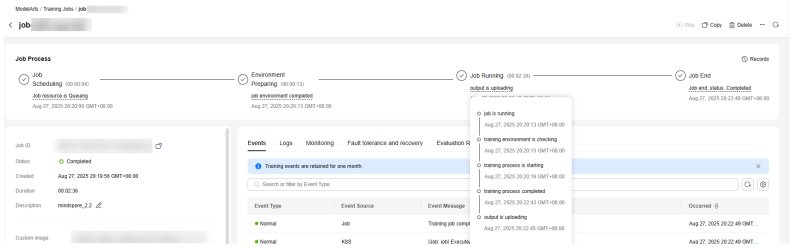
On the top of the training job details page, the training job module of ModelArts displays the job process details. There are four main phases: **Job Scheduling**, **Environment Preparing**, **Job Running**, and **Job End**.

You can check the subphases of each main phase.

- **Job Scheduling** logs all job activities, including successful or failed creations, scheduled jobs, and their exact timestamps.
- **Environment Preparing** documents essential steps for preparing the environment. It includes starting the setup, checking initial conditions, downloading training code, and completing the process, along with timestamps.
- **Job Running** tracks job details, including start and end times for each training step.
- **Job End** captures essential details like the date and time of the job's outcome.

Click **Records** in the upper right corner of the job process to see all its activity logs. This allows you to track details like repeated preemptions, rescheduling, or restarts during the job's execution.

Figure 12-3 Job process



12.4 Viewing Training Job Events

Scenario

Throughout the entire lifecycle of a training job, starting from the stage visible to you, the system backend records every key event point. You can view these records at any time on the details page of the corresponding training job. This allows you to clearly understand the progress and status of the training job, ensuring information transparency and traceability.

Event List

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: x hours)
- The training job has been manually deleted.
- Billing information synchronized.
- [worker-0] [Duration: second] Environment pre-check completed.
- [worker-0] [Duration: second] Pre-check failed. Exception:

- [worker-0] [Duration: second] Pre-check failed. Error:
- [worker-0] [Duration: second] Training code downloaded.
- [worker-0] [Duration: second] Failed to download the training code. Failure cause:
- [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] [Duration: second] Training output (parameter: xxx) uploaded.
- [worker-0] [Duration: second] Training output () prefetched.
- [worker-0] [Duration: second] Training pre-startup script executed.
- [worker-0] [Duration: second] Python dependency packages installed.
- [worker-0] rankTable training acceleration library installed.
- [worker-0] modelarts-turbo training acceleration library installed.
- [worker-0] turbo training acceleration library installed.
- [worker-0] Training discover library installed.
- [worker-0] The training output and log upload processes exit, and files will not be synchronized.
- [worker-0] Training container heartbeat detection timed out.
- [worker-0] The training job starts to run.
- [worker-0] Training started.
- [worker-0] Training completed, exit code.
- [worker-0] [Duration: second] Training output (parameter: xxx) uploaded.

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

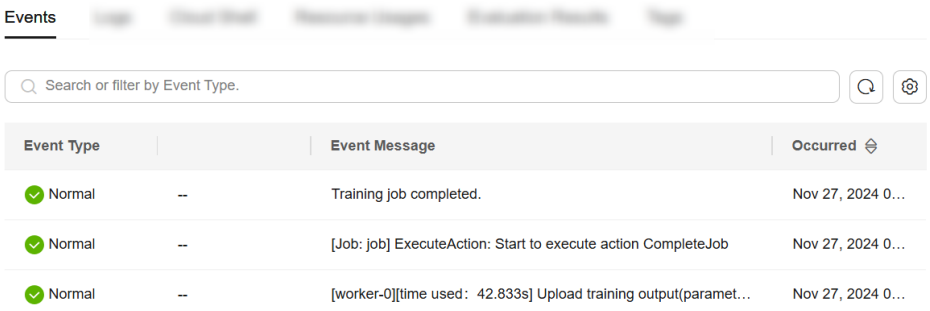
Notes and Constraints

The system automatically stores training job events for 30 days, and any expired events will be deleted.

Procedure

1. On the **ModelArts console**, choose **Model Build** > **Training** in the navigation pane. (On the old console, choose **Model Training** > **Training Jobs**.)
2. In the training job list, click the name of the target job to go to the training job details page.
3. On the training job details page, click the **Events** tab to view the event type, event source, event information, and event occurrence time.

Figure 12-4 Events



12.5 Viewing Training Job Logs

Scenario

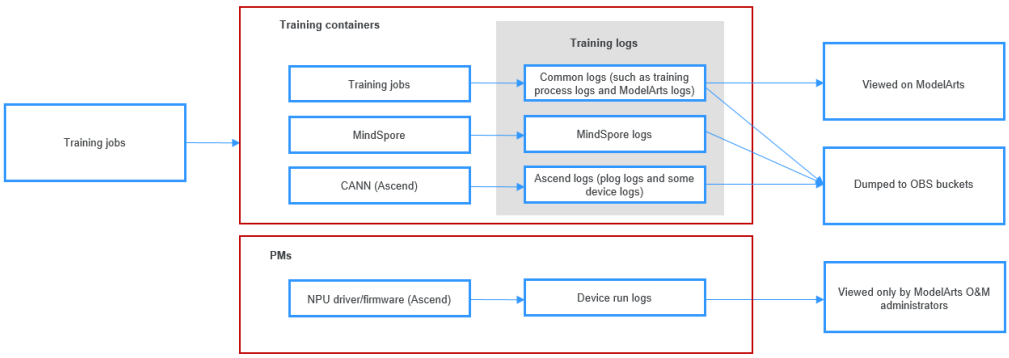
Training logs record the execution process and error information of training jobs, helping you quickly locate issues that occur during operation. Both standard output and standard error from your code are displayed in the training logs.

When encountering issues with training jobs in ModelArts, you should first check the logs. In most cases, problems can be directly identified through the error messages contained within the logs.

Training logs include common training logs and Ascend logs.

- **Common Logs**: When resources other than Ascend are used for training, only common training logs are generated. Common logs include the logs for **pip-requirement.txt**, training process, and ModelArts.
- **Ascend Logs**: When Ascend resources are used for training, device logs, plog logs, proc log for single-card training logs, MindSpore logs, and common logs are generated.

Figure 12-5 ModelArts training logs



NOTE

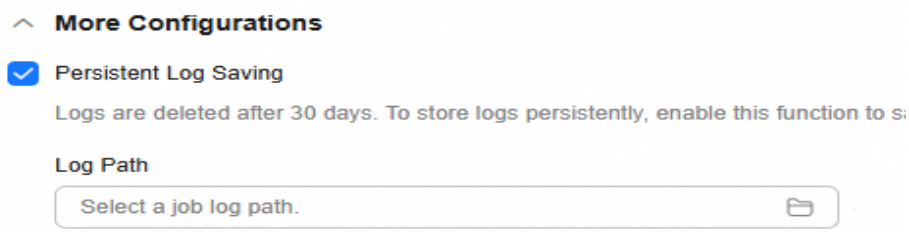
Separate MindSpore logs are generated only in the MindSpore+Ascend training scenario. Logs of other AI engines are contained in common logs.

Retention Period

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

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

Figure 12-6 Enabling persistent log saving



Real-time logs and historical logs have no difference in content. In the Ascend training scenario, permanent logs contain Ascend logs, which are not displayed on ModelArts.

Common Logs

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

Table 12-6 Common log types

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.

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 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
  ```

Ascend Logs

Ascend logs are generated when Ascend resources are used for training. When Ascend resources are used for training, device logs, plog logs, proc logs for single-card training logs, MindSpore logs, and common logs are generated.

Common logs in the Ascend training scenario include the logs for **pip-requirement.txt**, **ma-pre-start**, **davincirun**, training process, and ModelArts.

The following is an example of the Ascend log structure:

```
obs://dgg-test-user/snt9-test-cases/log-out/                                  # Job log path
├──modelarts-job-9ccf15f2-6610-42f9-ab99-059ba049a41e
   ├── ascend
      ├── process_log
            ├── rank_0
            ├── plog                                  # Plog logs
                  ...
         ├── device-0                                 # Device logs
                  ...
   ├── mindspore                                                  # MindSpore logs
├──modelarts-job-95f661bd-1527-41b8-971c-eca55e513254-worker-0.log             # Common logs
├──modelarts-job-95f661bd-1527-41b8-971c-eca55e513254-proc-rank-0-device-0.txt    # proc log for single-card training logs
```


Table 12-7 Ascend log description

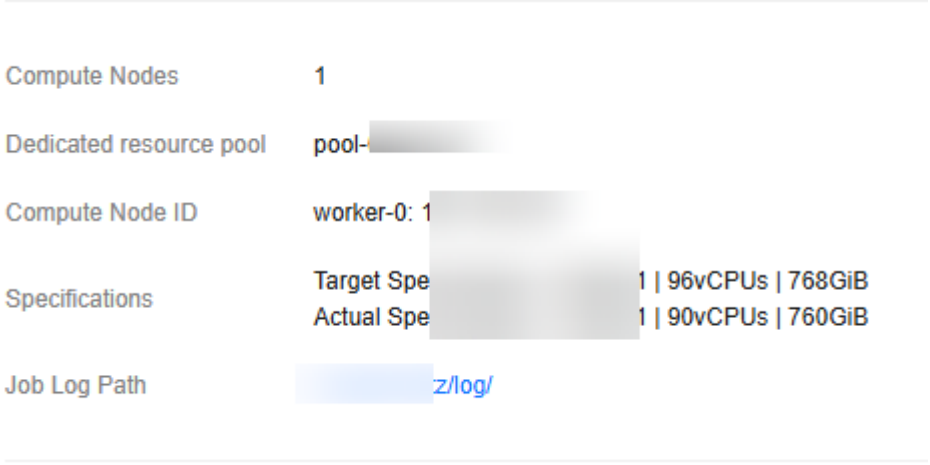
Type	Description	Name
Device logs	User process AICPU and HCCP logs generated on the device and sent back to the host (training container). If any of the following situations occur, device logs cannot be obtained: • The compute node restarts unexpectedly. • The compute node stops expectedly. After the training process ends, the log is generated in the training container.	**~/ascend/log/device-{device-id}/device-{pid}_{timestamp}.log** In the preceding command, **pid** indicates the user process ID on the host. Example: device-166_20220718191853764.log
Plog logs	User process logs, for example, ACL/GE. Plog logs are generated in the training container.	**~/ascend/log/plog/plog-{pid}_{timestamp}.log** In the preceding command, **pid** indicates the user process ID on the host. Example: plog-166_20220718191843620.log

Type	Description	Name
proc log	**proc log** is a redirection file of single-node training logs, helping you quickly obtain logs of a compute node. **proc log** for training using a preset MindSpore image and ranktable is generated in the training container and automatically saved in OBS. Training jobs using custom MindSpore images or other frameworks do not involve **proc log**.	**[modelarts-job-uuid]-proc-rank-[rank id]-device-[device logic id].txt** • **device id** indicates the ID of the NPU used in the training job. The value is 0 for a single NPU and 0 to 7 for eight NPUs. For example, if the Ascend specification is 8*Snt9, the value of device id ranges from 0 to 7. If the Ascend specification is 1*Snt9, the value of device id is 0. • **rank id** indicates the global NPU ID of the training job. The value ranges from 0 to the number of instances multiplied by the number of NPUs minus 1. If a single instance is used, the value of **rank id** is the same as that of **device id**. Example: modelarts-job-95f661bd-1527-41b8-971c-eca55e513254-proc-rank-0-device-0.txt
MindSpore logs	Separate MindSpore logs are generated in the MindSpore +Ascend training scenario. MindSpore logs are generated in the training container.	For details about MindSpore logs, visit the **MindSpore official website**.

Type	Description	Name
Common training logs	Common training logs are generated in the **/home/ma-user/modelarts/log** directory of the training container and automatically uploaded to OBS. The common training logs include these types: • Logs for **ma-pre-start** (specific to Ascend training): If the **ma-pre-start** script is defined, the script execution log is generated. • Logs for **davincirun** (specific to Ascend training): log generated when the Ascend training process is started using the **davincirun.py** file • Training process logs: standard output of user training code • Logs for **pip-requirement.txt**: If **pip-requirement.txt** is defined in training code, pip package installation logs are generated. • ModelArts logs: used by O&M personnel to locate service faults.	Contained in the **modelarts-job-[job id]-[task id].log** file. **task id** indicates the instance ID. If a single node is used, the value is **worker-0**. If multiple nodes are used, the value is **worker-0**, **worker-1**, ..., or **worker-*{n-1}***. **n** indicates the number of instances. Example: modelarts-job-95f661bd-1527-41b8-971c-eca55e513254-worker-0.log

In the Ascend training scenario, after the training process exits, ModelArts uploads the log files in the training container to the OBS directory specified by **Job Log Path**. On the job details page, you can obtain the job log path and click the OBS address to go to the OBS console to check logs.

Figure 12-7 Job Log Path



You can run the **ma-pre-start** script to modify the default environment variable configurations.

```
ASCEND_GLOBAL_LOG_LEVEL=3      # Log level, 0 for debug, 1 for info, 2 for warning, and 3 for error.
ASCEND_SLOG_PRINT_TO_STDOUT=1  # Whether to display plog logs. The value 1 indicates that plog logs are displayed by default.
ASCEND_GLOBAL_EVENT_ENABLE=1   # Event log level, 0 for disabling event logging and 1 for enabling event logging.
```

Place the **ma-pre-start.sh** or **ma-pre-start.py** script in the directory at the same level as the training boot file.

Before the training boot file is executed, the system executes the **ma-pre-start** script in **/home/work/user-job-dir/**. This method can be used to update the Ascend RUN package installed in the container image or set some additional global environment variables required for training.

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

  By default, **User logs** are displayed. You can select **System logs** or **All logs** to view log details.

  a. User logs: standard output of your training code

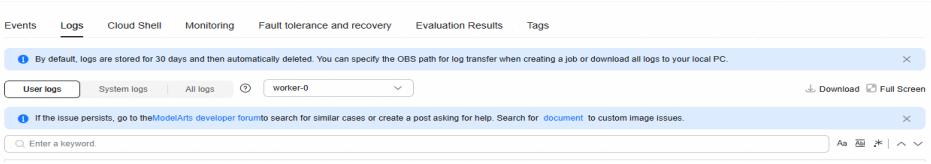
  b. System logs: ModelArts logs used by O&M personnel to locate service faults

  c. All logs: user logs and system logs If you select **All logs**, the size of the current log file is displayed.

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

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

Figure 12-8 Log preview



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 **Download** 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.

  By default, all logs are downloaded.

  If a training job is created on Ascend compute nodes, certain system logs cannot be downloaded in the training log pane. To obtain these logs, go to the **Job Log Path** you set when you created the training job.

- Searching for logs by keyword

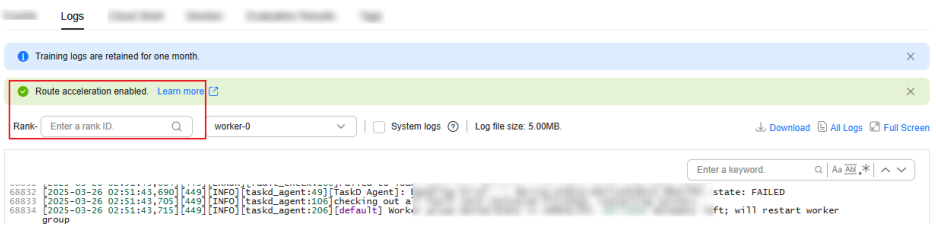
  You can search for a keyword in the log search box to query the specified log content.

  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.

- Viewing dynamic route acceleration logs

  If your training job has three or more instances, the **ROUTE_PLAN** environment variable is set to **true**, and you use Ascend resources, you can view the dynamic route acceleration logs by rank ID and check if dynamic route acceleration is enabled.

Figure 12-9 Viewing dynamic route acceleration logs



12.6 Using Cloud Shell to Debug a Production Training Job

ModelArts provides Cloud Shell, which allows you to log in to a running container to debug training jobs in the production environment.

Constraints

Only dedicated resource pools allow logging in to training containers using Cloud Shell. The training job must be running.

Preparation: Assigning the Cloud Shell Permission to an IAM User

1. Log in to the **Huawei Cloud console** as a tenant user, hover over your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. On the IAM console, choose **Permissions** > **Policies/Roles** from the navigation pane, click **Create Custom Policy** in the upper right corner, and configure the following parameters.
   - **Policy Name**: Enter a custom policy name, for example, **Using Cloud Shell to access a running job**.
   - **Policy View**: Select **Visual editor**.
   - **Policy Content**: Select **Allow**, **ModelArts Service**, **modelarts:trainJob:exec**, and default resources.
3. In the navigation pane, choose **User Groups**. Then, click **Authorize** in the **Operation** column of the target user group. On the **Authorize User Group** page, select the custom policies created in **2**, and click **Next**. Then, select the scope and click **OK**.

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

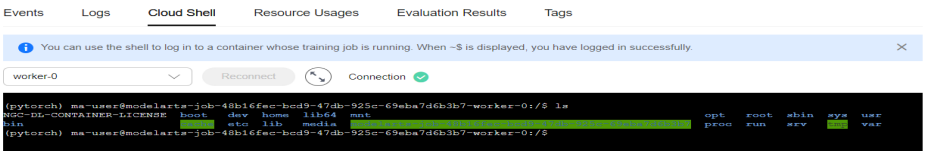
   If no user group is available, create a user group, add users using the user group management function, and configure authorization. If the target user is not in a user group, you can add the user to a user group through the user group management function.

Using Cloud Shell

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

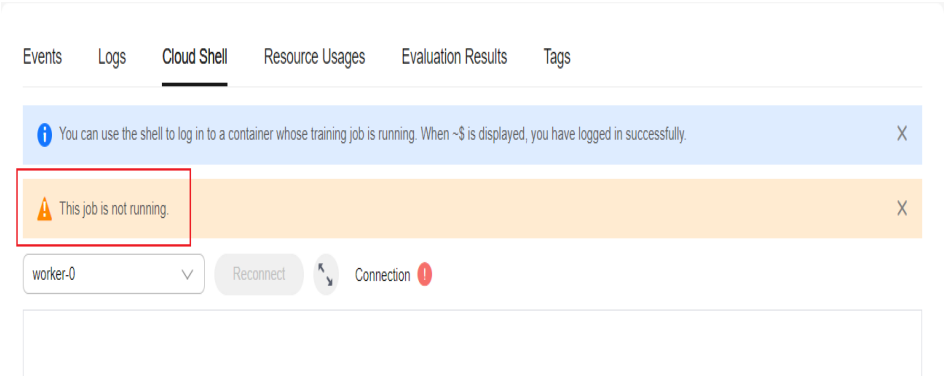
   Verify that the login is successful, as shown in the following figure.

Figure 12-10 Cloud Shell page



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

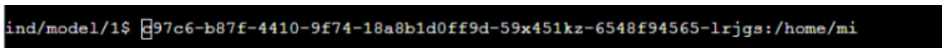
Figure 12-11 Error message



NOTE

If you encounter a path display issue when logging in to Cloud Shell, press **Enter** to resolve the problem.

Figure 12-12 Path display issue



Keeping a Training Job Running

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

Using the sleep Command

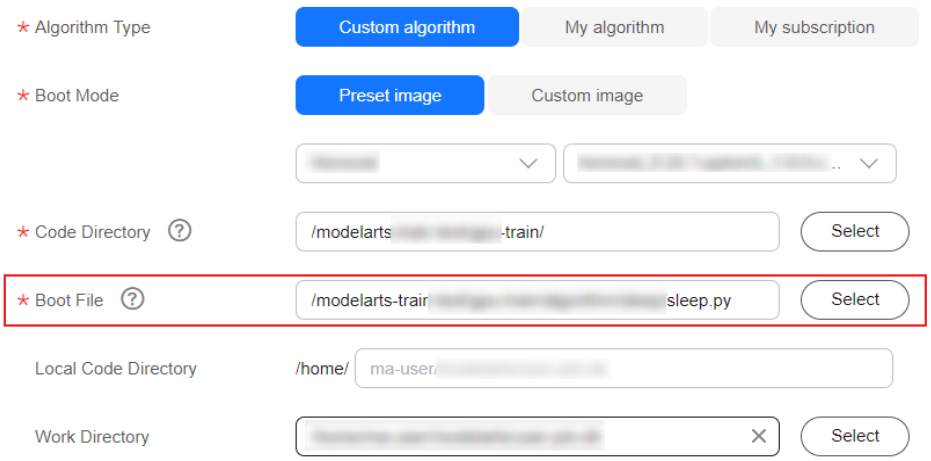
- For training jobs using a preset image

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

  Example of **sleep.py**

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

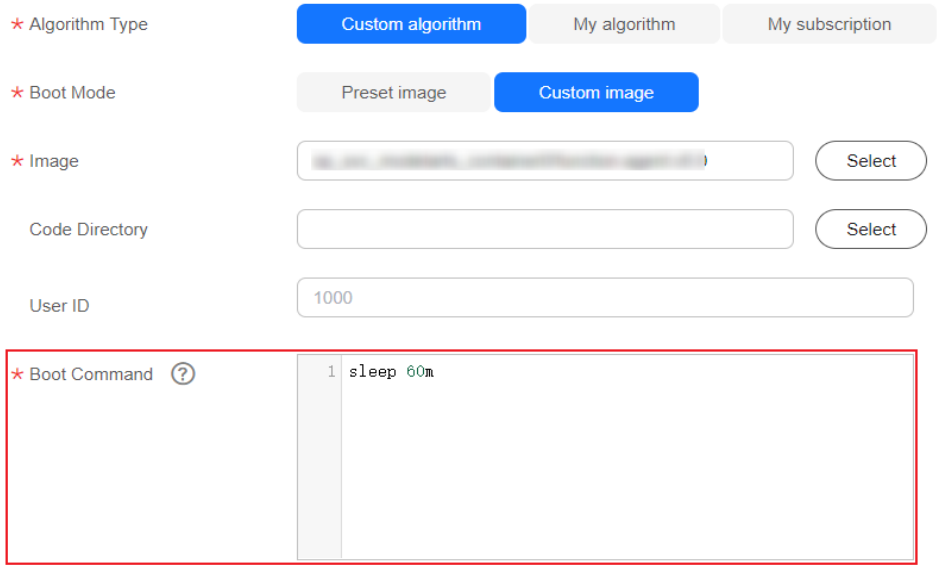

Figure 12-13 Using a preset image



- For training jobs using a custom image

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

Figure 12-14 Using a custom image



Keeping a Failed Job Running

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

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

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

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

Preventing Cloud Shell Session from Disconnection

To run a job for a long time, you can use the **screen** command to run the job in a remote terminal that stays active even if you disconnect. This prevents the job from failing due to disconnection.

1. If **screen** is not installed in the image, run **apt-get install screen** to install it.
2. Create a screen terminal.
   ```
   # Use -S to create a screen terminal named name.
   screen -S name
   ```
3. View the created screen terminals.
   ```
   screen -ls
   There are screens on:
   2433.pts-3.linux    (2013-10-20 16:48:59) (Detached)
   2428.pts-3.linux    (2013-10-20 16:48:05) (Detached)
   2284.pts-3.linux    (2013-10-20 16:14:55) (Detached)
   2276.pts-3.linux    (2013-10-20 16:13:18) (Detached)
   4 Sockets in /var/run/screen/S-root.
   ```
4. Connect to the screen terminal whose **screen_id** is **2276**.
   ```
   screen -r 2276
   ```
5. Press **Ctrl**+**A**+**D** to exit the screen terminal. After the exit, the screen session is still active and can be reconnected at any time.

For details about how to use screens, see **Screen User's Manual**.

Analyzing the Call Stack of the Suspended Process Using the py-spy Tool

Use py-spy to analyze the call stack of a suspended process and identify the issue.

Step 1 On the **ModelArts console**, choose **Model Build** > **Training**. (On the old console, choose **Model Training** > **Training Jobs**.)

Step 2 Click the target training job to go to its details page. On the page that appears, click the **Cloud Shell** tab and log in to the training container (the training job must be in the **Running** state).

Step 3 Install the py-spy tool.

```
# Use the utils.sh script to automatically configure the Python environment.
source /home/ma-user/modelarts/run/utils.sh

# Install py-spy.
pip install py-spy

# If the message "connection broken by 'ProxyError('Cannot connect to proxy.')" is displayed, disable the proxy.
export no_proxy=$no_proxy,repo.myhuaweicloud.com (Replace it with the pip source address of the corresponding region.)
pip install py-spy
```

Step 4 View the stack. For details about how to use the py-spy tool, see the **py-spy official document**.

```
# Find the PID of the training process.
ps -ef
```



```
# Check the process stack of process 12345.
# For a training job using eight cards, run the following command to check the stacks of the eight processes started by the main process in sequence.
py-spy dump --pid 12345
```

----End

12.7 Viewing the Resource Usage of a Training Job

Description

In the **Monitoring** tab of the training job details page, you can view the CPU, GPU, and NPU usage for the job or a single node.

Notes and Constraints

You can view the monitoring data of the entire training period. For details, see **Table 12-8**.

Usage data of training job resources is stored for 30 days before being automatically deleted.

Operations

1. On the **ModelArts console**, choose **Model Build** > **Training** in the navigation pane. (On the old console, choose **Model Training** > **Training Jobs**.)
2. In the training job list, click the name of the target job to go to the training job details page.
3. Click the **Monitoring** tab to view the resource usage of the training job. You can monitor resources in the following dimensions. **Table 12-8** describes the metrics.
   - Job monitoring: Monitors the overall resource usage of the current training job. You can select the last 15 minutes, last 30 minutes, last 1 hour, last 6 hours, last 12 hours, or last 24 hours, or specify a period.
   - Task monitoring: Monitors the resource usage of the training job's nodes. You can select the last 15 minutes, last 30 minutes, last 1 hour, last 6 hours, last 12 hours, or last 24 hours, or specify a period.
   - The system shows job monitoring data for the past 15 minutes if the jobs are in **Creating**, **Pending**, or **Running** states. For jobs in **Abnormal**, **Terminating**, **Terminated**, or **Completed** states, it displays data from their start to end times.

Table 12-8 Training job monitoring metrics

Category	Metric	Parameter	Description	Unit	Value Range	Presentation Form
CPU	CPU Usage	ma_container_cpu_util	CPU usage of a measured object	%	0%–100%	Line chart
	Used cores	ma_container_cpu_used_core	Number of CPU cores used by a measured object	Core	≥ 0	Bar chart
Memory	Physical Memory Usage	ma_container_memory_util	Percentage of the used physical memory to the total physical memory	%	0%–100%	Line chart
	Used Physical Memory	ma_container_memory_used_meg	Physical memory that has been used by a measured object (**container_memory_working_set_bytes** in the current working set) (Memory usage in a working set = Active anonymous page and cache, and file-baked page ≤ **container_memory_usage_bytes**)	MB	≥ 0	Bar chart
GPU	GPU Usage	ma_container_gpu_util	GPU usage of a measured object	%	0%–100%	Line chart
	GPU Memory Usage	ma_container_gpu_mem_util	Percentage of the used GPU memory to the total GPU memory	%	0%–100%	Line chart
	Used GPU Memory	ma_container_gpu_mem_used_megabytes	GPU memory used by a measured object	MB	≥ 0	Bar chart

Category	Metric	Parameter	Description	Unit	Value Range	Presentation Form
NPU	NPU Usage	ma_container_npu_ai_core_util	AI core usage of Ascend AI processors	%	0%–100%	Line chart
	NPU Memory Usage	ma_container_npu_memory_util	Percentage of the used NPU memory to the total NPU memory (To be replaced by **ma_container_npu_ddr_memory_util** for Snt3 series, and **ma_container_npu_hbm_util** for Snt9 series)	%	0%–100%	Line chart
	Used NPU Memory	ma_container_npu_memory_used_megabytes	NPU memory used by a measured object (To be replaced by **ma_container_npu_ddr_memory_usage_bytes** for Snt3 series, and **ma_container_npu_hbm_usage_bytes** for Snt9 series)	MB	≥ 0	Bar chart
Network	Network Downlink Rate	ma_container_network_receive_bytes	Inbound traffic rate of a measured object	Bytes/s	≥ 0	Line chart
	Network Uplink Rate	ma_container_network_transmit_bytes	Outbound traffic rate of a measured object	Bytes/s	≥ 0	Line chart
Disk	Disk Read Rate	ma_container_disk_read_kilobytes	Volume of data read from a disk per second	KB/s	≥ 0	Line chart

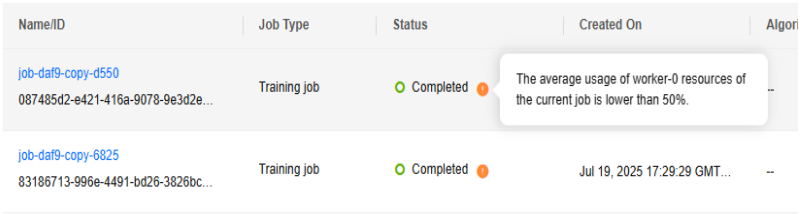
Category	Metric	Parameter	Description	Unit	Value Range	Presentation Form
	Disk Write Rate	ma_container_disk_write_kilobytes	Volume of data written into a disk per second	KB/s	≥ 0	Line chart

For more metrics, see **Viewing Monitoring Metrics of a Training Job**.

Alarms of Job Resource Usage

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

Figure 12-15 Job resource usage in the job list



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

Improving Job Resource Utilization

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

12.8 Viewing Monitoring Metrics of a Training Job

Receiving and promptly addressing alarms during a training job (for example, abnormal loss values) can save significant time and resources, preventing the waste caused by invalid job runs. Additionally, metric monitoring allows you to track the training job's progress in real time and the model's training status across different phases.

In the **Monitoring** tab of the training job details page of ModelArts, you can view the CPU, GPU, or NPU usage of training jobs. For details, see **Viewable Metrics on the ModelArts Console**.

For more metrics, go to the **AOM console**. For details, see **Viewing All ModelArts Monitoring Metrics on the AOM Console**.

With ModelArts, you can track custom metrics in AOM, such as loss values, step durations, and GPU throughput. These metrics are displayed in your training logs, making it easy to monitor trends and compare results across different jobs.

Collecting Prometheus Metrics for AOM Over HTTP: Configure the collection process and HTTP API to let the platform obtain metric data automatically.

- This method helps you simplify operations and reduce code changes.
- This method works well for users familiar with the Prometheus ecosystem and who already have collection tools.

Collecting Prometheus Metrics for AOM Using Commands: Configure the command and its parameters to let the platform obtain metric data automatically.

- This method works for users who do not want to show metric parsing.
- This method works well for users familiar with the Prometheus ecosystem and who already have collection tools.

Reporting Custom Metrics to AOM Using SDK: Integrate the SDK into your code to upload metric data to AOM manually.

- This method works well for users who need to add custom metrics, like adding custom dimensions or complex calculations.
- This method works well for complex service logic and when you need to control metric reporting in your code.
- This method works well for users who have experience with Huawei Cloud SDKs and know about cloud services.

Viewable Metrics on the ModelArts Console

Table 12-8 shows the metrics that can be viewed on the training job details page of the **ModelArts console**.

Table 12-9 shows the training metrics that can be viewed on the **Overview** page of the **ModelArts console**.

Table 12-9 Task-level metrics of training jobs

Metric	Description	How to View
Training job resource usage	CPU, GPU, or NPU usage of a training job.	**ModelArts console** > **Overview** > **Resource Usage of Training Jobs**
Card-hours	Running duration and number of cards used by a training job.	**ModelArts console** > **Overview** > **Resource Usage of Training Jobs** 

Collecting Prometheus Metrics for AOM Over HTTP

Configure the collection process and HTTP API to let the platform obtain Prometheus metric data and send it to the **AOM console** automatically. This solution lets you set up metric collection and exposure without manual uploads. It is ideal for monitoring training job performance in real time.

CAUTION

Follow these restrictions when using this solution:

- Metric format: Custom metrics must follow the cloud native exporter's standard format. If not, data might not be parsed correctly.

  Metric format example:

  ```
  mspti_marker_range_cost_time{name="Step: 0", source_kind="host", process_id="1887317", thread_id="1887317"} 261981960 1732357956859823920
  mspti_marker_range_cost_time{name="Step matmul: 0", source_kind="host", process_id="1887317", thread_id="1887757"} 2366640 1732357956863054960
  ```

- Data volume: If too much metric data (more than 32 KB) is reported within 10 seconds, data may be lost.
- Metric reporting frequency: If the metric reporting happens more often than every millisecond, some metrics might get lost because of repeated timestamps.

Step 1 Start the metric collection process in the training container. Start a separate process to collect custom metrics outside the training process.

For example, you can use the Flask framework to create a simple HTTP server.

```
from flask import Flask
app = Flask(__name__)

@app.route('/metrics')
def get_metrics():
    # Generate or obtain custom metric data.
    metrics = """
    mspti_marker_range_cost_time{name="Step: 0", source_kind="host", process_id="1887317", thread_id="1887317"} 261981960 1732357956859823920
```



```
    mspti_marker_range_cost_time{name="Step matmul: 0", source_kind="host", process_id="1887317",
thread_id="1887757"} 2366640 1732357956863054960
    """
    return metrics

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Step 2 Configure an HTTP API to ensure that the metric collection process provides an HTTP API (for example, **/metrics**) so that the training platform can periodically obtain metric data.

After each HTTP API call, promptly clear the collected metrics to avoid memory leaks or performance issues.

Step 3 Enable Prometheus metric collection.

When creating a training job, enable **Prometheus Metrics Collection**, set **Metrics Collection Method** to **HTTP**, and configure the collection URL and port. The parameter settings must meet the following requirements:

- Make sure the URL and port number match those of the metric collection process. If they do not, metrics might not report correctly.
- Ensure that the network environment where the training job is located allows access to the configured URL and port number to prevent metric collection failures due to network issues.

Step 4 View custom monitoring metrics on AOM.

When a training job is running, log in to the **AOM console** and view the custom Prometheus metrics on the **Metric Browsing** page.

----End

Collecting Prometheus Metrics for AOM Using Commands

Configure the command and its parameters to let the platform obtain Prometheus metric data and send it to the **AOM console** automatically. This solution lets you set up metric collection and exposure without manual uploads. You can fully customize the metric parsing process. It is ideal for monitoring training job performance in real time.

CAUTION

Follow these restrictions when using this solution:

- Metric format: The custom metric data must be in text format. Each metric should follow this format: <Metric name>{<Label name>=<Label value>,...} <Sampling value> [Millisecond timestamp].

  Metric data example:

  ```
  # HELP http_requests_total The total number of HTTP requests.
  # TYPE http_requests_total gauge
  html_http_requests_total{method="post",code="200"} 1656 1686660980680
  html_http_requests_total{method="post",code="400"} 2 1686660980681
  ```

- Data volume: If too much metric data (more than 32 KB) is reported within 10 seconds, data may be lost.
- Metric reporting frequency: If the metric reporting happens more often than every millisecond, some metrics might get lost because of repeated timestamps.

Step 1 Create a custom metric file, like **test.prom**. Store this file in your training image or code, ensuring it is accessible to the training container.

Step 2 Enable Prometheus metric collection.

When creating a training job, enable **Prometheus Metric Collection**, set **Metrics Collection Method** to **Command Line**, and configure the following parameters:

- **Command**: Enter the Linux command for reading metrics, for example, **cat**.
- **Command Parameters**: Enter the path to the custom metric file, for example, **/XXX/a.prom**.

Make sure the command and its parameters can reliably produce metrics with responses in seconds. If not, metric collection might fail.

Step 3 View custom monitoring metrics on AOM.

When a training job is running, log in to the **AOM console** and view the custom Prometheus metrics on the **Metric Browsing** page.

----End

Reporting Custom Metrics to AOM Using SDK

Integrate the SDK into your code to upload metric data to AOM manually.

This method works well for users who need to add custom metrics, like adding custom dimensions or complex calculations. It also works well for complex service logic and when you need to control metric reporting in your code.

1. Add the metric monitoring code to the training code. You can refer to the sample code below. For details about other requirements for preparing training code, see **Preparing Model Training Code**.

   Replace the region value **cn-southwest-2** in the last but one line of the code with the actual region. For details about the region values, see **Endpoints**.

   Add monitoring metrics to the code. For details about the parameters, see the **AOM documentation**.


```
# coding: utf-8
import os

from huaweicloudsdkaom.v2 import *
from huaweicloudsdkaom.v2.region.aom_region import AomRegion
from huaweicloudsdkcore.auth.credentials import BasicCredentials
from huaweicloudsdkcore.exceptions import exceptions
from moxing.framework import cloud_utils


def report2Aom(request,region):
    auth = cloud_utils.get_auth() # AK, SK, and temporary token, which are automatically obtained by the system.

    ak = auth.AK
    sk = auth.SK
    securityToken = auth.TOKEN

    projectId = os.environ.get("MA_IAM_PROJECT_ID")
    credentials = BasicCredentials(ak, sk, projectId).with_security_token(securityToken)
    client = AomClient.new_builder() \
        .with_credentials(credentials) \
        .with_region(AomRegion.value_of(region)) \
        .build()
    try:
        response = client.add_metric_data(request)
        print(response)
    except Exception as e:
        print(e)
if __name__ == "__main__":

    request = AddMetricDataRequest()
    listValuesBody = [
        # Enter the metric name, type, unit, and value, such as step_time and loss value.
        ValueData(
            metric_name="step_time", # Monitoring metric name, for example, step_time
            type="float",  # Data type of the metric. The value can only be int or float.
            unit="ms",     # Data unit, for example, ms. The value contains a maximum of 32 characters.
            value=135.572  # Value of the metric data. The value must be of a valid numeric type. The minimum value is 0.
        ),
        ValueData(
            metric_name="loss",
            type="float",
            value=0.6932
        )
    ]
    listDimensionsMetric = [
        # Enter the metric dimensions you want to view, such as thread and host.
        Dimension2(
            name="cluster_name",# This is only an example. Replace it with the actual metric dimension you want to view.
            value="fab2c5cf438b4f0c851fdcdf"# This is only an example. Replace it with the actual parameter value.
        ),
        Dimension2(
            name="user_name",
            value="modelarts_02" # This is only an example. Replace it with the actual parameter value.
        ),
        Dimension2(
            name="user_id",
            value="04f258c8fb00d42a1f6xxx" # This is only an example. Replace it with the actual parameter value.
        )
    ]
    metricBody = MetricItemInfo(
        dimensions=listDimensionsMetric,
        namespace="NOPAAS.ESC"  # Retain the default value.
    )
```



```
    listBodybody = [
        MetricDataItem(
            collect_time=int(round(time.time()*1000)),  # Time when monitoring metric data is collected. The value is the latest timestamp, in milliseconds.
            metric=metricBody,
            values=listValuesBody
        )
    ]
    request.body = listBodybody
    region = "cn-southwest-2"  # Replace the value with the actual region.
    response = report2Aom(request,region)
```

2. Add the commands below to the training code to load the required dependency packages. If a custom image is used, you can also install the dependencies during image creation. For details, see **Developing Code for Training Using a Custom Image**.

```
pip install huaweicloudsdkaom
pip install huaweicloudsdkcore
```

3. Create and run a training job. For details, see **Creating a Training Job**.
4. Log in to the **AOM console**. On the **Metric Browsing** page, view the reported metric data by specifying metrics.
5. Configure AOM alarm and notification rules. For details, see **Configuring Alarm Settings**.

12.9 Using CTS to Audit ModelArts

ModelArts can connect to CTS. With CTS, you can obtain operations associated with ModelArts for later query, audit, and backtrack operations.

After CTS is enabled, CTS starts recording operations on ModelArts. The CTS console stores the operation records generated in the last seven days. This section describes how to view operation records of the last seven days on the CTS console.

Prerequisites

You have enabled CTS. For details, see **Cloud Trace Service User Guide**.

Key Training Job Operations Traced by CTS

Table 12-10 Key training job operations traced by CTS

Operation	Resource Type	Trace
Creating a training job	JobV2	CreateJobV2
Modifying a training job	JobV2	UpdateJobV2
Deleting a training job	JobV2	DeleteJobV2
Stopping a training job	JobV2	StopJobV2
Creating an algorithm	AlgorithmV2	CreateAlgorithmV2
Modifying an algorithm	AlgorithmV2	UpdateAlgorithmV2
Deleting an algorithm	AlgorithmV2	DeleteAlgorithmV2

Operation	Resource Type	Trace
Creating an image saving task of a training job	SaveImageTask	CreateSaveImageTask
Adding a training job tag	JobV2	CreateJobTags
Deleting tags of a training job	JobV2	DeleteJobTags
Submitting a diagnosis task	DiagnosisTask	CreateDiagnosisTask
Creating a training experiment	Experiment	CreateExperiment
Updating a training experiment	Experiment	UpdateExperiment
Deleting a training experiment	Experiment	DeleteExperiment

Procedure

1. Log in to the **CTS console**.
2. Click  in the upper left corner and select a region.
3. In the navigation pane on the left, choose **Trace List**.
4. Specify filters as needed. You can query traces using a combination of the following filters:
   - **Trace Source**, **Resource Type**, and **Search By**:

     Select a filter from the drop-down list.

     When you select **Trace Name**, you need to enter a specific trace name.

     When you select **Resource ID**, you need to enter a specific resource ID.

     When you select **Resource Name**, you need to enter a specific resource name.
   - **Operator**: Select a specific operator (a user rather than a tenant).
   - **Trace Status**: Select **All trace statuses**, **Normal**, **Warning**, or **Incident**.
   - Time range: You can query traces generated during any time range of the last seven days.
5. Click ⌄ on the left of a trace to expand its details.
6. Locate the target trace and click **View Trace** in the **Operation** column. In the displayed **View Trace** dialog box, view the trace structure details.

   For details about the CTS trace structure, see **CTS User Guide**.

12.10 Intelligent O&M

Description

ModelArts monitors training jobs in real time for smooth operations. The training job details page includes intelligent O&M tools for easy monitoring and maintenance.

If a job ends with a **Failed** or **Terminated** status, choose suitable diagnosis tools under **Intelligent O&M**. Select the right tool based on your needs since their scope and duration differ.

Prerequisites

Fault monitoring requires that you enable **Auto Restart** when creating a training job.

Code error detection requires that you enable **Auto Restart** when creating a training job.

Performance monitoring requires that you enable **Performance Monitoring and Diagnosis** when creating a training job.

Real-Time Monitoring

Real-time monitoring in intelligent O&M is categorized into fault monitoring, code error detection, and performance monitoring.

When you enable the corresponding prerequisite features for a training job, the intelligent O&M system monitors the job in real-time. The interface displays status indicators such as **Unprotected**, **No risk**, **Low risk**, **Medium risk**, **High risk**, and **Monitoring**. If an anomaly is detected, the system provides a risk level and a detailed monitoring report, enabling you to handle abnormal jobs promptly.

Table 12-11 Real-time monitoring comparison

Real-Time Monitoring Type	Description	Anomaly Risk Level
Fault monitoring	When a fault occurs, the system automatically triggers restart and recovery to ensure the high availability of the training job.	Medium High
Code error detection	Performs real-time monitoring of the current training job code. When an anomaly appears, a diagnostic report is automatically generated to assist in troubleshooting.	Low Medium High

Real-Time Monitoring Type	Description	Anomaly Risk Level
Performance monitoring	Monitors the performance metrics of the training job in real-time. When metrics become abnormal, a monitoring report is automatically generated to assist in troubleshooting.	Medium

Diagnosis Tools

Currently, two types of diagnostic tools are supported: performance analysis and standard diagnosis.

Table 12-12

Diagnosis Tool	Description
Performance analysis	Designed for performance degradation issues such as unexpected step durations or imbalanced resource utilization. It provides real-time observation of step-time curves and supports manual collection of profiling data to generate visualized analysis results.
Standard diagnosis	Primarily detects training job environment information, job events, job logs, and device logs to identify runtime environment issues, code anomalies, and hardware failures. The diagnosis duration is positively correlated with the job cluster scale and log file size.

Click **Diagnose** on the right side of the corresponding tool to perform a diagnosis on the training job.

Once completed, you can view the diagnosis report.

Viewing Monitoring Reports

When an anomaly is detected during real-time monitoring, an anomaly detection report will be generated. Click **View Report** to see detailed diagnosis results.

- Code error detection report

  The results include the faulty device involved in the event, key logs, fault category, faulty component, faulty module, and recommended solutions.

Viewing Diagnosis Reports

After a diagnosis is complete, you can click **View Report** under the **Intelligent O&M** tab on the training job details page. Alternatively, you can find the

corresponding job in the list under **OM Management** > **Log Diagnosis** to view details.

The diagnosis report provides a detailed look at the basic information and results of the diagnostic task.

- Basic information

  Includes job ID, diagnosis duration, creation time, update time, creator, description, resource type, and training job ID.
- Diagnosis results

  Includes a description of the symptoms, a description of the fault, and recommended solutions for fault handling.

12.11 Viewing the Model Evaluation Result

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 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()
```

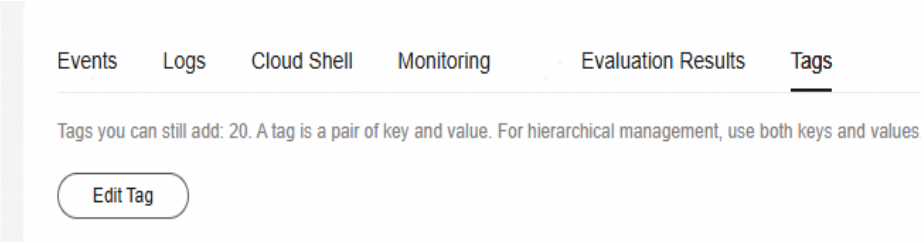
12.12 Viewing Training Job Tags

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

1. On the **ModelArts console**, choose **Model Build** > **Training** in the navigation pane. (On the old console, choose **Model Training** > **Training Jobs**.)
2. In the training job list, click the name of the target job to go to the training job details page.
3. Click **Tags**.

Tags can be added, modified, and deleted. For details about how to use tags, see **Using TMS Tags to Manage Resources by Group**.

Figure 12-16 Viewing training tags



NOTE

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

12.13 Priority of a Training Job

When using a dedicated resource pool for training jobs, you can set the job priority when creating a training job or adjust the priority when a job is in the **Pending** state for a long time. By adjusting the job priority, you can reduce the job queuing duration.

Overview

Some training jobs, such as unimportant tests or experiments, are of low priority. In this case, you need to prioritize training tasks (jobs). Jobs with higher priority are scheduled before those with lower priority.

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

Constraints

- You can set the priority of a training job only if it is created using a dedicated resource pool.
- The job priority can be set to **1**, **2**, or **3**. A larger number indicates a higher priority. The default priority is **1**, and the highest priority is **3**. To set the priority to **3**, you must have the permission. For details about how to configure the permission, see **Assigning the Permission to Set the Highest Job Scheduling Priority to an IAM User**.
- If you want to change the priority of a training job in the **Pending** state, it must be queued for resources in a dedicated resource pool.

Configuring the Priority

On the **Create Training Job** page, select **Increase job scheduling priority**. For details, see **Step 7: Configuring Scheduling Parameters**. The priority can be set to **1**, **2**, or **3**. A larger number indicates a higher priority. The default priority is **1**, and the highest priority is **3**.

Changing the Priority

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs** (new console: **Training**).
3. On the training job list page, find the target training job in the **Pending** status.
4. Click the target training job to access its details page.
5. Click  next to **Job Priority**. In the displayed dialog box, change the priority and click **OK**.

ModelArts manages training job queues using a priority system.

1. For jobs with equal priority, scheduling follows these rules:
   - If resources are enough, jobs run based on their submission order using the FIFO method.
   - If resources fall short, some jobs from separate logical pools might get prioritized instead, disrupting strict FIFO execution.
2. For jobs with adjusted priority, scheduling follows this rule:

   After a training job is delivered to a resource pool, its priority cannot be changed.

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

You can set the job priority to **1** or **2** by default. Once permission to set the highest priority is granted, you can set it to **1**, **2**, or **3**. If you have been assigned with the **modelarts:trainJob:setHighPriority** permission, you can set this parameter to **3**.

1. Log in to the **Huawei Cloud console** as a tenant user, hover 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.
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 set the training job priority to **1** to **3**.

   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.

12.14 Saving the Image of a Debug Training Job

During development and debugging, you may modify and optimize the image environment. After modifying the image, you can save the image on the console for future use.

Notes for saving an image:

1. The installed dependency packages will be retained, but the training data and code that need to be persistently stored will not be saved in the generated container image.
2. VS Code keeps the plug-ins you install on the server during remote development.

Key settings and dependencies are preserved, but dynamic data and code require separate handling.

Prerequisites

- The debug job is created in a dedicated resource pool.
- The job is in the **Running** state.

Notes and Constraints

- Only the image of the training job's worker-0 can be saved on the console.
- The image to be saved should not be larger than 35 GB and there should be no more than 125 layers. Otherwise, the image may fail to be saved due to the rootfs difference between node containers.
  - To save a larger image, log in to the ModelArts console. In the navigation pane on the left, choose **Resource Management** > **Dedicated Compute Resources** > **Resource Pools** (old console: **Resource Management** > **Dedicated Resource Pool**). On the displayed page, configure the container engine size as needed. For details, see **Resizing a Standard Dedicated Resource Pool**.
  - If the fault persists, contact technical support.

Procedure

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs** (new console: **Training**).
3. Click the target training job to access its details page. On the displayed page, click **Save Image** in the upper right corner.
4. In the **Save Image** dialog box, configure parameters. Click **OK** to save the image.

   Choose an organization from the **Organization** drop-down list. If no organization is available, click **Create** on the right to create one.

Users in an organization can share all images in the organization.

The image will be saved as a snapshot, which will take a while. During which, the image status is **Saving**.

NOTE

The time required for saving an image as a snapshot will be counted in the job runtime. If the job runtime expires before the snapshot is saved, saving the image will fail.

5. The saved image can be used to create a training job.

FAQs

1. What Can I Do If an Image Fails to Be Saved?
   a. If the image fails to be saved, check the event on the training job details page. For details, see **Viewing Training Job Events**.
   b. If a dedicated resource pool is used, you can adjust the container engine space on the **Resource Management** > **Lite Compute Resources** > **Lite Clusters** (old console: **Resource Management** > **Lite Cluster**) page as required. For details, see **Resizing a Standard Dedicated Resource Pool**.
   c. If the fault persists, contact technical support.
2. Why Cannot I Check Real-Time Logs During Image Saving?

   Saving an image may affect checking real-time logs. The Cloud Shell connection may be interrupted. After the image is saved, the connection is automatically restored.

12.15 Copying, Stopping, or Deleting a Training Job

Saving As an Algorithm

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs** (new console: **Training**). Click the target training job to go to the training job details page.
3. To modify the algorithm of a training job, click **Save As Algorithm** in the upper right corner of the training job details page. On the **Algorithms** page, the algorithm parameters for the last training job are automatically set. You can modify the settings.

NOTE

1. A subscribed algorithm cannot be saved as a new algorithm.

Copying a Training Job

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs** (new console: **Training**).
3. If you are not satisfied with a created training job, click **Copy** (or **Clone**) in the **Operation** column to re-create a training job. The page for re-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.

Stopping a Training Job

1. Log in to the **ModelArts console**.
2. In the navigation pane, choose **Model Training** > **Training Jobs** (new console: **Training**).
3. In the training job list, click **Stop** in the **Operation** column of a training job that is in creating, pending, or running state. Enter **YES** in the dialog box and confirm the operation. After a training job is stopped, its billing stops on ModelArts.

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

Deleting a Training Job

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

NOTE

Deleted training jobs cannot be restored.

- On the **Training Jobs** (**Model Training** in the new version) page, delete the training job that has finished running. Click **Delete** in the **Operation** column. In the displayed dialog box, enter **DELETE**, and click **OK** to confirm the deletion.
- Go to OBS and delete the OBS bucket and files used by the training job.

12.16 Managing 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

Notes and Constraints

When defining custom environment variables, avoid using names that start with **MA_** to prevent conflicts with system environment variables.

Configuring Environment Variables

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

NOTE

To ensure data security, do not enter sensitive information, such as plaintext passwords.

Environment Variables Preset in a Training Container

Table 12-13, Table 12-14, Table 12-15, Table 12-16, Table 12-17, Table 12-18, and Table 12-19 list environment variables preset in a training container.

The environment variable values are examples only.

Table 12-13 Path environment variables

Variable	Description	Example	Default Value
PATH	Executable file paths	**PATH=/usr/local/bin:/usr/local/cuda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin**	Not fixed
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/lib:/usr/local/nvidia/lib64**	Not fixed
LIBRARY_PATH	Static library paths	**LIBRARY_PATH=/usr/local/cuda/lib64/stubs**	Not fixed
MA_HOME	Main directory of a training job	**MA_HOME=/home/ma-user**	**/home/ma-user**
MA_JOB_DIR	Parent directory of the training algorithm folder	**MA_JOB_DIR=/home/ma-user/modelarts/user-job-dir**	**/home/ma-user/modelarts/user-job-dir**

Variable	Description	Example	Default Value
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**	**/home/ma-user/modelarts**
MA_LOG_DIR	Training log directory	**MA_LOG_DIR=/home/ma-user/modelarts/log**	**/home/ma-user/modelarts/log**
MA_SCRIPT_INTERPRETER	Training script interpreter	**MA_SCRIPT_INTERPRETER=**	Not set

Table 12-14 Environment variables of a distributed training job

Variable	Description	Example	Default Value
MA_CURRENT_IP	IP address of a job container.	**MA_CURRENT_IP=192.168.23.38**	Not fixed
MA_NUM_GPUS	Number of accelerator cards in a job container.	**MA_NUM_GPUS=8**	**0**
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**	**worker**

Variable	Description	Example	Default Value
MA_NUM_HOSTS	Number of instances which is automatically obtained from **Compute Nodes**.	**MA_NUM_HOSTS=4**	**1**
VC_TASK_INDEX	Container index, starting from **0**. This parameter is invalid for single-node training. In multi-node training jobs, you can use this parameter to determine the algorithm logic of the container.	**VC_TASK_INDEX=0**	**0**
VC_WORKER_NUM	Instances required for a training job.	**VC_WORKER_NUM=4**	**1**
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**	Not fixed

Variable	Description	Example	Default Value
${MA_VJ_NAME}-${MA_TASK_NAME}-N.${MA_VJ_NAME}	Communication domain name of a node. For example, the communication domain name of node 0 is **${MA_VJ_NAME}-${MA_TASK_NAME}-0.${MA_VJ_NAME}**. N indicates the number of instances. **WARNING** This method does not work for creating communication domain names for supernode resource pools. Instead, you can get the communication domain names for all nodes directly from **VC_WORKER_HOSTS** across all resource pools, including supernode ones.	For example, if there are four instances, the environment variables are as follows: **${MA_VJ_NAME}-${MA_TASK_NAME}-0.${MA_VJ_NAME}** **${MA_VJ_NAME}-${MA_TASK_NAME}-1.${MA_VJ_NAME}** **${MA_VJ_NAME}-${MA_TASK_NAME}-2.${MA_VJ_NAME}** **${MA_VJ_NAME}-${MA_TASK_NAME}-3.${MA_VJ_NAME}**	Not fixed

Table 12-15 NCCL environment variables

Variable	Description	Example	Default Value
NCCL_VERSION	NCCL version	**NCCL_VERSION=2.7.8**	Not fixed
NCCL_DEBUG	NCCL log level	**NCCL_DEBUG=INFO**	**INFO**
NCCL_IB_HCA	InfiniBand NIC to use for communication	**NCCL_IB_HCA=^mlx5_bond_0**	Not fixed
NCCL_IB_TIMEOUT	InfiniBand transmission timeout interval	**NCCL_IB_TIMEOUT=18**	**18**

Variable	Description	Example	Default Value
NCCL_IB_RETRY_CNT	Maximum number of InfiniBand transmission retries	**NCCL_IB_RETRY_CNT=15**	**15**
NCCL_IB_GID_INDEX	Global ID index used in RoCE mode	**NCCL_IB_GID_INDEX=3**	**3**
NCCL_IB_TC	InfiniBand traffic type	**NCCL_IB_TC=128**	**128**
NCCL_SOCKET_IFNAME	IP interface to use for communication	**NCCL_SOCKET_IFNAME=bond0,eth0**	Not fixed
NCCL_NET_PLUGIN	Network plug-in used by NCCL	**NCCL_NET_PLUGIN=none**	**none**

Table 12-16 OBS environment variables

Variable	Description	Example	Default Value
MA_S3_ENDPOINT	OBS address. Ensure that applications can properly connect to the specified OBS.	N/A	Not fixed
S3_VERIFY_SSL	Specifies whether to use SSL to access OBS. If this parameter is set to **0**, the SSL certificate is not verified. If this parameter is set to **1**, the SSL certificate is verified.	**S3_VERIFY_SSL=0**	**0**
S3_USE_HTTPS	Specifies whether to use HTTPS to access OBS. If the parameter is set to **1**, HTTPS is used. If the parameter is set to **0**, HTTP is used.	**S3_USE_HTTPS=1**	**1**

Table 12-17 Environment variables of the pip source and API Gateway address

Variable	Description	Example	Default Value
MA_PIP_HOST	Domain name of the pip source	**MA_PIP_HOST=repo.example.com**	Domain name of the Huawei internal pip source
MA_PIP_URL	Address of the pip source	**MA_PIP_URL=http://repo.example.com/repository/pypi/simple/**	Address of the Huawei internal pip source
MA_APIGW_ENDPOINT	ModelArts API Gateway address	**MA_APIGW_ENDPOINT=https://modelarts.region.xxx.example.com**	Not fixed

Table 12-18 Environment variables of job metadata

Variable	Description	Example	Default Value
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**	Not fixed

Table 12-19 Precheck environment variables

Variable	Description	Example	Default Value
MA_SKIP_IMAGE_DETECT	Specifies whether to enable ModelArts precheck. If this parameter is set to **1** or not set, precheck is enabled. If this parameter is set to **0**, pre-check is disabled. It is good practice to enable precheck to detect node and driver faults before they affect services. This parameter is not set by default and precheck is enabled.	**1**	Not set

Table 12-20 Suspension detection environment variables

Variable	Description	Example	Default Value
MA_HANG_DETECT_TIME	Suspension detection time. The job is considered suspended if its process I/O does not change for this time. Value range: 10 to 720 Unit: minute Default value: **30**	**30**	**30**

How Do I View Training Environment Variables?

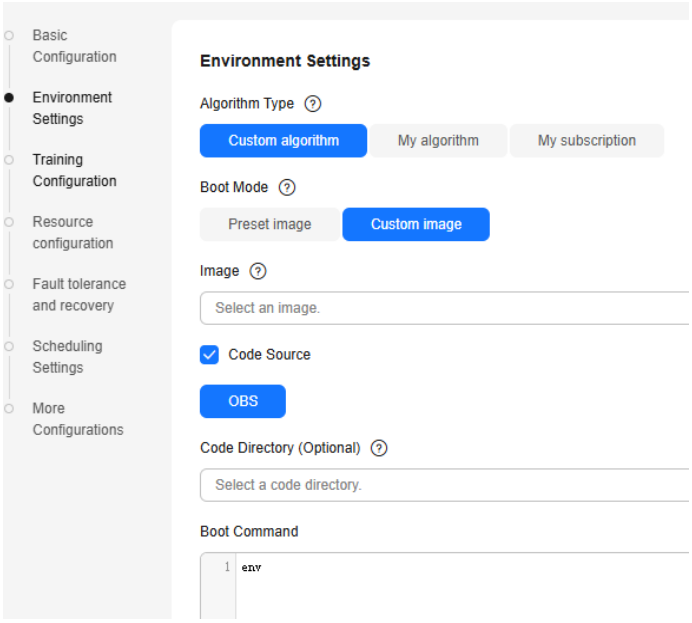
Environment variables may be injected into the container or processes, depending on the service. Environment variables injected into processes are invisible in Cloud Shell.

You are advised to use method 1 to view all environment variables.

1. View training environment variables using the boot command.

   When creating a training job, select **Custom algorithm** for **Algorithm Type** and **Custom image** for **Boot Mode**, enter **env** for **Boot Command**, and retain default settings for other parameters.

   Figure 12-17 Boot Command



After the training job is complete, check the **Logs** tab on the training job details page. The logs contain information about all environment variables.

Figure 12-18 Viewing logs

```
NV_LIBCUBLAS_DEV_VERSION=11.3.1.68-1
NV_CUDA_COMPAT_PACKAGE=cuda-compat-11-2
NV_CUDNN_PACKAGE_DEV=libcudnn8-dev=8.1.1.33-1+cuda11.2
LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvidia/lib64
NV_LIBNCCL_DEV_PACKAGE=libnccl-dev=2.8.4-1+cuda11.2
MA_ENGINE_VERSION=
MA_NUM_HOSTS=1
VC_WORKER_HOSTS=modelarts-job-5f8e4b52-630b-4c15-9bb6-c3f68a48ac47-worker-0.modelarts-job-5f8e4b52-630b-4c15-9bb6-c3f68a48ac47
VK_TASK_INDEX=0
_=/usr/bin/env
MA_SCRIPT_INTERPRETER=
NV_LIBNPP_DEV_PACKAGE=libnpp-dev-11-2=11.2.1.68-1
MA_MAX_BACKOFF=0
HOSTNAME=modelarts-job-5f8e4b52-630b-4c15-9bb6-c3f68a48ac47-worker-0
MA_IAM_USER_ID=79098163dd814fd986eca7ef0325d086
MA_CURRENT_IP=10.0.0.62
NV_LIBNPP_VERSION=11.2.1.68-1
NV_NVPROF_DEV_PACKAGE=cuda-nvprof-11-2=11.2.67-1
MA_MOUNT_PATH=/home/ma-user/modelarts
NVIDIA_VISIBLE_DEVICES=all
MA_ENGINE_TYPE=
NV_NVPROF_VERSION=11.2.67-1
NV_LIBCUSPARSE_VERSION=11.3.1.68-1
MODELARTS_SCC_SERVICE_PORT=60687
MA_HOME=/home/ma-user
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.247.0.1
NV_LIBCUBLAS_DEV_PACKAGE=libcublas-dev-11-2=11.3.1.68-1
MA_VJ_NAME=modelarts-job-5f8e4b52-630b-4c15-9bb6-c3f68a48ac47
MA_PIP_URL=http://repo.myhuaweicloud.com/repository/pypi/simple/
NCCL_VERSION=2.8.4-1
KUBERNETES_PORT=tcp://10.247.0.1:443
PWD=/
NVARCH=x86_64
HOME=/home/ma-user
```

2. Check training environment variables using Cloud Shell.

   Run the **env** command on Cloud Shell to obtain the environment variables.

   This method fails to obtain environment variables injected by the training platform during processes, such as **VC_TASK_INDEX**, **VC_WORKER_NUM**, and **VC_WORKER_HOSTS** in supernode scenarios. This method is not recommended.

12.17 Managing Training Experiments

Training Experiment

Managing many training jobs can become challenging when trying to find or track them. Training experiments can simplify this process. This is a way to group similar jobs together. You can organize jobs into different experiments as needed, with each experiment holding multiple related jobs.

You can add one job to an experiment, create jobs within existing experiments, check the experiment list, view experiment details, or delete experiments as needed.

NOTE

This section applies only to the old console. The experiment feature is not available in the new console. To proceed, you must switch back to the old console.

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:

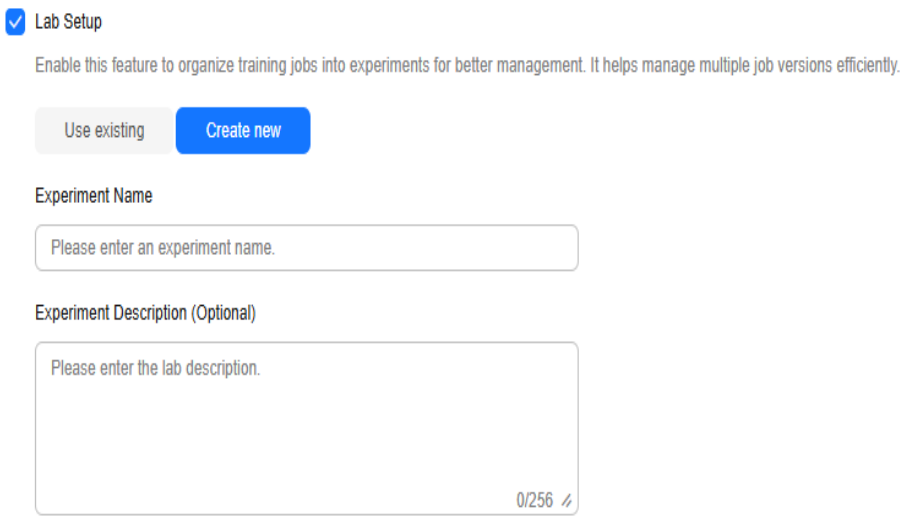
- **Create new**: An experiment can only be created when you create a training job. If you select this option, enter a new experiment name. After the job is submitted, the experiment is created and the job is added to the new experiment. The experiment name will be checked. If the name is already in use, 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 on the training job management page only shows jobs that have been added to an experiment.

Creating a Job to Be Added to an Experiment

Log in to the **ModelArts console**, choose **Model Training** > **Training Jobs**, and click **Create Training Job** in the upper right corner.

On this page, set **Experiment** to **Create new** and enter a name for the new experiment. Then, an experiment is created after you create the training job.

Figure 12-19 Creating a training job



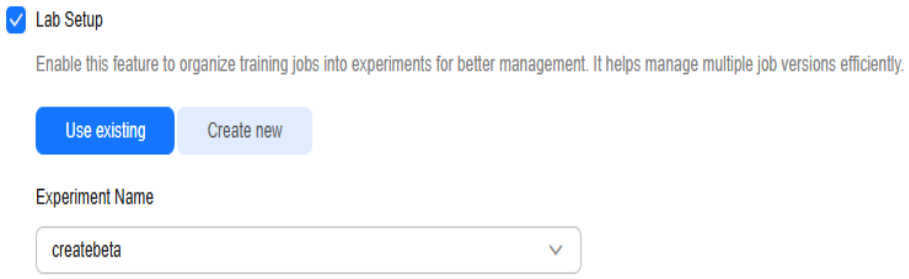
Adding a Created Job to an Experiment

Log in to the **ModelArts console**, choose **Model Training** > **Training Jobs**, and click **Copy** in the **Operation** column of the target job. Alternatively, click the job name or ID in the job list. On the job details page, click **Copy** in the upper right corner.

- For a job that has not been added to an experiment, set **Experiment** to **Create new** and enter a name for the new experiment. Then, an experiment is created after you copy the training job.

- For a job that has been added to an experiment, set **Experiment** to **Use existing** and select the target experiment.

Figure 12-20 Copying a training job



Viewing the Experiment List

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

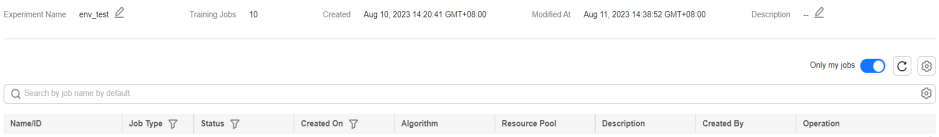
Table 12-21 Basic experiment information

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.

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 12-21 Viewing experiment details



- You can click to edit 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 are displayed.
- You can search for jobs by name, ID, algorithm, status, creation time, job type, or resource pool.
- You can refresh the job list. You can click the refresh button in the upper right corner of the job list to refresh it.
- 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.

Deleting an Experiment

WARNING

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.

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.

13 MoXing

13.1 Basic MoXing Functions

Description

When using ModelArts, you may encounter situations where you need to access Object Storage Service (OBS). OBS is an object-based massive storage service. Unlike traditional local file systems, files on OBS cannot be accessed directly through file paths.

When directly accessing OBS files, you may encounter the following issues:

- You cannot use the open() method as you would with local files.
- File read and write operations require network requests.
- Using the OBS Python SDK API can be relatively complex.

How can you more conveniently access and manage OBS files in ModelArts?

ModelArts provides the mox.file API, which offers a simpler solution for OBS operations. Below is an example of using the mox.file API to access OBS files:

```
# Example of accessing a file from OBS
import moxing as mox

# Open an OBS file.
with mox.file.File('obs://bucket_name/a.txt', 'r') as f:
    print(f.read())

# List OBS directories.
mox.file.list_directory('obs://bucket_name/my_dir/')
```

Precautions for Using the mox.file API

1. The mox.file API is primarily designed to enhance the ease of reading and downloading data from OBS.
2. For some APIs of OBS parallel file systems, there may be compatibility issues. It is recommended to use the OBS Python SDK directly for production business development.
3. For details about the APIs, see **API Overview of OBS SDK for Python**.

Through the mox.file API, you can access and manage OBS resources as conveniently as local files, thereby improving development efficiency.

Constraints

- The OBS bucket you need to access must be accessible by the current training job.
- You must have the read and write permissions on the OBS bucket.
- Moxing is preset in ModelArts standard images. Custom images are not supported.

File Copy

- Copy a file. **mox.file.copy** can only be used to perform operations on files. To perform operations on folders, use **mox.file.copy_parallel**.
  - Copy an OBS file from an OBS path to another OBS path. For example, copy **obs://bucket_name/obs_file.txt** to **obs://bucket_name/obs_file_2.txt**.
    ```
    import moxing as mox
    mox.file.copy('obs://bucket_name/obs_file.txt', 'obs://bucket_name/obs_file_2.txt')
    ```
  - Copy an OBS file to a local path, that is, download an OBS file. For example, download **obs://bucket_name/obs_file.txt** to **/tmp/obs_file.txt**.
    ```
    import moxing as mox
    mox.file.copy('obs://bucket_name/obs_file.txt', '/tmp/obs_file.txt')
    ```
  - Copy a local file to OBS, that is, upload an OBS file. For example, upload **/tmp/obs_file.txt** to **obs://bucket_name/obs_file.txt**.
    ```
    import moxing as mox
    mox.file.copy('/tmp/obs_file.txt', 'obs://bucket_name/obs_file.txt')
    # You can preload the file into the cache after copying it to OBS:
    mox.file.copy('/tmp/obs_file.txt', 'obs://bucket_name/obs_file.txt', is_preload=True)
    ```
  - Copy a local file to another local path. This operation is equivalent to **shutil.copyfile**. For example, copy **/tmp/obs_file.txt** to **/tmp/obs_file_2.txt**.
    ```
    import moxing as mox
    mox.file.copy('/tmp/obs_file.txt', '/tmp/obs_file_2.txt')
    ```
- For large files, **mox.file.copy** will use a segmented concurrent download method by default to speed up the process. The relevant parameters can be controlled through environment variables:

  a. Determine if a file is large: If the file size exceeds this threshold, segmented concurrent download will be enabled.
  ```
  MOX_FILE_PARTIAL_MAXIMUM_SIZE
  ```
  The default size is 5 GB. The unit is byte. To set the size to 5 GB, enter **5368709120**.

  b. Manage the size of file segments:
  ```
  MOX_FILE_LARGE_FILE_PART_SIZE
  ```
  The default size is 10 MB. The unit is byte. Due to the OBS limit of up to 10,000 segments, the segment size should be increased for files larger than 100 GB.

  c. Manage the number of concurrent download processes. The concurrent download process count determines how many threads are launched when downloading large files. In the new version, the default value is 8. If there are fewer nodes or fewer large files, you can increase the concurrency level appropriately to improve download performance.
  ```
  MOX_FILE_LARGE_FILE_TASK_NUM
  ```


The default value is **32**. If high concurrency causes OBS bucket throttling, reduce the concurrency number.

- Copy a folder. **mox.file.copy_parallel** can only be used to perform operations on folders. To perform operations on files, use **mox.file.copy**.
  - Copy an OBS file from an OBS path to another OBS path. For example, copy **obs://bucket_name/sub_dir_0** to **obs://bucket_name/sub_dir_1**.
    ```
    import moxing as mox
    mox.file.copy_parallel('obs://bucket_name/sub_dir_0', 'obs://bucket_name/sub_dir_1')
    ```
  - Copy an OBS folder to a local path, that is, download an OBS folder. For example, download **obs://bucket_name/sub_dir_0** to **/tmp/sub_dir_0**.
    ```
    import moxing as mox
    mox.file.copy_parallel('obs://bucket_name/sub_dir_0', '/tmp/sub_dir_0')
    ```
  - Copy a local folder to OBS, that is, upload an OBS folder. For example, upload **/tmp/sub_dir_0** to **obs://bucket_name/sub_dir_0**.
    ```
    import moxing as mox
    mox.file.copy_parallel('/tmp/sub_dir_0', 'obs://bucket_name/sub_dir_0')
    ```
  - Copy a local folder to another local path. This operation is equivalent to **shutil.copytree**. For example, copy **/tmp/sub_dir_0** to **/tmp/sub_dir_1**.
    ```
    import moxing as mox
    mox.file.copy_parallel('/tmp/sub_dir_0', '/tmp/sub_dir_1')
    ```

mox.file.copy_parallel uses the **threads** parameter to control the number of concurrent copy operations. The default value is **16**.

The **file_list** parameter specifies the files to be copied. For example, upload **/tmp/sub_dir_0/train/1.jpg** and **/tmp/sub_dir_0/eval/2.jpg** in **/tmp/sub_dir_0** to **obs://bucket_name/sub_dir_0**.

```
import moxing as mox
mox.file.copy_parallel('/tmp/sub_dir_0', 'obs://bucket_name/sub_dir_0', file_list=['train/1.jpg', 'eval/2.jpg'])
```

Read/Write

- Read an OBS file.

  For example, if you read the **obs://bucket_name/obs_file.txt** file, the content is returned as strings.
  ```
  import moxing as mox
  file_str = mox.file.read('obs://bucket_name/obs_file.txt')
  ```
  Alternatively, open the file object and read data from it.
  ```
  import moxing as mox
  with mox.file.File('obs://bucket_name/obs_file.txt', 'r') as f:
    file_str = f.read()
  ```
- Read a line from a file. A string that ends with a newline character is returned. You can also open the file object in OBS.
  ```
  import moxing as mox
  with mox.file.File('obs://bucket_name/obs_file.txt', 'r') as f:
    file_line = f.readline()
  ```
- Read all lines from a file. A list is returned, in which each element is a line and ends with a newline character.
  ```
  import moxing as mox
  with mox.file.File('obs://bucket_name/obs_file.txt', 'r') as f:
    file_line_list = f.readlines()
  ```
- Read an OBS file in binary mode.

  For example, if you read the **obs://bucket_name/obs_file.bin** file, the content is returned as bytes.
  ```
  import moxing as mox
  file_bytes = mox.file.read('obs://bucket_name/obs_file.bin',binary=True)
  ```


Alternatively, open the file object and read data from it.

```
import moxing as mox
with mox.file.File('obs://bucket_name/obs_file.bin', 'rb') as f:
  file_bytes = f.read()
```

One or all lines in a file opened in binary mode can be read with the same method.

- Write a string to a file (the content cannot exceed 2 GB).

  For example, write **Hello World!** into the **obs://bucket_name/obs_file.txt** file.

  ```
  import moxing as mox
  mox.file.write('obs://bucket_name/obs_file.txt', 'Hello World!')
  # You can preload the file into the cache after writing it to OBS:
  mox.file.write('obs://bucket_name/obs_file.txt', 'Hello World!', is_preload=True)
  ```

  You can also open the file object and write data into it. Both methods are the same.

  ```
  import moxing as mox
  with mox.file.File('obs://bucket_name/obs_file.txt', 'w') as f:
    f.write('Hello World!')
  ```

  NOTE

  When you open a file in write mode or call **mox.file.write**, if the file to be written does not exist, the file will be created. If the file to be written already exists, the file is overwritten.

- Append content to an OBS file.

  For example, append **Hello World!** to the **obs://bucket_name/obs_file.txt** file.

  ```
  import moxing as mox
  mox.file.append('obs://bucket_name/obs_file.txt', 'Hello World!')
  ```

  You can also open the file object and append content to it. Both methods are the same.

  ```
  import moxing as mox
  with mox.file.File('obs://bucket_name/obs_file.txt', 'a') as f:
    f.write('Hello World!')
  ```

  When you open a file in append mode or call **mox.file.append**, if the file to be appended does not exist, the file will be created. If the file to be appended already exists, the content is directly appended.

  If the size of the source file to be appended is large, for example, the **obs://bucket_name/obs_file.txt** file exceeds 5 MB, the append performance is low.

  NOTE

  If the file object is opened in write or append mode, when the **write** function is called, the content to be written is temporarily stored in the cache until the file object is closed (the file object is automatically closed when the **with** statement exits). Alternatively, you can call the **close()** or **flush()** function of the file object to write the file content.

List

- List an OBS directory. Only the top-level result (relative path) is returned. Recursive listing is not performed.

  For example, if you list **obs://bucket_name/object_dir**, all files and folders in the directory are returned, but recursive queries are not performed.

  Assume that **obs://bucket_name/object_dir** is in the following structure:


```
bucket_name
    |- object_dir
      |- dir0
        |- file00
      |- file1
```

Call the following code:

```
import moxing as mox
mox.file.list_directory('obs://bucket_name/object_dir')
```

The following list is returned:

```
['dir0', 'file1']
```

- Recursively list an OBS directory. All files and folders (relative paths) in the directory are returned, and recursive queries are performed.

  Assume that **obs://bucket_name/object_dir** is in the following structure:

  ```
  bucket_name
      |- object_dir
        |- dir0
          |- file00
        |- file1
  ```

  Call the following code:

  ```
  import moxing as mox
  mox.file.list_directory('obs://bucket_name/object_dir', recursive=True)
  ```

  The following list is returned:

  ```
  ['dir0', 'dir0/file00', 'file1']
  ```

Folder Creation

Create an OBS directory, that is, an OBS folder. Recursive creation is supported. That is, if the **sub_dir_0** folder does not exist, it is automatically created. If the **sub_dir_0** folder exists, no folder will be created.

```
import moxing as mox
mox.file.make_dirs('obs://bucket_name/sub_dir_0/sub_dir_1')
```

Query

- Check whether an OBS file exists. If the file exists, **True** is returned. If the file does not exist, **False** is returned.
  ```
  import moxing as mox
  mox.file.exists('obs://bucket_name/sub_dir_0/file.txt')
  ```
- Check whether an OBS folder exists. If the folder exists, **True** is returned. If the folder does not exist, **False** is returned.
  ```
  import moxing as mox
  mox.file.exists('obs://bucket_name/sub_dir_0/sub_dir_1')
  ```

  NOTE

  OBS allows files and folders with the same name (not allowed in UNIX). If a file or folder with the same name exists, for example, **obs://bucket_name/sub_dir_0/abc**, when **mox.file.exists** is called, **True** is returned regardless of whether **abc** is a file or folder.

- Check whether an OBS path is a folder. If it is a folder, **True** is returned. If it is not a folder, **False** is returned.
  ```
  import moxing as mox
  mox.file.is_directory('obs://bucket_name/sub_dir_0/sub_dir_1')
  ```


> **NOTE**
>
> OBS allows files and folders with the same name exist (not allowed in UNIX). If a file or folder with the same name exists, for example, **obs://bucket_name/sub_dir_0/abc**, when **mox.file.is_directory** is called, **True** is returned.

- Obtain the size of an OBS file, in bytes.

  For example, obtain the size of **obs://bucket_name/obs_file.txt**.

  ```
  import moxing as mox
  mox.file.get_size('obs://bucket_name/obs_file.txt')
  ```

- Recursively obtain the size of all files in an OBS folder, in bytes.

  For example, obtain the total size of all files in the **obs://bucket_name/object_dir** directory.

  ```
  import moxing as mox
  mox.file.get_size('obs://bucket_name/object_dir', recursive=True)
  ```

- Obtain the **stat** information about an OBS file or folder. The **stat** information contains the following:
  - **length**: File size.
  - **mtime_nsec**: Creation timestamp.
  - **is_directory**: Specifies whether the path is a folder.

  For example, if you want to query the OBS file **obs://bucket_name/obs_file.txt**, you can replace the file path with a folder path.

  ```
  import moxing as mox
  stat = mox.file.stat('obs://bucket_name/obs_file.txt')
  print(stat.length)
  print(stat.mtime_nsec)
  print(stat.is_directory)
  ```

Deletion

- Delete an OBS file.

  For example, delete **obs://bucket_name/obs_file.txt**.

  ```
  import moxing as mox
  mox.file.remove('obs://bucket_name/obs_file.txt')
  ```

- Delete an OBS folder and recursively delete all content in the folder. If the folder does not exist, an error is reported.

  For example, delete all content in **obs://bucket_name/sub_dir_0**.

  ```
  import moxing as mox
  mox.file.remove('obs://bucket_name/sub_dir_0', recursive=True)
  ```

Parameter Configuration

- Use the **mox.file.set_auth** function to set all configurable parameters.

  ```
  # Configure parameters for MoXing reading. Parameters that do not require modification can be left unspecified. This API should be executed globally once and must not be called multiple times.
  # ak – Access Key, string type. When using in ModelArts, the system configures it by default. For other environments, refer to unified identity authentication.
  # sk – Secret Access Key, string type. When using in ModelArts, the system configures it by default. For other environments, refer to unified identity authentication.
  # server – OBS server. When using in ModelArts, the system configures it by default. For other environments, refer to the configuration at https://support.huaweicloud.com/intl/en-us/productdesc-obs/obs_03_0152.html.
  # port – OBS server port. When using in ModelArts, the system configures it by default. For other environments, refer to the server configuration. Generally, a special port number is not provided and does not need to be configured.
  # is_secure – Specifies whether to use HTTPS. Boolean type. The default value is True.
  # ssl_verify – Specifies whether to use SSL verification. Boolean type. The default value is False.
  # long_conn_mode – Specifies whether to use long connection mode. Boolean type. The default value
  ```



```
is True.
# path_style – Specifies whether to use path style. Boolean type. HEC sets it to True by default, and public cloud sets it to False by default. It is generally not recommended to modify this.
# retry – Total number of attempts. Integer type. Integer type. The default value is 10, measured in times.
# retry_wait – Time to wait for each attempt. Float type. The default value is 0.1, measured in seconds. If not configured, the first attempt waits for 0.1 seconds, the second failure waits for 0.2 seconds, the third for 0.4 seconds, and so on, exponentially increasing. # client_timeout – OBS client timeout time. Integer type. The default value is 30, measured in seconds. # list_max_keys – Maximum number of objects listed per page, used for the list_directory API. Integer type. The default value is 1000.
#timeout_config - Sets the default timeout interval (in seconds) of each MoXing function. The default values are as follows:
{
  'read': 60,  # Timeout interval of the read function
  'listObjects': 10,  # Timeout interval of the list_directory function for listing 1,000 objects
  'deleteObject': 60,  # Timeout interval of the rename and remove functions for deleting objects
  'getObject': 60,  # Timeout interval of the copy function for obtaining OBS objects locally
  'getObjectMetadata': 30,  # Timeout interval of the copy and read functions for obtaining OBS object metadata
  'putFile': 120,  # Timeout interval of the copy function for uploading OBS files smaller than 5 GB locally
  'putContent': 30,  # Timeout interval of the copy_parallel function for creating OBS directories
  'copyPart': 120,  # Timeout interval of the copy function for copying file fragments between OBS buckets
  'copyObject': 120,  # Timeout interval of the copy function for copying objects between OBS buckets
  'uploadPart': 120,  # Timeout interval of the copy function for uploading OBS file fragments locally
  'initiateMultipartUpload': 30,  # Timeout interval of the copy function for initializing the upload of OBS files larger than 5 GB locally
  'completeMultipartUpload': 120,  # Timeout interval of the copy function for completing the upload of OBS files smaller than 5 GB locally
} # If the timeout occurs in the corresponding scenario, you can change the timeout interval of the corresponding function.
import moxing as mox
mox.file.set_auth(ak='xxx', sk='xxx')
```

- Delete an OBS folder and recursively delete all content in the folder. If the folder does not exist, an error is reported.

  For example, delete all content in **obs://bucket_name/sub_dir_0**.

  ```
  import moxing as mox
  mox.file.remove('obs://bucket_name/sub_dir_0', recursive=True)
  ```


14 Creating a Production Training Job (Old Version)

Developing models involves optimizing their performance effectively. Traditional methods require repeatedly testing various model designs, datasets, and hyperparameters, which takes significant time and effort but may still fail to deliver good results. ModelArts simplifies this process by offering tools for creating training jobs, tracking progress in real time, and managing versions. With ModelArts, users can test different configurations easily and identify the best-performing setup faster.

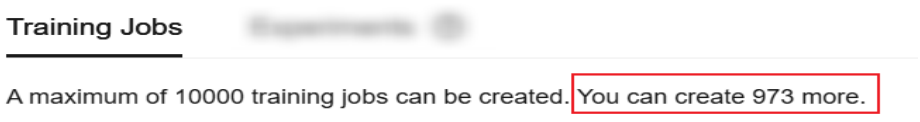
Create a production training job in either of the following ways:

- Use the ModelArts console. For details, see the following sections. This chapter provides the operation guide for the default GUI (that is, the old GUI). For details about the operation guide for the new GUI, see **Creating a Training Job**.
- Use the ModelArts API to create a production training job. For details, see **Using PyTorch to Create a Training Job (New-Version Training)**.

Notes and Constraints

By default, up to 10,000 training jobs can be created. You can view the remaining quota on the training job list page.

Figure 14-1 Viewing the remaining quota of a training job



Prerequisites

- Data for training uploaded to an OBS directory.
- At least one empty folder in OBS for storing training output.

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

- Account not in arrears (paid resources required for training jobs).
- OBS directory and ModelArts in the same region.
- Access authorization configured. If you have not yet configured access, follow the instructions in **Configuring Agency Authorization for ModelArts with One Click**.
- Training algorithm. For details, see **Creating an Algorithm**.

Billing

Model training in ModelArts uses compute and storage resources, which are billed. Compute resources are billed for running training jobs. Storage resources are billed for storing data in OBS or SFS. For details, see **Model Training Billing Items**.

Procedure

To create a training job, follow these steps:

Step 1 Follow the steps in **Accessing the Page for Creating a Training Job**.

Step 2 Follow the steps in **Configuring Basic Information**.

Step 3 Select an algorithm type.

- Use an existing algorithm to create a training job by referring to **Choosing an Algorithm Type (My Algorithm)**.
- Use a preset image to create a training job by referring to **Choosing an Algorithm Type (Custom Algorithm)**.
- Use a custom image to create a training job by referring to **Choosing a Boot Mode (Custom Image)**.

Step 4 Configure training parameters, including the input, output, hyperparameters, and environment variables. For details, see **Configuring Training Parameters**.

Step 5 Select a resource pool as required. A dedicated resource pool is recommended.

- **Configuring a Public Resource Pool**
- **Configuring a Dedicated Resource Pool**

Step 6 Select a training mode when you use a preset MindSpore engine and Ascend resources. For details, see **(Optional) Selecting a Training Mode**.

Step 7 Add tags if you want to manage training jobs by group. For details, see **(Optional) Adding Tags**.

Step 8 Perform follow-up procedure. For details, see **Follow-Up Operations**.

----End

Accessing the Page for Creating a Training Job

1. Log in to the **ModelArts console**.

2. In the navigation pane, choose **Model Training** > **Training Jobs**.
3. Click **Create Training Job** and click **Old Version** in the upper right corner. The old version of the **Create Training Job** page is displayed.

Configuring Basic Information

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

Table 14-1 Basic information

Parameter	Description
Name	Job name, which is mandatory. The system automatically generates a name, which you can then rename according to the following rules. • The name contains 1 to 64 characters. • Letters, digits, hyphens (-), and underscores (_) are allowed.
Description	Job description, which helps you learn about the job information in the training job list.
Experiment	Experiment for classifying and managing the job. • If you select **Create new**, enter the experiment name and description. • If you select **Use existing**, select an experiment name. • If you select **Not required**, this job will not be managed in any experiment.

Choosing an Algorithm Type (My Algorithm)

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

Choosing an Algorithm Type (Custom Algorithm)

If an algorithm is available in algorithm management, choose **My algorithm**. If no algorithm is available, choose **Custom algorithm**. If you use a custom algorithm to create a training job, select a boot mode by referring to **Table 14-2**.

Table 14-2 Creating a training job using a custom algorithm

Parameter	Description
Algorithm Type	Select **Custom algorithm**. This parameter is mandatory.

Parameter	Description
Boot Mode	Select **Preset image** and select the preset image engine and engine version to be used by the training job. If you select **Customize** for the engine version, select a custom image from **Image**.
Image	This parameter is displayed and mandatory only when the preset image version is set to **Customize**. You can set the container image path in either of the following ways: ● To select your image or an image shared by others, click **Select** on the right and select a container image for training. The required image must be uploaded to SWR beforehand. ● To select a public image, enter the address of the public image in SWR. Enter the image path in the format of "Organization name/Image name:Version name". Do not contain the domain name in the path because the system will automatically add the domain name to the path.
Code Source	Select a training code source. ● OBS: Select **OBS** if the training code is stored in an OBS bucket. ● SFS: Select **SFS** if the training code is stored in an SFS file system.
Code Directory	This parameter is available only when **Code Source** is set to **OBS**. Select the OBS directory where the training code file is stored. This parameter is mandatory. ● Upload code to the OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1000, and the folder depth cannot exceed 32. ● The training code file is automatically downloaded to the **${MA_JOB_DIR}/demo-code** directory of the training container when the training job is started. **demo-code** is the last-level OBS directory for storing the code. For example, if **Code Directory** is set to **/test/code**, the training code file is downloaded to the **${MA_JOB_DIR}/code** directory of the training container. **NOTE** Encrypt sensitive data before saving it to your OBS bucket.
Boot File	Select the Python boot script of the training job in the code directory. This parameter is mandatory. ModelArts supports only the boot file written in Python. Therefore, the boot file must end with .py.

Parameter	Description
Local Code Directory	This parameter is available in **More Configurations** only when **Code Source** is set to **OBS**. This parameter is optional. This parameter specifies the local directory of the training container. When training starts, the code directory is downloaded to this directory. The default local code directory is **/home/ma-user/modelarts/user-job-dir**. Cannot be under **/home/ma-user/modelarts/***, **/home/ma-user/modelarts-dev/***, **/home/ma-user/infer/***, or **/home/ma-user**. Click **Preview Runtime Environment** in the upper right corner of the page to view the work directory of the training job.
Work Directory	During training, the system automatically runs the **cd** command to execute the boot file in this directory.

Selecting a preset image with customization results in the same background behavior as running a training job directly with that 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.

Choosing a Boot Mode (Custom Image)

If you use a custom image to create a training job, select a boot mode by referring to **Table 14-3**.

Table 14-3 Creating a training job using a custom image

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

Parameter	Description
Image	Container image path. This parameter is mandatory. You can set the container image path in either of the following ways: ● To select your image or an image shared by others, click **Select** on the right and select a container image for training. The required image must be uploaded to SWR beforehand. ● To select a public image, enter the address of the public image in SWR. Enter the image path in the format of "Organization name/Image name:Version name". Do not contain the domain name in the path because the system will automatically add the domain name to the path.
Code Directory	OBS directory where the training code file is stored. Configure this parameter only if your custom image does not contain training code. ● Upload code to the OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1000, and the folder depth cannot exceed 32. ● The training code file is automatically downloaded to the **${MA_JOB_DIR}/demo-code** directory of the training container when the training job is started. **demo-code** is the last-level OBS directory for storing the code. For example, if **Code Directory** is set to **/test/code**, the training code file is downloaded to the **${MA_JOB_DIR}/code** directory of the training container.
User ID	User ID for running the container. The default value **1000** is recommended. 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** If the user ID is set to **0**, the user in the training container is **root**.

Parameter	Description

Boot Command	Command for booting an image. This parameter is mandatory.
	When a training job is running, the boot command is automatically executed after the code directory is downloaded.
	● If the training boot script is a .py file, **train.py** for example, the boot command is as follows:
	`python ${MA_JOB_DIR}/demo-code/train.py`
	● If the training boot script is a .sh file, **main.sh** for example, the boot command is as follows:
	`bash ${MA_JOB_DIR}/demo-code/main.sh`
	You can use semicolons (;) and ampersands (&&) to combine multiple commands. **demo-code** in the command is the last-level OBS directory where the code is stored. Replace it with the actual one.
	If there are input pipes, output pipes, or hyperparameters, ensure that the last command of the boot command runs the training script.
	Reason: The system appends input pipes, output pipes, and hyperparameters to the end of the boot command. If the last command is not the training script, an error will occur.
	Example: If the last line of the boot command is **python train.py** and the **--data_url** hyperparameter exists, the system executes **python train.py --data_url=/input** when running properly. However, if the boot command ends with another command, such as:
	`python train.py`
	`pwd  # The last command is pwd instead of the training script.`
	The system will execute **python train.py pwd --data_url=/input**, leading to an error.
	NOTE
	To ensure data security, do not enter sensitive information, such as plaintext passwords.

Local Code Directory	This parameter is available in **More Configurations** only when **Code Source** is set to **OBS**. This parameter is optional. This parameter specifies the local directory of the training container. When training starts, the code directory is downloaded to this directory. The default local code directory is **/home/ma-user/modelarts/user-job-dir**. Cannot be under **/home/ma-user/modelarts/***, **/home/ma-user/modelarts-dev/***, **/home/ma-user/infer/***, or **/home/ma-user**. Click **Preview Runtime Environment** in the upper right corner of the page to view the work directory of the training job.
Work Directory	During training, the system automatically runs the **cd** command to execute the boot file in this directory.

For details about how to use custom images supported by training, see **Boot Command Specifications for Custom Images**.

Configuring Training Parameters

Data is obtained from an OBS bucket or dataset for model training. The training output can also be stored in an OBS bucket. When creating a training job, you can configure parameters such as input, output, hyperparameters, and environment variables by referring to **Table 14-4**.

NOTE

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

Table 14-4 Configuring training parameters

Parameter	Sub-Parameter	Description
Input	Parameter name	The algorithm code reads the training input data based on the input parameter name. The recommended value is **data_url**. The training input parameters must match the input parameters of the selected algorithm. For details, see **Table 4-4**.
	Dataset	Click **Dataset** and select the target dataset and its version in the ModelArts dataset list. When the training job is started, ModelArts automatically downloads the data in the input path to the training container.
	Data path	Click **Data path** and select the storage path to the training input data from an OBS bucket. Files must not exceed 10 GB in total size, 1,000 in number, or 1 GB per file. When the training job is started, ModelArts automatically downloads the data in the input path to the training container.

	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", "")`

Parameter	Sub-Parameter	Description
Output	Parameter name	The algorithm code reads the training output data based on the output parameter name. The recommended value is **train_url**. The training output parameters must match the output parameters of the selected algorithm. For details, see **Table 4-5**.
	Data path	Click **Data path** and select the storage path for the training output data from an OBS bucket. Files must not exceed 1 GB in total size, 128 in number, or 128 MB per file. During training, the system automatically synchronizes files from the local code directory of the training container to the data path. **NOTE** The data path can only be an OBS path. To prevent any issues with data storage, choose an empty directory as the data path.

	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	Indicates whether to pre-download the files in the output directory to a local directory. • If you set **Predownload** to **No**, the system does not download the files in the training output data path to a local directory of the training container when the training job is started. • If you set **Predownload** to **Yes**, the system automatically downloads the files in the training output data path to a local directory of the training container when the training job is started. The larger the file size, the longer the download time. To avoid excessive training time, remove any unneeded files from the local code directory of the training container as soon as possible. To use **Resumable Training**, select **Yes**.

Parameter	Sub-Parameter	Description
Hyperparameter	N/A	Used for training tuning. This parameter is determined by the selected algorithm. If hyperparameters have been defined in the algorithm, all hyperparameters in the algorithm are displayed. Hyperparameters can be modified and deleted. The status depends on the hyperparameter constraint settings in the algorithm. For details, see **Table 4-6**. To import hyperparameters in batches, click **Upload**. You will need to fill in the hyperparameters based on the provided template. The total number of hyperparameters should not exceed 100, or the import will fail. **NOTE** To ensure data security, do not enter sensitive information, such as plaintext passwords.
Environment Variable	N/A	Add environment variables based on service requirements. For details about the environment variables preset in the training container, see **Managing Environment Variables of a Training Container**. To import environment variables in batches, click **Upload**. You will need to fill in the environment variables based on the provided template. The total number of environment variables should not exceed 100, or the import will fail. **NOTE** To ensure data security, do not enter sensitive information, such as plaintext passwords.

Parameter	Sub-Parameter	Description
Auto Restart	N/A	Once this feature is enabled, you can set the number of restarts and whether to enable unconditional auto restart and restart upon suspension. After you enable auto restart, ModelArts will handle any exceptions caused by environmental issues during a training job. It will either automatically handle the exception or isolate the faulty node and then restart the job, which helps to increase the success rate of the training. To avoid losing training progress and make full use of compute, ensure that your code logic supports resumable training before enabling this function. For details, see **Resumable Training**. The value ranges from **1** to **128**. The default value is **3**. The value cannot be changed once the training job is created. Set this parameter based on your needs. If **Unconditional auto restart** is selected, the training job will be restarted unconditionally once the system detects a training exception. To prevent invalid restarts, it supports a maximum of three consecutive unconditional restarts. ModelArts continuously monitors job processes to detect suspension and optimize resource usage. When **Restart Upon Suspension** is enabled, suspended jobs can be automatically restarted at the process level. CPU specifications do not support job restarts upon suspension. However, ModelArts does not verify code logic, and suspension detection is periodic, which may result in false reports. By enabling this feature, you acknowledge the possibility of false positives. To prevent unnecessary restarts, ModelArts limits consecutive restarts to three. If auto restart is triggered during training, the system records the restart information. You can check the fault recovery details on the training job details page. For details, see **Training Job Fault Tolerance Check**.

Configuring a Public Resource Pool

To configure a public resource pool, refer to **Table 14-5**.

Table 14-5 Configuring a public resource pool for a training job

Parameter	Description
Resource Pool	Select **Public resource pool**.

Parameter	Description
Resource Type	Select the resource type required for training. This parameter is mandatory. If a resource type has been defined in the training code, select a proper resource type based on algorithm constraints. For example, if the resource type defined in the training code is CPU and you select other types, the training job fails. If some resource types are invisible or unavailable for selection, they are not supported.
Specifications	Select the required resource specifications based on the resource type. If **Data path** is selected for **Input**, you can click **Check Input Size** on the right to ensure the storage is larger than the input data size.  **NOTE** • The instance specifications **GPU:*n**tnt004** (*n* indicates a specific number) do not support multi-process training. • Use a custom image with the same image and resource types as your instance when creating a job, for example, only GPU types. Otherwise, the training job will fail.
Compute Nodes	Select the number of instances as required. The default value is **1**. • If only one instance is used, a single-node training job is created. ModelArts starts one training container on this node. The training container exclusively uses the compute resources of the selected flavor. • If more than one instance is used, a distributed training job is created. For more information about distributed training configurations, see **Overview**. Before creating distributed training jobs, pre-install all required pip dependencies (see **Installing pip Dependencies in an Image**). If there are more than 10 nodes, the system automatically deletes the pip source configuration. Executing pip install commands during training may cause training failures.
Persistent Log Saving	If you select CPU or GPU flavors, **Persistent Log Saving** is available for you to configure. • After this feature is enabled (default), configure **Job Log Path**. The system permanently stores training logs to the specified OBS path. • After this feature is disabled, ModelArts automatically stores the logs for 30 days. You can download all logs on the job details page to a local path.

Parameter	Description
Job Log Path	When enabling **Persistent Log Saving** or selecting Ascend resources, select an empty OBS directory for **Job Log Path** to store log files generated by the training job. Ensure that you have read and write permissions to the selected OBS directory.
Event Notification	Indicates whether to enable event notification. • This feature is disabled by default, which means SMN is disabled. • After this feature is enabled, you will be notified of specific events, such as job status changes or suspected suspensions, via an SMS or email. Notifications will be billed based on SMN pricing. In this case, you must configure the topic name and events. – **Topic**: topic of event notifications. Click **Create Topic** to create a topic on the SMN console. – **Event**: events you want to subscribe to. Examples: **JobStarted**, **JobCompleted**, **JobFailed**, **JobTerminated**, and **JobHanged**. **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. For details, see **Adding a Subscription**. • SMN charges you for the number of notification messages. For details, see **Billing**. • Only training jobs using GPUs or NPUs support **JobHanged** events.
Auto Stop	When using paid resources, you can determine whether to enable auto stop. • This function is disabled by default; the training job keeps running until the training is completed. • If this function is enabled, configure the auto stop time. The value can be **1 hour**, **2 hours**, **4 hours**, **6 hours**, or **Customize**. The customized time must range from 1 hour to 720 hours. When you enable this feature, the training stops automatically when the time limit is reached. The time limit does not count down when the training is paused.
training_ssh_configure_nodes	Specifies whether to enable password-free SSH mutual trust between nodes. • This feature is disabled by default. • Enabling this feature requires you to configure the SSH key directory. This is where the automatically generated SSH key file will be stored within the training container. The default value is **/home/ma-user/.ssh**.

Configuring a Dedicated Resource Pool

To configure a dedicated resource pool, refer to **Table 14-6**.

Table 14-6 Configuring a dedicated resource pool for a training job

Parameter	Description
Resource Pool	Select a dedicated resource pool. If you select a dedicated resource pool, you can view the name, status, node specifications, number of idle/fragmented nodes, number of available/total nodes, and number of cards of the resource pool. Hover over **View** in the **Idle/Fragmented Nodes** column to check fragment details and check whether the resource pool meets the training requirements.
Specifications	Select the required resource specifications based on the resource type. If **Data path** is selected for **Input**, you can click **Check Input Size** on the right to ensure the storage is larger than the input data size.  **NOTE** The resource specifications **GPU:*n**tnt004** (*n* indicates a specific number) do not support multi-process training.
Compute Nodes	Select the number of instances as required. The default value is **1**. • If only one instance is used, a single-node training job is created. ModelArts starts one training container on this node. The training container exclusively uses the compute resources of the selected flavor. • If more than one instance is used, a distributed training job is created. For more information about distributed training configurations, see **Overview**. Before creating distributed training jobs, pre-install all required pip dependencies (see **Installing pip Dependencies in an Image**). If there are more than 10 nodes, the system automatically deletes the pip source configuration. Executing pip install commands during training may cause training failures.

Parameter	Description
Job Priority	• When using a dedicated resource pool, you can set and change the scheduling priority of the training job. This parameter is not supported when a public resource pool is used. • The platform handles jobs by prioritizing them from highest to lowest. If multiple jobs share the same priority, they are scheduled in the order they were submitted. When resources are available, the earliest-submitted job gets processed first. • Changing the number changes the priority of the job in the queue. The priority can be set to **1**, **2**, or **3**. A larger number indicates a higher priority. The default priority is **1**, and the highest priority is **3**. • To set the priority to **3**, you will also need the permission. For details about how to set the permission, see **Assigning the Permission to Set the Highest Job Priority to an IAM User**. • If a training job is in the **Pending** state for a long time, you can change the job priority to reduce the queuing duration. For details, see **Priority of a Training Job**.
Preemption	When enabled, jobs that allow preemption may be terminated and re-queued if resource pool capacity is insufficient. To avoid losing training progress, configure resumable training before enabling this function. For details, see **Resumable Training**.

Parameter	Description

SFS Turbo	When ModelArts and SFS Turbo are directly connected, multiple SFS Turbo file systems can be mounted to a training job to store training data. Click **Add Mount Configuration** and set the following parameters:
	• **Name**: Select an SFS Turbo file system.
	• **Mount Path**: Enter the SFS Turbo mounting path in the training container.
	• **Directory**: Specify the SFS Turbo storage location. If you have configured the folder control permission, select a storage location. If you have not configured the folder control permission, retain the default value **/** or customize a location.
	• **Mounting Mode**: Permission on the mounted SFS Turbo file system. This parameter is displayed as **Read/Write** or **Read-only** based on the permission of the SFS Turbo storage location. If you have not configured the folder control permission, this parameter is unavailable.
	• **Mount Options**: Configure SFS mount parameters to accelerate and optimize training. For details about the parameters, see **Configuring SFS Turbo Mount Options**. Alternatively, retain the default settings below:
	`mountOptions:`
	`- vers=3`
	`- timeo=600`
	`- nolock`
	`- hard`
	NOTE
	• You can mount a file system multiple times, but each mount path must be distinct. A maximum of five disks can be mounted to a training job.
	• The mounting path cannot be a **/** directory or a default mounting path, such as **/cache** and **/home/ma-user/modelarts**.
	• For details about how to set permissions for SFS Turbo folders, see **Permissions Management**.

Persistent Log Saving	If you select CPU or GPU flavors, **Persistent Log Saving** is available for you to configure. • After this feature is enabled (default), configure **Job Log Path**. The system permanently stores training logs to the specified OBS path. • After this feature is disabled, ModelArts automatically stores the logs for 30 days. You can download all logs on the job details page to a local path.
Job Log Path	When enabling **Persistent Log Saving** or selecting Ascend resources, select an empty OBS directory for **Job Log Path** to store log files generated by the training job. Ensure that you have read and write permissions to the selected OBS directory.

Parameter	Description
Event Notification	Indicates whether to enable event notification. • This feature is disabled by default, which means SMN is disabled. • After this feature is enabled, you will be notified of specific events, such as job status changes or suspected suspensions, via an SMS or email. Notifications will be billed based on SMN pricing. In this case, you must configure the topic name and events. – **Topic**: topic of event notifications. Click **Create Topic** to create a topic on the SMN console. – **Event**: events you want to subscribe to. Examples: **JobStarted**, **JobCompleted**, **JobFailed**, **JobTerminated**, and **JobHanged**. **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. For details, see **Adding a Subscription**. • SMN charges you for the number of notification messages. For details, see **Billing**. • Only training jobs using GPUs or NPUs support **JobHanged** events.
Auto Stop	When using paid resources, you can determine whether to enable auto stop. • This function is disabled by default; the training job keeps running until the training is completed. • If this function is enabled, configure the auto stop time. The value can be **1 hour**, **2 hours**, **4 hours**, **6 hours**, or **Customize**. The customized time must range from 1 hour to 720 hours. When you enable this feature, the training stops automatically when the time limit is reached. The time limit does not count down when the training is paused.
training_ssh_configure_nodes	Specifies whether to enable password-free SSH mutual trust between nodes. • This feature is disabled by default. • Enabling this feature requires you to configure the SSH key directory. This is where the automatically generated SSH key file will be stored within the training container. The default value is **/home/ma-user/.ssh**.

(Optional) Selecting a Training Mode

Select a training mode when you use a preset MindSpore engine and Ascend resources. ModelArts provides three training modes for you to select. You can obtain different diagnosis information based on the actual scenario.

- Common mode: It is the default training scenario.
- High performance mode: In this mode, certain O&M functions will be adjusted or even disabled to accelerate the running speed, but this will

deteriorate fault locating. This mode is suitable for stable networks requiring high performance.

- Fault diagnosis mode: In this 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.

(Optional) Adding Tags

If you want to manage training jobs by group using tags, select **Configure Now** for **Advanced Configuration** to set tags for training jobs. For details about how to use tags, see **Using TMS Tags to Manage Resources by Group**.

Follow-Up Operations

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

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

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

FAQs

1. **How Do I View the Resource Usage of a Training Job in ModelArts?**

