

ModelArts

Getting Started

Issue 01
Date 2025-08-29



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

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

Trademarks and Permissions



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

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

Notice

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

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

Huawei Cloud Computing Technologies Co., Ltd.

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

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

Contents

1 How to Use ModelArts..... 1

2 Building a Handwritten Digit Recognition Model with ModelArts Standard.....2

3 Practices for Beginners.....16

1 How to Use ModelArts

ModelArts is a one-stop development platform for AI developers. It provides lifecycle management of AI development, helping you quickly build models and deploy the models on devices, edge devices, and the cloud.

ModelArts supports automated machine learning, namely, ExeML, and provides multiple pre-trained models. In addition, it integrates JupyterLab Notebook to provide online code development environments.

This document provides tutorials to help you quickly understand ModelArts functions. You can select tutorials based on your AI experience.

Selecting a Use Mode Based on Your Experience

- If you are an AI engineer and are familiar with code compilation and debugging, you can use the online code development environment provided by ModelArts to compile training code for AI model development. For details, see [Modeling with Notebook Instances](#).
- If you have your own algorithms and want to migrate them to ModelArts for training and inference, see [Using a Custom Algorithm to Build a Handwritten Digit Recognition Model](#).

2 Building a Handwritten Digit Recognition Model with ModelArts Standard

This section describes how to modify a local custom algorithm to train and deploy models on ModelArts.

Scenarios

This case describes how to use PyTorch 1.8 to recognize handwritten digit images. An official MNIST dataset is used in this case.

Through this case, you can learn how to train jobs, deploy an inference model, and perform prediction on ModelArts.

Process

Before performing the following operations, complete necessary operations by referring to [Preparations](#).

1. **Step 1 Prepare Training Data:** Download the MNIST dataset.
2. **Step 2: Preparing Training Files and Inference Files:** Write training and inference code.
3. **Step 3: Creating an OBS Bucket and Upload Files to OBS:** Create an OBS bucket and folder, and upload the dataset, training script, inference script, and inference configuration file to OBS.
4. **Step 4 Create a Training Job:** Train a model.
5. **Step 5 Deploying the Model for Inference:** Import the trained model to ModelArts, create a model, and deploy the model as a real-time service.
6. **Step 6 Performing Prediction:** Upload a handwritten digit image and send an inference request to obtain the inference result.
7. **Step 7 Releasing Resources:** Stop the service and delete the data in OBS to stop billing.

Preparations

- You have registered a Huawei ID and enabled Huawei Cloud services, and the account is not in arrears or frozen.

- Configure an agency.
To use ModelArts, access to Object Storage Service (OBS), SoftWare Repository for Container (SWR), and Intelligent EdgeFabric (IEF) is required. If this is the first time you use ModelArts, configure an agency to authorize access to these services.
 - a. Log in to the **ModelArts console** using your Huawei Cloud account. In the navigation pane on the left, choose **Permission Management**. On the **Permission Management** page, click **Add Authorization**.
 - b. In the displayed dialog box, configure the following parameters:
 - **Authorized User:** All users.
 - **Agency:** Add agency.
 - **Permissions:** Common.Select "I have read and agree to the ModelArts Service Statement" and click **Create**.

Figure 2-1 Configuring an agency

The screenshot shows the 'Configuration' dialog in the ModelArts console. It has three main sections: 'Authorized User', 'Agency', and 'Permissions'. Under 'Authorized User', 'All users' is selected. Under 'Agency', 'Add agency' is selected. The 'Agency Name' field contains 'modelarts_agency_b63a'. Under 'Permissions', the 'Common' option is selected. A note at the bottom states: 'A maximum of 10000 agencies can be created. You can create 9559 more.'

- c. After the configuration, view the agency configurations of your account on the **Permission Management** page.

Figure 2-2 Viewing agency configurations

Authorized To	Authorized User	Authorization Type	Authorization Content	Creation Time	Operation
all-users	All users	Agency	modelarts_agency_b0b0	Mar 08, 2024 16:27:12 GMT+08:00	View Permissions Delete

Step 1 Prepare Training Data

Download the MNIST dataset from a web browser. Ensure the four files in **Figure 2-3** are all downloaded.

Figure 2-3 MNIST dataset

Four files are available on this site:

`train-images-idx3-ubyte.gz`: training set images (9912422 bytes)
`train-labels-idx1-ubyte.gz`: training set labels (28881 bytes)
`t10k-images-idx3-ubyte.gz`: test set images (1648877 bytes)
`t10k-labels-idx1-ubyte.gz`: test set labels (4542 bytes)

- **train-images-idx3-ubyte.gz**: compressed package of the training set, which contains 60,000 samples.
- **train-labels-idx1-ubyte.gz**: compressed package of the training set labels, which contains the labels of the 60,000 samples
- **t10k-images-idx3-ubyte.gz**: compressed package of the validation set, which contains 10,000 samples.
- **t10k-labels-idx1-ubyte.gz**: compressed package of the validation set labels, which contains the labels of the 10,000 samples

Step 2: Preparing Training Files and Inference Files

In this case, ModelArts provides the training script, inference script, and inference configuration file.

NOTE

When pasting code from a .py file, create a .py file. Otherwise, the error message "SyntaxError: 'gbk' codec can't decode byte 0xa4 in position 324: illegal multibyte sequence" may be displayed.

Create the training script **train.py** on the local host. The content is as follows:

```
# base on https://github.com/pytorch/examples/blob/main/mnist/main.py

from __future__ import print_function

import os
import gzip
import codecs
import argparse
from typing import IO, Union

import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR

import shutil

# Define a network model.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
```

```
self.dropout2 = nn.Dropout(0.5)
self.fc1 = nn.Linear(9216, 128)
self.fc2 = nn.Linear(128, 10)

def forward(self, x):
    x = self.conv1(x)
    x = F.relu(x)
    x = self.conv2(x)
    x = F.relu(x)
    x = F.max_pool2d(x, 2)
    x = self.dropout1(x)
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.dropout2(x)
    x = self.fc2(x)
    output = F.log_softmax(x, dim=1)
    return output

# Train the model. Set the model to the training mode, load the training data, calculate the loss function,
and perform gradient descent.
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{}] ({:.0f}%) \tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if args.dry_run:
                break

# Validate the model. Set the model to the validation mode, load the validation data, and calculate the loss
function and accuracy.
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%) \n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

# The following is PyTorch MNIST.
# https://github.com/pytorch/vision/blob/v0.9.0/torchvision/datasets/mnist.py
def get_int(b: bytes) -> int:
    return int(codecs.encode(b, 'hex'), 16)

def open_maybe_compressed_file(path: Union[str, IO]) -> Union[IO, gzip.GzipFile]:
    """Return a file object that possibly decompresses 'path' on the fly.
    Decompression occurs when argument 'path' is a string and ends with '.gz' or '.xz'.
```



```
"""
if not isinstance(path, torch._six.string_classes):
    return path
if path.endswith('.gz'):
    return gzip.open(path, 'rb')
if path.endswith('.xz'):
    return lzma.open(path, 'rb')
return open(path, 'rb')

SN3_PASCALVINCENT_TYPEMAP = {
    8: (torch.uint8, np.uint8, np.uint8),
    9: (torch.int8, np.int8, np.int8),
    11: (torch.int16, np.dtype('>i2'), 'i2'),
    12: (torch.int32, np.dtype('>i4'), 'i4'),
    13: (torch.float32, np.dtype('>f4'), 'f4'),
    14: (torch.float64, np.dtype('>f8'), 'f8')
}

def read_sn3_pascalvincent_tensor(path: Union[str, IO], strict: bool = True) -> torch.Tensor:
    """Read an SN3 file in "Pascal Vincent" format (Lush file 'libidx/idx-io.lsh').
    Argument may be a filename, compressed filename, or file object.
    """
    # read
    with open_maybe_compressed_file(path) as f:
        data = f.read()
    # parse
    magic = get_int(data[0:4])
    nd = magic % 256
    ty = magic // 256
    assert 1 <= nd <= 3
    assert 8 <= ty <= 14
    m = SN3_PASCALVINCENT_TYPEMAP[ty]
    s = [get_int(data[4 * (i + 1): 4 * (i + 2)]) for i in range(nd)]
    parsed = np.frombuffer(data, dtype=m[1], offset=(4 * (nd + 1)))
    assert parsed.shape[0] == np.prod(s) or not strict
    return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)

def read_label_file(path: str) -> torch.Tensor:
    with open(path, 'rb') as f:
        x = read_sn3_pascalvincent_tensor(f, strict=False)
    assert(x.dtype == torch.uint8)
    assert(x.ndimension() == 1)
    return x.long()

def read_image_file(path: str) -> torch.Tensor:
    with open(path, 'rb') as f:
        x = read_sn3_pascalvincent_tensor(f, strict=False)
    assert(x.dtype == torch.uint8)
    assert(x.ndimension() == 3)
    return x

def extract_archive(from_path, to_path):
    to_path = os.path.join(to_path, os.path.splitext(os.path.basename(from_path))[0])
    with open(to_path, "wb") as out_f, gzip.GzipFile(from_path) as zip_f:
        out_f.write(zip_f.read())
    # The above is pytorch mnist.
    # --- end

# Raw MNIST dataset processing
def convert_raw_mnist_dataset_to_pytorch_mnist_dataset(data_url):
    """
    raw
```

```
{data_url}/
train-images-idx3-ubyte.gz
train-labels-idx1-ubyte.gz
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz

processed

{data_url}/
train-images-idx3-ubyte.gz
train-labels-idx1-ubyte.gz
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz
MNIST/raw
    train-images-idx3-ubyte
    train-labels-idx1-ubyte
    t10k-images-idx3-ubyte
    t10k-labels-idx1-ubyte
MNIST/processed
    training.pt
    test.pt
"""

resources = [
    "train-images-idx3-ubyte.gz",
    "train-labels-idx1-ubyte.gz",
    "t10k-images-idx3-ubyte.gz",
    "t10k-labels-idx1-ubyte.gz"
]

pytorch_mnist_dataset = os.path.join(data_url, 'MNIST')

raw_folder = os.path.join(pytorch_mnist_dataset, 'raw')
processed_folder = os.path.join(pytorch_mnist_dataset, 'processed')

os.makedirs(raw_folder, exist_ok=True)
os.makedirs(processed_folder, exist_ok=True)

print('Processing...')

for f in resources:
    extract_archive(os.path.join(data_url, f), raw_folder)

training_set = (
    read_image_file(os.path.join(raw_folder, 'train-images-idx3-ubyte')),
    read_label_file(os.path.join(raw_folder, 'train-labels-idx1-ubyte'))
)
test_set = (
    read_image_file(os.path.join(raw_folder, 't10k-images-idx3-ubyte')),
    read_label_file(os.path.join(raw_folder, 't10k-labels-idx1-ubyte'))
)
with open(os.path.join(processed_folder, 'training.pt'), 'wb') as f:
    torch.save(training_set, f)
with open(os.path.join(processed_folder, 'test.pt'), 'wb') as f:
    torch.save(test_set, f)

print('Done!')

def main():
    # Define the preset running parameters of the training job.
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')

    parser.add_argument('--data_url', type=str, default=False,
                        help='mnist dataset path')
    parser.add_argument('--train_url', type=str, default=False,
                        help='mnist model path')

    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
```

```
parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                    help='input batch size for testing (default: 1000)')
parser.add_argument('--epochs', type=int, default=14, metavar='N',
                    help='number of epochs to train (default: 14)')
parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                    help='learning rate (default: 1.0)')
parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                    help='Learning rate step gamma (default: 0.7)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--dry-run', action='store_true', default=False,
                    help='quickly check a single pass')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')
parser.add_argument('--save-model', action='store_true', default=True,
                    help='For Saving the current Model')
args = parser.parse_args()

use_cuda = not args.no_cuda and torch.cuda.is_available()

torch.manual_seed(args.seed)

# Set whether to use GPU or CPU to run the algorithm.
device = torch.device("cuda" if use_cuda else "cpu")

train_kwargs = {'batch_size': args.batch_size}
test_kwargs = {'batch_size': args.test_batch_size}
if use_cuda:
    cuda_kwargs = {'num_workers': 1,
                   'pin_memory': True,
                   'shuffle': True}
    train_kwargs.update(cuda_kwargs)
    test_kwargs.update(cuda_kwargs)

# Define the data preprocessing method.
transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Convert the raw MNIST dataset to a PyTorch MNIST dataset.
convert_raw_mnist_dataset_to_pytorch_mnist_dataset(args.data_url)

# Create a training dataset and a validation dataset.
dataset1 = datasets.MNIST(args.data_url, train=True, download=False,
                          transform=transform)
dataset2 = datasets.MNIST(args.data_url, train=False, download=False,
                          transform=transform)

# Create iterators for the training dataset and the validation dataset.
train_loader = torch.utils.data.DataLoader(dataset1, **train_kwargs)
test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)

# Initialize the neural network model and copy the model to the compute device.
model = Net().to(device)
# Define the training optimizer and learning rate for gradient descent calculation.
optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)

# Train the neural network and perform validation in each epoch.
for epoch in range(1, args.epochs + 1):
    train(args, model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
    scheduler.step()

# Save the model and make it adapted to the ModelArts inference model package specifications.
if args.save_model:
```

```
# Create the model directory in the path specified in train_url.
model_path = os.path.join(args.train_url, 'model')
os.makedirs(model_path, exist_ok = True)

# Save the model to the model directory based on the ModelArts inference model package
specifications.
torch.save(model.state_dict(), os.path.join(model_path, 'mnist_cnn.pt'))

# Copy the inference code and configuration file to the model directory.
the_path_of_current_file = os.path.dirname(__file__)
shutil.copyfile(os.path.join(the_path_of_current_file, 'infer/customize_service.py'),
os.path.join(model_path, 'customize_service.py'))
shutil.copyfile(os.path.join(the_path_of_current_file, 'infer/config.json'), os.path.join(model_path,
'config.json'))

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

Create the inference script **customize_service.py** on the local host. The content is as follows:

```
import os
import log
import json

import torch.nn.functional as F
import torch.nn as nn
import torch
import torchvision.transforms as transforms

import numpy as np
from PIL import Image

from model_service.pytorch_model_service import PTServingBaseService

logger = log.getLogger(__name__)

# Define model preprocessing.
infer_transformation = transforms.Compose([
    transforms.Resize(28),
    transforms.CenterCrop(28),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Model inference service
class PTVisionService(PTServingBaseService):

    def __init__(self, model_name, model_path):
        # Call the constructor of the parent class.
        super(PTVisionService, self).__init__(model_name, model_path)

        # Call the customized function to load the model.
        self.model = Mnist(model_path)

        # Load labels.
        self.label = [0,1,2,3,4,5,6,7,8,9]

    # Receive the request data and convert it to the input format acceptable to the model.
    def _preprocess(self, data):
        preprocessed_data = {}
        for k, v in data.items():
            input_batch = []
            for file_name, file_content in v.items():
                with Image.open(file_content) as image1:
                    # Gray processing
                    image1 = image1.convert("L")
                    if torch.cuda.is_available():
                        input_batch.append(infer_transformation(image1).cuda())
```

```
        else:
            input_batch.append(infer_transformation(image1))
            input_batch_var = torch.autograd.Variable(torch.stack(input_batch, dim=0), volatile=True)
            print(input_batch_var.shape)
            preprocessed_data[k] = input_batch_var

    return preprocessed_data

# Post-process the inference result to obtain the expected output format. The result is the returned value.
def _postprocess(self, data):
    results = []
    for k, v in data.items():
        result = torch.argmax(v[0])
        result = {k: self.label[result]}
        results.append(result)
    return results

# Perform forward inference on the input data to obtain the inference result.
def _inference(self, data):
    result = {}
    for k, v in data.items():
        result[k] = self.model(v)

    return result

# Define a network.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

def Mnist(model_path, **kwargs):
    # Generate a network.
    model = Net()

    # Load the model.
    if torch.cuda.is_available():
        device = torch.device('cuda')
        model.load_state_dict(torch.load(model_path, map_location="cuda:0"))
    else:
        device = torch.device('cpu')
        model.load_state_dict(torch.load(model_path, map_location=device))

    # CPU or GPU mapping
    model.to(device)

    # Turn the model to inference mode.
```

```
model.eval()

return model
```

Infer the configuration file **config.json** on the local host. The content is as follows:

```
{
  "model_algorithm": "image_classification",
  "model_type": "PyTorch",
  "runtime": "pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64"
}
```

Step 3: Creating an OBS Bucket and Upload Files to OBS

Upload the data, code file, inference code file, and inference configuration file obtained in the previous step to an OBS bucket. When running a training job on ModelArts, read data and code files from the OBS bucket.

1. Log in to **OBS console** and create an OBS bucket and folder.

```
{OBS bucket}          # OBS bucket name, which is customizable, for example, test-modelarts-xx
- {OBS folder}        # OBS folder name, which is customizable, for example, pytorch
  - mnist-data        # OBS folder, which is used to store the training dataset. The folder name is customizable, for example, mnist-data.
  - mnist-code        # OBS folder, which is used to store training script train.py. The folder name is customizable, for example, mnist-code.
  - infer             # OBS folder, which is used to store inference script customize_service.py and configuration file config.json
  - mnist-output      # OBS folder, which is used to store trained models. The folder name is customizable, for example, mnist-output.
```

CAUTION

- The region where the created OBS bucket resides must be the same as that where ModelArts is used. Otherwise, the OBS bucket will be unavailable for training. For details, see [Check whether the OBS bucket and ModelArts are in the same region](#).
- When creating an OBS bucket, do not set the archive storage class. Otherwise, training models will fail.

2. Upload the MNIST dataset package obtained in [Step 1 Prepare Training Data](#) to the **mnist-data** folder on OBS.

CAUTION

- When uploading data to OBS, do not encrypt the data. Otherwise, the training will fail.
- Files do not need to be decompressed. Directly upload compressed packages to OBS.

3. Upload the training script **train.py** to the **mnist-code** folder.
4. Upload the inference script **customize_service.py** and inference configuration file **config.json** to the **infer** folder in **mnist-code**.

Step 4 Create a Training Job

1. Log in to the [ModelArts console](#) and select the same region as the OBS bucket.
2. In the navigation pane on the left, choose **Permission Management** and check whether access authorization has been configured for the current account. For details, see [Configuring Agency Authorization for ModelArts with One Click](#). If you have been authorized using access keys, clear the authorization and configure agency authorization.
3. In the navigation pane, choose **Model Training** > **Training Jobs**. On the **Training Jobs** page, click **Create Training Job**.
4. Set parameters.
 - **Algorithm Type**: Select **Custom algorithm**.
 - **Boot Mode**: Select **Preset image** and then select **PyTorch** and **pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64** from the drop-down lists.
 - **Code Directory**: Select the created OBS code directory, for example, **/test-modelarts-xx/pytorch/mnist-code/** (replace **test-modelarts-xx** with your OBS bucket name).
 - **Boot File**: Select the training script **train.py** uploaded to the code directory.
 - **Input**: Add one input and set its name to **data_url**. Set the data path to your OBS directory, for example, **/test-modelarts-xx/pytorch/mnist-data/** (replace **test-modelarts-xx** with your OBS bucket name).
 - **Output**: Add one output and set its name to **train_url**. Set the data path to your OBS directory, for example, **/test-modelarts-xx/pytorch/mnist-output/** (replace **test-modelarts-xx** with your OBS bucket name). Do not pre-download to a local directory.
 - **Resource Type**: Select the specifications of a single GPU.
 - Retain default settings for other parameters.

NOTE

The sample code runs on a single node with a single PU. If you select a flavor with multiple PUs, the training will fail.

5. Click **Submit**, confirm parameter settings for the training job, and click **Yes**. The system automatically switches back to the **Training Jobs** page. When the training job status changes to **Completed**, the model training is completed.


NOTE

In this case, the training job will take about 10 minutes.


6. Click the training job name. On the job details page that is displayed, check whether there are error messages in logs. If so, the training failed. Identify the cause and locate the fault based on the logs.
7. In the lower left corner of the training details page, click the training output path to go to OBS, as shown in [Figure 2-4](#). Then, check whether the **model** folder is available and whether there are any trained models in the folder. If there is no **model** folder or trained model, the training input may be incomplete. In this case, completely upload the training data and train the model again.

Figure 2-4 Output path

Input

Input Path	Param...	Obtain...	Local Path (Tr...
 -modelarts-x...	data_url	Hyperp...	/home/ma...

Output

Output Path	Param...	Obtain...	Local Path (Tr...
 -modelarts-x...	train_url	Hyperp...	/home/ma...

Step 5 Deploying the Model for Inference

After the model training is completed, create a model and deploy the model as a real-time service.

1. Log in to the [ModelArts console](#). In the navigation pane on the left, choose **Model Management**. On the displayed page, click **Create Model**.
2. On the **Create Model** page, configure the parameters and click **Create now**.
Choose **Training Job** for **Meta Model Source**. Select the training job completed in [Step 4 Create a Training Job](#) from the drop-down list and enable **Dynamic loading**. The values of **AI Engine** will be automatically configured.

Figure 2-5 Meta Model Source

★ Meta Model Source

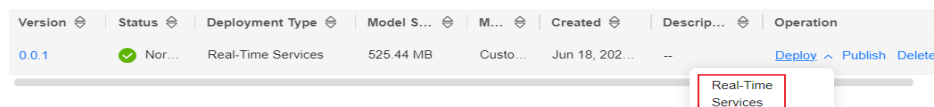
[Training job](#) [OBS](#) [Container image](#) [Template](#)

Import a model trained by a ModelArts training job. Select a job. For details of the model specification, see [Setting a Training Job as the Meta Model Source](#).
If the meta model is from a custom image, ensure the size of the meta model complies with [Restrictions on the Image Size for Importing an AI Application](#).

★ Training Job

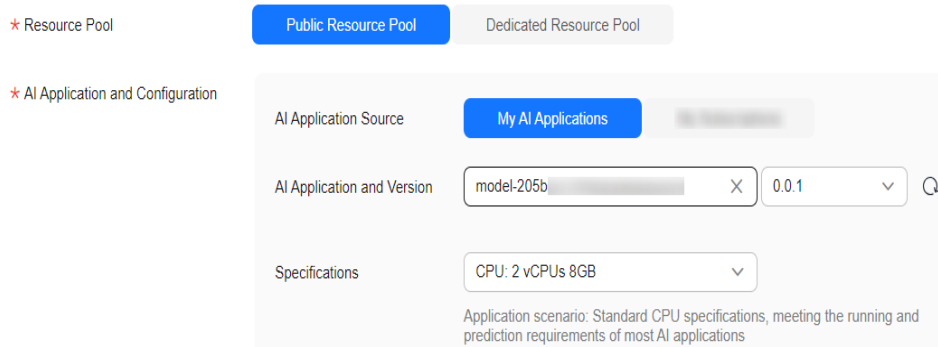
☒ Dynamic loading [?](#)

3. Wait until the model status changes to **Normal**. Then, the model is created. Locate the target model and click **Deploy** in the **Operation** column. On the displayed **Versions** page, locate the version and choose **Deploy** > **Real-Time Services** in the **Operation** column.

Figure 2-6 Deploying a real-time service

Version	Status	Deployment Type	Model S...	M...	Created	Descrip...	Operation
0.0.1	✓	Nor...	Real-Time Services	525.44 MB	Custo...	Jun 18, 202...	Deploy ^ Publish Delete

4. On the **Deploy** page, configure parameters and create a real-time service as prompted. In this example, use CPU specifications. If there are free CPU specifications, you can select them for deployment. (Each user can deploy only one real-time service for free. If you have deployed one, delete it first before deploying a new one for free.)

Figure 2-7 Deploying a model

★ Resource Pool

Public Resource Pool Dedicated Resource Pool

★ AI Application and Configuration

AI Application Source My AI Applications

AI Application and Version model-205b 0.0.1

Specifications CPU: 2 vCPUs 8GB

Application scenario: Standard CPU specifications, meeting the running and prediction requirements of most AI applications

After you submit the service deployment request, the system automatically switches to the **Real-Time Services** page. When the service status changes to **Running**, the service has been deployed.

Step 6 Performing Prediction

1. On the **Real-Time Services** page, click the name of the real-time service. The real-time service details page is displayed.
2. Click the **Prediction** tab, set **Request Type** to **multipart/form-data**, **Request Parameter** to **image**, click **Upload** to upload a sample image, and click **Predict**.

After the prediction is complete, the prediction result is displayed in the **Test Result** pane. According to the prediction result, the digit on the image is 2.

NOTE

The MNIST used in this case is a simple dataset used for demonstration, and its algorithms are also simple neural network algorithms used for teaching. The models generated using such data and algorithms are applicable only to teaching but not to complex prediction scenarios. The prediction is accurate only if the image used for prediction is similar to the image in the training dataset (white characters on black background).

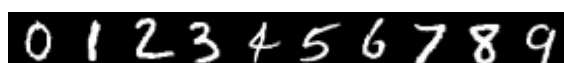
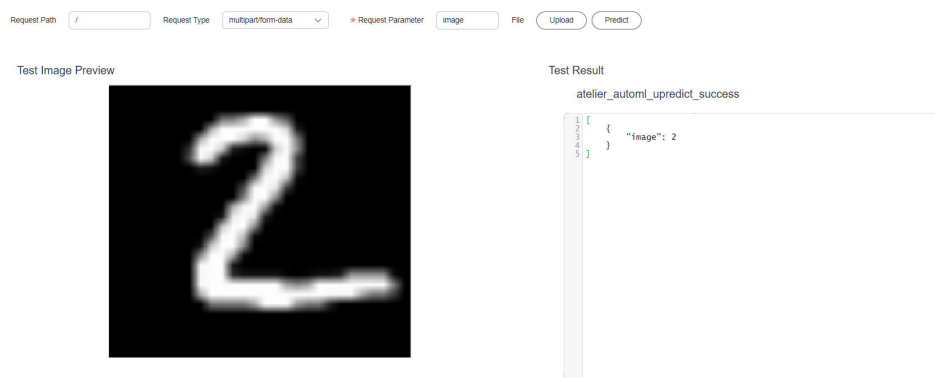
Figure 2-8 Example

Figure 2-9 Prediction results

Step 7 Releasing Resources

If you do not need to use this model and real-time service anymore, release the resources to stop billing.

- On the **Real-Time Services** page, locate the target service and click **Stop** or **Delete** in the **Operation** column.
- In the **My Model** tab, locate the target model and click **Delete** in the **Operation** column.
- On the **Training Jobs** page, delete the completed training job.
- Go to OBS and delete the OBS bucket, folders, and files used in this example.

FAQs

- **Why Is a Training Job Always Queuing?**
If the training job is always queuing, the selected resources are limited in the resource pool, and the job needs to be queued. In this case, wait for resources. For details, see [Why Is a Training Job Always Queuing](#).
- **Why Can't I Find My Created OBS Bucket After I Select an OBS Path in ModelArts?**
Ensure that the created bucket is in the same region as ModelArts. For details, see [Incorrect OBS Path on ModelArts](#).

3 Practices for Beginners

This section lists some common practices to help you understand and use ModelArts for AI development.

Table 3-1 Common best practices

Practice		Description
Assigning permissions for using ModelArts	Assigning Basic Permissions for Using ModelArts	Assign specific ModelArts operation permissions to the IAM users under a Huawei Cloud account. This prevents exceptions from occurring due to permissions when the IAM users access ModelArts.
Training a model	Example: Creating a Custom Image for Training (PyTorch + CPU/GPU)	This section describes how to create an image and use it for training on ModelArts. The AI engine used in the image is PyTorch, and the training runs on CPUs or GPUs.

Practice		Description
Deploying a service for inference	Creating a Custom Image and Using It to Create an AI Application	If you want to use an AI engine that is not supported by ModelArts, create a custom image, import the image to ModelArts, and use the image to create AI applications. This section describes how to use a custom image to create an AI application and deploy the application as a real-time service.