

ModelArts

Inference

Issue 01
Date 2023-12-05



Copyright © Huawei Technologies Co., Ltd. 2023. 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 Technologies Co., Ltd.

Trademarks and Permissions



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

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

Notice

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

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

Huawei Technologies Co., Ltd.

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

Website: <https://www.huawei.com>

Email: support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to "Vul. Response Process". For details about the policy, see the following website:<https://www.huawei.com/en/psirt/vul-response-process>
For enterprise customers who need to obtain vulnerability information, visit:<https://securitybulletin.huawei.com/enterprise/en/security-advisory>

Contents

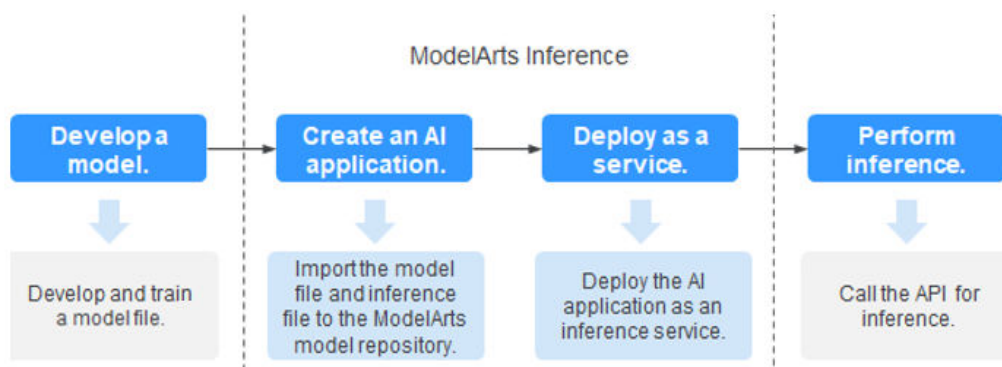
1 Introduction to Inference.....	1
2 Managing AI Applications.....	3
2.1 Introduction to AI Application Management.....	3
2.2 Creating an AI Application.....	5
2.2.1 Importing a Meta Model from a Training Job.....	5
2.2.2 Importing a Meta Model from OBS.....	7
2.2.3 Importing a Meta Model from a Container Image.....	9
2.3 Managing AI Applications.....	12
3 Deploying AI Applications as Real-Time Services.....	13
3.1 Deploying as a Real-Time Service.....	13
3.2 Viewing Service Details.....	15
3.3 Testing the Deployed Service.....	20
3.4 Upgrading a Real-Time Service.....	22
3.5 Accessing Real-Time Services.....	25
3.5.1 Access Authenticated Using a Token.....	26
3.5.2 Access Authenticated Using an AK/SK.....	29
4 Deploying AI Applications as Batch Services.....	34
4.1 Deploying as a Batch Service.....	34
4.2 Viewing the Batch Service Prediction Result.....	38
5 Upgrading a Service.....	40
6 Starting or Stopping a Service.....	42
7 Deleting a Service.....	43
8 Monitoring.....	44
8.1 ModelArts Metrics.....	44
8.2 Setting Alarm Rules.....	46
8.3 Viewing Monitoring Metrics.....	46
9 Inference Specifications.....	48
9.1 Model Package Specifications.....	48
9.1.1 Introduction to Model Package Specifications.....	48
9.1.2 Specifications for Editing a Model Configuration File.....	51

9.1.3 Specifications for Writing Model Inference Code.....	65
9.2 Examples of Custom Scripts.....	72
9.2.1 TensorFlow.....	72
9.2.2 PyTorch.....	78
9.2.3 Caffe.....	81
9.2.4 XGBoost.....	87
9.2.5 PySpark.....	88
9.2.6 Scikit Learn.....	89

1 Introduction to Inference

After an AI model is developed, you can use it to create an AI application and quickly deploy the application as an inference service. The AI inference capabilities can be integrated into your IT platform by calling APIs.

Figure 1-1 Inference



- **Develop a model:** Models can be developed in ModelArts or your local development environment. A locally developed model must be uploaded to OBS.
- **Create an AI application:** Import the model file and inference file to the ModelArts model repository and manage them by version. Use these files to build an executable AI application.
- **Deploy as a service:** Deploy the AI application as a container instance in the resource pool and register inference APIs that can be accessed externally.
- **Perform inference:** Add the function of calling the inference APIs to your application to integrate AI inference into the service process.

Deploying an AI Application as a Service

After an AI application is created, you can deploy it as a service on the **Deploy** page. ModelArts supports the following deployment types:

- **Real-time service**
Deploy an AI application as a web service with real-time test UI and monitoring supported.

- **Batch service**

Deploy an AI application as a batch service that performs inference on batch data and automatically stops after data processing is complete.

2 Managing AI Applications

2.1 Introduction to AI Application Management

AI development and optimization require frequent iterations and debugging. Changes in datasets, training code, or parameters affect the quality of models. If the metadata of the development process cannot be centrally managed, the optimal model may fail to be reproduced.

ModelArts AI application management allows you to import all meta models obtained through training, meta models uploaded to OBS, and meta models in container images. In this way, you can centrally manage all iterated and debugged AI applications.

Usage Restrictions

- In an ExeML project, after a model is deployed, the model is automatically uploaded to the AI application management list. However, AI applications generated by ExeML cannot be downloaded and can be used only for deployment and rollout.

Scenarios for Creating AI Applications

- **Importing a Meta Model from a Training Job:** You can create a training job on ModelArts and complete model training. After obtaining a satisfactory model, create an AI application for deployment.
- **Importing a Meta Model from OBS:** If you use a mainstream framework to develop and train a model locally, you can upload the model to an OBS bucket based on the model package specifications, import the model from OBS to ModelArts, and use it to create an AI application for service deployment.
- **Importing a Meta Model from a Container image:** If an AI engine is not supported by ModelArts, you can use it to build a model, import the model to ModelArts as a custom image, use the image to create an AI application, and deploy the AI application as services.

Functions of AI Application Management

Table 2-1 Functions of AI application management

Function	Description
Creating an AI Application	<p>Import the trained models to ModelArts and create AI applications for centralized management. The following provides the operation guide for each method of importing models.</p> <ul style="list-style-type: none"> • Importing a Meta Model from a Training Job • Importing a Meta Model from OBS • Importing a Meta Model from a Container Image
Managing AI Applications	<p>For model lineage and tuning, ModelArts provides AI application versioning.</p>

Supported AI Engines for ModelArts Inference

If you import a model from a template or OBS to create an AI application, the following AI engines and versions are supported.

 **NOTE**

- Runtime environments marked with **recommended** are unified runtime images, which will be used as mainstream base inference images.
- Images of the old version will be discontinued. Use unified images.
- A unified runtime image is named in the following format: *<AI engine and version> - <Hardware and version: CPU, CUDA, or CANN> - <Python version> - <OS version> - <CPU architecture>*

Table 2-2 Supported AI engines and their runtime

Engine	Runtime	Note
TensorFlow	tf1.13-python3.7-cpu tf1.13-python3.7-gpu tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64	The suffix contains cpu or gpu , indicating the model runs only on CPUs or GPUs.
Spark_MLlib	python2.7 python3.6	<ul style="list-style-type: none"> • Spark_MLlib 2.3.2 is used in python2.7 and python3.6. • python2.7 and python3.6 can only be used to run models on CPUs.

Engine	Runtime	Note
Scikit_Learn	python2.7 python3.6	<ul style="list-style-type: none">• Scikit_Learn 0.18.1 is used in python2.7 and python3.6.• python2.7 and python3.6 can only be used to run models on CPUs.
XGBoost	python2.7 python3.6	<ul style="list-style-type: none">• XGBoost 0.80 is used in python2.7 and python3.6.• python2.7 and python3.6 can only be used to run models on CPUs.
PyTorch	python3.7 pytorch_1.8.0- cuda_10.2-py_3.7- ubuntu_18.04-x86_64	<ul style="list-style-type: none">• PyTorch 1.0 is used in python3.7.• python3.7 indicates that the model can run on both CPUs and GPUs.• The default runtime is python3.6.

2.2 Creating an AI Application

2.2.1 Importing a Meta Model from a Training Job

You can create a training job on ModelArts and perform training to obtain a satisfactory model. Then import the model to **Model Management** for centralized management. In addition, you can quickly deploy the model as a service.

Background

- If a model generated by a ModelArts training job is used, ensure that the training job is completed and the model has been stored in the OBS directory for training outputs.
- A model generated from a training job that uses subscribed algorithms can be directly imported to ModelArts without the need to use the inference code or configuration file.
- If a model is generated from a training job that uses a mainstream framework or custom image, upload the inference code and configuration file to the storage directory of the model by referring to [Introduction to Model Package Specifications](#).
- The OBS directory you use and ModelArts are in the same region.
- ModelArts of the Arm version does not support model import from training.

Creating an AI Application

1. Log in to the ModelArts management console, and choose **AI Application Management > AI Applications** in the left navigation pane. The **AI Applications** page is displayed.

2. Click **Create** in the upper left corner.
3. On the displayed page, set the parameters.
 - a. Set basic information about the AI application. For details about the parameters, see [Table 2-3](#).

Table 2-3 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Version	Version of the AI application to be created. For the first import, the default value is 0.0.1 .
Description	Brief description of an AI application

- b. Select a meta model source and set parameters. If you select a training job, configure the parameters in [Table 2-4](#).

Table 2-4 Parameters of the meta model source

Parameter	Description
Meta Model Source	Select Training job . <ul style="list-style-type: none">• Select a training job that has completed training under the current account and a training version from the drop-down lists on the right of Training Job and Version respectively.
AI Engine	Inference engine used by the metamodel, which automatically match the training job you select
Runtime Dependency	Dependencies of the selected model on the environment.
AI Application Description	Provide AI application descriptions to help other AI application developers better understand and use your applications. Click Add AI Application Description and set the Document name and URL . You can add up to three AI application descriptions.
Deployment Type	Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select Real-time services here, you can only deploy the AI application as a real-time service after it is created.

- c. Check the information and click **Next**. The AI application is created. In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is

successfully created. On this page, you can perform such operations as creating new versions, quickly deploying AI applications, and publishing AI applications.

Follow-Up Procedure

Deploying the AI Applications as Services: In the AI application list, click the down arrow on the left of an AI application name to check all versions of the AI application. Locate the row that contains the target version, click **Deploy** in the **Operation** column, and select a deployment type from the drop-down list box. The AI application can be deployed in a deployment type selected during AI application creation.

2.2.2 Importing a Meta Model from OBS

If a model is developed and trained using a mainstream AI engine, import the model to ModelArts and use the model to create an AI application. In this way, the AI applications can be centrally managed on ModelArts.

Prerequisites

- The model has been developed and trained, and the type and version of the AI engine it uses is supported by ModelArts. For details, see [Supported AI Engines for ModelArts Inference](#).
- The imported model for creating an AI application, inference code, and configuration file must comply with the requirements of ModelArts. For details, see [Introduction to Model Package Specifications](#), [Specifications for Editing a Model Configuration File](#), and [Specifications for Writing Model Inference Code](#).
- The trained model package, inference code, and configuration file have been uploaded to OBS.
- The OBS directory you use and ModelArts are in the same region.
- ModelArts of the Arm version does not support model import from OBS.

Creating an AI Application

1. Log in to the ModelArts management console, and choose **AI Application Management > AI Applications** in the left navigation pane. The **AI Applications** page is displayed.
2. Click **Create** in the upper left corner.
3. On the displayed page, set the parameters.
 - a. Set basic information about the AI application. For details about the parameters, see [Table 2-5](#).

Table 2-5 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.

Parameter	Description
Version	Version of the AI application to be created. For the first import, the default value is 0.0.1 .
Description	Brief description of an AI application

- b. Select the meta model source and set related parameters. Set **Meta Model Source** to **OBS**. For details about the parameters, see [Table 2-6](#).

For the meta model imported from OBS, edit the inference code and configuration files by following [model package specifications](#) and place the inference code and configuration files in the **model** folder storing the meta model. If the selected directory does not comply with the model package specifications, the AI application cannot be created.

Table 2-6 Parameters of the meta model source

Parameter	Description
Meta Model	OBS path for storing the meta model. The OBS path cannot contain spaces. Otherwise, the AI application fails to be created.
AI Engine	The AI engine automatically associates with the meta model storage path you select.
Runtime Dependency	List the dependencies of the selected model on the environment.
AI Application Description	Provide AI application descriptions to help other AI application developers better understand and use your applications. Click Add AI Application Description and set the Document name and URL . You can add up to three AI application descriptions.
Configuration File	By default, the system associates the configuration file stored in OBS. After enabling this function, you can view and edit the model configuration file. NOTE This function is to be taken offline. After that, you can modify the model configuration by setting AI Engine , Runtime Dependency , and Apis .
Deployment Type	Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select Real-time services here, you can only deploy the AI application as a real-time service after it is created.
Apis	When you enable this function, you can edit RESTful APIs to define the AI application input and output formats.

- c. Check the information and click **Next**. The AI application is created.

In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is successfully created. On this page, you can perform such operations as creating new versions, quickly deploying AI applications, and publishing AI applications.

Follow-Up Procedure

Deploying the AI Applications as Services: In the AI application list, click the down arrow on the left of an AI application name to check all versions of the AI application. Locate the row that contains the target version, click **Deploy** in the **Operation** column, and select a deployment type from the drop-down list box. The AI application can be deployed in a deployment type selected during AI application creation.

2.2.3 Importing a Meta Model from a Container Image

For AI engines that are not supported by ModelArts, you can import the models you compile to ModelArts from custom images.

Prerequisites

- For details about the specifications and description of custom images, see [Custom Image Specifications for Creating an AI Application](#).
- The OBS directory you use and ModelArts are in the same region.

Creating an AI Application


1. Log in to the ModelArts management console, and choose **AI Application Management > AI Applications** in the left navigation pane. The **AI Applications** page is displayed.
2. Click **Create** in the upper left corner.
3. On the displayed page, set the parameters.
 - a. Set basic information about the AI application. For details about the parameters, see [Table 2-7](#).

Table 2-7 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Version	Version of the AI application to be created. For the first import, the default value is 0.0.1 .
Description	Brief description of an AI application

- b. Select the meta model source and set related parameters. Set **Meta Model Source** to **Container image**. For details about the parameters, see [Table 2-8](#).

Table 2-8 Parameters of the meta model source

Parameter	Description
Container Image Path	<p>Click  to import the model image from the container image. The model is of the Image type, and you do not need to use swr_location in the configuration file to specify the image location.</p> <p>NOTE The model image you select will be shared with the system administrator, so ensure that you have the permission to share the image (images shared with other accounts are unsupported). When you deploy a service, ModelArts deploys the image as an inference service. Ensure that your image can be properly started and provide an inference API.</p>
Image Replication	<p>Indicates whether to copy the model image in the container image to ModelArts.</p> <ul style="list-style-type: none"> • When this function is disabled, the model image is not copied, AI applications can be created quickly, but modifying or deleting images in the source directory of SWR may affect service deployment. • When this function is enabled, the model image is copied, AI applications cannot be created quickly, but you can modify or delete images in the source directory of SWR as that would not affect service deployment.

Parameter	Description
Health Check	<p>Health check on a model. This parameter is configurable only when the health check API is configured in the custom image. Otherwise, the AI application deployment will fail.</p> <ul style="list-style-type: none"> • Health Check URL: Enter the health check URL. • Health Check Period: Enter an integer greater than 0. The unit is second. • Maximum Failures: Enter an integer greater than 0. During service startup, if the number of consecutive health check failures reaches the specified value, the service will be abnormal. During service running, if the number of consecutive health check failures reaches the specified value, the service will enter the alarm status. <p>NOTE If health check is configured for an AI application, the deployed services using this AI application will stop 3 minutes after receiving the stop instruction.</p>
AI Application Description	<p>Provide AI application descriptions to help other AI application developers better understand and use your applications. Click Add AI Application Description and set the Document name and URL. You can add up to three AI application descriptions.</p>
Deployment Type	<p>Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select Real-time services here, you can only deploy the AI application as a real-time service after it is created.</p>
Start Command	<p>Customizable command to start a model.</p>
Apis	<p>When you enable this function, you can edit RESTful APIs to define the AI application input and output formats.</p>

- c. Check the information and click **Next**. The AI application is created. In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is successfully created. On this page, you can perform such operations as creating new versions, quickly deploying AI applications, and publishing AI applications.

Follow-Up Procedure

Deploying the AI Applications as Services: In the AI application list, click the down arrow on the left of an AI application name to check all versions of the AI

application. Locate the row that contains the target version, click **Deploy** in the **Operation** column, and select a deployment type from the drop-down list box. The AI application can be deployed in a deployment type selected during AI application creation.

2.3 Managing AI Applications

To facilitate source tracing and repeated AI application tuning, ModelArts provides the AI application version management function. You can manage models based on versions.

Prerequisites

An AI application has been created in ModelArts.

Creating a New Version

On the **AI Application Management > AI Applications** page, click **Create Version** in the **Operation** column. The **Create Version** page is displayed. Set related parameters by following the instructions in [Creating an AI Application](#) and click **Next**.

Deleting a Version

On the **AI Application Management > AI Applications** page, click the downward arrow on the left of the AI application name to expand an application version list. In the application version list, click **Delete** in the **Operation** column to delete the corresponding version.

NOTE

A deleted version cannot be recovered. Exercise caution when performing this operation.

Deleting an AI application

In the navigation pane, choose **AI Application Management > AI Applications**. On the **AI Applications** page, click **Delete** in the **Operation** column to delete the target AI application.

NOTE

If an AI application has been deployed as a service, delete the associated service before deleting the AI application. A deleted AI application cannot be recovered.

3 Deploying AI Applications as Real-Time Services

3.1 Deploying as a Real-Time Service

After an AI application is prepared, you can deploy the AI application as a real-time service and predict and call the service.

 **NOTE**

A maximum of 20 real-time services can be deployed by a user.

Prerequisites

- Data has been prepared. Specifically, you have created an AI application in the **Normal** state in ModelArts.

Procedure

1. Log in to the ModelArts management console. In the left navigation pane, choose **Service Deployment > Real-Time Services**. By default, the system switches to the **Real-Time Services** page.
2. In the real-time service list, click **Deploy** in the upper left corner. The **Deploy** page is displayed.
3. Set parameters for a real-time service.
 - a. Set basic information about model deployment. For details about the parameters, see [Table 3-1](#).

Table 3-1 Basic parameters of model deployment

Parameter	Description
Name	Name of the real-time service. Set this parameter as prompted.
Description	Brief description of the real-time service.

- b. Enter key information including the resource pool and AI application configurations. For details, see [Table 3-2](#).

Table 3-2 Parameters

Parameter	Sub-Parameter	Description
Resource Pool	Public resource pools	Instances in the public resource pool can be of the CPU or GPU type.
Resource Pool	Dedicated resource pools	Select a specification from the dedicated resource pool specifications.
AI Application and Configuration	AI Application Source	Select My AI Applications based on your requirements.
	AI Application and Version	Select the AI application and version that are in the Normal state. NOTE After a real-time service is deployed, the AI application and version cannot be changed. To change the AI application version, upgrade the real-time service. For details, see Upgrading a Real-Time Service .
	Streams	Set the traffic proportion of the current instance node. Service calling requests are allocated to the current version based on this proportion. If you deploy only one version of an AI application, set this parameter to 100% . If you select multiple versions for gated launch, ensure that the sum of the traffic ratios of multiple versions is 100% .
	Specifications	Select available specifications based on the list displayed on the console. The specifications in gray cannot be used in the current environment. If specifications in the public resource pools are unavailable, no public resource pool is available in the current environment. In this case, use a dedicated resource pool or contact the administrator to create a public resource pool. NOTE When the selected flavor is used to deploy the service, necessary system consumption is generated. Therefore, the resources actually occupied by the service are slightly greater than the selected flavor.

Parameter	Sub-Parameter	Description
	Compute Nodes	Set the number of instances for the current AI application version. If you set Instances to 1 , the standalone computing mode is used. If you set Instances to a value greater than 1, the distributed computing mode is used. Select a computing mode based on the actual requirements.
	Add AI Application Version and Configuration	If the selected AI application has multiple versions, you can add multiple versions and configure a traffic ratio. You can use grey launch to smoothly upgrade the AI application version. NOTE Free compute specifications do not support the grey launch of multiple versions.

4. After confirming the entered information, complete service deployment as prompted. Generally, service deployment jobs run for a period of time, which may be several minutes or tens of minutes depending on the amount of your selected data and resources.

 **NOTE**

After a real-time service is deployed, it is started immediately.

You can go to the real-time service list to check whether the deployment of the real-time service is complete. In the real-time service list, after the status of the newly deployed service changes from **Deploying** to **Running**, the service is deployed successfully.

3.2 Viewing Service Details

After an AI application is deployed as a real-time service, you can access the service page to view its details.

1. Log in to the ModelArts management console and choose **Service Deployment > Real-Time Services**.
2. On the **Real-Time Services** page, click the name of the target service. The service details page is displayed.

You can view the service name, status, and other information. For details, see [Table 3-3](#).


Table 3-3 Real-time service parameters

Parameter	Description
Name	Name of the real-time service.
Status	Status of the real-time service.

Parameter	Description
Source	AI application source of the real-time service.
Service ID	Real-time service ID
Failed Calls/ Total Calls	Number of service calls, which is counted from the time when the service was created. If the number of AI applications is changed or a service is invoked when an AI application is not ready, the number of calls is not counted.
Description	Service description, which can be edited after you click the edit button on the right side.
Resource Pool	Resource pool specifications used by the service.
Custom Settings	Customized configurations based on real-time service versions. This allows version-based traffic distribution policies and configurations. Enable this option and click View Settings to customize the settings. For details, see Modifying Customized Settings .

3. You can switch between tabs on the details page of a real-time service to view more details. For details, see [Table 3-4](#).

Table 3-4 Service details

Parameter	Description
Usage Guides	Displays the API address, AI application information, input parameters, and output parameters. You can click  to copy the API address to call the service.
Prediction	Performs a prediction test on the real-time service. For details, see Testing the Deployed Service .
Configuration Updates	Displays Existing Configuration and Historical Updates . <ul style="list-style-type: none"> • Current Configurations: includes the AI application name, version, status, specifications, traffic ratio, and compute nodes. • Historical Updates: displays historical AI application information.
Monitoring	Displays Resource Usage and AI Application Calls . <ul style="list-style-type: none"> • Resource Usage: includes the used and available CPU, memory, and GPU resources. • AI Application Calls: indicates the number of AI application calls. The statistics collection starts after the AI application status changes to Ready.

Parameter	Description
Logs	<p>Displays the log information about each AI application in the service. You can view logs generated in the latest 5 minutes, latest 30 minutes, latest 1 hour, and user-defined time segment.</p> <p>You can select the start time and end time when defining the time segment.</p> <p>If this function is enabled, the logs stored in LTS will be displayed. You can click View Complete Logs on LTS to view all logs.</p>

Modifying Customized Settings

A customized configuration rule consists of the configuration condition (**Setting**), access version (**Version**), and customized running parameters (including **Setting Name** and **Setting Value**).

You can configure different settings with customized running parameters for different versions of a real-time service.

The priorities of customized configuration rules are in descending order. You can change the priorities by dragging the sequence of customized configuration rules.

After a rule is matched, the system will no longer match subsequent rules. A maximum of 10 configuration rules can be configured.

Table 3-5 Parameters for Custom Settings

Parameter	Mandatory	Description
Setting	Yes	Expression of the Spring Expression Language (SPeL) rule. Only the equal, matches, and hashCode expressions of the character type are supported.
Version	Yes	Access version for a customized service configuration rule. When a rule is matched, the real-time service of the version is requested.
Setting Name	No	Key of a customized running parameter, consisting of a maximum of 128 characters. Configure this parameter if the HTTP message header is used to carry customized running parameters to a real-time service.
Setting Value	No	Value of a customized running parameter, consisting of a maximum of 256 characters. Configure this parameter if the HTTP message header is used to carry customized running parameters to a real-time service.

Customized settings can be used in the following scenarios:

- If multiple versions of a real-time service are deployed for gated launch, customized settings can be used to distribute traffic by user.

Table 3-6 Built-in variables

Built-in Variable	Description
DOMAIN_NAME	Account name that is used to invoke the inference request
DOMAIN_ID	Account ID that is used to invoke the inference request
PROJECT_NAME	Project name that is used to invoke the inference request
PROJECT_ID	Project ID that invokes the inference request
USER_NAME	Username that is used to invoke the inference request
USER_ID	User ID that is used to invoke the inference request

Pound key (#) indicates that a variable is referenced. The matched character string must be enclosed in single quotation marks.

```
#{Built-in variable} == 'Character string'
#{Built-in variable} matches 'Regular expression'
```

- Example 1:

If the account name for invoking the inference request is **User A**, the specified version is matched.

```
#DOMAIN_NAME == 'User A'
```

- Example 2:

If the account name in the inference request starts with **op**, the specified version is matched.

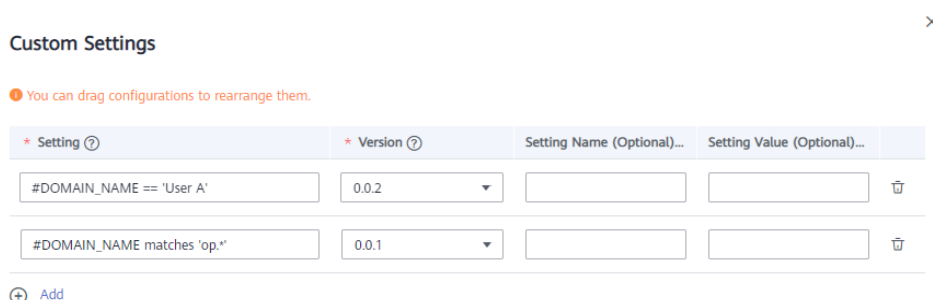
```
#DOMAIN_NAME matches 'op.*'
```

Table 3-7 Common regular expressions

Character	Description
.	Match any single character except \n. To match any character including \n, use (.\n).
*	Match the subexpression that it follows for zero or multiple times. For example, zo* can match z and zoo .

Character	Description
+	Match the subexpression that it follows for once or multiple times. For example, zo+ can match zo and zoo , but cannot match z .
?	Match the subexpression that it follows for zero or one time. For example, do(es)? can match does or do in does .
^	Match the start of the input string.
\$	Match the end of the input string.
{n}	Match for the number specified by <i>n</i> , a non-negative integer. For example, o{2} cannot match o in Bob , but can match two os in food .
x y	Match x or y. For example, z food can match z or food , and (z f)ood can match zood or food .
[xyz]	Match any single character contained in a character set. For example, [abc] can match a in plain .

Figure 3-1 Traffic distribution by user



- If multiple versions of a real-time service are deployed for gated launch, customized settings can be used to access different versions through the header.

Start with **#HEADER_**, indicating that the header is referenced as a condition.

#HEADER_{key} == '{value}'

#HEADER_{key} matches '{value}'

- Example 1:

If the header of an inference HTTP request contains a version and the value is **0.0.1**, the condition is met. Otherwise, the condition is not met.

#HEADER_version == '0.0.1'

- Example 2:

If the header of an inference HTTP request contains **testheader** and the value starts with **mock**, the rule is matched.

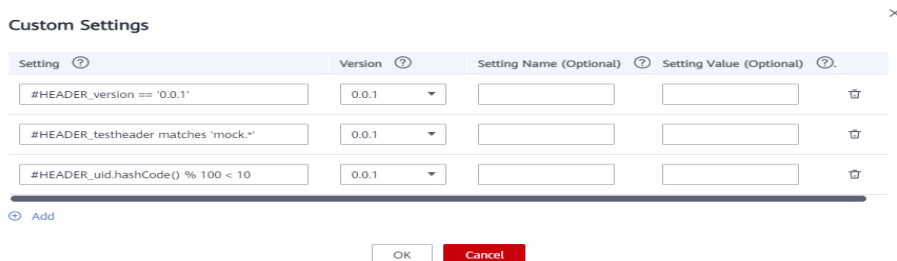
#HEADER_testheader matches 'mock.*'

- Example 3:

If the header of an inference HTTP request contains **uid** and the hash code value meets the conditions described in the following algorithm, the rule is matched.

```
#HEADER_uid.hashCode() % 100 < 10
```

Figure 3-2 Using the header to access different versions

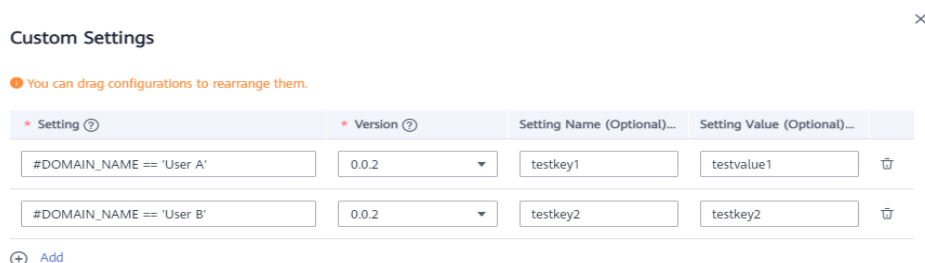


- If a real-time service version supports different running configurations, you can use **Setting Name** and **Setting Value** to specify customized running parameters so that different users can use different running configurations.

Example:

When user A accesses the AI application, the user uses configuration A. When user B accesses the AI application, the user uses configuration B. When matching a running configuration, ModelArts adds a header to the request and also the customized running parameters specified by **Setting Name** and **Setting Value**.

Figure 3-3 Customized running parameters added for a customized configuration rule



3.3 Testing the Deployed Service

After an AI application is deployed as a real-time service, you can debug code or add files for testing on the **Prediction** tab page. Based on the input request (JSON text or file) defined by the AI application, the service can be tested in either of the following ways:

1. **JSON Text Prediction:** If the input type of the AI application of the deployed service is JSON text, that is, the input does not contain files, you can enter the JSON code on the **Prediction** tab page for service testing.
2. **File Prediction:** If the input type of the AI application of the deployed service is file, including images, audios, and videos, you can add images on the **Prediction** tab page for service testing.

 NOTE

- If the input type is image, the size of a single image must be less than 8 MB.
- The following image types are supported: png, psd, jpg, jpeg, bmp, gif, webp, psd, svg, and tiff.
- If **Ascend** flavors are used during service deployment, PNG images with transparency cannot be predicted because Ascend supports only RGB-3 images.
- This function is used for commissioning. In actual production, you are advised to call APIs. You can select a test method based on the authentication mode. For details, see [Access Authenticated Using a Token](#).

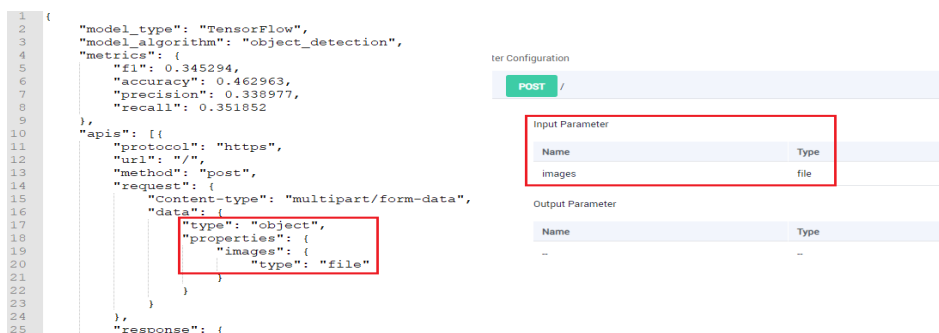
Input Parameters

For the service that you have deployed, you can learn about its input parameters of the service, that is, the input request type mentioned above, on the **Usage Guides** tab page of the service details page.

The input parameters displayed on the **Usage Guides** tab page depend on the AI application source that you select.

- If your meta model is from ExeML or a built-in algorithm, the input and output parameters are defined by ModelArts. For details, see the **Usage Guides** tab page. On the **Prediction** tab page, enter the target JSON text or file for testing.
- If you use a custom meta model with the inference code and configuration file written by yourself ([Specifications for Editing a Model Configuration File](#)), the **Usage Guides** tab page only displays your data. The following figure shows the mapping between the input parameters displayed on the **Usage Guides** tab page and the configuration file.

Figure 3-4 Mapping between the configuration file and Usage Guides



JSON Text Prediction

1. Log in to the ModelArts management console and choose **Service Deployment > Real-Time Services**.
2. On the **Real-Time Services** page, click the name of the target service. The service details page is displayed. On the **Prediction** tab page, enter the prediction code and click **Predict** to perform prediction.

File Prediction

1. Log in to the ModelArts management console and choose **Service Deployment > Real-Time Services**.
2. On the **Real-Time Services** page, click the name of the target service. The service details page is displayed. On the **Prediction** tab page, click **Upload** and select a test file. After the file is uploaded successfully, click **Predict** to perform a prediction test.

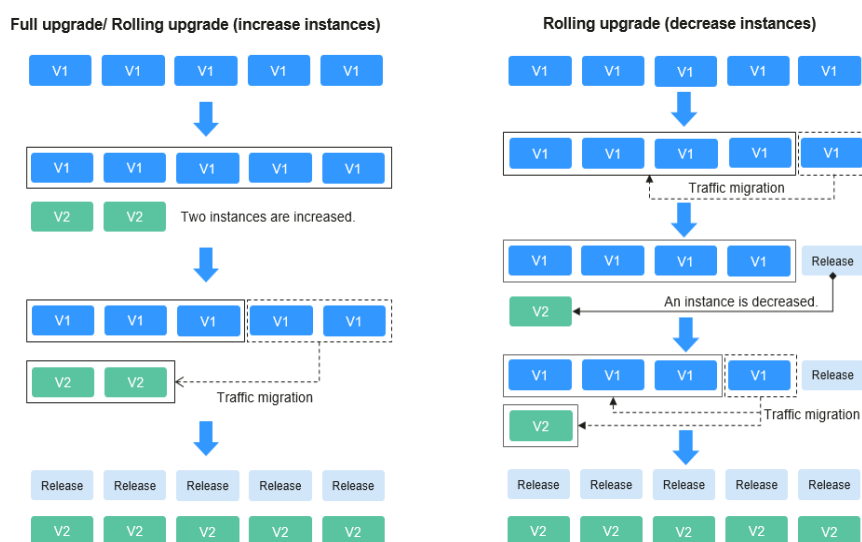
3.4 Upgrading a Real-Time Service

For a deployed service, you can change the AI application version to upgrade it.

Services can be upgraded in three modes: full upgrade, rolling upgrade (increase instances), and rolling upgrade (decrease instances). For details, see [Figure 1](#).

- Full upgrade
Twice the number of resources required by the service will be used to create new-version instances in full mode.
- Rolling upgrade (increase instances)
Extra resources will be used for a rolling upgrade. The more the instances, the faster the upgrade.
- Rolling upgrade (decrease instances)
Certain nodes that were reserved to run services will be used for a rolling upgrade. The more the instances for upgrade, the faster the upgrade, but with a higher probability of service interruption.

Figure 3-5 Upgrade process



Prerequisites

You can upgrade a service in the **Running**, **Abnormal**, **Alarm**, or **Stop** status.

Constraints

- Improper upgrade operations will interrupt services during the upgrade.
- ModelArts supports hitless rolling upgrade of real-time services in some scenarios. Prepare for and fully verify the upgrade.

For details about scenarios that support hitless upgrade of real-time services, see [Table 3-8](#).

Table 3-8 Scenarios for hitless upgrade

Meta Model Source for Creating an AI Application	Public Resource Pool	Dedicated Resource Pool
Training job	Not supported	Not supported
Template	Not supported	Not supported
Container image	Not supported	Supported The image must meet custom image specifications for creating AI applications . NOTICE If any of the following operations have been performed on an AI application version, hitless rolling upgrade is not supported: <ul style="list-style-type: none"> • Health check is not configured. • The protocol has been changed. For example, the HTTP protocol has been changed to the HTTPS protocol. • The port of the model has been changed.
OBS	Not supported	Not supported

Procedure

1. Log in to the ModelArts management console. In the navigation pane, choose **Service Deployment > Real-Time Services**. The list is sorted by **Update**.
2. Click the down arrow on the left of the target service name to show all AI application versions, and then click **Upgrade** in the **Operation** column of the target version.
3. On the **Upgrade Version** page, set parameters. For details, see [Table 3-9](#).
There are three upgrade scenarios: upgrade using the same public resource pool, upgrade using the same dedicated resource pool, and upgrade from a dedicated resource pool to another.

 **NOTE**

Services cannot be upgraded from a public resource pool to a dedicated resource pool, and vice versa.

Table 3-9 Parameters

Parameter	Description
Service Name	Name of the real-time service, which cannot be modified.
Current AI Application	Current AI application version, which cannot be modified.
Resource Pool	This parameter is available for real-time services deployed in a dedicated resource pool.
Resource Pool Spec	Select a dedicated resource pool for running the service.
AI Application and Version	New version of the AI application. Only the version can be selected.
Specifications	Select available specifications based on the list displayed on the console. The specifications in gray cannot be used in the current environment.
Environment Variable	Set environment variables and inject them to the container instance. To ensure data security, do not enter sensitive information in environment variables.

Parameter	Description
Upgrade Mode	<p>This parameter is available for real-time services deployed in a dedicated resource pool.</p> <ul style="list-style-type: none"> • Full upgrade: Twice the number of resources required by the service will be used for a one-time full upgrade. • Rolling upgrade (increase instances): Extra resources will be used for a rolling upgrade. The more the instances, the faster the upgrade. You can increase instances by quantity or proportion. <ul style="list-style-type: none"> – By quantity: The rolling upgrade is performed based on the specified number of new instances. – By proportion: The rolling upgrade is performed based on the specified proportion of new instances (rounded up). • Rolling upgrade (decrease instances): Certain nodes that were reserved to run services will be used for a rolling upgrade. The more the instances for upgrade, the faster the upgrade, but with a higher probability of service interruption. You can increase instances by quantity or proportion. <ul style="list-style-type: none"> – By quantity: The rolling upgrade is performed based on the specified number of instances that were reserved to run services. – By proportion: The rolling upgrade is performed based on the specified proportion of instances that were reserved to run services (rounded up). <p>NOTE The rolling upgrade (increase instances) and rolling upgrade (decrease instances) modes are supported only when services are upgraded in the same dedicated resource pool.</p>

4. Click **Next**. Then, confirm the information and click **Submit**. If the dedicated resource pool resources are insufficient, a message is displayed in the upper right corner of the page, notifying you to expand the capacity of the dedicated resource pool. If the resources are sufficient, the task is submitted and the real-time service list page is displayed.
5. View the service upgrade status in the real-time service list. During the service upgrade, the upgrade status and upgrade progress are displayed in the **Status** column. If the service upgrade fails, the upgrade status and rollback progress are displayed in the **Status** column. The upgrade failure time is displayed in the lower part of the status bar.

3.5 Accessing Real-Time Services

3.5.1 Access Authenticated Using a Token

If a real-time service is in the **Running** state, the real-time service has been deployed successfully. This service provides a standard RESTful API for users to call. Before integrating the API to the production environment, commission the API. You can use the following methods to send an inference request to the real-time service:

- **Method 1: Use GUI-based Software for Inference (Postman).** (Postman is recommended for Windows.)
- **Method 2: Run the cURL Command to Send an Inference Request** (curl commands are recommended for Linux.)
- **Method 3: Use Python to Send an Inference Request.**

Prerequisites

You have obtained a user token, local path to the inference file, URL of the real-time service, and input parameters of the real-time service.

- The local path to the inference file can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).
- You can obtain the service URL and input parameters of a real-time service on the Usage Guides tab page of its service details page.

The API address is the service URL of the real-time service.

Method 1: Use GUI-based Software for Inference (Postman)

1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 7.24.0 is recommended.
2. Open Postman.
3. Set parameters on Postman. The following uses image classification as an example.
 - Select a POST task and copy the API URL to the POST text box. On the **Headers** tab page, set **Key** to **X-Auth-Token** and **Value** to the user token.
 - On the **Body** tab page, file input and text input are available.
 - **File input**
Select **form-data**. Set **KEY** to the input parameter of the AI application, which must be the same as the input parameter of the real-time service. In this example, the **KEY** is **images**. Set **VALUE** to an image to be inferred (only one image can be inferred).
 - **Text input**
Select **raw** and then **JSON(application/json)**. Enter the request body in the text box below. An example request body is as follows:

```
{
  "meta": {
    "uuid": "10eb0091-887f-4839-9929-cbc884f1e20e"
  },
}
```

```
"data": {
  "req_data": [
    {
      "sepal_length": 3,
      "sepal_width": 1,
      "petal_length": 2.2,
      "petal_width": 4
    }
  ]
}
```

meta can carry a universally unique identifier (UUID). When you call an API, the system provides a UUID. When the inference result is returned, the UUID is returned to trace the request. If you do not need this function, leave **meta** blank. **data** contains a **req_data** array for one or multiple pieces of input data. The parameters of each piece of data are determined by the AI application, such as **sepal_length** and **sepal_width** in this example.

4. After setting the parameters, click **send** to send the request. The result will be displayed in **Response**.
 - Inference result using file input: The field values in the return result vary with the AI application.
 - Inference result using text input: The request body contains **meta** and **data**. If the request contains **uuid**, **uuid** will be returned in the response. Otherwise, **uuid** is left blank. **data** contains a **resp_data** array for the inference results of one or multiple pieces of input data. The parameters of each result are determined by the AI application, for example, **sepal_length** and **predictresult** in this example.

Method 2: Run the cURL Command to Send an Inference Request

The command for sending inference requests can be input as a file or text.

- File input

```
curl -kv -F 'images=@Image path' -H 'X-Auth-Token:Token value' -X POST Real-time service URL
```

- **-k** indicates that SSL websites can be accessed without using a security certificate.
- **-F** indicates file input. In this example, the parameter name is **images**, which can be changed as required. The image storage path follows **@**.
- **-H** indicates the header of a POST command. **X-Auth-Token** is the header key, which is fixed. *Token value* indicates the user token.
- **POST** is followed by the API URL of the real-time service.

The following is an example of the cURL command for inference with file input:

```
curl -kv -F 'images=@/home/data/test.png' -H 'X-Auth-Token:MIISkAY***80T9wHQ==' -X POST https://modelarts-infers-1.xxx/v1/infers/eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
```

- Text input

```
curl -kv -d '{"data":{"req_data":[{"sepal_length":3,"sepal_width":1,"petal_length":2.2,"petal_width":4}]}}' -H 'X-Auth-Token:MIISkAY***80T9wHQ==' -H 'Content-type: application/json' -X POST https://modelarts-infers-1.xxx/v1/infers/eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
```

-d indicates the text input of the request body.

Method 3: Use Python to Send an Inference Request

1. Download the Python SDK and configure it in the development tool. For details, see [Integrating the Python SDK for API request signing](#).
2. Create a request body for inference.

- **File input**

```
# coding=utf-8

import requests

if __name__ == '__main__':
    # Config url, token and file path.
    url = "URL of the real-time service"
    token = "User token"
    file_path = "Local path to the inference file"

    # Send request.
    headers = {
        'X-Auth-Token': token
    }
    files = {
        'images': open(file_path, 'rb')
    }
    resp = requests.post(url, headers=headers, files=files)

    # Print result.
    print(resp.status_code)
    print(resp.text)
```

The **files** name is determined by the input parameter of the real-time service. It is recommended that the parameter name be the same as that of the input parameter of the file type. The file **images** obtained in [Prerequisites](#) is used as an example.

- **Text input (JSON)**

The following is an example of the request body for reading the local inference file and performing Base64 encoding:

```
# coding=utf-8

import base64
import requests

if __name__ == '__main__':
    # Config url, token and file path
    url = "URL of the real-time service"
    token = "User token"
    file_path = "Local path to the inference file"
    with open(file_path, "rb") as file:
        base64_data = base64.b64encode(file.read()).decode("utf-8")

    # Set body,then send request
    headers = {
        'Content-Type': 'application/json',
        'X-Auth-Token': token
    }
    body = {
        'image': base64_data
    }
    resp = requests.post(url, headers=headers, json=body)

    # Print result
    print(resp.status_code)
    print(resp.text)
```

The **body** name is determined by the input parameter of the real-time service. It is recommended that the parameter name be the same as that of the input parameter of the string type.

3.5.2 Access Authenticated Using an AK/SK

If a real-time service is in the **Running** state, the real-time service has been deployed successfully. This service provides a standard RESTful API for users to call. Users can call the API using AK/SK-based authentication.

This section describes how to use the APIG SDK to access a real-time service. The process is as follows:

1. **Obtaining an AK/SK Pair**
2. **Obtaining Information About a Real-Time Service**
3. Sending an Inference Request
 - **Method 1: Use Python to Send an Inference Request**
 - **Method 2: Use Java to Send an Inference Request**

NOTE

1. AK/SK-based authentication supports API requests with a body not larger than 12 MB. For API requests with a larger body, token-based authentication is recommended.
2. The local time on the client must be synchronized with the clock server to avoid a large offset in the value of the **X-Sdk-Date** request header. API Gateway checks the time format and compares the time with the time when API Gateway receives the request. If the time difference exceeds 15 minutes, API Gateway will reject the request.

Obtaining an AK/SK Pair

If an AK/SK pair is already available, skip this step. Find the downloaded AK/SK file, which is usually named **credentials.csv**.

As shown in the following figure, the file contains the username, AK, and SK.

Figure 3-6 Content of the credential.csv file

	A	B	C
1	User Name	Access Key Id	Secret Access Key
2	hu*****dg	QTWA*****UT2QVKYUC	MFyfvk41ba2*****npdUKGpownRZImVmHc

Perform the following operations to generate an AK/SK pair:

1. Register with and log in to the management console.
2. Click the username and choose **My Credentials** from the drop-down list.
3. On the **My Credentials** page, choose **Access Keys** in the navigation pane.
4. Click **Create Access Key**. The **Identity Verification** dialog box is displayed.
5. Complete the identity authentication as prompted, download the access key, and keep it secure.

Obtaining Information About a Real-Time Service

When calling an API, you need to obtain the API address and input parameters of the real-time service. The procedure is as follows:

1. Log in to the ModelArts management console. In the left navigation pane, choose **Service Deployment > Real-Time Services**. By default, the system switches to the **Real-Time Services** page.
2. Click the name of the target service. The service details page is displayed.
3. On the details page of a real-time service, obtain the API address and input parameters of the service.

The API address is the service URL of the real-time service.

Method 1: Use Python to Send an Inference Request

1. Download the Python SDK and configure it in the development tool. For details, see [Integrating the Python SDK for API request signing](#).
2. Create a request body for inference.

– **File input**

```
# coding=utf-8

import requests
from apig_sdk import signer

if __name__ == '__main__':
    # Config url, ak, sk and file path.
    url = "URL of the real-time service"
    ak = "AK"
    sk = "SK"
    file_path = "Local path to the inference file"

    # Create request, set method, url, headers and body.
    method = 'POST'
    headers = {"x-sdk-content-sha256": "UNSIGNED-PAYLOAD"}
    request = signer.HttpRequest(method, url, headers)

    # Create sign, set the AK/SK to sign and authenticate the request.
    sig = signer.Signer()
    sig.Key = ak
    sig.Secret = sk
    sig.Sign(request)

    # Send request
    files = {'images': open(file_path, 'rb')}
    resp = requests.request(request.method, request.scheme + "://" + request.host + request.uri,
headers=request.headers, files=files)

    # Print result
    print(resp.status_code)
    print(resp.text)
```

file_path is the local path to the inference file. The path can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).

Request body format of **files**: `files = {"Request parameter": ("Load path", File content, "File type")}`. For details about parameters of **files**, see [Table 3-10](#).

Table 3-10 Parameters of files

Parameter	Mandatory	Description
Request parameter	Yes	Enter the parameter name of the real-time service.
Load path	No	Path in which the file is stored.
File content	Yes	Content of the file to be uploaded.
File type	No	Type of the file to be uploaded, which can be one of the following options: <ul style="list-style-type: none"> • txt: text/plain • jpg/jpeg: image/jpeg • png: image/png

– **Text input (JSON)**

The following is an example of the request body for reading the local inference file and performing Base64 encoding:

```
# coding=utf-8

import base64
import json
import requests
from apig_sdk import signer

if __name__ == '__main__':
    # Config url, ak, sk and file path.
    url = "URL of the real-time service"
    ak = "AK"
    sk = "SK"
    file_path = "Local path to the inference file"
    with open(file_path, "rb") as file:
        base64_data = base64.b64encode(file.read()).decode("utf-8")

    # Create request, set method, url, headers and body.
    method = 'POST'
    headers = {
        'Content-Type': 'application/json'
    }
    body = {
        'image': base64_data
    }
    request = signer.HttpRequest(method, url, headers, json.dumps(body))

    # Create sign, set the AK/SK to sign and authenticate the request.
    sig = signer.Signer()
    sig.Key = ak
    sig.Secret = sk
    sig.Sign(request)

    # Send request
    resp = requests.request(request.method, request.scheme + "://" + request.host + request.uri,
        headers=request.headers, data=request.body)

    # Print result
```

```
print(resp.status_code)
print(resp.text)
```

The **body** name is determined by the input parameter of the real-time service. It is recommended that the parameter name be the same as that of the input parameter of the string type.

Method 2: Use Java to Send an Inference Request

1. Download the Java SDK and configure it in the development tool. For details, see [Integrating the Java SDK for API request signing](#).
2. Create a Java request body for inference.

In the APiG Java SDK, **request.setBody()** can only be a string. Therefore, only text inference requests are supported.

The following is an example of the request body (JSON) for reading the local inference file and performing Base64 encoding:

```
// Package name of the demo.
package com.apig.sdk.demo;

import com.cloud.apigateway.sdk.utils.Client;
import com.cloud.apigateway.sdk.utils.Request;
import org.apache.http.HttpHeaders;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpRequestBase;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

public class MyAkSkTest {

    public static void main(String[] args) {
        String url = "URL of the real-time service";
        String ak = "AK";
        String sk = "SK";
        String body = "{}";

        try {
            // Create request
            Request request = new Request();

            // Set the AK/SK to sign and authenticate the request.
            request.setKey(ak);
            request.setSecret(sk);

            // Specify a request method, such as GET, PUT, POST, DELETE, HEAD, and PATCH.
            request.setMethod(HttpPost.METHOD_NAME);

            // Add header parameters
            request.addHeader(HttpHeaders.CONTENT_TYPE, "application/json");

            // Set a request URL in the format of https://{Endpoint}/{URI}.
            request.setUrl(url);

            // Special characters, such as the double quotation mark ("), contained in the body must be
            // escaped.
            request.setBody(body);

            // Sign the request.
            HttpRequestBase signedRequest = Client.sign(request);

            // Send request.
            CloseableHttpResponse response = HttpClients.createDefault().execute(signedRequest);


            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
        }
    }
}
```

```
        System.out.println(EntityUtils.toString(response.getEntity()));  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

body is determined by the text format. JSON is used as an example.

4 Deploying AI Applications as Batch Services

4.1 Deploying as a Batch Service

After an AI application is prepared, you can deploy it as a batch service. The **Service Deployment > Batch Services** page lists all batch services. You can enter a service name in the search box in the upper right corner and click  to query the service.

Prerequisites

- Data has been prepared. Specifically, you have created an AI application in the **Normal** state in ModelArts.
- Data to be batch processed is ready and has been upload to an OBS directory.
- At least one empty folder has been created in OBS for storing the output.

Background

- A maximum of 1,000 batch services can be created.
- Based on the input request (JSON or other file) defined by the AI application, different parameter are entered. If the AI application input is a JSON file, a configuration file is required to generate a mapping file. If the AI application input is other file, no mapping file is required.
- Batch services can only be deployed in a public resource pool, but not a dedicated resource pool.

Procedure

1. Log in to the ModelArts management console. In the left navigation pane, choose **Service Deployment > Batch Services**. By default, the **Batch Services** page is displayed.
2. In the batch service list, click **Deploy** in the upper left corner. The **Deploy** page is displayed.
3. Set parameters for a batch service.

- a. Set the basic information, including **Name** and **Description**. The name is generated by default, for example, **service-bc0d**. You can specify **Name** and **Description** according to actual requirements.
- b. Set other parameters, including AI application configurations. For details, see [Table 4-1](#).

Table 4-1 Parameters

Parameter	Description
AI Application Source	Select My AI Applications based on your requirements.
AI Application and Version	Select the AI application and version that are in the Normal state.
Input Path	Select the OBS directory where the data is to be uploaded. Select a folder or a .manifest file. For details about the specifications of the .manifest file, see Manifest File Specifications . NOTE <ul style="list-style-type: none">• If the input data is an image, ensure that the size of a single image is less than 10 MB.• If the input data is in CSV format, ensure that no Chinese character is included. To use Chinese, set the file encoding format to UTF-8. You can convert the file encoding format using code, or open the CSV file using Notepad and set the encoding format in the window displayed after you click Save As.
Output Path	Select the path for saving the batch prediction result. You can select the empty folder that you create.
Specifications	Select available specifications based on the list displayed on the console. The specifications in gray cannot be used at the current site.
Compute Nodes	Set the number of instances for the current AI application version. If you set Instances to 1 , the standalone computing mode is used. If you set Instances to a value greater than 1, the distributed computing mode is used. Select a computing mode based on the actual requirements.
Environment Variable	Set environment variables and inject them to the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables.

4. After setting the parameters, deploy the model as a batch service as prompted. Generally, service deployment jobs run for a period of time, which may be several minutes or tens of minutes depending on the amount of your selected data and resources.

You can go to the batch service list to view the basic information about the batch service. In the batch service list, after the status of the newly deployed service changes from **Deploying** to **Running**, the service is deployed successfully.

Manifest File Specifications

Batch services of the inference platform support the manifest file. The manifest file describes the input and output of data.

Example input manifest file

- File name: **test.manifest**
- File content:

```
{"source": "/test/data/1.jpg"}
{"source": "/xgboosterdata/data.csv?
AccessKeyId=2Q0V0TQ461N26DDL18RB&Expires=1550611914&Signature=wZBttZj5QZrReDhz1uDzwve
8GpY%3D&x-obs-security-token=gQpzb3V0aGNoaW5hixvY8V9a1SnsxmGoHYmB1SArYMyqnQT-
ZaMSxHvl68kKLAY5feYvLDM..."}
```
- File requirements:
 - a. The file name extension must be **.manifest**.
 - b. The file content is in JSON format. Each row describes a piece of input data, which must be accurate to a file instead of a folder.
 - c. The value of **source** is the OBS file path in the format of */{{Bucket name}}/{{Object name}}*.

Example output manifest file

A manifest file will be generated in the output directory of the batch services.

- Assume that the output path is **//test-bucket/test/**. The result is stored in the following path:

```
OBS bucket/directory name
├── test-bucket
│   └── test
│       ├── infer-result-{{index}}.manifest
│       ├── infer-result
│       │   ├── 1.jpg_result.txt
│       │   └── 2.jpg_result.txt
```
- Content of the **infer-result-0.manifest** file:

```
{"source": "/obs-data-bucket/test/data/1.jpg","inference-loc": "<obs path>/test-bucket/test/infer-result/1.jpg_result.txt"}
{"source": "/xgboosterdata/2.jpg?
AccessKeyId=2Q0V0TQ461N26DDL18RB&Expires=1550611914&Signature=wZBttZj5QZrReDhz1uDzwve
8GpY%3D&x-obs-security-token=gQpzb3V0aGNoaW5hixvY8V9a1SnsxmGoHYmB1SArYMyqnQT-
ZaMSxHvl68kKLAY5feYvLDMNZWxzHbz6Q-3HcoZMh9glSwQOVBwm4ZytB_m8sg1fL6isU7T3CnoL9jmv
DGgT9VBC7dC1EyfSjrUcqfB...", "inference-loc": "obs://test-bucket/test/infer-result/2.jpg_result.txt"}
```
- File format:
 - a. The file name is **infer-result-{{index}}.manifest**, where **index** is the instance ID. Each running instance of a batch service generates a manifest file.
 - b. The **infer-result** directory is created in the manifest directory to store the file processing result.
 - c. The file content is in JSON format. Each row describes the output result of a piece of input data.
 - d. The content contains multiple fields:

- i. **source**: input data description, which is the same as that of the input manifest file
- ii. **inference-loc**: output result path in the format of **<obs path>/{{Bucket name}}/{{Object name}}**

Example Mapping

The following example shows the relationship between the configuration file, mapping rule, CSV data, and inference request.

Assume that the **apis** parameter in the configuration file used by your model is as follows:

```
[
  {
    "protocol": "http",
    "method": "post",
    "url": "/",
    "request": {
      "type": "object",
      "properties": {
        "data": {
          "type": "object",
          "properties": {
            "req_data": {
              "type": "array",
              "items": [
                {
                  "type": "object",
                  "properties": {
                    "input_1": {
                      "type": "number"
                    },
                    "input_2": {
                      "type": "number"
                    },
                    "input_3": {
                      "type": "number"
                    },
                    "input_4": {
                      "type": "number"
                    }
                  }
                }
              ]
            }
          }
        }
      }
    }
  }
]
```

At this point, the corresponding mapping relationship is shown below. The ModelArts management console automatically resolves the mapping relationship from the configuration file. When calling a ModelArts API, write the mapping relationship by yourself according to the rule.

```
{
  "type": "object",
  "properties": {
    "data": {
      "type": "object",
      "properties": {
        "req_data": {
```

```
"type": "array",
"items": [
  {
    "type": "object",
    "properties": {
      "input_1": {
        "type": "number",
        "index": 0
      },
      "input_2": {
        "type": "number",
        "index": 1
      },
      "input_3": {
        "type": "number",
        "index": 2
      },
      "input_4": {
        "type": "number",
        "index": 3
      }
    }
  }
]
```

The data for inference, that is, the CSV data, is in the following format. The data must be separated by commas (,).

```
5.1,3.5,1.4,0.2
4.9,3.0,1.4,0.2
4.7,3.2,1.3,0.2
```

Depending on the defined mapping relationship, the inference request is shown below. The format is similar to the format used by the real-time service.


```
{
  "data": {
    "req_data": [{
      "input_1": 5.1,
      "input_2": 3.5,
      "input_3": 1.4,
      "input_4": 0.2
    }]
  }
}
```

4.2 Viewing the Batch Service Prediction Result

When deploying a batch service, you can select the location of the output data directory. You can view the running result of the batch service that is in the **Running completed** status.

Procedure

1. Log in to the ModelArts management console and choose **Service Deployment > Batch Services**.
2. Click the name of the target service in the **Running completed** status. The service details page is displayed.

- You can view the service name, status, ID, input path, output path, and description.
 - You can click  in the **Description** area to edit the description.
3. Obtain the detailed OBS path next to **Output Path**, switch to the path and obtain the batch service prediction results, including the prediction result file and the AI application prediction result.

If the prediction is successful, the directory contains the prediction result file and AI application prediction result. Otherwise, the directory contains only the prediction result file.

- Prediction result file: The file is in *xxx.manifest* format, which contains the file path and prediction result, and more.
- AI application prediction result:
 - If images are entered, a result file is generated for each image in the *Image name__result.txt* format, for example, **IMG_20180919_115016.jpg_result.txt**.
 - If audio files are entered, a result file is generated for each audio file in the *Audio file name__result.txt* format, for example, **1-36929-A-47.wav_result.txt**.
 - If table data is entered, the result file is generated in the *Table name__result.txt* format, for example, **train.csv_result.txt**.

5 Upgrading a Service

For a deployed service, you can modify its basic information to match service changes and change the AI application version to upgrade it.

You can modify the basic information about a service in either of the following ways:

[Method 1: Modify Service Information on the Service Management Page](#)

[Method 2: Modify Service Information on the Service Details Page](#)

Prerequisites

The service has been deployed. The service in the **Deploying** state cannot be upgraded by modifying the service information.

Constraints

- Improper upgrade operations will interrupt service running during the upgrade. Therefore, exercise caution when performing this operation.
- ModelArts supports hitless rolling upgrade of real-time services in some scenarios. Before upgrade, prepare for it and confirm the prerequisites.

Table 5-1 Scenarios for hitless rolling upgrade

Meta Model Source for Creating an AI Application	Using a Public Resource Pool	Using a Dedicated Resource Pool
Training job	Not supported	Not supported
Template	Not supported	Not supported
Container image	Not supported	Supported. The custom image for creating the AI application must comply with the custom image specifications .
OBS	Not supported	Not supported

Method 1: Modify Service Information on the Service Management Page

1. Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service.
2. In the service list, click **Modify** in the **Operation** column of the target service, modify basic service information, and submit the modification task as prompted.

When some parameters are modified, the system automatically restarts the service for the modification to take effect. When you submit a service modification task, if a restart is required, a dialog box will be displayed.

- For details about the real-time service parameters, see [Deploying as a Real-Time Service](#).
- For details about the batch service parameters, see [Deploying as a Batch Service](#).

Method 2: Modify Service Information on the Service Details Page

1. Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service.
2. Click the name of the target service. The service details page is displayed.
3. Click **Modify** in the upper right corner of the page, modify the service details, and submit the modification task as prompted.

When some parameters are modified, the system automatically restarts the service for the modification to take effect. When you submit a service modification task, if a restart is required, a dialog box will be displayed.

- For details about the real-time service parameters, see [Deploying as a Real-Time Service](#).
- For details about the batch service parameters, see [Deploying as a Batch Service](#).

6 Starting or Stopping a Service

Starting a Service

You can start services in the **Successful**, **Abnormal**, or **Stopped** status. Services in the **Deploying** state cannot be started. You can start a service in the following ways:

- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click **Start** in the **Operation** column to start a service. (For a real-time service, choose **More** > **Start** in the **Operation** column.)
- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click the name of the target service. The service details page is displayed. Click **Start** in the upper right corner of the page to start the service.

Stopping a Service

Stop a service in either of the following ways:

- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click **Stop** in the **Operation** column to stop a service. (For a real-time service, choose **More** > **Stop** in the **Operation** column.)
- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click the name of the target service. The service details page is displayed. Click **Stop** in the upper right corner of the page to stop the service.

7 Deleting a Service

If a service is no longer in use, you can delete it to release resources.

Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service.

- For a real-time service, choose **More > Delete** in the **Operation** column to delete the service
- For a batch service, click **Delete** in the **Operation** column to delete it.

NOTE

- A deleted service cannot be recovered.
- A service cannot be deleted without agency authorization.

8 Monitoring

8.1 ModelArts Metrics

Description

The cloud service platform provides Cloud Eye to help you better understand the status of your ModelArts real-time services and models. You can use Cloud Eye to automatically monitor your ModelArts real-time services and models in real time and manage alarms and notifications, so that you can keep track of performance metrics of ModelArts and models.

Namespace

SYS.ModelArts

Monitoring Metrics

Table 8-1 ModelArts metrics

Metric ID	Metric Name	Description	Value Range	Monitored Entity	Monitoring Interval
cpu_usage	CPU Usage	CPU usage of ModelArts Unit: %	≥ 0%	ModelArts models	1 minute
mem_usage	Memory Usage	Memory usage of ModelArts Unit: %	≥ 0%	ModelArts models	1 minute
gpu_util	GPU Usage	GPU usage of ModelArts Unit: %	≥ 0%	ModelArts models	1 minute

Metric ID	Metric Name	Description	Value Range	Monitored Entity	Monitoring Interval
successfully_called_times	Number of Successful Calls	Times that ModelArts has been successfully called Unit: Times/min	≥Count/min	ModelArts models ModelArts real-time services	1 minute
failed_called_times	Number of Failed Calls	Times that ModelArts failed to be called Unit: Times/min	≥Count/min	ModelArts models ModelArts real-time services	1 minute
total_called_times	API Calls	Times that ModelArts is called Unit: Times/min	≥Count/min	ModelArts models ModelArts real-time services	1 minute
<p>If a measurement object has multiple measurement dimensions, all the measurement dimensions are mandatory when you use an API to query monitoring metrics.</p> <ul style="list-style-type: none"> The following provides an example of using the multi-dimensional dim to query a single monitoring metric: dim.0=service_id,530cd6b0-86d7-4818-837f-935f6a27414d&dim.1="model_id,3773b058-5b4f-4366-9035-9bbd9964714a" The following provides an example of using the multi-dimensional dim to query monitoring metrics in batches: "dimensions": [<pre> { "name": "service_id", "value": "530cd6b0-86d7-4818-837f-935f6a27414d" } { "name": "model_id", "value": "3773b058-5b4f-4366-9035-9bbd9964714a" }],</pre> 					

Dimensions

Table 8-2 Dimension description

Key	Value
service_id	Real-time service ID
model_id	Model ID

8.2 Setting Alarm Rules

Scenario

Setting alarm rules allows you to customize the monitored objects and notification policies so that you can know the status of ModelArts real-time services and models in a timely manner.

An alarm rule includes the alarm rule name, monitored object, metric, threshold, monitoring interval, and whether to send a notification. This section describes how to set alarm rules for ModelArts services and models.

 **NOTE**

Only real-time services in the **Running** status can be interconnected with CES.

Prerequisites

You have created a ModelArts real-time service.

Procedure

1. Log in to the management console.
2. Click **Service List**. Under **Management & Deployment**, click **Cloud Eye**.
3. On the Cloud Eye page, click **Custom Monitoring**. Then, enable ModelArts monitoring as prompted.
4. In the left navigation pane, choose **Cloud Service Monitoring > ModelArts**.
5. Select a real-time service for which you want to create an alarm rule and click **Create Alarm Rule** in the **Operation** column.
6. On the **Create Alarm Rule** page, create an alarm rule for ModelArts real-time services and models as prompted.
7. After the setting is complete, click **Create**. When an alarm that meets the rule is generated, the system automatically sends a notification.

8.3 Viewing Monitoring Metrics

Scenario

Cloud Eye on the cloud service platform monitors the status of ModelArts real-time services and model loads. You can obtain the monitoring metrics of each



ModelArts real-time service and model loads on the management console. Monitored data requires a period of time for transmission and display. The status of ModelArts displayed on the Cloud Eye console is usually the status obtained 5 to 10 minutes before. You can view the monitored data of a newly created real-time service 5 to 10 minutes later.

Prerequisites

- The ModelArts real-time service is running properly.
- Alarm rules have been configured on the Cloud Eye page. For details, see [Setting Alarm Rules](#).
- The real-time service has been properly running for at least 10 minutes.
- The monitoring data and graphics are available for a new real-time service after the service runs for at least 10 minutes.
- Cloud Eye does not display the metrics of a faulty or deleted real-time service. The monitoring metrics can be viewed after the real-time service starts or recovers.

Monitoring data is unavailable without alarm rules configured on Cloud Eye. For details, see [Setting Alarm Rules](#).

Procedure

1. Log in to the management console.
2. Click **Service List**. Under **Management & Deployment**, click **Cloud Eye**.
3. In the left navigation pane, choose **Cloud Service Monitoring > ModelArts**.
4. View monitoring graphs.
 - Viewing monitoring graphs of the real-time service: Click **View Graph** in the **Operation** column.
 - Viewing monitoring graphs of the model loads: Click  next to the target real-time service, and select **View Graph** from the drop-down list for model loads in the **Operation** column.
5. In the monitoring area, you can select a duration to view the monitoring data. You can view the monitoring data in the recent 1 hour, 3 hours, or 12 hours. To view the monitoring curve of a longer time range, click  to enlarge the graph.

9 Inference Specifications

9.1 Model Package Specifications

9.1.1 Introduction to Model Package Specifications

When creating an AI application on the AI application management page, make sure that any meta model imported from OBS complies with certain specifications.

 **NOTE**

The model package specifications are used when you import one model. If you import multiple models, for example, there are multiple model files, use custom images.

The model package must contain the **model** directory. The **model** directory stores the model file, model configuration file, and model inference code file.

- **Model files:** The requirements for model files vary according to the model package structure. For details, see [Model Package Example](#).
- **Model configuration file:** The model configuration file must be available and its name is consistently to be **config.json**. There must be only one model configuration file. For details about how to edit a model configuration file, see [Specifications for Editing a Model Configuration File](#).
- **Model inference code file:** It is optional. If this file is required, the file name is consistently to be **customize_service.py**. There must be only one model inference code file. For details about how to edit model inference code, see [Specifications for Writing Model Inference Code](#).
 - The **.py** file on which **customize_service.py** depends can be directly stored in the **model** directory. Use a relative import mode to import the custom package.
 - The other files on which **customize_service.py** depends can be stored in the **model** directory. You must use absolute paths to access these files. For more details, see [Obtaining an Absolute Path](#).

ModelArts also provides custom script examples of common AI engines. For details, see [Examples of Custom Scripts](#).

Model Package Example

- Structure of the TensorFlow-based model package

When publishing the model, you only need to specify the **ocr** directory.

OBS bucket/directory name

```

|— ocr
|   |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   |— <<Custom Python package>> (Optional) User's Python package, which can be directly
|   |   |   |— saved_model.pb (Mandatory) Protocol buffer file, which contains the diagram description
|   |   |   |   |— variables Name of a fixed sub-directory, which contains the weight and deviation rate of
|   |   |   |   |   |— variables.index Mandatory
|   |   |   |   |   |— variables.data-00000-of-00001 Mandatory
|   |   |   |   |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json.
|   |   |   |   |   |— customize_service.py (Optional) Model inference code. The file name is fixed to
|   |   |   |   |   |   |— customize_service.py depends can be directly stored in the model directory.

```

- Structure of the MindSpore-based model package

OBS bucket/directory name

```

|— resnet
|   |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   |— <<Custom Python package>> (Optional) User's Python package, which can be directly
|   |   |   |— checkpoint_lenet_1-1_1875.ckpt (Mandatory) Model file in ckpt format trained using
|   |   |   |   |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json.
|   |   |   |   |   |— customize_service.py (Optional) Model inference code. The file name is fixed to
|   |   |   |   |   |   |— customize_service.py depends can be directly stored in the model directory.

```

- Structure of the MXNet-based model package

When publishing the model, you only need to specify the **resnet** directory.

OBS bucket/directory name

```

|— resnet
|   |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   |— <<Custom Python package>> (Optional) User's Python package, which can be directly
|   |   |   |— resnet-50-symbol.json (Mandatory) Model definition file, which contains the neural
|   |   |   |   |— resnet-50-0000.params (Mandatory) Model variable parameter file, which contains
|   |   |   |   |   |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json.
|   |   |   |   |   |   |— customize_service.py (Optional) Model inference code. The file name is fixed to
|   |   |   |   |   |   |   |— customize_service.py depends can be directly stored in the model directory.

```

- Structure of the Image-based model package

When publishing the model, you only need to specify the **resnet** directory.

OBS bucket/directory name

```

|— resnet
|   |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   |— config.json (Mandatory) Model configuration file (the address of the SWR image must be
|   |   |   |— customize_service.py (Optional) Model inference code. The file name is fixed to
|   |   |   |   |— customize_service.py depends can be directly stored in the model directory.

```

- Structure of the PySpark-based model package

When publishing the model, you only need to specify the **resnet** directory.

OBS bucket/directory name

```

|— resnet
|   |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   |— <<Custom Python package>> (Optional) User's Python package, which can be directly

```

```

referenced in model inference code
| | | |— spark_model (Mandatory) Model directory, which contains the model content saved by
PySpark
| | | |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json.
Only one model configuration file is supported.
| | | |— customize_service.py (Optional) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file exists. The files on which
customize_service.py depends can be directly stored in the model directory.

```

- Structure of the PyTorch-based model package

When publishing the model, you only need to specify the **resnet** directory.

```

OBS bucket/directory name
|— resnet
| |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
| | |— <<Custom Python package>> (Optional) User's Python package, which can be directly
referenced in model inference code
| | |— resnet50.pth (Mandatory) PyTorch model file, which contains variable and weight
information and is saved as state_dict
| | |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json.
Only one model configuration file is supported.
| | |— customize_service.py (Mandatory) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file exists. The files on which
customize_service.py depends can be directly stored in the model directory.

```

- Structure of the Caffe-based model package

When publishing the model, you only need to specify the **resnet** directory.

```

OBS bucket/directory name
|— resnet
| |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
| | |— <<Custom Python package>> (Optional) User's Python package, which can be directly
referenced in model inference code
| | |— deploy.prototxt (Mandatory) Caffe model file, which contains information such as the model
network structure
| | |— resnet.caffemodel (Mandatory) Caffe model file, which contains variable and weight
information
| | |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json. Only
one model configuration file is supported.
| | |— customize_service.py (Optional) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file exists. The files on which
customize_service.py depends can be directly stored in the model directory.

```

- Structure of the XGBoost-based model package

When publishing the model, you only need to specify the **resnet** directory.

```

OBS bucket/directory name
|— resnet
| |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
| | |— <<Custom Python package>> (Optional) User's Python package, which can be directly
referenced in model inference code
| | |— *.m (Mandatory): Model file whose extension name is .m
| | |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json. Only
one model configuration file is supported.
| | |— customize_service.py (Optional) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file exists. The files on which
customize_service.py depends can be directly stored in the model directory.

```

- Structure of the Scikit_Learn-based model package

When publishing the model, you only need to specify the **resnet** directory.

```

OBS bucket/directory name
|— resnet
| |— model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
| | |— <<Custom Python package>> (Optional) User's Python package, which can be directly
referenced in model inference code
| | |— *.m (Mandatory): Model file whose extension name is .m
| | |— config.json (Mandatory) Model configuration file. The file name is fixed to config.json. Only
one model configuration file is supported.
| | |— customize_service.py (Optional) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file exists. The files on which
customize_service.py depends can be directly stored in the model directory.

```

9.1.2 Specifications for Editing a Model Configuration File

A model developer needs to compile a configuration file when publishing a model. The model configuration file describes the model usage, computing framework, precision, inference code dependency package, and model API.

Configuration File Format

The configuration file is in JSON format. [Table 9-1](#) describes the parameters.

Table 9-1 Parameters

Parameter	Mandatory	Data Type	Description
model_algorithm	Yes	String	Model algorithm, which is set by the model developer to help model users understand the usage of the model. The value must start with a letter and contain no more than 36 characters. Special characters (&!\"<>=) are not allowed. Common model algorithms include image_classification (image classification), object_detection (object detection), and predict_analysis (prediction analysis).
model_type	Yes	String	Model AI engine, which indicates the computing framework used by a model. Common AI engines and Image are supported. <ul style="list-style-type: none">For details about supported AI engines, see Supported AI Engines for ModelArts Inference.If model_type is set to Image, the AI application is created using a custom image. In this case, parameter swr_location is mandatory.
runtime	No	String	Model runtime environment. Python3.6 is used by default. The value of runtime depends on the value of model_type . If model_type is set to Image , you do not need to set runtime . If model_type is set to another frequently-used framework, select the engine and runtime environment. For details about the supported running environments, see Supported AI Engines for ModelArts Inference . If your model needs to run on a specified CPU or GPU, select the runtime based on the suffix information. If the runtime does not contain the CPU or GPU information, read the description of each runtime in <i>Supported AI Engines for ModelArts Inference</i> .

Parameter	Mandatory	Data Type	Description
swr_location	No	String	<p>SWR image address.</p> <ul style="list-style-type: none"> If you import a custom image metamodel from a container image, you do not need to set swr_location. If you import a custom image metamodel from OBS (not recommended) and set model_type to Image, you must set swr_location. swr_location indicates the address of the Docker image on SWR, indicating that the Docker image on SWR is used to publish the model.
metrics	No	Object	<p>Model precision information, including the average value, recall rate, precision, and accuracy. For details about the metrics object structure, see Table 9-2.</p> <p>The result is displayed in the model precision area on the AI application details page.</p>
apis	No	api array	<p>Format of the requests received and returned by a model. The value is structure data. It is the RESTful API array provided by a model. For details about the API data structure, see Table 9-3.</p> <ul style="list-style-type: none"> When model_type is set to Image, that is, in the model scenario of a custom image, APIs with different paths can be declared in apis based on the request path exposed by the image. When model_type is not Image, only one API whose request path is / can be declared in apis because the preconfigured AI engine exposes only one inference API whose request path is /.
dependencies	No	dependency array	<p>Package on which the model inference code depends, which is structure data.</p> <p>Model developers need to provide the package name, installation mode, and version constraints. Only the pip installation mode is supported. Table 9-6 describes the dependency array.</p> <p>If the model package does not contain the customize_service.py file, you do not need to set this parameter. Dependency packages cannot be installed for custom image models.</p>

Parameter	Mandatory	Data Type	Description
health	No	health data structure	Configuration of an image health interface. This parameter is mandatory only when model_type is set to Image . If services cannot be interrupted during the rolling upgrade, a health check port must be provided for ModelArts to call. For details about the health data structure, see Table 9-8 .

Table 9-2 metrics object description

Parameter	Mandatory	Data Type	Description
f1	No	Number	F1 score. The value is rounded to 17 decimal places.
recall	No	Number	Recall rate. The value is rounded to 17 decimal places.
precision	No	Number	Precision. The value is rounded to 17 decimal places.
accuracy	No	Number	Accuracy. The value is rounded to 17 decimal places.

Table 9-3 api array

Parameter	Mandatory	Data Type	Description
protocol	No	String	Request protocol. Set the parameter value to http or https based on your custom image. If you use a metamodel imported from OBS, the default protocol is https . For details about other parameter, see Example of the Object Detection Model Configuration File .
url	No	String	Request path. The default value is a slash (/). For a custom image model (model_type is Image), set this parameter to the actual request path exposed in the image. For a non-custom image model (model_type is not Image), the URL can only be /.
method	No	String	Request method. The default value is POST .
request	No	Object	Request body. For details about the request structure, see Table 9-4 .

Parameter	Mandatory	Data Type	Description
response	No	Object	Response body. For details about the response structure, see Table 9-5 .

Table 9-4 request description

Parameter	Mandatory	Data Type	Description
Content-type	Yes for real-time services No for batch services	String	Data is sent in a specified content format. The default value is application/json . The options are as follows: <ul style="list-style-type: none"> • application/json: sends JSON data. • multipart/form-data: uploads a file. NOTE For machine learning models, only application/json is supported.
data	Yes for real-time services No for batch services	String	The request body is described in JSON schema. For details about the parameter description, see the official guide .

Table 9-5 response description

Parameter	Mandatory	Data Type	Description
Content-type	Yes for real-time services No for batch services	String	Data is sent in a specified content format. The default value is application/json . NOTE For machine learning models, only application/json is supported.
data	Yes for real-time services No for batch services	String	The response body is described in JSON schema. For details about the parameter description, see the official guide .

Table 9-6 dependency array

Parameter	Mandatory	Data Type	Description
installer	Yes	String	Installation method. Only pip is supported.
packages	Yes	package array	Dependency package collection. For details about the package structure array, see Table 9-7 .

Table 9-7 package array

Parameter	Mandatory	Type	Description
package_name	Yes	String	Dependency package name. Special characters (&!"<>=) are not allowed.
package_version	No	String	Dependency package version. If the dependency package does not rely on the version number, leave this field blank. Special characters (&!"<>=) are not allowed.

Parameter	Mandatory	Type	Description
restraint	No	String	<p>Version restriction. This parameter is mandatory only when package_version is configured. Possible values are EXACT, ATLEAST, and ATMOST.</p> <ul style="list-style-type: none"> • EXACT indicates that a specified version is installed. • ATLEAST indicates that the version of the installation package is not earlier than the specified version. • ATMOST indicates that the version of the installation package is not later than the specified version. <p>NOTE</p> <ul style="list-style-type: none"> • If there are specific requirements on the version, preferentially use EXACT. If EXACT conflicts with the system installation packages, you can select ATLEAST. • If there is no specific requirement on the version, retain only the package_name parameter and leave restraint and package_version blank.

Table 9-8 health data structure description

Parameter	Mandatory	Type	Description
url	Yes	String	Request URL of the health check interface
protocol	No	String	Request protocol of the health check interface. Only HTTP is supported.
initial_delay_seconds	No	String	After an instance is started, a health check starts after seconds configured in initial_delay_seconds .
timeout_seconds	No	String	Health check timeout

Example of the Object Detection Model Configuration File

The following code uses the TensorFlow engine as an example. You can modify the **model_type** parameter based on the actual engine type.

- Model input
Key: images
Value: image files

- Model output

```

{
  "detection_classes": [
    "face",
    "arm"
  ],
  "detection_boxes": [
    [
      33.6,
      42.6,
      104.5,
      203.4
    ],
    [
      103.1,
      92.8,
      765.6,
      945.7
    ]
  ],
  "detection_scores": [0.99, 0.73]
}

```

- Configuration file

```

{
  "model_type": "TensorFlow",
  "model_algorithm": "object_detection",
  "metrics": {
    "f1": 0.345294,
    "accuracy": 0.462963,
    "precision": 0.338977,
    "recall": 0.351852
  },
  "apis": [{
    "protocol": "http",
    "url": "/",
    "method": "post",
    "request": {
      "Content-type": "multipart/form-data",
      "data": {
        "type": "object",
        "properties": {
          "images": {
            "type": "file"
          }
        }
      }
    }
  ]
},
  "response": {
    "Content-type": "multipart/form-data",
    "data": {
      "type": "object",
      "properties": {
        "detection_classes": {
          "type": "array",
          "items": [{

```

```

        "type": "string"
    }
  },
  "detection_boxes": {
    "type": "array",
    "items": [{
      "type": "array",
      "minItems": 4,
      "maxItems": 4,
      "items": [{
        "type": "number"
      }]
    }]
  },
  "detection_scores": {
    "type": "array",
    "items": [{
      "type": "number"
    }]
  }
}
}
}
},
"dependencies": [{
  "installer": "pip",
  "packages": [{
    "restraint": "EXACT",
    "package_version": "1.15.0",
    "package_name": "numpy"
  },
  {
    "restraint": "EXACT",
    "package_version": "5.2.0",
    "package_name": "Pillow"
  }
]
}]
}
...

```

Example of the Image Classification Model Configuration File

The following code uses the TensorFlow engine as an example. You can modify the **model_type** parameter based on the actual engine type.

- Model input

Key: images

Value: image files

- Model output

```

...
{
  "predicted_label": "flower",
  "scores": [
    ["rose", 0.99],
    ["begonia", 0.01]
  ]
}
...

```

- Configuration file

```

...
{
  "model_type": "TensorFlow",
  "model_algorithm": "image_classification",
  "metrics": {

```

```

    "f1": 0.345294,
    "accuracy": 0.462963,
    "precision": 0.338977,
    "recall": 0.351852
  },
  "apis": [{
    "protocol": "http",
    "url": "/",
    "method": "post",
    "request": {
      "Content-type": "multipart/form-data",
      "data": {
        "type": "object",
        "properties": {
          "images": {
            "type": "file"
          }
        }
      }
    }
  }],
  "response": {
    "Content-type": "multipart/form-data",
    "data": {
      "type": "object",
      "properties": {
        "predicted_label": {
          "type": "string"
        },
        "scores": {
          "type": "array",
          "items": [{
            "type": "array",
            "minItems": 2,
            "maxItems": 2,
            "items": [
              {
                "type": "string"
              },
              {
                "type": "number"
              }
            ]
          }
        ]
      }
    }
  }
}],
  "dependencies": [{
    "installer": "pip",
    "packages": [{
      "restraint": "ATLEAST",
      "package_version": "1.15.0",
      "package_name": "numpy"
    },
    {
      "restraint": "",
      "package_version": "",
      "package_name": "Pillow"
    }
  ]
}]]
}

```


Example of the Predictive Analytics Model Configuration File

The following code uses the TensorFlow engine as an example. You can modify the **model_type** parameter based on the actual engine type.

- Model input

```
...
{
  "data": {
    "req_data": [
      {
        "buying_price": "high",
        "maint_price": "high",
        "doors": "2",
        "persons": "2",
        "lug_boot": "small",
        "safety": "low",
        "acceptability": "acc"
      },
      {
        "buying_price": "high",
        "maint_price": "high",
        "doors": "2",
        "persons": "2",
        "lug_boot": "small",
        "safety": "low",
        "acceptability": "acc"
      }
    ]
  }
}
...
```

- Model output

```
...
{
  "data": {
    "resp_data": [
      {
        "predict_result": "unacc"
      },
      {
        "predict_result": "unacc"
      }
    ]
  }
}
...
```

- Configuration file

```
...
{
  "model_type": "TensorFlow",
  "model_algorithm": "predict_analysis",
  "metrics": {
    "f1": 0.345294,
    "accuracy": 0.462963,
    "precision": 0.338977,
    "recall": 0.351852
  },
  "apis": [
    {
      "protocol": "http",
      "url": "/",
      "method": "post",
      "request": {
        "Content-type": "application/json",
        "data": {
          "type": "object",

```

```

"properties": {
  "data": {
    "type": "object",
    "properties": {
      "req_data": {
        "items": [
          {
            "type": "object",
            "properties": {
            }
          }
        ]
      },
      "type": "array"
    }
  }
},
"response": {
  "Content-type": "multipart/form-data",
  "data": {
    "type": "object",
    "properties": {
      "data": {
        "type": "object",
        "properties": {
          "resp_data": {
            "type": "array",
            "items": [
              {
                "type": "object",
                "properties": {
                }
              }
            ]
          }
        }
      }
    }
  }
},
"dependencies": [
  {
    "installer": "pip",
    "packages": [
      {
        "restraint": "EXACT",
        "package_version": "1.15.0",
        "package_name": "numpy"
      },
      {
        "restraint": "EXACT",
        "package_version": "5.2.0",
        "package_name": "Pillow"
      }
    ]
  }
]
}
...

```

Example of the Custom Image Model Configuration File

The model input and output are similar to those in [Example of the Object Detection Model Configuration File](#).

- If the input is an image, the request example is as follows.
In the example, a model prediction request containing the parameter **images** with the parameter type of **file** is received. For this example, the file upload

button is displayed on the inference page, and the inference is performed in file format.

```
{
  "Content-type": "multipart/form-data",
  "data": {
    "type": "object",
    "properties": {
      "images": {
        "type": "file"
      }
    }
  }
}
```

- If the input is JSON data, the request example is as follows.

In this example, the model prediction JSON request body is received. In the request, there is only one prediction request containing the parameter **input** with the parameter type of string. On the inference page, a text box is displayed for you to enter the prediction request.

```
{
  "Content-type": "application/json",
  "data": {
    "type": "object",
    "properties": {
      "input": {
        "type": "string"
      }
    }
  }
}
```

A complete request example is as follows:

```
{
  "model_algorithm": "image_classification",
  "model_type": "Image",

  "metrics": {
    "f1": 0.345294,
    "accuracy": 0.462963,
    "precision": 0.338977,
    "recall": 0.351852
  },
  "apis": [{
    "protocol": "http",
    "url": "/",
    "method": "post",
    "request": {
      "Content-type": "multipart/form-data",
      "data": {
        "type": "object",
        "properties": {
          "images": {
            "type": "file"
          }
        }
      }
    }
  ]
},
  "response": {
    "Content-type": "multipart/form-data",
    "data": {
      "type": "object",
      "required": [
        "predicted_label",
        "scores"
      ],
      "properties": {
```

```

    "predicted_label": {
      "type": "string"
    },
    "scores": {
      "type": "array",
      "items": [{
        "type": "array",
        "minItems": 2,
        "maxItems": 2,
        "items": [{
          "type": "string"
        },
        {
          "type": "number"
        }
      ]
    }
  ]
}

```

Example of the Machine Learning Model Configuration File

The following uses XGBoost as an example:

- Model input

```

{
  "data": {
    "req_data": [{
      "sepal_length": 5,
      "sepal_width": 3.3,
      "petal_length": 1.4,
      "petal_width": 0.2
    }, {
      "sepal_length": 5,
      "sepal_width": 2,
      "petal_length": 3.5,
      "petal_width": 1
    }, {
      "sepal_length": 6,
      "sepal_width": 2.2,
      "petal_length": 5,
      "petal_width": 1.5
    }
  ]
}

```

- Model output

```

{
  "data": {
    "resp_data": [{
      "predict_result": "Iris-setosa"
    }, {
      "predict_result": "Iris-versicolor"
    }
  ]
}

```

- Configuration file

```

{
  "model_type": "XGBoost",
  "model_algorithm": "xgboost_iris_test",
  "runtime": "python2.7",
  "metrics": {
    "f1": 0.345294,

```

```
"accuracy": 0.462963,  
"precision": 0.338977,  
"recall": 0.351852  
},  
"apis": [  
  {  
    "protocol": "http",  
    "url": "/",  
    "method": "post",  
    "request": {  
      "Content-type": "application/json",  
      "data": {  
        "type": "object",  
        "properties": {  
          "data": {  
            "type": "object",  
            "properties": {  
              "req_data": {  
                "items": [  
                  {  
                    "type": "object",  
                    "properties": {}  
                  }  
                ],  
                "type": "array"  
              }  
            }  
          }  
        }  
      }  
    }  
  },  
  "response": {  
    "Content-type": "applicaton/json",  
    "data": {  
      "type": "object",  
      "properties": {  
        "resp_data": {  
          "type": "array",  
          "items": [  
            {  
              "type": "object",  
              "properties": {  
                "predict_result": {  
                  "type": "number"  
                }  
              }  
            }  
          ]  
        }  
      }  
    }  
  }  
]  
}]  
}
```

Example of a Model Configuration File Using a Custom Dependency Package

The following example defines the NumPy 1.16.4 dependency environment.

```
{  
  "model_algorithm": "image_classification",  
  "model_type": "TensorFlow",  
  "runtime": "python3.6",  
  "apis": [{  
    "procotol": "http",  
    "url": "/",  
    "method": "post",
```

```

"request": {
  "Content-type": "multipart/form-data",
  "data": {
    "type": "object",
    "properties": {
      "images": {
        "type": "file"
      }
    }
  }
},
"response": {
  "Content-type": "applicaton/json",
  "data": {
    "type": "object",
    "properties": {
      "mnist_result": {
        "type": "array",
        "item": [{
          "type": "string"
        }]
      }
    }
  }
},
"metrics": {
  "f1": 0.124555,
  "recall": 0.171875,
  "precision": 0.0023493892851938493,
  "accuracy": 0.00746268656716417
},
"dependencies": [{
  "installer": "pip",
  "packages": [{
    "restraint": "EXACT",
    "package_version": "1.16.4",
    "package_name": "numpy"
  }]
}]
}

```

9.1.3 Specifications for Writing Model Inference Code

This section describes how to compile model inference code in ModelArts. The following also provides an example of inference code for the TensorFlow engine and an example of customizing inference logic in an inference script.

Specifications for Compiling Inference Code

1. In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

Table 9-9 Import statements of different types of parent model classes

Model Type	Parent Class	Import Statement
TensorFlow	TfServingBaseService	from model_service.tf-serving_model_service import TfServingBaseService

Model Type	Parent Class	Import Statement
MXNet	MXNetBaseService	from mms.model_service.mxnet_model_service import MXNetBaseService
PyTorch	PTServicingBaseService	from model_service.pytorch_model_service import PTServicingBaseService
Caffe	CaffeBaseService	from model_service.caffe_model_service import CaffeBaseService
MindSpore	SingleNodeService	from model_service.model_service import SingleNodeService

2. The following methods can be rewritten:

Table 9-10 Methods to be rewritten

Method	Description
<code>__init__(self, model_name, model_path)</code>	Initialization method, which is suitable for models created based on deep learning frameworks. Models and labels are loaded using this method. This method must be rewritten for models based on PyTorch and Caffe to implement the model loading logic.
<code>__init__(self, model_path)</code>	Initialization method, which is suitable for models created based on machine learning frameworks. The model path (self.model_path) is initialized using this method. In Spark_MLlib, this method also initializes SparkSession (self.spark).
<code>_preprocess(self, data)</code>	Preprocess method, which is called before an inference request and is used to convert the original request data of an API into the expected input data of a model
<code>_inference(self, data)</code>	Inference request method. You are advised not to rewrite the method because once the method is rewritten, the ModelArts built-in inference process will be overwritten and the custom inference logic will run.
<code>_postprocess(self, data)</code>	Postprocess method, which is called after an inference request is complete and is used to convert the model output to the API output

 **NOTE**

- You can choose to rewrite the preprocess and postprocess methods to implement preprocessing of the API input and postprocessing of the inference output.
- Rewriting the init method of the parent model class may cause an AI application to run abnormally.

- The attribute that can be used is the local path where the model resides. The attribute name is **self.model_path**. In addition, PySpark-based models can use **self.spark** to obtain the SparkSession object in **customize_service.py**.

NOTE

An absolute path is required for reading files in the inference code. You can obtain the local path of the model from the **self.model_path** attribute.

- When TensorFlow, Caffe, or MXNet is used, **self.model_path** indicates the path of the model file. See the following example:

```
# Store the label.json file in the model directory. The following information is read:  
with open(os.path.join(self.model_path, 'label.json')) as f:  
    self.label = json.load(f)
```
 - When PyTorch, Scikit_Learn, or PySpark is used, **self.model_path** indicates the path of the model file. See the following example:

```
# Store the label.json file in the model directory. The following information is read:  
dir_path = os.path.dirname(os.path.realpath(self.model_path))  
with open(os.path.join(dir_path, 'label.json')) as f:  
    self.label = json.load(f)
```
- data** imported through the API for pre-processing, actual inference request, and post-processing can be **multipart/form-data** or **application/json**.

– **multipart/form-data** request

```
curl -X POST \  
<modelarts-inference-endpoint> \  
-F image1=@cat.jpg \  
-F images2=@horse.jpg
```

The corresponding input data is as follows:

```
[  
  {  
    "image1":{  
      "cat.jpg":"<cat.jpg file io>"  
    }  
  },  
  {  
    "image2":{  
      "horse.jpg":"<horse.jpg file io>"  
    }  
  }  
]
```

– **application/json** request

```
curl -X POST \  
<modelarts-inference-endpoint> \  
-d '{  
  "images":"base64 encode image"  
}'
```

The corresponding input data is **python dict**.

```
{  
  "images":"base64 encode image"  
}
```

TensorFlow Inference Script Example

The following is an example of TensorFlow MnistService. For details about the inference code of other engines, see [PyTorch](#) and [Caffe](#).

- Inference code

```
from PIL import Image  
import numpy as np  
from model_service.tf-serving_model_service import TfServingBaseService
```



```
class MnistService(TfServingBaseService):  
  
    def _preprocess(self, data):  
        preprocessed_data = {}  
  
        for k, v in data.items():  
            for file_name, file_content in v.items():  
                image1 = Image.open(file_content)  
                image1 = np.array(image1, dtype=np.float32)  
                image1.resize((1, 784))  
                preprocessed_data[k] = image1  
  
        return preprocessed_data  
  
    def _postprocess(self, data):  
        infer_output = {}  
  
        for output_name, result in data.items():  
            infer_output["mnist_result"] = result[0].index(max(result[0]))  
  
        return infer_output
```

- Request
`curl -X POST \ Real-time service address \ -F images=@test.jpg`
- Response
`{"mnist_result": 7}`

The preceding code example resizes images imported to the user's form to adapt to the model input shape. The **32×32** image is read from the Pillow library and resized to **1×784** to match the model input. In subsequent processing, convert the model output into a list for the RESTful API to display.

XGBoost Inference Script Example

For details about the inference code of other machine learning engines, see [PySpark](#) and [Scikit Learn](#).

```
# coding:utf-8  
import collections  
import json  
import xgboost as xgb  
from model_service.python_model_service import XgSkServingBaseService  
  
class UserService(XgSkServingBaseService):  
  
    # request data preprocess  
    def _preprocess(self, data):  
        list_data = []  
        json_data = json.loads(data, object_pairs_hook=collections.OrderedDict)  
        for element in json_data["data"]["req_data"]:  
            array = []  
            for each in element:  
                array.append(element[each])  
            list_data.append(array)  
        return list_data  
  
    # predict  
    def _inference(self, data):  
        xg_model = xgb.Booster(model_file=self.model_path)  
        pre_data = xgb.DMatrix(data)  
        pre_result = xg_model.predict(pre_data)  
        pre_result = pre_result.tolist()  
        return pre_result
```

```
# predict result process
def _postprocess(self, data):
    resp_data = []
    for element in data:
        resp_data.append({"predict_result": element})
    return resp_data
```

Inference Script Example of the Custom Inference Logic

First, define a dependency package in the configuration file. For details, see [Example of a Model Configuration File Using a Custom Dependency Package](#). Then, use the following code example to implement the loading and inference of the model in **saved_model** format.

```
# -*- coding: utf-8 -*-
import json
import os
import threading

import numpy as np
import tensorflow as tf
from PIL import Image

from model_service.tferving_model_service import TfServingBaseService
import logging

logger = logging.getLogger(__name__)

class MnistService(TfServingBaseService):

    def __init__(self, model_name, model_path):
        self.model_name = model_name
        self.model_path = model_path
        self.model_inputs = {}
        self.model_outputs = {}

        # The label file can be loaded here and used in the post-processing function.
        # Directories for storing the label.txt file on OBS and in the model package

        # with open(os.path.join(self.model_path, 'label.txt')) as f:
        #     self.label = json.load(f)

        # Load the model in saved_model format in non-blocking mode to prevent blocking timeout.
        thread = threading.Thread(target=self.get_tf_sess)
        thread.start()

    def get_tf_sess(self):
        # Load the model in saved_model format.

        # The session will be reused. Do not use the with statement.
        sess = tf.Session(graph=tf.Graph())
        meta_graph_def = tf.saved_model.loader.load(sess, [tf.saved_model.tag_constants.SERVING],
self.model_path)
        signature_defs = meta_graph_def.signature_def

        self.sess = sess

        signature = []

        # only one signature allowed
        for signature_def in signature_defs:
            signature.append(signature_def)
        if len(signature) == 1:
            model_signature = signature[0]
        else:
            logger.warning("signatures more than one, use serving_default signature")
            model_signature = tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY
```

```
logger.info("model signature: %s", model_signature)

for signature_name in meta_graph_def.signature_def[model_signature].inputs:
    tensorinfo = meta_graph_def.signature_def[model_signature].inputs[signature_name]
    name = tensorinfo.name
    op = self.sess.graph.get_tensor_by_name(name)
    self.model_inputs[signature_name] = op

logger.info("model inputs: %s", self.model_inputs)

for signature_name in meta_graph_def.signature_def[model_signature].outputs:
    tensorinfo = meta_graph_def.signature_def[model_signature].outputs[signature_name]
    name = tensorinfo.name
    op = self.sess.graph.get_tensor_by_name(name)

    self.model_outputs[signature_name] = op

logger.info("model outputs: %s", self.model_outputs)

def _preprocess(self, data):
    # Two request modes using HTTPS
    # 1. The request in form-data file format is as follows: data = {"Request key value":{"File name":<File io>}}
    # 2. Request in JSON format is as follows: data = json.loads("JSON body transferred by the API")
    preprocessed_data = {}

    for k, v in data.items():
        for file_name, file_content in v.items():
            image1 = Image.open(file_content)
            image1 = np.array(image1, dtype=np.float32)
            image1.resize((1, 28, 28))
            preprocessed_data[k] = image1

    return preprocessed_data

def _inference(self, data):
    feed_dict = {}
    for k, v in data.items():
        if k not in self.model_inputs.keys():
            logger.error("input key %s is not in model inputs %s", k, list(self.model_inputs.keys()))
            raise Exception("input key %s is not in model inputs %s" % (k, list(self.model_inputs.keys())))
        feed_dict[self.model_inputs[k]] = v

    result = self.sess.run(self.model_outputs, feed_dict=feed_dict)
    logger.info('predict result : ' + str(result))

    return result

def _postprocess(self, data):
    infer_output = {"mnist_result": []}
    for output_name, results in data.items():

        for result in results:
            infer_output["mnist_result"].append(np.argmax(result))

    return infer_output

def __del__(self):
    self.sess.close()
```

MindSpore Inference Script Example

```
import threading

import mindspore
import mindspore.nn as nn
import numpy as np
```

```
import logging
from mindspore import Tensor, context
from mindspore.common.initializer import Normal
from mindspore.train.serialization import load_checkpoint, load_param_into_net
from model_service.model_service import SingleNodeService
from PIL import Image

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

context.set_context(mode=context.GRAPH_MODE, device_target="Ascend")

class LeNet5(nn.Cell):
    """Lenet network structure."""

    # define the operator required
    def __init__(self, num_class=10, num_channel=1):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(num_channel, 6, 5, pad_mode='valid')
        self.conv2 = nn.Conv2d(6, 16, 5, pad_mode='valid')
        self.fc1 = nn.Dense(16 * 5 * 5, 120, weight_init=Normal(0.02))
        self.fc2 = nn.Dense(120, 84, weight_init=Normal(0.02))
        self.fc3 = nn.Dense(84, num_class, weight_init=Normal(0.02))
        self.relu = nn.ReLU()
        self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()

    # use the preceding operators to construct networks
    def construct(self, x):
        x = self.max_pool2d(self.relu(self.conv1(x)))
        x = self.max_pool2d(self.relu(self.conv2(x)))
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class MnistService(SingleNodeService):
    def __init__(self, model_name, model_path):
        self.model_name = model_name
        self.model_path = model_path
        logger.info("self.model_name:%s self.model_path: %s", self.model_name,
                    self.model_path)
        self.network = None
        # Load the model in non-blocking mode to prevent blocking timeout.
        thread = threading.Thread(target=self.load_model)
        thread.start()

    def load_model(self):
        logger.info("load network ... \n")
        self.network = LeNet5()
        ckpt_file = self.model_path + "/checkpoint_lenet_1-1_1875.ckpt"
        logger.info("ckpt_file: %s", ckpt_file)
        param_dict = load_checkpoint(ckpt_file)
        load_param_into_net(self.network, param_dict)
        # Inference warm-up. Otherwise, the initial inference will take a long time.
        self.network_warmup()
        logger.info("load network successfully ! \n")

    def network_warmup(self):
        # Inference warm-up. Otherwise, the initial inference will take a long time.
        logger.info("warmup network ... \n")
        images = np.array(np.random.randn(1, 1, 32, 32), dtype=np.float32)
        inputs = Tensor(images, mindspore.float32)
        inference_result = self.network(inputs)
```

```
logger.info("warmup network successfully ! \n")

def _preprocess(self, input_data):
    preprocessed_result = {}
    images = []
    for k, v in input_data.items():
        for file_name, file_content in v.items():
            image1 = Image.open(file_content)
            image1 = image1.resize((1, 32 * 32))
            image1 = np.array(image1, dtype=np.float32)
            images.append(image1)

    images = np.array(images, dtype=np.float32)
    logger.info(images.shape)
    images.resize([len(input_data), 1, 32, 32])
    logger.info("images shape: %s", images.shape)
    inputs = Tensor(images, mindspore.float32)
    preprocessed_result['images'] = inputs

    return preprocessed_result

def _inference(self, preprocessed_result):
    inference_result = self.network(preprocessed_result['images'])
    return inference_result

def _postprocess(self, inference_result):
    return str(inference_result)
```

9.2 Examples of Custom Scripts

9.2.1 TensorFlow

TensorFlow has two types of APIs: Keras and tf. Keras and tf use different code for training and saving models, but the same code for inference.

Training a Model (Keras API)

```
from keras.models import Sequential
model = Sequential()
from keras.layers import Dense
import tensorflow as tf

# Import a training dataset.
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

print(x_train.shape)

from keras.layers import Dense
from keras.models import Sequential
import keras
from keras.layers import Dense, Activation, Flatten, Dropout

# Define a model network.
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(units=5120,activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(units=10, activation='softmax'))

# Define an optimizer and loss functions.
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
```

```
metrics=['accuracy'])

model.summary()
# Train the model.
model.fit(x_train, y_train, epochs=2)
# Evaluate the model.
model.evaluate(x_test, y_test)
```

Saving a Model (Keras API)

```
from keras import backend as K

# K.get_session().run(tf.global_variables_initializer())

# Define the inputs and outputs of the prediction API.
# The key values of the inputs and outputs dictionaries are used as the index keys for the input and output tensors of the model.
# The input and output definitions of the model must match the custom inference script.
predict_signature = tf.saved_model.signature_def_utils.predict_signature_def(
    inputs={"images" : model.input},
    outputs={"scores" : model.output}
)

# Define a save path.
builder = tf.saved_model.builder.SavedModelBuilder('./mnist_keras/')

builder.add_meta_graph_and_variables(

    sess = K.get_session(),
    # The tf.saved_model.tag_constants.SERVING tag needs to be defined for inference and deployment.
    tags=[tf.saved_model.tag_constants.SERVING],
    """
signature_def_map: Only single items can exist, or the corresponding key needs to be defined as follows:
    """
    signature_def_map={
        tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
            predict_signature
    }
)
builder.save()
```

Training a Model (tf API)

```
from __future__ import print_function

import gzip
import os
import urllib

import numpy
import tensorflow as tf
from six.moves import urllib

# Training data is obtained from the Yann LeCun official website http://yann.lecun.com/exdb/mnist/.
SOURCE_URL = 'http://yann.lecun.com/exdb/mnist/'
TRAIN_IMAGES = 'train-images-idx3-ubyte.gz'
TRAIN_LABELS = 'train-labels-idx1-ubyte.gz'
TEST_IMAGES = 't10k-images-idx3-ubyte.gz'
TEST_LABELS = 't10k-labels-idx1-ubyte.gz'
VALIDATION_SIZE = 5000

def maybe_download(filename, work_directory):
    """Download the data from Yann's website, unless it's already here."""
    if not os.path.exists(work_directory):
        os.mkdir(work_directory)
    filepath = os.path.join(work_directory, filename)
```

```
if not os.path.exists(filepath):
    filepath, _ = urllib.request.urlretrieve(SOURCE_URL + filename, filepath)
    statinfo = os.stat(filepath)
    print('Successfully downloaded %s %d bytes.' % (filename, statinfo.st_size))
    return filepath

def _read32(bytestream):
    dt = numpy.dtype(numpy.uint32).newbyteorder('>')
    return numpy.frombuffer(bytestream.read(4), dtype=dt)[0]

def extract_images(filename):
    """Extract the images into a 4D uint8 numpy array [index, y, x, depth]."""
    print('Extracting %s' % filename)
    with gzip.open(filename) as bytestream:
        magic = _read32(bytestream)
        if magic != 2051:
            raise ValueError(
                'Invalid magic number %d in MNIST image file: %s' %
                (magic, filename))
        num_images = _read32(bytestream)
        rows = _read32(bytestream)
        cols = _read32(bytestream)
        buf = bytestream.read(rows * cols * num_images)
        data = numpy.frombuffer(buf, dtype=numpy.uint8)
        data = data.reshape(num_images, rows, cols, 1)
        return data

def dense_to_one_hot(labels_dense, num_classes=10):
    """Convert class labels from scalars to one-hot vectors."""
    num_labels = labels_dense.shape[0]
    index_offset = numpy.arange(num_labels) * num_classes
    labels_one_hot = numpy.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
    return labels_one_hot

def extract_labels(filename, one_hot=False):
    """Extract the labels into a 1D uint8 numpy array [index]."""
    print('Extracting %s' % filename)
    with gzip.open(filename) as bytestream:
        magic = _read32(bytestream)
        if magic != 2049:
            raise ValueError(
                'Invalid magic number %d in MNIST label file: %s' %
                (magic, filename))
        num_items = _read32(bytestream)
        buf = bytestream.read(num_items)
        labels = numpy.frombuffer(buf, dtype=numpy.uint8)
        if one_hot:
            return dense_to_one_hot(labels)
        return labels

class DataSet(object):
    """Class encompassing test, validation and training MNIST data set."""

    def __init__(self, images, labels, fake_data=False, one_hot=False):
        """Construct a DataSet. one_hot arg is used only if fake_data is true."""

        if fake_data:
            self.num_examples = 10000
            self.one_hot = one_hot
        else:
            assert images.shape[0] == labels.shape[0], (
                'images.shape: %s labels.shape: %s' % (images.shape,
                                                         labels.shape))
```

```
self._num_examples = images.shape[0]

# Convert shape from [num examples, rows, columns, depth]
# to [num examples, rows*columns] (assuming depth == 1)
assert images.shape[3] == 1
images = images.reshape(images.shape[0],
                        images.shape[1] * images.shape[2])
# Convert from [0, 255] -> [0.0, 1.0].
images = images.astype(numpy.float32)
images = numpy.multiply(images, 1.0 / 255.0)
self._images = images
self._labels = labels
self._epochs_completed = 0
self._index_in_epoch = 0

@property
def images(self):
    return self._images

@property
def labels(self):
    return self._labels

@property
def num_examples(self):
    return self._num_examples

@property
def epochs_completed(self):
    return self._epochs_completed

def next_batch(self, batch_size, fake_data=False):
    """Return the next `batch_size` examples from this data set."""
    if fake_data:
        fake_image = [1] * 784
        if self.one_hot:
            fake_label = [1] + [0] * 9
        else:
            fake_label = 0
        return [fake_image for _ in range(batch_size)], [
            fake_label for _ in range(batch_size)
        ]
    start = self._index_in_epoch
    self._index_in_epoch += batch_size
    if self._index_in_epoch > self._num_examples:
        # Finished epoch
        self._epochs_completed += 1
        # Shuffle the data
        perm = numpy.arange(self._num_examples)
        numpy.random.shuffle(perm)
        self._images = self._images[perm]
        self._labels = self._labels[perm]
        # Start next epoch
        start = 0
        self._index_in_epoch = batch_size
        assert batch_size <= self._num_examples
    end = self._index_in_epoch
    return self._images[start:end], self._labels[start:end]

def read_data_sets(train_dir, fake_data=False, one_hot=False):
    """Return training, validation and testing data sets."""

    class DataSets(object):
        pass

    data_sets = DataSets()

    if fake_data:
```



```
data_sets.train = DataSet([], [], fake_data=True, one_hot=one_hot)
data_sets.validation = DataSet([], [], fake_data=True, one_hot=one_hot)
data_sets.test = DataSet([], [], fake_data=True, one_hot=one_hot)
return data_sets

local_file = maybe_download(TRAIN_IMAGES, train_dir)
train_images = extract_images(local_file)

local_file = maybe_download(TRAIN_LABELS, train_dir)
train_labels = extract_labels(local_file, one_hot=one_hot)

local_file = maybe_download(TEST_IMAGES, train_dir)
test_images = extract_images(local_file)

local_file = maybe_download(TEST_LABELS, train_dir)
test_labels = extract_labels(local_file, one_hot=one_hot)

validation_images = train_images[:VALIDATION_SIZE]
validation_labels = train_labels[:VALIDATION_SIZE]
train_images = train_images[VALIDATION_SIZE:]
train_labels = train_labels[VALIDATION_SIZE:]

data_sets.train = DataSet(train_images, train_labels)
data_sets.validation = DataSet(validation_images, validation_labels)
data_sets.test = DataSet(test_images, test_labels)
return data_sets

training_iteration = 1000

modelarts_example_path = './modelarts-mnist-train-save-deploy-example'

export_path = modelarts_example_path + '/model/'
data_path = './'

print('Training model...')
mnist = read_data_sets(data_path, one_hot=True)
sess = tf.InteractiveSession()
serialized_tf_example = tf.placeholder(tf.string, name='tf_example')
feature_configs = {'x': tf.FixedLenFeature(shape=[784], dtype=tf.float32), }
tf_example = tf.parse_example(serialized_tf_example, feature_configs)
x = tf.identity(tf_example['x'], name='x') # use tf.identity() to assign name
y_ = tf.placeholder('float', shape=[None, 10])
w = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
sess.run(tf.global_variables_initializer())
y = tf.nn.softmax(tf.matmul(x, w) + b, name='y')
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
values, indices = tf.nn.top_k(y, 10)
table = tf.contrib.lookup.index_to_string_table_from_tensor(
    tf.constant([str(i) for i in range(10)]))
prediction_classes = table.lookup(tf.to_int64(indices))
for _ in range(training_iteration):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, 'float'))
print('training accuracy %g' % sess.run(
    accuracy, feed_dict={
        x: mnist.test.images,
        y_: mnist.test.labels
    }))
print('Done training!')
```

Saving a Model (tf API)

```
# Export the model.
# The model needs to be saved using the saved_model API.
print('Exporting trained model to', export_path)
```

```
builder = tf.saved_model.builder.SavedModelBuilder(export_path)

tensor_info_x = tf.saved_model.utils.build_tensor_info(x)
tensor_info_y = tf.saved_model.utils.build_tensor_info(y)

# Define the inputs and outputs of the prediction API.
# The key values of the inputs and outputs dictionaries are used as the index keys for the input and output
tensors of the model.
# The input and output definitions of the model must match the custom inference script.
prediction_signature = (
    tf.saved_model.signature_def_utils.build_signature_def(
        inputs={'images': tensor_info_x},
        outputs={'scores': tensor_info_y},
        method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NAME))

legacy_init_op = tf.group(tf.tables_initializer(), name='legacy_init_op')
builder.add_meta_graph_and_variables(
    # Set tag to serve/tf.saved_model.tag_constants.SERVING.
    sess, [tf.saved_model.tag_constants.SERVING],
    signature_def_map={
        'predict_images':
            prediction_signature,
    },
    legacy_init_op=legacy_init_op)

builder.save()

print('Done exporting!')
```

Inference Code (Keras and tf APIs)

In the model inference code file `customize_service.py`, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

```
from PIL import Image
import numpy as np
from model_service.tf_serving_model_service import TfServingBaseService

class MnistService(TfServingBaseService):

    # Match the model input with the user's HTTPS API input during preprocessing.
    # The model input corresponding to the preceding training part is {"images":<array>}.
    def _preprocess(self, data):

        preprocessed_data = {}
        images = []
        # Iterate the input data.
        for k, v in data.items():
            for file_name, file_content in v.items():
                image1 = Image.open(file_content)
                image1 = np.array(image1, dtype=np.float32)
                image1.resize((1,784))
                images.append(image1)
        # Return the numpy array.
        images = np.array(images,dtype=np.float32)
        # Perform batch processing on multiple input samples and ensure that the shape is the same as that
inputted during training.
        images.resize((len(data), 784))
        preprocessed_data['images'] = images
        return preprocessed_data

    # Processing logic of the inference for invoking the parent class.

    # The output corresponding to model saving in the preceding training part is {"scores":<array>}.
    # Postprocess the HTTPS output.
```

```
def _postprocess(self, data):
    infer_output = {"mnist_result": []}
    # Iterate the model output.
    for output_name, results in data.items():
        for result in results:
            infer_output["mnist_result"].append(result.index(max(result)))
    return infer_output
```

9.2.2 PyTorch

Training a Model

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

# Define a network structure.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
    # The second dimension of the input must be 784.
        self.hidden1 = nn.Linear(784, 5120, bias=False)
        self.output = nn.Linear(5120, 10, bias=False)

    def forward(self, x):
        x = x.view(x.size()[0], -1)
        x = F.relu((self.hidden1(x)))
        x = F.dropout(x, 0.2)
        x = self.output(x)
        return F.log_softmax(x)

def train(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.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 10 == 0:
            print('Train Epoch: {} [{} / {}] ( {:.0f}%) \t Loss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

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() # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
            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)))

device = torch.device("cpu")
```

```
batch_size=64

kwargs={}

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('.', train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor()
        ])),
    batch_size=batch_size, shuffle=True, **kwargs)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('.', train=False, transform=transforms.Compose([
        transforms.ToTensor()
    ])),
    batch_size=1000, shuffle=True, **kwargs)

model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
optimizer = optim.Adam(model.parameters())

for epoch in range(1, 2 + 1):
    train(model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
```

Saving a Model

```
# The model must be saved using state_dict and can be deployed remotely.
torch.save(model.state_dict(), "pytorch_mnist/mnist_mlp.pt")
```

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

```
from PIL import Image
import log
from model_service.pytorch_model_service import PTServingBaseService
import torch.nn.functional as F

import torch.nn as nn
import torch
import json

import numpy as np

logger = log.getLogger(__name__)

import torchvision.transforms as transforms

# Define model preprocessing.
infer_transformation = transforms.Compose([
    transforms.Resize((28,28)),
    # Transform to a PyTorch tensor.
    transforms.ToTensor()
])

import os

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 tags.
self.label = [0,1,2,3,4,5,6,7,8,9]
# Labels can also be loaded by label file.
# Store the label.json file in the model directory. The following information is read:
dir_path = os.path.dirname(os.path.realpath(self.model_path))
with open(os.path.join(dir_path, 'label.json')) as f:
    self.label = json.load(f)

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

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

def _inference(self, data):

    result = {}
    for k, v in data.items():
        result[k] = self.model(v)

    return result

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden1 = nn.Linear(784, 5120, bias=False)
        self.output = nn.Linear(5120, 10, bias=False)

    def forward(self, x):
        x = x.view(x.size()[0], -1)
        x = F.relu((self.hidden1(x)))
        x = F.dropout(x, 0.2)
        x = self.output(x)
        return F.log_softmax(x)

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)
# Declare an inference mode.
model.eval()

return model
```

9.2.3 Caffe

Training and Saving a Model

lenet_train_test.prototxt file

```
name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
```

```
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

```
    }  
  }  
}  
layer {  
  name: "relu1"  
  type: "ReLU"  
  bottom: "ip1"  
  top: "ip1"  
}  
layer {  
  name: "ip2"  
  type: "InnerProduct"  
  bottom: "ip1"  
  top: "ip2"  
  param {  
    lr_mult: 1  
  }  
  param {  
    lr_mult: 2  
  }  
  inner_product_param {  
    num_output: 10  
    weight_filler {  
      type: "xavier"  
    }  
    bias_filler {  
      type: "constant"  
    }  
  }  
}  
layer {  
  name: "accuracy"  
  type: "Accuracy"  
  bottom: "ip2"  
  bottom: "label"  
  top: "accuracy"  
  include {  
    phase: TEST  
  }  
}  
layer {  
  name: "loss"  
  type: "SoftmaxWithLoss"  
  bottom: "ip2"  
  bottom: "label"  
  top: "loss"  
}
```

lenet_solver.prototxt file

```
# The train/test net protocol buffer definition  
net: "examples/mnist/lenet_train_test.prototxt"  
# test_iter specifies how many forward passes the test should carry out.  
# In the case of MNIST, we have test batch size 100 and 100 test iterations,  
# covering the full 10,000 testing images.  
test_iter: 100  
# Carry out testing every 500 training iterations.  
test_interval: 500  
# The base learning rate, momentum and the weight decay of the network.  
base_lr: 0.01  
momentum: 0.9  
weight_decay: 0.0005  
# The learning rate policy  
lr_policy: "inv"  
gamma: 0.0001  
power: 0.75  
# Display every 100 iterations  
display: 100  
# The maximum number of iterations  
max_iter: 1000
```



```
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: CPU
```

Train the model.

```
./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt
```

The **caffemodel** file is generated after model training. Rewrite the **lenet_train_test.prototxt** file to the **lenet_deploy.prototxt** file used for deployment by modifying input and output layers.

```
name: "LeNet"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 1 dim: 1 dim: 28 dim: 28 } }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 5
  }
}
```

```
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
```

```
name: "prob"  
type: "Softmax"  
bottom: "ip2"  
top: "prob"  
}
```

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

```
from model_service.caffe_model_service import CaffeBaseService  
  
import numpy as np  
  
import os, json  
  
import caffe  
  
from PIL import Image  
  
class LenetService(CaffeBaseService):  
  
    def __init__(self, model_name, model_path):  
        # Call the inference method of the parent class.  
        super(LenetService, self).__init__(model_name, model_path)  
  
        # Configure preprocessing information.  
        transformer = caffe.io.Transformer({'data': self.net.blobs['data'].data.shape})  
        # Transform to NCHW.  
        transformer.set_transpose('data', (2, 0, 1))  
        # Perform normalization.  
        transformer.set_raw_scale('data', 255.0)  
  
        # If the batch size is set to 1, inference is supported for only one image.  
        self.net.blobs['data'].reshape(1, 1, 28, 28)  
        self.transformer = transformer  
  
        # Define the class labels.  
        self.label = [0,1,2,3,4,5,6,7,8,9]  
  
    def _preprocess(self, data):  
  
        for k, v in data.items():  
            for file_name, file_content in v.items():  
                im = caffe.io.load_image(file_content, color=False)  
                # Pre-process the images.  
                self.net.blobs['data'].data[...] = self.transformer.preprocess('data', im)  
  
            return  
  
    def _postprocess(self, data):  
  
        data = data['prob'][0, :]  
        predicted = np.argmax(data)  
        predicted = {"predicted" : str(predicted) }  
  
        return predicted
```

9.2.4 XGBoost

Training and Saving a Model

```
import pandas as pd
import xgboost as xgb
from sklearn.model_selection import train_test_split

# Prepare training data and setting parameters
iris = pd.read_csv('/home/ma-user/work/iris.csv')
X = iris.drop(['variety'],axis=1)
y = iris[['variety']]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234565)
params = {
    'booster': 'gbtree',
    'objective': 'multi:softmax',
    'num_class': 3,
    'gamma': 0.1,
    'max_depth': 6,
    'lambda': 2,
    'subsample': 0.7,
    'colsample_bytree': 0.7,
    'min_child_weight': 3,
    'silent': 1,
    'eta': 0.1,
    'seed': 1000,
    'nthread': 4,
}
plst = params.items()
dtrain = xgb.DMatrix(X_train, y_train)
num_rounds = 500
model = xgb.train(plst, dtrain, num_rounds)
model.save_model('/tmp/xgboost.m')
```

Before training, download the **iris.csv** dataset, decompress it, and upload it to the **/home/ma-user/work/** directory of the notebook instance. Download the **iris.csv** dataset from <https://gist.github.com/netj/8836201>. For details about how to upload a file to a notebook instance, see [Uploading Files to JupyterLab](#).

After the model is saved, it must be uploaded to the OBS directory before being published. The **config.json** configuration and the **customize_service.py** inference code must be included during the publishing. For details about how to compile **config.json**, see [Specifications for Editing a Model Configuration File](#). For details about inference code, see [Inference Code](#).

Inference Code

In the model inference code file **customize_service.py**, add a child model class which inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

```
# coding:utf-8
import collections
import json
import xgboost as xgb
from model_service.python_model_service import XgSkIServingBaseService
class UserService(XgSkIServingBaseService):

    # request data preprocess
    def _preprocess(self, data):
        list_data = []
        json_data = json.loads(data, object_pairs_hook=collections.OrderedDict)
        for element in json_data["data"]["req_data"]:
            array = []
```

```
        for each in element:
            array.append(element[each])
        list_data.append(array)
    return list_data

# predict
def _inference(self, data):
    xg_model = xgb.Booster(model_file=self.model_path)
    pre_data = xgb.DMatrix(data)
    pre_result = xg_model.predict(pre_data)
    pre_result = pre_result.tolist()
    return pre_result

# predict result process
def _postprocess(self, data):
    resp_data = []
    for element in data:
        resp_data.append({"predictresult": element})
    return resp_data
```

9.2.5 PySpark

Training and Saving a Model

```
from pyspark.ml import Pipeline, PipelineModel
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import LogisticRegression

# Prepare training data using tuples.
# Prepare training data from a list of (label, features) tuples.
training = spark.createDataFrame([
    (1.0, Vectors.dense([0.0, 1.1, 0.1])),
    (0.0, Vectors.dense([2.0, 1.0, -1.0])),
    (0.0, Vectors.dense([2.0, 1.3, 1.0])),
    (1.0, Vectors.dense([0.0, 1.2, -0.5])), ["label", "features"])

# Create a training instance. The logistic regression algorithm is used for training.
# Create a LogisticRegression instance. This instance is an Estimator.
lr = LogisticRegression(maxIter=10, regParam=0.01)

# Train the logistic regression model.
# Learn a LogisticRegression model. This uses the parameters stored in lr.
model = lr.fit(training)

# Save the model to a local directory.
# Save model to local path.
model.save("/tmp/spark_model")
```

After the model is saved, it must be uploaded to the OBS directory before being published. The **config.json** configuration and the **customize_service.py** inference code must be included during the publishing. For details about how to compile **config.json**, see [Specifications for Editing a Model Configuration File](#). For details about inference code, see [Inference Code](#).

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

```
# coding:utf-8
import collections
import json
```

```
import traceback

import model_service.log as log
from model_service.spark_model_service import SparkServingBaseService
from pyspark.ml.classification import LogisticRegression

logger = log.getLogger(__name__)

class UserService(SparkServingBaseService):
    # Pre-process data.
    def _preprocess(self, data):
        logger.info("Begin to handle data from user data...")
        # Read data.
        req_json = json.loads(data, object_pairs_hook=collections.OrderedDict)
        try:
            # Convert data to the spark dataframe format.
            predict_spdf = self.spark.createDataFrame(pd.DataFrame(req_json["data"]["req_data"]))
        except Exception as e:
            logger.error("check your request data does meet the requirements ?")
            logger.error(traceback.format_exc())
            raise Exception("check your request data does meet the requirements ?")
        return predict_spdf

    # Perform model inference.
    def _inference(self, data):
        try:
            # Load a model file.
            predict_model = LogisticRegression.load(self.model_path)
            # Perform data inference.
            prediction_result = predict_model.transform(data)
        except Exception as e:
            logger.error(traceback.format_exc())
            raise Exception("Unable to load model and do dataframe transformation.")
        return prediction_result

    # Post-process data.
    def _postprocess(self, pre_data):
        logger.info("Get new data to respond...")
        predict_str = pre_data.toPandas().to_json(orient='records')
        predict_result = json.loads(predict_str)
        return predict_result
```

9.2.6 Scikit Learn

Training and Saving a Model

```
import json
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.externals import joblib
iris = pd.read_csv('/home/ma-user/work/iris.csv')
X = iris.drop(['variety'],axis=1)
y = iris[['variety']]
# Create a LogisticRegression instance and train model
logisticRegression = LogisticRegression(C=1000.0, random_state=0)
logisticRegression.fit(X,y)
# Save model to local path
joblib.dump(logisticRegression, '/tmp/sklearn.m')
```

Before training, download the **iris.csv** dataset, decompress it, and upload it to the **/home/ma-user/work/** directory of the notebook instance. Download the **iris.csv** dataset from <https://gist.github.com/netj/8836201>. For details about how to upload a file to a notebook instance, see [Uploading Files to JupyterLab](#).

After the model is saved, it must be uploaded to the OBS directory before being published. The **config.json** and **customize_service.py** files must be contained during publishing. For details about the definition method, see [Introduction to Model Package Specifications](#).

Inference Code

In the model inference code file **customize_service.py**, add a child model class which inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see [Table 9-9](#).

```
# coding:utf-8
import collections
import json
from sklearn.externals import joblib
from model_service.python_model_service import XgSkIServingBaseService

class UserService(XgSkIServingBaseService):

    # request data preprocess
    def _preprocess(self, data):
        list_data = []
        json_data = json.loads(data, object_pairs_hook=collections.OrderedDict)
        for element in json_data["data"]["req_data"]:
            array = []
            for each in element:
                array.append(element[each])
            list_data.append(array)
        return list_data

    # predict
    def _inference(self, data):
        sk_model = joblib.load(self.model_path)
        pre_result = sk_model.predict(data)
        pre_result = pre_result.tolist()
        return pre_result

    # predict result process
    def _postprocess(self,data):
        resp_data = []
        for element in data:
            resp_data.append({"predictresult": element})
        return resp_data
```