ModelArts

Getting Started

Issue 01

Date 2023-07-06

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

Notice

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

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

Huawei Technologies Co., Ltd.

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

Website: https://www.huawei.com

Email: support@huawei.com

Contents

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 a service developer and have no AI development experience, you can use ExeML of ModelArts to build AI models. For details, see **Service Developers: Building Models Using ExeML**.

2 Service Developers: Building Models Using ExeML

ModelArts provides ExeML for service developers, freeing you from model development and parameter tuning. With ExeML, you can finish an AI development project in just three steps, including data labeling, auto training, and service deployment.

As an example of object detection, this section describes how to detect Yunbao, the mascot of HUAWEI CLOUD, to help you quickly get started with ModelArts. By using the built-in Yunbao image dataset, the system automatically trains and generates a detection model, and deploys the generated model as a real-time service. After the deployment is completed, you can use the real-time service to identify whether an input image contains Yunbao.

Before you start, carefully complete the preparations described in **Preparations**. To use ExeML to build a model, perform the following steps:

- **Step 1: Prepare Data**
- **Step 2: Create an Object Detection Project**
- **Step 3: Label Data**
- **Step 4: Generate a Model with ExeML**
- **Step 5: Deploy the Model as a Real-Time Service**
- **Step 6: Test the Service**

Preparations

- Your HUAWEI CLOUD account is not in arrears or frozen.
- Access authorization has been configured for your account. For details, see **Configuring Agency Authorization**. If you have been authorized using access keys, clear the authorization and configure agency authorization.
- OBS buckets and folders are ready for model data storage. For details about how to create OBS buckets and folders, see **Creating a Bucket** and **Creating a Folder**. For normal data access, ensure that the created OBS bucket and ModelArts are in the same region.
- Data management is required for creating ExeML projects. Therefore, obtain the permission to access OBS from the **Data Management** module before using ExeML.

On the ModelArts management console, choose **Data Management** > **Datasets** in the left navigation pane. On the page that is displayed, click **Service Authorization** to apply for permission authorization. If you log in using an account, a dialog box is displayed, asking you to accept the authorization. If you log in as an IAM user (member account), your master account or a user with admin permissions grants authorization to you.

Step 1: Prepare Data

ModelArts provides a sample dataset of Yunbao named **Yunbao-Data-Custom**. This example uses this dataset to build a model. Perform the following operations to upload the dataset to the OBS directory **test-modelarts/dataset-yunbao** created in preparation. The OBS bucket name **test-modelarts** is for reference only. You need to customize an OBS bucket name.

If you want to use your own dataset, skip this step, upload the dataset to the OBS folder, and select this directory in **Step 2: Create an Object Detection Project**.

1. **Download the Yunbao-Data-Custom dataset** to the local PC.
2. Decompress the **Yunbao-Data-Custom.zip** file to the **Yunbao-Data-Custom** directory on the local PC.
3. Batch upload all files from the **Yunbao-Data-Custom** folder to the **test-modelarts/dataset-yunbao** directory on OBS. For details, see **Uploading a File**.

   The obtained dataset has two directories: **eval** and **train**. The data stored in **train** is used for model training, and the data stored in **eval** is used for model prediction.

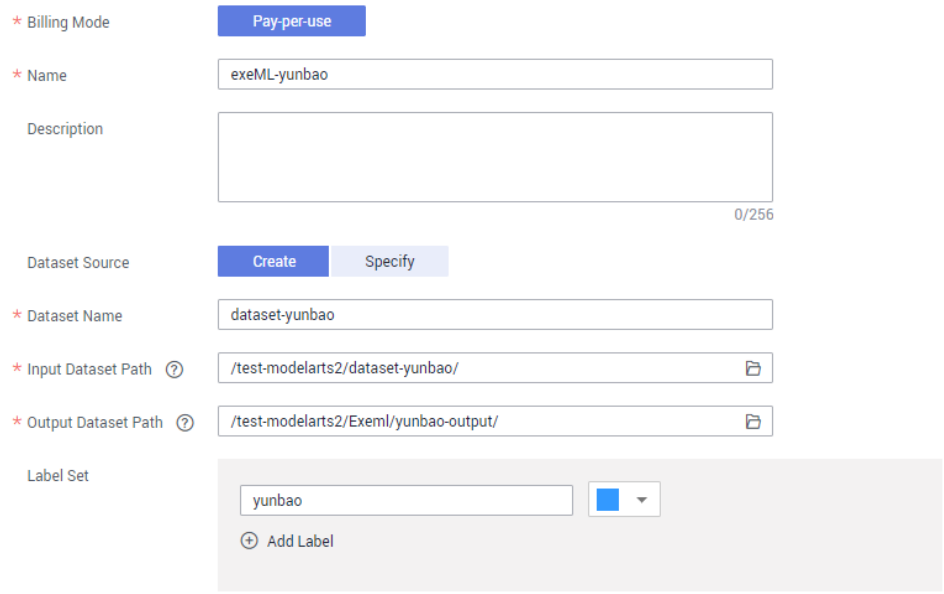
Step 2: Create an Object Detection Project

1. On the ModelArts management console, click **ExeML** in the left navigation pane.
2. In the **Object Detection** box, click **Create Project**. On the **Create Object Detection Project** page that is displayed, enter a project name and a dataset name, and select an input dataset path. The OBS path of the Yunbao dataset is **/test-modelarts/dataset-yunbao/train/**. Select an empty directory in **Output Dataset Path**.

   NOTE

   The Yunbao dataset has two directories: **eval** and **train**. Select the data in the **train** directory for training. If the upper-layer directory of **train** is selected, an error message is displayed, indicating that OBS has invalid data. As a result, the project will fail to be created.

Figure 2-1 Creating an object detection project



3. Click **Create Project**. The object detection project is created. After the project is created, the **ExeML > Label Data** page is displayed and data source synchronization is automatically performed.

Step 3: Label Data

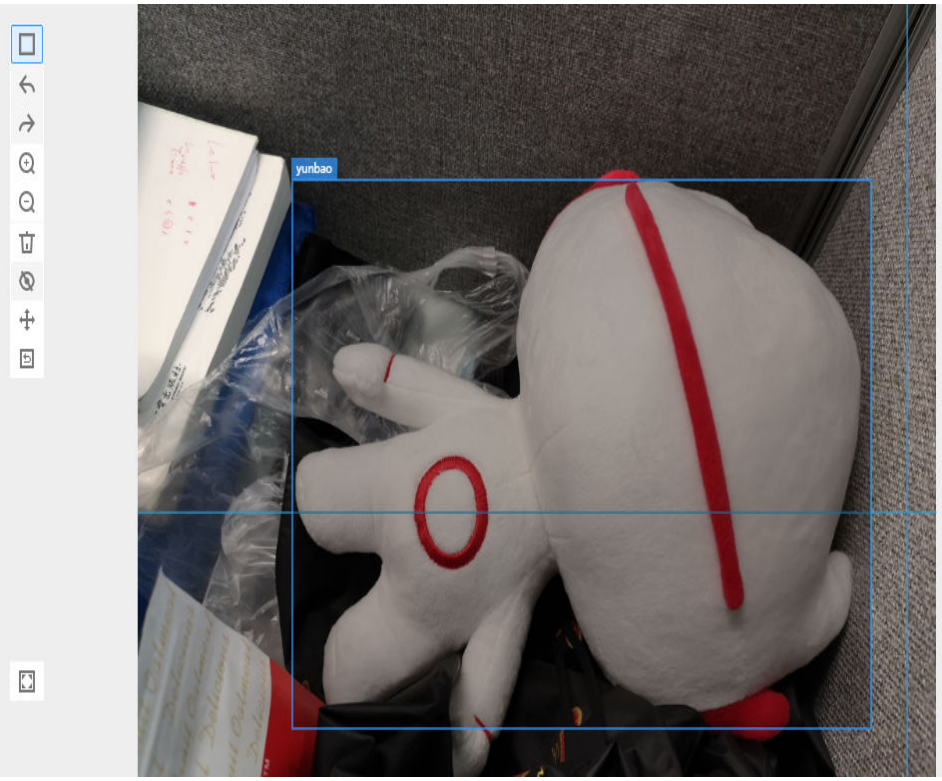
For an object detection project, labeling data is to locate an object in an image and assign a label to the object. The labeled data is used for model training. In the Yunbao dataset, part of data has been labeled. You can label the unlabeled data for trial use.

Data source synchronization is automatically performed when you create an ExeML project. Data source synchronization takes a certain period of time. If the synchronization fails, you can click **Synchronize Data Source** to manually execute the synchronization.

1. On the **ExeML > Label Data** page, click the **Unlabeled** tab. All unlabeled images are displayed. Click an image to go to the labeling page.
2. Left-click and drag the mouse to select the area where Yunbao is located. In the dialog box that is displayed, enter the label name, for example, **yunbao**, and press **Enter**. After the labeling is completed, the status of the image changes to **Labeled** in the left **Image Catalog** pane.

   You can select another image from the image catalog in the lower part of the page and repeat the preceding steps to label the image. If an image contains more than one Yunbao, you can label all. You are advised to label all images in the dataset to train a model with better precision.

Figure 2-2 Image labeling for object detection

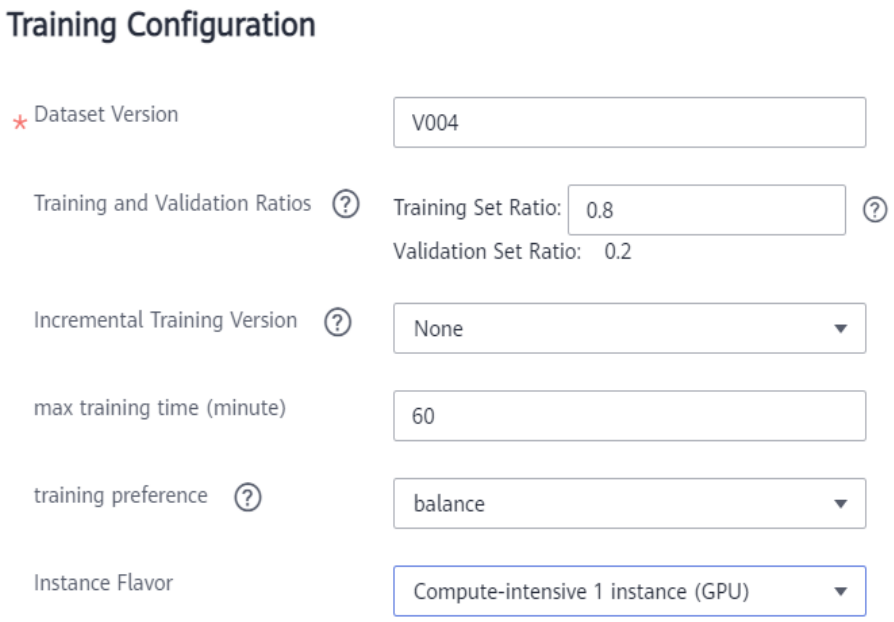


3. After all images in the image directory are labeled, click the project name in the upper left corner. In the dialog box that is displayed, click **OK** to save the labeling information. On the **Labeled** tab page, you can view the labeled images and view the label names and quantity in the right pane.

Step 4: Generate a Model with ExeML

1. After data labeling is completed, click **Train** in the upper right corner of the data labeling page. In the **Training Configuration** dialog box that is displayed, set related parameters. For details, see **Figure 2-3**.

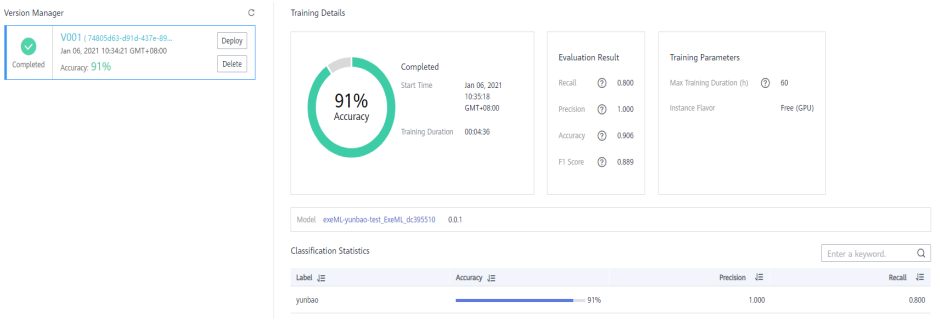
Figure 2-3 Setting training parameters



2. Click **Next**. On the configuration page that is displayed, confirm the specifications and click **Submit** to start auto model training. The training takes a certain period of time. If you close or exit the page, the system continues training until it is completed.

   After the training is completed, you can view the training details on the page, such as the accuracy, evaluation result, training parameters, and classification statistics.

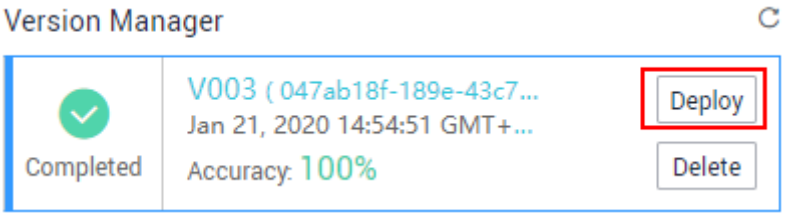
Figure 2-4 Model training



Step 5: Deploy the Model as a Real-Time Service

1. On the **Train Model** tab page, wait until the training status changes to **Completed**. Click **Deploy** in the **Version Manager** pane.

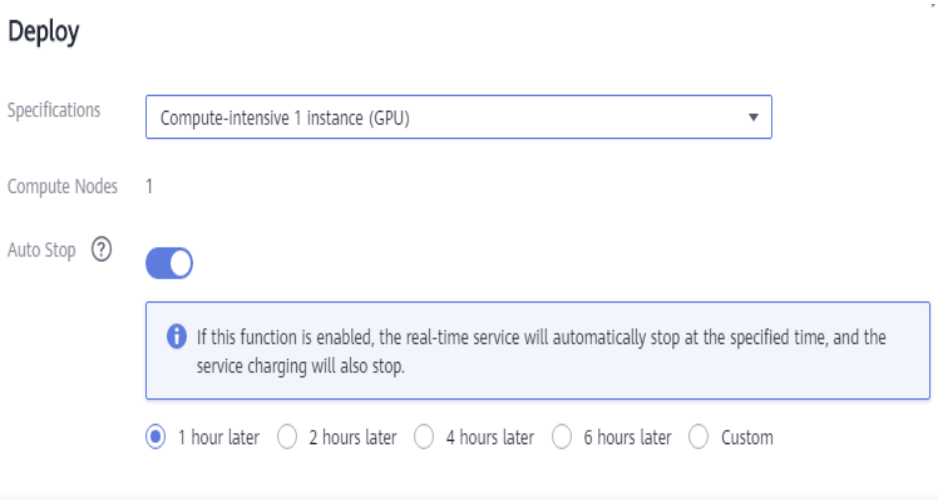
Figure 2-5 Deploying a service



2. In the displayed **Deploy** dialog box, set **Specifications** and **Auto Stop**, and click **OK** to deploy the object detection model as a real-time service.

   If you select free specifications, you do not need to set **Auto Stop**, because the node will be stopped one hour later.

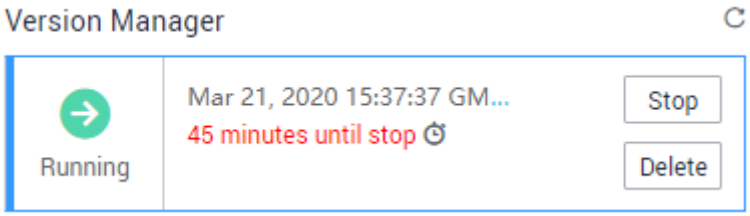
Figure 2-6 Deployment settings



3. After the deployment is started, the system automatically switches to the **Deploy Service** tab page. This page displays the deployment progress and status.

   The deployment takes a certain period of time. After the deployment is completed, the status in the **Version Manager** pane changes to **Running**.

Figure 2-7 Successful deployment

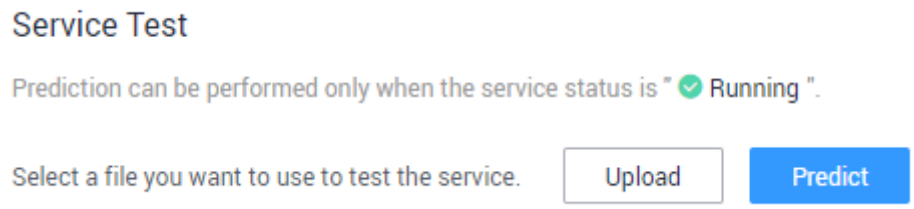


Step 6: Test the Service

After the model is deployed, you can test the service using an image.

1. On the **Deployment Online** tab page, select a running service version, and click **Upload** to upload a local image.

Figure 2-8 Uploading an image



2. Select an image from a local environment. The image must contain Yunbao. Click **Predict** to perform the test.

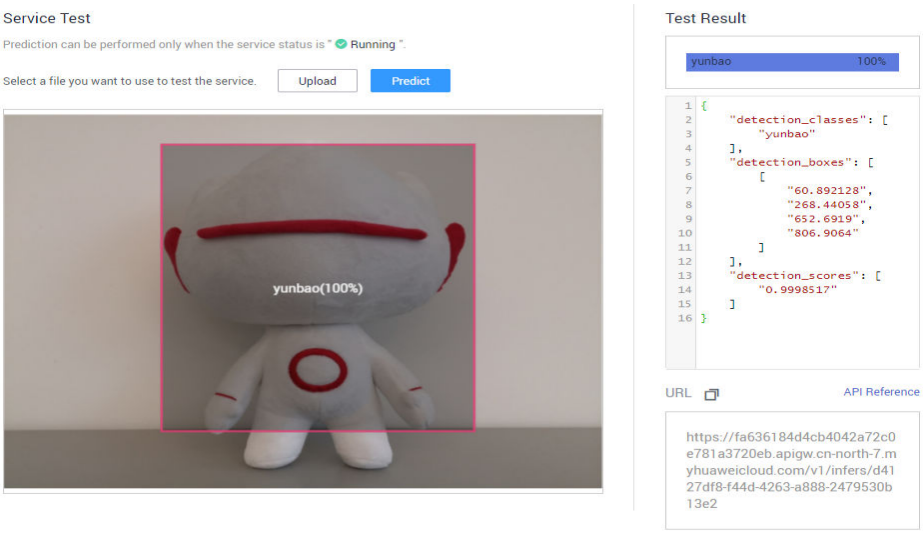
   After the prediction is completed, the label name **yunbao**, location coordinates, and confidence score are displayed in the prediction result pane on the right. In the prediction result, **detection_boxes** indicates the location of the object, **detection_scores** indicates the detection score of **yunbao**.

   If the model accuracy does not meet your expectation, add images on the **Label Data** tab page, label the images, and train and deploy the model again.

   NOTE

   A running real-time service keeps consuming the resources. If you do not need to use the real-time service, click **Stop** in the **Version Manager** pane to stop the service and avoid unnecessary billing. If you want to use the service again, click **Start**.

Figure 2-9 Test result



3 Using a Custom Algorithm to Build a Handwritten Digit Recognition Model

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. For details, see **Preparations**.

1. **Step 1 Prepare the Training Data**: Download the MNIST dataset.
2. **Step 2 Prepare Training Files and Inference Files**: Write training and inference code.
3. **Step 3 Create 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 Deploy the Model for Inference**: Import the trained model to ModelArts, create an AI application, and deploy the AI application as a real-time service.
6. **Step 6 Perform Prediction**: Upload a handwritten digit image and send an inference request to obtain the inference result.
7. **Step 7 Release 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.

- You have configured the agency-based authorization.

  Certain ModelArts functions require access to OBS, SWR, and IEF. Before using ModelArts, ensure your account has been authorized to access these services.

  a. Log in to the **ModelArts console** using your Huawei Cloud account. In the navigation pane on the left, choose **Settings**. Then, on the **Global Configuration** page, click **Add Authorization**.

  b. On the **Add Authorization** page that is displayed, set required parameters as follows:

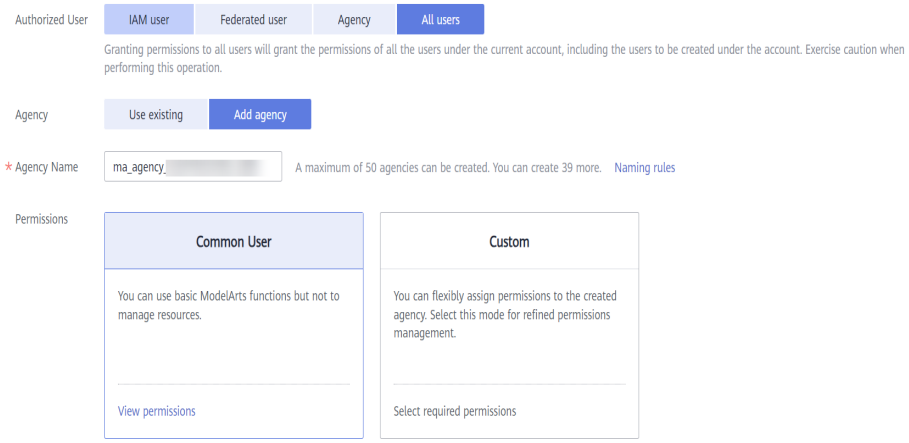
  Authorized User: Select **All users**.

  Agency: Select **Add agency**.

  Permissions: Select **Common User**.

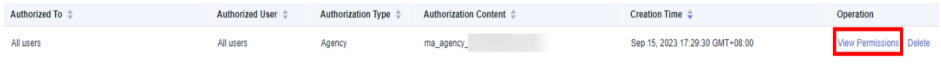
  Select "I have read and agree to the ModelArts Service Statement", and click **Create**.

  Figure 3-1 Configuring the agency-based authorization



  c. After the configuration is complete, view the agency configurations of your account on the **Global Configuration** page.

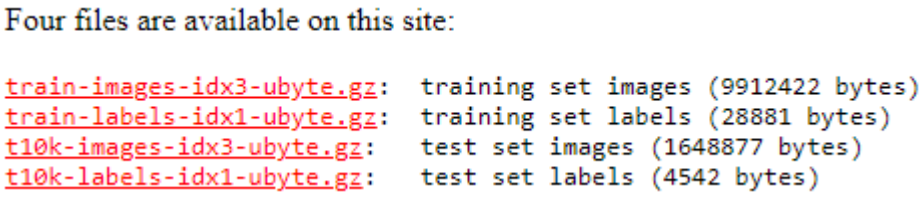
  Figure 3-2 Viewing agency configurations



Step 1 Prepare the Training Data

An MNIST dataset downloaded from the **MNIST official website** is used in this case. Ensure that the four files are all downloaded.

Figure 3-3 MNIST dataset



- **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 contain 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 contain the labels of the 10,000 samples

NOTE

If you are asked to enter the login information after you click the MNIST official website link, copy and paste this link in the address box of your browser: http://yann.lecun.com/exdb/mnist/

The login information is required when you open the link in HTTPS mode, which is not required if you open the link in HTTP mode.

Step 2 Prepare 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 a 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 Create an OBS Bucket and Upload Files to OBS

Upload the data, code file, inference code file, and inference configuration file obtained from **Step 2** to an OBS bucket. When running a training job on ModelArts, read data and code files from the OBS bucket.

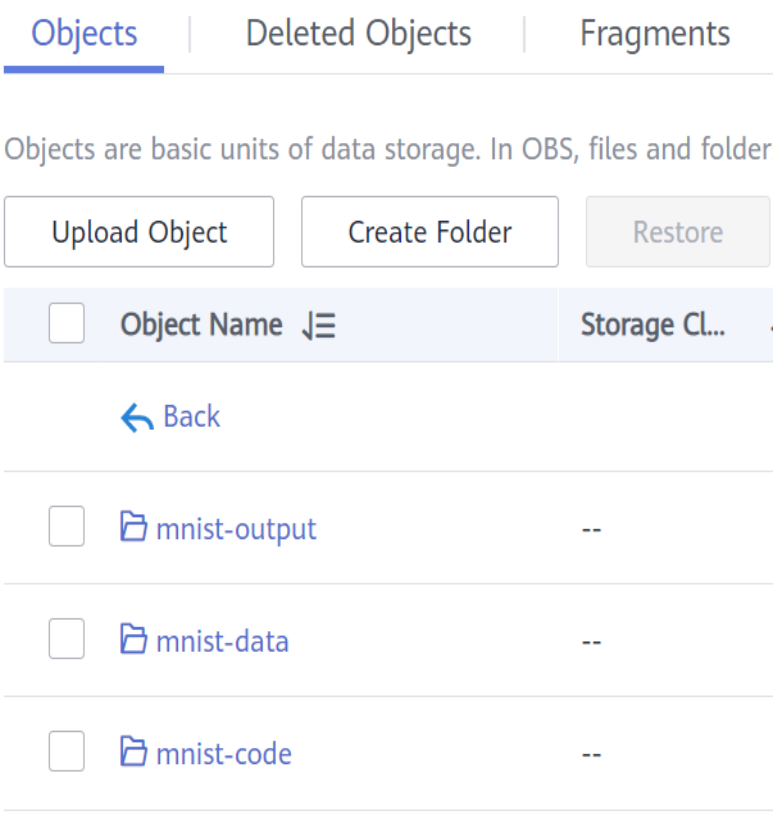
1. Log in to the OBS console and create an OBS bucket and folder. **Figure 3-4** shows an example of the created objects. For details, see **Creating a Bucket** and **Creating a Folder**.

   ```
   {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.

Figure 3-4 OBS file directory

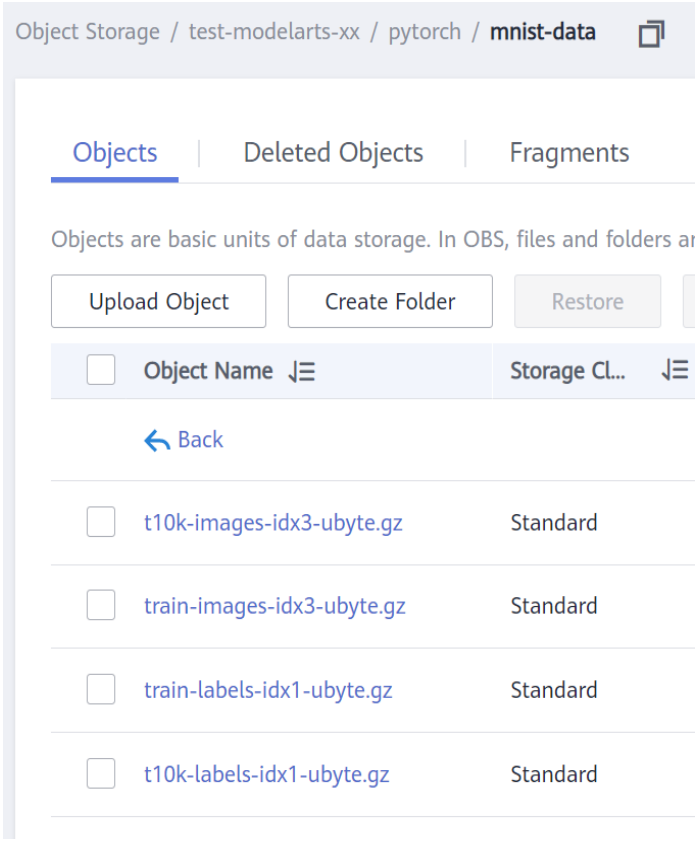


2. Upload the MNIST dataset package obtained in **Step 1 Prepare the Training Data** to OBS. For details, see **Uploading a File**.

CAUTION

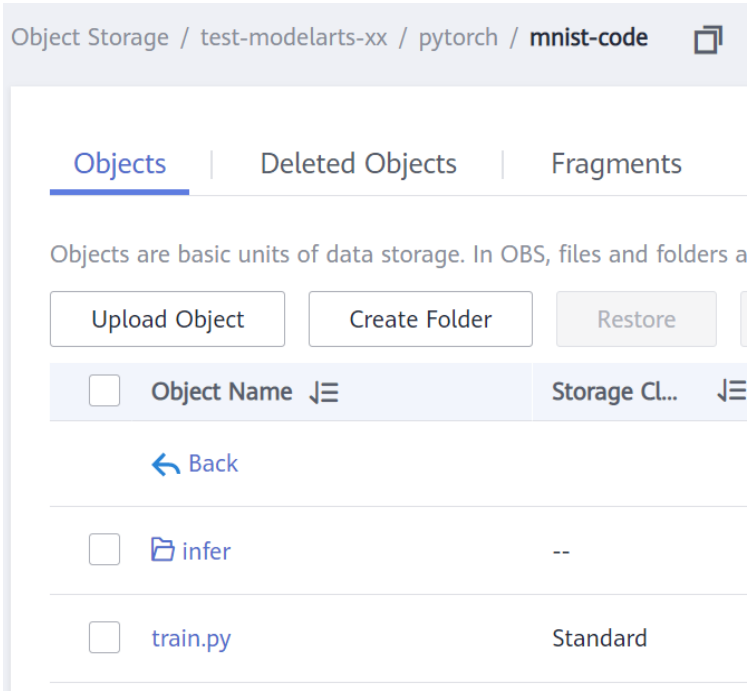
- When uploading data to OBS, do not encrypt the data. Otherwise, the training will fail.
- Files do not need to be decompressed. Directly upload compressed packages to OBS.

Figure 3-5 Uploading a dataset to the mnist-data folder



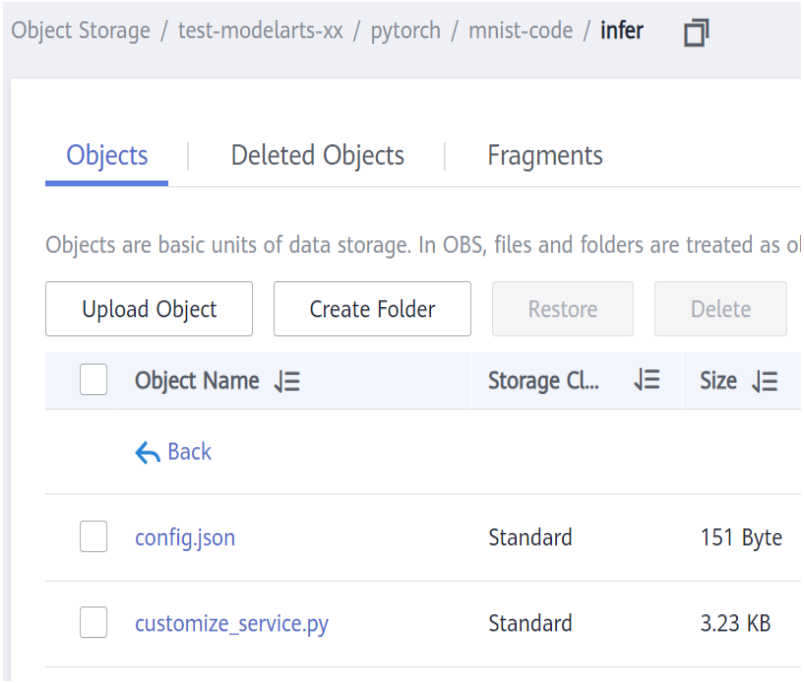
3. Upload the training script **train.py** to the **mnist-code** folder.

Figure 3-6 Uploading 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.

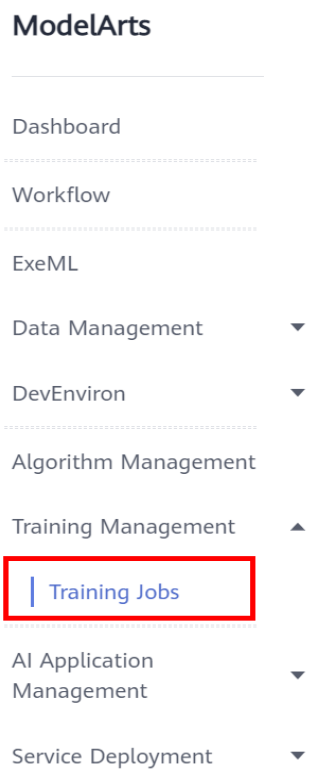
   Figure 3-7 Uploading **customize_service.py** and **config.json** to the **infer** folder



Step 4 Create a Training Job

1. Log in to the ModelArts management console and select the same region as the OBS bucket.
2. In the navigation pane on the left, choose **Settings** and check whether access authorization has been configured for the current account. For details, see **Configuring Access Authorization**. If you have been authorized using access keys, clear the authorization and configure agency authorization.
3. In the navigation pane on the left, choose **Training Management** > **Training Jobs**. On the displayed page, click **Create Training Job**.

Figure 3-8 Training Jobs



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 **GPU** and then **GPU: 1*NVIDIA-V100(16GB) | CPU: 8 vCPUs 64GB** (example). If there are free GPU specifications, you can select them for training.
   - Retain default settings for other parameters.

     NOTE

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

Figure 3-9 Training job settings

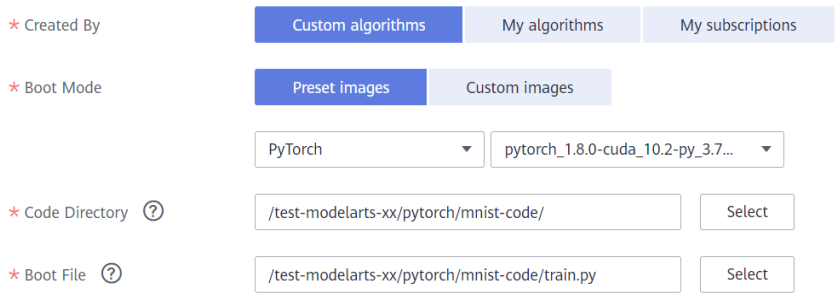


Figure 3-10 Setting training input and output

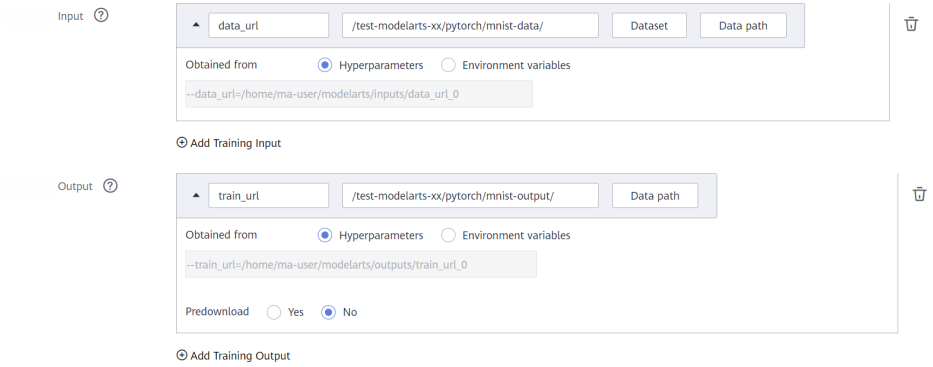
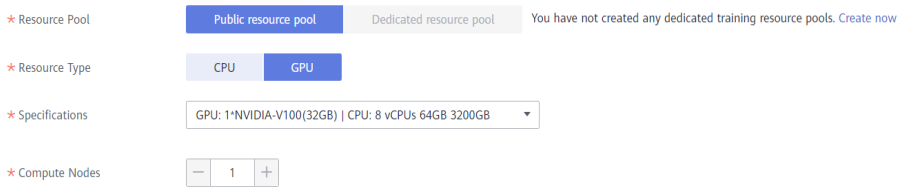


Figure 3-11 Configuring the resource type



5. Click **Submit**, confirm parameter settings for the training job, and click **Yes**.
6. 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 more than 10 minutes.
7. 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.
8. In the lower left corner of the training details page, click the training output path to go to OBS (as shown in **Figure 3-12**). Then, check whether the **model** folder is available and whether there are any trained models in the folder (as shown in **Figure 3-13**). 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 3-12 Output path

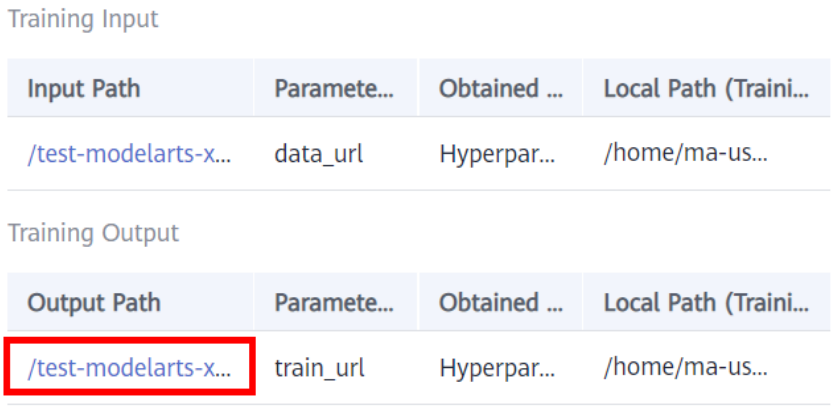
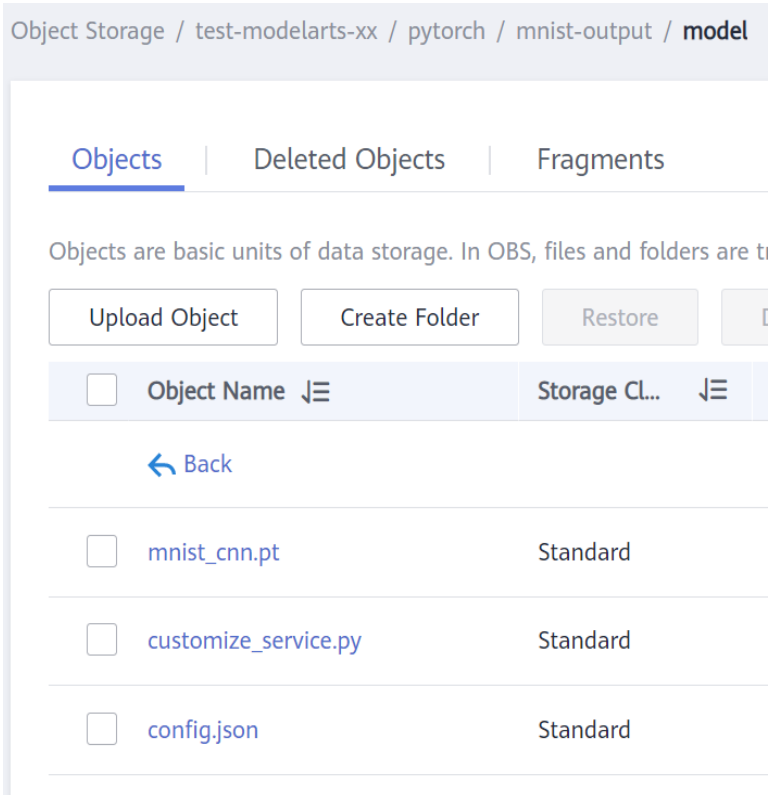


Figure 3-13 Trained model



Step 5 Deploy the Model for Inference

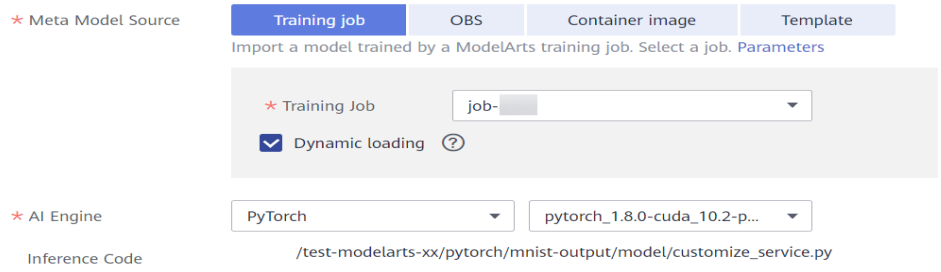
After the model training is complete, create an AI application and deploy it as a real-time service.

1. Log in to the ModelArts management console. In the navigation pane on the left, choose **AI Application Management** > **AI Applications**. On the **My AI Applications** page, click **Create**.
2. On the **Create** page, configure 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

select **Dynamic loading**. The values of **AI Engine** will be automatically configured.

Figure 3-14 Meta Model Source

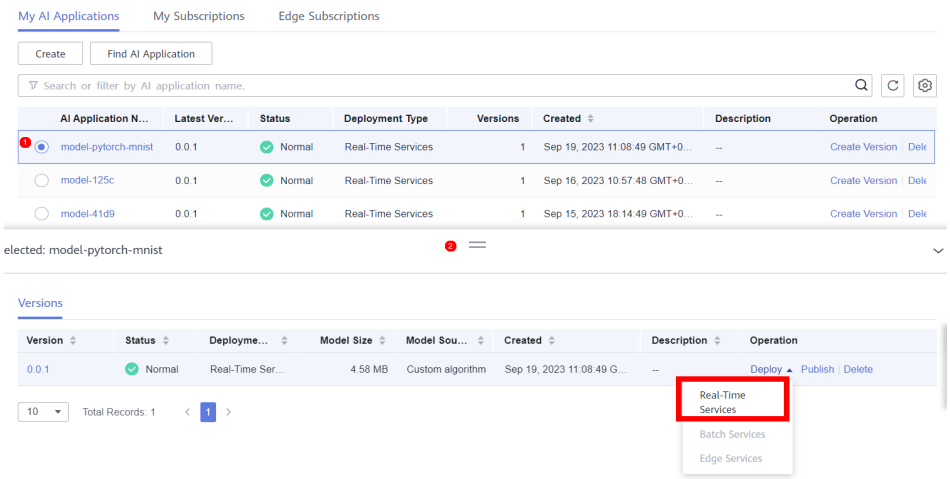


NOTE

If you have used **Training Jobs** of an old version, you can see both **Training Jobs** and **Training Jobs New** below **Training job**. In this case, select **Training Jobs New**.

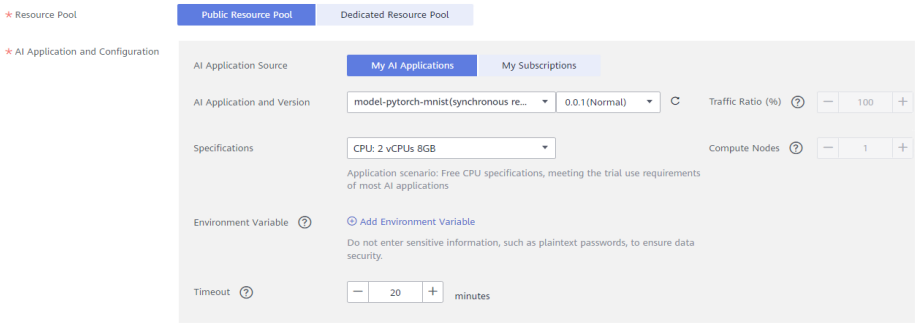
3. On the **AI Applications** page, if the application status changes to **Normal**, it has been created. Click the option button on the left of the AI application name to display the version list at the bottom of the list page, and choose **Deploy** > **Real-Time Services** in the **Operation** column to deploy the AI application as a real-time service.

Figure 3-15 Deploying a real-time service



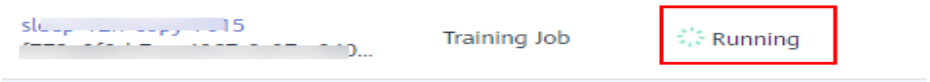
4. On the **Deploy** page, configure parameters and create a real-time service as prompted. In this example, use CPU specifications. If there are free CPU specifications, you can select them for deployment. (Each user can deploy only one real-time service for free. If you have deployed one, delete it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

Figure 3-16 Deploying a model



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.

Figure 3-17 Deployed service



Step 6 Perform 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**.

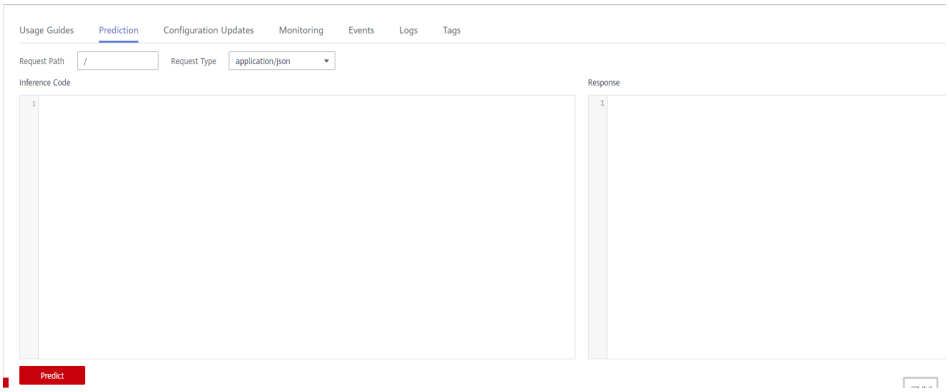
   After the prediction is complete, the prediction result is displayed in the **Test Result** pane. According to the prediction result, the digit on the image is **2**.

   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 3-18 Example

Figure 3-19 Prediction results



Step 7 Release 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 row containing the target service and click **Stop** or **Delete** in the **Operation** column.
- On the **AI Applications** page in **AI Application Management**, locate the row containing the target service and click **Delete** in the **Operation** column.
- On the **Training Jobs** page, click **Delete** in the **Operation** column to delete the finished 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**.

4 Practices for Beginners

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

Table 4-1 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 (Horovod-PyTorch and GPUs)**	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.

