

Contents

[Azure Machine Learning Documentation \(v1\)](#)

[Switch to current version documentation](#)

[Overview \(v1\)](#)

[Architecture & terms](#)

[Migrate from ML Studio \(classic\)](#)

[Migration overview](#)

[Migrate datasets](#)

[Rebuild experiments](#)

[Rebuild web services](#)

[Migrate client applications](#)

[Rebuild Execute R Script components](#)

[Concepts \(v1\)](#)

[Model training](#)

[Work with data](#)

[Data access](#)

[Studio network data access](#)

[Automated ML overview](#)

[Manage the ML lifecycle \(MLOps\)](#)

[MLOps capabilities](#)

[MLflow](#)

[Manage resources VS Code](#)

[Tutorials \(v1\)](#)

[Python get started \(Day 1\)](#)

[1. Run a Python script](#)

[2. Train your model](#)

[3. Use your own data](#)

[Train & deploy image classification](#)

[Build a training pipeline \(Python\)](#)

[Microsoft Power BI integration](#)

[Part 1: Train and deploy models](#)

[Part 2: Consume in Power BI](#)

[Samples \(v1\)](#)

[Jupyter Notebooks](#)

[Examples repository](#)

[How-to guides \(v1\)](#)

[Install and set up the CLI \(v1\)](#)

[Manage workspace using CLI \(v1\)](#)

[Set up software environments CLI \(v1\)](#)

[Create & manage compute resources](#)

[Workspace Diagnostics](#)

[Compute instance](#)

[Compute cluster](#)

[Azure Kubernetes Service](#)

[Security](#)

[Use managed identities for access control](#)

[Authenticate from the SDK v1](#)

[Network security](#)

[Network security overview \(v1\)](#)

[Use private endpoint \(v1\)](#)

[Secure training environment \(v1\)](#)

[Secure inferencing environment \(v1\)](#)

[Configure secure web services \(v1\)](#)

[Work with data](#)

[Access data](#)

[Connect to Azure storage with datastores](#)

[Identity-based data access to storage](#)

[Get data from storage with datasets](#)

[Connect to data \(UI\)](#)

[Manage & consume data](#)

[Train with datasets](#)

[Detect drift on datasets](#)

- [Version & track datasets](#)
- [Create datasets with labels](#)
- [Get & prepare data](#)
 - [Data ingestion with Azure Data Factory](#)
 - [Data preparation with Azure Synapse](#)
 - [DevOps for data ingestion](#)
 - [Import data in the designer](#)
- [Compliance](#)
 - [Export and delete data](#)
- [Train models](#)
 - [Configure & submit training run](#)
 - [Train with the Python SDK](#)
 - [Train with SDK v1](#)
 - [Distributed GPU guide](#)
 - [Scikit-learn](#)
 - [TensorFlow](#)
 - [Keras](#)
 - [PyTorch](#)
 - [Migrate from Estimators to ScriptRunConfig](#)
 - [Use Key Vault when training](#)
 - [Automated machine learning](#)
 - [Use automated ML \(Python\)](#)
 - [Auto-train a regression \(NYC Taxi data\)](#)
 - [Auto-train object detection model](#)
 - [Auto-train a natural language processing model](#)
 - [Set up AutoML to train computer vision models with Python](#)
 - [Local inference using ONNX](#)
 - [Track experiments with MLflow](#)
 - [Log & view metrics](#)
 - [Interpret ML models](#)
 - [Assess and mitigate model fairness](#)
 - [Prepare data for computer vision with AutoML](#)

[Reinforcement learning](#)

[Deploy models](#)

[Where and how to deploy](#)

[Hyperparameter tuning a model \(v1\)](#)

[Azure Kubernetes Service](#)

[Azure Container Instances](#)

[Deploy MLflow models](#)

[Update web service](#)

[Deployment targets](#)

[Deploy locally](#)

[Deploy locally on compute instance](#)

[GPU inference](#)

[Azure Cognitive Search](#)

[FPGA inference](#)

[Profile models](#)

[Troubleshoot deployment](#)

[Troubleshoot deployment \(local\)](#)

[Authenticate web service](#)

[Consume web service](#)

[Advanced entry script authoring](#)

[Prebuilt Docker images](#)

[Python extensibility](#)

[Dockerfile extensibility](#)

[Troubleshoot prebuilt docker images](#)

[Monitor web services](#)

[Collect & evaluate model data](#)

[Monitor with Application Insights](#)

[Deploy designer models](#)

[Package models](#)

[Manage the ML lifecycle](#)

[Build & use ML pipelines](#)

[Create ML pipelines \(Python\)](#)

- Moving data into and between ML pipeline steps (Python)
- Use automated ML in ML pipelines (Python)
- Deploy ML pipelines (Python)
- Use Azure Synapse Apache spark pools in an ML pipeline (Python)
- Trigger a pipeline
- Convert notebook code into Python scripts
- Troubleshoot & debug (v1)
 - Pipeline issues
 - Troubleshoot pipelines
 - Log pipeline data to Application Insights
 - Troubleshoot the ParallelRunStep
- Reference (v1)
 - Machine learning CLI pipeline YAML reference
 - Hyperparameters for AutoML computer vision tasks (v1)
 - Image data schemas for AutoML (v1)

Azure Machine Learning SDK & CLI (v1)

9/21/2022 • 2 minutes to read • [Edit Online](#)

APPLIES TO: Azure CLI ml extension v1 Python SDK azureml v1

All articles in this section document the use of the first version of Azure Machine Learning Python SDK (v1) or Azure CLI ml extension (v1).

SDK v1

The Azure SDK examples in articles in this section require the `azureml-core`, or Python SDK v1 for Azure Machine Learning. The Python SDK v2 is now available in preview.

The v1 and v2 Python SDK packages are incompatible, and v2 style of coding will not work for articles in this directory. However, machine learning workspaces and all underlying resources can be interacted with from either, meaning one user can create a workspace with the SDK v1 and another can submit jobs to the same workspace with the SDK v2.

We recommend not to install both versions of the SDK on the same environment, since it can cause clashes and confusion in the code.

How do I know which SDK version I have?

- To find out whether you have Azure ML Python SDK v1, run `pip show azureml-core`. (Or, in a Jupyter notebook, use `%pip show azureml-core`)
- To find out whether you have Azure ML Python SDK v2, run `pip show azure-ai-ml`. (Or, in a Jupyter notebook, use `%pip show azure-ai-ml`)

Based on the results of `pip show` you can determine which version of SDK you have.

CLI v1

The Azure CLI commands in articles in this section **require** the `azure-cli-ml`, or v1, extension for Azure Machine Learning. The enhanced v2 CLI using the `ml` extension is now available and recommended.

The extensions are incompatible, so v2 CLI commands will not work for articles in this directory. However, machine learning workspaces and all underlying resources can be interacted with from either, meaning one user can create a workspace with the v1 CLI and another can submit jobs to the same workspace with the v2 CLI.

How do I know which CLI extension I have?

To find which extensions you have installed, use `az extension list`.

- If the list of **Extensions** contains `azure-cli-ml`, you have the v1 extension.
- If the list contains `ml`, you have the v2 extension.

Next steps

For more information on installing and using the different extensions, see the following articles:

- `azure-cli-ml` - [Install, set up, and use the CLI \(v1\)](#)

- `ml` - [Install and set up the CLI \(v2\)](#)

For more information on installing and using the different SDK versions:

- `azureml-core` - [Install the Azure Machine Learning SDK \(v1\) for Python](#)
- `azure-ai-ml` - [Install the Azure Machine Learning SDK \(v2\) for Python](#)

How Azure Machine Learning works: Architecture and concepts (v1)

9/21/2022 • 13 minutes to read • [Edit Online](#)

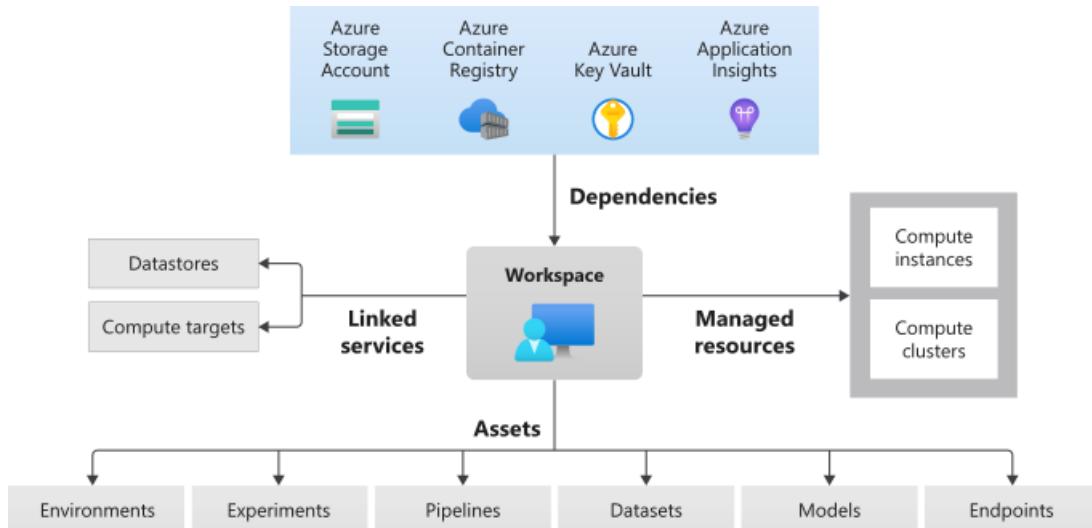
APPLIES TO: Azure CLI ml extension v1 Python SDK azureml v1

This article applies to the first version (v1) of the Azure Machine Learning CLI & SDK. For version two (v2), see [How Azure Machine Learning works \(v2\)](#).

Learn about the architecture and concepts for [Azure Machine Learning](#). This article gives you a high-level understanding of the components and how they work together to assist in the process of building, deploying, and maintaining machine learning models.

Workspace

A [machine learning workspace](#) is the top-level resource for Azure Machine Learning.



The workspace is the centralized place to:

- Manage resources you use for training and deployment of models, such as [computes](#)
- Store assets you create when you use Azure Machine Learning, including:
 - [Environments](#)
 - [Experiments](#)
 - [Pipelines](#)
 - [Datasets](#)
 - [Models](#)
 - [Endpoints](#)

A workspace includes other Azure resources that are used by the workspace:

- [Azure Container Registry \(ACR\)](#): Registers docker containers that you use during training and when you deploy a model. To minimize costs, ACR is only created when deployment images are created.
- [Azure Storage account](#): Is used as the default datastore for the workspace. Jupyter notebooks that are used with your Azure Machine Learning compute instances are stored here as well.
- [Azure Application Insights](#): Stores monitoring information about your models.

- [Azure Key Vault](#): Stores secrets that are used by compute targets and other sensitive information that's needed by the workspace.

You can share a workspace with others.

Computes

A [compute target](#) is any machine or set of machines you use to run your training script or host your service deployment. You can use your local machine or a remote compute resource as a compute target. With compute targets, you can start training on your local machine and then scale out to the cloud without changing your training script.

Azure Machine Learning introduces two fully managed cloud-based virtual machines (VM) that are configured for machine learning tasks:

- **Compute instance**: A compute instance is a VM that includes multiple tools and environments installed for machine learning. The primary use of a compute instance is for your development workstation. You can start running sample notebooks with no setup required. A compute instance can also be used as a compute target for training and inferencing jobs.
- **Compute clusters**: Compute clusters are a cluster of VMs with multi-node scaling capabilities. Compute clusters are better suited for compute targets for large jobs and production. The cluster scales up automatically when a job is submitted. Use as a training compute target or for dev/test deployment.

For more information about training compute targets, see [Training compute targets](#). For more information about deployment compute targets, see [Deployment targets](#).

Datasets and datastores

[Azure Machine Learning Datasets](#) make it easier to access and work with your data. By creating a dataset, you create a reference to the data source location along with a copy of its metadata. Because the data remains in its existing location, you incur no extra storage cost, and don't risk the integrity of your data sources.

For more information, see [Create and register Azure Machine Learning Datasets](#). For more examples using Datasets, see the [sample notebooks](#).

Datasets use [datastore](#) to securely connect to your Azure storage services. Datastores store connection information without putting your authentication credentials and the integrity of your original data source at risk. They store connection information, like your subscription ID and token authorization in your Key Vault associated with the workspace, so you can securely access your storage without having to hard code them in your script.

Environments

[Workspace](#) > [Environments](#)

An [environment](#) is the encapsulation of the environment where training or scoring of your machine learning model happens. The environment specifies the Python packages, environment variables, and software settings around your training and scoring scripts.

For code samples, see the "Manage environments" section of [How to use environments](#).

Experiments

[Workspace](#) > [Experiments](#)

An experiment is a grouping of many runs from a specified script. It always belongs to a workspace. When you

submit a run, you provide an experiment name. Information for the run is stored under that experiment. If the name doesn't exist when you submit an experiment, a new experiment is automatically created.

For an example of using an experiment, see [Tutorial: Train your first model](#).

Runs

[Workspace](#) > [Experiments](#) > [Run](#)

A run is a single execution of a training script. An experiment will typically contain multiple runs.

Azure Machine Learning records all runs and stores the following information in the experiment:

- Metadata about the run (timestamp, duration, and so on)
- Metrics that are logged by your script
- Output files that are autocollected by the experiment or explicitly uploaded by you
- A snapshot of the directory that contains your scripts, prior to the run

You produce a run when you submit a script to train a model. A run can have zero or more child runs. For example, the top-level run might have two child runs, each of which might have its own child run.

Run configurations

[Workspace](#) > [Experiments](#) > [Run](#) > [Run configuration](#)

A run configuration defines how a script should be run in a specified compute target. You use the configuration to specify the script, the compute target and Azure ML environment to run on, any distributed job-specific configurations, and some additional properties. For more information on the full set of configurable options for runs, see [ScriptRunConfig](#).

A run configuration can be persisted into a file inside the directory that contains your training script. Or it can be constructed as an in-memory object and used to submit a run.

For example run configurations, see [Configure a training run](#).

Snapshots

[Workspace](#) > [Experiments](#) > [Run](#) > [Snapshot](#)

When you submit a run, Azure Machine Learning compresses the directory that contains the script as a zip file and sends it to the compute target. The zip file is then extracted, and the script is run there. Azure Machine Learning also stores the zip file as a snapshot as part of the run record. Anyone with access to the workspace can browse a run record and download the snapshot.

Logging

Azure Machine Learning automatically logs standard run metrics for you. However, you can also [use the Python SDK to log arbitrary metrics](#).

There are multiple ways to view your logs: monitoring run status in real time, or viewing results after completion. For more information, see [Monitor and view ML run logs](#).

NOTE

To prevent unnecessary files from being included in the snapshot, make an ignore file (`.gitignore` or `.amlignore`) in the directory. Add the files and directories to exclude to this file. For more information on the syntax to use inside this file, see [syntax and patterns](#) for `.gitignore`. The `.amlignore` file uses the same syntax. *If both files exist, the `.amlignore` file is used and the `.gitignore` file is unused.*

Git tracking and integration

When you start a training run where the source directory is a local Git repository, information about the

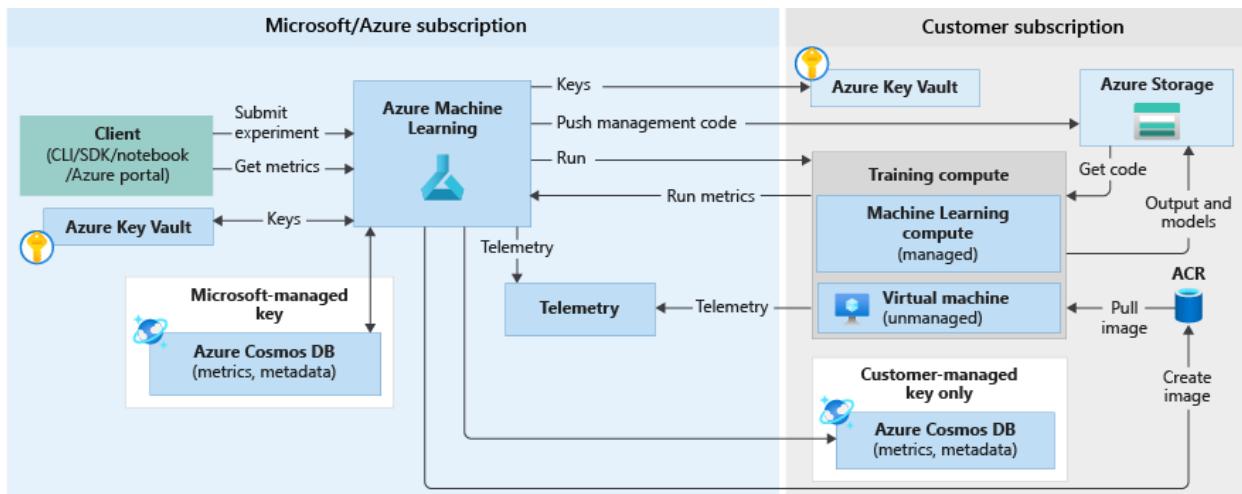
repository is stored in the run history. This works with runs submitted using a script run configuration or ML pipeline. It also works for runs submitted from the SDK or Machine Learning CLI.

For more information, see [Git integration for Azure Machine Learning](#).

Training workflow

When you run an experiment to train a model, the following steps happen. These are illustrated in the training workflow diagram below:

- Azure Machine Learning is called with the snapshot ID for the code snapshot saved in the previous section.
- Azure Machine Learning creates a run ID (optional) and a Machine Learning service token, which is later used by compute targets like Machine Learning Compute/VMs to communicate with the Machine Learning service.
- You can choose either a managed compute target (like Machine Learning Compute) or an unmanaged compute target (like VMs) to run training jobs. Here are the data flows for both scenarios:
 - VMs/HDIInsight, accessed by SSH credentials in a key vault in the Microsoft subscription. Azure Machine Learning runs management code on the compute target that:
 1. Prepares the environment. (Docker is an option for VMs and local computers. See the following steps for Machine Learning Compute to understand how running experiments on Docker containers works.)
 2. Downloads the code.
 3. Sets up environment variables and configurations.
 4. Runs user scripts (the code snapshot mentioned in the previous section).
 - Machine Learning Compute, accessed through a workspace-managed identity. Because Machine Learning Compute is a managed compute target (that is, it's managed by Microsoft) it runs under your Microsoft subscription.
 1. Remote Docker construction is kicked off, if needed.
 2. Management code is written to the user's Azure Files share.
 3. The container is started with an initial command. That is, management code as described in the previous step.
- After the run completes, you can query runs and metrics. In the flow diagram below, this step occurs when the training compute target writes the run metrics back to Azure Machine Learning from storage in the Cosmos DB database. Clients can call Azure Machine Learning. Machine Learning will in turn pull metrics from the Cosmos DB database and return them back to the client.



Models

At its simplest, a model is a piece of code that takes an input and produces output. Creating a machine learning model involves selecting an algorithm, providing it with data, and [tuning hyperparameters](#). Training is an iterative process that produces a trained model, which encapsulates what the model learned during the training process.

You can bring a model that was trained outside of Azure Machine Learning. Or you can train a model by submitting a [run](#) of an [experiment](#) to a [compute target](#) in Azure Machine Learning. Once you have a model, you [register the model](#) in the workspace.

Azure Machine Learning is framework agnostic. When you create a model, you can use any popular machine learning framework, such as Scikit-learn, XGBoost, PyTorch, TensorFlow, and Chainer.

For an example of training a model using Scikit-learn, see [Tutorial: Train an image classification model with Azure Machine Learning](#).

Model registry

[Workspace](#) > [Models](#)

The **model registry** lets you keep track of all the models in your Azure Machine Learning workspace.

Models are identified by name and version. Each time you register a model with the same name as an existing one, the registry assumes that it's a new version. The version is incremented, and the new model is registered under the same name.

When you register the model, you can provide additional metadata tags and then use the tags when you search for models.

TIP

A registered model is a logical container for one or more files that make up your model. For example, if you have a model that is stored in multiple files, you can register them as a single model in your Azure Machine Learning workspace. After registration, you can then download or deploy the registered model and receive all the files that were registered.

You can't delete a registered model that is being used by an active deployment.

For an example of registering a model, see [Train an image classification model with Azure Machine Learning](#).

Deployment

You deploy a [registered model](#) as a service endpoint. You need the following components:

- **Environment.** This environment encapsulates the dependencies required to run your model for inference.
- **Scoring code.** This script accepts requests, scores the requests by using the model, and returns the results.
- **Inference configuration.** The inference configuration specifies the environment, entry script, and other components needed to run the model as a service.

For more information about these components, see [Deploy models with Azure Machine Learning](#).

Endpoints

[Workspace](#) > [Endpoints](#)

An endpoint is an instantiation of your model into a web service that can be hosted in the cloud.

Web service endpoint

When deploying a model as a web service, the endpoint can be deployed on Azure Container Instances, Azure Kubernetes Service, or FPGAs. You create the service from your model, script, and associated files. These are placed into a base container image, which contains the execution environment for the model. The image has a

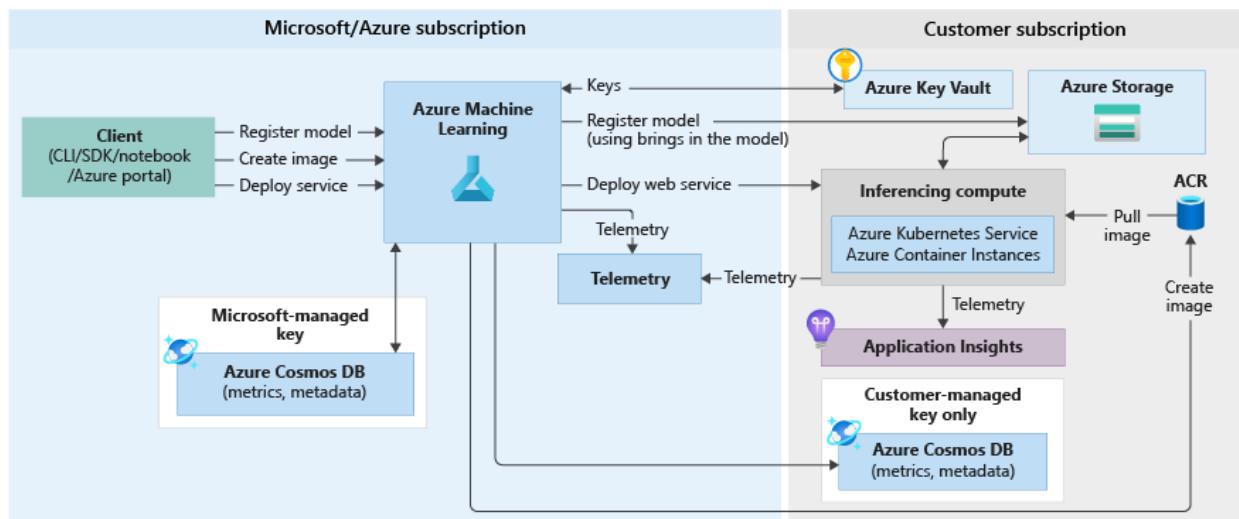
load-balanced, HTTP endpoint that receives scoring requests that are sent to the web service.

You can enable Application Insights telemetry or model telemetry to monitor your web service. The telemetry data is accessible only to you. It's stored in your Application Insights and storage account instances. If you've enabled automatic scaling, Azure automatically scales your deployment.

The following diagram shows the inference workflow for a model deployed as a web service endpoint:

Here are the details:

- The user registers a model by using a client like the Azure Machine Learning SDK.
- The user creates an image by using a model, a score file, and other model dependencies.
- The Docker image is created and stored in Azure Container Registry.
- The web service is deployed to the compute target (Container Instances/AKS) using the image created in the previous step.
- Scoring request details are stored in Application Insights, which is in the user's subscription.
- Telemetry is also pushed to the Microsoft Azure subscription.



For an example of deploying a model as a web service, see [Tutorial: Train and deploy a model](#).

Real-time endpoints

When you deploy a trained model in the designer, you can [deploy the model as a real-time endpoint](#). A real-time endpoint commonly receives a single request via the REST endpoint and returns a prediction in real-time. This is in contrast to batch processing, which processes multiple values at once and saves the results after completion to a datastore.

Pipeline endpoints

Pipeline endpoints let you call your [ML Pipelines](#) programmatically via a REST endpoint. Pipeline endpoints let you automate your pipeline workflows.

A pipeline endpoint is a collection of published pipelines. This logical organization lets you manage and call multiple pipelines using the same endpoint. Each published pipeline in a pipeline endpoint is versioned. You can select a default pipeline for the endpoint, or specify a version in the REST call.

Automation

Azure Machine Learning CLI

The [Azure Machine Learning CLI](#) is an extension to the Azure CLI, a cross-platform command-line interface for the Azure platform. This extension provides commands to automate your machine learning activities.

ML Pipelines

You use [machine learning pipelines](#) to create and manage workflows that stitch together machine learning

phases. For example, a pipeline might include data preparation, model training, model deployment, and inference/scoring phases. Each phase can encompass multiple steps, each of which can run unattended in various compute targets.

Pipeline steps are reusable, and can be run without rerunning the previous steps if the output of those steps hasn't changed. For example, you can retrain a model without rerunning costly data preparation steps if the data hasn't changed. Pipelines also allow data scientists to collaborate while working on separate areas of a machine learning workflow.

Monitoring and logging

Azure Machine Learning provides the following monitoring and logging capabilities:

- For **Data Scientists**, you can monitor your experiments and log information from your training runs. For more information, see the following articles:
 - [Start, monitor, and cancel training runs](#)
 - [Log metrics for training runs](#)
 - [Track experiments with MLflow](#)
 - [Visualize runs with TensorBoard](#)
- For **Administrators**, you can monitor information about the workspace, related Azure resources, and events such as resource creation and deletion by using Azure Monitor. For more information, see [How to monitor Azure Machine Learning](#).
- For **DevOps** or **MLOps**, you can monitor information generated by models deployed as web services to identify problems with the deployments and gather data submitted to the service. For more information, see [Collect model data and Monitor with Application Insights](#).

Interacting with your workspace

Studio

[Azure Machine Learning studio](#) provides a web view of all the artifacts in your workspace. You can view results and details of your datasets, experiments, pipelines, models, and endpoints. You can also manage compute resources and datastores in the studio.

The studio is also where you access the interactive tools that are part of Azure Machine Learning:

- [Azure Machine Learning designer](#) to perform workflow steps without writing code
- Web experience for [automated machine learning](#)
- [Azure Machine Learning notebooks](#) to write and run your own code in integrated Jupyter notebook servers.
- Data labeling projects to create, manage, and monitor projects for labeling [images](#) or [text](#).

Programming tools

IMPORTANT

Tools marked (preview) below are currently in public preview. The preview version is provided without a service level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

- Interact with the service in any Python environment with the [Azure Machine Learning SDK for Python](#).
- Use [Azure Machine Learning designer](#) to perform the workflow steps without writing code.
- Use [Azure Machine Learning CLI](#) for automation.

Next steps

To get started with Azure Machine Learning, see:

- [What is Azure Machine Learning?](#)
- [Create an Azure Machine Learning workspace](#)
- [Tutorial: Train and deploy a model](#)

Migrate to Azure Machine Learning from ML Studio (classic)

9/21/2022 • 9 minutes to read • [Edit Online](#)

IMPORTANT

Support for Machine Learning Studio (classic) will end on 31 August 2024. We recommend you transition to [Azure Machine Learning](#) by that date.

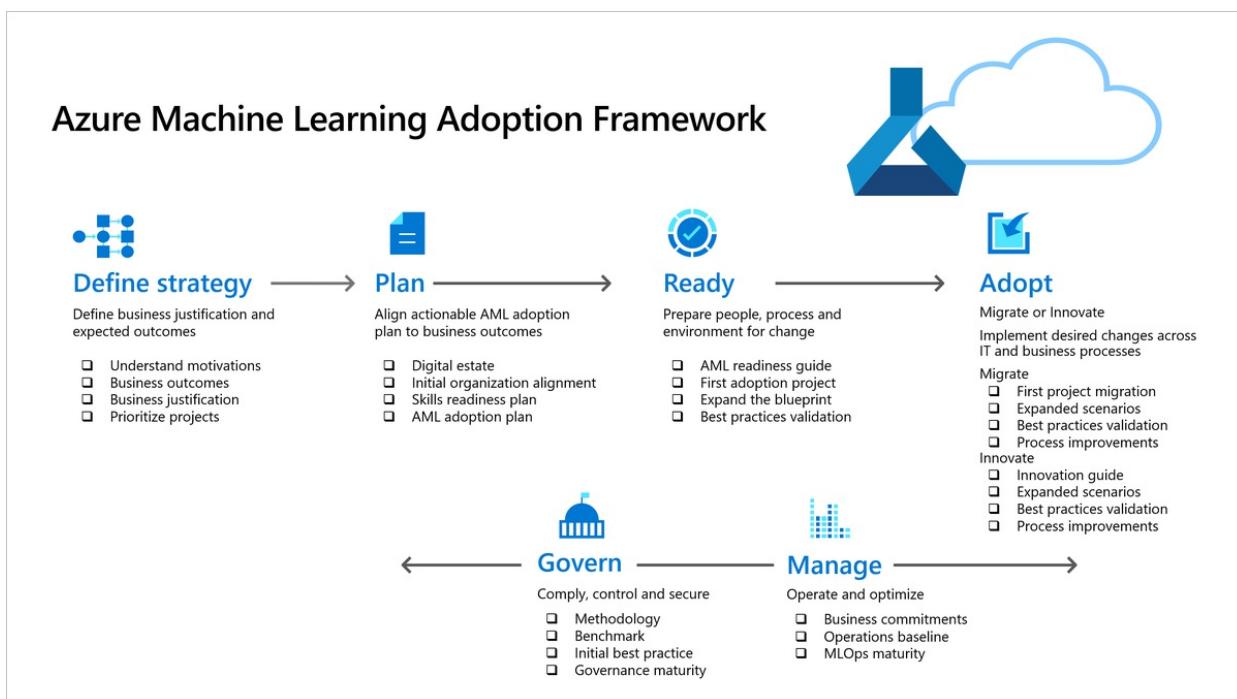
Beginning 1 December 2021, you will not be able to create new Machine Learning Studio (classic) resources. Through 31 August 2024, you can continue to use the existing Machine Learning Studio (classic) resources.

ML Studio (classic) documentation is being retired and may not be updated in the future.

Learn how to migrate from Studio (classic) to Azure Machine Learning. Azure Machine Learning provides a modernized data science platform that combines no-code and code-first approaches.

This is a guide for a basic "lift and shift" migration. If you want to optimize an existing machine learning workflow, or modernize a machine learning platform, see the [Azure Machine Learning adoption framework](#) for additional resources including digital survey tools, worksheets, and planning templates.

Please work with your Cloud Solution Architect on the migration.



Recommended approach

To migrate to Azure Machine Learning, we recommend the following approach:

- Step 1: Assess Azure Machine Learning
- Step 2: Define a strategy and plan
- Step 3: Rebuild experiments and web services
- Step 4: Integrate client apps

- Step 5: Clean up Studio (classic) assets
- Step 6: Review and expand scenarios

Step 1: Assess Azure Machine Learning

1. Learn about [Azure Machine Learning](#); its benefits, costs, and architecture.
2. Compare the capabilities of Azure Machine Learning and Studio (classic).

NOTE

The **designer** feature in Azure Machine Learning provides a similar drag-and-drop experience to Studio (classic). However, Azure Machine Learning also provides robust [code-first workflows](#) as an alternative. This migration series focuses on the designer, since it's most similar to the Studio (classic) experience.

The following table summarizes the key differences between ML Studio (classic) and Azure Machine Learning.

FEATURE	ML STUDIO (CLASSIC)	AZURE MACHINE LEARNING
Drag and drop interface	Classic experience	Updated experience - Azure Machine Learning designer
Code SDKs	Not supported	Fully integrated with Azure Machine Learning Python and R SDKs
Experiment	Scalable (10-GB training data limit)	Scale with compute target
Training compute targets	Proprietary compute target, CPU support only	Wide range of customizable training compute targets . Includes GPU and CPU support
Deployment compute targets	Proprietary web service format, not customizable	Wide range of customizable deployment compute targets . Includes GPU and CPU support
ML Pipeline	Not supported	Build flexible, modular pipelines to automate workflows
MLOps	Basic model management and deployment; CPU only deployments	Entity versioning (model, data, workflows), workflow automation, integration with CICD tooling, CPU and GPU deployments and more
Model format	Proprietary format, Studio (classic) only	Multiple supported formats depending on training job type
Automated model training and hyperparameter tuning	Not supported	Supported . Code-first and no-code options.
Data drift detection	Not supported	Supported
Data labeling projects	Not supported	Supported

FEATURE	ML STUDIO (CLASSIC)	AZURE MACHINE LEARNING
Role-Based Access Control (RBAC)	Only contributor and owner role	Flexible role definition and RBAC control
AI Gallery	Supported (https://gallery.azure.ai/)	Unsupported Learn with sample Python SDK notebooks .

3. Verify that your critical Studio (classic) modules are supported in Azure Machine Learning designer. For more information, see the [Studio \(classic\) and designer component-mapping](#) table below.
4. [Create an Azure Machine Learning workspace](#).

Step 2: Define a strategy and plan

1. Define business justifications and expected outcomes.
2. Align an actionable Azure Machine Learning adoption plan to business outcomes.
3. Prepare people, processes, and environments for change.

Please work with your Cloud Solution Architect to define your strategy.

See the [Azure Machine Learning Adoption Framework](#) for planning resources including a planning doc template.

Step 3: Rebuild your first model

After you've defined a strategy, migrate your first model.

1. [Migrate datasets to Azure Machine Learning](#).
2. Use the designer to [rebuild experiments](#).
3. Use the designer to [redeploy web services](#).

NOTE

Above guidance are built on top of AzureML v1 concepts and features. AzureML has CLI v2 and Python SDK v2. We suggest to rebuild your ML Studio(classic) models using v2 instead of v1. Start with AzureML v2 [here](#)

Step 4: Integrate client apps

1. Modify client applications that invoke Studio (classic) web services to use your new [Azure Machine Learning endpoints](#).

Step 5: Cleanup Studio (classic) assets

1. [Clean up Studio \(classic\) assets](#) to avoid extra charges. You may want to retain assets for fallback until you have validated Azure Machine Learning workloads.

Step 6: Review and expand scenarios

1. Review the model migration for best practices and validate workloads.
2. Expand scenarios and migrate additional workloads to Azure Machine Learning.

Studio (classic) and designer component-mapping

Consult the following table to see which modules to use while rebuilding Studio (classic) experiments in the designer.

IMPORTANT

The designer implements modules through open-source Python packages rather than C# packages like Studio (classic). Because of this difference, the output of designer components may vary slightly from their Studio (classic) counterparts.

CATEGORY	STUDIO (CLASSIC) MODULE	REPLACEMENT DESIGNER COMPONENT
Data input and output	<ul style="list-style-type: none">- Enter Data Manually- Export Data- Import Data- Load Trained Model- Unpack Zipped Datasets	<ul style="list-style-type: none">- Enter Data Manually- Export Data- Import Data
Data Format Conversions	<ul style="list-style-type: none">- Convert to CSV- Convert to Dataset- Convert to ARFF- Convert to SVMLight- Convert to TSV	<ul style="list-style-type: none">- Convert to CSV- Convert to Dataset
Data Transformation - Manipulation	<ul style="list-style-type: none">- Add Columns- Add Rows- Apply SQL Transformation- Cleaning Missing Data- Convert to Indicator Values- Edit Metadata- Join Data- Remove Duplicate Rows- Select Columns in Dataset- Select Columns Transform- SMOTE- Group Categorical Values	<ul style="list-style-type: none">- Add Columns- Add Rows- Apply SQL Transformation- Cleaning Missing Data- Convert to Indicator Values- Edit Metadata- Join Data- Remove Duplicate Rows- Select Columns in Dataset- Select Columns Transform- SMOTE
Data Transformation – Scale and Reduce	<ul style="list-style-type: none">- Clip Values- Group Data into Bins- Normalize Data- Principal Component Analysis	<ul style="list-style-type: none">- Clip Values- Group Data into Bins- Normalize Data
Data Transformation – Sample and Split	<ul style="list-style-type: none">- Partition and Sample- Split Data	<ul style="list-style-type: none">- Partition and Sample- Split Data
Data Transformation – Filter	<ul style="list-style-type: none">- Apply Filter- FIR Filter- IIR Filter- Median Filter- Moving Average Filter- Threshold Filter- User Defined Filter	

CATEGORY	STUDIO (CLASSIC) MODULE	REPLACEMENT DESIGNER COMPONENT
Data Transformation – Learning with Counts	<ul style="list-style-type: none"> - Build Counting Transform - Export Count Table - Import Count Table - Merge Count Transform - Modify Count Table Parameters 	
Feature Selection	<ul style="list-style-type: none"> - Filter Based Feature Selection - Fisher Linear Discriminant Analysis - Permutation Feature Importance 	<ul style="list-style-type: none"> - Filter Based Feature Selection - Permutation Feature Importance
Model - Classification	<ul style="list-style-type: none"> - Multiclass Decision Forest - Multiclass Decision Jungle - Multiclass Logistic Regression - Multiclass Neural Network - One-vs-All Multiclass - Two-Class Averaged Perceptron - Two-Class Bayes Point Machine - Two-Class Boosted Decision Tree - Two-Class Decision Forest - Two-Class Decision Jungle - Two-Class Locally-Deep SVM - Two-Class Logistic Regression - Two-Class Neural Network - Two-Class Support Vector Machine 	<ul style="list-style-type: none"> - Multiclass Decision Forest - Multiclass Boost Decision Tree - Multiclass Logistic Regression - Multiclass Neural Network - One-vs-All Multiclass - Two-Class Averaged Perceptron - Two-Class Boosted Decision Tree - Two-Class Decision Forest - Two-Class Logistic Regression - Two-Class Neural Network - Two-Class Support Vector Machine
Model - Clustering	<ul style="list-style-type: none"> - K-means clustering 	<ul style="list-style-type: none"> - K-means clustering
Model - Regression	<ul style="list-style-type: none"> - Bayesian Linear Regression - Boosted Decision Tree Regression - Decision Forest Regression - Fast Forest Quantile Regression - Linear Regression - Neural Network Regression - Ordinal Regression Poisson Regression 	<ul style="list-style-type: none"> - Boosted Decision Tree Regression - Decision Forest Regression - Fast Forest Quantile Regression - Linear Regression - Neural Network Regression - Poisson Regression
Model – Anomaly Detection	<ul style="list-style-type: none"> - One-Class SVM - PCA-Based Anomaly Detection 	<ul style="list-style-type: none"> - PCA-Based Anomaly Detection
Machine Learning – Evaluate	<ul style="list-style-type: none"> - Cross Validate Model - Evaluate Model - Evaluate Recommender 	<ul style="list-style-type: none"> - Cross Validate Model - Evaluate Model - Evaluate Recommender
Machine Learning – Train	<ul style="list-style-type: none"> - Sweep Clustering - Train Anomaly Detection Model - Train Clustering Model - Train Matchbox Recommender - Train Model - Tune Model Hyperparameters 	<ul style="list-style-type: none"> - Train Anomaly Detection Model - Train Clustering Model - Train Model - - Train PyTorch Model - Train SVD Recommender - Train Wide and Deep Recommender - Tune Model Hyperparameters
Machine Learning – Score	<ul style="list-style-type: none"> - Apply Transformation - Assign Data to clusters - Score Matchbox Recommender - Score Model 	<ul style="list-style-type: none"> - Apply Transformation - Assign Data to clusters - Score Image Model - Score Model - Score SVD Recommender - Score Wide and Deep Recommender

CATEGORY	STUDIO (CLASSIC) MODULE	REPLACEMENT DESIGNER COMPONENT
OpenCV Library Modules	<ul style="list-style-type: none"> - Import Images - Pre-trained Cascade Image Classification 	
Python Language Modules	<ul style="list-style-type: none"> - Execute Python Script 	<ul style="list-style-type: none"> - Execute Python Script - Create Python Model
R Language Modules	<ul style="list-style-type: none"> - Execute R Script - Create R Model 	<ul style="list-style-type: none"> - Execute R Script
Statistical Functions	<ul style="list-style-type: none"> - Apply Math Operation - Compute Elementary Statistics - Compute Linear Correlation - Evaluate Probability Function - Replace Discrete Values - Summarize Data - Test Hypothesis using t-Test 	<ul style="list-style-type: none"> - Apply Math Operation - Summarize Data
Text Analytics	<ul style="list-style-type: none"> - Detect Languages - Extract Key Phrases from Text - Extract N-Gram Features from Text - Feature Hashing - Latent Dirichlet Allocation - Named Entity Recognition - Preprocess Text - Score Vowpal Wabbit Version 7-10 Model - Score Vowpal Wabbit Version 8 Model - Train Vowpal Wabbit Version 7-10 Model - Train Vowpal Wabbit Version 8 Model 	<ul style="list-style-type: none"> - Convert Word to Vector - Extract N-Gram Features from Text - Feature Hashing - Latent Dirichlet Allocation - Preprocess Text - Score Vowpal Wabbit Model - Train Vowpal Wabbit Model
Time Series	<ul style="list-style-type: none"> - Time Series Anomaly Detection 	
Web Service	<ul style="list-style-type: none"> - Input - Output 	<ul style="list-style-type: none"> - Input - Output
Computer Vision		<ul style="list-style-type: none"> - Apply Image Transformation - Convert to Image Directory - Init Image Transformation - Split Image Directory - DenseNet Image Classification - ResNet Image Classification

For more information on how to use individual designer components, see the [designer component reference](#).

What if a designer component is missing?

Azure Machine Learning designer contains the most popular modules from Studio (classic). It also includes new modules that take advantage of the latest machine learning techniques.

If your migration is blocked due to missing modules in the designer, contact us by [creating a support ticket](#).

Example migration

The following experiment migration highlights some of the differences between Studio (classic) and Azure

Machine Learning.

Datasets

In Studio (classic), **datasets** were saved in your workspace and could only be used by Studio (classic).

The screenshot shows the Microsoft Azure Machine Learning Studio (classic) interface. On the left, there is a sidebar with icons for Projects, Experiments, Web Services, Datasets (which is selected and highlighted in blue), Trained Models, and Settings. The main area is titled "datasets" and contains two tabs: "MY DATASETS" and "SAMPLES". Below these tabs is a table with columns: NAME, SUBMITTED BY, DESCRIPTION, DATA TYPE, CREATED, SIZE, and PROJECT. The table lists various datasets, including "Automobile price data...", "Adult Census Income...", "Airport Codes Dataset", "Bike Rental UCI dataset", "Bill Gates RGB Image", "Blood donation data", "Book Reviews from A...", "Breast cancer data", "Breast Cancer Features", "Breast Cancer Info", "CRM Appettency Labe...", "CRM Churn Labels Sh...", and "CRM Dataset Shared". The row for "Automobile price data..." is selected, indicated by a blue background and a checked checkbox in the first column. At the bottom of the table are four buttons: DOWNLOAD, OPEN IN NOTEBOOK, GENERATE DATA ACCESS CODE..., and ADD TO PROJECT. There is also a "+ NEW" button on the far left of the bottom bar.

NAME	SUBMITTED BY	DESCRIPTION	DATA TYPE	CREATED	SIZE	PROJECT
<input type="checkbox"/> Adult Census Income...	Microsoft Corporation	Census Income dataset	GenericCSV	4/8/2015 3:10:05 PM	3.82 MB	None
<input type="checkbox"/> Airport Codes Dataset	Microsoft Corporation	Airport Codes Dataset	GenericCSV	4/8/2015 3:16:03 PM	16.64 KB	None
<input checked="" type="checkbox"/> Automobile price data...	Microsoft Corporation	Missing Value Scrubb...	GenericCSV	4/8/2015 3:10:42 PM	25.8 KB	None
<input type="checkbox"/> Bike Rental UCI dataset	Microsoft Corporation	Bike Rental UCI dataset	GenericCSV	4/8/2015 3:14:27 PM	1.1 MB	None
<input type="checkbox"/> Bill Gates RGB Image	Microsoft Corporation	Bill Gates RGB Image	GenericCSV	4/8/2015 3:20:16 PM	446.43 KB	None
<input type="checkbox"/> Blood donation data	Microsoft Corporation	Blood donor data tak...	GenericCSV	4/8/2015 3:11:19 PM	12.47 KB	None
<input type="checkbox"/> Book Reviews from A...	Microsoft Corporation	Book Reviews from A...	GenericTSVNoHeader	4/8/2015 3:30:27 PM	9.28 MB	None
<input type="checkbox"/> Breast cancer data	Microsoft Corporation	Breast cancer diagno...	ARFF	4/8/2015 3:08:46 PM	14.81 KB	None
<input type="checkbox"/> Breast Cancer Features	Microsoft Corporation	Breast Cancer Features	GenericTSVNoHeader	4/8/2015 3:28:50 PM	194.23 MB	None
<input type="checkbox"/> Breast Cancer Info	Microsoft Corporation	Breast Cancer Info	GenericTSVNoHeader	4/8/2015 3:25:41 PM	18.44 MB	None
<input type="checkbox"/> CRM Appettency Labe...	Microsoft Corporation	CRM Appettency Labels	GenericTSVNoHeader	4/8/2015 3:22:56 PM	194.44 KB	None
<input type="checkbox"/> CRM Churn Labels Sh...	Microsoft Corporation	CRM Churn Labels	GenericTSVNoHeader	4/8/2015 3:23:18 PM	191.73 KB	None
<input type="checkbox"/> CRM Dataset Shared	Microsoft Corporation	CRM Dataset	GenericTSV	4/8/2015 3:24:45 PM	24.99 MB	None

In Azure Machine Learning, **datasets** are registered to the workspace and can be used across all of Azure Machine Learning. For more information on the benefits of Azure Machine Learning datasets, see [Secure data access](#).

Pipeline

In Studio (classic), **experiments** contained the processing logic for your work. You created experiments with drag-and-drop modules.

Search experiment items

- Saved Datasets
- Trained Models
- Transforms
- Data Format Conversions
- Data Input and Output
- Data Transformation
- Feature Selection
- Machine Learning
- OpenCV Library Modules
- Python Language Modules
- R Language Modules
- Statistical Functions
- Text Analytics
- Time Series
- Web Service
- Deprecated

Training experiment Predictive experiment

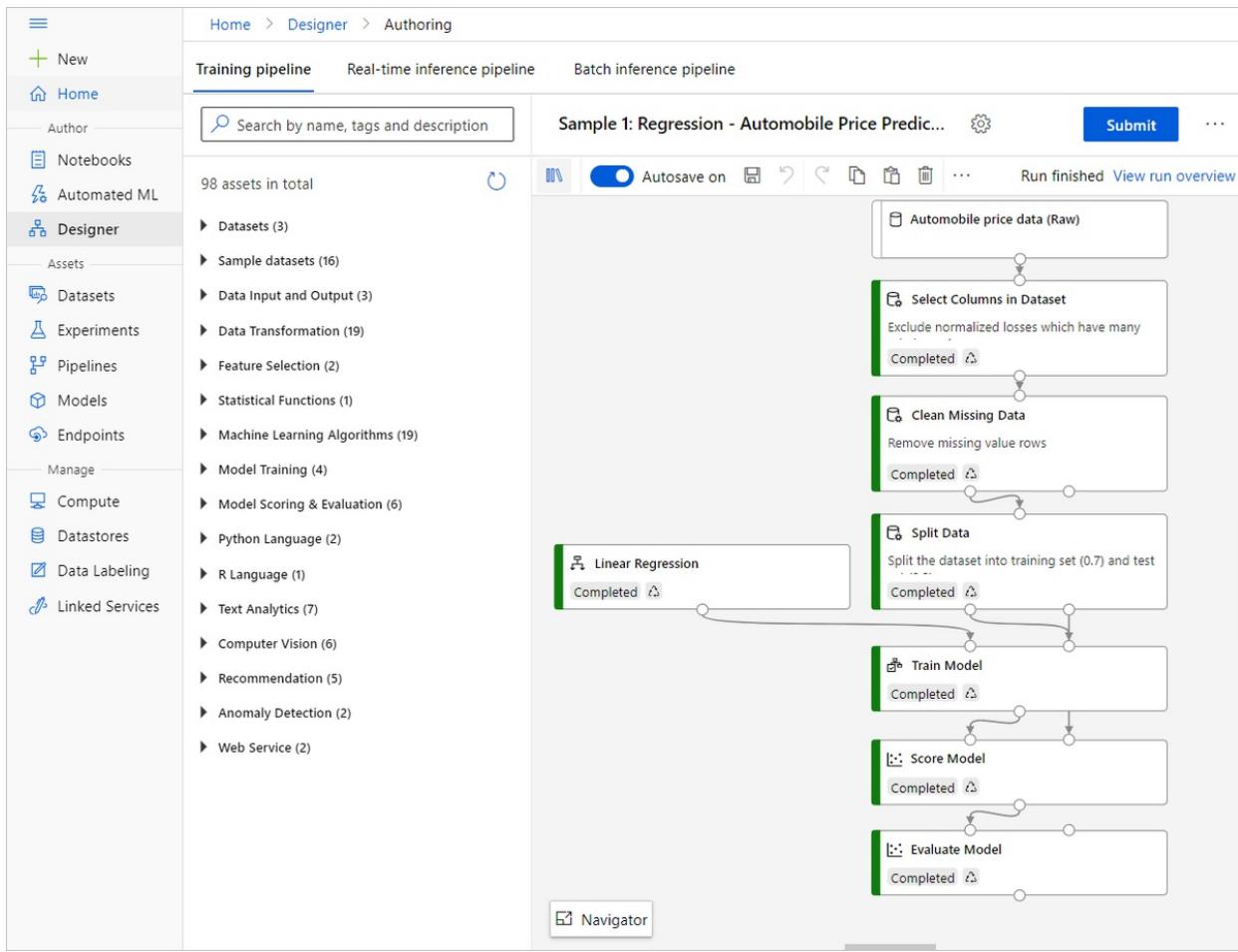
Automobile price prediction

Finished running ✓

```
graph TD; A[Automobile price data (Raw)] --> B[Clean Missing Data<br/>Handle the missing value in some columns]; B --> C[Select Columns in Dataset<br/>Exclude "num-of-doors" column from the feature set]; C --> D[Split Data<br/>Split the data into training set and testing set]; D --> E[Linear Regression]; E --> F[Train Model]; F --> G[Score Model<br/>Use the a different data set as the testing set to get the predicted price with Poisson ...]; G --> H[Evaluate Model];
```

NEW RUN HISTORY SAVE SAVE AS DISCARD CHANGES RUN SET UP WEB SERVICE PUBLISH TO GALLERY

In Azure Machine Learning, **pipelines** contain the processing logic for your work. You can create pipelines with either drag-and-drop modules or by writing code.



Web service endpoint

Studio (classic) used **REQUEST/RESPOND API** for real-time prediction and **BATCH EXECUTION API** for batch prediction or retraining.

Azure Machine Learning uses **real-time endpoints** (managed endpoints) for real-time prediction and **pipeline endpoints** for batch prediction or retraining.

sample-1-regression---automobile

Attributes

Service ID
sample-1-regression---automobile

Description
--

Deployment state
Unhealthy ⓘ

Compute type
AKS

Created by
User

Model ID
amlstudio-sample-1-regression:1

Created on
7/27/2020 1:46:24 PM

Last updated on
7/27/2020 1:46:38 PM

Compute target
plu-inference

Image ID
--

REST endpoint
 ⓘ

Key-based authentication enabled
true

Tags

CreatedByAMLStudio
true

Properties

LinkedPipelineDraftId
11111111-2222-3333-4444-aaaaaaaaaaaa

LinkedPipelineRunId
11111111-2222-3333-4444-aaaaaaaaaaab

hasInferenceSchema
True

hasHttps
False

Next steps

In this article, you learned the high-level requirements for migrating to Azure Machine Learning. For detailed steps, see the other articles in the Studio (classic) migration series:

1. [Migration overview.](#)
2. [Migrate dataset.](#)
3. [Rebuild a Studio \(classic\) training pipeline.](#)
4. [Rebuild a Studio \(classic\) web service.](#)
5. [Integrate an Azure Machine Learning web service with client apps.](#)
6. [Migrate Execute R Script.](#)

See the [Azure Machine Learning Adoption Framework](#) for additional migration resources.

Migrate a Studio (classic) dataset to Azure Machine Learning

9/21/2022 • 5 minutes to read • [Edit Online](#)

IMPORTANT

Support for Machine Learning Studio (classic) will end on 31 August 2024. We recommend you transition to [Azure Machine Learning](#) by that date.

Beginning 1 December 2021, you will not be able to create new Machine Learning Studio (classic) resources (workspace and web service plan). Through 31 August 2024, you can continue to use the existing Machine Learning Studio (classic) experiments and web services.

- See [information on moving machine learning projects from ML Studio \(classic\) to Azure Machine Learning](#).
- Learn more about [Azure Machine Learning](#)

ML Studio (classic) documentation is being retired and may not be updated in the future.

In this article, you learn how to migrate a Studio (classic) dataset to Azure Machine Learning. For more information on migrating from Studio (classic), see [the migration overview article](#).

You have three options to migrate a dataset to Azure Machine Learning. Read each section to determine which option is best for your scenario.

WHERE IS THE DATA?	MIGRATION OPTION
In Studio (classic)	Option 1: Download the dataset from Studio (classic) and upload it to Azure Machine Learning .
Cloud storage	Option 2: Register a dataset from a cloud source . Option 3: Use the Import Data module to get data from a cloud source .

NOTE

Azure Machine Learning also supports [code-first workflows](#) for creating and managing datasets.

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure Machine Learning workspace. [Create workspace resources](#).
- A Studio (classic) dataset to migrate.

Download the dataset from Studio (classic)

The simplest way to migrate a Studio (classic) dataset to Azure Machine Learning is to download your dataset and register it in Azure Machine Learning. This creates a new copy of your dataset and uploads it to an Azure Machine Learning datastore.

You can download the following Studio (classic) dataset types directly.

- Plain text (.txt)
- Comma-separated values (CSV) with a header (.csv) or without (.nh.csv)
- Tab-separated values (TSV) with a header (.tsv) or without (.nh.tsv)
- Excel file
- Zip file (.zip)

To download datasets directly:

1. Go to your Studio (classic) workspace (<https://studio.azureml.net>).
2. In the left navigation bar, select the **Datasets** tab.
3. Select the dataset(s) you want to download.
4. In the bottom action bar, select **Download**.

NAME	SUBMITTED BY	DESCRIPTION	DATA TYPE	CREATED	SIZE	PROJECT
Adult Census Income...	Microsoft Corporation	Census Income dataset	GenericCSV	4/8/2015 3:10:05 PM	3.82 MB	None
Airport Codes Dataset	Microsoft Corporation	Airport Codes Dataset	GenericCSV	4/8/2015 3:16:03 PM	16.64 KB	None
<input checked="" type="checkbox"/> Automobile price data...	Microsoft Corporation	Missing Value Scrub...	GenericCSV	4/8/2015 3:10:42 PM	25.8 KB	None
Bike Rental UCI dataset	Microsoft Corporation	Bike Rental UCI dataset	GenericCSV	4/8/2015 3:14:27 PM	1.1 MB	None
Blood donation data	Microsoft Corporation	Blood donor data tak...	GenericCSV	4/8/2015 3:11:19 PM	12.47 KB	None
Book Reviews from A...	Microsoft Corporation	Book Reviews from A...	GenericTSVNoHeader	4/8/2015 3:30:27 PM	9.28 MB	None
Breast cancer data	Microsoft Corporation	Breast cancer diagno...	ARFF	4/8/2015 3:08:46 PM	14.81 KB	None
Breast Cancer Features	Microsoft Corporation	Breast Cancer Features	GenericTSVNoHeader	4/8/2015 3:28:50 PM	194.23 MB	None
Breast Cancer Info	Microsoft Corporation	Breast Cancer Info	GenericTSVNoHeader	4/8/2015 3:25:41 PM	18.44 MB	None
CRM Appetency Labe...	Microsoft Corporation	CRM Appetency Labels	GenericTSVNoHeader	4/8/2015 3:22:56 PM	194.44 KB	None
CRM Churn Labels Sh...	Microsoft Corporation	CRM Churn Labels	GenericTSVNoHeader	4/8/2015 3:23:18 PM	191.73 KB	None
CRM Dataset Shared	Microsoft Corporation	CRM Dataset	GenericTSV	4/8/2015 3:24:45 PM	24.99 MB	None

For the following data types, you must use the **Convert to CSV** module to download datasets.

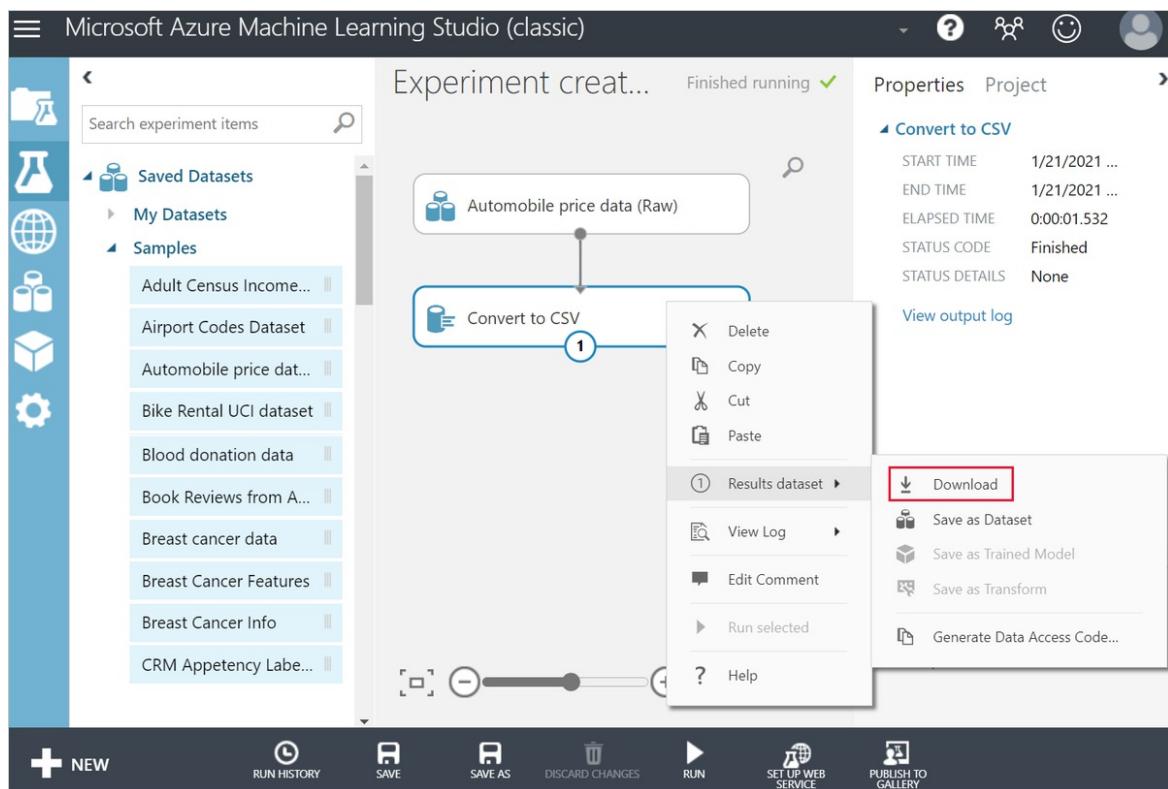
- SVMLight data (.svmlight)
- Attribute Relation File Format (ARFF) data (.arff)
- R object or workspace file (.RData)
- Dataset type (.data). Dataset type is Studio(classic) internal data type for module output.

To convert your dataset to a CSV and download the results:

1. Go to your Studio (classic) workspace (<https://studio.azureml.net>).
2. Create a new experiment.
3. Drag and drop the dataset you want to download onto the canvas.
4. Add a **Convert to CSV** module.
5. Connect the **Convert to CSV** input port to the output port of your dataset.
6. Run the experiment.

7. Right-click the **Convert to CSV** module.

8. Select **Results dataset > Download**.



Upload your dataset to Azure Machine Learning

After you download the data file, you can register the dataset in Azure Machine Learning:

1. Go to Azure Machine Learning studio (ml.azure.com).
2. In the left navigation pane, select the **Datasets** tab.
3. Select **Create dataset > From local files**.

Microsoft Azure Machine Learning Studio

The screenshot shows the Microsoft Azure Machine Learning Studio interface. On the left, there's a navigation sidebar with various options like New, Home, Author, Notebooks, Automated ML, Designer, Assets, Data (which is selected and highlighted with a red box), Jobs, Components, Pipelines, Environments, and Models. The main content area is titled 'Data' and shows a list of 'Registered data assets'. At the top of this list is a 'Create' button with a dropdown menu. The 'From local files' option in this dropdown is also highlighted with a red box. The main table has columns for 'Version' and 'Data source'. It lists two entries: one from 'mydatastore' and another from 'workspaceblobstore'.

Version	Data source
1	mydatastore
1	workspaceblobstore

4. Enter a name and description.

5. For **Dataset type**, select **Tabular**.

NOTE

You can also upload ZIP files as datasets. To upload a ZIP file, select **File** for **Dataset type**.

6. For **Datastore and file selection**, select the datastore you want to upload your dataset file to.

By default, Azure Machine Learning stores the dataset to the default workspace blobstore. For more information on datastores, see [Connect to storage services](#).

7. Set the data parsing settings and schema for your dataset. Then, confirm your settings.

Import data from cloud sources

If your data is already in a cloud storage service, and you want to keep your data in its native location. You can use either of the following options:

INGESTION METHOD	DESCRIPTION
------------------	-------------

INGESTION METHOD	DESCRIPTION
Register an Azure Machine Learning dataset	<p>Ingest data from local and online data sources (Blob, ADLS Gen1, ADLS Gen2, File share, SQL DB).</p> <p>Creates a reference to the data source, which is lazily evaluated at runtime. Use this option if you repeatedly access this dataset and want to enable advanced data features like data versioning and monitoring.</p>
Import Data module	<p>Ingest data from online data sources (Blob, ADLS Gen1, ADLS Gen2, File share, SQL DB).</p> <p>The dataset is only imported to the current designer pipeline run.</p>

NOTE

Studio (classic) users should note that the following cloud sources are not natively supported in Azure Machine Learning:

- Hive Query
- Azure Table
- Azure Cosmos DB
- On-premises SQL Database

We recommend that users migrate their data to a supported storage services using Azure Data Factory.

Register an Azure Machine Learning dataset

Use the following steps to register a dataset to Azure Machine Learning from a cloud service:

1. [Create a datastore](#), which links the cloud storage service to your Azure Machine Learning workspace.
2. [Register a dataset](#). If you are migrating a Studio (classic) dataset, select the **Tabular** dataset setting.

After you register a dataset in Azure Machine Learning, you can use it in designer:

1. Create a new designer pipeline draft.
2. In the module palette to the left, expand the **Datasets** section.
3. Drag your registered dataset onto the canvas.

Use the Import Data module

Use the following steps to import data directly to your designer pipeline:

1. [Create a datastore](#), which links the cloud storage service to your Azure Machine Learning workspace.

After you create the datastore, you can use the **Import Data** module in the designer to ingest data from it:

1. Create a new designer pipeline draft.
2. In the module palette to the left, find the **Import Data** module and drag it to the canvas.
3. Select the **Import Data** module, and use the settings in the right panel to configure your data source.

Next steps

In this article, you learned how to migrate a Studio (classic) dataset to Azure Machine Learning. The next step is to [rebuild a Studio \(classic\) training pipeline](#).

See the other articles in the Studio (classic) migration series:

1. [Migration overview.](#)
2. [Migrate datasets.](#)
3. [Rebuild a Studio \(classic\) training pipeline.](#)
4. [Rebuild a Studio \(classic\) web service.](#)
5. [Integrate an Azure Machine Learning web service with client apps.](#)
6. [Migrate Execute R Script.](#)

Rebuild a Studio (classic) experiment in Azure Machine Learning

9/21/2022 • 4 minutes to read • [Edit Online](#)

IMPORTANT

Support for Machine Learning Studio (classic) will end on 31 August 2024. We recommend you transition to [Azure Machine Learning](#) by that date.

Beginning 1 December 2021, you will not be able to create new Machine Learning Studio (classic) resources (workspace and web service plan). Through 31 August 2024, you can continue to use the existing Machine Learning Studio (classic) experiments and web services.

- See [information on moving machine learning projects from ML Studio \(classic\) to Azure Machine Learning](#).
- Learn more about [Azure Machine Learning](#)

ML Studio (classic) documentation is being retired and may not be updated in the future.

In this article, you learn how to rebuild an ML Studio (classic) experiment in Azure Machine Learning. For more information on migrating from Studio (classic), see [the migration overview article](#).

Studio (classic) **experiments** are similar to **pipelines** in Azure Machine Learning. However, in Azure Machine Learning pipelines are built on the same back-end that powers the SDK. This means that you have two options for machine learning development: the drag-and-drop designer or code-first SDKs.

For more information on building pipelines with the SDK, see [What are Azure Machine Learning pipelines](#).

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure Machine Learning workspace. [Create workspace resources](#).
- A Studio (classic) experiment to migrate.
- [Upload your dataset](#) to Azure Machine Learning.

Rebuild the pipeline

After you [migrate your dataset to Azure Machine Learning](#), you're ready to recreate your experiment.

In Azure Machine Learning, the visual graph is called a **pipeline draft**. In this section, you recreate your classic experiment as a pipeline draft.

1. Go to Azure Machine Learning studio (ml.azure.com)
2. In the left navigation pane, select **Designer** > **Easy-to-use prebuilt modules**

Microsoft > sdg-ws > Designer

Designer

New pipeline

Easy-to-use prebuilt components [\(i\)](#)

Image Classification using DenseNet [\(i\)](#)

3. Manually rebuild your experiment with designer components.

Consult the [module-mapping table](#) to find replacement modules. Many of Studio (classic)'s most popular modules have identical versions in the designer.

IMPORTANT

If your experiment uses the Execute R Script module, you need to perform additional steps to migrate your experiment. For more information, see [Migrate R Script modules](#).

4. Adjust parameters.

Select each module and adjust the parameters in the module settings panel to the right. Use the parameters to recreate the functionality of your Studio (classic) experiment. For more information on each module, see the [module reference](#).

Submit a job and check results

After you recreate your Studio (classic) experiment, it's time to submit a **pipeline job**.

A pipeline job executes on a **compute target** attached to your workspace. You can set a default compute target for the entire pipeline, or you can specify compute targets on a per-module basis.

Once you submit a job from a pipeline draft, it turns into a **pipeline job**. Each pipeline job is recorded and logged in Azure Machine Learning.

To set a default compute target for the entire pipeline:

1. Select the **Gear icon**  next to the pipeline name.
2. Select **Select compute target**.
3. Select an existing compute, or create a new compute by following the on-screen instructions.

Now that your compute target is set, you can submit a pipeline job:

1. At the top of the canvas, select **Submit**.
2. Select **Create new** to create a new experiment.

Experiments organize similar pipeline jobs together. If you run a pipeline multiple times, you can select

the same experiment for successive jobs. This is useful for logging and tracking.

3. Enter an experiment name. Then, select **Submit**.

The first job may take up to 20 minutes. Since the default compute settings have a minimum node size of 0, the designer must allocate resources after being idle. Successive jobs take less time, since the nodes are already allocated. To speed up the running time, you can create a compute resources with a minimum node size of 1 or greater.

After the job finishes, you can check the results of each module:

1. Right-click the module whose output you want to see.

2. Select either **Visualize**, **View Output**, or **View Log**.

- **Visualize**: Preview the results dataset.
- **View Output**: Open a link to the output storage location. Use this to explore or download the output.
- **View Log**: View driver and system logs. Use the **70_driver_log** to see information related to your user-submitted script such as errors and exceptions.

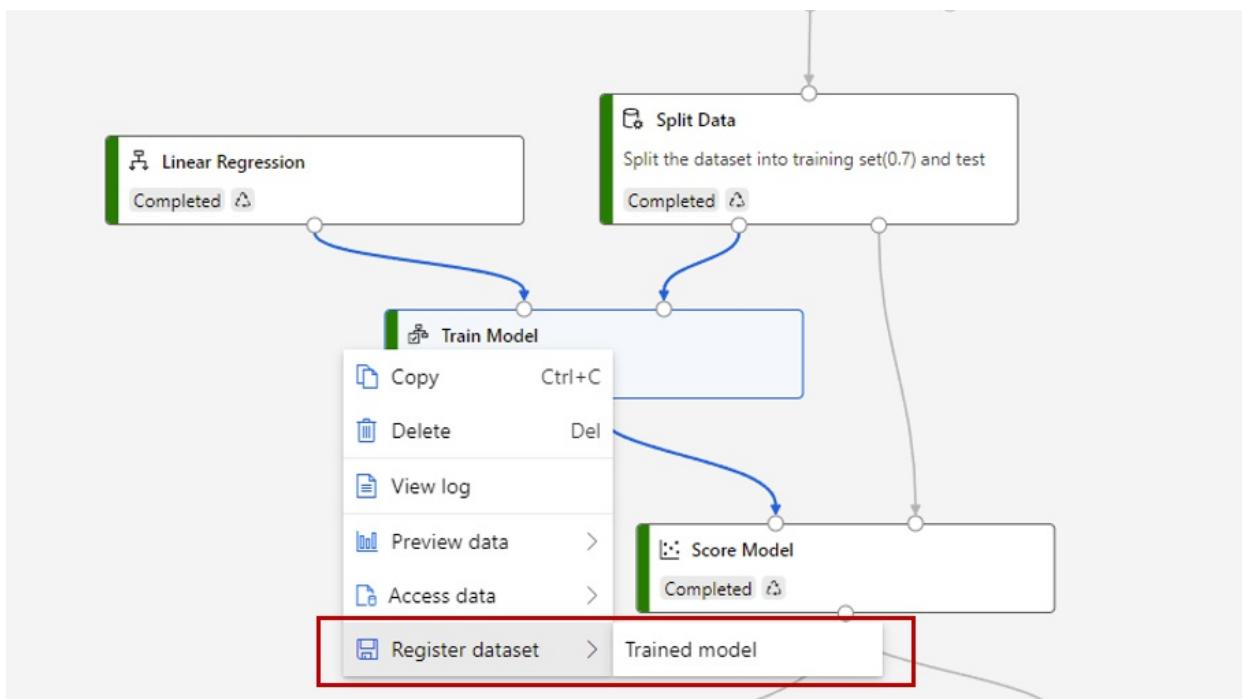
IMPORTANT

Designer components use open source Python packages to implement machine learning algorithms. However Studio (classic) uses a Microsoft internal C# library. Therefore, prediction result may vary between the designer and Studio (classic).

Save trained model to use in another pipeline

Sometimes you may want to save the model trained in a pipeline and use the model in another pipeline later. In Studio (classic), all trained models are saved in "Trained Models" category in the module list. In designer, the trained models are automatically registered as file dataset with a system generated name. Naming convention follows "MD - pipeline draft name - component name - Trained model ID" pattern.

To give a trained model a meaningful name, you can register the output of **Train Model** component as a **file dataset**. Give it the name you want, for example linear-regression-model.



You can find the trained model in "Dataset" category in the component list or search it by name. Then connect

the trained model to a **Score Model** component to use it for prediction.

The screenshot shows a search results page for 'linear-regression-model'. At the top, there's a search bar with the text 'linear-regression-model' and a close button 'X'. Below the search bar, it says '1 assets found.' with a refresh icon and a plus sign icon. The main area displays a single asset card for 'linear-regression-model'. The card includes a small icon, the name 'linear-regression-model', a 'Version 1' badge, a status box 'azureml.Designer: true', and a date '10/14/2021'.

Next steps

In this article, you learned how to rebuild a Studio (classic) experiment in Azure Machine Learning. The next step is to [rebuild web services in Azure Machine Learning](#).

See the other articles in the Studio (classic) migration series:

1. [Migration overview](#).
2. [Migrate dataset](#).
3. [Rebuild a Studio \(classic\) training pipeline](#).
4. [Rebuild a Studio \(classic\) web service](#).
5. [Integrate an Azure Machine Learning web service with client apps](#).
6. [Migrate Execute R Script](#).

Rebuild a Studio (classic) web service in Azure Machine Learning

9/21/2022 • 5 minutes to read • [Edit Online](#)

IMPORTANT

Support for Machine Learning Studio (classic) will end on 31 August 2024. We recommend you transition to [Azure Machine Learning](#) by that date.

Beginning 1 December 2021, you will not be able to create new Machine Learning Studio (classic) resources (workspace and web service plan). Through 31 August 2024, you can continue to use the existing Machine Learning Studio (classic) experiments and web services.

- See [information on moving machine learning projects from ML Studio \(classic\) to Azure Machine Learning](#).
- Learn more about [Azure Machine Learning](#)

ML Studio (classic) documentation is being retired and may not be updated in the future.

In this article, you learn how to rebuild an ML Studio (classic) web service as an **endpoint** in Azure Machine Learning.

Use Azure Machine Learning pipeline endpoints to make predictions, retrain models, or run any generic pipeline. The REST endpoint lets you run pipelines from any platform.

This article is part of the Studio (classic) to Azure Machine Learning migration series. For more information on migrating to Azure Machine Learning, see the [migration overview article](#).

NOTE

This migration series focuses on the drag-and-drop designer. For more information on deploying models programmatically, see [Deploy machine learning models in Azure](#).

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure Machine Learning workspace. [Create workspace resources](#).
- An Azure Machine Learning training pipeline. For more information, see [Rebuild a Studio \(classic\) experiment in Azure Machine Learning](#).

Real-time endpoint vs pipeline endpoint

Studio (classic) web services have been replaced by **endpoints** in Azure Machine Learning. Use the following table to choose which endpoint type to use:

STUDIO (CLASSIC) WEB SERVICE	AZURE MACHINE LEARNING REPLACEMENT
Request/respond web service (real-time prediction)	Real-time endpoint
Batch web service (batch prediction)	Pipeline endpoint

STUDIO (CLASSIC) WEB SERVICE	AZURE MACHINE LEARNING REPLACEMENT
Retraining web service (retraining)	Pipeline endpoint

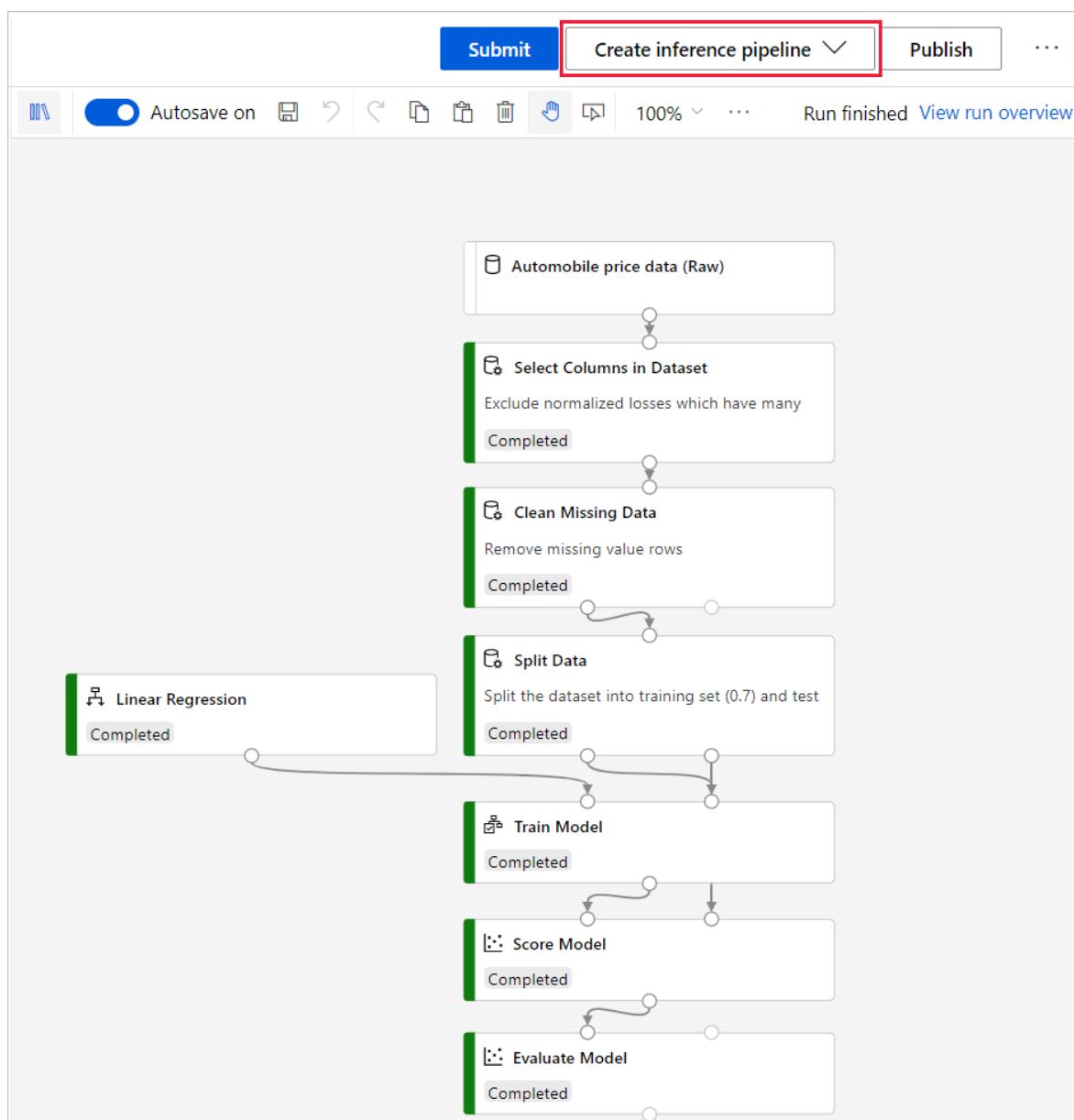
Deploy a real-time endpoint

In Studio (classic), you used a **REQUEST/RESPOND** web service to deploy a model for real-time predictions.

In Azure Machine Learning, you use a **real-time endpoint**.

There are multiple ways to deploy a model in Azure Machine Learning. One of the simplest ways is to use the designer to automate the deployment process. Use the following steps to deploy a model as a real-time endpoint:

1. Run your completed training pipeline at least once.
2. After the job completes, at the top of the canvas, select **Create inference pipeline > Real-time inference pipeline**.



The designer converts the training pipeline into a real-time inference pipeline. A similar conversion also occurs in Studio (classic).

In the designer, the conversion step also [registers the trained model to your Azure Machine Learning](#)

[workspace](#).

3. Select **Submit** to run the real-time inference pipeline, and verify that it runs successfully.
4. After you verify the inference pipeline, select **Deploy**.
5. Enter a name for your endpoint and a compute type.

The following table describes your deployment compute options in the designer:

COMPUTE TARGET	USED FOR	DESCRIPTION	CREATION
Azure Kubernetes Service (AKS)	Real-time inference	Large-scale, production deployments. Fast response time and service autoscaling.	User-created. For more information, see Create compute targets .
Azure Container Instances	Testing or development	Small-scale, CPU-based workloads that require less than 48 GB of RAM.	Automatically created by Azure Machine Learning.

Test the real-time endpoint

After deployment completes, you can see more details and test your endpoint:

1. Go the **Endpoints** tab.
2. Select your endpoint.
3. Select the **Test** tab.

Microsoft Azure Machine Learning

zhanxiaAML > Endpoints > migration-example-endpoint-aks

migration-example-endpoint-aks

Details **Test** Consume Deployment logs

Input data to test real-time endpoint **Test**

WebServiceInput0

symboling
3

normalized-losses
1

make
alfa-romero

fuel-type
gas

aspiration
std

num-of-doors
two

body-style
convertible

drive-wheels
rwd

engine-location
front

wheel-base
88.6

The 'Endpoints' option in the left sidebar is highlighted with a red box.

Publish a pipeline endpoint for batch prediction or retraining

You can also use your training pipeline to create a **pipeline endpoint** instead of a real-time endpoint. Use **pipeline endpoints** to perform either batch prediction or retraining.

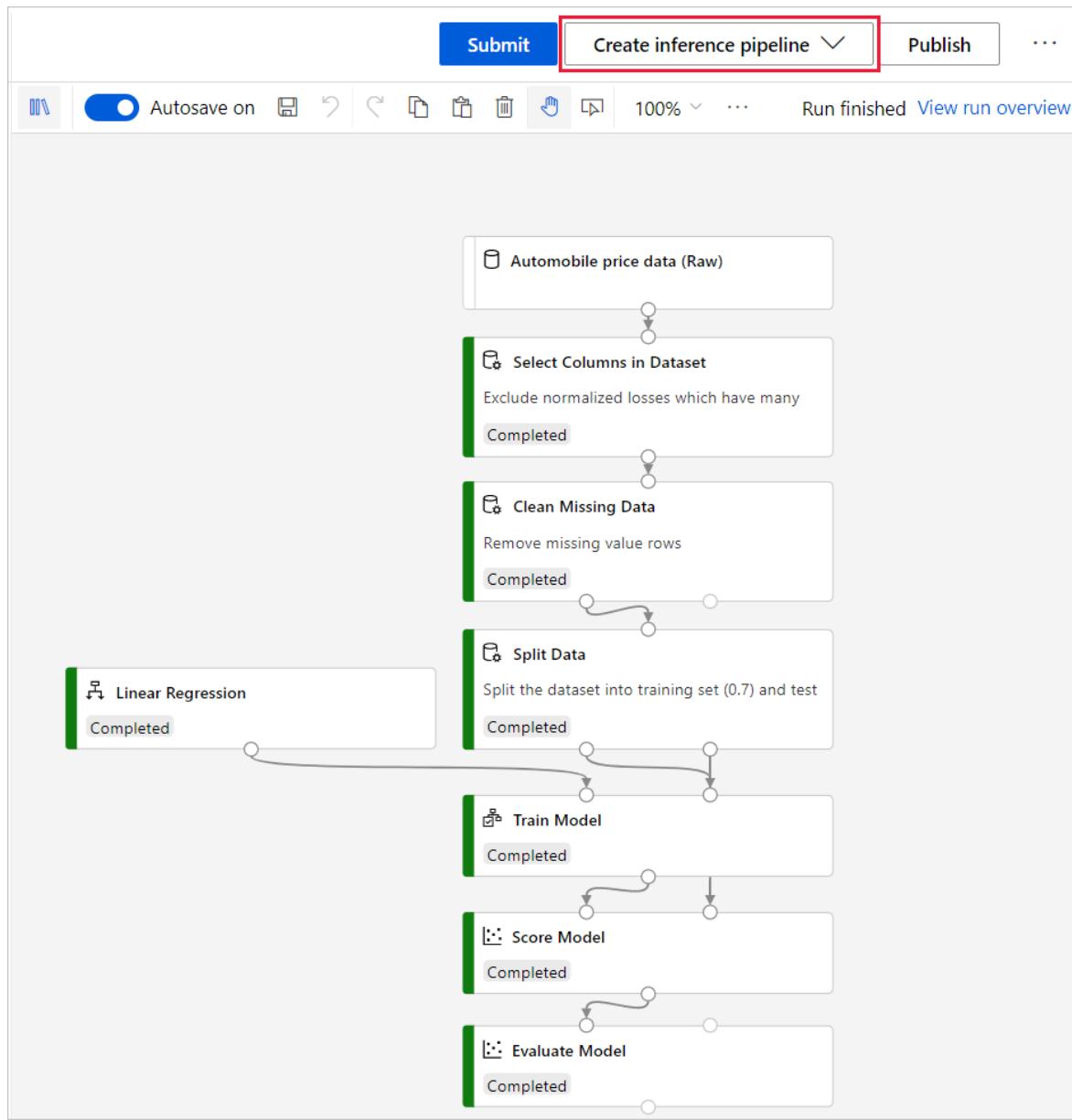
Pipeline endpoints replace Studio (classic) **batch execution endpoints** and **retraining web services**.

Publish a pipeline endpoint for batch prediction

Publishing a batch prediction endpoint is similar to the real-time endpoint.

Use the following steps to publish a pipeline endpoint for batch prediction:

1. Run your completed training pipeline at least once.
2. After the job completes, at the top of the canvas, select **Create inference pipeline > Batch inference pipeline**.



The designer converts the training pipeline into a batch inference pipeline. A similar conversion also occurs in Studio (classic).

In the designer, this step also [registers the trained model to your Azure Machine Learning workspace](#).

3. Select **Submit** to run the batch inference pipeline and verify that it successfully completes.
4. After you verify the inference pipeline, select **Publish**.
5. Create a new pipeline endpoint or select an existing one.

A new pipeline endpoint creates a new REST endpoint for your pipeline.

If you select an existing pipeline endpoint, you don't overwrite the existing pipeline. Instead, Azure Machine Learning versions each pipeline in the endpoint. You can specify which version to run in your REST call. You must also set a default pipeline if the REST call doesn't specify a version.

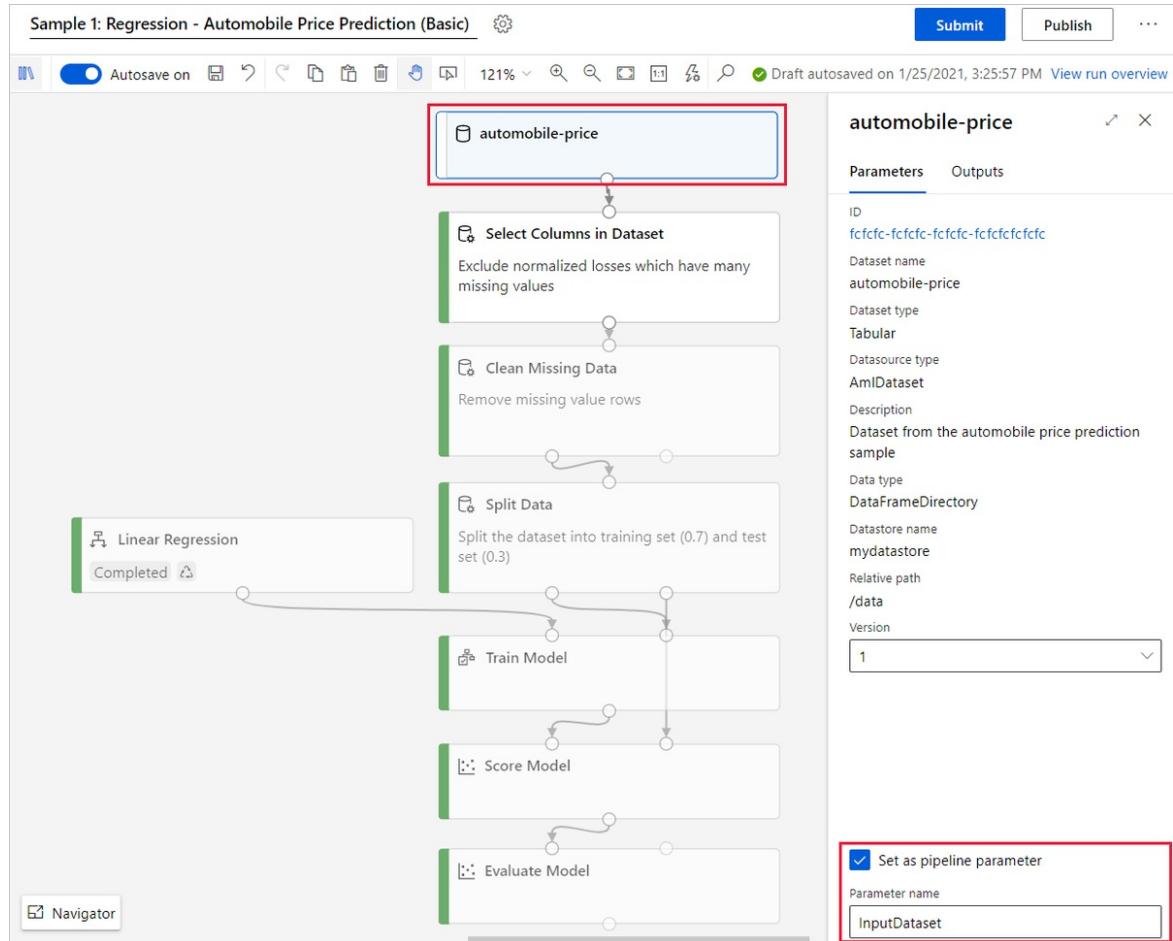
Publish a pipeline endpoint for retraining

To publish a pipeline endpoint for retraining, you must already have a pipeline draft that trains a model. For more information on building a training pipeline, see [Rebuild a Studio \(classic\) experiment](#).

To reuse your pipeline endpoint for retraining, you must create a **pipeline parameter** for your input dataset. This lets you dynamically set your training dataset, so that you can retrain your model.

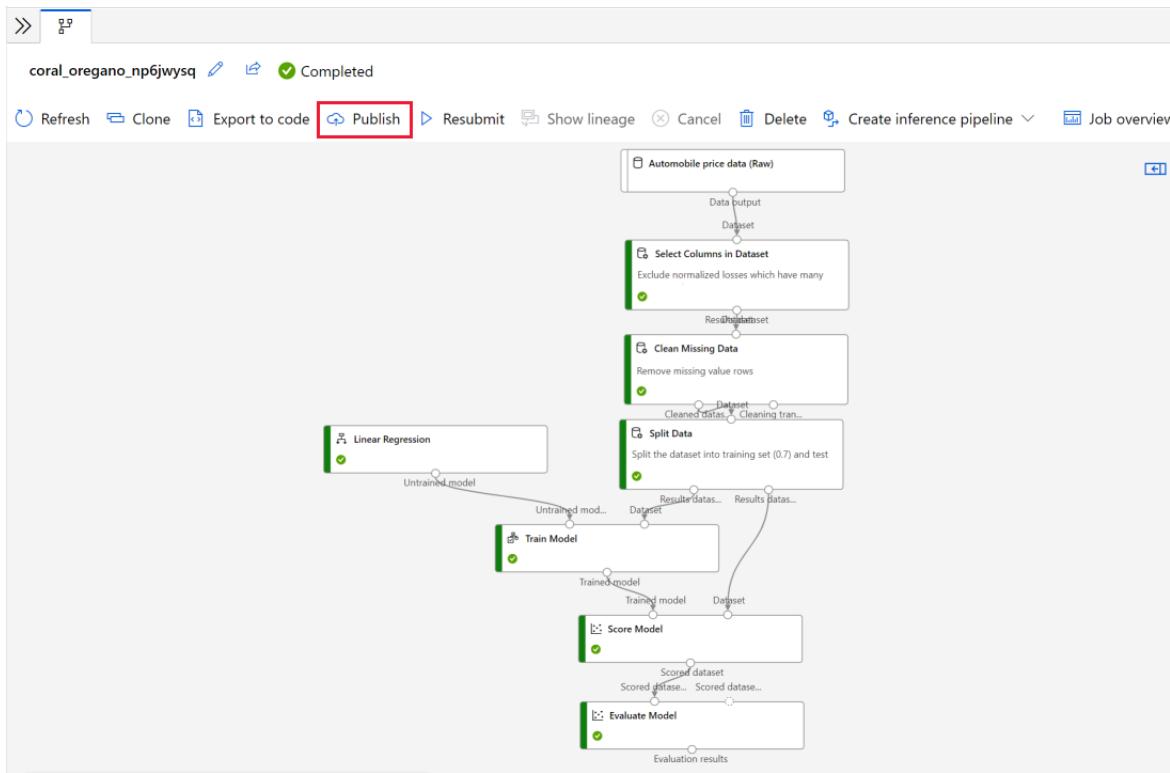
Use the following steps to publish retraining pipeline endpoint:

1. Run your training pipeline at least once.
2. After the run completes, select the dataset module.
3. In the module details pane, select **Set as pipeline parameter**.
4. Provide a descriptive name like "InputDataset".



This creates a pipeline parameter for your input dataset. When you call your pipeline endpoint for training, you can specify a new dataset to retrain the model.

5. Select **Publish**.



Call your pipeline endpoint from the studio

After you create your batch inference or retraining pipeline endpoint, you can call your endpoint directly from your browser.

1. Go to the **Pipelines** tab, and select **Pipeline endpoints**.
2. Select the pipeline endpoint you want to run.
3. Select **Submit**.

You can specify any pipeline parameters after you select **Submit**.

Next steps

In this article, you learned how to rebuild a Studio (classic) web service in Azure Machine Learning. The next step is to [integrate your web service with client apps](#).

See the other articles in the Studio (classic) migration series:

1. [Migration overview](#).
2. [Migrate dataset](#).
3. [Rebuild a Studio \(classic\) training pipeline](#).
4. [Rebuild a Studio \(classic\) web service](#).
5. [Integrate an Azure Machine Learning web service with client apps](#).
6. [Migrate Execute R Script](#).

Consume pipeline endpoints from client applications

9/21/2022 • 2 minutes to read • [Edit Online](#)

IMPORTANT

Support for Machine Learning Studio (classic) will end on 31 August 2024. We recommend you transition to [Azure Machine Learning](#) by that date.

Beginning 1 December 2021, you will not be able to create new Machine Learning Studio (classic) resources (workspace and web service plan). Through 31 August 2024, you can continue to use the existing Machine Learning Studio (classic) experiments and web services.

- See [information on moving machine learning projects from ML Studio \(classic\) to Azure Machine Learning](#).
- Learn more about [Azure Machine Learning](#)

ML Studio (classic) documentation is being retired and may not be updated in the future.

In this article, you learn how to integrate client applications with Azure Machine Learning endpoints. For more information on writing application code, see [Consume an Azure Machine Learning endpoint](#).

This article is part of the ML Studio (classic) to Azure Machine Learning migration series. For more information on migrating to Azure Machine Learning, see [the migration overview article](#).

Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- An Azure Machine Learning workspace. [Create workspace resources](#).
- An [Azure Machine Learning real-time endpoint](#) or [pipeline endpoint](#).

Consume a real-time endpoint

If you deployed your model as a *real-time endpoint*, you can find its REST endpoint, and pre-generated consumption code in C#, Python, and R:

1. Go to Azure Machine Learning studio ([ml.azure.com](#)).
2. Go the **Endpoints** tab.
3. Select your real-time endpoint.
4. Select **Consume**.

NOTE

You can also find the Swagger specification for your endpoint in the **Details** tab. Use the Swagger definition to understand your endpoint schema. For more information on Swagger definition, see [Swagger official documentation](#).

Consume a pipeline endpoint

There are two ways to consume a pipeline endpoint:

- REST API calls

- Integration with Azure Data Factory

Use REST API calls

Call the REST endpoint from your client application. You can use the Swagger specification for your endpoint to understand its schema:

1. Go to Azure Machine Learning studio (ml.azure.com).
2. Go the **Endpoints** tab.
3. Select **Pipeline endpoints**.
4. Select your pipeline endpoint.
5. In the **Pipeline endpoint overview** pane, select the link under **REST endpoint documentation**.

Use Azure Data Factory

You can call your Azure Machine Learning pipeline as a step in an Azure Data Factory pipeline. For more information, see [Execute Azure Machine Learning pipelines in Azure Data Factory](#).

Next steps

In this article, you learned how to find schema and sample code for your pipeline endpoints. For more information on consuming endpoints from the client application, see [Consume an Azure Machine Learning endpoint](#).

See the rest of the articles in the Azure Machine Learning migration series:

- [Migration overview](#).
- [Migrate dataset](#).
- [Rebuild a Studio \(classic\) training pipeline](#).
- [Rebuild a Studio \(classic\) web service](#).
- [Migrate Execute R Script](#).

Migrate Execute R Script modules in Studio (classic)

9/21/2022 • 2 minutes to read • [Edit Online](#)

IMPORTANT

Support for Machine Learning Studio (classic) will end on 31 August 2024. We recommend you transition to [Azure Machine Learning](#) by that date.

Beginning 1 December 2021, you will not be able to create new Machine Learning Studio (classic) resources (workspace and web service plan). Through 31 August 2024, you can continue to use the existing Machine Learning Studio (classic) experiments and web services.

- See [information on moving machine learning projects from ML Studio \(classic\) to Azure Machine Learning](#).
- Learn more about [Azure Machine Learning](#)

ML Studio (classic) documentation is being retired and may not be updated in the future.

In this article, you learn how to rebuild a Studio (classic) **Execute R Script** module in Azure Machine Learning.

For more information on migrating from Studio (classic), see the [migration overview article](#).

Execute R Script

Azure Machine Learning designer now runs on Linux. Studio (classic) runs on Windows. Due to the platform change, you must adjust your **Execute R Script** during migration, otherwise the pipeline will fail.

To migrate an **Execute R Script** module from Studio (classic), you must replace the `maml.mapInputPort` and `maml.mapOutputPort` interfaces with standard functions.

The following table summarizes the changes to the R Script module:

FEATURE	STUDIO (CLASSIC)	AZURE MACHINE LEARNING DESIGNER
Script Interface	<code>maml.mapInputPort</code> and <code>maml.mapOutputPort</code>	Function interface
Platform	Windows	Linux
Internet Accessible	No	Yes
Memory	14 GB	Dependent on Compute SKU

How to update the R script interface

Here are the contents of a sample **Execute R Script** module in Studio (classic):

```

#Map1-based optional input port to variables
dataset1<-maml.mapInputPort(1)#class:data.frame
dataset2<-maml.mapInputPort(2)#class:data.frame

#Content of optional Zipport are in ./src/
#source("src/yourfile.R");
#load("src/yourData.rdata");

#Sample operation
data.set=rbind(dataset1,dataset2);

#You'll see this output in the R Device port.
#It'll have your stdout, stderr and PNG graphics device(s).

plot(data.set);

#Select data.frame to be sent to the output Dataset port
maml.mapOutputPort("data.set");

```

Here are the updated contents in the designer. Notice that the `maml.mapInputPort` and `maml.mapOutputPort` have been replaced with the standard function interface `azureml_main`.

```

azureml_main<-function(dataframe1,dataframe2){
  #Use the parameters dataframe1 and dataframe2 directly
  dataset1<-dataframe1
  dataset2<-dataframe2

  #Content of optional Zipport are in ./src/
  #source("src/yourfile.R");
  #load("src/yourData.rdata");

  #Sample operation
  data.set=rbind(dataset1,dataset2);

  #You'll see this output in the R Device port.
  #It'll have your stdout, stderr and PNG graphics device(s).
  plot(data.set);

  #Return datasets as a NamedList

  return(list(dataset1=data.set))
}

```

For more information, see the designer [Execute R Script module reference](#).

Install R packages from the internet

Azure Machine Learning designer lets you install packages directly from CRAN.

This is an improvement over Studio (classic). Since Studio (classic) runs in a sandbox environment with no internet access, you had to upload scripts in a zip bundle to install more packages.

Use the following code to install CRAN packages in the designer's **Execute R Script** module:

```

if(!require(zoo)){
  install.packages("zoo",repos="http://cran.us.r-project.org")
}
library(zoo)

```

Next steps

In this article, you learned how to migrate Execute R Script modules to Azure Machine Learning.

See the other articles in the Studio (classic) migration series:

1. [Migration overview](#).
2. [Migrate dataset](#).
3. [Rebuild a Studio \(classic\) training pipeline](#).
4. [Rebuild a Studio \(classic\) web service](#).
5. [Integrate a Machine Learning web service with client apps](#).
6. **Migrate Execute R Script modules**.

Train models with Azure Machine Learning (v1)

9/21/2022 • 5 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

Azure Machine Learning provides several ways to train your models, from code-first solutions using the SDK to low-code solutions such as automated machine learning and the visual designer. Use the following list to determine which training method is right for you:

- [Azure Machine Learning SDK for Python](#): The Python SDK provides several ways to train models, each with different capabilities.

TRAINING METHOD	DESCRIPTION
Run configuration	A typical way to train models is to use a training script and job configuration. The job configuration provides the information needed to configure the training environment used to train your model. You can specify your training script, compute target, and Azure ML environment in your job configuration and run a training job.
Automated machine learning	Automated machine learning allows you to train models without extensive data science or programming knowledge . For people with a data science and programming background, it provides a way to save time and resources by automating algorithm selection and hyperparameter tuning. You don't have to worry about defining a job configuration when using automated machine learning.
Machine learning pipeline	Pipelines are not a different training method, but a way of defining a workflow using modular, reusable steps that can include training as part of the workflow. Machine learning pipelines support using automated machine learning and run configuration to train models. Since pipelines are not focused specifically on training, the reasons for using a pipeline are more varied than the other training methods. Generally, you might use a pipeline when: <ul style="list-style-type: none">* You want to schedule unattended processes such as long running training jobs or data preparation.* Use multiple steps that are coordinated across heterogeneous compute resources and storage locations.* Use the pipeline as a reusable template for specific scenarios, such as retraining or batch scoring.* Track and version data sources, inputs, and outputs for your workflow.* Your workflow is implemented by different teams that work on specific steps independently. Steps can then be joined together in a pipeline to implement the workflow.

- **Designer**: Azure Machine Learning designer provides an easy entry-point into machine learning for building proof of concepts, or for users with little coding experience. It allows you to train models using a

drag and drop web-based UI. You can use Python code as part of the design, or train models without writing any code.

- **Azure CLI:** The machine learning CLI provides commands for common tasks with Azure Machine Learning, and is often used for **scripting and automating tasks**. For example, once you've created a training script or pipeline, you might use the Azure CLI to start a training job on a schedule or when the data files used for training are updated. For training models, it provides commands that submit training jobs. It can submit jobs using run configurations or pipelines.

Each of these training methods can use different types of compute resources for training. Collectively, these resources are referred to as **compute targets**. A compute target can be a local machine or a cloud resource, such as an Azure Machine Learning Compute, Azure HDInsight, or a remote virtual machine.

Python SDK

The Azure Machine Learning SDK for Python allows you to build and run machine learning workflows with Azure Machine Learning. You can interact with the service from an interactive Python session, Jupyter Notebooks, Visual Studio Code, or other IDE.

- [What is the Azure Machine Learning SDK for Python](#)
- [Install/update the SDK](#)
- [Configure a development environment for Azure Machine Learning](#)

Run configuration

A generic training job with Azure Machine Learning can be defined using the [ScriptRunConfig](#). The script run configuration is then used, along with your training script(s) to train a model on a compute target.

You may start with a run configuration for your local computer, and then switch to one for a cloud-based compute target as needed. When changing the compute target, you only change the run configuration you use. A run also logs information about the training job, such as the inputs, outputs, and logs.

- [What is a run configuration?](#)
- [Tutorial: Train your first ML model](#)
- [Examples: Jupyter Notebook and Python examples of training models](#)
- [How to: Configure a training run](#)

Automated Machine Learning

Define the iterations, hyperparameter settings, featurization, and other settings. During training, Azure Machine Learning tries different algorithms and parameters in parallel. Training stops once it hits the exit criteria you defined.

TIP

In addition to the Python SDK, you can also use Automated ML through [Azure Machine Learning studio](#).

- [What is automated machine learning?](#)
- [Tutorial: Create your first classification model with automated machine learning](#)
- [Examples: Jupyter Notebook examples for automated machine learning](#)
- [How to: Configure automated ML experiments in Python](#)
- [How to: Autotrain a time-series forecast model](#)
- [How to: Create, explore, and deploy automated machine learning experiments with Azure Machine Learning studio](#)

Machine learning pipeline

Machine learning pipelines can use the previously mentioned training methods. Pipelines are more about creating a workflow, so they encompass more than just the training of models. In a pipeline, you can train a model using automated machine learning or run configurations.

- [What are ML pipelines in Azure Machine Learning?](#)
- [Create and run machine learning pipelines with Azure Machine Learning SDK](#)
- [Tutorial: Use Azure Machine Learning Pipelines for batch scoring](#)
- [Examples: Jupyter Notebook examples for machine learning pipelines](#)
- [Examples: Pipeline with automated machine learning](#)

Understand what happens when you submit a training job

The Azure training lifecycle consists of:

1. Zipping the files in your project folder, ignoring those specified in `.amlignore` or `.gitignore`
2. Scaling up your compute cluster
3. Building or downloading the dockerfile to the compute node
 - a. The system calculates a hash of:
 - The base image
 - Custom docker steps (see [Deploy a model using a custom Docker base image](#))
 - The conda definition YAML (see [Create & use software environments in Azure Machine Learning](#))
 - b. The system uses this hash as the key in a lookup of the workspace Azure Container Registry (ACR)
 - c. If it is not found, it looks for a match in the global ACR
 - d. If it is not found, the system builds a new image (which will be cached and registered with the workspace ACR)
4. Downloading your zipped project file to temporary storage on the compute node
5. Unzipping the project file
6. The compute node executing `python <entry script> <arguments>`
7. Saving logs, model files, and other files written to `./outputs` to the storage account associated with the workspace
8. Scaling down compute, including removing temporary storage

If you choose to train on your local machine ("configure as local run"), you do not need to use Docker. You may use Docker locally if you choose (see the section [Configure ML pipeline](#) for an example).

Azure Machine Learning designer

The designer lets you train models using a drag and drop interface in your web browser.

- [What is the designer?](#)
- [Tutorial: Predict automobile price](#)

Azure CLI

The machine learning CLI is an extension for the Azure CLI. It provides cross-platform CLI commands for working with Azure Machine Learning. Typically, you use the CLI to automate tasks, such as training a machine learning model.

- [Use the CLI extension for Azure Machine Learning](#)
- [MLOps on Azure](#)

Next steps

Learn how to [Configure a training run](#).

Data in Azure Machine Learning v1

9/21/2022 • 4 minutes to read • [Edit Online](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

APPLIES TO:  [Python SDK azureml v1](#)

Azure Machine Learning makes it easy to connect to your data in the cloud. It provides an abstraction layer over the underlying storage service, so you can securely access and work with your data without having to write code specific to your storage type. Azure Machine Learning also provides the following data capabilities:

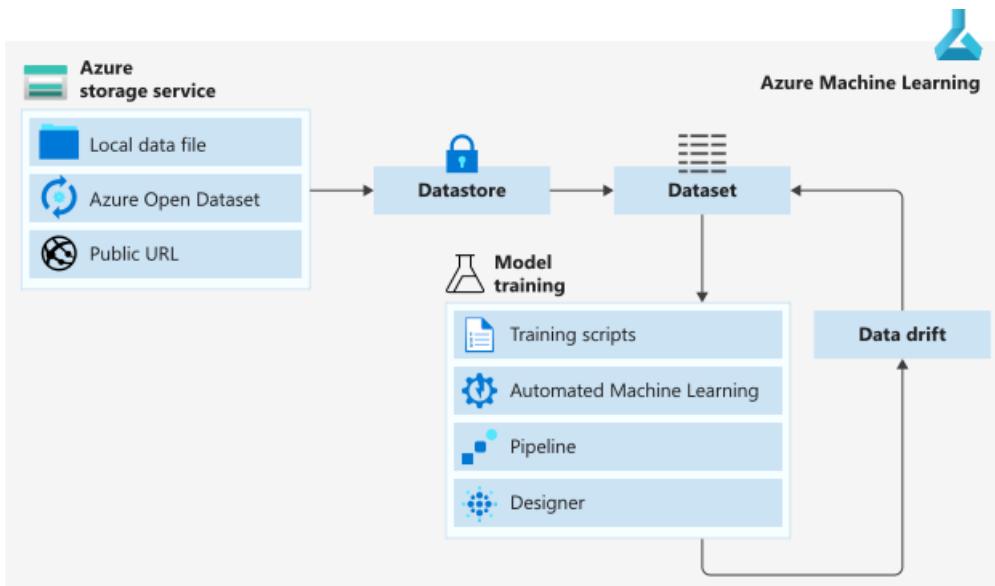
- Interoperability with Pandas and Spark DataFrames
- Versioning and tracking of data lineage
- Data labeling
- Data drift monitoring

Data workflow

When you're ready to use the data in your cloud-based storage solution, we recommend the following data delivery workflow. This workflow assumes you have an [Azure storage account](#) and data in a cloud-based storage service in Azure.

1. Create an [Azure Machine Learning datastore](#) to store connection information to your Azure storage.
 2. From that datastore, create an [Azure Machine Learning dataset](#) to point to a specific file(s) in your underlying storage.
 3. To use that dataset in your machine learning experiment you can either
 - Mount it to your experiment's compute target for model training.
- OR
4. Create [dataset monitors](#) for your model output dataset to detect for data drift.
 5. If data drift is detected, update your input dataset and retrain your model accordingly.

The following diagram provides a visual demonstration of this recommended workflow.



Connect to storage with datastores

Azure Machine Learning datastores securely keep the connection information to your data storage on Azure, so you don't have to code it in your scripts. [Register and create a datastore](#) to easily connect to your storage account, and access the data in your underlying storage service.

Supported cloud-based storage services in Azure that can be registered as datastores:

- Azure Blob Container
- Azure File Share
- Azure Data Lake
- Azure Data Lake Gen2
- Azure SQL Database
- Azure Database for PostgreSQL
- Databricks File System
- Azure Database for MySQL

TIP

You can create datastores with credential-based authentication for accessing storage services, like a service principal or shared access signature (SAS) token. These credentials can be accessed by users who have *Reader* access to the workspace.

If this is a concern, [create a datastore that uses identity-based data access](#) to connect to storage services.

Reference data in storage with datasets

Azure Machine Learning datasets aren't copies of your data. By creating a dataset, you create a reference to the data in its storage service, along with a copy of its metadata.

Because datasets are lazily evaluated, and the data remains in its existing location, you

- Incur no extra storage cost.
- Don't risk unintentionally changing your original data sources.
- Improve ML workflow performance speeds.

To interact with your data in storage, [create a dataset](#) to package your data into a consumable object for machine learning tasks. Register the dataset to your workspace to share and reuse it across different

experiments without data ingestion complexities.

Datasets can be created from local files, public urls, [Azure Open Datasets](#), or Azure storage services via datastores.

There are 2 types of datasets:

- A [FileDataset](#) references single or multiple files in your datastores or public URLs. If your data is already cleansed and ready to use in training experiments, you can [download or mount files](#) referenced by FileDatasets to your compute target.
- A [TabularDataset](#) represents data in a tabular format by parsing the provided file or list of files. You can load a TabularDataset into a pandas or Spark DataFrame for further manipulation and cleansing. For a complete list of data formats you can create TabularDatasets from, see the [TabularDatasetFactory class](#).

Additional datasets capabilities can be found in the following documentation:

- [Version and track](#) dataset lineage.
- [Monitor your dataset](#) to help with data drift detection.

Work with your data

With datasets, you can accomplish a number of machine learning tasks through seamless integration with Azure Machine Learning features.

- Create a [data labeling project](#).
- Train machine learning models:
 - [automated ML experiments](#)
 - the [designer](#)
 - [notebooks](#)
 - [Azure Machine Learning pipelines](#)
- Access datasets for scoring with [batch inference](#) in [machine learning pipelines](#).
- Set up a dataset monitor for [data drift](#) detection.

Label data with data labeling projects

Labeling large amounts of data has often been a headache in machine learning projects. Those with a computer vision component, such as image classification or object detection, generally require thousands of images and corresponding labels.

Azure Machine Learning gives you a central location to create, manage, and monitor labeling projects. Labeling projects help coordinate the data, labels, and team members, allowing you to more efficiently manage the labeling tasks. Currently supported tasks are image classification, either multi-label or multi-class, and object identification using bounded boxes.

Create an [image labeling project](#) or [text labeling project](#), and output a dataset for use in machine learning experiments.

Monitor model performance with data drift

In the context of machine learning, data drift is the change in model input data that leads to model performance degradation. It is one of the top reasons model accuracy degrades over time, thus monitoring data drift helps detect model performance issues.

See the [Create a dataset monitor](#) article, to learn more about how to detect and alert to data drift on new data in a dataset.

Next steps

- Create a dataset in Azure Machine Learning studio or with the Python SDK [using these steps](#).
- Try out dataset training examples with our [sample notebooks](#).

Network data access with Azure Machine Learning studio

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

APPLIES TO:  Azure CLI ml extension v1

Data access is complex and it's important to recognize that there are many pieces to it. For example, accessing data from Azure Machine Learning studio is different than using the SDK. When using the SDK on your local development environment, you're directly accessing data in the cloud. When using studio, you aren't always directly accessing the data store from your client. Studio relies on the workspace to access data on your behalf.

IMPORTANT

The information in this article is intended for Azure administrators who are creating the infrastructure required for an Azure Machine Learning solution.

TIP

Studio only supports reading data from the following datastore types in a VNet:

- Azure Storage Account (blob & file)
- Azure Data Lake Storage Gen1
- Azure Data Lake Storage Gen2
- Azure SQL Database

Data access

In general, data access from studio involves the following checks:

1. Who is accessing?
 - There are multiple different types of authentication depending on the storage type. For example, account key, token, service principal, managed identity, and user identity.
 - If authentication is made using a user identity, then it's important to know *which* user is trying to access storage. Learn more about [identity-based data access](#).
2. Do they have permission?
 - Are the credentials correct? If so, does the service principal, managed identity, etc., have the necessary permissions on the storage? Permissions are granted using Azure role-based access controls (Azure RBAC).
 - **Reader** of the storage account reads metadata of the storage.
 - **Storage Blob Data Reader** reads data within a blob container.
 - **Contributor** allows write access to a storage account.
 - More roles may be required depending on the type of storage.
3. Where is access from?
 - User: Is the client IP address in the VNet/subnet range?
 - Workspace: Is the workspace public or does it have a private endpoint in a VNet/subnet?

- Storage: Does the storage allow public access, or does it restrict access through a service endpoint or a private endpoint?

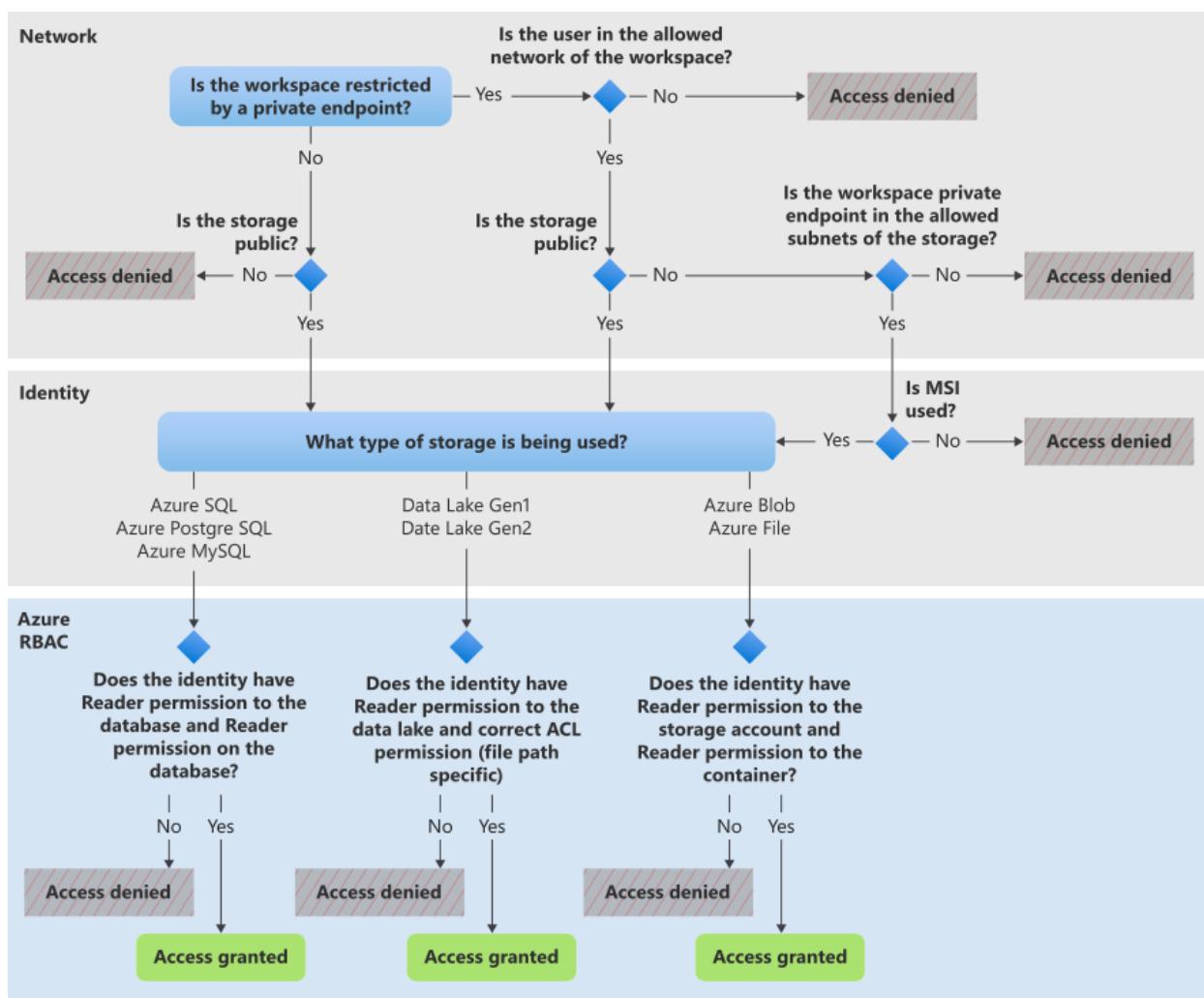
4. What operation is being performed?

- Create, read, update, and delete (CRUD) operations on a data store/dataset are handled by Azure Machine Learning.
- Data Access calls (such as preview or schema) go to the underlying storage and need extra permissions.

5. Where is this operation being run; compute resources in your Azure subscription or resources hosted in a Microsoft subscription?

- All calls to dataset and datastore services (except the "Generate Profile" option) use resources hosted in a **Microsoft subscription** to run the operations.
- Jobs, including the "Generate Profile" option for datasets, run on a compute resource in **your subscription**, and access the data from there. So the compute identity needs permission to the storage rather than the identity of the user submitting the job.

The following diagram shows the general flow of a data access call. In this example, a user is trying to make a data access call through a machine learning workspace, without using any compute resource.



Scenarios and identities

The following table lists what identities should be used for specific scenarios:

SCENARIO	USE WORKSPACE MANAGED SERVICE IDENTITY (MSI)	IDENTITY TO USE
Access from UI	Yes	Workspace MSI

SCENARIO	USE WORKSPACE MANAGED SERVICE IDENTITY (MSI)	IDENTITY TO USE
Access from UI	No	User's Identity
Access from Job	Yes/No	Compute MSI
Access from Notebook	Yes/No	User's identity

TIP

If you need to access data from outside Azure Machine Learning, such as using Azure Storage Explorer, *user* identity is probably what is used. Consult the documentation for the tool or service you are using for specific information. For more information on how Azure Machine Learning works with data, see [Identity-based data access to storage services on Azure](#).

Azure Storage Account

When using an Azure Storage Account from Azure Machine Learning studio, you must add the managed identity of the workspace to the following Azure RBAC roles for the storage account:

- [Blob Data Reader](#)
- If the storage account uses a private endpoint to connect to the VNet, you must grant the managed identity the [Reader](#) role for the storage account private endpoint.

For more information, see [Use Azure Machine Learning studio in an Azure Virtual Network](#).

See the following sections for information on limitations when using Azure Storage Account with your workspace in a VNet.

Secure communication with Azure Storage Account

To secure communication between Azure Machine Learning and Azure Storage Accounts, configure storage to [Grant access to trusted Azure services](#).

Azure Storage firewall

When an Azure Storage account is behind a virtual network, the storage firewall can normally be used to allow your client to directly connect over the internet. However, when using studio it isn't your client that connects to the storage account; it's the Azure Machine Learning service that makes the request. The IP address of the service isn't documented and changes frequently. **Enabling the storage firewall will not allow studio to access the storage account in a VNet configuration.**

Azure Storage endpoint type

When the workspace uses a private endpoint and the storage account is also in the VNet, there are extra validation requirements when using studio:

- If the storage account uses a **service endpoint**, the workspace private endpoint and storage service endpoint must be in the same subnet of the VNet.
- If the storage account uses a **private endpoint**, the workspace private endpoint and storage service endpoint must be in the same VNet. In this case, they can be in different subnets.

Azure Data Lake Storage Gen1

When using Azure Data Lake Storage Gen1 as a datastore, you can only use POSIX-style access control lists. You can assign the workspace's managed identity access to resources just like any other security principal. For more

information, see [Access control in Azure Data Lake Storage Gen1](#).

Azure Data Lake Storage Gen2

When using Azure Data Lake Storage Gen2 as a datastore, you can use both Azure RBAC and POSIX-style access control lists (ACLs) to control data access inside of a virtual network.

To use Azure RBAC, follow the steps in the [Datastore: Azure Storage Account](#) section of the 'Use Azure Machine Learning studio in an Azure Virtual Network' article. Data Lake Storage Gen2 is based on Azure Storage, so the same steps apply when using Azure RBAC.

To use ACLs, the managed identity of the workspace can be assigned access just like any other security principal. For more information, see [Access control lists on files and directories](#).

Azure SQL Database

To access data stored in an Azure SQL Database with a managed identity, you must create a SQL contained user that maps to the managed identity. For more information on creating a user from an external provider, see [Create contained users mapped to Azure AD identities](#).

After you create a SQL contained user, grant permissions to it by using the [GRANT T-SQL command](#).

Secure communication with Azure SQL Database

To secure communication between Azure Machine Learning and Azure SQL Database, there are two options:

IMPORTANT

Both options allow the possibility of services outside your subscription connecting to your SQL Database. Make sure that your SQL login and user permissions limit access to authorized users only.

- **Allow Azure services and resources to access the Azure SQL Database server.** Enabling this setting *allows all connections from Azure*, including **connections from the subscriptions of other customers**, to your database server.

For information on enabling this setting, see [IP firewall rules - Azure SQL Database and Synapse Analytics](#).

- **Allow the IP address range of the Azure Machine Learning service in Firewalls and virtual networks** for the Azure SQL Database. Allowing the IP addresses through the firewall limits **connections to the Azure Machine Learning service for a region**.

WARNING

The IP ranges for the Azure Machine Learning service may change over time. There is no built-in way to automatically update the firewall rules when the IPs change.

To get a list of the IP addresses for Azure Machine Learning, download the [Azure IP Ranges and Service Tags](#) and search the file for `AzureMachineLearning.<region>`, where `<region>` is the Azure region that contains your Azure Machine Learning workspace.

To add the IP addresses to your Azure SQL Database, see [IP firewall rules - Azure SQL Database and Synapse Analytics](#).

Next steps

For information on enabling studio in a network, see [Use Azure Machine Learning studio in an Azure Virtual Network](#).

Automated machine learning (AutoML)?

9/21/2022 • 15 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

Automated machine learning, also referred to as automated ML or AutoML, is the process of automating the time-consuming, iterative tasks of machine learning model development. It allows data scientists, analysts, and developers to build ML models with high scale, efficiency, and productivity all while sustaining model quality. Automated ML in Azure Machine Learning is based on a breakthrough from our [Microsoft Research division](#).

Traditional machine learning model development is resource-intensive, requiring significant domain knowledge and time to produce and compare dozens of models. With automated machine learning, you'll accelerate the time it takes to get production-ready ML models with great ease and efficiency.

Ways to use AutoML in Azure Machine Learning

Azure Machine Learning offers the following two experiences for working with automated ML. See the following sections to understand [feature availability in each experience \(v1\)](#).

- For code-experienced customers, [Azure Machine Learning Python SDK](#). Get started with [Tutorial: Use automated machine learning to predict taxi fares \(v1\)](#).
- For limited/no-code experience customers, Azure Machine Learning studio at <https://ml.azure.com>. Get started with these tutorials:
 - [Tutorial: Create a classification model with automated ML in Azure Machine Learning](#).
 - [Tutorial: Forecast demand with automated machine learning](#)

Experiment settings

The following settings allow you to configure your automated ML experiment.

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Split data into train/validation sets	✓	✓
Supports ML tasks: classification, regression, & forecasting	✓	✓
Supports computer vision tasks (preview): image classification, object detection & instance segmentation	✓	
Optimizes based on primary metric	✓	✓
Supports Azure ML compute as compute target	✓	✓
Configure forecast horizon, target lags & rolling window	✓	✓

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Set exit criteria	✓	✓
Set concurrent iterations	✓	✓
Drop columns	✓	✓
Block algorithms	✓	✓
Cross validation	✓	✓
Supports training on Azure Databricks clusters	✓	
View engineered feature names	✓	
Featurization summary	✓	
Featurization for holidays	✓	
Log file verbosity levels	✓	

Model settings

These settings can be applied to the best model as a result of your automated ML experiment.

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Best model registration, deployment, explainability	✓	✓
Enable voting ensemble & stack ensemble models	✓	✓
Show best model based on non-primary metric	✓	
Enable/disable ONNX model compatibility	✓	
Test the model	✓	✓ (preview)

Job control settings

These settings allow you to review and control your experiment jobs and its child jobs.

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Job summary table	✓	✓
Cancel jobs & child jobs	✓	✓
Get guardrails	✓	✓

	THE PYTHON SDK	THE STUDIO WEB EXPERIENCE
Pause & resume jobs	✓	

When to use AutoML: classification, regression, forecasting, computer vision & NLP

Apply automated ML when you want Azure Machine Learning to train and tune a model for you using the target metric you specify. Automated ML democratizes the machine learning model development process, and empowers its users, no matter their data science expertise, to identify an end-to-end machine learning pipeline for any problem.

ML professionals and developers across industries can use automated ML to:

- Implement ML solutions without extensive programming knowledge
- Save time and resources
- Leverage data science best practices
- Provide agile problem-solving

Classification

Classification is a common machine learning task. Classification is a type of supervised learning in which models learn using training data, and apply those learnings to new data. Azure Machine Learning offers featurizations specifically for these tasks, such as deep neural network text featurizers for classification. Learn more about [featurization \(v1\) options](#).

The main goal of classification models is to predict which categories new data will fall into based on learnings from its training data. Common classification examples include fraud detection, handwriting recognition, and object detection. Learn more and see an example at [Create a classification model with automated ML \(v1\)](#).

See examples of classification and automated machine learning in these Python notebooks: [Fraud Detection](#), [Marketing Prediction](#), and [Newsgroup Data Classification](#)

Regression

Similar to classification, regression tasks are also a common supervised learning task.

Different from classification where predicted output values are categorical, regression models predict numerical output values based on independent predictors. In regression, the objective is to help establish the relationship among those independent predictor variables by estimating how one variable impacts the others. For example, automobile price based on features like, gas mileage, safety rating, etc. Learn more and see an example of [regression with automated machine learning \(v1\)](#).

See examples of regression and automated machine learning for predictions in these Python notebooks: [CPU Performance Prediction](#),

Time-series forecasting

Building forecasts is an integral part of any business, whether it's revenue, inventory, sales, or customer demand. You can use automated ML to combine techniques and approaches and get a recommended, high-quality time-series forecast. Learn more with this how-to: [automated machine learning for time series forecasting \(v1\)](#).

An automated time-series experiment is treated as a multivariate regression problem. Past time-series values are "pivoted" to become additional dimensions for the regressor together with other predictors. This approach, unlike classical time series methods, has an advantage of naturally incorporating multiple contextual variables and their relationship to one another during training. Automated ML learns a single, but often internally branched model for all items in the dataset and prediction horizons. More data is thus available to estimate

model parameters and generalization to unseen series becomes possible.

Advanced forecasting configuration includes:

- holiday detection and featurization
- time-series and DNN learners (Auto-ARIMA, Prophet, ForecastTCN)
- many models support through grouping
- rolling-origin cross validation
- configurable lags
- rolling window aggregate features

See examples of regression and automated machine learning for predictions in these Python notebooks: [Sales Forecasting](#), [Demand Forecasting](#), and [Forecasting GitHub's Daily Active Users](#).

Computer vision (preview)

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

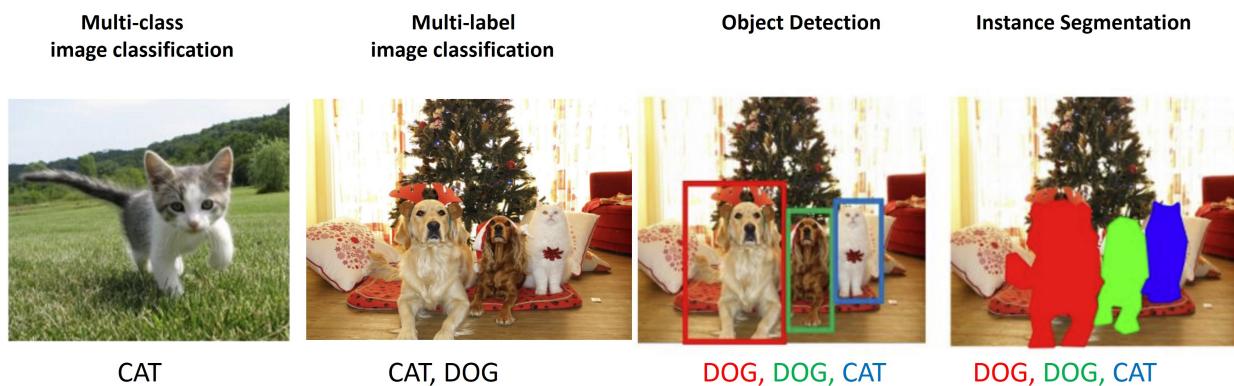
Support for computer vision tasks allows you to easily generate models trained on image data for scenarios like image classification and object detection.

With this capability you can:

- Seamlessly integrate with the [Azure Machine Learning data labeling](#) capability
- Use labeled data for generating image models
- Optimize model performance by specifying the model algorithm and tuning the hyperparameters.
- Download or deploy the resulting model as a web service in Azure Machine Learning.
- Operationalize at scale, leveraging Azure Machine Learning [MLOps](#) and [ML Pipelines \(v1\)](#) capabilities.

Authoring AutoML models for vision tasks is supported via the Azure ML Python SDK. The resulting experimentation jobs, models, and outputs can be accessed from the Azure Machine Learning studio UI.

Learn how to [set up AutoML training for computer vision models](#).



Automated ML for images supports the following computer vision tasks:

TASK	DESCRIPTION

TASK	DESCRIPTION
Multi-class image classification	Tasks where an image is classified with only a single label from a set of classes - e.g. each image is classified as either an image of a 'cat' or a 'dog' or a 'duck'
Multi-label image classification	Tasks where an image could have one or more labels from a set of labels - e.g. an image could be labeled with both 'cat' and 'dog'
Object detection	Tasks to identify objects in an image and locate each object with a bounding box e.g. locate all dogs and cats in an image and draw a bounding box around each.
Instance segmentation	Tasks to identify objects in an image at the pixel level, drawing a polygon around each object in the image.

Natural language processing: NLP (preview)

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Support for natural language processing (NLP) tasks in automated ML allows you to easily generate models trained on text data for text classification and named entity recognition scenarios. Authoring automated ML trained NLP models is supported via the Azure Machine Learning Python SDK. The resulting experimentation jobs, models, and outputs can be accessed from the Azure Machine Learning studio UI.

The NLP capability supports:

- End-to-end deep neural network NLP training with the latest pre-trained BERT models
- Seamless integration with [Azure Machine Learning data labeling](#)
- Use labeled data for generating NLP models
- Multi-lingual support with 104 languages
- Distributed training with Horovod

Learn how to [set up AutoML training for NLP models \(v1\)](#).

How automated ML works

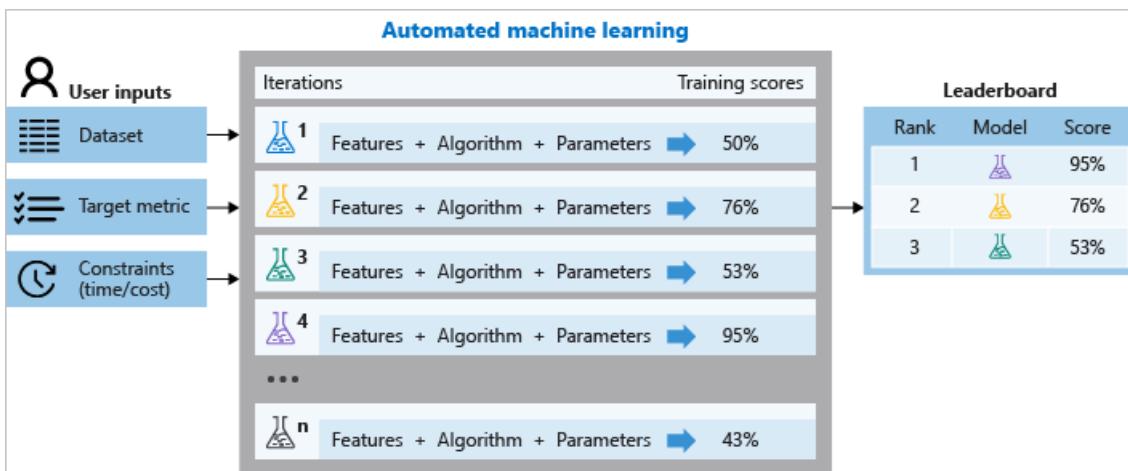
During training, Azure Machine Learning creates a number of pipelines in parallel that try different algorithms and parameters for you. The service iterates through ML algorithms paired with feature selections, where each iteration produces a model with a training score. The higher the score, the better the model is considered to "fit" your data. It will stop once it hits the exit criteria defined in the experiment.

Using [Azure Machine Learning](#), you can design and run your automated ML training experiments with these steps:

1. **Identify the ML problem** to be solved: classification, forecasting, regression or computer vision (preview).
2. **Choose whether you want to use the Python SDK or the studio web experience:** Learn about the parity between the [Python SDK](#) and [studio web experience](#).

- For limited or no code experience, try the Azure Machine Learning studio web experience at <https://ml.azure.com>
 - For Python developers, check out the [Azure Machine Learning Python SDK \(v1\)](#)
3. **Specify the source and format of the labeled training data:** Numpy arrays or Pandas dataframe
 4. **Configure the compute target for model training,** such as your [local computer, Azure Machine Learning Computes, remote VMs, or Azure Databricks with SDK v1](#).
 5. **Configure the automated machine learning parameters** that determine how many iterations over different models, hyperparameter settings, advanced preprocessing/featurization, and what metrics to look at when determining the best model.
 6. **Submit the training job.**
 7. **Review the results**

The following diagram illustrates this process.



You can also inspect the logged job information, which [contains metrics](#) gathered during the job. The training job produces a Python serialized object (`.pk1` file) that contains the model and data preprocessing.

While model building is automated, you can also [learn how important or relevant features are](#) to the generated models.

Guidance on local vs. remote managed ML compute targets

The web interface for automated ML always uses a remote [compute target](#). But when you use the Python SDK, you will choose either a local compute or a remote compute target for automated ML training.

- **Local compute:** Training occurs on your local laptop or VM compute.
- **Remote compute:** Training occurs on Machine Learning compute clusters.

Choose compute target

Consider these factors when choosing your compute target:

- **Choose a local compute:** If your scenario is about initial explorations or demos using small data and short trains (i.e. seconds or a couple of minutes per child job), training on your local computer might be a better choice. There is no setup time, the infrastructure resources (your PC or VM) are directly available.
- **Choose a remote ML compute cluster:** If you are training with larger datasets like in production training creating models which need longer trains, remote compute will provide much better end-to-end time performance because [AutoML](#) will parallelize trains across the cluster's nodes. On a remote compute, the

start-up time for the internal infrastructure will add around 1.5 minutes per child job, plus additional minutes for the cluster infrastructure if the VMs are not yet up and running.

Pros and cons

Consider these pros and cons when choosing to use local vs. remote.

	PROS (ADVANTAGES)	CONS (HANDICAPS)
Local compute target	<ul style="list-style-type: none">No environment start-up time	<ul style="list-style-type: none">Subset of featuresCan't parallelize jobsWorse for large data.No data streaming while trainingNo DNN-based featurizationPython SDK only
Remote ML compute clusters	<ul style="list-style-type: none">Full set of featuresParallelize child jobsLarge data supportDNN-based featurizationDynamic scalability of compute cluster on demandNo-code experience (web UI) also available	<ul style="list-style-type: none">Start-up time for cluster nodesStart-up time for each child job

Feature availability

More features are available when you use the remote compute, as shown in the table below.

FEATURE	REMOTE	LOCAL
Data streaming (Large data support, up to 100 GB)	✓	
DNN-BERT-based text featurization and training	✓	
Out-of-the-box GPU support (training and inference)	✓	
Image Classification and Labeling support	✓	
Auto-ARIMA, Prophet and ForecastTCN models for forecasting	✓	
Multiple jobs/iterations in parallel	✓	
Create models with interpretability in AutoML studio web experience UI	✓	
Feature engineering customization in studio web experience UI	✓	
Azure ML hyperparameter tuning	✓	
Azure ML Pipeline workflow support	✓	

FEATURE	REMOTE	LOCAL
Continue a job	✓	
Forecasting	✓	✓
Create and run experiments in notebooks	✓	✓
Register and visualize experiment's info and metrics in UI	✓	✓
Data guardrails	✓	✓

Training, validation and test data

With automated ML you provide the **training data** to train ML models, and you can specify what type of model validation to perform. Automated ML performs model validation as part of training. That is, automated ML uses **validation data** to tune model hyperparameters based on the applied algorithm to find the best combination that best fits the training data. However, the same validation data is used for each iteration of tuning, which introduces model evaluation bias since the model continues to improve and fit to the validation data.

To help confirm that such bias isn't applied to the final recommended model, automated ML supports the use of **test data** to evaluate the final model that automated ML recommends at the end of your experiment. When you provide test data as part of your AutoML experiment configuration, this recommended model is tested by default at the end of your experiment (preview).

IMPORTANT

Testing your models with a test dataset to evaluate generated models is a preview feature. This capability is an [experimental](#) preview feature, and may change at any time.

Learn how to [configure AutoML experiments to use test data \(preview\)](#) with the [SDK \(v1\)](#) or with the [Azure Machine Learning studio](#).

You can also [test any existing automated ML model \(preview\) \(v1\)](#), including models from child jobs, by providing your own test data or by setting aside a portion of your training data.

Feature engineering

Feature engineering is the process of using domain knowledge of the data to create features that help ML algorithms learn better. In Azure Machine Learning, scaling and normalization techniques are applied to facilitate feature engineering. Collectively, these techniques and feature engineering are referred to as featurization.

For automated machine learning experiments, featurization is applied automatically, but can also be customized based on your data. [Learn more about what featurization is included \(v1\)](#) and how AutoML helps [prevent overfitting and imbalanced data](#) in your models.

NOTE

Automated machine learning featurization steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same featurization steps applied during training are applied to your input data automatically.

Customize featurization

Additional feature engineering techniques such as, encoding and transforms are also available.

Enable this setting with:

- Azure Machine Learning studio: Enable **Automatic featurization** in the **View additional configuration** section [with these \(v1\) steps](#).
- Python SDK: Specify `"featurization": "auto" / "off" / FeaturizationConfig'` in your [AutoMLConfig](#) object. Learn more about [enabling featurization \(v1\)](#).

Ensemble models

Automated machine learning supports ensemble models, which are enabled by default. Ensemble learning improves machine learning results and predictive performance by combining multiple models as opposed to using single models. The ensemble iterations appear as the final iterations of your job. Automated machine learning uses both voting and stacking ensemble methods for combining models:

- **Voting**: predicts based on the weighted average of predicted class probabilities (for classification tasks) or predicted regression targets (for regression tasks).
- **Stacking**: stacking combines heterogenous models and trains a meta-model based on the output from the individual models. The current default meta-models are LogisticRegression for classification tasks and ElasticNet for regression/forecasting tasks.

The [Caruana ensemble selection algorithm](#) with sorted ensemble initialization is used to decide which models to use within the ensemble. At a high level, this algorithm initializes the ensemble with up to five models with the best individual scores, and verifies that these models are within 5% threshold of the best score to avoid a poor initial ensemble. Then for each ensemble iteration, a new model is added to the existing ensemble and the resulting score is calculated. If a new model improved the existing ensemble score, the ensemble is updated to include the new model.

See the [how-to \(v1\)](#) for changing default ensemble settings in automated machine learning.

AutoML & ONNX

With Azure Machine Learning, you can use automated ML to build a Python model and have it converted to the ONNX format. Once the models are in the ONNX format, they can be run on a variety of platforms and devices. Learn more about [accelerating ML models with ONNX](#).

See how to convert to ONNX format in [this Jupyter notebook example](#). Learn which [algorithms are supported in ONNX \(v1\)](#).

The ONNX runtime also supports C#, so you can use the model built automatically in your C# apps without any need for recoding or any of the network latencies that REST endpoints introduce. Learn more about [using an AutoML ONNX model in a .NET application with ML.NET](#) and [inferencing ONNX models with the ONNX runtime C# API](#).

Next steps

There are multiple resources to get you up and running with AutoML.

Tutorials/ how-tos

Tutorials are end-to-end introductory examples of AutoML scenarios.

- For a code first experience, follow the [Tutorial: Train a regression model with AutoML and Python \(v1\)](#).

- For a low or no-code experience, see the [Tutorial: Train a classification model with no-code AutoML in Azure Machine Learning studio](#).
- For using AutoML to train computer vision models, see the [Tutorial: Train an object detection model \(preview\) with AutoML and Python \(v1\)](#).

How-to articles provide additional detail into what functionality automated ML offers. For example,

- Configure the settings for automatic training experiments
 - [Without code in the Azure Machine Learning studio](#).
 - [With the Python SDK v1](#).
- Learn how to train forecasting models with time series data (v1).
- Learn how to train computer vision models with Python (v1).
- Learn how to view the generated code from your automated ML models.

Jupyter notebook samples

Review detailed code examples and use cases in the [GitHub notebook repository for automated machine learning samples](#).

Python SDK reference

Deepen your expertise of SDK design patterns and class specifications with the [AutoML class reference documentation](#).

NOTE

Automated machine learning capabilities are also available in other Microsoft solutions such as, [ML.NET](#), [HDInsight](#), [Power BI](#) and [SQL Server](#)

MLOps: Model management, deployment, lineage, and monitoring with Azure Machine Learning v1

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO: [Azure CLI ml extension v1](#) [Python SDK azureml v1](#)

In this article, learn about how do Machine Learning Operations (MLOps) in Azure Machine Learning to manage the lifecycle of your models. MLOps improves the quality and consistency of your machine learning solutions.

What is MLOps?

Machine Learning Operations (MLOps) is based on [DevOps](#) principles and practices that increase the efficiency of workflows. For example, continuous integration, delivery, and deployment. MLOps applies these principles to the machine learning process, with the goal of:

- Faster experimentation and development of models
- Faster deployment of models into production
- Quality assurance and end-to-end lineage tracking

MLOps in Azure Machine Learning

Azure Machine Learning provides the following MLOps capabilities:

- **Create reproducible ML pipelines.** Machine Learning pipelines allow you to define repeatable and reusable steps for your data preparation, training, and scoring processes.
- **Create reusable software environments** for training and deploying models.
- **Register, package, and deploy models from anywhere.** You can also track associated metadata required to use the model.
- **Capture the governance data for the end-to-end ML lifecycle.** The logged lineage information can include who is publishing models, why changes were made, and when models were deployed or used in production.
- **Notify and alert on events in the ML lifecycle.** For example, experiment completion, model registration, model deployment, and data drift detection.
- **Monitor ML applications for operational and ML-related issues.** Compare model inputs between training and inference, explore model-specific metrics, and provide monitoring and alerts on your ML infrastructure.
- **Automate the end-to-end ML lifecycle with Azure Machine Learning and Azure Pipelines.** Using pipelines allows you to frequently update models, test new models, and continuously roll out new ML models alongside your other applications and services.

For more information on MLOps, see [Machine Learning DevOps \(MLOps\)](#).

Create reproducible ML pipelines

Use ML pipelines from Azure Machine Learning to stitch together all of the steps involved in your model training process.

An ML pipeline can contain steps from data preparation to feature extraction to hyperparameter tuning to model evaluation. For more information, see [ML pipelines](#).

If you use the [Designer](#) to create your ML pipelines, you may at any time click the "..." at the top-right of the Designer page and then select **Clone**. Cloning your pipeline allows you to iterate your pipeline design without losing your old versions.

Create reusable software environments

Azure Machine Learning environments allow you to track and reproduce your projects' software dependencies as they evolve. Environments allow you to ensure that builds are reproducible without manual software configurations.

Environments describe the pip and Conda dependencies for your projects, and can be used for both training and deployment of models. For more information, see [What are Azure Machine Learning environments](#).

Register, package, and deploy models from anywhere

Register and track ML models

Model registration allows you to store and version your models in the Azure cloud, in your workspace. The model registry makes it easy to organize and keep track of your trained models.

TIP

A registered model is a logical container for one or more files that make up your model. For example, if you have a model that is stored in multiple files, you can register them as a single model in your Azure Machine Learning workspace. After registration, you can then download or deploy the registered model and receive all the files that were registered.

Registered models are identified by name and version. Each time you register a model with the same name as an existing one, the registry increments the version. Additional metadata tags can be provided during registration. These tags are then used when searching for a model. Azure Machine Learning supports any model that can be loaded using Python 3.5.2 or higher.

TIP

You can also register models trained outside Azure Machine Learning.

You can't delete a registered model that is being used in an active deployment. For more information, see the register model section of [Deploy models](#).

IMPORTANT

When using Filter by `Tags` option on the Models page of Azure Machine Learning Studio, instead of using `TagName : TagValue` customers should use `TagName=TagValue` (without space)

Package and debug models

Before deploying a model into production, it is packaged into a Docker image. In most cases, image creation happens automatically in the background during deployment. You can manually specify the image.

If you run into problems with the deployment, you can deploy on your local development environment for troubleshooting and debugging.

For more information, see [Deploy models](#) and [Troubleshooting deployments](#).

Convert and optimize models

Converting your model to [Open Neural Network Exchange](#) (ONNX) may improve performance. On average,

converting to ONNX can yield a 2x performance increase.

For more information on ONNX with Azure Machine Learning, see the [Create and accelerate ML models](#) article.

Use models

Trained machine learning models are deployed as web services in the cloud or locally. Deployments use CPU, GPU, or field-programmable gate arrays (FPGA) for inferencing. You can also use models from Power BI.

When using a model as a web service, you provide the following items:

- The model(s) that are used to score data submitted to the service/device.
- An entry script. This script accepts requests, uses the model(s) to score the data, and return a response.
- An Azure Machine Learning environment that describes the pip and Conda dependencies required by the model(s) and entry script.
- Any additional assets such as text, data, etc. that are required by the model(s) and entry script.

You also provide the configuration of the target deployment platform. For example, the VM family type, available memory, and number of cores when deploying to Azure Kubernetes Service.

When the image is created, components required by Azure Machine Learning are also added. For example, assets needed to run the web service.

Batch scoring

Batch scoring is supported through ML pipelines. For more information, see [Batch predictions on big data](#).

Real-time web services

You can use your models in **web services** with the following compute targets:

- Azure Container Instance
- Azure Kubernetes Service
- Local development environment

To deploy the model as a web service, you must provide the following items:

- The model or ensemble of models.
- Dependencies required to use the model. For example, a script that accepts requests and invokes the model, conda dependencies, etc.
- Deployment configuration that describes how and where to deploy the model.

For more information, see [Deploy models](#).

Controlled rollout

When deploying to Azure Kubernetes Service, you can use controlled rollout to enable the following scenarios:

- Create multiple versions of an endpoint for a deployment
- Perform A/B testing by routing traffic to different versions of the endpoint.
- Switch between endpoint versions by updating the traffic percentage in endpoint configuration.

For more information, see [Controlled rollout of ML models](#).

Analytics

Microsoft Power BI supports using machine learning models for data analytics. For more information, see [Azure Machine Learning integration in Power BI \(preview\)](#).

Capture the governance data required for MLOps

Azure ML gives you the capability to track the end-to-end audit trail of all of your ML assets by using metadata.

- Azure ML [integrates with Git](#) to track information on which repository / branch / commit your code came from.
- [Azure ML Datasets](#) help you track, profile, and version data.
- [Interpretability](#) allows you to explain your models, meet regulatory compliance, and understand how models arrive at a result for given input.
- Azure ML Run history stores a snapshot of the code, data, and computes used to train a model.
- The Azure ML Model Registry captures all of the metadata associated with your model (which experiment trained it, where it is being deployed, if its deployments are healthy).
- [Integration with Azure](#) allows you to act on events in the ML lifecycle. For example, model registration, deployment, data drift, and training (run) events.

TIP

While some information on models and datasets is automatically captured, you can add additional information by using [tags](#). When looking for registered models and datasets in your workspace, you can use tags as a filter.

Associating a dataset with a registered model is an optional step. For information on referencing a dataset when registering a model, see the [Model class reference](#).

Notify, automate, and alert on events in the ML lifecycle

Azure ML publishes key events to Azure Event Grid, which can be used to notify and automate on events in the ML lifecycle. For more information, please see [this document](#).

Monitor for operational & ML issues

Monitoring enables you to understand what data is being sent to your model, and the predictions that it returns.

This information helps you understand how your model is being used. The collected input data may also be useful in training future versions of the model.

For more information, see [How to enable model data collection](#).

Retrain your model on new data

Often, you'll want to validate your model, update it, or even retrain it from scratch, as you receive new information. Sometimes, receiving new data is an expected part of the domain. Other times, as discussed in [Detect data drift \(preview\) on datasets](#), model performance can degrade in the face of such things as changes to a particular sensor, natural data changes such as seasonal effects, or features shifting in their relation to other features.

There is no universal answer to "How do I know if I should retrain?" but Azure ML event and monitoring tools previously discussed are good starting points for automation. Once you have decided to retrain, you should:

- Preprocess your data using a repeatable, automated process
- Train your new model
- Compare the outputs of your new model to those of your old model
- Use predefined criteria to choose whether to replace your old model

A theme of the above steps is that your retraining should be automated, not ad hoc. [Azure Machine Learning pipelines](#) are a good answer for creating workflows relating to data preparation, training, validation, and deployment. Read [Retrain models with Azure Machine Learning designer](#) to see how pipelines and the Azure Machine Learning designer fit into a retraining scenario.

Automate the ML lifecycle

You can use GitHub and Azure Pipelines to create a continuous integration process that trains a model. In a typical scenario, when a Data Scientist checks a change into the Git repo for a project, the Azure Pipeline will start a training run. The results of the run can then be inspected to see the performance characteristics of the trained model. You can also create a pipeline that deploys the model as a web service.

The [Azure Machine Learning extension](#) makes it easier to work with Azure Pipelines. It provides the following enhancements to Azure Pipelines:

- Enables workspace selection when defining a service connection.
- Enables release pipelines to be triggered by trained models created in a training pipeline.

For more information on using Azure Pipelines with Azure Machine Learning, see the following links:

- [Continuous integration and deployment of ML models with Azure Pipelines](#)
- [Azure Machine Learning MLOps repository](#)
- [Azure Machine Learning MLOpsPython repository](#)

You can also use Azure Data Factory to create a data ingestion pipeline that prepares data for use with training. For more information, see [Data ingestion pipeline](#).

Next steps

Learn more by reading and exploring the following resources:

- [How & where to deploy models](#) with Azure Machine Learning
- [Tutorial: Train and deploy a model.](#)
- [End-to-end MLOps examples repo](#)
- [CI/CD of ML models with Azure Pipelines](#)
- Create clients that [consume a deployed model](#)
- [Machine learning at scale](#)
- [Azure AI reference architectures & best practices rep](#)

MLflow and Azure Machine Learning (v1)

9/21/2022 • 2 minutes to read • [Edit Online](#)

APPLIES TO: Azure CLI ml extension v1 Python SDK azureml v1

MLflow is an open-source library for managing the life cycle of your machine learning experiments. MLflow's tracking URI and logging API are collectively known as [MLflow Tracking](#). This component of MLflow logs and tracks your training run metrics and model artifacts, no matter where your experiment's environment is--on your computer, on a remote compute target, on a virtual machine, or in an Azure Databricks cluster.

Together, MLflow Tracking and Azure Machine learning allow you to track an experiment's run metrics and store model artifacts in your Azure Machine Learning workspace.

Compare MLflow and Azure Machine Learning clients

The following table summarizes the clients that can use Azure Machine Learning and their respective capabilities.

MLflow Tracking offers metric logging and artifact storage functionalities that are otherwise available only through the [Azure Machine Learning Python SDK](#).

CAPABILITY	MLFLOW TRACKING AND DEPLOYMENT	AZURE MACHINE LEARNING PYTHON SDK	AZURE MACHINE LEARNING CLI	AZURE MACHINE LEARNING STUDIO
Manage a workspace		✓	✓	✓
Use data stores		✓	✓	
Log metrics	✓	✓		
Upload artifacts	✓	✓		
View metrics	✓	✓	✓	✓
Manage compute		✓	✓	✓
Deploy models	✓	✓	✓	✓
Monitor model performance		✓		
Detect data drift		✓		✓

Track experiments

With MLflow Tracking, you can connect Azure Machine Learning as the back end of your MLflow experiments.

You can then do the following tasks:

- Track and log experiment metrics and artifacts in your [Azure Machine Learning workspace](#). If you already use MLflow Tracking for your experiments, the workspace provides a centralized, secure, and scalable

location to store training metrics and models. Learn more at [Track machine learning models with MLflow and Azure Machine Learning](#).

- Track and manage models in MLflow and the Azure Machine Learning model registry.
- [Track Azure Databricks training runs](#).

Train MLflow projects (preview)

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

You can use MLflow Tracking to submit training jobs with [MLflow Projects](#) and Azure Machine Learning backend support (preview). You can submit jobs locally with Azure Machine Learning tracking or migrate your runs to the cloud via [Azure Machine Learning compute](#).

Learn more at [Train machine learning models with MLflow projects and Azure Machine Learning \(preview\)](#).

Deploy MLflow experiments

You can [deploy your MLflow model as an Azure web service](#) so that you can apply the model management and data drift detection capabilities in Azure Machine Learning to your production models.

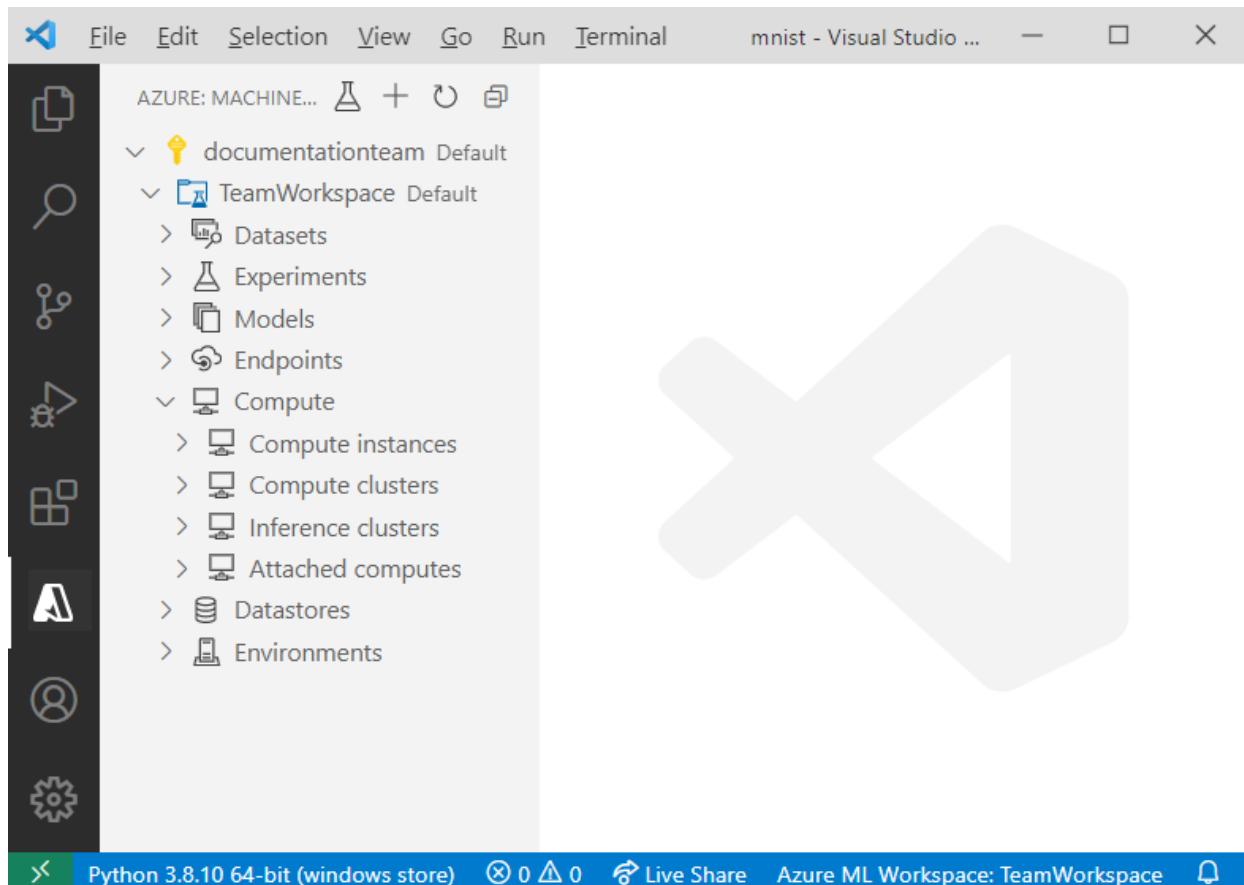
Next steps

- [Track machine learning models with MLflow and Azure Machine Learning](#)
- [Train machine learning models with MLflow projects and Azure Machine Learning \(preview\)](#)
- [Track Azure Databricks runs with MLflow](#)
- [Deploy models with MLflow](#)

Manage Azure Machine Learning resources with the VS Code Extension (preview)

9/21/2022 • 11 minutes to read • [Edit Online](#)

Learn how to manage Azure Machine Learning resources with the VS Code extension.



Prerequisites

- Azure subscription. If you don't have one, sign up to try the [free or paid version of Azure Machine Learning](#).
- Visual Studio Code. If you don't have it, [install it](#).
- Azure Machine Learning extension. Follow the [Azure Machine Learning VS Code extension installation guide](#) to set up the extension.

Create resources

The quickest way to create resources is using the extension's toolbar.

1. Open the Azure Machine Learning view.
2. Select + in the activity bar.
3. Choose your resource from the dropdown list.
4. Configure the specification file. The information required depends on the type of resource you want to create.
5. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, you can create a resource by using the command palette:

1. Open the command palette **View > Command Palette**

2. Enter `> Azure ML: Create <RESOURCE-TYPE>` into the text box. Replace `<RESOURCE-TYPE>` with the type of resource you want to create.
3. Configure the specification file.
4. Open the command palette **View > Command Palette**
5. Enter `> Azure ML: Create Resource` into the text box.

Version resources

Some resources like environments, datasets, and models allow you to make changes to a resource and store the different versions.

To version a resource:

1. Use the existing specification file that created the resource or follow the create resources process to create a new specification file.
2. Increment the version number in the template.
3. Right-click the specification file and select **Azure ML: Execute YAML**.

As long as the name of the updated resource is the same as the previous version, Azure Machine Learning picks up the changes and creates a new version.

Workspaces

For more information, see [workspaces](#).

Create a workspace

1. In the Azure Machine Learning view, right-click your subscription node and select **Create Workspace**.
2. A specification file appears. Configure the specification file.
3. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Workspace` command in the command palette.

Remove workspace

1. Expand the subscription node that contains your workspace.
2. Right-click the workspace you want to remove.
3. Select whether you want to remove:
 - *Only the workspace*: This option deletes **only** the workspace Azure resource. The resource group, storage accounts, and any other resources the workspace was attached to are still in Azure.
 - *With associated resources*: This option deletes the workspace **and** all resources associated with it.

Alternatively, use the `> Azure ML: Remove Workspace` command in the command palette.

Datastores

The extension currently supports datastores of the following types:

- Azure Blob
- Azure Data Lake Gen 1
- Azure Data Lake Gen 2
- Azure File

For more information, see [datastore](#).

Create a datastore

1. Expand the subscription node that contains your workspace.
2. Expand the workspace node you want to create the datastore under.
3. Right-click the **Datastores** node and select **Create Datastore**.
4. Choose the datastore type.
5. A specification file appears. Configure the specification file.
6. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Datastore` command in the command palette.

Manage a datastore

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Datastores** node inside your workspace.
4. Right-click the datastore you want to:
 - *Unregister Datastore*. Removes datastore from your workspace.
 - *View Datastore*. Display read-only datastore settings

Alternatively, use the `> Azure ML: Unregister Datastore` and `> Azure ML: View Datastore` commands respectively in the command palette.

Datasets

The extension currently supports the following dataset types:

- *Tabular*: Allows you to materialize data into a DataFrame.
- *File*: A file or collection of files. Allows you to download or mount files to your compute.

For more information, see [datasets](#)

Create dataset

1. Expand the subscription node that contains your workspace.
2. Expand the workspace node you want to create the dataset under.
3. Right-click the **Datasets** node and select **Create Dataset**.
4. A specification file appears. Configure the specification file.
5. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Dataset` command in the command palette.

Manage a dataset

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Datasets** node.
4. Right-click the dataset you want to:
 - **View Dataset Properties**. Lets you view metadata associated with a specific dataset. If you have multiple version of a dataset, you can choose to only view the dataset properties of a specific version by expanding the dataset node and performing the same steps described in this section on the version of interest.
 - **Preview dataset**. View your dataset directly in the VS Code Data Viewer. Note that this option is only available for tabular datasets.
 - **Unregister dataset**. Removes a dataset and all versions of it from your workspace.

Alternatively, use the `> Azure ML: View Dataset Properties` and `> Azure ML: Unregister Dataset` commands

respectively in the command palette.

Environments

For more information, see [environments](#).

Create environment

1. Expand the subscription node that contains your workspace.
2. Expand the workspace node you want to create the datastore under.
3. Right-click the **Environments** node and select **Create Environment**.
4. A specification file appears. Configure the specification file.
5. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Environment` command in the command palette.

View environment configurations

To view the dependencies and configurations for a specific environment in the extension:

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Environments** node.
4. Right-click the environment you want to view and select **View Environment**.

Alternatively, use the `> Azure ML: View Environment` command in the command palette.

Experiments

For more information, see [experiments](#).

Create job

The quickest way to create a job is by clicking the **Create Job** icon in the extension's activity bar.

Using the resource nodes in the Azure Machine Learning view:

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Right-click the **Experiments** node in your workspace and select **Create Job**.
4. Choose your job type.
5. A specification file appears. Configure the specification file.
6. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Job` command in the command palette.

View job

To view your job in Azure Machine Learning studio:

1. Expand the subscription node that contains your workspace.
2. Expand the **Experiments** node inside your workspace.
3. Right-click the experiment you want to view and select **View Experiment in Studio**.
4. A prompt appears asking you to open the experiment URL in Azure Machine Learning studio. Select **Open**.

Alternatively, use the `> Azure ML: View Experiment in Studio` command respectively in the command palette.

Track job progress

As you're running your job, you may want to see its progress. To track the progress of a job in Azure Machine

Learning studio from the extension:

1. Expand the subscription node that contains your workspace.
2. Expand the **Experiments** node inside your workspace.
3. Expand the job node you want to track progress for.
4. Right-click the job and select **View Job in Studio**.
5. A prompt appears asking you to open the job URL in Azure Machine Learning studio. Select **Open**.

Download job logs & outputs

Once a job is complete, you may want to download the logs and assets such as the model generated as part of a job.

1. Expand the subscription node that contains your workspace.
2. Expand the **Experiments** node inside your workspace.
3. Expand the job node you want to download logs and outputs for.
4. Right-click the job:
 - To download the outputs, select **Download outputs**.
 - To download the logs, select **Download logs**.

Alternatively, use the `> Azure ML: Download Outputs` and `> Azure ML: Download Logs` commands respectively in the command palette.

Compute instances

For more information, see [compute instances](#).

Create compute instance

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Compute** node.
4. Right-click the **Compute instances** node in your workspace and select **Create Compute**.
5. A specification file appears. Configure the specification file.
6. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Compute` command in the command palette.

Connect to compute instance

To use a compute instance as a development environment or remote Jupyter server, see [Connect to a compute instance](#).

Stop or restart compute instance

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Compute instances** node inside your **Compute** node.
4. Right-click the compute instance you want to stop or restart and select **Stop Compute instance** or **Restart Compute instance** respectively.

Alternatively, use the `> Azure ML: Stop Compute instance` and `> Azure ML: Restart Compute instance` commands respectively in the command palette.

View compute instance configuration

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.

3. Expand the **Compute instances** node inside your **Compute** node.
4. Right-click the compute instance you want to inspect and select **View Compute instance Properties**.

Alternatively, use the `Azure ML: View Compute instance Properties` command in the command palette.

Delete compute instance

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Compute instances** node inside your **Compute** node.
4. Right-click the compute instance you want to delete and select **Delete Compute instance**.

Alternatively, use the `Azure ML: Delete Compute instance` command in the command palette.

Compute clusters

For more information, see [training compute targets](#).

Create compute cluster

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Compute** node.
4. Right-click the **Compute clusters** node in your workspace and select **Create Compute**.
5. A specification file appears. Configure the specification file.
6. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Compute` command in the command palette.

View compute configuration

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Compute clusters** node inside your **Compute** node.
4. Right-click the compute you want to view and select **View Compute Properties**.

Alternatively, use the `> Azure ML: View Compute Properties` command in the command palette.

Delete compute cluster

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Compute clusters** node inside your **Compute** node.
4. Right-click the compute you want to delete and select **Remove Compute**.

Alternatively, use the `> Azure ML: Remove Compute` command in the command palette.

Inference Clusters

For more information, see [compute targets for inference](#).

Manage inference clusters

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Inference clusters** node inside your **Compute** node.
4. Right-click the compute you want to:
 - **View Compute Properties**. Displays read-only configuration data about your attached compute.

- **Detach compute.** Detaches the compute from your workspace.

Alternatively, use the `> Azure ML: View Compute Properties` and `> Azure ML: Detach Compute` commands respectively in the command palette.

Delete inference clusters

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Attached computes** node inside your **Compute** node.
4. Right-click the compute you want to delete and select **Remove Compute**.

Alternatively, use the `> Azure ML: Remove Compute` command in the command palette.

Attached Compute

For more information, see [unmanaged compute](#).

Manage attached compute

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Expand the **Attached computes** node inside your **Compute** node.
4. Right-click the compute you want to:
 - **View Compute Properties.** Displays read-only configuration data about your attached compute.
 - **Detach compute.** Detaches the compute from your workspace.

Alternatively, use the `> Azure ML: View Compute Properties` and `> Azure ML: Detach Compute` commands respectively in the command palette.

Models

For more information, see [models](#)

Create model

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Right-click the **Models** node in your workspace and select **Create Model**.
4. A specification file appears. Configure the specification file.
5. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Model` command in the command palette.

View model properties

1. Expand the subscription node that contains your workspace.
2. Expand the **Models** node inside your workspace.
3. Right-click the model whose properties you want to see and select **View Model Properties**. A file opens in the editor containing your model properties.

Alternatively, use the `> Azure ML: View Model Properties` command in the command palette.

Download model

1. Expand the subscription node that contains your workspace.
2. Expand the **Models** node inside your workspace.
3. Right-click the model you want to download and select **Download Model File**.

Alternatively, use the `> Azure ML: Download Model File` command in the command palette.

Delete a model

1. Expand the subscription node that contains your workspace.
2. Expand the **Models** node inside your workspace.
3. Right-click the model you want to delete and select **Remove Model**.
4. A prompt appears confirming you want to remove the model. Select **Ok**.

Alternatively, use the `> Azure ML: Remove Model` command in the command palette.

Endpoints

For more information, see [endpoints](#).

Create endpoint

1. Expand the subscription node that contains your workspace.
2. Expand your workspace node.
3. Right-click the **Models** node in your workspace and select **Create Endpoint**.
4. Choose your endpoint type.
5. A specification file appears. Configure the specification file.
6. Right-click the specification file and select **Azure ML: Execute YAML**.

Alternatively, use the `> Azure ML: Create Endpoint` command in the command palette.

Delete endpoint

1. Expand the subscription node that contains your workspace.
2. Expand the **Endpoints** node inside your workspace.
3. Right-click the deployment you want to remove and select **Remove Service**.
4. A prompt appears confirming you want to remove the service. Select **Ok**.

Alternatively, use the `> Azure ML: Remove Service` command in the command palette.

View service properties

In addition to creating and deleting deployments, you can view and edit settings associated with the deployment.

1. Expand the subscription node that contains your workspace.
2. Expand the **Endpoints** node inside your workspace.
3. Right-click the deployment you want to manage:
 - To view deployment configuration settings, select **View Service Properties**.

Alternatively, use the `> Azure ML: View Service Properties` command in the command palette.

Next steps

[Train an image classification model](#) with the VS Code extension.

Tutorial: Get started with a Python script in Azure Machine Learning (SDK v1, part 1 of 3)

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO: Python SDK azureml v1

In this tutorial, you run your first Python script in the cloud with Azure Machine Learning. This tutorial is *part 1 of a three-part tutorial series*.

This tutorial avoids the complexity of training a machine learning model. You will run a "Hello World" Python script in the cloud. You will learn how a control script is used to configure and create a run in Azure Machine Learning.

In this tutorial, you will:

- Create and run a "Hello world!" Python script.
- Create a Python control script to submit "Hello world!" to Azure Machine Learning.
- Understand the Azure Machine Learning concepts in the control script.
- Submit and run the "Hello world!" script.
- View your code output in the cloud.

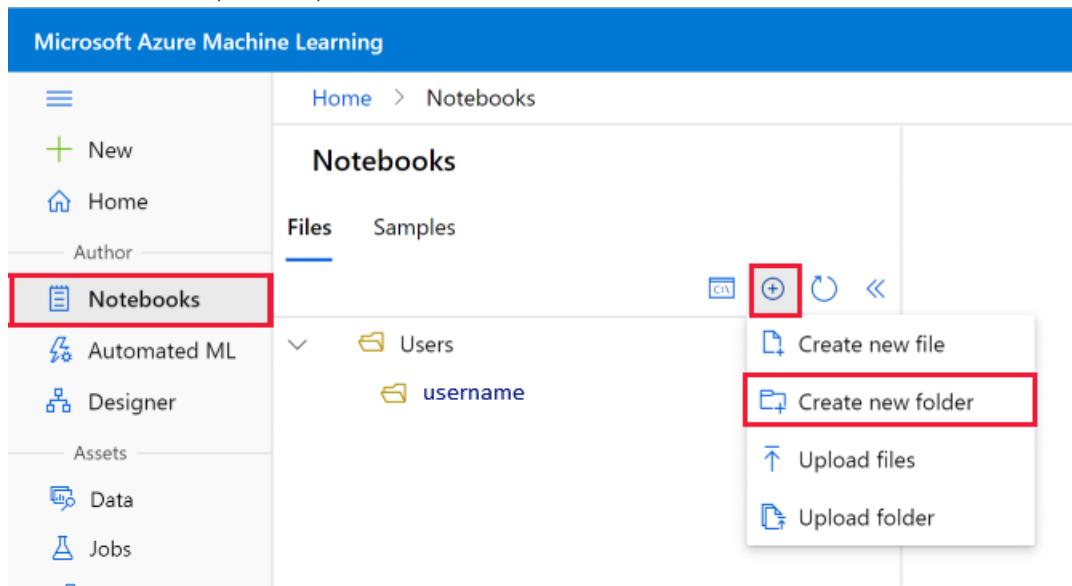
Prerequisites

- Complete [Quickstart: Set up your workspace to get started with Azure Machine Learning](#) to create a workspace, compute instance, and compute cluster to use in this tutorial series.

Create and run a Python script

This tutorial will use the compute instance as your development computer. First create a few folders and the script:

1. Sign in to the [Azure Machine Learning studio](#) and select your workspace if prompted.
2. On the left, select **Notebooks**
3. In the **Files** toolbar, select +, then select **Create new folder**.



4. Name the folder **get-started**.

5. To the right of the folder name, use the ... to create another folder under get-started.

The screenshot shows the Microsoft Azure Machine Learning interface. On the left, there's a sidebar with options like New, Home, Author, Notebooks (which is selected), Automated ML, Designer, Assets, Data, Jobs, Components, Pipelines, Environments, Models, Endpoints, Manage, and Compute. The main area shows a file structure under 'Notebooks': 'Users' > 'username' > 'get-started'. To the right of 'get-started', there's a three-dot menu icon (...). A context menu is open, with 'Create new folder' highlighted by a red box. Other options in the menu include Create new file, Upload files, Upload folder, Rename, Duplicate, Move, Copy folder path, and Delete.

6. Name the new folder **src**. Use the **Edit location** link if the file location is not correct.

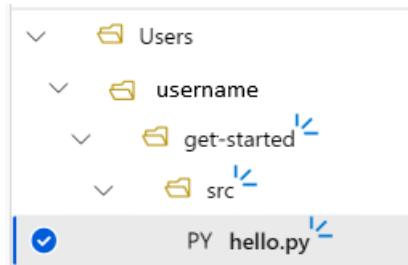
7. To the right of the **src** folder, use the ... to create a new file in the **src** folder.

8. Name your file *hello.py*. Switch the **File type** to *Python (.py)**

Copy this code into your file:

```
# src/hello.py
print("Hello world!")
```

Your project folder structure will now look like:



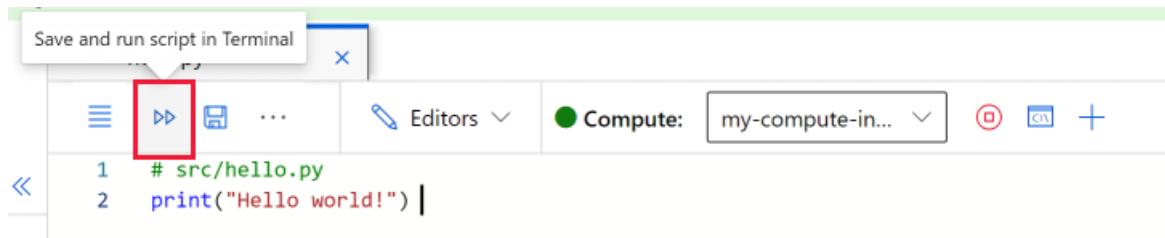
Test your script

You can run your code locally, which in this case means on the compute instance. Running code locally has the benefit of interactive debugging of code.

If you have previously stopped your compute instance, start it now with the **Start compute** tool to the right of the compute dropdown. Wait about a minute for state to change to *Running*.



Select **Save and run script in terminal** to run the script.



You'll see the output of the script in the terminal window that opens. Close the tab and select **Terminate** to close the session.

Create a control script

A *control script* allows you to run your `hello.py` script on different compute resources. You use the control script to control how and where your machine learning code is run.

Select the ... at the end of **get-started** folder to create a new file. Create a Python file called `run-hello.py` and copy/paste the following code into that file:

```
# get-started/run-hello.py
from azureml.core import Workspace, Experiment, Environment, ScriptRunConfig

ws = Workspace.from_config()
experiment = Experiment(workspace=ws, name='day1-experiment-hello')

config = ScriptRunConfig(source_directory='./src', script='hello.py', compute_target='cpu-cluster')

run = experiment.submit(config)
aml_url = run.get_portal_url()
print(aml_url)
```

TIP

If you used a different name when you created your compute cluster, make sure to adjust the name in the code `compute_target='cpu-cluster'` as well.

Understand the code

Here's a description of how the control script works:

```
ws = Workspace.from_config()
```

[Workspace](#) connects to your Azure Machine Learning workspace, so that you can communicate with your Azure Machine Learning resources.

```
experiment = Experiment(...)
```

[Experiment](#) provides a simple way to organize multiple jobs under a single name. Later you can see how experiments make it easy to compare metrics between dozens of jobs.

```
config = ScriptRunConfig(...)
```

`ScriptRunConfig` wraps your `hello.py` code and passes it to your workspace. As the name suggests, you can use this class to *configure* how you want your *script* to *run* in Azure Machine Learning. It also specifies what compute target the script will run on. In this code, the target is the compute cluster that you created in the [setup tutorial](#).

```
run = experiment.submit(config)
```

Submits your script. This submission is called a `run`. In v2, it has been renamed to a job. A run/job encapsulates a single execution of your code. Use a job to monitor the script progress, capture the output, analyze the results, visualize metrics, and more.

```
aml_url = run.get_portal_url()
```

The `run` object provides a handle on the execution of your code. Monitor its progress from the Azure Machine Learning studio with the URL that's printed from the Python script.

Submit and run your code in the cloud

1. Select **Save and run script in terminal** to run your control script, which in turn runs `hello.py` on the compute cluster that you created in the [setup tutorial](#).
2. In the terminal, you may be asked to sign in to authenticate. Copy the code and follow the link to complete this step.
3. Once you're authenticated, you'll see a link in the terminal. Select the link to view the job.

NOTE

You may see some warnings starting with *Failure while loading azureml_run_type_providers....* You can ignore these warnings. Use the link at the bottom of these warnings to view your output.

View the output

1. In the page that opens, you'll see the job status.
2. When the status of the job is **Completed**, select **Output + logs** at the top of the page.
3. Select **std_log.txt** to view the output of your job.

Monitor your code in the cloud in the studio

The output from your script will contain a link to the studio that looks something like this:

```
https://ml.azure.com/experiments/hello-world/runs/<run-id>?wsid=/subscriptions/<subscription-id>/resourcegroups/<resource-group>/workspaces/<workspace-name>
```

Follow the link. At first, you'll see a status of **Queued** or **Preparing**. The very first run will take 5-10 minutes to complete. This is because the following occurs:

- A docker image is built in the cloud
- The compute cluster is resized from 0 to 1 node
- The docker image is downloaded to the compute.

Subsequent jobs are much quicker (~15 seconds) as the docker image is cached on the compute. You can test this by resubmitting the code below after the first job has completed.

Wait about 10 minutes. You'll see a message that the job has completed. Then use **Refresh** to see the status change to *Completed*. Once the job completes, go to the **Outputs + logs** tab. There you can see a `std_log.txt`

file that looks like this:

```
1: [2020-08-04T22:15:44.407305] Entering context manager injector.
2: [context_manager_injector.py] Command line Options: Namespace(inject=
['ProjectPythonPath:context_managers.ProjectPythonPath', 'RunHistory:context_managers.RunHistory',
'TrackUserError:context_managers.TrackUserError', 'UserExceptions:context_managers.UserExceptions'],
invocation=['hello.py'])
3: Starting the daemon thread to refresh tokens in background for process with pid = 31263
4: Entering Job History Context Manager.
5: Preparing to call script [ hello.py ] with arguments: []
6: After variable expansion, calling script [ hello.py ] with arguments: []
7:
8: Hello world!
9: Starting the daemon thread to refresh tokens in background for process with pid = 31263
10:
11:
12: The experiment completed successfully. Finalizing job...
13: Logging experiment finalizing status in history service.
14: [2020-08-04T22:15:46.541334] TimeoutHandler __init__
15: [2020-08-04T22:15:46.541396] TimeoutHandler __enter__
16: Cleaning up all outstanding Job operations, waiting 300.0 seconds
17: 1 items cleaning up...
18: Cleanup took 0.1812913417816162 seconds
19: [2020-08-04T22:15:47.040203] TimeoutHandler __exit__
```

On line 8, you see the "Hello world!" output.

The `70_driver_log.txt` file contains the standard output from a job. This file can be useful when you're debugging remote jobs in the cloud.

Next steps

In this tutorial, you took a simple "Hello world!" script and ran it on Azure. You saw how to connect to your Azure Machine Learning workspace, create an experiment, and submit your `hello.py` code to the cloud.

In the next tutorial, you build on these learnings by running something more interesting than

```
print("Hello world!") .
```

[Tutorial: Train a model](#)

NOTE

If you want to finish the tutorial series here and not progress to the next step, remember to [clean up your resources](#).

Tutorial: Train your first machine learning model (SDK v1, part 2 of 3)

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This tutorial shows you how to train a machine learning model in Azure Machine Learning. This tutorial is *part 2 of a three-part tutorial series*.

In [Part 1: Run "Hello world!"](#) of the series, you learned how to use a control script to run a job in the cloud.

In this tutorial, you take the next step by submitting a script that trains a machine learning model. This example will help you understand how Azure Machine Learning eases consistent behavior between local debugging and remote runs.

In this tutorial, you:

- Create a training script.
- Use Conda to define an Azure Machine Learning environment.
- Create a control script.
- Understand Azure Machine Learning classes ([Environment](#), [Run](#), [Metrics](#)).
- Submit and run your training script.
- View your code output in the cloud.
- Log metrics to Azure Machine Learning.
- View your metrics in the cloud.

Prerequisites

- Completion of [part 1](#) of the series.

Create training scripts

First you define the neural network architecture in a *model.py* file. All your training code will go into the [src](#) subdirectory, including *model.py*.

The training code is taken from [this introductory example](#) from PyTorch. Note that the Azure Machine Learning concepts apply to any machine learning code, not just PyTorch.

1. Create a *model.py* file in the [src](#) subfolder. Copy this code into the file:

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

2. On the toolbar, select **Save** to save the file. Close the tab if you wish.
3. Next, define the training script, also in the `src` subfolder. This script downloads the CIFAR10 dataset by using PyTorch `torchvision.dataset` APIs, sets up the network defined in `model.py`, and trains it for two epochs by using standard SGD and cross-entropy loss.

Create a `train.py` script in the `src` subfolder:

```

import torch
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

from model import Net

# download CIFAR 10 data
trainset = torchvision.datasets.CIFAR10(
    root="../data",
    train=True,
    download=True,
    transform=torchvision.transforms.ToTensor(),
)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=4, shuffle=True, num_workers=2
)

if __name__ == "__main__":
    # define convolutional network
    net = Net()

    # set up pytorch loss / optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    # train the network
    for epoch in range(2):

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # unpack the data
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

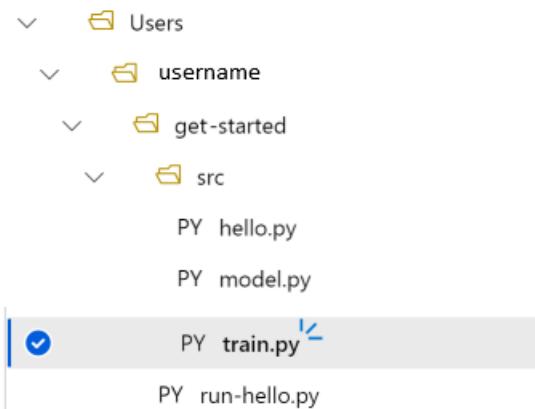
            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:
                loss = running_loss / 2000
                print(f"epoch={epoch + 1}, batch={i + 1:5}: loss {loss:.2f}")
                running_loss = 0.0

    print("Finished Training")

```

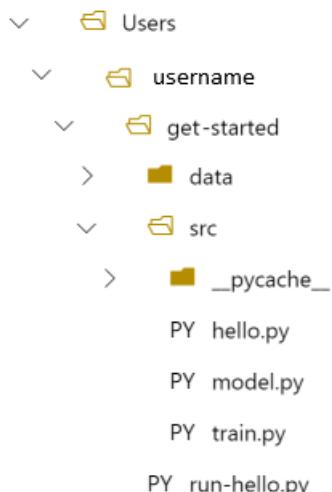
4. You now have the following folder structure:



Test locally

Select **Save and run script in terminal** to run the *train.py* script directly on the compute instance.

After the script completes, select **Refresh** above the file folders. You'll see the new data folder called **get-started/data**. Expand this folder to view the downloaded data.



Create a Python environment

Azure Machine Learning provides the concept of an [environment](#) to represent a reproducible, versioned Python environment for running experiments. It's easy to create an environment from a local Conda or pip environment.

First you'll create a file with the package dependencies.

1. Create a new file in the **get-started** folder called `pytorch-env.yml`:

```
name: pytorch-env
channels:
  - defaults
  - pytorch
dependencies:
  - python=3.6.2
  - pytorch
  - torchvision
```

2. On the toolbar, select **Save** to save the file. Close the tab if you wish.

Create the control script

The difference between the following control script and the one that you used to submit "Hello world!" is that you add a couple of extra lines to set the environment.

Create a new Python file in the **get-started** folder called `run-pytorch.py`:

```
# run-pytorch.py
from azureml.core import Workspace
from azureml.core import Experiment
from azureml.core import Environment
from azureml.core import ScriptRunConfig

if __name__ == "__main__":
    ws = Workspace.from_config()
    experiment = Experiment(workspace=ws, name='day1-experiment-train')
    config = ScriptRunConfig(source_directory='./src',
                            script='train.py',
                            compute_target='cpu-cluster')

    # set up pytorch environment
    env = Environment.from_conda_specification(
        name='pytorch-env',
        file_path='pytorch-env.yml'
    )
    config.run_config.environment = env

    run = experiment.submit(config)

    aml_url = run.get_portal_url()
    print(aml_url)
```

TIP

If you used a different name when you created your compute cluster, make sure to adjust the name in the code `compute_target='cpu-cluster'` as well.

Understand the code changes

`env = ...`

References the dependency file you created above.

`config.run_config.environment = env`

Adds the environment to [ScriptRunConfig](#).

Submit the run to Azure Machine Learning

1. Select **Save and run script in terminal** to run the `run-pytorch.py` script.
2. You'll see a link in the terminal window that opens. Select the link to view the job.

NOTE

You may see some warnings starting with *Failure while loading azureml_run_type_providers...*. You can ignore these warnings. Use the link at the bottom of these warnings to view your output.

View the output

1. In the page that opens, you'll see the job status. The first time you run this script, Azure Machine Learning will build a new Docker image from your PyTorch environment. The whole job might take around 10 minutes to

complete. This image will be reused in future jobs to make them run much quicker.

2. You can see view Docker build logs in the Azure Machine Learning studio. Select the **Outputs + logs** tab, and then select `20_image_build_log.txt`.
3. When the status of the job is **Completed**, select **Output + logs**.
4. Select `std_log.txt` to view the output of your job.

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data/cifar-10-python.tar.gz
Extracting ../data/cifar-10-python.tar.gz to ../data
epoch=1, batch= 2000: loss 2.19
epoch=1, batch= 4000: loss 1.82
epoch=1, batch= 6000: loss 1.66
...
epoch=2, batch= 8000: loss 1.51
epoch=2, batch=10000: loss 1.49
epoch=2, batch=12000: loss 1.46
Finished Training
```

If you see an error `Your total snapshot size exceeds the limit`, the **data** folder is located in the `source_directory` value used in `ScriptRunConfig`.

Select the ... at the end of the folder, then select **Move** to move **data** to the **get-started** folder.

Log training metrics

Now that you have a model training in Azure Machine Learning, start tracking some performance metrics.

The current training script prints metrics to the terminal. Azure Machine Learning provides a mechanism for logging metrics with more functionality. By adding a few lines of code, you gain the ability to visualize metrics in the studio and to compare metrics between multiple jobs.

Modify `train.py` to include logging

1. Modify your `train.py` script to include two more lines of code:

```

import torch
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from model import Net
from azureml.core import Run

# ADDITIONAL CODE: get run from the current context
run = Run.get_context()

# download CIFAR 10 data
trainset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
trainloader = torch.utils.data.DataLoader(
    trainset,
    batch_size=4,
    shuffle=True,
    num_workers=2
)

if __name__ == "__main__":
    # define convolutional network
    net = Net()
    # set up pytorch loss / optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
    # train the network
    for epoch in range(2):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # unpack the data
            inputs, labels = data
            # zero the parameter gradients
            optimizer.zero_grad()
            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:
                loss = running_loss / 2000
                # ADDITIONAL CODE: log loss metric to AML
                run.log('loss', loss)
                print(f'epoch={epoch + 1}, batch={i + 1:5}: loss {loss:.2f}')
                running_loss = 0.0
        print('Finished Training')

```

2. Save this file, then close the tab if you wish.

Understand the additional two lines of code

In *train.py*, you access the *run* object from *within* the training script itself by using the `Run.get_context()` method and use it to log metrics:

```
# ADDITIONAL CODE: get run from the current context
run = Run.get_context()

...
# ADDITIONAL CODE: log loss metric to AML
run.log('loss', loss)
```

Metrics in Azure Machine Learning are:

- Organized by experiment and run, so it's easy to keep track of and compare metrics.
- Equipped with a UI so you can visualize training performance in the studio.
- Designed to scale, so you keep these benefits even as you run hundreds of experiments.

Update the Conda environment file

The `train.py` script just took a new dependency on `azureml.core`. Update `pytorch-env.yml` to reflect this change:

```
name: pytorch-env
channels:
  - defaults
  - pytorch
dependencies:
  - python=3.6.2
  - pytorch
  - torchvision
  - pip
  - pip:
    - azureml-sdk
```

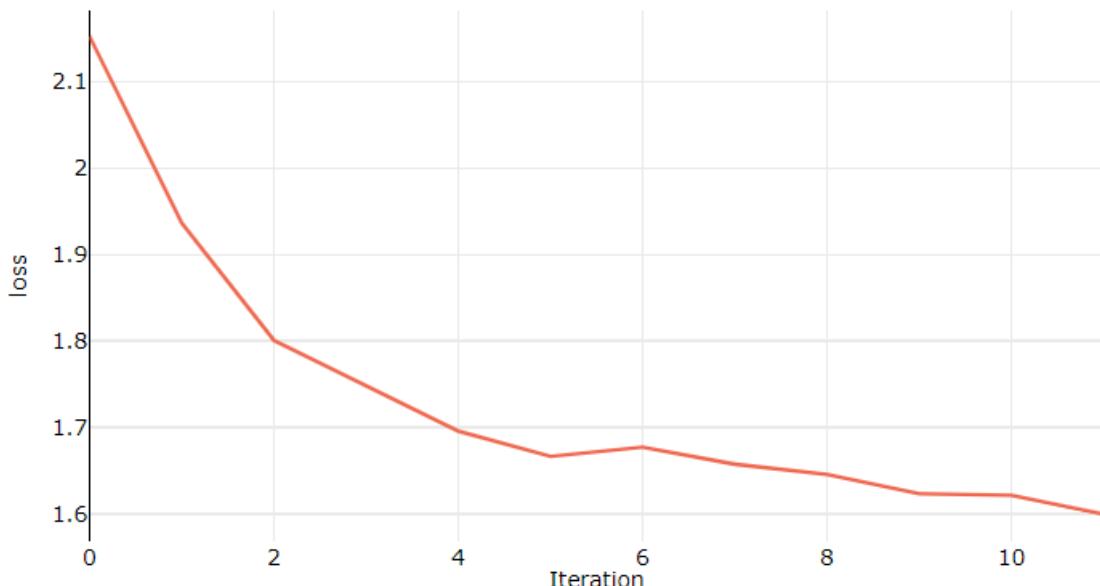
Make sure you save this file before you submit the run.

Submit the run to Azure Machine Learning

Select the tab for the `run-pytorch.py` script, then select **Save and run script in terminal** to re-run the `run-pytorch.py` script. Make sure you've saved your changes to `pytorch-env.yml` first.

This time when you visit the studio, go to the **Metrics** tab where you can now see live updates on the model training loss! It may take a 1 to 2 minutes before the training begins.

loss



Next steps

In this session, you upgraded from a basic "Hello world!" script to a more realistic training script that required a specific Python environment to run. You saw how to use curated Azure Machine Learning environments. Finally, you saw how in a few lines of code you can log metrics to Azure Machine Learning.

There are other ways to create Azure Machine Learning environments, including [from a pip requirements.txt file](#) or [from an existing local Conda environment](#).

In the next session, you'll see how to work with data in Azure Machine Learning by uploading the CIFAR10 dataset to Azure.

[Tutorial: Bring your own data](#)

NOTE

If you want to finish the tutorial series here and not progress to the next step, remember to [clean up your resources](#).

Tutorial: Upload data and train a model (SDK v1, part 3 of 3)

9/21/2022 • 7 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This tutorial shows you how to upload and use your own data to train machine learning models in Azure Machine Learning. This tutorial is *part 3 of a three-part tutorial series*.

In [Part 2: Train a model](#), you trained a model in the cloud, using sample data from `PyTorch`. You also downloaded that data through the `torchvision.datasets.CIFAR10` method in the PyTorch API. In this tutorial, you'll use the downloaded data to learn the workflow for working with your own data in Azure Machine Learning.

In this tutorial, you:

- Upload data to Azure.
- Create a control script.
- Understand the new Azure Machine Learning concepts (passing parameters, datasets, datastores).
- Submit and run your training script.
- View your code output in the cloud.

Prerequisites

You'll need the data that was downloaded in the previous tutorial. Make sure you have completed these steps:

1. [Create the training script](#).
2. [Test locally](#).

Adjust the training script

By now you have your training script (`get-started/src/train.py`) running in Azure Machine Learning, and you can monitor the model performance. Let's parameterize the training script by introducing arguments. Using arguments will allow you to easily compare different hyperparameters.

Our training script is currently set to download the CIFAR10 dataset on each run. The following Python code has been adjusted to read the data from a directory.

NOTE

The use of `argparse` parameterizes the script.

1. Open `train.py` and replace it with this code:

```
import os
import argparse
import torch
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from model import Net
from azureml.core import Run
```

```

from gaudi.cmscore import Run
run = Run.get_context()
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--data_path',
        type=str,
        help='Path to the training data'
    )
    parser.add_argument(
        '--learning_rate',
        type=float,
        default=0.001,
        help='Learning rate for SGD'
    )
    parser.add_argument(
        '--momentum',
        type=float,
        default=0.9,
        help='Momentum for SGD'
    )
args = parser.parse_args()
print("===== DATA =====")
print("DATA PATH: " + args.data_path)
print("LIST FILES IN DATA PATH...")
print(os.listdir(args.data_path))
print("=====")
# prepare DataLoader for CIFAR10 data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
trainset = torchvision.datasets.CIFAR10(
    root=args.data_path,
    train=True,
    download=False,
    transform=transform,
)
trainloader = torch.utils.data.DataLoader(
    trainset,
    batch_size=4,
    shuffle=True,
    num_workers=2
)
# define convolutional network
net = Net()
# set up pytorch loss / optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(
    net.parameters(),
    lr=args.learning_rate,
    momentum=args.momentum,
)
# train the network
for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # unpack the data
        inputs, labels = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:
            loss = running_loss / 2000

```

```

    loss = running_loss / 2000
    run.log('loss', loss) # log loss metric to AML
    print(f'epoch={epoch + 1}, batch={i + 1:5}: loss {loss:.2f}')
    running_loss = 0.0
print('Finished Training')

```

- Save the file. Close the tab if you wish.

Understanding the code changes

The code in `train.py` has used the `argparse` library to set up `data_path`, `learning_rate`, and `momentum`.

```

# .... other code
parser = argparse.ArgumentParser()
parser.add_argument('--data_path', type=str, help='Path to the training data')
parser.add_argument('--learning_rate', type=float, default=0.001, help='Learning rate for SGD')
parser.add_argument('--momentum', type=float, default=0.9, help='Momentum for SGD')
args = parser.parse_args()
# ... other code

```

Also, the `train.py` script was adapted to update the optimizer to use the user-defined parameters:

```

optimizer = optim.SGD(
    net.parameters(),
    lr=args.learning_rate,      # get learning rate from command-line argument
    momentum=args.momentum,    # get momentum from command-line argument
)

```

Upload the data to Azure

To run this script in Azure Machine Learning, you need to make your training data available in Azure. Your Azure Machine Learning workspace comes equipped with a *default* datastore. This is an Azure Blob Storage account where you can store your training data.

NOTE

Azure Machine Learning allows you to connect other cloud-based datastores that store your data. For more details, see the [datastores documentation](#).

- Create a new Python control script in the `get-started` folder (make sure it is in `get-started`, *not* in the `/src` folder). Name the script `upload-data.py` and copy this code into the file:

```

# upload-data.py
from azureml.core import Workspace
from azureml.core import Dataset
from azureml.data.datapath import DataPath

ws = Workspace.from_config()
datastore = ws.get_default_datastore()
Dataset.File.upload_directory(src_dir='data',
                              target=DataPath(datastore, "datasets/cifar10"))
)

```

The `target_path` value specifies the path on the datastore where the CIFAR10 data will be uploaded.

TIP

While you're using Azure Machine Learning to upload the data, you can use [Azure Storage Explorer](#) to upload ad hoc files. If you need an ETL tool, you can use [Azure Data Factory](#) to ingest your data into Azure.

2. Select **Save and run script in terminal** to run the *upload-data.py* script.

You should see the following standard output:

```
Uploading ./data\cifar-10-batches-py\data_batch_2
Uploaded ./data\cifar-10-batches-py\data_batch_2, 4 files out of an estimated total of 9
.
.
Uploading ./data\cifar-10-batches-py\data_batch_5
Uploaded ./data\cifar-10-batches-py\data_batch_5, 9 files out of an estimated total of 9
Uploaded 9 files
```

Create a control script

As you've done previously, create a new Python control script called *run-pytorch-data.py* in the **get-started** folder:

```
# run-pytorch-data.py
from azureml.core import Workspace
from azureml.core import Experiment
from azureml.core import Environment
from azureml.core import ScriptRunConfig
from azureml.core import Dataset

if __name__ == "__main__":
    ws = Workspace.from_config()
    datastore = ws.get_default_datastore()
    dataset = Dataset.File.from_files(path=(datastore, 'datasets/cifar10'))

    experiment = Experiment(workspace=ws, name='day1-experiment-data')

    config = ScriptRunConfig(
        source_directory='./src',
        script='train.py',
        compute_target='cpu-cluster',
        arguments=[
            '--data_path', dataset.as_named_input('input').as_mount(),
            '--learning_rate', 0.003,
            '--momentum', 0.92],
    )

    # set up pytorch environment
    env = Environment.from_conda_specification(
        name='pytorch-env',
        file_path='pytorch-env.yml'
    )
    config.run_config.environment = env

    run = experiment.submit(config)
    aml_url = run.get_portal_url()
    print("Submitted to compute cluster. Click link below")
    print("")
    print(aml_url)
```

TIP

If you used a different name when you created your compute cluster, make sure to adjust the name in the code

```
compute_target='cpu-cluster'
```

as well.

Understand the code changes

The control script is similar to the one from [part 3 of this series](#), with the following new lines:

```
dataset = Dataset.File.from_files( ... )
```

A **dataset** is used to reference the data you uploaded to Azure Blob Storage. Datasets are an abstraction layer on top of your data that are designed to improve reliability and trustworthiness.

```
config = ScriptRunConfig(...)
```

ScriptRunConfig is modified to include a list of arguments that will be passed into `train.py`. The `dataset.as_named_input('input').as_mount()` argument means the specified directory will be *mounted* to the compute target.

Submit the run to Azure Machine Learning

Select **Save and run script in terminal** to run the `run-pytorch-data.py` script. This run will train the model on the compute cluster using the data you uploaded.

This code will print a URL to the experiment in the Azure Machine Learning studio. If you go to that link, you'll be able to see your code running.

NOTE

You may see some warnings starting with *Failure while loading azureml_run_type_providers....* You can ignore these warnings. Use the link at the bottom of these warnings to view your output.

Inspect the log file

In the studio, go to the experiment job (by selecting the previous URL output) followed by **Outputs + logs**.

Select the `std_log.txt` file. Scroll down through the log file until you see the following output:

```

Processing 'input'.
Processing dataset FileDataset
{
  "source": [
    "('workspaceblobstore', 'datasets/cifar10')"
  ],
  "definition": [
    "GetDatastoreFiles"
  ],
  "registration": {
    "id": "XXXXX",
    "name": null,
    "version": null,
    "workspace": "Workspace.create(name='XXXX', subscription_id='XXXX', resource_group='X')"
  }
}
Mounting input to /tmp/tmp9kituvp3.
Mounted input to /tmp/tmp9kituvp3 as folder.
Exit __enter__ of DatasetContextManager
Entering Job History Context Manager.
Current directory: /mnt/batch/tasks/shared/LS_root/jobs/dsvm-aml/azureml/tutorial-session-3_1600171983_763c5381/mounts/workspaceblobstore/azureml/tutorial-session-3_1600171983_763c5381
Preparing to call script [ train.py ] with arguments: ['--data_path', '$input', '--learning_rate', '0.003', '--momentum', '0.92']
After variable expansion, calling script [ train.py ] with arguments: ['--data_path', '/tmp/tmp9kituvp3', '--learning_rate', '0.003', '--momentum', '0.92']

Script type = None
===== DATA =====
DATA PATH: /tmp/tmp9kituvp3
LIST FILES IN DATA PATH...
['cifar-10-batches-py', 'cifar-10-python.tar.gz']

```

Notice:

- Azure Machine Learning has mounted Blob Storage to the compute cluster automatically for you.
- The `dataset.as_named_input('input').as_mount()` used in the control script resolves to the mount point.

Clean up resources

If you plan to continue now to another tutorial, or to start your own training jobs, skip to [Next steps](#).

Stop compute instance

If you're not going to use it now, stop the compute instance:

1. In the studio, on the left, select **Compute**.
2. In the top tabs, select **Compute instances**
3. Select the compute instance in the list.
4. On the top toolbar, select **Stop**.

Delete all resources

IMPORTANT

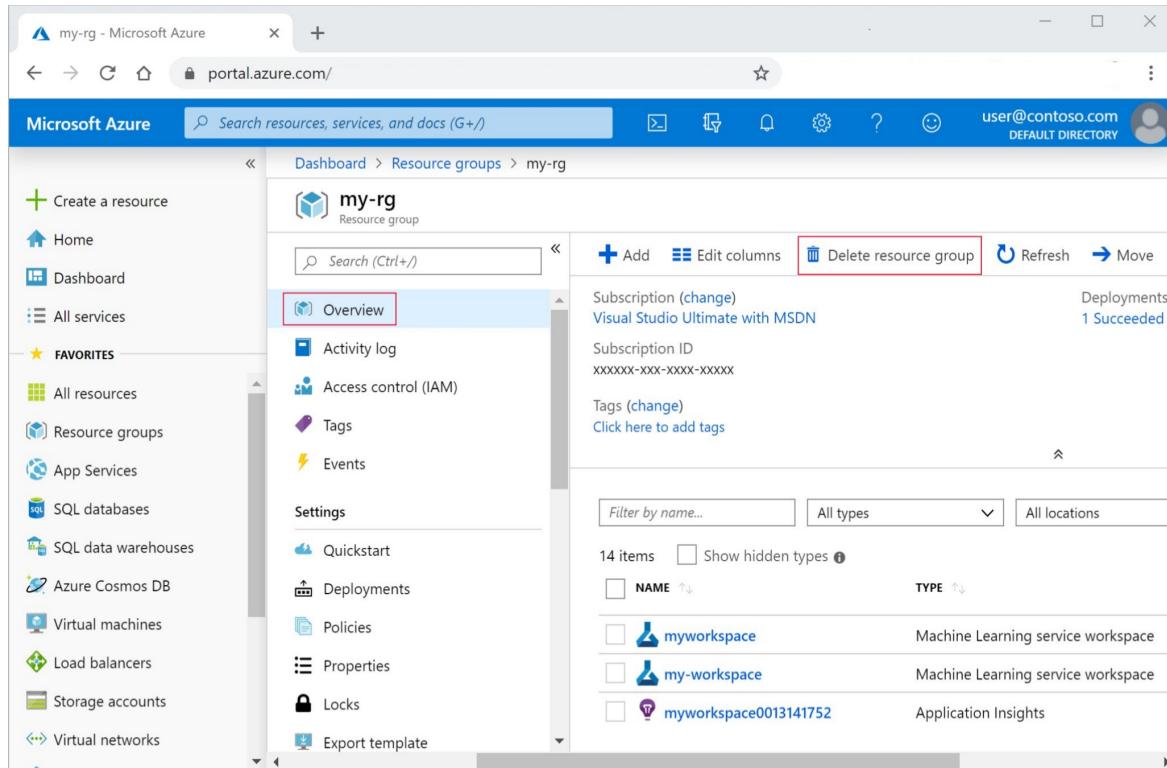
The resources that you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use any of the resources that you created, delete them so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.

2. From the list, select the resource group that you created.

3. Select **Delete resource group**.



The screenshot shows the Microsoft Azure portal interface. The left sidebar has a 'FAVORITES' section with items like All resources, Resource groups, App Services, etc. The main area shows a 'Resource group' named 'my-rg'. The 'Overview' tab is selected. At the top right, there are buttons for Add, Edit columns, Delete resource group (which is highlighted with a red box), Refresh, and Move. Below these are sections for Subscription (Visual Studio Ultimate with MSDN), Deployment (1 Succeeded), Activity log, Access control (IAM), Tags, Events, and Settings. Under Settings, there's a list of 14 items, including 'myworkspace' (Machine Learning service workspace), 'my-workspace' (Machine Learning service workspace), and 'myworkspace0013141752' (Application Insights). A 'Filter by name...' search bar is at the bottom.

4. Enter the resource group name. Then select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties and select **Delete**.

Next steps

In this tutorial, we saw how to upload data to Azure by using **Datastore**. The datastore served as cloud storage for your workspace, giving you a persistent and flexible place to keep your data.

You saw how to modify your training script to accept a data path via the command line. By using **Dataset**, you were able to mount a directory to the remote job.

Now that you have a model, learn:

[How to deploy MLflow models.](#)

Tutorial: Train and deploy an image classification model with an example Jupyter Notebook

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this tutorial, you train a machine learning model on remote compute resources. You'll use the training and deployment workflow for Azure Machine Learning in a Python Jupyter Notebook. You can then use the notebook as a template to train your own machine learning model with your own data.

This tutorial trains a simple logistic regression by using the [MNIST](#) dataset and [scikit-learn](#) with Azure Machine Learning. MNIST is a popular dataset consisting of 70,000 grayscale images. Each image is a handwritten digit of 28 x 28 pixels, representing a number from zero to nine. The goal is to create a multi-class classifier to identify the digit a given image represents.

Learn how to take the following actions:

- Download a dataset and look at the data.
- Train an image classification model and log metrics using MLflow.
- Deploy the model to do real-time inference.

Prerequisites

- Complete the [Quickstart: Get started with Azure Machine Learning](#) to:
 - Create a workspace.
 - Create a cloud-based compute instance to use for your development environment.

Run a notebook from your workspace

Azure Machine Learning includes a cloud notebook server in your workspace for an install-free and pre-configured experience. Use [your own environment](#) if you prefer to have control over your environment, packages, and dependencies.

Clone a notebook folder

You complete the following experiment setup and run steps in Azure Machine Learning studio. This consolidated interface includes machine learning tools to perform data science scenarios for data science practitioners of all skill levels.

1. Sign in to [Azure Machine Learning studio](#).
2. Select your subscription and the workspace you created.
3. On the left, select **Notebooks**.
4. Select the **Open terminal** tool to open a terminal window.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. On the left, there's a sidebar with navigation links: 'Microsoft' (with a back arrow), 'New' (with a plus sign), 'Home' (with a house icon), 'Author' (with a person icon), 'Notebooks' (which is highlighted with a red box), and 'Automated ML'. The main area has a breadcrumb path: 'Microsoft > my-ws > Notebooks'. Below the path, the word 'Notebooks' is displayed in bold. There are two tabs: 'Files' (which is selected) and 'Samples'. A tooltip for 'Open terminal' points to a terminal icon. The 'Files' section shows a folder structure: 'C:\Users\tqpublic'. The 'C:' icon is also highlighted with a red box.

5. On the top bar, select the compute instance you created during the [Quickstart: Get started with Azure Machine Learning](#) to use if it's not already selected. Start the compute instance if it is stopped.
6. In the terminal window, clone the MachineLearningNotebooks repository:

```
git clone --depth 1 https://github.com/Azure/MachineLearningNotebooks
```

7. If necessary, refresh the list of files with the Refresh tool to see the newly cloned folder under your user folder.

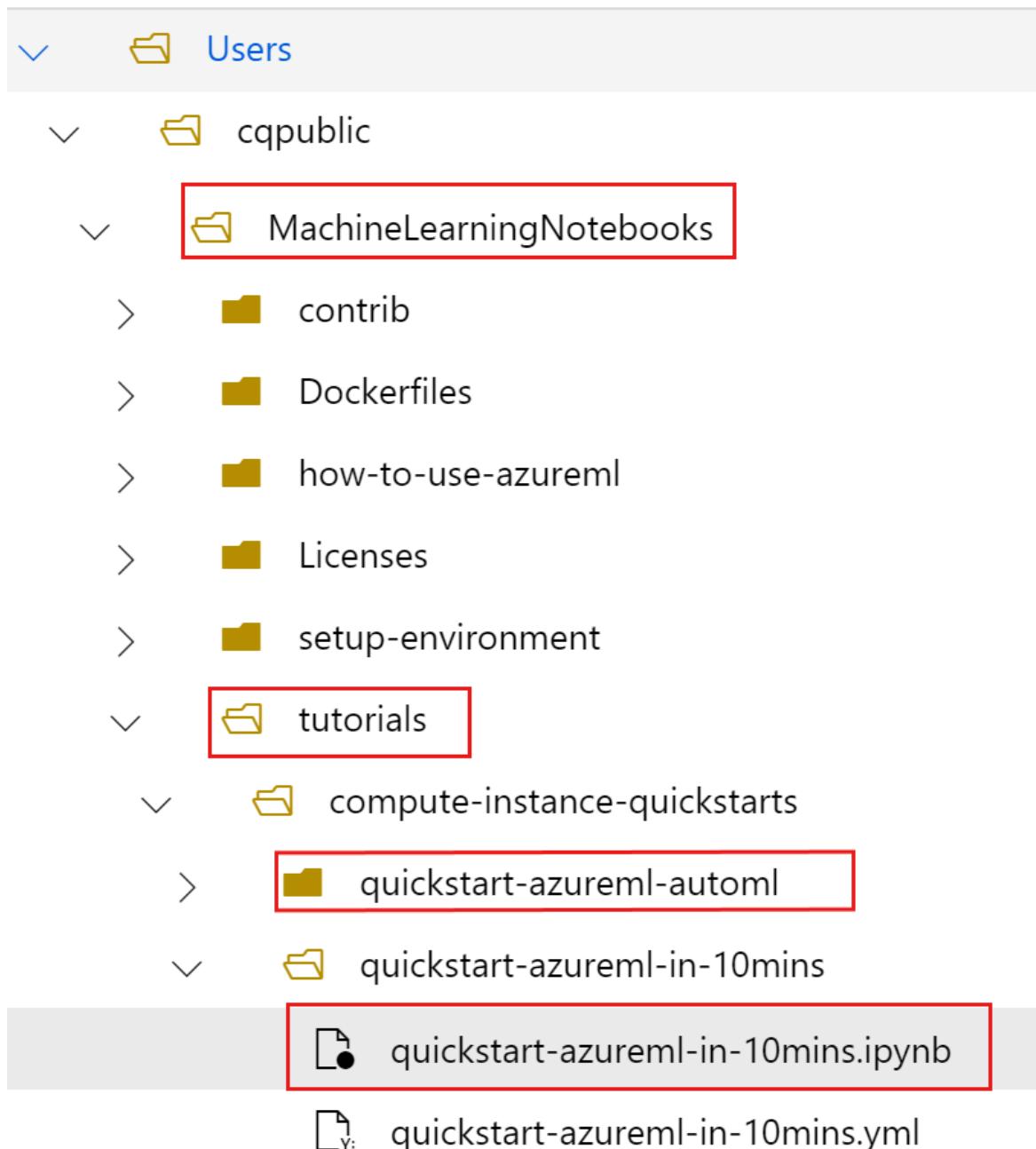
Open the cloned notebook

1. Open the MachineLearningNotebooks folder that was cloned into your Files section.
2. Select the quickstart-azureml-in-10mins.ipynb file from your MachineLearningNotebooks/tutorials/compute-instance-quickstarts/quickstart-azureml-in-10mins folder.

Notebooks

Files Samples

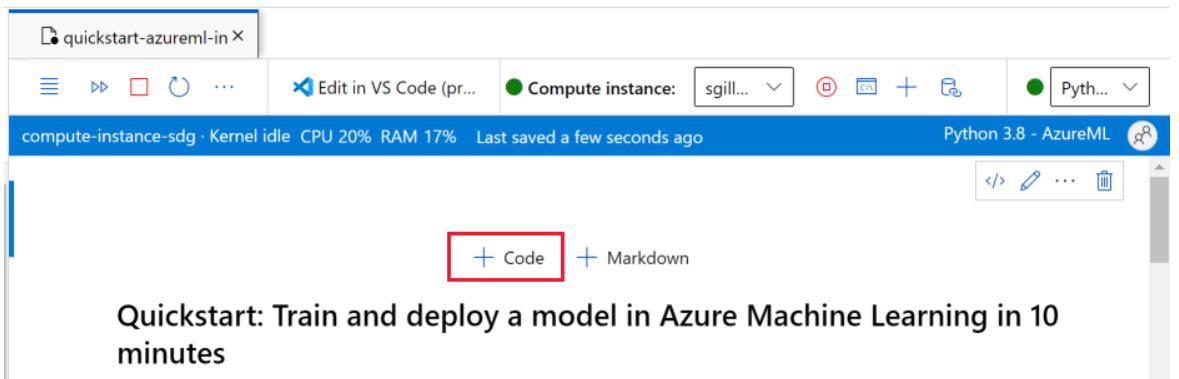
C:\ + ⌂ <



Install packages

Once the compute instance is running and the kernel appears, add a new code cell to install packages needed for this tutorial.

1. At the top of the notebook, add a code cell.



2. Add the following into the cell and then run the cell, either by using the **Run** tool or by using **Shift+Enter**.

```
%pip install scikit-learn==0.22.1  
%pip install scipy==1.5.2
```

You may see a few install warnings. These can safely be ignored.

Run the notebook

This tutorial and accompanying `utils.py` file is also available on [GitHub](#) if you wish to use it on your own [local environment](#). If you aren't using the compute instance, add

```
%pip install azureml-sdk[notebooks] azureml-opendatasets matplotlib
```

IMPORTANT

The rest of this article contains the same content as you see in the notebook.

Switch to the Jupyter Notebook now if you want to run the code while you read along. To run a single code cell in a notebook, click the code cell and hit **Shift+Enter**. Or, run the entire notebook by choosing **Run all** from the top toolbar.

Import data

Before you train a model, you need to understand the data you're using to train it. In this section, learn how to:

- Download the MNIST dataset
- Display some sample images

You'll use Azure Open Datasets to get the raw MNIST data files. Azure Open Datasets are curated public datasets that you can use to add scenario-specific features to machine learning solutions for better models. Each dataset has a corresponding class, `MNIST` in this case, to retrieve the data in different ways.

```
import os  
from azureml.opendatasets import MNIST  
  
data_folder = os.path.join(os.getcwd(), "/tmp/qs_data")  
os.makedirs(data_folder, exist_ok=True)  
  
mnist_file_dataset = MNIST.get_file_dataset()  
mnist_file_dataset.download(data_folder, overwrite=True)
```

Take a look at the data

Load the compressed files into `numpy` arrays. Then use `matplotlib` to plot 30 random images from the dataset

with their labels above them.

Note this step requires a `load_data` function that's included in an `utils.py` file. This file is placed in the same folder as this notebook. The `load_data` function simply parses the compressed files into numpy arrays.

```
from utils import load_data
import matplotlib.pyplot as plt
import numpy as np
import glob

# note we also shrink the intensity values (X) from 0-255 to 0-1. This helps the model converge faster.
X_train = (
    load_data(
        glob.glob(
            os.path.join(data_folder, "**/train-images-idx3-ubyte.gz"), recursive=True
        )[0],
        False,
    )
    / 255.0
)
X_test = (
    load_data(
        glob.glob(
            os.path.join(data_folder, "**/t10k-images-idx3-ubyte.gz"), recursive=True
        )[0],
        False,
    )
    / 255.0
)
y_train = load_data(
    glob.glob(
        os.path.join(data_folder, "**/train-labels-idx1-ubyte.gz"), recursive=True
    )[0],
    True,
).reshape(-1)
y_test = load_data(
    glob.glob(
        os.path.join(data_folder, "**/t10k-labels-idx1-ubyte.gz"), recursive=True
    )[0],
    True,
).reshape(-1)

# now let's show some randomly chosen images from the training set.
count = 0
sample_size = 30
plt.figure(figsize=(16, 6))
for i in np.random.permutation(X_train.shape[0])[:sample_size]:
    count = count + 1
    plt.subplot(1, sample_size, count)
    plt.axhline("")
    plt.axvline("")
    plt.text(x=10, y=-10, s=y_train[i], fontsize=18)
    plt.imshow(X_train[i].reshape(28, 28), cmap=plt.cm.Greys)
plt.show()
```

The code above displays a random set of images with their labels, similar to this:

4	9	9	2	4	6	4	3	8	1	2	8	4	7	2	3	1	7	8	3	2	2	8	1	7	4	2	9	2	9
4	9	9	2	4	6	4	3	8	1	2	8	4	7	2	3	1	7	8	3	2	2	8	1	7	4	2	9	2	9

Train model and log metrics with MLflow

You'll train the model using the code below. Note that you are using MLflow autologging to track metrics and log model artifacts.

You'll be using the [LogisticRegression](#) classifier from the [SciKit Learn framework](#) to classify the data.

NOTE

The model training takes approximately 2 minutes to complete.**

```
# create the model
import mlflow
import numpy as np
from sklearn.linear_model import LogisticRegression
from azureml.core import Workspace

# connect to your workspace
ws = Workspace.from_config()

# create experiment and start logging to a new run in the experiment
experiment_name = "azure-ml-in10-mins-tutorial"

# set up MLflow to track the metrics
mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())
mlflow.set_experiment(experiment_name)
mlflow.autolog()

# set up the Logistic regression model
reg = 0.5
clf = LogisticRegression(
    C=1.0 / reg, solver="liblinear", multi_class="auto", random_state=42
)

# train the model
with mlflow.start_run() as run:
    clf.fit(X_train, y_train)
```

View experiment

In the left-hand menu in Azure Machine Learning studio, select **Jobs** and then select your job (**azure-ml-in10-mins-tutorial**). A job is a grouping of many runs from a specified script or piece of code. Multiple jobs can be grouped together as an experiment.

Information for the run is stored under that job. If the name doesn't exist when you submit a job, if you select your run you will see various tabs containing metrics, logs, explanations, etc.

Version control your models with the model registry

You can use model registration to store and version your models in your workspace. Registered models are identified by name and version. Each time you register a model with the same name as an existing one, the registry increments the version. The code below registers and versions the model you trained above. Once you have executed the code cell below you will be able to see the model in the registry by selecting **Models** in the left-hand menu in Azure Machine Learning studio.

```
# register the model
model_uri = "runs:/{}//model".format(run.info.run_id)
model = mlflow.register_model(model_uri, "sklearn_mnist_model")
```

Deploy the model for real-time inference

In this section you learn how to deploy a model so that an application can consume (inference) the model over REST.

Create deployment configuration

The code cell gets a *curated environment*, which specifies all the dependencies required to host the model (for example, the packages like scikit-learn). Also, you create a *deployment configuration*, which specifies the amount of compute required to host the model. In this case, the compute will have 1CPU and 1GB memory.

```
# create environment for the deploy
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.webservice import AciWebservice

# get a curated environment
env = Environment.get(
    workspace=ws,
    name="AzureML-sklearn-0.24.1-ubuntu18.04-py37-cpu-inference",
    version=1
)
env.inferencing_stack_version='latest'

# create deployment config i.e. compute resources
aciconfig = AciWebservice.deploy_configuration(
    cpu_cores=1,
    memory_gb=1,
    tags={"data": "MNIST", "method": "sklearn"},
    description="Predict MNIST with sklearn",
)
```

Deploy model

This next code cell deploys the model to Azure Container Instance.

NOTE

The deployment takes approximately 3 minutes to complete.**

```

%%time
import uuid
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment
from azureml.core.model import Model

# get the registered model
model = Model(ws, "sklearn_mnist_model")

# create an inference config i.e. the scoring script and environment
inference_config = InferenceConfig(entry_script="score.py", environment=env)

# deploy the service
service_name = "sklearn-mnist-svc-" + str(uuid.uuid4())[:4]
service = Model.deploy(
    workspace=ws,
    name=service_name,
    models=[model],
    inference_config=inference_config,
    deployment_config=aciconfig,
)
service.wait_for_deployment(show_output=True)

```

The scoring script file referenced in the code above can be found in the same folder as this notebook, and has two functions:

1. An `init` function that executes once when the service starts - in this function you normally get the model from the registry and set global variables
2. A `run(data)` function that executes each time a call is made to the service. In this function, you normally format the input data, run a prediction, and output the predicted result.

View endpoint

Once the model has been successfully deployed, you can view the endpoint by navigating to **Endpoints** in the left-hand menu in Azure Machine Learning studio. You will be able to see the state of the endpoint (healthy/unhealthy), logs, and consume (how applications can consume the model).

Test the model service

You can test the model by sending a raw HTTP request to test the web service.

```

# send raw HTTP request to test the web service.
import requests

# send a random row from the test set to score
random_index = np.random.randint(0, len(X_test) - 1)
input_data = '{"data": [' + str(list(X_test[random_index])) + "]}"

headers = {"Content-Type": "application/json"}

resp = requests.post(service.scoring_uri, input_data, headers=headers)

print("POST to url", service.scoring_uri)
print("label:", y_test[random_index])
print("prediction:", resp.text)

```

Clean up resources

If you're not going to continue to use this model, delete the Model service using:

```
# if you want to keep workspace and only delete endpoint (it will incur cost while running)
service.delete()
```

If you want to control cost further, stop the compute instance by selecting the "Stop compute" button next to the **Compute** dropdown. Then start the compute instance again the next time you need it.

Delete everything

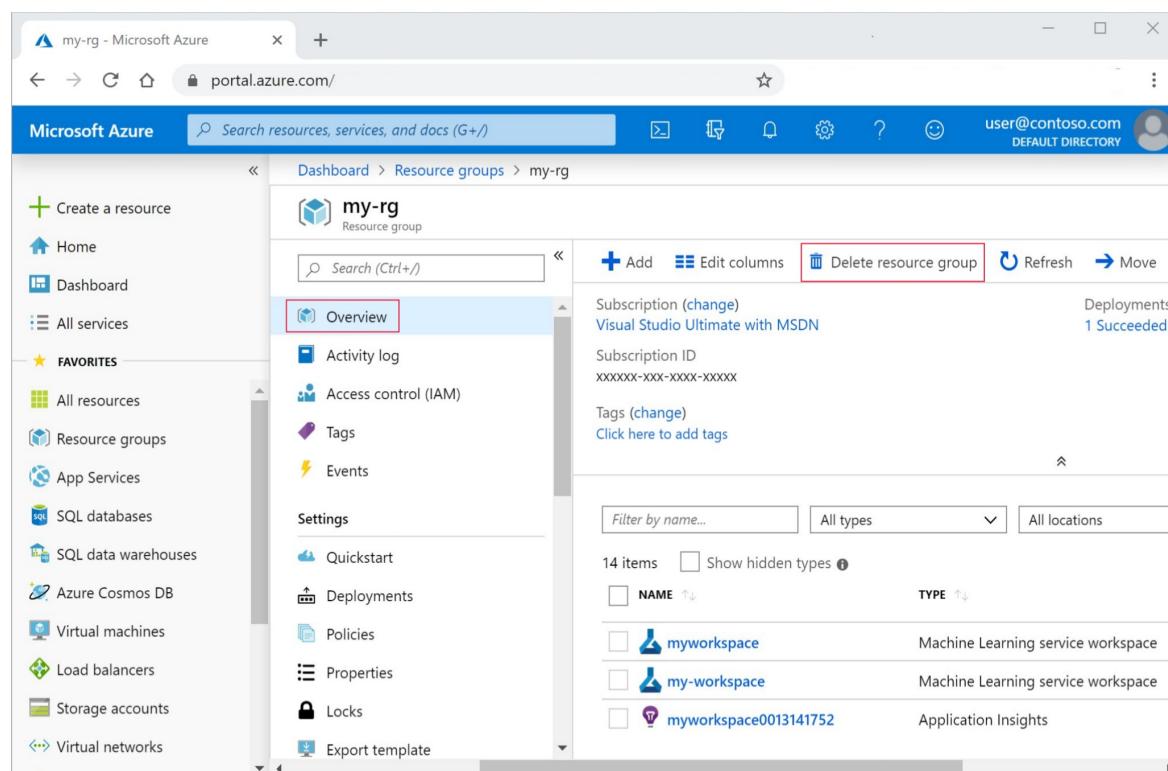
Use these steps to delete your Azure Machine Learning workspace and all compute resources.

IMPORTANT

The resources that you created can be used as prerequisites to other Azure Machine Learning tutorials and how-to articles.

If you don't plan to use any of the resources that you created, delete them so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.
2. From the list, select the resource group that you created.
3. Select **Delete resource group**.



4. Enter the resource group name. Then select **Delete**.

Next steps

- Learn about all of the [deployment options for Azure Machine Learning](#).
- Learn how to [authenticate to the deployed model](#).
- [Make predictions on large quantities of data](#) asynchronously.
- Monitor your Azure Machine Learning models with [Application Insights](#).

Tutorial: Build an Azure Machine Learning pipeline for image classification

9/21/2022 • 14 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

NOTE

For a tutorial that uses SDK v2 to build a pipeline, see [Tutorial: Use ML pipelines for production ML workflows with Python SDK v2 \(preview\) in a Jupyter Notebook](#).

In this tutorial, you learn how to build an [Azure Machine Learning pipeline](#) to prepare data and train a machine learning model. Machine learning pipelines optimize your workflow with speed, portability, and reuse, so you can focus on machine learning instead of infrastructure and automation.

The example trains a small [Keras](#) convolutional neural network to classify images in the [Fashion MNIST](#) dataset.

In this tutorial, you complete the following tasks:

- Configure workspace
- Create an Experiment to hold your work
- Provision a ComputeTarget to do the work
- Create a Dataset in which to store compressed data
- Create a pipeline step to prepare the data for training
- Define a runtime Environment in which to perform training
- Create a pipeline step to define the neural network and perform the training
- Compose a Pipeline from the pipeline steps
- Run the pipeline in the experiment
- Review the output of the steps and the trained neural network
- Register the model for further use

If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#) today.

Prerequisites

- Complete the [Quickstart: Get started with Azure Machine Learning](#) if you don't already have an Azure Machine Learning workspace.
- A Python environment in which you've installed both the `azureml-core` and `azureml-pipeline` packages. This environment is for defining and controlling your Azure Machine Learning resources and is separate from the environment used at runtime for training.

IMPORTANT

Currently, the most recent Python release compatible with `azureml-pipeline` is Python 3.8. If you've difficulty installing the `azureml-pipeline` package, ensure that `python --version` is a compatible release. Consult the documentation of your Python virtual environment manager (`venv`, `conda`, and so on) for instructions.

Start an interactive Python session

This tutorial uses the Python SDK for Azure ML to create and control an Azure Machine Learning pipeline. The tutorial assumes that you'll be running the code snippets interactively in either a Python REPL environment or a Jupyter notebook.

- This tutorial is based on the `image-classification.ipynb` notebook found in the `python-sdk/tutorial/using-pipelines` directory of the [AzureML Examples](#) repository. The source code for the steps themselves is in the `keras-mnist-fashion` subdirectory.

Import types

Import all the Azure Machine Learning types that you'll need for this tutorial:

```
import os
import azureml.core
from azureml.core import (
    Workspace,
    Experiment,
    Dataset,
    Datastore,
    ComputeTarget,
    Environment,
    ScriptRunConfig
)
from azureml.data import OutputFileDatasetConfig
from azureml.core.compute import AmlCompute
from azureml.core.compute_target import ComputeTargetException
from azureml.pipeline.steps import PythonScriptStep
from azureml.pipeline.core import Pipeline

# check core SDK version number
print("Azure ML SDK Version: ", azureml.core.VERSION)
```

The Azure ML SDK version should be 1.37 or greater. If it isn't, upgrade with `pip install --upgrade azureml-core`.

Configure workspace

Create a workspace object from the existing Azure Machine Learning workspace.

```
workspace = Workspace.from_config()
```

IMPORTANT

This code snippet expects the workspace configuration to be saved in the current directory or its parent. For more information on creating a workspace, see [Create workspace resources](#). For more information on saving the configuration to file, see [Create a workspace configuration file](#).

Create the infrastructure for your pipeline

Create an `Experiment` object to hold the results of your pipeline runs:

```
exp = Experiment(workspace=workspace, name="keras-mnist-fashion")
```

Create a `ComputeTarget` that represents the machine resource on which your pipeline will run. The simple neural network used in this tutorial trains in just a few minutes even on a CPU-based machine. If you wish to use a GPU for training, set `use_gpu` to `True`. Provisioning a compute target generally takes about five minutes.

```
use_gpu = False

# choose a name for your cluster
cluster_name = "gpu-cluster" if use_gpu else "cpu-cluster"

found = False
# Check if this compute target already exists in the workspace.
cts = workspace.compute_targets
if cluster_name in cts and cts[cluster_name].type == "AmlCompute":
    found = True
    print("Found existing compute target.")
    compute_target = cts[cluster_name]
if not found:
    print("Creating a new compute target...")
    compute_config = AmlCompute.provisioning_configuration(
        vm_size= "STANDARD_NC6" if use_gpu else "STANDARD_D2_V2"
        # vm_priority = 'lowpriority', # optional
        max_nodes=4,
    )

    # Create the cluster.
    compute_target = ComputeTarget.create(workspace, cluster_name, compute_config)

    # Can poll for a minimum number of nodes and for a specific timeout.
    # If no min_node_count is provided, it will use the scale settings for the cluster.
    compute_target.wait_for_completion(
        show_output=True, min_node_count=None, timeout_in_minutes=10
    )
# For a more detailed view of current AmlCompute status, use
get_status().print(compute_target.get_status().serialize())
```

NOTE

GPU availability depends on the quota of your Azure subscription and upon Azure capacity. See [Manage and increase quotas for resources with Azure Machine Learning](#).

Create a dataset for the Azure-stored data

Fashion-MNIST is a dataset of fashion images divided into 10 classes. Each image is a 28x28 grayscale image and there are 60,000 training and 10,000 test images. As an image classification problem, Fashion-MNIST is harder than the classic MNIST handwritten digit database. It's distributed in the same compressed binary form as the original [handwritten digit database](#).

To create a `Dataset` that references the Web-based data, run:

```
data_urls = ["https://data4mldemo6150520719.blob.core.windows.net/demo/mnist-fashion"]
fashion_ds = Dataset.File.from_files(data_urls)

# list the files referenced by fashion_ds
print(fashion_ds.to_path())
```

This code completes quickly. The underlying data remains in the Azure storage resource specified in the `data_urls` array.

Create the data-preparation pipeline step

The first step in this pipeline will convert the compressed data files of `fashion_ds` into a dataset in your own workspace consisting of CSV files ready for use in training. Once registered with the workspace, your collaborators can access this data for their own analysis, training, and so on

```
datastore = workspace.get_default_datastore()
prepared_fashion_ds = OutputFileDatasetConfig(
    destination=(datastore, "outputdataset/{run-id}")
).register_on_complete(name="prepared_fashion_ds")
```

The above code specifies a dataset that is based on the output of a pipeline step. The underlying processed files will be put in the workspace's default datastore's blob storage at the path specified in `destination`. The dataset will be registered in the workspace with the name `prepared_fashion_ds`.

Create the pipeline step's source

The code that you've executed so far has created and controlled Azure resources. Now it's time to write code that does the first step in the domain.

If you're following along with the example in the [AzureML Examples repo](#), the source file is already available as `keras-mnist-fashion/prepare.py`.

If you're working from scratch, create a subdirectory called `keras-mnist-fashion/`. Create a new file, add the following code to it, and name the file `prepare.py`.

```

# prepare.py
# Converts MNIST-formatted files at the passed-in input path to a passed-in output path
import os
import sys

# Conversion routine for MNIST binary format
def convert(imgf, labelf, outf, n):
    f = open(imgf, "rb")
    l = open(labelf, "rb")
    o = open(outf, "w")

    f.read(16)
    l.read(8)
    images = []

    for i in range(n):
        image = [ord(l.read(1))]
        for j in range(28 * 28):
            image.append(ord(f.read(1)))
        images.append(image)

    for image in images:
        o.write(",".join(str(pix) for pix in image) + "\n")
    f.close()
    o.close()
    l.close()

# The MNIST-formatted source
mounted_input_path = sys.argv[1]
# The output directory at which the outputs will be written
mounted_output_path = sys.argv[2]

# Create the output directory
os.makedirs(mounted_output_path, exist_ok=True)

# Convert the training data
convert(
    os.path.join(mounted_input_path, "mnist-fashion/train-images-idx3-ubyte"),
    os.path.join(mounted_input_path, "mnist-fashion/train-labels-idx1-ubyte"),
    os.path.join(mounted_output_path, "mnist_train.csv"),
    60000,
)
# Convert the test data
convert(
    os.path.join(mounted_input_path, "mnist-fashion/t10k-images-idx3-ubyte"),
    os.path.join(mounted_input_path, "mnist-fashion/t10k-labels-idx1-ubyte"),
    os.path.join(mounted_output_path, "mnist_test.csv"),
    10000,
)

```

The code in `prepare.py` takes two command-line arguments: the first is assigned to `mounted_input_path` and the second to `mounted_output_path`. If that subdirectory doesn't exist, the call to `os.makedirs` creates it. Then, the program converts the training and testing data and outputs the comma-separated files to the `mounted_output_path`.

Specify the pipeline step

Back in the Python environment you're using to specify the pipeline, run this code to create a `PythonScriptStep` for your preparation code:

```

script_folder = "./keras-mnist-fashion"

prep_step = PythonScriptStep(
    name="prepare step",
    script_name="prepare.py",
    # On the compute target, mount fashion_ds dataset as input, prepared_fashion_ds as output
    arguments=[fashion_ds.as_named_input("fashion_ds").as_mount(), prepared_fashion_ds],
    source_directory=script_folder,
    compute_target=compute_target,
    allow_reuse=True,
)

```

The call to `PythonScriptStep` specifies that, when the pipeline step is run:

- All the files in the `script_folder` directory are uploaded to the `compute_target`
- Among those uploaded source files, the file `prepare.py` will be run
- The `fashion_ds` and `prepared_fashion_ds` datasets will be mounted on the `compute_target` and appear as directories
- The path to the `fashion_ds` files will be the first argument to `prepare.py`. In `prepare.py`, this argument is assigned to `mounted_input_path`
- The path to the `prepared_fashion_ds` will be the second argument to `prepare.py`. In `prepare.py`, this argument is assigned to `mounted_output_path`
- Because `allow_reuse` is `True`, it won't be rerun until its source files or inputs change
- This `PythonScriptStep` will be named `prepare step`

Modularity and reuse are key benefits of pipelines. Azure Machine Learning can automatically determine source code or Dataset changes. The output of a step that isn't affected will be reused without rerunning the steps again if `allow_reuse` is `True`. If a step relies on a data source external to Azure Machine Learning that may change (for instance, a URL that contains sales data), set `allow_reuse` to `False` and the pipeline step will run every time the pipeline is run.

Create the training step

Once the data has been converted from the compressed format to CSV files, it can be used for training a convolutional neural network.

Create the training step's source

With larger pipelines, it's a good practice to put each step's source code in a separate directory (`src/prepare/`, `src/train/`, and so on) but for this tutorial, just use or create the file `train.py` in the same `keras-mnist-fashion/` source directory.

```

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
from keras.utils import to_categorical
from keras.callbacks import Callback

import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from azureml.core import Run

# dataset object from the run

```

```

run = Run.get_context()
dataset = run.input_datasets["prepared_fashion_ds"]

# split dataset into train and test set
(train_dataset, test_dataset) = dataset.random_split(percentage=0.8, seed=111)

# load dataset into pandas dataframe
data_train = train_dataset.to_pandas_dataframe()
data_test = test_dataset.to_pandas_dataframe()

img_rows, img_cols = 28, 28
input_shape = (img_rows, img_cols, 1)

X = np.array(data_train.iloc[:, 1:])
y = to_categorical(np.array(data_train.iloc[:, 0]))

# here we split validation data to optimiza classifier during training
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=13)

# test data
X_test = np.array(data_test.iloc[:, 1:])
y_test = to_categorical(np.array(data_test.iloc[:, 0]))

X_train = (
    X_train.reshape(X_train.shape[0], img_rows, img_cols, 1).astype("float32") / 255
)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1).astype("float32") / 255
X_val = X_val.reshape(X_val.shape[0], img_rows, img_cols, 1).astype("float32") / 255

batch_size = 256
num_classes = 10
epochs = 10

# construct neuron network
model = Sequential()
model.add(
    Conv2D(
        32,
        kernel_size=(3, 3),
        activation="relu",
        kernel_initializer="he_normal",
        input_shape=input_shape,
    )
)
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), activation="relu"))
model.add(Dropout(0.4))
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dropout(0.3))
model.add(Dense(num_classes, activation="softmax"))

model.compile(
    loss=keras.losses.categorical_crossentropy,
    optimizer=keras.optimizers.Adam(),
    metrics=["accuracy"],
)

# start an Azure ML run
run = Run.get_context()

class LogRunMetrics(Callback):
    # callback at the end of every epoch

```

```

def on_epoch_end(self, epoch, log):
    # log a value repeated which creates a list
    run.log("Loss", log["loss"])
    run.log("Accuracy", log["accuracy"])


history = model.fit(
    X_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_val, y_val),
    callbacks=[LogRunMetrics()],
)
score = model.evaluate(X_test, y_test, verbose=0)

# log a single value
run.log("Final test loss", score[0])
print("Test loss:", score[0])

run.log("Final test accuracy", score[1])
print("Test accuracy:", score[1])

plt.figure(figsize=(6, 3))
plt.title("Fashion MNIST with Keras ({}) epochs)".format(epochs), fontsize=14)
plt.plot(history.history["accuracy"], "b-", label="Accuracy", lw=4, alpha=0.5)
plt.plot(history.history["loss"], "r--", label="Loss", lw=4, alpha=0.5)
plt.legend(fontsize=12)
plt.grid(True)

# log an image
run.log_image("Loss v.s. Accuracy", plot=plt)

# create a ./outputs/model folder in the compute target
# files saved in the "./outputs" folder are automatically uploaded into run history
os.makedirs("./outputs/model", exist_ok=True)

# serialize NN architecture to JSON
model_json = model.to_json()
# save model JSON
with open("./outputs/model/model.json", "w") as f:
    f.write(model_json)
# save model weights
model.save_weights("./outputs/model/model.h5")
print("model saved in ./outputs/model folder")

```

Most of this code should be familiar to ML developers:

- The data is partitioned into train and validation sets for training, and a separate test subset for final scoring
- The input shape is 28x28x1 (only 1 because the input is grayscale), there will be 256 inputs in a batch, and there are 10 classes
- The number of training epochs will be 10
- The model has three convolutional layers, with max pooling and dropout, followed by a dense layer and softmax head
- The model is fitted for 10 epochs and then evaluated
- The model architecture is written to `outputs/model/model.json` and the weights to `outputs/model/model.h5`

Some of the code, though, is specific to Azure Machine Learning. `run = Run.get_context()` retrieves a `Run` object, which contains the current service context. The `train.py` source uses this `run` object to retrieve the input dataset via its name (an alternative to the code in `prepare.py` that retrieved the dataset via the `argv` array of script arguments).

The `run` object is also used to log the training progress at the end of every epoch and, at the end of training, to log the graph of loss and accuracy over time.

Create the training pipeline step

The training step has a slightly more complex configuration than the preparation step. The preparation step used only standard Python libraries. More commonly, you'll need to modify the runtime environment in which your source code runs.

Create a file `conda_dependencies.yml` with the following contents:

```
dependencies:
- python=3.6.2
- pip:
  - azureml-core
  - azureml-dataset-runtime
  - keras==2.4.3
  - tensorflow==2.4.3
  - numpy
  - scikit-learn
  - pandas
  - matplotlib
```

The `Environment` class represents the runtime environment in which a machine learning task runs. Associate the above specification with the training code with:

```
keras_env = Environment.from_conda_specification(
    name="keras-env", file_path="./conda_dependencies.yml"
)

train_cfg = ScriptRunConfig(
    source_directory=script_folder,
    script="train.py",
    compute_target=compute_target,
    environment=keras_env,
)
```

Creating the training step itself uses code similar to the code used to create the preparation step:

```
train_step = PythonScriptStep(
    name="train step",
    arguments=[
        prepared_fashion_ds.read_delimited_files().as_input(name="prepared_fashion_ds")
    ],
    source_directory=train_cfg.source_directory,
    script_name=train_cfg.script,
    runconfig=train_cfg.run_config,
)
```

Create and run the pipeline

Now that you've specified data inputs and outputs and created your pipeline's steps, you can compose them into a pipeline and run it:

```
pipeline = Pipeline(workspace, steps=[prep_step, train_step])
run = exp.submit(pipeline)
```

The `Pipeline` object you create runs in your `workspace` and is composed of the preparation and training steps

you've specified.

NOTE

This pipeline has a simple dependency graph: the training step relies on the preparation step and the preparation step relies on the `fashion_ds` dataset. Production pipelines will often have much more complex dependencies. Steps may rely on multiple upstream steps, a source code change in an early step may have far-reaching consequences, and so on. Azure Machine Learning tracks these concerns for you. You need only pass in the array of `steps` and Azure Machine Learning takes care of calculating the execution graph.

The call to `submit` the `Experiment` completes quickly, and produces output similar to:

```
Submitted PipelineRun 5968530a-abcd-1234-9cc1-46168951b5eb  
Link to Azure Machine Learning Portal: https://ml.azure.com/runs/abc-xyz...
```

You can monitor the pipeline run by opening the link or you can block until it completes by running:

```
run.wait_for_completion(show_output=True)
```

IMPORTANT

The first pipeline run takes roughly *15 minutes*. All dependencies must be downloaded, a Docker image is created, and the Python environment is provisioned and created. Running the pipeline again takes significantly less time because those resources are reused instead of created. However, total run time for the pipeline depends on the workload of your scripts and the processes that are running in each pipeline step.

Once the pipeline completes, you can retrieve the metrics you logged in the training step:

```
run.find_step_run("train step")[0].get_metrics()
```

If you're satisfied with the metrics, you can register the model in your workspace:

```
run.find_step_run("train step")[0].register_model(  
    model_name="keras-model",  
    model_path="outputs/model/",  
    datasets=[("train test data", fashion_ds)],  
)
```

Clean up resources

Don't complete this section if you plan to run other Azure Machine Learning tutorials.

Stop the compute instance

If you used a compute instance, stop the VM when you aren't using it to reduce cost.

1. In your workspace, select **Compute**.
2. From the list, select the name of the compute instance.
3. Select **Stop**.
4. When you're ready to use the server again, select **Start**.

Delete everything

If you don't plan to use the resources you created, delete them, so you don't incur any charges:

1. In the Azure portal, in the left menu, select **Resource groups**.
2. In the list of resource groups, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then, select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties, and then select **Delete**.

Next steps

In this tutorial, you used the following types:

- The `Workspace` represents your Azure Machine Learning workspace. It contained:
 - The `Experiment` that contains the results of training runs of your pipeline
 - The `Dataset` that lazily loaded the data held in the Fashion-MNIST datastore
 - The `ComputeTarget` that represents the machine(s) on which the pipeline steps run
 - The `Environment` that is the runtime environment in which the pipeline steps run
 - The `Pipeline` that composes the `PythonScriptStep` steps into a whole
 - The `Model` that you registered after being satisfied with the training process

The `Workspace` object contains references to other resources (notebooks, endpoints, and so on) that weren't used in this tutorial. For more, see [What is an Azure Machine Learning workspace?](#).

The `outputFileDatasetConfig` promotes the output of a run to a file-based dataset. For more information on datasets and working with data, see [How to access data](#).

For more on compute targets and environments, see [What are compute targets in Azure Machine Learning?](#) and [What are Azure Machine Learning environments?](#)

The `ScriptRunConfig` associates a `ComputeTarget` and `Environment` with Python source files. A `PythonScriptStep` takes that `ScriptRunConfig` and defines its inputs and outputs, which in this pipeline was the file dataset built by the `outputFileDatasetConfig`.

For more examples of how to build pipelines by using the machine learning SDK, see the [example repository](#).

Tutorial: Power BI integration - Create the predictive model with a Jupyter Notebook (part 1 of 2)

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO: Python SDK azureml v1

In part 1 of this tutorial, you train and deploy a predictive machine learning model by using code in a Jupyter Notebook. You also create a scoring script to define the input and output schema of the model for integration into Power BI. In part 2, you use the model to predict outcomes in Microsoft Power BI.

In this tutorial, you:

- Create a Jupyter Notebook.
- Create an Azure Machine Learning compute instance.
- Train a regression model by using scikit-learn.
- Write a scoring script that defines the input and output for easy integration into Microsoft Power BI.
- Deploy the model to a real-time scoring endpoint.

Prerequisites

- An Azure subscription. If you don't already have a subscription, you can use a [free trial](#).
- An Azure Machine Learning workspace. If you don't already have a workspace, see [Create workspace resources](#).
- Introductory knowledge of the Python language and machine learning workflows.

Create a notebook and compute

On the [Azure Machine Learning Studio](#) home page, select **Create new > Notebook**:

The screenshot shows the Azure Machine Learning studio interface. On the left, there's a sidebar with navigation links like 'New', 'Home', 'Author', 'Notebooks', 'Automated ML', 'Designer', 'Assets', 'Data', 'Jobs', and 'Components'. The 'Notebooks' link is currently selected. The main area is titled 'Azure Machine Learning studio' and contains four cards: 'Create new' (with a plus icon), 'Notebooks' (with a document icon), 'Automated ML' (with a lightning bolt icon), and 'Designer' (with a cube icon). The 'Create new' card has a red box around its 'Create new' button. Below each card is a 'Start now' button.

On the [Create a new file](#) page:

1. Name your notebook (for example, *my_model_notebook*).
2. Change the **File Type** to **Notebook**.
3. Select **Create**.

Next, to run code cells, create a compute instance and attach it to your notebook. Start by selecting the plus icon at the top of the notebook:

The screenshot shows the top navigation bar of a Jupyter Notebook. The title bar says "my_model_notebook". The top right has a "Compute" dropdown set to "No computes found", a "+" button (highlighted with a red box), and a "No kernel connected" message. A yellow status bar at the bottom says "Your document is currently not connected to a compute. Switch to a running compute or create a new compute to run a cell."

On the **Create compute instance** page:

1. Choose a CPU virtual machine size. For this tutorial, you can choose a **Standard_D11_v2**, with 2 cores and 14 GB of RAM.
2. Select **Next**.
3. On the **Configure Settings** page, provide a valid **Compute name**. Valid characters are uppercase and lowercase letters, digits, and hyphens (-).
4. Select **Create**.

In the notebook, you might notice the circle next to **Compute** turned cyan. This color change indicates that the compute instance is being created:

The screenshot shows the top navigation bar again. The "Compute" dropdown now shows "pbi-tutorial - Creating" (highlighted with a red box). The status bar at the bottom remains the same.

NOTE

The compute instance can take 2 to 4 minutes to be provisioned.

After the compute is provisioned, you can use the notebook to run code cells. For example, in the cell you can type the following code:

```
import numpy as np  
np.sin(3)
```

Then select Shift + Enter (or select Control + Enter or select the Play button next to the cell). You should see the following output:

The screenshot shows the Jupyter Notebook interface with a code cell containing "import numpy as np" and "np.sin(3)". The cell is marked with a green play button icon. The output below the cell shows the result: "0.1411200080598672". The top navigation bar shows "Compute: pbi-tutorial - Running".

Now you're ready to build a machine learning model.

Build a model by using scikit-learn

In this tutorial, you use the [Diabetes dataset](#). This dataset is available in [Azure Open Datasets](#).

Import data

To import your data, copy the following code and paste it into a new *code cell* in your notebook.

```
from azureml.opendatasets import Diabetes

diabetes = Diabetes.get_tabular_dataset()
X = diabetes.drop_columns("Y")
y = diabetes.keep_columns("Y")
X_df = X.to_pandas_dataframe()
y_df = y.to_pandas_dataframe()
X_df.info()
```

The `X_df` pandas data frame contains 10 baseline input variables. These variables include age, sex, body mass index, average blood pressure, and six blood serum measurements. The `y_df` pandas data frame is the target variable. It contains a quantitative measure of disease progression one year after the baseline. The data frame contains 442 records.

Train the model

Create a new *code cell* in your notebook. Then copy the following code and paste it into the cell. This code snippet constructs a ridge regression model and serializes the model by using the Python pickle format.

```
import joblib
from sklearn.linear_model import Ridge

model = Ridge().fit(X_df,y_df)
joblib.dump(model, 'sklearn_regression_model.pkl')
```

Register the model

In addition to the content of the model file itself, your registered model will store metadata. The metadata includes the model description, tags, and framework information.

Metadata is useful when you're managing and deploying models in your workspace. By using tags, for instance, you can categorize your models and apply filters when you list models in your workspace. Also, if you mark this model with the scikit-learn framework, you'll simplify deploying it as a web service.

Copy the following code and then paste it into a new *code cell* in your notebook.

```

import sklearn

from azureml.core import Workspace
from azureml.core import Model
from azureml.core.resource_configuration import ResourceConfiguration

ws = Workspace.from_config()

model = Model.register(workspace=ws,
                       model_name='my-sklearn-model',           # Name of the registered model in your
workspace.
                       model_path='./sklearn_regression_model.pkl', # Local file to upload and register as
a model.
                       model_framework=Model.Framework.SCIKITLEARN, # Framework used to create the model.
                       model_framework_version=sklearn.__version__, # Version of scikit-learn used to
create the model.
                       sample_input_dataset=X,
                       sample_output_dataset=y,
                       resource_configuration=ResourceConfiguration(cpu=2, memory_in_gb=4),
                       description='Ridge regression model to predict diabetes progression.',
                       tags={'area': 'diabetes', 'type': 'regression'})

print('Name:', model.name)
print('Version:', model.version)

```

You can also view the model in Azure Machine Learning Studio. In the menu on the left, select **Models**:

Name	Version	Experiment	Job (Run ID)	Created on	Tags
my-sklearn-model	1			Dec 22, 2021 3:21 PM	area:diabetes type:re...
bidaf_onnx	1			Dec 22, 2021 12:32 PM	...
keras-model	1	keras-mnist-fashion	93f84cb0-b1d4-4f00-94c1-7ed...	Jan 27, 2022 2:12 PM	...
sklearn_mnist_model	1		0d3ab202-2e14-4487-845b-e5...	Jan 5, 2022 4:45 PM	...

Define the scoring script

When you deploy a model that will be integrated into Power BI, you need to define a Python *scoring script* and custom environment. The scoring script contains two functions:

- The `init()` function runs when the service starts. It loads the model (which is automatically downloaded from the model registry) and deserializes it.
- The `run(data)` function runs when a call to the service includes input data that needs to be scored.

NOTE

The Python decorators in the code below define the schema of the input and output data, which is important for integration into Power BI.

Copy the following code and paste it into a new *code cell* in your notebook. The following code snippet has cell magic that writes the code to a file named *score.py*.

```

%%writefile score.py

import json
import pickle
import numpy as np
import pandas as pd
import os
import joblib
from azureml.core.model import Model

from inference_schema.schema_decorators import input_schema, output_schema
from inference_schema.parameter_types.numpy_parameter_type import NumpyParameterType
from inference_schema.parameter_types.pandas_parameter_type import PandasParameterType


def init():
    global model
    # Replace filename if needed.
    path = os.getenv('AZUREML_MODEL_DIR')
    model_path = os.path.join(path, 'sklearn_regression_model.pkl')
    # Deserialize the model file back into a sklearn model.
    model = joblib.load(model_path)

input_sample = pd.DataFrame(data=[{
    "AGE": 5,
    "SEX": 2,
    "BMI": 3.1,
    "BP": 3.1,
    "S1": 3.1,
    "S2": 3.1,
    "S3": 3.1,
    "S4": 3.1,
    "S5": 3.1,
    "S6": 3.1
}])

# This is an integer type sample. Use the data type that reflects the expected result.
output_sample = np.array([0])

# To indicate that we support a variable length of data input,
# set enforce_shape=False
@input_schema('data', PandasParameterType(input_sample))
@output_schema(NumpyParameterType(output_sample))
def run(data):
    try:
        print("input_data....")
        print(data.columns)
        print(type(data))
        result = model.predict(data)
        print("result....")
        print(result)
    # You can return any data type, as long as it can be serialized by JSON.
    # return result.tolist()
    except Exception as e:
        error = str(e)
        return error

```

Define the custom environment

Next, define the environment to score the model. In the environment, define the Python packages, such as pandas and scikit-learn, that the scoring script (*score.py*) requires.

To define the environment, copy the following code and paste it into a new *code cell* in your notebook.

```

from azureml.core.model import InferenceConfig
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

environment = Environment('my-sklearn-environment')
environment.python.conda_dependencies = CondaDependencies.create(pip_packages=[
    'azureml-defaults',
    'inference-schema[numpy-support]',
    'joblib',
    'numpy',
    'pandas',
    'scikit-learn=={}'.format(sklearn.__version__)
])

inference_config = InferenceConfig(entry_script='./score.py', environment=environment)

```

Deploy the model

To deploy the model, copy the following code and paste it into a new *code cell* in your notebook:

```

service_name = 'my-diabetes-model'

service = Model.deploy(ws, service_name, [model], inference_config, overwrite=True)
service.wait_for_deployment(show_output=True)

```

NOTE

The service can take 2 to 4 minutes to deploy.

If the service deploys successfully, you should see the following output:

```

Tips: You can try get_logs(): https://aka.ms/debugimage#dockerlog or local deployment:
https://aka.ms/debugimage#debug-locally to debug if deployment takes longer than 10 minutes.
Running.....
Succeeded
ACI service creation operation finished, operation "Succeeded"

```

You can also view the service in Azure Machine Learning Studio. In the menu on the left, select **Endpoints**:

Name	Description	Created on	Created by	Updated on	Compute type
my-diabetes-model		Dec 22, 2021 2:22 PM	Chris Q. Public	Dec 22, 2021 2:22 PM	Container instance

We recommend that you test the web service to ensure it works as expected. To return your notebook, in Azure

Machine Learning Studio, in the menu on the left, select **Notebooks**. Then copy the following code and paste it into a new *code cell* in your notebook to test the service.

```
import json

input_payload = json.dumps({
    'data': X_df[0:2].values.tolist()
})

output = service.run(input_payload)

print(output)
```

The output should look like this JSON structure: `{'predict': [[205.59], [68.84]]}`.

Next steps

In this tutorial, you saw how to build and deploy a model so that it can be consumed by Power BI. In the next part, you'll learn how to consume this model in a Power BI report.

[Tutorial: Consume a model in Power BI](#)

Explore Azure Machine Learning with Jupyter Notebooks (v1)

9/21/2022 • 2 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

The [Azure Machine Learning Notebooks repository](#) includes Azure Machine Learning Python SDK (v1) samples. These Jupyter notebooks are designed to help you explore the SDK and serve as models for your own machine learning projects. In this repository, you'll find tutorial notebooks in the **tutorials** folder and feature-specific notebooks in the **how-to-use-azureml** folder.

This article shows you how to access the repositories from the following environments:

- Azure Machine Learning compute instance
- Bring your own notebook server
- Data Science Virtual Machine

Option 1: Access on Azure Machine Learning compute instance (recommended)

The easiest way to get started with the samples is to complete the [Quickstart: Get started with Azure Machine Learning](#). Once completed, you'll have a dedicated notebook server pre-loaded with the SDK and the Azure Machine Learning Notebooks repository. No downloads or installation necessary.

Option 2: Access on your own notebook server

If you'd like to bring your own notebook server for local development, follow these steps on your computer.

1. Use the instructions at [Azure Machine Learning SDK](#) to install the Azure Machine Learning SDK (v1) for Python
2. Create an [Azure Machine Learning workspace](#).
3. Write a [configuration file](#) file (`aml_config/config.json`).
4. Clone [the Machine Learning Notebooks repository](#).

```
git clone https://github.com/Azure/MachineLearningNotebooks.git --depth 1
```

5. Start the notebook server from the directory containing your clone.

```
jupyter notebook
```

These instructions install the base SDK packages necessary for the quickstart and tutorial notebooks. Other sample notebooks may require you to install extra components. For more information, see [Install the Azure Machine Learning SDK for Python](#).

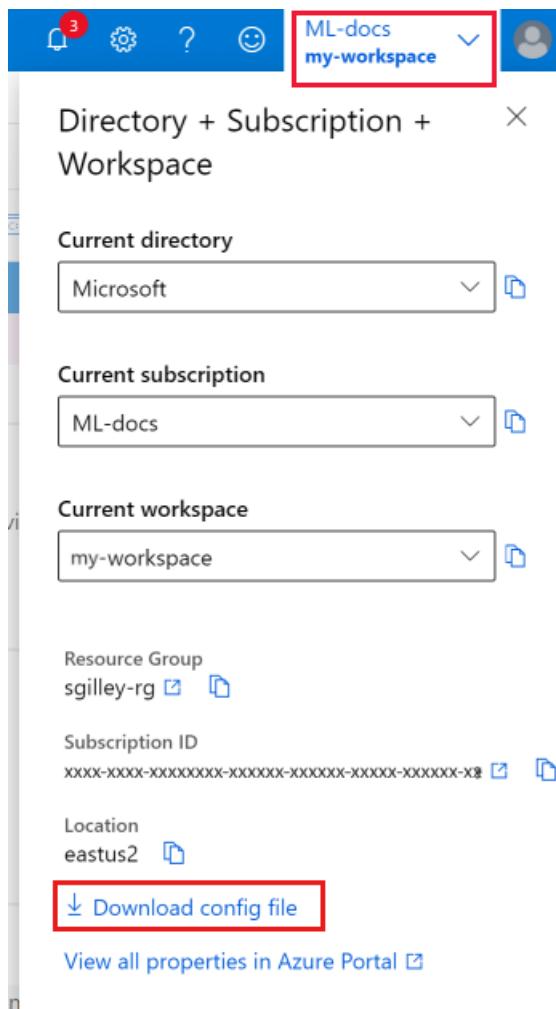
Option 3: Access on a DSVM

The Data Science Virtual Machine (DSVM) is a customized VM image built specifically for doing data science. If

you [create a DSVM](#), the SDK and notebook server are installed and configured for you. However, you'll still need to create a workspace and clone the sample repository.

1. [Create an Azure Machine Learning workspace](#).
2. Add a workspace configuration file using either of these methods:

- In [Azure Machine Learning studio](#), select your workspace settings in the upper right, then select [Download config file](#).



- Create a new workspace using code in the [configuration.ipynb](#) notebook.
3. From the directory where you added the configuration file, clone [the Machine Learning Notebooks repository](#).

```
git clone https://github.com/Azure/MachineLearningNotebooks.git --depth 1
```

4. Start the notebook server from the directory, which now contains the clone and the config file.

```
jupyter notebook
```

Next steps

Explore the [MachineLearningNotebooks](#) repository to discover what Azure Machine Learning can do.

For more GitHub sample projects and examples, see these repos:

- [Microsoft/MLOps](#)
- [Microsoft/MLOpsPython](#)

Install & use the CLI (v1)

9/21/2022 • 17 minutes to read • [Edit Online](#)

APPLIES TO:  Azure CLI ml extension v1

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

The Azure Machine Learning CLI is an extension to the [Azure CLI](#), a cross-platform command-line interface for the Azure platform. This extension provides commands for working with Azure Machine Learning. It allows you to automate your machine learning activities. The following list provides some example actions that you can do with the CLI extension:

- Run experiments to create machine learning models
- Register machine learning models for customer usage
- Package, deploy, and track the lifecycle of your machine learning models

The CLI is not a replacement for the Azure Machine Learning SDK. It is a complementary tool that is optimized to handle highly parameterized tasks which suit themselves well to automation.

Prerequisites

- To use the CLI, you must have an Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#) today.
- To use the CLI commands in this document from your **local environment**, you need the [Azure CLI](#).

If you use the [Azure Cloud Shell](#), the CLI is accessed through the browser and lives in the cloud.

Full reference docs

Find the [full reference docs for the azure-cli-ml extension of Azure CLI](#).

Connect the CLI to your Azure subscription

IMPORTANT

If you are using the Azure Cloud Shell, you can skip this section. The cloud shell automatically authenticates you using the account you log into your Azure subscription.

There are several ways that you can authenticate to your Azure subscription from the CLI. The most basic is to interactively authenticate using a browser. To authenticate interactively, open a command line or terminal and use the following command:

```
az login
```

If the CLI can open your default browser, it will do so and load a sign-in page. Otherwise, you need to open a browser and follow the instructions on the command line. The instructions involve browsing to <https://aka.ms/devicelogin> and entering an authorization code.

TIP

After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the [Azure portal](#) by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.

To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see [Use multiple Azure Subscriptions](#).

For other methods of authenticating, see [Sign in with Azure CLI](#).

Install the extension

To install the CLI (v1) extension:

```
az extension add -n azure-cli-ml
```

Update the extension

To update the Machine Learning CLI extension, use the following command:

```
az extension update -n azure-cli-ml
```

Remove the extension

To remove the CLI extension, use the following command:

```
az extension remove -n azure-cli-ml
```

Resource management

The following commands demonstrate how to use the CLI to manage resources used by Azure Machine Learning.

- If you do not already have one, create a resource group:

```
az group create -n myresourcegroup -l westus2
```

- Create an Azure Machine Learning workspace:

```
az ml workspace create -w myworkspace -g myresourcegroup
```

For more information, see [az ml workspace create](#).

- Attach a workspace configuration to a folder to enable CLI contextual awareness.

```
az ml folder attach -w myworkspace -g myresourcegroup
```

This command creates a `.azureml` subdirectory that contains example runconfig and conda environment files. It also contains a `config.json` file that is used to communicate with your Azure Machine Learning workspace.

For more information, see [az ml folder attach](#).

- Attach an Azure blob container as a Datastore.

```
az ml datastore attach-blob -n datastorename -a accountname -c containername
```

For more information, see [az ml datastore attach-blob](#).

- Upload files to a Datastore.

```
az ml datastore upload -n datastorename -p sourcepath
```

For more information, see [az ml datastore upload](#).

- Attach an AKS cluster as a Compute Target.

```
az ml computetarget attach aks -n myaks -i myaksresourceid -g myresourcegroup -w myworkspace
```

For more information, see [az ml computetarget attach aks](#)

Compute clusters

- Create a new managed compute cluster.

```
az ml computetarget create amlcompute -n cpu --min-nodes 1 --max-nodes 1 -s STANDARD_D3_V2
```

- Create a new managed compute cluster with managed identity

- User-assigned managed identity

```
az ml computetarget create amlcompute --name cpu-cluster --vm-size Standard_NC6 --max-nodes 5  
--assign-identity  
'/subscriptions/<subscription_id>/resourcegroups/<resource_group>/providers/Microsoft.ManagedId  
entity/userAssignedIdentities/<user_assigned_identity>'
```

- System-assigned managed identity

```
az ml computetarget create amlcompute --name cpu-cluster --vm-size Standard_NC6 --max-nodes 5  
--assign-identity '[system]'
```

- Add a managed identity to an existing cluster:

- User-assigned managed identity

```
az ml computetarget amlcompute identity assign --name cpu-cluster  
'/subscriptions/<subscription_id>/resourcegroups/<resource_group>/providers/Microsoft.ManagedId  
entity/userAssignedIdentities/<user_assigned_identity>'
```

- System-assigned managed identity

```
az ml computetarget amlcompute identity assign --name cpu-cluster '[system]'
```

For more information, see [az ml computetarget create amlcompute](#).

NOTE

Azure Machine Learning compute clusters support only **one system-assigned identity or multiple user-assigned identities**, not both concurrently.

Compute instance

Manage compute instances. In all the examples below, the name of the compute instance is **cpu**

- Create a new computeinstance.

```
az ml computetarget create computeinstance -n cpu -s "STANDARD_D3_V2" -v
```

For more information, see [az ml computetarget create computeinstance](#).

- Stop a computeinstance.

```
az ml computetarget computeinstance stop -n cpu -v
```

For more information, see [az ml computetarget computeinstance stop](#).

- Start a computeinstance.

```
az ml computetarget computeinstance start -n cpu -v
```

For more information, see [az ml computetarget computeinstance start](#).

- Restart a computeinstance.

```
az ml computetarget computeinstance restart -n cpu -v
```

For more information, see [az ml computetarget computeinstance restart](#).

- Delete a computeinstance.

```
az ml computetarget delete -n cpu -v
```

For more information, see [az ml computetarget delete computeinstance](#).

Run experiments

- Start a run of your experiment. When using this command, specify the name of the runconfig file (the text before *.runconfig if you are looking at your file system) against the -c parameter.

```
az ml run submit-script -c sklearn -e testexperiment train.py
```

TIP

The `az ml folder attach` command creates a `.azureml` subdirectory, which contains two example runconfig files.

If you have a Python script that creates a run configuration object programmatically, you can use `RunConfig.save()` to save it as a runconfig file.

The full runconfig schema can be found in this [JSON file](#). The schema is self-documenting through the `description` key of each object. Additionally, there are enums for possible values, and a template snippet at the end.

For more information, see [az ml run submit-script](#).

- View a list of experiments:

```
az ml experiment list
```

For more information, see [az ml experiment list](#).

HyperDrive run

You can use HyperDrive with Azure CLI to perform parameter tuning runs. First, create a HyperDrive configuration file in the following format. See [Tune hyperparameters for your model](#) article for details on hyperparameter tuning parameters.

```
# hdconfig.yml
sampling:
    type: random # Supported options: Random, Grid, Bayesian
    parameter_space: # specify a name|expression|values tuple for each parameter.
        - name: --penalty # The name of a script parameter to generate values for.
          expression: choice # supported options: choice, randint, uniform, quniform, loguniform, qloguniform, normal, qnormal, lognormal, qlognormal
          values: [0.5, 1, 1.5] # The list of values, the number of values is dependent on the expression specified.
    policy:
        type: BanditPolicy # Supported options: BanditPolicy, MedianStoppingPolicy, TruncationSelectionPolicy, NoTerminationPolicy
        evaluation_interval: 1 # Policy properties are policy specific. See the above link for policy specific parameter details.
        slack_factor: 0.2
primary_metric_name: Accuracy # The metric used when evaluating the policy
primary_metric_goal: Maximize # Maximize|Minimize
max_total_runs: 8 # The maximum number of runs to generate
max_concurrent_runs: 2 # The number of runs that can run concurrently.
max_duration_minutes: 100 # The maximum length of time to run the experiment before cancelling.
```

Add this file alongside the run configuration files. Then submit a HyperDrive run using:

```
az ml run submit-hyperdrive -e <experiment> -c <runconfig> --hyperdrive-configuration-name <hdconfig>  
my_train.py
```

Note the *arguments* section in runconfig and *parameter space* in HyperDrive config. They contain the command-line arguments to be passed to training script. The value in runconfig stays the same for each iteration, while the range in HyperDrive config is iterated over. Do not specify the same argument in both files.

Dataset management

The following commands demonstrate how to work with datasets in Azure Machine Learning:

- Register a dataset:

```
az ml dataset register -f mydataset.json
```

For information on the format of the JSON file used to define the dataset, use

```
az ml dataset register --show-template .
```

For more information, see [az ml dataset register](#).

- List all datasets in a workspace:

```
az ml dataset list
```

For more information, see [az ml dataset list](#).

- Get details of a dataset:

```
az ml dataset show -n dataset-name
```

For more information, see [az ml dataset show](#).

- Unregister a dataset:

```
az ml dataset unregister -n dataset-name
```

For more information, see [az ml dataset unregister](#).

Environment management

The following commands demonstrate how to create, register, and list Azure Machine Learning environments for your workspace:

- Create scaffolding files for an environment:

```
az ml environment scaffold -n myenv -d myenvdirectory
```

For more information, see [az ml environment scaffold](#).

- Register an environment:

```
az ml environment register -d myenvdirectory
```

For more information, see [az ml environment register](#).

- List registered environments:

```
az ml environment list
```

For more information, see [az ml environment list](#).

- Download a registered environment:

```
az ml environment download -n myenv -d downloaddirectory
```

For more information, see [az ml environment download](#).

Environment configuration schema

If you used the `az ml environment scaffold` command, it generates a template `azureml_environment.json` file that can be modified and used to create custom environment configurations with the CLI. The top level object loosely maps to the [Environment](#) class in the Python SDK.

```
{
    "name": "testenv",
    "version": null,
    "environmentVariables": {
        "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
    },
    "python": {
        "userManagedDependencies": false,
        "interpreterPath": "python",
        "condaDependenciesFile": null,
        "baseCondaEnvironment": null
    },
    "docker": {
        "enabled": false,
        "baseImage": "mcr.microsoft.com/azureml/openmpi3.1.2-ubuntu18.04:20210615.v1",
        "baseDockerfile": null,
        "sharedVolumes": true,
        "shmSize": "2g",
        "arguments": [],
        "baseImageRegistry": {
            "address": null,
            "username": null,
            "password": null
        }
    },
    "spark": {
        "repositories": [],
        "packages": [],
        "precachePackages": true
    },
    "databricks": {
        "mavenLibraries": [],
        "pypiLibraries": [],
        "rcranLibraries": [],
        "jarLibraries": [],
        "eggLibraries": []
    },
    "inferencingStackVersion": null
}
```

The following table details each top-level field in the JSON file, its type, and a description. If an object type is linked to a class from the Python SDK, there is a loose 1:1 match between each JSON field and the public

variable name in the Python class. In some cases the field may map to a constructor argument rather than a class variable. For example, the `environmentVariables` field maps to the `environment_variables` variable in the [Environment](#) class.

JSON FIELD	TYPE	DESCRIPTION
<code>name</code>	<code>string</code>	Name of the environment. Do not start name with Microsoft or AzureML .
<code>version</code>	<code>string</code>	Version of the environment.
<code>environmentVariables</code>	<code>{string: string}</code>	A hash-map of environment variable names and values.
<code>python</code>	PythonSection	hat defines the Python environment and interpreter to use on target compute resource.
<code>docker</code>	DockerSection	Defines settings to customize the Docker image built to the environment's specifications.
<code>spark</code>	SparkSection	The section configures Spark settings. It is only used when framework is set to PySpark.
<code>databricks</code>	DatabricksSection	Configures Databricks library dependencies.
<code>inferencingStackVersion</code>	<code>string</code>	Specifies the inferencing stack version added to the image. To avoid adding an inferencing stack, leave this field <code>null</code> . Valid value: "latest".

ML pipeline management

The following commands demonstrate how to work with machine learning pipelines:

- Create a machine learning pipeline:

```
az ml pipeline create -n mypipeline -y mypipeline.yml
```

For more information, see [az ml pipeline create](#).

For more information on the pipeline YAML file, see [Define machine learning pipelines in YAML](#).

- Run a pipeline:

```
az ml run submit-pipeline -n myexperiment -y mypipeline.yml
```

For more information, see [az ml run submit-pipeline](#).

For more information on the pipeline YAML file, see [Define machine learning pipelines in YAML](#).

- Schedule a pipeline:

```
az ml pipeline create-schedule -n myschedule -e myexperiment -i mypipelineid -y myschedule.yml
```

For more information, see [az ml pipeline create-schedule](#).

Model registration, profiling, deployment

The following commands demonstrate how to register a trained model, and then deploy it as a production service:

- Register a model with Azure Machine Learning:

```
az ml model register -n mymodel -p sklearn_regression_model.pkl
```

For more information, see [az ml model register](#).

- **OPTIONAL** Profile your model to get optimal CPU and memory values for deployment.

```
az ml model profile -n myprofile -m mymodel:1 --ic inferenceconfig.json -d "{\"data\": [[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}" -t myprofileresult.json
```

For more information, see [az ml model profile](#).

- Deploy your model to AKS

```
az ml model deploy -n myservice -m mymodel:1 --ic inferenceconfig.json --dc deploymentconfig.json --ct akscomputetarget
```

For more information on the inference configuration file schema, see [Inference configuration schema](#).

For more information on the deployment configuration file schema, see [Deployment configuration schema](#).

For more information, see [az ml model deploy](#).

Inference configuration schema

The entries in the `inferenceconfig.json` document map to the parameters for the `InferenceConfig` class. The following table describes the mapping between entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>entryScript</code>	<code>entry_script</code>	Path to a local file that contains the code to run for the image.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
sourceDirectory	source_directory	Optional. Path to folders that contain all files to create the image, which makes it easy to access any files within this folder or subfolder. You can upload an entire folder from your local machine as dependencies for the Webservice. Note: your entry_script, conda_file, and extra_docker_file_steps paths are relative paths to the source_directory path.
environment	environment	Optional. Azure Machine Learning environment.

You can include full specifications of an Azure Machine Learning [environment](#) in the inference configuration file. If this environment doesn't exist in your workspace, Azure Machine Learning will create it. Otherwise, Azure Machine Learning will update the environment if necessary. The following JSON is an example:

```
{
    "entryScript": "score.py",
    "environment": {
        "docker": {
            "arguments": [],
            "baseDockerfile": null,
            "baseImage": "mcr.microsoft.com/azureml/intelmpi2018.3-ubuntu18.04",
            "enabled": false,
            "sharedVolumes": true,
            "shmSize": null
        },
        "environmentVariables": {
            "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
        },
        "name": "my-deploy-env",
        "python": {
            "baseCondaEnvironment": null,
            "condaDependencies": {
                "channels": [
                    "conda-forge"
                ],
                "dependencies": [
                    "python=3.6.2",
                    {
                        "pip": [
                            "azureml-defaults",
                            "azureml-telemetry",
                            "scikit-learn==0.22.1",
                            "inference-schema[numpy-support]"
                        ]
                    }
                ],
                "name": "project_environment"
            },
            "condaDependenciesFile": null,
            "interpreterPath": "python",
            "userManagedDependencies": false
        },
        "version": "1"
    }
}
```

You can also use an existing Azure Machine Learning [environment](#) in separated CLI parameters and remove the

"environment" key from the inference configuration file. Use -e for the environment name, and --ev for the environment version. If you don't specify --ev, the latest version will be used. Here is an example of an inference configuration file:

```
{  
    "entryScript": "score.py",  
    "sourceDirectory": null  
}
```

The following command demonstrates how to deploy a model using the previous inference configuration file (named myInferenceConfig.json).

It also uses the latest version of an existing Azure Machine Learning [environment](#) (named AzureML-Minimal).

```
az ml model deploy -m mymodel:1 --ic myInferenceConfig.json -e AzureML-Minimal --dc deploymentconfig.json
```

Deployment configuration schema

Local deployment configuration schema

The entries in the `deploymentconfig.json` document map to the parameters for

[LocalWebservice.deploy_configuration](#). The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>computeType</code>	NA	The compute target. For local targets, the value must be <code>local</code> .
<code>port</code>	<code>port</code>	The local port on which to expose the service's HTTP endpoint.

This JSON is an example deployment configuration for use with the CLI:

```
{  
    "computeType": "local",  
    "port": 32267  
}
```

Save this JSON as a file called `deploymentconfig.json`.

Azure Container Instance deployment configuration schema

The entries in the `deploymentconfig.json` document map to the parameters for

[AciWebservice.deploy_configuration](#). The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>computeType</code>	NA	The compute target. For ACI, the value must be <code>ACI</code> .
<code>containerResourceRequirements</code>	NA	Container for the CPU and memory entities.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
cpu	cpu_cores	The number of CPU cores to allocate. Defaults, 0.1
memoryInGB	memory_gb	The amount of memory (in GB) to allocate for this web service. Default, 0.5
location	location	The Azure region to deploy this Webservice to. If not specified the Workspace location will be used. More details on available regions can be found here: ACI Regions
authEnabled	auth_enabled	Whether to enable auth for this Webservice. Defaults to False
sslEnabled	ssl_enabled	Whether to enable SSL for this Webservice. Defaults to False.
appInsightsEnabled	enable_app_insights	Whether to enable AppInsights for this Webservice. Defaults to False
sslCertificate	ssl_cert_pem_file	The cert file needed if SSL is enabled
sslKey	ssl_key_pem_file	The key file needed if SSL is enabled
cname	ssl_cname	The cname for if SSL is enabled
dnsNameLabel	dns_name_label	The dns name label for the scoring endpoint. If not specified a unique dns name label will be generated for the scoring endpoint.

The following JSON is an example deployment configuration for use with the CLI:

```
{
  "computeType": "aci",
  "containerResourceRequirements":
  {
    "cpu": 0.5,
    "memoryInGB": 1.0
  },
  "authEnabled": true,
  "sslEnabled": false,
  "appInsightsEnabled": false
}
```

Azure Kubernetes Service deployment configuration schema

The entries in the `deploymentconfig.json` document map to the parameters for `AksWebservice.deploy_configuration`. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>computeType</code>	NA	The compute target. For AKS, the value must be <code>aks</code> .
<code>autoScaler</code>	NA	Contains configuration elements for autoscale. See the autoscaler table.
<code>autoscaleEnabled</code>	<code>autoscale_enabled</code>	Whether to enable autoscaling for the web service. If <code>numReplicas</code> = <code>0</code> , <code>True</code> ; otherwise, <code>False</code> .
<code>minReplicas</code>	<code>autoscale_min_replicas</code>	The minimum number of containers to use when autoscaling this web service. Default, <code>1</code> .
<code>maxReplicas</code>	<code>autoscale_max_replicas</code>	The maximum number of containers to use when autoscaling this web service. Default, <code>10</code> .
<code>refreshPeriodInSeconds</code>	<code>autoscale_refresh_seconds</code>	How often the autoscaler attempts to scale this web service. Default, <code>1</code> .
<code>targetUtilization</code>	<code>autoscale_target_utilization</code>	The target utilization (in percent out of 100) that the autoscaler should attempt to maintain for this web service. Default, <code>70</code> .
<code>dataCollection</code>	NA	Contains configuration elements for data collection.
<code>storageEnabled</code>	<code>collect_model_data</code>	Whether to enable model data collection for the web service. Default, <code>False</code> .
<code>authEnabled</code>	<code>auth_enabled</code>	Whether or not to enable key authentication for the web service. Both <code>tokenAuthEnabled</code> and <code>authEnabled</code> cannot be <code>True</code> . Default, <code>True</code> .
<code>tokenAuthEnabled</code>	<code>token_auth_enabled</code>	Whether or not to enable token authentication for the web service. Both <code>tokenAuthEnabled</code> and <code>authEnabled</code> cannot be <code>True</code> . Default, <code>False</code> .
<code>containerResourceRequirements</code>	NA	Container for the CPU and memory entities.
<code>cpu</code>	<code>cpu_cores</code>	The number of CPU cores to allocate for this web service. Defaults, <code>0.1</code> .
<code>memoryInGB</code>	<code>memory_gb</code>	The amount of memory (in GB) to allocate for this web service. Default, <code>0.5</code> .

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
appInsightsEnabled	enable_app_insights	Whether to enable Application Insights logging for the web service. Default, <code>False</code> .
scoringTimeoutMs	scoring_timeout_ms	A timeout to enforce for scoring calls to the web service. Default, <code>60000</code> .
maxConcurrentRequestsPerContainer	replica_max_concurrent_requests	The maximum concurrent requests per node for this web service. Default, <code>1</code> .
maxQueueWaitMs	max_request_wait_time	The maximum time a request will stay in the queue (in milliseconds) before a 503 error is returned. Default, <code>500</code> .
numReplicas	num_replicas	The number of containers to allocate for this web service. No default value. If this parameter is not set, the autoscaler is enabled by default.
keys	NA	Contains configuration elements for keys.
primaryKey	primary_key	A primary auth key to use for this Webservice
secondaryKey	secondary_key	A secondary auth key to use for this Webservice
gpuCores	gpu_cores	The number of GPU cores (per-container replica) to allocate for this Webservice. Default is 1. Only supports whole number values.
livenessProbeRequirements	NA	Contains configuration elements for liveness probe requirements.
periodSeconds	period_seconds	How often (in seconds) to perform the liveness probe. Default to 10 seconds. Minimum value is 1.
initialDelaySeconds	initial_delay_seconds	Number of seconds after the container has started before liveness probes are initiated. Defaults to 310
timeoutSeconds	timeout_seconds	Number of seconds after which the liveness probe times out. Defaults to 2 seconds. Minimum value is 1
successThreshold	success_threshold	Minimum consecutive successes for the liveness probe to be considered successful after having failed. Defaults to 1. Minimum value is 1.

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>failureThreshold</code>	<code>failure_threshold</code>	When a Pod starts and the liveness probe fails, Kubernetes will try failureThreshold times before giving up. Defaults to 3. Minimum value is 1.
<code>namespace</code>	<code>namespace</code>	The Kubernetes namespace that the webservice is deployed into. Up to 63 lowercase alphanumeric ('a'-'z', '0'-'9') and hyphen ('-') characters. The first and last characters can't be hyphens.

The following JSON is an example deployment configuration for use with the CLI:

```
{
  "computeType": "aks",
  "autoScaler":
  {
    "autoscaleEnabled": true,
    "minReplicas": 1,
    "maxReplicas": 3,
    "refreshPeriodInSeconds": 1,
    "targetUtilization": 70
  },
  "dataCollection":
  {
    "storageEnabled": true
  },
  "authEnabled": true,
  "containerResourceRequirements":
  {
    "cpu": 0.5,
    "memoryInGB": 1.0
  }
}
```

Next steps

- [Command reference for the Machine Learning CLI extension.](#)
- [Train and deploy machine learning models using Azure Pipelines](#)

Manage Azure Machine Learning workspaces using Azure CLI extension v1

9/21/2022 • 12 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

APPLIES TO:  Azure CLI ml extension v1

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning.

Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

In this article, you learn how to create and manage Azure Machine Learning workspaces using the Azure CLI. The Azure CLI provides commands for managing Azure resources and is designed to get you working quickly with Azure, with an emphasis on automation. The machine learning extension for the CLI provides commands for working with Azure Machine Learning resources.

Prerequisites

- An [Azure subscription](#). If you don't have one, try the [free or paid version of Azure Machine Learning](#).
- To use the CLI commands in this document from your [local environment](#), you need the [Azure CLI](#).

If you use the [Azure Cloud Shell](#), the CLI is accessed through the browser and lives in the cloud.

Limitations

- When creating a new workspace, you can either automatically create services needed by the workspace or use existing services. If you want to use **existing services from a different Azure subscription** than the workspace, you must register the Azure Machine Learning namespace in the subscription that contains those services. For example, creating a workspace in subscription A that uses a storage account from subscription B, the Azure Machine Learning namespace must be registered in subscription B before you can use the storage account with the workspace.

The resource provider for Azure Machine Learning is **Microsoft.MachineLearningServices**. For information on how to see if it is registered and how to register it, see the [Azure resource providers and types](#) article.

IMPORTANT

This only applies to resources provided during workspace creation; Azure Storage Accounts, Azure Container Register, Azure Key Vault, and Application Insights.

TIP

An Azure Application Insights instance is created when you create the workspace. You can delete the Application Insights instance after cluster creation if you want. Deleting it limits the information gathered from the workspace, and may make it more difficult to troubleshoot problems. **If you delete the Application Insights instance created by the workspace, you cannot re-create it without deleting and recreating the workspace.**

For more information on using this Application Insights instance, see [Monitor and collect data from Machine Learning web service endpoints](#).

Secure CLI communications

Some of the Azure CLI commands communicate with Azure Resource Manager over the internet. This communication is secured using HTTPS/TLS 1.2.

With the Azure Machine Learning CLI extension v1 (`azure-cli-ml`), only some of the commands communicate with the Azure Resource Manager. Specifically, commands that create, update, delete, list, or show Azure resources. Operations such as submitting a training job communicate directly with the Azure Machine Learning workspace. **If your workspace is secured with a private endpoint, that is enough to secure commands provided by the `azure-cli-ml` extension.**

Connect the CLI to your Azure subscription

IMPORTANT

If you are using the Azure Cloud Shell, you can skip this section. The cloud shell automatically authenticates you using the account you log into your Azure subscription.

There are several ways that you can authenticate to your Azure subscription from the CLI. The most simple is to interactively authenticate using a browser. To authenticate interactively, open a command line or terminal and use the following command:

```
az login
```

If the CLI can open your default browser, it will do so and load a sign-in page. Otherwise, you need to open a browser and follow the instructions on the command line. The instructions involve browsing to <https://aka.ms/devicelogin> and entering an authorization code.

TIP

After logging in, you see a list of subscriptions associated with your Azure account. The subscription information with `isDefault: true` is the currently activated subscription for Azure CLI commands. This subscription must be the same one that contains your Azure Machine Learning workspace. You can find the subscription ID from the [Azure portal](#) by visiting the overview page for your workspace. You can also use the SDK to get the subscription ID from the workspace object. For example, `Workspace.from_config().subscription_id`.

To select another subscription, use the `az account set -s <subscription name or ID>` command and specify the subscription name or ID to switch to. For more information about subscription selection, see [Use multiple Azure Subscriptions](#).

For other methods of authenticating, see [Sign in with Azure CLI](#).

Create a resource group

The Azure Machine Learning workspace must be created inside a resource group. You can use an existing resource group or create a new one. To **create a new resource group**, use the following command. Replace `<resource-group-name>` with the name to use for this resource group. Replace `<location>` with the Azure region to use for this resource group:

NOTE

You should select a region where Azure Machine Learning is available. For information, see [Products available by region](#).

```
az group create --name <resource-group-name> --location <location>
```

The response from this command is similar to the following JSON. You can use the output values to locate the created resources or parse them as input to subsequent CLI steps for automation.

```
{
  "id": "/subscriptions/<subscription-GUID>/resourceGroups/<resourcegroupname>",
  "location": "<location>",
  "managedBy": null,
  "name": "<resource-group-name>",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": null
}
```

For more information on working with resource groups, see [az group](#).

Create a workspace

When you deploy an Azure Machine Learning workspace, various other services are [required as dependent associated resources](#). When you use the CLI to create the workspace, the CLI can either create new associated resources on your behalf or you could attach existing resources.

IMPORTANT

When attaching your own storage account, make sure that it meets the following criteria:

- The storage account is *not* a premium account (Premium_LRS and Premium_GRS)
- Both Azure Blob and Azure File capabilities enabled
- Hierarchical Namespace (ADLS Gen 2) is disabled These requirements are only for the *default* storage account used by the workspace.

When attaching Azure container registry, you must have the [admin account](#) enabled before it can be used with an Azure Machine Learning workspace.

- [Create with new resources](#)
- [Bring existing resources](#)

To create a new workspace where the **services are automatically created**, use the following command:

```
az ml workspace create -w <workspace-name> -g <resource-group-name>
```

IMPORTANT

When you attaching existing resources, you don't have to specify all. You can specify one or more. For example, you can specify an existing storage account and the workspace will create the other resources.

The output of the workspace creation command is similar to the following JSON. You can use the output values to locate the created resources or parse them as input to subsequent CLI steps.

```
{  
  "applicationInsights": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-name>/providers/microsoft.insights/components/<application-insight-name>",  
  "containerRegistry": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-name>/providers/microsoft.containerregistry/registries/<acr-name>",  
  "creationTime": "2019-08-30T20:24:19.6984254+00:00",  
  "description": "",  
  "friendlyName": "<workspace-name>",  
  "id": "/subscriptions/<service-GUID>/resourceGroups/<resource-group-name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>",  
  "identityPrincipalId": "<GUID>",  
  "identityTenantId": "<GUID>",  
  "identityType": "SystemAssigned",  
  "keyVault": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-name>/providers/microsoft.keyvault/vaults/<key-vault-name>",  
  "location": "<location>",  
  "name": "<workspace-name>",  
  "resourceGroup": "<resource-group-name>",  
  "storageAccount": "/subscriptions/<service-GUID>/resourcegroups/<resource-group-name>/providers/microsoft.storage/storageaccounts/<storage-account-name>",  
  "type": "Microsoft.MachineLearningServices/workspaces",  
  "workspaceid": "<GUID>"  
}
```

Advanced configurations

Configure workspace for private network connectivity

Dependent on your use case and organizational requirements, you can choose to configure Azure Machine Learning using private network connectivity. You can use the Azure CLI to deploy a workspace and a Private link endpoint for the workspace resource. For more information on using a private endpoint and virtual network (VNet) with your workspace, see [Virtual network isolation and privacy overview](#). For complex resource configurations, also refer to template based deployment options including [Azure Resource Manager](#).

If you want to restrict workspace access to a virtual network, you can use the following parameters as part of the `az ml workspace create` command or use the `az ml workspace private-endpoint` commands.

```
az ml workspace create -w <workspace-name>  
  -g <resource-group-name>  
  --pe-name "<pe name>"  
  --pe-auto-approval "<pe-autoapproval>"  
  --pe-resource-group "<pe name>"  

```

- `--pe-name` : The name of the private endpoint that is created.
- `--pe-auto-approval` : Whether private endpoint connections to the workspace should be automatically approved.
- `--pe-resource-group` : The resource group to create the private endpoint in. Must be the same group that

contains the virtual network.

- `--pe-vnet-name` : The existing virtual network to create the private endpoint in.
- `--pe-subnet-name` : The name of the subnet to create the private endpoint in. The default value is `default`.

For more information on how to use these commands, see the [CLI reference pages](#).

Customer-managed key and high business impact workspace

By default, metadata for the workspace is stored in an Azure Cosmos DB instance that Microsoft maintains. This data is encrypted using Microsoft-managed keys. Instead of using the Microsoft-managed key, you can also provide your own key. Doing so creates an extra set of resources in your Azure subscription to store your data.

To learn more about the resources that are created when you bring your own key for encryption, see [Data encryption with Azure Machine Learning](#).

Use the `--cmk-keyvault` parameter to specify the Azure Key Vault that contains the key, and `--resource-cmk-uri` to specify the resource ID and uri of the key within the vault.

To [limit the data that Microsoft collects](#) on your workspace, you can additionally specify the `--hbi-workspace` parameter.

```
az ml workspace create -w <workspace-name>
    -g <resource-group-name>
    --cmk-keyvault "<cmk keyvault name>"
    --resource-cmk-uri "<resource cmk uri>"
```

NOTE

Authorize the **Machine Learning App** (in Identity and Access Management) with contributor permissions on your subscription to manage the data encryption additional resources.

NOTE

Azure Cosmos DB is **not** used to store information such as model performance, information logged by experiments, or information logged from your model deployments. For more information on monitoring these items, see the [Monitoring and logging](#) section of the architecture and concepts article.

IMPORTANT

Selecting high business impact can only be done when creating a workspace. You cannot change this setting after workspace creation.

For more information on customer-managed keys and high business impact workspace, see [Enterprise security for Azure Machine Learning](#).

Using the CLI to manage workspaces

Get workspace information

To get information about a workspace, use the following command:

```
az ml workspace show -w <workspace-name> -g <resource-group-name>
```

Update a workspace

To update a workspace, use the following command:

```
az ml workspace update -n <workspace-name> -g <resource-group-name>
```

Sync keys for dependent resources

If you change access keys for one of the resources used by your workspace, it takes around an hour for the workspace to synchronize to the new key. To force the workspace to sync the new keys immediately, use the following command:

```
az ml workspace sync-keys -w <workspace-name> -g <resource-group-name>
```

For more information on changing keys, see [Regenerate storage access keys](#).

Delete a workspace

WARNING

Once an Azure Machine Learning workspace has been deleted, it cannot be recovered.

To delete a workspace after it's no longer needed, use the following command:

```
az ml workspace delete -w <workspace-name> -g <resource-group-name>
```

IMPORTANT

Deleting a workspace does not delete the application insight, storage account, key vault, or container registry used by the workspace.

You can also delete the resource group, which deletes the workspace and all other Azure resources in the resource group. To delete the resource group, use the following command:

```
az group delete -g <resource-group-name>
```

If you accidentally deleted your workspace, are still able to retrieve your notebooks. For more information, see the [workspace deletion](#) section of the disaster recovery article.

Troubleshooting

Resource provider errors

When creating an Azure Machine Learning workspace, or a resource used by the workspace, you may receive an error similar to the following messages:

- No registered resource provider found for location {location}
- The subscription is not registered to use namespace {resource-provider-namespace}

Most resource providers are automatically registered, but not all. If you receive this message, you need to register the provider mentioned.

The following table contains a list of the resource providers required by Azure Machine Learning:

RESOURCE PROVIDER	WHY IT'S NEEDED
Microsoft.MachineLearningServices	Creating the Azure Machine Learning workspace.
Microsoft.Storage	Azure Storage Account is used as the default storage for the workspace.
Microsoft.ContainerRegistry	Azure Container Registry is used by the workspace to build Docker images.
Microsoft.KeyVault	Azure Key Vault is used by the workspace to store secrets.
Microsoft.Notebooks/NotebookProxies	Integrated notebooks on Azure Machine Learning compute instance.
Microsoft.ContainerService	If you plan on deploying trained models to Azure Kubernetes Services.

If you plan on using a customer-managed key with Azure Machine Learning, then the following service providers must be registered:

RESOURCE PROVIDER	WHY IT'S NEEDED
Microsoft.DocumentDB/databaseAccounts	Azure CosmosDB instance that logs metadata for the workspace.
Microsoft.Search/searchServices	Azure Search provides indexing capabilities for the workspace.

For information on registering resource providers, see [Resolve errors for resource provider registration](#).

Moving the workspace

WARNING

Moving your Azure Machine Learning workspace to a different subscription, or moving the owning subscription to a new tenant, is not supported. Doing so may cause errors.

Deleting the Azure Container Registry

The Azure Machine Learning workspace uses Azure Container Registry (ACR) for some operations. It will automatically create an ACR instance when it first needs one.

WARNING

Once an Azure Container Registry has been created for a workspace, do not delete it. Doing so will break your Azure Machine Learning workspace.

Next steps

For more information on the Azure CLI extension for machine learning, see the [az ml](#) (v1) documentation.

To check for problems with your workspace, see [How to use workspace diagnostics](#).

To learn how to move a workspace to a new Azure subscription, see [How to move a workspace](#).

For information on how to keep your Azure ML up to date with the latest security updates, see [Vulnerability management](#).

Create & use software environments in Azure Machine Learning with CLI v1

9/21/2022 • 2 minutes to read • [Edit Online](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

In this article, learn how to create and manage Azure Machine Learning [environments](#) using CLI v1. Use the environments to track and reproduce your projects' software dependencies as they evolve. The [Azure Machine Learning CLI](#) v1 mirrors most of the functionality of the Python SDK v1. You can use it to create and manage environments.

Software dependency management is a common task for developers. You want to ensure that builds are reproducible without extensive manual software configuration. The Azure Machine Learning [Environment](#) class accounts for local development solutions such as pip and Conda and distributed cloud development through Docker capabilities.

For a high-level overview of how environments work in Azure Machine Learning, see [What are ML environments?](#) For information about managing environments in the Azure ML studio, see [Manage environments in the studio](#). For information about configuring development environments, see [Set up a Python development environment for Azure ML](#).

Prerequisites

- An [Azure Machine Learning workspace](#)

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Scaffold an environment

The following command scaffolds the files for a default environment definition in the specified directory. These files are JSON files. They work like the corresponding class in the SDK. You can use the files to create new environments that have custom settings.

```
az ml environment scaffold -n myenv -d myenvdir
```

Register an environment

Run the following command to register an environment from a specified directory:

```
az ml environment register -d myenvdir
```

List environments

Run the following command to list all registered environments:

```
az ml environment list
```

Download an environment

To download a registered environment, use the following command:

```
az ml environment download -n myenv -d downloaddir
```

Next steps

- After you have a trained model, learn [how and where to deploy models](#).
- View the [Environment class SDK reference](#).

How to use workspace diagnostics (SDK v1)

9/21/2022 • 2 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

Azure Machine Learning provides a diagnostic API that can be used to identify problems with your workspace. Errors returned in the diagnostics report include information on how to resolve the problem.

In this article, learn how to use the workspace diagnostics from the Azure Machine Learning Python SDK v1.

Prerequisites

- An Azure Machine learning workspace. If you don't have one, see [Create a workspace](#).
- The [Azure Machine Learning SDK for Python](#).

Diagnostics from Python

The following snippet demonstrates how to use workspace diagnostics from Python

APPLIES TO:  Python SDK azureml v1

```
from azureml.core import Workspace

ws = Workspace.from_config()

diag_param = {
    "value": {
    }
}

resp = ws.diagnose_workspace(diag_param)
print(resp)
```

The response is a JSON document that contains information on any problems detected with the workspace. The following JSON is an example response:

```
{  
    "value": {  
        "user_defined_route_results": [],  
        "network_security_rule_results": [],  
        "resource_lock_results": [],  
        "dns_resolution_results": [{  
            "code": "CustomDnsInUse",  
            "level": "Warning",  
            "message": "It is detected VNet '/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/Microsoft.Network/virtualNetworks/<virtual-network-name>' of private endpoint '/subscriptions/<subscription-id>/resourceGroups/larrygroup0916/providers/Microsoft.Network/privateEndpoints/<workspace-private-endpoint>' is not using Azure default DNS. You need to configure your DNS server and check https://learn.microsoft.com/azure/machine-learning/how-to-custom-dns to make sure the custom DNS is set up correctly."  
        }],  
        "storage_account_results": [],  
        "key_vault_results": [],  
        "container_registry_results": [],  
        "application_insights_results": [],  
        "other_results": []  
    }  
}
```

If no problems are detected, an empty JSON document is returned.

For more information, see the [Workspace.diagnose_workspace\(\)](#) reference.

Next steps

- [Workspace.diagnose_workspace\(\)](#)
- [How to manage workspaces in portal or SDK](#)

Create and manage an Azure Machine Learning compute instance with CLI v1

9/21/2022 • 5 minutes to read • [Edit Online](#)

APPLIES TO: Azure CLI ml extension v1 Python SDK azureml v1

Learn how to create and manage a [compute instance](#) in your Azure Machine Learning workspace with CLI v1.

Use a compute instance as your fully configured and managed development environment in the cloud. For development and testing, you can also use the instance as a [training compute target](#) or for an [inference target](#). A compute instance can run multiple jobs in parallel and has a job queue. As a development environment, a compute instance can't be shared with other users in your workspace.

Compute instances can run jobs securely in a [virtual network environment](#), without requiring enterprises to open up SSH ports. The job executes in a containerized environment and packages your model dependencies in a Docker container.

In this article, you learn how to:

- Create a compute instance
- Manage (start, stop, restart, delete) a compute instance

NOTE

This article covers only how to do these tasks using CLI v1. For more recent ways to manage a compute instance, see [Create an Azure Machine Learning compute cluster](#).

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create an Azure Machine Learning workspace](#).
- The [Azure CLI extension for Machine Learning service \(v1\)](#) or [Azure Machine Learning Python SDK \(v1\)](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Create

IMPORTANT

Items marked (preview) below are currently in public preview. The preview version is provided without a service level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Time estimate: Approximately 5 minutes.

Creating a compute instance is a one time process for your workspace. You can reuse the compute as a development workstation or as a compute target for training. You can have multiple compute instances attached to your workspace.

The dedicated cores per region per VM family quota and total regional quota, which applies to compute instance creation, is unified and shared with Azure Machine Learning training compute cluster quota. Stopping the compute instance doesn't release quota to ensure you'll be able to restart the compute instance. It isn't possible to change the virtual machine size of compute instance once it's created.

The following example demonstrates how to create a compute instance:

- [Python SDK](#)
- [Azure CLI](#)

APPLIES TO:  [Python SDK azureml v1](#)

```
import datetime
import time

from azureml.core.compute import ComputeTarget, ComputeInstance
from azureml.core.compute_target import ComputeTargetException

# Choose a name for your instance
# Compute instance name should be unique across the azure region
compute_name = "ci{}".format(ws._workspace_id)[:10]

# Verify that instance does not exist already
try:
    instance = ComputeInstance(workspace=ws, name=compute_name)
    print('Found existing instance, use it.')
except ComputeTargetException:
    compute_config = ComputeInstance.provisioning_configuration(
        vm_size='STANDARD_D3_V2',
        ssh_public_access=False,
        # vnet_resourcegroup_name='<my-resource-group>',
        # vnet_name='<my-vnet-name>',
        # subnet_name='default',
        # admin_user_ssh_public_key='<my-sshkey>'
    )
    instance = ComputeInstance.create(ws, compute_name, compute_config)
    instance.wait_for_completion(show_output=True)
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- [ComputeInstance class](#)
- [ComputeTarget.create](#)
- [ComputeInstance.wait_for_completion](#)

Manage

Start, stop, restart, and delete a compute instance. A compute instance doesn't automatically scale down, so make sure to stop the resource to prevent ongoing charges. Stopping a compute instance deallocates it. Then start it again when you need it. While stopping the compute instance stops the billing for compute hours, you'll still be billed for disk, public IP and standard load balancer.

TIP

The compute instance has 120GB OS disk. If you run out of disk space, [use the terminal](#) to clear at least 1-2 GB before you stop or restart the compute instance. Please do not stop the compute instance by issuing sudo shutdown from the terminal. The temp disk size on compute instance depends on the VM size chosen and is mounted on /mnt.

- [Python SDK](#)
- [Azure CLI](#)

APPLIES TO:  [Python SDK azureml v1](#)

In the examples below, the name of the compute instance is `instance`.

- Get status

```
# get_status() gets the latest status of the ComputeInstance target
instance.get_status()
```

- Stop

```
# stop() is used to stop the ComputeInstance
# Stopping ComputeInstance will stop the billing meter and persist the state on the disk.
# Available Quota will not be changed with this operation.
instance.stop(wait_for_completion=True, show_output=True)
```

- Start

```
# start() is used to start the ComputeInstance if it is in stopped state
instance.start(wait_for_completion=True, show_output=True)
```

- Restart

```
# restart() is used to restart the ComputeInstance
instance.restart(wait_for_completion=True, show_output=True)
```

- Delete

```
# delete() is used to delete the ComputeInstance target. Useful if you want to re-use the compute
name
instance.delete(wait_for_completion=True, show_output=True)
```

[Azure RBAC](#) allows you to control which users in the workspace can create, delete, start, stop, restart a compute instance. All users in the workspace contributor and owner role can create, delete, start, stop, and restart compute instances across the workspace. However, only the creator of a specific compute instance, or the user assigned if it was created on their behalf, is allowed to access Jupyter, JupyterLab, and RStudio on that compute instance. A compute instance is dedicated to a single user who has root access. That user has access to Jupyter/JupyterLab/RStudio running on the instance. Compute instance will have single-user sign in and all actions will use that user's identity for Azure RBAC and attribution of experiment runs. SSH access is controlled

through public/private key mechanism.

These actions can be controlled by Azure RBAC:

- *Microsoft.MachineLearningServices/workspaces/computes/read*
- *Microsoft.MachineLearningServices/workspaces/computes/write*
- *Microsoft.MachineLearningServices/workspaces/computes/delete*
- *Microsoft.MachineLearningServices/workspaces/computes/start/action*
- *Microsoft.MachineLearningServices/workspaces/computes/stop/action*
- *Microsoft.MachineLearningServices/workspaces/computes/restart/action*
- *Microsoft.MachineLearningServices/workspaces/computes/updateSchedules/action*

To create a compute instance, you'll need permissions for the following actions:

- *Microsoft.MachineLearningServices/workspaces/computes/write*
- *Microsoft.MachineLearningServices/workspaces/checkComputeNameAvailability/action*

Next steps

- [Access the compute instance terminal](#)
- [Create and manage files](#)
- [Update the compute instance to the latest VM image](#)
- [Submit a training run](#)

Create an Azure Machine Learning compute cluster with CLI v1

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO: [Azure CLI ml extension v1](#) [Python SDK azureml v1](#)

Learn how to create and manage a [compute cluster](#) in your Azure Machine Learning workspace.

You can use Azure Machine Learning compute cluster to distribute a training or batch inference process across a cluster of CPU or GPU compute nodes in the cloud. For more information on the VM sizes that include GPUs, see [GPU-optimized virtual machine sizes](#).

In this article, learn how to:

- Create a compute cluster
- Lower your compute cluster cost
- Set up a [managed identity](#) for the cluster

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create an Azure Machine Learning workspace](#).
- The [Azure CLI extension for Machine Learning service \(v1\)](#), [Azure Machine Learning Python SDK](#), or the [Azure Machine Learning Visual Studio Code extension](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- If using the Python SDK, [set up your development environment with a workspace](#). Once your environment is set up, attach to the workspace in your Python script:

APPLIES TO: [Python SDK azureml v1](#)

```
from azureml.core import Workspace  
  
ws = Workspace.from_config()
```

What is a compute cluster?

Azure Machine Learning compute cluster is a managed-compute infrastructure that allows you to easily create a single or multi-node compute. The compute cluster is a resource that can be shared with other users in your workspace. The compute scales up automatically when a job is submitted, and can be put in an Azure Virtual Network. Compute cluster supports [no public IP \(preview\)](#) deployment as well in virtual network. The compute executes in a containerized environment and packages your model dependencies in a [Docker](#)

container.

Compute clusters can run jobs securely in a [virtual network environment](#), without requiring enterprises to open up SSH ports. The job executes in a containerized environment and packages your model dependencies in a Docker container.

Limitations

- Some of the scenarios listed in this document are marked as [preview](#). Preview functionality is provided without a service level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).
- Compute clusters can be created in a different region than your workspace. This functionality is in [preview](#), and is only available for **compute clusters**, not compute instances. This preview is not available if you are using a private endpoint-enabled workspace.

WARNING

When using a compute cluster in a different region than your workspace or datastores, you may see increased network latency and data transfer costs. The latency and costs can occur when creating the cluster, and when running jobs on it.

- We currently support only creation (and not updating) of clusters through [ARM templates](#). For updating compute, we recommend using the SDK, Azure CLI or UX for now.
- Azure Machine Learning Compute has default limits, such as the number of cores that can be allocated. For more information, see [Manage and request quotas for Azure resources](#).
- Azure allows you to place *locks* on resources, so that they cannot be deleted or are read only. **Do not apply resource locks to the resource group that contains your workspace**. Applying a lock to the resource group that contains your workspace will prevent scaling operations for Azure ML compute clusters. For more information on locking resources, see [Lock resources to prevent unexpected changes](#).

TIP

Clusters can generally scale up to 100 nodes as long as you have enough quota for the number of cores required. By default clusters are setup with inter-node communication enabled between the nodes of the cluster to support MPI jobs for example. However you can scale your clusters to 1000s of nodes by simply [raising a support ticket](#), and requesting to allow list your subscription, or workspace, or a specific cluster for disabling inter-node communication.

Create

Time estimate: Approximately 5 minutes.

Azure Machine Learning Compute can be reused across runs. The compute can be shared with other users in the workspace and is retained between runs, automatically scaling nodes up or down based on the number of runs submitted, and the `max_nodes` set on your cluster. The `min_nodes` setting controls the minimum nodes available.

The dedicated cores per region per VM family quota and total regional quota, which applies to compute cluster creation, is unified and shared with Azure Machine Learning training compute instance quota.

IMPORTANT

To avoid charges when no jobs are running, **set the minimum nodes to 0**. This setting allows Azure Machine Learning to de-allocate the nodes when they aren't in use. Any value larger than 0 will keep that number of nodes running, even if they are not in use.

The compute autoscales down to zero nodes when it isn't used. Dedicated VMs are created to run your jobs as needed.

- [Python SDK](#)
- [Azure CLI](#)

To create a persistent Azure Machine Learning Compute resource in Python, specify the `vm_size` and `max_nodes` properties. Azure Machine Learning then uses smart defaults for the other properties.

- `vm_size`: The VM family of the nodes created by Azure Machine Learning Compute.
- `max_nodes`: The max number of nodes to autoscale up to when you run a job on Azure Machine Learning Compute.

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# Choose a name for your CPU cluster
cpu_cluster_name = "cpucluster"

# Verify that cluster does not exist already
try:
    cpu_cluster = ComputeTarget(workspace=ws, name=cpu_cluster_name)
    print('Found existing cluster, use it.')
except ComputeTargetException:
    # To use a different region for the compute, add a location='<region>' parameter
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                            max_nodes=4)
    cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)

cpu_cluster.wait_for_completion(show_output=True)
```

You can also configure several advanced properties when you create Azure Machine Learning Compute. The properties allow you to create a persistent cluster of fixed size, or within an existing Azure Virtual Network in your subscription. See the [AmlCompute class](#) for details.

WARNING

When setting the `location` parameter, if it is a different region than your workspace or datastores you may see increased network latency and data transfer costs. The latency and costs can occur when creating the cluster, and when running jobs on it.

Lower your compute cluster cost

You may also choose to use [low-priority VMs](#) to run some or all of your workloads. These VMs do not have guaranteed availability and may be preempted while in use. You will have to restart a preempted job.

- [Python SDK](#)
- [Azure CLI](#)

APPLIES TO:  Python SDK azureml v1

```
compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                       vm_priority='lowpriority',
                                                       max_nodes=4)
```

Set up managed identity

Azure Machine Learning compute clusters also support [managed identities](#) to authenticate access to Azure resources without including credentials in your code. There are two types of managed identities:

- A **system-assigned managed identity** is enabled directly on the Azure Machine Learning compute cluster. The life cycle of a system-assigned identity is directly tied to the compute cluster. If the compute cluster is deleted, Azure automatically cleans up the credentials and the identity in Azure AD.
- A **user-assigned managed identity** is a standalone Azure resource provided through Azure Managed Identity service. You can assign a user-assigned managed identity to multiple resources, and it persists for as long as you want. This managed identity needs to be created beforehand and then passed as the `identity_id` as a required parameter.

NOTE

Managed identity is supported on [compute clusters](#), but is not supported on [compute instances](#).

- [Python SDK](#)
- [Azure CLI](#)

APPLIES TO:  Python SDK azureml v1

- Configure managed identity in your provisioning configuration:

- System assigned managed identity created in a workspace named `ws`

```
# configure cluster with a system-assigned managed identity
compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                       max_nodes=5,
                                                       identity_type="SystemAssigned",
                                                       )
cpu_cluster_name = "cpu-cluster"
cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)
```

- User-assigned managed identity created in a workspace named `ws`

```
# configure cluster with a user-assigned managed identity
compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                       max_nodes=5,
                                                       identity_type="UserAssigned",
                                                       identity_id=
['/subscriptions/<subscription_id>/resourcegroups/<resource_group>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<user_assigned_identity>'])
cpu_cluster_name = "cpu-cluster"
cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)
```

- Add managed identity to an existing compute cluster named `cpu_cluster`
 - System-assigned managed identity:

```
# add a system-assigned managed identity
cpu_cluster.add_identity(identity_type="SystemAssigned")
```

- User-assigned managed identity:

```
# add a user-assigned managed identity
cpu_cluster.add_identity(identity_type="UserAssigned",
                        identity_id=
                        ['/subscriptions/<subscription_id>/resourcegroups/<resource_group>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<user_assigned_identity>'])
```

NOTE

Azure Machine Learning compute clusters support only **one system-assigned identity or multiple user-assigned identities**, not both concurrently.

Managed identity usage

The **default managed identity** is the system-assigned managed identity or the first user-assigned managed identity.

During a run there are two applications of an identity:

1. The system uses an identity to set up the user's storage mounts, container registry, and datastores.
 - In this case, the system will use the default-managed identity.
2. The user applies an identity to access resources from within the code for a submitted run
 - In this case, provide the *client_id* corresponding to the managed identity you want to use to retrieve a credential.
 - Alternatively, get the user-assigned identity's client ID through the *DEFAULT_IDENTITY_CLIENT_ID* environment variable.

For example, to retrieve a token for a datastore with the default-managed identity:

```
client_id = os.environ.get('DEFAULT_IDENTITY_CLIENT_ID')
credential = ManagedIdentityCredential(client_id=client_id)
token = credential.get_token('https://storage.azure.com/')
```

Troubleshooting

There is a chance that some users who created their Azure Machine Learning workspace from the Azure portal before the GA release might not be able to create AmlCompute in that workspace. You can either raise a support request against the service or create a new workspace through the portal or the SDK to unblock yourself immediately.

Stuck at resizing

If your Azure Machine Learning compute cluster appears stuck at resizing (0 -> 0) for the node state, this may be caused by Azure resource locks.

Azure allows you to place *locks* on resources, so that they cannot be deleted or are read only. **Locking a resource can lead to unexpected results**. Some operations that don't seem to modify the resource actually require actions that are blocked by the lock.

With Azure Machine Learning, applying a delete lock to the resource group for your workspace will prevent

scaling operations for Azure ML compute clusters. To work around this problem we recommend **removing** the lock from resource group and instead applying it to individual items in the group.

IMPORTANT

Do not apply the lock to the following resources:

RESOURCE NAME	RESOURCE TYPE
<GUID>-azurebatch-cloudservicenetworksecuritygroup	Network security group
<GUID>-azurebatch-cloudservicepublicip	Public IP address
<GUID>-azurebatch-cloudserviceloadbalancer	Load balancer

These resources are used to communicate with, and perform operations such as scaling on, the compute cluster. Removing the resource lock from these resources should allow autoscaling for your compute clusters.

For more information on resource locking, see [Lock resources to prevent unexpected changes](#).

Next steps

Use your compute cluster to:

- [Submit a training run](#)
- [Run batch inference](#).

Create and attach an Azure Kubernetes Service cluster

9/21/2022 • 17 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

APPLIES TO:  Azure CLI ml extension v1

Azure Machine Learning can deploy trained machine learning models to Azure Kubernetes Service. However, you must first either **create** an Azure Kubernetes Service (AKS) cluster from your Azure ML workspace, or **attach** an existing AKS cluster. This article provides information on both creating and attaching a cluster.

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create an Azure Machine Learning workspace](#).
- The [Azure CLI extension \(v1\) for Machine Learning service](#), [Azure Machine Learning Python SDK](#), or the [Azure Machine Learning Visual Studio Code extension](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- If you plan on using an Azure Virtual Network to secure communication between your Azure ML workspace and the AKS cluster, your workspace and its associated resources (storage, key vault, Azure Container Registry) must have private endpoints or service endpoints in the same VNET as AKS cluster's VNET. Please follow tutorial [create a secure workspace](#) to add those private endpoints or service endpoints to your VNET.

Limitations

- If you need a **Standard Load Balancer(SLB)** deployed in your cluster instead of a Basic Load Balancer(BLB), create a cluster in the AKS portal/CLI/SDK and then **attach** it to the AzureML workspace.
- If you have an Azure Policy that restricts the creation of Public IP addresses, then AKS cluster creation will fail. AKS requires a Public IP for [egress traffic](#). The egress traffic article also provides guidance to lock down egress traffic from the cluster through the Public IP, except for a few fully qualified domain names. There are 2 ways to enable a Public IP:
 - The cluster can use the Public IP created by default with the BLB or SLB, Or
 - The cluster can be created without a Public IP and then a Public IP is configured with a firewall with a user defined route. For more information, see [Customize cluster egress with a user-defined-route](#).

The AzureML control plane does not talk to this Public IP. It talks to the AKS control plane for deployments.

- To attach an AKS cluster, the service principal/user performing the operation must be assigned the **Owner or contributor** Azure role-based access control (Azure RBAC) role on the Azure resource group that contains the cluster. The service principal/user must also be assigned [Azure Kubernetes Service Cluster Admin Role](#) on the cluster.
- If you **attach** an AKS cluster, which has an [Authorized IP range enabled to access the API server](#), enable the AzureML control plane IP ranges for the AKS cluster. The AzureML control plane is deployed across paired regions and deploys inference pods on the AKS cluster. Without access to the API server, the inference pods cannot be deployed. Use the [IP ranges](#) for both the [paired regions](#) when enabling the IP ranges in an AKS cluster.

Authorized IP ranges only works with Standard Load Balancer.

- If you want to use a private AKS cluster (using Azure Private Link), you must create the cluster first, and then **attach** it to the workspace. For more information, see [Create a private Azure Kubernetes Service cluster](#).
- Using a [public fully qualified domain name \(FQDN\)](#) with a private AKS cluster is **not supported** with Azure Machine learning.
- The compute name for the AKS cluster MUST be unique within your Azure ML workspace. It can include letters, digits and dashes. It must start with a letter, end with a letter or digit, and be between 3 and 24 characters in length.
- If you want to deploy models to **GPU** nodes or **FPGA** nodes (or any specific SKU), then you must create a cluster with the specific SKU. There is no support for creating a secondary node pool in an existing cluster and deploying models in the secondary node pool.
- When creating or attaching a cluster, you can select whether to create the cluster for **dev-test** or **production**. If you want to create an AKS cluster for **development, validation, and testing** instead of production, set the **cluster purpose** to **dev-test**. If you do not specify the cluster purpose, a **production** cluster is created.

IMPORTANT

A **dev-test** cluster is not suitable for production level traffic and may increase inference times. Dev/test clusters also do not guarantee fault tolerance.

- When creating or attaching a cluster, if the cluster will be used for **production**, then it must contain at least **3 nodes**. For a **dev-test** cluster, it must contain at least 1 node.
- The Azure Machine Learning SDK does not provide support scaling an AKS cluster. To scale the nodes in the cluster, use the UI for your AKS cluster in the Azure Machine Learning studio. You can only change the node count, not the VM size of the cluster. For more information on scaling the nodes in an AKS cluster, see the following articles:
 - [Manually scale the node count in an AKS cluster](#)
 - [Set up cluster autoscaler in AKS](#)
- Do not directly update the cluster by using a YAML configuration.** While Azure Kubernetes Services supports updates via YAML configuration, Azure Machine Learning deployments will override your changes. The only two YAML fields that will not overwritten are **request limits** and **cpu and memory**.
- Creating an AKS cluster using the Azure Machine Learning studio UI, SDK, or CLI extension is **not idempotent**. Attempting to create the resource again will result in an error that a cluster with the same name already exists.

- Using an Azure Resource Manager template and the [Microsoft.MachineLearningServices/workspaces/computes](#) resource to create an AKS cluster is also **not idempotent**. If you attempt to use the template again to update an already existing resource, you will receive the same error.

Azure Kubernetes Service version

Azure Kubernetes Service allows you to create a cluster using a variety of Kubernetes versions. For more information on available versions, see [supported Kubernetes versions in Azure Kubernetes Service](#).

When **creating** an Azure Kubernetes Service cluster using one of the following methods, you *do not have a choice in the version* of the cluster that is created:

- Azure Machine Learning studio, or the Azure Machine Learning section of the Azure portal.
- Machine Learning extension for Azure CLI.
- Azure Machine Learning SDK.

These methods of creating an AKS cluster use the **default** version of the cluster. *The default version changes over time* as new Kubernetes versions become available.

When **attaching** an existing AKS cluster, we support all currently supported AKS versions.

IMPORTANT

Azure Kubernetes Service uses [Blobfuse FlexVolume driver](#) for the versions <=1.16 and [Blob CSI driver](#) for the versions >=1.17. Therefore, it is important to re-deploy or [update the web service](#) after cluster upgrade in order to deploy to correct blobfuse method for the cluster version.

NOTE

There may be edge cases where you have an older cluster that is no longer supported. In this case, the attach operation will return an error and list the currently supported versions.

You can attach **preview** versions. Preview functionality is provided without a service level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. Support for using preview versions may be limited. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Available and default versions

To find the available and default AKS versions, use the [Azure CLI](#) command `az aks get-versions`. For example, the following command returns the versions available in the West US region:

```
az aks get-versions -l westus -o table
```

The output of this command is similar to the following text:

KubernetesVersion	Upgrades
1.18.6(preview)	None available
1.18.4(preview)	1.18.6(preview)
1.17.9	1.18.4(preview), 1.18.6(preview)
1.17.7	1.17.9, 1.18.4(preview), 1.18.6(preview)
1.16.13	1.17.7, 1.17.9
1.16.10	1.16.13, 1.17.7, 1.17.9
1.15.12	1.16.10, 1.16.13
1.15.11	1.15.12, 1.16.10, 1.16.13

To find the default version that is used when **creating** a cluster through Azure Machine Learning, you can use the `--query` parameter to select the default version:

```
az aks get-versions -l westus --query "orchestrators[?default == `true`].orchestratorVersion" -o table
```

The output of this command is similar to the following text:

Result

1.16.13

If you'd like to **programmatically check the available versions**, use the Container Service Client - List Orchestrators REST API. To find the available versions, look at the entries where `orchestratorType` is `Kubernetes`. The associated `orchestrationVersion` entries contain the available versions that can be **attached** to your workspace.

To find the default version that is used when **creating** a cluster through Azure Machine Learning, find the entry where `orchestratorType` is `Kubernetes` and `default` is `true`. The associated `orchestratorVersion` value is the default version. The following JSON snippet shows an example entry:

```
...
{
    "orchestratorType": "Kubernetes",
    "orchestratorVersion": "1.16.13",
    "default": true,
    "upgrades": [
        {
            "orchestratorType": "",
            "orchestratorVersion": "1.17.7",
            "isPreview": false
        }
    ],
},
...
```

Create a new AKS cluster

Time estimate: Approximately 10 minutes.

Creating or attaching an AKS cluster is a one time process for your workspace. You can reuse this cluster for multiple deployments. If you delete the cluster or the resource group that contains it, you must create a new cluster the next time you need to deploy. You can have multiple AKS clusters attached to your workspace.

The following example demonstrates how to create a new AKS cluster using the SDK and CLI:

- [Python SDK](#)

- Azure CLI
- Studio

APPLIES TO:  Python SDK azureml v1

```
from azureml.core.compute import AksCompute, ComputeTarget

# Use the default configuration (you can also provide parameters to customize this).
# For example, to create a dev/test cluster, use:
# prov_config = AksCompute.provisioning_configuration(cluster_purpose = AksCompute.ClusterPurpose.DEV_TEST)
prov_config = AksCompute.provisioning_configuration()

# Example configuration to use an existing virtual network
# prov_config.vnet_name = "mynetwork"
# prov_config.vnet_resourcegroup_name = "mygroup"
# prov_config.subnet_name = "default"
# prov_config.service_cidr = "10.0.0.0/16"
# prov_config.dns_service_ip = "10.0.0.10"
# prov_config.docker_bridge_cidr = "172.17.0.1/16"

aks_name = 'myaks'
# Create the cluster
aks_target = ComputeTarget.create(workspace = ws,
                                   name = aks_name,
                                   provisioning_configuration = prov_config)

# Wait for the create process to complete
aks_target.wait_for_completion(show_output = True)
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- [AksCompute.ClusterPurpose](#)
- [AksCompute.provisioning_configuration](#)
- [ComputeTarget.create](#)
- [ComputeTarget.wait_for_completion](#)

Attach an existing AKS cluster

Time estimate: Approximately 5 minutes.

If you already have AKS cluster in your Azure subscription, you can use it with your workspace.

TIP

The existing AKS cluster can be in a Azure region other than your Azure Machine Learning workspace.

WARNING

Do not create multiple, simultaneous attachments to the same AKS cluster from your workspace. For example, attaching one AKS cluster to a workspace using two different names. Each new attachment will break the previous existing attachment(s).

If you want to re-attach an AKS cluster, for example to change TLS or other cluster configuration setting, you must first remove the existing attachment by using [AksCompute.detach\(\)](#).

For more information on creating an AKS cluster using the Azure CLI or portal, see the following articles:

- [Create an AKS cluster \(CLI\)](#)
- [Create an AKS cluster \(portal\)](#)
- [Create an AKS cluster \(ARM Template on Azure Quickstart Templates\)](#)

The following example demonstrates how to attach an existing AKS cluster to your workspace:

- [Python SDK](#)
- [Azure CLI](#)
- [Studio](#)

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core.compute import AksCompute, ComputeTarget
# Set the resource group that contains the AKS cluster and the cluster name
resource_group = 'myresourcegroup'
cluster_name = 'myexistingcluster'

# Attach the cluster to your workgroup. If the cluster has less than 12 virtual CPUs, use the following instead:
# attach_config = AksCompute.attach_configuration(resource_group = resource_group,
#                                                 cluster_name = cluster_name,
#                                                 cluster_purpose = AksCompute.ClusterPurpose.DEV_TEST)
attach_config = AksCompute.attach_configuration(resource_group = resource_group,
                                                cluster_name = cluster_name)
aks_target = ComputeTarget.attach(ws, 'myaks', attach_config)

# Wait for the attach process to complete
aks_target.wait_for_completion(show_output = True)
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- [AksCompute.attach_configuration\(\)](#)
- [AksCompute.ClusterPurpose](#)
- [AksCompute.attach](#)

Create or attach an AKS cluster with TLS termination

When you [create or attach an AKS cluster](#), you can enable TLS termination with [AksCompute.provisioning_configuration\(\)](#) and [AksCompute.attach_configuration\(\)](#) configuration objects. Both methods return a configuration object that has an `enable_ssl` method, and you can use `enable_ssl` method to enable TLS.

Following example shows how to enable TLS termination with automatic TLS certificate generation and configuration by using Microsoft certificate under the hood.

APPLIES TO:  [Python SDK azureml v1](#)

```

from azureml.core.compute import AksCompute, ComputeTarget

# Enable TLS termination when you create an AKS cluster by using provisioning_config object enable_ssl
method

# Leaf domain label generates a name using the formula
# "<leaf-domain-label>#####.<azure-region>.cloudapp.azure.com"
# where "#####" is a random series of characters
provisioning_config.enable_ssl(leaf_domain_label = "contoso")

# Enable TLS termination when you attach an AKS cluster by using attach_config object enable_ssl method

# Leaf domain label generates a name using the formula
# "<leaf-domain-label>#####.<azure-region>.cloudapp.azure.com"
# where "#####" is a random series of characters
attach_config.enable_ssl(leaf_domain_label = "contoso")

```

Following example shows how to enable TLS termination with custom certificate and custom domain name. With custom domain and certificate, you must update your DNS record to point to the IP address of scoring endpoint, please see [Update your DNS](#)

APPLIES TO:  Python SDK azureml v1

```

from azureml.core.compute import AksCompute, ComputeTarget

# Enable TLS termination with custom certificate and custom domain when creating an AKS cluster

provisioning_config.enable_ssl(ssl_cert_pem_file="cert.pem",
                               ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")

# Enable TLS termination with custom certificate and custom domain when attaching an AKS cluster

attach_config.enable_ssl(ssl_cert_pem_file="cert.pem",
                        ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")

```

NOTE

For more information about how to secure model deployment on AKS cluster, please see [use TLS to secure a web service through Azure Machine Learning](#)

Create or attach an AKS cluster to use Internal Load Balancer with private IP

When you create or attach an AKS cluster, you can configure the cluster to use an Internal Load Balancer. With an Internal Load Balancer, scoring endpoints for your deployments to AKS will use a private IP within the virtual network. Following code snippets show how to configure an Internal Load Balancer for an AKS cluster.

- [Create](#)
- [Attach](#)

APPLIES TO:  Python SDK azureml v1

To create an AKS cluster that uses an Internal Load Balancer, use the `load_balancer_type` and `load_balancer_subnet` parameters:

```
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute import AksCompute, ComputeTarget

# Change to the name of the subnet that contains AKS
subnet_name = "default"
# When you create an AKS cluster, you can specify Internal Load Balancer to be created with
provisioning_config object
provisioning_config = AksCompute.provisioning_configuration(load_balancer_type = 'InternalLoadBalancer',
load_balancer_subnet = subnet_name)

# Create the cluster
aks_target = ComputeTarget.create(workspace = ws,
                                   name = aks_name,
                                   provisioning_configuration = provisioning_config)

# Wait for the create process to complete
aks_target.wait_for_completion(show_output = True)
```

IMPORTANT

If your AKS cluster is configured with an Internal Load Balancer, using a Microsoft provided certificate is not supported and you must use [custom certificate to enable TLS](#).

NOTE

For more information about how to secure inferencing environment, please see [Secure an Azure Machine Learning Inferencing Environment](#)

Detach an AKS cluster

To detach a cluster from your workspace, use one of the following methods:

WARNING

Using the Azure Machine Learning studio, SDK, or the Azure CLI extension for machine learning to detach an AKS cluster **does not delete the AKS cluster**. To delete the cluster, see [Use the Azure CLI with AKS](#).

- [Python SDK](#)
- [Azure CLI](#)
- [Studio](#)

APPLIES TO:  [Python SDK azureml v1](#)

```
aks_target.detach()
```

Troubleshooting

Update the cluster

Updates to Azure Machine Learning components installed in an Azure Kubernetes Service cluster must be manually applied.

You can apply these updates by detaching the cluster from the Azure Machine Learning workspace and reattaching the cluster to the workspace.

APPLIES TO:  Python SDK azureml v1

```
compute_target = ComputeTarget(workspace=ws, name=clusterWorkspaceName)
compute_target.detach()
compute_target.wait_for_completion(show_output=True)
```

Before you can re-attach the cluster to your workspace, you need to first delete any `azureml-fe` related resources. If there is no active service in the cluster, you can delete your `azureml-fe` related resources with the following code.

```
kubectl delete sa azureml-fe
kubectl delete clusterrole azureml-fe-role
kubectl delete clusterrolebinding azureml-fe-binding
kubectl delete svc azureml-fe
kubectl delete svc azureml-fe-int-http
kubectl delete deploy azureml-fe
kubectl delete secret azuremlfessl
kubectl delete cm azuremlfeconfig
```

If TLS is enabled in the cluster, you will need to supply the TLS/SSL certificate and private key when reattaching the cluster.

APPLIES TO:  Python SDK azureml v1

```
attach_config = AksCompute.attach_configuration(resource_group=resourceGroup,
cluster_name=kubernetesClusterName)

# If SSL is enabled.
attach_config.enable_ssl(
    ssl_cert_pem_file="cert.pem",
    ssl_key_pem_file="key.pem",
    ssl_cname=sslCname)

attach_config.validate_configuration()

compute_target = ComputeTarget.attach(workspace=ws, name=args.clusterWorkspaceName,
attach_configuration=attach_config)
compute_target.wait_for_completion(show_output=True)
```

If you no longer have the TLS/SSL certificate and private key, or you are using a certificate generated by Azure Machine Learning, you can retrieve the files prior to detaching the cluster by connecting to the cluster using `kubectl` and retrieving the secret `azuremlfessl`.

```
kubectl get secret/azuremlfessl -o yaml
```

NOTE

Kubernetes stores the secrets in Base64-encoded format. You will need to Base64-decode the `cert.pem` and `key.pem` components of the secrets prior to providing them to `attach_config.enable_ssl`.

Webservice failures

Many webservice failures in AKS can be debugged by connecting to the cluster using `kubectl`. You can get the `kubeconfig.json` for an AKS cluster by running

APPLIES TO:  Azure CLI ml extension v1

```
az aks get-credentials -g <rg> -n <aks cluster name>
```

Delete azureml-fe related resources

After detaching cluster, if there is none active service in cluster, please delete the `azureml-fe` related resources before attaching again:

```
kubectl delete sa azureml-fe
kubectl delete clusterrole azureml-fe-role
kubectl delete clusterrolebinding azureml-fe-binding
kubectl delete svc azureml-fe
kubectl delete svc azureml-fe-int-http
kubectl delete deploy azureml-fe
kubectl delete secret azuremlfessl
kubectl delete cm azuremlfeconfig
```

Load balancers should not have public IPs

When trying to create or attach an AKS cluster, you may receive a message that the request has been denied because "Load Balancers should not have public IPs". This message is returned when an administrator has applied a policy that prevents using an AKS cluster with a public IP address.

To resolve this problem, create/attach the cluster by using the `load_balancer_type` and `load_balancer_subnet` parameters. For more information, see [Internal Load Balancer \(private IP\)](#).

Next steps

- [Use Azure RBAC for Kubernetes authorization](#)
- [How and where to deploy a model](#)

Use Managed identities with Azure Machine Learning CLI v1

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

APPLIES TO:  Azure CLI ml extension v1

Managed identities allow you to configure your workspace with the *minimum required permissions to access resources*.

When configuring Azure Machine Learning workspace in trustworthy manner, it is important to ensure that different services associated with the workspace have the correct level of access. For example, during machine learning workflow the workspace needs access to Azure Container Registry (ACR) for Docker images, and storage accounts for training data.

Furthermore, managed identities allow fine-grained control over permissions, for example you can grant or revoke access from specific compute resources to a specific ACR.

In this article, you'll learn how to use managed identities to:

- Configure and use ACR for your Azure Machine Learning workspace without having to enable admin user access to ACR.
- Access a private ACR external to your workspace, to pull base images for training or inference.
- Create workspace with user-assigned managed identity to access associated resources.

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create workspace resources](#).
- The [Azure CLI extension for Machine Learning service](#)

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- The [Azure Machine Learning Python SDK](#).
- To assign roles, the login for your Azure subscription must have the [Managed Identity Operator](#) role, or other role that grants the required actions (such as [Owner](#)).
- You must be familiar with creating and working with [Managed Identities](#).

Configure managed identities

In some situations, it's necessary to disallow admin user access to Azure Container Registry. For example, the ACR may be shared and you need to disallow admin access by other users. Or, creating ACR with admin user

enabled is disallowed by a subscription level policy.

IMPORTANT

When using Azure Machine Learning for inference on Azure Container Instance (ACI), admin user access on ACR is required. Do not disable it if you plan on deploying models to ACI for inference.

When you create ACR without enabling admin user access, managed identities are used to access the ACR to build and pull Docker images.

You can bring your own ACR with admin user disabled when you create the workspace. Alternatively, let Azure Machine Learning create workspace ACR and disable admin user afterwards.

Bring your own ACR

If ACR admin user is disallowed by subscription policy, you should first create ACR without admin user, and then associate it with the workspace. Also, if you have existing ACR with admin user disabled, you can attach it to the workspace.

[Create ACR from Azure CLI](#) without setting `--admin-enabled` argument, or from Azure portal without enabling admin user. Then, when creating Azure Machine Learning workspace, specify the Azure resource ID of the ACR. The following example demonstrates creating a new Azure ML workspace that uses an existing ACR:

TIP

To get the value for the `--container-registry` parameter, use the `az acr show` command to show information for your ACR. The `id` field contains the resource ID for your ACR.

```
az ml workspace create -w <workspace name> \
-g <workspace resource group> \
-l <region> \
--container-registry /subscriptions/<subscription id>/resourceGroups/<acr resource group>/providers/Microsoft.ContainerRegistry/registries/<acr name>
```

Let Azure Machine Learning service create workspace ACR

If you do not bring your own ACR, Azure Machine Learning service will create one for you when you perform an operation that needs one. For example, submit a training run to Machine Learning Compute, build an environment, or deploy a web service endpoint. The ACR created by the workspace will have admin user enabled, and you need to disable the admin user manually.

1. Create a new workspace

```
az ml workspace show -n <my workspace> -g <my resource group>
```

2. Perform an action that requires ACR. For example, the [tutorial on training a model](#).

3. Get the ACR name created by the cluster:

```
az ml workspace show -w <my workspace> \
-g <my resource group>
--query containerRegistry
```

This command returns a value similar to the following text. You only want the last portion of the text, which is the ACR instance name:

```
/subscriptions/<subscription id>/resourceGroups/<my resource group>/providers/MicrosoftContainerRegistry/registries/<ACR instance name>
```

4. Update the ACR to disable the admin user:

```
az acr update --name <ACR instance name> --admin-enabled false
```

Create compute with managed identity to access Docker images for training

To access the workspace ACR, create machine learning compute cluster with system-assigned managed identity enabled. You can enable the identity from Azure portal or Studio when creating compute, or from Azure CLI using the below. For more information, see [using managed identity with compute clusters](#).

- [Python SDK](#)
- [Azure CLI](#)
- [Studio](#)

When creating a compute cluster with the [AmlComputeProvisioningConfiguration](#), use the `identity_type` parameter to set the managed identity type.

A managed identity is automatically granted ACRPull role on workspace ACR to enable pulling Docker images for training.

NOTE

If you create compute first, before workspace ACR has been created, you have to assign the ACRPull role manually.

Access base images from private ACR

By default, Azure Machine Learning uses Docker base images that come from a public repository managed by Microsoft. It then builds your training or inference environment on those images. For more information, see [What are ML environments?](#).

To use a custom base image internal to your enterprise, you can use managed identities to access your private ACR. There are two use cases:

- Use base image for training as is.
- Build Azure Machine Learning managed image with custom image as a base.

Pull Docker base image to machine learning compute cluster for training as is

Create machine learning compute cluster with system-assigned managed identity enabled as described earlier. Then, determine the principal ID of the managed identity.

APPLIES TO:  [Azure CLI ml extension v1](#)

```
az ml computetarget amlcompute identity show --name <cluster name> -w <workspace> -g <resource group>
```

Optionally, you can update the compute cluster to assign a user-assigned managed identity:

APPLIES TO:  [Azure CLI ml extension v1](#)

```
az ml computetarget amlcompute identity assign --name <cluster name> \
-w $mlws -g $mlrg --identities <my-identity-id>
```

To allow the compute cluster to pull the base images, grant the managed service identity ACRPull role on the private ACR

```
az role assignment create --assignee <principal ID> \  
--role acrpull \  
--scope "/subscriptions/<subscription ID>/resourceGroups/<private ACR resource  
group>/providers/Microsoft.ContainerRegistry/registries/<private ACR name>"
```

Finally, when submitting a training run, specify the base image location in the [environment definition](#).

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core import Environment  
env = Environment(name="private-acr")  
env.docker.base_image = "<ACR name>.azurecr.io/<base image repository>/<base image version>"  
env.python.user_managed_dependencies = True
```

IMPORTANT

To ensure that the base image is pulled directly to the compute resource, set `user_managed_dependencies = True` and do not specify a Dockerfile. Otherwise Azure Machine Learning service will attempt to build a new Docker image and fail, because only the compute cluster has access to pull the base image from ACR.

Build Azure Machine Learning managed environment into base image from private ACR for training or inference

APPLIES TO:  [Azure CLI ml extension v1](#)

In this scenario, Azure Machine Learning service builds the training or inference environment on top of a base image you supply from a private ACR. Because the image build task happens on the workspace ACR using ACR Tasks, you must perform more steps to allow access.

1. Create **user-assigned managed identity** and grant the identity ACRPull access to the **private ACR**.
2. Grant the workspace **system-assigned managed identity** a Managed Identity Operator role on the **user-assigned managed identity** from the previous step. This role allows the workspace to assign the user-assigned managed identity to ACR Task for building the managed environment.
 - a. Obtain the principal ID of workspace system-assigned managed identity:

```
az ml workspace show -w <workspace name> -g <resource group> --query identityPrincipalId
```

- b. Grant the Managed Identity Operator role:

```
az role assignment create --assignee <principal ID> --role managedidentityoperator --scope  
<user-assigned managed identity resource ID>
```

The user-assigned managed identity resource ID is Azure resource ID of the user assigned identity, in the format

```
/subscriptions/<subscription ID>/resourceGroups/<resource  
group>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<user-assigned managed  
identity name>
```

3. Specify the external ACR and client ID of the **user-assigned managed identity** in workspace connections by using [Workspace.set_connection method](#):

APPLIES TO:  Python SDK azureml v1

```
workspace.set_connection(  
    name="privateAcr",  
    category="ACR",  
    target = "<acr url>",  
    authType = "RegistryConnection",  
    value={"ResourceId": "<user-assigned managed identity resource id>", "ClientId": "<user-assigned managed identity client ID>"})
```

- Once the configuration is complete, you can use the base images from private ACR when building environments for training or inference. The following code snippet demonstrates how to specify the base image ACR and image name in an environment definition:

APPLIES TO:  Python SDK azureml v1

```
from azureml.core import Environment  
  
env = Environment(name="my-env")  
env.docker.base_image = "<acr url>/my-repo/my-image:latest"
```

Optionally, you can specify the managed identity resource URL and client ID in the environment definition itself by using [RegistryIdentity](#). If you use registry identity explicitly, it overrides any workspace connections specified earlier:

APPLIES TO:  Python SDK azureml v1

```
from azureml.core.container_registry import RegistryIdentity  
  
identity = RegistryIdentity()  
identity.resource_id= "<user-assigned managed identity resource ID>"  
identity.client_id="<user-assigned managed identity client ID>"  
env.docker.base_image_registry.registry_identity=identity  
env.docker.base_image = "my-acr.azurecr.io/my-repo/my-image:latest"
```

Use Docker images for inference

Once you've configured ACR without admin user as described earlier, you can access Docker images for inference without admin keys from your Azure Kubernetes service (AKS). When you create or attach AKS to workspace, the cluster's service principal is automatically assigned ACRPull access to workspace ACR.

NOTE

If you bring your own AKS cluster, the cluster must have service principal enabled instead of managed identity.

Create workspace with user-assigned managed identity

When creating a workspace, you can bring your own [user-assigned managed identity](#) that will be used to access the associated resources: ACR, KeyVault, Storage, and App Insights.

IMPORTANT

When creating workspace with user-assigned managed identity, you must create the associated resources yourself, and grant the managed identity roles on those resources. Use the [role assignment ARM template](#) to make the assignments.

Use Azure CLI or Python SDK to create the workspace. When using the CLI, specify the ID using the `--primary-user-assigned-identity` parameter. When using the SDK, use `primary_user_assigned_identity`. The following are examples of using the Azure CLI and Python to create a new workspace using these parameters:

Azure CLI

APPLIES TO:  Azure CLI ml extension v1

```
az ml workspace create -w <workspace name> -g <resource group> --primary-user-assigned-identity <managed identity ARM ID>
```

Python

APPLIES TO:  Python SDK azureml v1

```
from azureml.core import Workspace

ws = Workspace.create(name="workspace name",
                      subscription_id="subscription id",
                      resource_group="resource group name",
                      primary_user_assigned_identity="managed identity ARM ID")
```

You can also use [an ARM template](#) to create a workspace with user-assigned managed identity.

For a workspace with [customer-managed keys for encryption](#), you can pass in a user-assigned managed identity to authenticate from storage to Key Vault. Use argument `user-assigned-identity-for-cmk-encryption` (CLI) or `user_assigned_identity_for_cmk_encryption` (SDK) to pass in the managed identity. This managed identity can be the same or different as the workspace primary user assigned managed identity.

Next steps

- Learn more about [enterprise security in Azure Machine Learning](#)
- Learn about [identity-based data access](#)
- Learn about [managed identities on compute cluster](#).

Set up authentication for Azure Machine Learning resources and workflows using SDK v1

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

Learn how to set up authentication to your Azure Machine Learning workspace. Authentication to your Azure Machine Learning workspace is based on **Azure Active Directory** (Azure AD) for most things. In general, there are four authentication workflows that you can use when connecting to the workspace:

- **Interactive:** You use your account in Azure Active Directory to either directly authenticate, or to get a token that is used for authentication. Interactive authentication is used during *experimentation and iterative development*. Interactive authentication enables you to control access to resources (such as a web service) on a per-user basis.
- **Service principal:** You create a service principal account in Azure Active Directory, and use it to authenticate or get a token. A service principal is used when you need an *automated process to authenticate* to the service without requiring user interaction. For example, a continuous integration and deployment script that trains and tests a model every time the training code changes.
- **Azure CLI session:** You use an active Azure CLI session to authenticate. Azure CLI authentication is used during *experimentation and iterative development*, or when you need an *automated process to authenticate* to the service using a pre-authenticated session. You can log in to Azure via the Azure CLI on your local workstation, without storing credentials in Python code or prompting the user to authenticate. Similarly, you can reuse the same scripts as part of continuous integration and deployment pipelines, while authenticating the Azure CLI with a service principal identity.
- **Managed identity:** When using the Azure Machine Learning SDK *on an Azure Virtual Machine*, you can use a managed identity for Azure. This workflow allows the VM to connect to the workspace using the managed identity, without storing credentials in Python code or prompting the user to authenticate. Azure Machine Learning compute clusters can also be configured to use a managed identity to access the workspace when *training models*.

Regardless of the authentication workflow used, Azure role-based access control (Azure RBAC) is used to scope the level of access (authorization) allowed to the resources. For example, an admin or automation process might have access to create a compute instance, but not use it, while a data scientist could use it, but not delete or create it. For more information, see [Manage access to Azure Machine Learning workspace](#).

Azure AD Conditional Access can be used to further control or restrict access to the workspace for each authentication workflow. For example, an admin can allow workspace access from managed devices only.

Prerequisites

- Create an [Azure Machine Learning workspace](#).
- [Configure your development environment](#) to install the Azure Machine Learning SDK, or use a [Azure Machine Learning compute instance](#) with the SDK already installed.

Azure Active Directory

All the authentication workflows for your workspace rely on Azure Active Directory. If you want users to authenticate using individual accounts, they must have accounts in your Azure AD. If you want to use service

principals, they must exist in your Azure AD. Managed identities are also a feature of Azure AD.

For more on Azure AD, see [What is Azure Active Directory authentication](#).

Once you've created the Azure AD accounts, see [Manage access to Azure Machine Learning workspace](#) for information on granting them access to the workspace and other operations in Azure Machine Learning.

Configure a service principal

To use a service principal (SP), you must first create the SP. Then grant it access to your workspace. As mentioned earlier, Azure role-based access control (Azure RBAC) is used to control access, so you must also decide what access to grant the SP.

IMPORTANT

When using a service principal, grant it the **minimum access required for the task** it is used for. For example, you would not grant a service principal owner or contributor access if all it is used for is reading the access token for a web deployment.

The reason for granting the least access is that a service principal uses a password to authenticate, and the password may be stored as part of an automation script. If the password is leaked, having the minimum access required for a specific tasks minimizes the malicious use of the SP.

The easiest way to create an SP and grant access to your workspace is by using the [Azure CLI](#). To create a service principal and grant it access to your workspace, use the following steps:

NOTE

You must be an admin on the subscription to perform all of these steps.

1. Authenticate to your Azure subscription:

```
az login
```

If the CLI can open your default browser, it will do so and load a sign-in page. Otherwise, you need to open a browser and follow the instructions on the command line. The instructions involve browsing to <https://aka.ms/devicelogin> and entering an authorization code.

If you have multiple Azure subscriptions, you can use the `az account set -s <subscription name or ID>` command to set the subscription. For more information, see [Use multiple Azure subscriptions](#).

For other methods of authenticating, see [Sign in with Azure CLI](#).

2. Create the service principal. In the following example, an SP named **ml-auth** is created:

```
az ad sp create-for-rbac --sdk-auth --name ml-auth --role Contributor --scopes /subscriptions/<subscription id>
```

The output will be a JSON similar to the following. Take note of the `clientId`, `clientSecret`, and `tenantId` fields, as you'll need them for other steps in this article.

```
{  
    "clientId": "your-client-id",  
    "clientSecret": "your-client-secret",  
    "subscriptionId": "your-sub-id",  
    "tenantId": "your-tenant-id",  
    "activeDirectoryEndpointUrl": "https://login.microsoftonline.com",  
    "resourceManagerEndpointUrl": "https://management.azure.com",  
    "activeDirectoryGraphResourceId": "https://graph.windows.net",  
    "sqlManagementEndpointUrl": "https://management.core.windows.net:5555",  
    "galleryEndpointUrl": "https://gallery.azure.com/",  
    "managementEndpointUrl": "https://management.core.windows.net"  
}
```

3. Retrieve the details for the service principal by using the `clientId` value returned in the previous step:

```
az ad sp show --id your-client-id
```

The following JSON is a simplified example of the output from the command. Take note of the `objectId` field, as you'll need its value for the next step.

```
{  
    "accountEnabled": "True",  
    "addIns": [],  
    "appDisplayName": "ml-auth",  
    ...  
    ...  
    ...  
    "objectId": "your-sp-object-id",  
    "objectType": "ServicePrincipal"  
}
```

4. To grant access to the workspace and other resources used by Azure Machine Learning, use the information in the following articles:

- [How to assign roles and actions in AzureML](#)
- [How to assign roles in the CLI](#)

IMPORTANT

Owner access allows the service principal to do virtually any operation in your workspace. It is used in this document to demonstrate how to grant access; in a production environment Microsoft recommends granting the service principal the minimum access needed to perform the role you intend it for. For information on creating a custom role with the access needed for your scenario, see [Manage access to Azure Machine Learning workspace](#).

Configure a managed identity

IMPORTANT

Managed identity is only supported when using the Azure Machine Learning SDK from an Azure Virtual Machine or with an Azure Machine Learning compute cluster.

Managed identity with a VM

1. Enable a [system-assigned managed identity for Azure resources on the VM](#).
2. From the [Azure portal](#), select your workspace and then select **Access Control (IAM)**.

3. Select **Add**, **Add Role Assignment** to open the **Add role assignment** page.

4. Assign the following role. For detailed steps, see [Assign Azure roles using the Azure portal](#).

SETTING	VALUE
Role	The role you want to assign.
Assign access to	Managed Identity
Members	The managed identity you created earlier

A role definition is a collection of permissions. You can use the built-in roles or you can create your own custom roles. [Learn more](#)

Name ↑↓	Description ↑↓	Type ↑↓	Category ↑↓	Details
Owner	Grants full access to manage all resources, including the ability to a...	BuiltinRole	General	View
Contributor	Grants full access to manage all resources, but does not allow you ...	BuiltinRole	General	View
Reader	View all resources, but does not allow you to make any changes.	BuiltinRole	General	View
AcrDelete	acr delete	BuiltinRole	Containers	View
AcrImageSigner	acr image signer	BuiltinRole	Containers	View
AcrPull	acr pull	BuiltinRole	Containers	View
AcrPush	acr push	BuiltinRole	Containers	View
AcrQuarantineReader	acr quarantine data reader	BuiltinRole	Containers	View
AcrQuarantineWriter	acr quarantine data writer	BuiltinRole	Containers	View

Managed identity with compute cluster

For more information, see [Set up managed identity for compute cluster](#).

Use interactive authentication

IMPORTANT

Interactive authentication uses your browser, and requires cookies (including 3rd party cookies). If you have disabled cookies, you may receive an error such as "we couldn't sign you in." This error may also occur if you have enabled [Azure AD Multi-Factor Authentication](#).

Most examples in the documentation and samples use interactive authentication. For example, when using the SDK there are two function calls that will automatically prompt you with a UI-based authentication flow:

- Calling the `from_config()` function will issue the prompt.

```
from azureml.core import Workspace
ws = Workspace.from_config()
```

The `from_config()` function looks for a JSON file containing your workspace connection information.

- Using the `Workspace` constructor to provide subscription, resource group, and workspace information, will also prompt for interactive authentication.

```
ws = Workspace(subscription_id="your-sub-id",
               resource_group="your-resource-group-id",
               workspace_name="your-workspace-name"
              )
```

TIP

If you have access to multiple tenants, you may need to import the class and explicitly define what tenant you are targeting. Calling the constructor for `InteractiveLoginAuthentication` will also prompt you to login similar to the calls above.

```
from azureml.core.authentication import InteractiveLoginAuthentication
interactive_auth = InteractiveLoginAuthentication(tenant_id="your-tenant-id")
```

When using the Azure CLI, the `az login` command is used to authenticate the CLI session. For more information, see [Get started with Azure CLI](#).

TIP

If you are using the SDK from an environment where you have previously authenticated interactively using the Azure CLI, you can use the `AzureCliAuthentication` class to authenticate to the workspace using the credentials cached by the CLI:

```
from azureml.core.authentication import AzureCliAuthentication
cli_auth = AzureCliAuthentication()
ws = Workspace(subscription_id="your-sub-id",
               resource_group="your-resource-group-id",
               workspace_name="your-workspace-name",
               auth=cli_auth
              )
```

Use service principal authentication

To authenticate to your workspace from the SDK, using a service principal, use the `ServicePrincipalAuthentication` class constructor. Use the values you got when creating the service provider as the parameters. The `tenant_id` parameter maps to `tenantID` from above, `service_principal_id` maps to `clientId`, and `service_principal_password` maps to `clientSecret`.

```
from azureml.core.authentication import ServicePrincipalAuthentication

sp = ServicePrincipalAuthentication(tenant_id="your-tenant-id", # tenantID
                                    service_principal_id="your-client-id", # clientId
                                    service_principal_password="your-client-secret") # clientSecret
```

The `sp` variable now holds an authentication object that you use directly in the SDK. In general, it's a good idea to store the ids/secrets used above in environment variables as shown in the following code. Storing in environment variables prevents the information from being accidentally checked into a GitHub repo.

```
import os

sp = ServicePrincipalAuthentication(tenant_id=os.environ['AML_TENANT_ID'],
                                    service_principal_id=os.environ['AML_PRINCIPAL_ID'],
                                    service_principal_password=os.environ['AML_PRINCIPAL_PASS'])
```

For automated workflows that run in Python and use the SDK primarily, you can use this object as-is in most cases for your authentication. The following code authenticates to your workspace using the auth object you created.

```
from azureml.core import Workspace

ws = Workspace.get(name="ml-example",
                   auth=sp,
                   subscription_id="your-sub-id",
                   resource_group="your-rg-name")
ws.get_details()
```

Use managed identity authentication

To authenticate to the workspace from a VM or compute cluster that is configured with a managed identity, use the `MsiAuthentication` class. The following example demonstrates how to use this class to authenticate to a workspace:

```
from azureml.core.authentication import MsiAuthentication

msi_auth = MsiAuthentication()

ws = Workspace(subscription_id="your-sub-id",
               resource_group="your-resource-group-id",
               workspace_name="your-workspace-name",
               auth=msi_auth
              )
```

Use Conditional Access

As an administrator, you can enforce [Azure AD Conditional Access policies](#) for users signing in to the workspace. For example, you can require two-factor authentication, or allow sign in only from managed devices. To use Conditional Access for Azure Machine Learning workspaces specifically, [assign the Conditional Access policy](#) to Machine Learning Cloud app.

Next steps

- [How to use secrets in training.](#)
- [How to configure authentication for models deployed as a web service.](#)
- [Consume an Azure Machine Learning model deployed as a web service.](#)

Secure Azure Machine Learning workspace resources using virtual networks (SDK/CLI v1)

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

APPLIES TO:  Azure CLI ml extension v1

Secure Azure Machine Learning workspace resources and compute environments using virtual networks (VNets). This article uses an example scenario to show you how to configure a complete virtual network.

TIP

This article is part of a series on securing an Azure Machine Learning workflow. See the other articles in this series:

- [Secure the workspace resources](#)
- [Secure the training environment \(v1\)](#)
- [Secure inference environment \(v1\)](#)
- [Enable studio functionality](#)
- [Use custom DNS](#)
- [Use a firewall](#)
- [API platform network isolation](#)

For a tutorial on creating a secure workspace, see [Tutorial: Create a secure workspace](#) or [Tutorial: Create a secure workspace using a template](#).

Prerequisites

This article assumes that you have familiarity with the following topics:

- [Azure Virtual Networks](#)
- [IP networking](#)
- [Azure Machine Learning workspace with private endpoint](#)
- [Network Security Groups \(NSG\)](#)
- [Network firewalls](#)

Example scenario

In this section, you learn how a common network scenario is set up to secure Azure Machine Learning communication with private IP addresses.

The following table compares how services access different parts of an Azure Machine Learning network with and without a VNet:

SCENARIO	WORKSPACE	ASSOCIATED RESOURCES	TRAINING COMPUTE ENVIRONMENT	INFERENCE COMPUTE ENVIRONMENT
No virtual network	Public IP	Public IP	Public IP	Public IP

SCENARIO	WORKSPACE	ASSOCIATED RESOURCES	TRAINING COMPUTE ENVIRONMENT	INFERRING COMPUTE ENVIRONMENT
Public workspace, all other resources in a virtual network	Public IP	Public IP (service endpoint) - or - Private IP (private endpoint)	Public IP	Private IP
Secure resources in a virtual network	Private IP (private endpoint)	Public IP (service endpoint) - or - Private IP (private endpoint)	Private IP	Private IP

- **Workspace** - Create a private endpoint for your workspace. The private endpoint connects the workspace to the vnet through several private IP addresses.
 - **Public access** - You can optionally enable public access for a secured workspace.
- **Associated resource** - Use service endpoints or private endpoints to connect to workspace resources like Azure storage, Azure Key Vault. For Azure Container Services, use a private endpoint.
 - **Service endpoints** provide the identity of your virtual network to the Azure service. Once you enable service endpoints in your virtual network, you can add a virtual network rule to secure the Azure service resources to your virtual network. Service endpoints use public IP addresses.
 - **Private endpoints** are network interfaces that securely connect you to a service powered by Azure Private Link. Private endpoint uses a private IP address from your VNet, effectively bringing the service into your VNet.
- **Training compute access** - Access training compute targets like Azure Machine Learning Compute Instance and Azure Machine Learning Compute Clusters with public or private IP addresses.
- **Inference compute access** - Access Azure Kubernetes Services (AKS) compute clusters with private IP addresses.

The next sections show you how to secure the network scenario described above. To secure your network, you must:

1. Secure the [workspace and associated resources](#).
2. Secure the [training environment \(v1\)](#).
3. Secure the [inferencing environment \(v1\)](#).
4. Optionally: [enable studio functionality](#).
5. Configure [firewall settings](#).
6. Configure [DNS name resolution](#).

Public workspace and secured resources

If you want to access the workspace over the public internet while keeping all the associated resources secured in a virtual network, use the following steps:

1. Create an [Azure Virtual Network](#) that will contain the resources used by the workspace.
2. Use **one** of the following options to create a publicly accessible workspace:
 - Create an Azure Machine Learning workspace that **does not** use the virtual network. For more information, see [Manage Azure Machine Learning workspaces](#).
 - Create a [Private Link-enabled workspace](#) to enable communication between your VNet and workspace. Then [enable public access to the workspace](#).

3. Add the following services to the virtual network by using *either* a service endpoint or a private endpoint. Also allow trusted Microsoft services to access these services:

SERVICE	ENDPOINT INFORMATION	ALLOW TRUSTED INFORMATION
Azure Key Vault	Service endpoint Private endpoint	Allow trusted Microsoft services to bypass this firewall
Azure Storage Account	Service and private endpoint Private endpoint	Grant access to trusted Azure services
Azure Container Registry	Private endpoint	Allow trusted services

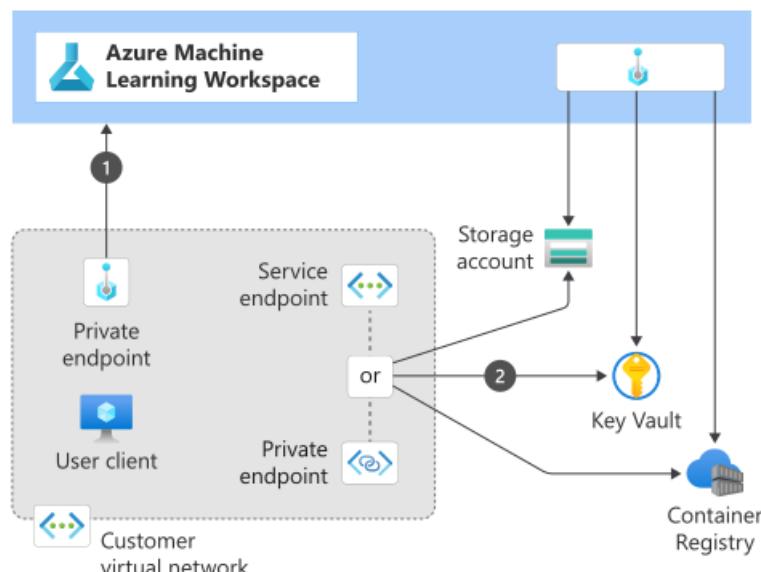
4. In properties for the Azure Storage Account(s) for your workspace, add your client IP address to the allowed list in firewall settings. For more information, see [Configure firewalls and virtual networks](#).

Secure the workspace and associated resources

Use the following steps to secure your workspace and associated resources. These steps allow your services to communicate in the virtual network.

- Create an [Azure Virtual Networks](#) that will contain the workspace and other resources. Then create a [Private Link-enabled workspace](#) to enable communication between your VNet and workspace.
- Add the following services to the virtual network by using *either* a service endpoint or a private endpoint. Also allow trusted Microsoft services to access these services:

SERVICE	ENDPOINT INFORMATION	ALLOW TRUSTED INFORMATION
Azure Key Vault	Service endpoint Private endpoint	Allow trusted Microsoft services to bypass this firewall
Azure Storage Account	Service and private endpoint Private endpoint	Grant access from Azure resource instances or Grant access to trusted Azure services
Azure Container Registry	Private endpoint	Allow trusted services



For detailed instructions on how to complete these steps, see [Secure an Azure Machine Learning workspace](#).

Limitations

Securing your workspace and associated resources within a virtual network have the following limitations:

- All resources must be behind the same VNet. However, subnets within the same VNet are allowed.

Secure the training environment

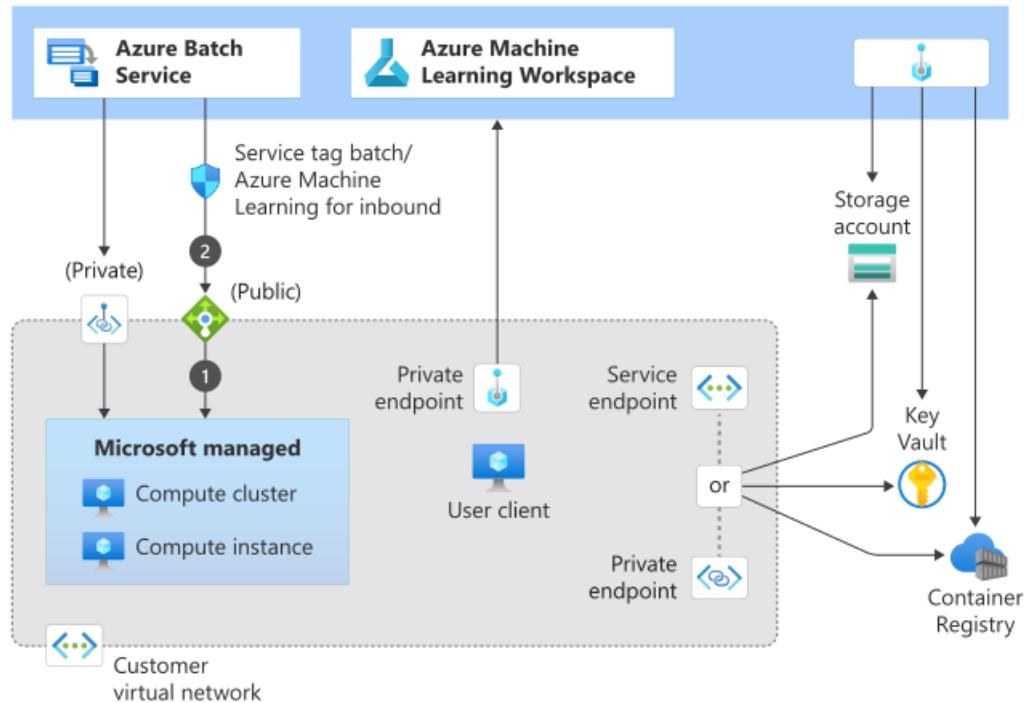
In this section, you learn how to secure the training environment in Azure Machine Learning. You also learn how Azure Machine Learning completes a training job to understand how the network configurations work together.

To secure the training environment, use the following steps:

1. Create an Azure Machine Learning [compute instance and computer cluster in the virtual network](#) to run the training job.
2. If your compute cluster or compute instance uses a public IP address, you must [Allow inbound communication](#) so that management services can submit jobs to your compute resources.

TIP

Compute cluster and compute instance can be created with or without a public IP address. If created with a public IP address, you get a load balancer with a public IP to accept the inbound access from Azure batch service and Azure Machine Learning service. You need to configure User Defined Routing (UDR) if you use a firewall. If created without a public IP, you get a private link service to accept the inbound access from Azure batch service and Azure Machine Learning service without a public IP.



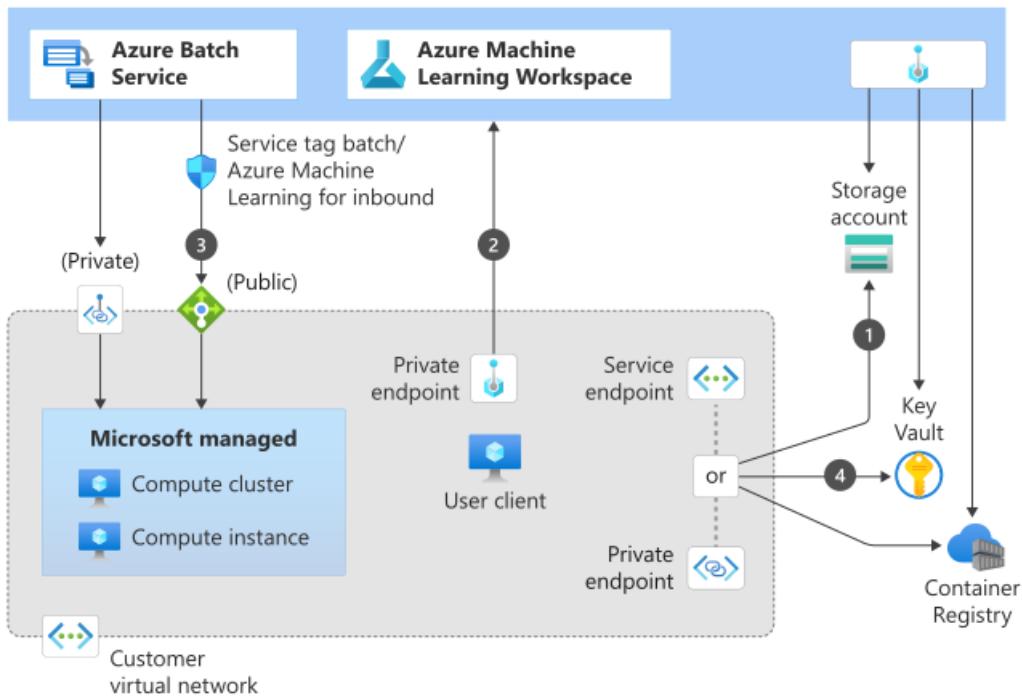
For detailed instructions on how to complete these steps, see [Secure a training environment](#).

Example training job submission

In this section, you learn how Azure Machine Learning securely communicates between services to submit a training job. This shows you how all your configurations work together to secure communication.

1. The client uploads training scripts and training data to storage accounts that are secured with a service or private endpoint.

2. The client submits a training job to the Azure Machine Learning workspace through the private endpoint.
3. Azure Batch service receives the job from the workspace. It then submits the training job to the compute environment through the public load balancer for the compute resource.
4. The compute resource receives the job and begins training. The compute resource uses information stored in key vault to access storage accounts to download training files and upload output.



Limitations

- Azure Compute Instance and Azure Compute Clusters must be in the same VNet, region, and subscription as the workspace and its associated resources.

Secure the inferencing environment (v1)

APPLIES TO: [Python SDK azureml v1](#)

APPLIES TO: [Azure CLI ml extension v1](#)

In this section, you learn the options available for securing an inferencing environment when using the Azure CLI extension for ML v1 or the Azure ML Python SDK v1. When doing a v1 deployment, we recommend that you use Azure Kubernetes Services (AKS) clusters for high-scale, production deployments.

You have two options for AKS clusters in a virtual network:

- Deploy or attach a default AKS cluster to your VNet.
- Attach a private AKS cluster to your VNet.

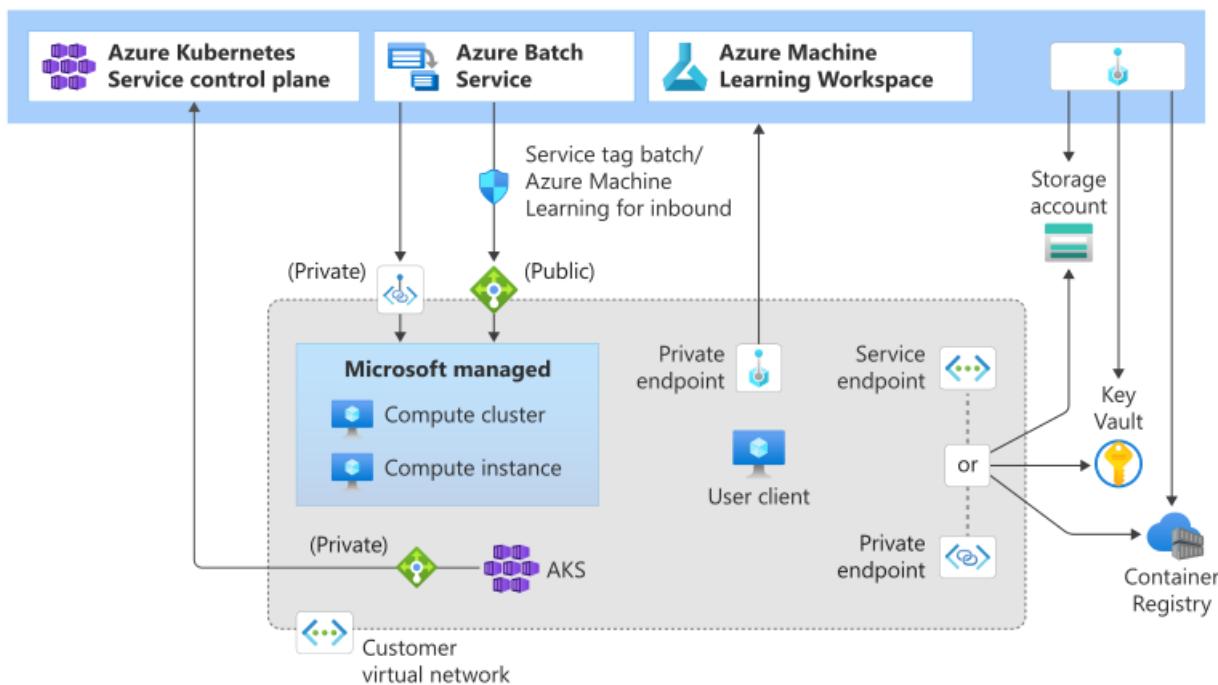
Default AKS clusters have a control plane with public IP addresses. You can add a default AKS cluster to your VNet during the deployment or attach a cluster after it's created.

Private AKS clusters have a control plane, which can only be accessed through private IPs. Private AKS clusters must be attached after the cluster is created.

For detailed instructions on how to add default and private clusters, see [Secure an inferencing environment](#).

Regardless default AKS cluster or private AKS cluster used, if your AKS cluster is behind of VNET, your workspace and its associate resources (storage, key vault, and ACR) must have private endpoints or service endpoints in the same VNET as the AKS cluster.

The following network diagram shows a secured Azure Machine Learning workspace with a private AKS cluster attached to the virtual network.



Optional: Enable public access

You can secure the workspace behind a VNet using a private endpoint and still allow access over the public internet. The initial configuration is the same as [securing the workspace and associated resources](#).

After securing the workspace with a private endpoint, use the following steps to enable clients to develop remotely using either the SDK or Azure Machine Learning studio:

1. [Enable public access](#) to the workspace.
2. [Configure the Azure Storage firewall](#) to allow communication with the IP address of clients that connect over the public internet.

Optional: Enable studio functionality

If your storage is in a VNet, you must use extra configuration steps to enable full functionality in studio. By default, the following features are disabled:

- Preview data in the studio.
- Visualize data in the designer.
- Deploy a model in the designer.
- Submit an AutoML experiment.
- Start a labeling project.

To enable full studio functionality, see [Use Azure Machine Learning studio in a virtual network](#).

Limitations

[ML-assisted data labeling](#) doesn't support a default storage account behind a virtual network. Instead, use a storage account other than the default for ML assisted data labeling.

TIP

As long as it is not the default storage account, the account used by data labeling can be secured behind the virtual network.

Configure firewall settings

Configure your firewall to control traffic between your Azure Machine Learning workspace resources and the public internet. While we recommend Azure Firewall, you can use other firewall products.

For more information on firewall settings, see [Use workspace behind a Firewall](#).

Custom DNS

If you need to use a custom DNS solution for your virtual network, you must add host records for your workspace.

For more information on the required domain names and IP addresses, see [how to use a workspace with a custom DNS server](#).

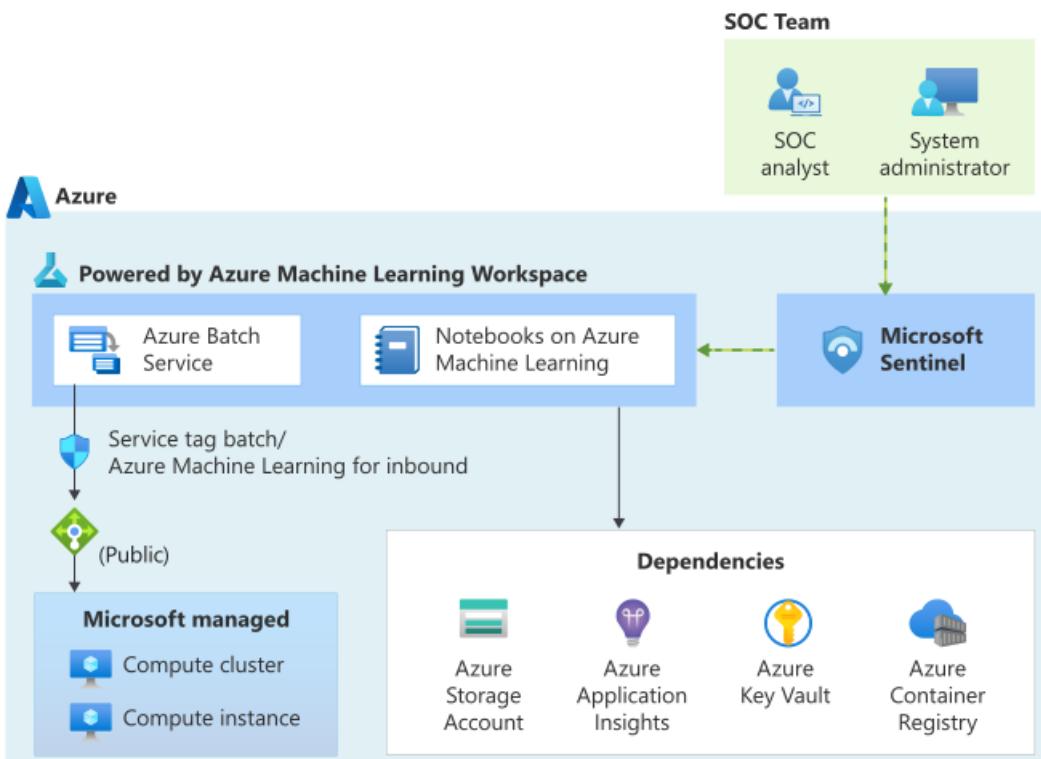
Microsoft Sentinel

Microsoft Sentinel is a security solution that can integrate with Azure Machine Learning. For example, using Jupyter notebooks provided through Azure Machine Learning. For more information, see [Use Jupyter notebooks to hunt for security threats](#).

Public access

Microsoft Sentinel can automatically create a workspace for you if you are OK with a public endpoint. In this configuration, the security operations center (SOC) analysts and system administrators connect to notebooks in your workspace through Sentinel.

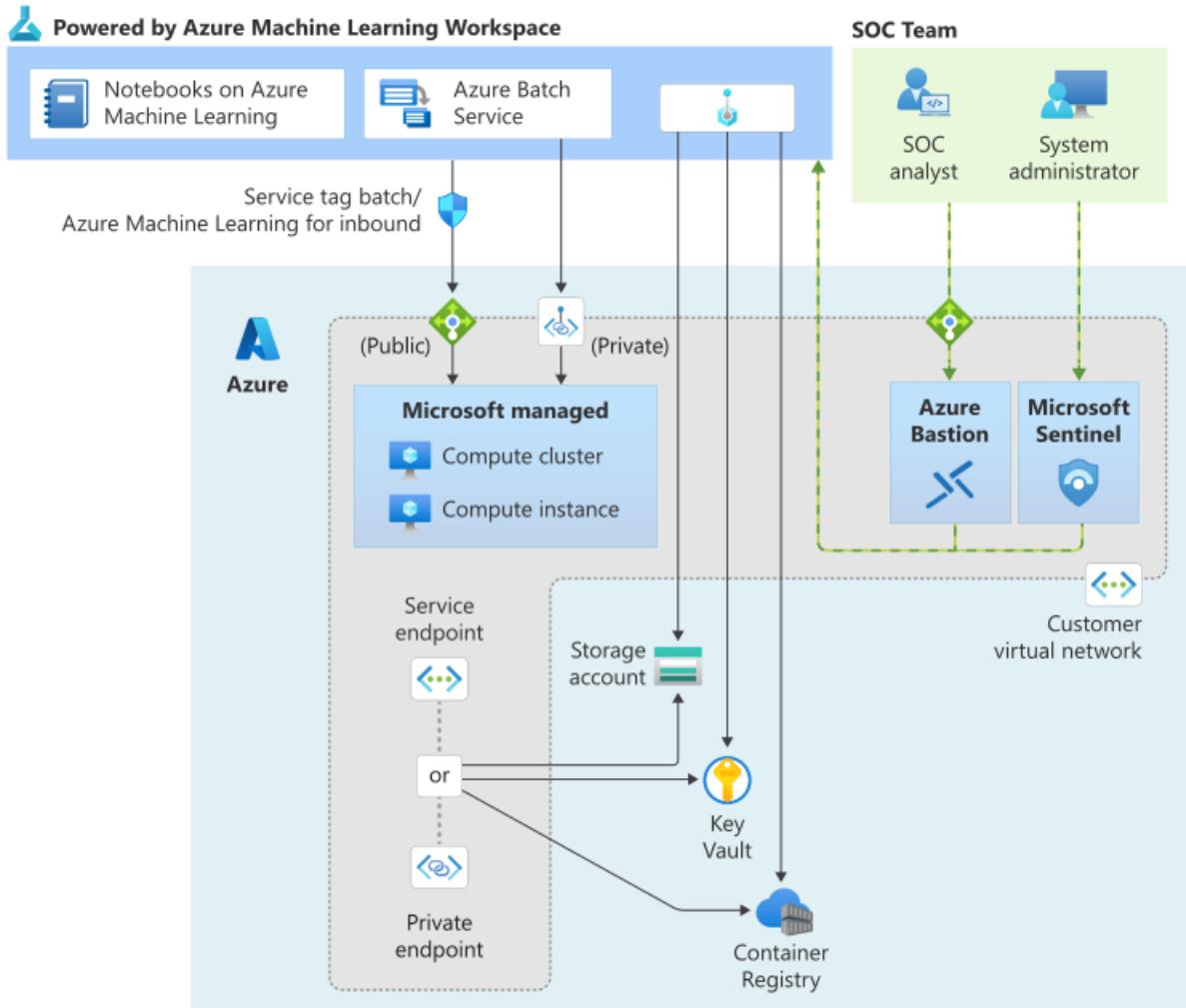
For information on this process, see [Create an Azure ML workspace from Microsoft Sentinel](#)



Private endpoint

If you want to secure your workspace and associated resources in a VNet, you must create the Azure Machine Learning workspace first. You must also create a virtual machine 'jump box' in the same VNet as your workspace, and enable Azure Bastion connectivity to it. Similar to the public configuration, SOC analysts and administrators can connect using Microsoft Sentinel, but some operations must be performed using Azure Bastion to connect to the VM.

For more information on this configuration, see [Create an Azure ML workspace from Microsoft Sentinel](#)



Next steps

This article is part of a series on securing an Azure Machine Learning workflow. See the other articles in this series:

- [Secure the workspace resources](#)
- [Secure the training environment \(v1\)](#)
- [Secure inference environment \(v1\)](#)
- [Enable studio functionality](#)
- [Use custom DNS](#)
- [Use a firewall](#)
- [API platform network isolation](#)

Configure a private endpoint for an Azure Machine Learning workspace with SDK and CLI v1

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO: [Azure CLI ml extension v1](#) [Python SDK azureml v1](#)

In this document, you learn how to configure a private endpoint for your Azure Machine Learning workspace. For information on creating a virtual network for Azure Machine Learning, see [Virtual network isolation and privacy overview](#).

Azure Private Link enables you to connect to your workspace using a private endpoint. The private endpoint is a set of private IP addresses within your virtual network. You can then limit access to your workspace to only occur over the private IP addresses. A private endpoint helps reduce the risk of data exfiltration. To learn more about private endpoints, see the [Azure Private Link](#) article.

WARNING

Securing a workspace with private endpoints does not ensure end-to-end security by itself. You must secure all of the individual components of your solution. For example, if you use a private endpoint for the workspace, but your Azure Storage Account is not behind the VNet, traffic between the workspace and storage does not use the VNet for security.

For more information on securing resources used by Azure Machine Learning, see the following articles:

- [Virtual network isolation and privacy overview](#).
- [Secure workspace resources](#).
- [Secure training environments \(v1\)](#).
- [Secure inference environment \(v1\)](#)
- [Use Azure Machine Learning studio in a VNet](#).
- [API platform network isolation](#).

Prerequisites

- You must have an existing virtual network to create the private endpoint in.
- [Disable network policies for private endpoints](#) before adding the private endpoint.

Limitations

- If you enable public access for a workspace secured with private endpoint and use Azure Machine Learning studio over the public internet, some features such as the designer may fail to access your data. This problem happens when the data is stored on a service that is secured behind the VNet. For example, an Azure Storage Account.
- You may encounter problems trying to access the private endpoint for your workspace if you're using Mozilla Firefox. This problem may be related to DNS over HTTPS in Mozilla Firefox. We recommend using Microsoft Edge or Google Chrome as a workaround.
- Using a private endpoint doesn't affect Azure control plane (management operations) such as deleting the workspace or managing compute resources. For example, creating, updating, or deleting a compute target. These operations are performed over the public Internet as normal. Data plane operations, such as using Azure Machine Learning studio, APIs (including published pipelines), or the SDK use the private

endpoint.

- When creating a compute instance or compute cluster in a workspace with a private endpoint, the compute instance and compute cluster must be in the same Azure region as the workspace.
- When creating or attaching an Azure Kubernetes Service cluster to a workspace with a private endpoint, the cluster must be in the same region as the workspace.
- When using a workspace with multiple private endpoints, one of the private endpoints must be in the same VNet as the following dependency services:
 - Azure Storage Account that provides the default storage for the workspace
 - Azure Key Vault for the workspace
 - Azure Container Registry for the workspace.

For example, one VNet ('services' VNet) would contain a private endpoint for the dependency services and the workspace. This configuration allows the workspace to communicate with the services. Another VNet ('clients') might only contain a private endpoint for the workspace, and be used only for communication between client development machines and the workspace.

Create a workspace that uses a private endpoint

Use one of the following methods to create a workspace with a private endpoint. Each of these methods **requires an existing virtual network**:

TIP

If you'd like to create a workspace, private endpoint, and virtual network at the same time, see [Use an Azure Resource Manager template to create a workspace for Azure Machine Learning](#).

- [Python SDK](#)
- [Azure CLI](#)

The Azure Machine Learning Python SDK provides the [PrivateEndpointConfig](#) class, which can be used with [Workspace.create\(\)](#) to create a workspace with a private endpoint. This class requires an existing virtual network.

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core import Workspace
from azureml.core import PrivateEndPointConfig

pe = PrivateEndPointConfig(name='myprivateendpoint', vnet_name='myvnet', vnet_subnet_name='default')
ws = Workspace.create(name='myworkspace',
    subscription_id='<my-subscription-id>',
    resource_group='myresourcegroup',
    location='eastus2',
    private_endpoint_config=pe,
    private_endpoint_auto_approval=True,
    show_output=True)
```

Add a private endpoint to a workspace

Use one of the following methods to add a private endpoint to an existing workspace:

WARNING

If you have any existing compute targets associated with this workspace, and they are not behind the same virtual network tha the private endpoint is created in, they will not work.

- [Python](#)
- [Azure CLI](#)

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core import Workspace
from azureml.core import PrivateEndPointConfig

pe = PrivateEndPointConfig(name='myprivateendpoint', vnet_name='myvnet', vnet_subnet_name='default')
ws = Workspace.from_config()
ws.add_private_endpoint(private_endpoint_config=pe, private_endpoint_auto_approval=True, show_output=True)
```

For more information on the classes and methods used in this example, see [PrivateEndpointConfig](#) and [Workspace.add_private_endpoint](#).

Remove a private endpoint

You can remove one or all private endpoints for a workspace. Removing a private endpoint removes the workspace from the VNet that the endpoint was associated with. Removing the private endpoint may prevent the workspace from accessing resources in that VNet, or resources in the VNet from accessing the workspace. For example, if the VNet doesn't allow access to or from the public internet.

WARNING

Removing the private endpoints for a workspace **doesn't make it publicly accessible**. To make the workspace publicly accessible, use the steps in the [Enable public access](#) section.

To remove a private endpoint, use the following information:

- [Python SDK](#)
- [Azure CLI](#)

To remove a private endpoint, use [Workspace.delete_private_endpoint_connection](#). The following example demonstrates how to remove a private endpoint:

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core import Workspace

ws = Workspace.from_config()
# get the connection name
_, _, connection_name = ws.get_details()['privateEndpointConnections'][0]['id'].rpartition('/')
ws.delete_private_endpoint_connection(private_endpoint_connection_name=connection_name)
```

Enable public access

In some situations, you may want to allow someone to connect to your secured workspace over a public endpoint, instead of through the VNet. Or you may want to remove the workspace from the VNet and re-enable public access.

IMPORTANT

Enabling public access doesn't remove any private endpoints that exist. All communications between components behind the VNet that the private endpoint(s) connect to are still secured. It enables public access only to the workspace, in addition to the private access through any private endpoints.

WARNING

When connecting over the public endpoint while the workspace uses a private endpoint to communicate with other resources:

- Some features of studio will fail to access your data. This problem happens when the *data is stored on a service that is secured behind the VNet*. For example, an Azure Storage Account.
- Using Jupyter, JupyterLab, and RStudio on a compute instance, including running notebooks, is not supported.

To enable public access, use the following steps:

TIP

There are two possible properties that you can configure:

- `allow_public_access_when_behind_vnet` - used by the Python SDK and CLI v2
- `public_network_access` - used by the Python SDK and CLI v2 Each property overrides the other. For example, setting `public_network_access` will override any previous setting to `allow_public_access_when_behind_vnet`.

Microsoft recommends using `public_network_access` to enable or disable public access to a workspace.

- [Python SDK](#)
- [Azure CLI](#)

To enable public access, use `Workspace.update` and set `allow_public_access_when_behind_vnet=True`.

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core import Workspace

ws = Workspace.from_config()
ws.update(allow_public_access_when_behind_vnet=True)
```

Securely connect to your workspace

To connect to a workspace that's secured behind a VNet, use one of the following methods:

- [Azure VPN gateway](#) - Connects on-premises networks to the VNet over a private connection. Connection is made over the public internet. There are two types of VPN gateways that you might use:
 - [Point-to-site](#): Each client computer uses a VPN client to connect to the VNet.
 - [Site-to-site](#): A VPN device connects the VNet to your on-premises network.
- [ExpressRoute](#) - Connects on-premises networks into the cloud over a private connection. Connection is made using a connectivity provider.
- [Azure Bastion](#) - In this scenario, you create an Azure Virtual Machine (sometimes called a jump box) inside the VNet. You then connect to the VM using Azure Bastion. Bastion allows you to connect to the VM using either an RDP or SSH session from your local web browser. You then use the jump box as your

development environment. Since it is inside the VNet, it can directly access the workspace. For an example of using a jump box, see [Tutorial: Create a secure workspace](#).

IMPORTANT

When using a **VPN gateway** or **ExpressRoute**, you will need to plan how name resolution works between your on-premises resources and those in the VNet. For more information, see [Use a custom DNS server](#).

If you have problems connecting to the workspace, see [Troubleshoot secure workspace connectivity](#).

Multiple private endpoints

Azure Machine Learning supports multiple private endpoints for a workspace. Multiple private endpoints are often used when you want to keep different environments separate. The following are some scenarios that are enabled by using multiple private endpoints:

- Client development environments in a separate VNet.
- An Azure Kubernetes Service (AKS) cluster in a separate VNet.
- Other Azure services in a separate VNet. For example, Azure Synapse and Azure Data Factory can use a Microsoft managed virtual network. In either case, a private endpoint for the workspace can be added to the managed VNet used by those services. For more information on using a managed virtual network with these services, see the following articles:
 - [Synapse managed private endpoints](#).
 - [Azure Data Factory managed virtual network](#).

IMPORTANT

[Synapse's data exfiltration protection](#) is not supported with Azure Machine Learning.

IMPORTANT

Each VNet that contains a private endpoint for the workspace must also be able to access the Azure Storage Account, Azure Key Vault, and Azure Container Registry used by the workspace. For example, you might create a private endpoint for the services in each VNet.

Adding multiple private endpoints uses the same steps as described in the [Add a private endpoint to a workspace](#) section.

Scenario: Isolated clients

If you want to isolate the development clients, so they don't have direct access to the compute resources used by Azure Machine Learning, use the following steps:

NOTE

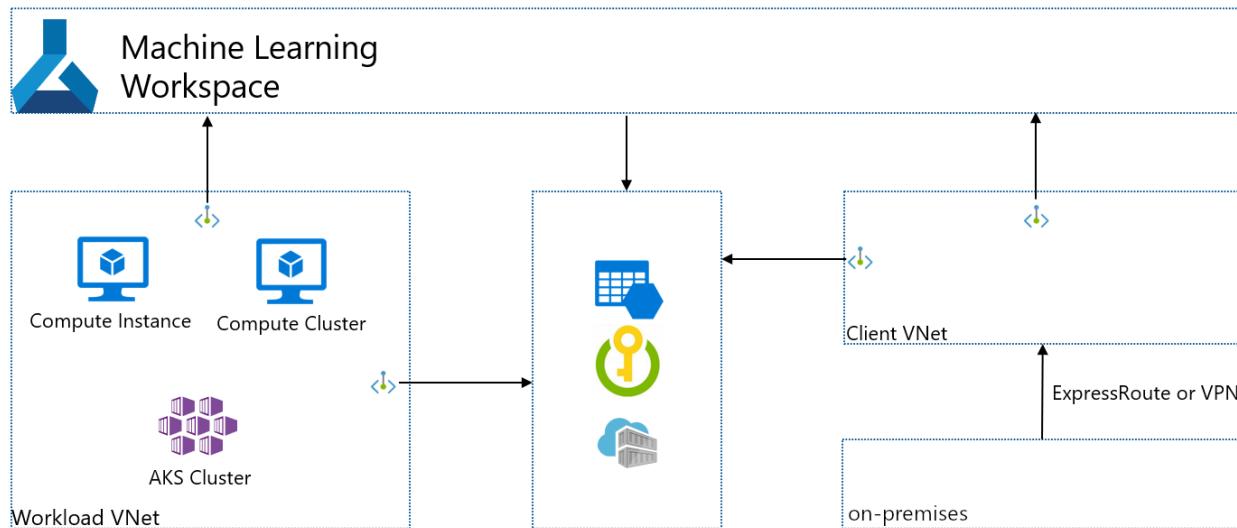
These steps assume that you have an existing workspace, Azure Storage Account, Azure Key Vault, and Azure Container Registry. Each of these services has a private endpoints in an existing VNet.

1. Create another VNet for the clients. This VNet might contain Azure Virtual Machines that act as your clients, or it may contain a VPN Gateway used by on-premises clients to connect to the VNet.
2. Add a new private endpoint for the Azure Storage Account, Azure Key Vault, and Azure Container Registry

used by your workspace. These private endpoints should exist in the client VNet.

3. If you have another storage that is used by your workspace, add a new private endpoint for that storage. The private endpoint should exist in the client VNet and have private DNS zone integration enabled.
4. Add a new private endpoint to your workspace. This private endpoint should exist in the client VNet and have private DNS zone integration enabled.
5. Use the steps in the [Use studio in a virtual network](#) article to enable studio to access the storage account(s).

The following diagram illustrates this configuration. The **Workload VNet** contains computes created by the workspace for training & deployment. The **Client VNet** contains clients or client ExpressRoute/VPN connections. Both VNets contain private endpoints for the workspace, Azure Storage Account, Azure Key Vault, and Azure Container Registry.



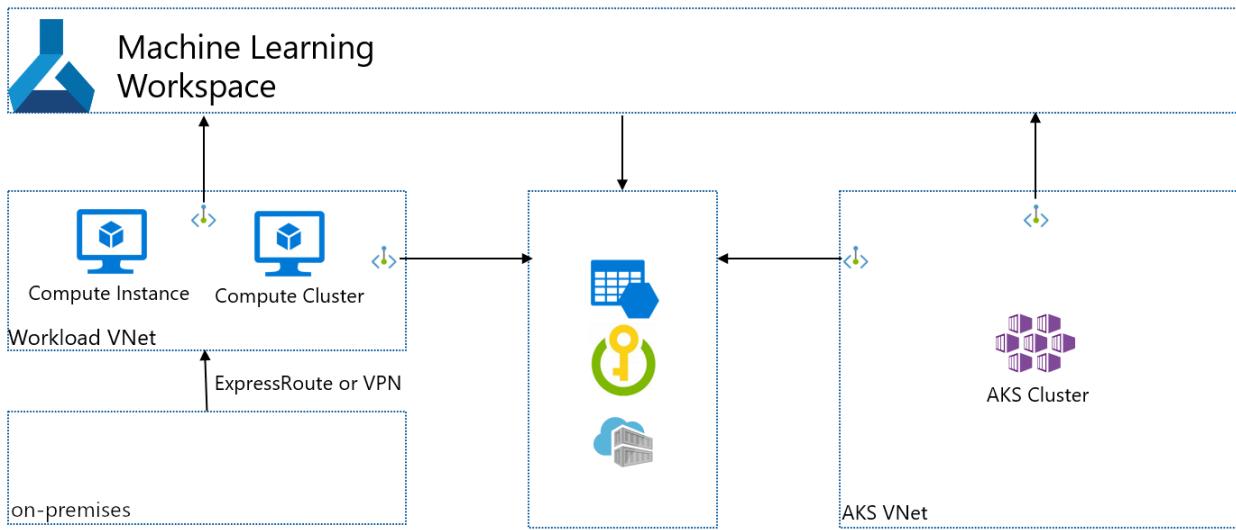
Scenario: Isolated Azure Kubernetes Service

If you want to create an isolated Azure Kubernetes Service used by the workspace, use the following steps:

NOTE

These steps assume that you have an existing workspace, Azure Storage Account, Azure Key Vault, and Azure Container Registry. Each of these services has a private endpoints in an existing VNet.

1. Create an Azure Kubernetes Service instance. During creation, AKS creates a VNet that contains the AKS cluster.
2. Add a new private endpoint for the Azure Storage Account, Azure Key Vault, and Azure Container Registry used by your workspace. These private endpoints should exist in the client VNet.
3. If you have other storage that is used by your workspace, add a new private endpoint for that storage. The private endpoint should exist in the client VNet and have private DNS zone integration enabled.
4. Add a new private endpoint to your workspace. This private endpoint should exist in the client VNet and have private DNS zone integration enabled.
5. Attach the AKS cluster to the Azure Machine Learning workspace. For more information, see [Create and attach an Azure Kubernetes Service cluster](#).



Next steps

- For more information on securing your Azure Machine Learning workspace, see the [Virtual network isolation and privacy overview](#) article.
- If you plan on using a custom DNS solution in your virtual network, see [how to use a workspace with a custom DNS server](#).
- [API platform network isolation](#)

Secure an Azure Machine Learning training environment with virtual networks (SDKv1)

9/21/2022 • 19 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to secure training environments with a virtual network in Azure Machine Learning using the Python SDK v1.

TIP

For information on using the Azure Machine Learning studio and the Python SDK v2, see [Secure training environment \(v2\)](#).

For a tutorial on creating a secure workspace, see [Tutorial: Create a secure workspace in Azure portal](#) or [Tutorial: Create a secure workspace using a template](#).

In this article you learn how to secure the following training compute resources in a virtual network:

- Azure Machine Learning compute cluster
- Azure Machine Learning compute instance
- Azure Databricks
- Virtual Machine
- HDInsight cluster

Prerequisites

- Read the [Network security overview](#) article to understand common virtual network scenarios and overall virtual network architecture.
- An existing virtual network and subnet to use with your compute resources.
- To deploy resources into a virtual network or subnet, your user account must have permissions to the following actions in Azure role-based access control (Azure RBAC):
 - "Microsoft.Network/virtualNetworks/*/read" on the virtual network resource. This permission isn't needed for Azure Resource Manager (ARM) template deployments.
 - "Microsoft.Network/virtualNetworks/subnet/join/action" on the subnet resource.

For more information on Azure RBAC with networking, see the [Networking built-in roles](#)

Azure Machine Learning compute cluster/instance

- Compute clusters and instances create the following resources. If they're unable to create these resources (for example, if there's a resource lock on the resource group) then creation, scale out, or scale in, may fail.
 - IP address.
 - Network Security Group (NSG).
 - Load balancer.
- The virtual network must be in the same subscription as the Azure Machine Learning workspace.
- The subnet used for the compute instance or cluster must have enough unassigned IP addresses.

- A compute cluster can dynamically scale. If there aren't enough unassigned IP addresses, the cluster will be partially allocated.
- A compute instance only requires one IP address.
- To create a compute cluster or instance [without a public IP address](#) (a preview feature), your workspace must use a private endpoint to connect to the VNet. For more information, see [Configure a private endpoint for Azure Machine Learning workspace](#).
- If you plan to secure the virtual network by restricting traffic, see the [Required public internet access](#) section.
- The subnet used to deploy compute cluster/instance shouldn't be delegated to any other service. For example, it shouldn't be delegated to ACI.

Azure Databricks

- The virtual network must be in the same subscription and region as the Azure Machine Learning workspace.
- If the Azure Storage Account(s) for the workspace are also secured in a virtual network, they must be in the same virtual network as the Azure Databricks cluster.

Limitations

Azure Machine Learning compute cluster/instance

- If you put multiple compute instances or clusters in one virtual network, you may need to request a quota increase for one or more of your resources. The Machine Learning compute instance or cluster automatically allocates networking resources in the **resource group that contains the virtual network**. For each compute instance or cluster, the service allocates the following resources:
 - One network security group (NSG). This NSG contains the following rules, which are specific to compute cluster and compute instance:
 - Allow inbound TCP traffic on ports 29876-29877 from the `BatchNodeManagement` service tag.
 - Allow inbound TCP traffic on port 44224 from the `AzureMachineLearning` service tag.

The following screenshot shows an example of these rules:

Priority ↑↓	Name ↑↓	Port ↑↓	Protocol ↑↓	Source ↑↓	Destination ↑↓	Action ↑↓	
Inbound Security Rules							
120	BatchServiceRule	29876-29877	tcp	BatchNodeManagement.Sou...	Any	Allow	
149	⚠ NodeAgentRule-DenyAll	29876-29877	tcp	Any	Any	Deny	
160	JupyterServerPort	44224	tcp	AzureMachineLearning	Any	Allow	
170	SSH	22	tcp	Internet	Any	Deny	
65000	AllowVnetInBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow	
65001	AllowAzureLoadBalancerinBo...	Any	Any	AzureLoadBalancer	Any	Allow	
65500	DenyAllInBound	Any	Any	Any	Any	Deny	
Outbound Security Rules							
65000	AllowVnetOutBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow	
65001	AllowinternetOutBound	Any	Any	Any	Internet	Allow	
65500	DenyAllOutBound	Any	Any	Any	Any	Deny	

TIP

If your compute cluster or instance does not use a public IP address (a preview feature), these inbound NSG rules are not required.

- For compute cluster or instance, it's now possible to remove the public IP address (a preview feature). If you have Azure Policy assignments prohibiting Public IP creation, then deployment of the compute cluster or instance will succeed.
- One load balancer

For compute clusters, these resources are deleted every time the cluster scales down to 0 nodes and created when scaling up.

For a compute instance, these resources are kept until the instance is deleted. Stopping the instance doesn't remove the resources.

IMPORTANT

These resources are limited by the subscription's [resource quotas](#). If the virtual network resource group is locked then deletion of compute cluster/instance will fail. Load balancer cannot be deleted until the compute cluster/instance is deleted. Also please ensure there is no Azure Policy assignment which prohibits creation of network security groups.

- If you create a compute instance and plan to use the no public IP address configuration, your Azure Machine Learning workspace's managed identity must be assigned the **Reader** role for the virtual network that contains the workspace. For more information on assigning roles, see [Steps to assign an Azure role](#).
- If you have configured Azure Container Registry for your workspace behind the virtual network, you must use a compute cluster to build Docker images. You can't use a compute cluster with the no public IP address configuration. For more information, see [Enable Azure Container Registry](#).
- If the Azure Storage Accounts for the workspace are also in the virtual network, use the following guidance on subnet limitations:
 - If you plan to use Azure Machine Learning studio to visualize data or use designer, the storage account must be **in the same subnet as the compute instance or cluster**.
 - If you plan to use the **SDK**, the storage account can be in a different subnet.

NOTE

Adding a resource instance for your workspace or selecting the checkbox for "Allow trusted Microsoft services to access this account" is not sufficient to allow communication from the compute.

- When your workspace uses a private endpoint, the compute instance can only be accessed from inside the virtual network. If you use a custom DNS or hosts file, add an entry for `<instance-name>.instances.azureml.ms`. Map this entry to the private IP address of the workspace private endpoint. For more information, see the [custom DNS](#) article.
- Virtual network service endpoint policies don't work for compute cluster/instance system storage accounts.
- If storage and compute instance are in different regions, you may see intermittent timeouts.
- If the Azure Container Registry for your workspace uses a private endpoint to connect to the virtual network, you can't use a managed identity for the compute instance. To use a managed identity with the compute instance, don't put the container registry in the VNet.
- If you want to use Jupyter Notebooks on a compute instance:
 - Don't disable websocket communication. Make sure your network allows websocket communication to `*.instances.azureml.net` and `*.instances.azureml.ms`.
 - Make sure that your notebook is running on a compute resource behind the same virtual network and subnet as your data. When creating the compute instance, use **Advanced settings > Configure virtual network** to select the network and subnet.
- **Compute clusters** can be created in a different region than your workspace. This functionality is in

[preview](#), and is only available for **compute clusters**, not compute instances. When using a different region for the cluster, the following limitations apply:

- If your workspace associated resources, such as storage, are in a different virtual network than the cluster, set up global virtual network peering between the networks. For more information, see [Virtual network peering](#).
- You may see increased network latency and data transfer costs. The latency and costs can occur when creating the cluster, and when running jobs on it.

Guidance such as using NSG rules, user-defined routes, and input/output requirements, apply as normal when using a different region than the workspace.

WARNING

If you are using a **private endpoint-enabled workspace**, creating the cluster in a different region is not supported.

Azure Databricks

- In addition to the **databricks-private** and **databricks-public** subnets used by Azure Databricks, the **default** subnet created for the virtual network is also required.
- Azure Databricks doesn't use a private endpoint to communicate with the virtual network.

For more information on using Azure Databricks in a virtual network, see [Deploy Azure Databricks in your Azure Virtual Network](#).

Azure HDInsight or virtual machine

- Azure Machine Learning supports only virtual machines that are running Ubuntu.

Required public internet access

Azure Machine Learning requires both inbound and outbound access to the public internet. The following tables provide an overview of what access is required and what it is for. The **protocol** for all items is **TCP**. For service tags that end in `.region`, replace `region` with the Azure region that contains your workspace. For example,

`Storage.westus`:

DIRECTION	PORTS	SERVICE TAG	PURPOSE
Inbound	29876-29877	BatchNodeManagement	Create, update, and delete of Azure Machine Learning compute instance and compute cluster. It isn't required if you use No Public IP option.
Inbound	44224	AzureMachineLearning	Create, update, and delete of Azure Machine Learning compute instance. It isn't required if you use No Public IP option.
Outbound	80, 443	AzureActiveDirectory	Authentication using Azure AD.
Outbound	443, 8787, 18881	AzureMachineLearning	Using Azure Machine Learning services.

DIRECTION	PORTS	SERVICE TAG	PURPOSE
Outbound	443	AzureResourceManager	Creation of Azure resources with Azure Machine Learning.
Outbound	443, 445 (*)	Storage.region	<p>Access data stored in the Azure Storage Account for compute cluster and compute instance. This outbound can be used to exfiltrate data. For more information, see Data exfiltration protection.</p> <p>(*) 445 is only required if you have a firewall between your virtual network for Azure ML and a private endpoint for your storage accounts.</p>
Outbound	443	AzureFrontDoor.FrontEnd * Not needed in Azure China.	<p>Global entry point for Azure Machine Learning studio. Store images and environments for AutoML.</p>
Outbound	443	MicrosoftContainerRegistry.region Note that this tag has a dependency on the AzureFrontDoor.FirstParty tag	<p>Access docker images provided by Microsoft. Setup of the Azure Machine Learning router for Azure Kubernetes Service.</p>
Outbound	443	AzureMonitor	Used to log monitoring and metrics to App Insights and Azure Monitor.
Outbound	443	Keyvault.region	Access the key vault for the Azure Batch service. Only needed if your workspace was created with the hbi_workspace flag enabled.

TIP

If you need the IP addresses instead of service tags, use one of the following options:

- Download a list from [Azure IP Ranges and Service Tags](#).
- Use the Azure CLI `az network list-service-tags` command.
- Use the Azure PowerShell `Get-AzNetworkServiceTag` command.

The IP addresses may change periodically.

IMPORTANT

When using a compute cluster that is configured for **no public IP address**, you must allow the following traffic:

- **Inbound** from source of **VirtualNetwork** and any port source, to destination of **VirtualNetwork**, and destination port of **29876, 29877**.
- **Inbound** from source **AzureLoadBalancer** and any port source to destination **VirtualNetwork** and port **44224** destination.

You may also need to allow **outbound** traffic to Visual Studio Code and non-Microsoft sites for the installation of packages required by your machine learning project. The following table lists commonly used repositories for machine learning:

HOST NAME	PURPOSE
anaconda.com *.anaconda.com	Used to install default packages.
*.anaconda.org	Used to get repo data.
pypi.org	Used to list dependencies from the default index, if any, and the index isn't overwritten by user settings. If the index is overwritten, you must also allow *.pythonhosted.org.
cloud.r-project.org	Used when installing CRAN packages for R development.
*pytorch.org	Used by some examples based on PyTorch.
*.tensorflow.org	Used by some examples based on Tensorflow.
code.visualstudio.com	Required to download and install VS Code desktop. This is not required for VS Code Web.
update.code.visualstudio.com .vo.msecnd.net	Used to retrieve VS Code server bits that are installed on the compute instance through a setup script.
marketplace.visualstudio.com vscode.blob.core.windows.net *.gallerycdn.vsassets.io	Required to download and install VS Code extensions. These enable the remote connection to Compute Instances provided by the Azure ML extension for VS Code, see Connect to an Azure Machine Learning compute instance in Visual Studio Code for more information.
raw.githubusercontent.com/microsoft/vscode-tools-for-ai/master/azureml_remote_websocket_server/*	Used to retrieve websocket server bits, which are installed on the compute instance. The websocket server is used to transmit requests from Visual Studio Code client (desktop application) to Visual Studio Code server running on the compute instance.

When using Azure Kubernetes Service (AKS) with Azure Machine Learning, allow the following traffic to the AKS VNet:

- General inbound/outbound requirements for AKS as described in the [Restrict egress traffic in Azure Kubernetes Service](#) article.
- **Outbound** to mcr.microsoft.com.
- When deploying a model to an AKS cluster, use the guidance in the [Deploy ML models to Azure Kubernetes](#)

[Service](#) article.

For information on using a firewall solution, see [Use a firewall with Azure Machine Learning](#).

Compute cluster

APPLIES TO:  Python SDK azureml v1

The following code creates a new Machine Learning Compute cluster in the `default` subnet of a virtual network named `mynetwork`:

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# The Azure virtual network name, subnet, and resource group
vnet_name = 'mynetwork'
subnet_name = 'default'
vnet_resourcegroup_name = 'mygroup'

# Choose a name for your CPU cluster
cpu_cluster_name = "cpucluster"

# Verify that cluster does not exist already
try:
    cpu_cluster = ComputeTarget(workspace=ws, name=cpu_cluster_name)
    print("Found existing cputcluster")
except ComputeTargetException:
    print("Creating new cputcluster")

# Specify the configuration for the new cluster
compute_config = AmlCompute.provisioning_configuration(vm_size="STANDARD_D2_V2",
                                                       min_nodes=0,
                                                       max_nodes=4,
                                                       location="westus2",
                                                       vnet_resourcegroup_name=vnet_resourcegroup_name,
                                                       vnet_name=vnet_name,
                                                       subnet_name=subnet_name)

# Create the cluster with the specified name and configuration
cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)

# Wait for the cluster to be completed, show the output log
cpu_cluster.wait_for_completion(show_output=True)
```

When the creation process finishes, you train your model by using the cluster in an experiment. For more information, see [Select and use a compute target for training](#).

NOTE

You may choose to use [low-priority VMs](#) to run some or all of your workloads. See how to [create a low-priority VM](#).

No public IP for compute clusters (preview)

When you enable **No public IP**, your compute cluster doesn't use a public IP for communication with any dependencies. Instead, it communicates solely within the virtual network using Azure Private Link ecosystem and service/private endpoints, eliminating the need for a public IP entirely. No public IP removes access and discoverability of compute cluster nodes from the internet thus eliminating a significant threat vector. **No public IP** clusters help comply with no public IP policies many enterprises have.

WARNING

By default, you do not have public internet access from No Public IP Compute Cluster. You need to configure User Defined Routing (UDR) to reach to a public IP to access the internet. For example, you can use a public IP of your firewall, or you can use [Virtual Network NAT](#) with a public IP.

A compute cluster with **No public IP** enabled has **no inbound communication requirements** from public internet. Specifically, neither inbound NSG rule (`BatchNodeManagement`, `AzureMachineLearning`) is required. You still need to allow inbound from source of **VirtualNetwork** and any port source, to destination of **VirtualNetwork**, and destination port of **29876**, **29877** and inbound from source **AzureLoadBalancer** and any port source to destination **VirtualNetwork** and port **44224** destination.

No public IP clusters are dependent on [Azure Private Link](#) for Azure Machine Learning workspace. A compute cluster with **No public IP** also requires you to disable private endpoint network policies and private link service network policies. These requirements come from Azure private link service and private endpoints and aren't Azure Machine Learning specific. Follow instruction from [Disable network policies for Private Link service](#) to set the parameters `disable-private-endpoint-network-policies` and `disable-private-link-service-network-policies` on the virtual network subnet.

For **outbound connections** to work, you need to set up an egress firewall such as Azure firewall with user defined routes. For instance, you can use a firewall set up with [inbound/outbound configuration](#) and route traffic there by defining a route table on the subnet in which the compute cluster is deployed. The route table entry can set up the next hop of the private IP address of the firewall with the address prefix of 0.0.0.0/0.

You can use a service endpoint or private endpoint for your Azure container registry and Azure storage in the subnet in which cluster is deployed.

To create a no public IP address compute cluster (a preview feature) in studio, set **No public IP** checkbox in the virtual network section. You can also create no public IP compute cluster through an ARM template. In the ARM template set `enableNodePublicIP` parameter to false.

NOTE

Support for compute instances without public IP addresses is currently available and in public preview for the following regions: France Central, East Asia, West Central US, South Central US, West US 2, East US, East US 2, North Europe, West Europe, Central US, North Central US, West US, Australia East, Japan East, Japan West.

Support for compute clusters without public IP addresses is currently available and in public preview for the following regions: France Central, East Asia, West Central US, South Central US, West US 2, East US, North Europe, East US 2, Central US, West Europe, North Central US, West US, Australia East, Japan East, Japan West.

Troubleshooting

- If you get this error message during creation of cluster
`The specified subnet has PrivateLinkServiceNetworkPolicies or PrivateEndpointNetworkEndpoints enabled`, follow the instructions from [Disable network policies for Private Link service](#) and [Disable network policies for Private Endpoint](#).
- If job execution fails with connection issues to ACR or Azure Storage, verify that customer has added ACR and Azure Storage service endpoint/private endpoints to subnet and ACR/Azure Storage allows the access from the subnet.
- To ensure that you've created a no public IP cluster, in Studio when looking at cluster details you'll see **No Public IP** property is set to **true** under resource properties.

Compute instance

For steps on how to create a compute instance deployed in a virtual network, see [Create and manage an Azure Machine Learning compute instance](#).

No public IP for compute instances (preview)

When you enable **No public IP**, your compute instance doesn't use a public IP for communication with any dependencies. Instead, it communicates solely within the virtual network using Azure Private Link ecosystem and service/private endpoints, eliminating the need for a public IP entirely. No public IP removes access and discoverability of compute instance node from the internet thus eliminating a significant threat vector. Compute instances will also do packet filtering to reject any traffic from outside virtual network. **No public IP** instances are dependent on [Azure Private Link](#) for Azure Machine Learning workspace.

WARNING

By default, you do not have public internet access from No Public IP Compute Instance. You need to configure User Defined Routing (UDR) to reach to a public IP to access the internet. For example, you can use a public IP of your firewall, or you can use [Virtual Network NAT](#) with a public IP.

For **outbound connections** to work, you need to set up an egress firewall such as Azure firewall with user defined routes. For instance, you can use a firewall set up with [inbound/outbound configuration](#) and route traffic there by defining a route table on the subnet in which the compute instance is deployed. The route table entry can set up the next hop of the private IP address of the firewall with the address prefix of 0.0.0.0/0.

A compute instance with **No public IP** enabled has **no inbound communication requirements** from public internet. Specifically, neither inbound NSG rule (`BatchNodeManagement`, `AzureMachineLearning`) is required. You still need to allow inbound from source of **VirtualNetwork**, any port source, destination of **VirtualNetwork**, and destination port of **29876, 29877, 44224**.

A compute instance with **No public IP** also requires you to disable private endpoint network policies and private link service network policies. These requirements come from Azure private link service and private endpoints and aren't Azure Machine Learning specific. Follow instruction from [Disable network policies for Private Link service source IP](#) to set the parameters `disable-private-endpoint-network-policies` and `disable-private-link-service-network-policies` on the virtual network subnet.

To create a no public IP address compute instance (a preview feature) in studio, set **No public IP** checkbox in the virtual network section. You can also create no public IP compute instance through an ARM template. In the ARM template set `enableNodePublicIP` parameter to false.

Next steps:

- [Use custom DNS](#)
- [Use a firewall](#)

NOTE

Support for compute instances without public IP addresses is currently available and in public preview for the following regions: France Central, East Asia, West Central US, South Central US, West US 2, East US, East US 2, North Europe, West Europe, Central US, North Central US, West US, Australia East, Japan East, Japan West.

Support for compute clusters without public IP addresses is currently available and in public preview for the following regions: France Central, East Asia, West Central US, South Central US, West US 2, East US, North Europe, East US 2, Central US, West Europe, North Central US, West US, Australia East, Japan East, Japan West.

Inbound traffic

When using Azure Machine Learning **compute instance** (with a public IP) or **compute cluster**, allow inbound traffic from Azure Batch management and Azure Machine Learning services. Compute instance with no public IP (preview) does not require this inbound communication. A Network Security Group allowing this traffic is dynamically created for you, however you may need to also create user-defined routes (UDR) if you have a firewall. When creating a UDR for this traffic, you can use either **IP Addresses** or **service tags** to route the traffic.

IMPORTANT

Using service tags with user-defined routes is now GA. For more information, see [Virtual Network routing](#).

TIP

While a compute instance without a public IP (a preview feature) does not need a UDR for this inbound traffic, you will still need these UDRs if you also use a compute cluster or a compute instance with a public IP.

- [IP Address routes](#)
- [Service tag routes](#)

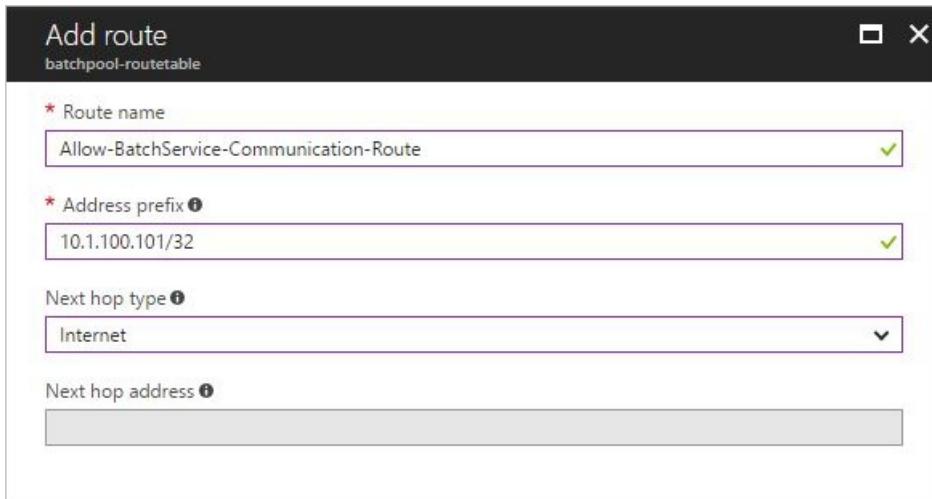
For the Azure Machine Learning service, you must add the IP address of both the **primary** and **secondary** regions. To find the secondary region, see the [Cross-region replication in Azure](#). For example, if your Azure Machine Learning service is in East US 2, the secondary region is Central US.

To get a list of IP addresses of the Batch service and Azure Machine Learning service, download the [Azure IP Ranges and Service Tags](#) and search the file for `BatchNodeManagement.<region>` and `AzureMachineLearning.<region>`, where `<region>` is your Azure region.

IMPORTANT

The IP addresses may change over time.

When creating the UDR, set the **Next hop type** to **Internet**. This means the inbound communication from Azure skips your firewall to access the load balancers with public IPs of Compute Instance and Compute Cluster. UDR is required because Compute Instance and Compute Cluster will get random public IPs at creation, and you cannot know the public IPs before creation to register them on your firewall to allow the inbound from Azure to specific IPs for Compute Instance and Compute Cluster. The following image shows an example IP address based UDR in the Azure portal:



For information on configuring UDR, see [Route network traffic with a routing table](#).

For more information on input and output traffic requirements for Azure Machine Learning, see [Use a workspace behind a firewall](#).

Next steps

This article is part of a series on securing an Azure Machine Learning workflow. See the other articles in this series:

- [Virtual network overview \(v1\)](#)
- [Secure the workspace resources](#)
- [Secure inference environment \(v1\)](#)
- [Enable studio functionality](#)
- [Use custom DNS](#)
- [Use a firewall](#)

Secure an Azure Machine Learning inferencing environment with virtual networks (v1)

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

APPLIES TO:  Azure CLI ml extension v1

In this article, you learn how to secure inferencing environments with a virtual network in Azure Machine Learning. This article is specific to the SDK/CLI v1 deployment workflow of deploying a model as a web service.

TIP

This article is part of a series on securing an Azure Machine Learning workflow. See the other articles in this series:

- [Virtual network overview](#)
- [Secure the workspace resources](#)
- [Secure the training environment](#)
- [Enable studio functionality](#)
- [Use custom DNS](#)
- [Use a firewall](#)

For a tutorial on creating a secure workspace, see [Tutorial: Create a secure workspace](#) or [Tutorial: Create a secure workspace using a template](#).

In this article you learn how to secure the following inferencing resources in a virtual network:

- Default Azure Kubernetes Service (AKS) cluster
- Private AKS cluster
- AKS cluster with private link

Prerequisites

- Read the [Network security overview](#) article to understand common virtual network scenarios and overall virtual network architecture.
- An existing virtual network and subnet to use with your compute resources.
- To deploy resources into a virtual network or subnet, your user account must have permissions to the following actions in Azure role-based access control (Azure RBAC):
 - "Microsoft.Network/virtualNetworks/join/action" on the virtual network resource.
 - "Microsoft.Network/virtualNetworks/subnet/join/action" on the subnet resource.

For more information on Azure RBAC with networking, see the [Networking built-in roles](#)

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Limitations

Azure Container Instances

When your Azure Machine Learning workspace is configured with a private endpoint, deploying to Azure Container Instances in a VNet is not supported. Instead, consider using a [Managed online endpoint with network isolation](#).

Azure Kubernetes Service

- If your AKS cluster is behind of a VNET, your workspace and its associated resources (storage, key vault, Azure Container Registry) must have private endpoints or service endpoints in the same VNET as AKS cluster's VNET. Please read tutorial [create a secure workspace](#) to add those private endpoints or service endpoints to your VNET.
- If your workspace has a **private endpoint**, the Azure Kubernetes Service cluster must be in the same Azure region as the workspace.
- Using a [public fully qualified domain name \(FQDN\)](#) with a private AKS cluster is **not supported** with Azure Machine learning.

Azure Kubernetes Service

IMPORTANT

To use an AKS cluster in a virtual network, first follow the prerequisites in [Configure advanced networking in Azure Kubernetes Service \(AKS\)](#).

To add AKS in a virtual network to your workspace, use the following steps:

1. Sign in to [Azure Machine Learning studio](#), and then select your subscription and workspace.
2. Select **Compute** on the left, **Inference clusters** from the center, and then select **+ New**.

☰

↶ Microsoft

+ New

HomeAspx

Author

Notebooks

Automated ML

Designer

Assets

Data

Jobs

Components

Pipelines

Environments

Models

Endpoints

Manage

+ Compute

Datastores

Home > Compute

Compute

Compute instances

Compute clusters

Inference clusters

Attached computes



Deploy your model to Azure Kubernetes Service for large scale inferencing

Create a new Azure Kubernetes Service (AKS) cluster or attach an existing AKS cluster to your workspace, then deploy your model as a REST endpoint. [Learn more](#)

+ New

[View Azure Machine Learning tutorials](#)

- From the **Create inference cluster** dialog, select **Create new** and the VM size to use for the cluster. Finally, select **Next**.

Create inference cluster

X

Virtual Machine

V

Select virtual machine

Select the virtual machine size you would like to use for your inference cluster.

Kubernetes Service

Create new Use existing

Location *

South Central US

[+ Add filter](#)

Standard_DS3

X

Showing 3 of 360 VM sizes | Current selection: Standard_DS3_v2

Total available quota: 100 cores [\(i\)](#)

Name ↑	Category	Available quota (i)
<input type="radio"/> Standard_DS3 4 cores, 14GB RAM, 28GB storage	General purpose	100 cores
<input checked="" type="radio"/> Standard_DS3_v2 4 cores, 14GB RAM, 28GB storage	General purpose	100 cores
<input type="radio"/> Standard_DS3_v2_Promo 4 cores, 14GB RAM, 28GB storage	General purpose	100 cores

Back

Next

Cancel

4. From the **Configure Settings** section, enter a **Compute name**, select the **Cluster Purpose**, **Number of nodes**, and then select **Advanced** to display the network settings. In the **Configure virtual network** area, set the following values:

- Set the **Virtual network** to use.

TIP

If your workspace uses a private endpoint to connect to the virtual network, the **Virtual network** selection field is greyed out.

- Set the **Subnet** to create the cluster in.
- In the **Kubernetes Service address range** field, enter the Kubernetes service address range. This address range uses a Classless Inter-Domain Routing (CIDR) notation IP range to define the IP addresses that are available for the cluster. It must not overlap with any subnet IP ranges (for example, 10.0.0.0/16).
- In the **Kubernetes DNS service IP address** field, enter the Kubernetes DNS service IP address. This IP address is assigned to the Kubernetes DNS service. It must be within the Kubernetes service address range (for example, 10.0.0.10).
- In the **Docker bridge address** field, enter the Docker bridge address. This IP address is assigned

to Docker Bridge. It must not be in any subnet IP ranges, or the Kubernetes service address range (for example, 172.18.0.1/16).

Create inference cluster X

Settings ▼

Configure Settings
Configure compute cluster settings for your selected virtual machine size.

Name	Category	Cores	Available quota	RAM	Storage
Standard_DS3_v2	General purpose	4	100 cores	14 GB	28 GB

Compute name * (i)

Cluster purpose
 Production Dev-test

Number of nodes * (i)

Network configuration (i)
 Basic Advanced

Configure virtual network (i)

Virtual network
docs-ml-vnet (docs-ml-rg)
(i) Your workspace is linked to a virtual network using a private endpoint connection. In order to communicate properly with the workspace, your compute resource must be provisioned in the same virtual network.

Subnet *

Kubernetes Service address range * (i)

Kubernetes DNS Service IP address * (i)

Docker Bridge address * (i)

Enable SSL configuration (i)

Back Create Download a template for automation Cancel

5. When you deploy a model as a web service to AKS, a scoring endpoint is created to handle inferencing requests. Make sure that the network security group (NSG) that controls the virtual network has an inbound security rule enabled for the IP address of the scoring endpoint if you want to call it from outside the virtual network.

To find the IP address of the scoring endpoint, look at the scoring URI for the deployed service. For information on viewing the scoring URI, see [Consume a model deployed as a web service](#).

IMPORTANT

Keep the default outbound rules for the NSG. For more information, see the default security rules in [Security groups](#).

- Inbound security rules

PRIORITY	NAME	PORT	PROTOCOL	SOURCE	DESTINATION	ACTION	...
1000	ScoringIP	80	TCP	Internet	137.135.117.175	Allow	...
1002	AML	Any	Any	AzureMachineLearning	Any	Allow	...
65000	AllowVnetInBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow	...
65001	AllowAzureLoadBalancerInBound	Any	Any	AzureLoadBalancer	Any	Allow	...
65500	DenyAllInBound	Any	Any	Any	Any	Deny	...

](./media/how-to-secure-inferencing-vnet/aks-vnet-inbound-nsg-scoring.png#lightbox)

IMPORTANT

The IP address shown in the image for the scoring endpoint will be different for your deployments. While the same IP is shared by all deployments to one AKS cluster, each AKS cluster will have a different IP address.

You can also use the Azure Machine Learning SDK to add Azure Kubernetes Service in a virtual network. If you already have an AKS cluster in a virtual network, attach it to the workspace as described in [How to deploy to AKS](#). The following code creates a new AKS instance in the `default` subnet of a virtual network named `mynetwork`:

APPLIES TO: [Python SDK azureml v1](#)

```
from azureml.core.compute import ComputeTarget, AksCompute

# Create the compute configuration and set virtual network information
config = AksCompute.provisioning_configuration(location="eastus2")
config.vnet_resourcegroup_name = "mygroup"
config.vnet_name = "mynetwork"
config.subnet_name = "default"
config.service_cidr = "10.0.0.0/16"
config.dns_service_ip = "10.0.0.10"
config.docker_bridge_cidr = "172.17.0.1/16"

# Create the compute target
aks_target = ComputeTarget.create(workspace=ws,
                                   name="myaks",
                                   provisioning_configuration=config)
```

When the creation process is completed, you can run inference, or model scoring, on an AKS cluster behind a virtual network. For more information, see [How to deploy to AKS](#).

For more information on using Role-Based Access Control with Kubernetes, see [Use Azure RBAC for Kubernetes authorization](#).

Network contributor role

IMPORTANT

If you create or attach an AKS cluster by providing a virtual network you previously created, you must grant the service principal (SP) or managed identity for your AKS cluster the *Network Contributor* role to the resource group that contains the virtual network.

To add the identity as network contributor, use the following steps:

1. To find the service principal or managed identity ID for AKS, use the following Azure CLI commands.

Replace `<aks-cluster-name>` with the name of the cluster. Replace `<resource-group-name>` with the name of the resource group that *contains the AKS cluster*.

```
az aks show -n <aks-cluster-name> --resource-group <resource-group-name> --query  
servicePrincipalProfile.clientId
```

If this command returns a value of `msi`, use the following command to identify the principal ID for the managed identity:

```
az aks show -n <aks-cluster-name> --resource-group <resource-group-name> --query identity.principalId
```

2. To find the ID of the resource group that contains your virtual network, use the following command.

Replace `<resource-group-name>` with the name of the resource group that *contains the virtual network*.

```
az group show -n <resource-group-name> --query id
```

3. To add the service principal or managed identity as a network contributor, use the following command.

Replace `<SP-or-managed-identity>` with the ID returned for the service principal or managed identity.

Replace `<resource-group-id>` with the ID returned for the resource group that contains the virtual network:

```
az role assignment create --assignee <SP-or-managed-identity> --role 'Network Contributor' --scope  
<resource-group-id>
```

For more information on using the internal load balancer with AKS, see [Use internal load balancer with Azure Kubernetes Service](#).

Secure VNet traffic

There are two approaches to isolate traffic to and from the AKS cluster to the virtual network:

- **Private AKS cluster:** This approach uses Azure Private Link to secure communications with the cluster for deployment/management operations.
- **Internal AKS load balancer:** This approach configures the endpoint for your deployments to AKS to use a private IP within the virtual network.

Private AKS cluster

By default, AKS clusters have a control plane, or API server, with public IP addresses. You can configure AKS to use a private control plane by creating a private AKS cluster. For more information, see [Create a private Azure Kubernetes Service cluster](#).

After you create the private AKS cluster, [attach the cluster to the virtual network](#) to use with Azure Machine Learning.

Internal AKS load balancer

By default, AKS deployments use a [public load balancer](#). In this section, you learn how to configure AKS to use an internal load balancer. An internal (or private) load balancer is used where only private IPs are allowed as frontend. Internal load balancers are used to load balance traffic inside a virtual network.

A private load balancer is enabled by configuring AKS to use an *internal load balancer*.

Enable private load balancer

IMPORTANT

You cannot enable private IP when creating the Azure Kubernetes Service cluster in Azure Machine Learning studio. You can create one with an internal load balancer when using the Python SDK or Azure CLI extension for machine learning.

The following examples demonstrate how to **create a new AKS cluster with a private IP/internal load balancer** using the SDK and CLI:

- [Python SDK](#)
- [Azure CLI](#)

APPLIES TO:  [Python SDK azureml v1](#)

```
import azureml.core
from azureml.core.compute import AksCompute, ComputeTarget

# Verify that cluster does not exist already
try:
    aks_target = AksCompute(workspace=ws, name=aks_cluster_name)
    print("Found existing aks cluster")

except:
    print("Creating new aks cluster")

# Subnet to use for AKS
subnet_name = "default"
# Create AKS configuration
prov_config=AksCompute.provisioning_configuration(load_balancer_type="InternalLoadBalancer")
# Set info for existing virtual network to create the cluster in
prov_config.vnet_resourcegroup_name = "myvnetresourcegroup"
prov_config.vnet_name = "myvnetname"
prov_config.service_cidr = "10.0.0.0/16"
prov_config.dns_service_ip = "10.0.0.10"
prov_config.subnet_name = subnet_name
prov_config.load_balancer_subnet = subnet_name
prov_config.docker_bridge_cidr = "172.17.0.1/16"

# Create compute target
aks_target = ComputeTarget.create(workspace = ws, name = "myaks", provisioning_configuration =
prov_config)
# Wait for the operation to complete
aks_target.wait_for_completion(show_output = True)
```

When attaching an existing cluster to your workspace, use the `load_balancer_type` and `load_balancer_subnet` parameters of [AksCompute.attach_configuration\(\)](#) to configure the load balancer.

For information on attaching a cluster, see [Attach an existing AKS cluster](#).

Limit outbound connectivity from the virtual network

If you don't want to use the default outbound rules and you do want to limit the outbound access of your virtual

network, you must allow access to Azure Container Registry. For example, make sure that your Network Security Groups (NSG) contains a rule that allows access to the `AzureContainerRegistry.RegionName` service tag where `{RegionName}` is the name of an Azure region.

Next steps

This article is part of a series on securing an Azure Machine Learning workflow. See the other articles in this series:

- [Virtual network overview](#)
- [Secure the workspace resources](#)
- [Secure the training environment](#)
- [Enable studio functionality](#)
- [Use custom DNS](#)
- [Use a firewall](#)

Use TLS to secure a web service through Azure Machine Learning

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article shows you how to secure a web service that's deployed through Azure Machine Learning.

You use [HTTPS](#) to restrict access to web services and secure the data that clients submit. HTTPS helps secure communications between a client and a web service by encrypting communications between the two.

Encryption uses [Transport Layer Security \(TLS\)](#). TLS is sometimes still referred to as *Secure Sockets Layer (SSL)*, which was the predecessor of TLS.

TIP

The Azure Machine Learning SDK uses the term "SSL" for properties that are related to secure communications. This doesn't mean that your web service doesn't use *TLS*. SSL is just a more commonly recognized term.

Specifically, web services deployed through Azure Machine Learning support TLS version 1.2 for AKS and ACI. For ACI deployments, if you are on older TLS version, we recommend re-deploying to get the latest TLS version.

TLS version 1.3 for Azure Machine Learning - AKS Inference is unsupported.

TLS and SSL both rely on *digital certificates*, which help with encryption and identity verification. For more information on how digital certificates work, see the Wikipedia topic [Public key infrastructure](#).

WARNING

If you don't use HTTPS for your web service, data that's sent to and from the service might be visible to others on the internet.

HTTPS also enables the client to verify the authenticity of the server that it's connecting to. This feature protects clients against [man-in-the-middle attacks](#).

This is the general process to secure a web service:

1. Get a domain name.
2. Get a digital certificate.
3. Deploy or update the web service with TLS enabled.
4. Update your DNS to point to the web service.

IMPORTANT

If you're deploying to Azure Kubernetes Service (AKS), you can purchase your own certificate or use a certificate that's provided by Microsoft. If you use a certificate from Microsoft, you don't need to get a domain name or TLS/SSL certificate. For more information, see the [Enable TLS and deploy](#) section of this article.

There are slight differences when you secure across [deployment targets](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Get a domain name

If you don't already own a domain name, purchase one from a *domain name registrar*. The process and price differ among registrars. The registrar provides tools to manage the domain name. You use these tools to map a fully qualified domain name (FQDN) (such as www.contoso.com) to the IP address that hosts your web service.

Get a TLS/SSL certificate

There are many ways to get an TLS/SSL certificate (digital certificate). The most common is to purchase one from a *certificate authority* (CA). Regardless of where you get the certificate, you need the following files:

- A **certificate**. The certificate must contain the full certificate chain, and it must be "PEM-encoded."
- A **key**. The key must also be PEM-encoded.

When you request a certificate, you must provide the FQDN of the address that you plan to use for the web service (for example, www.contoso.com). The address that's stamped into the certificate and the address that the clients use are compared to verify the identity of the web service. If those addresses don't match, the client gets an error message.

TIP

If the certificate authority can't provide the certificate and key as PEM-encoded files, you can use a utility such as [OpenSSL](#) to change the format.

WARNING

Use *self-signed* certificates only for development. Don't use them in production environments. Self-signed certificates can cause problems in your client applications. For more information, see the documentation for the network libraries that your client application uses.

Enable TLS and deploy

For AKS deployment, you can enable TLS termination when you [create or attach an AKS cluster](#) in AzureML workspace. At AKS model deployment time, you can disable TLS termination with deployment configuration object, otherwise all AKS model deployment by default will have TLS termination enabled at AKS cluster create or attach time.

For ACI deployment, you can enable TLS termination at model deployment time with deployment configuration object.

Deploy on Azure Kubernetes Service

NOTE

The information in this section also applies when you deploy a secure web service for the designer. If you aren't familiar with using the Python SDK, see [What is the Azure Machine Learning SDK for Python?](#).

When you [create or attach an AKS cluster](#) in AzureML workspace, you can enable TLS termination with `AksCompute.provisioning_configuration()` and `AksCompute.attach_configuration()` configuration objects. Both methods return a configuration object that has an `enable_ssl` method, and you can use `enable_ssl` method to enable TLS.

You can enable TLS either with Microsoft certificate or a custom certificate purchased from CA.

- When you use a certificate from Microsoft, you must use the `leaf_domain_label` parameter. This parameter generates the DNS name for the service. For example, a value of "contoso" creates a domain name of "contoso<six-random-characters>.<azureregion>.cloudapp.azure.com", where <azureregion> is the region that contains the service. Optionally, you can use the `overwrite_existing_domain` parameter to overwrite the existing `leaf_domain_label`. The following example demonstrates how to create a configuration that enables an TLS with Microsoft certificate:

```
from azureml.core.compute import AksCompute

# Config used to create a new AKS cluster and enable TLS
provisioning_config = AksCompute.provisioning_configuration()

# Leaf domain label generates a name using the formula
# "<leaf-domain-label>#####.<azure-region>.cloudapp.azure.com"
# where "#####" is a random series of characters
provisioning_config.enable_ssl(leaf_domain_label = "contoso")

# Config used to attach an existing AKS cluster to your workspace and enable TLS
attach_config = AksCompute.attach_configuration(resource_group = resource_group,
                                                cluster_name = cluster_name)

# Leaf domain label generates a name using the formula
# "<leaf-domain-label>#####.<azure-region>.cloudapp.azure.com"
# where "#####" is a random series of characters
attach_config.enable_ssl(leaf_domain_label = "contoso")
```

IMPORTANT

When you use a certificate from Microsoft, you don't need to purchase your own certificate or domain name.

- When you use a custom certificate that you purchased, you use the `ssl_cert_pem_file`, `ssl_key_pem_file`, and `ssl_cname` parameters. The following example demonstrates how to use .pem files to create a configuration that uses a TLS/SSL certificate that you purchased:

```

from azureml.core.compute import AksCompute

# Config used to create a new AKS cluster and enable TLS
provisioning_config = AksCompute.provisioning_configuration()
provisioning_config.enable_ssl(ssl_cert_pem_file="cert.pem",
                               ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")

# Config used to attach an existing AKS cluster to your workspace and enable SSL
attach_config = AksCompute.attach_configuration(resource_group = resource_group,
                                                cluster_name = cluster_name)
attach_config.enable_ssl(ssl_cert_pem_file="cert.pem",
                        ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")

```

For more information about `enable_ssl`, see [AksProvisioningConfiguration.enable_ssl\(\)](#) and [AksAttachConfiguration.enable_ssl\(\)](#).

Deploy on Azure Container Instances

When you deploy to Azure Container Instances, you provide values for TLS-related parameters, as the following code snippet shows:

```

from azureml.core.webservice import AciWebservice

aci_config = AciWebservice.deploy_configuration(
    ssl_enabled=True, ssl_cert_pem_file="cert.pem", ssl_key_pem_file="key.pem", ssl_cname="www.contoso.com")

```

For more information, see [AciWebservice.deploy_configuration\(\)](#).

Update your DNS

For either AKS deployment with custom certificate or ACI deployment, you must update your DNS record to point to the IP address of scoring endpoint.

IMPORTANT

When you use a certificate from Microsoft for AKS deployment, you don't need to manually update the DNS value for the cluster. The value should be set automatically.

You can follow following steps to update DNS record for your custom domain name:

1. Get scoring endpoint IP address from scoring endpoint URI, which is usually in the format of `http://104.214.29.152:80/api/v1/service/<service-name>/score`. In this example, the IP address is 104.214.29.152.
2. Use the tools from your domain name registrar to update the DNS record for your domain name. The record maps the FQDN (for example, www.contoso.com) to the IP address. The record must point to the IP address of scoring endpoint.

TIP

Microsoft does is not responsible for updating the DNS for your custom DNS name or certificate. You must update it with your domain name registrar.

3. After DNS record update, you can validate DNS resolution using `nslookup custom-domain-name` command. If DNS record is correctly updated, the custom domain name will point to the IP address of scoring endpoint.

There can be a delay of minutes or hours before clients can resolve the domain name, depending on the registrar and the "time to live" (TTL) that's configured for the domain name.

For more information on DNS resolution with Azure Machine Learning, see [How to use your workspace with a custom DNS server](#).

Update the TLS/SSL certificate

TLS/SSL certificates expire and must be renewed. Typically this happens every year. Use the information in the following sections to update and renew your certificate for models deployed to Azure Kubernetes Service:

Update a Microsoft generated certificate

If the certificate was originally generated by Microsoft (when using the `leaf_domain_label` to create the service), it will automatically renew when needed. If you want to manually renew it, use one of the following examples to update the certificate:

IMPORTANT

- If the existing certificate is still valid, use `renew=True` (SDK) or `--ssl-renew` (CLI) to force the configuration to renew it. For example, if the existing certificate is still valid for 10 days and you don't use `renew=True`, the certificate may not be renewed.
- When the service was originally deployed, the `leaf_domain_label` is used to create a DNS name using the pattern `<leaf-domain-label>#####.<azure-region>.cloudapp.azure.com`. To preserve the existing name (including the 6 digits originally generated), use the original `leaf_domain_label` value. Do not include the 6 digits that were generated.

Use the SDK

```
from azureml.core.compute import AksCompute
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute.aks import SslConfiguration

# Get the existing cluster
aks_target = AksCompute(ws, clustername)

# Update the existing certificate by referencing the leaf domain label
ssl_configuration = SslConfiguration(leaf_domain_label="myaks", overwrite_existing_domain=True, renew=True)
update_config = AksUpdateConfiguration(ssl_configuration)
aks_target.update(update_config)
```

Use the CLI

APPLIES TO:  [Azure CLI ml extension v1](#)

```
az ml computetarget update aks -g "myresourcegroup" -w "myresourceworkspace" -n "myaks" --ssl-leaf-domain-label "myaks" --ssl-overwrite-domain True --ssl-renew
```

For more information, see the following reference docs:

- [SslConfiguration](#)
- [AksUpdateConfiguration](#)

Update custom certificate

If the certificate was originally generated by a certificate authority, use the following steps:

1. Use the documentation provided by the certificate authority to renew the certificate. This process creates

new certificate files.

2. Use either the SDK or CLI to update the service with the new certificate:

Use the SDK

```
from azureml.core.compute import AksCompute
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute.aks import SslConfiguration

# Read the certificate file
def get_content(file_name):
    with open(file_name, 'r') as f:
        return f.read()

# Get the existing cluster
aks_target = AksCompute(ws, clustername)

# Update cluster with custom certificate
ssl_configuration = SslConfiguration(cname="myaks", cert=get_content('cert.pem'),
key=get_content('key.pem'))
update_config = AksUpdateConfiguration(ssl_configuration)
aks_target.update(update_config)
```

Use the CLI

APPLIES TO:  Azure CLI ml extension v1

```
az ml computetarget update aks -g "myresourcegroup" -w "myresourceworkspace" -n "myaks" --ssl-cname
"myaks"--ssl-cert-file "cert.pem" --ssl-key-file "key.pem"
```

For more information, see the following reference docs:

- [SslConfiguration](#)
- [AksUpdateConfiguration](#)

Disable TLS

To disable TLS for a model deployed to Azure Kubernetes Service, create an `SslConfiguration` with `status="Disabled"`, then perform an update:

```
from azureml.core.compute import AksCompute
from azureml.core.compute.aks import AksUpdateConfiguration
from azureml.core.compute.aks import SslConfiguration

# Get the existing cluster
aks_target = AksCompute(ws, clustername)

# Disable TLS
ssl_configuration = SslConfiguration(status="Disabled")
update_config = AksUpdateConfiguration(ssl_configuration)
aks_target.update(update_config)
```

Next steps

Learn how to:

- [Consume a machine learning model deployed as a web service](#)
- [Virtual network isolation and privacy overview](#)

- How to use your workspace with a custom DNS server

Connect to storage services on Azure with datastores

9/21/2022 • 13 minutes to read • [Edit Online](#)

APPLIES TO: Python SDK azureml v1

APPLIES TO: Azure CLI ml extension v1

In this article, learn how to connect to data storage services on Azure with Azure Machine Learning datastores and the [Azure Machine Learning Python SDK](#).

Datastores securely connect to your storage service on Azure without putting your authentication credentials and the integrity of your original data source at risk. They store connection information, like your subscription ID and token authorization in your [Key Vault](#) that's associated with the workspace, so you can securely access your storage without having to hard code them in your scripts. You can create datastores that connect to [these Azure storage solutions](#).

To understand where datastores fit in Azure Machine Learning's overall data access workflow, see the [Securely access data](#) article.

For a low code experience, see how to use the [Azure Machine Learning studio to create and register datastores](#).

TIP

This article assumes you want to connect to your storage service with credential-based authentication credentials, like a service principal or a shared access signature (SAS) token. Keep in mind, if credentials are registered with datastores, all users with workspace *Reader* role are able to retrieve these credentials. [Learn more about workspace Reader role..](#)

If this is a concern, learn how to [Connect to storage services with identity based access](#).

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#).
- An Azure storage account with a [supported storage type](#).
- The [Azure Machine Learning SDK for Python](#).
- An Azure Machine Learning workspace.

Either [create an Azure Machine Learning workspace](#) or use an existing one via the Python SDK.

Import the `Workspace` and `Datastore` class, and load your subscription information from the file `config.json` using the function `from_config()`. This looks for the JSON file in the current directory by default, but you can also specify a path parameter to point to the file using `from_config(path="your/file/path")`.

```
import azureml.core
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()
```

When you create a workspace, an Azure blob container and an Azure file share are automatically registered as datastores to the workspace. They're named `workspaceblobstore` and `workspacefilestore`, respectively. The `workspaceblobstore` is used to store workspace artifacts and your machine learning experiment logs. It's also set as the **default datastore** and can't be deleted from the workspace. The `workspacefilestore` is used to store notebooks and R scripts authorized via [compute instance](#).

NOTE

Azure Machine Learning designer will create a datastore named `azureml_globaldatasets` automatically when you open a sample in the designer homepage. This datastore only contains sample datasets. Please **do not** use this datastore for any confidential data access.

Supported data storage service types

Datastores currently support storing connection information to the storage services listed in the following matrix.

TIP

For **unsupported storage solutions** (those not listed in the table below), you may run into issues connecting and working with your data. We suggest you [move your data](#) to a supported Azure storage solution. Doing this may also help with additional scenarios, like saving data egress cost during ML experiments.

STORAGE TYPE	AUTHENTICATION TYPE	AZURE MACHINE LEARNING STUDIO	AZURE MACHINE LEARNING PYTHON SDK	AZURE MACHINE LEARNING CLI	AZURE MACHINE LEARNING REST API	VS CODE
Azure Blob Storage	Account key SAS token	✓	✓	✓	✓	✓
Azure File Share	Account key SAS token	✓	✓	✓	✓	✓
Azure Data Lake Storage Gen 1	Service principal	✓	✓	✓	✓	
Azure Data Lake Storage Gen 2	Service principal	✓	✓	✓	✓	
Azure SQL Database	SQL authentication Service principal	✓	✓	✓	✓	
Azure PostgreSQL	SQL authentication	✓	✓	✓	✓	
Azure Database for MySQL	SQL authentication		✓*	✓*	✓*	

STORAGE TYPE	AUTHENTICATION TYPE	AZURE MACHINE LEARNING STUDIO	AZURE MACHINE LEARNING PYTHON SDK	AZURE MACHINE LEARNING CLI	AZURE MACHINE LEARNING REST API	VS CODE
Databricks File System	No authentication		✓**	✓ **	✓**	

- MySQL is only supported for pipeline [DataTransferStep](#).
- Databricks is only supported for pipeline [DatabricksStep](#).

Storage guidance

We recommend creating a datastore for an [Azure Blob container](#). Both standard and premium storage are available for blobs. Although premium storage is more expensive, its faster throughput speeds might improve the speed of your training runs, particularly if you train against a large dataset. For information about the cost of storage accounts, see the [Azure pricing calculator](#).

[Azure Data Lake Storage Gen2](#) is built on top of Azure Blob storage and designed for enterprise big data analytics. A fundamental part of Data Lake Storage Gen2 is the addition of a [hierarchical namespace](#) to Blob storage. The hierarchical namespace organizes objects/files into a hierarchy of directories for efficient data access.

Storage access and permissions

To ensure you securely connect to your Azure storage service, Azure Machine Learning requires that you have permission to access the corresponding data storage container. This access depends on the authentication credentials used to register the datastore.

NOTE

This guidance also applies to [datastores created with identity-based data access](#).

Virtual network

Azure Machine Learning requires extra configuration steps to communicate with a storage account that is behind a firewall or within a virtual network. If your storage account is behind a firewall, you can [add your client's IP address to an allowlist](#) via the Azure portal.

Azure Machine Learning can receive requests from clients outside of the virtual network. To ensure that the entity requesting data from the service is safe and to enable data being displayed in your workspace, [use a private endpoint with your workspace](#).

For Python SDK users, to access your data via your training script on a compute target, the compute target needs to be inside the same virtual network and subnet of the storage. You can [use a compute cluster in the same virtual network](#) or [use a compute instance in the same virtual network](#).

For Azure Machine Learning studio users, several features rely on the ability to read data from a dataset, such as dataset previews, profiles, and automated machine learning. For these features to work with storage behind virtual networks, use a [workspace managed identity in the studio](#) to allow Azure Machine Learning to access the storage account from outside the virtual network.

NOTE

If your data storage is an Azure SQL Database behind a virtual network, be sure to set *Deny public access* to **No** via the [Azure portal](#) to allow Azure Machine Learning to access the storage account.

Access validation

WARNING

Cross tenant access to storage accounts is not supported. If cross tenant access is needed for your scenario, please reach out to the AzureML Data Support team alias at amldatasupport@microsoft.com for assistance with a custom code solution.

As part of the initial datastore creation and registration process, Azure Machine Learning automatically validates that the underlying storage service exists and the user provided principal (username, service principal, or SAS token) has access to the specified storage.

After datastore creation, this validation is only performed for methods that require access to the underlying storage container, **not** each time datastore objects are retrieved. For example, validation happens if you want to download files from your datastore; but if you just want to change your default datastore, then validation doesn't happen.

To authenticate your access to the underlying storage service, you can provide either your account key, shared access signatures (SAS) tokens, or service principal in the corresponding `register_azure_*()` method of the datastore type you want to create. The [storage type matrix](#) lists the supported authentication types that correspond to each datastore type.

You can find account key, SAS token, and service principal information on your [Azure portal](#).

- If you plan to use an account key or SAS token for authentication, select **Storage Accounts** on the left pane, and choose the storage account that you want to register.
 - The **Overview** page provides information such as the account name, container, and file share name.
 - For account keys, go to **Access keys** on the **Settings** pane.
 - For SAS tokens, go to **Shared access signatures** on the **Settings** pane.
- If you plan to use a service principal for authentication, go to your **App registrations** and select which app you want to use.
 - Its corresponding **Overview** page will contain required information like tenant ID and client ID.

IMPORTANT

If you need to change your access keys for an Azure Storage account (account key or SAS token), be sure to sync the new credentials with your workspace and the datastores connected to it. Learn how to [sync your updated credentials](#).

Permissions

For Azure blob container and Azure Data Lake Gen 2 storage, make sure your authentication credentials have **Storage Blob Data Reader** access. Learn more about [Storage Blob Data Reader](#). An account SAS token defaults to no permissions.

- For data **read access**, your authentication credentials must have a minimum of list and read permissions for containers and objects.
- For data **write access**, write and add permissions also are required.

Create and register datastores

When you register an Azure storage solution as a datastore, you automatically create and register that datastore to a specific workspace. Review the [storage access & permissions](#) section for guidance on virtual network scenarios, and where to find required authentication credentials.

Within this section are examples for how to create and register a datastore via the Python SDK for the following storage types. The parameters provided in these examples are the **required parameters** to create and register a datastore.

- [Azure blob container](#)
- [Azure file share](#)
- [Azure Data Lake Storage Generation 2](#)

To create datastores for other supported storage services, see the [reference documentation for the applicable `register_azure_*` methods](#).

If you prefer a low code experience, see [Connect to data with Azure Machine Learning studio](#).

IMPORTANT

If you unregister and re-register a datastore with the same name, and it fails, the Azure Key Vault for your workspace may not have soft-delete enabled. By default, soft-delete is enabled for the key vault instance created by your workspace, but it may not be enabled if you used an existing key vault or have a workspace created prior to October 2020. For information on how to enable soft-delete, see [Turn on Soft Delete for an existing key vault](#).

NOTE

Datastore name should only consist of lowercase letters, digits and underscores.

Azure blob container

To register an Azure blob container as a datastore, use `register_azure_blob_container()`.

The following code creates and registers the `blob_datastore_name` datastore to the `ws` workspace. This datastore accesses the `my-container-name` blob container on the `my-account-name` storage account, by using the provided account access key. Review the [storage access & permissions](#) section for guidance on virtual network scenarios, and where to find required authentication credentials.

```
blob_datastore_name='azblobsdk' # Name of the datastore to workspace
container_name=os.getenv("BLOB_CONTAINER", "<my-container-name>") # Name of Azure blob container
account_name=os.getenv("BLOB_ACCOUNTNAME", "<my-account-name>") # Storage account name
account_key=os.getenv("BLOB_ACCOUNT_KEY", "<my-account-key>") # Storage account access key

blob_datastore = Datastore.register_azure_blob_container(workspace=ws,
                                                       datastore_name=blob_datastore_name,
                                                       container_name=container_name,
                                                       account_name=account_name,
                                                       account_key=account_key)
```

Azure file share

To register an Azure file share as a datastore, use `register_azure_file_share()`.

The following code creates and registers the `file_datastore_name` datastore to the `ws` workspace. This datastore accesses the `my-fileshare-name` file share on the `my-account-name` storage account, by using the provided account access key. Review the [storage access & permissions](#) section for guidance on virtual network scenarios, and where to find required authentication credentials.

```

file_datastore_name='azfilesharesdk' # Name of the datastore to workspace
file_share_name=os.getenv("FILE_SHARE_CONTAINER", "<my-fileshare-name>") # Name of Azure file share
container
account_name=os.getenv("FILE_SHARE_ACCOUNTNAME", "<my-account-name>") # Storage account name
account_key=os.getenv("FILE_SHARE_ACCOUNT_KEY", "<my-account-key>") # Storage account access key

file_datastore = Datastore.register_azure_file_share(workspace=ws,
                                                    datastore_name=file_datastore_name,
                                                    file_share_name=file_share_name,
                                                    account_name=account_name,
                                                    account_key=account_key)

```

Azure Data Lake Storage Generation 2

For an Azure Data Lake Storage Generation 2 (ADLS Gen 2) datastore, use [register_azure_data_lake_gen2\(\)](#) to register a credential datastore connected to an Azure DataLake Gen 2 storage with [service principal permissions](#).

In order to utilize your service principal, you need to [register your application](#) and grant the service principal data access via either Azure role-based access control (Azure RBAC) or access control lists (ACL). Learn more about [access control set up for ADLS Gen 2](#).

The following code creates and registers the `adlsgen2_datastore_name` datastore to the `ws` workspace. This datastore accesses the file system `test` in the `account_name` storage account, by using the provided service principal credentials. Review the [storage access & permissions](#) section for guidance on virtual network scenarios, and where to find required authentication credentials.

```

adlsgen2_datastore_name = 'adlsgen2datastore'

subscription_id=os.getenv("ADL_SUBSCRIPTION", "<my_subscription_id>") # subscription id of ADLS account
resource_group=os.getenv("ADL_RESOURCE_GROUP", "<my_resource_group>") # resource group of ADLS account

account_name=os.getenv("ADLSGEN2_ACCOUNTNAME", "<my_account_name>") # ADLS Gen2 account name
tenant_id=os.getenv("ADLSGEN2_TENANT", "<my_tenant_id>") # tenant id of service principal
client_id=os.getenv("ADLSGEN2_CLIENTID", "<my_client_id>") # client id of service principal
client_secret=os.getenv("ADLSGEN2_CLIENT_SECRET", "<my_client_secret>") # the secret of service principal

adlsgen2_datastore = Datastore.register_azure_data_lake_gen2(workspace=ws,
                                                               datastore_name=adlsgen2_datastore_name,
                                                               account_name=account_name, # ADLS Gen2 account
                                                               name
                                                               filesystem='test', # ADLS Gen2 filesystem
                                                               tenant_id=tenant_id, # tenant id of service
                                                               principal
                                                               client_id=client_id, # client id of service
                                                               principal
                                                               client_secret=client_secret) # the secret of
                                                               service principal

```

Create datastores with other Azure tools

In addition to creating datastores with the Python SDK and the studio, you can also use Azure Resource Manager templates or the Azure Machine Learning VS Code extension.

Azure Resource Manager

There are several templates at <https://github.com/Azure/azure-quickstart-templates/tree/master/quickstarts/microsoft.machinelearningservices> that can be used to create datastores.

For information on using these templates, see [Use an Azure Resource Manager template to create a workspace for Azure Machine Learning](#).

VS Code extension

If you prefer to create and manage datastores using the Azure Machine Learning VS Code extension, visit the [VS Code resource management how-to guide](#) to learn more.

Use data in your datastores

After you create a datastore, [create an Azure Machine Learning dataset](#) to interact with your data. Datasets package your data into a lazily evaluated consumable object for machine learning tasks, like training.

With datasets, you can [download or mount](#) files of any format from Azure storage services for model training on a compute target. [Learn more about how to train ML models with datasets.](#)

Get datastores from your workspace

To get a specific datastore registered in the current workspace, use the `get()` static method on the `Datastore` class:

```
# Get a named datastore from the current workspace
datastore = Datastore.get(ws, datastore_name='your datastore name')
```

To get the list of datastores registered with a given workspace, you can use the `datastores` property on a workspace object:

```
# List all datastores registered in the current workspace
datastores = ws.datastores
for name, datastore in datastores.items():
    print(name, datastore.datastore_type)
```

To get the workspace's default datastore, use this line:

```
datastore = ws.get_default_datastore()
```

You can also change the default datastore with the following code. This ability is only supported via the SDK.

```
ws.set_default_datastore(new_default_datastore)
```

Access data during scoring

Azure Machine Learning provides several ways to use your models for scoring. Some of these methods don't provide access to datastores. Use the following table to understand which methods allow you to access datastores during scoring:

METHOD	DATASTORE ACCESS	DESCRIPTION
Batch prediction	✓	Make predictions on large quantities of data asynchronously.
Web service		Deploy models as a web service.

For situations where the SDK doesn't provide access to datastores, you might be able to create custom code by using the relevant Azure SDK to access the data. For example, the [Azure Storage SDK for Python](#) is a client library that you can use to access data stored in blobs or files.

Move data to supported Azure storage solutions

Azure Machine Learning supports accessing data from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL. If you're using unsupported storage, we recommend that you move your data to supported Azure storage solutions by using [Azure Data Factory and these steps](#). Moving data to supported storage can help you save data egress costs during machine learning experiments.

Azure Data Factory provides efficient and resilient data transfer with more than 80 prebuilt connectors at no extra cost. These connectors include Azure data services, on-premises data sources, Amazon S3 and Redshift, and Google BigQuery.

Next steps

- [Create an Azure machine learning dataset](#)
- [Train a model](#)
- [Deploy a model](#)

Connect to storage by using identity-based data access with SDK v1

9/21/2022 • 10 minutes to read • [Edit Online](#)

In this article, you learn how to connect to storage services on Azure by using identity-based data access and Azure Machine Learning datastores via the [Azure Machine Learning SDK for Python](#).

Typically, datastores use **credential-based authentication** to confirm you have permission to access the storage service. They keep connection information, like your subscription ID and token authorization, in the **key vault** that's associated with the workspace. When you create a datastore that uses **identity-based data access**, your Azure account ([Azure Active Directory token](#)) is used to confirm you have permission to access the storage service. In the **identity-based data access** scenario, no authentication credentials are saved. Only the storage account information is stored in the datastore.

To create datastores with **identity-based** data access via the Azure Machine Learning studio UI, see [Connect to data with the Azure Machine Learning studio](#).

To create datastores that use **credential-based** authentication, like access keys or service principals, see [Connect to storage services on Azure](#).

Identity-based data access in Azure Machine Learning

There are two scenarios in which you can apply identity-based data access in Azure Machine Learning. These scenarios are a good fit for identity-based access when you're working with confidential data and need more granular data access management:

WARNING

Identity-based data access is not supported for [automated ML experiments](#).

- Accessing storage services
- Training machine learning models with private data

Accessing storage services

You can connect to storage services via identity-based data access with Azure Machine Learning datastores or [Azure Machine Learning datasets](#).

Your authentication credentials are usually kept in a datastore, which is used to ensure you have permission to access the storage service. When these credentials are registered via datastores, any user with the workspace Reader role can retrieve them. That scale of access can be a security concern for some organizations. [Learn more about the workspace Reader role](#).

When you use identity-based data access, Azure Machine Learning prompts you for your Azure Active Directory token for data access authentication instead of keeping your credentials in the datastore. That approach allows for data access management at the storage level and keeps credentials confidential.

The same behavior applies when you:

- [Create a dataset directly from storage URLs](#).
- Work with data interactively via a Jupyter Notebook on your local computer or [compute instance](#).

NOTE

Credentials stored via credential-based authentication include subscription IDs, shared access signature (SAS) tokens, and storage access key and service principal information, like client IDs and tenant IDs.

Model training on private data

Certain machine learning scenarios involve training models with private data. In such cases, data scientists need to run training workflows without being exposed to the confidential input data. In this scenario, a [managed identity](#) of the training compute is used for data access authentication. This approach allows storage admins to grant Storage Blob Data Reader access to the managed identity that the training compute uses to run the training job. The individual data scientists don't need to be granted access. For more information, see [Set up managed identity on a compute cluster](#).

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#).
- An Azure storage account with a supported storage type. These storage types are supported:
 - [Azure Blob Storage](#)
 - [Azure Data Lake Storage Gen1](#)
 - [Azure Data Lake Storage Gen2](#)
 - [Azure SQL Database](#)
- The [Azure Machine Learning SDK for Python](#).
- An Azure Machine Learning workspace.

Either [create an Azure Machine Learning workspace](#) or use an [existing one via the Python SDK](#).

Create and register datastores

When you register a storage service on Azure as a datastore, you automatically create and register that datastore to a specific workspace. See [Storage access permissions](#) for guidance on required permission types. You also have the option to manually create the storage you want to connect to without any special permissions, and you just need the name.

See [Work with virtual networks](#) for details on how to connect to data storage behind virtual networks.

In the following code, notice the absence of authentication parameters like `sas_token`, `account_key`, `subscription_id`, and the service principal `client_id`. This omission indicates that Azure Machine Learning will use identity-based data access for authentication. Creation of datastores typically happens interactively in a notebook or via the studio. So your Azure Active Directory token is used for data access authentication.

NOTE

Datastore names should consist only of lowercase letters, numbers, and underscores.

Azure blob container

To register an Azure blob container as a datastore, use `register_azure_blob_container()`.

The following code creates the `credentialless_blob` datastore, registers it to the `ws` workspace, and assigns it to the `blob_datastore` variable. This datastore accesses the `my_container_name` blob container on the `my-account-name` storage account.

```
# Create blob datastore without credentials.
blob_datastore = Datastore.register_azure_blob_container(workspace=ws,
                                                       datastore_name='credentialless_blob',
                                                       container_name='my_container_name',
                                                       account_name='my_account_name')
```

Azure Data Lake Storage Gen1

Use [register_azure_data_lake\(\)](#) to register a datastore that connects to Azure Data Lake Storage Gen1.

The following code creates the `credentialless_adls1` datastore, registers it to the `workspace` workspace, and assigns it to the `adls_dstore` variable. This datastore accesses the `adls_storage` Azure Data Lake Storage account.

```
# Create Azure Data Lake Storage Gen1 datastore without credentials.
adls_dstore = Datastore.register_azure_data_lake(workspace = workspace,
                                                 datastore_name='credentialless_adls1',
                                                 store_name='adls_storage')
```

Azure Data Lake Storage Gen2

Use [register_azure_data_lake_gen2\(\)](#) to register a datastore that connects to Azure Data Lake Storage Gen2.

The following code creates the `credentialless_adls2` datastore, registers it to the `ws` workspace, and assigns it to the `adls2_dstore` variable. This datastore accesses the file system `tabular` in the `myadls2` storage account.

```
# Create Azure Data Lake Storage Gen2 datastore without credentials.
adls2_dstore = Datastore.register_azure_data_lake_gen2(workspace=ws,
                                                       datastore_name='credentialless_adls2',
                                                       filesystem='tabular',
                                                       account_name='myadls2')
```

Azure SQL database

For an Azure SQL database, use [register_azure_sql_database\(\)](#) to register a datastore that connects to an Azure SQL database storage.

The following code creates and registers the `credentialless_sqldb` datastore to the `ws` workspace and assigns it to the variable, `sqldb_dstore`. This datastore accesses the database `mydb` in the `myserver` SQL DB server.

```
# Create a sqldatabase datastore without credentials

sqldb_dstore = Datastore.register_azure_sql_database(workspace=ws,
                                                       datastore_name='credentialless_sqldb',
                                                       server_name='myserver',
                                                       database_name='mydb')
```

Storage access permissions

To help ensure that you securely connect to your storage service on Azure, Azure Machine Learning requires that you have permission to access the corresponding data storage.

WARNING

Cross tenant access to storage accounts is not supported. If cross tenant access is needed for your scenario, please reach out to the AzureML Data Support team alias at amldatasupport@microsoft.com for assistance with a custom code solution.

Identity-based data access supports connections to **only** the following storage services.

- Azure Blob Storage
- Azure Data Lake Storage Gen1
- Azure Data Lake Storage Gen2
- Azure SQL Database

To access these storage services, you must have at least [Storage Blob Data Reader](#) access to the storage account. Only storage account owners can [change your access level via the Azure portal](#).

If you prefer to not use your user identity (Azure Active Directory), you also have the option to grant a workspace managed-system identity (MSI) permission to create the datastore. To do so, you must have Owner permissions to the storage account and add the `grant_workspace_access= True` parameter to your data register method.

If you're training a model on a remote compute target and want to access the data for training, the compute identity must be granted at least the Storage Blob Data Reader role from the storage service. Learn how to [set up managed identity on a compute cluster](#).

Work with virtual networks

By default, Azure Machine Learning can't communicate with a storage account that's behind a firewall or in a virtual network.

You can configure storage accounts to allow access only from within specific virtual networks. This configuration requires additional steps to ensure data isn't leaked outside of the network. This behavior is the same for credential-based data access. For more information, see [How to configure virtual network scenarios](#).

If your storage account has virtual network settings, that dictates what identity type and permissions access is needed. For example for data preview and data profile, the virtual network settings determine what type of identity is used to authenticate data access.

- In scenarios where only certain IPs and subnets are allowed to access the storage, then Azure Machine Learning uses the workspace MSI to accomplish data previews and profiles.
- If your storage is ADLS Gen 2 or Blob and has virtual network settings, customers can use either user identity or workspace MSI depending on the datastore settings defined during creation.
- If the virtual network setting is "Allow Azure services on the trusted services list to access this storage account", then Workspace MSI is used.

Use data in storage

We recommend that you use [Azure Machine Learning datasets](#) when you interact with your data in storage with Azure Machine Learning.

IMPORTANT

Datasets using identity-based data access are not supported for [automated ML experiments](#).

Datasets package your data into a lazily evaluated consumable object for machine learning tasks like training. Also, with datasets you can [download or mount](#) files of any format from Azure storage services like Azure Blob Storage and Azure Data Lake Storage to a compute target.

To create a dataset, you can reference paths from datastores that also use identity-based data access .

- If you're underlying storage account type is Blob or ADLS Gen 2, your user identity needs Blob Reader role.
- If your underlying storage is ADLS Gen 1, permissions need can be set via the storage's Access Control List (ACL).

In the following example, `blob_datastore` already exists and uses identity-based data access.

```
blob_dataset = Dataset.Tabular.from_delimited_files(blob_datastore, 'test.csv')
```

Another option is to skip datastore creation and create datasets directly from storage URLs. This functionality currently supports only Azure blobs and Azure Data Lake Storage Gen1 and Gen2. For creation based on storage URL, only the user identity is needed to authenticate.

```
blob_dset = Dataset.File.from_files('https://myblob.blob.core.windows.net/may/keras-mnist-fashion/')
```

When you submit a training job that consumes a dataset created with identity-based data access, the managed identity of the training compute is used for data access authentication. Your Azure Active Directory token isn't used. For this scenario, ensure that the managed identity of the compute is granted at least the Storage Blob Data Reader role from the storage service. For more information, see [Set up managed identity on compute clusters](#).

Access data for training jobs on compute clusters (preview)

APPLIES TO:  [Azure CLI ml extension v2 \(current\)](#)

When training on [Azure Machine Learning compute clusters](#), you can authenticate to storage with your Azure Active Directory token.

This authentication mode allows you to:

- Set up fine-grained permissions, where different workspace users can have access to different storage accounts or folders within storage accounts.
- Audit storage access because the storage logs show which identities were used to access data.

IMPORTANT

This functionality has the following limitations

- Feature is only supported for experiments submitted via the [Azure Machine Learning CLI](#)
- Only CommandJobs, and PipelineJobs with CommandSteps and AutoMLSteps are supported
- User identity and compute managed identity cannot be used for authentication within same job.

WARNING

This feature is **public preview** and is not secure for production workloads. Ensure that only trusted users have permissions to access your workspace and storage accounts.

Preview features are provided without a service-level agreement, and are not recommended for production workloads. Certain features might not be supported or might have constrained capabilities.

For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

The following steps outline how to set up identity-based data access for training jobs on compute clusters.

1. Grant the user identity access to storage resources. For example, grant StorageBlobReader access to the specific storage account you want to use or grant ACL-based permission to specific folders or files in Azure Data Lake Gen 2 storage.
2. Create an Azure Machine Learning datastore without cached credentials for the storage account. If a datastore has cached credentials, such as storage account key, those credentials are used instead of user identity.
3. Submit a training job with property **identity** set to **type: user_identity**, as shown in following job specification. During the training job, the authentication to storage happens via the identity of the user that submits the job.

NOTE

If the **identity** property is left unspecified and datastore does not have cached credentials, then compute managed identity becomes the fallback option.

```
command: |
    echo "--census-csv: ${{inputs.census_csv}}"
    python hello-census.py --census-csv ${{inputs.census_csv}}
code: src
inputs:
  census_csv:
    type: uri_file
    path: azureml://datastores/mydata/paths/census.csv
environment: azureml:AzureML-sklearn-1.0-ubuntu20.04-py38-cpu@latest
compute: azureml:cpu-cluster
identity:
  type: user_identity
```

Next steps

- [Create an Azure Machine Learning dataset](#)
- [Train with datasets](#)
- [Create a datastore with key-based data access](#)

Create Azure Machine Learning datasets

9/21/2022 • 13 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this article, you learn how to create Azure Machine Learning datasets to access data for your local or remote experiments with the Azure Machine Learning Python SDK. To understand where datasets fit in Azure Machine Learning's overall data access workflow, see the [Securely access data](#) article.

By creating a dataset, you create a reference to the data source location, along with a copy of its metadata. Because the data remains in its existing location, you incur no extra storage cost, and don't risk the integrity of your data sources. Also datasets are lazily evaluated, which aids in workflow performance speeds. You can create datasets from datastores, public URLs, and [Azure Open Datasets](#).

For a low-code experience, [Create Azure Machine Learning datasets with the Azure Machine Learning studio](#).

With Azure Machine Learning datasets, you can:

- Keep a single copy of data in your storage, referenced by datasets.
- Seamlessly access data during model training without worrying about connection strings or data paths.
[Learn more about how to train with datasets](#).
- Share data and collaborate with other users.

Prerequisites

To create and work with datasets, you need:

- An Azure subscription. If you don't have one, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#).
- An [Azure Machine Learning workspace](#).
- The [Azure Machine Learning SDK for Python](#) installed, which includes the `azureml-datasets` package.
 - Create an [Azure Machine Learning compute instance](#), which is a fully configured and managed development environment that includes integrated notebooks and the SDK already installed.

OR

- Work on your own Jupyter notebook and [install the SDK yourself](#).

NOTE

Some dataset classes have dependencies on the `azureml-dataprep` package, which is only compatible with 64-bit Python. If you are developing on [Linux](#), these classes rely on .NET Core 2.1, and are only supported on specific distributions. For more information on the supported distros, see the .NET Core 2.1 column in the [Install .NET on Linux](#) article.

IMPORTANT

While the package may work on older versions of Linux distros, we do not recommend using a distro that is out of mainstream support. Distros that are out of mainstream support may have security vulnerabilities, as they do not receive the latest updates. We recommend using the latest supported version of your distro that is compatible with .

Compute size guidance

When creating a dataset, review your compute processing power and the size of your data in memory. The size of your data in storage is not the same as the size of data in a dataframe. For example, data in CSV files can expand up to 10x in a dataframe, so a 1 GB CSV file can become 10 GB in a dataframe.

If your data is compressed, it can expand further; 20 GB of relatively sparse data stored in compressed parquet format can expand to ~800 GB in memory. Since Parquet files store data in a columnar format, if you only need half of the columns, then you only need to load ~400 GB in memory.

[Learn more about optimizing data processing in Azure Machine Learning.](#)

Dataset types

There are two dataset types, based on how users consume them in training; FileDatasets and TabularDatasets. Both types can be used in Azure Machine Learning training workflows involving, estimators, AutoML, hyperDrive and pipelines.

FileDataset

A [FileDataset](#) references single or multiple files in your datastores or public URLs. If your data is already cleansed, and ready to use in training experiments, you can [download or mount](#) the files to your compute as a FileDataset object.

We recommend FileDatasets for your machine learning workflows, since the source files can be in any format, which enables a wider range of machine learning scenarios, including deep learning.

Create a FileDataset with the [Python SDK](#) or the [Azure Machine Learning studio](#).

TabularDataset

A [TabularDataset](#) represents data in a tabular format by parsing the provided file or list of files. This provides you with the ability to materialize the data into a pandas or Spark DataFrame so you can work with familiar data preparation and training libraries without having to leave your notebook. You can create a [TabularDataset](#) object from .csv, .tsv, [.parquet](#), [.jsonl](#) files, and from [SQL query results](#).

With TabularDatasets, you can specify a time stamp from a column in the data or from wherever the path pattern data is stored to enable a time series trait. This specification allows for easy and efficient filtering by time. For an example, see [Tabular time series-related API demo with NOAA weather data](#).

Create a TabularDataset with the [Python SDK](#) or [Azure Machine Learning studio](#).

NOTE

Automated ML workflows generated via the Azure Machine Learning studio currently only support TabularDatasets.

NOTE

For TabularDatasets generating from [SQL query results](#), T-SQL (e.g. 'WITH' sub query) or duplicate column name is not supported. Complex queries like T-SQL can cause performance issues. Duplicate column names in a dataset can cause ambiguity issues.

Access datasets in a virtual network

If your workspace is in a virtual network, you must configure the dataset to skip validation. For more information on how to use datastores and datasets in a virtual network, see [Secure a workspace and associated resources](#).

Create datasets from datastores

For the data to be accessible by Azure Machine Learning, datasets must be created from paths in [Azure Machine Learning datastores](#) or web URLs.

TIP

You can create datasets directly from storage urls with identity-based data access. Learn more at [Connect to storage with identity-based data access](#).

To create datasets from a datastore with the Python SDK:

1. Verify that you have `contributor` or `owner` access to the underlying storage service of your registered Azure Machine Learning datastore. [Check your storage account permissions in the Azure portal](#).
2. Create the dataset by referencing paths in the datastore. You can create a dataset from multiple paths in multiple datastores. There is no hard limit on the number of files or data size that you can create a dataset from.

NOTE

For each data path, a few requests will be sent to the storage service to check whether it points to a file or a folder. This overhead may lead to degraded performance or failure. A dataset referencing one folder with 1000 files inside is considered referencing one data path. We recommend creating dataset referencing less than 100 paths in datastores for optimal performance.

Create a FileDataset

Use the `from_files()` method on the `FileDatasetFactory` class to load files in any format and to create an unregistered FileDataset.

If your storage is behind a virtual network or firewall, set the parameter `validate=False` in your `from_files()` method. This bypasses the initial validation step, and ensures that you can create your dataset from these secure files. Learn more about how to [use datastores and datasets in a virtual network](#).

```
from azureml.core import Workspace, Datastore, Dataset

# create a FileDataset pointing to files in 'animals' folder and its subfolders recursively
datastore_paths = [(datastore, 'animals')]
animal_ds = Dataset.File.from_files(path=datastore_paths)

# create a FileDataset from image and label files behind public web urls
web_paths = ['https://azuredataportage.blob.core.windows.net/mnist/train-images-idx3-ubyte.gz',
            'https://azuredataportage.blob.core.windows.net/mnist/train-labels-idx1-ubyte.gz']
mnist_ds = Dataset.File.from_files(path=web_paths)
```

If you want to upload all the files from a local directory, create a FileDataset in a single method with `upload_directory()`. This method uploads data to your underlying storage, and as a result incur storage costs.

```
from azureml.core import Workspace, Datastore, Dataset
from azureml.data.datapath import DataPath

ws = Workspace.from_config()
datastore = Datastore.get(ws, '<name of your datastore>')
ds = Dataset.File.upload_directory(src_dir='<path to your data>',
                                    target=DataPath(datastore, '<path on the datastore>'),
                                    show_progress=True)
```

To reuse and share datasets across experiment in your workspace, [register your dataset](#).

Create a TabularDataset

Use the `from_delimited_files()` method on the `TabularDatasetFactory` class to read files in .csv or .tsv format, and to create an unregistered TabularDataset. To read in files from .parquet format, use the `from_parquet_files()` method. If you're reading from multiple files, results will be aggregated into one tabular representation.

See the [TabularDatasetFactory reference documentation](#) for information about supported file formats, as well as syntax and design patterns such as [multiline support](#).

If your storage is behind a virtual network or firewall, set the parameter `validate=False` in your `from_delimited_files()` method. This bypasses the initial validation step, and ensures that you can create your dataset from these secure files. Learn more about how to use [datastores and datasets in a virtual network](#).

The following code gets the existing workspace and the desired datastore by name. And then passes the datastore and file locations to the `path` parameter to create a new TabularDataset, `weather_ds`.

```
from azureml.core import Workspace, Datastore, Dataset

datastore_name = 'your datastore name'

# get existing workspace
workspace = Workspace.from_config()

# retrieve an existing datastore in the workspace by name
datastore = Datastore.get(workspace, datastore_name)

# create a TabularDataset from 3 file paths in datastore
datastore_paths = [(datastore, 'weather/2018/11.csv'),
                   (datastore, 'weather/2018/12.csv'),
                   (datastore, 'weather/2019/*.csv')]

weather_ds = Dataset.Tabular.from_delimited_files(path=datastore_paths)
```

Set data schema

By default, when you create a TabularDataset, column data types are inferred automatically. If the inferred types don't match your expectations, you can update your dataset schema by specifying column types with the following code. The parameter `infer_column_type` is only applicable for datasets created from delimited files.

[Learn more about supported data types](#).

```

from azureml.core import Dataset
from azureml.data.dataset_factory import DataType

# create a TabularDataset from a delimited file behind a public web url and convert column "Survived" to
# boolean
web_path ='https://dprepdata.blob.core.windows.net/demo/Titanic.csv'
titanic_ds = Dataset.Tabular.from_delimited_files(path=web_path, set_column_types={'Survived':
(DataType.to_bool())})

# preview the first 3 rows of titanic_ds
titanic_ds.take(3).to_pandas_dataframe()

```

(INDEX)	PASS ENGE RID	SURV IVED	PCLA SS	NAM E	SEX	AGE	SIBSP	PARC H	TICK ET	FARE	CABI N	EMB ARKE D
0	1	False	3	Brau nd, Mr. Owe n Harri s	male	22.0	1	0	A/5 2117 1	7.25 00		S
1	2	True	1	Cumi ngs, Mrs. John Bradl ey (Florence Briggs Th...	femal e	38.0	1	0	PC 1759 9	71.2 833	C85	C
2	3	True	3	Heik kinen , Miss. Laina	femal e	26.0	0	0	STON /O2. 3101 282	7.92 50		S

To reuse and share datasets across experiments in your workspace, [register your dataset](#).

Wrangle data

After you create and [register](#) your dataset, you can load it into your notebook for data wrangling and [exploration](#) prior to model training.

If you don't need to do any data wrangling or exploration, see how to consume datasets in your training scripts for submitting ML experiments in [Train with datasets](#).

Filter datasets (preview)

Filtering capabilities depends on the type of dataset you have.

IMPORTANT

Filtering datasets with the preview method, `filter()` is an [experimental](#) preview feature, and may change at any time.

For **TabularDatasets**, you can keep or remove columns with the [keep_columns\(\)](#) and [drop_columns\(\)](#) methods.

To filter out rows by a specific column value in a TabularDataset, use the [filter\(\)](#) method (preview).

The following examples return an unregistered dataset based on the specified expressions.

```
# TabularDataset that only contains records where the age column value is greater than 15
tabular_dataset = tabular_dataset.filter(tabular_dataset['age'] > 15)

# TabularDataset that contains records where the name column value contains 'Bri' and the age column value
# is greater than 15
tabular_dataset = tabular_dataset.filter((tabular_dataset['name'].contains('Bri')) & (tabular_dataset['age'] > 15))
```

In **FileDatasets**, each row corresponds to a path of a file, so filtering by column value is not helpful. But, you can [filter\(\)](#) out rows by metadata like, CreationTime, Size etc.

The following examples return an unregistered dataset based on the specified expressions.

```
# FileDataset that only contains files where Size is less than 100000
file_dataset = file_dataset.filter(file_dataset.file_metadata['Size'] < 100000)

# FileDataset that only contains files that were either created prior to Jan 1, 2020 or where
file_dataset = file_dataset.filter((file_dataset.file_metadata['CreatedTime'] < datetime(2020,1,1)) |
(file_dataset.file_metadata['CanSeek'] == False))
```

Labeled datasets created from [image labeling projects](#) are a special case. These datasets are a type of TabularDataset made up of image files. For these types of datasets, you can [filter\(\)](#) images by metadata, and by column values like `label` and `image_details`.

```
# Dataset that only contains records where the label column value is dog
labeled_dataset = labeled_dataset.filter(labeled_dataset['label'] == 'dog')

# Dataset that only contains records where the label and isCrowd columns are True and where the file size is
# larger than 100000
labeled_dataset = labeled_dataset.filter((labeled_dataset['label']['isCrowd'] == True) &
(labeled_dataset.file_metadata['Size'] > 100000))
```

Partition data

You can partition a dataset by including the `partitions_format` parameter when creating a TabularDataset or FileDataset.

When you partition a dataset, the partition information of each file path is extracted into columns based on the specified format. The format should start from the position of first partition key until the end of file path.

For example, given the path `../Accounts/2019/01/01/data.jsonl` where the partition is by department name and time; the `partition_format='/{Department}/{PartitionDate:yyyy/MM/dd}/data.jsonl'` creates a string column 'Department' with the value 'Accounts' and a datetime column 'PartitionDate' with the value `2019-01-01`.

If your data already has existing partitions and you want to preserve that format, include the `partitioned_format` parameter in your [from_files\(\)](#) method to create a FileDataset.

To create a TabularDataset that preserves existing partitions, include the `partitioned_format` parameter in the [from_parquet_files\(\)](#) or the [from_delimited_files\(\)](#) method.

The following example,

- Creates a FileDataset from partitioned files.
- Gets the partition keys
- Creates a new, indexed FileDataset using

```
file_dataset = Dataset.File.from_files(data_paths, partition_format = '{userid}/*.wav')
ds.register(name='speech_dataset')

# access partition_keys
indexes = file_dataset.partition_keys # ['userid']

# get all partition key value pairs should return [{'userid': 'user1'}, {'userid': 'user2'}]
partitions = file_dataset.get_partition_key_values()

partitions = file_dataset.get_partition_key_values(['userid'])
# return [{'userid': 'user1'}, {'userid': 'user2'}]

# filter API, this will only download data from user1/ folder
new_file_dataset = file_dataset.filter(ds['userid'] == 'user1').download()
```

You can also create a new partitions structure for TabularDatasets with the [partitions_by\(\)](#) method.

```
dataset = Dataset.get_by_name('test') # indexed by country, state, partition_date

# call partition_by locally
new_dataset = ds.partition_by(name="repartitioned_ds", partition_keys=['country'],
target=DataPath(datastore, "repartition"))
partition_keys = new_dataset.partition_keys # ['country']
```

Explore data

After you're done wrangling your data, you can [register](#) your dataset, and then load it into your notebook for data exploration prior to model training.

For FileDatasets, you can either **mount** or **download** your dataset, and apply the Python libraries you'd normally use for data exploration. [Learn more about mount vs download](#).

```
# download the dataset
dataset.download(target_path='.', overwrite=False)

# mount dataset to the temp directory at `mounted_path`

import tempfile
mounted_path = tempfile.mkdtemp()
mount_context = dataset.mount(mounted_path)

mount_context.start()
```

For TabularDatasets, use the [to_pandas_dataframe\(\)](#) method to view your data in a dataframe.

```
# preview the first 3 rows of titanic_ds
titanic_ds.take(3).to_pandas_dataframe()
```

(INDEX)	PASS ENGERID	SURVIVED	PCLASS	NAME	SEX	AGE	SIBSP	PARC H	TICK ET	FARE	CABIN	EMB ARKED
0	1	False	3	Braund, Mr. Owen Harri s	male	22.0	1	0	A/5 2117 1	7.25 00		S
1	2	True	1	Cumings, Mrs. John Bradl ey (Florence Briggs Th...	femal e	38.0	1	0	PC 1759 9	71.2 833	C85	C
2	3	True	3	Heikkinen , Miss. Laina	femal e	26.0	0	0	STON /O2. 3101 282	7.92 50		S

Create a dataset from pandas dataframe

To create a TabularDataset from an in memory pandas dataframe use the [register_pandas_dataframe\(\)](#) method. This method registers the TabularDataset to the workspace and uploads data to your underlying storage, which incurs storage costs.

```
from azureml.core import Workspace, Datastore, Dataset
import pandas as pd

pandas_df = pd.read_csv('<path to your csv file>')
ws = Workspace.from_config()
datastore = Datastore.get(ws, '<name of your datastore>')
dataset = Dataset.Tabular.register_pandas_dataframe(pandas_df, datastore, "dataset_from_pandas_df",
show_progress=True)
```

TIP

Create and register a TabularDataset from an in memory spark dataframe or a dask dataframe with the public preview methods, [register_spark_dataframe\(\)](#) and [register_dask_dataframe\(\)](#). These methods are **experimental** preview features, and may change at any time.

These methods upload data to your underlying storage, and as a result incur storage costs.

Register datasets

To complete the creation process, register your datasets with a workspace. Use the [register\(\)](#) method to register datasets with your workspace in order to share them with others and reuse them across experiments in

your workspace:

```
titanic_ds = titanic_ds.register(workspace=workspace,
                                 name='titanic_ds',
                                 description='titanic training data')
```

Create datasets using Azure Resource Manager

There are many templates at <https://github.com/Azure/azure-quickstart-templates/tree/master//quickstarts/microsoft.machinelearningservices> that can be used to create datasets.

For information on using these templates, see [Use an Azure Resource Manager template to create a workspace for Azure Machine Learning](#).

Train with datasets

Use your datasets in your machine learning experiments for training ML models. [Learn more about how to train with datasets](#).

Version datasets

You can register a new dataset under the same name by creating a new version. A dataset version is a way to bookmark the state of your data so that you can apply a specific version of the dataset for experimentation or future reproduction. Learn more about [dataset versions](#).

```
# create a TabularDataset from Titanic training data
web_paths = ['https://dprepdata.blob.core.windows.net/demo/Titanic.csv',
             'https://dprepdata.blob.core.windows.net/demo/Titanic2.csv']
titanic_ds = Dataset.Tabular.from_delimited_files(path=web_paths)

# create a new version of titanic_ds
titanic_ds = titanic_ds.register(workspace = workspace,
                                  name = 'titanic_ds',
                                  description = 'new titanic training data',
                                  create_new_version = True)
```

Next steps

- Learn [how to train with datasets](#).
- Use automated machine learning to [train with TabularDatasets](#).
- For more dataset training examples, see the [sample notebooks](#).

Connect to data with the Azure Machine Learning studio

9/21/2022 • 10 minutes to read • [Edit Online](#)

In this article, learn how to access your data with the [Azure Machine Learning studio](#). Connect to your data in storage services on Azure with [Azure Machine Learning datastores](#), and then package that data for tasks in your ML workflows with [Azure Machine Learning datasets](#).

The following table defines and summarizes the benefits of datastores and datasets.

OBJECT	DESCRIPTION	BENEFITS
Datastores	Securely connect to your storage service on Azure, by storing your connection information, like your subscription ID and token authorization in your Key Vault associated with the workspace	Because your information is securely stored, you Don't put authentication credentials or original data sources at risk. <ul style="list-style-type: none">• No longer need to hard code them in your scripts.
Datasets	By creating a dataset, you create a reference to the data source location, along with a copy of its metadata. With datasets you can, <ul style="list-style-type: none">• Access data during model training.• Share data and collaborate with other users.• Use open-source libraries, like pandas, for data exploration.	Because datasets are lazily evaluated, and the data remains in its existing location, you <ul style="list-style-type: none">• Keep a single copy of data in your storage.• Incur no extra storage cost• Don't risk unintentionally changing your original data sources.• Improve ML workflow performance speeds.

To understand where datastores and datasets fit in Azure Machine Learning's overall data access workflow, see the [Securely access data](#) article.

For a code first experience, see the following articles to use the [Azure Machine Learning Python SDK](#) to:

- [Connect to Azure storage services with datastores](#).
- [Create Azure Machine Learning datasets](#).

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#).
- Access to [Azure Machine Learning studio](#).
- An Azure Machine Learning workspace. [Create workspace resources](#).
 - When you create a workspace, an Azure blob container and an Azure file share are automatically registered as datastores to the workspace. They're named `workspaceblobstore` and `workspacefilestore`, respectively. If blob storage is sufficient for your needs, the `workspaceblobstore` is set as the default datastore, and already configured for use. Otherwise, you need a storage account on Azure with a [supported storage type](#).

Create datastores

You can create datastores from [these Azure storage solutions](#). For unsupported storage solutions, and to save data egress cost during ML experiments, you must [move your data](#) to a supported Azure storage solution. [Learn more about datastores](#).

You can create datastores with credential-based access or identity-based access.

- [Credential-based](#)
- [Identity-based](#)

Create a new datastore in a few steps with the Azure Machine Learning studio.

IMPORTANT

If your data storage account is in a virtual network, additional configuration steps are required to ensure the studio has access to your data. See [Network isolation & privacy](#) to ensure the appropriate configuration steps are applied.

1. Sign in to [Azure Machine Learning studio](#).
2. Select **Datastores** on the left pane under **Manage**.
3. Select **+ New datastore**.
4. Complete the form to create and register a new datastore. The form intelligently updates itself based on your selections for Azure storage type and authentication type. See the [storage access and permissions section](#) to understand where to find the authentication credentials you need to populate this form.

The following example demonstrates what the form looks like when you create an **Azure blob datastore**:

New datastore

Datastore name *

Datastore type *



Account selection method

 From Azure subscription Enter manually

Subscription ID *



Storage account *



Blob container *



Authentication type



Account key * 

Create datasets

After you create a datastore, create a dataset to interact with your data. Datasets package your data into a lazily evaluated consumable object for machine learning tasks, like training. [Learn more about datasets](#).

There are two types of datasets, FileDataset and TabularDataset. [FileDatasets](#) create references to single or multiple files or public URLs. Whereas [TabularDatasets](#) represent your data in a tabular format. You can create TabularDatasets from .csv, .tsv, .parquet, .jsonl files, and from SQL query results.

The following steps and animation show how to create a dataset in [Azure Machine Learning studio](#).

NOTE

Datasets created through Azure Machine Learning studio are automatically registered to the workspace.

To create a dataset in the studio:

1. Sign in to the [Azure Machine Learning studio](#).
2. Select **Datasets** in the **Assets** section of the left pane.
3. Select **Create Dataset** to choose the source of your dataset. This source can be local files, a datastore, public URLs, or [Azure Open Datasets](#).
4. Select **Tabular** or **File** for Dataset type.
5. Select **Next** to open the **Datastore and file selection** form. On this form you select where to keep your dataset after creation, and select what data files to use for your dataset.
 - a. Enable skip validation if your data is in a virtual network. Learn more about [virtual network isolation and privacy](#).
6. Select **Next** to populate the **Settings and preview** and **Schema** forms; they're intelligently populated based on file type and you can further configure your dataset prior to creation on these forms.
 - a. On the Settings and preview form, you can indicate if your data contains multi-line data.
 - b. On the Schema form, you can specify that your TabularDataset has a time component by selecting type: **Timestamp** for your date or time column.
 - a. If your data is formatted into subsets, for example time windows, and you want to use those subsets for training, select type **Partition timestamp**. Doing so enables time series operations on your dataset. Learn more about how to [use partitions in your dataset for training](#).
7. Select **Next** to review the **Confirm details** form. Check your selections and create an optional data profile for your dataset. Learn more about [data profiling](#).
8. Select **Create** to complete your dataset creation.

Data profile and preview

After you create your dataset, verify you can view the profile and preview in the studio with the following steps.

1. Sign in to the [Azure Machine Learning studio](#)
2. Select **Datasets** in the **Assets** section of the left pane.
3. Select the name of the dataset you want to view.
4. Select the **Explore** tab.
5. Select the **Preview or Profile** tab.

The screenshot shows the Azure Machine Learning Studio interface. On the left, a sidebar menu includes options like New, Home, Author, Notebooks, Automated ML, Designer, Assets (Datasets, Experiments, Modules, Pipelines, Models, Endpoints), Manage (Compute, Environments (preview), Datastores, Data Labeling, Linked Services), and Documentation. The main area is titled "Welcome to the Azure Machine Learning Studio". It features four cards: "Create new" (Notebooks, Automated ML, Designer), "Recent resources" (Runs, Compute, Models, Datasets, showing no runs to display), and "Documentation" (Learning modules, Tutorials, Additional resources). The "Recent resources" card has tabs for Runs, Compute, Models, and Datasets, and columns for Run, Run ID, Experiment, Status, Submitted time, Submitted by, and Run type.

You can get a vast variety of summary statistics across your data set to verify whether your data set is ML-ready. For non-numeric columns, they include only basic statistics like min, max, and error count. For numeric columns, you can also review their statistical moments and estimated quantiles.

Specifically, Azure Machine Learning dataset's data profile includes:

NOTE

Blank entries appear for features with irrelevant types.

STATISTIC	DESCRIPTION
Feature	Name of the column that is being summarized.
Profile	In-line visualization based on the type inferred. For example, strings, booleans, and dates will have value counts, while decimals (numerics) have approximated histograms. This allows you to gain a quick understanding of the distribution of the data.

STATISTIC	DESCRIPTION
Type distribution	In-line value count of types within a column. Nulls are their own type, so this visualization is useful for detecting odd or missing values.
Type	Inferred type of the column. Possible values include: strings, booleans, dates, and decimals.
Min	Minimum value of the column. Blank entries appear for features whose type doesn't have an inherent ordering (like, booleans).
Max	Maximum value of the column.
Count	Total number of missing and non-missing entries in the column.
Not missing count	Number of entries in the column that aren't missing. Empty strings and errors are treated as values, so they won't contribute to the "not missing count."
Quantiles	Approximated values at each quantile to provide a sense of the distribution of the data.
Mean	Arithmetic mean or average of the column.
Standard deviation	Measure of the amount of dispersion or variation of this column's data.
Variance	Measure of how far spread out this column's data is from its average value.
Skewness	Measure of how different this column's data is from a normal distribution.
Kurtosis	Measure of how heavily tailed this column's data is compared to a normal distribution.

Storage access and permissions

To ensure you securely connect to your Azure storage service, Azure Machine Learning requires that you have permission to access the corresponding data storage. This access depends on the authentication credentials used to register the datastore.

Virtual network

If your data storage account is in a **virtual network**, extra configuration steps are required to ensure Azure Machine Learning has access to your data. See [Use Azure Machine Learning studio in a virtual network](#) to ensure the appropriate configuration steps are applied when you create and register your datastore.

Access validation

WARNING

Cross tenant access to storage accounts is not supported. If cross tenant access is needed for your scenario, please reach out to the AzureML Data Support team alias at amldatasupport@microsoft.com for assistance with a custom code solution.

As part of the initial datastore creation and registration process, Azure Machine Learning automatically validates that the underlying storage service exists and the user provided principal (username, service principal, or SAS token) has access to the specified storage.

After datastore creation, this validation is only performed for methods that require access to the underlying storage container, **not** each time datastore objects are retrieved. For example, validation happens if you want to download files from your datastore; but if you just want to change your default datastore, then validation doesn't happen.

To authenticate your access to the underlying storage service, you can provide either your account key, shared access signatures (SAS) tokens, or service principal according to the datastore type you want to create. The [storage type matrix](#) lists the supported authentication types that correspond to each datastore type.

You can find account key, SAS token, and service principal information on your [Azure portal](#).

- If you plan to use an account key or SAS token for authentication, select **Storage Accounts** on the left pane, and choose the storage account that you want to register.
 - The **Overview** page provides information such as the account name, container, and file share name.
 1. For account keys, go to **Access keys** on the **Settings** pane.
 2. For SAS tokens, go to **Shared access signatures** on the **Settings** pane.
- If you plan to use a [service principal](#) for authentication, go to your **App registrations** and select which app you want to use.
 - Its corresponding **Overview** page will contain required information like tenant ID and client ID.

IMPORTANT

- If you need to change your access keys for an Azure Storage account (account key or SAS token), be sure to sync the new credentials with your workspace and the datastores connected to it. Learn how to [sync your updated credentials](#).
- If you unregister and re-register a datastore with the same name, and it fails, the Azure Key Vault for your workspace may not have soft-delete enabled. By default, soft-delete is enabled for the key vault instance created by your workspace, but it may not be enabled if you used an existing key vault or have a workspace created prior to October 2020. For information on how to enable soft-delete, see [Turn on Soft Delete for an existing key vault](#).

Permissions

For Azure blob container and Azure Data Lake Gen 2 storage, make sure your authentication credentials have **Storage Blob Data Reader** access. Learn more about [Storage Blob Data Reader](#). An account SAS token defaults to no permissions.

- For data **read access**, your authentication credentials must have a minimum of list and read permissions for containers and objects.
- For data **write access**, write and add permissions also are required.

Train with datasets

Use your datasets in your machine learning experiments for training ML models. [Learn more about how to train](#)

with datasets.

Next steps

- A step-by-step example of training with TabularDatasets and automated machine learning.
- Train a model.
- For more dataset training examples, see the [sample notebooks](#).

Train models with Azure Machine Learning datasets

9/21/2022 • 10 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this article, you learn how to work with [Azure Machine Learning datasets](#) to train machine learning models. You can use datasets in your local or remote compute target without worrying about connection strings or data paths.

- For structured data, see [Consume datasets in machine learning training scripts](#).
- For unstructured data, see [Mount files to remote compute targets](#).

Azure Machine Learning datasets provide a seamless integration with Azure Machine Learning training functionality like [ScriptRunConfig](#), [HyperDrive](#), and [Azure Machine Learning pipelines](#).

If you aren't ready to make your data available for model training, but want to load your data to your notebook for data exploration, see how to [explore the data in your dataset](#).

Prerequisites

To create and train with datasets, you need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#) today.
- An [Azure Machine Learning workspace](#).
- The [Azure Machine Learning SDK for Python](#) installed (>= 1.13.0), which includes the `azureml-datasets` package.

NOTE

Some Dataset classes have dependencies on the `azureml-dataprep` package. For Linux users, these classes are supported only on the following distributions: Red Hat Enterprise Linux, Ubuntu, Fedora, and CentOS.

Consume datasets in machine learning training scripts

If you have structured data not yet registered as a dataset, create a `TabularDataset` and use it directly in your training script for your local or remote experiment.

In this example, you create an unregistered `TabularDataset` and specify it as a script argument in the `ScriptRunConfig` object for training. If you want to reuse this `TabularDataset` with other experiments in your workspace, see [how to register datasets to your workspace](#).

Create a `TabularDataset`

The following code creates an unregistered `TabularDataset` from a web url.

```
from azureml.core.dataset import Dataset  
  
web_path ='https://dprepdata.blob.core.windows.net/demo/Titanic.csv'  
titanic_ds = Dataset.Tabular.from_delimited_files(path=web_path)
```

TabularDataset objects provide the ability to load the data in your TabularDataset into a pandas or Spark DataFrame so that you can work with familiar data preparation and training libraries without having to leave your notebook.

Access dataset in training script

The following code configures a script argument `--input-data` that you'll specify when you configure your training run (see next section). When the tabular dataset is passed in as the argument value, Azure ML will resolve that to ID of the dataset, which you can then use to access the dataset in your training script (without having to hardcode the name or ID of the dataset in your script). It then uses the `to_pandas_dataframe()` method to load that dataset into a pandas dataframe for further data exploration and preparation prior to training.

NOTE

If your original data source contains NaN, empty strings or blank values, when you use `to_pandas_dataframe()`, then those values are replaced as a `Null` value.

If you need to load the prepared data into a new dataset from an in-memory pandas dataframe, write the data to a local file, like a parquet, and create a new dataset from that file. Learn more about [how to create datasets](#).

```
%%writefile $script_folder/train_titanic.py

import argparse
from azureml.core import Dataset, Run

parser = argparse.ArgumentParser()
parser.add_argument("--input-data", type=str)
args = parser.parse_args()

run = Run.get_context()
ws = run.experiment.workspace

# get the input dataset by ID
dataset = Dataset.get_by_id(ws, id=args.input_data)

# load the TabularDataset to pandas DataFrame
df = dataset.to_pandas_dataframe()
```

Configure the training run

A `ScriptRunConfig` object is used to configure and submit the training run.

This code creates a `ScriptRunConfig` object, `src`, that specifies:

- A script directory for your scripts. All the files in this directory are uploaded into the cluster nodes for execution.
- The training script, `train_titanic.py`.
- The input dataset for training, `titanic_ds`, as a script argument. Azure ML will resolve this to corresponding ID of the dataset when it's passed to your script.
- The compute target for the run.
- The environment for the run.

```

from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory=script_folder,
                      script='train_titanic.py',
                      # pass dataset as an input with friendly name 'titanic'
                      arguments=['--input-data', titanic_ds.as_named_input('titanic')],
                      compute_target=compute_target,
                      environment=myenv)

# Submit the run configuration for your training run
run = experiment.submit(src)
run.wait_for_completion(show_output=True)

```

Mount files to remote compute targets

If you have unstructured data, create a [FileDataset](#) and either mount or download your data files to make them available to your remote compute target for training. Learn about when to use [mount vs. download](#) for your remote training experiments.

The following example,

- Creates an input FileDataset, `mnist_ds`, for your training data.
- Specifies where to write training results, and to promote those results as a FileDataset.
- Mounts the input dataset to the compute target.

NOTE

If you are using a custom Docker base image, you will need to install fuse via `apt-get install -y fuse` as a dependency for dataset mount to work. Learn how to [build a custom build image](#).

For the notebook example, see [How to configure a training run with data input and output](#).

Create a FileDataset

The following example creates an unregistered FileDataset, `mnist_data` from web urls. This FileDataset is the input data for your training run.

Learn more about [how to create datasets](#) from other sources.

```

from azureml.core.dataset import Dataset

web_paths = [
    'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
]

mnist_ds = Dataset.File.from_files(path = web_paths)

```

Where to write training output

You can specify where to write your training results with an [OutputFileDatasetConfig](#) object.

[OutputFileDatasetConfig](#) objects allow you to:

- Mount or upload the output of a run to cloud storage you specify.

- Save the output as a FileDataset to these supported storage types:
 - Azure blob
 - Azure file share
 - Azure Data Lake Storage generations 1 and 2
- Track the data lineage between training runs.

The following code specifies that training results should be saved as a FileDataset in the `outputdataset` folder in the default blob datastore, `def_blob_store`.

```
from azureml.core import Workspace
from azureml.data import OutputFileDatasetConfig

ws = Workspace.from_config()

def_blob_store = ws.get_default_datastore()
output = OutputFileDatasetConfig(destination=(def_blob_store, 'sample/outputdataset'))
```

Configure the training run

We recommend passing the dataset as an argument when mounting via the `arguments` parameter of the `ScriptRunConfig` constructor. By doing so, you get the data path (mounting point) in your training script via arguments. This way, you're able to use the same training script for local debugging and remote training on any cloud platform.

The following example creates a `ScriptRunConfig` that passes in the `FileDataset` via `arguments`. After you submit the run, data files referred to by the dataset `mnist_ds` are mounted to the compute target, and training results are saved to the specified `outputdataset` folder in the default datastore.

```
from azureml.core import ScriptRunConfig

input_data= mnist_ds.as_named_input('input').as_mount()# the dataset will be mounted on the remote compute

src = ScriptRunConfig(source_directory=script_folder,
                      script='dummy_train.py',
                      arguments=[input_data, output],
                      compute_target=compute_target,
                      environment=myenv)

# Submit the run configuration for your training run
run = experiment.submit(src)
run.wait_for_completion(show_output=True)
```

Simple training script

The following script is submitted through the `ScriptRunConfig`. It reads the `mnist_ds` dataset as input, and writes the file to the `outputdataset` folder in the default blob datastore, `def_blob_store`.

```

%%writefile $source_directory/dummy_train.py

# Copyright (c) Microsoft Corporation. All rights reserved.
# Licensed under the MIT License.
import sys
import os

print("*****")
print("Hello Azure ML!")

mounted_input_path = sys.argv[1]
mounted_output_path = sys.argv[2]

print("Argument 1: %s" % mounted_input_path)
print("Argument 2: %s" % mounted_output_path)

with open(mounted_input_path, 'r') as f:
    content = f.read()
    with open(os.path.join(mounted_output_path, 'output.csv'), 'w') as fw:
        fw.write(content)

```

Mount vs download

Mounting or downloading files of any format are supported for datasets created from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL.

When you **mount** a dataset, you attach the files referenced by the dataset to a directory (mount point) and make it available on the compute target. Mounting is supported for Linux-based computes, including Azure Machine Learning Compute, virtual machines, and HDInsight. If your data size exceeds the compute disk size, downloading isn't possible. For this scenario, we recommend mounting since only the data files used by your script are loaded at the time of processing.

When you **download** a dataset, all the files referenced by the dataset will be downloaded to the compute target. Downloading is supported for all compute types. If your script processes all files referenced by the dataset, and your compute disk can fit your full dataset, downloading is recommended to avoid the overhead of streaming data from storage services. For multi-node downloads, see [how to avoid throttling](#).

NOTE

The download path name should not be longer than 255 alpha-numeric characters for Windows OS. For Linux OS, the download path name should not be longer than 4,096 alpha-numeric characters. Also, for Linux OS the file name (which is the last segment of the download path `/path/to/file/{filename}`) should not be longer than 255 alpha-numeric characters.

The following code mounts `dataset` to the temp directory at `mounted_path`

```

import tempfile
mounted_path = tempfile.mkdtemp()

# mount dataset onto the mounted_path of a Linux-based compute
mount_context = dataset.mount(mounted_path)

mount_context.start()

import os
print(os.listdir(mounted_path))
print (mounted_path)

```

Get datasets in machine learning scripts

Registered datasets are accessible both locally and remotely on compute clusters like the Azure Machine Learning compute. To access your registered dataset across experiments, use the following code to access your workspace and get the dataset that was used in your previously submitted run. By default, the `get_by_name()` method on the `Dataset` class returns the latest version of the dataset that's registered with the workspace.

```
%%writefile $script_folder/train.py

from azureml.core import Dataset, Run

run = Run.get_context()
workspace = run.experiment.workspace

dataset_name = 'titanic_ds'

# Get a dataset by name
titanic_ds = Dataset.get_by_name(workspace=workspace, name=dataset_name)

# Load a TabularDataset into pandas DataFrame
df = titanic_ds.to_pandas_dataframe()
```

Access source code during training

Azure Blob storage has higher throughput speeds than an Azure file share and will scale to large numbers of jobs started in parallel. For this reason, we recommend configuring your runs to use Blob storage for transferring source code files.

The following code example specifies in the run configuration which blob datastore to use for source code transfers.

```
# workspaceblobstore is the default blob storage
src.run_config.source_directory_data_store = "workspaceblobstore"
```

Notebook examples

- For more dataset examples and concepts, see the [dataset notebooks](#).
- See how to [parametrize datasets in your ML pipelines](#).

Troubleshooting

Dataset initialization failed: Waiting for mount point to be ready has timed out:

- If you don't have any outbound `network security group` rules and are using `azureml-sdk>=1.12.0`, update `azureml-dataset-runtime` and its dependencies to be the latest for the specific minor version, or if you're using it in a run, recreate your environment so it can have the latest patch with the fix.
- If you're using `azureml-sdk<1.12.0`, upgrade to the latest version.
- If you have outbound NSG rules, make sure there's an outbound rule that allows all traffic for the service tag `AzureResourceMonitor`.

Dataset initialization failed: StreamAccessException was caused by ThrottlingException

For multi-node file downloads, all nodes may attempt to download all files in the file dataset from the Azure Storage service, which results in a throttling error. To avoid throttling, initially set the environment variable `AZUREML_DOWNLOAD_CONCURRENCY` to a value of eight times the number of CPU cores divided by the number of

nodes. Setting up a value for this environment variable may require some experimentation, so the aforementioned guidance is a starting point.

The following example assumes 32 cores and 4 nodes.

```
from azureml.core.environment import Environment
myenv = Environment(name="myenv")
myenv.environment_variables = {"AZUREML_DOWNLOAD_CONCURRENCY":64}
```

AzureFile storage

Unable to upload project files to working directory in AzureFile because the storage is overloaded:

- If you're using file share for other workloads, such as data transfer, the recommendation is to use blobs so that file share is free to be used for submitting runs.
- Another option is to split the workload between two different workspaces.

ConfigException: Could not create a connection to the AzureFileService due to missing credentials. Either an Account Key or SAS token needs to be linked the default workspace blob store.

To ensure your storage access credentials are linked to the workspace and the associated file datastore, complete the following steps:

1. Navigate to your workspace in the [Azure portal](#).
2. Select the storage link on the workspace **Overview** page.
3. On the storage page, select **Access keys** on the left side menu.
4. Copy the key.
5. Navigate to the [Azure Machine Learning studio](#) for your workspace.
6. In the studio, select the file datastore for which you want to provide authentication credentials.
7. Select **Update authentication**.
8. Paste the key from the previous steps.
9. Select **Save**.

Passing data as input

TypeError: FileNotFoundError: No such file or directory: This error occurs if the file path you provide isn't where the file is located. You need to make sure the way you refer to the file is consistent with where you mounted your dataset on your compute target. To ensure a deterministic state, we recommend using the abstract path when mounting a dataset to a compute target. For example, in the following code we mount the dataset under the root of the filesystem of the compute target, `/tmp`.

```
# Note the leading / in '/tmp/dataset'
script_params = {
    '--data-folder': dset.as_named_input('dogscats_train').as_mount('/tmp/dataset'),
}
```

If you don't include the leading forward slash, `'/'`, you'll need to prefix the working directory for example, `/mnt/batch/.../tmp/dataset` on the compute target to indicate where you want the dataset to be mounted.

Next steps

- [Auto train machine learning models](#) with TabularDatasets.
- [Train image classification models](#) with FileDatasets.
- [Train with datasets using pipelines](#).

Detect data drift (preview) on datasets

9/21/2022 • 15 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

Learn how to monitor data drift and set alerts when drift is high.

With Azure Machine Learning dataset monitors (preview), you can:

- **Analyze drift in your data** to understand how it changes over time.
- **Monitor model data** for differences between training and serving datasets. Start by [collecting model data from deployed models](#).
- **Monitor new data** for differences between any baseline and target dataset.
- **Profile features in data** to track how statistical properties change over time.
- **Set up alerts on data drift** for early warnings to potential issues.
- **Create a new dataset version** when you determine the data has drifted too much.

An [Azure Machine learning dataset](#) is used to create the monitor. The dataset must include a timestamp column.

You can view data drift metrics with the Python SDK or in Azure Machine Learning studio. Other metrics and insights are available through the [Azure Application Insights](#) resource associated with the Azure Machine Learning workspace.

IMPORTANT

Data drift detection for datasets is currently in public preview. The preview version is provided without a service level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Prerequisites

To create and work with dataset monitors, you need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#) today.
- An [Azure Machine Learning workspace](#).
- The [Azure Machine Learning SDK for Python installed](#), which includes the `azureml-datasets` package.
- Structured (tabular) data with a timestamp specified in the file path, file name, or column in the data.

What is data drift?

Data drift is one of the top reasons model accuracy degrades over time. For machine learning models, data drift is the change in model input data that leads to model performance degradation. Monitoring data drift helps detect these model performance issues.

Causes of data drift include:

- Upstream process changes, such as a sensor being replaced that changes the units of measurement from inches to centimeters.
- Data quality issues, such as a broken sensor always reading 0.
- Natural drift in the data, such as mean temperature changing with the seasons.

- Change in relation between features, or covariate shift.

Azure Machine Learning simplifies drift detection by computing a single metric abstracting the complexity of datasets being compared. These datasets may have hundreds of features and tens of thousands of rows. Once drift is detected, you drill down into which features are causing the drift. You then inspect feature level metrics to debug and isolate the root cause for the drift.

This top down approach makes it easy to monitor data instead of traditional rules-based techniques. Rules-based techniques such as allowed data range or allowed unique values can be time consuming and error prone.

In Azure Machine Learning, you use dataset monitors to detect and alert for data drift.

Dataset monitors

With a dataset monitor you can:

- Detect and alert to data drift on new data in a dataset.
- Analyze historical data for drift.
- Profile new data over time.

The data drift algorithm provides an overall measure of change in data and indication of which features are responsible for further investigation. Dataset monitors produce a number of other metrics by profiling new data in the `timeseries` dataset.

Custom alerting can be set up on all metrics generated by the monitor through [Azure Application Insights](#). Dataset monitors can be used to quickly catch data issues and reduce the time to debug the issue by identifying likely causes.

Conceptually, there are three primary scenarios for setting up dataset monitors in Azure Machine Learning.

SCENARIO	DESCRIPTION
Monitor a model's serving data for drift from the training data	Results from this scenario can be interpreted as monitoring a proxy for the model's accuracy, since model accuracy degrades when the serving data drifts from the training data.
Monitor a time series dataset for drift from a previous time period.	This scenario is more general, and can be used to monitor datasets involved upstream or downstream of model building. The target dataset must have a timestamp column. The baseline dataset can be any tabular dataset that has features in common with the target dataset.
Perform analysis on past data.	This scenario can be used to understand historical data and inform decisions in settings for dataset monitors.

Dataset monitors depend on the following Azure services.

AZURE SERVICE	DESCRIPTION
<i>Dataset</i>	Drift uses Machine Learning datasets to retrieve training data and compare data for model training. Generating profile of data is used to generate some of the reported metrics such as min, max, distinct values, distinct values count.
<i>Azureml pipeline and compute</i>	The drift calculation job is hosted in azureml pipeline. The job is triggered on demand or by schedule to run on a compute configured at drift monitor creation time.

AZURE SERVICE	DESCRIPTION
<i>Application insights</i>	Drift emits metrics to Application Insights belonging to the machine learning workspace.
<i>Azure blob storage</i>	Drift emits metrics in json format to Azure blob storage.

Baseline and target datasets

You monitor [Azure machine learning datasets](#) for data drift. When you create a dataset monitor, you will reference your:

- Baseline dataset - usually the training dataset for a model.
 - Target dataset - usually model input data - is compared over time to your baseline dataset. This comparison means that your target dataset must have a timestamp column specified.

The monitor will compare the baseline and target datasets.

Create target dataset

The target dataset needs the `timeseries` trait set on it by specifying the timestamp column either from a column in the data or a virtual column derived from the path pattern of the files. Create the dataset with a timestamp through the [Python SDK](#) or [Azure Machine Learning studio](#). A column representing a "timestamp" must be specified to add `timeseries` trait to the dataset. If your data is partitioned into folder structure with time info, such as '{yyyy/MM/dd}', create a virtual column through the path pattern setting and set it as the "partition timestamp" to enable time series API functionality.

- Python SDK
 - Studio

APPLIES TO: Python SDK azureml v1

The `Dataset` class `with_timestamp_columns()` method defines the time stamp column for the dataset.

```
from azureml.core import Workspace, Dataset, Datastore

# get workspace object
ws = Workspace.from_config()

# get datastore object
dstore = Datastore.get(ws, 'your datastore name')

# specify datastore paths
dstore_paths = [(dstore, 'weather/*/*/*/*data.parquet')]

# specify partition format
partition_format = 'weather/{state}/{date:yyyy/MM/dd}/data.parquet'

# create the Tabular dataset with 'state' and 'date' as virtual columns
dset = Dataset.Tabular.from_parquet_files(path=dstore_paths, partition_format=partition_format)

# assign the timestamp attribute to a real or virtual column in the dataset
dset = dset.with_timestamp_columns('date')

# register the dataset as the target dataset
dset = dset.register(ws, 'target')
```

TIP

For a full example of using the `timeseries` trait of datasets, see the [example notebook](#) or the [datasets SDK documentation](#).

Create dataset monitor

Create a dataset monitor to detect and alert to data drift on a new dataset. Use either the [Python SDK](#) or [Azure Machine Learning studio](#).

- [Python SDK](#)
- [Studio](#)

APPLIES TO:  [Python SDK azureml v1](#)

See the [Python SDK reference documentation on data drift](#) for full details.

The following example shows how to create a dataset monitor using the Python SDK

```
from azureml.core import Workspace, Dataset
from azureml.datadrift import DataDriftDetector
from datetime import datetime

# get the workspace object
ws = Workspace.from_config()

# get the target dataset
target = Dataset.get_by_name(ws, 'target')

# set the baseline dataset
baseline = target.time_before(datetime(2019, 2, 1))

# set up feature list
features = ['latitude', 'longitude', 'elevation', 'windAngle', 'windSpeed', 'temperature', 'snowDepth',
'stationName', 'countryOrRegion']

# set up data drift detector
monitor = DataDriftDetector.create_from_datasets(ws, 'drift-monitor', baseline, target,
                                                 compute_target='cpu-cluster',
                                                 frequency='Week',
                                                 feature_list=None,
                                                 drift_threshold=.6,
                                                 latency=24)

# get data drift detector by name
monitor = DataDriftDetector.get_by_name(ws, 'drift-monitor')

# update data drift detector
monitor = monitor.update(feature_list=features)

# run a backfill for January through May
backfill1 = monitor.backfill(datetime(2019, 1, 1), datetime(2019, 5, 1))

# run a backfill for May through today
backfill1 = monitor.backfill(datetime(2019, 5, 1), datetime.today())

# disable the pipeline schedule for the data drift detector
monitor = monitor.disable_schedule()

# enable the pipeline schedule for the data drift detector
monitor = monitor.enable_schedule()
```

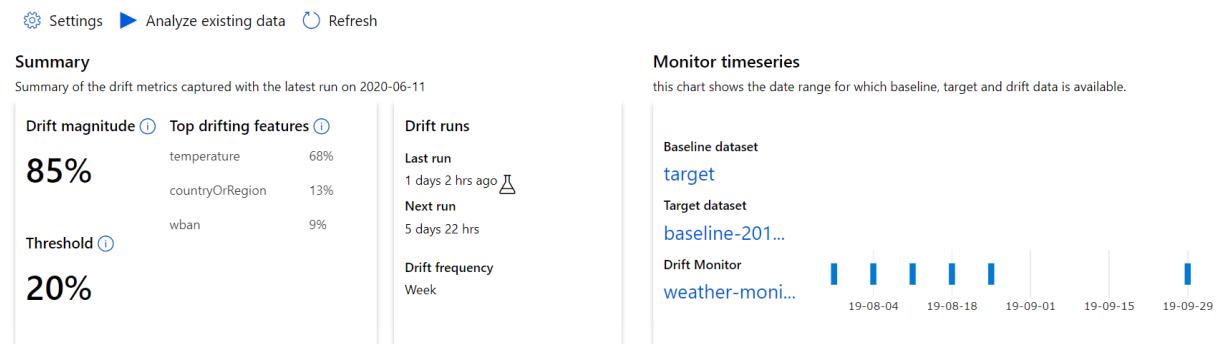
TIP

For a full example of setting up a `timeseries` dataset and data drift detector, see our [example notebook](#).

Understand data drift results

This section shows you the results of monitoring a dataset, found in the **Datasets / Dataset monitors** page in Azure studio. You can update the settings as well as analyze existing data for a specific time period on this page.

Start with the top-level insights into the magnitude of data drift and a highlight of features to be further investigated.



METRIC	DESCRIPTION
Data drift magnitude	A percentage of drift between the baseline and target dataset over time. Ranging from 0 to 100, 0 indicates identical datasets and 100 indicates the Azure Machine Learning data drift model can completely tell the two datasets apart. Noise in the precise percentage measured is expected due to machine learning techniques being used to generate this magnitude.
Top drifting features	Shows the features from the dataset that have drifted the most and are therefore contributing the most to the Drift Magnitude metric. Due to covariate shift, the underlying distribution of a feature does not necessarily need to change to have relatively high feature importance.
Threshold	Data Drift magnitude beyond the set threshold will trigger alerts. This can be configured in the monitor settings.

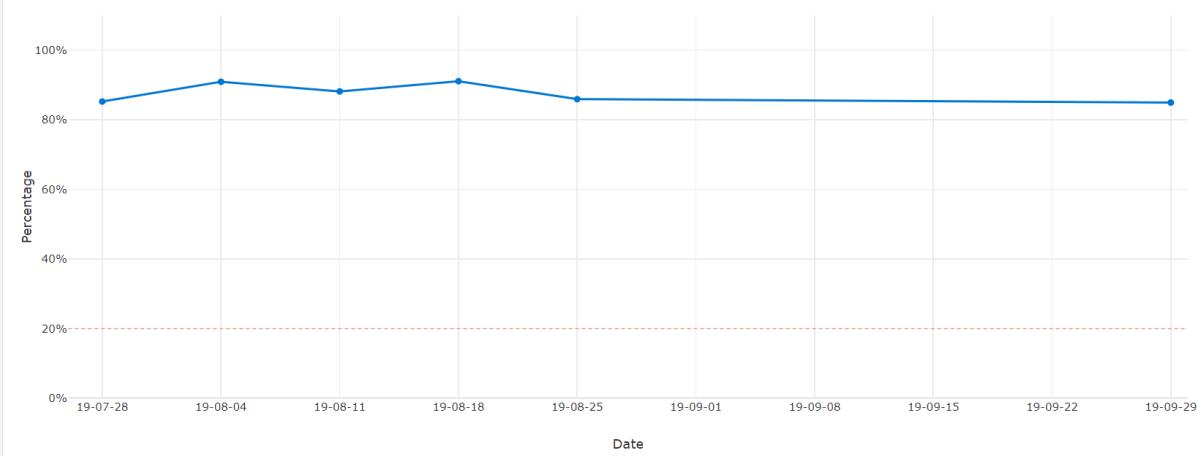
Drift magnitude trend

See how the dataset differs from the target dataset in the specified time period. The closer to 100%, the more the two datasets differ.

Start date:	End date:
7/27/2019	9/29/2019

Drift magnitude trend

Drift magnitude metric measures the difference between target dataset for the time interval and baseline dataset. 0% implies that the target data is identical to the baseline data and 100% implies the target data is completely different from the baseline data.



Drift magnitude by features

This section contains feature-level insights into the change in the selected feature's distribution, as well as other statistics, over time.

The target dataset is also profiled over time. The statistical distance between the baseline distribution of each feature is compared with the target dataset's over time. Conceptually, this is similar to the data drift magnitude. However this statistical distance is for an individual feature rather than all features. Min, max, and mean are also available.

In the Azure Machine Learning studio, click on a bar in the graph to see the feature-level details for that date. By default, you see the baseline dataset's distribution and the most recent job's distribution of the same feature.



These metrics can also be retrieved in the Python SDK through the `get_metrics()` method on a `DataDriftDetector` object.

Feature details

Finally, scroll down to view details for each individual feature. Use the dropdowns above the chart to select the feature, and additionally select the metric you want to view.

weather_monitor_daily

Settings Analyze existing data Refresh

Feature details

Select feature: Select metrics:

temperature

Wasserstein distance



Wasserstein distance



Feature distribution



baseline
2019-06-27

Metrics in the chart depend on the type of feature.

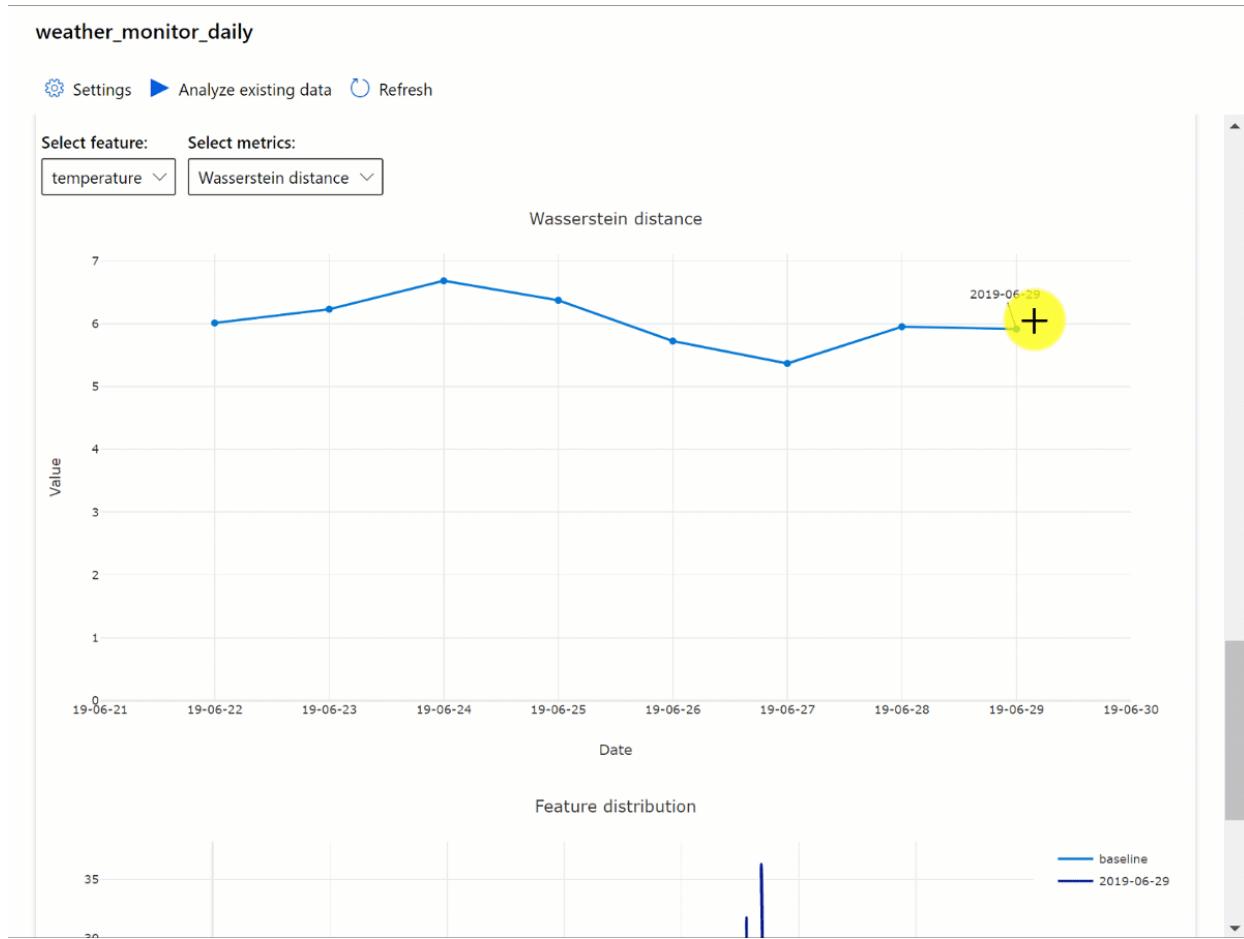
- Numeric features

METRIC	DESCRIPTION
Wasserstein distance	Minimum amount of work to transform baseline distribution into the target distribution.
Mean value	Average value of the feature.
Min value	Minimum value of the feature.
Max value	Maximum value of the feature.

- Categorical features

METRIC	DESCRIPTION
Euclidian distance	Computed for categorical columns. Euclidean distance is computed on two vectors, generated from empirical distribution of the same categorical column from two datasets. 0 indicates there is no difference in the empirical distributions. The more it deviates from 0, the more this column has drifted. Trends can be observed from a time series plot of this metric and can be helpful in uncovering a drifting feature.
Unique values	Number of unique values (cardinality) of the feature.

On this chart, select a single date to compare the feature distribution between the target and this date for the displayed feature. For numeric features, this shows two probability distributions. If the feature is numeric, a bar chart is shown.



Metrics, alerts, and events

Metrics can be queried in the [Azure Application Insights](#) resource associated with your machine learning workspace. You have access to all features of Application Insights including set up for custom alert rules and action groups to trigger an action such as, an Email/SMS/Push/Voice or Azure Function. Refer to the complete Application Insights documentation for details.

To get started, navigate to the [Azure portal](#) and select your workspace's **Overview** page. The associated Application Insights resource is on the far right:

The screenshot shows the "Overview" page for the "NCUS-AzureML" workspace. On the left, there is a sidebar with links: "Overview" (highlighted with a red box), "Activity log", "Access control (IAM)", "Tags", and "Diagnose and solve problems". The main pane displays resource details: "Resource group : ncusazureml", "Location : North Central US", "Subscription : Feedback-FeatureSynthesizer", "Subscription ID : [redacted]", "Storage : ncusazureml", "Registry : ncusazureml", "Key Vault : ncusazureml", and "Application insights : ncusazureml" (also highlighted with a red box). There are also buttons for "Download config.json" and "Delete".

Select Logs (Analytics) under Monitoring on the left pane:

The screenshot shows the Azure Application Insights portal interface. At the top, there's a search bar labeled "Search (Ctrl+ /)" and a "ncusazureml" application name. Below the search bar is a navigation menu with several sections:

- Overview**
- Activity log**
- Access control (IAM)**
- Tags**
- Diagnose and solve problems**

Investigate

- Application map**
- Smart Detection**
- Live Metrics Stream**
- Search**
- Availability**
- Failures**
- Performance**
- Servers**
- Browser**
- Troubleshooting guides (pre...)**

Monitoring

- Alerts**
- Metrics**
- Logs (Analytics)** (This item is highlighted with a red box.)
- Workbooks**

Usage

- Users**

The dataset monitor metrics are stored as `customMetrics`. You can write and run a query after setting up a dataset monitor to view them:

The screenshot shows the Azure Application Insights Logs Analytics interface. A search query `customMetrics | take 1000` is run against the `ncusazureml` resource. The results table displays 1,000 records from the last 7 days, grouped by timestamp [UTC], name, value, valueCount, valueSum, valueMin, valueMax, customDimensions, and operation_Syn. The table includes columns for timestamp, name, value, valueCount, valueSum, valueMin, valueMax, customDimensions, and operation_Syn. The data shows various metrics like energy_distance, datadrift_contribution, max, min, and measures over time.

After identifying metrics to set up alert rules, create a new alert rule:

The screenshot shows the "Create rule" wizard in the Azure portal under Rules management. It's step 1 of 4: Set condition. The condition is defined as "Whenever the Custom log search is <logic undefined>". The condition details show a monthly cost of \$1.50. The alert actions section is currently empty, with a note that Azure Alerts are limited to one signal per rule. The "Customize Actions" section includes options for Email subject and Include custom Json payload for webhook.

You can use an existing action group, or create a new one to define the action to be taken when the set conditions are met:

Add action group

Action group name * ⓘ	<input type="text"/>			
Short name * ⓘ	<input type="text"/>			
Subscription * ⓘ	Feedback-FeatureSynthesizer			
Resource group * ⓘ	Default-ActivityLogAlerts			
Actions				
Action Name *	Action Type *	Status	Configure	Actions
<input type="text"/> Unique name for the action	<input type="text"/> Select an action type <ul style="list-style-type: none"> Automation Runbook Azure Function Email Azure Resource Manager Role Email/SMS/Push/Voice ITSM LogicApp Secure Webhook Webhook 			
Privacy Statement Pricing <p>i Have a consistent format in email details. Learn more</p>				
<small>Optional. You can add multiple actions to an action group. Each action will receive the monitoring source data. You can enable per action by editing its configuration.</small>				
<input type="button" value="OK"/>				

Troubleshooting

Limitations and known issues for data drift monitors:

- The time range when analyzing historical data is limited to 31 intervals of the monitor's frequency setting.
- Limitation of 200 features, unless a feature list is not specified (all features used).
- Compute size must be large enough to handle the data.
- Ensure your dataset has data within the start and end date for a given monitor job.
- Dataset monitors will only work on datasets that contain 50 rows or more.
- Columns, or features, in the dataset are classified as categorical or numeric based on the conditions in the following table. If the feature does not meet these conditions - for instance, a column of type string with >100 unique values - the feature is dropped from our data drift algorithm, but is still profiled.

FEATURE TYPE	DATA TYPE	CONDITION	LIMITATIONS
--------------	-----------	-----------	-------------

FEATURE TYPE	DATA TYPE	CONDITION	LIMITATIONS
Categorical	string, bool, int, float	The number of unique values in the feature is less than 100 and less than 5% of the number of rows.	Null is treated as its own category.
Numerical	int, float	The values in the feature are of a numerical data type and do not meet the condition for a categorical feature.	Feature dropped if >15% of values are null.

- When you have created a data drift monitor but cannot see data on the **Dataset monitors** page in Azure Machine Learning studio, try the following.
 - Check if you have selected the right date range at the top of the page.
 - On the **Dataset Monitors** tab, select the experiment link to check job status. This link is on the far right of the table.
 - If the job completed successfully, check the driver logs to see how many metrics have been generated or if there's any warning messages. Find driver logs in the **Output + logs** tab after you click on an experiment.
- If the SDK `backfill()` function does not generate the expected output, it may be due to an authentication issue. When you create the compute to pass into this function, do not use `Run.get_context().experiment.workspace.compute_targets`. Instead, use `ServicePrincipalAuthentication` such as the following to create the compute that you pass into that `backfill()` function:

```
auth = ServicePrincipalAuthentication(
    tenant_id=tenant_id,
    service_principal_id=app_id,
    service_principal_password=client_secret
)
ws = Workspace.get("xxx", auth=auth, subscription_id="xxx", resource_group="xxx")
compute = ws.compute_targets.get("xxx")
```

- From the Model Data Collector, it can take up to (but usually less than) 10 minutes for data to arrive in your blob storage account. In a script or Notebook, wait 10 minutes to ensure cells below will run.

```
import time
time.sleep(600)
```

Next steps

- Head to the [Azure Machine Learning studio](#) or the [Python notebook](#) to set up a dataset monitor.
- See how to set up data drift on [models deployed to Azure Kubernetes Service](#).
- Set up dataset drift monitors with [Azure Event Grid](#).

Version and track Azure Machine Learning datasets

9/21/2022 • 5 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you'll learn how to version and track Azure Machine Learning datasets for reproducibility. Dataset versioning is a way to bookmark the state of your data so that you can apply a specific version of the dataset for future experiments.

Typical versioning scenarios:

- When new data is available for retraining
- When you're applying different data preparation or feature engineering approaches

Prerequisites

For this tutorial, you need:

- [Azure Machine Learning SDK for Python installed](#). This SDK includes the `azureml-datasets` package.
- An [Azure Machine Learning workspace](#). Retrieve an existing one by running the following code, or [create a new workspace](#).

```
import azureml.core
from azureml.core import Workspace

ws = Workspace.from_config()
```

- An [Azure Machine Learning dataset](#).

Register and retrieve dataset versions

By registering a dataset, you can version, reuse, and share it across experiments and with colleagues. You can register multiple datasets under the same name and retrieve a specific version by name and version number.

Register a dataset version

The following code registers a new version of the `titanic_ds` dataset by setting the `create_new_version` parameter to `True`. If there's no existing `titanic_ds` dataset registered with the workspace, the code creates a new dataset with the name `titanic_ds` and sets its version to 1.

```
titanic_ds = titanic_ds.register(workspace = workspace,
                                 name = 'titanic_ds',
                                 description = 'titanic training data',
                                 create_new_version = True)
```

Retrieve a dataset by name

By default, the `get_by_name()` method on the `Dataset` class returns the latest version of the dataset registered with the workspace.

The following code gets version 1 of the `titanic_ds` dataset.

```

from azureml.core import Dataset
# Get a dataset by name and version number
titanic_ds = Dataset.get_by_name(workspace = workspace,
                                 name = 'titanic_ds',
                                 version = 1)

```

Versioning best practice

When you create a dataset version, you're *not* creating an extra copy of data with the workspace. Because datasets are references to the data in your storage service, you have a single source of truth, managed by your storage service.

IMPORTANT

If the data referenced by your dataset is overwritten or deleted, calling a specific version of the dataset does *not* revert the change.

When you load data from a dataset, the current data content referenced by the dataset is always loaded. If you want to make sure that each dataset version is reproducible, we recommend that you not modify data content referenced by the dataset version. When new data comes in, save new data files into a separate data folder and then create a new dataset version to include data from that new folder.

The following image and sample code show the recommended way to structure your data folders and to create dataset versions that reference those folders:

NAME	ACCESS TIER	ACCESS TIER LAST MODIFIED	LAST MODIFIED	BLOB TYPE	CONTENT TYPE	SIZE
week 27				Folder		
week 28				Folder		
week 29				Folder		

```

from azureml.core import Dataset

# get the default datastore of the workspace
datastore = workspace.get_default_datastore()

# create & register weather_ds version 1 pointing to all files in the folder of week 27
datastore_path1 = [(datastore, 'Weather/week 27')]
dataset1 = Dataset.File.from_files(path=datastore_path1)
dataset1.register(workspace = workspace,
                  name = 'weather_ds',
                  description = 'weather data in week 27',
                  create_new_version = True)

# create & register weather_ds version 2 pointing to all files in the folder of week 27 and 28
datastore_path2 = [(datastore, 'Weather/week 27'), (datastore, 'Weather/week 28')]
dataset2 = Dataset.File.from_files(path = datastore_path2)
dataset2.register(workspace = workspace,
                  name = 'weather_ds',
                  description = 'weather data in week 27, 28',
                  create_new_version = True)

```

Version an ML pipeline output dataset

You can use a dataset as the input and output of each [ML pipeline](#) step. When you rerun pipelines, the output of each pipeline step is registered as a new dataset version.

ML pipelines populate the output of each step into a new folder every time the pipeline reruns. This behavior allows the versioned output datasets to be reproducible. Learn more about [datasets in pipelines](#).

```
from azureml.core import Dataset
from azureml.pipeline.steps import PythonScriptStep
from azureml.pipeline.core import Pipeline, PipelineData
from azureml.core.runconfig import CondaDependencies, RunConfiguration

# get input dataset
input_ds = Dataset.get_by_name(workspace, 'weather_ds')

# register pipeline output as dataset
output_ds = PipelineData('prepared_weather_ds', datastore=datastore).as_dataset()
output_ds = output_ds.register(name='prepared_weather_ds', create_new_version=True)

conda = CondaDependencies.create(
    pip_packages=['azureml-defaults', 'azureml-dataprep[fuse,pandas]'],
    pin_sdk_version=False)

run_config = RunConfiguration()
run_config.environment.docker.enabled = True
run_config.environment.python.conda_dependencies = conda

# configure pipeline step to use dataset as the input and output
prep_step = PythonScriptStep(script_name="prepare.py",
                             inputs=[input_ds.as_named_input('weather_ds')],
                             outputs=[output_ds],
                             runconfig=run_config,
                             compute_target=compute_target,
                             source_directory=project_folder)
```

Track data in your experiments

Azure Machine Learning tracks your data throughout your experiment as input and output datasets.

The following are scenarios where your data is tracked as an **input dataset**.

- As a `DatasetConsumptionConfig` object through either the `inputs` or `arguments` parameter of your `ScriptRunConfig` object when submitting the experiment job.
- When methods like, `get_by_name()` or `get_by_id()` are called in your script. For this scenario, the name assigned to the dataset when you registered it to the workspace is the name displayed.

The following are scenarios where your data is tracked as an **output dataset**.

- Pass an `OutputFileDatasetConfig` object through either the `outputs` or `arguments` parameter when submitting an experiment job. `OutputFileDatasetConfig` objects can also be used to persist data between pipeline steps. See [Move data between ML pipeline steps](#).
- Register a dataset in your script. For this scenario, the name assigned to the dataset when you registered it to the workspace is the name displayed. In the following example, `training_ds` is the name that would be displayed.

```
training_ds = unregistered_ds.register(workspace = workspace,
                                         name = 'training_ds',
                                         description = 'training data'
                                         )
```

- Submit child job with an unregistered dataset in script. This results in an anonymous saved dataset.

Trace datasets in experiment jobs

For each Machine Learning experiment, you can easily trace the datasets used as input with the experiment `Job` object.

The following code uses the `get_details()` method to track which input datasets were used with the experiment run:

```
# get input datasets
inputs = run.get_details()['inputDatasets']
input_dataset = inputs[0]['dataset']

# list the files referenced by input_dataset
input_dataset.to_path()
```

You can also find the `input_datasets` from experiments by using the [Azure Machine Learning studio](#).

The following image shows where to find the input dataset of an experiment on Azure Machine Learning studio. For this example, go to your **Experiments** pane and open the **Properties** tab for a specific run of your experiment, `keras-mnist`.

The screenshot shows the Azure Machine Learning studio interface. On the left, there's a sidebar with navigation links like 'New', 'Home', 'Notebooks', 'Automated ML', 'Designer', 'Assets', 'Datasets', 'Experiments' (which is selected), 'Pipelines', 'Models', 'Endpoints', 'Compute', and 'Environments'. The main area is titled 'Run 24' under 'mayworkspace > Experiments > keras-mnist > keras-mnist_1570739212_92bc7af'. It has tabs for 'Properties', 'Metrics', 'Images', 'Child runs', 'Outputs', 'Logs', 'Snapshot', and 'Raw JSON'. The 'Properties' tab is active. It displays various details about the run, including Status (Completed), Created (Oct 10, 2019 1:31 PM), Duration (1m 55.69s), Compute target (local), Run ID (keras-mnist_1570739212_92bc7af), Run number (24), Script name (keras_mnist.py), and Input datasets (keras-mnist). The 'Input datasets' section is highlighted with a red box.

Use the following code to register models with datasets:

```
model = run.register_model(model_name='keras-mlp-mnist',
                           model_path=model_path,
                           datasets =[('training data',train_dataset)])
```

After registration, you can see the list of models registered with the dataset by using Python or go to the [studio](#).

The following view is from the **Datasets** pane under **Assets**. Select the dataset and then select the **Models** tab

for a list of the models that are registered with the dataset.

The screenshot shows a Jupyter Notebook interface with the following details:

- Header: mayworkspace > Data > mnist dataset
- Section: mnist dataset Version 1 (latest) ▾
- Actions: Refresh, Unregister, New version ▾
- Navigation: Overview, **Models** (highlighted with a red box), Search to filter items...
- Table: A list of registered models with the following columns: Name, Model version, Usage, Registered on, and Description.

Name	Model version	Usage	Registered on	Description
keras-mlp-mnist	4	training data	Oct 11, 2019 10:50 AM	--
keras-mlp-mnist	3	training data	Oct 10, 2019 10:16 PM	--
keras-mlp-mnist	2	training data	Oct 10, 2019 10:00 AM	--
keras-mlp-mnist	1	training data	Oct 8, 2019 11:24 PM	--
- Pagination: < Prev, Next >

Next steps

- [Train with datasets](#)
- [More sample dataset notebooks](#)

Create and explore Azure Machine Learning dataset with labels

9/21/2022 • 2 minutes to read • [Edit Online](#)

In this article, you'll learn how to export the data labels from an Azure Machine Learning data labeling project and load them into popular formats such as, a pandas dataframe for data exploration.

What are datasets with labels

Azure Machine Learning datasets with labels are referred to as labeled datasets. These specific datasets are [TabularDatasets](#) with a dedicated label column and are only created as an output of Azure Machine Learning data labeling projects. Create a data labeling project [for image labeling](#) or [text labeling](#). Machine Learning supports data labeling projects for image classification, either multi-label or multi-class, and object identification together with bounded boxes.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- The [Azure Machine Learning SDK for Python](#), or access to [Azure Machine Learning studio](#).
- A Machine Learning workspace. See [Create workspace resources](#).
- Access to an Azure Machine Learning data labeling project. If you don't have a labeling project, first create one for [image labeling](#) or [text labeling](#).

Export data labels

When you complete a data labeling project, you can [export the label data from a labeling project](#). Doing so, allows you to capture both the reference to the data and its labels, and export them in [COCO format](#) or as an Azure Machine Learning dataset.

Use the **Export** button on the **Project details** page of your labeling project.

The screenshot shows the Azure Machine Learning Studio interface. On the left is a navigation sidebar with options like New, Home, Notebooks, Automated ML, Designer, Assets, and Data. The main area shows a breadcrumb path: my-ws > Data Labeling > tutorial-cats-n-dogs. Below the path is the project name 'tutorial-cats-n-dogs'. Underneath it are two buttons: 'Export' with a dropdown arrow and 'Refresh'. At the bottom of the main area are three tabs: 'Dashboard', 'Data', and 'Details', with 'Details' being the active tab. To the right of the tabs is a section titled 'Project details' which is currently empty. At the very bottom is a section titled 'Input datasets'.

COCO

The COCO file is created in the default blob store of the Azure Machine Learning workspace in a folder within `export/coco`.

NOTE

In object detection projects, the exported "bbox": [x,y,width,height]" values in COCO file are normalized. They are scaled to 1. Example : a bounding box at (10, 10) location, with 30 pixels width , 60 pixels height, in a 640x480 pixel image will be annotated as (0.015625. 0.02083, 0.046875, 0.125). Since the coordinates are normalized, it will show as '0.0' as "width" and "height" for all images. The actual width and height can be obtained using Python library like OpenCV or Pillow(PIL).

Azure Machine Learning dataset

You can access the exported Azure Machine Learning dataset in the **Datasets** section of your Azure Machine Learning studio. The dataset **Details** page also provides sample code to access your labels from Python.

The screenshot shows the 'Details' page for a dataset named 'tutorial-cats-n-dogs2_20220527_174752'. The left sidebar is the navigation menu for the workspace, with the 'Data' item highlighted by a red box. The main content area displays the dataset's attributes, tags, description, and data sources. The 'Attributes' section includes properties like 'Tabular', created by 'Service Principal', and files in the dataset (1). The 'Tags' section lists labels such as 'labelingCreatedBy', 'labelingProjectType', and 'SourceDatastoreName'. The 'Description' section contains a detailed text about the dataset, and the 'Data sources' section is currently empty.

TIP

Once you have exported your labeled data to an Azure Machine Learning dataset, you can use AutoML to build computer vision models trained on your labeled data. Learn more at [Set up AutoML to train computer vision models with Python \(preview\)](#)

Explore labeled datasets via pandas dataframe

Load your labeled datasets into a pandas dataframe to leverage popular open-source libraries for data exploration with the `to_pandas_dataframe()` method from the `azureml-dataprep` class.

Install the class with the following shell command:

```
pip install azureml-dataprep
```

In the following code, the `animal_labels` dataset is the output from a labeling project previously saved to the workspace. The exported dataset is a `TabularDataset`. If you plan to use `download()` or `mount()` methods, be sure to set the parameter `stream column = 'image_url'`.

NOTE

The public preview methods `download()` and `mount()` are [experimental](#) preview features, and may change at any time.

APPLIES TO:  Python SDK azureml v1

```
import azureml.core
from azureml.core import Dataset, Workspace

# get animal_labels dataset from the workspace
animal_labels = Dataset.get_by_name(workspace, 'animal_labels')
animal_pd = animal_labels.to_pandas_dataframe()

# download the images to local
download_path = animal_labels.download(stream_column='image_url')

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

#read images from downloaded path
img = mpimg.imread(download_path[0])
imgplot = plt.imshow(img)
```

Next steps

- Learn to [train image classification models in Azure](#)
- [Set up AutoML to train computer vision models with Python \(preview\)](#)

Data ingestion with Azure Data Factory

9/21/2022 • 5 minutes to read • [Edit Online](#)

In this article, you learn about the available options for building a data ingestion pipeline with [Azure Data Factory](#). This Azure Data Factory pipeline is used to ingest data for use with [Azure Machine Learning](#). Data Factory allows you to easily extract, transform, and load (ETL) data. Once the data has been transformed and loaded into storage, it can be used to train your machine learning models in Azure Machine Learning.

Simple data transformation can be handled with native Data Factory activities and instruments such as [data flow](#). When it comes to more complicated scenarios, the data can be processed with some custom code. For example, Python or R code.

Compare Azure Data Factory data ingestion pipelines

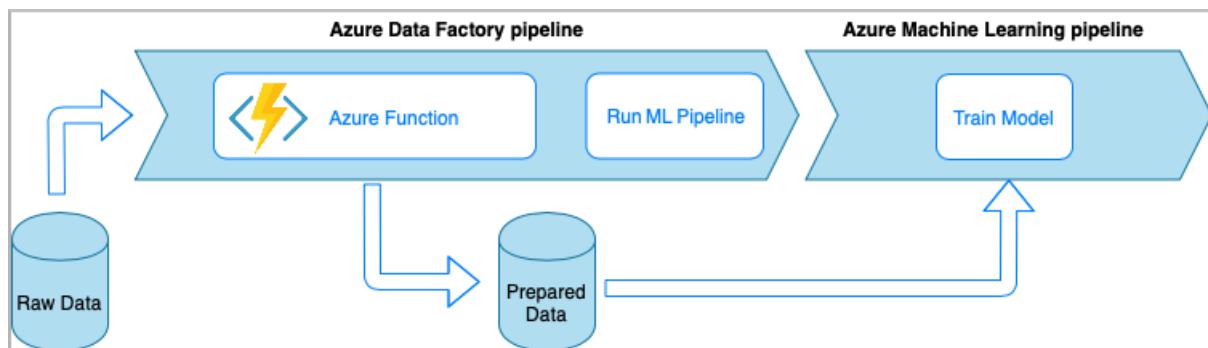
There are several common techniques of using Data Factory to transform data during ingestion. Each technique has advantages and disadvantages that help determine if it's a good fit for a specific use case:

TECHNIQUE	ADVANTAGES	DISADVANTAGES
Data Factory + Azure Functions	<ul style="list-style-type: none">• Low latency, serverless compute• Stateful functions• Reusable functions	Only good for short running processing
Data Factory + custom component	<ul style="list-style-type: none">• Large-scale parallel computing• Suited for heavy algorithms	<ul style="list-style-type: none">• Requires wrapping code into an executable• Complexity of handling dependencies and IO
Data Factory + Azure Databricks notebook	<ul style="list-style-type: none">• Apache Spark• Native Python environment	<ul style="list-style-type: none">• Can be expensive• Creating clusters initially takes time and adds latency

Azure Data Factory with Azure functions

Azure Functions allows you to run small pieces of code (functions) without worrying about application infrastructure. In this option, the data is processed with custom Python code wrapped into an Azure Function.

The function is invoked with the [Azure Data Factory Azure Function activity](#). This approach is a good option for lightweight data transformations.

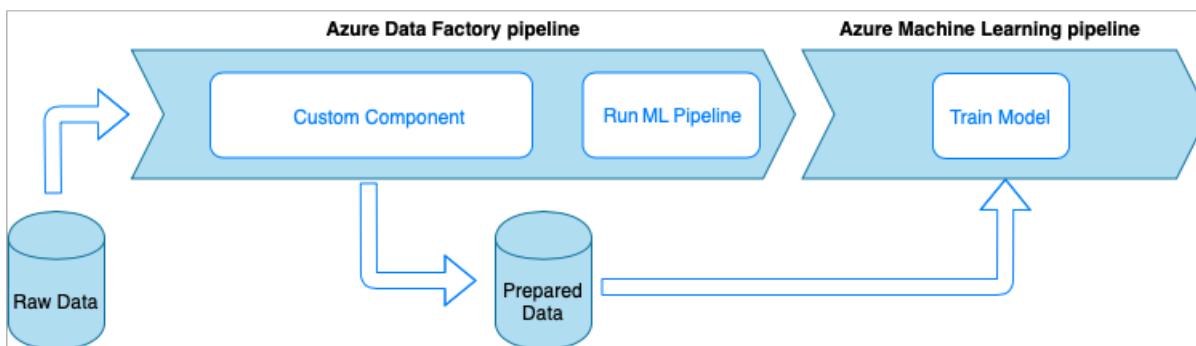


- Advantages:
 - The data is processed on a serverless compute with a relatively low latency

- Data Factory pipeline can invoke a [Durable Azure Function](#) that may implement a sophisticated data transformation flow
- The details of the data transformation are abstracted away by the Azure Function that can be reused and invoked from other places
- Disadvantages:
 - The Azure Functions must be created before use with ADF
 - Azure Functions is good only for short running data processing

Azure Data Factory with Custom Component activity

In this option, the data is processed with custom Python code wrapped into an executable. It's invoked with an [Azure Data Factory Custom Component activity](#). This approach is a better fit for large data than the previous technique.

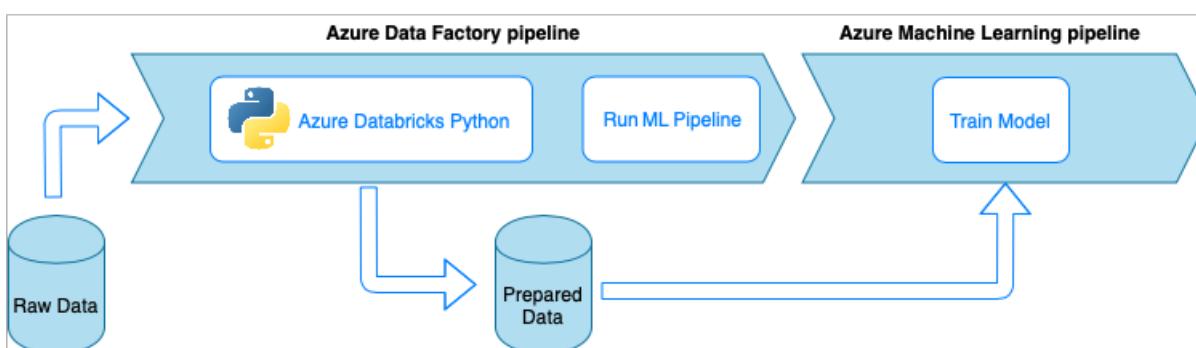


- Advantages:
 - The data is processed on [Azure Batch](#) pool, which provides large-scale parallel and high-performance computing
 - Can be used to run heavy algorithms and process significant amounts of data
- Disadvantages:
 - Azure Batch pool must be created before use with Data Factory
 - Over engineering related to wrapping Python code into an executable. Complexity of handling dependencies and input/output parameters

Azure Data Factory with Azure Databricks Python notebook

[Azure Databricks](#) is an Apache Spark-based analytics platform in the Microsoft cloud.

In this technique, the data transformation is performed by a [Python notebook](#), running on an Azure Databricks cluster. This is probably, the most common approach that uses the full power of an Azure Databricks service. It's designed for distributed data processing at scale.



- Advantages:
 - The data is transformed on the most powerful data processing Azure service, which is backed up by Apache Spark environment

- Native support of Python along with data science frameworks and libraries including TensorFlow, PyTorch, and scikit-learn
- There's no need to wrap the Python code into functions or executable modules. The code works as is.
- Disadvantages:
 - Azure Databricks infrastructure must be created before use with Data Factory
 - Can be expensive depending on Azure Databricks configuration
 - Spinning up compute clusters from "cold" mode takes some time that brings high latency to the solution

Consume data in Azure Machine Learning

The Data Factory pipeline saves the prepared data to your cloud storage (such as Azure Blob or Azure Data Lake).

Consume your prepared data in Azure Machine Learning by,

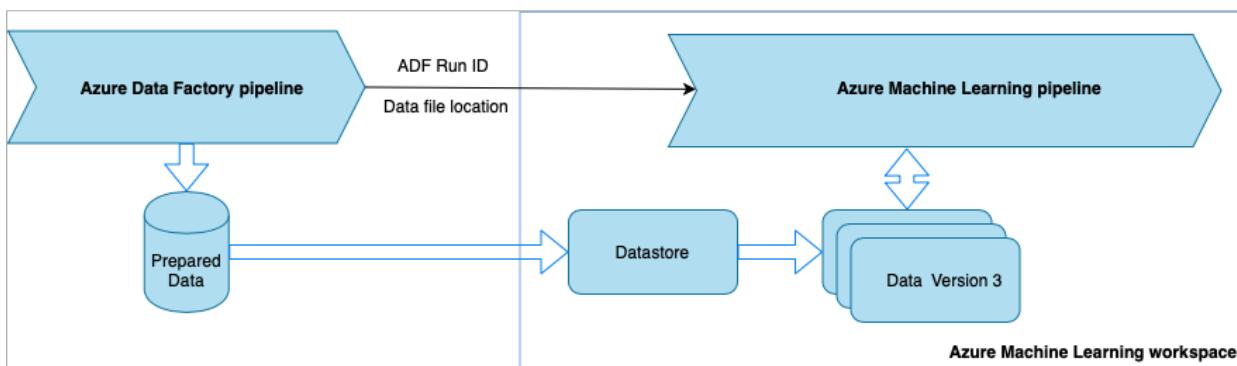
- Invoking an Azure Machine Learning pipeline from your Data Factory pipeline.
- OR
- Creating an [Azure Machine Learning datastore](#).

Invoke Azure Machine Learning pipeline from Data Factory

This method is recommended for [Machine Learning Operations \(MLOps\) workflows](#). If you don't want to set up an Azure Machine Learning pipeline, see [Read data directly from storage](#).

Each time the Data Factory pipeline runs,

1. The data is saved to a different location in storage.
 2. To pass the location to Azure Machine Learning, the Data Factory pipeline calls an [Azure Machine Learning pipeline](#). When calling the ML pipeline, the data location and job ID are sent as parameters.
 3. The ML pipeline can then create an Azure Machine Learning datastore and dataset with the data location.
- Learn more in [Execute Azure Machine Learning pipelines in Data Factory](#).



TIP

Datasets [support versioning](#), so the ML pipeline can register a new version of the dataset that points to the most recent data from the ADF pipeline.

Once the data is accessible through a datastore or dataset, you can use it to train an ML model. The training process might be part of the same ML pipeline that is called from ADF. Or it might be a separate process such as experimentation in a Jupyter notebook.

Since datasets support versioning, and each job from the pipeline creates a new version, it's easy to understand which version of the data was used to train a model.

[Read data directly from storage](#)

If you don't want to create an ML pipeline, you can access the data directly from the storage account where your prepared data is saved with an Azure Machine Learning datastore and dataset.

The following Python code demonstrates how to create a datastore that connects to Azure DataLake Generation 2 storage. [Learn more about datastores and where to find service principal permissions](#).

APPLIES TO:  Python SDK azureml v1

```
ws = Workspace.from_config()
adlsgen2_datastore_name = '<ADLS gen2 storage account alias>' #set ADLS Gen2 storage account alias in
AzureML

subscription_id=os.getenv("ADL_SUBSCRIPTION", "<ADLS account subscription ID>") # subscription id of ADLS
account
resource_group=os.getenv("ADL_RESOURCE_GROUP", "<ADLS account resource group>") # resource group of ADLS
account

account_name=os.getenv("ADLSEGEN2_ACCOUNTNAME", "<ADLS account name>") # ADLS Gen2 account name
tenant_id=os.getenv("ADLSEGEN2_TENANT", "<tenant id of service principal>") # tenant id of service principal
client_id=os.getenv("ADLSEGEN2_CLIENTID", "<client id of service principal>") # client id of service
principal
client_secret=os.getenv("ADLSEGEN2_CLIENT_SECRET", "<secret of service principal>") # the secret of service
principal

adlsgen2_datastore = Datastore.register_azure_data_lake_gen2(
    workspace=ws,
    datastore_name=adlsgen2_datastore_name,
    account_name=account_name, # ADLS Gen2 account name
    filesystem='<filesystem name>', # ADLS Gen2 filesystem
    tenant_id=tenant_id, # tenant id of service principal
    client_id=client_id, # client id of service principal
```

Next, create a dataset to reference the file(s) you want to use in your machine learning task.

The following code creates a TabularDataset from a csv file, `prepared-data.csv`. Learn more about [dataset types and accepted file formats](#).

APPLIES TO:  Python SDK azureml v1

```
from azureml.core import Workspace, Datastore, Dataset
from azureml.core.experiment import Experiment
from azureml.train.automl import AutoMLConfig

# retrieve data via AzureML datastore
datastore = Datastore.get(ws, adlsgen2_datastore)
datastore_path = [(datastore, '/data/prepared-data.csv')]

prepared_dataset = Dataset.Tabular.from_delimited_files(path=datastore_path)
```

From here, use `prepared_dataset` to reference your prepared data, like in your training scripts. Learn how to [Train models with datasets in Azure Machine Learning](#).

Next steps

- [Run a Databricks notebook in Azure Data Factory](#)
- [Access data in Azure storage services](#)
- [Train models with datasets in Azure Machine Learning](#).
- [DevOps for a data ingestion pipeline](#)

Data wrangling with Apache Spark pools (preview)

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK `azureml` v1

In this article, you learn how to perform data wrangling tasks interactively within a dedicated Synapse session, powered by [Azure Synapse Analytics](#), in a Jupyter notebook using the [Azure Machine Learning Python SDK](#).

If you prefer to use Azure Machine Learning pipelines, see [How to use Apache Spark \(powered by Azure Synapse Analytics\) in your machine learning pipeline \(preview\)](#).

For guidance on how to use Azure Synapse Analytics with a Synapse workspace, see the [Azure Synapse Analytics get started series](#).

IMPORTANT

The Azure Machine Learning and Azure Synapse Analytics integration is in preview. The capabilities presented in this article employ the `azureml-synapse` package which contains [experimental](#) preview features that may change at any time.

Azure Machine Learning and Azure Synapse Analytics integration

The Azure Synapse Analytics integration with Azure Machine Learning (preview) allows you to attach an Apache Spark pool backed by Azure Synapse for interactive data exploration and preparation. With this integration, you can have a dedicated compute for data wrangling at scale, all within the same Python notebook you use for training your machine learning models.

Prerequisites

- The [Azure Machine Learning Python SDK installed](#).
- [Create an Azure Machine Learning workspace](#).
- [Create an Azure Synapse Analytics workspace in Azure portal](#).
- [Create Apache Spark pool using Azure portal, web tools, or Synapse Studio](#).
- [Configure your development environment](#) to install the Azure Machine Learning SDK, or use an [Azure Machine Learning compute instance](#) with the SDK already installed.
- Install the `azureml-synapse` package (preview) with the following code:

```
pip install azureml-synapse
```

- Link your Azure Machine Learning workspace and Azure Synapse Analytics workspace with the [Azure Machine Learning Python SDK](#) or via the [Azure Machine Learning studio](#)
- [Attach a Synapse Spark pool](#) as a compute target.

Launch Synapse Spark pool for data wrangling tasks

To begin data preparation with the Apache Spark pool, specify the attached Spark Synapse compute name. This

name can be found via the Azure Machine Learning studio under the **Attached computes** tab.

The screenshot shows the Azure Machine Learning studio interface. On the left, there's a sidebar with various options like New, Home, Notebooks, Automated ML, Designer, Assets, Datasets, Experiments, Modules, Pipelines, Models, Endpoints, and Manage. Under Manage, the 'Compute' option is selected and highlighted with a red box. The main area is titled 'Compute' and shows tabs for Compute instances, Compute clusters, Inference clusters, and Attached computes. The Attached computes tab is also highlighted with a red box. Below these tabs are buttons for '+ New', 'Detach', 'Refresh', 'Edit columns', and 'Reset view'. A search bar is present. At the bottom, there's a table with columns: Name, State, Type, Created on, and Active runs. One row is shown: 'abespark', 'Succeeded', 'Synapse Spark pool (preview)', 'May 14, 2021 10:22 AM', and '0'.

IMPORTANT

To continue use of the Apache Spark pool you must indicate which compute resource to use throughout your data wrangling tasks with `%synapse` for single lines of code and `%%synapse` for multiple lines. [Learn more about the %synapse magic command.](#)

```
%synapse start -c SynapseSparkPoolAlias
```

After the session starts, you can check the session's metadata.

```
%synapse meta
```

You can specify an [Azure Machine Learning environment](#) to use during your Apache Spark session. Only Conda dependencies specified in the environment will take effect. Docker image isn't supported.

WARNING

Python dependencies specified in environment Conda dependencies are not supported in Apache Spark pools. Currently, only fixed Python versions are supported. Check your Python version by including `sys.version_info` in your script.

The following code, creates the environment, `myenv`, which installs `azureml-core` version 1.20.0 and `numpy` version 1.17.0 before the session begins. You can then include this environment in your Apache Spark session `start` statement.

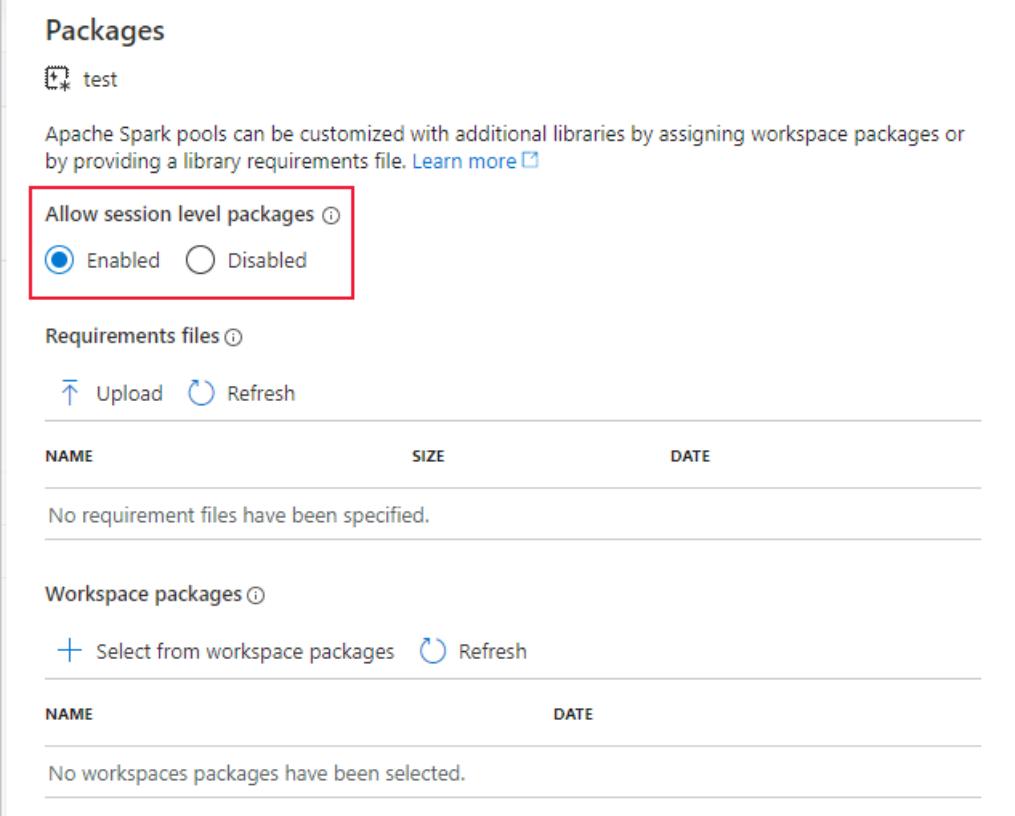
```
from azureml.core import Workspace, Environment

# creates environment with numpy and azureml-core dependencies
ws = Workspace.from_config()
env = Environment(name="myenv")
env.python.conda_dependencies.add_pip_package("azureml-core==1.20.0")
env.python.conda_dependencies.add_conda_package("numpy==1.17.0")
env.register(workspace=ws)
```

To begin data preparation with the Apache Spark pool and your custom environment, specify the Apache Spark pool name and which environment to use during the Apache Spark session. Furthermore, you can provide your subscription ID, the machine learning workspace resource group, and the name of the machine learning workspace.

IMPORTANT

Make sure to [Allow session level packages](#) is enabled in the linked Synapse workspace.



Packages

Apache Spark pools can be customized with additional libraries by assigning workspace packages or by providing a library requirements file. [Learn more](#)

Allow session level packages ⓘ

Enabled Disabled

Requirements files ⓘ

Upload Refresh

NAME	SIZE	DATE
No requirement files have been specified.		

Workspace packages ⓘ

Select from workspace packages Refresh

NAME	DATE
No workspaces packages have been selected.	

```
%synapse start -c SynapseSparkPoolAlias -e myenv -s AzureMLworkspaceSubscriptionID -r AzureMLworkspaceResourceGroupName -w AzureMLworkspaceName
```

Load data from storage

Once your Apache Spark session starts, read in the data that you wish to prepare. Data loading is supported for Azure Blob storage and Azure Data Lake Storage Generations 1 and 2.

There are two ways to load data from these storage services:

- Directly load data from storage using its Hadoop Distributed Files System (HDFS) path.
- Read in data from an existing [Azure Machine Learning dataset](#).

To access these storage services, you need **Storage Blob Data Reader** permissions. If you plan to write data back to these storage services, you need **Storage Blob Data Contributor** permissions. [Learn more about storage permissions and roles](#).

Load data with Hadoop Distributed Files System (HDFS) path

To load and read data in from storage with the corresponding HDFS path, you need to have your data access authentication credentials readily available. These credentials differ depending on your storage type.

The following code demonstrates how to read data from an **Azure Blob storage** into a Spark dataframe with

either your shared access signature (SAS) token or access key.

```
%%synapse

# setup access key or SAS token
sc._jsc.hadoopConfiguration().set("fs.azure.account.key.<storage account name>.blob.core.windows.net", "<access key>")
sc._jsc.hadoopConfiguration().set("fs.azure.sas.<container name>.<storage account name>.blob.core.windows.net", "<sas token>")

# read from blob
df = spark.read.option("header", "true").csv("wasbs://demo@dprepdata.blob.core.windows.net/Titanic.csv")
```

The following code demonstrates how to read data in from **Azure Data Lake Storage Generation 1 (ADLS Gen 1)** with your service principal credentials.

```
%%synapse

# setup service principal which has access of the data
sc._jsc.hadoopConfiguration().set("fs.adl.account.<storage account name>.oauth2.access.token.provider.type", "ClientCredential")

sc._jsc.hadoopConfiguration().set("fs.adl.account.<storage account name>.oauth2.client.id", "<client id>")

sc._jsc.hadoopConfiguration().set("fs.adl.account.<storage account name>.oauth2.credential", "<client secret>")

sc._jsc.hadoopConfiguration().set("fs.adl.account.<storage account name>.oauth2.refresh.url",
"https://login.microsoftonline.com/<tenant id>/oauth2/token")

df = spark.read.csv("adl://<storage account name>.azuredatalakestore.net/<path>")
```

The following code demonstrates how to read data in from **Azure Data Lake Storage Generation 2 (ADLS Gen 2)** with your service principal credentials.

```
%%synapse

# setup service principal which has access of the data
sc._jsc.hadoopConfiguration().set("fs.azure.account.auth.type.<storage account name>.dfs.core.windows.net", "OAuth")
sc._jsc.hadoopConfiguration().set("fs.azure.account.oauth.provider.type.<storage account name>.dfs.core.windows.net", "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider")
sc._jsc.hadoopConfiguration().set("fs.azure.account.oauth2.client.id.<storage account name>.dfs.core.windows.net", "<client id>")
sc._jsc.hadoopConfiguration().set("fs.azure.account.oauth2.client.secret.<storage account name>.dfs.core.windows.net", "<client secret>")
sc._jsc.hadoopConfiguration().set("fs.azure.account.oauth2.client.endpoint.<storage account name>.dfs.core.windows.net",
"https://login.microsoftonline.com/<tenant id>/oauth2/token")

df = spark.read.csv("abfss://<container name>@<storage account>.dfs.core.windows.net/<path>")
```

Read in data from registered datasets

You can also get an existing registered dataset in your workspace and perform data preparation on it by converting it into a spark dataframe.

The following example authenticates to the workspace, gets a registered TabularDataset, `blob_dset`, that references files in blob storage, and converts it into a spark dataframe. When you convert your datasets into a

spark dataframe, you can use `pyspark` data exploration and preparation libraries.

```
%%synapse

from azureml.core import Workspace, Dataset

subscription_id = "<enter your subscription ID>"
resource_group = "<enter your resource group>"
workspace_name = "<enter your workspace name>

ws = Workspace(workspace_name = workspace_name,
               subscription_id = subscription_id,
               resource_group = resource_group)

dset = Dataset.get_by_name(ws, "blob_dset")
spark_df = dset.to_spark_dataframe()
```

Perform data wrangling tasks

After you've retrieved and explored your data, you can perform data wrangling tasks.

The following code, expands upon the HDFS example in the previous section and filters the data in spark dataframe, `df`, based on the **Survivor** column and groups that list by **Age**

```
%%synapse

from pyspark.sql.functions import col, desc

df.filter(col('Survived') == 1).groupBy('Age').count().orderBy(desc('count')).show(10)

df.show()
```

Save data to storage and stop spark session

Once your data exploration and preparation is complete, store your prepared data for later use in your storage account on Azure.

In the following example, the prepared data is written back to Azure Blob storage and overwrites the original `Titanic.csv` file in the `training_data` directory. To write back to storage, you need **Storage Blob Data Contributor** permissions. [Learn more about storage permissions and roles](#).

```
%% synapse

df.write.format("csv").mode("overwrite").save("wasbs://demo@dprepdata.blob.core.windows.net/training_data/Titanic.csv")
```

When you've completed data preparation and saved your prepared data to storage, stop using your Apache Spark pool with the following command.

```
%synapse stop
```

Create dataset to represent prepared data

When you're ready to consume your prepared data for model training, connect to your storage with an [Azure Machine Learning datastore](#), and specify which file(s) you want to use with an [Azure Machine Learning dataset](#).

The following code example,

- Assumes you already created a datastore that connects to the storage service where you saved your prepared data.
- Gets that existing datastore, `mydatastore`, from the workspace, `ws` with the `get()` method.
- Creates a `FileDataset`, `train_ds`, that references the prepared data files located in the `training_data` directory in `mydatastore`.
- Creates the variable `input1`, which can be used at a later time to make the data files of the `train_ds` dataset available to a compute target for your training tasks.

```
from azureml.core import Datastore, Dataset

datastore = Datastore.get(ws, datastore_name='mydatastore')

datastore_paths = [(datastore, '/training_data/')]
train_ds = Dataset.File.from_files(path=datastore_paths, validate=True)
input1 = train_ds.as_mount()
```

Use a `ScriptRunConfig` to submit an experiment run to a Synapse Spark pool

If you're ready to automate and productionize your data wrangling tasks, you can submit an experiment run to an attached [Synapse Spark pool](#) with the `ScriptRunConfig` object.

Similarly, if you have an Azure Machine Learning pipeline, you can use the [SynapseSparkStep](#) to specify your [Synapse Spark pool as the compute target](#) for the data preparation step in your pipeline.

Making your data available to the Synapse Spark pool depends on your dataset type.

- For a `FileDataset`, you can use the `as_hdfs()` method. When the run is submitted, the dataset is made available to the Synapse Spark pool as a Hadoop distributed file system (HDFS).
- For a `TabularDataset`, you can use the `as_named_input()` method.

The following code,

- Creates the variable `input2` from the `FileDataset` `train_ds` that was created in the previous code example.
- Creates the variable `output` with the `HDFSOutputDatasetConfiguration` class. After the run is complete, this class allows us to save the output of the run as the dataset, `test` in the datastore, `mydatastore`. In the Azure Machine Learning workspace, the `test` dataset is registered under the name `registered_dataset`.
- Configures settings the run should use in order to perform on the Synapse Spark pool.
- Defines the `ScriptRunConfig` parameters to,
 - Use the `dataprep.py`, for the run.
 - Specify which data to use as input and how to make it available to the Synapse Spark pool.
 - Specify where to store output data, `output`.

```

from azureml.core import Dataset, HDFSOutputDatasetConfig
from azureml.core.environment import CondaDependencies
from azureml.core import RunConfiguration
from azureml.core import ScriptRunConfig
from azureml.core import Experiment

input2 = train_ds.as_hdbsql()
output = HDFSOutputDatasetConfig(destination=(datastore,
"test").register_on_complete(name="registered_dataset"))

run_config = RunConfiguration(framework="pyspark")
run_config.target = synapse_compute_name

run_config.spark.configuration["spark.driver.memory"] = "1g"
run_config.spark.configuration["spark.driver.cores"] = 2
run_config.spark.configuration["spark.executor.memory"] = "1g"
run_config.spark.configuration["spark.executor.cores"] = 1
run_config.spark.configuration["spark.executor.instances"] = 1

conda_dep = CondaDependencies()
conda_dep.add_pip_package("azureml-core==1.20.0")

run_config.environment.python.conda_dependencies = conda_dep

script_run_config = ScriptRunConfig(source_directory = './code',
                                    script= 'dataprep.py',
                                    arguments = ["--file_input", input2,
                                                "--output_dir", output],
                                    run_config = run_config)

```

For more information about `run_config.spark.configuration` and general Spark configuration, see [SparkConfiguration Class](#) and [Apache Spark's configuration documentation](#).

Once your `ScriptRunConfig` object is set up, you can submit the run.

```

from azureml.core import Experiment

exp = Experiment(workspace=ws, name="synapse-spark")
run = exp.submit(config=script_run_config)
run

```

For more details, like the `dataprep.py` script used in this example, see the [example notebook](#).

After your data is prepared, you can then use it as input for your training jobs. In the aforementioned code example, the `registered_dataset` is what you would specify as your input data for training jobs.

Example notebooks

See the example notebooks for more concepts and demonstrations of the Azure Synapse Analytics and Azure Machine Learning integration capabilities.

- [Run an interactive Spark session from a notebook in your Azure Machine Learning workspace.](#)
- [Submit an Azure Machine Learning experiment run with a Synapse Spark pool as your compute target.](#)

Next steps

- [Train a model.](#)
- [Train with Azure Machine Learning dataset.](#)

DevOps for a data ingestion pipeline

9/21/2022 • 11 minutes to read • [Edit Online](#)

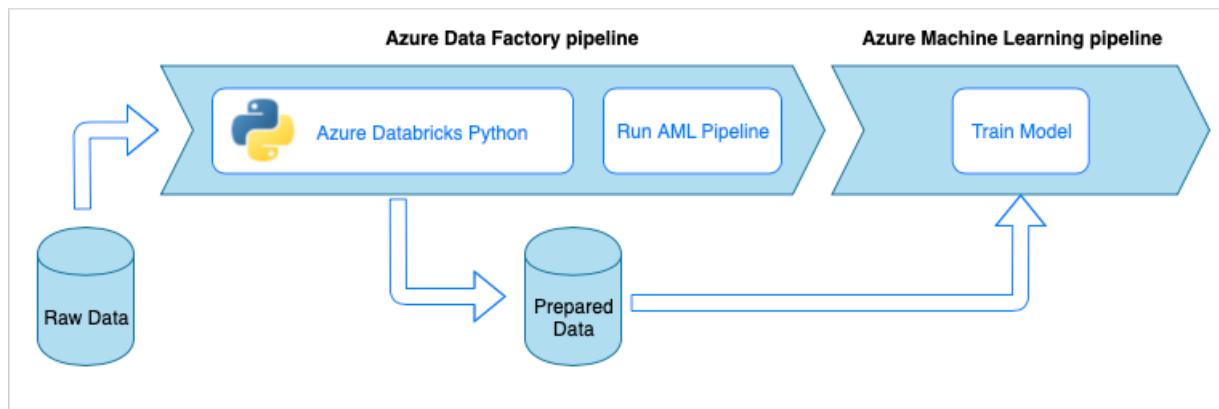
In most scenarios, a data ingestion solution is a composition of scripts, service invocations, and a pipeline orchestrating all the activities. In this article, you learn how to apply DevOps practices to the development lifecycle of a common data ingestion pipeline that prepares data for machine learning model training. The pipeline is built using the following Azure services:

- **Azure Data Factory**: Reads the raw data and orchestrates data preparation.
- **Azure Databricks**: Runs a Python notebook that transforms the data.
- **Azure Pipelines**: Automates a continuous integration and development process.

Data ingestion pipeline workflow

The data ingestion pipeline implements the following workflow:

1. Raw data is read into an Azure Data Factory (ADF) pipeline.
2. The ADF pipeline sends the data to an Azure Databricks cluster, which runs a Python notebook to transform the data.
3. The data is stored to a blob container, where it can be used by Azure Machine Learning to train a model.



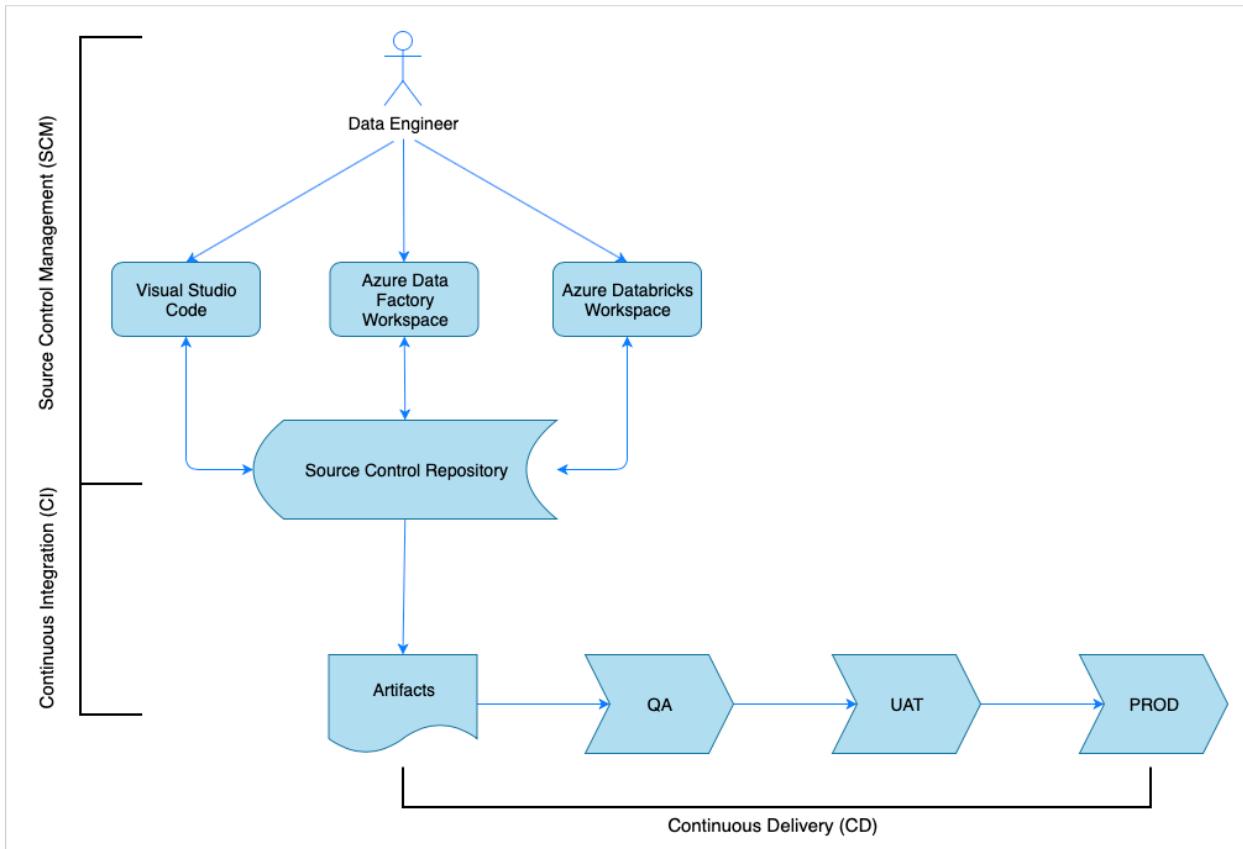
Continuous integration and delivery overview

As with many software solutions, there is a team (for example, Data Engineers) working on it. They collaborate and share the same Azure resources such as Azure Data Factory, Azure Databricks, and Azure Storage accounts. The collection of these resources is a Development environment. The data engineers contribute to the same source code base.

A continuous integration and delivery system automates the process of building, testing, and delivering (deploying) the solution. The Continuous Integration (CI) process performs the following tasks:

- Assembles the code
- Checks it with the code quality tests
- Runs unit tests
- Produces artifacts such as tested code and Azure Resource Manager templates

The Continuous Delivery (CD) process deploys the artifacts to the downstream environments.



This article demonstrates how to automate the CI and CD processes with [Azure Pipelines](#).

Source control management

Source control management is needed to track changes and enable collaboration between team members. For example, the code would be stored in an Azure DevOps, GitHub, or GitLab repository. The collaboration workflow is based on a branching model. For example, [GitFlow](#).

Python Notebook Source Code

The data engineers work with the Python notebook source code either locally in an IDE (for example, [Visual Studio Code](#)) or directly in the Databricks workspace. Once the code changes are complete, they are merged to the repository following a branching policy.

TIP

We recommend storing the code in `.py` files rather than in `.ipynb` Jupyter Notebook format. It improves the code readability and enables automatic code quality checks in the CI process.

Azure Data Factory Source Code

The source code of Azure Data Factory pipelines is a collection of JSON files generated by an Azure Data Factory workspace. Normally the data engineers work with a visual designer in the Azure Data Factory workspace rather than with the source code files directly.

To configure the workspace to use a source control repository, see [Author with Azure Repos Git integration](#).

Continuous integration (CI)

The ultimate goal of the Continuous Integration process is to gather the joint team work from the source code and prepare it for the deployment to the downstream environments. As with the source code management this process is different for the Python notebooks and Azure Data Factory pipelines.

Python Notebook CI

The CI process for the Python Notebooks gets the code from the collaboration branch (for example, *master* or *develop*) and performs the following activities:

- Code linting
- Unit testing
- Saving the code as an artifact

The following code snippet demonstrates the implementation of these steps in an Azure DevOps *yaml* pipeline:

```
steps:  
- script: |  
  flake8 --output-file=$(Build.BinariesDirectory)/lint-testresults.xml --format junit-xml  
  workingDirectory: '$(Build.SourcesDirectory)'  
  displayName: 'Run flake8 (code style analysis)'  
  
- script: |  
  python -m pytest --junitxml=$(Build.BinariesDirectory)/unit-testresults.xml $(Build.SourcesDirectory)  
  displayName: 'Run unit tests'  
  
- task: PublishTestResults@2  
  condition: succeededOrFailed()  
  inputs:  
    testResultsFiles: '$(Build.BinariesDirectory)/*-testresults.xml'  
    testRunTitle: 'Linting & Unit tests'  
    failTaskOnFailedTests: true  
  displayName: 'Publish linting and unit test results'  
  
- publish: $(Build.SourcesDirectory)  
  artifact: di-notebooks
```

The pipeline uses [flake8](#) to do the Python code linting. It runs the unit tests defined in the source code and publishes the linting and test results so they're available in the Azure Pipelines execution screen.

If the linting and unit testing is successful, the pipeline will copy the source code to the artifact repository to be used by the subsequent deployment steps.

Azure Data Factory CI

CI process for an Azure Data Factory pipeline is a bottleneck for a data ingestion pipeline. There's no continuous integration. A deployable artifact for Azure Data Factory is a collection of Azure Resource Manager templates. The only way to produce those templates is to click the *publish* button in the Azure Data Factory workspace.

1. The data engineers merge the source code from their feature branches into the collaboration branch, for example, *master* or *develop*.
2. Someone with the granted permissions clicks the *publish* button to generate Azure Resource Manager templates from the source code in the collaboration branch.
3. The workspace validates the pipelines (think of it as of linting and unit testing), generates Azure Resource Manager templates (think of it as of building) and saves the generated templates to a technical branch *adf_publish* in the same code repository (think of it as of publishing artifacts). This branch is created automatically by the Azure Data Factory workspace.

For more information on this process, see [Continuous integration and delivery in Azure Data Factory](#).

It's important to make sure that the generated Azure Resource Manager templates are environment agnostic. This means that all values that may differ between environments are parametrized. Azure Data Factory is smart enough to expose the majority of such values as parameters. For example, in the following template the connection properties to an Azure Machine Learning workspace are exposed as parameters:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "factoryName": {
            "value": "devops-ds-adf"
        },
        "AzureMLService_servicePrincipalKey": {
            "value": ""
        },
        "AzureMLService_properties_typeProperties_subscriptionId": {
            "value": "0fe1c235-5cfa-4152-17d7-5dff45a8d4ba"
        },
        "AzureMLService_properties_typeProperties_resourceGroupName": {
            "value": "devops-ds-rg"
        },
        "AzureMLService_properties_typeProperties_servicePrincipalId": {
            "value": "6e35e589-3b22-4edb-89d0-2ab7fc08d488"
        },
        "AzureMLService_properties_typeProperties_tenant": {
            "value": "72f988bf-86f1-41af-912b-2d7cd611db47"
        }
    }
}
```

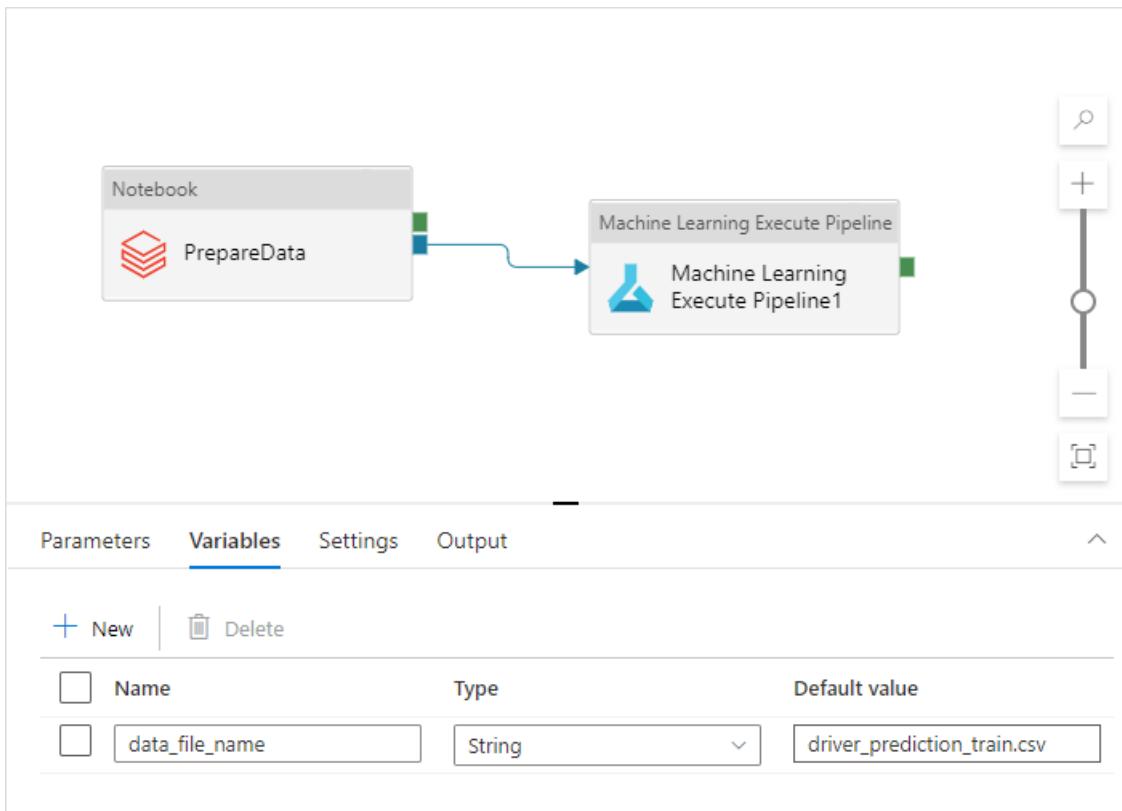
However, you may want to expose your custom properties that are not handled by the Azure Data Factory workspace by default. In the scenario of this article an Azure Data Factory pipeline invokes a Python notebook processing the data. The notebook accepts a parameter with the name of an input data file.

```
import pandas as pd
import numpy as np

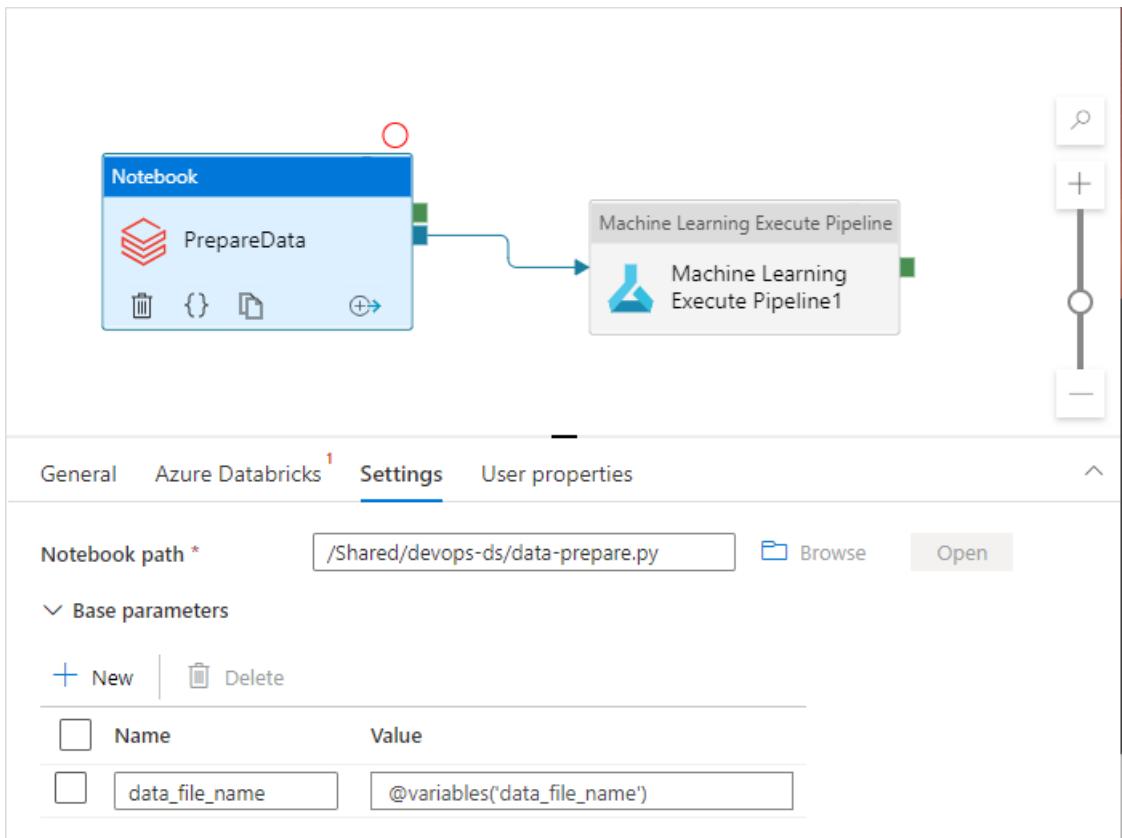
data_file_name = getArgument("data_file_name")
data = pd.read_csv(data_file_name)

labels = np.array(data['target'])
...
```

This name is different for *Dev*, *QA*, *UAT*, and *PROD* environments. In a complex pipeline with multiple activities, there can be several custom properties. It's good practice to collect all those values in one place and define them as pipeline *variables*:



The pipeline activities may refer to the pipeline variables while actually using them:



The Azure Data Factory workspace *doesn't* expose pipeline variables as Azure Resource Manager templates parameters by default. The workspace uses the [Default Parameterization Template](#) dictating what pipeline properties should be exposed as Azure Resource Manager template parameters. To add pipeline variables to the list, update the `"Microsoft.DataFactory/factories/pipelines"` section of the [Default Parameterization Template](#) with the following snippet and place the result.json file in the root of the source folder:

```
"Microsoft.DataFactory/factories/pipelines": {  
    "properties": {  
        "variables": {  
            "*": {  
                "defaultValue": "="  
            }  
        }  
    }  
}
```

Doing so will force the Azure Data Factory workspace to add the variables to the parameters list when the **publish** button is clicked:

```
{  
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",  
    "contentVersion": "1.0.0.0",  
    "parameters": {  
        "factoryName": {  
            "value": "devops-ds-adf"  
        },  
        ...  
        "data-ingestion-pipeline_properties_variables_data_file_nameDefaultValue": {  
            "value": "driver_prediction_train.csv"  
        }  
    }  
}
```

The values in the JSON file are default values configured in the pipeline definition. They're expected to be overridden with the target environment values when the Azure Resource Manager template is deployed.

Continuous delivery (CD)

The Continuous Delivery process takes the artifacts and deploys them to the first target environment. It makes sure that the solution works by running tests. If successful, it continues to the next environment.

The CD Azure Pipelines consists of multiple stages representing the environments. Each stage contains [deployments](#) and [jobs](#) that perform the following steps:

- Deploy a Python Notebook to Azure Databricks workspace
- Deploy an Azure Data Factory pipeline
- Run the pipeline
- Check the data ingestion result

The pipeline stages can be configured with [approvals](#) and [gates](#) that provide additional control on how the deployment process evolves through the chain of environments.

Deploy a Python Notebook

The following code snippet defines an Azure Pipeline [deployment](#) that copies a Python notebook to a Databricks cluster:

```

- stage: 'Deploy_to_QA'
  displayName: 'Deploy to QA'
  variables:
    - group: devops-ds-qa-vg
  jobs:
    - deployment: "Deploy_to_Databricks"
      displayName: 'Deploy to Databricks'
      timeoutInMinutes: 0
      environment: qa
      strategy:
        runOnce:
          deploy:
            steps:
              - task: UsePythonVersion@0
                inputs:
                  versionSpec: '3.x'
                  addToPath: true
                  architecture: 'x64'
                displayName: 'Use Python3'

              - task: configuredatabricks@0
                inputs:
                  url: '$(DATABRICKS_URL)'
                  token: '$(DATABRICKS_TOKEN)'
                displayName: 'Configure Databricks CLI'

              - task: deploynotebooks@0
                inputs:
                  notebooksFolderPath: '$(Pipeline.Workspace)/di-notebooks'
                  workspaceFolder: '/Shared/devops-ds'
                displayName: 'Deploy (copy) data processing notebook to the Databricks cluster'

```

The artifacts produced by the CI are automatically copied to the deployment agent and are available in the `$(Pipeline.Workspace)` folder. In this case, the deployment task refers to the `di-notebooks` artifact containing the Python notebook. This [deployment](#) uses the [Databricks Azure DevOps extension](#) to copy the notebook files to the Databricks workspace.

The `Deploy_to_QA` stage contains a reference to the `devops-ds-qa-vg` variable group defined in the Azure DevOps project. The steps in this stage refer to the variables from this variable group (for example, `$(DATABRICKS_URL)` and `$(DATABRICKS_TOKEN)`). The idea is that the next stage (for example, `Deploy_to_UAT`) will operate with the same variable names defined in its own UAT-scoped variable group.

Deploy an Azure Data Factory pipeline

A deployable artifact for Azure Data Factory is an Azure Resource Manager template. It's going to be deployed with the [*Azure Resource Group Deployment*](#) task as it is demonstrated in the following snippet:

```
- deployment: "Deploy_to_ADF"
  displayName: 'Deploy to ADF'
  timeoutInMinutes: 0
  environment: qa
  strategy:
    runOnce:
      deploy:
        steps:
          - task: AzureResourceGroupDeployment@2
            displayName: 'Deploy ADF resources'
            inputs:
              azureSubscription: $(AZURE_RM_CONNECTION)
              resourceGroupName: $(RESOURCE_GROUP)
              location: $(LOCATION)
              csmFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateForFactory.json'
              csmParametersFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateParametersForFactory.json'
              overrideParameters: -data-ingestion-
            pipeline_properties_variables_data_file_nameDefaultValue "$(DATA_FILE_NAME)"
```

The value of the data filename parameter comes from the `$(DATA_FILE_NAME)` variable defined in a QA stage variable group. Similarly, all parameters defined in *ARMTemplateForFactory.json* can be overridden. If they are not, then the default values are used.

Run the pipeline and check the data ingestion result

The next step is to make sure that the deployed solution is working. The following job definition runs an Azure Data Factory pipeline with a [PowerShell script](#) and executes a Python notebook on an Azure Databricks cluster. The notebook checks if the data has been ingested correctly and validates the result data file with `$(bin_FILE_NAME)` name.

```

- job: "Integration_test_job"
  displayName: "Integration test job"
  dependsOn: [Deploy_to_Databricks, Deploy_to_ADF]
  pool:
    vmImage: 'ubuntu-latest'
  timeoutInMinutes: 0
  steps:
    - task: AzurePowerShell@4
      displayName: 'Execute ADF Pipeline'
      inputs:
        azureSubscription: $(AZURE_RM_CONNECTION)
        ScriptPath: '$(Build.SourcesDirectory)/adf/utils/Invoke-ADFPipeline.ps1'
        ScriptArguments: '-ResourceGroupName $(RESOURCE_GROUP) -DataFactoryName $(DATA_FACTORY_NAME) -PipelineName $(PIPELINE_NAME)'
        azurePowerShellVersion: LatestVersion
    - task: UsePythonVersion@0
      inputs:
        versionSpec: '3.x'
        addToPath: true
        architecture: 'x64'
      displayName: 'Use Python3'

    - task: configuredatabricks@0
      inputs:
        url: '$(DATABRICKS_URL)'
        token: '$(DATABRICKS_TOKEN)'
      displayName: 'Configure Databricks CLI'

    - task: executenotebook@0
      inputs:
        notebookPath: '/Shared/devops-ds/test-data-ingestion'
        existingClusterId: '$(DATABRICKS_CLUSTER_ID)'
        executionParams: '{"bin_file_name":"'$(bin_FILE_NAME)'"}'
      displayName: 'Test data ingestion'

    - task: waitexecution@0
      displayName: 'Wait until the testing is done'

```

The final task in the job checks the result of the notebook execution. If it returns an error, it sets the status of pipeline execution to failed.

Putting pieces together

The complete CI/CD Azure Pipeline consists of the following stages:

- CI
- Deploy To QA
 - Deploy to Databricks + Deploy to ADF
 - Integration Test

It contains a number of ***Deploy*** stages equal to the number of target environments you have. Each ***Deploy*** stage contains two **deployments** that run in parallel and a **job** that runs after deployments to test the solution on the environment.

A sample implementation of the pipeline is assembled in the following ***yaml*** snippet:

```

variables:
- group: devops-ds-vg

stages:
- stage: 'CI'
  displayName: 'CI'
  ...

```

```

jobs:
- job: "CI_Job"
  displayName: "CI Job"
  pool:
    vmImage: 'ubuntu-latest'
  timeoutInMinutes: 0
  steps:
- task: UsePythonVersion@0
  inputs:
    versionSpec: '3.x'
    addToPath: true
    architecture: 'x64'
  displayName: 'Use Python3'
- script: pip install --upgrade flake8 flake8_formatter_junit_xml
  displayName: 'Install flake8'
- checkout: self
- script: |
  flake8 --output-file=$(Build.BinariesDirectory)/lint-testresults.xml --format junit-xml
  workingDirectory: '$(Build.SourcesDirectory)'
  displayName: 'Run flake8 (code style analysis)'
- script: |
  python -m pytest --junitxml=$(Build.BinariesDirectory)/unit-testresults.xml $(Build.SourcesDirectory)
  displayName: 'Run unit tests'
- task: PublishTestResults@2
  condition: succeededOrFailed()
  inputs:
    testResultsFiles: '$(Build.BinariesDirectory)/*-testresults.xml'
    testRunTitle: 'Linting & Unit tests'
    failTaskOnFailedTests: true
  displayName: 'Publish linting and unit test results'

# The CI stage produces two artifacts (notebooks and ADF pipelines).
# The pipelines Azure Resource Manager templates are stored in a technical branch "adf_publish"
- publish: $(Build.SourcesDirectory)/$(Build.Repository.Name)/code/dataingestion
  artifact: di-notebooks
- checkout: git://${{variables['System.TeamProject']} }@adf_publish
- publish: $(Build.SourcesDirectory)/$(Build.Repository.Name)/devops-ds-adf
  artifact: adf-pipelines

- stage: 'Deploy_to_QA'
  displayName: 'Deploy to QA'
  variables:
  - group: devops-ds-qa-vg
  jobs:
  - deployment: "Deploy_to_Databricks"
    displayName: 'Deploy to Databricks'
    timeoutInMinutes: 0
    environment: qa
    strategy:
      runOnce:
        deploy:
          steps:
            - task: UsePythonVersion@0
              inputs:
                versionSpec: '3.x'
                addToPath: true
                architecture: 'x64'
              displayName: 'Use Python3'

            - task: configuredatabricks@0
              inputs:
                url: '$(DATABRICKS_URL)'
                token: '$(DATABRICKS_TOKEN)'
              displayName: 'Configure Databricks CLI'

            - task: deploynotebooks@0
              inputs:
                notebooksFolderPath: '$(Pipeline.Workspace)/di-notebooks'
                workspaceFolder: '/Shared/devops-ds'

```

```

        displayName: 'Deploy (copy) data processing notebook to the Databricks cluster'
- deployment: "Deploy_to_ADF"
  displayName: 'Deploy to ADF'
  timeoutInMinutes: 0
  environment: qa
  strategy:
    runOnce:
      deploy:
        steps:
          - task: AzureResourceGroupDeployment@2
            displayName: 'Deploy ADF resources'
            inputs:
              azureSubscription: $(AZURE_RM_CONNECTION)
              resourceGroupName: $(RESOURCE_GROUP)
              location: $(LOCATION)
              csmFile: '$(Pipeline.Workspace)/adf-pipelines/ARMTemplateForFactory.json'
              csmParametersFile: '$(Pipeline.Workspace)/adf-
pipelines/ARMTemplateParametersForFactory.json'
            overrideParameters: -data-ingestion-
  pipeline_properties_variables_data_file_name_defaultValue "$(DATA_FILE_NAME)"
  - job: "Integration_test_job"
    displayName: "Integration test job"
    dependsOn: [Deploy_to_Databricks, Deploy_to_ADF]
    pool:
      vmImage: 'ubuntu-latest'
    timeoutInMinutes: 0
    steps:
      - task: AzurePowerShell@4
        displayName: 'Execute ADF Pipeline'
        inputs:
          azureSubscription: $(AZURE_RM_CONNECTION)
          ScriptPath: '$(Build.SourcesDirectory)/adf/utils/Invoke-ADFPipeline.ps1'
          ScriptArguments: '-ResourceGroupName $(RESOURCE_GROUP) -DataFactoryName $(DATA_FACTORY_NAME) - -
PipelineName $(PIPELINE_NAME)'
          azurePowerShellVersion: LatestVersion
      - task: UsePythonVersion@0
        inputs:
          versionSpec: '3.x'
          addToPath: true
          architecture: 'x64'
        displayName: 'Use Python3'

      - task: configuredatabricks@0
        inputs:
          url: '$(DATABRICKS_URL)'
          token: '$(DATABRICKS_TOKEN)'
        displayName: 'Configure Databricks CLI'

      - task: executenotebook@0
        inputs:
          notebookPath: '/Shared/devops-ds/test-data-ingestion'
          existingClusterId: '$(DATABRICKS_CLUSTER_ID)'
          executionParams: '{"bin_file_name": "$(bin_FILE_NAME)"}'
        displayName: 'Test data ingestion'

      - task: waitexecution@0
        displayName: 'Wait until the testing is done'

```

Next steps

- [Source Control in Azure Data Factory](#)
- [Continuous integration and delivery in Azure Data Factory](#)
- [DevOps for Azure Databricks](#)

Import data into Azure Machine Learning designer

9/21/2022 • 4 minutes to read • [Edit Online](#)

In this article, you learn how to import your own data in the designer to create custom solutions. There are two ways you can import data into the designer:

- **Azure Machine Learning datasets** - Register [datasets](#) in Azure Machine Learning to enable advanced features that help you manage your data.
 - **Import Data component** - Use the [Import Data](#) component to directly access data from online data sources.

IMPORTANT

If you do not see graphical elements mentioned in this document, such as buttons in studio or designer, you may not have the right level of permissions to the workspace. Please contact your Azure subscription administrator to verify that you have been granted the correct level of access. For more information, see [Manage users and roles](#).

Use Azure Machine Learning datasets

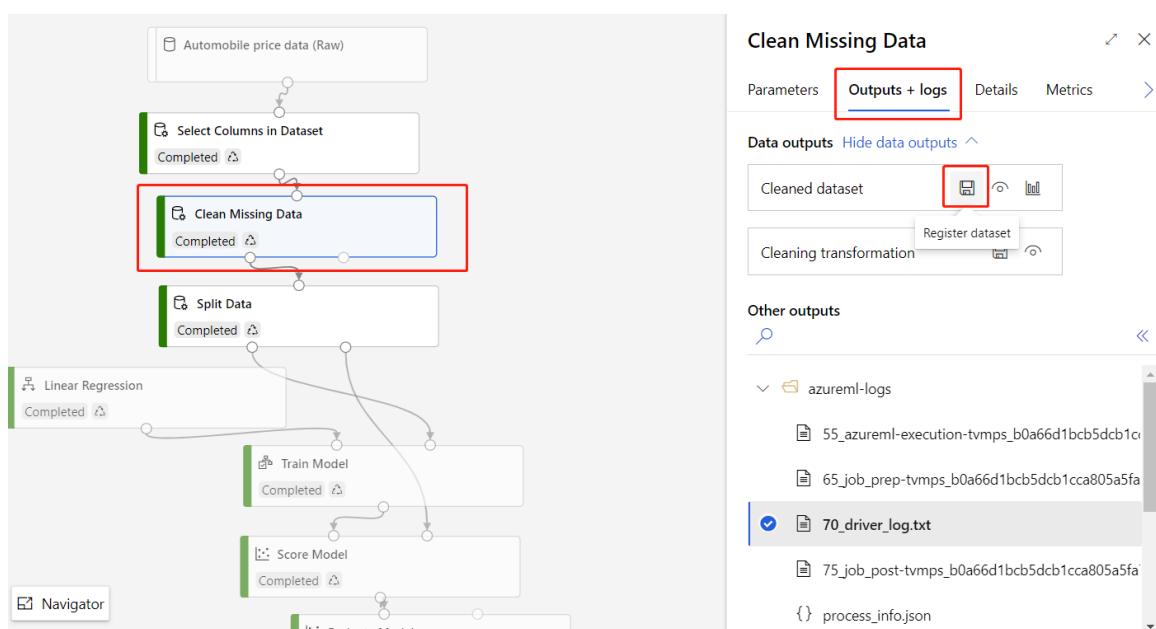
We recommend that you use [datasets](#) to import data into the designer. When you register a dataset, you can take full advantage of advanced data features like [versioning and tracking](#) and [data monitoring](#).

Register a dataset

You can register existing datasets [programmatically with the SDK](#) or [visually in Azure Machine Learning studio](#).

You can also register the output for any designer component as a dataset.

1. Select the component that outputs the data you want to register.
 2. In the properties pane, select **Outputs + logs > Register dataset**.



If the component output data is in a tabular format, you must choose to register the output as a **file dataset** or **tabular dataset**.

- **File dataset** registers the component's output folder as a file dataset. The output folder contains a data

file and meta files that the designer uses internally. Select this option if you want to continue to use the registered dataset in the designer.

- **Tabular dataset** registers only the component's the output data file as a tabular dataset. This format is easily consumed by other tools, for example in Automated Machine Learning or the Python SDK. Select this option if you plan to use the registered dataset outside of the designer.

Use a dataset

Your registered datasets can be found in the component palette, under **Datasets**. To use a dataset, drag and drop it onto the pipeline canvas. Then, connect the output port of the dataset to other components in the canvas.

If you register a file dataset, the output port type of the dataset is **AnyDirectory**. If you register a Tabular dataset, the output port type of the dataset if **DataFrameDirectory**. Note that if you connect the output port of the dataset to other components in the designer, the port type of datasets and components need to be aligned.

The screenshot shows the 'Datasets' section of the component palette. A search bar at the top says 'Search by name, tags and description'. Below it, it says '99 assets in total' and has a refresh icon. A red box highlights the 'Datasets (5)' section. Three datasets are listed: 'New Cleaned dataset' (9/9/2020), 'MD-Sample_pipeline-Train_Model-Trained...' (9/9/2020), and 'Cleaned dataset' (9/9/2020). The 'Cleaned dataset' item is also highlighted with a red box.

NOTE

The designer supports [dataset versioning](#). Specify the dataset version in the property panel of the dataset component.

Limitations

- Currently you can only visualize tabular dataset in the designer. If you register a file dataset outside designer, you cannot visualize it in the designer canvas.
- Currently the designer only supports preview outputs which are stored in **Azure blob storage**. You can check and change your output datastore in the **Output settings** under **Parameters** tab in the right panel of the component.
- If your data is stored in virtual network (VNet) and you want to preview, you need to enable workspace managed identity of the datastore.

1. Go the related datastore and click **Update authentication**

workspaceblobstore (Default)

[Overview](#) [Browse \(preview\)](#)

Refresh

Update credentials

Create dataset

2. Select Yes to enable workspace managed identity.

Update datastore credentials

Authentication type * ⓘ

Account key

Account key * ⓘ

Use workspace managed identity for data preview and profiling in Azure Machine Learning studio ⓘ

No Yes

Note: Azure Machine Learning service does not validate whether the underlying data source exi... ↴

Save Cancel

The screenshot shows a configuration dialog for updating datastore credentials. It includes fields for authentication type (selected as 'Account key') and account key, and a toggle switch for using workspace managed identity (with 'Yes' selected). A note at the bottom indicates that the service does not validate the underlying data source. The 'Yes' button is highlighted with a red box.

Import data using the Import Data component

While we recommend that you use datasets to import data, you can also use the [Import Data](#) component. The Import Data component skips registering your dataset in Azure Machine Learning and imports data directly from a [datastore](#) or HTTP URL.

For detailed information on how to use the Import Data component, see the [Import Data reference page](#).

NOTE

If your dataset has too many columns, you may encounter the following error: "Validation failed due to size limitation". To avoid this, [register the dataset in the Datasets interface](#).

Supported sources

This section lists the data sources supported by the designer. Data comes into the designer from either a

datastore or from [tabular dataset](#).

Datastore sources

For a list of supported datastore sources, see [Access data in Azure storage services](#).

Tabular dataset sources

The designer supports tabular datasets created from the following sources:

- Delimited files
- JSON files
- Parquet files
- SQL queries

Data types

The designer internally recognizes the following data types:

- String
- Integer
- Decimal
- Boolean
- Date

The designer uses an internal data type to pass data between components. You can explicitly convert your data into data table format using the [Convert to Dataset](#) component. Any component that accepts formats other than the internal format will convert the data silently before passing it to the next component.

Data constraints

Modules in the designer are limited by the size of the compute target. For larger datasets, you should use a larger Azure Machine Learning compute resource. For more information on Azure Machine Learning compute, see [What are compute targets in Azure Machine Learning?](#)

Access data in a virtual network

If your workspace is in a virtual network, you must perform additional configuration steps to visualize data in the designer. For more information on how to use datastores and datasets in a virtual network, see [Use Azure Machine Learning studio in an Azure virtual network](#).

Next steps

Learn the designer fundamentals with this [Tutorial: Predict automobile price with the designer](#).

Export or delete your Machine Learning service workspace data

9/21/2022 • 2 minutes to read • [Edit Online](#)

In Azure Machine Learning, you can export or delete your workspace data using either the portal's graphical interface or the Python SDK. This article describes both options.

NOTE

For information about viewing or deleting personal data, see [Azure Data Subject Requests for the GDPR](#). For more information about GDPR, see the [GDPR section of the Microsoft Trust Center](#) and the [GDPR section of the Service Trust portal](#).

NOTE

This article provides steps about how to delete personal data from the device or service and can be used to support your obligations under the GDPR. For general information about GDPR, see the [GDPR section of the Microsoft Trust Center](#) and the [GDPR section of the Service Trust portal](#).

Control your workspace data

In-product data stored by Azure Machine Learning is available for export and deletion. You can export and delete using Azure Machine Learning studio, CLI, and SDK. Telemetry data can be accessed through the Azure Privacy portal.

In Azure Machine Learning, personal data consists of user information in job history documents.

Delete high-level resources using the portal

When you create a workspace, Azure creates several resources within the resource group:

- The workspace itself
- A storage account
- A container registry
- An Applications Insights instance
- A key vault

These resources can be deleted by selecting them from the list and choosing **Delete**

The screenshot shows the Microsoft Azure portal interface. On the left, there's a navigation sidebar with sections like Home, Overview, Activity log, Access control (IAM), Tags, Events, Settings (Quickstart, Deployments, Policies, Properties, Locks, Export template), Cost Management (Cost analysis, Cost alerts (preview), Budgets, Advisor recommendations), Monitoring (Insights (preview), Alerts, Metrics, Diagnostic settings), and Storage Explorer (preview). The main content area is titled 'my-resource-group' and shows a list of resources. At the top of this list, there are buttons for Add, Edit columns, Delete resource group, Refresh, Move, Export to CSV, Open query, Assign tags, and Delete. The 'Delete' button is highlighted with a red box. Below these buttons, it says 'Deployments : 1 Succeeded'. The resource list includes:

Name	Type	Location
designerpython	Machine Learning	East US
designerpythash2138ef	Container registry	East US
designerpython3584489984	Application Insights	East US
designerpython4720032234	Storage account	East US
designerpython6289077319	Key vault	East US

Job history documents, which may contain personal user information, are stored in the storage account in blob storage, in subfolders of `/azureml`. You can download and delete the data from the portal.

The screenshot shows the Azure Storage Explorer (preview) interface. The left sidebar has options for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, and Storage Explorer (preview). The main area shows a tree view of blob containers under 'BLOB CONTAINERS'. One container, 'azureml', is expanded, showing subfolders: 'azureml', 'azureml-blobstore-89f54357-8ac2-', 'revisions', 'snapshots', and 'snapshotzips'. To the right, there's a detailed view of the 'azureml' folder with a table:

NAME	ACCESS TIER	ACCESS TIER LAST MODIFIED	LAST MODIFIED	BLOB TYPE
ComputeRecord				
Dataset				
ExperimentRun				
Labeling				

Export and delete machine learning resources using Azure Machine Learning studio

Azure Machine Learning studio provides a unified view of your machine learning resources, such as notebooks, datasets, models, and experiments. Azure Machine Learning studio emphasizes preserving a record of your data and experiments. Computational resources such as pipelines and compute resources can be deleted using the browser. For these resources, navigate to the resource in question and choose **Delete**.

Datasets can be unregistered and Experiments can be archived, but these operations don't delete the data. To entirely remove the data, datasets and experiment data must be deleted at the storage level. Deleting at the storage level is done using the portal, as described previously. An individual Job can be deleted directly in studio. Deleting a Job deletes the Job's data.

NOTE

Prior to unregistering a Dataset, use its **Data source** link to find the specific Data URL to delete.

You can download training artifacts from experimental jobs using the Studio. Choose the **Experiment** and **Job** in which you're interested. Choose **Output + logs** and navigate to the specific artifacts you wish to download. Choose ... and **Download**.

You can download a registered model by navigating to the **Model** and choosing **Download**.

my-workspace > Models > test:1

test:1

⟳ Refresh ⚡ Deploy ⬇ Download

Export and delete resources using the Python SDK

You can download the outputs of a particular job using:

```
# Retrieved from Azure Machine Learning web UI
run_id = 'aaaaaaaa-bbbb-cccc-dddd-0123456789AB'
experiment = ws.experiments['my-experiment']
run = next(run for run in ex.get_runs() if run.id == run_id)
metrics_output_port = run.get_pipeline_output('metrics_output')
model_output_port = run.get_pipeline_output('model_output')

metrics_output_port.download('.', show_progress=True)
model_output_port.download('.', show_progress=True)
```

The following machine learning resources can be deleted using the Python SDK:

TYPE	FUNCTION CALL	NOTES
Workspace	delete	Use delete-dependent-resources to cascade the delete
Model	delete	
ComputeTarget	delete	
WebService	delete	

Configure and submit training jobs

9/21/2022 • 10 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to configure and submit Azure Machine Learning jobs to train your models. Snippets of code explain the key parts of configuration and submission of a training script. Then use one of the [example notebooks](#) to find the full end-to-end working examples.

When training, it is common to start on your local computer, and then later scale out to a cloud-based cluster. With Azure Machine Learning, you can run your script on various compute targets without having to change your training script.

All you need to do is define the environment for each compute target within a **script job configuration**. Then, when you want to run your training experiment on a different compute target, specify the job configuration for that compute.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#) today
- The [Azure Machine Learning SDK for Python](#) (>= 1.13.0)
- An [Azure Machine Learning workspace](#), 
- A compute target,  [Create a compute target](#)

What's a script run configuration?

A [ScriptRunConfig](#) is used to configure the information necessary for submitting a training job as part of an experiment.

You submit your training experiment with a ScriptRunConfig object. This object includes the:

- **source_directory**: The source directory that contains your training script
- **script**: The training script to run
- **compute_target**: The compute target to run on
- **environment**: The environment to use when running the script
- and some additional configurable options (see the [reference documentation](#) for more information)

Train your model

The code pattern to submit a training job is the same for all types of compute targets:

1. Create an experiment to run
2. Create an environment where the script will run
3. Create a ScriptRunConfig, which specifies the compute target and environment
4. Submit the job
5. Wait for the job to complete

Or you can:

- Submit a HyperDrive run for [hyperparameter tuning](#).

- Submit an experiment via the [VS Code extension](#).

Create an experiment

Create an [experiment](#) in your workspace. An experiment is a light-weight container that helps to organize job submissions and keep track of code.

```
from azureml.core import Experiment

experiment_name = 'my_experiment'
experiment = Experiment(workspace=ws, name=experiment_name)
```

Select a compute target

Select the compute target where your training script will run on. If no compute target is specified in the `ScriptRunConfig`, or if `compute_target='local'`, Azure ML will execute your script locally.

The example code in this article assumes that you have already created a compute target `my_compute_target` from the "Prerequisites" section.

NOTE

Azure Databricks is not supported as a compute target for model training. You can use Azure Databricks for data preparation and deployment tasks.

NOTE

To create and attach a compute target for training on Azure Arc-enabled Kubernetes cluster, see [Configure Azure Arc-enabled Machine Learning](#)

Create an environment

Azure Machine Learning [environments](#) are an encapsulation of the environment where your machine learning training happens. They specify the Python packages, Docker image, environment variables, and software settings around your training and scoring scripts. They also specify runtimes (Python, Spark, or Docker).

You can either define your own environment, or use an Azure ML curated environment. [Curated environments](#) are predefined environments that are available in your workspace by default. These environments are backed by cached Docker images which reduces the job preparation cost. See [Azure Machine Learning Curated Environments](#) for the full list of available curated environments.

For a remote compute target, you can use one of these popular curated environments to start with:

```
from azureml.core import Workspace, Environment

ws = Workspace.from_config()
myenv = Environment.get(workspace=ws, name="AzureML-Minimal")
```

For more information and details about environments, see [Create & use software environments in Azure Machine Learning](#).

Local compute target

If your compute target is your **local machine**, you are responsible for ensuring that all the necessary packages

are available in the Python environment where the script runs. Use `python.user_managed_dependencies` to use your current Python environment (or the Python on the path you specify).

```
from azureml.core import Environment

myenv = Environment("user-managed-env")
myenv.python.user_managed_dependencies = True

# You can choose a specific Python environment by pointing to a Python path
# myenv.python.interpreter_path = '/home/johndoe/miniconda3/envs/myenv/bin/python'
```

Create the script job configuration

Now that you have a compute target (`my_compute_target`, see [Prerequisites](#) and environment (`myenv`, see [Create an environment](#)), create a script job configuration that runs your training script (`train.py`) located in your `project_folder` directory:

```
from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory=project_folder,
                      script='train.py',
                      compute_target=my_compute_target,
                      environment=myenv)

# Set compute target
# Skip this if you are running on your local computer
script_run_config.run_config.target = my_compute_target
```

If you do not specify an environment, a default environment will be created for you.

If you have command-line arguments you want to pass to your training script, you can specify them via the `arguments` parameter of the `ScriptRunConfig` constructor, e.g.

```
arguments=['--arg1', arg1_val, '--arg2', arg2_val].
```

If you want to override the default maximum time allowed for the job, you can do so via the `max_run_duration_seconds` parameter. The system will attempt to automatically cancel the job if it takes longer than this value.

Specify a distributed job configuration

If you want to run a [distributed training](#) job, provide the distributed job-specific config to the `distributed_job_config` parameter. Supported config types include [MpiConfiguration](#), [TensorflowConfiguration](#), and [PyTorchConfiguration](#).

For more information and examples on running distributed Horovod, TensorFlow and PyTorch jobs, see:

- [Train TensorFlow models](#)
- [Train PyTorch models](#)

Submit the experiment

```
run = experiment.submit(config=src)
run.wait_for_completion(show_output=True)
```

IMPORTANT

When you submit the training job, a snapshot of the directory that contains your training scripts is created and sent to the compute target. It is also stored as part of the experiment in your workspace. If you change files and submit the job again, only the changed files will be uploaded.

To prevent unnecessary files from being included in the snapshot, make an ignore file (`.gitignore` or `.amlignore`) in the directory. Add the files and directories to exclude to this file. For more information on the syntax to use inside this file, see [syntax and patterns](#) for `.gitignore`. The `.amlignore` file uses the same syntax. *If both files exist, the `.amlignore` file is used and the `.gitignore` file is unused.*

For more information about snapshots, see [Snapshots](#).

IMPORTANT

Special Folders Two folders, *outputs* and *logs*, receive special treatment by Azure Machine Learning. During training, when you write files to folders named *outputs* and *logs* that are relative to the root directory (`./outputs` and `./logs`, respectively), the files will automatically upload to your job history so that you have access to them once your job is finished.

To create artifacts during training (such as model files, checkpoints, data files, or plotted images) write these to the `./outputs` folder.

Similarly, you can write any logs from your training job to the `./logs` folder. To utilize Azure Machine Learning's [TensorBoard integration](#) make sure you write your TensorBoard logs to this folder. While your job is in progress, you will be able to launch TensorBoard and stream these logs. Later, you will also be able to restore the logs from any of your previous jobs.

For example, to download a file written to the *outputs* folder to your local machine after your remote training job:

```
run.download_file(name='outputs/my_output_file', output_file_path='my_destination_path')
```

Git tracking and integration

When you start a training job where the source directory is a local Git repository, information about the repository is stored in the job history. For more information, see [Git integration for Azure Machine Learning](#).

Notebook examples

See these notebooks for examples of configuring jobs for various training scenarios:

- [Training on various compute targets](#)
- [Training with ML frameworks](#)
- [tutorials/img-classification-part1-training.ipynb](#)

Learn how to run notebooks by following the article [Use Jupyter notebooks to explore this service](#).

Troubleshooting

- **AttributeError: 'RoundTripLoader' object has no attribute 'comment_handling'**: This error comes from the new version (v0.17.5) of `ruamel-yaml`, an `azureml-core` dependency, that introduces a breaking change to `azureml-core`. In order to fix this error, please uninstall `ruamel-yaml` by running `pip uninstall ruamel-yaml` and installing a different version of `ruamel-yaml`; the supported versions are v0.15.35 to v0.17.4 (inclusive). You can do this by running `pip install "ruamel-yaml>=0.15.35,<0.17.5"`.
- **Job fails with `jwt.exceptions.DecodeError`**: Exact error message:

```
jwt.exceptions.DecodeError: It is required that you pass in a value for the "algorithms" argument when calling decode()
```

Consider upgrading to the latest version of azureml-core: `pip install -U azureml-core`.

If you are running into this issue for local jobs, check the version of PyJWT installed in your environment where you are starting jobs. The supported versions of PyJWT are < 2.0.0. Uninstall PyJWT from the environment if the version is $\geq 2.0.0$. You may check the version of PyJWT, uninstall and install the right version as follows:

1. Start a command shell, activate conda environment where azureml-core is installed.
2. Enter `pip freeze` and look for `PyJWT`, if found, the version listed should be < 2.0.0
3. If the listed version is not a supported version, `pip uninstall PyJWT` in the command shell and enter y for confirmation.
4. Install using `pip install 'PyJWT<2.0.0'`

If you are submitting a user-created environment with your job, consider using the latest version of azureml-core in that environment. Versions $\geq 1.18.0$ of azureml-core already pin PyJWT < 2.0.0. If you need to use a version of azureml-core < 1.18.0 in the environment you submit, make sure to specify PyJWT < 2.0.0 in your pip dependencies.

- **ModuleErrors (No module named):** If you are running into ModuleErrors while submitting experiments in Azure ML, the training script is expecting a package to be installed but it isn't added. Once you provide the package name, Azure ML installs the package in the environment used for your training job.

If you are using Estimators to submit experiments, you can specify a package name via `pip_packages` or `conda_packages` parameter in the estimator based on from which source you want to install the package. You can also specify a yml file with all your dependencies using `conda_dependencies_file` or list all your pip requirements in a txt file using `pip_requirements_file` parameter. If you have your own Azure ML Environment object that you want to override the default image used by the estimator, you can specify that environment via the `environment` parameter of the estimator constructor.

Azure ML maintained docker images and their contents can be seen in [AzureML Containers](#). Framework-specific dependencies are listed in the respective framework documentation:

- [Chainer](#)
- [PyTorch](#)
- [TensorFlow](#)
- [SKLearn](#)

NOTE

If you think a particular package is common enough to be added in Azure ML maintained images and environments please raise a GitHub issue in [AzureML Containers](#).

- **NameError (Name not defined), AttributeError (Object has no attribute):** This exception should come from your training scripts. You can look at the log files from Azure portal to get more information about the specific name not defined or attribute error. From the SDK, you can use `run.get_details()` to look at the error message. This will also list all the log files generated for your job. Please make sure to take a look at your training script and fix the error before resubmitting your job.
- **Job or experiment deletion:** Experiments can be archived by using the [Experiment.archive](#) method, or from the Experiment tab view in Azure Machine Learning studio client via the "Archive experiment" button. This action hides the experiment from list queries and views, but does not delete it.

Permanent deletion of individual experiments or jobs is not currently supported. For more information on deleting Workspace assets, see [Export or delete your Machine Learning service workspace data](#).

- **Metric Document is too large:** Azure Machine Learning has internal limits on the size of metric objects that can be logged at once from a training job. If you encounter a "Metric Document is too large" error when logging a list-valued metric, try splitting the list into smaller chunks, for example:

```
run.log_list("my metric name", my_metric[:N])
run.log_list("my metric name", my_metric[N:])
```

Internally, Azure ML concatenates the blocks with the same metric name into a contiguous list.

- **Compute target takes a long time to start:** The Docker images for compute targets are loaded from Azure Container Registry (ACR). By default, Azure Machine Learning creates an ACR that uses the *basic* service tier. Changing the ACR for your workspace to standard or premium tier may reduce the time it takes to build and load images. For more information, see [Azure Container Registry service tiers](#).

Next steps

- [Tutorial: Train and deploy a model](#) uses a managed compute target to train a model.
- See how to train models with specific ML frameworks, such as [Scikit-learn](#), [TensorFlow](#), and [PyTorch](#).
- Learn how to [efficiently tune hyperparameters](#) to build better models.
- Once you have a trained model, learn [how and where to deploy models](#).
- View the [ScriptRunConfig class](#) SDK reference.
- [Use Azure Machine Learning with Azure Virtual Networks](#)

Train models with the Azure Machine Learning Python SDK (v1)

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

Learn how to attach Azure compute resources to your Azure Machine Learning workspace with SDK v1. Then you can use these resources as training and inference [compute targets](#) in your machine learning tasks.

In this article, learn how to set up your workspace to use these compute resources:

- Your local computer
- Remote virtual machines
- Apache Spark pools (powered by Azure Synapse Analytics)
- Azure HDInsight
- Azure Batch
- Azure Databricks - used as a training compute target only in [machine learning pipelines](#)
- Azure Data Lake Analytics
- Azure Container Instance
- Azure Machine Learning Kubernetes

To use compute targets managed by Azure Machine Learning, see:

- [Azure Machine Learning compute instance](#)
- [Azure Machine Learning compute cluster](#)
- [Azure Kubernetes Service cluster](#)

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create workspace resources](#).
- The [Azure CLI extension for Machine Learning service](#), [Azure Machine Learning Python SDK](#), or the [Azure Machine Learning Visual Studio Code extension](#).

Limitations

- **Do not create multiple, simultaneous attachments to the same compute** from your workspace.
For example, attaching one Azure Kubernetes Service cluster to a workspace using two different names.
Each new attachment will break the previous existing attachment(s).

If you want to reattach a compute target, for example to change TLS or other cluster configuration setting, you must first remove the existing attachment.

What's a compute target?

With Azure Machine Learning, you can train your model on various resources or environments, collectively referred to as [compute targets](#). A compute target can be a local machine or a cloud resource, such as an Azure Machine Learning Compute, Azure HDInsight, or a remote virtual machine. You also use compute targets for model deployment as described in "[Where and how to deploy your models](#)".

Local computer

When you use your local computer for **training**, there is no need to create a compute target. Just submit the [training run](#) from your local machine.

When you use your local computer for **inference**, you must have Docker installed. To perform the deployment, use [LocalWebservice.deploy_configuration\(\)](#) to define the port that the web service will use. Then use the normal deployment process as described in [Deploy models with Azure Machine Learning](#).

Remote virtual machines

Azure Machine Learning also supports attaching an Azure Virtual Machine. The VM must be an Azure Data Science Virtual Machine (DSVM). The VM offers a curated choice of tools and frameworks for full-lifecycle machine learning development. For more information on how to use the DSVM with Azure Machine Learning, see [Configure a development environment](#).

TIP

Instead of a remote VM, we recommend using the [Azure Machine Learning compute instance](#). It is a fully managed, cloud-based compute solution that is specific to Azure Machine Learning. For more information, see [create and manage Azure Machine Learning compute instance](#).

1. **Create:** Azure Machine Learning cannot create a remote VM for you. Instead, you must create the VM and then attach it to your Azure Machine Learning workspace. For information on creating a DSVM, see [Provision the Data Science Virtual Machine for Linux \(Ubuntu\)](#).

WARNING

Azure Machine Learning only supports virtual machines that run **Ubuntu**. When you create a VM or choose an existing VM, you must select a VM that uses Ubuntu.

Azure Machine Learning also requires the virtual machine to have a **public IP address**.

2. **Attach:** To attach an existing virtual machine as a compute target, you must provide the resource ID, user name, and password for the virtual machine. The resource ID of the VM can be constructed using the subscription ID, resource group name, and VM name using the following string format:

```
/subscriptions/<subscription_id>/resourceGroups/<resource_group>/providers/Microsoft.Compute/virtualMachines/<vm_name>
```

```
from azureml.core.compute import RemoteCompute, ComputeTarget

# Create the compute config
compute_target_name = "attach-dsvm"

attach_config = RemoteCompute.attach_configuration(resource_id='<resource_id>',
                                                    ssh_port=22,
                                                    username='<username>',
                                                    password="<password>")

# Attach the compute
compute = ComputeTarget.attach(ws, compute_target_name, attach_config)

compute.wait_for_completion(show_output=True)
```

Or you can attach the DSVM to your workspace [using Azure Machine Learning studio](#).

WARNING

Do not create multiple, simultaneous attachments to the same DSVM from your workspace. Each new attachment will break the previous existing attachment(s).

3. **Configure:** Create a run configuration for the DSVM compute target. Docker and conda are used to create and configure the training environment on the DSVM.

```
from azureml.core import ScriptRunConfig
from azureml.core.environment import Environment
from azureml.core.conda_dependencies import CondaDependencies

# Create environment
myenv = Environment(name="myenv")

# Specify the conda dependencies
myenv.python.conda_dependencies = CondaDependencies.create(conda_packages=['scikit-learn'])

# If no base image is explicitly specified the default CPU image
#"azureml.core.runconfig.DEFAULT_CPU_IMAGE" will be used
# To use GPU in DSVM, you should specify the default GPU base Docker image or another GPU-enabled
#image:
# myenv.docker.enabled = True
# myenv.docker.base_image = azureml.core.runconfig.DEFAULT_GPU_IMAGE

# Configure the run configuration with the Linux DSVM as the compute target and the environment
# defined above
src = ScriptRunConfig(source_directory=".", script="train.py", compute_target=compute,
environment=myenv)
```

TIP

If you want to **remove** (detach) a VM from your workspace, use the [RemoteCompute.detach\(\)](#) method.

Azure Machine Learning does not delete the VM for you. You must manually delete the VM using the Azure portal, CLI, or the SDK for Azure VM.

Apache Spark pools

The Azure Synapse Analytics integration with Azure Machine Learning (preview) allows you to attach an Apache Spark pool backed by Azure Synapse for interactive data exploration and preparation. With this integration, you can have a dedicated compute for data wrangling at scale. For more information, see [How to attach Apache Spark pools powered by Azure Synapse Analytics](#).

Azure HDInsight

Azure HDInsight is a popular platform for big-data analytics. The platform provides Apache Spark, which can be used to train your model.

1. **Create:** Azure Machine Learning cannot create an HDInsight cluster for you. Instead, you must create the cluster and then attach it to your Azure Machine Learning workspace. For more information, see [Create a Spark Cluster in HDInsight](#).

WARNING

Azure Machine Learning requires the HDInsight cluster to have a **public IP address**.

When you create the cluster, you must specify an SSH user name and password. Take note of these values, as you need them to use HDInsight as a compute target.

After the cluster is created, connect to it with the hostname <clustername>-ssh.azurehdinsight.net, where <clustername> is the name that you provided for the cluster.

2. **Attach:** To attach an HDInsight cluster as a compute target, you must provide the resource ID, user name, and password for the HDInsight cluster. The resource ID of the HDInsight cluster can be constructed using the subscription ID, resource group name, and HDInsight cluster name using the following string format:

```
/subscriptions/<subscription_id>/resourceGroups/<resource_group>/providers/Microsoft.HDInsight/clusters/<cluster_name>
```

```
from azureml.core.compute import ComputeTarget, HDInsightCompute
from azureml.exceptions import ComputeTargetException

try:
    # if you want to connect using SSH key instead of username/password you can provide parameters
    private_key_file and private_key_passphrase

    attach_config = HDInsightCompute.attach_configuration(resource_id='<resource_id>',
                                                            ssh_port=22,
                                                            username='<ssh-username>',
                                                            password='<ssh-pwd>')

    hdi_compute = ComputeTarget.attach(workspace=ws,
                                       name='myhdi',
                                       attach_configuration=attach_config)

except ComputeTargetException as e:
    print("Caught = {}".format(e.message))

hdi_compute.wait_for_completion(show_output=True)
```

Or you can attach the HDInsight cluster to your workspace [using Azure Machine Learning studio](#).

WARNING

Do not create multiple, simultaneous attachments to the same HDInsight from your workspace. Each new attachment will break the previous existing attachment(s).

3. **Configure:** Create a run configuration for the HDI compute target.

```
from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies

# use pyspark framework
run_hdi = RunConfiguration(framework="pyspark")

# Set compute target to the HDI cluster
run_hdi.target = hdi_compute.name

# specify CondaDependencies object to ask system installing numpy
cd = CondaDependencies()
cd.add_conda_package('numpy')
run_hdi.environment.python.conda_dependencies = cd
```

TIP

If you want to **remove** (detach) an HDInsight cluster from the workspace, use the [HDInsightCompute.detach\(\)](#) method.

Azure Machine Learning does not delete the HDInsight cluster for you. You must manually delete it using the Azure portal, CLI, or the SDK for Azure HDInsight.

Azure Batch

Azure Batch is used to run large-scale parallel and high-performance computing (HPC) applications efficiently in the cloud. AzureBatchStep can be used in an Azure Machine Learning Pipeline to submit jobs to an Azure Batch pool of machines.

To attach Azure Batch as a compute target, you must use the Azure Machine Learning SDK and provide the following information:

- **Azure Batch compute name:** A friendly name to be used for the compute within the workspace
- **Azure Batch account name:** The name of the Azure Batch account
- **Resource Group:** The resource group that contains the Azure Batch account.

The following code demonstrates how to attach Azure Batch as a compute target:

```
from azureml.core.compute import ComputeTarget, BatchCompute
from azureml.exceptions import ComputeTargetException

# Name to associate with new compute in workspace
batch_compute_name = 'mybatchcompute'

# Batch account details needed to attach as compute to workspace
batch_account_name = "<batch_account_name>" # Name of the Batch account
# Name of the resource group which contains this account
batch_resource_group = "<batch_resource_group>"

try:
    # check if the compute is already attached
    batch_compute = BatchCompute(ws, batch_compute_name)
except ComputeTargetException:
    print('Attaching Batch compute...')
    provisioning_config = BatchCompute.attach_configuration(
        resource_group=batch_resource_group, account_name=batch_account_name)
    batch_compute = ComputeTarget.attach(
        ws, batch_compute_name, provisioning_config)
    batch_compute.wait_for_completion()
    print("Provisioning state:{}".format(batch_compute.provisioning_state))
    print("Provisioning errors:{}".format(batch_compute.provisioning_errors))

print("Using Batch compute:{}".format(batch_compute.cluster_resource_id))
```

WARNING

Do not create multiple, simultaneous attachments to the same Azure Batch from your workspace. Each new attachment will break the previous existing attachment(s).

Azure Databricks

Azure Databricks is an Apache Spark-based environment in the Azure cloud. It can be used as a compute target with an Azure Machine Learning pipeline.

IMPORTANT

Azure Machine Learning cannot create an Azure Databricks compute target. Instead, you must create an Azure Databricks workspace, and then attach it to your Azure Machine Learning workspace. To create a workspace resource, see the [Run a Spark job on Azure Databricks](#) document.

To attach an Azure Databricks workspace from a different Azure subscription, you (your Azure AD account) must be granted the **Contributor** role on the Azure Databricks workspace. Check your access in the [Azure portal](#).

To attach Azure Databricks as a compute target, provide the following information:

- **Databricks compute name:** The name you want to assign to this compute resource.
- **Databricks workspace name:** The name of the Azure Databricks workspace.
- **Databricks access token:** The access token used to authenticate to Azure Databricks. To generate an access token, see the [Authentication](#) document.

The following code demonstrates how to attach Azure Databricks as a compute target with the Azure Machine Learning SDK:

```
import os
from azureml.core.compute import ComputeTarget, DatabricksCompute
from azureml.exceptions import ComputeTargetException

databricks_compute_name = os.environ.get(
    "AML_DATABRICKS_COMPUTE_NAME", "<databricks_compute_name>")
databricks_workspace_name = os.environ.get(
    "AML_DATABRICKS_WORKSPACE", "<databricks_workspace_name>")
databricks_resource_group = os.environ.get(
    "AML_DATABRICKS_RESOURCE_GROUP", "<databricks_resource_group>")
databricks_access_token = os.environ.get(
    "AML_DATABRICKS_ACCESS_TOKEN", "<databricks_access_token>")

try:
    databricks_compute = ComputeTarget(
        workspace=ws, name=databricks_compute_name)
    print('Compute target already exists')
except ComputeTargetException:
    print('compute not found')
    print('databricks_compute_name {}'.format(databricks_compute_name))
    print('databricks_workspace_name {}'.format(databricks_workspace_name))
    print('databricks_access_token {}'.format(databricks_access_token))

    # Create attach config
    attach_config = DatabricksCompute.attach_configuration(resource_group=databricks_resource_group,
                                                            workspace_name=databricks_workspace_name,
                                                            access_token=databricks_access_token)

    databricks_compute = ComputeTarget.attach(
        ws,
        databricks_compute_name,
        attach_config
    )

    databricks_compute.wait_for_completion(True)
```

For a more detailed example, see an [example notebook](#) on GitHub.

WARNING

Do not create multiple, simultaneous attachments to the same Azure Databricks from your workspace. Each new attachment will break the previous existing attachment(s).

Azure Data Lake Analytics

Azure Data Lake Analytics is a big data analytics platform in the Azure cloud. It can be used as a compute target with an Azure Machine Learning pipeline.

Create an Azure Data Lake Analytics account before using it. To create this resource, see the [Get started with Azure Data Lake Analytics](#) document.

To attach Data Lake Analytics as a compute target, you must use the Azure Machine Learning SDK and provide the following information:

- **Compute name:** The name you want to assign to this compute resource.
- **Resource Group:** The resource group that contains the Data Lake Analytics account.
- **Account name:** The Data Lake Analytics account name.

The following code demonstrates how to attach Data Lake Analytics as a compute target:

```
import os
from azureml.core.compute import ComputeTarget, AdlaCompute
from azureml.exceptions import ComputeTargetException

adla_compute_name = os.environ.get(
    "AML_ADLA_COMPUTE_NAME", "<adla_compute_name>")
adla_resource_group = os.environ.get(
    "AML_ADLA_RESOURCE_GROUP", "<adla_resource_group>")
adla_account_name = os.environ.get(
    "AML_ADLA_ACCOUNT_NAME", "<adla_account_name>")

try:
    adla_compute = ComputeTarget(workspace=ws, name=adla_compute_name)
    print('Compute target already exists')
except ComputeTargetException:
    print('compute not found')
    print('adla_compute_name {}'.format(adla_compute_name))
    print('adla_resource_id {}'.format(adla_resource_group))
    print('adla_account_name {}'.format(adla_account_name))
    # create attach config
    attach_config = AdlaCompute.attach_configuration(resource_group=adla_resource_group,
                                                      account_name=adla_account_name)

    # Attach ADLA
    adla_compute = ComputeTarget.attach(
        ws,
        adla_compute_name,
        attach_config
    )

    adla_compute.wait_for_completion(True)
```

For a more detailed example, see an [example notebook](#) on GitHub.

WARNING

Do not create multiple, simultaneous attachments to the same ADLA from your workspace. Each new attachment will break the previous existing attachment(s).

TIP

Azure Machine Learning pipelines can only work with data stored in the default data store of the Data Lake Analytics account. If the data you need to work with is in a non-default store, you can use a [DataTransferStep](#) to copy the data before training.

Azure Container Instance

Azure Container Instances (ACI) are created dynamically when you deploy a model. You cannot create or attach ACI to your workspace in any other way. For more information, see [Deploy a model to Azure Container Instances](#).

Kubernetes

Azure Machine Learning provides you with the option to attach your own Kubernetes clusters for training and inferencing. See [Configure Kubernetes cluster for Azure Machine Learning](#).

To detach a Kubernetes cluster from your workspace, use the following method:

```
compute_target.detach()
```

WARNING

Detaching a cluster **does not delete the cluster**. To delete an Azure Kubernetes Service cluster, see [Use the Azure CLI with AKS](#). To delete an Azure Arc-enabled Kubernetes cluster, see [Azure Arc quickstart](#).

Notebook examples

See these notebooks for examples of training with various compute targets:

- [how-to-use-azureml/training](#)
- [tutorials/img-classification-part1-training.ipynb](#)

Learn how to run notebooks by following the article [Use Jupyter notebooks to explore this service](#).

Next steps

- Use the compute resource to [configure and submit a training run](#).
- [Tutorial: Train and deploy a model](#) uses a managed compute target to train a model.
- Learn how to [efficiently tune hyperparameters](#) to build better models.
- Once you have a trained model, learn [how and where to deploy models](#).
- [Use Azure Machine Learning with Azure Virtual Networks](#)

Distributed GPU training guide (SDK v1)

9/21/2022 • 13 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

Learn more about how to use distributed GPU training code in Azure Machine Learning (ML). This article will not teach you about distributed training. It will help you run your existing distributed training code on Azure Machine Learning. It offers tips and examples for you to follow for each framework:

- Message Passing Interface (MPI)
 - Horovod
 - DeepSpeed
 - Environment variables from Open MPI
- PyTorch
 - Process group initialization
 - Launch options
 - DistributedDataParallel (per-process-launch)
 - Using `torch.distributed.launch` (per-node-launch)
 - PyTorch Lightning
 - Hugging Face Transformers
- TensorFlow
 - Environment variables for TensorFlow (TF_CONFIG)
- Accelerate GPU training with InfiniBand

Prerequisites

Review these [basic concepts of distributed GPU training](#) such as *data parallelism*, *distributed data parallelism*, and *model parallelism*.

TIP

If you don't know which type of parallelism to use, more than 90% of the time you should use **Distributed Data Parallelism**.

MPI

Azure ML offers an [MPI job](#) to launch a given number of processes in each node. You can adopt this approach to run distributed training using either per-process-launcher or per-node-launcher, depending on whether `process_count_per_node` is set to 1 (the default) for per-node-launcher, or equal to the number of devices/GPUs for per-process-launcher. Azure ML constructs the full MPI launch command (`mpirun`) behind the scenes. You can't provide your own full head-node-launcher commands like `mpirun` or `DeepSpeed launcher`.

TIP

The base Docker image used by an Azure Machine Learning MPI job needs to have an MPI library installed. Open MPI is included in all the [AzureML GPU base images](#). When you use a custom Docker image, you are responsible for making sure the image includes an MPI library. Open MPI is recommended, but you can also use a different MPI implementation such as Intel MPI. Azure ML also provides [curated environments](#) for popular frameworks.

To run distributed training using MPI, follow these steps:

1. Use an Azure ML environment with the preferred deep learning framework and MPI. AzureML provides [curated environment](#) for popular frameworks.
2. Define `MpiConfiguration` with `process_count_per_node` and `node_count`. `process_count_per_node` should be equal to the number of GPUs per node for per-process-launch, or set to 1 (the default) for per-node-launch if the user script will be responsible for launching the processes per node.
3. Pass the `MpiConfiguration` object to the `distributed_job_config` parameter of `ScriptRunConfig`.

```
from azureml.core import Workspace, ScriptRunConfig, Environment, Experiment
from azureml.core.runconfig import MpiConfiguration

curated_env_name = 'AzureML-PyTorch-1.6-GPU'
pytorch_env = Environment.get(workspace=ws, name=curated_env_name)
distr_config = MpiConfiguration(process_count_per_node=4, node_count=2)

run_config = ScriptRunConfig(
    source_directory= './src',
    script='train.py',
    compute_target=compute_target,
    environment=pytorch_env,
    distributed_job_config=distr_config,
)

# submit the run configuration to start the job
run = Experiment(ws, "experiment_name").submit(run_config)
```

Horovod

Use the MPI job configuration when you use [Horovod](#) for distributed training with the deep learning framework.

Make sure your code follows these tips:

- The training code is instrumented correctly with Horovod before adding the Azure ML parts
- Your Azure ML environment contains Horovod and MPI. The PyTorch and TensorFlow curated GPU environments come pre-configured with Horovod and its dependencies.
- Create an `MpiConfiguration` with your desired distribution.

Horovod example

- [azureml-examples: TensorFlow distributed training using Horovod](#)

DeepSpeed

Don't use DeepSpeed's custom launcher to run distributed training with the [DeepSpeed](#) library on Azure ML. Instead, configure an MPI job to launch the training job [with MPI](#).

Make sure your code follows these tips:

- Your Azure ML environment contains DeepSpeed and its dependencies, Open MPI, and mpi4py.
- Create an `MpiConfiguration` with your distribution.

DeepSeed example

- [azureml-examples: Distributed training with DeepSpeed on CIFAR-10](#)

Environment variables from Open MPI

When running MPI jobs with Open MPI images, the following environment variables for each process launched:

1. `OMPI_COMM_WORLD_RANK` - the rank of the process
2. `OMPI_COMM_WORLD_SIZE` - the world size
3. `AZ_BATCH_MASTER_NODE` - primary address with port, `MASTER_ADDR:MASTER_PORT`
4. `OMPI_COMM_WORLD_LOCAL_RANK` - the local rank of the process on the node
5. `OMPI_COMM_WORLD_LOCAL_SIZE` - number of processes on the node

TIP

Despite the name, environment variable `OMPI_COMM_WORLD_NODE_RANK` does not correspond to the `NODE_RANK`. To use per-node-launcher, set `process_count_per_node=1` and use `OMPI_COMM_WORLD_RANK` as the `NODE_RANK`.

PyTorch

Azure ML supports running distributed jobs using PyTorch's native distributed training capabilities (`torch.distributed`).

TIP

For data parallelism, the [official PyTorch guidance](#) is to use `DistributedDataParallel` (DDP) over `DataParallel` for both single-node and multi-node distributed training. PyTorch also [recommends using `DistributedDataParallel` over the multiprocessing package](#). Azure Machine Learning documentation and examples will therefore focus on `DistributedDataParallel` training.

Process group initialization

The backbone of any distributed training is based on a group of processes that know each other and can communicate with each other using a backend. For PyTorch, the process group is created by calling `torch.distributed.init_process_group` in **all distributed processes** to collectively form a process group.

```
torch.distributed.init_process_group(backend='nccl', init_method='env://', ...)
```

The most common communication backends used are `mpi`, `nccl`, and `gloo`. For GPU-based training `nccl` is recommended for best performance and should be used whenever possible.

`init_method` tells how each process can discover each other, how they initialize and verify the process group using the communication backend. By default if `init_method` is not specified PyTorch will use the environment variable initialization method (`env://`). `init_method` is the recommended initialization method to use in your training code to run distributed PyTorch on Azure ML. PyTorch will look for the following environment variables for initialization:

- `MASTER_ADDR` - IP address of the machine that will host the process with rank 0.
- `MASTER_PORT` - A free port on the machine that will host the process with rank 0.
- `WORLD_SIZE` - The total number of processes. Should be equal to the total number of devices (GPU) used for distributed training.
- `RANK` - The (global) rank of the current process. The possible values are 0 to (world size - 1).

For more information on process group initialization, see the [PyTorch documentation](#).

Beyond these, many applications will also need the following environment variables:

- `LOCAL_RANK` - The local (relative) rank of the process within the node. The possible values are 0 to (# of processes on the node - 1). This information is useful because many operations such as data preparation only should be performed once per node --- usually on `local_rank` = 0.
- `NODE_RANK` - The rank of the node for multi-node training. The possible values are 0 to (total # of nodes - 1).

PyTorch launch options

The Azure ML PyTorch job supports two types of options for launching distributed training:

- **Per-process-launcher:** The system will launch all distributed processes for you, with all the relevant information (such as environment variables) to set up the process group.
- **Per-node-launcher:** You provide Azure ML with the utility launcher that will get run on each node. The utility launcher will handle launching each of the processes on a given node. Locally within each node, `RANK` and `LOCAL_RANK` are set up by the launcher. The `torch.distributed.launch` utility and PyTorch Lightning both belong in this category.

There are no fundamental differences between these launch options. The choice is largely up to your preference or the conventions of the frameworks/libraries built on top of vanilla PyTorch (such as Lightning or Hugging Face).

The following sections go into more detail on how to configure Azure ML PyTorch jobs for each of the launch options.

DistributedDataParallel (per-process-launch)

You don't need to use a launcher utility like `torch.distributed.launch`. To run a distributed PyTorch job:

1. Specify the training script and arguments
2. Create a `PyTorchConfiguration` and specify the `process_count` and `node_count`. The `process_count` corresponds to the total number of processes you want to run for your job. `process_count` should typically equal `# GPUs per node x # nodes`. If `process_count` isn't specified, Azure ML will by default launch one process per node.

Azure ML will set the `MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, and `NODE_RANK` environment variables on each node, and set the process-level `RANK` and `LOCAL_RANK` environment variables.

To use this option for multi-process-per-node training, use Azure ML Python SDK `>= 1.22.0`. `Process_count` was introduced in 1.22.0.

```
from azureml.core import ScriptRunConfig, Environment, Experiment
from azureml.core.runconfig import PyTorchConfiguration

curated_env_name = 'AzureML-PyTorch-1.6-GPU'
pytorch_env = Environment.get(workspace=ws, name=curated_env_name)
distr_config = PyTorchConfiguration(process_count=8, node_count=2)

run_config = ScriptRunConfig(
    source_directory='./src',
    script='train.py',
    arguments=['--epochs', 50],
    compute_target=compute_target,
    environment=pytorch_env,
    distributed_job_config=distr_config,
)

run = Experiment(ws, 'experiment_name').submit(run_config)
```

TIP

If your training script passes information like local rank or rank as script arguments, you can reference the environment variable(s) in the arguments:

```
arguments=['--epochs', 50, '--local_rank', $LOCAL_RANK]
```

Pytorch per-process-launch example

- [azureml-examples: Distributed training with PyTorch on CIFAR-10](#)

Using `torch.distributed.launch` (per-node-launch)

PyTorch provides a launch utility in `torch.distributed.launch` that you can use to launch multiple processes per node. The `torch.distributed.launch` module spawns multiple training processes on each of the nodes.

The following steps demonstrate how to configure a PyTorch job with a per-node-launcher on Azure ML. The job achieves the equivalent of running the following command:

```
python -m torch.distributed.launch --nproc_per_node <num processes per node> \
--nnodes <num nodes> --node_rank $NODE_RANK --master_addr $MASTER_ADDR \
--master_port $MASTER_PORT --use_env \
<your training script> <your script arguments>
```

1. Provide the `torch.distributed.launch` command to the `command` parameter of the `ScriptRunConfig` constructor. Azure ML runs this command on each node of your training cluster. `--nproc_per_node` should be less than or equal to the number of GPUs available on each node. `MASTER_ADDR`, `MASTER_PORT`, and `NODE_RANK` are all set by Azure ML, so you can just reference the environment variables in the command. Azure ML sets `MASTER_PORT` to `6105`, but you can pass a different value to the `--master_port` argument of `torch.distributed.launch` command if you wish. (The launch utility will reset the environment variables.)
2. Create a `PyTorchConfiguration` and specify the `node_count`.

```
from azureml.core import ScriptRunConfig, Environment, Experiment
from azureml.core.runconfig import PyTorchConfiguration

curated_env_name = 'AzureML-PyTorch-1.6-GPU'
pytorch_env = Environment.get(workspace=ws, name=curated_env_name)
distr_config = PyTorchConfiguration(node_count=2)
launch_cmd = "python -m torch.distributed.launch --nproc_per_node 4 --nnodes 2 --node_rank $NODE_RANK --master_addr $MASTER_ADDR --master_port $MASTER_PORT --use_env train.py --epochs 50".split()

run_config = ScriptRunConfig(
    source_directory='./src',
    command=launch_cmd,
    compute_target=compute_target,
    environment=pytorch_env,
    distributed_job_config=distr_config,
)

run = Experiment(ws, 'experiment_name').submit(run_config)
```

TIP

Single-node multi-GPU training: If you are using the launch utility to run single-node multi-GPU PyTorch training, you do not need to specify the `distributed_job_config` parameter of `ScriptRunConfig`.

```
launch_cmd = "python -m torch.distributed.launch --nproc_per_node 4 --use_env train.py --epochs 50".split()

run_config = ScriptRunConfig(
    source_directory='./src',
    command=launch_cmd,
    compute_target=compute_target,
    environment=pytorch_env,
)
```

PyTorch per-node-launch example

- [azureml-examples: Distributed training with PyTorch on CIFAR-10](#)

PyTorch Lightning

[PyTorch Lightning](#) is a lightweight open-source library that provides a high-level interface for PyTorch. Lightning abstracts away many of the lower-level distributed training configurations required for vanilla PyTorch.

Lightning allows you to run your training scripts in single GPU, single-node multi-GPU, and multi-node multi-GPU settings. Behind the scene, it launches multiple processes for you similar to `torch.distributed.launch`.

For single-node training (including single-node multi-GPU), you can run your code on Azure ML without needing to specify a `distributed_job_config`. To run an experiment using multiple nodes with multiple GPUs, there are 2 options:

- Using PyTorch configuration (recommended): Define `PyTorchConfiguration` and specify `communication_backend="Nccl"`, `node_count`, and `process_count` (note that this is the total number of processes, ie, `num_nodes * process_count_per_node`). In Lightning Trainer module, specify both `num_nodes` and `gpus` to be consistent with `PyTorchConfiguration`. For example, `num_nodes = node_count` and `gpus = process_count_per_node`.
- Using MPI Configuration:
 - Define `MpiConfiguration` and specify both `node_count` and `process_count_per_node`. In Lightning Trainer, specify both `num_nodes` and `gpus` to be respectively the same as `node_count` and `process_count_per_node` from `MpiConfiguration`.
 - For multi-node training with MPI, Lightning requires the following environment variables to be set on each node of your training cluster:
 - `MASTER_ADDR`
 - `MASTER_PORT`
 - `NODE_RANK`
 - `LOCAL_RANK`

Manually set these environment variables that Lightning requires in the main training scripts:

```

import os
from argparse import ArgumentParser

def set_environment_variables_for_mpi(num_nodes, gpus_per_node, master_port=54965):
    if num_nodes > 1:
        os.environ["MASTER_ADDR"], os.environ["MASTER_PORT"] =
os.environ["AZ_BATCH_MASTER_NODE"].split(":")
    else:
        os.environ["MASTER_ADDR"] = os.environ["AZ_BATCHAI_MPI_MASTER_NODE"]
        os.environ["MASTER_PORT"] = str(master_port)

    try:
        os.environ["NODE_RANK"] = str(int(os.environ.get("OMPI_COMM_WORLD_RANK")) // gpus_per_node)
        # additional variables
        os.environ["MASTER_ADDRESS"] = os.environ["MASTER_ADDR"]
        os.environ["LOCAL_RANK"] = os.environ["OMPI_COMM_WORLD_LOCAL_RANK"]
        os.environ["WORLD_SIZE"] = os.environ["OMPI_COMM_WORLD_SIZE"]
    except:
        # fails when used with pytorch configuration instead of mpi
        pass

if __name__ == "__main__":
    parser = ArgumentParser()
    parser.add_argument("--num_nodes", type=int, required=True)
    parser.add_argument("--gpus_per_node", type=int, required=True)
    args = parser.parse_args()
    set_environment_variables_for_mpi(args.num_nodes, args.gpus_per_node)

    trainer = Trainer(
        num_nodes=args.num_nodes,
        gpus=args.gpus_per_node
    )

```

Lightning handles computing the world size from the Trainer flags `--gpus` and `--num_nodes`.

```

from azureml.core import ScriptRunConfig, Experiment
from azureml.core.runconfig import MpiConfiguration

nnodes = 2
gpus_per_node = 4
args = ['--max_epochs', 50, '--gpus_per_node', gpus_per_node, '--accelerator', 'ddp', '--num_nodes',
nnodes]
distr_config = MpiConfiguration(node_count=nnodes, process_count_per_node=gpus_per_node)

run_config = ScriptRunConfig(
    source_directory='./src',
    script='train.py',
    arguments=args,
    compute_target=compute_target,
    environment=pytorch_env,
    distributed_job_config=distr_config,
)

run = Experiment(ws, 'experiment_name').submit(run_config)

```

Hugging Face Transformers

Hugging Face provides many [examples](#) for using its Transformers library with `torch.distributed.launch` to run distributed training. To run these examples and your own custom training scripts using the Transformers Trainer API, follow the [Using `torch.distributed.launch`](#) section.

Sample job configuration code to fine-tune the BERT large model on the text classification MNLI task using the `run_glue.py` script on one node with 8 GPUs:

```

from azureml.core import ScriptRunConfig
from azureml.core.runconfig import PyTorchConfiguration

distr_config = PyTorchConfiguration() # node_count defaults to 1
launch_cmd = "python -m torch.distributed.launch --nproc_per_node 8 text-classification/run_glue.py --
model_name_or_path bert-large-uncased-whole-word-masking --task_name mnli --do_train --do_eval --
max_seq_length 128 --per_device_train_batch_size 8 --learning_rate 2e-5 --num_train_epochs 3.0 --output_dir
/tmp/mnli_output".split()

run_config = ScriptRunConfig(
    source_directory='./src',
    command=launch_cmd,
    compute_target=compute_target,
    environment=pytorch_env,
    distributed_job_config=distr_config,
)

```

You can also use the [per-process-launch](#) option to run distributed training without using `torch.distributed.launch`. One thing to keep in mind if using this method is that the transformers [TrainingArguments](#) expect the local rank to be passed in as an argument (`--local_rank`). `torch.distributed.launch` takes care of this when `--use_env=False`, but if you are using per-process-launch you'll need to explicitly pass the local rank in as an argument to the training script `--local_rank=$LOCAL_RANK` as Azure ML only sets the `LOCAL_RANK` environment variable.

TensorFlow

If you're using [native distributed TensorFlow](#) in your training code, such as TensorFlow 2.x's `tf.distribute.Strategy` API, you can launch the distributed job via Azure ML using the `TensorflowConfiguration`.

To do so, specify a `TensorflowConfiguration` object to the `distributed_job_config` parameter of the `ScriptRunConfig` constructor. If you're using `tf.distribute.experimental.MultiWorkerMirroredStrategy`, specify the `worker_count` in the `TensorflowConfiguration` corresponding to the number of nodes for your training job.

```

from azureml.core import ScriptRunConfig, Environment, Experiment
from azureml.core.runconfig import TensorflowConfiguration

curated_env_name = 'AzureML-TensorFlow-2.3-GPU'
tf_env = Environment.get(workspace=ws, name=curated_env_name)
distr_config = TensorflowConfiguration(worker_count=2, parameter_server_count=0)

run_config = ScriptRunConfig(
    source_directory='./src',
    script='train.py',
    compute_target=compute_target,
    environment=tf_env,
    distributed_job_config=distr_config,
)

# submit the run configuration to start the job
run = Experiment(ws, "experiment_name").submit(run_config)

```

If your training script uses the parameter server strategy for distributed training, such as for legacy TensorFlow 1.x, you'll also need to specify the number of parameter servers to use in the job, for example,

```
tf_config = TensorflowConfiguration(worker_count=2, parameter_server_count=1).
```

TF_CONFIG

In TensorFlow, the `TF_CONFIG` environment variable is required for training on multiple machines. For TensorFlow jobs, Azure ML will configure and set the `TF_CONFIG` variable appropriately for each worker before

executing your training script.

You can access TF_CONFIG from your training script if you need to: `os.environ['TF_CONFIG']`.

Example TF_CONFIG set on a chief worker node:

```
TF_CONFIG='{
    "cluster": {
        "worker": [ "host0:2222", "host1:2222" ]
    },
    "task": { "type": "worker", "index": 0},
    "environment": "cloud"
}'
```

TensorFlow example

- [azureml-examples: Distributed TensorFlow training with MultiWorkerMirroredStrategy](#)

Accelerating distributed GPU training with InfiniBand

As the number of VMs training a model increases, the time required to train that model should decrease. The decrease in time, ideally, should be linearly proportional to the number of training VMs. For instance, if training a model on one VM takes 100 seconds, then training the same model on two VMs should ideally take 50 seconds. Training the model on four VMs should take 25 seconds, and so on.

InfiniBand can be an important factor in attaining this linear scaling. InfiniBand enables low-latency, GPU-to-GPU communication across nodes in a cluster. InfiniBand requires specialized hardware to operate. Certain Azure VM series, specifically the NC, ND, and H-series, now have RDMA-capable VMs with SR-IOV and InfiniBand support. These VMs communicate over the low latency and high-bandwidth InfiniBand network, which is much more performant than Ethernet-based connectivity. SR-IOV for InfiniBand enables near bare-metal performance for any MPI library (MPI is used by many distributed training frameworks and tooling, including NVIDIA's NCCL software.) These SKUs are intended to meet the needs of computationally intensive, GPU-accelerated machine learning workloads. For more information, see [Accelerating Distributed Training in Azure Machine Learning with SR-IOV](#).

Typically, VM SKUs with an 'r' in their name contain the required InfiniBand hardware, and those without an 'r' typically do not. ('r' is a reference to RDMA, which stands for "remote direct memory access.") For instance, the VM SKU `Standard_NC24rs_v3` is InfiniBand-enabled, but the SKU `Standard_NC24s_v3` is not. Aside from the InfiniBand capabilities, the specs between these two SKUs are largely the same – both have 24 cores, 448 GB RAM, 4 GPUs of the same SKU, etc. [Learn more about RDMA- and InfiniBand-enabled machine SKUs](#).

WARNING

The older-generation machine SKU `Standard_NC24r` is RDMA-enabled, but it does not contain SR-IOV hardware required for InfiniBand.

If you create an `AmlCompute` cluster of one of these RDMA-capable, InfiniBand-enabled sizes, the OS image will come with the Mellanox OFED driver required to enable InfiniBand preinstalled and preconfigured.

Next steps

- [Deploy machine learning models to Azure](#)
- [Reference architecture for distributed deep learning training in Azure](#)

Train scikit-learn models at scale with Azure Machine Learning (SDK v1)

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, learn how to run your scikit-learn training scripts with Azure Machine Learning.

The example scripts in this article are used to classify iris flower images to build a machine learning model based on scikit-learn's [iris dataset](#).

Whether you're training a machine learning scikit-learn model from the ground-up or you're bringing an existing model into the cloud, you can use Azure Machine Learning to scale out open-source training jobs using elastic cloud compute resources. You can build, deploy, version, and monitor production-grade models with Azure Machine Learning.

Prerequisites

You can run this code in either an Azure Machine Learning compute instance, or your own Jupyter Notebook:

- Azure Machine Learning compute instance
 - Complete the [Quickstart: Get started with Azure Machine Learning](#) to create a compute instance. Every compute instance includes a dedicated notebook server pre-loaded with the SDK and the notebooks sample repository.
 - Select the notebook tab in the Azure Machine Learning studio. In the samples training folder, find a completed and expanded notebook by navigating to this directory: **how-to-use-azureml > ml-frameworks > scikit-learn > train-hyperparameter-tune-deploy-with-sklearn** folder.
 - You can use the pre-populated code in the sample training folder to complete this tutorial.
- Create a Jupyter Notebook server and run the code in the following sections.
 - [Install the Azure Machine Learning SDK \(>= 1.13.0\)](#).
 - [Create a workspace configuration file](#).

Set up the experiment

This section sets up the training experiment by loading the required Python packages, initializing a workspace, defining the training environment, and preparing the training script.

Initialize a workspace

The [Azure Machine Learning workspace](#) is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the [prerequisites section](#).

```
from azureml.core import Workspace  
  
ws = Workspace.from_config()
```

Prepare scripts

In this tutorial, the [training script `train_iris.py`](#) is already provided for you. In practice, you should be able to take any custom training script as is and run it with Azure ML without having to modify your code.

NOTE

- The provided training script shows how to log some metrics to your Azure ML run using the `Run` object within the script.
- The provided training script uses example data from the `iris = datasets.load_iris()` function. To use and access your own data, see [how to train with datasets](#) to make data available during training.

Define your environment

To define the Azure ML [Environment](#) that encapsulates your training script's dependencies, you can either define a custom environment or use an Azure ML curated environment.

Use a curated environment

Optionally, Azure ML provides prebuilt, [curated environments](#) if you don't want to define your own environment.

If you want to use a curated environment, you can run the following command instead:

```
from azureml.core import Environment  
  
sklearn_env = Environment.get(workspace=ws, name='AzureML-Tutorial')
```

Create a custom environment

You can also create your own custom environment. Define your conda dependencies in a YAML file; in this example the file is named `conda_dependencies.yml`.

```
dependencies:  
- python=3.6.2  
- scikit-learn  
- numpy  
- pip:  
- azureml-defaults
```

Create an Azure ML environment from this Conda environment specification. The environment will be packaged into a Docker container at runtime.

```
from azureml.core import Environment  
  
sklearn_env = Environment.from_conda_specification(name='sklearn-env', file_path='conda_dependencies.yml')
```

For more information on creating and using environments, see [Create and use software environments in Azure Machine Learning](#).

Configure and submit your training run

Create a ScriptRunConfig

Create a `ScriptRunConfig` object to specify the configuration details of your training job, including your training script, environment to use, and the compute target to run on. Any arguments to your training script will be passed via command line if specified in the `arguments` parameter.

The following code will configure a `ScriptRunConfig` object for submitting your job for execution on your local machine.

```
from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory='.',
                      script='train_iris.py',
                      arguments=['--kernel', 'linear', '--penalty', 1.0],
                      environment=sklearn_env)
```

If you want to instead run your job on a remote cluster, you can specify the desired compute target to the `compute_target` parameter of `ScriptRunConfig`.

```
from azureml.core import ScriptRunConfig

compute_target = ws.compute_targets['<my-cluster-name>']
src = ScriptRunConfig(source_directory='.',
                      script='train_iris.py',
                      arguments=['--kernel', 'linear', '--penalty', 1.0],
                      compute_target=compute_target,
                      environment=sklearn_env)
```

Submit your run

```
from azureml.core import Experiment

run = Experiment(ws,'Tutorial-TrainIRIS').submit(src)
run.wait_for_completion(show_output=True)
```

WARNING

Azure Machine Learning runs training scripts by copying the entire source directory. If you have sensitive data that you don't want to upload, use a [.ignore file](#) or don't include it in the source directory . Instead, access your data using an [Azure ML dataset](#).

What happens during run execution

As the run is executed, it goes through the following stages:

- **Preparing:** A docker image is created according to the environment defined. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress. If a curated environment is specified instead, the cached image backing that curated environment will be used.
- **Scaling:** The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running:** All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the `script` is executed. Outputs from stdout and the `./logs` folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing:** The `./outputs` folder of the run is copied over to the run history.

Save and register the model

Once you've trained the model, you can save and register it to your workspace. Model registration lets you store and version your models in your workspace to simplify [model management and deployment](#).

Add the following code to your training script, `train_iris.py`, to save the model.

```
import joblib

joblib.dump(svm_model_linear, 'model.joblib')
```

Register the model to your workspace with the following code. By specifying the parameters `model_framework`, `model_framework_version`, and `resource_configuration`, no-code model deployment becomes available. No-code model deployment allows you to directly deploy your model as a web service from the registered model, and the `ResourceConfiguration` object defines the compute resource for the web service.

```
from azureml.core import Model
from azureml.core.resource_configuration import ResourceConfiguration

model = run.register_model(model_name='sklearn-iris',
                           model_path='outputs/model.joblib',
                           model_framework=Model.Framework.SCIKITLEARN,
                           model_framework_version='0.19.1',
                           resource_configuration=ResourceConfiguration(cpu=1, memory_in_gb=0.5))
```

Deployment

The model you just registered can be deployed the exact same way as any other registered model in Azure ML. The deployment how-to contains a section on registering models, but you can skip directly to [creating a compute target how-to-deploy-and-where.md#choose-a-compute-target) for deployment, since you already have a registered model.

(Preview) No-code model deployment

Instead of the traditional deployment route, you can also use the no-code deployment feature (preview) for scikit-learn. No-code model deployment is supported for all built-in scikit-learn model types. By registering your model as shown above with the `model_framework`, `model_framework_version`, and `resource_configuration` parameters, you can simply use the `deploy()` static function to deploy your model.

```
web_service = Model.deploy(ws, "scikit-learn-service", [model])
```

NOTE

These dependencies are included in the pre-built scikit-learn inference container.

```
- azureml-defaults
- inference-schema[numpy-support]
- scikit-learn
- numpy
```

The full [how-to](#) covers deployment in Azure Machine Learning in greater depth.

Next steps

In this article, you trained and registered a scikit-learn model, and learned about deployment options. See these other articles to learn more about Azure Machine Learning.

- [Track run metrics during training](#)
- [Tune hyperparameters](#)

Train TensorFlow models at scale with Azure Machine Learning SDK (v1)

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, learn how to run your [TensorFlow](#) training scripts at scale using Azure Machine Learning.

This example trains and registers a TensorFlow model to classify handwritten digits using a deep neural network (DNN).

Whether you're developing a TensorFlow model from the ground-up or you're bringing an [existing model](#) into the cloud, you can use Azure Machine Learning to scale out open-source training jobs to build, deploy, version, and monitor production-grade models.

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
 - Complete the [Quickstart: Get started with Azure Machine Learning](#) to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
 - In the samples deep learning folder on the notebook server, find a completed and expanded notebook by navigating to this directory: `how-to-use-azureml > ml-frameworks > tensorflow > train-hyperparameter-tune-deploy-with-tensorflow` folder.
- Your own Jupyter Notebook server
 - [Install the Azure Machine Learning SDK \(>= 1.15.0\).](#)
 - [Create a workspace configuration file.](#)
 - [Download the sample script files](#) `tf_mnist.py` and `utils.py`

You can also find a completed [Jupyter Notebook version](#) of this guide on the GitHub samples page. The notebook includes expanded sections covering intelligent hyperparameter tuning, model deployment, and notebook widgets.

Before you can run the code in this article to create a GPU cluster, you'll need to [request a quota increase](#) for your workspace.

Set up the experiment

This section sets up the training experiment by loading the required Python packages, initializing a workspace, creating the compute target, and defining the training environment.

Import packages

First, import the necessary Python libraries.

```
import os
import urllib
import shutil
import azureml

from azureml.core import Experiment
from azureml.core import Workspace, Run
from azureml.core import Environment

from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
```

Initialize a workspace

The [Azure Machine Learning workspace](#) is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the [prerequisites section](#).

```
ws = Workspace.from_config()
```

Create a file dataset

A `FileDataset` object references one or multiple files in your workspace datastore or public urls. The files can be of any format, and the class provides you with the ability to download or mount the files to your compute. By creating a `FileDataset`, you create a reference to the data source location. If you applied any transformations to the data set, they'll be stored in the data set as well. The data remains in its existing location, so no extra storage cost is incurred. For more information the `Dataset` package, see the [How to create register datasets article](#).

```
from azureml.core.dataset import Dataset

web_paths = [
    'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
]
dataset = Dataset.File.from_files(path = web_paths)
```

Use the `register()` method to register the data set to your workspace so they can be shared with others, reused across various experiments, and referred to by name in your training script.

```
dataset = dataset.register(workspace=ws,
                           name='mnist-dataset',
                           description='training and test dataset',
                           create_new_version=True)

# list the files referenced by dataset
dataset.to_path()
```

Create a compute target

Create a compute target for your TensorFlow job to run on. In this example, create a GPU-enabled Azure Machine Learning compute cluster.

IMPORTANT

Before you can create a GPU cluster, you'll need to [request a quota increase](#) for your workspace.

```
cluster_name = "gpu-cluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                           max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

NOTE

You may choose to use [low-priority VMs](#) to run some or all of your workloads. See how to [create a low-priority VM](#).

For more information on compute targets, see the [what is a compute target](#) article.

Define your environment

To define the Azure ML [Environment](#) that encapsulates your training script's dependencies, you can either define a custom environment or use an Azure ML curated environment.

Use a curated environment

Azure ML provides prebuilt, curated environments if you don't want to define your own environment. Azure ML has several CPU and GPU curated environments for TensorFlow corresponding to different versions of TensorFlow. For more info, see [Azure ML Curated Environments](#).

If you want to use a curated environment, you can run the following command instead:

```
curated_env_name = 'AzureML-TensorFlow-2.2-GPU'
tf_env = Environment.get(workspace=ws, name=curated_env_name)
```

To see the packages included in the curated environment, you can write out the conda dependencies to disk:

```
tf_env.save_to_directory(path=curated_env_name)
```

Make sure the curated environment includes all the dependencies required by your training script. If not, you'll have to modify the environment to include the missing dependencies. If the environment is modified, you'll have to give it a new name, as the 'AzureML' prefix is reserved for curated environments. If you modified the conda dependencies YAML file, you can create a new environment from it with a new name, for example:

```
tf_env = Environment.from_conda_specification(name='tensorflow-2.2-gpu',
                                               file_path='./conda_dependencies.yml')
```

If you had instead modified the curated environment object directly, you can clone that environment with a new name:

```
tf_env = tf_env.clone(new_name='tensorflow-2.2-gpu')
```

Create a custom environment

You can also create your own Azure ML environment that encapsulates your training script's dependencies.

First, define your conda dependencies in a YAML file; in this example the file is named `conda_dependencies.yml`.

```
channels:
- conda-forge
dependencies:
- python=3.6.2
- pip:
  - azureml-defaults
  - tensorflow-gpu==2.2.0
```

Create an Azure ML environment from this conda environment specification. The environment will be packaged into a Docker container at runtime.

By default if no base image is specified, Azure ML will use a CPU image

`azureml.core.environment.DEFAULT_CPU_IMAGE` as the base image. Since this example runs training on a GPU cluster, you'll need to specify a GPU base image that has the necessary GPU drivers and dependencies. Azure ML maintains a set of base images published on Microsoft Container Registry (MCR) that you can use, see the [Azure/AzureML-Containers GitHub repo](#) for more information.

```
tf_env = Environment.from_conda_specification(name='tensorflow-2.2-gpu',
file_path='./conda_dependencies.yml')

# Specify a GPU base image
tf_env.docker.enabled = True
tf_env.docker.base_image = 'mcr.microsoft.com/azureml/openmpi3.1.2-cuda10.1-cudnn7-ubuntu18.04'
```

TIP

Optionally, you can just capture all your dependencies directly in a custom Docker image or Dockerfile, and create your environment from that. For more information, see [Train with custom image](#).

For more information on creating and using environments, see [Create and use software environments in Azure Machine Learning](#).

Configure and submit your training run

Create a ScriptRunConfig

Create a `ScriptRunConfig` object to specify the configuration details of your training job, including your training script, environment to use, and the compute target to run on. Any arguments to your training script will be passed via command line if specified in the `arguments` parameter.

```
from azureml.core import ScriptRunConfig

args = ['--data-folder', dataset.as_mount(),
       '--batch-size', 64,
       '--first-layer-neurons', 256,
       '--second-layer-neurons', 128,
       '--learning-rate', 0.01]

src = ScriptRunConfig(source_directory=script_folder,
                      script='tf_mnist.py',
                      arguments=args,
                      compute_target=compute_target,
                      environment=tf_env)
```

WARNING

Azure Machine Learning runs training scripts by copying the entire source directory. If you have sensitive data that you don't want to upload, use a `.ignore` file or don't include it in the source directory. Instead, access your data using an Azure ML [dataset](#).

For more information on configuring jobs with `ScriptRunConfig`, see [Configure and submit training runs](#).

WARNING

If you were previously using the TensorFlow estimator to configure your TensorFlow training jobs, please note that Estimators have been deprecated as of the 1.19.0 SDK release. With Azure ML SDK \geq 1.15.0, `ScriptRunConfig` is the recommended way to configure training jobs, including those using deep learning frameworks. For common migration questions, see the [Estimator to ScriptRunConfig migration guide](#).

Submit a run

The [Run object](#) provides the interface to the run history while the job is running and after it has completed.

```
run = Experiment(workspace=ws, name='Tutorial-TF-Mnist').submit(src)
run.wait_for_completion(show_output=True)
```

What happens during run execution

As the run is executed, it goes through the following stages:

- **Preparing:** A docker image is created according to the environment defined. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress. If a curated environment is specified instead, the cached image backing that curated environment will be used.
- **Scaling:** The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running:** All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the `script` is executed. Outputs from `stdout` and the `./logs` folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing:** The `./outputs` folder of the run is copied over to the run history.

Register or download a model

Once you've trained the model, you can register it to your workspace. Model registration lets you store and

version your models in your workspace to simplify [model management and deployment](#).

Optional: by specifying the parameters `model_framework`, `model_framework_version`, and `resource_configuration`, no-code model deployment becomes available. This allows you to directly deploy your model as a web service from the registered model, and the `ResourceConfiguration` object defines the compute resource for the web service.

```
from azureml.core import Model
from azureml.core.resource_configuration import ResourceConfiguration

model = run.register_model(model_name='tf-mnist',
                           model_path='outputs/model',
                           model_framework=Model.Framework.TENSORFLOW,
                           model_framework_version='2.0',
                           resource_configuration=ResourceConfiguration(cpu=1, memory_in_gb=0.5))
```

You can also download a local copy of the model by using the Run object. In the training script `tf_mnist.py`, a TensorFlow saver object persists the model to a local folder (local to the compute target). You can use the Run object to download a copy.

```
# Create a model folder in the current directory
os.makedirs('./model', exist_ok=True)
run.download_files(prefix='outputs/model', output_directory='./model', append_prefix=False)
```

Distributed training

Azure Machine Learning also supports multi-node distributed TensorFlow jobs so that you can scale your training workloads. You can easily run distributed TensorFlow jobs and Azure ML will manage the orchestration for you.

Azure ML supports running distributed TensorFlow jobs with both Horovod and TensorFlow's built-in distributed training API.

For more information about distributed training, see the [Distributed GPU training guide](#).

Deploy a TensorFlow model

The deployment how-to contains a section on registering models, but you can skip directly to [creating a compute target](#) for deployment, since you already have a registered model.

(Preview) No-code model deployment

Instead of the traditional deployment route, you can also use the no-code deployment feature (preview) for TensorFlow. By registering your model as shown above with the `model_framework`, `model_framework_version`, and `resource_configuration` parameters, you can use the `deploy()` static function to deploy your model.

```
service = Model.deploy(ws, "tensorflow-web-service", [model])
```

The full [how-to](#) covers deployment in Azure Machine Learning in greater depth.

Next steps

In this article, you trained and registered a TensorFlow model, and learned about options for deployment. See these other articles to learn more about Azure Machine Learning.

- [Track run metrics during training](#)

- Tune hyperparameters
- Reference architecture for distributed deep learning training in Azure

Train Keras models at scale with Azure Machine Learning (SDK v1)

9/21/2022 • 7 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, learn how to run your Keras training scripts with Azure Machine Learning.

The example code in this article shows you how to train and register a Keras classification model built using the TensorFlow backend with Azure Machine Learning. It uses the popular [MNIST dataset](#) to classify handwritten digits using a deep neural network (DNN) built using the [Keras Python library](#) running on top of [TensorFlow](#).

Keras is a high-level neural network API capable of running top of other popular DNN frameworks to simplify development. With Azure Machine Learning, you can rapidly scale out training jobs using elastic cloud compute resources. You can also track your training runs, version models, deploy models, and much more.

Whether you're developing a Keras model from the ground-up or you're bringing an existing model into the cloud, Azure Machine Learning can help you build production-ready models.

NOTE

If you are using the Keras API `tf.keras` built into TensorFlow and not the standalone Keras package, refer instead to [Train TensorFlow models](#).

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
 - Complete the [Quickstart: Get started with Azure Machine Learning](#) to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
 - In the samples folder on the notebook server, find a completed and expanded notebook by navigating to this directory: `how-to-use-azureml > ml-frameworks > keras > train-hyperparameter-tune-deploy-with-keras` folder.
- Your own Jupyter Notebook server
 - [Install the Azure Machine Learning SDK \(>= 1.15.0\).](#)
 - [Create a workspace configuration file.](#)
 - [Download the sample script files](#) `keras_mnist.py` and `utils.py`

You can also find a completed [Jupyter Notebook version](#) of this guide on the GitHub samples page. The notebook includes expanded sections covering intelligent hyperparameter tuning, model deployment, and notebook widgets.

Before you can run the code in this article to create a GPU cluster, you'll need to [request a quota increase](#) for your workspace.

Set up the experiment

This section sets up the training experiment by loading the required Python packages, initializing a workspace,

creating the FileDataset for the input training data, creating the compute target, and defining the training environment.

Import packages

First, import the necessary Python libraries.

```
import os
import azureml
from azureml.core import Experiment
from azureml.core import Environment
from azureml.core import Workspace, Run
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
```

Initialize a workspace

The [Azure Machine Learning workspace](#) is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the [prerequisites section](#).

```
ws = Workspace.from_config()
```

Create a file dataset

A `FileDataset` object references one or multiple files in your workspace datastore or public urls. The files can be of any format, and the class provides you with the ability to download or mount the files to your compute. By creating a `FileDataset`, you create a reference to the data source location. If you applied any transformations to the data set, they will be stored in the data set as well. The data remains in its existing location, so no extra storage cost is incurred. See the [how-to guide](#) on the `Dataset` package for more information.

```
from azureml.core.dataset import Dataset

web_paths = [
    'http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
    'http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz'
]
dataset = Dataset.File.from_files(path=web_paths)
```

You can use the `register()` method to register the data set to your workspace so they can be shared with others, reused across various experiments, and referred to by name in your training script.

```
dataset = dataset.register(workspace=ws,
                           name='mnist-dataset',
                           description='training and test dataset',
                           create_new_version=True)
```

Create a compute target

Create a compute target for your training job to run on. In this example, create a GPU-enabled Azure Machine Learning compute cluster.

IMPORTANT

Before you can create a GPU cluster, you'll need to [request a quota increase](#) for your workspace.

```
cluster_name = "gpu-cluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                           max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

NOTE

You may choose to use [low-priority VMs](#) to run some or all of your workloads. See how to [create a low-priority VM](#).

For more information on compute targets, see the [what is a compute target](#) article.

Define your environment

Define the Azure ML [Environment](#) that encapsulates your training script's dependencies.

First, define your conda dependencies in a YAML file; in this example the file is named `conda_dependencies.yml`.

```
channels:
- conda-forge
dependencies:
- python=3.6.2
- pip:
  - azureml-defaults
  - tensorflow-gpu==2.0.0
  - keras<=2.3.1
  - matplotlib
```

Create an Azure ML environment from this conda environment specification. The environment will be packaged into a Docker container at runtime.

By default if no base image is specified, Azure ML will use a CPU image

`azureml.core.environment.DEFAULT_CPU_IMAGE` as the base image. Since this example runs training on a GPU cluster, you will need to specify a GPU base image that has the necessary GPU drivers and dependencies. Azure ML maintains a set of base images published on Microsoft Container Registry (MCR) that you can use, see the [Azure/AzureML-Containers](#) GitHub repo for more information.

```
keras_env = Environment.from_conda_specification(name='keras-env', file_path='conda_dependencies.yml')

# Specify a GPU base image
keras_env.docker.enabled = True
keras_env.docker.base_image = 'mcr.microsoft.com/azureml/openmpi3.1.2-cuda10.0-cudnn7-ubuntu18.04'
```

For more information on creating and using environments, see [Create and use software environments in Azure Machine Learning](#).

Configure and submit your training run

Create a ScriptRunConfig

First get the data from the workspace datastore using the `Dataset` class.

```
dataset = Dataset.get_by_name(ws, 'mnist-dataset')

# list the files referenced by mnist-dataset
dataset.to_path()
```

Create a `ScriptRunConfig` object to specify the configuration details of your training job, including your training script, environment to use, and the compute target to run on.

Any arguments to your training script will be passed via command line if specified in the `arguments` parameter. The `DatasetConsumptionConfig` for our `FileDataset` is passed as an argument to the training script, for the `--data-folder` argument. Azure ML will resolve this `DatasetConsumptionConfig` to the mount-point of the backing datastore, which can then be accessed from the training script.

```
from azureml.core import ScriptRunConfig

args = ['--data-folder', dataset.as_mount(),
        '--batch-size', 50,
        '--first-layer-neurons', 300,
        '--second-layer-neurons', 100,
        '--learning-rate', 0.001]

src = ScriptRunConfig(source_directory=script_folder,
                      script='keras_mnist.py',
                      arguments=args,
                      compute_target=compute_target,
                      environment=keras_env)
```

For more information on configuring jobs with `ScriptRunConfig`, see [Configure and submit training runs](#).

WARNING

If you were previously using the TensorFlow estimator to configure your Keras training jobs, please note that Estimators have been deprecated as of the 1.19.0 SDK release. With Azure ML SDK >= 1.15.0, `ScriptRunConfig` is the recommended way to configure training jobs, including those using deep learning frameworks. For common migration questions, see the [Estimator to ScriptRunConfig migration guide](#).

Submit your run

The `Run` object provides the interface to the run history while the job is running and after it has completed.

```
run = Experiment(workspace=ws, name='Tutorial-Keras-Minst').submit(src)
run.wait_for_completion(show_output=True)
```

What happens during run execution

As the run is executed, it goes through the following stages:

- **Preparing:** A docker image is created according to the environment defined. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress. If a curated environment is specified instead, the cached image backing that curated environment will be used.

- **Scaling:** The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running:** All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the `script` is executed. Outputs from stdout and the `./logs` folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing:** The `./outputs` folder of the run is copied over to the run history.

Register the model

Once you've trained the model, you can register it to your workspace. Model registration lets you store and version your models in your workspace to simplify [model management and deployment](#).

```
model = run.register_model(model_name='keras-mnist', model_path='outputs/model')
```

TIP

The deployment how-to contains a section on registering models, but you can skip directly to [creating a compute target](#) for deployment, since you already have a registered model.

You can also download a local copy of the model. This can be useful for doing additional model validation work locally. In the training script, `keras_mnist.py`, a TensorFlow saver object persists the model to a local folder (local to the compute target). You can use the Run object to download a copy from the run history.

```
# Create a model folder in the current directory
os.makedirs('./model', exist_ok=True)

for f in run.get_file_names():
    if f.startswith('outputs/model'):
        output_file_path = os.path.join('./model', f.split('/')[-1])
        print('Downloading from {} to {}'.format(f, output_file_path))
        run.download_file(name=f, output_file_path=output_file_path)
```

Next steps

In this article, you trained and registered a Keras model on Azure Machine Learning. To learn how to deploy a model, continue on to our model deployment article.

- [How and where to deploy models](#)
- [Track run metrics during training](#)
- [Tune hyperparameters](#)
- [Reference architecture for distributed deep learning training in Azure](#)

Train PyTorch models at scale with Azure Machine Learning SDK (v1)

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, learn how to run your [PyTorch](#) training scripts at enterprise scale using Azure Machine Learning.

The example scripts in this article are used to classify chicken and turkey images to build a deep learning neural network (DNN) based on [PyTorch's transfer learning tutorial](#). Transfer learning is a technique that applies knowledge gained from solving one problem to a different but related problem. Transfer learning shortens the training process by requiring less data, time, and compute resources than training from scratch. To learn more about transfer learning, see the [deep learning vs machine learning](#) article.

Whether you're training a deep learning PyTorch model from the ground-up or you're bringing an existing model into the cloud, you can use Azure Machine Learning to scale out open-source training jobs using elastic cloud compute resources. You can build, deploy, version, and monitor production-grade models with Azure Machine Learning.

Prerequisites

Run this code on either of these environments:

- Azure Machine Learning compute instance - no downloads or installation necessary
 - Complete the [Quickstart: Get started with Azure Machine Learning](#) to create a dedicated notebook server pre-loaded with the SDK and the sample repository.
 - In the samples deep learning folder on the notebook server, find a completed and expanded notebook by navigating to this directory: `how-to-use-azureml > ml-frameworks > pytorch > train-hyperparameter-tune-deploy-with-pytorch` folder.
- Your own Jupyter Notebook server
 - [Install the Azure Machine Learning SDK \(>= 1.15.0\)](#).
 - [Create a workspace configuration file](#).
 - [Download the sample script files](#) `pytorch_train.py`

You can also find a completed [Jupyter Notebook version](#) of this guide on the GitHub samples page. The notebook includes expanded sections covering intelligent hyperparameter tuning, model deployment, and notebook widgets.

Before you can run the code in this article to create a GPU cluster, you'll need to [request a quota increase](#) for your workspace.

Set up the experiment

This section sets up the training experiment by loading the required Python packages, initializing a workspace, creating the compute target, and defining the training environment.

Import packages

First, import the necessary Python libraries.

```
import os
import shutil

from azureml.core.workspace import Workspace
from azureml.core import Experiment
from azureml.core import Environment

from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException
```

Initialize a workspace

The [Azure Machine Learning workspace](#) is the top-level resource for the service. It provides you with a centralized place to work with all the artifacts you create. In the Python SDK, you can access the workspace artifacts by creating a `workspace` object.

Create a workspace object from the `config.json` file created in the [prerequisites section](#).

```
ws = Workspace.from_config()
```

Get the data

The dataset consists of about 120 training images each for turkeys and chickens, with 100 validation images for each class. We'll download and extract the dataset as part of our training script `pytorch_train.py`. The images are a subset of the [Open Images v5 Dataset](#). For more steps on creating a JSONL to train with your own data, see this [Jupyter notebook](#).

Prepare training script

In this tutorial, the training script, `pytorch_train.py`, is already provided. In practice, you can take any custom training script, as is, and run it with Azure Machine Learning.

Create a folder for your training script(s).

```
project_folder = './pytorch-birds'
os.makedirs(project_folder, exist_ok=True)
shutil.copy('pytorch_train.py', project_folder)
```

Create a compute target

Create a compute target for your PyTorch job to run on. In this example, create a GPU-enabled Azure Machine Learning compute cluster.

IMPORTANT

Before you can create a GPU cluster, you'll need to [request a quota increase](#) for your workspace.

```

# Choose a name for your CPU cluster
cluster_name = "gpu-cluster"

# Verify that cluster does not exist already
try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                          max_nodes=4)

    # Create the cluster with the specified name and configuration
    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    # Wait for the cluster to complete, show the output log
    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

```

If you instead want to create a CPU cluster, provide a different VM size to the `vm_size` parameter, such as `STANDARD_D2_V2`.

NOTE

You may choose to use [low-priority VMs](#) to run some or all of your workloads. See how to [create a low-priority VM](#).

For more information on compute targets, see the [what is a compute target](#) article.

Define your environment

To define the [Azure ML Environment](#) that encapsulates your training script's dependencies, you can either define a custom environment or use an Azure ML curated environment.

Use a curated environment

Azure ML provides prebuilt, [curated environments](#) if you don't want to define your own environment. There are several CPU and GPU curated environments for PyTorch corresponding to different versions of PyTorch.

If you want to use a curated environment, you can run the following command instead:

```

curated_env_name = 'AzureML-PyTorch-1.6-GPU'
pytorch_env = Environment.get(workspace=ws, name=curated_env_name)

```

To see the packages included in the curated environment, you can write out the conda dependencies to disk:

```

pytorch_env.save_to_directory(path=curated_env_name)

```

Make sure the curated environment includes all the dependencies required by your training script. If not, you'll have to modify the environment to include the missing dependencies. If the environment is modified, you'll have to give it a new name, as the 'AzureML' prefix is reserved for curated environments. If you modified the conda dependencies YAML file, you can create a new environment from it with a new name, for example:

```

pytorch_env = Environment.from_conda_specification(name='pytorch-1.6-gpu',
                                                    file_path='./conda_dependencies.yml')

```

If you had instead modified the curated environment object directly, you can clone that environment with a new name:

```
pytorch_env = pytorch_env.clone(new_name='pytorch-1.6-gpu')
```

Create a custom environment

You can also create your own Azure ML environment that encapsulates your training script's dependencies.

First, define your conda dependencies in a YAML file; in this example the file is named `conda_dependencies.yml`.

```
channels:
- conda-forge
dependencies:
- python=3.6.2
- pip=21.3.1
- pip:
  - azureml-defaults
  - torch==1.6.0
  - torchvision==0.7.0
  - future==0.17.1
- pillow
```

Create an Azure ML environment from this conda environment specification. The environment will be packaged into a Docker container at runtime.

By default if no base image is specified, Azure ML will use a CPU image

`azureml.core.environment.DEFAULT_CPU_IMAGE` as the base image. Since this example runs training on a GPU cluster, you'll need to specify a GPU base image that has the necessary GPU drivers and dependencies. Azure ML maintains a set of base images published on Microsoft Container Registry (MCR) that you can use. For more information, see [AzureML-Containers GitHub repo](#).

```
pytorch_env = Environment.from_conda_specification(name='pytorch-1.6-gpu',
file_path='./conda_dependencies.yml')

# Specify a GPU base image
pytorch_env.docker.enabled = True
pytorch_env.docker.base_image = 'mcr.microsoft.com/azureml/openmpi3.1.2-cuda10.1-cudnn7-ubuntu18.04'
```

TIP

Optionally, you can just capture all your dependencies directly in a custom Docker image or Dockerfile, and create your environment from that. For more information, see [Train with custom image](#).

For more information on creating and using environments, see [Create and use software environments in Azure Machine Learning](#).

Configure and submit your training run

Create a ScriptRunConfig

Create a `ScriptRunConfig` object to specify the configuration details of your training job, including your training script, environment to use, and the compute target to run on. Any arguments to your training script will be passed via command line if specified in the `arguments` parameter. The following code will configure a single-node PyTorch job.

```
from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory=project_folder,
                      script='pytorch_train.py',
                      arguments=['--num_epochs', 30, '--output_dir', './outputs'],
                      compute_target=compute_target,
                      environment=pytorch_env)
```

WARNING

Azure Machine Learning runs training scripts by copying the entire source directory. If you have sensitive data that you don't want to upload, use a [.ignore file](#) or don't include it in the source directory . Instead, access your data using an Azure ML [dataset](#).

For more information on configuring jobs with ScriptRunConfig, see [Configure and submit training runs](#).

WARNING

If you were previously using the PyTorch estimator to configure your PyTorch training jobs, please note that Estimators have been deprecated as of the 1.19.0 SDK release. With Azure ML SDK >= 1.15.0, ScriptRunConfig is the recommended way to configure training jobs, including those using deep learning frameworks. For common migration questions, see the [Estimator to ScriptRunConfig migration guide](#).

Submit your run

The [Run object](#) provides the interface to the run history while the job is running and after it has completed.

```
run = Experiment(ws, name='Tutorial-pytorch-birds').submit(src)
run.wait_for_completion(show_output=True)
```

What happens during run execution

As the run is executed, it goes through the following stages:

- **Preparing:** A docker image is created according to the environment defined. The image is uploaded to the workspace's container registry and cached for later runs. Logs are also streamed to the run history and can be viewed to monitor progress. If a curated environment is specified instead, the cached image backing that curated environment will be used.
- **Scaling:** The cluster attempts to scale up if the Batch AI cluster requires more nodes to execute the run than are currently available.
- **Running:** All scripts in the script folder are uploaded to the compute target, data stores are mounted or copied, and the `script` is executed. Outputs from stdout and the `./logs` folder are streamed to the run history and can be used to monitor the run.
- **Post-Processing:** The `./outputs` folder of the run is copied over to the run history.

Register or download a model

Once you've trained the model, you can register it to your workspace. Model registration lets you store and version your models in your workspace to simplify [model management and deployment](#).

```
model = run.register_model(model_name='pytorch-birds', model_path='outputs/model.pt')
```

TIP

The deployment how-to contains a section on registering models, but you can skip directly to [creating a compute target](#) for deployment, since you already have a registered model.

You can also download a local copy of the model by using the Run object. In the training script

`pytorch_train.py`, a PyTorch save object persists the model to a local folder (local to the compute target). You can use the Run object to download a copy.

```
# Create a model folder in the current directory
os.makedirs('./model', exist_ok=True)

# Download the model from run history
run.download_file(name='outputs/model.pt', output_file_path='./model/model.pt'),
```

Distributed training

Azure Machine Learning also supports multi-node distributed PyTorch jobs so that you can scale your training workloads. You can easily run distributed PyTorch jobs and Azure ML will manage the orchestration for you.

Azure ML supports running distributed PyTorch jobs with both Horovod and PyTorch's built-in `DistributedDataParallel` module.

For more information about distributed training, see the [Distributed GPU training guide](#).

Export to ONNX

To optimize inference with the [ONNX Runtime](#), convert your trained PyTorch model to the ONNX format. Inference, or model scoring, is the phase where the deployed model is used for prediction, most commonly on production data. For an example, see the [Exporting model from PyTorch to ONNX tutorial](#).

Next steps

In this article, you trained and registered a deep learning, neural network using PyTorch on Azure Machine Learning. To learn how to deploy a model, continue on to our model deployment article.

- [How and where to deploy models](#)
- [Track run metrics during training](#)
- [Tune hyperparameters](#)
- [Reference architecture for distributed deep learning training in Azure](#)

Migrating from Estimators to ScriptRunConfig

9/21/2022 • 3 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

Up until now, there have been multiple methods for configuring a training job in Azure Machine Learning via the SDK, including Estimators, ScriptRunConfig, and the lower-level RunConfiguration. To address this ambiguity and inconsistency, we are simplifying the job configuration process in Azure ML. You should now use ScriptRunConfig as the recommended option for configuring training jobs.

Estimators are deprecated with the 1.19. release of the Python SDK. You should also generally avoid explicitly instantiating a RunConfiguration object yourself, and instead configure your job using the ScriptRunConfig class.

This article covers common considerations when migrating from Estimators to ScriptRunConfig.

IMPORTANT

To migrate to ScriptRunConfig from Estimators, make sure you are using >= 1.15.0 of the Python SDK.

ScriptRunConfig documentation and samples

Azure Machine Learning documentation and samples have been updated to use [ScriptRunConfig](#) for job configuration and submission.

For information on using ScriptRunConfig, refer to the following documentation:

- [Configure and submit training jobs](#)
- [Configuring PyTorch training jobs](#)
- [Configuring TensorFlow training jobs](#)
- [Configuring scikit-learn training jobs](#)

In addition, refer to the following samples & tutorials:

- [Azure/MachineLearningNotebooks](#)
- [Azure/azureml-examples](#)

Defining the training environment

While the various framework estimators have preconfigured environments that are backed by Docker images, the Dockerfiles for these images are private. Therefore you do not have a lot of transparency into what these environments contain. In addition, the estimators take in environment-related configurations as individual parameters (such as `pip_packages`, `custom_docker_image`) on their respective constructors.

When using ScriptRunConfig, all environment-related configurations are encapsulated in the `Environment` object that gets passed into the `environment` parameter of the ScriptRunConfig constructor. To configure a training job, provide an environment that has all the dependencies required for your training script. If no environment is provided, Azure ML will use one of the Azure ML base images, specifically the one defined by `azureml.core.environment.DEFAULT_CPU_IMAGE`, as the default environment. There are a couple of ways to provide an environment:

- [Use a curated environment](#) - curated environments are predefined environments available in your

workspace by default. There is a corresponding curated environment for each of the preconfigured framework/version Docker images that backed each framework estimator.

- [Define your own custom environment](#)

Here is an example of using the curated environment for training:

```
from azureml.core import Workspace, ScriptRunConfig, Environment

curated_env_name = '<add Pytorch curated environment name here>'
pytorch_env = Environment.get(workspace=ws, name=curated_env_name)

compute_target = ws.compute_targets['my-cluster']
src = ScriptRunConfig(source_directory='.',
                      script='train.py',
                      compute_target=compute_target,
                      environment=pytorch_env)
```

TIP

For a list of curated environments, see [curated environments](#).

If you want to specify **environment variables** that will get set on the process where the training script is executed, use the Environment object:

```
myenv.environment_variables = {"MESSAGE": "Hello from Azure Machine Learning"}
```

For information on configuring and managing Azure ML environments, see:

- [How to use environments](#)
- [Curated environments](#)
- [Train with a custom Docker image](#)

Using data for training

Datasets

If you are using an Azure ML dataset for training, pass the dataset as an argument to your script using the `arguments` parameter. By doing so, you will get the data path (mounting point or download path) in your training script via arguments.

The following example configures a training job where the FileDataset, `mnist_ds`, will get mounted on the remote compute.

```
src = ScriptRunConfig(source_directory='.',
                      script='train.py',
                      arguments=['--data-folder', mnist_ds.as_mount()], # or mnist_ds.as_download() to
download
                      compute_target=compute_target,
                      environment=pytorch_env)
```

DataReference (old)

While we recommend using Azure ML Datasets over the old DataReference way, if you are still using DataReferences for any reason, you must configure your job as follows:

```

# if you want to pass a DataReference object, such as the below:
datastore = ws.get_default_datastore()
data_ref = datastore.path('./foo').as_mount()

src = ScriptRunConfig(source_directory='.',
                      script='train.py',
                      arguments=['--data-folder', str(data_ref)], # cast the DataReference object to str
                      compute_target=compute_target,
                      environment=pytorch_env)
src.run_config.data_references = {data_ref.data_reference_name: data_ref.to_config()} # set a dict of the
DataReference(s) you want to the `data_references` attribute of the ScriptRunConfig's underlying
RunConfiguration object.

```

For more information on using data for training, see:

- [Train with datasets in Azure ML](#)

Distributed training

If you need to configure a distributed job for training, do so by specifying the `distributed_job_config` parameter in the `ScriptRunConfig` constructor. Pass in an [Mpiconfiguration](#), [PyTorchConfiguration](#), or [TensorflowConfiguration](#) for distributed jobs of the respective types.

The following example configures a PyTorch training job to use distributed training with MPI/Horovod:

```

from azureml.core.runconfig import Mpiconfiguration

src = ScriptRunConfig(source_directory='.',
                      script='train.py',
                      compute_target=compute_target,
                      environment=pytorch_env,
                      distributed_job_config=Mpiconfiguration(node_count=2, process_count_per_node=2))

```

For more information, see:

- [Distributed training with PyTorch](#)
- [Distributed training with TensorFlow](#)

Miscellaneous

If you need to access the underlying `RunConfiguration` object for a `ScriptRunConfig` for any reason, you can do so as follows:

```
src.run_config
```

Next steps

- [Configure and submit training jobs](#)

Use authentication credential secrets in Azure Machine Learning training jobs

9/21/2022 • 2 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to use secrets in training jobs securely. Authentication information such as your user name and password are secrets. For example, if you connect to an external database in order to query training data, you would need to pass your username and password to the remote job context. Coding such values into training scripts in cleartext is insecure as it would expose the secret.

Instead, your Azure Machine Learning workspace has an associated resource called a [Azure Key Vault](#). Use this Key Vault to pass secrets to remote jobs securely through a set of APIs in the Azure Machine Learning Python SDK.

The standard flow for using secrets is:

1. On local computer, log in to Azure and connect to your workspace.
2. On local computer, set a secret in Workspace Key Vault.
3. Submit a remote job.
4. Within the remote job, get the secret from Key Vault and use it.

Set secrets

In the Azure Machine Learning, the [KeyVault](#) class contains methods for setting secrets. In your local Python session, first obtain a reference to your workspace Key Vault, and then use the [set_secret\(\)](#) method to set a secret by name and value. The [set_secret](#) method updates the secret value if the name already exists.

```
from azureml.core import Workspace
from azureml.core import Keyvault
import os

ws = Workspace.from_config()
my_secret = os.environ.get("MY_SECRET")
keyvault = ws.get_default_keyvault()
keyvault.set_secret(name="mysecret", value = my_secret)
```

Do not put the secret value in your Python code as it is insecure to store it in file as cleartext. Instead, obtain the secret value from an environment variable, for example Azure DevOps build secret, or from interactive user input.

You can list secret names using the [list_secrets\(\)](#) method and there is also a batch version, [set_secrets\(\)](#) that allows you to set multiple secrets at a time.

IMPORTANT

Using `list_secrets()` will only list secrets created through `set_secret()` or `set_secrets()` using the Azure ML SDK. It will not list secrets created by something other than the SDK. For example, a secret created using the Azure portal or Azure PowerShell will not be listed.

You can use `get_secret()` to get a secret value from the key vault, regardless of how it was created. So you can retrieve secrets that are not listed by `list_secrets()`.

Get secrets

In your local code, you can use the `get_secret()` method to get the secret value by name.

For jobs submitted the `Experiment.submit`, use the `get_secret()` method with the `Run` class. Because a submitted run is aware of its workspace, this method shortcuts the Workspace instantiation and returns the secret value directly.

```
# Code in submitted job
from azureml.core import Experiment, Run

run = Run.get_context()
secret_value = run.get_secret(name="mysecret")
```

Be careful not to expose the secret value by writing or printing it out.

There is also a batch version, `get_secrets()` for accessing multiple secrets at once.

Next steps

- [View example notebook](#)
- [Learn about enterprise security with Azure Machine Learning](#)

Set up AutoML training with Python

9/21/2022 • 21 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this guide, learn how to set up an automated machine learning, AutoML, training run with the [Azure Machine Learning Python SDK](#) using Azure Machine Learning automated ML. Automated ML picks an algorithm and hyperparameters for you and generates a model ready for deployment. This guide provides details of the various options that you can use to configure automated ML experiments.

For an end to end example, see [Tutorial: AutoML- train regression model](#).

If you prefer a no-code experience, you can also [Set up no-code AutoML training in the Azure Machine Learning studio](#).

Prerequisites

For this article you need,

- An Azure Machine Learning workspace. To create the workspace, see [Create workspace resources](#).
- The Azure Machine Learning Python SDK installed. To install the SDK you can either,
 - Create a compute instance, which automatically installs the SDK and is preconfigured for ML workflows. See [Create and manage an Azure Machine Learning compute instance](#) for more information.
 - [Install the `automl` package yourself](#), which includes the [default installation](#) of the SDK.

IMPORTANT

The Python commands in this article require the latest `azureml-train-automl` package version.

- [Install the latest `azureml-train-automl` package to your local environment](#).
- For details on the latest `azureml-train-automl` package, see the [release notes](#).

WARNING

Python 3.8 is not compatible with `automl`.

Select your experiment type

Before you begin your experiment, you should determine the kind of machine learning problem you are solving. Automated machine learning supports task types of `classification`, `regression`, and `forecasting`. Learn more about [task types](#).

NOTE

Support for computer vision tasks: image classification (multi-class and multi-label), object detection, and instance segmentation is available in public preview. [Learn more about computer vision tasks in automated ML](#).

Support for natural language processing (NLP) tasks: image classification (multi-class and multi-label) and named entity recognition is available in public preview. [Learn more about NLP tasks in automated ML](#).

These preview capabilities are provided without a service-level agreement. Certain features might not be supported or might have constrained functionality. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

The following code uses the `task` parameter in the `AutoMLConfig` constructor to specify the experiment type as `classification`.

```
from azureml.train.automl import AutoMLConfig

# task can be one of classification, regression, forecasting
automl_config = AutoMLConfig(task = "classification")
```

Data source and format

Automated machine learning supports data that resides on your local desktop or in the cloud such as Azure Blob Storage. The data can be read into a **Pandas DataFrame** or an **Azure Machine Learning TabularDataset**. [Learn more about datasets](#).

Requirements for training data in machine learning:

- Data must be in tabular form.
- The value to predict, target column, must be in the data.

IMPORTANT

Automated ML experiments do not support training with datasets that use [identity-based data access](#).

For **remote experiments**, training data must be accessible from the remote compute. Automated ML only accepts **Azure Machine Learning TabularDatasets** when working on a remote compute.

Azure Machine Learning datasets expose functionality to:

- Easily transfer data from static files or URL sources into your workspace.
- Make your data available to training scripts when running on cloud compute resources. See [How to train with datasets](#) for an example of using the `Dataset` class to mount data to your remote compute target.

The following code creates a TabularDataset from a web url. See [Create a TabularDataset](#) for code examples on how to create datasets from other sources like local files and datastores.

```
from azureml.core.dataset import Dataset
data = "https://automlsamplenotebookdata.blob.core.windows.net/automl-sample-notebook-data/creditcard.csv"
dataset = Dataset.Tabular.from_delimited_files(data)
```

For **local compute experiments**, we recommend pandas dataframes for faster processing times.

```

import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv("your-local-file.csv")
train_data, test_data = train_test_split(df, test_size=0.1, random_state=42)
label = "label-col-name"

```

Training, validation, and test data

You can specify separate **training data** and **validation data** sets directly in the `AutoMLConfig` constructor.

Learn more about [how to configure training, validation, cross validation, and test data](#) for your AutoML experiments.

If you do not explicitly specify a `validation_data` or `n_cross_validation` parameter, automated ML applies default techniques to determine how validation is performed. This determination depends on the number of rows in the dataset assigned to your `training_data` parameter.

TRAINING DATA SIZE	VALIDATION TECHNIQUE
Larger than 20,000 rows	Train/validation data split is applied. The default is to take 10% of the initial training data set as the validation set. In turn, that validation set is used for metrics calculation.
Smaller than 20,000 rows	Cross-validation approach is applied. The default number of folds depends on the number of rows. If the dataset is less than 1,000 rows , 10 folds are used. If the rows are between 1,000 and 20,000 , then three folds are used.

TIP

You can upload **test data (preview)** to evaluate models that automated ML generated for you. These features are [experimental](#) preview capabilities, and may change at any time. Learn how to:

- Pass in test data to your `AutoMLConfig` object.
- Test the models automated ML generated for your experiment.

If you prefer a no-code experience, see [step 12 in Set up AutoML with the studio UI](#)

Large data

Automated ML supports a limited number of algorithms for training on large data that can successfully build models for big data on small virtual machines. Automated ML heuristics depend on properties such as data size, virtual machine memory size, experiment timeout and featurization settings to determine if these large data algorithms should be applied. [Learn more about what models are supported in automated ML](#).

- For regression, [Online Gradient Descent Regressor](#) and [Fast Linear Regressor](#)
- For classification, [Averaged Perceptron Classifier](#) and [Linear SVM Classifier](#); where the Linear SVM classifier has both large data and small data versions.

If you want to override these heuristics, apply the following settings:

TASK	SETTING	NOTES
Block data streaming algorithms	<code>blocked_models</code> in your <code>AutoMLConfig</code> object and list the model(s) you don't want to use.	Results in either run failure or long run time
Use data streaming algorithms	<code>allowed_models</code> in your <code>AutoMLConfig</code> object and list the model(s) you want to use.	
Use data streaming algorithms (studio UI experiments)	Block all models except the big data algorithms you want to use.	

Compute to run experiment

Next determine where the model will be trained. An automated ML training experiment can run on the following compute options.

- **Choose a local compute:** If your scenario is about initial explorations or demos using small data and short trains (i.e. seconds or a couple of minutes per child run), training on your local computer might be a better choice. There is no setup time, the infrastructure resources (your PC or VM) are directly available. See [this notebook](#) for a local compute example.
- **Choose a remote ML compute cluster:** If you are training with larger datasets like in production training creating models which need longer trains, remote compute will provide much better end-to-end time performance because `AutoML` will parallelize trains across the cluster's nodes. On a remote compute, the start-up time for the internal infrastructure will add around 1.5 minutes per child run, plus additional minutes for the cluster infrastructure if the VMs are not yet up and running. [Azure Machine Learning Managed Compute](#) is a managed service that enables the ability to train machine learning models on clusters of Azure virtual machines. Compute instance is also supported as a compute target.
- An **Azure Databricks cluster** in your Azure subscription. You can find more details in [Set up an Azure Databricks cluster for automated ML](#). See this [GitHub site](#) for examples of notebooks with Azure Databricks.

Consider these factors when choosing your compute target:

	PROS (ADVANTAGES)	CONS (HANDICAPS)
Local compute target	<ul style="list-style-type: none"> • No environment start-up time 	<ul style="list-style-type: none"> • Subset of features • Can't parallelize runs • Worse for large data. • No data streaming while training • No DNN-based featurization • Python SDK only
Remote ML compute clusters	<ul style="list-style-type: none"> • Full set of features • Parallelize child runs • Large data support • DNN-based featurization • Dynamic scalability of compute cluster on demand • No-code experience (web UI) also available 	<ul style="list-style-type: none"> • Start-up time for cluster nodes • Start-up time for each child run

Configure your experiment settings

There are several options that you can use to configure your automated ML experiment. These parameters are set by instantiating an `AutoMLConfig` object. See the [AutoMLConfig class](#) for a full list of parameters.

The following example is for a classification task. The experiment uses AUC weighted as the **primary metric** and has an experiment time out set to 30 minutes and 2 cross-validation folds.

```
automl_classifier=AutoMLConfig(task='classification',
                                primary_metric='AUC_weighted',
                                experiment_timeout_minutes=30,
                                blocked_models=['XGBoostClassifier'],
                                training_data=train_data,
                                label_column_name=label,
                                n_cross_validations=2)
```

You can also configure forecasting tasks, which requires extra setup. See the [Set up AutoML for time-series forecasting](#) article for more details.

```
time_series_settings = {
    'time_column_name': time_column_name,
    'time_series_id_column_names': time_series_id_column_names,
    'forecast_horizon': n_test_periods
}

automl_config = AutoMLConfig(
    task = 'forecasting',
    debug_log='automl_oj_sales_errors.log',
    primary_metric='normalized_root_mean_squared_error',
    experiment_timeout_minutes=20,
    training_data=train_data,
    label_column_name=label,
    n_cross_validations=5,
    path=project_folder,
    verbosity=logging.INFO,
    **time_series_settings
)
```

Supported models

Automated machine learning tries different models and algorithms during the automation and tuning process. As a user, there is no need for you to specify the algorithm.

The three different `task` parameter values determine the list of algorithms, or models, to apply. Use the `allowed_models` or `blocked_models` parameters to further modify iterations with the available models to include or exclude. The following table summarizes the supported models by task type.

NOTE

If you plan to export your automated ML created models to an [ONNX model](#), only those algorithms indicated with an * (asterisk) are able to be converted to the ONNX format. Learn more about [converting models to ONNX](#).

Also note, ONNX only supports classification and regression tasks at this time.

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
Logistic Regression*	Elastic Net*	AutoARIMA

CLASSIFICATION	REGRESSION	TIME SERIES FORECASTING
Light GBM*	Light GBM*	Prophet
Gradient Boosting*	Gradient Boosting*	Elastic Net
Decision Tree*	Decision Tree*	Light GBM
K Nearest Neighbors*	K Nearest Neighbors*	Gradient Boosting
Linear SVC*	LARS Lasso*	Decision Tree
Support Vector Classification (SVC)*	Stochastic Gradient Descent (SGD)*	Arimax
Random Forest*	Random Forest	LARS Lasso
Extremely Randomized Trees*	Extremely Randomized Trees*	Stochastic Gradient Descent (SGD)
Xgboost*	Xgboost*	Random Forest
Averaged Perceptron Classifier	Online Gradient Descent Regressor	Xgboost
Naive Bayes*	Fast Linear Regressor	ForecastTCN
Stochastic Gradient Descent (SGD)*		Naive
Linear SVM Classifier*		SeasonalNaive
		Average
		SeasonalAverage
		ExponentialSmoothing

Primary metric

The `primary_metric` parameter determines the metric to be used during model training for optimization. The available metrics you can select is determined by the task type you choose.

Choosing a primary metric for automated ML to optimize depends on many factors. We recommend your primary consideration be to choose a metric that best represents your business needs. Then consider if the metric is suitable for your dataset profile (data size, range, class distribution, etc.). The following sections summarize the recommended primary metrics based on task type and business scenario.

Learn about the specific definitions of these metrics in [Understand automated machine learning results](#).

Metrics for classification scenarios

Threshold-dependent metrics, like `accuracy`, `recall_score_weighted`, `norm_macro_recall`, and `precision_score_weighted` may not optimize as well for datasets that are small, have very large class skew (class imbalance), or when the expected metric value is very close to 0.0 or 1.0. In those cases, `AUC_weighted` can be a better choice for the primary metric. After automated ML completes, you can choose the winning model based on the metric best suited to your business needs.

METRIC	EXAMPLE USE CASE(S)
<code>accuracy</code>	Image classification, Sentiment analysis, Churn prediction
<code>AUC_weighted</code>	Fraud detection, Image classification, Anomaly detection/spam detection
<code>average_precision_score_weighted</code>	Sentiment analysis
<code>norm_macro_recall</code>	Churn prediction
<code>precision_score_weighted</code>	

Metrics for regression scenarios

`r2_score`, `normalized_mean_absolute_error` and `normalized_root_mean_squared_error` are all trying to minimize prediction errors. `r2_score` and `normalized_root_mean_squared_error` are both minimizing average squared errors while `normalized_mean_absolute_error` is minimizing the average absolute value of errors. Absolute value treats errors at all magnitudes alike and squared errors will have a much larger penalty for errors with larger absolute values. Depending on whether larger errors should be punished more or not, one can choose to optimize squared error or absolute error.

The main difference between `r2_score` and `normalized_root_mean_squared_error` is the way they are normalized and their meanings. `normalized_root_mean_squared_error` is root mean squared error normalized by range and can be interpreted as the average error magnitude for prediction. `r2_score` is mean squared error normalized by an estimate of variance of data. It is the proportion of variation that can be captured by the model.

NOTE

`r2_score` and `normalized_root_mean_squared_error` also behave similarly as primary metrics. If a fixed validation set is applied, these two metrics are optimizing the same target, mean squared error, and will be optimized by the same model. When only a training set is available and cross-validation is applied, they would be slightly different as the normalizer for `normalized_root_mean_squared_error` is fixed as the range of training set, but the normalizer for `r2_score` would vary for every fold as it's the variance for each fold.

If the rank, instead of the exact value is of interest, `spearman_correlation` can be a better choice as it measures the rank correlation between real values and predictions.

However, currently no primary metrics for regression addresses relative difference. All of `r2_score`, `normalized_mean_absolute_error`, and `normalized_root_mean_squared_error` treat a \$20k prediction error the same for a worker with a \$30k salary as a worker making \$20M, if these two data points belongs to the same dataset for regression, or the same time series specified by the time series identifier. While in reality, predicting only \$20k off from a \$20M salary is very close (a small 0.1% relative difference), whereas \$20k off from \$30k is not close (a large 67% relative difference). To address the issue of relative difference, one can train a model with available primary metrics, and then select the model with best `mean_absolute_percentage_error` or `root_mean_squared_log_error`.

METRIC	EXAMPLE USE CASE(S)
<code>spearman_correlation</code>	
<code>normalized_root_mean_squared_error</code>	Price prediction (house/product/tip), Review score prediction

METRIC	EXAMPLE USE CASE(S)
r2_score	Airline delay, Salary estimation, Bug resolution time
normalized_mean_absolute_error	

Metrics for time series forecasting scenarios

The recommendations are similar to those noted for regression scenarios.

METRIC	EXAMPLE USE CASE(S)
normalized_root_mean_squared_error	Price prediction (forecasting), Inventory optimization, Demand forecasting
r2_score	Price prediction (forecasting), Inventory optimization, Demand forecasting
normalized_mean_absolute_error	

Data featurization

In every automated ML experiment, your data is automatically scaled and normalized to help *certain* algorithms that are sensitive to features that are on different scales. This scaling and normalization is referred to as featurization. See [Featurization in AutoML](#) for more detail and code examples.

NOTE

Automated machine learning featurization steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same featurization steps applied during training are applied to your input data automatically.

When configuring your experiments in your `AutoMLConfig` object, you can enable/disable the setting `featurization`. The following table shows the accepted settings for featurization in the `AutoMLConfig` object.

FEATURIZATION CONFIGURATION	DESCRIPTION
<code>"featurization": "auto"</code>	Indicates that as part of preprocessing, data guardrails and featurization steps are performed automatically. Default setting .
<code>"featurization": "off"</code>	Indicates featurization step shouldn't be done automatically.
<code>"featurization": "FeaturizationConfig"</code>	Indicates customized featurization step should be used. Learn how to customize featurization .

Ensemble configuration

Ensemble models are enabled by default, and appear as the final run iterations in an AutoML run. Currently **VotingEnsemble** and **StackEnsemble** are supported.

Voting implements soft-voting, which uses weighted averages. The stacking implementation uses a two layer implementation, where the first layer has the same models as the voting ensemble, and the second layer model is used to find the optimal combination of the models from the first layer.

If you are using ONNX models, or have model-explainability enabled, stacking is disabled and only voting is

utilized.

Ensemble training can be disabled by using the `enable_voting_ensemble` and `enable_stack_ensemble` boolean parameters.

```
automl_classifier = AutoMLConfig(  
    task='classification',  
    primary_metric='AUC_weighted',  
    experiment_timeout_minutes=30,  
    training_data=data_train,  
    label_column_name=label,  
    n_cross_validations=5,  
    enable_voting_ensemble=False,  
    enable_stack_ensemble=False  
)
```

To alter the default ensemble behavior, there are multiple default arguments that can be provided as `kwargs` in an `AutoMLConfig` object.

IMPORTANT

The following parameters aren't explicit parameters of the `AutoMLConfig` class.

- `ensemble_download_models_timeout_sec` : During **VotingEnsemble** and **StackEnsemble** model generation, multiple fitted models from the previous child runs are downloaded. If you encounter this error: `AutoMLEnsembleException: Could not find any models for running ensembling`, then you may need to provide more time for the models to be downloaded. The default value is 300 seconds for downloading these models in parallel and there is no maximum timeout limit. Configure this parameter with a higher value than 300 secs, if more time is needed.

NOTE

If the timeout is reached and there are models downloaded, then the ensembling proceeds with as many models it has downloaded. It's not required that all the models need to be downloaded to finish within that timeout. The following parameters only apply to **StackEnsemble** models:

- `stack_meta_learner_type` : the meta-learner is a model trained on the output of the individual heterogeneous models. Default meta-learners are `LogisticRegression` for classification tasks (or `LogisticRegressionCV` if cross-validation is enabled) and `ElasticNet` for regression/forecasting tasks (or `ElasticNetCV` if cross-validation is enabled). This parameter can be one of the following strings: `LogisticRegression` , `LogisticRegressionCV` , `LightGBMClassifier` , `ElasticNet` , `ElasticNetCV` , `LightGBMRegressor` , or `LinearRegression` .
- `stack_meta_learner_train_percentage` : specifies the proportion of the training set (when choosing train and validation type of training) to be reserved for training the meta-learner. Default value is `0.2` .
- `stack_meta_learner_kwargs` : optional parameters to pass to the initializer of the meta-learner. These parameters and parameter types mirror the parameters and parameter types from the corresponding model constructor, and are forwarded to the model constructor.

The following code shows an example of specifying custom ensemble behavior in an `AutoMLConfig` object.

```

ensemble_settings = {
    "ensemble_download_models_timeout_sec": 600
    "stack_meta_learner_type": "LogisticRegressionCV",
    "stack_meta_learner_train_percentage": 0.3,
    "stack_meta_learner_kwargs": {
        "refit": True,
        "fit_intercept": False,
        "class_weight": "balanced",
        "multi_class": "auto",
        "n_jobs": -1
    }
}
automl_classifier = AutoMLConfig(
    task='classification',
    primary_metric='AUC_weighted',
    experiment_timeout_minutes=30,
    training_data=train_data,
    label_column_name=label,
    n_cross_validations=5,
    **ensemble_settings
)

```

Exit criteria

There are a few options you can define in your AutoMLConfig to end your experiment.

CRITERIA	DESCRIPTION
No criteria	If you do not define any exit parameters the experiment continues until no further progress is made on your primary metric.
After a length of time	<p>Use <code>experiment_timeout_minutes</code> in your settings to define how long, in minutes, your experiment should continue to run.</p> <p>To help avoid experiment time out failures, there is a minimum of 15 minutes, or 60 minutes if your row by column size exceeds 10 million.</p>
A score has been reached	Use <code>experiment_exit_score</code> completes the experiment after a specified primary metric score has been reached.

Run experiment

WARNING

If you run an experiment with the same configuration settings and primary metric multiple times, you'll likely see variation in each experiments final metrics score and generated models. The algorithms automated ML employs have inherent randomness that can cause slight variation in the models output by the experiment and the recommended model's final metrics score, like accuracy. You'll likely also see results with the same model name, but different hyperparameters used.

For automated ML, you create an `Experiment` object, which is a named object in a `Workspace` used to run experiments.

```
from azureml.core.experiment import Experiment

ws = Workspace.from_config()

# Choose a name for the experiment and specify the project folder.
experiment_name = 'Tutorial-automl'
project_folder = './sample_projects/automl-classification'

experiment = Experiment(ws, experiment_name)
```

Submit the experiment to run and generate a model. Pass the `AutoMLConfig` to the `submit` method to generate the model.

```
run = experiment.submit(automl_config, show_output=True)
```

NOTE

Dependencies are first installed on a new machine. It may take up to 10 minutes before output is shown. Setting `show_output` to `True` results in output being shown on the console.

Multiple child runs on clusters

Automated ML experiment child runs can be performed on a cluster that is already running another experiment. However, the timing depends on how many nodes the cluster has, and if those nodes are available to run a different experiment.

Each node in the cluster acts as an individual virtual machine (VM) that can accomplish a single training run; for automated ML this means a child run. If all the nodes are busy, the new experiment is queued. But if there are free nodes, the new experiment will run automated ML child runs in parallel in the available nodes/VMs.

To help manage child runs and when they can be performed, we recommend you create a dedicated cluster per experiment, and match the number of `max_concurrent_iterations` of your experiment to the number of nodes in the cluster. This way, you use all the nodes of the cluster at the same time with the number of concurrent child runs/iterations you want.

Configure `max_concurrent_iterations` in your `AutoMLConfig` object. If it is not configured, then by default only one concurrent child run/iteration is allowed per experiment. In case of compute instance, `max_concurrent_iterations` can be set to be the same as number of cores on the compute instance VM.

Explore models and metrics

Automated ML offers options for you to monitor and evaluate your training results.

- You can view your training results in a widget or inline if you are in a notebook. See [Monitor automated machine learning runs](#) for more details.
- For definitions and examples of the performance charts and metrics provided for each run, see [Evaluate automated machine learning experiment results](#).
- To get a featurization summary and understand what features were added to a particular model, see [Featurization transparency](#).

You can view the hyperparameters, the scaling and normalization techniques, and algorithm applied to a specific automated ML run with the [custom code solution](#), `print_model()`.

TIP

Automated ML also let's you [view the generated model training code for Auto ML trained models](#). This functionality is in public preview and can change at any time.

Monitor automated machine learning runs

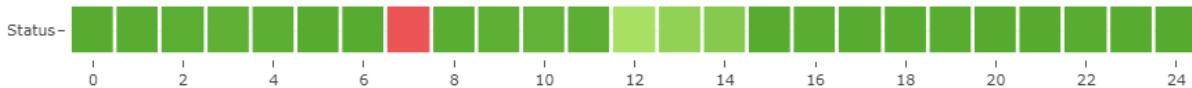
For automated ML runs, to access the charts from a previous run, replace `<<experiment_name>>` with the appropriate experiment name:

```
from azureml.widgets import RunDetails
from azureml.core.run import Run

experiment = Experiment (workspace, <<experiment_name>>)
run_id = 'autoML_my_runID' #replace with run_ID
run = Run(experiment, run_id)
RunDetails(run).show()
```

AutoML_ed181129-3876-452a-82b0-39f33a8290b7:

Status: **Completed**

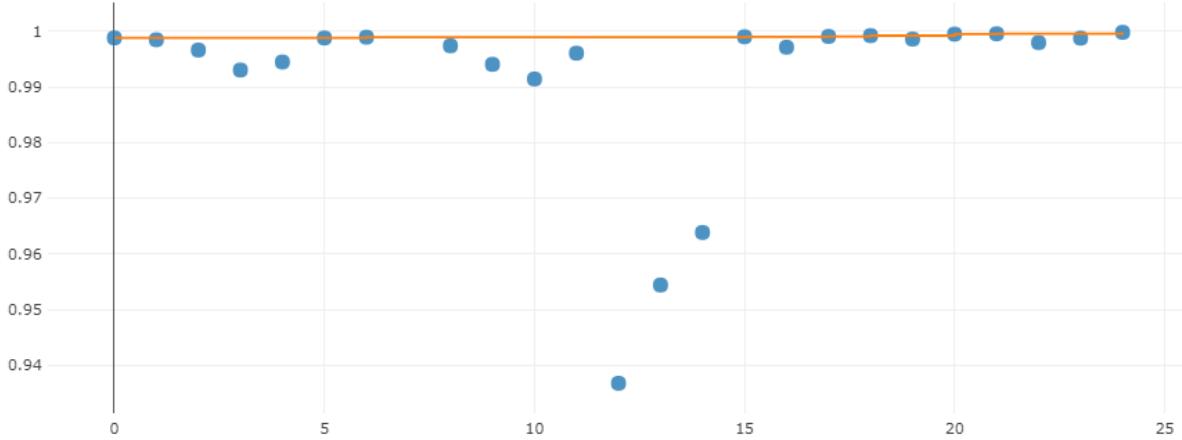


Iteration	Pipeline	Iteration metric	Best metric	Status	Duration	Started
0	StandardScalerWrapper, KNN	0.99883057	0.99883057	Completed	0:00:00	Nov 26, 2018 1:30 PM
1	StandardScalerWrapper, KNN	0.99849239	0.99883057	Completed	0:00:00	Nov 26, 2018 1:30 PM
2	MaxAbsScaler, LightGBM	0.99664006	0.99883057	Completed	0:00:00	Nov 26, 2018 1:30 PM
3	StandardScalerWrapper, LightGBM	0.9930566	0.99883057	Completed	0:00:00	Nov 26, 2018 1:31 PM
4	StandardScalerWrapper, LogisticRegression	0.9944965	0.99883057	Completed	0:00:00	Nov 26, 2018 1:31 PM

Pages: [1](#) [2](#) [3](#) [4](#) [5](#) [Next](#) [Last](#) [5](#) per page

[AUC_weighted](#)

Run with metric : AUC_weighted



[Click here to see the run in Azure portal](#)

Test models (preview)

IMPORTANT

Testing your models with a test dataset to evaluate automated ML generated models is a preview feature. This capability is an [experimental](#) preview feature, and may change at any time.

WARNING

This feature is not available for the following automated ML scenarios

- Computer vision tasks (preview)
- Many models and hierarchical time series forecasting training (preview)
- Forecasting tasks where deep learning neural networks (DNN) are enabled
- Automated ML runs from local computes or Azure Databricks clusters

Passing the `test_data` or `test_size` parameters into the `AutoMLConfig`, automatically triggers a remote test run that uses the provided test data to evaluate the best model that automated ML recommends upon completion of the experiment. This remote test run is done at the end of the experiment, once the best model is determined. See how to [pass test data into your `AutoMLConfig`](#).

Get test job results

You can get the predictions and metrics from the remote test job from the [Azure Machine Learning studio](#) or with the following code.

```
best_run, fitted_model = remote_run.get_output()
test_run = next(best_run.get_children(type='automl.model_test'))
test_run.wait_for_completion(show_output=False, wait_post_processing=True)

# Get test metrics
test_run_metrics = test_run.get_metrics()
for name, value in test_run_metrics.items():
    print(f"{name}: {value}")

# Get test predictions as a Dataset
test_run_details = test_run.get_details()
dataset_id = test_run_details['outputDatasets'][0]['identifier']['savedId']
test_run_predictions = Dataset.get_by_id(workspace, dataset_id)
predictions_df = test_run_predictions.to_pandas_dataframe()

# Alternatively, the test predictions can be retrieved via the run outputs.
test_run.download_file("predictions/predictions.csv")
predictions_df = pd.read_csv("predictions.csv")
```

The model test job generates the `predictions.csv` file that's stored in the default datastore created with the workspace. This datastore is visible to all users with the same subscription. Test jobs are not recommended for scenarios if any of the information used for or created by the test job needs to remain private.

Test existing automated ML model

To test other existing automated ML models created, best job or child job, use `ModelProxy()` to test a model after the main AutoML run has completed. `ModelProxy()` already returns the predictions and metrics and does not require further processing to retrieve the outputs.

NOTE

`ModelProxy` is an [experimental](#) preview class, and may change at any time.

The following code demonstrates how to test a model from any run by using `ModelProxy.test()` method. In the `test()` method you have the option to specify if you only want to see the predictions of the test run with the `include_predictions_only` parameter.

```
from azureml.train.automl.model_proxy import ModelProxy

model_proxy = ModelProxy(child_run=my_run, compute_target=cpu_cluster)
predictions, metrics = model_proxy.test(test_data, include_predictions_only= True
)
```

Register and deploy models

After you test a model and confirm you want to use it in production, you can register it for later use and

To register a model from an automated ML run, use the `register_model()` method.

```
best_run = run.get_best_child()
print(fitted_model.steps)

model_name = best_run.properties['model_name']
description = 'AutoML forecast example'
tags = None

model = run.register_model(model_name = model_name,
                           description = description,
                           tags = tags)
```

For details on how to create a deployment configuration and deploy a registered model to a web service, see [how and where to deploy a model](#).

TIP

For registered models, one-click deployment is available via the [Azure Machine Learning studio](#). See [how to deploy registered models from the studio](#).

Model interpretability

Model interpretability allows you to understand why your models made predictions, and the underlying feature importance values. The SDK includes various packages for enabling model interpretability features, both at training and inference time, for local and deployed models.

See how to [enable interpretability features](#) specifically within automated ML experiments.

For general information on how model explanations and feature importance can be enabled in other areas of the SDK outside of automated machine learning, see the [concept article on interpretability](#).

NOTE

The ForecastTCN model is not currently supported by the Explanation Client. This model will not return an explanation dashboard if it is returned as the best model, and does not support on-demand explanation runs.

Next steps

- Learn more about [how and where to deploy a model](#).

- Learn more about [how to train a regression model with Automated machine learning](#).
- [Troubleshoot automated ML experiments](#).

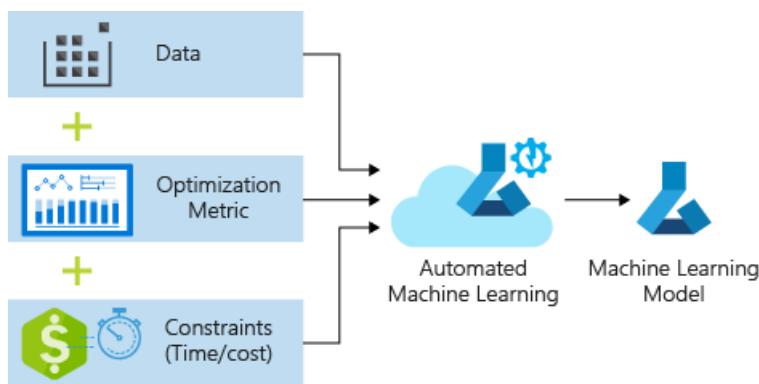
Train a regression model with AutoML and Python (SDK v1)

9/21/2022 • 12 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to train a regression model with the Azure Machine Learning Python SDK using Azure Machine Learning automated ML. This regression model predicts NYC taxi fares.

This process accepts training data and configuration settings, and automatically iterates through combinations of different feature normalization/standardization methods, models, and hyperparameter settings to arrive at the best model.



You'll write code using the Python SDK in this article. You'll learn the following tasks:

- Download, transform, and clean data using Azure Open Datasets
- Train an automated machine learning regression model
- Calculate model accuracy

For no-code AutoML, try the following tutorials:

- [Tutorial: Train no-code classification models](#)
- [Tutorial: Forecast demand with automated machine learning](#)

Prerequisites

If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version](#) of Azure Machine Learning today.

- Complete the [Quickstart: Get started with Azure Machine Learning](#) if you don't already have an Azure Machine Learning workspace or a compute instance.
- After you complete the quickstart:
 1. Select **Notebooks** in the studio.
 2. Select the **Samples** tab.
 3. Open the `tutorials/regression-automl-nyc-taxi-data/regression-automated-ml.ipynb` notebook.
 4. To run each cell in the tutorial, select **Clone this notebook**

This article is also available on [GitHub](#) if you wish to run it in your own [local environment](#). To get the required packages,

- Install the full `automl` client.
 - Run `pip install azureml-opendatasets azureml-widgets` to get the required packages.

Download and prepare data

Import the necessary packages. The Open Datasets package contains a class representing each data source (`NycTlcGreen` for example) to easily filter date parameters before downloading.

```
from azureml.opendatasets import NycTlcGreen
import pandas as pd
from datetime import datetime
from dateutil.relativedelta import relativedelta
```

Begin by creating a dataframe to hold the taxi data. When working in a non-Spark environment, Open Datasets only allows downloading one month of data at a time with certain classes to avoid `MemoryError` with large datasets.

To download taxi data, iteratively fetch one month at a time, and before appending it to `green_taxi_df` randomly sample 2,000 records from each month to avoid bloating the dataframe. Then preview the data.

```
green_taxi_df = pd.DataFrame([])
start = datetime.strptime("1/1/2015", "%m/%d/%Y")
end = datetime.strptime("1/31/2015", "%m/%d/%Y")

for sample_month in range(12):
    temp_df_green = NycTlcGreen(start + relativedelta(months=sample_month), end +
relativedelta(months=sample_month)) \
        .to_pandas_dataframe()
    green_taxi_df = green_taxi_df.append(temp_df_green.sample(2000))

green_taxi_df.head(10)
```

L	P	E	P	D	R	O	P	I	M	T	T	O	L	A	T	R
P	P	E	P	P	O	P	O	C	P	P	E	S	S	H	A	I
P	P	E	P	P	O	P	O	E	P	O	E	S	S	M	M	E
V	D	D	D	F	F	F	F	N	A	A	A	E	E	H	H	T
E	A	A	A	E	E	E	E	E	E	E	E	A	A	A	A	A
N	T	T	T	E	E	E	E	N	T	T	T	L	L	L	L	L
D	T	T	T	E	E	E	E	E	T	T	T	A	A	A	A	A
O	E	E	E	C	C	C	C	O	E	E	E	M	M	M	M	M
R	E	E	E	O	O	O	O	E	E	E	E	A	A	A	A	A
I	T	T	T	N	N	N	N	N	T	T	T	M	M	M	M	M
D	T	T	T	O	O	O	O	O	O	O	O	A	A	A	A	A
I	E	E	E	U	U	U	U	U	U	U	U	M	M	M	M	M
M	E	E	E	N	N	N	N	N	N	N	N	A	A	A	A	A

VENDOR ID	LPEPPEPPICKUPPDATEETIME	TIME	LPEPPEPPICKUPPDATEETIME	TRIP COUNT	PASSENGER ID	TRIP ID	LOCATION ID	DOCK LOCATION ID	PICKUP LONGITUDE	PICKUP LATITUDE	DROPOFF LONGITUDE	DROPOFF LATITUDE	PAYMENT TYPE	FARE AMOUNT	EXTRA	MATA TAX	IMPROVEMENTS SURCHARGE	TIP AMOUNT	TOLLSAMOUNT	EHAILFEE	TOTAL AMOUNT	TRIP TYPE
131969	20115:-0111005:344:404	22001111005:45:03	20115:-0111005:45:03	3.84	None	None	-73.	4084	-73.94	4084	-73.94	...	200	1500	0	0	0	0	0	0	nan	16.30
1129817	20115:-0111005:45:03	22001111005:45:03	20115:-0111005:45:03	1.99	None	None	-73.96	4081	-73.96	4081	-73.96	...	200	4500	100	0	0	0	0	0	nana	6.30

VENDOR ID	NAME	TIME	LPEPDPICKUPPICKUPDROPOFFPAYMENTTYPE	TRIPLOCATIONID	DOLONGITUDE	PICKUPLATITUDE	DROPFLONGITUDE	FAREAMOUNT	EXTRA	MATAUTAX	IMPROVEMENTTSURCHARGE	TIPAMOUNT	TOLLSAMOUNT	EHAILFEE	TOTALAMOUNT	TRIPTYPE
12695627	10105:04:10	2005:06:23	21050	None	-73.92	40.76	-7.92	2	4.	0.50	0.50	0.00	0.00	0.00	nan	5.00
811755	10419:07:15	20150	110	None	-73.96	40.72	-7.95	2	6.50	0.50	0.50	0.30	0.00	0.00	nan	7.80

VENDORID	LPEPPEP	LPEPPEP		PASSE		TRIPL		DOKUPL		PIKUPL		DROPFL		PAYMEN		FAREAM		IMPROVE		MENTTSUR		TIPAMO		TOLLSA		TOTALAMOUNT		TRIPTYPE	
		IME	MEME	CIDE	ID	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE	IDE
4	2	2	N	N	-	4	-	...	2	5	0	0	0	0	0	0	n	6											
3	0	0	o	o	7	0	7			a	.											
2	1	1	n	n	3	.	3			0	5	5	3	0	0	0	n	3											
1	5	e	e	.	7	.	9			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	-	0	9	1	9		4	4																					
6	1																												
	-																												
	2																												
	2																												
	3																												
	:																												
	1																												
	6																												
	:																												
	3																												
	3																												
	2																												
	0																												
	0																												
	1																												
	5																												
	-																												
	0																												
	1																												
	2																												
	2																												
	3																												
	:																												
	2																												
	0																												
	1																												
	3																												
	1																												
	0																												
	.																												
	6																												
	5																												

Remove some of the columns that you won't need for training or additional feature building. Automate machine learning will automatically handle time-based features such as `lpepPickupDatetime`.

```

columns_to_remove = ["lpepDropoffDatetime", "puLocationId", "doLocationId", "extra", "mtaTax",
                     "improvementSurcharge", "tollsAmount", "ehailFee", "tripType", "rateCodeID",
                     "storeAndFwdFlag", "paymentType", "fareAmount", "tipAmount"]
]
for col in columns_to_remove:
    green_taxi_df.pop(col)

green_taxi_df.head(5)

```

Cleanse data

Run the `describe()` function on the new dataframe to see summary statistics for each field.

```
green_taxi_df.describe()
```

VENDO RID	PASSE NGERC OUNT	TRIPDI STANC E	PICKU PLONG ITUDE	PICKU PLATIT UDE	DROPO FFLO N GITU D E	DROPO FFLATI TUDE	TOTAL AMOU NT	MONT H_NU M DAY_O F_MON TH	DAY_O F_WEE K	HOUR_OF_DA Y
count	48000.00	48000.00	48000.00	48000.00	48000.00	48000.00	48000.00	48000.00	48000.00	48000.00
mean	1.78	1.37	2.87	-73.83	40.69	-73.84	40.70	14.75	6.50	15.13
std	0.41	1.04	2.93	2.76	1.52	2.61	1.44	12.08	3.45	8.45
min	1.00	0.00	0.00	-74.66	0.00	-74.66	0.00	-300.00	1.00	1.00
25%	2.00	1.00	1.06	-73.96	40.70	-73.97	40.70	7.80	3.75	8.00
50%	2.00	1.00	1.90	-73.94	40.75	-73.94	40.75	11.30	6.50	15.00
75%	2.00	1.00	3.60	-73.92	40.80	-73.91	40.79	17.80	9.25	22.00
max	2.00	9.00	97.57	0.00	41.93	0.00	41.94	450.00	12.00	30.00

From the summary statistics, you see that there are several fields that have outliers or values that will reduce model accuracy. First filter the lat/long fields to be within the bounds of the Manhattan area. This will filter out longer taxi trips or trips that are outliers in respect to their relationship with other features.

Additionally filter the `tripDistance` field to be greater than zero but less than 31 miles (the haversine distance between the two lat/long pairs). This eliminates long outlier trips that have inconsistent trip cost.

Lastly, the `totalAmount` field has negative values for the taxi fares, which don't make sense in the context of our model, and the `passengerCount` field has bad data with the minimum values being zero.

Filter out these anomalies using query functions, and then remove the last few columns unnecessary for training.

```
final_df = green_taxi_df.query("pickupLatitude>=40.53 and pickupLatitude<=40.88")
final_df = final_df.query("pickupLongitude>=-74.09 and pickupLongitude<=-73.72")
final_df = final_df.query("tripDistance>=0.25 and tripDistance<31")
final_df = final_df.query("passengerCount>0 and totalAmount>0")

columns_to_remove_for_training = ["pickupLongitude", "pickupLatitude", "dropoffLongitude",
"dropoffLatitude"]
for col in columns_to_remove_for_training:
    final_df.pop(col)
```

Call `describe()` again on the data to ensure cleansing worked as expected. You now have a prepared and cleansed set of taxi, holiday, and weather data to use for machine learning model training.

```
final_df.describe()
```

Configure workspace

Create a workspace object from the existing workspace. A [Workspace](#) is a class that accepts your Azure subscription and resource information. It also creates a cloud resource to monitor and track your model runs.

`Workspace.from_config()` reads the file `config.json` and loads the authentication details into an object named `ws`. `ws` is used throughout the rest of the code in this article.

```
from azureml.core.workspace import Workspace
ws = Workspace.from_config()
```

Split the data into train and test sets

Split the data into training and test sets by using the `train_test_split` function in the `scikit-learn` library. This function segregates the data into the **x (features)** data set for model training and the **y (values to predict)** data set for testing.

The `test_size` parameter determines the percentage of data to allocate to testing. The `random_state` parameter sets a seed to the random generator, so that your train-test splits are deterministic.

```
from sklearn.model_selection import train_test_split
x_train, x_test = train_test_split(final_df, test_size=0.2, random_state=223)
```

The purpose of this step is to have data points to test the finished model that haven't been used to train the model, in order to measure true accuracy.

In other words, a well-trained model should be able to accurately make predictions from data it hasn't already seen. You now have data prepared for auto-training a machine learning model.

Automatically train a model

To automatically train a model, take the following steps:

1. Define settings for the experiment run. Attach your training data to the configuration, and modify settings that control the training process.
2. Submit the experiment for model tuning. After submitting the experiment, the process iterates through different machine learning algorithms and hyperparameter settings, adhering to your defined constraints. It chooses the best-fit model by optimizing an accuracy metric.

Define training settings

Define the experiment parameter and model settings for training. View the full list of [settings](#). Submitting the experiment with these default settings will take approximately 5-20 min, but if you want a shorter run time, reduce the `experiment_timeout_hours` parameter.

PROPERTY	VALUE IN THIS ARTICLE	DESCRIPTION
<code>iteration_timeout_minutes</code>	10	Time limit in minutes for each iteration. Increase this value for larger datasets that need more time for each iteration.
<code>experiment_timeout_hours</code>	0.3	Maximum amount of time in hours that all iterations combined can take before the experiment terminates.
<code>enable_early_stopping</code>	True	Flag to enable early termination if the score is not improving in the short term.
<code>primary_metric</code>	<code>spearman_correlation</code>	Metric that you want to optimize. The best-fit model will be chosen based on this metric.
<code>featurization</code>	<code>auto</code>	By using <code>auto</code> , the experiment can preprocess the input data (handling missing data, converting text to numeric, etc.)
<code>verbosity</code>	<code>logging.INFO</code>	Controls the level of logging.
<code>n_cross_validations</code>	5	Number of cross-validation splits to perform when validation data is not specified.

```
import logging

automl_settings = {
    "iteration_timeout_minutes": 10,
    "experiment_timeout_hours": 0.3,
    "enable_early_stopping": True,
    "primary_metric": 'spearman_correlation',
    "featurization": 'auto',
    "verbosity": logging.INFO,
    "n_cross_validations": 5
}
```

Use your defined training settings as a `**kwargs` parameter to an `AutoMLConfig` object. Additionally, specify your training data and the type of model, which is `regression` in this case.

```
from azureml.train.automl import AutoMLConfig

automl_config = AutoMLConfig(task='regression',
                             debug_log='automated_ml_errors.log',
                             training_data=x_train,
                             label_column_name="totalAmount",
                             **automl_settings)
```

NOTE

Automated machine learning pre-processing steps (feature normalization, handling missing data, converting text to numeric, etc.) become part of the underlying model. When using the model for predictions, the same pre-processing steps applied during training are applied to your input data automatically.

Train the automatic regression model

Create an experiment object in your workspace. An experiment acts as a container for your individual jobs. Pass the defined `automl_config` object to the experiment, and set the output to `True` to view progress during the job.

After starting the experiment, the output shown updates live as the experiment runs. For each iteration, you see the model type, the run duration, and the training accuracy. The field `BEST` tracks the best running training score based on your metric type.

```
from azureml.core.experiment import Experiment
experiment = Experiment(ws, "Tutorial-NYCTaxi")
local_run = experiment.submit(automl_config, show_output=True)
```

```
Running on local machine
Parent Run ID: AutoML_1766cdf7-56cf-4b28-a340-c4aeee15b12b
Current status: DatasetFeaturization. Beginning to featurize the dataset.
Current status: DatasetEvaluation. Gathering dataset statistics.
Current status: FeaturesGeneration. Generating features for the dataset.
Current status: DatasetFeaturizationCompleted. Completed featurizing the dataset.
Current status: DatasetCrossValidationSplit. Generating individually featurized CV splits.
Current status: ModelSelection. Beginning model selection.

*****
ITERATION: The iteration being evaluated.
PIPELINE: A summary description of the pipeline being evaluated.
DURATION: Time taken for the current iteration.
METRIC: The result of computing score on the fitted pipeline.
BEST: The best observed score thus far.
*****
```

ITERATION	PIPELINE	DURATION	METRIC	BEST
0	StandardScalerWrapper RandomForest	0:00:16	0.8746	0.8746
1	MinMaxScaler RandomForest	0:00:15	0.9468	0.9468
2	StandardScalerWrapper ExtremeRandomTrees	0:00:09	0.9303	0.9468
3	StandardScalerWrapper LightGBM	0:00:10	0.9424	0.9468
4	RobustScaler DecisionTree	0:00:09	0.9449	0.9468
5	StandardScalerWrapper LassoLars	0:00:09	0.9440	0.9468
6	StandardScalerWrapper LightGBM	0:00:10	0.9282	0.9468
7	StandardScalerWrapper RandomForest	0:00:12	0.8946	0.9468
8	StandardScalerWrapper LassoLars	0:00:16	0.9439	0.9468
9	MinMaxScaler ExtremeRandomTrees	0:00:35	0.9199	0.9468
10	RobustScaler ExtremeRandomTrees	0:00:19	0.9411	0.9468
11	StandardScalerWrapper ExtremeRandomTrees	0:00:13	0.9077	0.9468
12	StandardScalerWrapper LassoLars	0:00:15	0.9433	0.9468
13	MinMaxScaler ExtremeRandomTrees	0:00:14	0.9186	0.9468
14	RobustScaler RandomForest	0:00:10	0.8810	0.9468
15	StandardScalerWrapper LassoLars	0:00:55	0.9433	0.9468
16	StandardScalerWrapper ExtremeRandomTrees	0:00:13	0.9026	0.9468
17	StandardScalerWrapper RandomForest	0:00:13	0.9140	0.9468
18	VotingEnsemble	0:00:23	0.9471	0.9471
19	StackEnsemble	0:00:27	0.9463	0.9471

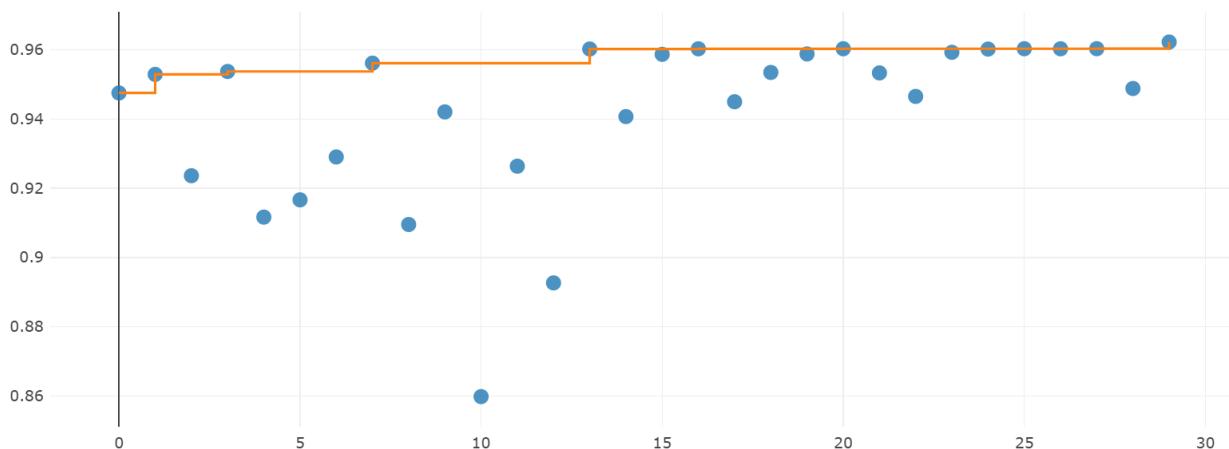
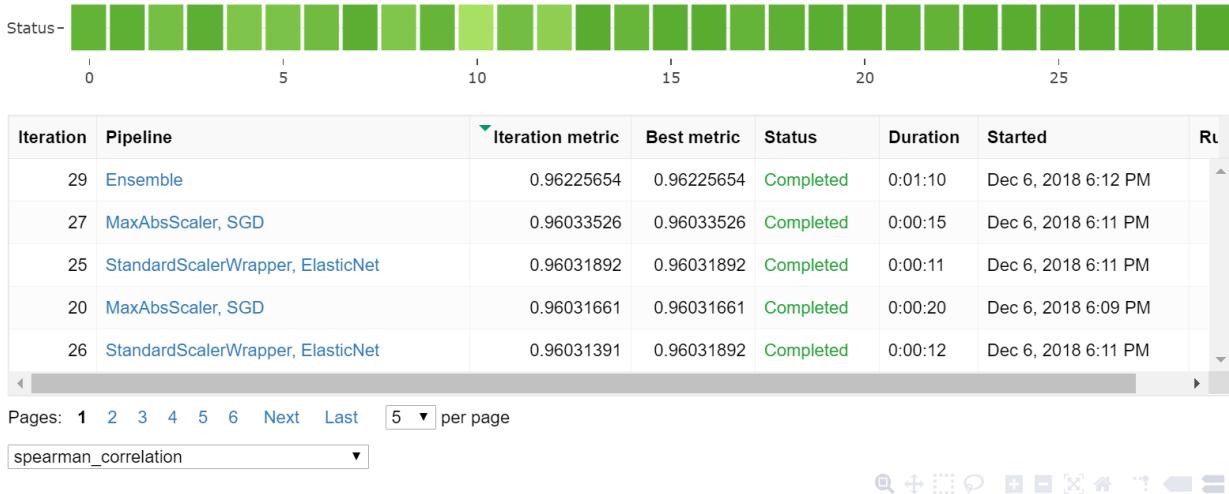
Explore the results

Explore the results of automatic training with a [Jupyter widget](#). The widget allows you to see a graph and table of all individual job iterations, along with training accuracy metrics and metadata. Additionally, you can filter on different accuracy metrics than your primary metric with the dropdown selector.

```
from azureml.widgets import RunDetails  
RunDetails(local_run).show()
```

AutoML_797ff8a7-a369-4986-8180-e9bbbed938259:

Status: Completed



Retrieve the best model

Select the best model from your iterations. The `get_output` function returns the best run and the fitted model for the last fit invocation. By using the overloads on `get_output`, you can retrieve the best run and fitted model for any logged metric or a particular iteration.

```
best_run, fitted_model = local_run.get_output()  
print(best_run)  
print(fitted_model)
```

Test the best model accuracy

Use the best model to run predictions on the test data set to predict taxi fares. The function `predict` uses the best model and predicts the values of `y_trip cost`, from the `x_test` data set. Print the first 10 predicted cost values from `y_predict`.

```
y_test = x_test.pop("totalAmount")

y_predict = fitted_model.predict(x_test)
print(y_predict[:10])
```

Calculate the `root mean squared error` of the results. Convert the `y_test` dataframe to a list to compare to the predicted values. The function `mean_squared_error` takes two arrays of values and calculates the average squared error between them. Taking the square root of the result gives an error in the same units as the y variable, `cost`. It indicates roughly how far the taxi fare predictions are from the actual fares.

```
from sklearn.metrics import mean_squared_error
from math import sqrt

y_actual = y_test.values.flatten().tolist()
rmse = sqrt(mean_squared_error(y_actual, y_predict))
rmse
```

Run the following code to calculate mean absolute percent error (MAPE) by using the full `y_actual` and `y_predict` data sets. This metric calculates an absolute difference between each predicted and actual value and sums all the differences. Then it expresses that sum as a percent of the total of the actual values.

```
sum_actuals = sum_errors = 0

for actual_val, predict_val in zip(y_actual, y_predict):
    abs_error = actual_val - predict_val
    if abs_error < 0:
        abs_error = abs_error * -1

    sum_errors = sum_errors + abs_error
    sum_actuals = sum_actuals + actual_val

mean_abs_percent_error = sum_errors / sum_actuals
print("Model MAPE:")
print(mean_abs_percent_error)
print()
print("Model Accuracy:")
print(1 - mean_abs_percent_error)
```

```
Model MAPE:
0.14353867606052823

Model Accuracy:
0.8564613239394718
```

From the two prediction accuracy metrics, you see that the model is fairly good at predicting taxi fares from the data set's features, typically within $\pm \$4.00$, and approximately 15% error.

The traditional machine learning model development process is highly resource-intensive, and requires significant domain knowledge and time investment to run and compare the results of dozens of models. Using automated machine learning is a great way to rapidly test many different models for your scenario.

Clean up resources

Do not complete this section if you plan on running other Azure Machine Learning tutorials.

Stop the compute instance

If you used a compute instance, stop the VM when you aren't using it to reduce cost.

1. In your workspace, select **Compute**.
2. From the list, select the name of the compute instance.
3. Select **Stop**.
4. When you're ready to use the server again, select **Start**.

Delete everything

If you don't plan to use the resources you created, delete them, so you don't incur any charges.

1. In the Azure portal, select **Resource groups** on the far left.
2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties and select **Delete**.

Next steps

In this automated machine learning article, you did the following tasks:

- Configured a workspace and prepared data for an experiment.
- Trained by using an automated regression model locally with custom parameters.
- Explored and reviewed training results.

[Set up AutoML to train computer vision models with Python \(v1\)](#)

Tutorial: Train an object detection model (preview) with AutoML and Python (v1)

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

The features presented in this article are in preview. They should be considered [experimental](#) preview features that might change at any time.

In this tutorial, you learn how to train an object detection model using Azure Machine Learning automated ML with the Azure Machine Learning Python SDK. This object detection model identifies whether the image contains objects, such as a can, carton, milk bottle, or water bottle.

Automated ML accepts training data and configuration settings, and automatically iterates through combinations of different feature normalization/standardization methods, models, and hyperparameter settings to arrive at the best model.

You'll write code using the Python SDK in this tutorial and learn the following tasks:

- Download and transform data
- Train an automated machine learning object detection model
- Specify hyperparameter values for your model
- Perform a hyperparameter sweep
- Deploy your model
- Visualize detections

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version](#) of Azure Machine Learning today.
- Python 3.6 or 3.7 are supported for this feature
- Complete the [Quickstart: Get started with Azure Machine Learning](#) if you don't already have an Azure Machine Learning workspace.
- Download and unzip the [*odFridgeObjects.zip](#) data file. The dataset is annotated in Pascal VOC format, where each image corresponds to an xml file. Each xml file contains information on where its corresponding image file is located and also contains information about the bounding boxes and the object labels. In order to use this data, you first need to convert it to the required JSONL format as seen in the [Convert the downloaded data to JSONL](#) section of the notebook.

This tutorial is also available in the [azureml-examples repository on GitHub](#) if you wish to run it in your own [local environment](#). To get the required packages,

- Run `pip install azureml`
- [Install the full automl client](#)

Compute target setup

You first need to set up a compute target to use for your automated ML model training. Automated ML models for image tasks require GPU SKUs.

This tutorial uses the NCsv3-series (with V100 GPUs) as this type of compute target leverages multiple GPUs to speed up training. Additionally, you can set up multiple nodes to take advantage of parallelism when tuning hyperparameters for your model.

The following code creates a GPU compute of size Standard_NC24s_v3 with four nodes that are attached to the workspace, `ws`.

WARNING

Ensure your subscription has sufficient quota for the compute target you wish to use.

```
from azureml.core.compute import AmlCompute, ComputeTarget

cluster_name = "gpu-nc24sv3"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target.')
except KeyError:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='Standard_NC24s_v3',
                                                          idle_seconds_before_scaledown=1800,
                                                          min_nodes=0,
                                                          max_nodes=4)

    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

#If no min_node_count is provided, the scale settings are used for the cluster.
compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)
```

Experiment setup

Next, create an `Experiment` in your workspace to track your model training runs.

```
from azureml.core import Experiment

experiment_name = 'automl-image-object-detection'
experiment = Experiment(ws, name=experiment_name)
```

Visualize input data

Once you have the input image data prepared in [JSONL](#) (JSON Lines) format, you can visualize the ground truth bounding boxes for an image. To do so, be sure you have `matplotlib` installed.

```
%pip install --upgrade matplotlib
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

```

import matplotlib.patches as patches
from PIL import Image as pil_image
import numpy as np
import json
import os

def plot_ground_truth_boxes(image_file, ground_truth_boxes):
    # Display the image
    plt.figure()
    img_np = mpimg.imread(image_file)
    img = pil_image.fromarray(img_np.astype("uint8"), "RGB")
    img_w, img_h = img.size

    fig,ax = plt.subplots(figsize=(12, 16))
    ax.imshow(img_np)
    ax.axis("off")

    label_to_color_mapping = {}

    for gt in ground_truth_boxes:
        label = gt["label"]

        xmin, ymin, xmax, ymax = gt["topX"], gt["topY"], gt["bottomX"], gt["bottomY"]
        topleft_x, topleft_y = img_w * xmin, img_h * ymin
        width, height = img_w * (xmax - xmin), img_h * (ymax - ymin)

        if label in label_to_color_mapping:
            color = label_to_color_mapping[label]
        else:
            # Generate a random color. If you want to use a specific color, you can use something like
            "red".
            color = np.random.rand(3)
            label_to_color_mapping[label] = color

        # Display bounding box
        rect = patches.Rectangle((topleft_x, topleft_y), width, height,
                                linewidth=2, edgecolor=color, facecolor="none")
        ax.add_patch(rect)

        # Display label
        ax.text(topleft_x, topleft_y - 10, label, color=color, fontsize=20)

    plt.show()

def plot_ground_truth_boxes_jsonl(image_file, jsonl_file):
    image_base_name = os.path.basename(image_file)
    ground_truth_data_found = False
    with open(jsonl_file) as fp:
        for line in fp.readlines():
            line_json = json.loads(line)
            filename = line_json["image_url"]
            if image_base_name in filename:
                ground_truth_data_found = True
                plot_ground_truth_boxes(image_file, line_json["label"])
                break
    if not ground_truth_data_found:
        print("Unable to find ground truth information for image: {}".format(image_file))

def plot_ground_truth_boxes_dataset(image_file, dataset_pd):
    image_base_name = os.path.basename(image_file)
    image_pd = dataset_pd[dataset_pd['portable_path'].str.contains(image_base_name)]
    if not image_pd.empty:
        ground_truth_boxes = image_pd.iloc[0]["label"]
        plot_ground_truth_boxes(image_file, ground_truth_boxes)
    else:
        print("Unable to find ground truth information for image: {}".format(image_file))

```

Using the above helper functions, for any given image, you can run the following code to display the bounding boxes.

```
image_file = "./odFridgeObjects/images/31.jpg"
jsonl_file = "./odFridgeObjects/train_annotations.jsonl"

plot_ground_truth_boxes_jsonl(image_file, jsonl_file)
```

Upload data and create dataset

In order to use the data for training, upload it to your workspace via a datastore. The datastore provides a mechanism for you to upload or download data, and interact with it from your remote compute targets.

```
ds = ws.get_default_datastore()
ds.upload(src_dir='./odFridgeObjects', target_path='odFridgeObjects')
```

Once uploaded to the datastore, you can create an Azure Machine Learning dataset from the data. Datasets package your data into a consumable object for training.

The following code creates a dataset for training. Since no validation dataset is specified, by default 20% of your training data is used for validation.

```
from azureml.core import Dataset
from azureml.data import DataType

training_dataset_name = 'odFridgeObjectsTrainingDataset'
if training_dataset_name in ws.datasets:
    training_dataset = ws.datasets.get(training_dataset_name)
    print('Found the training dataset', training_dataset_name)
else:
    # create training dataset
    # create training dataset
    training_dataset = Dataset.Tabular.from_json_lines_files(
        path=ds.path('odFridgeObjects/train_annotations.jsonl'),
        set_column_types={"image_url": DataType.to_stream(ds.workspace)},
    )
    training_dataset = training_dataset.register(workspace=ws, name=training_dataset_name)

print("Training dataset name: " + training_dataset.name)
```

Visualize dataset

You can also visualize the ground truth bounding boxes for an image from this dataset.

Load the dataset into a pandas dataframe.

```
import azureml.dataprep as dprep

from azureml.dataprep.api.functions import get_portable_path

# Get pandas dataframe from the dataset
dflow = training_dataset._dataflow.add_column(get_portable_path(dprep.col("image_url")),
                                              "portable_path", "image_url")
dataset_pd = dflow.to_pandas_dataframe(extended_types=True)
```

For any given image, you can run the following code to display the bounding boxes.

```
image_file = "./odFridgeObjects/images/31.jpg"
plot_ground_truth_boxes_dataset(image_file, dataset_pd)
```

Configure your object detection experiment

To configure automated ML runs for image-related tasks, use the `AutoMLImageConfig` object. In your `AutoMLImageConfig`, you can specify the model algorithms with the `model_name` parameter and configure the settings to perform a hyperparameter sweep over a defined parameter space to find the optimal model.

In this example, we use the `AutoMLImageConfig` to train an object detection model with `yolov5` and `fasterrcnn_resnet50_fpn`, both of which are pretrained on COCO, a large-scale object detection, segmentation, and captioning dataset that contains over thousands of labeled images with over 80 label categories.

Hyperparameter sweeping for image tasks

You can perform a hyperparameter sweep over a defined parameter space to find the optimal model.

The following code, defines the parameter space in preparation for the hyperparameter sweep for each defined algorithm, `yolov5` and `fasterrcnn_resnet50_fpn`. In the parameter space, specify the range of values for `learning_rate`, `optimizer`, `lr_scheduler`, etc., for AutoML to choose from as it attempts to generate a model with the optimal primary metric. If hyperparameter values are not specified, then default values are used for each algorithm.

For the tuning settings, use random sampling to pick samples from this parameter space by importing the `GridParameterSampling`, `RandomParameterSampling` and `BayesianParameterSampling` classes. Doing so, tells automated ML to try a total of 20 iterations with these different samples, running four iterations at a time on our compute target, which was set up using four nodes. The more parameters the space has, the more iterations you need to find optimal models.

The Bandit early termination policy is also used. This policy terminates poor performing configurations; that is, those configurations that are not within 20% slack of the best performing configuration, which significantly saves compute resources.

```

from azureml.train.hyperdrive import RandomParameterSampling
from azureml.train.hyperdrive import BanditPolicy, HyperDriveConfig
from azureml.train.hyperdrive import choice, uniform

parameter_space = {
    'model': choice(
        {
            'model_name': choice('yolov5'),
            'learning_rate': uniform(0.0001, 0.01),
            #'model_size': choice('small', 'medium'), # model-specific
            'img_size': choice(640, 704, 768), # model-specific
        },
        {
            'model_name': choice('fasterrcnn_resnet50_fpn'),
            'learning_rate': uniform(0.0001, 0.001),
            #'warmup_cosine_lr_warmup_epochs': choice(0, 3),
            'optimizer': choice('sgd', 'adam', 'adamw'),
            'min_size': choice(600, 800), # model-specific
        }
    )
}

tuning_settings = {
    'iterations': 20,
    'max_concurrent_iterations': 4,
    'hyperparameter_sampling': RandomParameterSampling(parameter_space),
    'policy': BanditPolicy(evaluation_interval=2, slack_factor=0.2, delay_evaluation=6)
}

```

Once the parameter space and tuning settings are defined, you can pass them into your `AutoMLImageConfig` object and then submit the experiment to train an image model using your training dataset.

```

from azureml.train.automl import AutoMLImageConfig
automl_image_config = AutoMLImageConfig(task='image-object-detection',
                                         compute_target=compute_target,
                                         training_data=training_dataset,
                                         validation_data=validation_dataset,
                                         primary_metric='mean_average_precision',
                                         **tuning_settings)

automl_image_run = experiment.submit(automl_image_config)
automl_image_run.wait_for_completion(wait_post_processing=True)

```

When doing a hyperparameter sweep, it can be useful to visualize the different configurations that were tried using the HyperDrive UI. You can navigate to this UI by going to the 'Child runs' tab in the UI of the main `automl_image_run` from above, which is the HyperDrive parent run. Then you can go into the 'Child runs' tab of this one. Alternatively, here below you can see directly the HyperDrive parent run and navigate to its 'Child runs' tab:

```

from azureml.core import Run
hyperdrive_run = Run(experiment=experiment, run_id=automl_image_run.id + '_HD')
hyperdrive_run

```

Register the best model

Once the run completes, we can register the model that was created from the best run.

```
best_child_run = automl_image_run.get_best_child()
model_name = best_child_run.properties['model_name']
model = best_child_run.register_model(model_name = model_name, model_path='outputs/model.pt')
```

Deploy model as a web service

Once you have your trained model, you can deploy the model on Azure. You can deploy your trained model as a web service on Azure Container Instances (ACI) or Azure Kubernetes Service (AKS). ACI is the perfect option for testing deployments, while AKS is better suited for high-scale, production usage.

In this tutorial, we deploy the model as a web service in AKS.

1. Create an AKS compute cluster. In this example, a GPU virtual machine SKU is used for the deployment cluster

```
from azureml.core.compute import ComputeTarget, AksCompute
from azureml.exceptions import ComputeTargetException

# Choose a name for your cluster
aks_name = "cluster-aks-gpu"

# Check to see if the cluster already exists
try:
    aks_target = ComputeTarget(workspace=ws, name=aks_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    # Provision AKS cluster with GPU machine
    prov_config = AksCompute.provisioning_configuration(vm_size="STANDARD_NC6",
                                                          location="eastus2")
    # Create the cluster
    aks_target = ComputeTarget.create(workspace=ws,
                                      name=aks_name,
                                      provisioning_configuration=prov_config)
    aks_target.wait_for_completion(show_output=True)
```

2. Define the inference configuration that describes how to set up the web-service containing your model.

You can use the scoring script and the environment from the training run in your inference config.

NOTE

To change the model's settings, open the downloaded scoring script and modify the `model_settings` variable before deploying the model.

```
from azureml.core.model import InferenceConfig

best_child_run.download_file('outputs/scoring_file_v_1_0_0.py', output_file_path='score.py')
environment = best_child_run.get_environment()
inference_config = InferenceConfig(entry_script='score.py', environment=environment)
```

3. You can then deploy the model as an AKS web service.

```
from azureml.core.webservice import AksWebservice
from azureml.core.webservice import Webservice
from azureml.core.model import Model
from azureml.core.environment import Environment

aks_config = AksWebservice.deploy_configuration(autoscale_enabled=True,
                                                cpu_cores=1,
                                                memory_gb=50,
                                                enable_app_insights=True)

aks_service = Model.deploy(ws,
                           models=[model],
                           inference_config=inference_config,
                           deployment_config=aks_config,
                           deployment_target=aks_target,
                           name='automl-image-test',
                           overwrite=True)
aks_service.wait_for_deployment(show_output=True)
print(aks_service.state)
```

Test the web service

You can test the deployed web service to predict new images. For this tutorial, pass a random image from the dataset and pass it to the scoring URL.

```
import requests

# URL for the web service
scoring_uri = aks_service.scoring_uri

# If the service is authenticated, set the key or token
key, _ = aks_service.get_keys()

sample_image = './test_image.jpg'

# Load image data
data = open(sample_image, 'rb').read()

# Set the content type
headers = {'Content-Type': 'application/octet-stream'}

# If authentication is enabled, set the authorization header
headers['Authorization'] = f'Bearer {key}'

# Make the request and display the response
resp = requests.post(scoring_uri, data, headers=headers)
print(resp.text)
```

Visualize detections

Now that you have scored a test image, you can visualize the bounding boxes for this image. To do so, be sure you have matplotlib installed.

```
%pip install --upgrade matplotlib
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import matplotlib.patches as patches
from PIL import Image
import numpy as np
import json

IMAGE_SIZE = (18,12)
plt.figure(figsize=IMAGE_SIZE)
img_np=mpimg.imread(sample_image)
img = Image.fromarray(img_np.astype('uint8'), 'RGB')
x, y = img.size

fig,ax = plt.subplots(1, figsize=(15,15))
# Display the image
ax.imshow(img_np)

# draw box and label for each detection
detections = json.loads(resp.text)
for detect in detections['boxes']:
    label = detect['label']
    box = detect['box']
    conf_score = detect['score']
    if conf_score > 0.6:
        ymin, xmin, ymax, xmax = box['topY'],box['topX'], box['bottomY'],box['bottomX']
        topleft_x, topleft_y = x * xmin, y * ymin
        width, height = x * (xmax - xmin), y * (ymax - ymin)
        print('{}: [{}], [{}], [{}], [{}], [{}].format(detect['label'], round(topleft_x, 3),
                                                       round(topleft_y, 3), round(width, 3),
                                                       round(height, 3), round(conf_score, 3)))')

        color = np.random.rand(3) #'red'
        rect = patches.Rectangle((topleft_x, topleft_y), width, height,
                               linewidth=3, edgecolor=color, facecolor='none')

        ax.add_patch(rect)
        plt.text(topleft_x, topleft_y - 10, label, color=color, fontsize=20)

plt.show()
```

Clean up resources

Do not complete this section if you plan on running other Azure Machine Learning tutorials.

If you don't plan to use the resources you created, delete them, so you don't incur any charges.

1. In the Azure portal, select **Resource groups** on the far left.
2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

You can also keep the resource group but delete a single workspace. Display the workspace properties and select **Delete**.

Next steps

In this automated machine learning tutorial, you did the following tasks:

- Configured a workspace and prepared data for an experiment.
- Trained an automated object detection model

- Specified hyperparameter values for your model
 - Performed a hyperparameter sweep
 - Deployed your model
 - Visualized detections
-
- [Learn more about computer vision in automated ML \(preview\).](#)
 - [Learn how to set up AutoML to train computer vision models with Python \(preview\).](#)
 - [Learn how to configure incremental training on computer vision models.](#)
 - See [what hyperparameters are available for computer vision tasks.](#)
-
- Review detailed code examples and use cases in the [GitHub notebook repository for automated machine learning samples](#). Please check the folders with 'image-' prefix for samples specific to building computer vision models.

NOTE

Use of the fridge objects dataset is available through the license under the [MIT License](#).

Set up AutoML to train a natural language processing model with Python (preview)

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this article, you learn how to train natural language processing (NLP) models with [automated ML](#) in the [Azure Machine Learning Python SDK](#).

Automated ML supports NLP which allows ML professionals and data scientists to bring their own text data and build custom models for tasks such as, multi-class text classification, multi-label text classification, and named entity recognition (NER).

You can seamlessly integrate with the [Azure Machine Learning data labeling](#) capability to label your text data or bring your existing labeled data. Automated ML provides the option to use distributed training on multi-GPU compute clusters for faster model training. The resulting model can be operationalized at scale by leveraging Azure ML's MLOps capabilities.

Prerequisites

- Azure subscription. If you don't have an Azure subscription, sign up to try the [free or paid version of Azure Machine Learning](#) today.
- An Azure Machine Learning workspace with a GPU training compute. To create the workspace, see [Create workspace resources](#). See [GPU optimized virtual machine sizes](#) for more details of GPU instances provided by Azure.

WARNING

Support for multilingual models and the use of models with longer max sequence length is necessary for several NLP use cases, such as non-english datasets and longer range documents. As a result, these scenarios may require higher GPU memory for model training to succeed, such as the NC_v3 series or the ND series.

- The Azure Machine Learning Python SDK installed.

To install the SDK you can either,

- Create a compute instance, which automatically installs the SDK and is pre-configured for ML workflows. See [Create and manage an Azure Machine Learning compute instance](#) for more information.
- [Install the `automl` package yourself](#), which includes the [default installation](#) of the SDK.

IMPORTANT

The Python commands in this article require the latest `azureml-train-automl` package version.

- [Install the latest `azureml-train-automl` package to your local environment.](#)
- For details on the latest `azureml-train-automl` package, see the [release notes](#).

- This article assumes some familiarity with setting up an automated machine learning experiment. Follow the [tutorial](#) or [how-to](#) to see the main automated machine learning experiment design patterns.

Select your NLP task

Determine what NLP task you want to accomplish. Currently, automated ML supports the follow deep neural network NLP tasks.

TASK	AUTOMLCONFIG SYNTAX	DESCRIPTION
Multi-class text classification	<code>task = 'text-classification'</code>	<p>There are multiple possible classes and each sample can be classified as exactly one class. The task is to predict the correct class for each sample.</p> <p>For example, classifying a movie script as "Comedy" or "Romantic".</p>
Multi-label text classification	<code>task = 'text-classification-multilabel'</code>	<p>There are multiple possible classes and each sample can be assigned any number of classes. The task is to predict all the classes for each sample</p> <p>For example, classifying a movie script as "Comedy", or "Romantic", or "Comedy and Romantic".</p>
Named Entity Recognition (NER)	<code>task = 'text-ner'</code>	<p>There are multiple possible tags for tokens in sequences. The task is to predict the tags for all the tokens for each sequence.</p> <p>For example, extracting domain-specific entities from unstructured text, such as contracts or financial documents</p>

Preparing data

For NLP experiments in automated ML, you can bring an Azure Machine Learning dataset with `.csv` format for multi-class and multi-label classification tasks. For NER tasks, two-column `.txt` files that use a space as the separator and adhere to the CoNLL format are supported. The following sections provide additional detail for the data format accepted for each task.

Multi-class

For multi-class classification, the dataset can contain several text columns and exactly one label column. The following example has only one text column.

```

text,labels
"I love watching Chicago Bulls games.", "NBA"
"Tom Brady is a great player.", "NFL"
"There is a game between Yankees and Orioles tonight", "MLB"
"Stephen Curry made the most number of 3-Pointers", "NBA"

```

Multi-label

For multi-label classification, the dataset columns would be the same as multi-class, however there are special format requirements for data in the label column. The two accepted formats and examples are in the following table.

LABEL COLUMN FORMAT OPTIONS	MULTIPLE LABELS	ONE LABEL	NO LABELS
Plain text	"label1, label2, label3"	"label1"	""
Python list with quotes	"['label1', 'label2', 'label3']"	["['label1']"]	["[]"]

IMPORTANT

Different parsers are used to read labels for these formats. If you are using the plain text format, only use alphabetical, numerical and `_` in your labels. All other characters are recognized as the separator of labels.

For example, if your label is `"cs.AI"`, it's read as `"cs"` and `"AI"`. Whereas with the Python list format, the label would be `["['cs.AI']"]`, which is read as `"cs.AI"`.

Example data for multi-label in plain text format.

```

text,labels
"I love watching Chicago Bulls games.", "basketball"
"The four most popular leagues are NFL, MLB, NBA and NHL", "football,baseball,basketball,hockey"
"I like drinking beer.", ""

```

Example data for multi-label in Python list with quotes format.

```

text,labels
"I love watching Chicago Bulls games.", ["'basketball']"
"The four most popular leagues are NFL, MLB, NBA and NHL", ["'football','baseball','basketball','hockey']"
"I like drinking beer.", []

```

Named entity recognition (NER)

Unlike multi-class or multi-label, which takes `.csv` format datasets, named entity recognition requires CoNLL format. The file must contain exactly two columns and in each row, the token and the label is separated by a single space.

For example,

```

Hudson B-loc
Square I-loc
is O
a O
famous O
place O
in O
New B-loc
York I-loc
City I-loc

Stephen B-per
Curry I-per
got O
three O
championship O
rings O

```

Data validation

Before training, automated ML applies data validation checks on the input data to ensure that the data can be preprocessed correctly. If any of these checks fail, the run fails with the relevant error message. The following are the requirements to pass data validation checks for each task.

NOTE

Some data validation checks are applicable to both the training and the validation set, whereas others are applicable only to the training set. If the test dataset could not pass the data validation, that means that automated ML couldn't capture it and there is a possibility of model inference failure, or a decline in model performance.

TASK	DATA VALIDATION CHECK
All tasks	<ul style="list-style-type: none"> - Both training and validation sets must be provided - At least 50 training samples are required
Multi-class and Multi-label	<p>The training data and validation data must have</p> <ul style="list-style-type: none"> - The same set of columns - The same order of columns from left to right - The same data type for columns with the same name - At least two unique labels - Unique column names within each dataset (For example, the training set can't have multiple columns named Age)
Multi-class only	None
Multi-label only	<ul style="list-style-type: none"> - The label column format must be in accepted format - At least one sample should have 0 or 2+ labels, otherwise it should be a <code>multiclass</code> task - All labels should be in <code>str</code> or <code>int</code> format, with no overlapping. You should not have both label <code>1</code> and label <code>'1'</code>

TASK	DATA VALIDATION CHECK
NER only	<ul style="list-style-type: none"> - The file should not start with an empty line - Each line must be an empty line, or follow format <code>{token} {label}</code>, where there is exactly one space between the token and the label and no white space after the label - All labels must start with <code>I-</code>, <code>B-</code>, or be exactly <code>O</code>. Case sensitive - Exactly one empty line between two samples - Exactly one empty line at the end of the file

Configure experiment

Automated ML's NLP capability is triggered through `AutoMLConfig`, which is the same workflow for submitting automated ML experiments for classification, regression and forecasting tasks. You would set most of the parameters as you would for those experiments, such as `task`, `compute_target` and data inputs.

However, there are key differences:

- You can ignore `primary_metric`, as it is only for reporting purpose. Currently, automated ML only trains one model per run for NLP and there is no model selection.
- The `label_column_name` parameter is only required for multi-class and multi-label text classification tasks.
- If the majority of the samples in your dataset contain more than 128 words, it's considered long range. For this scenario, you can enable the long range text option with the `enable_long_range_text=True` parameter in your `AutoMLConfig`. Doing so, helps improve model performance but requires longer training times.
 - If you enable long range text, then a GPU with higher memory is required such as, `NCv3` series or `ND` series.
 - The `enable_long_range_text` parameter is only available for multi-class classification tasks.

```
automl_settings = {
    "verbosity": logging.INFO,
    "enable_long_range_text": True, # # You only need to set this parameter if you want to enable the long-range text setting
}

automl_config = AutoMLConfig(
    task="text-classification",
    debug_log="automl_errors.log",
    compute_target=compute_target,
    training_data=train_dataset,
    validation_data=val_dataset,
    label_column_name=target_column_name,
    **automl_settings
)
```

Language settings

As part of the NLP functionality, automated ML supports 104 languages leveraging language specific and multilingual pre-trained text DNN models, such as the BERT family of models. Currently, language selection defaults to English.

The following table summarizes what model is applied based on task type and language. See the full list of [supported languages and their codes](#).

Task Type	Syntax for <code>dataset_language</code>	Text Model Algorithm
Multi-label text classification	<pre>'eng' 'deu' 'mul'</pre>	English BERT uncased German BERT Multilingual BERT For all other languages, automated ML applies multilingual BERT
Multi-class text classification	<pre>'eng' 'deu' 'mul'</pre>	English BERT cased Multilingual BERT For all other languages, automated ML applies multilingual BERT
Named entity recognition (NER)	<pre>'eng' 'deu' 'mul'</pre>	English BERT cased German BERT Multilingual BERT For all other languages, automated ML applies multilingual BERT

You can specify your dataset language in your `FeaturizationConfig`. BERT is also used in the featurization process of automated ML experiment training, learn more about [BERT integration and featurization in automated ML](#).

```
from azureml.automl.core.featurization import FeaturizationConfig  
  
featurization_config = FeaturizationConfig(dataset_language='{your language code}')  
automl_config = AutoMLConfig("featurization": featurization_config)
```

Distributed training

You can also run your NLP experiments with distributed training on an Azure ML compute cluster. This is handled automatically by automated ML when the parameters `max_concurrent_iterations = number_of_vms` and `enable_distributed_dnn_training = True` are provided in your `AutoMLConfig` during experiment set up.

```
max_concurrent_iterations = number_of_vms  
enable_distributed_dnn_training = True
```

Doing so, schedules distributed training of the NLP models and automatically scales to every GPU on your virtual machine or cluster of virtual machines. The max number of virtual machines allowed is 32. The training is scheduled with number of virtual machines that is in powers of two.

Example notebooks

See the sample notebooks for detailed code examples for each NLP task.

- [Multi-class text classification](#)
- [Multi-label text classification](#)
- [Named entity recognition](#)

Next steps

- Learn more about [how and where to deploy a model](#).

- Troubleshoot automated ML experiments.

Set up AutoML to train computer vision models with Python (v1)

9/21/2022 • 15 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this article, you learn how to train computer vision models on image data with automated ML in the [Azure Machine Learning Python SDK](#).

Automated ML supports model training for computer vision tasks like image classification, object detection, and instance segmentation. Authoring AutoML models for computer vision tasks is currently supported via the Azure Machine Learning Python SDK. The resulting experimentation runs, models, and outputs are accessible from the Azure Machine Learning studio UI. [Learn more about automated ml for computer vision tasks on image data](#).

NOTE

Automated ML for computer vision tasks is only available via the Azure Machine Learning Python SDK.

Prerequisites

- An Azure Machine Learning workspace. To create the workspace, see [Create workspace resources](#).
- The Azure Machine Learning Python SDK installed. To install the SDK you can either,
 - Create a compute instance, which automatically installs the SDK and is pre-configured for ML workflows. For more information, see [Create and manage an Azure Machine Learning compute instance](#).
 - Install the `automl` package yourself, which includes the [default installation](#) of the SDK.

NOTE

Only Python 3.6 and 3.7 are compatible with automated ML support for computer vision tasks.

Select your task type

Automated ML for images supports the following task types:

TASK TYPE	AUTOMLIMAGE CONFIG SYNTAX
image classification	<code>ImageTask.IMAGE_CLASSIFICATION</code>
image classification multi-label	<code>ImageTask.IMAGE_CLASSIFICATION_MULTILABEL</code>
image object detection	<code>ImageTask.IMAGE_OBJECT_DETECTION</code>
image instance segmentation	<code>ImageTask.IMAGE_INSTANCE_SEGMENTATION</code>

This task type is a required parameter and is passed in using the `task` parameter in the [AutoMLImageConfig](#).

For example:

```
from azureml.train.automl import AutoMLImageConfig  
from azureml.automl.core.shared.constants import ImageTask  
automl_image_config = AutoMLImageConfig(task=ImageTask.IMAGE_OBJECT_DETECTION)
```

Training and validation data

In order to generate computer vision models, you need to bring labeled image data as input for model training in the form of an Azure Machine Learning [TabularDataset](#). You can either use a `TabularDataset` that you have [exported from a data labeling project](#), or create a new `TabularDataset` with your labeled training data.

If your training data is in a different format (like, pascal VOC or COCO), you can apply the helper scripts included with the sample notebooks to convert the data to JSONL. Learn more about how to [prepare data for computer vision tasks with automated ML](#).

WARNING

Creation of TabularDatasets from data in JSONL format is supported using the SDK only, for this capability. Creating the dataset via UI is not supported at this time. As of now, the UI doesn't recognize the StreamInfo datatype, which is the datatype used for image URLs in JSONL format.

NOTE

The training dataset needs to have at least 10 images in order to be able to submit an AutoML run.

JSONL schema samples

The structure of the TabularDataset depends upon the task at hand. For computer vision task types, it consists of the following fields:

FIELD	DESCRIPTION
<code>image_url</code>	Contains filepath as a StreamInfo object
<code>image_details</code>	Image metadata information consists of height, width, and format. This field is optional and hence may or may not exist.

FIELD	DESCRIPTION
label	A json representation of the image label, based on the task type.

The following is a sample JSONL file for image classification:

```
{
  "image_url": "AmlDatastore://image_data/Image_01.png",
  "image_details":
  {
    "format": "png",
    "width": "2230px",
    "height": "4356px"
  },
  "label": "cat"
}

{
  "image_url": "AmlDatastore://image_data/Image_02.jpeg",
  "image_details":
  {
    "format": "jpeg",
    "width": "3456px",
    "height": "3467px"
  },
  "label": "dog"
}
```

The following code is a sample JSONL file for object detection:

```
{
  "image_url": "AmlDatastore://image_data/Image_01.png",
  "image_details":
  {
    "format": "png",
    "width": "2230px",
    "height": "4356px"
  },
  "label":
  {
    "label": "cat",
    "topX": "1",
    "topY": "0",
    "bottomX": "0",
    "bottomY": "1",
    "isCrowd": "true",
  }
}
{
  "image_url": "AmlDatastore://image_data/Image_02.png",
  "image_details":
  {
    "format": "jpeg",
    "width": "1230px",
    "height": "2356px"
  },
  "label":
  {
    "label": "dog",
    "topX": "0",
    "topY": "1",
    "bottomX": "0",
    "bottomY": "1",
    "isCrowd": "false",
  }
}
```

Consume data

Once your data is in JSONL format, you can create a TabularDataset with the following code:

```
ws = Workspace.from_config()
ds = ws.get_default_datastore()
from azureml.core import Dataset

training_dataset = Dataset.Tabular.from_json_lines_files(
    path=ds.path('odFridgeObjects/odFridgeObjects.jsonl'),
    set_column_types={'image_url': DataType.to_stream(ds.workspace)})
training_dataset = training_dataset.register(workspace=ws, name=training_dataset_name)
```

Automated ML does not impose any constraints on training or validation data size for computer vision tasks. Maximum dataset size is only limited by the storage layer behind the dataset (i.e. blob store). There is no minimum number of images or labels. However, we recommend to start with a minimum of 10-15 samples per label to ensure the output model is sufficiently trained. The higher the total number of labels/classes, the more samples you need per label.

Training data is a required and is passed in using the `training_data` parameter. You can optionally specify another TabularDataset as a validation dataset to be used for your model with the `validation_data` parameter of the AutoMLImageConfig. If no validation dataset is specified, 20% of your training data will be used for validation by default, unless you pass `validation_size` argument with a different value.

For example:

```
from azureml.train.automl import AutoMLImageConfig
automl_image_config = AutoMLImageConfig(training_data=training_dataset)
```

Compute to run experiment

Provide a [compute target](#) for automated ML to conduct model training. Automated ML models for computer vision tasks require GPU SKUs and support NC and ND families. We recommend the NCv3-series (with v100 GPUs) for faster training. A compute target with a multi-GPU VM SKU leverages multiple GPUs to also speed up training. Additionally, when you set up a compute target with multiple nodes you can conduct faster model training through parallelism when tuning hyperparameters for your model.

The compute target is a required parameter and is passed in using the `compute_target` parameter of the `AutoMLImageConfig`. For example:

```
from azureml.train.automl import AutoMLImageConfig
automl_image_config = AutoMLImageConfig(compute_target=compute_target)
```

Configure model algorithms and hyperparameters

With support for computer vision tasks, you can control the model algorithm and sweep hyperparameters. These model algorithms and hyperparameters are passed in as the parameter space for the sweep.

The model algorithm is required and is passed in via `model_name` parameter. You can either specify a single `model_name` or choose between multiple.

Supported model algorithms

The following table summarizes the supported models for each computer vision task.

TASK	MODEL ALGORITHMS	STRING LITERAL SYNTAX <small>DEFAULT_MODEL * DENOTED WITH *</small>
Image classification (multi-class and multi-label)	MobileNet : Light-weighted models for mobile applications ResNet : Residual networks ResNeSt : Split attention networks SE-ResNeXt50 : Squeeze-and-Excitation networks ViT : Vision transformer networks	<code>mobilenetv2</code> <code>resnet18</code> <code>resnet34</code> <code>resnet50</code> <code>resnet101</code> <code>resnet152</code> <code>resnest50</code> <code>resnest101</code> <code>seresnext</code> <code>vits16r224</code> (small) <code>vitb16r224</code> * (base) <code>vitl16r224</code> (large)
Object detection	YOLOv5 : One stage object detection model Faster RCNN ResNet FPN : Two stage object detection models RetinaNet ResNet FPN : address class imbalance with Focal Loss <i>Note: Refer to <code>model_size</code> hyperparameter for YOLOv5 model sizes.</i>	<code>yolov5</code> * <code>fasterrcnn_resnet18_fpn</code> <code>fasterrcnn_resnet34_fpn</code> <code>fasterrcnn_resnet50_fpn</code> <code>fasterrcnn_resnet101_fpn</code> <code>fasterrcnn_resnet152_fpn</code> <code>retinanet_resnet50_fpn</code>

Task	Model Algorithms	String Literal Syntax <small>DEFAULT_MODEL * DENOTED WITH *</small>
Instance segmentation	MaskRCNN ResNet FPN	maskrcnn_resnet18_fpn maskrcnn_resnet34_fpn maskrcnn_resnet50_fpn * maskrcnn_resnet101_fpn maskrcnn_resnet152_fpn maskrcnn_resnet50_fpn

In addition to controlling the model algorithm, you can also tune hyperparameters used for model training. While many of the hyperparameters exposed are model-agnostic, there are instances where hyperparameters are task-specific or model-specific. [Learn more about the available hyperparameters for these instances.](#)

Data augmentation

In general, deep learning model performance can often improve with more data. Data augmentation is a practical technique to amplify the data size and variability of a dataset which helps to prevent overfitting and improve the model's generalization ability on unseen data. Automated ML applies different data augmentation techniques based on the computer vision task, before feeding input images to the model. Currently, there is no exposed hyperparameter to control data augmentations.

Task	Impacted Dataset	Data Augmentation Technique(s) Applied
Image classification (multi-class and multi-label)	Training Validation & Test	Random resize and crop, horizontal flip, color jitter (brightness, contrast, saturation, and hue), normalization using channel-wise ImageNet's mean and standard deviation Resize, center crop, normalization
Object detection, instance segmentation	Training Validation & Test	Random crop around bounding boxes, expand, horizontal flip, normalization, resize Normalization, resize
Object detection using yolov5	Training Validation & Test	Mosaic, random affine (rotation, translation, scale, shear), horizontal flip Letterbox resizing

Configure your experiment settings

Before doing a large sweep to search for the optimal models and hyperparameters, we recommend trying the default values to get a first baseline. Next, you can explore multiple hyperparameters for the same model before sweeping over multiple models and their parameters. This way, you can employ a more iterative approach, because with multiple models and multiple hyperparameters for each, the search space grows exponentially and you need more iterations to find optimal configurations.

If you wish to use the default hyperparameter values for a given algorithm (say yolov5), you can specify the config for your AutoML Image runs as follows:

```

from azureml.train.automl import AutoMLImageConfig
from azureml.train.hyperdrive import GridParameterSampling, choice
from azureml.core.shared.constants import ImageTask

automl_image_config_yolov5 = AutoMLImageConfig(task=ImageTask.IMAGE_OBJECT_DETECTION,
                                              compute_target=compute_target,
                                              training_data=training_dataset,
                                              validation_data=validation_dataset,
                                              hyperparameter_sampling=GridParameterSampling({'model_name':
choice('yolov5')}),
                                              iterations=1)

```

Once you've built a baseline model, you might want to optimize model performance in order to sweep over the model algorithm and hyperparameter space. You can use the following sample config to sweep over the hyperparameters for each algorithm, choosing from a range of values for learning_rate, optimizer, lr_scheduler, etc., to generate a model with the optimal primary metric. If hyperparameter values are not specified, then default values are used for the specified algorithm.

Primary metric

The primary metric used for model optimization and hyperparameter tuning depends on the task type. Using other primary metric values is currently not supported.

- `accuracy` for IMAGE_CLASSIFICATION
- `iou` for IMAGE_CLASSIFICATION_MULTILABEL
- `mean_average_precision` for IMAGE_OBJECT_DETECTION
- `mean_average_precision` for IMAGE_INSTANCE_SEGMENTATION

Experiment budget

You can optionally specify the maximum time budget for your AutoML Vision experiment using `experiment_timeout_hours` - the amount of time in hours before the experiment terminates. If none specified, default experiment timeout is seven days (maximum 60 days).

Sweeping hyperparameters for your model

When training computer vision models, model performance depends heavily on the hyperparameter values selected. Often, you might want to tune the hyperparameters to get optimal performance. With support for computer vision tasks in automated ML, you can sweep hyperparameters to find the optimal settings for your model. This feature applies the hyperparameter tuning capabilities in Azure Machine Learning. [Learn how to tune hyperparameters](#).

Define the parameter search space

You can define the model algorithms and hyperparameters to sweep in the parameter space.

- See [Configure model algorithms and hyperparameters](#) for the list of supported model algorithms for each task type.
- See [Hyperparameters for computer vision tasks](#) hyperparameters for each computer vision task type.
- See [details on supported distributions for discrete and continuous hyperparameters](#).

Sampling methods for the sweep

When sweeping hyperparameters, you need to specify the sampling method to use for sweeping over the defined parameter space. Currently, the following sampling methods are supported with the `hyperparameter_sampling` parameter:

- [Random sampling](#)
- [Grid sampling](#)

- [Bayesian sampling](#)

NOTE

Currently only random sampling supports conditional hyperparameter spaces.

Early termination policies

You can automatically end poorly performing runs with an early termination policy. Early termination improves computational efficiency, saving compute resources that would have been otherwise spent on less promising configurations. Automated ML for images supports the following early termination policies using the `early_termination_policy` parameter. If no termination policy is specified, all configurations are run to completion.

- [Bandit policy](#)
- [Median stopping policy](#)
- [Truncation selection policy](#)

Learn more about [how to configure the early termination policy for your hyperparameter sweep](#).

Resources for the sweep

You can control the resources spent on your hyperparameter sweep by specifying the `iterations` and the `max_concurrent_iterations` for the sweep.

PARAMETER	DETAIL
<code>iterations</code>	Required parameter for maximum number of configurations to sweep. Must be an integer between 1 and 1000. When exploring just the default hyperparameters for a given model algorithm, set this parameter to 1.
<code>max_concurrent_iterations</code>	Maximum number of runs that can run concurrently. If not specified, all runs launch in parallel. If specified, must be an integer between 1 and 100. NOTE: The number of concurrent runs is gated on the resources available in the specified compute target. Ensure that the compute target has the available resources for the desired concurrency.

NOTE

For a complete sweep configuration sample, please refer to this [tutorial](#).

Arguments

You can pass fixed settings or parameters that don't change during the parameter space sweep as arguments. Arguments are passed in name-value pairs and the name must be prefixed by a double dash.

```
from azureml.train.automl import AutoMLImageConfig
arguments = ["--early_stopping", 1, "--evaluation_frequency", 2]
automl_image_config = AutoMLImageConfig(arguments=arguments)
```

Incremental training (optional)

Once the training run is done, you have the option to further train the model by loading the trained model

checkpoint. You can either use the same dataset or a different one for incremental training.

There are two available options for incremental training. You can,

- Pass the run ID that you want to load the checkpoint from.
- Pass the checkpoints through a FileDataset.

Pass the checkpoint via run ID

To find the run ID from the desired model, you can use the following code.

```
# find a run id to get a model checkpoint from
target_checkpoint_run = automl_image_run.get_best_child()
```

To pass a checkpoint via the run ID, you need to use the `checkpoint_run_id` parameter.

```
automl_image_config = AutoMLImageConfig(task='image-object-detection',
                                         compute_target=compute_target,
                                         training_data=training_dataset,
                                         validation_data=validation_dataset,
                                         checkpoint_run_id= target_checkpoint_run.id,
                                         primary_metric='mean_average_precision',
                                         **tuning_settings)

automl_image_run = experiment.submit(automl_image_config)
automl_image_run.wait_for_completion(wait_post_processing=True)
```

Pass the checkpoint via FileDataset

To pass a checkpoint via a FileDataset, you need to use the `checkpoint_dataset_id` and `checkpoint_filename` parameters.

```
# download the checkpoint from the previous run
model_name = "outputs/model.pt"
model_local = "checkpoints/model_yolo.pt"
target_checkpoint_run.download_file(name=model_name, output_file_path=model_local)

# upload the checkpoint to the blob store
ds.upload(src_dir="checkpoints", target_path='checkpoints')

# create a FileDataset for the checkpoint and register it with your workspace
ds_path = ds.path('checkpoints/model_yolo.pt')
checkpoint_yolo = Dataset.File.from_files(path=ds_path)
checkpoint_yolo = checkpoint_yolo.register(workspace=ws, name='yolo_checkpoint')

automl_image_config = AutoMLImageConfig(task='image-object-detection',
                                         compute_target=compute_target,
                                         training_data=training_dataset,
                                         validation_data=validation_dataset,
                                         checkpoint_dataset_id= checkpoint_yolo.id,
                                         checkpoint_filename='model_yolo.pt',
                                         primary_metric='mean_average_precision',
                                         **tuning_settings)

automl_image_run = experiment.submit(automl_image_config)
automl_image_run.wait_for_completion(wait_post_processing=True)
```

Submit the run

When you have your `AutoMLImageConfig` object ready, you can submit the experiment.

```
ws = Workspace.from_config()
experiment = Experiment(ws, "Tutorial-automl-image-object-detection")
automl_image_run = experiment.submit(automl_image_config)
```

Outputs and evaluation metrics

The automated ML training runs generates output model files, evaluation metrics, logs and deployment artifacts like the scoring file and the environment file which can be viewed from the outputs and logs and metrics tab of the child runs.

TIP

Check how to navigate to the job results from the [View run results](#) section.

For definitions and examples of the performance charts and metrics provided for each run, see [Evaluate automated machine learning experiment results](#)

Register and deploy model

Once the run completes, you can register the model that was created from the best run (configuration that resulted in the best primary metric)

```
best_child_run = automl_image_run.get_best_child()
model_name = best_child_run.properties['model_name']
model = best_child_run.register_model(model_name = model_name, model_path='outputs/model.pt')
```

After you register the model you want to use, you can deploy it as a web service on [Azure Container Instances \(ACI\)](#) or [Azure Kubernetes Service \(AKS\)](#). ACI is the perfect option for testing deployments, while AKS is better suited for high-scale, production usage.

This example deploys the model as a web service in AKS. To deploy in AKS, first create an AKS compute cluster or use an existing AKS cluster. You can use either GPU or CPU VM SKUs for your deployment cluster.

```
from azureml.core.compute import ComputeTarget, AksCompute
from azureml.exceptions import ComputeTargetException

# Choose a name for your cluster
aks_name = "cluster-aks-gpu"

# Check to see if the cluster already exists
try:
    aks_target = ComputeTarget(workspace=ws, name=aks_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    # Provision AKS cluster with GPU machine
    prov_config = AksCompute.provisioning_configuration(vm_size="STANDARD_NC6",
                                                          location="eastus2")
    # Create the cluster
    aks_target = ComputeTarget.create(workspace=ws,
                                      name=aks_name,
                                      provisioning_configuration=prov_config)
    aks_target.wait_for_completion(show_output=True)
```

Next, you can define the inference configuration, that describes how to set up the web-service containing your

model. You can use the scoring script and the environment from the training run in your inference config.

```
from azureml.core.model import InferenceConfig

best_child_run.download_file('outputs/scoring_file_v_1_0_0.py', output_file_path='score.py')
environment = best_child_run.get_environment()
inference_config = InferenceConfig(entry_script='score.py', environment=environment)
```

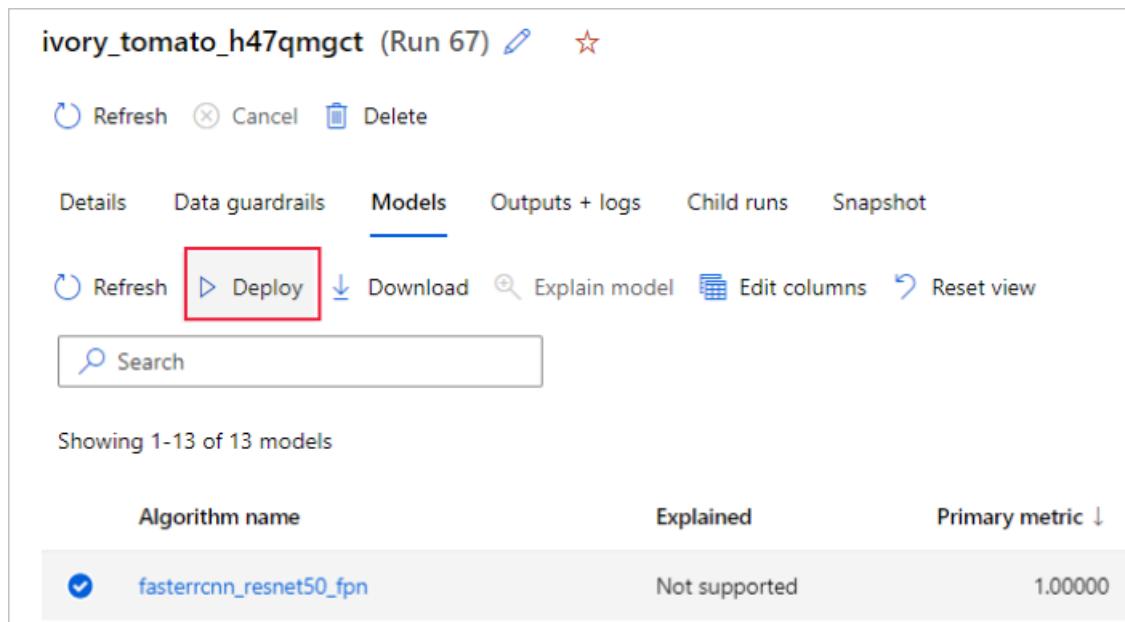
You can then deploy the model as an AKS web service.

```
# Deploy the model from the best run as an AKS web service
from azureml.core.webservice import AksWebservice
from azureml.core.webservice import Webservice
from azureml.core.model import Model
from azureml.core.environment import Environment

aks_config = AksWebservice.deploy_configuration(autoscale_enabled=True,
                                                cpu_cores=1,
                                                memory_gb=50,
                                                enable_app_insights=True)

aks_service = Model.deploy(ws,
                           models=[model],
                           inference_config=inference_config,
                           deployment_config=aks_config,
                           deployment_target=aks_target,
                           name='automl-image-test',
                           overwrite=True)
aks_service.wait_for_deployment(show_output=True)
print(aks_service.state)
```

Alternatively, you can deploy the model from the [Azure Machine Learning studio UI](#). Navigate to the model you wish to deploy in the **Models** tab of the automated ML run and select the **Deploy**.



The screenshot shows the Azure Machine Learning studio UI for a run named "ivory_tomato_h47qmgct (Run 67)". The "Models" tab is selected. At the top, there are buttons for Refresh, Cancel, Delete, and a star icon. Below the tabs, there are buttons for Refresh, Deploy (which is highlighted with a red box), Download, Explain model, Edit columns, and Reset view. A search bar is also present. The main area displays 13 models, with one row visible:

Algorithm name	Explained	Primary metric ↓
fasterrcnn_resnet50_fpn	Not supported	1.00000

You can configure the model deployment endpoint name and the inferencing cluster to use for your model deployment in the **Deploy a model** pane.

Deploy a model

Name * [\(i\)](#)

Description [\(i\)](#)

Compute type * [\(i\)](#)

Azure Kubernetes Service

Compute name * [\(i\)](#)

Select or search by name

Models: AutoMLfdbacdb4b0

Enable authentication

Type

Token-based authentication

This model supports [no-code deployment](#). You may **optionally** override the default environment and driver file.

Use custom deployment assets

Use custom deployment assets

[Advanced](#)

[Deploy](#) [Cancel](#)

Update inference configuration

In the previous step, we downloaded the scoring file `outputs/scoring_file_v_1_0_0.py` from the best model into a local `score.py` file and we used it to create an `InferenceConfig` object. This script can be modified to change the model specific inference settings if needed after it has been downloaded and before creating the `InferenceConfig`. For instance, this is the code section that initializes the model in the scoring file:

```

...
def init():
    ...
    try:
        logger.info("Loading model from path: {}".format(model_path))
        model_settings = {...}
        model = load_model(TASK_TYPE, model_path, **model_settings)
        logger.info("Loading successful.")
    except Exception as e:
        logging_utilities.log_traceback(e, logger)
        raise
...

```

Each of the tasks (and some models) have a set of parameters in the `model_settings` dictionary. By default, we use the same values for the parameters that were used during the training and validation. Depending on the behavior that we need when using the model for inference, we can change these parameters. Below you can find a list of parameters for each task type and model.

TASK	PARAMETER NAME	DEFAULT
Image classification (multi-class and multi-label)	<code>valid_resize_size</code> <code>valid_crop_size</code>	256 224
Object detection	<code>min_size</code> <code>max_size</code> <code>box_score_thresh</code> <code>nms_iou_thresh</code> <code>box_detections_per_img</code>	600 1333 0.3 0.5 100
Object detection using <code>yolov5</code>	<code>img_size</code> <code>model_size</code> <code>box_score_thresh</code> <code>nms_iou_thresh</code>	640 medium 0.1 0.5
Instance segmentation	<code>min_size</code> <code>max_size</code> <code>box_score_thresh</code> <code>nms_iou_thresh</code> <code>box_detections_per_img</code> <code>mask_pixel_score_threshold</code> <code>max_number_of_polygon_points</code> <code>export_as_image</code> <code>image_type</code>	600 1333 0.3 0.5 100 0.5 100 False JPEG

For a detailed description on task specific hyperparameters, please refer to [Hyperparameters for computer vision tasks in automated machine learning](#).

If you want to use tiling, and want to control tiling behavior, the following parameters are available:

`tile_grid_size`, `tile_overlap_ratio` and `tile_predictions_nms_thresh`. For more details on these parameters please check [Train a small object detection model using AutoML](#).

Example notebooks

Review detailed code examples and use cases in the [GitHub notebook repository for automated machine learning samples](#). Please check the folders with 'image-' prefix for samples specific to building computer vision models.

Next steps

- [Tutorial: Train an object detection model \(preview\) with AutoML and Python.](#)
- [Make predictions with ONNX on computer vision models from AutoML](#)
- [Troubleshoot automated ML experiments.](#)

Make predictions with ONNX on computer vision models from AutoML (v1)

9/21/2022 • 33 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

In this article, you learn how to use Open Neural Network Exchange (ONNX) to make predictions on computer vision models generated from automated machine learning (AutoML) in Azure Machine Learning.

To use ONNX for predictions, you need to:

1. Download ONNX model files from an AutoML training run.
2. Understand the inputs and outputs of an ONNX model.
3. Preprocess your data so it's in the required format for input images.
4. Perform inference with ONNX Runtime for Python.
5. Visualize predictions for object detection and instance segmentation tasks.

ONNX is an open standard for machine learning and deep learning models. It enables model import and export (interoperability) across the popular AI frameworks. For more details, explore the [ONNX GitHub project](#).

ONNX Runtime is an open-source project that supports cross-platform inference. ONNX Runtime provides APIs across programming languages (including Python, C++, C#, C, Java, and JavaScript). You can use these APIs to perform inference on input images. After you have the model that has been exported to ONNX format, you can use these APIs on any programming language that your project needs.

In this guide, you'll learn how to use [Python APIs for ONNX Runtime](#) to make predictions on images for popular vision tasks. You can use these ONNX exported models across languages.

Prerequisites

- Get an AutoML-trained computer vision model for any of the supported image tasks: classification, object detection, or instance segmentation. [Learn more about AutoML support for computer vision tasks](#).
- Install the `onnxruntime` package. The methods in this article have been tested with versions 1.3.0 to 1.8.0.

Download ONNX model files

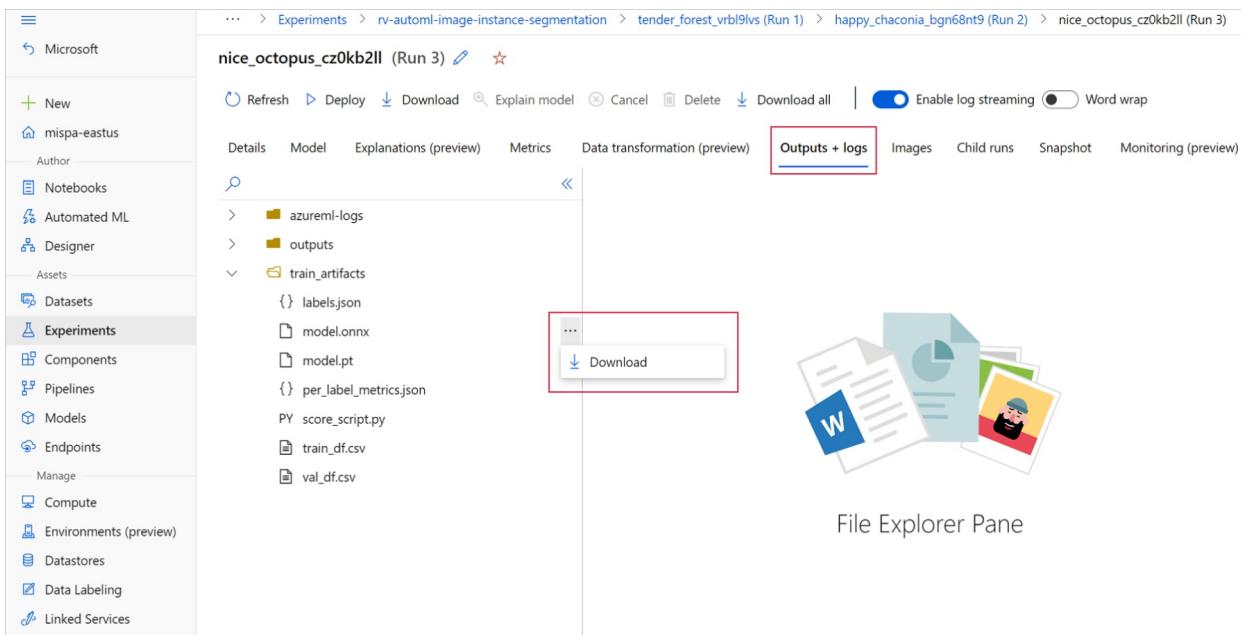
You can download ONNX model files from AutoML runs by using the Azure Machine Learning studio UI or the Azure Machine Learning Python SDK. We recommend downloading via the SDK with the experiment name and parent run ID.

Azure Machine Learning studio

On Azure Machine Learning studio, go to your experiment by using the hyperlink to the experiment generated in the training notebook, or by selecting the experiment name on the **Experiments** tab under **Assets**. Then select the best child run.

Within the best child run, go to **Outputs+logs > train_artifacts**. Use the **Download** button to manually download the following files:

- *labels.json*: File that contains all the classes or labels in the training dataset.
- *model.onnx*: Model in ONNX format.



Save the downloaded model files in a directory. The example in this article uses the */automl_models* directory.

Azure Machine Learning Python SDK

With the SDK, you can select the best child run (by primary metric) with the experiment name and parent run ID. Then, you can download the *labels.json* and *model.onnx* files.

The following code returns the best child run based on the relevant primary metric.

```
from azureml.train.automl.run import AutoMLRun

# Select the best child run
run_id = '' # Specify the run ID
automl_image_run = AutoMLRun(experiment=experiment, run_id=run_id)
best_child_run = automl_image_run.get_best_child()
```

Download the *labels.json* file, which contains all the classes and labels in the training dataset.

```
labels_file = 'automl_models/labels.json'
best_child_run.download_file(name='train_artifacts/labels.json', output_file_path=labels_file)
```

Download the *model.onnx* file.

```
onnx_model_path = 'automl_models/model.onnx'
best_child_run.download_file(name='train_artifacts/model.onnx', output_file_path=onnx_model_path)
```

Model generation for batch scoring

By default, AutoML for Images supports batch scoring for classification. But object detection and instance

segmentation models don't support batch inferencing. In case of batch inference for object detection and instance segmentation, use the following procedure to generate an ONNX model for the required batch size. Models generated for a specific batch size don't work for other batch sizes.

```
from azureml.core.script_run_config import ScriptRunConfig
from azureml.train.automl.run import AutoMLRun
from azureml.core.workspace import Workspace
from azureml.core import Experiment

# specify experiment name
experiment_name = ''
# specify workspace parameters
subscription_id = ''
resource_group = ''
workspace_name = ''
# load the workspace and compute target
ws = ''
compute_target = ''
experiment = Experiment(ws, name=experiment_name)

# specify the run id of the automl run
run_id = ''
automl_image_run = AutoMLRun(experiment=experiment, run_id=run_id)
best_child_run = automl_image_run.get_best_child()
```

Use the following model specific arguments to submit the script. For more details on arguments, refer to [model specific hyperparameters](#) and for supported object detection model names refer to the [supported model algorithm section](#).

To get the argument values needed to create the batch scoring model, refer to the scoring scripts generated under the outputs folder of the Automl training runs. Use the hyperparameter values available in the model settings variable inside the scoring file for the best child run.

- [Multi-class image classification](#)
- [Multi-label image classification](#)
- [Object detection with Faster R-CNN or RetinaNet](#)
- [Object detection with YOLO](#)
- [Instance segmentation](#)

For multi-class image classification, the generated ONNX model for the best child-run supports batch scoring by default. Therefore, no model specific arguments are needed for this task type and you can skip to the [Load the labels and ONNX model files](#) section.

Download and keep the `ONNX_batch_model_generator_automl_for_images.py` file in the current directory and submit the script. Use `ScriptRunConfig` to submit the script `ONNX_batch_model_generator_automl_for_images.py` available in the [azureml-examples GitHub repository](#), to generate an ONNX model of a specific batch size. In the following code, the trained model environment is used to submit this script to generate and save the ONNX model to the outputs directory.

```
script_run_config = ScriptRunConfig(source_directory='.',
                                    script='ONNX_batch_model_generator_automl_for_images.py',
                                    arguments=arguments,
                                    compute_target=compute_target,
                                    environment=best_child_run.get_environment())

remote_run = experiment.submit(script_run_config)
remote_run.wait_for_completion(wait_post_processing=True)
```

Once the batch model is generated, either download it from **Outputs+logs > outputs** manually, or use the following method:

```
batch_size= 8 # use the batch size used to generate the model
onnx_model_path = 'automl_models/model.onnx' # local path to save the model
remote_run.download_file(name='outputs/model_'+str(batch_size)+'.onnx', output_file_path=onnx_model_path)
```

After the model downloading step, you use the ONNX Runtime Python package to perform inferencing by using the *model.onnx* file. For demonstration purposes, this article uses the datasets from [How to prepare image datasets](#) for each vision task.

We've trained the models for all vision tasks with their respective datasets to demonstrate ONNX model inference.

Load the labels and ONNX model files

The following code snippet loads *labels.json*, where class names are ordered. That is, if the ONNX model predicts a label ID as 2, then it corresponds to the label name given at the third index in the *labels.json* file.

```
import json
import onnxruntime

labels_file = "automl_models/labels.json"
with open(labels_file) as f:
    classes = json.load(f)
print(classes)
try:
    session = onnxruntime.InferenceSession(onnx_model_path)
    print("ONNX model loaded...")
except Exception as e:
    print("Error loading ONNX file: ", str(e))
```

Get expected input and output details for an ONNX model

When you have the model, it's important to know some model-specific and task-specific details. These details include the number of inputs and number of outputs, expected input shape or format for preprocessing the image, and output shape so you know the model-specific or task-specific outputs.

```
sess_input = session.get_inputs()
sess_output = session.get_outputs()
print(f"No. of inputs : {len(sess_input)}, No. of outputs : {len(sess_output)}")

for idx, input_ in enumerate(range(len(sess_input))):
    input_name = sess_input[input_].name
    input_shape = sess_input[input_].shape
    input_type = sess_input[input_].type
    print(f"{idx} Input name : {input_name}, Input shape : {input_shape}, \
Input type : {input_type}")

for idx, output in enumerate(range(len(sess_output))):
    output_name = sess_output[output].name
    output_shape = sess_output[output].shape
    output_type = sess_output[output].type
    print(f" {idx} Output name : {output_name}, Output shape : {output_shape}, \
Output type : {output_type}")
```

Expected input and output formats for the ONNX model

Every ONNX model has a predefined set of input and output formats.

- [Multi-class image classification](#)
- [Multi-label image classification](#)
- [Object detection with Faster R-CNN or RetinaNet](#)
- [Object detection with YOLO](#)
- [Instance segmentation](#)

This example applies the model trained on the [fridgeObjects](#) dataset with 134 images and 4 classes/labels to explain ONNX model inference. For more information on training an image classification task, see the [multi-class image classification notebook](#).

Input format

The input is a preprocessed image.

INPUT NAME	INPUT SHAPE	INPUT TYPE	DESCRIPTION
input1	(batch_size, num_channels, height, width)	ndarray(float)	Input is a preprocessed image, with the shape (1, 3, 224, 224) for a batch size of 1, and a height and width of 224. These numbers correspond to the values used for crop_size in the training example.

Output format

The output is an array of logits for all the classes/labels.

OUTPUT NAME	OUTPUT SHAPE	OUTPUT TYPE	DESCRIPTION
output1	(batch_size, num_classes)	ndarray(float)	Model returns logits (without softmax). For instance, for batch size 1 and 4 classes, it returns (1, 4).

Preprocessing

- [Multi-class image classification](#)
- [Multi-label image classification](#)
- [Object detection with Faster R-CNN or RetinaNet](#)
- [Object detection with YOLO](#)
- [Instance segmentation](#)

Perform the following preprocessing steps for the ONNX model inference:

1. Convert the image to RGB.
2. Resize the image to `valid_resize_size` and `valid_resize_size` values that correspond to the values used in the transformation of the validation dataset during training. The default value for `valid_resize_size` is 256.
3. Center crop the image to `height_onnx_crop_size` and `width_onnx_crop_size`. It corresponds to `valid_crop_size` with the default value of 224.
4. Change `HxWxC` to `CxHxW`.
5. Convert to float type.
6. Normalize with ImageNet's `mean` = [0.485, 0.456, 0.406] and `std` = [0.229, 0.224, 0.225].

If you chose different values for the [hyperparameters](#) `valid_resize_size` and `valid_crop_size` during training, then those values should be used.

Get the input shape needed for the ONNX model.

```
batch, channel, height_onnx_crop_size, width_onnx_crop_size = session.get_inputs()[0].shape  
batch, channel, height_onnx_crop_size, width_onnx_crop_size
```

Without PyTorch

```

import glob
import numpy as np
from PIL import Image

def preprocess(image, resize_size, crop_size_onnx):
    """Perform pre-processing on raw input image

    :param image: raw input image
    :type image: PIL image
    :param resize_size: value to resize the image
    :type image: Int
    :param crop_size_onnx: expected height of an input image in onnx model
    :type crop_size_onnx: Int
    :return: pre-processed image in numpy format
    :rtype: ndarray 1xCxHxW
    """

    image = image.convert('RGB')
    # resize
    image = image.resize((resize_size, resize_size))
    # center crop
    left = (resize_size - crop_size_onnx)/2
    top = (resize_size - crop_size_onnx)/2
    right = (resize_size + crop_size_onnx)/2
    bottom = (resize_size + crop_size_onnx)/2
    image = image.crop((left, top, right, bottom))

    np_image = np.array(image)
    # HWC -> CHW
    np_image = np_image.transpose(2, 0, 1) # CxHxW
    # normalize the image
    mean_vec = np.array([0.485, 0.456, 0.406])
    std_vec = np.array([0.229, 0.224, 0.225])
    norm_img_data = np.zeros(np_image.shape).astype('float32')
    for i in range(np_image.shape[0]):
        norm_img_data[i,:,:] = (np_image[i,:,:]/255 - mean_vec[i])/std_vec[i]

    np_image = np.expand_dims(norm_img_data, axis=0) # 1xCxHxW
    return np_image

# following code loads only batch_size number of images for demonstrating ONNX inference
# make sure that the data directory has at least batch_size number of images

test_images_path = "automl_models_multi_cls/test_images_dir/*" # replace with path to images
# Select batch size needed
batch_size = 8
# you can modify resize_size based on your trained model
resize_size = 256
# height and width will be the same for classification
crop_size_onnx = height_onnx_crop_size

image_files = glob.glob(test_images_path)
img_processed_list = []
for i in range(batch_size):
    img = Image.open(image_files[i])
    img_processed_list.append(preprocess(img, resize_size, crop_size_onnx))

if len(img_processed_list) > 1:
    img_data = np.concatenate(img_processed_list)
elif len(img_processed_list) == 1:
    img_data = img_processed_list[0]
else:
    img_data = None

assert batch_size == img_data.shape[0]

```

With PyTorch

```
import glob
import torch
import numpy as np
from PIL import Image
from torchvision import transforms

def _make_3d_tensor(x) -> torch.Tensor:
    """This function is for images that have less channels.

    :param x: input tensor
    :type x: torch.Tensor
    :return: return a tensor with the correct number of channels
    :rtype: torch.Tensor
    """
    return x if x.shape[0] == 3 else x.expand((3, x.shape[1], x.shape[2]))

def preprocess(image, resize_size, crop_size_onnx):
    transform = transforms.Compose([
        transforms.Resize(resize_size),
        transforms.CenterCrop(crop_size_onnx),
        transforms.ToTensor(),
        transforms.Lambda(_make_3d_tensor),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]))]

    img_data = transform(image)
    img_data = img_data.numpy()
    img_data = np.expand_dims(img_data, axis=0)
    return img_data

# following code loads only batch_size number of images for demonstrating ONNX inference
# make sure that the data directory has at least batch_size number of images

test_images_path = "automl_models_multi_cls/test_images_dir/*" # replace with path to images
# Select batch size needed
batch_size = 8
# you can modify resize_size based on your trained model
resize_size = 256
# height and width will be the same for classification
crop_size_onnx = height_onnx_crop_size

image_files = glob.glob(test_images_path)
img_processed_list = []
for i in range(batch_size):
    img = Image.open(image_files[i])
    img_processed_list.append(preprocess(img, resize_size, crop_size_onnx))

if len(img_processed_list) > 1:
    img_data = np.concatenate(img_processed_list)
elif len(img_processed_list) == 1:
    img_data = img_processed_list[0]
else:
    img_data = None

assert batch_size == img_data.shape[0]
```

Inference with ONNX Runtime

Inferencing with ONNX Runtime differs for each computer vision task.

- [Multi-class image classification](#)
- [Multi-label image classification](#)
- [Object detection with Faster R-CNN or RetinaNet](#)

- [Object detection with YOLO](#)
- [Instance segmentation](#)

```
def get_predictions_from_ONNX(onnx_session, img_data):
    """Perform predictions with ONNX runtime

    :param onnx_session: onnx model session
    :type onnx_session: class InferenceSession
    :param img_data: pre-processed numpy image
    :type img_data: ndarray with shape 1xCxHxW
    :return: scores with shapes
        (1, No. of classes in training dataset)
    :rtype: numpy array
    """

    sess_input = onnx_session.get_inputs()
    sess_output = onnx_session.get_outputs()
    print(f"No. of inputs : {len(sess_input)}, No. of outputs : {len(sess_output)}")
    # predict with ONNX Runtime
    output_names = [output.name for output in sess_output]
    scores = onnx_session.run(output_names=output_names,\n                             input_feed={sess_input[0].name: img_data})

    return scores[0]

scores = get_predictions_from_ONNX(session, img_data)
```

Postprocessing

- [Multi-class image classification](#)
- [Multi-label image classification](#)
- [Object detection with Faster R-CNN or RetinaNet](#)
- [Object detection with YOLO](#)
- [Instance segmentation](#)

Apply `softmax()` over predicted values to get classification confidence scores (probabilities) for each class. Then the prediction will be the class with the highest probability.

Without PyTorch

```
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e_x / np.sum(e_x, axis=1, keepdims=True)

conf_scores = softmax(scores)
class_preds = np.argmax(conf_scores, axis=1)
print("predicted classes:", [(class_idx, classes[class_idx]) for class_idx in class_preds])
```

With PyTorch

```
conf_scores = torch.nn.functional.softmax(torch.from_numpy(scores), dim=1)
class_preds = torch.argmax(conf_scores, dim=1)
print("predicted classes:", [(class_idx.item(), classes[class_idx]) for class_idx in class_preds])
```

Visualize predictions

- Multi-class image classification
- Multi-label image classification
- Object detection with Faster R-CNN or RetinaNet
- Object detection with YOLO
- Instance segmentation

Visualize an input image with labels

```

import matplotlib.image as mpimg
import matplotlib.pyplot as plt
%matplotlib inline

sample_image_index = 0 # change this for an image of interest from image_files list
IMAGE_SIZE = (18, 12)
plt.figure(figsize=IMAGE_SIZE)
img_np = mpimg.imread(image_files[sample_image_index])

img = Image.fromarray(img_np.astype('uint8'), 'RGB')
x, y = img.size

fig,ax = plt.subplots(1, figsize=(15, 15))
# Display the image
ax.imshow(img_np)

label = class_preds[sample_image_index]
if torch.is_tensor(label):
    label = label.item()

conf_score = conf_scores[sample_image_index]
if torch.is_tensor(conf_score):
    conf_score = np.max(conf_score.tolist())
else:
    conf_score = np.max(conf_score)

display_text = '{} ({})'.format(label, round(conf_score, 3))
print(display_text)

color = 'red'
plt.text(30, 30, display_text, color=color, fontsize=30)

plt.show()

```

Next steps

- Learn more about computer vision tasks in AutoML
- Troubleshoot AutoML experiments

Track ML models with MLflow and Azure Machine Learning

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, learn how to enable [MLflow Tracking](#) to connect Azure Machine Learning as the backend of your MLflow experiments.

[MLflow](#) is an open-source library for managing the lifecycle of your machine learning experiments. MLflow Tracking is a component of MLflow that logs and tracks your training run metrics and model artifacts, no matter your experiment's environment--locally on your computer, on a remote compute target, a virtual machine, or an [Azure Databricks cluster](#).

See [MLflow and Azure Machine Learning](#) for all supported MLflow and Azure Machine Learning functionality including MLflow Project support (preview) and model deployment.

TIP

If you want to track experiments running on Azure Databricks or Azure Synapse Analytics, see the dedicated articles [Track Azure Databricks ML experiments with MLflow and Azure Machine Learning](#) or [Track Azure Synapse Analytics ML experiments with MLflow and Azure Machine Learning](#).

NOTE

The information in this document is primarily for data scientists and developers who want to monitor the model training process. If you are an administrator interested in monitoring resource usage and events from Azure Machine Learning, such as quotas, completed training jobs, or completed model deployments, see [Monitoring Azure Machine Learning](#).

Prerequisites

- Install the `mlflow` package.
 - You can use the [MLflow Skinny](#) which is a lightweight MLflow package without SQL storage, server, UI, or data science dependencies. This is recommended for users who primarily need the tracking and logging capabilities without importing the full suite of MLflow features including deployments.
- Install the `azureml-mlflow` package.
- [Create an Azure Machine Learning Workspace](#).
 - See which access permissions you need to perform your [MLflow operations with your workspace](#).
- Install and [set up Azure Machine Learning CLI \(v1\)](#) and make sure you install the ml extension.

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-m1`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `m1`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- Install and set up [Azure Machine Learning SDK for Python](#).

Track runs from your local machine or remote compute

Tracking using MLflow with Azure Machine Learning lets you store the logged metrics and artifacts runs that were executed on your local machine into your Azure Machine Learning workspace.

Set up tracking environment

To track a run that is not running on Azure Machine Learning compute (from now on referred to as "*local compute*"), you need to point your local compute to the Azure Machine Learning MLflow Tracking URI.

NOTE

When running on Azure Compute (Azure Notebooks, Jupyter Notebooks hosted on Azure Compute Instances or Compute Clusters) you don't have to configure the tracking URI. It's automatically configured for you.

- [Using the Azure ML SDK](#)
- [Using an environment variable](#)
- [Building the MLflow tracking URI](#)

APPLIES TO: [Python SDK azureml v1](#)

You can get the Azure ML MLflow tracking URI using the [Azure Machine Learning SDK v1 for Python](#). Ensure you have the library `azureml-sdk` installed in the cluster you are using. The following sample gets the unique MLFLow tracking URI associated with your workspace. Then the method `set_tracking_uri()` points the MLflow tracking URI to that URI.

1. Using the workspace configuration file:

```
from azureml.core import Workspace
import mlflow

ws = Workspace.from_config()
mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())
```

TIP

You can download the workspace configuration file by:

1. Navigate to [Azure ML studio](#)
2. Click on the upper-right corner of the page -> Download config file.
3. Save the file `config.json` in the same directory where you are working on.

2. Using the subscription ID, resource group name and workspace name:

```
from azureml.core import Workspace
import mlflow

#Enter details of your AzureML workspace
subscription_id = '<SUBSCRIPTION_ID>'
resource_group = '<RESOURCE_GROUP>'
workspace_name = '<AZUREML_WORKSPACE_NAME>'

ws = Workspace.get(name=workspace_name,
                    subscription_id=subscription_id,
                    resource_group=resource_group)

mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())
```

Set experiment name

All MLflow runs are logged to the active experiment. By default, runs are logged to an experiment named `Default` that is automatically created for you. To configure the experiment you want to work on use MLflow command `mlflow.set_experiment()`.

```
experiment_name = 'experiment_with_mlflow'
mlflow.set_experiment(experiment_name)
```

TIP

When submitting jobs using Azure ML SDK, you can set the experiment name using the property `experiment_name` when you submit it. You don't have to configure it on your training script.

Start training run

After you set the MLflow experiment name, you can start your training run with `start_run()`. Then use `log_metric()` to activate the MLflow logging API and begin logging your training run metrics.

```
import os
from random import random

with mlflow.start_run() as mlflow_run:
    mlflow.log_param("hello_param", "world")
    mlflow.log_metric("hello_metric", random())
    os.system(f"echo 'hello world' > helloworld.txt")
    mlflow.log_artifact("helloworld.txt")
```

For details about how to log metrics, parameters and artifacts in a run using MLflow view [How to log and view metrics](#).

Track runs running on Azure Machine Learning

APPLIES TO:  [Python SDK azurerm v1](#)

Remote runs (jobs) let you train your models in a more robust and repetitive way. They can also leverage more powerful computes, such as Machine Learning Compute clusters. See [Use compute targets for model training](#) to learn about different compute options.

When submitting runs, Azure Machine Learning automatically configures MLflow to work with the workspace the run is running in. This means that there is no need to configure the MLflow tracking URI. On top of that, experiments are automatically named based on the details of the experiment submission.

IMPORTANT

When submitting training jobs to Azure Machine Learning, you don't have to configure the MLflow tracking URI on your training logic as it is already configured for you. You don't need to configure the experiment name in your training routine neither.

Creating a training routine

First, you should create a `src` subdirectory and create a file with your training code in a `train.py` file in the `src` subdirectory. All your training code will go into the `src` subdirectory, including `train.py`.

The training code is taken from this [MLflow example](#) in the Azure Machine Learning example repo.

Copy this code into the file:

```
# imports
import os
import mlflow

from random import random

# define functions
def main():
    mlflow.log_param("hello_param", "world")
    mlflow.log_metric("hello_metric", random())
    os.system(f"echo 'hello world' > helloworld.txt")
    mlflow.log_artifact("helloworld.txt")

# run functions
if __name__ == "__main__":
    # run main function
    main()
```

Configuring the experiment

You will need to use Python to submit the experiment to Azure Machine Learning. In a notebook or Python file, configure your compute and training run environment with the [Environment](#) class.

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

env = Environment(name="mlflow-env")

# Specify conda dependencies with scikit-learn and temporary pointers to mlflow extensions
cd = CondaDependencies.create(
    conda_packages=["scikit-learn", "matplotlib"],
    pip_packages=["azureml-mlflow", "pandas", "numpy"]
)

env.python.conda_dependencies = cd
```

Then, construct [ScriptRunConfig](#) with your remote compute as the compute target.

```
from azureml.core import ScriptRunConfig

src = ScriptRunConfig(source_directory="src",
                      script=training_script,
                      compute_target=<COMPUTE_NAME>,
                      environment=env)
```

With this compute and training run configuration, use the `Experiment.submit()` method to submit a run. This method automatically sets the MLflow tracking URI and directs the logging from MLflow to your Workspace.

```
from azureml.core import Experiment
from azureml.core import Workspace
ws = Workspace.from_config()

experiment_name = "experiment_with_mlflow"
exp = Experiment(workspace=ws, name=experiment_name)

run = exp.submit(src)
```

View metrics and artifacts in your workspace

The metrics and artifacts from MLflow logging are tracked in your workspace. To view them anytime, navigate to your workspace and find the experiment by name in your workspace in [Azure Machine Learning studio](#). Or run the below code.

Retrieve run metric using MLflow [get_run\(\)](#).

```
from mlflow.tracking import MlflowClient

# Use MLFlow to retrieve the run that was just completed
client = MlflowClient()
run_id = mlflow_run.info.run_id
finished_mlflow_run = MlflowClient().get_run(run_id)

metrics = finished_mlflow_run.data.metrics
tags = finished_mlflow_run.data.tags
params = finished_mlflow_run.data.params

print(metrics,tags,params)
```

To view the artifacts of a run, you can use [MlflowClient.list_artifacts\(\)](#)

```
client.list_artifacts(run_id)
```

To download an artifact to the current directory, you can use [MLflowClient.download_artifacts\(\)](#)

```
client.download_artifacts(run_id, "helloworld.txt", ".")
```

For more details about how to retrieve information from experiments and runs in Azure Machine Learning using MLflow view [Manage experiments and runs with MLflow](#).

Compare and query

Compare and query all MLflow runs in your Azure Machine Learning workspace with the following code. [Learn more about how to query runs with MLflow](#).

```
from mlflow.entities import ViewType

all_experiments = [exp.experiment_id for exp in MlflowClient().list_experiments()]
query = "metrics.hello_metric > 0"
runs = mlflow.search_runs(experiment_ids=all_experiments, filter_string=query, run_view_type=ViewType.ALL)

runs.head(10)
```

Automatic logging

With Azure Machine Learning and MLFlow, users can log metrics, model parameters and model artifacts automatically when training a model. A [variety of popular machine learning libraries](#) are supported.

To enable [automatic logging](#) insert the following code before your training code:

```
mlflow.autolog()
```

[Learn more about Automatic logging with MLflow.](#)

Manage models

Register and track your models with the [Azure Machine Learning model registry](#), which supports the MLflow model registry. Azure Machine Learning models are aligned with the MLflow model schema making it easy to export and import these models across different workflows. The MLflow-related metadata, such as run ID, is also tracked with the registered model for traceability. Users can submit training runs, register, and deploy models produced from MLflow runs.

If you want to deploy and register your production ready model in one step, see [Deploy and register MLflow models](#).

To register and view a model from a run, use the following steps:

- Once a run is complete, call the `register_model()` method.

```
# the model folder produced from a run is registered. This includes the MLmodel file, model.pkl and
# the conda.yaml.
model_path = "model"
model_uri = 'runs:/{}{}'.format(run_id, model_path)
mlflow.register_model(model_uri,"registered_model_name")
```

- View the registered model in your workspace with [Azure Machine Learning studio](#).

In the following example the registered model, `my-model` has MLflow tracking metadata tagged.

Microsoft Azure Machine Learning Studio

Microsoft > my-ws > Models > my-model:1

my-model:1

Details Versions Artifacts Endpoints Data Explanations (preview) Fairness (preview) Responsible AI (preview)

Refresh Deploy Download all Create Responsible AI dashboard (preview)

Responsible AI dashboard currently only supports MLflow format models with scikit-learn flavor. Learn more about model conversion to MLflow.

Attributes

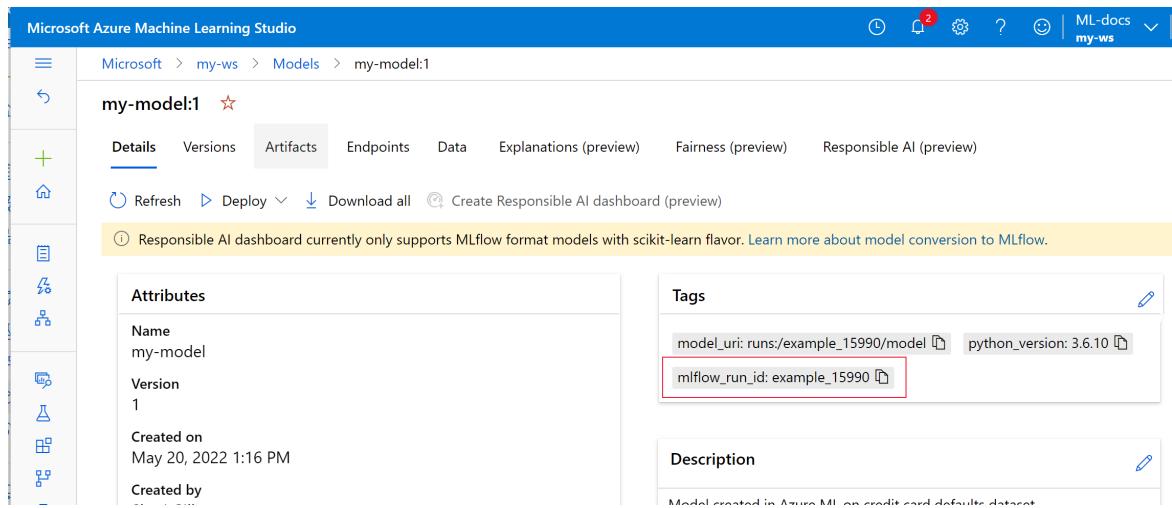
Name: my-model
Version: 1
Created on: May 20, 2022 1:16 PM
Created by: ...

Tags

model_uri: runs/example_15990/model
python_version: 3.6.10
mlflow_run_id: example_15990

Description

Model created in Azure ML on credit card default dataset



3. Select the **Artifacts** tab to see all the model files that align with the MLflow model schema (conda.yaml, MLmodel, model.pkl).

e2edemos > Models > my_model:1

my_model:1

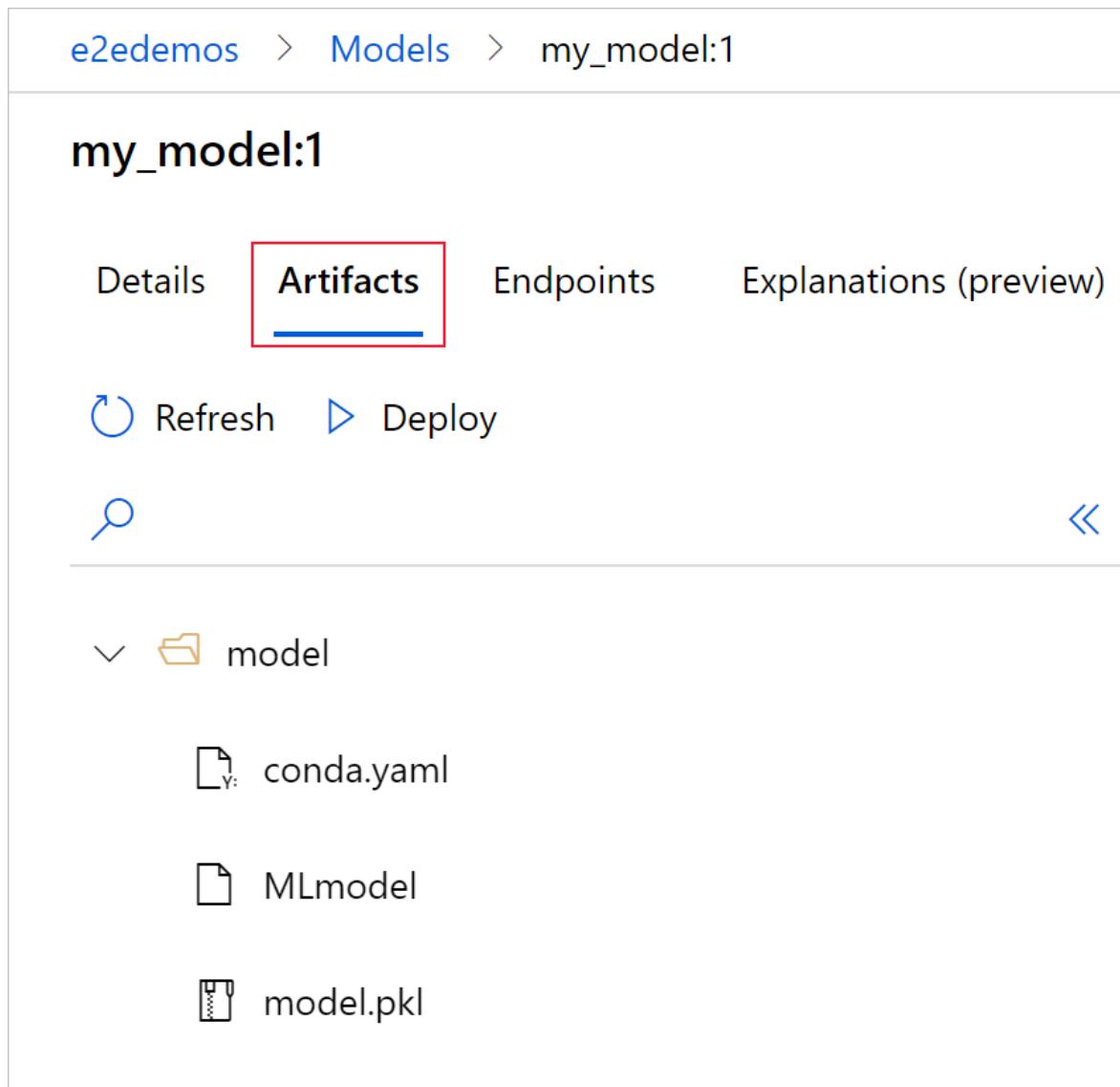
Details Artifacts Endpoints Explanations (preview)

Refresh Deploy

🔍 ⏪

model

- conda.yaml
- MLmodel
- model.pkl



4. Select MLmodel to see the MLmodel file generated by the run.

```

1 artifact_path: model
2 flavors:
3   python_function:
4     env: conda.yaml
5     loader_module: mlflow.sklearn
6     model_path: model.pkl
7     python_version: 3.6.2
8     sklearn:
9       pickled_model: model.pkl
10      serialization_format:云pickle
11      sklearn_version: 0.23.2
12 run_id: example_15990
13 utc_time_created: '2020-09-02 16:54:26.734153'
14

```

Clean up resources

If you don't plan to use the logged metrics and artifacts in your workspace, the ability to delete them individually is currently unavailable. Instead, delete the resource group that contains the storage account and workspace, so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left.

2. From the list, select the resource group you created.
3. Select **Delete resource group**.
4. Enter the resource group name. Then select **Delete**.

Example notebooks

The [MLflow with Azure ML notebooks](#) demonstrate and expand upon concepts presented in this article. Also see the community-driven repository, [AzureML-Examples](#).

Next steps

- Deploy models with MLflow.
- Monitor your production models for [data drift](#).
- [Track Azure Databricks runs with MLflow](#).
- [Manage your models](#).

Log & view metrics and log files v1

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

Log real-time information using both the default Python logging package and Azure Machine Learning Python SDK-specific functionality. You can log locally and send logs to your workspace in the portal.

Logs can help you diagnose errors and warnings, or track performance metrics like parameters and model performance. In this article, you learn how to enable logging in the following scenarios:

- Log run metrics
- Interactive training sessions
- Submitting training jobs using `ScriptRunConfig`
- Python native `logging` settings
- Logging from additional sources

TIP

This article shows you how to monitor the model training process. If you're interested in monitoring resource usage and events from Azure Machine learning, such as quotas, completed training runs, or completed model deployments, see [Monitoring Azure Machine Learning](#).

Data types

You can log multiple data types including scalar values, lists, tables, images, directories, and more. For more information, and Python code examples for different data types, see the [Run class reference page](#).

Logging run metrics

Use the following methods in the logging APIs to influence the metrics visualizations. Note the [service limits](#) for these logged metrics.

LOGGED VALUE	EXAMPLE CODE	FORMAT IN PORTAL
Log an array of numeric values	<pre>run.log_list(name='Fibonacci', value=[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89])</pre>	single-variable line chart
Log a single numeric value with the same metric name repeatedly used (like from within a for loop)	<pre>for i in tqdm(range(-10, 10)): run.log(name='Sigmoid', value=1 / (1 + np.exp(-i))) angle = i / 2.0</pre>	Single-variable line chart
Log a row with 2 numerical columns repeatedly	<pre>run.log_row(name='Cosine Wave', angle=angle, cos=np.cos(angle)) sines['angle'].append(angle) sines['sine'].append(np.sin(angle))</pre>	Two-variable line chart
Log table with 2 numerical columns	<pre>run.log_table(name='Sine Wave', value=sines)</pre>	Two-variable line chart

LOGGED VALUE	EXAMPLE CODE	FORMAT IN PORTAL
Log image	<pre>run.log_image(name='food', path='./breadpudding.jpg', plot=None, description='desert')</pre>	Use this method to log an image file or a matplotlib plot to the run. These images will be visible and comparable in the run record

Logging with MLflow

We recommend logging your models, metrics and artifacts with MLflow as it's open source and it supports local mode to cloud portability. The following table and code examples show how to use MLflow to log metrics and artifacts from your training runs. [Learn more about MLflow's logging methods and design patterns.](#)

Be sure to install the `mlflow` and `azureml-mlflow` pip packages to your workspace.

```
pip install mlflow
pip install azureml-mlflow
```

Set the MLflow tracking URI to point at the Azure Machine Learning backend to ensure that your metrics and artifacts are logged to your workspace.

```
from azureml.core import Workspace
import mlflow
from mlflow.tracking import MlflowClient

ws = Workspace.from_config()
mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())

mlflow.create_experiment("mlflow-experiment")
mlflow.set_experiment("mlflow-experiment")
mlflow_run = mlflow.start_run()
```

LOGGED VALUE	EXAMPLE CODE	NOTES
Log a numeric value (int or float)	<code>mlflow.log_metric('my_metric', 1)</code>	
Log a boolean value	<code>mlflow.log_metric('my_metric', 0)</code>	0 = True, 1 = False
Log a string	<code>mlflow.log_text('foo', 'my_string')</code>	Logged as an artifact
Log numpy metrics or PIL image objects	<code>mlflow.log_image(img, 'figure.png')</code>	
Log matplotlib plot or image file	<code>mlflow.log_figure(fig, "figure.png")</code>	

View run metrics via the SDK

You can view the metrics of a trained model using `run.get_metrics()`.

```

from azureml.core import Run
run = Run.get_context()
run.log('metric-name', metric_value)

metrics = run.get_metrics()
# metrics is of type Dict[str, List[float]] mapping metric names
# to a list of the values for that metric in the given run.

metrics.get('metric-name')
# list of metrics in the order they were recorded

```

You can also access run information using MLflow through the run object's data and info properties. See the [MLflow.entities.Run object](#) documentation for more information.

After the run completes, you can retrieve it using the `MLflowClient()`.

```

from mlflow.tracking import MlflowClient

# Use MLFlow to retrieve the run that was just completed
client = MlflowClient()
finished_mlflow_run = MlflowClient().get_run(mlflow_run.info.run_id)

```

You can view the metrics, parameters, and tags for the run in the data field of the run object.

```

metrics = finished_mlflow_run.data.metrics
tags = finished_mlflow_run.data.tags
params = finished_mlflow_run.data.params

```

NOTE

The metrics dictionary under `mlflow.entities.Run.data.metrics` only returns the most recently logged value for a given metric name. For example, if you log, in order, 1, then 2, then 3, then 4 to a metric called `sample_metric`, only 4 is present in the metrics dictionary for `sample_metric`.

To get all metrics logged for a particular metric name, you can use `MLflowClient.get_metric_history()`.

View run metrics in the studio UI

You can browse completed run records, including logged metrics, in the [Azure Machine Learning studio](#).

Navigate to the **Experiments** tab. To view all your runs in your Workspace across Experiments, select the **All runs** tab. You can drill down on runs for specific Experiments by applying the Experiment filter in the top menu bar.

For the individual Experiment view, select the **All experiments** tab. On the experiment run dashboard, you can see tracked metrics and logs for each run.

You can also edit the run list table to select multiple runs and display either the last, minimum, or maximum logged value for your runs. Customize your charts to compare the logged metrics values and aggregates across multiple runs. You can plot multiple metrics on the y-axis of your chart and customize your x-axis to plot your logged metrics.

View and download log files for a run

Log files are an essential resource for debugging the Azure ML workloads. After submitting a training job, drill down to a specific run to view its logs and outputs:

1. Navigate to the **Experiments** tab.
2. Select the runID for a specific run.
3. Select **Outputs and logs** at the top of the page.
4. Select **Download all** to download all your logs into a zip folder.
5. You can also download individual log files by choosing the log file and selecting **Download**

The screenshot shows the Azure Machine Learning interface with the 'Experiments' tab selected. In the center, a 'Run 37' card is displayed with a 'Completed' status. Below it, the 'Outputs + logs' tab is selected, indicated by a red border. To its right is a 'Download all' button, also highlighted with a red box. The main area shows a hierarchical log viewer with sections like 'azurerm-logs' and 'logs'. Under 'logs', there is a file named '70_driver_log.txt' which is currently selected and expanded, showing its contents. The log entries are numbered from 1 to 21, providing details about the experiment's execution.

```

1 [2020-09-01T03:48:03.383595] Entering context manager injector.
2 [[context_manager_injector.py] Command line Options: Namespace(inject=['ProjectPythonPath:context_managers.P
3 Starting the daemon thread to refresh tokens in background for process with pid = 5396
4 Entering Run History Context Manager.
5 Current directory: /tmp/azureml_runs/knn_project_1598931903_4234e2b8
6 Preparing to call script [ knn_train.py ] with arguments: []
7 After variable expansion, calling script [ knn_train.py ] with arguments: []
8
9 Script type = None
10 Starting the daemon thread to refresh tokens in background for process with pid = 5396
11
12
13 The experiment completed successfully. Finalizing run...
14 Logging experiment finalizing status in history service.
15 [2020-09-01T03:48:15.456191] TimeoutHandler __init__
16 [2020-09-01T03:48:15.456228] TimeoutHandler __enter__
17 Cleaning up all outstanding Run operations, waiting 300.0 seconds
18 2 items cleaning up...
19 Cleanup took 0.20339488983154297 seconds
20 [2020-09-01T03:48:16.035950] TimeoutHandler __exit__
21

```

user_logs folder

This folder contains information about the user generated logs. This folder is open by default, and the **std_log.txt** log is selected. The **std_log.txt** is where your code's logs (for example, print statements) show up. This file contains **stdout** log and **stderr** logs from your control script and training script, one per process. In the majority of cases, you will monitor the logs here.

system_logs folder

This folder contains the logs generated by Azure Machine Learning and it will be closed by default. The logs generated by the system are grouped into different folders, based on the stage of the job in the runtime.

Other folders

For jobs training on multi-compute clusters, logs are present for each node IP. The structure for each node is the same as single node jobs. There is one more logs folder for overall execution, stderr, and stdout logs.

Azure Machine Learning logs information from various sources during training, such as AutoML or the Docker container that runs the training job. Many of these logs are not documented. If you encounter problems and contact Microsoft support, they may be able to use these logs during troubleshooting.

Interactive logging session

Interactive logging sessions are typically used in notebook environments. The method [Experiment.start_logging\(\)](#) starts an interactive logging session. Any metrics logged during the session are added to the run record in the experiment. The method [run.complete\(\)](#) ends the sessions and marks the run as completed.

ScriptRun logs

In this section, you learn how to add logging code inside of runs created when configured with [ScriptRunConfig](#). You can use the [ScriptRunConfig](#) class to encapsulate scripts and environments for repeatable runs. You can also use this option to show a visual Jupyter Notebooks widget for monitoring.

This example performs a parameter sweep over alpha values and captures the results using the [run.log\(\)](#) method.

1. Create a training script that includes the logging logic, `train.py`.

```
# Copyright (c) Microsoft. All rights reserved.  
# Licensed under the MIT license.  
  
from sklearn.datasets import load_diabetes  
from sklearn.linear_model import Ridge  
from sklearn.metrics import mean_squared_error  
from sklearn.model_selection import train_test_split  
from azureml.core.run import Run  
import os  
import numpy as np  
import mylib  
# sklearn.externals.joblib is removed in 0.23  
try:  
    from sklearn.externals import joblib  
except ImportError:  
    import joblib  
  
os.makedirs('./outputs', exist_ok=True)  
  
X, y = load_diabetes(return_X_y=True)  
  
run = Run.get_context()  
  
X_train, X_test, y_train, y_test = train_test_split(X, y,  
                                                    test_size=0.2,  
                                                    random_state=0)  
data = {"train": {"X": X_train, "y": y_train},  
        "test": {"X": X_test, "y": y_test}}  
  
# list of numbers from 0.0 to 1.0 with a 0.05 interval  
alphas = mylib.get_alphas()  
  
for alpha in alphas:  
    # Use Ridge algorithm to create a regression model  
    reg = Ridge(alpha=alpha)  
    reg.fit(data["train"]["X"], data["train"]["y"])  
  
    preds = reg.predict(data["test"]["X"])  
    mse = mean_squared_error(preds, data["test"]["y"])  
    run.log('alpha', alpha)  
    run.log('mse', mse)  
  
    model_file_name = 'ridge_{0:.2f}.pkl'.format(alpha)  
    # save model in the outputs folder so it automatically get uploaded  
    with open(model_file_name, "wb") as file:  
        joblib.dump(value=reg, filename=os.path.join('./outputs/',  
                                                    model_file_name))  
  
    print('alpha is {0:.2f}, and mse is {1:.2f}'.format(alpha, mse))
```

2. Submit the `train.py` script to run in a user-managed environment. The entire script folder is submitted for training.

```
from azureml.core import ScriptRunConfig  
  
src = ScriptRunConfig(source_directory='./', script='train.py', environment=user_managed_env)
```

```
run = exp.submit(src)
```

The `show_output` parameter turns on verbose logging, which lets you see details from the training

process as well as information about any remote resources or compute targets. Use the following code to turn on verbose logging when you submit the experiment.

```
run = exp.submit(src, show_output=True)
```

You can also use the same parameter in the `wait_for_completion` function on the resulting run.

```
run.wait_for_completion(show_output=True)
```

Native Python logging

Some logs in the SDK may contain an error that instructs you to set the logging level to DEBUG. To set the logging level, add the following code to your script.

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Other logging sources

Azure Machine Learning can also log information from other sources during training, such as automated machine learning runs, or Docker containers that run the jobs. These logs aren't documented, but if you encounter problems and contact Microsoft support, they may be able to use these logs during troubleshooting.

For information on logging metrics in Azure Machine Learning designer, see [How to log metrics in the designer](#)

Example notebooks

The following notebooks demonstrate concepts in this article:

- [how-to-use-azureml/training/train-on-local](#)
- [how-to-use-azureml/track-and-monitor-experiments/logging-api](#)

Learn how to run notebooks by following the article [Use Jupyter notebooks to explore this service](#).

Next steps

See these articles to learn more on how to use Azure Machine Learning:

- See an example of how to register the best model and deploy it in the tutorial, [Train an image classification model with Azure Machine Learning](#).

Use the Python interpretability package to explain ML models & predictions (preview)

9/21/2022 • 16 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this how-to guide, you learn to use the interpretability package of the Azure Machine Learning Python SDK to perform the following tasks:

- Explain the entire model behavior or individual predictions on your personal machine locally.
- Enable interpretability techniques for engineered features.
- Explain the behavior for the entire model and individual predictions in Azure.
- Upload explanations to Azure Machine Learning Run History.
- Use a visualization dashboard to interact with your model explanations, both in a Jupyter Notebook and in the Azure Machine Learning studio.
- Deploy a scoring explainer alongside your model to observe explanations during inferencing.

For more information on the supported interpretability techniques and machine learning models, see [Model interpretability in Azure Machine Learning](#) and [sample notebooks](#).

For guidance on how to enable interpretability for models trained with automated machine learning see, [Interpretability: model explanations for automated machine learning models \(preview\)](#).

Generate feature importance value on your personal machine

The following example shows how to use the interpretability package on your personal machine without contacting Azure services.

1. Install the `azureml-interpret` package.

```
pip install azureml-interpret
```

2. Train a sample model in a local Jupyter Notebook.

```
# load breast cancer dataset, a well-known small dataset that comes with scikit-learn
from sklearn.datasets import load_breast_cancer
from sklearn import svm
from sklearn.model_selection import train_test_split
breast_cancer_data = load_breast_cancer()
classes = breast_cancer_data.target_names.tolist()

# split data into train and test
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(breast_cancer_data.data,
                                                    breast_cancer_data.target,
                                                    test_size=0.2,
                                                    random_state=0)

clf = svm.SVC(gamma=0.001, C=100., probability=True)
model = clf.fit(x_train, y_train)
```

3. Call the explainer locally.

- To initialize an explainer object, pass your model and some training data to the explainer's constructor.
- To make your explanations and visualizations more informative, you can choose to pass in feature names and output class names if doing classification.

The following code blocks show how to instantiate an explainer object with `TabularExplainer`, `MimicExplainer`, and `PFIExplainer` locally.

- `TabularExplainer` calls one of the three SHAP explainers underneath (`TreeExplainer`, `DeepExplainer`, or `KernelExplainer`).
- `TabularExplainer` automatically selects the most appropriate one for your use case, but you can call each of its three underlying explainers directly.

```
from interpret.ext.blackbox import TabularExplainer

# "features" and "classes" fields are optional
explainer = TabularExplainer(model,
                             x_train,
                             features=breast_cancer_data.feature_names,
                             classes=classes)
```

or

```
from interpret.ext.blackbox import MimicExplainer

# you can use one of the following four interpretable models as a global surrogate to the black box
model

from interpret.ext.glassbox import LGBMExplainableModel
from interpret.ext.glassbox import LinearExplainableModel
from interpret.ext.glassbox import SGDExplainableModel
from interpret.ext.glassbox import DecisionTreeExplainableModel

# "features" and "classes" fields are optional
# augment_data is optional and if true, oversamples the initialization examples to improve surrogate
model accuracy to fit original model. Useful for high-dimensional data where the number of rows is
less than the number of columns.
# max_num_of_augmentations is optional and defines max number of times we can increase the input data
size.
# LGBMExplainableModel can be replaced with LinearExplainableModel, SGDExplainableModel, or
DecisionTreeExplainableModel
explainer = MimicExplainer(model,
                           x_train,
                           LGBMExplainableModel,
                           augment_data=True,
                           max_num_of_augmentations=10,
                           features=breast_cancer_data.feature_names,
                           classes=classes)
```

or

```
from interpret.ext.blackbox import PFIExplainer

# "features" and "classes" fields are optional
explainer = PFIExplainer(model,
                         features=breast_cancer_data.feature_names,
                         classes=classes)
```

Explain the entire model behavior (global explanation)

Refer to the following example to help you get the aggregate (global) feature importance values.

```
# you can use the training data or the test data here, but test data would allow you to use Explanation
Exploration
global_explanation = explainer.explain_global(x_test)

# if you used the PFIExplainer in the previous step, use the next line of code instead
# global_explanation = explainer.explain_global(x_train, true_labels=y_train)

# sorted feature importance values and feature names
sorted_global_importance_values = global_explanation.get_ranked_global_values()
sorted_global_importance_names = global_explanation.get_ranked_global_names()
dict(zip(sorted_global_importance_names, sorted_global_importance_values))

# alternatively, you can print out a dictionary that holds the top K feature names and values
global_explanation.get_feature_importance_dict()
```

Explain an individual prediction (local explanation)

Get the individual feature importance values of different datapoints by calling explanations for an individual instance or a group of instances.

NOTE

PFIExplainer does not support local explanations.

```
# get explanation for the first data point in the test set
local_explanation = explainer.explain_local(x_test[0:5])

# sorted feature importance values and feature names
sorted_local_importance_names = local_explanation.get_ranked_local_names()
sorted_local_importance_values = local_explanation.get_ranked_local_values()
```

Raw feature transformations

You can opt to get explanations in terms of raw, untransformed features rather than engineered features. For this option, you pass your feature transformation pipeline to the explainer in `train_explain.py`. Otherwise, the explainer provides explanations in terms of engineered features.

The format of supported transformations is the same as described in [sklearn-pandas](#). In general, any transformations are supported as long as they operate on a single column so that it's clear they're one-to-many.

Get an explanation for raw features by using a `sklearn.compose.ColumnTransformer` or with a list of fitted transformer tuples. The following example uses `sklearn.compose.ColumnTransformer`.

In case you want to run the example with the list of fitted transformer tuples, use the following code:

Generate feature importance values via remote runs

The following example shows how you can use the `ExplanationClient` class to enable model interpretability for remote runs. It's conceptually similar to the local process, except you:

- Use the `ExplanationClient` in the remote run to upload the interpretability context.
- Download the context later in a local environment.

1. Install the `azureml-interpret` package.

```
pip install azureml-interpret
```

2. Create a training script in a local Jupyter Notebook. For example, `train_explain.py`.

```
from azureml.interpret import ExplanationClient
from azureml.core.run import Run
from interpret.ext.blackbox import TabularExplainer

run = Run.get_context()
client = ExplanationClient.from_run(run)

# write code to get and split your data into train and test sets here
# write code to train your model here

# explain predictions on your local machine
# "features" and "classes" fields are optional
explainer = TabularExplainer(model,
                             x_train,
                             features=feature_names,
                             classes=classes)

# explain overall model predictions (global explanation)
global_explanation = explainer.explain_global(x_test)

# uploading global model explanation data for storage or visualization in webUX
# the explanation can then be downloaded on any compute
# multiple explanations can be uploaded
client.upload_model_explanation(global_explanation, comment='global explanation: all features')
# or you can only upload the explanation object with the top k feature info
#client.upload_model_explanation(global_explanation, top_k=2, comment='global explanation: Only top 2 features')
```

3. Set up an Azure Machine Learning Compute as your compute target and submit your training run. See [Create and manage Azure Machine Learning compute clusters](#) for instructions. You might also find the [example notebooks](#) helpful.

4. Download the explanation in your local Jupyter Notebook.

```
from azureml.interpret import ExplanationClient

client = ExplanationClient.from_run(run)

# get model explanation data
explanation = client.download_model_explanation()
# or only get the top k (e.g., 4) most important features with their importance values
explanation = client.download_model_explanation(top_k=4)

global_importance_values = explanation.get_ranked_global_values()
global_importance_names = explanation.get_ranked_global_names()
print('global importance values: {}'.format(global_importance_values))
print('global importance names: {}'.format(global_importance_names))
```

Visualizations

After you download the explanations in your local Jupyter Notebook, you can use the visualizations in the explanations dashboard to understand and interpret your model. To load the explanations dashboard widget in your Jupyter Notebook, use the following code:

```
from raiwidgets import ExplanationDashboard

ExplanationDashboard(global_explanation, model, datasetX=x_test)
```

The visualizations support explanations on both engineered and raw features. Raw explanations are based on the features from the original dataset and engineered explanations are based on the features from the dataset with feature engineering applied.

When attempting to interpret a model with respect to the original dataset, it's recommended to use raw explanations as each feature importance will correspond to a column from the original dataset. One scenario where engineered explanations might be useful is when examining the impact of individual categories from a categorical feature. If a one-hot encoding is applied to a categorical feature, then the resulting engineered explanations will include a different importance value per category, one per one-hot engineered feature. This encoding can be useful when narrowing down which part of the dataset is most informative to the model.

NOTE

Engineered and raw explanations are computed sequentially. First an engineered explanation is created based on the model and featurization pipeline. Then the raw explanation is created based on that engineered explanation by aggregating the importance of engineered features that came from the same raw feature.

Create, edit, and view dataset cohorts

The top ribbon shows the overall statistics on your model and data. You can slice and dice your data into dataset cohorts, or subgroups, to investigate or compare your model's performance and explanations across these defined subgroups. By comparing your dataset statistics and explanations across those subgroups, you can get a sense of why possible errors are happening in one group versus another.

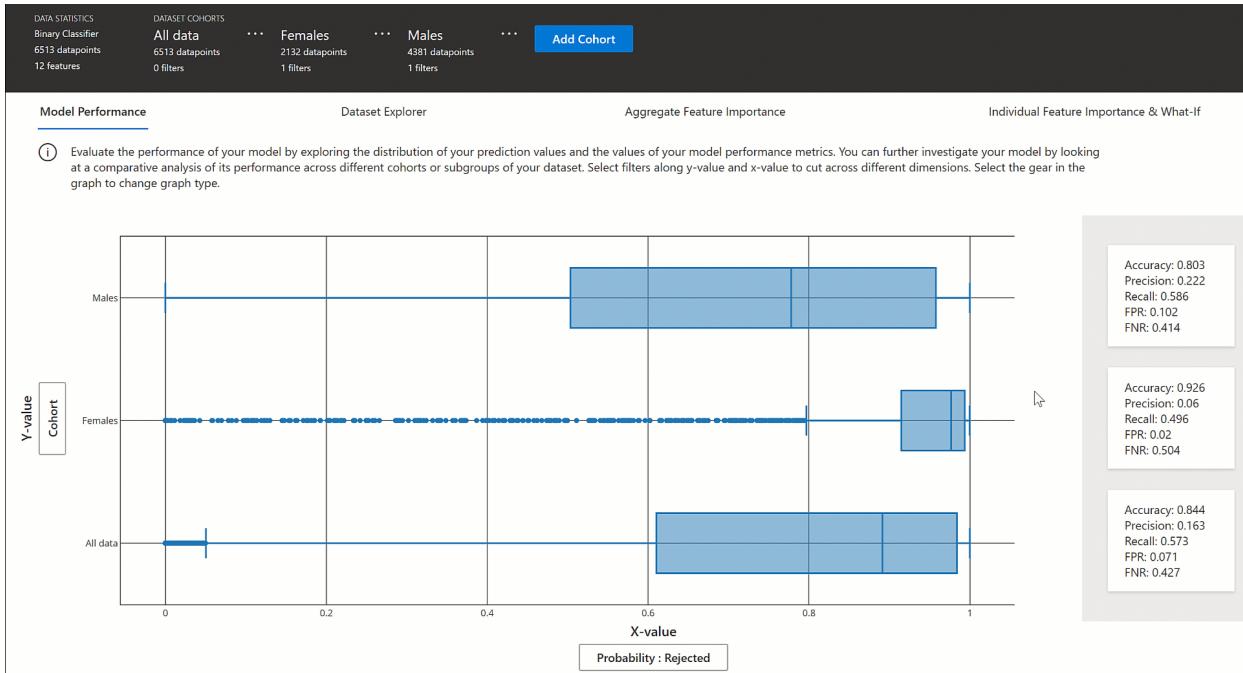


Understand entire model behavior (global explanation)

The first three tabs of the explanation dashboard provide an overall analysis of the trained model along with its predictions and explanations.

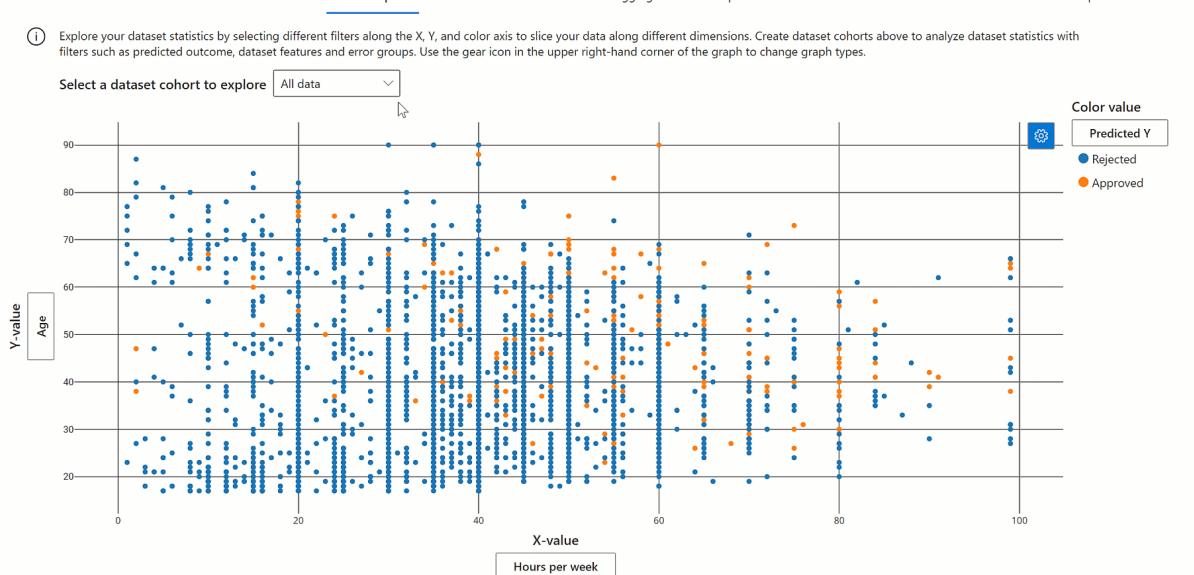
Model performance

Evaluate the performance of your model by exploring the distribution of your prediction values and the values of your model performance metrics. You can further investigate your model by looking at a comparative analysis of its performance across different cohorts or subgroups of your dataset. Select filters along y-value and x-value to cut across different dimensions. View metrics such as accuracy, precision, recall, false positive rate (FPR), and false negative rate (FNR).



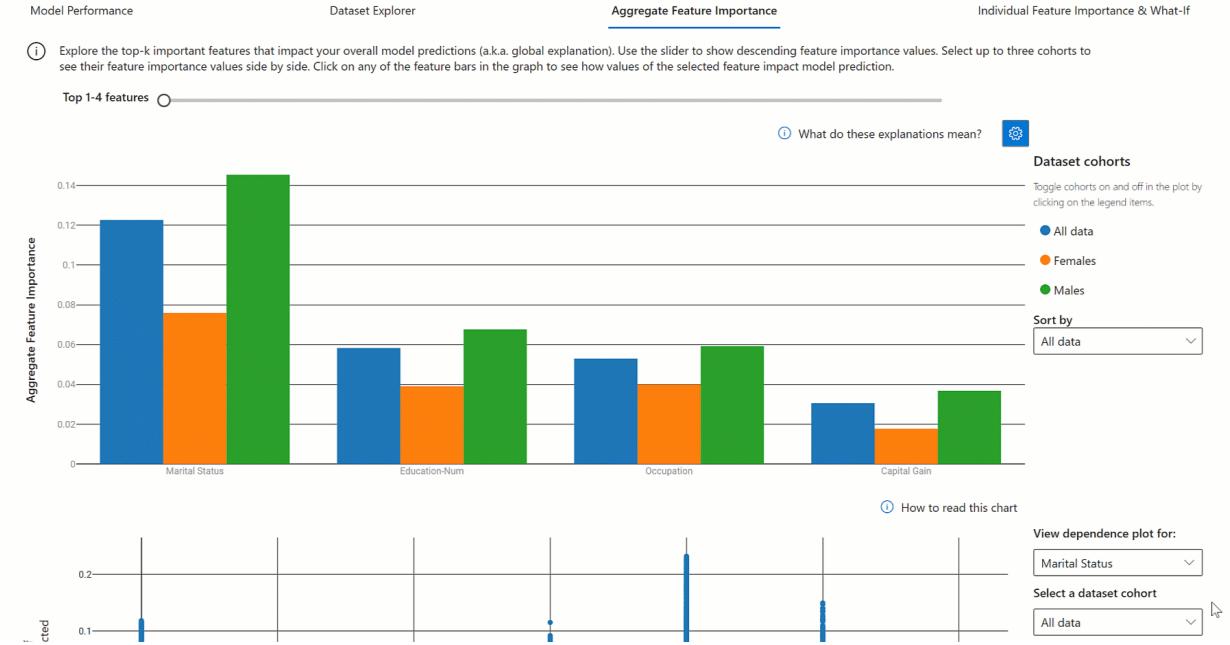
Dataset explorer

Explore your dataset statistics by selecting different filters along the X, Y, and color axes to slice your data along different dimensions. Create dataset cohorts above to analyze dataset statistics with filters such as predicted outcome, dataset features and error groups. Use the gear icon in the upper right-hand corner of the graph to change graph types.



Aggregate feature importance

Explore the top-k important features that impact your overall model predictions (also known as global explanation). Use the slider to show descending feature importance values. Select up to three cohorts to see their feature importance values side by side. Select any of the feature bars in the graph to see how values of the selected feature impact model prediction in the dependence plot below.

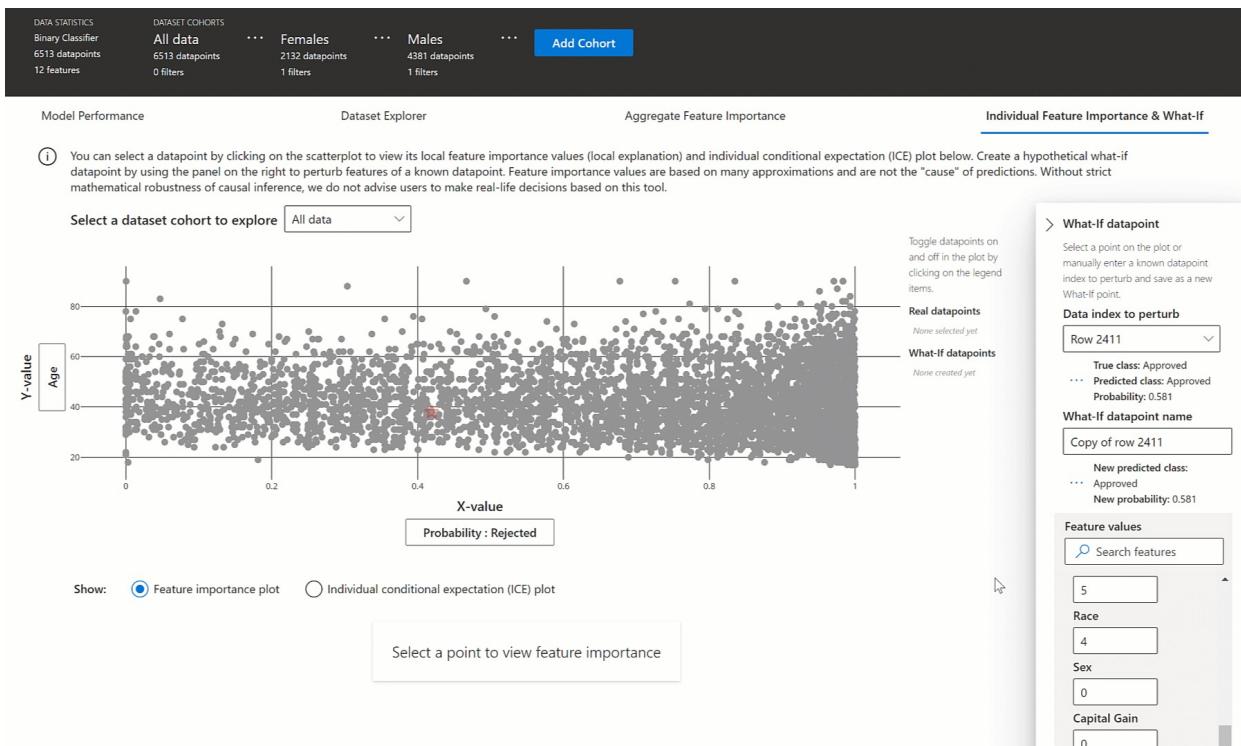


Understand individual predictions (local explanation)

The fourth tab of the explanation tab lets you drill into an individual datapoint and their individual feature importances. You can load the individual feature importance plot for any data point by clicking on any of the individual data points in the main scatter plot or selecting a specific datapoint in the panel wizard on the right.

PLOT	DESCRIPTION
------	-------------

PLOT	DESCRIPTION
Individual feature importance	Shows the top-k important features for an individual prediction. Helps illustrate the local behavior of the underlying model on a specific data point.
What-If analysis	Allows changes to feature values of the selected real data point and observe resulting changes to prediction value by generating a hypothetical datapoint with the new feature values.
Individual Conditional Expectation (ICE)	Allows feature value changes from a minimum value to a maximum value. Helps illustrate how the data point's prediction changes when a feature changes.



NOTE

These are explanations based on many approximations and are not the "cause" of predictions. Without strict mathematical robustness of causal inference, we do not advise users to make real-life decisions based on the feature perturbations of the What-If tool. This tool is primarily for understanding your model and debugging.

Visualization in Azure Machine Learning studio

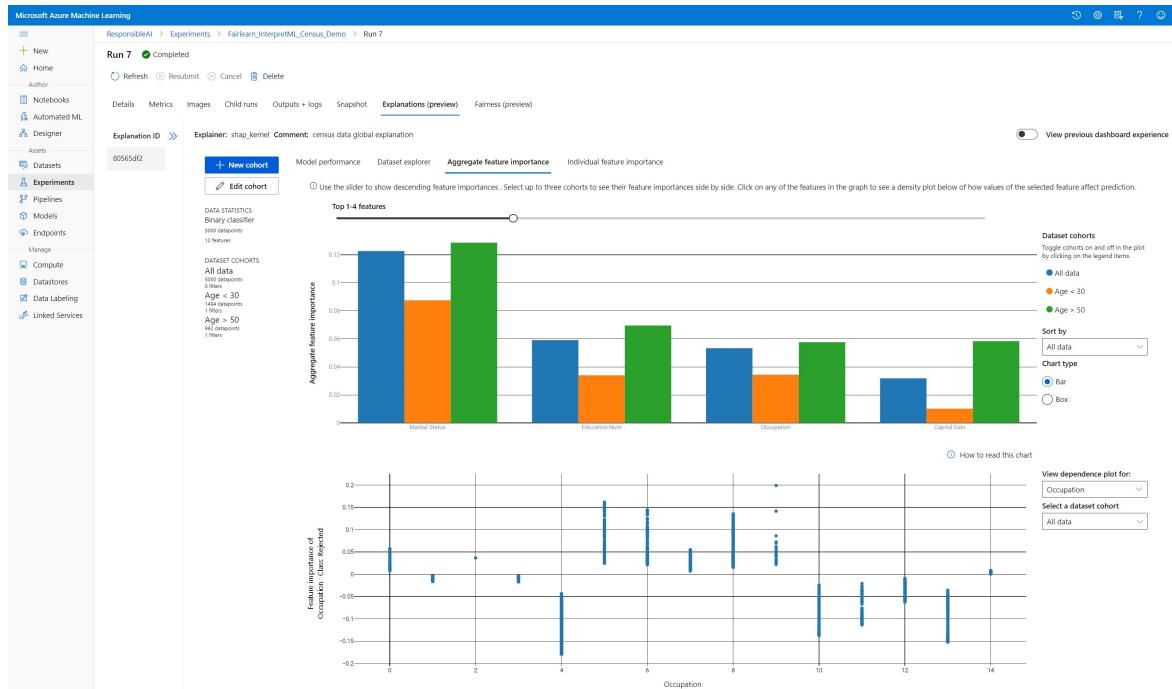
If you complete the [remote interpretability](#) steps (uploading generated explanations to Azure Machine Learning Run History), you can view the visualizations on the explanations dashboard in [Azure Machine Learning studio](#). This dashboard is a simpler version of the dashboard widget that's generated within your Jupyter Notebook. What-If datapoint generation and ICE plots are disabled as there's no active compute in Azure Machine Learning studio that can perform their real-time computations.

If the dataset, global, and local explanations are available, data populates all of the tabs. However, if only a global explanation is available, the Individual feature importance tab will be disabled.

Follow one of these paths to access the explanations dashboard in Azure Machine Learning studio:

- Experiments pane (Preview)

1. Select **Experiments** in the left pane to see a list of experiments that you've run on Azure Machine Learning.
2. Select a particular experiment to view all the runs in that experiment.
3. Select a run, and then the **Explanations** tab to the explanation visualization dashboard.



- **Models** pane

1. If you registered your original model by following the steps in [Deploy models with Azure Machine Learning](#), you can select **Models** in the left pane to view it.
2. Select a model, and then the **Explanations** tab to view the explanations dashboard.

Interpretability at inference time

You can deploy the explainer along with the original model and use it at inference time to provide the individual feature importance values (local explanation) for any new datapoint. We also offer lighter-weight scoring explainers to improve interpretability performance at inference time, which is currently supported only in Azure Machine Learning SDK. The process of deploying a lighter-weight scoring explainer is similar to deploying a model and includes the following steps:

1. Create an explanation object. For example, you can use `TabularExplainer`:

```
from interpret.ext.blackbox import TabularExplainer

explainer = TabularExplainer(model,
                             initialization_examples=x_train,
                             features=dataset_feature_names,
                             classes=dataset_classes,
                             transformations=transformations)
```

2. Create a scoring explainer with the explanation object.

```
from azureml.interpret.scoring.scoring_explainer import KernelScoringExplainer, save

# create a lightweight explainer at scoring time
scoring_explainer = KernelScoringExplainer(explainer)

# pickle scoring explainer
# pickle scoring explainer locally
OUTPUT_DIR = 'my_directory'
save(scoring_explainer, directory=OUTPUT_DIR, exist_ok=True)
```

3. Configure and register an image that uses the scoring explainer model.

```
# register explainer model using the path from ScoringExplainer.save - could be done on remote
compute
# scoring_explainer.pkl is the filename on disk, while my_scoring_explainer.pkl will be the filename
in cloud storage
run.upload_file('my_scoring_explainer.pkl', os.path.join(OUTPUT_DIR, 'scoring_explainer.pkl'))

scoring_explainer_model = run.register_model(model_name='my_scoring_explainer',
                                              model_path='my_scoring_explainer.pkl')
print(scoring_explainer_model.name, scoring_explainer_model.id, scoring_explainer_model.version, sep
      = '\t')
```

4. As an optional step, you can retrieve the scoring explainer from cloud and test the explanations.

```
from azureml.interpret.scoring.scoring_explainer import load

# retrieve the scoring explainer model from cloud"
scoring_explainer_model = Model(ws, 'my_scoring_explainer')
scoring_explainer_model_path = scoring_explainer_model.download(target_dir=os.getcwd(),
                                                               exist_ok=True)

# load scoring explainer from disk
scoring_explainer = load(scoring_explainer_model_path)

# test scoring explainer locally
preds = scoring_explainer.explain(x_test)
print(preds)
```

5. Deploy the image to a compute target, by following these steps:

- If needed, register your original prediction model by following the steps in [Deploy models with Azure Machine Learning](#).
- Create a scoring file.

```

%%writefile score.py
import json
import numpy as np
import pandas as pd
import os
import pickle
from sklearn.externals import joblib
from sklearn.linear_model import LogisticRegression
from azureml.core.model import Model

def init():

    global original_model
    global scoring_model

    # retrieve the path to the model file using the model name
    # assume original model is named original_prediction_model
    original_model_path = Model.get_model_path('original_prediction_model')
    scoring_explainer_path = Model.get_model_path('my_scoring_explainer')

    original_model = joblib.load(original_model_path)
    scoring_explainer = joblib.load(scoring_explainer_path)

def run(raw_data):
    # get predictions and explanations for each data point
    data = pd.read_json(raw_data)
    # make prediction
    predictions = original_model.predict(data)
    # retrieve model explanations
    local_importance_values = scoring_explainer.explain(data)
    # you can return any data type as long as it is JSON-serializable
    return {'predictions': predictions.tolist(), 'local_importance_values':
local_importance_values}

```

c. Define the deployment configuration.

This configuration depends on the requirements of your model. The following example defines a configuration that uses one CPU core and one GB of memory.

```

from azureml.core.webservice import AciWebservice

aciconfig = AciWebservice.deploy_configuration(cpu_cores=1,
                                                memory_gb=1,
                                                tags={"data": "NAME_OF_THE_DATASET",
                                                      "method" : "local_explanation"},
                                                description='Get local explanations for
NAME_OF_THE_PROBLEM')

```

d. Create a file with environment dependencies.

```

from azureml.core.conda_dependencies import CondaDependencies

# WARNING: to install this, g++ needs to be available on the Docker image and is not by
# default (look at the next cell)

azureml_pip_packages = ['azureml-defaults', 'azureml-core', 'azureml-telemetry', 'azureml-
interpret']

# specify CondaDependencies obj
myenv = CondaDependencies.create(conda_packages=['scikit-learn', 'pandas'],
                                 pip_packages=['sklearn-pandas'] + azureml_pip_packages,
                                 pin_sdk_version=False)

with open("myenv.yml","w") as f:
    f.write(myenv.serialize_to_string())

with open("myenv.yml","r") as f:
    print(f.read())

```

e. Create a custom dockerfile with g++ installed.

```

%%writefile dockerfile
RUN apt-get update && apt-get install -y g++

```

f. Deploy the created image.

This process takes approximately five minutes.

```

from azureml.core.webservice import Webservice
from azureml.core.image import ContainerImage

# use the custom scoring, docker, and conda files we created above
image_config = ContainerImage.image_configuration(execution_script="score.py",
                                                 docker_file="dockerfile",
                                                 runtime="python",
                                                 conda_file="myenv.yml")

# use configs and models generated above
service = Webservice.deploy_from_model(workspace=ws,
                                         name='model-scoring-service',
                                         deployment_config=aciconfig,
                                         models=[scoring_explainer_model, original_model],
                                         image_config=image_config)

service.wait_for_deployment(show_output=True)

```

6. Test the deployment.

```

import requests

# create data to test service with
examples = x_list[:4]
input_data = examples.to_json()

headers = {'Content-Type': 'application/json'}

# send request to service
resp = requests.post(service.scoring_uri, input_data, headers=headers)

print("POST to url", service.scoring_uri)
# can convert back to Python objects from json string if desired
print("prediction:", resp.text)

```

7. Clean up.

To delete a deployed web service, use `service.delete()`.

Troubleshooting

- Sparse data not supported:** The model explanation dashboard breaks/slows down substantially with a large number of features, therefore we currently don't support sparse data format. Additionally, general memory issues will arise with large datasets and large number of features.
- Supported explanations features matrix**

SUPPORTED EXPLANATION TAB	RAW FEATURES (DENSE)	RAW FEATURES (SPARSE)	ENGINEERED FEATURES (DENSE)	ENGINEERED FEATURES (SPARSE)
Model performance	Supported (not forecasting)	Supported (not forecasting)	Supported	Supported
Dataset explorer	Supported (not forecasting)	Not supported. Since sparse data isn't uploaded and UI has issues rendering sparse data.	Supported	Not supported. Since sparse data isn't uploaded and UI has issues rendering sparse data.
Aggregate feature importance	Supported	Supported	Supported	Supported
Individual feature importance	Supported (not forecasting)	Not supported. Since sparse data isn't uploaded and UI has issues rendering sparse data.	Supported	Not supported. Since sparse data isn't uploaded and UI has issues rendering sparse data.

- Forecasting models not supported with model explanations:** Interpretability, best model explanation, isn't available for AutoML forecasting experiments that recommend the following algorithms as the best model: TCNForecaster, AutoArima, Prophet, ExponentialSmoothing, Average, Naive, Seasonal Average, and Seasonal Naive. AutoML Forecasting regression models support explanations. However, in the explanation dashboard, the "Individual feature importance" tab isn't supported for forecasting because of complexity in their data pipelines.
- Local explanation for data index:** The explanation dashboard doesn't support relating local importance values to a row identifier from the original validation dataset if that dataset is greater than 5000 datapoints as the dashboard randomly downsamples the data. However, the dashboard shows raw

dataset feature values for each datapoint passed into the dashboard under the Individual feature importance tab. Users can map local importances back to the original dataset through matching the raw dataset feature values. If the validation dataset size is less than 5000 samples, the `index` feature in AzureML studio will correspond to the index in the validation dataset.

- **What-if/ICE plots not supported in studio:** What-If and Individual Conditional Expectation (ICE) plots aren't supported in Azure Machine Learning studio under the Explanations tab since the uploaded explanation needs an active compute to recalculate predictions and probabilities of perturbed features. It's currently supported in Jupyter notebooks when run as a widget using the SDK.

Next steps

[Techniques for model interpretability in Azure ML](#)

[Check out Azure Machine Learning interpretability sample notebooks](#)

Use Azure Machine Learning with the Fairlearn open-source package to assess the fairness of ML models (preview)

9/21/2022 • 7 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this how-to guide, you will learn to use the [Fairlearn](#) open-source Python package with Azure Machine Learning to perform the following tasks:

- Assess the fairness of your model predictions. To learn more about fairness in machine learning, see the [fairness in machine learning article](#).
- Upload, list and download fairness assessment insights to/from Azure Machine Learning studio.
- See a fairness assessment dashboard in Azure Machine Learning studio to interact with your model(s)' fairness insights.

NOTE

Fairness assessment is not a purely technical exercise. This package can help you assess the fairness of a machine learning model, but only you can configure and make decisions as to how the model performs. While this package helps to identify quantitative metrics to assess fairness, developers of machine learning models must also perform a qualitative analysis to evaluate the fairness of their own models.

Azure Machine Learning Fairness SDK

The Azure Machine Learning Fairness SDK, `azureml-contrib-fairness`, integrates the open-source Python package, [Fairlearn](#), within Azure Machine Learning. To learn more about Fairlearn's integration within Azure Machine Learning, check out these [sample notebooks](#). For more information on Fairlearn, see the [example guide](#) and [sample notebooks](#).

Use the following commands to install the `azureml-contrib-fairness` and `fairlearn` packages:

```
pip install azureml-contrib-fairness
pip install fairlearn==0.4.6
```

Later versions of Fairlearn should also work in the following example code.

Upload fairness insights for a single model

The following example shows how to use the fairness package. We will upload model fairness insights into Azure Machine Learning and see the fairness assessment dashboard in Azure Machine Learning studio.

1. Train a sample model in Jupyter Notebook.

For the dataset, we use the well-known adult census dataset, which we fetch from OpenML. We pretend we have a loan decision problem with the label indicating whether an individual repaid a previous loan. We will train a model to predict if previously unseen individuals will repay a loan. Such a model might be used in making loan decisions.

```

import copy
import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.datasets import fetch_openml
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import make_column_selector as selector
from sklearn.pipeline import Pipeline

from raiwidgets import FairnessDashboard

# Load the census dataset
data = fetch_openml(data_id=1590, as_frame=True)
X_raw = data.data
y = (data.target == ">50K") * 1

# (Optional) Separate the "sex" and "race" sensitive features out and drop them from the main data
prior to training your model
X_raw = data.data
y = (data.target == ">50K") * 1
A = X_raw[["race", "sex"]]
X = X_raw.drop(labels=['sex', 'race'], axis=1)

# Split the data in "train" and "test" sets
(X_train, X_test, y_train, y_test, A_train, A_test) = train_test_split(
    X_raw, y, A, test_size=0.3, random_state=12345, stratify=y
)

# Ensure indices are aligned between X, y and A,
# after all the slicing and splitting of DataFrames
# and Series
X_train = X_train.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)
y_test = y_test.reset_index(drop=True)
A_train = A_train.reset_index(drop=True)
A_test = A_test.reset_index(drop=True)

# Define a processing pipeline. This happens after the split to avoid data leakage
numeric_transformer = Pipeline(
    steps=[
        ("impute", SimpleImputer()),
        ("scaler", StandardScaler()),
    ]
)
categorical_transformer = Pipeline(
    [
        ("impute", SimpleImputer(strategy="most_frequent")),
        ("ohe", OneHotEncoder(handle_unknown="ignore")),
    ]
)
preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, selector(dtype_exclude="category")),
        ("cat", categorical_transformer, selector(dtype_include="category")),
    ]
)

# Put an estimator onto the end of the pipeline
lr_predictor = Pipeline(
    steps=[
        ("preprocessor", copy.deepcopy(preprocessor)),
        (
            "classifier",

```

```

        LogisticRegression(solver="liblinear", fit_intercept=True),
    ),
]
)

# Train the model on the test data
lr_predictor.fit(X_train, y_train)

# (Optional) View this model in the fairness dashboard, and see the disparities which appear:
from raiwidgets import FairnessDashboard
FairnessDashboard(sensitive_features=A_test,
                  y_true=y_test,
                  y_pred={"lr_model": lr_predictor.predict(X_test)})

```

2. Log into Azure Machine Learning and register your model.

The fairness dashboard can integrate with registered or unregistered models. Register your model in Azure Machine Learning with the following steps:

```

from azureml.core import Workspace, Experiment, Model
import joblib
import os

ws = Workspace.from_config()
ws.get_details()

os.makedirs('models', exist_ok=True)

# Function to register models into Azure Machine Learning
def register_model(name, model):
    print("Registering ", name)
    model_path = "models/{0}.pkl".format(name)
    joblib.dump(value=model, filename=model_path)
    registered_model = Model.register(model_path=model_path,
                                       model_name=name,
                                       workspace=ws)
    print("Registered ", registered_model.id)
    return registered_model.id

# Call the register_model function
lr_reg_id = register_model("fairness_logistic_regression", lr_predictor)

```

3. Precompute fairness metrics.

Create a dashboard dictionary using Fairlearn's `metrics` package. The `_create_group_metric_set` method has arguments similar to the Dashboard constructor, except that the sensitive features are passed as a dictionary (to ensure that names are available). We must also specify the type of prediction (binary classification in this case) when calling this method.

```

# Create a dictionary of model(s) you want to assess for fairness
sf = { 'Race': A_test.race, 'Sex': A_test.sex}
ys_pred = { lr_reg_id:lr_predictor.predict(X_test) }
from fairlearn.metrics._group_metric_set import _create_group_metric_set

dash_dict = _create_group_metric_set(y_true=y_test,
                                      predictions=ys_pred,
                                      sensitive_features=sf,
                                      prediction_type='binary_classification')

```

4. Upload the precomputed fairness metrics.

Now, import `azureml.contrib.fairness` package to perform the upload:

```
from azureml.contrib.fairness import upload_dashboard_dictionary, download_dashboard_by_upload_id
```

Create an Experiment, then a Run, and upload the dashboard to it:

```
exp = Experiment(ws, "Test_Fairness_Census_Demo")
print(exp)

run = exp.start_logging()

# Upload the dashboard to Azure Machine Learning
try:
    dashboard_title = "Fairness insights of Logistic Regression Classifier"
    # Set validate_model_ids parameter of upload_dashboard_dictionary to False if you have not
    registered your model(s)
    upload_id = upload_dashboard_dictionary(run,
                                              dash_dict,
                                              dashboard_name=dashboard_title)
    print("\nUploaded to id: {0}\n".format(upload_id))

    # To test the dashboard, you can download it back and ensure it contains the right information
    downloaded_dict = download_dashboard_by_upload_id(run, upload_id)
finally:
    run.complete()
```

5. Check the fairness dashboard from Azure Machine Learning studio

If you complete the previous steps (uploading generated fairness insights to Azure Machine Learning), you can view the fairness dashboard in [Azure Machine Learning studio](#). This dashboard is the same visualization dashboard provided in Fairlearn, enabling you to analyze the disparities among your sensitive feature's subgroups (e.g., male vs. female). Follow one of these paths to access the visualization dashboard in Azure Machine Learning studio:

- **Jobs pane (Preview)**
 - a. Select **Jobs** in the left pane to see a list of experiments that you've run on Azure Machine Learning.
 - b. Select a particular experiment to view all the runs in that experiment.
 - c. Select a run, and then the **Fairness** tab to the explanation visualization dashboard.
 - d. Once landing on the **Fairness** tab, click on a **fairness id** from the menu on the right.
 - e. Configure your dashboard by selecting your sensitive attribute, performance metric, and fairness metric of interest to land on the fairness assessment page.
 - f. Switch chart type from one to another to observe both **allocation** harms and **quality of service** harms.

Microsoft Azure Machine Learning

ResponsibleAI > Experiments > Test_Fairlearn_GridSearch_Census_Demo > Run 1

Run 1 Completed

Refresh ↻ Resubmit 🕒 Cancel ✖ Delete

Fairness (preview)

Sensitive feature: race | Performance metric: Accuracy | Fairness metric: Demographic parity difference

	Accuracy	Selection rate	Demographic parity difference	False positive rate	False negative rate
Overall	84.7%	9.9%	17.6%	7.3%	40.1%
Amer-Indian-Eskimo	92.4%	10.3%	2.4%	40%	
Asian-Pac-Islander	81.4%	24.8%	11.6%	38.3%	
Black	91.2%	8.7%	3.3%	50%	
Other	86.5%	7.2%	4.1%	73.3%	
White	84%	21.2%	7.8%	39.7%	

Charts

Selection rate

How to read this chart

Demographic Group	Selection rate (%)
Amer-Indian-Eskimo	25%
Asian-Pac-Islander	23%
Black	8.8%
Other	7.2%
White	2.1%

Microsoft Azure Machine Learning

ResponsibleAI > Experiments > Test_Fairlearn_GridSearch_Census_Demo > Run 1

Run 1 Completed

Refresh ↻ Resubmit 🕒 Cancel ✖ Delete

Fairness (preview)

Sensitive feature: race | Performance metric: Accuracy | Fairness metric: Demographic parity difference

	Accuracy	Selection rate	Demographic parity difference	False positive rate	False negative rate
Overall	84.7%	9.9%	17.6%	7.3%	40.1%
Amer-Indian-Eskimo	92.4%	10.3%	2.4%	40%	
Asian-Pac-Islander	81.4%	24.8%	11.6%	38.3%	
Black	91.2%	8.7%	3.3%	50%	
Other	86.5%	7.2%	4.1%	73.3%	
White	84%	21.2%	7.8%	39.7%	

Charts

False positive and false negative rates

How to read this chart

Demographic Group	False negative rate (%)	False positive rate (%)
Amer-Indian-Eskimo	5.4	0.04
Asian-Pac-Islander	3.3	0.11
Black	0.3	0.34
Other	0.73	0.06
White	0.4	0.079

- **Models pane**

- If you registered your original model by following the previous steps, you can select **Models** in the left pane to view it.
- Select a model, and then the **Fairness** tab to view the explanation visualization dashboard.

To learn more about the visualization dashboard and what it contains, check out Fairlearn's [user guide](#).

Upload fairness insights for multiple models

To compare multiple models and see how their fairness assessments differ, you can pass more than one model to the visualization dashboard and compare their performance-fairness trade-offs.

1. Train your models:

We now create a second classifier, based on a Support Vector Machine estimator, and upload a fairness dashboard dictionary using Fairlearn's `metrics` package. We assume that the previously trained model is still available.

```

# Put an SVM predictor onto the preprocessing pipeline
from sklearn import svm
svm_predictor = Pipeline(
    steps=[
        ("preprocessor", copy.deepcopy(preprocessor)),
        (
            "classifier",
            svm.SVC(),
        ),
    ],
)

# Train your second classification model
svm_predictor.fit(X_train, y_train)

```

2. Register your models

Next register both models within Azure Machine Learning. For convenience, store the results in a dictionary, which maps the `id` of the registered model (a string in `name:version` format) to the predictor itself:

```

model_dict = {}

lr_reg_id = register_model("fairness_logistic_regression", lr_predictor)
model_dict[lr_reg_id] = lr_predictor

svm_reg_id = register_model("fairness_svm", svm_predictor)
model_dict[svm_reg_id] = svm_predictor

```

3. Load the Fairness dashboard locally

Before uploading the fairness insights into Azure Machine Learning, you can examine these predictions in a locally invoked Fairness dashboard.

```

# Generate models' predictions and load the fairness dashboard locally
ys_pred = {}
for n, p in model_dict.items():
    ys_pred[n] = p.predict(X_test)

from raiwidgets import FairnessDashboard

FairnessDashboard(sensitive_features=A_test,
                  y_true=y_test.tolist(),
                  y_pred=ys_pred)

```

4. Precompute fairness metrics.

Create a dashboard dictionary using Fairlearn's `metrics` package.

```

sf = { 'Race': A_test.race, 'Sex': A_test.sex }

from fairlearn.metrics._group_metric_set import _create_group_metric_set

dash_dict = _create_group_metric_set(y_true=Y_test,
                                      predictions=ys_pred,
                                      sensitive_features=sf,
                                      prediction_type='binary_classification')

```

5. Upload the precomputed fairness metrics.

Now, import `azureml.contrib.fairness` package to perform the upload:

```
from azureml.contrib.fairness import upload_dashboard_dictionary, download_dashboard_by_upload_id
```

Create an Experiment, then a Run, and upload the dashboard to it:

```
exp = Experiment(ws, "Compare_Two_Models_Fairness_Census_Demo")
print(exp)

run = exp.start_logging()

# Upload the dashboard to Azure Machine Learning
try:
    dashboard_title = "Fairness Assessment of Logistic Regression and SVM Classifiers"
    # Set validate_model_ids parameter of upload_dashboard_dictionary to False if you have not
    registered your model(s)
    upload_id = upload_dashboard_dictionary(run,
                                              dash_dict,
                                              dashboard_name=dashboard_title)
    print("\nUploaded to id: {0}\n".format(upload_id))

    # To test the dashboard, you can download it back and ensure it contains the right information
    downloaded_dict = download_dashboard_by_upload_id(run, upload_id)
finally:
    run.complete()
```

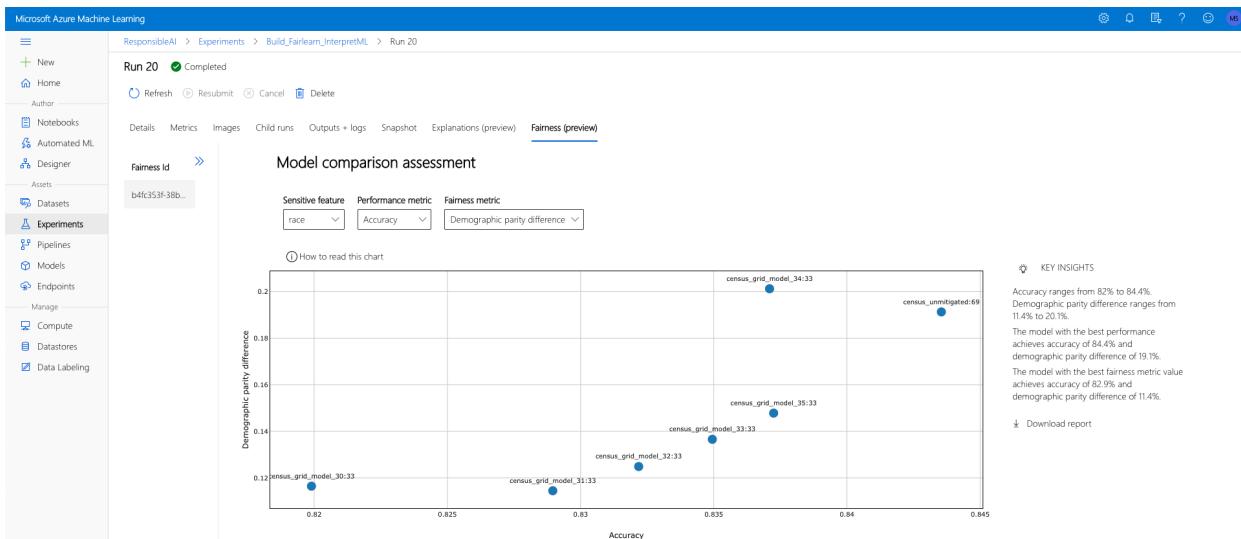
Similar to the previous section, you can follow one of the paths described above (via **Experiments** or **Models**) in Azure Machine Learning studio to access the visualization dashboard and compare the two models in terms of fairness and performance.

Upload unmitigated and mitigated fairness insights

You can use Fairlearn's [mitigation algorithms](#), compare their generated mitigated model(s) to the original unmitigated model, and navigate the performance/fairness trade-offs among compared models.

To see an example that demonstrates the use of the [Grid Search](#) mitigation algorithm (which creates a collection of mitigated models with different fairness and performance trade offs) check out this [sample notebook](#).

Uploading multiple models' fairness insights in a single Run allows for comparison of models with respect to fairness and performance. You can click on any of the models displayed in the model comparison chart to see the detailed fairness insights of the particular model.



Next steps

[Learn more about model fairness](#)

[Check out Azure Machine Learning Fairness sample notebooks](#)

Prepare data for computer vision tasks with automated machine learning v1

9/21/2022 • 3 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

IMPORTANT

Support for training computer vision models with automated ML in Azure Machine Learning is an experimental public preview feature. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this article, you learn how to prepare image data for training computer vision models with [automated machine learning in Azure Machine Learning](#).

To generate models for computer vision tasks with automated machine learning, you need to bring labeled image data as input for model training in the form of an [Azure Machine Learning TabularDataset](#).

To ensure your TabularDataset contains the accepted schema for consumption in automated ML, you can use the Azure Machine Learning data labeling tool or use a conversion script.

Prerequisites

- Familiarize yourself with the accepted [schemas for JSONL files for AutoML computer vision experiments](#).
- Labeled data you want to use to train computer vision models with automated ML.

Azure Machine Learning data labeling

If you don't have labeled data, you can use Azure Machine Learning's [data labeling tool](#) to manually label images. This tool automatically generates the data required for training in the accepted format.

It helps to create, manage, and monitor data labeling tasks for

- Image classification (multi-class and multi-label)
- Object detection (bounding box)
- Instance segmentation (polygon)

If you already have a data labeling project and you want to use that data, you can [export your labeled data as an Azure Machine Learning TabularDataset](#), which can then be used directly with automated ML for training computer vision models.

Use conversion scripts

If you have labeled data in popular computer vision data formats, like VOC or COCO, [helper scripts](#) to generate JSONL files for training and validation data are available in [notebook examples](#).

If your data doesn't follow any of the previously mentioned formats, you can use your own script to generate JSON Lines files based on schemas defined in [Schema for JSONL files for AutoML image experiments](#).

After your data file(s) are converted to the accepted JSONL format, you can upload them to your storage account on Azure.

Upload the JSONL file and images to storage

To use the data for automated ML training, upload the data to your [Azure Machine Learning workspace](#) via a [datastore](#). The datastore provides a mechanism for you to upload/download data to storage on Azure, and interact with it from your remote compute targets.

Upload the entire parent directory consisting of images and JSONL files to the default datastore that is automatically created upon workspace creation. This datastore connects to the default Azure blob storage container that was created as part of workspace creation.

```
# Retrieve default datastore that's automatically created when we setup a workspace
ds = ws.get_default_datastore()
ds.upload(src_dir='./fridgeObjects', target_path='fridgeObjects')
```

Once the data upload is done, you can create an [Azure Machine Learning TabularDataset](#) and register it to your workspace for future use as input to your automated ML experiments for computer vision models.

```
from azureml.core import Dataset
from azureml.data import DataType

training_dataset_name = 'fridgeObjectsTrainingDataset'
# create training dataset
training_dataset =
    Dataset.Tabular.from_json_lines_files(path=ds.path("fridgeObjects/train_annotations.jsonl"),
                                            set_column_types={"image_url":
                                                DataType.to_stream(ds.workspace)})
    )
training_dataset = training_dataset.register( workspace=ws, name=training_dataset_name)

print("Training dataset name: " + training_dataset.name)
```

Next steps

- [Train computer vision models with automated machine learning](#).
- [Train a small object detection model with automated machine learning](#).
- [Tutorial: Train an object detection model \(preview\) with AutoML and Python](#).

Reinforcement learning (preview) with Azure Machine Learning

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

WARNING

Azure Machine Learning reinforcement learning via the `azureml.contrib.train.rl` package will no longer be supported after June 2022. We recommend customers use the [Ray on Azure Machine Learning library](#) for reinforcement learning experiments with Azure Machine Learning. For an example, see the notebook [Reinforcement Learning in Azure Machine Learning - Pong problem](#).

In this article, you learn how to train a reinforcement learning (RL) agent to play the video game Pong. You use the open-source Python library [Ray RLlib](#) with Azure Machine Learning to manage the complexity of distributed RL.

In this article you learn how to:

- Set up an experiment
- Define head and worker nodes
- Create an RL estimator
- Submit an experiment to start a job
- View results

This article is based on the [RLlib Pong example](#) that can be found in the Azure Machine Learning notebook [GitHub repository](#).

Prerequisites

Run this code in either of these environments. We recommend you try Azure Machine Learning compute instance for the fastest start-up experience. You can quickly clone and run the reinforcement sample notebooks on an Azure Machine Learning compute instance.

- Azure Machine Learning compute instance
 - Learn how to clone sample notebooks in [Tutorial: Train and deploy a model](#).
 - Clone the **how-to-use-azureml** folder instead of **tutorials**
 - Run the virtual network setup notebook located at `/how-to-use-azureml/reinforcement-learning/setup/devenv_setup.ipynb` to open network ports used for distributed reinforcement learning.
 - Run the sample notebook `/how-to-use-azureml/reinforcement-learning/atari-on-distributed-compute/pong_rlrb.ipynb`
- Your own Jupyter Notebook server
 - Install the [Azure Machine Learning SDK](#).
 - Install the [Azure Machine Learning RL SDK](#):
`pip install --upgrade azureml-contrib-reinforcementlearning`
 - Create a [workspace configuration file](#).

- Run the virtual network to open network ports used for distributed reinforcement learning.

How to train a Pong-playing agent

Reinforcement learning (RL) is an approach to machine learning that learns by doing. While other machine learning techniques learn by passively taking input data and finding patterns within it, RL uses **training agents** to actively make decisions and learn from their outcomes.

Your training agents learn to play Pong in a **simulated environment**. Training agents make a decision every frame of the game to move the paddle up, down, or stay in place. It looks at the state of the game (an RGB image of the screen) to make a decision.

RL uses **rewards** to tell the agent if its decisions are successful. In this example, the agent gets a positive reward when it scores a point and a negative reward when a point is scored against it. Over many iterations, the training agent learns to choose the action, based on its current state, that optimizes for the sum of expected future rewards. It's common to use **deep neural networks** (DNN) to perform this optimization in RL.

Training ends when the agent reaches an average reward score of 18 in a training epoch. This means that the agent has beaten its opponent by an average of at least 18 points in matches up to 21.

The process of iterating through simulation and retraining a DNN is computationally expensive, and requires a lot of data. One way to improve performance of RL jobs is by **parallelizing work** so that multiple training agents can act and learn simultaneously. However, managing a distributed RL environment can be a complex undertaking.

Azure Machine Learning provides the framework to manage these complexities to scale out your RL workloads.

Set up the environment

Set up the local RL environment by:

1. Loading the required Python packages
2. Initializing your workspace
3. Creating an experiment
4. Specifying a configured virtual network.

Import libraries

Import the necessary Python packages to run the rest of this example.

```
# Azure ML Core imports
import azureml.core
from azureml.core import Workspace
from azureml.core import Experiment
from azureml.core.compute import AmlCompute
from azureml.core.compute import ComputeTarget
from azureml.core.runconfig import EnvironmentDefinition
from azureml.widgets import RunDetails
from azureml.tensorboard import Tensorboard

# Azure ML Reinforcement Learning imports
from azureml.contrib.train.rl import ReinforcementLearningEstimator, Ray
from azureml.contrib.train.rl import WorkerConfiguration
```

Initialize a workspace

Initialize a **workspace** object from the `config.json` file created in the [prerequisites section](#). If you are executing this code in an Azure Machine Learning Compute Instance, the configuration file has already been created for you.

```
ws = Workspace.from_config()
```

Create a reinforcement learning experiment

Create an [experiment](#) to track your reinforcement learning job. In Azure Machine Learning, experiments are logical collections of related trials to organize job logs, history, outputs, and more.

```
experiment_name='rllib-pong-multi-node'  
  
exp = Experiment(workspace=ws, name=experiment_name)
```

Specify a virtual network

For RL jobs that use multiple compute targets, you must specify a virtual network with open ports that allow worker nodes and head nodes to communicate with each other.

The virtual network can be in any resource group, but it should be in the same region as your workspace. For more information on setting up your virtual network, see the workspace setup notebook in the prerequisites section. Here, you specify the name of the virtual network in your resource group.

```
vnet = 'your_vnet'
```

Define head and worker compute targets

This example uses separate compute targets for the Ray head and workers nodes. These settings let you scale your compute resources up and down depending on your workload. Set the number of nodes, and the size of each node, based on your needs.

Head computing target

You can use a GPU-equipped head cluster to improve deep learning performance. The head node trains the neural network that the agent uses to make decisions. The head node also collects data points from the worker nodes to train the neural network.

The head compute uses a single [STANDARD_NC6](#) virtual machine (VM). It has 6 virtual CPUs to distribute work across.

```

from azureml.core.compute import AmlCompute, ComputeTarget

# choose a name for the Ray head cluster
head_compute_name = 'head-gpu'
head_compute_min_nodes = 0
head_compute_max_nodes = 2

# This example uses GPU VM. For using CPU VM, set SKU to STANDARD_D2_V2
head_vm_size = 'STANDARD_NC6'

if head_compute_name in ws.compute_targets:
    head_compute_target = ws.compute_targets[head_compute_name]
    if head_compute_target and type(head_compute_target) is AmlCompute:
        print(f'found head compute target. just use it {head_compute_name}')
else:
    print('creating a new head compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size = head_vm_size,
                                                               min_nodes = head_compute_min_nodes,
                                                               max_nodes = head_compute_max_nodes,
                                                               vnet_resourcegroup_name = ws.resource_group,
                                                               vnet_name = vnet_name,
                                                               subnet_name = 'default')

# create the cluster
head_compute_target = ComputeTarget.create(ws, head_compute_name, provisioning_config)

# can poll for a minimum number of nodes and for a specific timeout.
# if no min node count is provided it will use the scale settings for the cluster
head_compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

# For a more detailed view of current AmlCompute status, use get_status()
print(head_compute_target.get_status().serialize())

```

NOTE

You may choose to use [low-priority VMs](#) to run some or all of your workloads. See how to [create a low-priority VM](#).

Worker computing cluster

This example uses four [STANDARD_D2_V2](#) VMs for the worker compute target. Each worker node has 2 available CPUs for a total of 8 available CPUs.

Gpus aren't necessary for the worker nodes since they aren't performing deep learning. The workers run the game simulations and collect data.

```

# choose a name for your Ray worker cluster
worker_compute_name = 'worker-cpu'
worker_compute_min_nodes = 0
worker_compute_max_nodes = 4

# This example uses CPU VM. For using GPU VM, set SKU to STANDARD_NC6
worker_vm_size = 'STANDARD_D2_V2'

# Create the compute target if it hasn't been created already
if worker_compute_name in ws.compute_targets:
    worker_compute_target = ws.compute_targets[worker_compute_name]
    if worker_compute_target and type(worker_compute_target) is AmlCompute:
        print(f'found worker compute target. just use it {worker_compute_name}')
else:
    print('creating a new worker compute target...')
provisioning_config = AmlCompute.provisioning_configuration(vm_size = worker_vm_size,
                                                            min_nodes = worker_compute_min_nodes,
                                                            max_nodes = worker_compute_max_nodes,
                                                            vnet_resourcegroup_name = ws.resource_group,
                                                            vnet_name = vnet_name,
                                                            subnet_name = 'default')

# create the cluster
worker_compute_target = ComputeTarget.create(ws, worker_compute_name, provisioning_config)

# can poll for a minimum number of nodes and for a specific timeout.
# if no min node count is provided it will use the scale settings for the cluster
worker_compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

# For a more detailed view of current AmlCompute status, use get_status()
print(worker_compute_target.get_status().serialize())

```

Create a reinforcement learning estimator

Use the [ReinforcementLearningEstimator](#) to submit a training job to Azure Machine Learning.

Azure Machine Learning uses estimator classes to encapsulate job configuration information. This lets you specify how to configure a script execution.

Define a worker configuration

The WorkerConfiguration object tells Azure Machine Learning how to initialize the worker cluster that runs the entry script.

```

# Pip packages we will use for both head and worker
pip_packages=["ray[rllib]==0.8.3"] # Latest version of Ray has fixes for issues related to object transfers

# Specify the Ray worker configuration
worker_conf = WorkerConfiguration(
    # Azure ML compute cluster to run Ray workers
    compute_target=worker_compute_target,

    # Number of worker nodes
    node_count=4,

    # GPU
    use_gpu=False,

    # PIP packages to use
    pip_packages=pip_packages
)

```

Define script parameters

The entry script `pong_rllib.py` accepts a list of parameters that defines how to execute the training job. Passing these parameters through the estimator as a layer of encapsulation makes it easy to change script parameters and run configurations independently of each other.

Specifying the correct `num_workers` makes the most out of your parallelization efforts. Set the number of workers to the same as the number of available CPUs. For this example, you can use the following calculation:

The head node is a [Standard_NC6](#) with 6 vCPUs. The worker cluster is 4 [Standard_D2_V2 VMs](#) with 2 CPUs each, for a total of 8 CPUs. However, you must subtract 1 CPU from the worker count since 1 must be dedicated to the head node role.

6 CPUs + 8 CPUs - 1 head CPU = 13 simultaneous workers. Azure Machine Learning uses head and worker clusters to distinguish compute resources. However, Ray does not distinguish between head and workers, and all CPUs are available as worker threads.

```
training_algorithm = "IMPALA"
rl_environment = "PongNoFrameskip-v4"

# Training script parameters
script_params = {

    # Training algorithm, IMPALA in this case
    "--run": training_algorithm,

    # Environment, Pong in this case
    "--env": rl_environment,

    # Add additional single quotes at the both ends of string values as we have spaces in the
    # string parameters, outermost quotes are not passed to scripts as they are not actually part of string
    # Number of GPUs
    # Number of ray workers
    "--config": '\'{\"num_gpus\": 1, \"num_workers\": 13}\'',

    # Target episode reward mean to stop the training
    # Total training time in seconds
    "--stop": '\'{\"episode_reward_mean\": 18, \"time_total_s\": 3600}\'',

}
```

Define the reinforcement learning estimator

Use the parameter list and the worker configuration object to construct the estimator.

```

# RL estimator
rl_estimator = ReinforcementLearningEstimator(
    # Location of source files
    source_directory='files',
    # Python script file
    entry_script="pong_rllib.py",
    # Parameters to pass to the script file
    # Defined above.
    script_params=script_params,
    # The Azure ML compute target set up for Ray head nodes
    compute_target=head_compute_target,
    # Pip packages
    pip_packages=pip_packages,
    # GPU usage
    use_gpu=True,
    # RL framework. Currently must be Ray.
    rl_framework=Ray(),
    # Ray worker configuration defined above.
    worker_configuration=worker_conf,
    # How long to wait for whole cluster to start
    cluster_coordination_timeout_seconds=3600,
    # Maximum time for the whole Ray job to run
    # This will cut off the job after an hour
    max_run_duration_seconds=3600,
    # Allow the docker container Ray runs in to make full use
    # of the shared memory available from the host OS.
    shm_size=24*1024*1024*1024
)

```

Entry script

The [entry script](#) `pong_rllib.py` trains a neural network using the [OpenAI Gym environment](#) `PongNoFrameSkip-v4`. OpenAI Gyms are standardized interfaces to test reinforcement learning algorithms on classic Atari games.

This example uses a training algorithm known as [IMPALA](#) (Importance Weighted Actor-Learner Architecture). IMPALA parallelizes each individual learning actor to scale across many compute nodes without sacrificing speed or stability.

[Ray Tune](#) orchestrates the IMPALA worker tasks.

```

import ray
import ray.tune as tune
from ray.rllib import train

import os
import sys

from azureml.core import Run
from utils import callbacks

DEFAULT_RAY_ADDRESS = 'localhost:6379'

if __name__ == "__main__":

    # Parse arguments
    train_parser = train.create_parser()

    args = train_parser.parse_args()
    print("Algorithm config:", args.config)

    if args.ray_address is None:
        args.ray_address = DEFAULT_RAY_ADDRESS

    ray.init(address=args.ray_address)

    tune.run(run_or_experiment=args.run,
             config={
                 "env": args.env,
                 "num_gpus": args.config["num_gpus"],
                 "num_workers": args.config["num_workers"],
                 "callbacks": {"on_train_result": callbacks.on_train_result},
                 "sample_batch_size": 50,
                 "train_batch_size": 1000,
                 "num_sgd_iter": 2,
                 "num_data_loader_buffers": 2,
                 "model": {
                     "dim": 42
                 },
                 "stop":args.stop,
                 "local_dir='./logs'}
             )

```

Logging callback function

The entry script uses a utility function to define a [custom RLLib callback function](#) to log metrics to your Azure Machine Learning workspace. Learn how to view these metrics in the [Monitor and view results](#) section.

```

'''RLLib callbacks module:
Common callback methods to be passed to RLLib trainer.
'''

from azureml.core import Run

def on_train_result(info):
    '''Callback on train result to record metrics returned by trainer.
    '''
    run = Run.get_context()
    run.log(
        name='episode_reward_mean',
        value=info["result"]["episode_reward_mean"])
    run.log(
        name='episodes_total',
        value=info["result"]["episodes_total"])

```

Submit a job

Run handles the run history of in-progress or complete jobs.

```
run = exp.submit(config=rl_estimator)
```

NOTE

The run may take up to 30 to 45 minutes to complete.

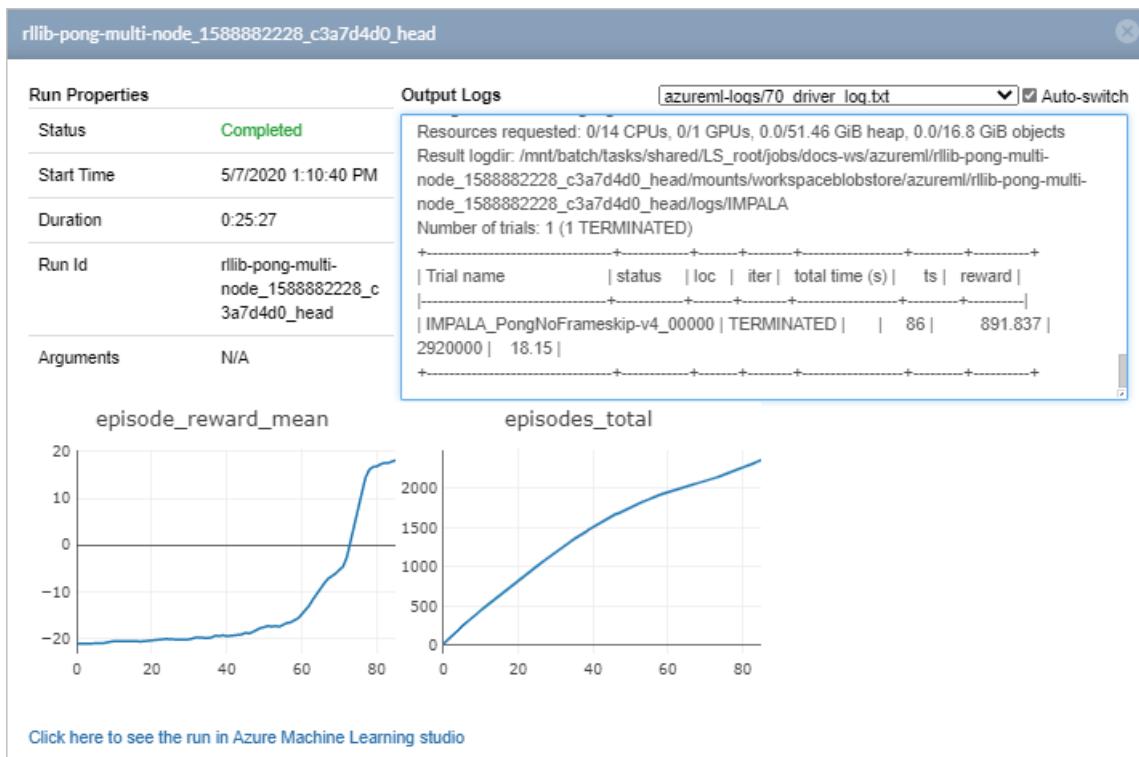
Monitor and view results

Use the Azure Machine Learning Jupyter widget to see the status of your jobs in real time. The widget shows two child jobs: one for head and one for workers.

```
from azureml.widgets import RunDetails  
  
RunDetails(run).show()  
run.wait_for_completion()
```

1. Wait for the widget to load.
2. Select the head job in the list of jobs.

Select [Click here to see the job in Azure Machine Learning studio](#) for additional job information in the studio. You can access this information while the job is in progress or after it completes.



The **episode_reward_mean** plot shows the mean number of points scored per training epoch. You can see that the training agent initially performed poorly, losing its matches without scoring a single point (shown by a reward_mean of -21). Within 100 iterations, the training agent learned to beat the computer opponent by an average of 18 points.

If you browse logs of the child job, you can see the evaluation results recorded in driver_log.txt file. You may need to wait several minutes before these metrics become available on the Job page.

In short work, you have learned to configure multiple compute resources to train a reinforcement learning agent to play Pong very well against a computer opponent.

Next steps

In this article, you learned how to train a reinforcement learning agent using an IMPALA learning agent. To see additional examples, go to the [Azure Machine Learning Reinforcement Learning GitHub repository](#).

Deploy machine learning models to Azure

9/21/2022 • 17 minutes to read • [Edit Online](#)

APPLIES TO: [Azure CLI ml extension v1](#) [Python SDK azureml v1](#)

Learn how to deploy your machine learning or deep learning model as a web service in the Azure cloud.

NOTE

Azure Machine Learning Endpoints (preview) provide an improved, simpler deployment experience. Endpoints support both real-time and batch inference scenarios. Endpoints provide a unified interface to invoke and manage model deployments across compute types. See [What are Azure Machine Learning endpoints \(preview\)?](#).

Workflow for deploying a model

The workflow is similar no matter where you deploy your model:

1. Register the model.
2. Prepare an entry script.
3. Prepare an inference configuration.
4. Deploy the model locally to ensure everything works.
5. Choose a compute target.
6. Deploy the model to the cloud.
7. Test the resulting web service.

For more information on the concepts involved in the machine learning deployment workflow, see [Manage, deploy, and monitor models with Azure Machine Learning](#).

Prerequisites

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO: [Azure CLI ml extension v1](#)

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- An Azure Machine Learning workspace. For more information, see [Create workspace resources](#).
- A model. The examples in this article use a pre-trained model.
- A machine that can run Docker, such as a [compute instance](#).

Connect to your workspace

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO: [Azure CLI ml extension v1](#)

To see the workspaces that you have access to, use the following commands:

```
az login  
az account set -s <subscription>  
az ml workspace list --resource-group=<resource-group>
```

Register the model

A typical situation for a deployed machine learning service is that you need the following components:

- Resources representing the specific model that you want deployed (for example: a pytorch model file).
- Code that you will be running in the service, that executes the model on a given input.

Azure Machine Learnings allows you to separate the deployment into two separate components, so that you can keep the same code, but merely update the model. We define the mechanism by which you upload a model *separately* from your code as "registering the model".

When you register a model, we upload the model to the cloud (in your workspace's default storage account) and then mount it to the same compute where your webservice is running.

The following examples demonstrate how to register a model.

IMPORTANT

You should use only models that you create or obtain from a trusted source. You should treat serialized models as code, because security vulnerabilities have been discovered in a number of popular formats. Also, models might be intentionally trained with malicious intent to provide biased or inaccurate output.

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO: [Azure CLI ml extension v1](#)

The following commands download a model and then register it with your Azure Machine Learning workspace:

```
wget https://aka.ms/bidaf-9-model -O model.onnx --show-progress  
az ml model register -n bidaf_onnx \  
-p ./model.onnx \  
-g <resource-group> \  
-w <workspace-name>
```

Set `-p` to the path of a folder or a file that you want to register.

For more information on `az ml model register`, see the [reference documentation](#).

Register a model from an Azure ML training job

If you need to register a model that was created previously through an Azure Machine Learning training job, you can specify the experiment, run, and path to the model:

```
az ml model register -n bidaf_onnx --asset-path outputs/model.onnx --experiment-name myexperiment --run-id myrunid --tag area=qna
```

The `--asset-path` parameter refers to the cloud location of the model. In this example, the path of a single file is used. To include multiple files in the model registration, set `--asset-path` to the path of a folder that contains the files.

For more information on `az ml model register`, see the [reference documentation](#).

Define a dummy entry script

The entry script receives data submitted to a deployed web service and passes it to the model. It then returns the model's response to the client. *The script is specific to your model.* The entry script must understand the data that the model expects and returns.

The two things you need to accomplish in your entry script are:

1. Loading your model (using a function called `init()`)
2. Running your model on input data (using a function called `run()`)

For your initial deployment, use a dummy entry script that prints the data it receives.

```
import json

def init():
    print("This is init")

def run(data):
    test = json.loads(data)
    print(f"received data {test}")
    return f"test is {test}"
```

Save this file as `echo_score.py` inside of a directory called `source_dir`. This dummy script returns the data you send to it, so it doesn't use the model. But it is useful for testing that the scoring script is running.

Define an inference configuration

An inference configuration describes the Docker container and files to use when initializing your web service. All of the files within your source directory, including subdirectories, will be zipped up and uploaded to the cloud when you deploy your web service.

The inference configuration below specifies that the machine learning deployment will use the file `echo_score.py` in the `./source_dir` directory to process incoming requests and that it will use the Docker image with the Python packages specified in the `project_environment` environment.

You can use any [Azure Machine Learning inference curated environments](#) as the base Docker image when creating your project environment. We will install the required dependencies on top and store the resulting Docker image into the repository that is associated with your workspace.

NOTE

Azure machine learning [inference source directory](#) upload does not respect `.gitignore` or `.amlignore`

- [Python SDK](#)

APPLIES TO:  Azure CLI ml extension v1

A minimal inference configuration can be written as:

```
{
  "entryScript": "echo_score.py",
  "sourceDirectory": "./source_dir",
  "environment": {
    "docker": {
      "arguments": [],
      "baseDockerfile": null,
      "baseImage": "mcr.microsoft.com/azureml/openmpi3.1.2-ubuntu18.04",
      "enabled": false,
      "sharedVolumes": true,
      "shmSize": null
    },
    "environmentVariables": {
      "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"
    },
    "name": "my-deploy-env",
    "python": {
      "baseCondaEnvironment": null,
      "condaDependencies": {
        "channels": [],
        "dependencies": [
          "python=3.6.2",
          {
            "pip": [
              "azureml-defaults"
            ]
          }
        ],
        "name": "project_environment"
      },
      "condaDependenciesFile": null,
      "interpreterPath": "python",
      "userManagedDependencies": false
    },
    "version": "1"
  }
}
```

Save this file with the name `dummyinferenceconfig.json`.

[See this article](#) for a more thorough discussion of inference configurations.

Define a deployment configuration

A deployment configuration specifies the amount of memory and cores your webservice needs in order to run. It also provides configuration details of the underlying webservice. For example, a deployment configuration lets you specify that your service needs 2 gigabytes of memory, 2 CPU cores, 1 GPU core, and that you want to enable autoscaling.

The options available for a deployment configuration differ depending on the compute target you choose. In a local deployment, all you can specify is which port your webservice will be served on.

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  Azure CLI ml extension v1

The entries in the `deploymentconfig.json` document map to the parameters for `LocalWebservice.deploy_configuration`. The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>computeType</code>	NA	The compute target. For local targets, the value must be <code>local</code> .
<code>port</code>	<code>port</code>	The local port on which to expose the service's HTTP endpoint.

This JSON is an example deployment configuration for use with the CLI:

```
{  
    "computeType": "local",  
    "port": 32267  
}
```

Save this JSON as a file called `deploymentconfig.json`.

For more information, see the [deployment schema](#).

Deploy your machine learning model

You are now ready to deploy your model.

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  Azure CLI ml extension v1

Replace `bidaf_onnx:1` with the name of your model and its version number.

```
az ml model deploy -n myservice \  
    -m bidaf_onnx:1 \  
    --overwrite \  
    --ic dummyinferenceconfig.json \  
    --dc deploymentconfig.json \  
    -g <resource-group> \  
    -w <workspace-name>
```

Call into your model

Let's check that your echo model deployed successfully. You should be able to do a simple liveness request, as well as a scoring request:

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  Azure CLI ml extension v1

```
curl -v http://localhost:32267
curl -v -X POST -H "content-type:application/json" \
-d '{"query": "What color is the fox", "context": "The quick brown fox jumped over the lazy dog."}' \
http://localhost:32267/score
```

Define an entry script

Now it's time to actually load your model. First, modify your entry script:

```
import json
import numpy as np
import os
import onnxruntime
from nltk import word_tokenize
import nltk

def init():
    nltk.download("punkt")
    global sess
    sess = onnxruntime.InferenceSession(
        os.path.join(os.getenv("AZUREML_MODEL_DIR"), "model.onnx")
    )

def run(request):
    print(request)
    text = json.loads(request)
    qw, qc = preprocess(text["query"])
    cw, cc = preprocess(text["context"])

    # Run inference
    test = sess.run(
        None,
        {"query_word": qw, "query_char": qc, "context_word": cw, "context_char": cc},
    )
    start = np.asscalar(test[0])
    end = np.asscalar(test[1])
    ans = [w for w in cw[start : end + 1].reshape(-1)]
    print(ans)
    return ans

def preprocess(word):
    tokens = word_tokenize(word)

    # split into lower-case word tokens, in numpy array with shape of (seq, 1)
    words = np.asarray([w.lower() for w in tokens]).reshape(-1, 1)

    # split words into chars, in numpy array with shape of (seq, 1, 1, 16)
    chars = [[c for c in t][:16] for t in tokens]
    chars = [cs + [""] * (16 - len(cs)) for cs in chars]
    chars = np.asarray(chars).reshape(-1, 1, 1, 16)
    return words, chars
```

Save this file as `score.py` inside of `source_dir`.

Notice the use of the `AZUREML_MODEL_DIR` environment variable to locate your registered model. Now that you've added some pip packages.

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  Azure CLI ml extension v1

```
{  
    "entryScript": "score.py",  
    "sourceDirectory": "./source_dir",  
    "environment": {  
        "docker": {  
            "arguments": [],  
            "baseDockerfile": null,  
            "baseImage": "mcr.microsoft.com/azureml/openmpi3.1.2-ubuntu18.04",  
            "enabled": false,  
            "sharedVolumes": true,  
            "shmSize": null  
        },  
        "environmentVariables": {  
            "EXAMPLE_ENV_VAR": "EXAMPLE_VALUE"  
        },  
        "name": "my-deploy-env",  
        "python": {  
            "baseCondaEnvironment": null,  
            "condaDependencies": {  
                "channels": [],  
                "dependencies": [  
                    "python=3.6.2",  
                    {  
                        "pip": [  
                            "azureml-defaults",  
                            "nltk",  
                            "numpy",  
                            "onnxruntime"  
                        ]  
                    }  
                ],  
                "name": "project_environment"  
            },  
            "condaDependenciesFile": null,  
            "interpreterPath": "python",  
            "userManagedDependencies": false  
        },  
        "version": "2"  
    }  
}
```

Save this file as `inferenceconfig.json`

Deploy again and call your service

Deploy your service again:

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  Azure CLI ml extension v1

Replace `bidaf_onnx:1` with the name of your model and its version number.

```
az ml model deploy -n myservice \
-m bidaf_ponnx:1 \
--overwrite \
--ic inferenceconfig.json \
--dc deploymentconfig.json \
-g <resource-group> \
-w <workspace-name>
```

Then ensure you can send a post request to the service:

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

```
curl -v -X POST -H "content-type:application/json" \
-d '{"query": "What color is the fox", "context": "The quick brown fox jumped over the lazy dog."}' \
http://localhost:32267/score
```

Choose a compute target

The compute target you use to host your model will affect the cost and availability of your deployed endpoint. Use this table to choose an appropriate compute target.

COMPUTE TARGET	USED FOR	GPU SUPPORT	DESCRIPTION
Local web service	Testing/debugging		Use for limited testing and troubleshooting. Hardware acceleration depends on use of libraries in the local system.
Azure Machine Learning Kubernetes	Real-time inference Batch inference	Yes	Run inferencing workloads on on-premises, cloud, and edge Kubernetes clusters.
Azure Container Instances	Real-time inference Recommended for dev/test purposes only.		Use for low-scale CPU-based workloads that require less than 48 GB of RAM. Doesn't require you to manage a cluster. Supported in the designer.
Azure Machine Learning compute clusters	Batch inference	Yes (machine learning pipeline)	Run batch scoring on serverless compute. Supports normal and low-priority VMs. No support for real-time inference.

NOTE

Although compute targets like local, and Azure Machine Learning compute clusters support GPU for training and experimentation, using GPU for inference *when deployed as a web service* is supported only on Azure Machine Learning Kubernetes.

Using a GPU for inference *when scoring with a machine learning pipeline* is supported only on Azure Machine Learning compute.

When choosing a cluster SKU, first scale up and then scale out. Start with a machine that has 150% of the RAM your model requires, profile the result and find a machine that has the performance you need. Once you've learned that, increase the number of machines to fit your need for concurrent inference.

NOTE

- Container instances are suitable only for small models less than 1 GB in size.

Deploy to cloud

Once you've confirmed your service works locally and chosen a remote compute target, you are ready to deploy to the cloud.

Change your deploy configuration to correspond to the compute target you've chosen, in this case Azure Container Instances:

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

The options available for a deployment configuration differ depending on the compute target you choose.

```
{  
  "computeType": "aci",  
  "containerResourceRequirements":  
  {  
    "cpu": 0.5,  
    "memoryInGB": 1.0  
  },  
  "authEnabled": true,  
  "sslEnabled": false,  
  "appInsightsEnabled": false  
}
```

Save this file as `re-deploymentconfig.json`.

For more information, see [this reference](#).

Deploy your service again:

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

Replace `bidaf_onnx:1` with the name of your model and its version number.

```
az ml model deploy -n myservice \
-m bidaf_onnéx:1 \
--overwrite \
--ic inferenceconfig.json \
--dc re-deploymentconfig.json \
-g <resource-group> \
-w <workspace-name>
```

To view the service logs, use the following command:

```
az ml service get-logs -n myservice \
-g <resource-group> \
-w <workspace-name>
```

Call your remote webservice

When you deploy remotely, you may have key authentication enabled. The example below shows how to get your service key with Python in order to make an inference request.

```
import requests
import json
from azureml.core import Webservice

service = Webservice(workspace=ws, name="myservice")
scoring_uri = service.scoring_uri

# If the service is authenticated, set the key or token
key, _ = service.get_keys()

# Set the appropriate headers
headers = {"Content-Type": "application/json"}
headers["Authorization"] = f"Bearer {key}"

# Make the request and display the response and logs
data = {
    "query": "What color is the fox",
    "context": "The quick brown fox jumped over the lazy dog."
}
data = json.dumps(data)
resp = requests.post(scoring_uri, data=data, headers=headers)
print(resp.text)
```

```
print(service.get_logs())
```

See the article on [client applications to consume web services](#) for more example clients in other languages.

How to configure emails in the studio (preview)

To start receiving emails when your job, online endpoint, or batch endpoint is complete or if there's an issue (failed, canceled), follow the preceding instructions.

1. In Azure ML studio, go to settings by selecting the gear icon.
2. Select the **Email notifications** tab.
3. Toggle to enable or disable email notifications for a specific event.

Settings

X

Language

Email Notifications

These settings apply across all workspaces you have access to

Job succeeded

Send me an email when my job succeeded



Enabled

Issue with my job

Send me an email when my job failed or cancelled



Enabled

Issue with my endpoint deployment

Send me an email when my inferencing endpoint deployment failed or cancelled



Enabled

Endpoint deployment succeeded

Send me an email when my inferencing endpoint deployment succeeded



Enabled

Understanding service state

During model deployment, you may see the service state change while it fully deploys.

The following table describes the different service states:

WEBSERVICE STATE	DESCRIPTION	FINAL STATE?
Transitioning	The service is in the process of deployment.	No

WEBSERVICE STATE	DESCRIPTION	FINAL STATE?
Unhealthy	The service has deployed but is currently unreachable.	No
Unschedulable	The service cannot be deployed at this time due to lack of resources.	No
Failed	The service has failed to deploy due to an error or crash.	Yes
Healthy	The service is healthy and the endpoint is available.	Yes

TIP

When deploying, Docker images for compute targets are built and loaded from Azure Container Registry (ACR). By default, Azure Machine Learning creates an ACR that uses the *basic* service tier. Changing the ACR for your workspace to standard or premium tier may reduce the time it takes to build and deploy images to your compute targets. For more information, see [Azure Container Registry service tiers](#).

NOTE

If you are deploying a model to Azure Kubernetes Service (AKS), we advise you enable [Azure Monitor](#) for that cluster. This will help you understand overall cluster health and resource usage. You might also find the following resources useful:

- [Check for Resource Health events impacting your AKS cluster](#)
- [Azure Kubernetes Service Diagnostics](#)

If you are trying to deploy a model to an unhealthy or overloaded cluster, it is expected to experience issues. If you need help troubleshooting AKS cluster problems please contact AKS Support.

Delete resources

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

```
# Get the current model id
import os

stream = os.popen(
    'az ml model list --model-name=bidaf_onnx --latest --query "[0].id" -o tsv'
)
MODEL_ID = stream.read()[0:-1]
MODEL_ID
```

```
az ml service delete -n myservice
az ml service delete -n myaciservice
az ml model delete --model-id=<MODEL_ID>
```

To delete a deployed webservice, use `az ml service delete <name of webservice>`.

To delete a registered model from your workspace, use `az ml model delete <model id>`

Read more about [deleting a webservice](#) and [deleting a model](#).

Next steps

- [Troubleshoot a failed deployment](#)
- [Update web service](#)
- [One click deployment for automated ML runs in the Azure Machine Learning studio](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Create event alerts and triggers for model deployments](#)

Hyperparameter tuning a model with Azure Machine Learning (v1)

9/21/2022 • 14 minutes to read • [Edit Online](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Automate efficient hyperparameter tuning by using Azure Machine Learning (v1) [HyperDrive package](#). Learn how to complete the steps required to tune hyperparameters with the [Azure Machine Learning SDK](#):

1. Define the parameter search space
2. Specify a primary metric to optimize
3. Specify early termination policy for low-performing runs
4. Create and assign resources
5. Launch an experiment with the defined configuration
6. Visualize the training runs
7. Select the best configuration for your model

What is hyperparameter tuning?

Hyperparameters are adjustable parameters that let you control the model training process. For example, with neural networks, you decide the number of hidden layers and the number of nodes in each layer. Model performance depends heavily on hyperparameters.

Hyperparameter tuning, also called **hyperparameter optimization**, is the process of finding the configuration of hyperparameters that results in the best performance. The process is typically computationally expensive and manual.

Azure Machine Learning lets you automate hyperparameter tuning and run experiments in parallel to efficiently optimize hyperparameters.

Define the search space

Tune hyperparameters by exploring the range of values defined for each hyperparameter.

Hyperparameters can be discrete or continuous, and has a distribution of values described by a [parameter expression](#).

Discrete hyperparameters

Discrete hyperparameters are specified as a `choice` among discrete values. `choice` can be:

- one or more comma-separated values

- a `range` object
- any arbitrary `list` object

```
{
    "batch_size": choice(16, 32, 64, 128)
    "number_of_hidden_layers": choice(range(1,5))
}
```

In this case, `batch_size` one of the values [16, 32, 64, 128] and `number_of_hidden_layers` takes one of the values [1, 2, 3, 4].

The following advanced discrete hyperparameters can also be specified using a distribution:

- `quniform(low, high, q)` - Returns a value like $\text{round}(\text{uniform}(\text{low}, \text{high}) / q) * q$
- `qloguniform(low, high, q)` - Returns a value like $\text{round}(\exp(\text{uniform}(\text{low}, \text{high})) / q) * q$
- `qnormal(mu, sigma, q)` - Returns a value like $\text{round}(\text{normal}(\mu, \sigma) / q) * q$
- `qlognormal(mu, sigma, q)` - Returns a value like $\text{round}(\exp(\text{normal}(\mu, \sigma)) / q) * q$

Continuous hyperparameters

The Continuous hyperparameters are specified as a distribution over a continuous range of values:

- `uniform(low, high)` - Returns a value uniformly distributed between low and high
- `loguniform(low, high)` - Returns a value drawn according to $\exp(\text{uniform}(\text{low}, \text{high}))$ so that the logarithm of the return value is uniformly distributed
- `normal(mu, sigma)` - Returns a real value that's normally distributed with mean mu and standard deviation sigma
- `lognormal(mu, sigma)` - Returns a value drawn according to $\exp(\text{normal}(\mu, \sigma))$ so that the logarithm of the return value is normally distributed

An example of a parameter space definition:

```
{
    "learning_rate": normal(10, 3),
    "keep_probability": uniform(0.05, 0.1)
}
```

This code defines a search space with two parameters - `learning_rate` and `keep_probability`. `learning_rate` has a normal distribution with mean value 10 and a standard deviation of 3. `keep_probability` has a uniform distribution with a minimum value of 0.05 and a maximum value of 0.1.

Sampling the hyperparameter space

Specify the parameter sampling method to use over the hyperparameter space. Azure Machine Learning supports the following methods:

- Random sampling
- Grid sampling
- Bayesian sampling

Random sampling

[Random sampling](#) supports discrete and continuous hyperparameters. It supports early termination of low-performance runs. Some users do an initial search with random sampling and then refine the search space to improve results.

In random sampling, hyperparameter values are randomly selected from the defined search space.

```

from azureml.train.hyperdrive import RandomParameterSampling
from azureml.train.hyperdrive import normal, uniform, choice
param_sampling = RandomParameterSampling( {
    "learning_rate": normal(10, 3),
    "keep_probability": uniform(0.05, 0.1),
    "batch_size": choice(16, 32, 64, 128)
}
)

```

Grid sampling

[Grid sampling](#) supports discrete hyperparameters. Use grid sampling if you can budget to exhaustively search over the search space. Supports early termination of low-performance runs.

Grid sampling does a simple grid search over all possible values. Grid sampling can only be used with `choice` hyperparameters. For example, the following space has six samples:

```

from azureml.train.hyperdrive import GridParameterSampling
from azureml.train.hyperdrive import choice
param_sampling = GridParameterSampling( {
    "num_hidden_layers": choice(1, 2, 3),
    "batch_size": choice(16, 32)
}
)

```

Bayesian sampling

[Bayesian sampling](#) is based on the Bayesian optimization algorithm. It picks samples based on how previous samples did, so that new samples improve the primary metric.

Bayesian sampling is recommended if you have enough budget to explore the hyperparameter space. For best results, we recommend a maximum number of runs greater than or equal to 20 times the number of hyperparameters being tuned.

The number of concurrent runs has an impact on the effectiveness of the tuning process. A smaller number of concurrent runs may lead to better sampling convergence, since the smaller degree of parallelism increases the number of runs that benefit from previously completed runs.

Bayesian sampling only supports `choice`, `uniform`, and `quniform` distributions over the search space.

```

from azureml.train.hyperdrive import BayesianParameterSampling
from azureml.train.hyperdrive import uniform, choice
param_sampling = BayesianParameterSampling( {
    "learning_rate": uniform(0.05, 0.1),
    "batch_size": choice(16, 32, 64, 128)
}
)

```

Specify primary metric

Specify the [primary metric](#) you want hyperparameter tuning to optimize. Each training run is evaluated for the primary metric. The early termination policy uses the primary metric to identify low-performance runs.

Specify the following attributes for your primary metric:

- `primary_metric_name` : The name of the primary metric needs to exactly match the name of the metric logged by the training script
- `primary_metric_goal` : It can be either `PrimaryMetricGoal.MAXIMIZE` or `PrimaryMetricGoal.MINIMIZE` and determines whether the primary metric will be maximized or minimized when evaluating the runs.

```
primary_metric_name="accuracy",
primary_metric_goal=PrimaryMetricGoal.MAXIMIZE
```

This sample maximizes "accuracy".

Log metrics for hyperparameter tuning

The training script for your model **must** log the primary metric during model training so that HyperDrive can access it for hyperparameter tuning.

Log the primary metric in your training script with the following sample snippet:

```
from azureml.core.run import Run
run_logger = Run.get_context()
run_logger.log("accuracy", float(val_accuracy))
```

The training script calculates the `val_accuracy` and logs it as the primary metric "accuracy". Each time the metric is logged, it's received by the hyperparameter tuning service. It's up to you to determine the frequency of reporting.

For more information on logging values in model training runs, see [Enable logging in Azure ML training runs](#).

Specify early termination policy

Automatically end poorly performing runs with an early termination policy. Early termination improves computational efficiency.

You can configure the following parameters that control when a policy is applied:

- `evaluation_interval` : the frequency of applying the policy. Each time the training script logs the primary metric counts as one interval. An `evaluation_interval` of 1 will apply the policy every time the training script reports the primary metric. An `evaluation_interval` of 2 will apply the policy every other time. If not specified, `evaluation_interval` is set to 1 by default.
- `delay_evaluation` : delays the first policy evaluation for a specified number of intervals. This is an optional parameter that avoids premature termination of training runs by allowing all configurations to run for a minimum number of intervals. If specified, the policy applies every multiple of `evaluation_interval` that is greater than or equal to `delay_evaluation`.

Azure Machine Learning supports the following early termination policies:

- [Bandit policy](#)
- [Median stopping policy](#)
- [Truncation selection policy](#)
- [No termination policy](#)

Bandit policy

[Bandit policy](#) is based on slack factor/slack amount and evaluation interval. Bandit ends runs when the primary metric isn't within the specified slack factor/slack amount of the most successful run.

NOTE

Bayesian sampling does not support early termination. When using Bayesian sampling, set

```
early_termination_policy = None
```

Specify the following configuration parameters:

- `slack_factor` or `slack_amount`: the slack allowed with respect to the best performing training run. `slack_factor` specifies the allowable slack as a ratio. `slack_amount` specifies the allowable slack as an absolute amount, instead of a ratio.

For example, consider a Bandit policy applied at interval 10. Assume that the best performing run at interval 10 reported a primary metric is 0.8 with a goal to maximize the primary metric. If the policy specifies a `slack_factor` of 0.2, any training runs whose best metric at interval 10 is less than 0.66 ($0.8/(1+ \text{slack_factor})$) will be terminated.

- `evaluation_interval` : (optional) the frequency for applying the policy
- `delay_evaluation` : (optional) delays the first policy evaluation for a specified number of intervals

```
from azureml.train.hyperdrive import BanditPolicy
early_termination_policy = BanditPolicy(slack_factor = 0.1, evaluation_interval=1, delay_evaluation=5)
```

In this example, the early termination policy is applied at every interval when metrics are reported, starting at evaluation interval 5. Any run whose best metric is less than $(1/(1+0.1))$ or 91% of the best performing run will be terminated.

Median stopping policy

[Median stopping](#) is an early termination policy based on running averages of primary metrics reported by the runs. This policy computes running averages across all training runs and stops runs whose primary metric value is worse than the median of the averages.

This policy takes the following configuration parameters:

- `evaluation_interval` : the frequency for applying the policy (optional parameter).
- `delay_evaluation` : delays the first policy evaluation for a specified number of intervals (optional parameter).

```
from azureml.train.hyperdrive import MedianStoppingPolicy
early_termination_policy = MedianStoppingPolicy(evaluation_interval=1, delay_evaluation=5)
```

In this example, the early termination policy is applied at every interval starting at evaluation interval 5. A run is stopped at interval 5 if its best primary metric is worse than the median of the running averages over intervals 1:5 across all training runs.

Truncation selection policy

[Truncation selection](#) cancels a percentage of lowest performing runs at each evaluation interval. Runs are compared using the primary metric.

This policy takes the following configuration parameters:

- `truncation_percentage` : the percentage of lowest performing runs to terminate at each evaluation interval. An integer value between 1 and 99.
- `evaluation_interval` : (optional) the frequency for applying the policy
- `delay_evaluation` : (optional) delays the first policy evaluation for a specified number of intervals
- `exclude_finished_jobs` : specifies whether to exclude finished jobs when applying the policy

```
from azureml.train.hyperdrive import TruncationSelectionPolicy
early_termination_policy = TruncationSelectionPolicy(evaluation_interval=1, truncation_percentage=20,
delay_evaluation=5, exclude_finished_jobs=true)
```

In this example, the early termination policy is applied at every interval starting at evaluation interval 5. A run

terminates at interval 5 if its performance at interval 5 is in the lowest 20% of performance of all runs at interval 5 and will exclude finished jobs when applying the policy.

No termination policy (default)

If no policy is specified, the hyperparameter tuning service will let all training runs execute to completion.

```
policy=None
```

Picking an early termination policy

- For a conservative policy that provides savings without terminating promising jobs, consider a Median Stopping Policy with `evaluation_interval` 1 and `delay_evaluation` 5. These are conservative settings, that can provide approximately 25%-35% savings with no loss on primary metric (based on our evaluation data).
- For more aggressive savings, use Bandit Policy with a smaller allowable slack or Truncation Selection Policy with a larger truncation percentage.

Create and assign resources

Control your resource budget by specifying the maximum number of training runs.

- `max_total_runs` : Maximum number of training runs. Must be an integer between 1 and 1000.
- `max_duration_minutes` : (optional) Maximum duration, in minutes, of the hyperparameter tuning experiment. Runs after this duration are canceled.

NOTE

If both `max_total_runs` and `max_duration_minutes` are specified, the hyperparameter tuning experiment terminates when the first of these two thresholds is reached.

Additionally, specify the maximum number of training runs to run concurrently during your hyperparameter tuning search.

- `max_concurrent_runs` : (optional) Maximum number of runs that can run concurrently. If not specified, all runs launch in parallel. If specified, must be an integer between 1 and 100.

NOTE

The number of concurrent runs is gated on the resources available in the specified compute target. Ensure that the compute target has the available resources for the desired concurrency.

```
max_total_runs=20,  
max_concurrent_runs=4
```

This code configures the hyperparameter tuning experiment to use a maximum of 20 total runs, running four configurations at a time.

Configure hyperparameter tuning experiment

To [configure your hyperparameter tuning](#) experiment, provide the following:

- The defined hyperparameter search space
- Your early termination policy
- The primary metric

- Resource allocation settings
- ScriptRunConfig `script_run_config`

The ScriptRunConfig is the training script that will run with the sampled hyperparameters. It defines the resources per job (single or multi-node), and the compute target to use.

NOTE

The compute target used in `script_run_config` must have enough resources to satisfy your concurrency level. For more information on ScriptRunConfig, see [Configure training runs](#).

Configure your hyperparameter tuning experiment:

```
from azureml.train.hyperdrive import HyperDriveConfig
from azureml.train.hyperdrive import RandomParameterSampling, BanditPolicy, uniform, PrimaryMetricGoal

param_sampling = RandomParameterSampling( {
    'learning_rate': uniform(0.0005, 0.005),
    'momentum': uniform(0.9, 0.99)
})

early_termination_policy = BanditPolicy(slack_factor=0.15, evaluation_interval=1, delay_evaluation=10)

hd_config = HyperDriveConfig(run_config=script_run_config,
                             hyperparameter_sampling=param_sampling,
                             policy=early_termination_policy,
                             primary_metric_name="accuracy",
                             primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
                             max_total_runs=100,
                             max_concurrent_runs=4)
```

The `HyperDriveConfig` sets the parameters passed to the `ScriptRunConfig script_run_config`. The `script_run_config`, in turn, passes parameters to the training script. The above code snippet is taken from the sample notebook [Train, hyperparameter tune, and deploy with PyTorch](#). In this sample, the `learning_rate` and `momentum` parameters will be tuned. Early stopping of runs will be determined by a `BanditPolicy`, which stops a run whose primary metric falls outside the `slack_factor` (see [BanditPolicy class reference](#)).

The following code from the sample shows how the being-tuned values are received, parsed, and passed to the training script's `fine_tune_model` function:

```

# from pytorch_train.py
def main():
    print("Torch version:", torch.__version__)

    # get command-line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('--num_epochs', type=int, default=25,
                        help='number of epochs to train')
    parser.add_argument('--output_dir', type=str, help='output directory')
    parser.add_argument('--learning_rate', type=float,
                        default=0.001, help='learning rate')
    parser.add_argument('--momentum', type=float, default=0.9, help='momentum')
    args = parser.parse_args()

    data_dir = download_data()
    print("data directory is: " + data_dir)
    model = fine_tune_model(args.num_epochs, data_dir,
                            args.learning_rate, args.momentum)
    os.makedirs(args.output_dir, exist_ok=True)
    torch.save(model, os.path.join(args.output_dir, 'model.pt'))

```

IMPORTANT

Every hyperparameter run restarts the training from scratch, including rebuilding the model and *all the data loaders*. You can minimize this cost by using an Azure Machine Learning pipeline or manual process to do as much data preparation as possible prior to your training runs.

Submit hyperparameter tuning experiment

After you define your hyperparameter tuning configuration, [submit the experiment](#):

```

from azureml.core.experiment import Experiment
experiment = Experiment(workspace, experiment_name)
hyperdrive_run = experiment.submit(hd_config)

```

Warm start hyperparameter tuning (optional)

Finding the best hyperparameter values for your model can be an iterative process. You can reuse knowledge from the five previous runs to accelerate hyperparameter tuning.

Warm starting is handled differently depending on the sampling method:

- **Bayesian sampling:** Trials from the previous run are used as prior knowledge to pick new samples, and to improve the primary metric.
- **Random sampling or grid sampling:** Early termination uses knowledge from previous runs to determine poorly performing runs.

Specify the list of parent runs you want to warm start from.

```

from azureml.train.hyperdrive import HyperDriveRun

warmstart_parent_1 = HyperDriveRun(experiment, "warmstart_parent_run_ID_1")
warmstart_parent_2 = HyperDriveRun(experiment, "warmstart_parent_run_ID_2")
warmstart_parents_to_resume_from = [warmstart_parent_1, warmstart_parent_2]

```

If a hyperparameter tuning experiment is canceled, you can resume training runs from the last checkpoint. However, your training script must handle checkpoint logic.

The training run must use the same hyperparameter configuration and mounted the outputs folders. The training script must accept the `resume-from` argument, which contains the checkpoint or model files from which to resume the training run. You can resume individual training runs using the following snippet:

```
from azureml.core.run import Run

resume_child_run_1 = Run(experiment, "resume_child_run_ID_1")
resume_child_run_2 = Run(experiment, "resume_child_run_ID_2")
child_runs_to_resume = [resume_child_run_1, resume_child_run_2]
```

You can configure your hyperparameter tuning experiment to warm start from a previous experiment or resume individual training runs using the optional parameters `resume_from` and `resume_child_runs` in the config:

```
from azureml.train.hyperdrive import HyperDriveConfig

hd_config = HyperDriveConfig(run_config=script_run_config,
                             hyperparameter_sampling=param_sampling,
                             policy=early_termination_policy,
                             resume_from=warmstart_parents_to_resume_from,
                             resume_child_runs=child_runs_to_resume,
                             primary_metric_name="accuracy",
                             primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
                             max_total_runs=100,
                             max_concurrent_runs=4)
```

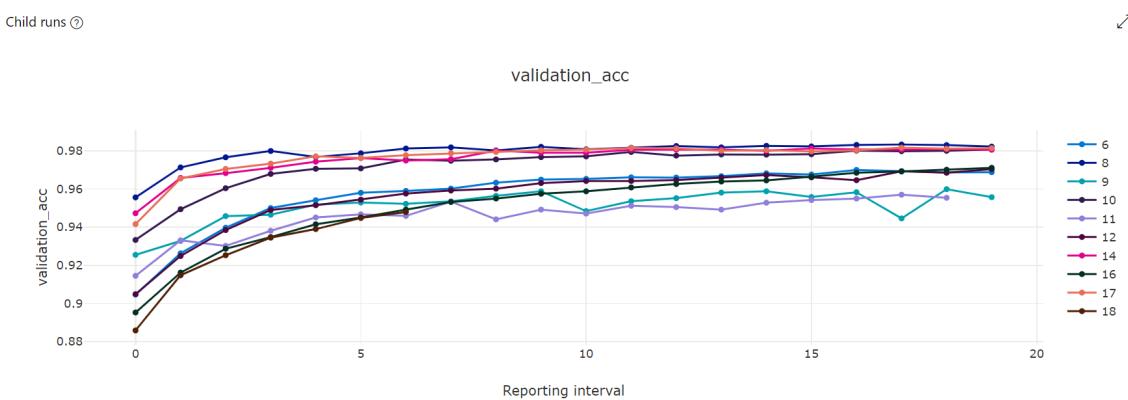
Visualize hyperparameter tuning runs

You can visualize your hyperparameter tuning runs in the Azure Machine Learning studio, or you can use a notebook widget.

Studio

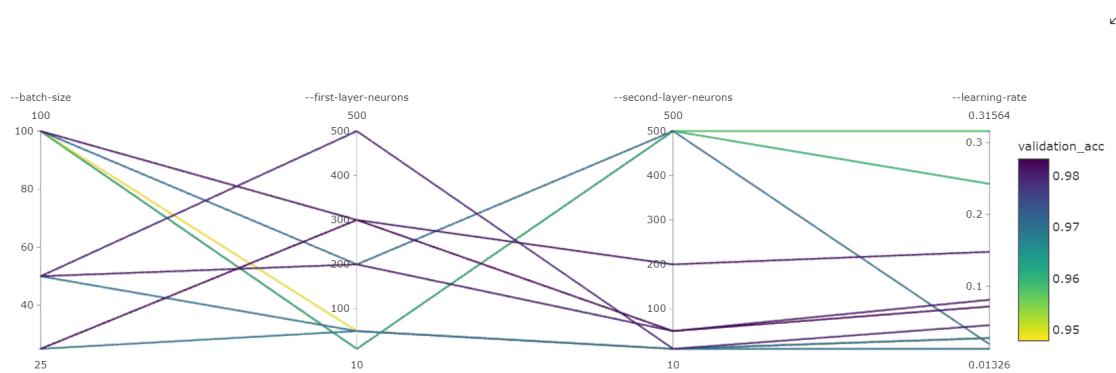
You can visualize all of your hyperparameter tuning runs in the [Azure Machine Learning studio](#). For more information on how to view an experiment in the portal, see [View run records in the studio](#).

- **Metrics chart:** This visualization tracks the metrics logged for each hyperdrive child run over the duration of hyperparameter tuning. Each line represents a child run, and each point measures the primary metric value at that iteration of runtime.

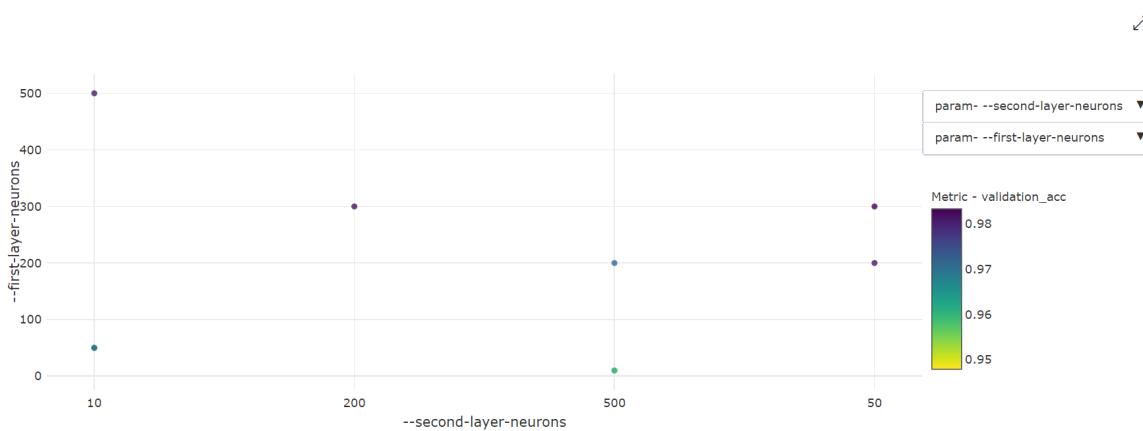


- **Parallel Coordinates Chart:** This visualization shows the correlation between primary metric performance and individual hyperparameter values. The chart is interactive via movement of axes (click and drag by the axis label), and by highlighting values across a single axis (click and drag vertically along a single axis to highlight a range of desired values). The parallel coordinates chart includes an axis on the right most portion of the chart that plots the best metric value corresponding to the hyperparameters set

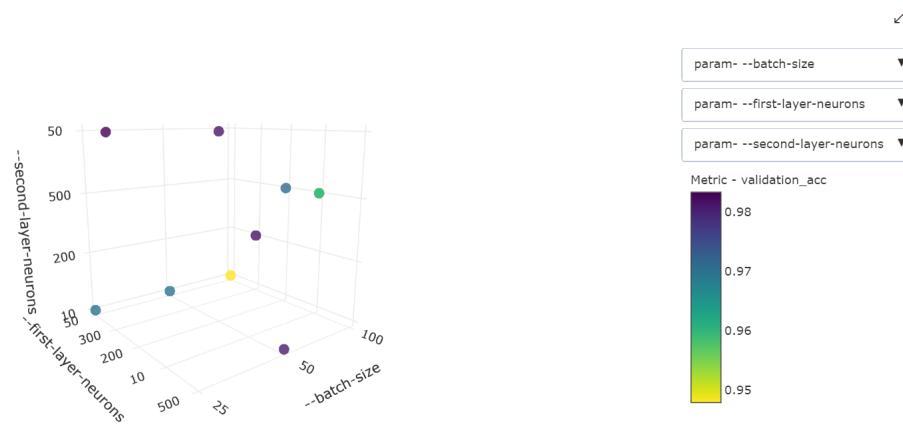
for that run instance. This axis is provided in order to project the chart gradient legend onto the data in a more readable fashion.



- **2-Dimensional Scatter Chart:** This visualization shows the correlation between any two individual hyperparameters along with their associated primary metric value.



- **3-Dimensional Scatter Chart:** This visualization is the same as 2D but allows for three hyperparameter dimensions of correlation with the primary metric value. You can also click and drag to reorient the chart to view different correlations in 3D space.



Notebook widget

Use the [Notebook widget](#) to visualize the progress of your training runs. The following snippet visualizes all your hyperparameter tuning runs in one place in a Jupyter notebook:

```
from azureml.widgets import RunDetails  
RunDetails(hyperdrive_run).show()
```

This code displays a table with details about the training runs for each of the hyperparameter configurations.

Run Properties

Status	Completed
Start Time	9/4/2018 2:55:54 PM
Duration	7 days, 0:01:41
Run Id	cifar_1536098154351
Max concurrent runs	4
Max total runs	100

Output Logs

```
previous status = 'RUNNING')
[2018-09-11T21:57:36.389083][CONTROLLER][INFO]Experiment
was 'ExperimentStatus.RUNNING', is 'ExperimentStatus.FINISHED'.
Run is completed.
```

Failed (6)	Completed (36)	Canceled (58)

Run Best Metric* Status Started Duration Run Id

Run	Best Metric*	Status	Started	Duration	Run Id
103	0.909600019454956	Completed	Sep 4, 2018 2:56 PM	5:58:16	swatig-cifar_1536098154351_0
105	0.9283999800682068	Completed	Sep 4, 2018 2:56 PM	6:04:28	swatig-cifar_1536098154351_3
104	0.9247000217437744	Completed	Sep 4, 2018 2:56 PM	3:32:35	swatig-cifar_1536098154351_2
106	0.9251000285148621	Completed	Sep 4, 2018 2:56 PM	6:01:17	swatig-cifar_1536098154351_1
107	0.9108999967575073	Completed	Sep 4, 2018 6:29 PM	4:18:35	swatig-cifar_1536098154351_4

Pages: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [11](#) [12](#) [13](#) [14](#) [15](#) ... [Next](#) [Last](#)

* The best metric field is obtained from the min/max of primary metric achieved by a run

You can also visualize the performance of each of the runs as training progresses.

Find the best model

Once all of the hyperparameter tuning runs have completed, identify the best performing configuration and hyperparameter values:

```
best_run = hyperdrive_run.get_best_run_by_primary_metric()
best_run_metrics = best_run.get_metrics()
parameter_values = best_run.get_details()['runDefinition']['arguments']

print('Best Run Id: ', best_run.id)
print('\n Accuracy:', best_run_metrics['accuracy'])
print('\n learning rate:',parameter_values[3])
print('\n keep probability:',parameter_values[5])
print('\n batch size:',parameter_values[7])
```

Sample notebook

Refer to train-hyperparameter-* notebooks in this folder:

- [how-to-use-azureml/ml-frameworks](#)

Learn how to run notebooks by following the article [Use Jupyter notebooks to explore this service](#).

Next steps

- [Track an experiment](#)
- [Deploy a trained model](#)

Deploy a model to an Azure Kubernetes Service cluster with v1

9/21/2022 • 16 minutes to read • [Edit Online](#)

IMPORTANT

This article shows how to use the CLI and SDK v1 to deploy a model. For the recommended approach for v2, see [Deploy and score a machine learning model by using an online endpoint](#).

Learn how to use Azure Machine Learning to deploy a model as a web service on Azure Kubernetes Service (AKS). Azure Kubernetes Service is good for high-scale production deployments. Use Azure Kubernetes service if you need one or more of the following capabilities:

- **Fast response time**
- **Autoscaling** of the deployed service
- **Logging**
- **Model data collection**
- **Authentication**
- **TLS termination**
- **Hardware acceleration** options such as GPU and field-programmable gate arrays (FPGA)

When deploying to Azure Kubernetes Service, you deploy to an AKS cluster that is **connected to your workspace**. For information on connecting an AKS cluster to your workspace, see [Create and attach an Azure Kubernetes Service cluster](#).

IMPORTANT

We recommend that you debug locally before deploying to the web service. For more information, see [Debug Locally](#)

You can also refer to Azure Machine Learning - [Deploy to Local Notebook](#)

NOTE

Azure Machine Learning Endpoints (preview) provide an improved, simpler deployment experience. Endpoints support both real-time and batch inference scenarios. Endpoints provide a unified interface to invoke and manage model deployments across compute types. See [What are Azure Machine Learning endpoints \(preview\)?](#).

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create an Azure Machine Learning workspace](#).
- A machine learning model registered in your workspace. If you don't have a registered model, see [How and where to deploy models](#).
- The [Azure CLI extension \(v1\)](#) for Machine Learning service, [Azure Machine Learning Python SDK](#), or the [Azure Machine Learning Visual Studio Code extension](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-m1`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `m1`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- The Python code snippets in this article assume that the following variables are set:
 - `ws` - Set to your workspace.
 - `model` - Set to your registered model.
 - `inference_config` - Set to the inference configuration for the model.
- For more information on setting these variables, see [How and where to deploy models](#).
- The CLI snippets in this article assume that you've created an `inferenceconfig.json` document. For more information on creating this document, see [How and where to deploy models](#).
- An Azure Kubernetes Service cluster connected to your workspace. For more information, see [Create and attach an Azure Kubernetes Service cluster](#).
 - If you want to deploy models to GPU nodes or FPGA nodes (or any specific SKU), then you must create a cluster with the specific SKU. There's no support for creating a secondary node pool in an existing cluster and deploying models in the secondary node pool.

Understand the deployment processes

The word "deployment" is used in both Kubernetes and Azure Machine Learning. "Deployment" has different meanings in these two contexts. In Kubernetes, a `Deployment` is a concrete entity, specified with a declarative YAML file. A Kubernetes `Deployment` has a defined lifecycle and concrete relationships to other Kubernetes entities such as `Pods` and `ReplicaSets`. You can learn about Kubernetes from docs and videos at [What is Kubernetes?](#)

In Azure Machine Learning, "deployment" is used in the more general sense of making available and cleaning up your project resources. The steps that Azure Machine Learning considers part of deployment are:

1. Zipping the files in your project folder, ignoring those specified in `.amlignore` or `.gitignore`
2. Scaling up your compute cluster (Relates to Kubernetes)
3. Building or downloading the dockerfile to the compute node (Relates to Kubernetes)
 - a. The system calculates a hash of:
 - The base image
 - Custom docker steps (see [Deploy a model using a custom Docker base image](#))
 - The conda definition YAML (see [Create & use software environments in Azure Machine Learning](#))
 - b. The system uses this hash as the key in a lookup of the workspace Azure Container Registry (ACR)
 - c. If it isn't found, it looks for a match in the global ACR
 - d. If it isn't found, the system builds a new image (which will be cached and pushed to the workspace ACR)
4. Downloading your zipped project file to temporary storage on the compute node
5. Unzipping the project file
6. The compute node executing `python <entry script> <arguments>`
7. Saving logs, model files, and other files written to `./outputs` to the storage account associated with the

workspace

- Scaling down compute, including removing temporary storage (Relates to Kubernetes)

Azure ML router

The front-end component (azureml-fe) that routes incoming inference requests to deployed services automatically scales as needed. Scaling of azureml-fe is based on the AKS cluster purpose and size (number of nodes). The cluster purpose and nodes are configured when you [create or attach an AKS cluster](#). There's one azureml-fe service per cluster, which may be running on multiple pods.

IMPORTANT

When using a cluster configured as **dev-test**, the self-scaler is **disabled**. Even for FastProd/DenseProd clusters, Self-Scaler is only enabled when telemetry shows that it's needed.

Azureml-fe scales both up (vertically) to use more cores, and out (horizontally) to use more pods. When making the decision to scale up, the time that it takes to route incoming inference requests is used. If this time exceeds the threshold, a scale-up occurs. If the time to route incoming requests continues to exceed the threshold, a scale-out occurs.

When scaling down and in, CPU usage is used. If the CPU usage threshold is met, the front end will first be scaled down. If the CPU usage drops to the scale-in threshold, a scale-in operation happens. Scaling up and out will only occur if there are enough cluster resources available.

When scale-up or scale-down, azureml-fe pods will be restarted to apply the cpu/memory changes. Inferencing requests aren't affected by the restarts.

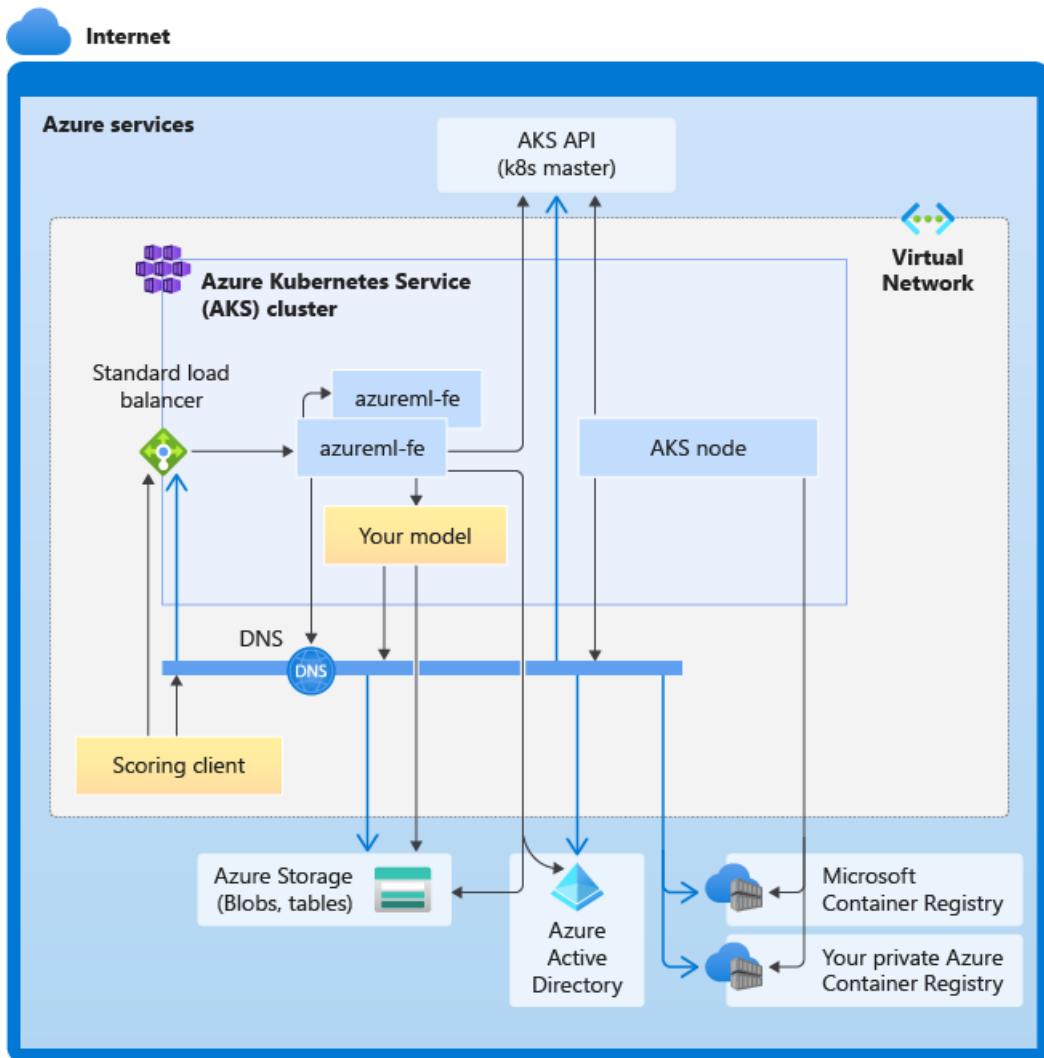
Understand connectivity requirements for AKS inferencing cluster

When Azure Machine Learning creates or attaches an AKS cluster, AKS cluster is deployed with one of the following two network models:

- Kubenet networking - The network resources are typically created and configured as the AKS cluster is deployed.
- Azure Container Networking Interface (CNI) networking - The AKS cluster is connected to an existing virtual network resource and configurations.

For Kubenet networking, the network is created and configured properly for Azure Machine Learning service. For the CNI networking, you need to understand the connectivity requirements and ensure DNS resolution and outbound connectivity for AKS inferencing. For example, you may be using a firewall to block network traffic.

The following diagram shows the connectivity requirements for AKS inferencing. Black arrows represent actual communication, and blue arrows represent the domain names. You may need to add entries for these hosts to your firewall or to your custom DNS server.



For general AKS connectivity requirements, see [Control egress traffic for cluster nodes in Azure Kubernetes Service](#).

For accessing Azure ML services behind a firewall, see [How to access azureml behind firewall](#).

Overall DNS resolution requirements

DNS resolution within an existing VNet is under your control. For example, a firewall or custom DNS server. The following hosts must be reachable:

HOST NAME	USED BY
<cluster>.hcp.<region>.azmk8s.io	AKS API server
mcr.microsoft.com	Microsoft Container Registry (MCR)
<ACR name>.azurecr.io	Your Azure Container Registry (ACR)
<account>.table.core.windows.net	Azure Storage Account (table storage)
<account>.blob.core.windows.net	Azure Storage Account (blob storage)
api.azureml.ms	Azure Active Directory (Azure AD) authentication
ingest-vienna<region>.kusto.windows.net	Kusto endpoint for uploading telemetry

HOST NAME	USED BY
<leaf-domain-label + auto-generated suffix>. <region>.cloudapp.azure.com	Endpoint domain name, if you autogenerated by Azure Machine Learning. If you used a custom domain name, you don't need this entry.

Connectivity requirements in chronological order: from cluster creation to model deployment

In the process of AKS create or attach, Azure ML router (azureml-fe) is deployed into the AKS cluster. In order to deploy Azure ML router, AKS node should be able to:

- Resolve DNS for AKS API server
- Resolve DNS for MCR in order to download docker images for Azure ML router
- Download images from MCR, where outbound connectivity is required

Right after azureml-fe is deployed, it will attempt to start and this requires to:

- Resolve DNS for AKS API server
- Query AKS API server to discover other instances of itself (it's a multi-pod service)
- Connect to other instances of itself

Once azureml-fe is started, it requires the following connectivity to function properly:

- Connect to Azure Storage to download dynamic configuration
- Resolve DNS for Azure AD authentication server api.azureml.ms and communicate with it when the deployed service uses Azure AD authentication.
- Query AKS API server to discover deployed models
- Communicate to deployed model PODs

At model deployment time, for a successful model deployment AKS node should be able to:

- Resolve DNS for customer's ACR
- Download images from customer's ACR
- Resolve DNS for Azure BLOBs where model is stored
- Download models from Azure BLOBs

After the model is deployed and service starts, azureml-fe will automatically discover it using AKS API, and will be ready to route request to it. It must be able to communicate to model PODs.

NOTE

If the deployed model requires any connectivity (e.g. querying external database or other REST service, downloading a BLOB etc), then both DNS resolution and outbound communication for these services should be enabled.

Deploy to AKS

To deploy a model to Azure Kubernetes Service, create a **deployment configuration** that describes the compute resources needed. For example, number of cores and memory. You also need an **inference configuration**, which describes the environment needed to host the model and web service. For more information on creating the inference configuration, see [How and where to deploy models](#).

NOTE

The number of models to be deployed is limited to 1,000 models per deployment (per container).

- Python SDK
 - Azure CLI
 - Visual Studio Code

APPLIES TO: Python SDK azureml v1

```
from azureml.core.webservice import AksWebservice, Webservice
from azureml.core.model import Model
from azureml.core.compute import AksCompute

aks_target = AksCompute(ws,"myaks")
# If deploying to a cluster configured for dev/test, ensure that it was created with enough
# cores and memory to handle this deployment configuration. Note that memory is also used by
# things such as dependencies and AML components.
deployment_config = AksWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)
service = Model.deploy(ws, "myservice", [model], inference_config, deployment_config, aks_target)
service.wait_for_deployment(show_output = True)
print(service.state)
print(service.get_logs())
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- AksCompute
 - AksWebservice.deploy_configuration
 - Model.deploy
 - Webservice.wait_for_deployment

Autoscaling

APPLIES TO: Python SDK azureml v1

The component that handles autoscaling for Azure ML model deployments is `azureml-fe`, which is a smart request router. Since all inference requests go through it, it has the necessary data to automatically scale the deployed model(s).

IMPORTANT

- **Do not enable Kubernetes Horizontal Pod Autoscaler (HPA) for model deployments.** Doing so would cause the two auto-scaling components to compete with each other. Azureml-fe is designed to auto-scale models deployed by Azure ML, where HPA would have to guess or approximate model utilization from a generic metric like CPU usage or a custom metric configuration.
 - **Azureml-fe does not scale the number of nodes in an AKS cluster**, because this could lead to unexpected cost increases. Instead, it scales the number of replicas for the model within the physical cluster boundaries. If you need to scale the number of nodes within the cluster, you can manually scale the cluster or [configure the AKS cluster autoscaler](#).

Autoscaling can be controlled by setting `autoscale_target_utilization`, `autoscale_min_replicas`, and `autoscale_max_replicas` for the AKS web service. The following example demonstrates how to enable autoscaling:

Decisions to scale up/down is based off of utilization of the current container replicas. The number of replicas that are busy (processing a request) divided by the total number of current replicas is the current utilization. If this number exceeds `autoscale_target_utilization`, then more replicas are created. If it's lower, then replicas are reduced. By default, the target utilization is 70%.

Decisions to add replicas are eager and fast (around 1 second). Decisions to remove replicas are conservative (around 1 minute).

You can calculate the required replicas by using the following code:

```
from math import ceil
# target requests per second
targetRps = 20
# time to process the request (in seconds)
reqTime = 10
# Maximum requests per container
maxReqPerContainer = 1
# target_utilization. 70% in this example
targetUtilization = .7

concurrentRequests = targetRps * reqTime / targetUtilization

# Number of container replicas
replicas = ceil(concurrentRequests / maxReqPerContainer)
```

For more information on setting `autoscale_target_utilization`, `autoscale_max_replicas`, and `autoscale_min_replicas`, see the [AksWebservice](#) module reference.

Web service authentication

When deploying to Azure Kubernetes Service, **key-based** authentication is enabled by default. You can also enable **token-based** authentication. Token-based authentication requires clients to use an Azure Active Directory account to request an authentication token, which is used to make requests to the deployed service.

To **disable** authentication, set the `auth_enabled=False` parameter when creating the deployment configuration. The following example disables authentication using the SDK:

```
deployment_config = AksWebservice.deploy_configuration(cpu_cores=1, memory_gb=1, auth_enabled=False)
```

For information on authenticating from a client application, see the [Consume an Azure Machine Learning model deployed as a web service](#).

Authentication with keys

If key authentication is enabled, you can use the `get_keys` method to retrieve a primary and secondary authentication key:

```
primary, secondary = service.get_keys()
print(primary)
```

IMPORTANT

If you need to regenerate a key, use `service.regen_key`

Authentication with tokens

To enable token authentication, set the `token_auth_enabled=True` parameter when you're creating or updating a

deployment. The following example enables token authentication using the SDK:

```
deployment_config = AksWebservice.deploy_configuration(cpu_cores=1, memory_gb=1, token_auth_enabled=True)
```

If token authentication is enabled, you can use the `get_token` method to retrieve a JWT token and that token's expiration time:

```
token, refresh_by = service.get_token()  
print(token)
```

IMPORTANT

You will need to request a new token after the token's `refresh_by` time.

Microsoft strongly recommends that you create your Azure Machine Learning workspace in the same region as your Azure Kubernetes Service cluster. To authenticate with a token, the web service will make a call to the region in which your Azure Machine Learning workspace is created. If your workspace's region is unavailable, then you will not be able to fetch a token for your web service even, if your cluster is in a different region than your workspace. This effectively results in Token-based Authentication being unavailable until your workspace's region is available again. In addition, the greater the distance between your cluster's region and your workspace's region, the longer it will take to fetch a token.

To retrieve a token, you must use the Azure Machine Learning SDK or the `az ml service get-access-token` command.

Vulnerability scanning

Microsoft Defender for Cloud provides unified security management and advanced threat protection across hybrid cloud workloads. You should allow Microsoft Defender for Cloud to scan your resources and follow its recommendations. For more, see [Azure Kubernetes Services integration with Defender for Cloud](#).

Next steps

- [Use Azure RBAC for Kubernetes authorization](#)
- [Secure inferencing environment with Azure Virtual Network](#)
- [How to deploy a model using a custom Docker image](#)
- [Deployment troubleshooting](#)
- [Update web service](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Consume a ML Model deployed as a web service](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)

Deploy a model to Azure Container Instances with CLI (v1)

9/21/2022 • 4 minutes to read • [Edit Online](#)

IMPORTANT

This article shows how to use the CLI and SDK v1 to deploy a model. For the recommended approach for v2, see [Deploy and score a machine learning model by using an online endpoint](#).

Learn how to use Azure Machine Learning to deploy a model as a web service on Azure Container Instances (ACI). Use Azure Container Instances if you:

- prefer not to manage your own Kubernetes cluster
- Are OK with having only a single replica of your service, which may impact uptime

For information on quota and region availability for ACI, see [Quotas and region availability for Azure Container Instances](#) article.

IMPORTANT

It is highly advised to debug locally before deploying to the web service, for more information see [Debug Locally](#)

You can also refer to Azure Machine Learning - [Deploy to Local Notebook](#)

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create an Azure Machine Learning workspace](#).
- A machine learning model registered in your workspace. If you don't have a registered model, see [How and where to deploy models](#).
- The [Azure CLI extension \(v1\)](#) for Machine Learning service, [Azure Machine Learning Python SDK](#), or the [Azure Machine Learning Visual Studio Code extension](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-m1`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `m1`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

- The **Python** code snippets in this article assume that the following variables are set:
 - `ws` - Set to your workspace.
 - `model` - Set to your registered model.
 - `inference_config` - Set to the inference configuration for the model.

For more information on setting these variables, see [How and where to deploy models](#).

- The CLI snippets in this article assume that you've created an `inferenceconfig.json` document. For more information on creating this document, see [How and where to deploy models](#).

Limitations

When your Azure Machine Learning workspace is configured with a private endpoint, deploying to Azure Container Instances in a VNet is not supported. Instead, consider using a [Managed online endpoint with network isolation](#).

Deploy to ACI

To deploy a model to Azure Container Instances, create a **deployment configuration** that describes the compute resources needed. For example, number of cores and memory. You also need an **inference configuration**, which describes the environment needed to host the model and web service. For more information on creating the inference configuration, see [How and where to deploy models](#).

NOTE

- ACI is suitable only for small models that are under 1 GB in size.
- We recommend using single-node AKS to dev-test larger models.
- The number of models to be deployed is limited to 1,000 models per deployment (per container).

Using the SDK

APPLIES TO:  Python SDK azureml v1

```
from azureml.core.webservice import AciWebservice, Webservice
from azureml.core.model import Model

deployment_config = AciWebservice.deploy_configuration(cpu_cores = 1, memory_gb = 1)
service = Model.deploy(ws, "aciservice", [model], inference_config, deployment_config)
service.wait_for_deployment(show_output = True)
print(service.state)
```

For more information on the classes, methods, and parameters used in this example, see the following reference documents:

- [AciWebservice.deploy_configuration](#)
- [Model.deploy](#)
- [Webservice.wait_for_deployment](#)

Using the Azure CLI

APPLIES TO:  Azure CLI ml extension v1

To deploy using the CLI, use the following command. Replace `mymodel:1` with the name and version of the registered model. Replace `myservice` with the name to give this service:

```
az ml model deploy -n myservice -m mymodel:1 --ic inferenceconfig.json --dc deploymentconfig.json
```

The entries in the `deploymentconfig.json` document map to the parameters for [AciWebservice.deploy_configuration](#). The following table describes the mapping between the entities in the JSON document and the parameters for the method:

JSON ENTITY	METHOD PARAMETER	DESCRIPTION
<code>computeType</code>	NA	The compute target. For ACI, the value must be <code>ACI</code> .
<code>containerResourceRequirements</code>	NA	Container for the CPU and memory entities.
<code>cpu</code>	<code>cpu_cores</code>	The number of CPU cores to allocate. Defaults, <code>0.1</code>
<code>memoryInGB</code>	<code>memory_gb</code>	The amount of memory (in GB) to allocate for this web service. Default, <code>0.5</code>
<code>location</code>	<code>location</code>	The Azure region to deploy this Webservice to. If not specified the Workspace location will be used. More details on available regions can be found here: ACI Regions
<code>authEnabled</code>	<code>auth_enabled</code>	Whether to enable auth for this Webservice. Defaults to False
<code>sslEnabled</code>	<code>ssl_enabled</code>	Whether to enable SSL for this Webservice. Defaults to False.
<code>appInsightsEnabled</code>	<code>enable_app_insights</code>	Whether to enable AppInsights for this Webservice. Defaults to False
<code>sslCertificate</code>	<code>ssl_cert_pem_file</code>	The cert file needed if SSL is enabled
<code>sslKey</code>	<code>ssl_key_pem_file</code>	The key file needed if SSL is enabled
<code>cname</code>	<code>ssl_cname</code>	The cname for if SSL is enabled
<code>dnsNameLabel</code>	<code>dns_name_label</code>	The dns name label for the scoring endpoint. If not specified a unique dns name label will be generated for the scoring endpoint.

The following JSON is an example deployment configuration for use with the CLI:

```
{
  "computeType": "aci",
  "containerResourceRequirements":
  {
    "cpu": 0.5,
    "memoryInGB": 1.0
  },
  "authEnabled": true,
  "sslEnabled": false,
  "appInsightsEnabled": false
}
```

For more information, see the [az ml model deploy](#) reference.

Using VS Code

See [how to manage resources in VS Code](#).

IMPORTANT

You don't need to create an ACI container to test in advance. ACI containers are created as needed.

IMPORTANT

We append hashed workspace id to all underlying ACI resources which are created, all ACI names from same workspace will have same suffix. The Azure Machine Learning service name would still be the same customer provided "service_name" and all the user facing Azure Machine Learning SDK APIs do not need any change. We do not give any guarantees on the names of underlying resources being created.

Next steps

- [How to deploy a model using a custom Docker image](#)
- [Deployment troubleshooting](#)
- [Update the web service](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Consume a ML Model deployed as a web service](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)

Deploy MLflow models as Azure web services

9/21/2022 • 4 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this article, learn how to deploy your [MLflow](#) model as an Azure web service, so you can leverage and apply Azure Machine Learning's model management and data drift detection capabilities to your production models. See [MLflow and Azure Machine Learning](#) for additional MLflow and Azure Machine Learning functionality integrations.

Azure Machine Learning offers deployment configurations for:

- Azure Container Instance (ACI) which is a suitable choice for a quick dev-test deployment.
- Azure Kubernetes Service (AKS) which is recommended for scalable production deployments.

NOTE

Azure Machine Learning Endpoints (preview) provide an improved, simpler deployment experience. Endpoints support both real-time and batch inference scenarios. Endpoints provide a unified interface to invoke and manage model deployments across compute types. See [What are Azure Machine Learning endpoints \(preview\)?](#).

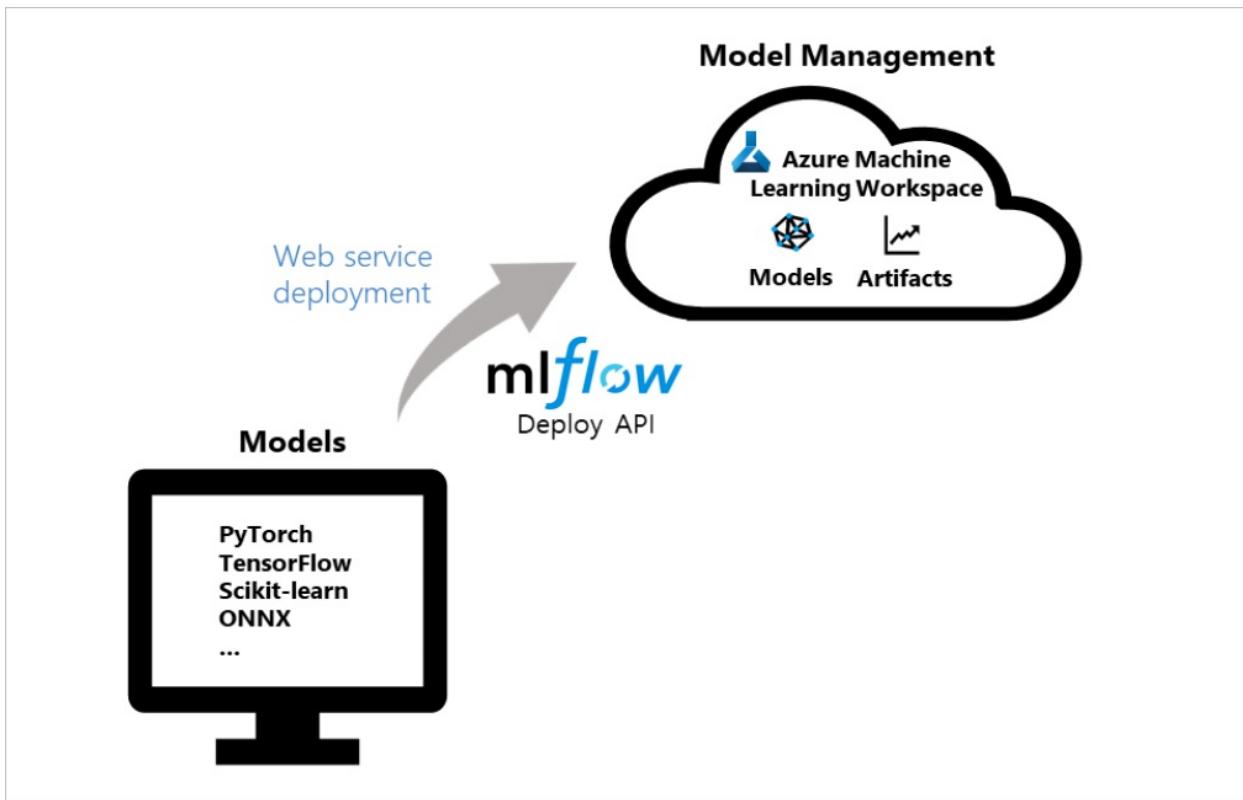
TIP

The information in this document is primarily for data scientists and developers who want to deploy their MLflow model to an Azure Machine Learning web service endpoint. If you are an administrator interested in monitoring resource usage and events from Azure Machine Learning, such as quotas, completed training runs, or completed model deployments, see [Monitoring Azure Machine Learning](#).

MLflow with Azure Machine Learning deployment

MLflow is an open-source library for managing the life cycle of your machine learning experiments. Its integration with Azure Machine Learning allows for you to extend this management beyond model training to the deployment phase of your production model.

The following diagram demonstrates that with the MLflow deploy API and Azure Machine Learning, you can deploy models created with popular frameworks, like PyTorch, Tensorflow, scikit-learn, etc., as Azure web services and manage them in your workspace.



Prerequisites

- A machine learning model. If you don't have a trained model, find the notebook example that best fits your compute scenario in [this repo](#) and follow its instructions.
- [Set up the MLflow Tracking URI to connect Azure Machine Learning](#).
- Install the `azureml-mlflow` package.
 - This package automatically brings in `azureml-core` of the [The Azure Machine Learning Python SDK](#), which provides the connectivity for MLflow to access your workspace.
- See which [access permissions you need to perform your MLflow operations with your workspace](#).

Deploy to Azure Container Instance (ACI)

To deploy your MLflow model to an Azure Machine Learning web service, your model must be set up with the [MLflow Tracking URI to connect with Azure Machine Learning](#).

In order to deploy to ACI, you don't need to define any deployment configuration, the service will default to an ACI deployment when a config is not provided. Then, register and deploy the model in one step with MLflow's `deploy` method for Azure Machine Learning.

```
from mlflow.deployments import get_deploy_client

# set the tracking uri as the deployment client
client = get_deploy_client(mlflow.get_tracking_uri())

# set the model path
model_path = "model"

# define the model path and the name is the service name
# the model gets registered automatically and a name is autogenerated using the "name" parameter below
client.create_deployment(name="mlflow-test-aci", model_uri='runs:/{}{}'.format(run.id, model_path))
```

Customize deployment configuration

If you prefer not to use the defaults, you can set up your deployment configuration with a deployment config

json file that uses parameters from the [deploy_configuration\(\)](#) method as reference.

For your deployment config json file, each of the deployment config parameters need to be defined in the form of a dictionary. The following is an example. [Learn more about what your deployment configuration json file can contain.](#)

Azure Container Instance deployment configuration schema

```
{"computeType": "aci",
 "containerResourceRequirements": {"cpu": 1, "memoryInGB": 1},
 "location": "eastus2"
}
```

Your json file can then be used to create your deployment.

```
# set the deployment config
deploy_path = "deployment_config.json"
test_config = {'deploy-config-file': deploy_path}

client.create_deployment(model_uri='runs:/{}{}'.format(run.id, model_path),
                         config=test_config,
                         name="mlflow-test-aci")
```

Deploy to Azure Kubernetes Service (AKS)

To deploy your MLflow model to an Azure Machine Learning web service, your model must be set up with the [MLflow Tracking URI to connect with Azure Machine Learning](#).

To deploy to AKS, first create an AKS cluster. Create an AKS cluster using the [ComputeTarget.create\(\)](#) method. It may take 20-25 minutes to create a new cluster.

```
from azureml.core.compute import AksCompute, ComputeTarget

# Use the default configuration (can also provide parameters to customize)
prov_config = AksCompute.provisioning_configuration()

aks_name = 'aks-mlflow'

# Create the cluster
aks_target = ComputeTarget.create(workspace=ws,
                                   name=aks_name,
                                   provisioning_configuration=prov_config)

aks_target.wait_for_completion(show_output = True)

print(aks_target.provisioning_state)
print(aks_target.provisioning_errors)
```

Create a deployment config json using [deploy_configuration\(\)](#) method values as a reference. Each of the deployment config parameters simply need to be defined as a dictionary. Here's an example below:

```
{"computeType": "aks", "computeTargetName": "aks-mlflow"}
```

Then, register and deploy the model in one step with MLflow's [deployment client](#).

```
from mlflow.deployments import get_deploy_client

# set the tracking uri as the deployment client
client = get_deploy_client(mlflow.get_tracking_uri())

# set the model path
model_path = "model"

# set the deployment config
deploy_path = "deployment_config.json"
test_config = {'deploy-config-file': deploy_path}

# define the model path and the name is the service name
# the model gets registered automatically and a name is autogenerated using the "name" parameter below
client.create_deployment(model_uri='runs:/{}{}'.format(run.id, model_path),
                          config=test_config,
                          name="mlflow-test-aci")
```

The service deployment can take several minutes.

Clean up resources

If you don't plan to use your deployed web service, use `service.delete()` to delete it from your notebook. For more information, see the documentation for [WebService.delete\(\)](#).

Example notebooks

The [MLflow with Azure Machine Learning notebooks](#) demonstrate and expand upon concepts presented in this article.

NOTE

A community-driven repository of examples using mlflow can be found at <https://github.com/Azure/azureml-examples>.

Next steps

- [Manage your models](#).
- Monitor your production models for [data drift](#).
- [Track Azure Databricks runs with MLflow](#).

Update a deployed web service (v1)

9/21/2022 • 3 minutes to read • [Edit Online](#)

APPLIES TO: Azure CLI ml extension v1 Python SDK azureml v1

In this article, you learn how to update a web service that was deployed with Azure Machine Learning.

Prerequisites

- This article assumes you have already deployed a web service with Azure Machine Learning. If you need to learn how to deploy a web service, [follow these steps](#).
- The code snippets in this article assume that the `ws` variable has already been initialized to your workspace by using the `Workflow()` constructor or loading a saved configuration with `Workspace.from_config()`. The following snippet demonstrates how to use the constructor:

APPLIES TO: Python SDK azureml v1

```
from azureml.core import Workspace
ws = Workspace(subscription_id="mysubscriptionid",
               resource_group="myresourcegroup",
               workspace_name="myworkspace")
```

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Update web service

To update a web service, use the `update` method. You can update the web service to use a new model, a new entry script, or new dependencies that can be specified in an inference configuration. For more information, see the documentation for [Webservice.update](#).

See [AKS Service Update Method](#).

See [ACI Service Update Method](#).

IMPORTANT

When you create a new version of a model, you must manually update each service that you want to use it.

You can not use the SDK to update a web service published from the Azure Machine Learning designer.

IMPORTANT

Azure Kubernetes Service uses [Blobfuse FlexVolume driver](#) for the versions <=1.16 and [Blob CSI driver](#) for the versions >=1.17.

Therefore, it is important to re-deploy or update the web service after cluster upgrade in order to deploy to correct blobfuse method for the cluster version.

NOTE

When an operation is already in progress, any new operation on that same web service will respond with 409 conflict error. For example, If create or update web service operation is in progress and if you trigger a new Delete operation it will throw an error.

Using the SDK

The following code shows how to use the SDK to update the model, environment, and entry script for a web service:

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core import Environment
from azureml.core.webservice import Webservice
from azureml.core.model import Model, InferenceConfig

# Register new model.
new_model = Model.register(model_path="outputs/sklearn_mnist_model.pkl",
                           model_name="sklearn_mnist",
                           tags={"key": "0.1"},
                           description="test",
                           workspace=ws)

# Use version 3 of the environment.
deploy_env = Environment.get(workspace=ws, name="myenv", version="3")
inference_config = InferenceConfig(entry_script="score.py",
                                   environment=deploy_env)

service_name = 'myservice'
# Retrieve existing service.
service = Webservice(name=service_name, workspace=ws)

# Update to new model(s).
service.update(models=[new_model], inference_config=inference_config)
service.wait_for_deployment(show_output=True)
print(service.state)
print(service.get_logs())
```

Using the CLI

You can also update a web service by using the ML CLI. The following example demonstrates registering a new model and then updating a web service to use the new model:

APPLIES TO:  [Azure CLI ml extension v1](#)

```
az ml model register -n sklearn_mnist --asset-path outputs/sklearn_mnist_model.pkl --experiment-name myexperiment --output-metadata-file modelinfo.json
az ml service update -n myservice --model-metadata-file modelinfo.json
```

TIP

In this example, a JSON document is used to pass the model information from the registration command into the update command.

To update the service to use a new entry script or environment, create an [inference configuration file](#) and specify it with the `ic` parameter.

For more information, see the [az ml service update](#) documentation.

Next steps

- [Troubleshoot a failed deployment](#)
- [Create client applications to consume web services](#)
- [How to deploy a model using a custom Docker image](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)
- [Create event alerts and triggers for model deployments](#)

Deploy models trained with Azure Machine Learning on your local machines

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article describes how to use your local computer as a target for training or deploying models created in Azure Machine Learning. Azure Machine Learning is flexible enough to work with most Python machine learning frameworks. Machine learning solutions generally have complex dependencies that can be difficult to duplicate. This article will show you how to balance total control with ease of use.

Scenarios for local deployment include:

- Quickly iterating data, scripts, and models early in a project.
- Debugging and troubleshooting in later stages.
- Final deployment on user-managed hardware.

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create workspace resources](#).
- A model and an environment. If you don't have a trained model, you can use the model and dependency files provided in [this tutorial](#).
- The [Azure Machine Learning SDK for Python](#).
- A conda manager, like Anaconda or Miniconda, if you want to mirror Azure Machine Learning package dependencies.
- Docker, if you want to use a containerized version of the Azure Machine Learning environment.

Prepare your local machine

The most reliable way to locally run an Azure Machine Learning model is with a Docker image. A Docker image provides an isolated, containerized experience that duplicates, except for hardware issues, the Azure execution environment. For more information on installing and configuring Docker for development scenarios, see [Overview of Docker remote development on Windows](#).

It's possible to attach a debugger to a process running in Docker. (See [Attach to a running container](#).) But you might prefer to debug and iterate your Python code without involving Docker. In this scenario, it's important that your local machine uses the same libraries that are used when you run your experiment in Azure Machine Learning. To manage Python dependencies, Azure uses [conda](#). You can re-create the environment by using other package managers, but installing and configuring conda on your local machine is the easiest way to synchronize.

IMPORTANT

GPU base images can't be used for local deployment, unless the local deployment is on an Azure Machine Learning compute instance. GPU base images are supported only on Microsoft Azure Services such as Azure Machine Learning compute clusters and instances, Azure Container Instance (ACI), Azure VMs, or Azure Kubernetes Service (AKS).

Prepare your entry script

Even if you use Docker to manage the model and dependencies, the Python scoring script must be local. The script must have two methods:

- An `init()` method that takes no arguments and returns nothing
- A `run()` method that takes a JSON-formatted string and returns a JSON-serializable object

The argument to the `run()` method will be in this form:

```
{  
    "data": <model-specific-data-structure>  
}
```

The object you return from the `run()` method must implement `toJSON() -> string`.

The following example demonstrates how to load a registered scikit-learn model and score it by using NumPy data. This example is based on the model and dependencies of [this tutorial](#).

```
import json  
import numpy as np  
import os  
import pickle  
import joblib  
  
def init():  
    global model  
    # AZUREML_MODEL_DIR is an environment variable created during deployment.  
    # It's the path to the model folder (./azureml-models/$MODEL_NAME/$VERSION).  
    # For multiple models, it points to the folder containing all deployed models (./azureml-models).  
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_mnist_model.pkl')  
    model = joblib.load(model_path)  
  
def run(raw_data):  
    data = np.array(json.loads(raw_data)['data'])  
    # Make prediction.  
    y_hat = model.predict(data)  
    # You can return any data type as long as it's JSON-serializable.  
    return y_hat.tolist()
```

For more advanced examples, including automatic Swagger schema generation and scoring binary data (for example, images), see [Advanced entry script authoring](#).

Deploy as a local web service by using Docker

The easiest way to replicate the environment used by Azure Machine Learning is to deploy a web service by using Docker. With Docker running on your local machine, you will:

1. Connect to the Azure Machine Learning workspace in which your model is registered.
2. Create a `Model` object that represents the model.
3. Create an `Environment` object that contains the dependencies and defines the software environment in which your code will run.
4. Create an `InferenceConfig` object that associates the entry script with the `Environment`.
5. Create a `DeploymentConfiguration` object of the subclass `LocalWebserviceDeploymentConfiguration`.
6. Use `Model.deploy()` to create a `Webservice` object. This method downloads the Docker image and associates it with the `Model`, `InferenceConfig`, and `DeploymentConfiguration`.
7. Activate the `Webservice` by using `Webservice.wait_for_deployment()`.

The following code shows these steps:

```
from azureml.core.webservice import LocalWebservice
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment
from azureml.core import Workspace
from azureml.core.model import Model

ws = Workspace.from_config()
model = Model(ws, 'sklearn_mnist')

myenv = Environment.get(workspace=ws, name="tutorial-env", version="1")
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)

deployment_config = LocalWebservice.deploy_configuration(port=6789)

local_service = Model.deploy(workspace=ws,
                             name='sklearn-mnist-local',
                             models=[model],
                             inference_config=inference_config,
                             deployment_config = deployment_config)

local_service.wait_for_deployment(show_output=True)
print(f"Scoring URI is : {local_service.scoring_uri}")
```

The call to `Model.deploy()` can take a few minutes. After you've initially deployed the web service, it's more efficient to use the `update()` method rather than starting from scratch. See [Update a deployed web service](#).

Test your local deployment

When you run the previous deployment script, it will output the URI to which you can POST data for scoring (for example, `http://localhost:6789/score`). The following sample shows a script that scores sample data by using the `"sklearn-mnist-local"` locally deployed model. The model, if properly trained, infers that `normalized_pixel_values` should be interpreted as a "2".

Download and run your model directly

Using Docker to deploy your model as a web service is the most common option. But you might want to run your code directly by using local Python scripts. You'll need two important components:

- The model itself
 - The dependencies upon which the model relies

You can download the model:

- From the portal, by selecting the **Models** tab, selecting the desired model, and on the **Details** page, selecting **Download**.
 - From the command line, by using `az ml model download`. (See [model download](#).)
 - By using the Python SDK `Model.download()` method. (See [Model class](#).)

An Azure model may be in whatever form your framework uses but is generally one or more serialized Python objects, packaged as a Python pickle file (`.pk1` extension). The contents of the pickle file depend on the machine learning library or technique used to train the model. For example, if you're using the model from the tutorial, you might load the model with:

```
import pickle

with open('sklearn_mnist_model.pkl', 'rb') as f :
    logistic_model = pickle.load(f, encoding='latin1')
```

Dependencies are always tricky to get right, especially with machine learning, where there can often be a dizzying web of specific version requirements. You can re-create an Azure Machine Learning environment on your local machine either as a complete conda environment or as a Docker image by using the `build_local()` method of the `Environment` class:

```
ws = Workspace.from_config()
myenv = Environment.get(workspace=ws, name="tutorial-env", version="1")
myenv.build_local(workspace=ws, useDocker=False) #Creates conda environment.
```

If you set the `build_local()` `useDocker` argument to `True`, the function will create a Docker image rather than a conda environment. If you want more control, you can use the `save_to_directory()` method of `Environment`, which writes `conda_dependencies.yml` and `azureml_environment.json` definition files that you can fine-tune and use as the basis for extension.

The `Environment` class has many other methods for synchronizing environments across your compute hardware, your Azure workspace, and Docker images. For more information, see [Environment class](#).

After you download the model and resolve its dependencies, there are no Azure-defined restrictions on how you perform scoring, fine-tune the model, use transfer learning, and so forth.

Upload a retrained model to Azure Machine Learning

If you have a locally trained or retrained model, you can register it with Azure. After it's registered, you can continue tuning it by using Azure compute or deploy it by using Azure facilities like [Azure Kubernetes Service](#) or [Triton Inference Server \(Preview\)](#).

To be used with the Azure Machine Learning Python SDK, a model must be stored as a serialized Python object in pickle format (a `.pkl` file). It must also implement a `predict(data)` method that returns a JSON-serializable object. For example, you might store a locally trained scikit-learn diabetes model with:

```
import joblib

from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge

dataset_x, dataset_y = load_diabetes(return_X_y=True)

sk_model = Ridge().fit(dataset_x, dataset_y)

joblib.dump(sk_model, "sklearn_regression_model.pkl")
```

To make the model available in Azure, you can then use the `register()` method of the `Model` class:

```
from azureml.core.model import Model

model = Model.register(model_path="sklearn_regression_model.pkl",
                      model_name="sklearn_regression_model",
                      tags={'area': "diabetes", 'type': "regression"},
                      description="Ridge regression model to predict diabetes",
                      workspace=ws)
```

You can then find your newly registered model on the Azure Machine Learning Model tab:

The screenshot shows the Microsoft Azure Machine Learning Studio interface. On the left, there's a sidebar with various options like Notebooks, Automated ML, Designer, Data, Jobs, Components, Pipelines, Environments, Models (which is selected), Endpoints, Compute, Datastores, and Linked Services. The main area is titled 'sklearn_regression_model:1' and has tabs for Details, Versions, Artifacts, Endpoints, Data, Explanations (preview), and Fairness (preview). Under 'Details', there are sections for Attributes (Version 1, ID sklearn_regression_model:1, Date registered 11/19/2020, 11:29:02 AM, Framework Custom, Framework version --, Experiment name --, Run ID --, Created by Jelena Maric), Tags (area : diabetes, type : regression), Properties (No properties), and Description (Ridge regression model to predict diabetes).

For more information on uploading and updating models and environments, see [Register model and deploy locally with advanced usages](#).

Next steps

- For information on using VS Code with Azure Machine Learning, see [Connect to compute instance in Visual Studio Code \(preview\)](#)
- For more information on managing environments, see [Create & use software environments in Azure Machine Learning](#).
- To learn about accessing data from your datastore, see [Connect to storage services on Azure](#).

Deploy a model locally

9/21/2022 • 2 minutes to read • [Edit Online](#)

Learn how to use Azure Machine Learning to deploy a model as a web service on your Azure Machine Learning compute instance. Use compute instances if one of the following conditions is true:

- You need to quickly deploy and validate your model.
- You are testing a model that is under development.

TIP

Deploying a model from a Jupyter Notebook on a compute instance, to a web service on the same VM is a *local deployment*. In this case, the 'local' computer is the compute instance.

NOTE

Azure Machine Learning Endpoints (preview) provide an improved, simpler deployment experience. Endpoints support both real-time and batch inference scenarios. Endpoints provide a unified interface to invoke and manage model deployments across compute types. See [What are Azure Machine Learning endpoints \(preview\)?](#).

Prerequisites

- An Azure Machine Learning workspace with a compute instance running. For more information, see [Quickstart: Get started with Azure Machine Learning](#).

Deploy to the compute instances

An example notebook that demonstrates local deployments is included on your compute instance. Use the following steps to load the notebook and deploy the model as a web service on the VM:

1. From [Azure Machine Learning studio](#), select "Notebooks", and then select how-to-use-azureml/deployment/deploy-to-local/register-model-deploy-local.ipynb under "Sample notebooks". Clone this notebook to your user folder.
2. Find the notebook cloned in step 1, choose or create a Compute Instance to run the notebook.

```

from azureml.core.webservice import LocalWebservice

#this is optional, if not provided we choose random port
deployment_config = LocalWebservice.deploy_configuration(port=6789)

local_service = Model.deploy(ws, "test", [model], inference_config, deployment_config)

local_service.wait_for_deployment()

Requirement already satisfied: pyjwt>=1.0.0 in /opt/miniconda/lib/python3.6/site-packages (from adal>=0.4.5>azureml-model-management-sdk==1.0.1b6.post1->r /var/azureml-app/requirements.txt (line 1)) (1.7.1)

Requirement already satisfied: cffi!=1.11.3,>=1.8 in /opt/miniconda/lib/python3.6/site-packages (from cryptography>=1.1.0->adal>=0.4.5>azureml-model-management-sdk==1.0.1b6.post1->r /var/azureml-app/requirements.txt (line 1)) (1.12.3)
Requirement already satisfied: asn1crypto>=0.21.0 in /opt/miniconda/lib/python3.6/site-packages (from cryptography>=1.1.0->adal>=0.4.5>azureml-model-management-sdk==1.0.1b6.post1->r /var/azureml-app/requirements.txt (line 1)) (0.24.0)
Requirement already satisfied: pycparser in /opt/miniconda/lib/python3.6/site-packages (from cffi!=1.11.3,>=1.8>cryptograph>=1.1.0->adal>=0.4.5>azureml-model-management-sdk==1.0.1b6.post1->r /var/azureml-app/requirements.txt (line 1)) (2.19)
    --> fe3a3661fb0b
Step 7/7 : CMD ["runsvdir","/var/runit"]
--> Running in 3611ce505d62
--> ece2403f94cc
Successfully built ece2403f94cc
Successfully tagged test:latest
Downloading model sklearn_regression_model.pkl:1 to /tmp/azureml_models_test/sklearn_regression_model.pkl:1
Starting Docker container...
Docker container running.
Checking container health...
Local web service is running at http://localhost:6789

```

- The notebook displays the URL and port that the service is running on. For example,

<https://localhost:6789>. You can also run the cell containing

```
print('Local service port: {}'.format(local_service.port))
```

```
print('Local service port: {}'.format(local_service.port))
Local service port: 6789
```

- To test the service from a compute instance, use the https://localhost:<local_service.port> URL. To test from a remote client, get the public URL of the service running on the compute instance. The public URL can be determined use the following formula;

- Notebook VM:

```
https://<vm\_name>-<local\_service\_port>.<azure\_region\_of\_workspace>.notebooks.azureml.net\(score
```

- Compute instance:

```
https://<vm\_name>-<local\_service\_port>.<azure\_region\_of\_workspace>.instances.azureml.net\(score
```

For example,

- Notebook VM: [https://vm-name-6789.northcentralus.notebooks.azureml.net\(score](https://vm-name-6789.northcentralus.notebooks.azureml.net(score)

- Compute instance: [https://vm-name-6789.northcentralus.instances.azureml.net\(score](https://vm-name-6789.northcentralus.instances.azureml.net(score)

Test the service

To submit sample data to the running service, use the following code. Replace the value of `service_url` with the URL of from the previous step:

NOTE

When authenticating to a deployment on the compute instance, the authentication is made using Azure Active Directory. The call to `interactive_auth.get_authentication_header()` in the example code authenticates you using AAD, and returns a header that can then be used to authenticate to the service on the compute instance. For more information, see [Set up authentication for Azure Machine Learning resources and workflows](#).

When authenticating to a deployment on Azure Kubernetes Service or Azure Container Instances, a different authentication method is used. For more information on, see [Configure authentication for Azure Machine models deployed as web services](#).

```
import requests
import json
from azureml.core.authentication import InteractiveLoginAuthentication

# Get a token to authenticate to the compute instance from remote
interactive_auth = InteractiveLoginAuthentication()
auth_header = interactive_auth.get_authentication_header()

# Create and submit a request using the auth header
headers = auth_header
# Add content type header
headers.update({'Content-Type':'application/json'})

# Sample data to send to the service
test_sample = json.dumps({'data': [
    [1,2,3,4,5,6,7,8,9,10],
    [10,9,8,7,6,5,4,3,2,1]
]})
test_sample = bytes(test_sample,encoding = 'utf8')

# Replace with the URL for your compute instance, as determined from the previous section
service_url = "https://vm-name-6789.northcentralus.notebooks.azureml.net/score"
# for a compute instance, the url would be https://vm-name-6789.northcentralus.instances.azureml.net/score
resp = requests.post(service_url, test_sample, headers=headers)
print("prediction:", resp.text)
```

Next steps

- [How to deploy a model using a custom Docker image](#)
- [Deployment troubleshooting](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Consume a ML Model deployed as a web service](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)

Deploy a deep learning model for inference with GPU

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article teaches you how to use Azure Machine Learning to deploy a GPU-enabled model as a web service. The information in this article is based on deploying a model on Azure Kubernetes Service (AKS). The AKS cluster provides a GPU resource that is used by the model for inference.

Inference, or model scoring, is the phase where the deployed model is used to make predictions. Using GPUs instead of CPUs offers performance advantages on highly parallelizable computation.

NOTE

Azure Machine Learning Endpoints (preview) provide an improved, simpler deployment experience. Endpoints support both real-time and batch inference scenarios. Endpoints provide a unified interface to invoke and manage model deployments across compute types. See [What are Azure Machine Learning endpoints \(preview\)?](#).

IMPORTANT

When using the Azure ML SDK v1, GPU inference is only supported on Azure Kubernetes Service. When using the Azure ML SDK v2 or CLI v2, you can use an online endpoint for GPU inference. For more information, see [Deploy and score a machine learning model with an online endpoint](#).

For inference using a **machine learning pipeline**, GPUs are only supported on Azure Machine Learning Compute. For more information on using ML pipelines, see [Tutorial: Build an Azure Machine Learning pipeline for batch scoring](#).

TIP

Although the code snippets in this article use a TensorFlow model, you can apply the information to any machine learning framework that supports GPUs.

NOTE

The information in this article builds on the information in the [How to deploy to Azure Kubernetes Service](#) article. Where that article generally covers deployment to AKS, this article covers GPU specific deployment.

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create an Azure Machine Learning workspace](#).
- A Python development environment with the Azure Machine Learning SDK installed. For more information, see [Azure Machine Learning SDK](#).

- A registered model that uses a GPU.
 - To learn how to register models, see [Deploy Models](#).
 - To create and register the Tensorflow model used to create this document, see [How to Train a TensorFlow Model](#).
- A general understanding of [How and where to deploy models](#).

Connect to your workspace

To connect to an existing workspace, use the following code:

IMPORTANT

This code snippet expects the workspace configuration to be saved in the current directory or its parent. For more information on creating a workspace, see [Create workspace resources](#). For more information on saving the configuration to file, see [Create a workspace configuration file](#).

```
from azureml.core import Workspace

# Connect to the workspace
ws = Workspace.from_config()
```

Create a Kubernetes cluster with GPUs

Azure Kubernetes Service provides many different GPU options. You can use any of them for model inference. See [the list of N-series VMs](#) for a full breakdown of capabilities and costs.

The following code demonstrates how to create a new AKS cluster for your workspace:

```
from azureml.core.compute import ComputeTarget, AksCompute
from azureml.exceptions import ComputeTargetException

# Choose a name for your cluster
aks_name = "aks-gpu"

# Check to see if the cluster already exists
try:
    aks_target = ComputeTarget(workspace=ws, name=aks_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    # Provision AKS cluster with GPU machine
    prov_config = AksCompute.provisioning_configuration(vm_size="Standard_NC6")

    # Create the cluster
    aks_target = ComputeTarget.create(
        workspace=ws, name=aks_name, provisioning_configuration=prov_config
    )

    aks_target.wait_for_completion(show_output=True)
```

IMPORTANT

Azure will bill you as long as the AKS cluster exists. Make sure to delete your AKS cluster when you're done with it.

For more information on using AKS with Azure Machine Learning, see [How to deploy to Azure Kubernetes Service](#).

Write the entry script

The entry script receives data submitted to the web service, passes it to the model, and returns the scoring results. The following script loads the Tensorflow model on startup, and then uses the model to score data.

TIP

The entry script is specific to your model. For example, the script must know the framework to use with your model, data formats, etc.

```
import json
import numpy as np
import os
import tensorflow as tf

from azureml.core.model import Model


def init():
    global X, output, sess
    tf.reset_default_graph()
    model_root = os.getenv('AZUREML_MODEL_DIR')
    # the name of the folder in which to look for tensorflow model files
    tf_model_folder = 'model'
    saver = tf.train.import_meta_graph(
        os.path.join(model_root, tf_model_folder, 'mnist-tf.model.meta'))
    X = tf.get_default_graph().get_tensor_by_name("network/X:0")
    output = tf.get_default_graph().get_tensor_by_name("network/output/MatMul:0")

    sess = tf.Session()
    saver.restore(sess, os.path.join(model_root, tf_model_folder, 'mnist-tf.model'))


def run(raw_data):
    data = np.array(json.loads(raw_data)['data'])
    # make prediction
    out = output.eval(session=sess, feed_dict={X: data})
    y_hat = np.argmax(out, axis=1)
    return y_hat.tolist()
```

This file is named `score.py`. For more information on entry scripts, see [How and where to deploy](#).

Define the conda environment

The conda environment file specifies the dependencies for the service. It includes dependencies required by both the model and the entry script. Please note that you must indicate `azureml-defaults` with version `>= 1.0.45` as a pip dependency, because it contains the functionality needed to host the model as a web service. The following YAML defines the environment for a Tensorflow model. It specifies `tensorflow-gpu`, which will make use of the GPU used in this deployment:

```
name: project_environment
dependencies:
    # The Python interpreter version.
    # Currently Azure ML only supports 3.5.2 and later.
    - python=3.6.2

    - pip:
        # You must list azureml-defaults as a pip dependency
        - azureml-defaults>=1.0.45
        - numpy
        - tensorflow-gpu=1.12
channels:
    - conda-forge
```

For this example, the file is saved as `myenv.yml`.

Define the deployment configuration

IMPORTANT

AKS does not allow pods to share GPUs, you can have only as many replicas of a GPU-enabled web service as there are GPUs in the cluster.

The deployment configuration defines the Azure Kubernetes Service environment used to run the web service:

```
from azureml.core.webservice import AksWebservice

gpu_aks_config = AksWebservice.deploy_configuration(autoscale_enabled=False,
                                                    num_replicas=3,
                                                    cpu_cores=2,
                                                    memory_gb=4)
```

For more information, see the reference documentation for [AksService.deploy_configuration](#).

Define the inference configuration

The inference configuration points to the entry script and an environment object, which uses a docker image with GPU support. Please note that the YAML file used for environment definition must list `azureml-defaults` with version $\geq 1.0.45$ as a pip dependency, because it contains the functionality needed to host the model as a web service.

```
from azureml.core.model import InferenceConfig
from azureml.core.environment import Environment, DEFAULT_GPU_IMAGE

myenv = Environment.from_conda_specification(name="myenv", file_path="myenv.yml")
myenv.docker.base_image = DEFAULT_GPU_IMAGE
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
```

For more information on environments, see [Create and manage environments for training and deployment](#). For more information, see the reference documentation for [InferenceConfig](#).

Deploy the model

Deploy the model to your AKS cluster and wait for it to create your service.

```
from azureml.core.model import Model

# Name of the web service that is deployed
aks_service_name = 'aks-dnn-mnist'
# Get the registered model
model = Model(ws, "tf-dnn-mnist")
# Deploy the model
aks_service = Model.deploy(ws,
    models=[model],
    inference_config=inference_config,
    deployment_config=gpu_aks_config,
    deployment_target=aks_target,
    name=aks_service_name)

aks_service.wait_for_deployment(show_output=True)
print(aks_service.state)
```

For more information, see the reference documentation for [Model](#).

Issue a sample query to your service

Send a test query to the deployed model. When you send a jpeg image to the model, it scores the image. The following code sample downloads test data and then selects a random test image to send to the service.

```

# Used to test your webservice
import os
import urllib
import gzip
import numpy as np
import struct
import requests

# load compressed MNIST gz files and return numpy arrays
def load_data(filename, label=False):
    with gzip.open(filename) as gz:
        struct.unpack('I', gz.read(4))
        n_items = struct.unpack('>I', gz.read(4))
        if not label:
            n_rows = struct.unpack('>I', gz.read(4))[0]
            n_cols = struct.unpack('>I', gz.read(4))[0]
            res = np.frombuffer(gz.read(n_items[0] * n_rows * n_cols), dtype=np.uint8)
            res = res.reshape(n_items[0], n_rows * n_cols)
        else:
            res = np.frombuffer(gz.read(n_items[0]), dtype=np.uint8)
            res = res.reshape(n_items[0], 1)
    return res

# one-hot encode a 1-D array
def one_hot_encode(array, num_of_classes):
    return np.eye(num_of_classes)[array.reshape(-1)]

# Download test data
os.makedirs('./data/mnist', exist_ok=True)
urllib.request.urlretrieve('http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz',
filename='./data/mnist/test-images.gz')
urllib.request.urlretrieve('http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz',
filename='./data/mnist/test-labels.gz')

# Load test data from model training
X_test = load_data('./data/mnist/test-images.gz', False) / 255.0
y_test = load_data('./data/mnist/test-labels.gz', True).reshape(-1)

# send a random row from the test set to score
random_index = np.random.randint(0, len(X_test)-1)
input_data = "{\"data\": [" + str(list(X_test[random_index])) + "]}""

api_key = aks_service.get_keys()[0]
headers = {'Content-Type': 'application/json',
           'Authorization': ('Bearer ' + api_key)}
resp = requests.post(aks_service.scoring_uri, input_data, headers=headers)

print("POST to url", aks_service.scoring_uri)
print("label:", y_test[random_index])
print("prediction:", resp.text)

```

For more information on creating a client application, see [Create client to consume deployed web service](#).

Clean up the resources

If you created the AKS cluster specifically for this example, delete your resources after you're done.

IMPORTANT

Azure bills you based on how long the AKS cluster is deployed. Make sure to clean it up after you are done with it.

```
aks_service.delete()  
aks_target.delete()
```

Next steps

- [Deploy model on FPGA](#)
- [Deploy model with ONNX](#)
- [Train TensorFlow DNN Models](#)

Deploy a model for use with Cognitive Search

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

This article teaches you how to use Azure Machine Learning to deploy a model for use with [Azure Cognitive Search](#).

Cognitive Search performs content processing over heterogenous content, to make it queryable by humans or applications. This process can be enhanced by using a model deployed from Azure Machine Learning.

Azure Machine Learning can deploy a trained model as a web service. The web service is then embedded in a Cognitive Search *skill*, which becomes part of the processing pipeline.

IMPORTANT

The information in this article is specific to the deployment of the model. It provides information on the supported deployment configurations that allow the model to be used by Cognitive Search.

For information on how to configure Cognitive Search to use the deployed model, see the [Build and deploy a custom skill with Azure Machine Learning](#) tutorial.

For the sample that the tutorial is based on, see <https://github.com/Azure-Samples/azure-search-python-samples/tree/master/AzureML-Custom-Skill>.

When deploying a model for use with Azure Cognitive Search, the deployment must meet the following requirements:

- Use Azure Kubernetes Service to host the model for inference.
- Enable transport layer security (TLS) for the Azure Kubernetes Service. TLS is used to secure HTTPS communications between Cognitive Search and the deployed model.
- The entry script must use the `inference_schema` package to generate an OpenAPI (Swagger) schema for the service.
- The entry script must also accept JSON data as input, and generate JSON as output.

Prerequisites

- An Azure Machine Learning workspace. For more information, see [Create workspace resources](#).
- A Python development environment with the Azure Machine Learning SDK installed. For more information, see [Azure Machine Learning SDK](#).
- A registered model. If you do not have a model, use the example notebook at <https://github.com/Azure-Samples/azure-search-python-samples/tree/master/AzureML-Custom-Skill>.
- A general understanding of [How and where to deploy models](#).

Connect to your workspace

An Azure Machine Learning workspace provides a centralized place to work with all the artifacts you create when you use Azure Machine Learning. The workspace keeps a history of all training jobs, including logs, metrics, output, and a snapshot of your scripts.

To connect to an existing workspace, use the following code:

IMPORTANT

This code snippet expects the workspace configuration to be saved in the current directory or its parent. For more information, see [Create and manage Azure Machine Learning workspaces](#). For more information on saving the configuration to file, see [Create a workspace configuration file](#).

```
from azureml.core import Workspace

try:
    # Load the workspace configuration from local cached info
    ws = Workspace.from_config()
    print(ws.name, ws.location, ws.resource_group, ws.location, sep='\t')
    print('Library configuration succeeded')
except:
    print('Workspace not found')
```

Create a Kubernetes cluster

Time estimate: Approximately 20 minutes.

A Kubernetes cluster is a set of virtual machine instances (called nodes) that are used for running containerized applications.

When you deploy a model from Azure Machine Learning to Azure Kubernetes Service, the model and all the assets needed to host it as a web service are packaged into a Docker container. This container is then deployed onto the cluster.

The following code demonstrates how to create a new Azure Kubernetes Service (AKS) cluster for your workspace:

TIP

You can also attach an existing Azure Kubernetes Service to your Azure Machine Learning workspace. For more information, see [How to deploy models to Azure Kubernetes Service](#).

IMPORTANT

Notice that the code uses the `enable_ssl()` method to enable transport layer security (TLS) for the cluster. This is required when you plan on using the deployed model from Cognitive Services.

```

from azureml.core.compute import AksCompute, ComputeTarget
# Create or attach to an AKS inferencing cluster

# Create the provisioning configuration with defaults
prov_config = AksCompute.provisioning_configuration()

# Enable TLS (sometimes called SSL) communications
# Leaf domain label generates a name using the formula
# "<leaf-domain-label>#####.<azure-region>.cloudapp.azure.com"
# where "#####" is a random series of characters
prov_config.enable_ssl(leaf_domain_label = "contoso")

cluster_name = 'amlskills'
# Try to use an existing compute target by that name.
# If one doesn't exist, create one.
try:

    aks_target = ComputeTarget(ws, cluster_name)
    print("Attaching to existing cluster")
except Exception as e:
    print("Creating new cluster")
    aks_target = ComputeTarget.create(workspace = ws,
                                       name = cluster_name,
                                       provisioning_configuration = prov_config)
# Wait for the create process to complete
aks_target.wait_for_completion(show_output = True)

```

IMPORTANT

Azure will bill you as long as the AKS cluster exists. Make sure to delete your AKS cluster when you're done with it.

For more information on using AKS with Azure Machine Learning, see [How to deploy to Azure Kubernetes Service](#).

Write the entry script

The entry script receives data submitted to the web service, passes it to the model, and returns the scoring results. The following script loads the model on startup, and then uses the model to score data. This file is sometimes called `score.py`.

TIP

The entry script is specific to your model. For example, the script must know the framework to use with your model, data formats, etc.

IMPORTANT

When you plan on using the deployed model from Azure Cognitive Services, you must use the `inference_schema` package to enable schema generation for the deployment. This package provides decorators that allow you to define the input and output data format for the web service that performs inference using the model.

```

from azureml.core.model import Model
from nlp_architect.models.absa.inference.inference import SentimentInference
from spacy.cli.download import download as spacy_download
import traceback
import json
# Inference schema for schema discovery
from inference_schema.schema_decorators import input_schema, output_schema
from inference_schema.parameter_types.numpy_parameter_type import NumpyParameterType
from inference_schema.parameter_types.standard_py_parameter_type import StandardPythonParameterType

def init():
    """
    Set up the ABSA model for Inference
    """
    global SentInference
    spacy_download('en')
    aspect_lex = Model.get_model_path('hotel_aspect_lex')
    opinion_lex = Model.get_model_path('hotel_opinion_lex')
    SentInference = SentimentInference(aspect_lex, opinion_lex)

# Use inference schema decorators and sample input/output to
# build the OpenAPI (Swagger) schema for the deployment
standard_sample_input = {'text': 'a sample input record containing some text' }
standard_sample_output = {"sentiment": {"sentence": "This place makes false booking prices, when you get there, they say they do not have the reservation for that day.",
                                       "terms": [{"text": "hotels", "type": "AS", "polarity": "POS",
                                                  "score": 1.0, "start": 300, "len": 6},
                                                 {"text": "nice", "type": "OP", "polarity": "POS", "score": 1.0, "start": 295, "len": 4}]}}
@input_schema('raw_data', StandardPythonParameterType(standard_sample_input))
@output_schema(StandardPythonParameterType(standard_sample_output))
def run(raw_data):
    try:
        # Get the value of the 'text' field from the JSON input and perform inference
        input_txt = raw_data["text"]
        doc = SentInference.run(doc=input_txt)
        if doc is None:
            return None
        sentences = doc._sentences
        result = {"sentence": doc._doc_text}
        terms = []
        for sentence in sentences:
            for event in sentence._events:
                for x in event:
                    term = {"text": x._text, "type": x._type.value, "polarity": x._polarity.value, "score": x._score, "start": x._start, "len": x._len}
                    terms.append(term)
        result["terms"] = terms
        print("Success!")
        # Return the results to the client as a JSON document
        return {"sentiment": result}
    except Exception as e:
        result = str(e)
        # return error message back to the client
        print("Failure!")
        print(traceback.format_exc())
        return json.dumps({"error": result, "tb": traceback.format_exc()})

```

For more information on entry scripts, see [How and where to deploy](#).

Define the software environment

The environment class is used to define the Python dependencies for the service. It includes dependencies required by both the model and the entry script. In this example, it installs packages from the regular pypi index, as well as from a GitHub repo.

```
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core import Environment

conda = None
pip = ["azureml-defaults", "azureml-monitoring",
       "git+https://github.com/NervanaSystems/nlp-architect.git@absa", 'nlp-architect', 'inference-schema',
       "spacy==2.0.18"]

conda_deps = CondaDependencies.create(conda_packages=None, pip_packages=pip)

myenv = Environment(name='myenv')
myenv.python.conda_dependencies = conda_deps
```

For more information on environments, see [Create and manage environments for training and deployment](#).

Define the deployment configuration

The deployment configuration defines the Azure Kubernetes Service hosting environment used to run the web service.

TIP

If you aren't sure about the memory, CPU, or GPU needs of your deployment, you can use profiling to learn these. For more information, see [How and where to deploy a model](#).

```
from azureml.core.model import Model
from azureml.core.webservice import Webservice
from azureml.core.image import ContainerImage
from azureml.core.webservice import AksWebservice, Webservice

# If deploying to a cluster configured for dev/test, ensure that it was created with enough
# cores and memory to handle this deployment configuration. Note that memory is also used by
# things such as dependencies and AzureML components.

aks_config = AksWebservice.deploy_configuration(autoscale_enabled=True,
                                                autoscale_min_replicas=1,
                                                autoscale_max_replicas=3,
                                                autoscale_refresh_seconds=10,
                                                autoscale_target_utilization=70,
                                                auth_enabled=True,
                                                cpu_cores=1, memory_gb=2,
                                                scoring_timeout_ms=5000,
                                                replica_max_concurrent_requests=2,
                                                max_request_wait_time=5000)
```

For more information, see the reference documentation for [AksService.deploy_configuration](#).

Define the inference configuration

The inference configuration points to the entry script and the environment object:

```
from azureml.core.model import InferenceConfig
inf_config = InferenceConfig(entry_script='score.py', environment=myenv)
```

For more information, see the reference documentation for [InferenceConfig](#).

Deploy the model

Deploy the model to your AKS cluster and wait for it to create your service. In this example, two registered models are loaded from the registry and deployed to AKS. After deployment, the `score.py` file in the deployment loads these models and uses them to perform inference.

```
from azureml.core.webservice import AksWebservice, Webservice

c_aspect_lex = Model(ws, 'hotel_aspect_lex')
c_opinion_lex = Model(ws, 'hotel_opinion_lex')
service_name = "hotel-absa-v2"

aks_service = Model.deploy(workspace=ws,
                           name=service_name,
                           models=[c_aspect_lex, c_opinion_lex],
                           inference_config=inference_config,
                           deployment_config=aks_config,
                           deployment_target=aks_target,
                           overwrite=True)

aks_service.wait_for_deployment(show_output = True)
print(aks_service.state)
```

For more information, see the reference documentation for [Model](#).

Issue a sample query to your service

The following example uses the deployment information stored in the `aks_service` variable by the previous code section. It uses this variable to retrieve the scoring URL and authentication token needed to communicate with the service:

```
import requests
import json

primary, secondary = aks_service.get_keys()

# Test data
input_data = '{"raw_data": {"text": "This is a nice place for a relaxing evening out with friends. The owners seem pretty nice, too. I have been there a few times including last night. Recommend."}}'

# Since authentication was enabled for the deployment, set the authorization header.
headers = {'Content-Type':'application/json', 'Authorization':('Bearer ' + primary)}

# Send the request and display the results
resp = requests.post(aks_service.scoring_uri, input_data, headers=headers)
print(resp.text)
```

The result returned from the service is similar to the following JSON:

```
{"sentiment": {"sentence": "This is a nice place for a relaxing evening out with friends. The owners seem pretty nice, too. I have been there a few times including last night. Recommend.", "terms": [{"text": "place", "type": "AS", "polarity": "POS", "score": 1.0, "start": 15, "len": 5}, {"text": "nice", "type": "OP", "polarity": "POS", "score": 1.0, "start": 10, "len": 4}]}}
```

Connect to Cognitive Search

For information on using this model from Cognitive Search, see the [Build and deploy a custom skill with Azure Machine Learning](#) tutorial.

Clean up the resources

If you created the AKS cluster specifically for this example, delete your resources after you're done testing it with Cognitive Search.

IMPORTANT

Azure bills you based on how long the AKS cluster is deployed. Make sure to clean it up after you are done with it.

```
aks_service.delete()  
aks_target.delete()
```

Next steps

- [Build and deploy a custom skill with Azure Machine Learning](#)

Deploy ML models to field-programmable gate arrays (FPGAs) with Azure Machine Learning

9/21/2022 • 9 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn about FPGAs and how to deploy your ML models to an Azure FPGA using the [hardware-accelerated models Python package](#) from [Azure Machine Learning](#).

What are FPGAs?

FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects. The interconnects allow these blocks to be configured in various ways after manufacturing. Compared to other chips, FPGAs provide a combination of programmability and performance.

FPGAs make it possible to achieve low latency for real-time inference (or model scoring) requests.

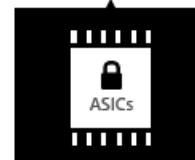
Asynchronous requests (batching) aren't needed. Batching can cause latency, because more data needs to be processed. Implementations of neural processing units don't require batching; therefore the latency can be many times lower, compared to CPU and GPU processors.

You can reconfigure FPGAs for different types of machine learning models. This flexibility makes it easier to accelerate the applications based on the most optimal numerical precision and memory model being used. Because FPGAs are reconfigurable, you can stay current with the requirements of rapidly changing AI algorithms.

Silicon alternatives

TRAINING
CPUs and GPUs, limited FPGAs,
ASICs under investigation

EVALUATION
CPUs and FPGAs,
ASICs under investigation



FLEXIBILITY

EFFICIENCY

PROCESSOR	ABBREVIATION	DESCRIPTION
Application-specific integrated circuits	ASICs	Custom circuits, such as Google's Tensor Processor Units (TPU), provide the highest efficiency. They can't be reconfigured as your needs change.

PROCESSOR	ABBREVIATION	DESCRIPTION
Field-programmable gate arrays	FPGAs	FPGAs, such as those available on Azure, provide performance close to ASICs. They're also flexible and reconfigurable over time, to implement new logic.
Graphics processing units	GPUs	A popular choice for AI computations. GPUs offer parallel processing capabilities, making it faster at image rendering than CPUs.
Central processing units	CPUs	General-purpose processors, the performance of which isn't ideal for graphics and video processing.

FPGA support in Azure

Microsoft Azure is the world's largest cloud investment in FPGAs. Microsoft uses FPGAs for deep neural networks (DNN) evaluation, Bing search ranking, and software defined networking (SDN) acceleration to reduce latency, while freeing CPUs for other tasks.

FPGAs on Azure are based on Intel's FPGA devices, which data scientists and developers use to accelerate real-time AI calculations. This FPGA-enabled architecture offers performance, flexibility, and scale, and is available on Azure.

Azure FPGAs are integrated with Azure Machine Learning. Azure can parallelize pre-trained DNN across FPGAs to scale out your service. The DNNs can be pre-trained, as a deep featurizer for transfer learning, or fine-tuned with updated weights.

SCENARIOS & CONFIGURATIONS ON AZURE	SUPPORTED DNN MODELS	REGIONAL SUPPORT
<ul style="list-style-type: none"> + Image classification and recognition scenarios + TensorFlow deployment (requires Tensorflow 1.x) + Intel FPGA hardware 	<ul style="list-style-type: none"> - ResNet 50 - ResNet 152 - DenseNet-121 - VGG-16 - SSD-VGG 	<ul style="list-style-type: none"> - East US - Southeast Asia - West Europe - West US 2

To optimize latency and throughput, your client sending data to the FPGA model should be in one of the regions above (the one you deployed the model to).

The **PBS Family of Azure VMs** contains Intel Arria 10 FPGAs. It will show as "Standard PBS Family vCPUs" when you check your Azure quota allocation. The PB6 VM has six vCPUs and one FPGA. PB6 VM is automatically provisioned by Azure Machine Learning during model deployment to an FPGA. It's only used with Azure ML, and it can't run arbitrary bitstreams. For example, you won't be able to flash the FPGA with bitstreams to do encryption, encoding, etc.

Deploy models on FPGAs

You can deploy a model as a web service on FPGAs with [Azure Machine Learning Hardware Accelerated Models](#). Using FPGAs provides ultra-low latency inference, even with a single batch size.

In this example, you create a TensorFlow graph to preprocess the input image, make it a featurizer using ResNet 50 on an FPGA, and then run the features through a classifier trained on the ImageNet data set. Then, the model is deployed to an AKS cluster.

Prerequisites

- An Azure subscription. If you don't have one, create a [pay-as-you-go](#) account (free Azure accounts aren't eligible for FPGA quota).
- An Azure Machine Learning workspace and the Azure Machine Learning SDK for Python installed, as described in [Create a workspace](#).
- The hardware-accelerated models package: `pip install --upgrade azureml-accel-models[cpu]`
- The [Azure CLI](#)
- FPGA quota. Submit a [request for quota](#), or run this CLI command to check quota:

```
az vm list-usage --location "eastus" -o table --query "[?localName=='Standard PBS Family vCPUs']"
```

Make sure you have at least 6 vCPUs under the **CurrentValue** returned.

Define the TensorFlow model

Begin by using the [Azure Machine Learning SDK for Python](#) to create a service definition. A service definition is a file describing a pipeline of graphs (input, featurizer, and classifier) based on TensorFlow. The deployment command compresses the definition and graphs into a ZIP file, and uploads the ZIP to Azure Blob storage. The DNN is already deployed to run on the FPGA.

1. Load Azure Machine Learning workspace

```
import os
import tensorflow as tf

from azureml.core import Workspace

ws = Workspace.from_config()
print(ws.name, ws.resource_group, ws.location, ws.subscription_id, sep='\n')
```

2. Preprocess image.

The input to the web service is a JPEG image. The first step is to decode the JPEG image and preprocess it. The JPEG images are treated as strings and the result are tensors that will be the input to the ResNet 50 model.

```
# Input images as a two-dimensional tensor containing an arbitrary number of images represented as
# strings
import azureml.accel.models.utils as utils
tf.reset_default_graph()

in_images = tf.placeholder(tf.string)
image_tensors = utils.preprocess_array(in_images)
print(image_tensors.shape)
```

3. Load featurizer.

Initialize the model and download a TensorFlow checkpoint of the quantized version of ResNet50 to be used as a featurizer. Replace "QuantizedResnet50" in the code snippet to import other deep neural networks:

- QuantizedResnet152
- QuantizedVgg16
- Densenet121

```
from azureml.accel.models import QuantizedResnet50
save_path = os.path.expanduser('~/models')
model_graph = QuantizedResnet50(save_path, is_frozen=True)
feature_tensor = model_graph.import_graph_def(image_tensors)
print(model_graph.version)
print(feature_tensor.name)
print(feature_tensor.shape)
```

4. Add a classifier. This classifier was trained on the ImageNet data set.

```
classifier_output = model_graph.get_default_classifier(feature_tensor)
print(classifier_output)
```

5. Save the model. Now that the preprocessor, ResNet 50 featurizer, and the classifier have been loaded, save the graph and associated variables as a model.

```
model_name = "resnet50"
model_save_path = os.path.join(save_path, model_name)
print("Saving model in {}".format(model_save_path))

with tf.Session() as sess:
    model_graph.restore_weights(sess)
    tf.saved_model.simple_save(sess, model_save_path,
                               inputs={'images': in_images},
                               outputs={'output_alias': classifier_output})
```

6. Save input and output tensors as you will use them for model conversion and inference requests.

```
input_tensors = in_images.name
output_tensors = classifier_output.name

print(input_tensors)
print(output_tensors)
```

The following models are listed with their classifier output tensors for inference if you used the default classifier.

- Resnet50, QuantizedResnet50

```
output_tensors = "classifier_1/resnet_v1_50/predictions/Softmax:0"
```

- Resnet152, QuantizedResnet152

```
output_tensors = "classifier/resnet_v1_152/predictions/Softmax:0"
```

- Densenet121, QuantizedDensenet121

```
output_tensors = "classifier/densenet121/predictions/Softmax:0"
```

- Vgg16, QuantizedVgg16

```
output_tensors = "classifier/vgg_16/fc8/squeezed:0"
```

- SsdVgg, QuantizedSsdVgg

```
output_tensors = ['ssd_300_vgg/block4_box/Reshape_1:0', 'ssd_300_vgg/block7_box/Reshape_1:0',
'ssd_300_vgg/block8_box/Reshape_1:0', 'ssd_300_vgg/block9_box/Reshape_1:0',
:ssd_300_vgg/block10_box/Reshape_1:0', 'ssd_300_vgg/block11_box/Reshape_1:0',
:ssd_300_vgg/block4_box/Reshape:0', 'ssd_300_vgg/block7_box/Reshape:0',
:ssd_300_vgg/block8_box/Reshape:0', 'ssd_300_vgg/block9_box/Reshape:0',
:ssd_300_vgg/block10_box/Reshape:0', 'ssd_300_vgg/block11_box/Reshape:0']
```

Convert the model to the Open Neural Network Exchange format (ONNX)

Before you can deploy to FPGAs, convert the model to the [ONNX](#) format.

1. [Register](#) the model by using the SDK with the ZIP file in Azure Blob storage. Adding tags and other metadata about the model helps you keep track of your trained models.

```
from azureml.core.model import Model

registered_model = Model.register(workspace=ws,
                                   model_path=model_save_path,
                                   model_name=model_name)

print("Successfully registered: ", registered_model.name,
      registered_model.description, registered_model.version, sep='\t')
```

If you've already registered a model and want to load it, you may retrieve it.

```
from azureml.core.model import Model
model_name = "resnet50"
# By default, the latest version is retrieved. You can specify the version, i.e. version=1
registered_model = Model(ws, name="resnet50")
print(registered_model.name, registered_model.description,
      registered_model.version, sep='\t')
```

2. Convert the TensorFlow graph to the ONNX format. You must provide the names of the input and output tensors, so your client can use them when you consume the web service.

```
from azureml.accel import AccelOnnxConverter

convert_request = AccelOnnxConverter.convert_tf_model(
    ws, registered_model, input_tensors, output_tensors)

# If it fails, you can run wait_for_completion again with show_output=True.
convert_request.wait_for_completion(show_output=False)

# If the above call succeeded, get the converted model
converted_model = convert_request.result
print("\nSuccessfully converted: ", converted_model.name, converted_model.url,
      converted_model.version,
      converted_model.id, converted_model.created_time, '\n')
```

Containerize and deploy the model

Next, create a Docker image from the converted model and all dependencies. This Docker image can then be deployed and instantiated. Supported deployment targets include Azure Kubernetes Service (AKS) in the cloud or an edge device such as [Azure Stack Edge](#). You can also add tags and descriptions for your registered Docker image.

```

from azureml.core.image import Image
from azureml.accel import AccelContainerImage

image_config = AccelContainerImage.image_configuration()
# Image name must be lowercase
image_name = "{}-image".format(model_name)

image = Image.create(name=image_name,
                     models=[converted_model],
                     image_config=image_config,
                     workspace=ws)
image.wait_for_creation(show_output=False)

```

List the images by tag and get the detailed logs for any debugging.

```

for i in Image.list(workspace=ws):
    print('{}(v.{} [{}]) stored at {} with build log {}'.format(
        i.name, i.version, i.creation_state, i.image_location, i.image_build_log_uri))

```

Deploy to an Azure Kubernetes Service Cluster

- To deploy your model as a high-scale production web service, use AKS. You can create a new one using the Azure Machine Learning SDK, CLI, or [Azure Machine Learning studio](#).

```

from azureml.core.compute import AksCompute, ComputeTarget

# Specify the Standard_PB6s Azure VM and location. Values for location may be "eastus",
#"southeastasia", "westeurope", or "westus2". If no value is specified, the default is "eastus".
prov_config = AksCompute.provisioning_configuration(vm_size = "Standard_PB6s",
                                                    agent_count = 1,
                                                    location = "eastus")

aks_name = 'my-aks-cluster'
# Create the cluster
aks_target = ComputeTarget.create(workspace=ws,
                                    name=aks_name,
                                    provisioning_configuration=prov_config)

```

The AKS deployment may take around 15 minutes. Check to see if the deployment succeeded.

```

aks_target.wait_for_completion(show_output=True)
print(aks_target.provisioning_state)
print(aks_target.provisioning_errors)

```

- Deploy the container to the AKS cluster.

```

from azureml.core.webservice import Webservice, AksWebservice

# For this deployment, set the web service configuration without enabling auto-scaling or
# authentication for testing
aks_config = AksWebservice.deploy_configuration(autoscale_enabled=False,
                                                num_replicas=1,
                                                auth_enabled=False)

aks_service_name = 'my-aks-service'

aks_service = Webservice.deploy_from_image(workspace=ws,
                                            name=aks_service_name,
                                            image=image,
                                            deployment_config=aks_config,
                                            deployment_target=aks_target)
aks_service.wait_for_deployment(show_output=True)

```

Deploy to a local edge server

All [Azure Stack Edge devices](#) contain an FPGA for running the model. Only one model can be running on the FPGA at one time. To run a different model, just deploy a new container. Instructions and sample code can be found in [this Azure Sample](#).

Consume the deployed model

Lastly, use the sample client to call into the Docker image to get predictions from the model. Sample client code is available:

- [Python](#)
- [C#](#)

The Docker image supports gRPC and the TensorFlow Serving "predict" API.

You can also download a sample client for TensorFlow Serving.

```

# Using the grpc client in Azure ML Accelerated Models SDK package
from azureml.accel import PredictionClient

address = aks_service.scoring_uri
ssl_enabled = address.startswith("https")
address = address[address.find('/')+2:].strip('/')
port = 443 if ssl_enabled else 80

# Initialize Azure ML Accelerated Models client
client = PredictionClient(address=address,
                           port=port,
                           use_ssl=ssl_enabled,
                           service_name=aks_service.name)

```

Since this classifier was trained on the ImageNet data set, map the classes to human-readable labels.

```

import requests
classes_entries = requests.get(
    "https://raw.githubusercontent.com/Lasagne/Recipes/master/examples/resnet50/imagenet_classes.txt").text.splitlines()

# Score image with input and output tensor names
results = client.score_file(path='./snowleopardgaze.jpg',
                             input_name=input_tensors,
                             outputs=output_tensors)

# map results [class_id] => [confidence]
results = enumerate(results)
# sort results by confidence
sorted_results = sorted(results, key=lambda x: x[1], reverse=True)
# print top 5 results
for top in sorted_results[:5]:
    print(classes_entries[top[0]], 'confidence:', top[1])

```

Clean up resources

To avoid unnecessary costs, clean up your resources **in this order**: web service, then image, and then the model.

```

aks_service.delete()
aks_target.delete()
image.delete()
registered_model.delete()
converted_model.delete()

```

Next steps

- Learn how to [secure your web services](#) document.
- Learn about FPGA and [Azure Machine Learning pricing and costs](#).
- [Hyperscale hardware: ML at scale on top of Azure + FPGA: Build 2018 \(video\)](#)
- [Project Brainwave for real-time AI](#)
- [Automated optical inspection system](#)

Profile your model to determine resource utilization

9/21/2022 • 3 minutes to read • [Edit Online](#)

APPLIES TO: Azure CLI ml extension v1 Python SDK azureml v1

This article shows how to profile a machine learning model to determine how much CPU and memory you will need to allocate for the model when deploying it as a web service.

IMPORTANT

This article applies to CLI v1 and SDK v1. This profiling technique is not available for v2 of either CLI or SDK.

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Prerequisites

This article assumes you have trained and registered a model with Azure Machine Learning. See the [sample tutorial here](#) for an example of training and registering a scikit-learn model with Azure Machine Learning.

Limitations

- Profiling will not work when the Azure Container Registry (ACR) for your workspace is behind a virtual network.

Run the profiler

Once you have registered your model and prepared the other components necessary for its deployment, you can determine the CPU and memory the deployed service will need. Profiling tests the service that runs your model and returns information such as the CPU usage, memory usage, and response latency. It also provides a recommendation for the CPU and memory based on resource usage.

In order to profile your model, you will need:

- A registered model.
- An inference configuration based on your entry script and inference environment definition.
- A single column tabular dataset, where each row contains a string representing sample request data.

IMPORTANT

At this point we only support profiling of services that expect their request data to be a string, for example: string serialized json, text, string serialized image, etc. The content of each row of the dataset (string) will be put into the body of the HTTP request and sent to the service encapsulating the model for scoring.

IMPORTANT

We only support profiling up to 2 CPUs in ChinaEast2 and USGovArizona region.

Below is an example of how you can construct an input dataset to profile a service that expects its incoming request data to contain serialized json. In this case, we created a dataset based 100 instances of the same request data content. In real world scenarios we suggest that you use larger datasets containing various inputs, especially if your model resource usage/behavior is input dependent.

APPLIES TO:  Python SDK azureml v1

```
import json
from azureml.core import Datastore
from azureml.core.dataset import Dataset
from azureml.data import dataset_type_definitions

input_json = {'data': [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                      [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]]}
# create a string that can be utf-8 encoded and
# put in the body of the request
serialized_input_json = json.dumps(input_json)
dataset_content = []
for i in range(100):
    dataset_content.append(serialized_input_json)
dataset_content = '\n'.join(dataset_content)
file_name = 'sample_request_data.txt'
f = open(file_name, 'w')
f.write(dataset_content)
f.close()

# upload the txt file created above to the Datastore and create a dataset from it
data_store = Datastore.get_default(ws)
data_store.upload_files(['./' + file_name], target_path='sample_request_data')
datastore_path = [(data_store, 'sample_request_data' + '/' + file_name)]
sample_request_data = Dataset.Tabular.from_delimited_files(
    datastore_path, separator='\n',
    infer_column_types=True,
    header=dataset_type_definitions.PromoteHeadersBehavior.NO_HEADERS)
sample_request_data = sample_request_data.register(workspace=ws,
                                                 name='sample_request_data',
                                                 create_new_version=True)
```

Once you have the dataset containing sample request data ready, create an inference configuration. Inference configuration is based on the score.py and the environment definition. The following example demonstrates how to create the inference configuration and run profiling:

```

from azureml.core.model import InferenceConfig, Model
from azureml.core.dataset import Dataset

model = Model(ws, id=model_id)
inference_config = InferenceConfig(entry_script='path-to-score.py',
                                    environment=myenv)
input_dataset = Dataset.get_by_name(workspace=ws, name='sample_request_data')
profile = Model.profile(ws,
                        'unique_name',
                        [model],
                        inference_config,
                        input_dataset=input_dataset)

profile.wait_for_completion(True)

# see the result
details = profile.get_details()

```

APPLIES TO:  Azure CLI ml extension v1

The following command demonstrates how to profile a model by using the CLI:

```
az ml model profile -g <resource-group-name> -w <workspace-name> --inference-config-file <path-to-inf-config.json> -m <model-id> --idi <input-dataset-id> -n <unique-name>
```

TIP

To persist the information returned by profiling, use tags or properties for the model. Using tags or properties stores the data with the model in the model registry. The following examples demonstrate adding a new tag containing the `requestedCpu` and `requestedMemoryInGb` information:

```
model.add_tags({'requestedCpu': details['requestedCpu'],
                'requestedMemoryInGb': details['requestedMemoryInGb']})
```

```
az ml model profile -g <resource-group-name> -w <workspace-name> --i <model-id> --add-tag requestedCpu=1
--add-tag requestedMemoryInGb=0.5
```

Next steps

- [Troubleshoot a failed deployment](#)
- [Deploy to Azure Kubernetes Service](#)
- [Create client applications to consume web services](#)
- [Update web service](#)
- [How to deploy a model using a custom Docker image](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)
- [Create event alerts and triggers for model deployments](#)

Troubleshooting remote model deployment

9/21/2022 • 9 minutes to read • [Edit Online](#)

Learn how to troubleshoot and solve, or work around, common errors you may encounter when deploying a model to Azure Container Instances (ACI) and Azure Kubernetes Service (AKS) using Azure Machine Learning.

NOTE

If you are deploying a model to Azure Kubernetes Service (AKS), we advise you enable [Azure Monitor](#) for that cluster. This will help you understand overall cluster health and resource usage. You might also find the following resources useful:

- [Check for Resource Health events impacting your AKS cluster](#)
- [Azure Kubernetes Service Diagnostics](#)

If you are trying to deploy a model to an unhealthy or overloaded cluster, it is expected to experience issues. If you need help troubleshooting AKS cluster problems please contact AKS Support.

Prerequisites

- An [Azure subscription](#). Try the [free or paid version of Azure Machine Learning](#).
- The [Azure Machine Learning SDK](#).
- The [Azure CLI](#).
- The [CLI extension v1 for Azure Machine Learning](#).

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-m1`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `m1`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Steps for Docker deployment of machine learning models

When you deploy a model to non-local compute in Azure Machine Learning, the following things happen:

1. The Dockerfile you specified in your Environments object in your InferenceConfig is sent to the cloud, along with the contents of your source directory
2. If a previously built image isn't available in your container registry, a new Docker image is built in the cloud and stored in your workspace's default container registry.
3. The Docker image from your container registry is downloaded to your compute target.
4. Your workspace's default Blob store is mounted to your compute target, giving you access to registered models
5. Your web server is initialized by running your entry script's `init()` function
6. When your deployed model receives a request, your `run()` function handles that request

The main difference when using a local deployment is that the container image is built on your local machine,

which is why you need to have Docker installed for a local deployment.

Understanding these high-level steps should help you understand where errors are happening.

Get deployment logs

The first step in debugging errors is to get your deployment logs. First, follow the [instructions here to connect to your workspace](#).

- [Azure CLI](#)
- [Python SDK](#)

APPLIES TO:  [Azure CLI ml extension v1](#)

To get the logs from a deployed webservice, do:

```
az ml service get-logs --verbose --workspace-name <my workspace name> --name <service name>
```

Debug locally

If you have problems when deploying a model to ACI or AKS, deploy it as a local web service. Using a local web service makes it easier to troubleshoot problems. To troubleshoot a deployment locally, see the [local troubleshooting article](#).

Azure Machine learning inference HTTP server

The local inference server allows you to quickly debug your entry script (`score.py`). In case the underlying score script has a bug, the server will fail to initialize or serve the model. Instead, it will throw an exception & the location where the issues occurred. [Learn more about Azure Machine Learning inference HTTP Server](#)

1. Install the `azureml-inference-server-http` package from the [pypi](#) feed:

```
python -m pip install azureml-inference-server-http
```

2. Start the server and set `score.py` as the entry script:

```
azmlinfsrv --entry_script score.py
```

3. Send a scoring request to the server using `curl`:

```
curl -p 127.0.0.1:5001/score
```

NOTE

[Learn frequently asked questions](#) about Azure machine learning Inference HTTP server.

Container can't be scheduled

When deploying a service to an Azure Kubernetes Service compute target, Azure Machine Learning will attempt to schedule the service with the requested amount of resources. If there are no nodes available in the cluster with the appropriate amount of resources after 5 minutes, the deployment will fail. The failure message is

Couldn't Schedule because the kubernetes cluster didn't have available resources after trying for 00:05:00 .

You can address this error by either adding more nodes, changing the SKU of your nodes, or changing the resource requirements of your service.

The error message will typically indicate which resource you need more of - for instance, if you see an error message indicating `0/3 nodes are available: 3 Insufficient nvidia.com/gpu` that means that the service requires GPUs and there are three nodes in the cluster that don't have available GPUs. This could be addressed by adding more nodes if you're using a GPU SKU, switching to a GPU enabled SKU if you aren't or changing your environment to not require GPUs.

Service launch fails

After the image is successfully built, the system attempts to start a container using your deployment configuration. As part of container starting-up process, the `init()` function in your scoring script is invoked by the system. If there are uncaught exceptions in the `init()` function, you might see **CrashLoopBackOff** error in the error message.

Use the info in the [Inspect the Docker log](#) article.

Container azureml-fe-aci launch fails

When deploying a service to an Azure Container Instance compute target, Azure Machine Learning attempts to create a front-end container that has the name `azureml-fe-aci` for the inference request. If `azureml-fe-aci` crashes, you can see logs by running

```
az container logs --name MyContainerGroup --resource-group MyResourceGroup --subscription MySubscription --container-name azureml-fe-aci
```

. You can follow the error message in the logs to make the fix.

The most common failure for `azureml-fe-aci` is that the provided SSL certificate or key is invalid.

Function fails: get_model_path()

Often, in the `init()` function in the scoring script, `Model.get_model_path()` function is called to locate a model file or a folder of model files in the container. If the model file or folder can't be found, the function fails. The easiest way to debug this error is to run the below Python code in the Container shell:

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core.model import Model
import logging
logging.basicConfig(level=logging.DEBUG)
print(Model.get_model_path(model_name='my-best-model'))
```

This example prints the local path (relative to `/var/azureml-app`) in the container where your scoring script is expecting to find the model file or folder. Then you can verify if the file or folder is indeed where it's expected to be.

Setting the logging level to DEBUG may cause additional information to be logged, which may be useful in identifying the failure.

Function fails: run(input_data)

If the service is successfully deployed, but it crashes when you post data to the scoring endpoint, you can add error catching statement in your `run(input_data)` function so that it returns detailed error message instead. For example:

```

def run(input_data):
    try:
        data = json.loads(input_data)['data']
        data = np.array(data)
        result = model.predict(data)
        return json.dumps({"result": result.tolist()})
    except Exception as e:
        result = str(e)
        # return error message back to the client
        return json.dumps({"error": result})

```

Note: Returning error messages from the `run(input_data)` call should be done for debugging purpose only. For security reasons, you shouldn't return error messages this way in a production environment.

HTTP status code 502

A 502 status code indicates that the service has thrown an exception or crashed in the `run()` method of the `score.py` file. Use the information in this article to debug the file.

HTTP status code 503

Azure Kubernetes Service deployments support autoscaling, which allows replicas to be added to support extra load. The autoscaler is designed to handle **gradual** changes in load. If you receive large spikes in requests per second, clients may receive an HTTP status code 503. Even though the autoscaler reacts quickly, it takes AKS a significant amount of time to create more containers.

Decisions to scale up/down is based off of utilization of the current container replicas. The number of replicas that are busy (processing a request) divided by the total number of current replicas is the current utilization. If this number exceeds `autoscale_target_utilization`, then more replicas are created. If it's lower, then replicas are reduced. Decisions to add replicas are eager and fast (around 1 second). Decisions to remove replicas are conservative (around 1 minute). By default, autoscaling target utilization is set to **70%**, which means that the service can handle spikes in requests per second (RPS) of **up to 30%**.

There are two things that can help prevent 503 status codes:

TIP

These two approaches can be used individually or in combination.

- Change the utilization level at which autoscaling creates new replicas. You can adjust the utilization target by setting the `autoscale_target_utilization` to a lower value.

IMPORTANT

This change does not cause replicas to be created *faster*. Instead, they are created at a lower utilization threshold. Instead of waiting until the service is 70% utilized, changing the value to 30% causes replicas to be created when 30% utilization occurs.

If the web service is already using the current max replicas and you're still seeing 503 status codes, increase the `autoscale_max_replicas` value to increase the maximum number of replicas.

- Change the minimum number of replicas. Increasing the minimum replicas provides a larger pool to handle the incoming spikes.

To increase the minimum number of replicas, set `autoscale_min_replicas` to a higher value. You can

calculate the required replicas by using the following code, replacing values with values specific to your project:

```
from math import ceil
# target requests per second
targetRps = 20
# time to process the request (in seconds)
reqTime = 10
# Maximum requests per container
maxReqPerContainer = 1
# target_utilization. 70% in this example
targetUtilization = .7

concurrentRequests = targetRps * reqTime / targetUtilization

# Number of container replicas
replicas = ceil(concurrentRequests / maxReqPerContainer)
```

NOTE

If you receive request spikes larger than the new minimum replicas can handle, you may receive 503s again. For example, as traffic to your service increases, you may need to increase the minimum replicas.

For more information on setting `autoscale_target_utilization`, `autoscale_max_replicas`, and `autoscale_min_replicas` for, see the [AksWebservice](#) module reference.

HTTP status code 504

A 504 status code indicates that the request has timed out. The default timeout is 1 minute.

You can increase the timeout or try to speed up the service by modifying the score.py to remove unnecessary calls. If these actions don't correct the problem, use the information in this article to debug the score.py file. The code may be in a non-responsive state or an infinite loop.

Other error messages

Take these actions for the following errors:

ERROR	RESOLUTION
409 conflict error	When an operation is already in progress, any new operation on that same web service will respond with 409 conflict error. For example, If create or update web service operation is in progress and if you trigger a new Delete operation it will throw an error.
Image building failure when deploying web service	Add "pynacl==1.2.1" as a pip dependency to Conda file for image configuration
<code>['DaskOnBatch:context_managers.DaskOnBatch', 'setup.py']' died with <Signals.SIGKILL: 9></code>	Change the SKU for VMs used in your deployment to one that has more memory.
FPGA failure	You won't be able to deploy models on FPGAs until you've requested and been approved for FPGA quota. To request access, fill out the quota request form: https://aka.ms/aml-real-time-ai

Advanced debugging

You may need to interactively debug the Python code contained in your model deployment. For example, if the entry script is failing and the reason can't be determined by extra logging. By using Visual Studio Code and the debugpy, you can attach to the code running inside the Docker container.

For more information, visit the [interactive debugging in VS Code guide](#).

Model deployment user forum

Next steps

Learn more about deployment:

- [How to deploy and wherezzs](#)
- [How to run and debug experiments locally](#)

Troubleshooting with a local model deployment

9/21/2022 • 4 minutes to read • [Edit Online](#)

Try a local model deployment as a first step in troubleshooting deployment to Azure Container Instances (ACI) or Azure Kubernetes Service (AKS). Using a local web service makes it easier to spot and fix common Azure Machine Learning Docker web service deployment errors.

Prerequisites

- An **Azure subscription**. Try the [free or paid version of Azure Machine Learning](#).
- Option A (**Recommended**) - Debug locally on Azure Machine Learning Compute Instance
 - An Azure Machine Learning Workspace with [compute instance](#) running
- Option B - Debug locally on your compute
 - The [Azure Machine Learning SDK](#).
 - The [Azure CLI](#).
 - The [CLI extension for Azure Machine Learning](#).
 - Have a working Docker installation on your local system.
 - To verify your Docker installation, use the command `docker run hello-world` from a terminal or command prompt. For information on installing Docker, or troubleshooting Docker errors, see the [Docker Documentation](#).
- Option C - Enable local debugging with Azure Machine Learning inference HTTP server.
 - The Azure Machine Learning inference HTTP server ([preview](#)) is a Python package that allows you to easily validate your entry script (`score.py`) in a local development environment. If there's a problem with the scoring script, the server will return an error. It will also return the location where the error occurred.
 - The server can also be used when creating validation gates in a continuous integration and deployment pipeline. For example, start the server with the candidate script and run the test suite against the local endpoint.

Azure Machine learning inference HTTP server

The local inference server allows you to quickly debug your entry script (`score.py`). In case the underlying score script has a bug, the server will fail to initialize or serve the model. Instead, it will throw an exception & the location where the issues occurred. [Learn more about Azure Machine Learning inference HTTP Server](#)

1. Install the `azureml-inference-server-http` package from the [pypi](#) feed:

```
python -m pip install azureml-inference-server-http
```

2. Start the server and set `score.py` as the entry script:

```
azmlinfsrv --entry_script score.py
```

3. Send a scoring request to the server using `curl`:

```
curl -p 127.0.0.1:5001/score
```

NOTE

Learn [frequently asked questions](#) about Azure machine learning Inference HTTP server.

Debug locally

You can find a sample [local deployment notebook](#) in the [MachineLearningNotebooks](#) repo to explore a runnable example.

WARNING

Local web service deployments are not supported for production scenarios.

To deploy locally, modify your code to use `LocalWebservice.deploy_configuration()` to create a deployment configuration. Then use `Model.deploy()` to deploy the service. The following example deploys a model (contained in the `model` variable) as a local web service:

APPLIES TO:  [Python SDK azureml v1](#)

```
from azureml.core.environment import Environment
from azureml.core.model import InferenceConfig, Model
from azureml.core.webservice import LocalWebservice

# Create inference configuration based on the environment definition and the entry script
myenv = Environment.from_conda_specification(name="env", file_path="myenv.yml")
inference_config = InferenceConfig(entry_script="score.py", environment=myenv)
# Create a local deployment, using port 8890 for the web service endpoint
deployment_config = LocalWebservice.deploy_configuration(port=8890)
# Deploy the service
service = Model.deploy(
    ws, "mymodel", [model], inference_config, deployment_config)
# Wait for the deployment to complete
service.wait_for_deployment(True)
# Display the port that the web service is available on
print(service.port)
```

If you are defining your own conda specification YAML, list `azureml-defaults` version $\geq 1.0.45$ as a pip dependency. This package is needed to host the model as a web service.

At this point, you can work with the service as normal. The following code demonstrates sending data to the service:

```
import json

test_sample = json.dumps({'data': [
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
]})

test_sample = bytes(test_sample, encoding='utf8')

prediction = service.run(input_data=test_sample)
print(prediction)
```

For more information on customizing your Python environment, see [Create and manage environments for training and deployment](#).

Update the service

During local testing, you may need to update the `score.py` file to add logging or attempt to resolve any problems that you've discovered. To reload changes to the `score.py` file, use `reload()`. For example, the following code reloads the script for the service, and then sends data to it. The data is scored using the updated `score.py` file:

IMPORTANT

The `reload` method is only available for local deployments. For information on updating a deployment to another compute target, see [how to update your webservice](#).

```
service.reload()  
print(service.run(input_data=test_sample))
```

NOTE

The script is reloaded from the location specified by the `InferenceConfig` object used by the service.

To change the model, Conda dependencies, or deployment configuration, use `update()`. The following example updates the model used by the service:

```
service.update([different_model], inference_config, deployment_config)
```

Delete the service

To delete the service, use `delete()`.

Inspect the Docker log

You can print out detailed Docker engine log messages from the service object. You can view the log for ACI, AKS, and Local deployments. The following example demonstrates how to print the logs.

```
# if you already have the service object handy  
print(service.get_logs())  
  
# if you only know the name of the service (note there might be multiple services with the same name but  
# different version number)  
print(ws.webservices['mysvc'].get_logs())
```

If you see the line `Booting worker with pid: <pid>` occurring multiple times in the logs, it means, there isn't enough memory to start the worker. You can address the error by increasing the value of `memory_gb` in `deployment_config`

Next steps

Learn more about deployment:

- [How to troubleshoot remote deployments](#)
- [Azure Machine Learning inference HTTP Server](#)
- [How to run and debug experiments locally](#)

Configure authentication for models deployed as web services

9/21/2022 • 3 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

Azure Machine Learning allows you to deploy your trained machine learning models as web services. In this article, learn how to configure authentication for these deployments.

The model deployments created by Azure Machine Learning can be configured to use one of two authentication methods:

- **key-based**: A static key is used to authenticate to the web service.
- **token-based**: A temporary token must be obtained from the Azure Machine Learning workspace (using Azure Active Directory) and used to authenticate to the web service. This token expires after a period of time, and must be refreshed to continue working with the web service.

NOTE

Token-based authentication is only available when deploying to Azure Kubernetes Service.

Key-based authentication

Web-services deployed on Azure Kubernetes Service (AKS) have key-based auth *enabled* by default.

Azure Container Instances (ACI) deployed services have key-based auth *disabled* by default, but you can enable it by setting `auth_enabled=True` when creating the ACI web-service. The following code is an example of creating an ACI deployment configuration with key-based auth enabled.

```
from azureml.core.webservice import AciWebservice

aci_config = AciWebservice.deploy_configuration(cpu_cores = 1,
                                                memory_gb = 1,
                                                auth_enabled=True)
```

Then you can use the custom ACI configuration in deployment using the `Model` class.

```
from azureml.core.model import Model, InferenceConfig

inference_config = InferenceConfig(entry_script="score.py",
                                   environment=myenv)
aci_service = Model.deploy(workspace=ws,
                           name="aci_service_sample",
                           models=[model],
                           inference_config=inference_config,
                           deployment_config=aci_config)
aci_service.wait_for_deployment(True)
```

To fetch the auth keys, use `aci_service.get_keys()`. To regenerate a key, use the `regen_key()` function and pass either **Primary** or **Secondary**.

```
aci_service.regen_key("Primary")
# or
aci_service.regen_key("Secondary")
```

Token-based authentication

When you enable token authentication for a web service, users must present an Azure Machine Learning JSON Web Token to the web service to access it. The token expires after a specified time-frame and needs to be refreshed to continue making calls.

- Token authentication is **disabled by default** when you deploy to Azure Kubernetes Service.
- Token authentication **isn't supported** when you deploy to Azure Container Instances.
- Token authentication **can't be used at the same time as key-based authentication**.

To control token authentication, use the `token_auth_enabled` parameter when you create or update a deployment:

```
from azureml.core.webservice import AksWebservice
from azureml.core.model import Model, InferenceConfig

# Create the config
aks_config = AksWebservice.deploy_configuration()

# Enable token auth and disable (key) auth on the webservice
aks_config = AksWebservice.deploy_configuration(token_auth_enabled=True, auth_enabled=False)

aks_service_name ='aks-service-1'

# deploy the model
aks_service = Model.deploy(workspace=ws,
                           name=aks_service_name,
                           models=[model],
                           inference_config=inference_config,
                           deployment_config=aks_config,
                           deployment_target=aks_target)

aks_service.wait_for_deployment(show_output = True)
```

If token authentication is enabled, you can use the `get_token` method to retrieve a JSON Web Token (JWT) and that token's expiration time:

TIP

If you use a service principal to get the token, and want it to have the minimum required access to retrieve a token, assign it to the **reader** role for the workspace.

```
token, refresh_by = aks_service.get_token()
print(token)
```

IMPORTANT

You'll need to request a new token after the token's `refresh_by` time. If you need to refresh tokens outside of the Python SDK, one option is to use the REST API with service-principal authentication to periodically make the `service.get_token()` call, as discussed previously.

We strongly recommend that you create your Azure Machine Learning workspace in the same region as your Azure Kubernetes Service cluster.

To authenticate with a token, the web service will make a call to the region in which your Azure Machine Learning workspace is created. If your workspace region is unavailable, you won't be able to fetch a token for your web service, even if your cluster is in a different region from your workspace. The result is that Azure AD Authentication is unavailable until your workspace region is available again.

Also, the greater the distance between your cluster's region and your workspace region, the longer it will take to fetch a token.

Next steps

For more information on authenticating to a deployed model, see [Create a client for a model deployed as a web service](#).

Consume an Azure Machine Learning model deployed as a web service

9/21/2022 • 13 minutes to read • [Edit Online](#)

Deploying an Azure Machine Learning model as a web service creates a REST API endpoint. You can send data to this endpoint and receive the prediction returned by the model. In this document, learn how to create clients for the web service by using C#, Go, Java, and Python.

You create a web service when you deploy a model to your local environment, Azure Container Instances, Azure Kubernetes Service, or field-programmable gate arrays (FPGA). You retrieve the URI used to access the web service by using the [Azure Machine Learning SDK](#). If authentication is enabled, you can also use the SDK to get the authentication keys or tokens.

The general workflow for creating a client that uses a machine learning web service is:

1. Use the SDK to get the connection information.
2. Determine the type of request data used by the model.
3. Create an application that calls the web service.

TIP

The examples in this document are manually created without the use of OpenAPI (Swagger) specifications. If you've enabled an OpenAPI specification for your deployment, you can use tools such as [swagger-codegen](#) to create client libraries for your service.

IMPORTANT

Some of the Azure CLI commands in this article use the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Connection information

NOTE

Use the Azure Machine Learning SDK to get the web service information. This is a Python SDK. You can use any language to create a client for the service.

The `azureml.core.WebService` class provides the information you need to create a client. The following `WebService` properties are useful for creating a client application:

- `auth_enabled` - If key authentication is enabled, `True`; otherwise, `False`.
- `token_auth_enabled` - If token authentication is enabled, `True`; otherwise, `False`.
- `scoring_uri` - The REST API address.
- `swagger_uri` - The address of the OpenAPI specification. This URI is available if you enabled automatic

schema generation. For more information, see [Deploy models with Azure Machine Learning](#).

There are several ways to retrieve this information for deployed web services:

- [Python SDK](#)
- [Azure CLI](#)
- [Studio](#)

APPLIES TO:  [Python SDK azureml v1](#)

- When you deploy a model, a `Webservice` object is returned with information about the service:

```
service = Model.deploy(ws, "myservice", [model], inference_config, deployment_config)
service.wait_for_deployment(show_output = True)
print(service.scoring_uri)
print(service.swagger_uri)
```

- You can use `Webservice.list` to retrieve a list of deployed web services for models in your workspace. You can add filters to narrow the list of information returned. For more information about what can be filtered on, see the [Webservice.list](#) reference documentation.

```
services = Webservice.list(ws)
print(services[0].scoring_uri)
print(services[0].swagger_uri)
```

- If you know the name of the deployed service, you can create a new instance of `Webservice`, and provide the workspace and service name as parameters. The new object contains information about the deployed service.

```
service = Webservice(workspace=ws, name='myservice')
print(service.scoring_uri)
print(service.swagger_uri)
```

The following table shows what these URIs look like:

URI TYPE	EXAMPLE
Scoring URI	<code>http://104.214.29.152:80/api/v1/service/<service-name>/score</code>
Swagger URI	<code>http://104.214.29.152/api/v1/service/<service-name>/swagger.json</code>

TIP

The IP address will be different for your deployment. Each AKS cluster will have its own IP address that is shared by deployments to that cluster.

Secured web service

If you secured the deployed web service using a TLS/SSL certificate, you can use [HTTPS](#) to connect to the service using the scoring or swagger URI. HTTPS helps secure communications between a client and a web service by encrypting communications between the two. Encryption uses [Transport Layer Security \(TLS\)](#). TLS is sometimes still referred to as *Secure Sockets Layer (SSL)*, which was the predecessor of TLS.

IMPORTANT

Web services deployed by Azure Machine Learning only support TLS version 1.2. When creating a client application, make sure that it supports this version.

For more information, see [Use TLS to secure a web service through Azure Machine Learning](#).

Authentication for services

Azure Machine Learning provides two ways to control access to your web services.

AUTHENTICATION METHOD	ACI	AKS
Key	Disabled by default	Enabled by default
Token	Not Available	Disabled by default

When sending a request to a service that is secured with a key or token, use the **Authorization** header to pass the key or token. The key or token must be formatted as `Bearer <key-or-token>`, where `<key-or-token>` is your key or token value.

The primary difference between keys and tokens is that **keys are static and can be regenerated manually**, and **tokens need to be refreshed upon expiration**. Key-based auth is supported for Azure Container Instance and Azure Kubernetes Service deployed web-services, and token-based auth is **only** available for Azure Kubernetes Service deployments. For more information on configuring authentication, see [Configure authentication for models deployed as web services](#).

Authentication with keys

When you enable authentication for a deployment, you automatically create authentication keys.

- Authentication is enabled by default when you're deploying to Azure Kubernetes Service.
- Authentication is disabled by default when you're deploying to Azure Container Instances.

To control authentication, use the `auth_enabled` parameter when you're creating or updating a deployment.

If authentication is enabled, you can use the `get_keys` method to retrieve a primary and secondary authentication key:

```
primary, secondary = service.get_keys()  
print(primary)
```

IMPORTANT

If you need to regenerate a key, use `service.regen_key`.

Authentication with tokens

When you enable token authentication for a web service, a user must provide an Azure Machine Learning JWT token to the web service to access it.

- Token authentication is disabled by default when you're deploying to Azure Kubernetes Service.
- Token authentication isn't supported when you're deploying to Azure Container Instances.

To control token authentication, use the `token_auth_enabled` parameter when you're creating or updating a deployment.

If token authentication is enabled, you can use the `get_token` method to retrieve a bearer token and that token's expiration time:

```
token, refresh_by = service.get_token()  
print(token)
```

If you have the [Azure CLI and the machine learning extension](#), you can use the following command to get a token:

APPLIES TO:  Azure CLI ml extension v1

```
az ml service get-access-token -n <service-name>
```

IMPORTANT

Currently the only way to retrieve the token is by using the Azure Machine Learning SDK or the Azure CLI machine learning extension.

You'll need to request a new token after the token's `refresh_by` time.

Request data

The REST API expects the body of the request to be a JSON document with the following structure:

```
{  
    "data":  
        [  
            <model-specific-data-structure>  
        ]  
}
```

IMPORTANT

The structure of the data needs to match what the scoring script and model in the service expect. The scoring script might modify the data before passing it to the model.

Binary data

For information on how to enable support for binary data in your service, see [Binary data](#).

TIP

Enabling support for binary data happens in the score.py file used by the deployed model. From the client, use the HTTP functionality of your programming language. For example, the following snippet sends the contents of a JPG file to a web service:

```
import requests  
# Load image data  
data = open('example.jpg', 'rb').read()  
# Post raw data to scoring URI  
res = request.post(url='<scoring-uri>', data=data, headers={'Content-Type': 'application/octet-stream'})
```

Cross-origin resource sharing (CORS)

For information on enabling CORS support in your service, see [Cross-origin resource sharing](#).

Call the service (C#)

This example demonstrates how to use C# to call the web service created from the [Train within notebook](#) example:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json;

namespace MLWebServiceClient
{
    // The data structure expected by the service
    internal class InputData
    {
        [JsonProperty("data")]
        // The service used by this example expects an array containing
        // one or more arrays of doubles
        internal double[,] data;
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Set the scoring URI and authentication key or token
            string scoringUri = "<your web service URI>";
            string authKey = "<your key or token>";

            // Set the data to be sent to the service.
            // In this case, we are sending two sets of data to be scored.
            InputData payload = new InputData();
            payload.data = new double[,] {
                {
                    0.0199132141783263,
                    0.0506801187398187,
                    0.104808689473925,
                    0.0700725447072635,
                    -0.0359677812752396,
                    -0.0266789028311707,
                    -0.0249926566315915,
                    -0.00259226199818282,
                    0.00371173823343597,
                    0.0403433716478807
                },
                {
                    -0.0127796318808497,
                    -0.044641636506989,
                    0.0606183944448076,
                    0.0528581912385822,
                    0.0479653430750293,
                    0.0293746718291555,
                    -0.0176293810234174,
                    0.0343088588777263,
                    0.0702112981933102,
                    0.00720651632920303
                }
            };
            // Create the HTTP client
            HttpClient client = new HttpClient();
            // Set the auth header. Only needed if the web service requires authentication.
            client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", authKey);
        }
    }
}
```

```

    // Make the request
    try {
        var request = new HttpRequestMessage(HttpMethod.Post, new Uri(scoringUri));
        request.Content = new StringContent(JsonConvert.SerializeObject(payload));
        request.Content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
        var response = client.SendAsync(request).Result;
        // Display the response from the web service
        Console.WriteLine(response.Content.ReadAsStringAsync().Result);
    }
    catch (Exception e)
    {
        Console.Out.WriteLine(e.Message);
    }
}
}
}

```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Call the service (Go)

This example demonstrates how to use Go to call the web service created from the [Train within notebook](#) example:

```

package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
)

// Features for this model are an array of decimal values
type Features []float64

// The web service input can accept multiple sets of values for scoring
type InputData struct {
    Data []Features `json:"data",omitempty"`
}

// Define some example data
var exampleData = []Features{
    []float64{
        0.0199132141783263,
        0.0506801187398187,
        0.104808689473925,
        0.0700725447072635,
        -0.0359677812752396,
        -0.0266789028311707,
        -0.0249926566315915,
        -0.00259226199818282,
        0.00371173823343597,
        0.0403433716478807,
    },
    []float64{
        -0.0127796318808497,
        -0.044641636506989,
        0.0606183944448076,
        0.0528581912385822,
    }
}

```

```

        0.0479653430750293,
        0.0293746718291555,
        -0.0176293810234174,
        0.0343088588777263,
        0.0702112981933102,
        0.00720651632920303,
    },
}

// Set to the URI for your service
var serviceUri string = "<your web service URI>"
// Set to the authentication key or token (if any) for your service
var authKey string = "<your key or token>

func main() {
    // Create the input data from example data
    jsonData := InputData{
        Data: exampleData,
    }
    // Create JSON from it and create the body for the HTTP request
    jsonValue, _ := json.Marshal(jsonData)
    body := bytes.NewBuffer(jsonValue)

    // Create the HTTP request
    client := &http.Client{}
    request, err := http.NewRequest("POST", serviceUri, body)
    request.Header.Add("Content-Type", "application/json")

    // These next two are only needed if using an authentication key
    bearer := fmt.Sprintf("Bearer %v", authKey)
    request.Header.Add("Authorization", bearer)

    // Send the request to the web service
    resp, err := client.Do(request)
    if err != nil {
        fmt.Println("Failure: ", err)
    }

    // Display the response received
    respBody, _ := ioutil.ReadAll(resp.Body)
    fmt.Println(string(respBody))
}

```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Call the service (Java)

This example demonstrates how to use Java to call the web service created from the [Train within notebook](#) example:

```

import java.io.IOException;
import org.apache.http.client.fluent.*;
import org.apache.http.entity.ContentType;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;

public class App {
    // Handle making the request
    public static void sendRequest(String data) {
        // Replace with the scoring_uri of your service
        String uri = "<your web service URI>";
        // If using authentication, replace with the auth key or token
    }
}

```

```

String key = "<your key or token>";
try {
    // Create the request
    Content content = Request.Post(uri)
        .addHeader("Content-Type", "application/json")
        // Only needed if using authentication
        .addHeader("Authorization", "Bearer " + key)
        // Set the JSON data as the body
        .bodyString(data, ContentType.APPLICATION_JSON)
        // Make the request and display the response.
        .execute().returnContent();
    System.out.println(content);
}
catch (IOException e) {
    System.out.println(e);
}
}

public static void main(String[] args) {
    // Create the data to send to the service
    JSONObject obj = new JSONObject();
    // In this case, it's an array of arrays
    JSONArray dataItems = new JSONArray();
    // Inner array has 10 elements
    JSONArray item1 = new JSONArray();
    item1.add(0.0199132141783263);
    item1.add(0.0506801187398187);
    item1.add(0.104808689473925);
    item1.add(0.0700725447072635);
    item1.add(-0.0359677812752396);
    item1.add(-0.0266789028311707);
    item1.add(-0.0249926566315915);
    item1.add(-0.00259226199818282);
    item1.add(0.00371173823343597);
    item1.add(0.0403433716478807);
    // Add the first set of data to be scored
    dataItems.add(item1);
    // Create and add the second set
    JSONArray item2 = new JSONArray();
    item2.add(-0.0127796318808497);
    item2.add(-0.044641636506989);
    item2.add(0.0606183944448076);
    item2.add(0.0528581912385822);
    item2.add(0.0479653430750293);
    item2.add(0.0293746718291555);
    item2.add(-0.0176293810234174);
    item2.add(0.0343088588777263);
    item2.add(0.0702112981933102);
    item2.add(0.00720651632920303);
    dataItems.add(item2);
    obj.put("data", dataItems);

    // Make the request using the JSON document string
    sendRequest(obj.toJSONString());
}
}

```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Call the service (Python)

This example demonstrates how to use Python to call the web service created from the [Train within notebook](#) example:

```

import requests
import json

# URL for the web service
scoring_uri = '<your web service URI>'
# If the service is authenticated, set the key or token
key = '<your key or token>'

# Two sets of data to score, so we get two results back
data = {"data":
    [
        [
            0.0199132141783263,
            0.0506801187398187,
            0.104808689473925,
            0.0700725447072635,
            -0.0359677812752396,
            -0.0266789028311707,
            -0.0249926566315915,
            -0.00259226199818282,
            0.00371173823343597,
            0.0403433716478807
        ],
        [
            -0.0127796318808497,
            -0.044641636506989,
            0.0606183944448076,
            0.0528581912385822,
            0.0479653430750293,
            0.0293746718291555,
            -0.0176293810234174,
            0.0343088588777263,
            0.0702112981933102,
            0.00720651632920303
        ]
    ]
}

# Convert to JSON string
input_data = json.dumps(data)

# Set the content type
headers = {'Content-Type': 'application/json'}
# If authentication is enabled, set the authorization header
headers['Authorization'] = f'Bearer {key}'

# Make the request and display the response
resp = requests.post(scoring_uri, input_data, headers=headers)
print(resp.text)

```

The results returned are similar to the following JSON document:

```
[217.67978776218715, 224.78937091757172]
```

Web service schema (OpenAPI specification)

If you used automatic schema generation with your deployment, you can get the address of the OpenAPI specification for the service by using the [swagger_uri property](#). (For example, `print(service.swagger_uri)`) Use a GET request or open the URI in a browser to retrieve the specification.

The following JSON document is an example of a schema (OpenAPI specification) generated for a deployment:

```
{
    "swagger": "2.0",
    "info": {
        "title": "Scoring API"
    }
}
```

```

    "info": {
      "title": "myservice",
      "description": "API specification for Azure Machine Learning myservice",
      "version": "1.0"
    },
    "schemes": [
      "https"
    ],
    "consumes": [
      "application/json"
    ],
    "produces": [
      "application/json"
    ],
    "securityDefinitions": {
      "Bearer": {
        "type": "apiKey",
        "name": "Authorization",
        "in": "header",
        "description": "For example: Bearer abc123"
      }
    },
    "paths": {
      "/": {
        "get": {
          "operationId": "ServiceHealthCheck",
          "description": "Simple health check endpoint to ensure the service is up at any given point.",
          "responses": {
            "200": {
              "description": "If service is up and running, this response will be returned with the content 'Healthy'",
              "schema": {
                "type": "string"
              },
              "examples": {
                "application/json": "Healthy"
              }
            },
            "default": {
              "description": "The service failed to execute due to an error.",
              "schema": {
                "$ref": "#/definitions/ErrorResponse"
              }
            }
          }
        }
      },
      "/score": {
        "post": {
          "operationId": "RunMLService",
          "description": "Run web service's model and get the prediction output",
          "security": [
            {
              "Bearer": []
            }
          ],
          "parameters": [
            {
              "name": "serviceInputPayload",
              "in": "body",
              "description": "The input payload for executing the real-time machine learning service.",
              "schema": {
                "$ref": "#/definitions/ServiceInput"
              }
            }
          ],
          "responses": {
            "200": {
              "description": "The predicted output from the machine learning model"
            }
          }
        }
      }
    }
  }
}

```

```

        "200": {
            "description": "The service processed the input correctly and provided a result prediction, if applicable.",
            "schema": {
                "$ref": "#/definitions/ServiceOutput"
            }
        },
        "default": {
            "description": "The service failed to execute due to an error.",
            "schema": {
                "$ref": "#/definitions/ErrorResponse"
            }
        }
    }
},
"definitions": {
    "ServiceInput": {
        "type": "object",
        "properties": {
            "data": {
                "type": "array",
                "items": {
                    "type": "array",
                    "items": {
                        "type": "integer",
                        "format": "int64"
                    }
                }
            }
        },
        "example": {
            "data": [
                [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
            ]
        }
    },
    "ServiceOutput": {
        "type": "array",
        "items": {
            "type": "number",
            "format": "double"
        },
        "example": [
            3726.995
        ]
    },
    "ErrorResponse": {
        "type": "object",
        "properties": {
            "status_code": {
                "type": "integer",
                "format": "int32"
            },
            "message": {
                "type": "string"
            }
        }
    }
}
}

```

For more information, see [OpenAPI specification](#).

For a utility that can create client libraries from the specification, see [swagger-codegen](#).

TIP

You can retrieve the schema JSON document after you deploy the service. Use the [swagger_uri](#) property from the deployed web service (for example, `service.swagger_uri`) to get the URI to the local web service's Swagger file.

Consume the service from Power BI

Power BI supports consumption of Azure Machine Learning web services to enrich the data in Power BI with predictions.

To generate a web service that's supported for consumption in Power BI, the schema must support the format that's required by Power BI. [Learn how to create a Power BI-supported schema](#).

Once the web service is deployed, it's consumable from Power BI dataflows. [Learn how to consume an Azure Machine Learning web service from Power BI](#).

Next steps

To view a reference architecture for real-time scoring of Python and deep learning models, go to the [Azure architecture center](#).

Advanced entry script authoring

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article shows how to write entry scripts for specialized use cases.

Prerequisites

This article assumes you already have a trained machine learning model that you intend to deploy with Azure Machine Learning. To learn more about model deployment, see [How to deploy and where](#).

Automatically generate a Swagger schema

To automatically generate a schema for your web service, provide a sample of the input and/or output in the constructor for one of the defined type objects. The type and sample are used to automatically create the schema. Azure Machine Learning then creates an [OpenAPI](#) (Swagger) specification for the web service during deployment.

WARNING

You must not use sensitive or private data for sample input or output. The Swagger page for AML-hosted inferencing exposes the sample data.

These types are currently supported:

- `pandas`
- `numpy`
- `pyspark`
- Standard Python object

To use schema generation, include the open-source `inference-schema` package version 1.1.0 or above in your dependencies file. For more information on this package, see <https://github.com/Azure/InferenceSchema>. In order to generate conforming swagger for automated web service consumption, scoring script `run()` function must have API shape of:

- A first parameter of type "StandardPythonParameterType", named **Inputs** and nested.
- An optional second parameter of type "StandardPythonParameterType", named **GlobalParameters**.
- Return a dictionary of type "StandardPythonParameterType" named **Results** and nested.

Define the input and output sample formats in the `input_sample` and `output_sample` variables, which represent the request and response formats for the web service. Use these samples in the input and output function decorators on the `run()` function. The following scikit-learn example uses schema generation.

Power BI compatible endpoint

The following example demonstrates how to define API shape according to above instruction. This method is supported for consuming the deployed web service from Power BI. ([Learn more about how to consume the web service from Power BI](#).)

```

import json
import pickle
import numpy as np
import pandas as pd
import azureml.train.automl
from sklearn.externals import joblib
from sklearn.linear_model import Ridge

from inference_schema.schema_decorators import input_schema, output_schema
from inference_schema.parameter_types.standard_py_parameter_type import StandardPythonParameterType
from inference_schema.parameter_types.numpy_parameter_type import NumpyParameterType
from inference_schema.parameter_types.pandas_parameter_type import PandasParameterType


def init():
    global model
    # Replace filename if needed.
    model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_regression_model.pkl')
    # Deserialize the model file back into a sklearn model.
    model = joblib.load(model_path)

# providing 3 sample inputs for schema generation
numpy_sample_input = NumpyParameterType(np.array([[1,2,3,4,5,6,7,8,9,10],
[10,9,8,7,6,5,4,3,2,1]],dtype='float64'))
pandas_sample_input = PandasParameterType(pd.DataFrame({'name': ['Sarah', 'John'], 'age': [25, 26]}))
standard_sample_input = StandardPythonParameterType(0.0)

# This is a nested input sample, any item wrapped by `ParameterType` will be described by schema
sample_input = StandardPythonParameterType({'input1': numpy_sample_input,
                                             'input2': pandas_sample_input,
                                             'input3': standard_sample_input})

sample_global_parameters = StandardPythonParameterType(1.0) # this is optional
sample_output = StandardPythonParameterType([1.0, 1.0])
outputs = StandardPythonParameterType({'Results':sample_output}) # 'Results' is case sensitive

@input_schema('Inputs', sample_input)
# 'Inputs' is case sensitive

@input_schema('GlobalParameters', sample_global_parameters)
# this is optional, 'GlobalParameters' is case sensitive

@output_schema(outputs)

def run(Inputs, GlobalParameters):
    # the parameters here have to match those in decorator, both 'Inputs' and
    # 'GlobalParameters' here are case sensitive
    try:
        data = Inputs['input1']
        # data will be convert to target format
        assert isinstance(data, np.ndarray)
        result = model.predict(data)
        return result.tolist()
    except Exception as e:
        error = str(e)
        return error

```

TIP

The return value from the script can be any Python object that is serializable to JSON. For example, if your model returns a Pandas dataframe that contains multiple columns, you might use an output decorator similar to the following code:

```
output_sample = pd.DataFrame(data=[{"a1": 5, "a2": 6}])
@output_schema(PandasParameterType(output_sample))
...
result = model.predict(data)
return result
```

Binary (that is, image) data

If your model accepts binary data, like an image, you must modify the `score.py` file used for your deployment to accept raw HTTP requests. To accept raw data, use the `AMLRequest` class in your entry script and add the `@rawhttp` decorator to the `run()` function.

Here's an example of a `score.py` that accepts binary data:

```
from azureml.contrib.servicesaml_request import AMLRequest, rawhttp
from azureml.contrib.servicesaml_response import AMLResponse
from PIL import Image
import json

def init():
    print("This is init()")

@rawhttp
def run(request):
    print("This is run()")

    if request.method == 'GET':
        # For this example, just return the URL for GETs.
        respBody = str.encode(request.full_path)
        return AMLResponse(respBody, 200)
    elif request.method == 'POST':
        file_bytes = request.files["image"]
        image = Image.open(file_bytes).convert('RGB')
        # For a real-world solution, you would load the data from reqBody
        # and send it to the model. Then return the response.

        # For demonstration purposes, this example just returns the size of the image as the response..
        return AMLResponse(json.dumps(image.size), 200)
    else:
        return AMLResponse("bad request", 500)
```

IMPORTANT

The `AMLRequest` class is in the `azureml.contrib` namespace. Entities in this namespace change frequently as we work to improve the service. Anything in this namespace should be considered a preview that's not fully supported by Microsoft.

If you need to test this in your local development environment, you can install the components by using the following command:

```
pip install azureml-contrib-services
```

The `AMLRequest` class only allows you to access the raw posted data in the `score.py`, there's no client-side component. From a client, you post data as normal. For example, the following Python code reads an image file and posts the data:

```
import requests

uri = service.scoring_uri
image_path = 'test.jpg'
files = {'image': open(image_path, 'rb').read()}
response = requests.post(uri, files=files)

print(response.json)
```

Cross-origin resource sharing (CORS)

Cross-origin resource sharing is a way to allow resources on a webpage to be requested from another domain. CORS works via HTTP headers sent with the client request and returned with the service response. For more information on CORS and valid headers, see [Cross-origin resource sharing](#) in Wikipedia.

To configure your model deployment to support CORS, use the `AMLResponse` class in your entry script. This class allows you to set the headers on the response object.

The following example sets the `Access-Control-Allow-Origin` header for the response from the entry script:

```

from azureml.contrib.services.aml_request import AMLRequest, rawhttp
from azureml.contrib.services.aml_response import AMLResponse

def init():
    print("This is init()")

@rawhttp
def run(request):
    print("This is run()")
    print("Request: [{0}].format(request)")
    if request.method == 'GET':
        # For this example, just return the URL for GET.
        # For a real-world solution, you would load the data from URL params or headers
        # and send it to the model. Then return the response.
        respBody = str.encode(request.full_path)
        resp = AMLResponse(respBody, 200)
        resp.headers["Allow"] = "OPTIONS, GET, POST"
        resp.headers["Access-Control-Allow-Methods"] = "OPTIONS, GET, POST"
        resp.headers['Access-Control-Allow-Origin'] = "http://www.example.com"
        resp.headers['Access-Control-Allow-Headers'] = "*"
        return resp
    elif request.method == 'POST':
        reqBody = request.get_data(False)
        # For a real-world solution, you would load the data from reqBody
        # and send it to the model. Then return the response.
        resp = AMLResponse(reqBody, 200)
        resp.headers["Allow"] = "OPTIONS, GET, POST"
        resp.headers["Access-Control-Allow-Methods"] = "OPTIONS, GET, POST"
        resp.headers['Access-Control-Allow-Origin'] = "http://www.example.com"
        resp.headers['Access-Control-Allow-Headers'] = "*"
        return resp
    elif request.method == 'OPTIONS':
        resp = AMLResponse("", 200)
        resp.headers["Allow"] = "OPTIONS, GET, POST"
        resp.headers["Access-Control-Allow-Methods"] = "OPTIONS, GET, POST"
        resp.headers['Access-Control-Allow-Origin'] = "http://www.example.com"
        resp.headers['Access-Control-Allow-Headers'] = "*"
        return resp
    else:
        return AMLResponse("bad request", 400)

```

IMPORTANT

The `AMLResponse` class is in the `azureml.contrib` namespace. Entities in this namespace change frequently as we work to improve the service. Anything in this namespace should be considered a preview that's not fully supported by Microsoft.

If you need to test this in your local development environment, you can install the components by using the following command:

```
pip install azureml-contrib-services
```

WARNING

Azure Machine Learning will route only POST and GET requests to the containers running the scoring service. This can cause errors due to browsers using OPTIONS requests to pre-flight CORS requests.

Load registered models

There are two ways to locate models in your entry script:

- `AZUREML_MODEL_DIR`: An environment variable containing the path to the model location.
- `Model.get_model_path`: An API that returns the path to model file using the registered model name.

AZUREML_MODEL_DIR

`AZUREML_MODEL_DIR` is an environment variable created during service deployment. You can use this environment variable to find the location of the deployed model(s).

The following table describes the value of `AZUREML_MODEL_DIR` depending on the number of models deployed:

DEPLOYMENT	ENVIRONMENT VARIABLE VALUE
Single model	The path to the folder containing the model.
Multiple models	The path to the folder containing all models. Models are located by name and version in this folder (<code>\$MODEL_NAME/\$VERSION</code>)

During model registration and deployment, Models are placed in the `AZUREML_MODEL_DIR` path, and their original filenames are preserved.

To get the path to a model file in your entry script, combine the environment variable with the file path you're looking for.

Single model example

```
# Example when the model is a file
model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'sklearn_regression_model.pkl')

# Example when the model is a folder containing a file
file_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'my_model_folder', 'sklearn_regression_model.pkl')
```

Multiple model example

In this scenario, two models are registered with the workspace:

- `my_first_model`: Contains one file (`my_first_model.pkl`) and there's only one version (`1`).
- `my_second_model`: Contains one file (`my_second_model.pkl`) and there are two versions; `1` and `2`.

When the service was deployed, both models are provided in the deploy operation:

```
first_model = Model(ws, name="my_first_model", version=1)
second_model = Model(ws, name="my_second_model", version=2)
service = Model.deploy(ws, "myservice", [first_model, second_model], inference_config, deployment_config)
```

In the Docker image that hosts the service, the `AZUREML_MODEL_DIR` environment variable contains the directory where the models are located. In this directory, each of the models is located in a directory path of `MODEL_NAME/VERSION`. Where `MODEL_NAME` is the name of the registered model, and `VERSION` is the version of the model. The files that make up the registered model are stored in these directories.

In this example, the paths would be `$AZUREML_MODEL_DIR/my_first_model/1/my_first_model.pkl` and `$AZUREML_MODEL_DIR/my_second_model/2/my_second_model.pkl`.

```
# Example when the model is a file, and the deployment contains multiple models
first_model_name = 'my_first_model'
first_model_version = '1'
first_model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), first_model_name, first_model_version,
'my_first_model.pkl')
second_model_name = 'my_second_model'
second_model_version = '2'
second_model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), second_model_name, second_model_version,
'my_second_model.pkl')
```

get_model_path

When you register a model, you provide a model name that's used for managing the model in the registry. You use this name with the [Model.get_model_path\(\)](#) method to retrieve the path of the model file or files on the local file system. If you register a folder or a collection of files, this API returns the path of the directory that contains those files.

When you register a model, you give it a name. The name corresponds to where the model is placed, either locally or during service deployment.

Framework-specific examples

More entry script examples for specific machine learning use cases can be found below:

- [PyTorch](#)
- [TensorFlow](#)
- [Keras](#)
- [AutoML](#)
- [ONNX](#)

Next steps

- [Troubleshoot a failed deployment](#)
- [Deploy to Azure Kubernetes Service](#)
- [Create client applications to consume web services](#)
- [Update web service](#)
- [How to deploy a model using a custom Docker image](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)
- [Create event alerts and triggers for model deployments](#)

Python package extensibility for prebuilt Docker images (preview)

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

The [prebuilt Docker images for model inference](#) contain packages for popular machine learning frameworks. There are two methods that can be used to add Python packages **without rebuilding the Docker image**:

- **Dynamic installation:** This approach uses a [requirements](#) file to automatically restore Python packages when the Docker container boots.

Consider this method for **rapid prototyping**. When the image starts, packages are restored using the `requirements.txt` file. This method increases startup of the image, and you must wait longer before the deployment can handle requests.

- **Pre-installed Python packages:** You provide a directory containing preinstalled Python packages. During deployment, this directory is mounted into the container for your entry script (`score.py`) to use.

Use this approach for **production deployments**. Since the directory containing the packages is mounted to the image, it can be used even when your deployments don't have public internet access. For example, when deployed into a secured Azure Virtual Network.

IMPORTANT

Using Python package extensibility for prebuilt Docker images with Azure Machine Learning is currently in preview. Preview functionality is provided "as-is", with no guarantee of support or service level agreement. For more information, see the [Supplemental terms of use for Microsoft Azure previews](#).

Prerequisites

- An Azure Machine Learning workspace. For a tutorial on creating a workspace, see [Get started with Azure Machine Learning](#).
- Familiarity with using Azure Machine Learning [environments](#).
- Familiarity with [Where and how to deploy models](#) with Azure Machine Learning.

Dynamic installation

This approach uses a [requirements](#) file to automatically restore Python packages when the image starts up.

To extend your prebuilt docker container image through a `requirements.txt`, follow these steps:

1. Create a `requirements.txt` file alongside your `score.py` script.
2. Add **all** of your required packages to the `requirements.txt` file.
3. Set the `AZUREML_EXTRA_REQUIREMENTS_TXT` environment variable in your Azure Machine Learning [environment](#) to the location of `requirements.txt` file.

Once deployed, the packages will automatically be restored for your score script.

TIP

Even while prototyping, we recommend that you pin each package version in `requirements.txt`. For example, use `scipy == 1.2.3` instead of just `scipy` or even `scipy > 1.2.3`. If you don't pin an exact version and `scipy` releases a new version, this can break your scoring script and cause failures during deployment and scaling.

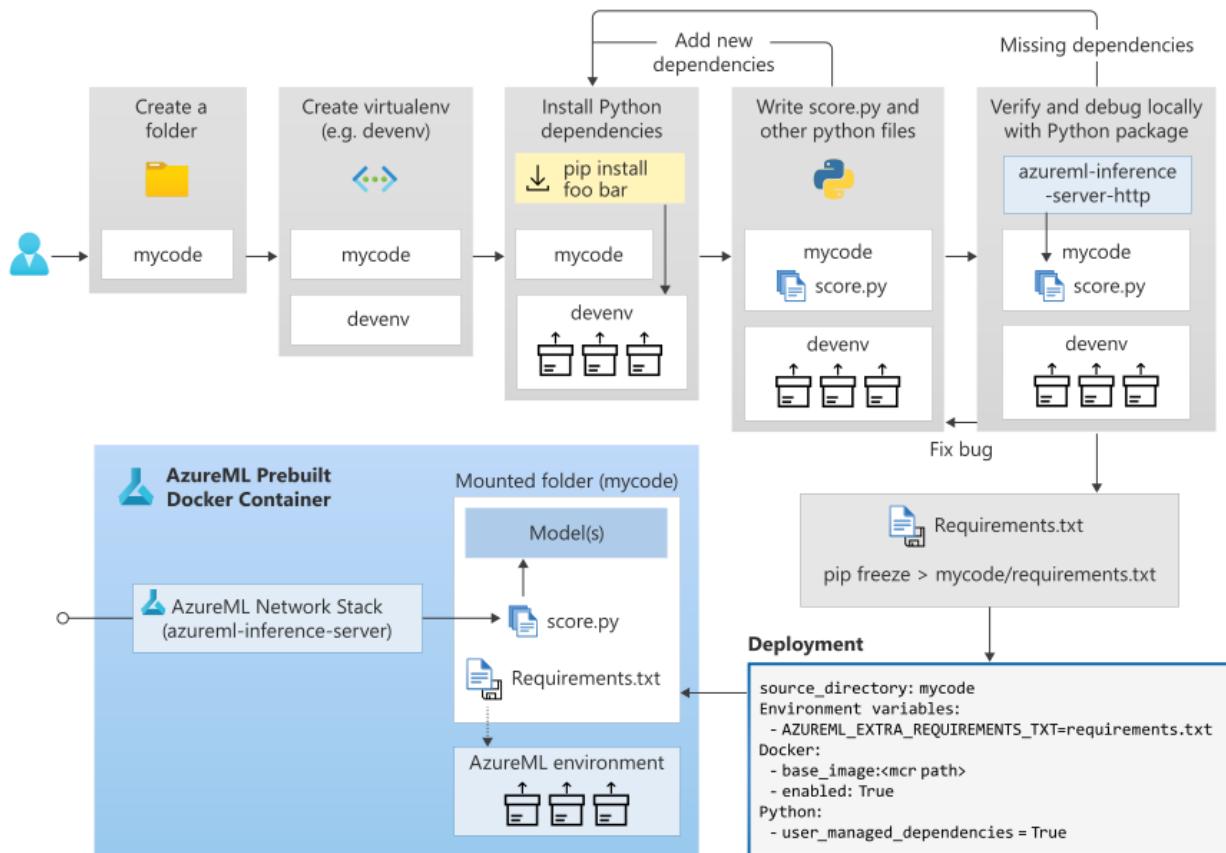
The following example demonstrates setting the `AZUREML_EXTRA_REQUIREMENTS_TXT` environment variable:

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

myenv = Environment(name="my_azureml_env")
myenv.docker.enabled = True
myenv.docker.base_image = <MCR-path>
myenv.python.user_managed_dependencies = True

myenv.environment_variables = {
    "AZUREML_EXTRA_REQUIREMENTS_TXT": "requirements.txt"
}
```

The following diagram is a visual representation of the dynamic installation process:



Pre-installed Python packages

This approach mounts a directory that you provide into the image. The Python packages from this directory can then be used by the entry script (`score.py`).

To extend your prebuilt docker container image through pre-installed Python packages, follow these steps:

IMPORTANT

You must use packages compatible with Python 3.7. All current images are pinned to Python 3.7.

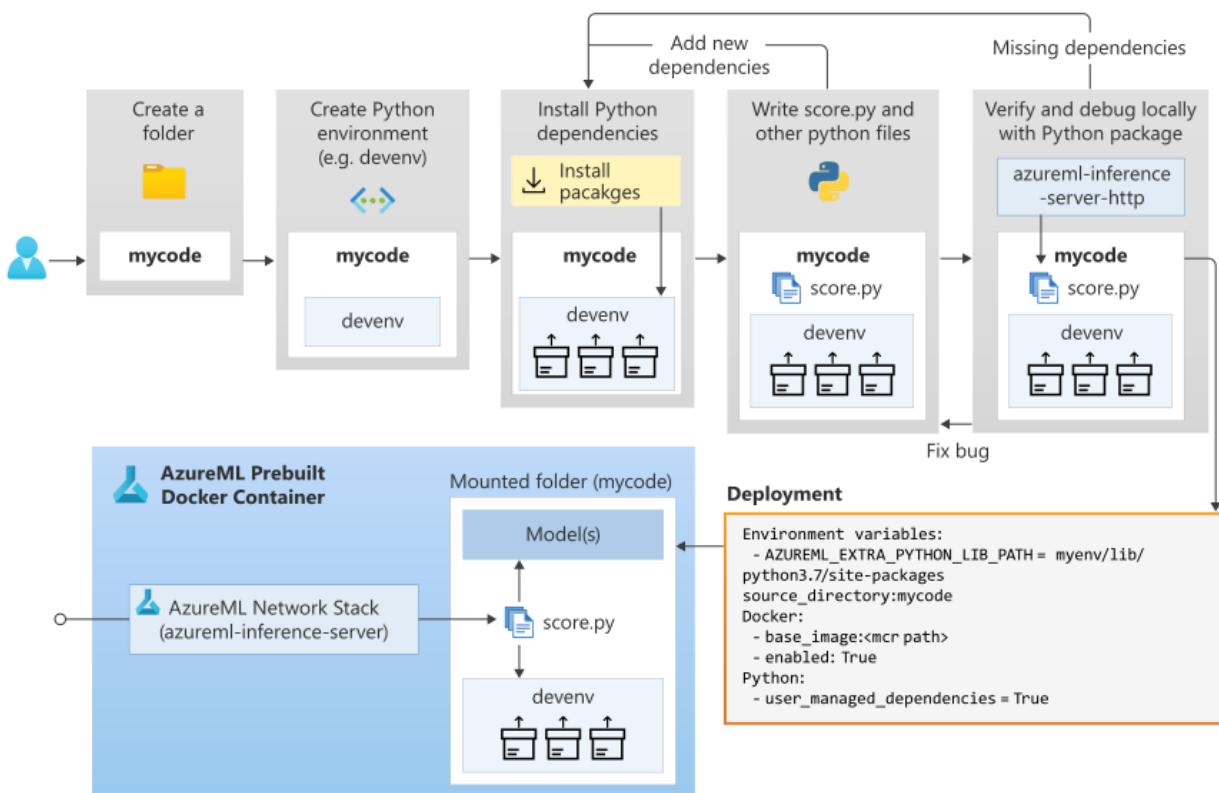
1. Create a virtual environment using [virtualenv](#).
2. Install your Dependencies. If you have a list of dependencies in a `requirements.txt`, for example, you can use that to install with `pip install -r requirements.txt` or just `pip install` individual dependencies.
3. When you specify the `AZUREML_EXTRA_PYTHON_LIB_PATH` environment variable, make sure that you point to the correct site packages directory, which will vary depending on your environment name and Python version. The following code demonstrates setting the path for a virtual environment named `myenv` and Python 3.7:

```
from azureml.core import Environment
from azureml.core.conda_dependencies import CondaDependencies

myenv = Environment(name='my_azureml_env')
myenv.docker.enabled = True
myenv.docker.base_image = <MCR-path>
myenv.python.user_managed_dependencies = True

myenv.environment_variables = {
    "AZUREML_EXTRA_PYTHON_LIB_PATH": "myenv/lib/python3.7/site-packages"
}
```

The following diagram is a visual representation of the pre-installed packages process:



Common problems

The mounting solution will only work when your `myenv` site packages directory contains all of your dependencies. If your local environment is using dependencies installed in a different location, they won't be available in the image.

Here are some things that may cause this problem:

- `virtualenv` creates an isolated environment by default. Once you activate the virtual environment, **global dependencies cannot be used**.
- If you have a `PYTHONPATH` environment variable pointing to your global dependencies, it **may interfere with your virtual environment**. Run `pip list` and `pip freeze` after activating your environment to make sure no unwanted dependencies are in your environment.
- **Conda and `virtualenv` environments can interfere**. Make sure that not to use [Conda environment](#) and `virtualenv` at the same time.

Limitations

`Model.package()`

- The [Model.package\(\)](#) method lets you create a model package in the form of a Docker image or Dockerfile build context. Using `Model.package()` with prebuilt inference docker images triggers an intermediate image build that changes the non-root user to root user.
- We encourage you to use our Python package extensibility solutions. If other dependencies are required (such as `apt` packages), create your own [Dockerfile extending from the inference image](#).

Frequently asked questions

- In the `requirements.txt` extensibility approach is it mandatory for the file name to be `requirements.txt`?

```
myenv.environment_variables = {
    "AZUREML_EXTRA_REQUIREMENTS_TXT": "name of your pip requirements file goes here"
}
```

- Can you summarize the `requirements.txt` approach versus the *mounting approach*?

Start prototyping with the *requirements.txt* approach. After some iteration, when you're confident about which packages (and versions) you need for a successful model deployment, switch to the *Mounting Solution*.

Here's a detailed comparison.

COMPARED ITEM	REQUIREMENTS.TXT (DYNAMIC INSTALLATION)	PACKAGE MOUNT
Solution	Create a <code>requirements.txt</code> that installs the specified packages when the container starts.	Create a local Python environment with all of the dependencies. Mount this directory into container at runtime.
Package Installation	No extra installation (assuming pip already installed)	Virtual environment or conda environment installation.
Virtual environment Setup	No extra setup of virtual environment required, as users can pull the current local user environment with <code>pip freeze</code> as needed to create the <code>requirements.txt</code> .	Need to set up a clean virtual environment, may take extra steps depending on the current user local environment.

COMPARED ITEM	REQUIREMENTS.TXT (DYNAMIC INSTALLATION)	PACKAGE MOUNT
Debugging	Easy to set up and debug server, since dependencies are clearly listed.	Unclean virtual environment could cause problems when debugging of server. For example, it may not be clear if errors come from the environment or user code.
Consistency during scaling out	Not consistent as dependent on external PyPi packages and users pinning their dependencies. These external downloads could be flaky.	Relies solely on user environment, so no consistency issues.

- Why are my `requirements.txt` and mounted dependencies directory not found in the container?

Locally, verify the environment variables are set properly. Next, verify the paths that are specified are spelled properly and exist. Check if you have set your source directory correctly in the [inference config](#) constructor.

- Can I override Python package dependencies in prebuilt inference docker image?

Yes. If you want to use other version of Python package that is already installed in an inference image, our extensibility solution will respect your version. Make sure there are no conflicts between the two versions.

Best Practices

- Refer to the [Load registered model](#) docs. When you register a model directory, don't include your scoring script, your mounted dependencies directory, or `requirements.txt` within that directory.
- For more information on how to load a registered or local model, see [Where and how to deploy](#).

Bug Fixes

2021-07-26

- `AZUREML_EXTRA_REQUIREMENTS_TXT` and `AZUREML_EXTRA_PYTHON_LIB_PATH` are now always relative to the directory of the score script. For example, if both the requirements.txt and score script is in `my_folder`, then `AZUREML_EXTRA_REQUIREMENTS_TXT` will need to be set to requirements.txt. No longer will `AZUREML_EXTRA_REQUIREMENTS_TXT` be set to `my_folder/requirements.txt`.

Next steps

To learn more about deploying a model, see [How to deploy a model](#).

To learn how to troubleshoot prebuilt docker image deployments, see [how to troubleshoot prebuilt Docker image deployments](#).

Extend a prebuilt Docker image

9/21/2022 • 3 minutes to read • [Edit Online](#)

In some cases, the [prebuilt Docker images for model inference](#) and [extensibility](#) solutions for Azure Machine Learning may not meet your inference service needs.

In this case, you can use a Dockerfile to create a new image, using one of the prebuilt images as the starting point. By extending from an existing prebuilt Docker image, you can use the Azure Machine Learning network stack and libraries without creating an image from scratch.

Benefits and tradeoffs

Using a Dockerfile allows for full customization of the image before deployment. It allows you to have maximum control over what dependencies or environment variables, among other things, are set in the container.

The main tradeoff for this approach is that an extra image build will take place during deployment, which slows down the deployment process. If you can use the [Python package extensibility](#) method, deployment will be faster.

Prerequisites

- An Azure Machine Learning workspace. For a tutorial on creating a workspace, see [Get started with Azure Machine Learning](#).
- Familiarity with authoring a [Dockerfile](#).
- Either a local working installation of [Docker](#), including the `docker` CLI, OR an Azure Container Registry (ACR) associated with your Azure Machine Learning workspace.

WARNING

The Azure Container Registry for your workspace is created the first time you train or deploy a model using the workspace. If you've created a new workspace, but not trained or created a model, no Azure Container Registry will exist for the workspace.

Create and build Dockerfile

Below is a sample Dockerfile that uses an Azure Machine Learning prebuilt Docker image as a base image:

```
FROM mcr.microsoft.com/azureml/<image_name>:<tag>

COPY requirements.txt /tmp/requirements.txt

RUN pip install -r /tmp/requirements.txt
```

Then put the above Dockerfile into the directory with all the necessary files and run the following command to build the image:

```
docker build -f <above dockerfile> -t <image_name>:<tag> .
```

TIP

More details about `docker build` can be found here in the [Docker documentation](#).

If the `docker build` command isn't available locally, use the Azure Container Registry ACR for your Azure Machine Learning Workspace to build the Docker image in the cloud. For more information, see [Tutorial: Build and deploy container images with Azure Container Registry](#).

IMPORTANT

Microsoft recommends that you first validate that your Dockerfile works locally before trying to create a custom base image via Azure Container Registry.

The following sections contain more specific details on the Dockerfile.

Install extra packages

If there are any other `apt` packages that need to be installed in the Ubuntu container, you can add them in the Dockerfile. The following example demonstrates how to use the `apt-get` command from a Dockerfile:

```
FROM <prebuilt docker image from MCR>

# Switch to root to install apt packages
USER root:root

RUN apt-get update && \
    apt-get install -y \
    <package-1> \
    ...
    <package-n> && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/*

# Switch back to non-root user
USER dockeruser
```

You can also install addition pip packages from a Dockerfile. The following example demonstrates using

```
pip install :
```

```
RUN pip install <library>
```

Build model and code into images

If the model and code need to be built into the image, the following environment variables need to be set in the Dockerfile:

- `AZUREML_ENTRY_SCRIPT` : The entry script of your code. This file contains the `init()` and `run()` methods.
- `AZUREML_MODEL_DIR` : The directory that contains the model file(s). The entry script should use this directory as the root directory of the model.

The following example demonstrates setting these environment variables in the Dockerfile:

```
FROM <prebuilt docker image from MCR>

# Code
COPY <local_code_directory> /var/azureml-app
ENV AZUREML_ENTRY_SCRIPT=<entryscript_file_name>

# Model
COPY <model_directory> /var/azureml-app/azureml-models
ENV AZUREML_MODEL_DIR=/var/azureml-app/azureml-models
```

Example Dockerfile

The following example demonstrates installing `apt` packages, setting environment variables, and including code and models as part of the Dockerfile:

```
FROM mcr.microsoft.com/azureml/pytorch-1.6-ubuntu18.04-py37-cpu-inference:latest

USER root:root

# Install libpng-tools and opencv
RUN apt-get update && \
    apt-get install -y \
    libpng-tools \
    python3-opencv && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/*

# Switch back to non-root user
USER dockeruser

# Code
COPY code /var/azureml-app
ENV AZUREML_ENTRY_SCRIPT=score.py

# Model
COPY model /var/azureml-app/azureml-models
ENV AZUREML_MODEL_DIR=/var/azureml-app/azureml-models
```

Next steps

To use a Dockerfile with the Azure Machine Learning Python SDK, see the following documents:

- [Use your own local Dockerfile](#)
- [Use a pre-built Docker image and create a custom base image](#)

To learn more about deploying a model, see [How to deploy a model](#).

To learn how to troubleshoot prebuilt docker image deployments, see [how to troubleshoot prebuilt Docker image deployments](#).

Troubleshooting prebuilt docker images for inference

9/21/2022 • 3 minutes to read • [Edit Online](#)

Learn how to troubleshoot problems you may see when using prebuilt docker images for inference with Azure Machine Learning.

IMPORTANT

Using [Python package extensibility for prebuilt Docker images](#) with Azure Machine Learning is currently in preview. Preview functionality is provided "as-is", with no guarantee of support or service level agreement. For more information, see the [Supplemental terms of use for Microsoft Azure previews](#).

Model deployment failed

If model deployment fails, you won't see logs in [Azure Machine Learning studio](#) and `service.get_logs()` will return None. If there is a problem in the `init()` function of `score.py`, `service.get_logs()` will return logs for the same.

So you'll need to run the container locally using one of the commands shown below and replace `<mcr-path>` with an image path. For a list of the images and paths, see [Prebuilt Docker images for inference](#).

Mounting extensibility solution

Go to the directory containing `score.py` and run:

```
docker run -it -v $(pwd):/var/azureml-app -e AZUREML_EXTRA_PYTHON_LIB_PATH="myenv/lib/python3.7/site-packages" <mcr-path>
```

requirements.txt extensibility solution

Go to the directory containing `score.py` and run:

```
docker run -it -v $(pwd):/var/azureml-app -e AZUREML_EXTRA_REQUIREMENTS_TXT="requirements.txt" <mcr-path>
```

Enable local debugging

The local inference server allows you to quickly debug your entry script (`score.py`). In case the underlying score script has a bug, the server will fail to initialize or serve the model. Instead, it will throw an exception & the location where the issues occurred. [Learn more about Azure Machine Learning inference HTTP Server](#)

For common model deployment issues

For problems when deploying a model from Azure Machine Learning to Azure Container Instances (ACI) or Azure Kubernetes Service (AKS), see [Troubleshoot model deployment](#).

init() or run() failing to write a file

HTTP server in our Prebuilt Docker Images run as *non-root user*, it may not have access right to all directories.

Only write to directories you have access rights to. For example, the `/tmp` directory in the container.

Extra Python packages not installed

- Check if there's a typo in the environment variable or file name.
- Check the container log to see if `pip install -r <your_requirements.txt>` is installed or not.
- Check if source directory is set correctly in the [inference config](#) constructor.
- If installation not found and log says "file not found", check if the file name shown in the log is correct.
- If installation started but failed or timed out, try to install the same `requirements.txt` locally with same Python and pip version in clean environment (that is, no cache directory;
`pip install --no-cache-dir -r requirements.txt`). See if the problem can be reproduced locally.

Mounting solution failed

- Check if there's a typo in the environment variable or directory name.
- The environment variable must be set to the relative path of the `score.py` file.
- Check if source directory is set correctly in the [inference config](#) constructor.
- The directory needs to be the "site-packages" directory of the environment.
- If `score.py` still returns `ModuleNotFoundError` and the module is supposed to be in the directory mounted, try to print the `sys.path` in `init()` or `run()` to see if any path is missing.

Building an image based on the prebuilt Docker image failed

- If failed during apt package installation, check if the user has been set to root before running the apt command? (Make sure switch back to non-root user)

Run doesn't complete on GPU local deployment

GPU base images can't be used for local deployment, unless the local deployment is on an Azure Machine Learning compute instance. GPU base images are supported only on Microsoft Azure Services such as Azure Machine Learning compute clusters and instances, Azure Container Instance (ACI), Azure VMs, or Azure Kubernetes Service (AKS).

Image built based on the prebuilt Docker image can't boot up

- The non-root user needs to be `dockeruser`. Otherwise, the owner of the following directories must be set to the user name you want to use when running the image:

```
/var/run
/var/log
/var/lib/nginx
/run
/opt/miniconda
/var/azureml-app
```

- If the `ENTRYPOINT` has been changed in the new built image, then the HTTP server and related components need to be loaded by `runsvdir /var/run`

Next steps

- [Add Python packages to prebuilt images.](#)
- [Use a prebuilt package as a base for a new Dockerfile.](#)

Collect data from models in production

9/21/2022 • 4 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article shows how to collect data from an Azure Machine Learning model deployed on an Azure Kubernetes Service (AKS) cluster. The collected data is then stored in Azure Blob storage.

Once collection is enabled, the data you collect helps you:

- [Monitor data drifts](#) on the production data you collect.
- Analyze collected data using [Power BI](#) or [Azure Databricks](#)
- Make better decisions about when to retrain or optimize your model.
- Retrain your model with the collected data.

What is collected and where it goes

The following data can be collected:

- Model input data from web services deployed in an AKS cluster. Voice audio, images, and video are *not* collected.
- Model predictions using production input data.

NOTE

Preaggregation and precalculations on this data are not currently part of the collection service.

The output is saved in Blob storage. Because the data is added to Blob storage, you can choose your favorite tool to run the analysis.

The path to the output data in the blob follows this syntax:

```
/modeldata/<subscriptionid>/<resourcegroup>/<workspace>/<webservice>/<model>/<version>/<designation>/<year>/<month>/<day>/data.csv  
# example: /modeldata/1a2b3c4d-5e6f-7g8h-9i10-j11k12l13m14/myresourcegrp/myWorkspace/aks-w-collv9/best_model/10/inputs/2018/12/31/data.csv
```

NOTE

In versions of the Azure Machine Learning SDK for Python earlier than version 0.1.0a16, the `designation` argument is named `identifier`. If you developed your code with an earlier version, you need to update it accordingly.

Prerequisites

- If you don't have an Azure subscription, create a [free account](#) before you begin.
- An Azure Machine Learning workspace, a local directory containing your scripts, and the Azure Machine Learning SDK for Python must be installed. To learn how to install them, see [How to configure a](#)

development environment.

- You need a trained machine-learning model to be deployed to AKS. If you don't have a model, see the [Train image classification model](#) tutorial.
- You need an AKS cluster. For information on how to create one and deploy to it, see [Deploy machine learning models to Azure](#).
- [Set up your environment](#) and install the [Azure Machine Learning Monitoring SDK](#).
- Use a docker image based on Ubuntu 18.04, which is shipped with `libssl 1.0.0`, the essential dependency of `modeldatacollector`. You can refer to [prebuilt images](#).

Enable data collection

You can enable [data collection](#) regardless of the model you deploy through Azure Machine Learning or other tools.

To enable data collection, you need to:

1. Open the scoring file.
2. Add the following code at the top of the file:

```
from azureml.monitoring import ModelDataCollector
```

3. Declare your data collection variables in your `init` function:

```
global inputs_dc, prediction_dc
inputs_dc = ModelDataCollector("best_model", designation="inputs", feature_names=["feat1", "feat2",
"feat3", "feat4", "feat5", "feat6"])
prediction_dc = ModelDataCollector("best_model", designation="predictions", feature_names=
["prediction1", "prediction2"])
```

`CorrelationId` is an optional parameter. You don't need to use it if your model doesn't require it. Use of `CorrelationId` does help you more easily map with other data, such as `LoanNumber` or `CustomerId`.

The `Identifier` parameter is later used for building the folder structure in your blob. You can use it to differentiate raw data from processed data.

4. Add the following lines of code to the `run(input_df)` function:

```
data = np.array(data)
result = model.predict(data)
inputs_dc.collect(data) #this call is saving our input data into Azure Blob
prediction_dc.collect(result) #this call is saving our prediction data into Azure Blob
```

5. Data collection is *not* automatically set to `true` when you deploy a service in AKS. Update your configuration file, as in the following example:

```
aks_config = AksWebservice.deploy_configuration(collect_model_data=True)
```

You can also enable Application Insights for service monitoring by changing this configuration:

```
aks_config = AksWebservice.deploy_configuration(collect_model_data=True, enable_app_insights=True)
```

6. To create a new image and deploy the machine learning model, see [Deploy machine learning models to Azure](#).

7. Add the 'Azure-Monitoring' pip package to the conda-dependencies of the web service environment:

```
env = Environment('webserviceenv')
env.python.conda_dependencies = CondaDependencies.create(conda_packages=['numpy'],pip_packages=['azureml-defaults','azureml-monitoring','inference-schema[numpy-support]'])
```

Disable data collection

You can stop collecting data at any time. Use Python code to disable data collection.

```
## replace <service_name> with the name of the web service
<service_name>.update(collect_model_data=False)
```

Validate and analyze your data

You can choose a tool of your preference to analyze the data collected in your Blob storage.

Quickly access your blob data

1. Sign in to [Azure portal](#).
2. Open your workspace.
3. Select **Storage**.

The screenshot shows the Azure Machine Learning service workspace overview page. On the left, there's a sidebar with links: Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. The main area has a search bar and a 'Delete' button. Under 'Resource group', it shows '<resource group>'. Below that, 'Location' is listed as 'East US 2'. Under 'Subscription', it shows '<subscription name>' and 'Subscription ID' with '<SubscriptionID>'. On the right, there's a 'Storage' section with fields for 'Registry' (<registry name>), 'Key Vault' (<key vault name>), and 'Application Insights' (<App Insights name>). The field for 'Storage' is labeled '<blob storage name>' and is highlighted with a red box.

4. Follow the path to the blob's output data with this syntax:

```
/modeldata/<subscriptionid>/<resourcegroup>/<workspace>/<webservice>/<model>/<version>/<designation>/<year>/<month>/<day>/data.csv
# example: /modeldata/1a2b3c4d-5e6f-7g8h-9i10-j11k12l13m14/myresourcegrp/myworkspace/aks-w-
collv9/best_model/10/inputs/2018/12/31/data.csv
```

Analyze model data using Power BI

1. Download and open [Power BI Desktop](#).
2. Select **Get Data** and select [Azure Blob Storage](#).

X

Get Data

X

All

Azure

Online Services

All

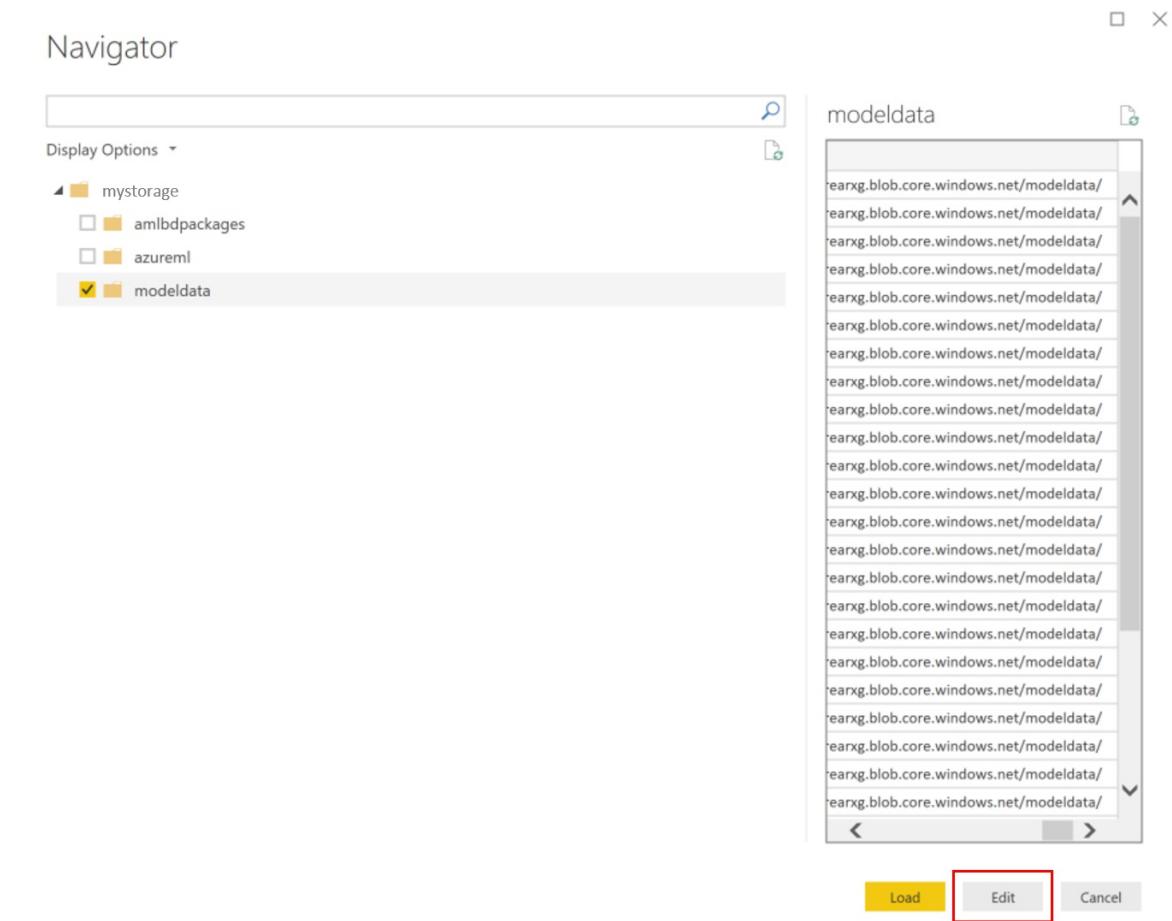
- Azure SQL database
- Azure SQL Data Warehouse
- Azure Analysis Services database
- Azure Blob Storage
- Azure Table Storage
- Azure Cosmos DB (Beta)
- Azure Data Lake Store
- Azure HDInsight (HDFS)
- Azure HDInsight Spark
- Microsoft Azure Consumption Insights (Beta)
- Azure KustoDB (Beta)

[Certified Connectors](#)

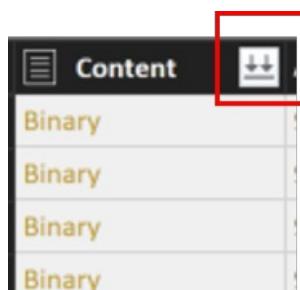
Connect Cancel

3. Add your storage account name and enter your storage key. You can find this information by selecting **Settings > Access keys** in your blob.

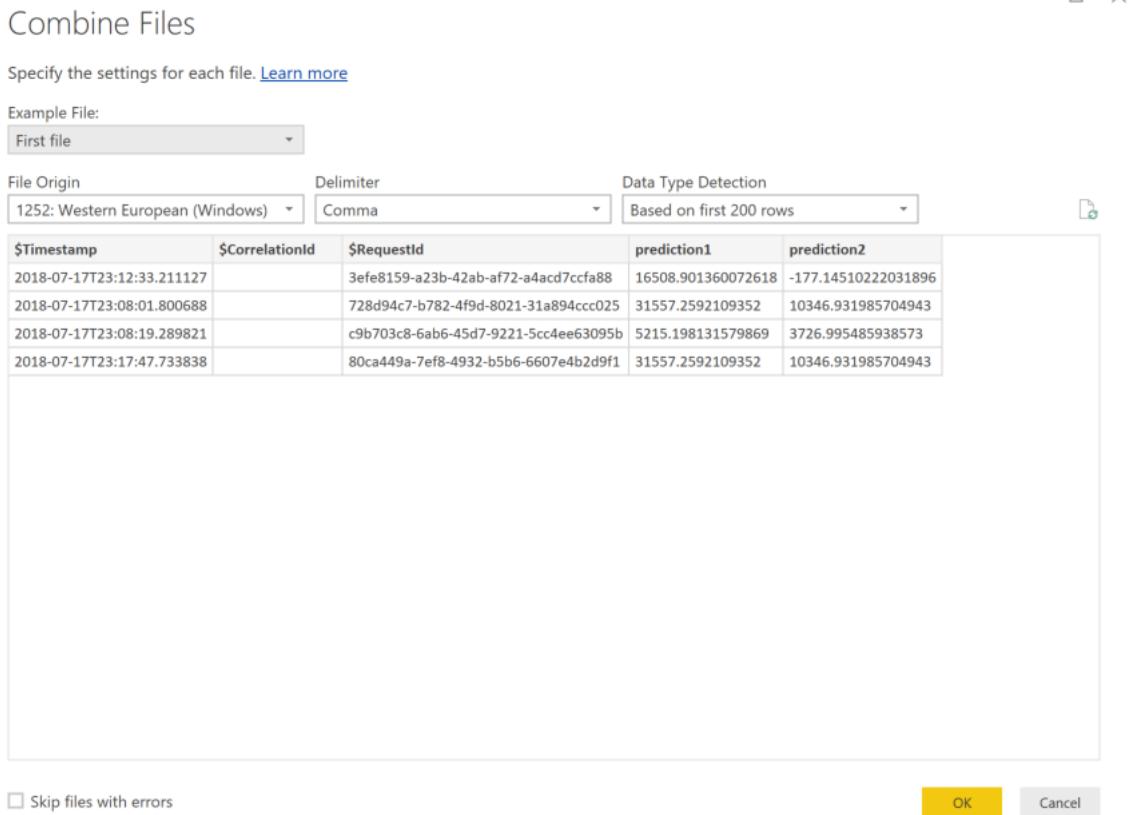
4. Select the **model data** container and select **Edit**.



- In the query editor, click under the **Name** column and add your storage account.
 - Enter your model path into the filter. If you want to look only into files from a specific year or month, just expand the filter path. For example, to look only into March data, use this filter path:
`/modeldata/<subscriptionid>/<resourcegroupname>/<workspacename>/<webservicename>/<modelname>/<modelversion>/<designation>/<year>/3`
 - Filter the data that is relevant to you based on **Name** values. If you stored predictions and inputs, you need to create a query for each.
 - Select the downward double arrows next to the **Content** column heading to combine the files.



9. Select OK. The data preloads.



10. Select **Close and Apply**.

11. If you added inputs and predictions, your tables are automatically ordered by **RequestId** values.

12. Start building your custom reports on your model data.

Analyze model data using Azure Databricks

1. Create an [Azure Databricks workspace](#).
2. Go to your Databricks workspace.
3. In your Databricks workspace, select **Upload Data**.



Azure Databricks



Explore the Quickstart Tutorial

Spin up a cluster, run queries on preloaded data, and display results in 5 minutes.

Quickly im

Common Tasks

New Notebook

Upload Data

Create Table

New Cluster

New Job

Import Library

Read Documentation

Recents

2018-11-(

2018-06-(

setup2

2018-07-(

setup

4. Select **Create New Table** and select **Other Data Sources > Azure Blob Storage > Create Table in Notebook**.

Create New Table

Data source

Upload File DBFS Other Data Sources

Connector

Azure Blob Storage

Create Table in Notebook

5. Update the location of your data. Here is an example:

```
file_location = "wasbs://mycontainer@storageaccountname.blob.core.windows.net/modeldata/1a2b3c4d-5e6f-7g8h-9i10-j11k12l13m14/myresourcegrp/myworkspace/aks-w-col1v9/best_model/10/inputs/2018/**/data.csv"
file_type = "csv"
```

Cmd. 2

Step 1: Set the data location and type

There are two ways to access Azure Blob storage: account keys and shared access signatures (SAS).

To get started, we need to set the location and type of the file.

Cmd. 3

```
1 storage_account_name = "mystorage"
2 storage_account_access_key = "Jhsduhwe908rjjfoieudnf 98h v9udfhgb987yh g908ufgb98yfgb9 ufgb78gy8 gfo89ug98yfdg7yfg8ytg9fyg87"
```

Cmd. 4

```
1 file_location = "wasbs://mycontainer@storageaccountname.blob.core.windows.net/modeldata/1a2b3c4d-5e6f-7g8h-9i10-j11k12l13m14/myresourcegrp/myworkspace/aks-w-col1v9/best_model/10/inputs/2018/**/data.csv"
2 file_type = "csv"
```

Cmd. 5

```
1 spark.conf.set(
2   "fs.azure.account.key."+storage_account_name+".blob.core.windows.net",
3   storage_account_access_key)
```

6. Follow the steps on the template to view and analyze your data.

Next steps

[Detect data drift](#) on the data you have collected.

Monitor and collect data from ML web service endpoints

9/21/2022 • 5 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to collect data from models deployed to web service endpoints in Azure Kubernetes Service (AKS) or Azure Container Instances (ACI). Use [Azure Application Insights](#) to collect the following data from an endpoint:

- Output data
- Responses
- Request rates, response times, and failure rates
- Dependency rates, response times, and failure rates
- Exceptions

The [enable-app-insights-in-production-service.ipynb](#) notebook demonstrates concepts in this article.

Learn how to run notebooks by following the article [Use Jupyter notebooks to explore this service](#).

IMPORTANT

The information in this article relies on the Azure Application Insights instance that was created with your workspace. If you deleted this Application Insights instance, there is no way to re-create it other than deleting and recreating the workspace.

TIP

If you are using online endpoints instead, use the information in the [Monitor online endpoints](#) article instead.

Prerequisites

- An Azure subscription - try the [free or paid version of Azure Machine Learning](#).
- An Azure Machine Learning workspace, a local directory that contains your scripts, and the Azure Machine Learning SDK for Python installed. To learn more, see [How to configure a development environment](#).
- A trained machine learning model. To learn more, see the [Train image classification model](#) tutorial.

Configure logging with the Python SDK

In this section, you learn how to enable Application Insight logging by using the Python SDK.

Update a deployed service

Use the following steps to update an existing web service:

1. Identify the service in your workspace. The value for `ws` is the name of your workspace

```
from azureml.core.webservice import Webservice  
aks_service= Webservice(ws, "my-service-name")
```

2. Update your service and enable Azure Application Insights

```
aks_service.update(enable_app_insights=True)
```

Log custom traces in your service

IMPORTANT

Azure Application Insights only logs payloads of up to 64kb. If this limit is reached, you may see errors such as out of memory, or no information may be logged. If the data you want to log is larger 64kb, you should instead store it to blob storage using the information in [Collect Data for models in production](#).

For more complex situations, like model tracking within an AKS deployment, we recommend using a third-party library like [OpenCensus](#).

To log custom traces, follow the standard deployment process for AKS or ACI in the [How to deploy and where](#) document. Then, use the following steps:

1. Update the scoring file by adding print statements to send data to Application Insights during inference.
For more complex information, such as the request data and the response, use a JSON structure.

The following example `score.py` file logs when the model was initialized, input and output during inference, and the time any errors occur.

```

import pickle
import json
import numpy
from sklearn.externals import joblib
from sklearn.linear_model import Ridge
from azureml.core.model import Model
import time

def init():
    global model
    #Print statement for appinsights custom traces:
    print ("model initialized" + time.strftime("%H:%M:%S"))

    # note here "sklearn_regression_model.pkl" is the name of the model registered under the
    workspace
    # this call should return the path to the model.pkl file on the local disk.
    model_path = Model.get_model_path(model_name = 'sklearn_regression_model.pkl')

    # deserialize the model file back into a sklearn model
    model = joblib.load(model_path)

# note you can pass in multiple rows for scoring
def run(raw_data):
    try:
        data = json.loads(raw_data)['data']
        data = numpy.array(data)
        result = model.predict(data)
        # Log the input and output data to appinsights:
        info = {
            "input": raw_data,
            "output": result.tolist()
        }
        print(json.dumps(info))
        # you can return any datatype as long as it is JSON-serializable
        return result.tolist()
    except Exception as e:
        error = str(e)
        print (error + time.strftime("%H:%M:%S"))
        return error

```

2. Update the service configuration, and make sure to enable Application Insights.

```
config = Webservice.deploy_configuration(enable_app_insights=True)
```

3. Build an image and deploy it on AKS or ACI. For more information, see [How to deploy and where](#).

Disable tracking in Python

To disable Azure Application Insights, use the following code:

```
## replace <service_name> with the name of the web service
<service_name>.update(enable_app_insights=False)
```

Configure logging with Azure Machine Learning studio

You can also enable Azure Application Insights from Azure Machine Learning studio. When you're ready to deploy your model as a web service, use the following steps to enable Application Insights:

1. Sign in to the studio at <https://ml.azure.com>.

2. Go to **Models** and select the model you want to deploy.

3. Select **+Deploy**.

4. Populate the **Deploy model** form.

5. Expand the **Advanced** menu.

Deploy a model

Customers should not include personal data or other sensitive information in fields marked with because the content in these fields may be logged and shared across Microsoft systems to facilitate operations and troubleshooting. [Learn more](#) X

Name *

Description

Compute type * *

ACI ▼

Models: AutoML6e225a7b963:1

Enable authentication

Entry script file *

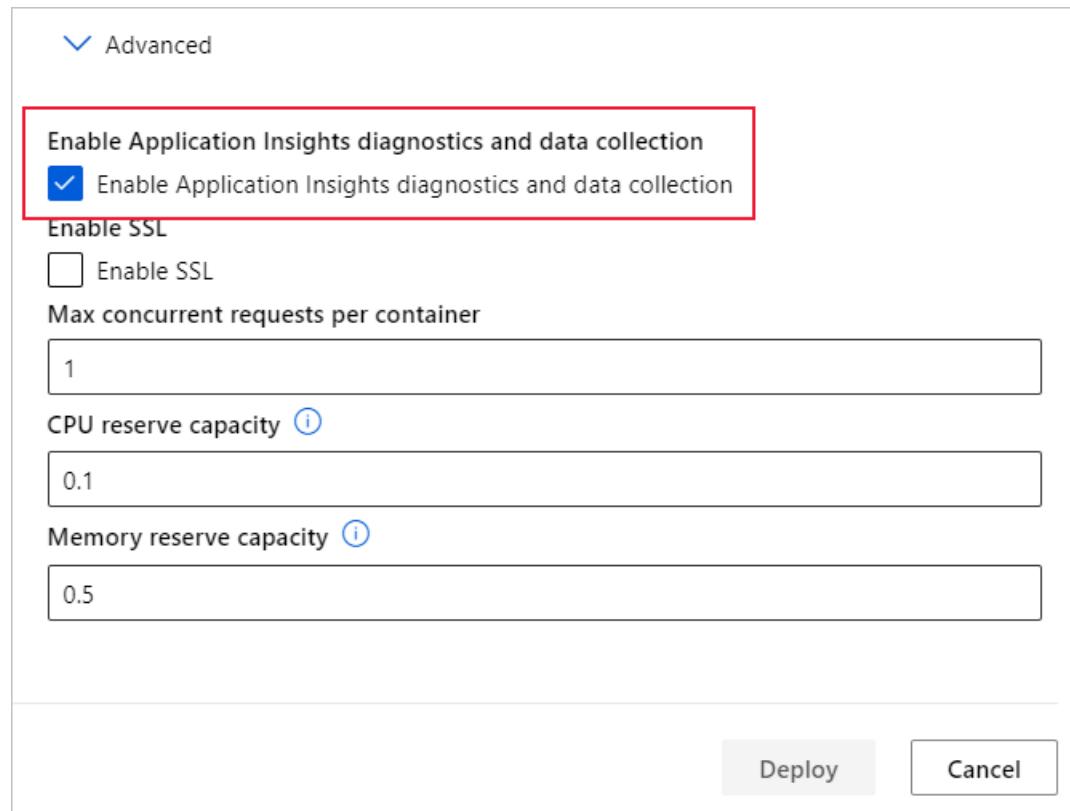
*

Conda dependencies file * *

*

Dependencies

6. Select **Enable Application Insights diagnostics and data collection**.



View metrics and logs

Query logs for deployed models

Logs of online endpoints are customer data. You can use the `get_logs()` function to retrieve logs from a previously deployed web service. The logs may contain detailed information about any errors that occurred during deployment.

```
from azureml.core import Workspace
from azureml.core.webservice import Webservice

ws = Workspace.from_config()

# load existing web service
service = Webservice(name="service-name", workspace=ws)
logs = service.get_logs()
```

If you have multiple Tenants, you may need to add the following authenticate code before

```
ws = Workspace.from_config()
```

```
from azureml.core.authentication import InteractiveLoginAuthentication
interactive_auth = InteractiveLoginAuthentication(tenant_id="the tenant_id in which your workspace resides")
```

View logs in the studio

Azure Application Insights stores your service logs in the same resource group as the Azure Machine Learning workspace. Use the following steps to view your data using the studio:

1. Go to your Azure Machine Learning workspace in the [studio](#).
2. Select **Endpoints**.
3. Select the deployed service.
4. Select the [Application Insights url](#) link.

Microsoft Azure Machine Learning

my-ws > Endpoints > aks-service-appinsights

aks-service-appinsights

--

CPU
0.1

Memory
0.5 GB

Application Insights enabled
true

Application Insights url
https://portal.azure.com#@microsoft.onmicrosoft.com/resource/subscriptions/xxx_XXXXXX...

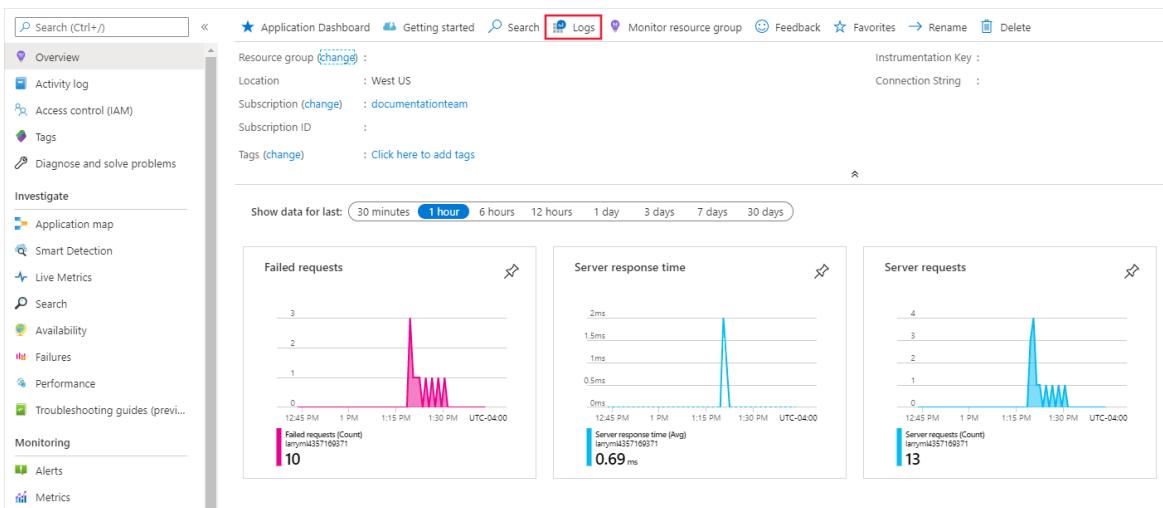
Event Hubs enabled
false

Storage enabled
false

Autoscale enabled
true

Min replicas
1

5. In Application Insights, from the **Overview** tab or the **Monitoring** section, select **Logs**.



6. To view information logged from the score.py file, look at the **traces** table. The following query searches for logs where the **input** value was logged:

```
traces
| where customDimensions contains "input"
| limit 10
```

The screenshot shows the Azure Application Insights Log Analytics workspace. On the left, there's a navigation sidebar with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Investigate (Application map, Smart Detection, Live Metrics, Search, Availability, Failures, Performance, Troubleshooting guides), Monitoring (Alerts, Metrics, Logs, Workbooks), Usage (Users, Sessions, Events), and a favorites section. The main area shows a query editor with the following code:

```
traces
| where timestamp > ago(24h) and customDimensions contains "input"
| limit 10
```

The results table has columns: timestamp [Local Time], message, severityLevel, itemType, and customDimensions. It shows three records from June 8, 2020, at 11:56 AM, 11:56:49 AM, and 2:08 PM. The customDimensions column shows JSON data for each entry.

For more information on how to use Azure Application Insights, see [What is Application Insights?](#).

Web service metadata and response data

IMPORTANT

Azure Application Insights only logs payloads of up to 64kb. If this limit is reached then you may see errors such as out of memory, or no information may be logged.

To log web service request information, add `print` statements to your score.py file. Each `print` statement results in one entry in the Application Insights trace table under the message `STDOUT`. Application Insights stores the `print` statement outputs in `customDimensions` and in the `Contents` trace table. Printing JSON strings produces a hierarchical data structure in the trace output under `Contents`.

Export data for retention and processing

IMPORTANT

Azure Application Insights only supports exports to blob storage. For more information on the limits of this implementation, see [Export telemetry from App Insights](#).

Use Application Insights' [continuous export](#) to export data to a blob storage account where you can define retention settings. Application Insights exports the data in JSON format.

Dashboard > azureml1588320359 - Continuous export

azureml1588320359 - Continuous export

Application Insights

Search (Ctrl+ /)

Add Refresh Help

Usage

- Users
- Sessions
- Events
- Funnels
- User Flows
- Retention
- Impact
- Cohorts
- More

Configure

- Properties
- Smart Detection settings
- Usage and estimated costs
- Continuous export
- Performance Testing
- API Access

STATUS	STORAGE ACCOUNT	LAST EXPORT
✓	copeterstest3321914124	9/27/2019, 8:55:10 AM
✓	chiworkspace3965959982	9/27/2019, 8:55:10 AM
✓	chichesstoragebmskrivc	9/27/2019, 8:55:47 AM
✓	azureml5386059871	9/27/2019, 8:55:47 AM

Next steps

In this article, you learned how to enable logging and view logs for web service endpoints. Try these articles for next steps:

- [How to deploy a model to an AKS cluster](#)
- [How to deploy a model to Azure Container Instances](#)
- [MLOps: Manage, deploy, and monitor models with Azure Machine Learning](#) to learn more about leveraging data collected from models in production. Such data can help to continually improve your machine learning process.

Use the studio to deploy models trained in the designer

9/21/2022 • 7 minutes to read • [Edit Online](#)

In this article, you learn how to deploy a designer model as an online (real-time) endpoint in Azure Machine Learning studio.

Once registered or downloaded, you can use designer trained models just like any other model. Exported models can be deployed in use cases such as internet of things (IoT) and local deployments.

Deployment in the studio consists of the following steps:

1. Register the trained model.
2. Download the entry script and conda dependencies file for the model.
3. (Optional) Configure the entry script.
4. Deploy the model to a compute target.

You can also deploy models directly in the designer to skip model registration and file download steps. This can be useful for rapid deployment. For more information see, [Deploy a model with the designer](#).

Models trained in the designer can also be deployed through the SDK or command-line interface (CLI). For more information, see [Deploy your existing model with Azure Machine Learning](#).

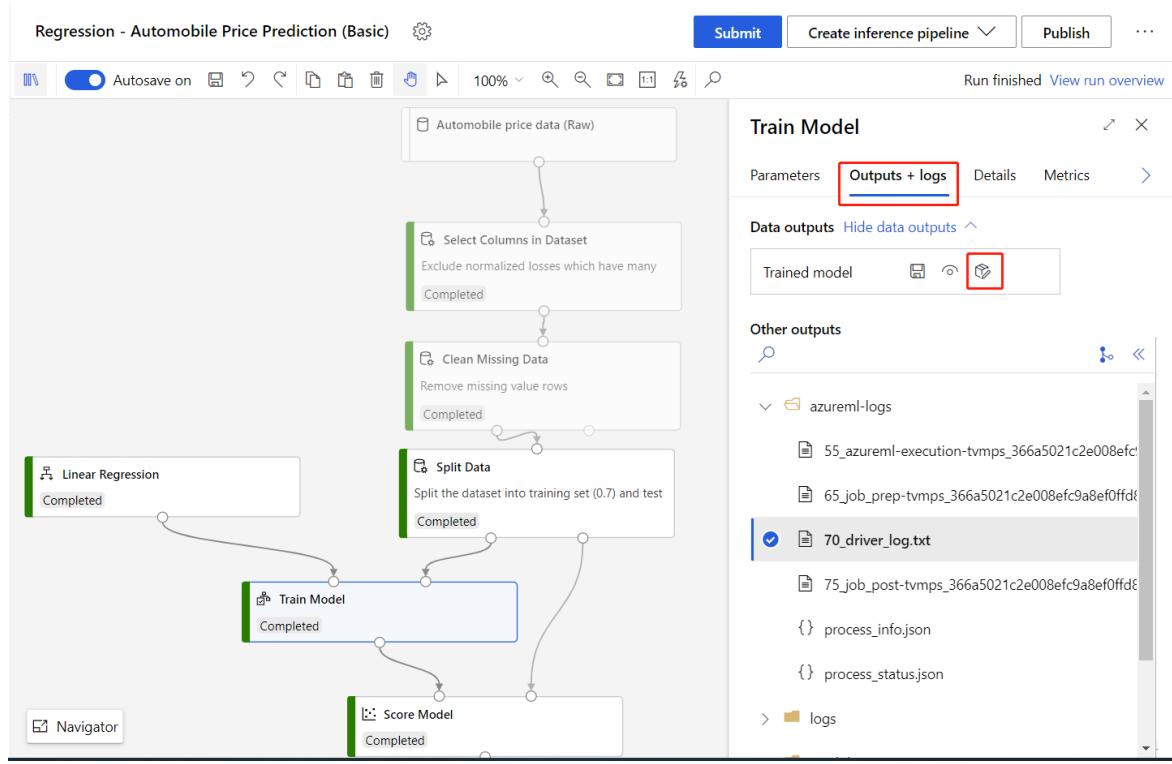
Prerequisites

- [An Azure Machine Learning workspace](#)
- A completed training pipeline containing one of following components:
 - [Train Model Component](#)
 - [Train Anomaly Detection Model component](#)
 - [Train Clustering Model component](#)
 - [Train Pytorch Model component](#)
 - [Train SVD Recommender component](#)
 - [Train Vowpal Wabbit Model component](#)
 - [Train Wide & Deep Model component](#)

Register the model

After the training pipeline completes, register the trained model to your Azure Machine Learning workspace to access the model in other projects.

1. Select the [Train Model component](#).
2. Select the **Outputs + logs** tab in the right pane.
3. Select the **Register Model** icon .



4. Enter a name for your model, then select **Save**.

After registering your model, you can find it in the **Models** asset page in the studio.

Model List						
	Name	Version	Experiment	Job (Run ID)	Created on	Tags
	PricePredictionModel	1	Sample1	84a9bbe3-d387-40b7-8eb7-8d...	Aug 26, 2020 2:21 PM	CreatedByAMLStudio: true
	amlstudio-rename-sample-test	1	Sample1	dd215f48-e5ea-4c8e-9c6a-26...	Aug 24, 2020 1:44 PM	CreatedByAMLStudio: true
	amlstudio-rename-sample-1	3	Sample1	dd215f48-e5ea-4c8e-9c6a-26...	Aug 24, 2020 1:43 PM	CreatedByAMLStudio: true
	amlstudio-testdeployssample1	1	Sample1	dd215f48-e5ea-4c8e-9c6a-26...	Aug 24, 2020 1:41 PM	CreatedByAMLStudio: true
	amlstudio-rename-sample-1	2	Sample1	dd215f48-e5ea-4c8e-9c6a-26...	Aug 14, 2020 7:42 AM	CreatedByAMLStudio: true

Download the entry script file and conda dependencies file

You need the following files to deploy a model in Azure Machine Learning studio:

- **Entry script file** - loads the trained model, processes input data from requests, does real-time inferences, and returns the result. The designer automatically generates a `score.py` entry script file when the **Train Model** component completes.
- **Conda dependencies file** - specifies which pip and conda packages your webservice depends on. The designer automatically creates a `conda_env.yaml` file when the **Train Model** component completes.

You can download these two files in the right pane of the **Train Model** component:

1. Select the **Train Model** component.
2. In the **Outputs + logs** tab, select the folder `trained_model_outputs`.
3. Download the `conda_env.yaml` file and `score.py` file.

The screenshot shows the Azure Machine Learning Studio interface with the following details:

- Top Bar:** Regression - Automobile Price Prediction (Basic), **Submit**, **Create inference pipeline**, **Publish**, and three ellipsis buttons.
- Toolbar:** Autosave on, various icons for file operations like Open, Save, Print, and a magnifying glass.
- Main Area:** A flowchart of the data pipeline:
 - An input dataset "Automobile price data" feeds into a "Select Columns in Data" step (Completed).
 - "Select Columns in Data" feeds into "Clean Missing Data" (Completed).
 - "Clean Missing Data" feeds into "Split Data" (Completed).
 - "Split Data" splits the data into training and testing sets, which then feed into a "Linear Regression" step (Completed).
 - "Linear Regression" feeds into a "Train Model" step (Completed).
 - "Train Model" outputs a "Trained_model".
- Right Panel - Train Model:** Shows the "Outputs + logs" tab selected. It displays the "Trained model" output and other outputs:
 - Data outputs:** Hide data outputs ▾
 - Trained model
 - Other outputs:**
 - _metadata.yaml
 - _samples.json
 - _schema.json
 - conda_env.yaml** (highlighted with a red box)
 - data.learner** (highlighted with a red box)
 - data.metadata**
 - model_spec.yaml**
 - score.py** (highlighted with a red box)
 - Trained_model**

Alternatively, you can download the files from the **Models** asset page after registering your model:

1. Navigate to the **Models** asset page.
 2. Select the model you want to deploy.
 3. Select the **Artifacts** tab.
 4. Select the `trained_model_outputs` folder.
 5. Download the `conda_env.yaml` file and `score.py` file.

```
PY score.py
1 import os
2 import json
3
4 from azureml.studio.core.io.model_directory import ModelDirectory
5 from pathlib import Path
6 from azureml.studio.modules.ml.score.score_generic_module.score_generic_module import ScoreModelModule
7 from azureml.designer.serving.dagengine.converter import create_dfd_from_dict
8 from collections import defaultdict
9 from azureml.designer.serving.dagengine.utils import decode_nan
10 from azureml.studio.common.datatable.DataTable import DataTable
11
12
13 model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'trained_model_outputs')
14 schema_file_path = Path(model_path) / '_schema.json'
15 with open(schema_file_path) as fp:
16     schema_data = json.load(fp)
17
18
19 def init():
20     global model
21     model = ModelDirectory.load(model_path).model
22
23
24 def run(data):
25     data = json.loads(data)
26     input_entry = defaultdict(list)
27     for row in data:
```

NOTE

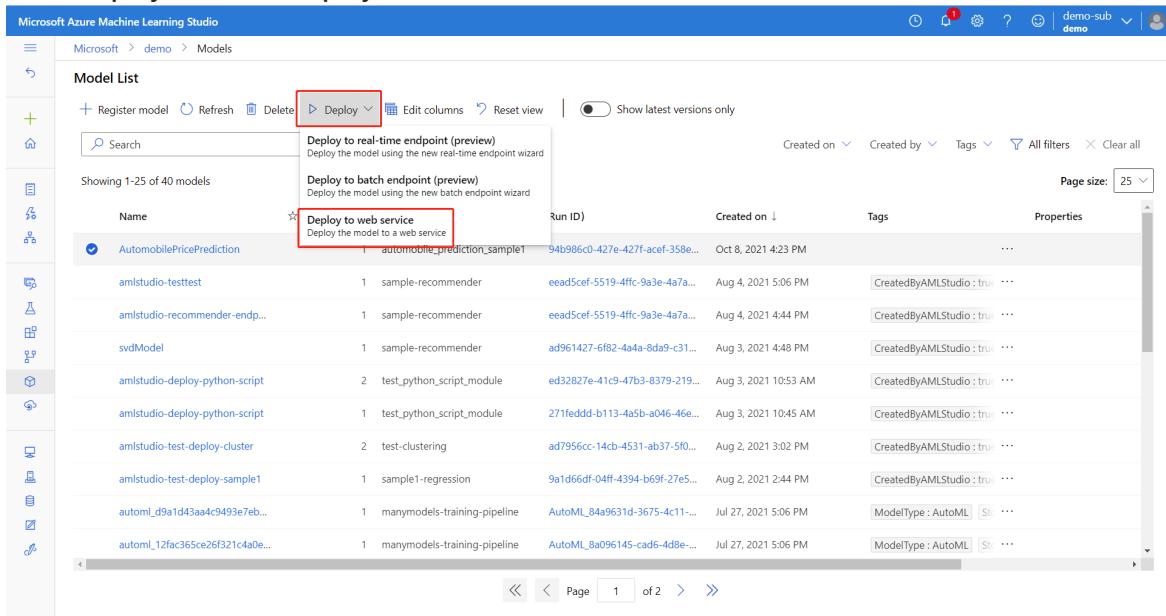
The `score.py` file provides nearly the same functionality as the **Score Model** components. However, some components like **Score SVD Recommender**, **Score Wide and Deep Recommender**, and **Score Vowpal Wabbit Model** have parameters for different scoring modes. You can also change those parameters in the entry script.

For more information on setting parameters in the `score.py` file, see the section, [Configure the entry script](#).

Deploy the model

After downloading the necessary files, you're ready to deploy the model.

1. In the **Models** asset page, select the registered model.
2. Select **Deploy** and select **Deploy to web service**.



The screenshot shows the Microsoft Azure Machine Learning Studio interface. On the left, there's a sidebar with various icons. The main area is titled "Model List". At the top, there's a "Deploy" dropdown menu with several options: "Deploy to real-time endpoint (preview)", "Deploy to batch endpoint (preview)", and "Deploy to web service". The "Deploy to web service" option is highlighted with a red box. Below the dropdown, there's a table listing 1-25 of 40 models. The columns include Name, Run ID, Created on, Tags, and Properties. One row is selected, showing "AutomobilePricePrediction" with a run ID of "94b586c0-427e-427f-acef-358e...". At the bottom of the table, there are navigation buttons for pages 1 and 2.

3. In the configuration menu, enter the following information:

- Input a name for the endpoint.
- Select to deploy the model to [Azure Kubernetes Service](#) or [Azure Container Instance](#).
- Upload the `score.py` for the **Entry script file**.
- Upload the `conda_env.yml` for the **Conda dependencies file**.

TIP

In **Advanced** setting, you can set CPU/Memory capacity and other parameters for deployment. These settings are important for certain models such as PyTorch models, which consume considerable amount of memory (about 4 GB).

4. Select **Deploy** to deploy your model as an online endpoint.

Deploy a model

Name * ⓘ



*

Description ⓘ

Compute type * ⓘ



*

Models: PricePredictionModel:1

Enable authentication



Type



Entry script file * ⓘ

*

Conda dependencies file * ⓘ

*

Dependencies

Consume the online endpoint

After deployment succeeds, you can find the endpoint in the **Endpoints** asset page. Once there, you will find a REST endpoint, which clients can use to submit requests to the endpoint.

NOTE

The designer also generates a sample data json file for testing, you can download `_samples.json` in the `trained_model_outputs` folder.

Use the following code sample to consume an online endpoint.

```
import json
from pathlib import Path
from azureml.core.workspace import Workspace, Webservice

service_name = 'YOUR_SERVICE_NAME'
ws = Workspace.get(
    name='WORKSPACE_NAME',
    subscription_id='SUBSCRIPTION_ID',
    resource_group='RESOURCEGROUP_NAME'
)
service = Webservice(ws, service_name)
sample_file_path = '_samples.json'

with open(sample_file_path, 'r') as f:
    sample_data = json.load(f)
score_result = service.run(json.dumps(sample_data))
print(f'Inference result = {score_result}')
```

Consume computer vision related online endpoints

When consuming computer vision related online endpoints, you need to convert images to bytes, since web service only accepts string as input. Following is the sample code:

```

import base64
import json
from copy import deepcopy
from pathlib import Path
from azureml.studio.core.io.image_directory import (IMG_EXTS, image_from_file, image_to_bytes)
from azureml.studio.core.io.transformation_directory import ImageTransformationDirectory

# image path
image_path = Path('YOUR_IMAGE_FILE_PATH')

# provide the same parameter setting as in the training pipeline. Just an example here.
image_transform = [
    # format: (op, args). {} means using default parameter values of torchvision.transforms.
    # See https://pytorch.org/docs/stable/torchvision/transforms.html
    ('Resize', 256),
    ('CenterCrop', 224),
    ('Pad', 0),
    ('ColorJitter', {}),
    ('Grayscale', {}),
    ('RandomResizedCrop', 256),
    ('RandomCrop', 224),
    ('RandomHorizontalFlip', {}),
    ('RandomVerticalFlip', {}),
    ('RandomRotation', 0),
    ('RandomAffine', 0),
    ('RandomGrayscale', {}),
    ('RandomPerspective', {}),
]
transform = ImageTransformationDirectory.create(transforms=image_transform).torch_transform

# download _samples.json file under Outputs+logs tab in the right pane of Train Pytorch Model component
sample_file_path = '_samples.json'
with open(sample_file_path, 'r') as f:
    sample_data = json.load(f)

# use first sample item as the default value
default_data = sample_data[0]
data_list = []
for p in image_path.iterdir():
    if p.suffix.lower() in IMG_EXTS:
        data = deepcopy(default_data)
        # convert image to bytes
        data['image'] = base64.b64encode(image_to_bytes(transform(image_from_file(p))).decode())
        data_list.append(data)

# use data.json as input of consuming the endpoint
data_file_path = 'data.json'
with open(data_file_path, 'w') as f:
    json.dump(data_list, f)

```

Configure the entry script

Some components in the designer like [Score SVD Recommender](#), [Score Wide and Deep Recommender](#), and [Score Vowpal Wabbit Model](#) have parameters for different scoring modes.

In this section, you learn how to update these parameters in the entry script file too.

The following example updates the default behavior for a trained **Wide & Deep recommender** model. By default, the `score.py` file tells the web service to predict ratings between users and items.

You can modify the entry script file to make item recommendations, and return recommended items by changing the `recommender_prediction_kind` parameter.

```

import os
import json
from pathlib import Path
from collections import defaultdict
from azureml.studio.core.io.model_directory import ModelDirectory
from azureml.designer.modules.recommendation.dnn.wide_and_deep.score. \
    score_wide_and_deep_recommender import ScoreWideAndDeepRecommenderModule
from azureml.designer.serving.dagengine.utils import decode_nan
from azureml.designer.serving.dagengine.converter import create_dfd_from_dict

model_path = os.path.join(os.getenv('AZUREML_MODEL_DIR'), 'trained_model_outputs')
schema_file_path = Path(model_path) / '_schema.json'
with open(schema_file_path) as fp:
    schema_data = json.load(fp)

def init():
    global model
    model = ModelDirectory.load(load_from_dir=model_path)

def run(data):
    data = json.loads(data)
    input_entry = defaultdict(list)
    for row in data:
        for key, val in row.items():
            input_entry[key].append(decode_nan(val))

    data_frame_directory = create_dfd_from_dict(input_entry, schema_data)

    # The parameter names can be inferred from Score Wide and Deep Recommender component parameters:
    # convert the letters to lower cases and replace whitespaces to underscores.
    score_params = dict(
        trained_wide_and_deep_recommendation_model=model,
        dataset_to_score=data_frame_directory,
        training_data=None,
        user_features=None,
        item_features=None,
        ##### Note #####
        # Set 'Recommender prediction kind' parameter to enable item recommendation model
        recommender_prediction_kind='Item Recommendation',
        recommended_item_selection='From All Items',
        maximum_number_of_items_to_recommend_to_a_user=5,
        whether_to_return_the_predicted_ratings_of_the_items_along_with_the_labels='True')
    result_dfd, = ScoreWideAndDeepRecommenderModule().run(**score_params)
    result_df = result_dfd.data
    return json.dumps(result_df.to_dict("list"))

```

For **Wide & Deep recommender** and **Vowpal Wabbit** models, you can configure the scoring mode parameter using the following methods:

- The parameter names are the lowercase and underscore combinations of parameter names for [Score Vowpal Wabbit Model](#) and [Score Wide and Deep Recommender](#);
- Mode type parameter values are strings of the corresponding option names. Take **Recommender prediction kind** in the above codes as example, the value can be `'Rating Prediction'` or `'Item Recommendation'`. Other values are not allowed.

For **SVD recommender** trained model, the parameter names and values maybe less obvious, and you can look up the tables below to decide how to set parameters.

PARAMETER NAME IN SCORE SVD RECOMMENDER	PARAMETER NAME IN THE ENTRY SCRIPT FILE
Recommender prediction kind	prediction_kind
Recommended item selection	recommended_item_selection
Minimum size of the recommendation pool for a single user	min_recommendation_pool_size
Maximum number of items to recommend to a user	max_recommended_item_count
Whether to return the predicted ratings of the items along with the labels	return_ratings

The following code shows you how to set parameters for an SVD recommender, which uses all six parameters to recommend rated items with predicted ratings attached.

```
score_params = dict(
    learner=model,
    test_data=DataTable.from_dfd(data_frame_directory),
    training_data=None,
    # RecommenderPredictionKind has 2 members, 'RatingPrediction' and 'ItemRecommendation'. You
    # can specify prediction_kind parameter with one of them.
    prediction_kind=RecommenderPredictionKind.ItemRecommendation,
    # RecommendedItemSelection has 3 members, 'FromAllItems', 'FromRatedItems', 'FromUndatedItems'.
    # You can specify recommended_item_selection parameter with one of them.
    recommended_item_selection=RecommendedItemSelection.FromRatedItems,
    min_recommendation_pool_size=1,
    max_recommended_item_count=3,
    return_ratings=True,
)
```

Next steps

- [Train a model in the designer](#)
- [Deploy models with Azure Machine Learning SDK](#)
- [Troubleshoot a failed deployment](#)
- [Deploy to Azure Kubernetes Service](#)
- [Create client applications to consume web services](#)
- [Update web service](#)

How to package a registered model with Docker

9/21/2022 • 4 minutes to read • [Edit Online](#)

This article shows how to package a registered Azure Machine Learning model with Docker.

Prerequisites

This article assumes you have already trained and registered a model in your machine learning workspace. To learn how to train and register a scikit-learn model, [follow this tutorial](#).

Package models

In some cases, you might want to create a Docker image without deploying the model. Or you might want to download the image and run it on a local Docker installation. You might even want to download the files used to build the image, inspect them, modify them, and build the image manually.

Model packaging enables you to do these things. It packages all the assets needed to host a model as a web service and allows you to download either a fully built Docker image or the files needed to build one. There are two ways to use model packaging:

Download a packaged model: Download a Docker image that contains the model and other files needed to host it as a web service.

Generate a Dockerfile: Download the Dockerfile, model, entry script, and other assets needed to build a Docker image. You can then inspect the files or make changes before you build the image locally.

Both packages can be used to get a local Docker image.

TIP

Creating a package is similar to deploying a model. You use a registered model and an inference configuration.

IMPORTANT

To download a fully built image or build an image locally, you need to have [Docker](#) installed in your development environment.

Download a packaged model

The following example builds an image, which is registered in the Azure container registry for your workspace:

```
package = Model.package(ws, [model], inference_config)
package.wait_for_creation(show_output=True)
```

After you create a package, you can use `package.pull()` to pull the image to your local Docker environment. The output of this command will display the name of the image. For example:

```
Status: Downloaded newer image for myworkspacef78fd10.azurecr.io/package:20190822181338 .
```

After you download the model, use the `docker images` command to list the local images:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myworkspacef78fd10.azurecr.io/package	20190822181338	7ff48015d5bd	4 minutes ago	1.43 GB

To start a local container based on this image, use the following command to start a named container from the shell or command line. Replace the `<imageid>` value with the image ID returned by the `docker images` command.

```
docker run -p 6789:5001 --name mycontainer <imageid>
```

This command starts the latest version of the image named `myimage`. It maps local port 6789 to the port in the container on which the web service is listening (5001). It also assigns the name `mycontainer` to the container, which makes the container easier to stop. After the container is started, you can submit requests to `http://localhost:6789/score`.

Generate a Dockerfile and dependencies

The following example shows how to download the Dockerfile, model, and other assets needed to build an image locally. The `generate_dockerfile=True` parameter indicates that you want the files, not a fully built image.

```
package = Model.package(ws, [model], inference_config, generate_dockerfile=True)
package.wait_for_creation(show_output=True)
# Download the package.
package.save("./imagefiles")
# Get the Azure container registry that the model/Dockerfile uses.
acr=package.get_container_registry()
print("Address:", acr.address)
print("Username:", acr.username)
print("Password:", acr.password)
```

This code downloads the files needed to build the image to the `imagefiles` directory. The Dockerfile included in the saved files references a base image stored in an Azure container registry. When you build the image on your local Docker installation, you need to use the address, user name, and password to authenticate to the registry. Use the following steps to build the image by using a local Docker installation:

- From a shell or command-line session, use the following command to authenticate Docker with the Azure container registry. Replace `<address>`, `<username>`, and `<password>` with the values retrieved by `package.get_container_registry()`.

```
docker login <address> -u <username> -p <password>
```

- To build the image, use the following command. Replace `<imagefiles>` with the path of the directory where `package.save()` saved the files.

```
docker build --tag myimage <imagefiles>
```

This command sets the image name to `myimage`.

To verify that the image is built, use the `docker images` command. You should see the `myimage` image in the list:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	2d5ee0bf3b3b	49 seconds ago	1.43 GB
myimage	latest	739f22498d64	3 minutes ago	1.43 GB

To start a new container based on this image, use the following command:

```
docker run -p 6789:5001 --name mycontainer myimage:latest
```

This command starts the latest version of the image named `myimage`. It maps local port 6789 to the port in the container on which the web service is listening (5001). It also assigns the name `mycontainer` to the container, which makes the container easier to stop. After the container is started, you can submit requests to `http://localhost:6789/score`.

Example client to test the local container

The following code is an example of a Python client that can be used with the container:

```
import requests
import json

# URL for the web service.
scoring_uri = 'http://localhost:6789/score'

# Two sets of data to score, so we get two results back.
data = {"data":
    [
        [ 1,2,3,4,5,6,7,8,9,10 ],
        [ 10,9,8,7,6,5,4,3,2,1 ]
    ]
}

# Convert to JSON string.
input_data = json.dumps(data)

# Set the content type.
headers = {'Content-Type': 'application/json'}

# Make the request and display the response.
resp = requests.post(scoring_uri, input_data, headers=headers)
print(resp.text)
```

For example clients in other programming languages, see [Consume models deployed as web services](#).

Stop the Docker container

To stop the container, use the following command from a different shell or command line:

```
docker kill mycontainer
```

Next steps

- [Troubleshoot a failed deployment](#)
- [Deploy to Azure Kubernetes Service](#)
- [Create client applications to consume web services](#)
- [Update web service](#)
- [How to deploy a model using a custom Docker image](#)
- [Use TLS to secure a web service through Azure Machine Learning](#)
- [Monitor your Azure Machine Learning models with Application Insights](#)
- [Collect data for models in production](#)
- [Create event alerts and triggers for model deployments](#)

Create and run machine learning pipelines with Azure Machine Learning SDK

9/21/2022 • 14 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to create and run [machine learning pipelines](#) by using the [Azure Machine Learning SDK](#). Use [ML pipelines](#) to create a workflow that stitches together various ML phases. Then, publish that pipeline for later access or sharing with others. Track ML pipelines to see how your model is performing in the real world and to detect data drift. ML pipelines are ideal for batch scoring scenarios, using various computes, reusing steps instead of rerunning them, and sharing ML workflows with others.

This article isn't a tutorial. For guidance on creating your first pipeline, see [Tutorial: Build an Azure Machine Learning pipeline for batch scoring](#) or [Use automated ML in an Azure Machine Learning pipeline in Python](#).

While you can use a different kind of pipeline called an [Azure Pipeline](#) for CI/CD automation of ML tasks, that type of pipeline isn't stored in your workspace. [Compare these different pipelines](#).

The ML pipelines you create are visible to the members of your Azure Machine Learning [workspace](#).

ML pipelines execute on compute targets (see [What are compute targets in Azure Machine Learning](#)). Pipelines can read and write data to and from supported [Azure Storage](#) locations.

If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#).

Prerequisites

- An Azure Machine Learning workspace. [Create workspace resources](#).
- [Configure your development environment](#) to install the Azure Machine Learning SDK, or use an [Azure Machine Learning compute instance](#) with the SDK already installed.

Start by attaching your workspace:

```
import azureml.core
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()
```

Set up machine learning resources

Create the resources required to run an ML pipeline:

- Set up a datastore used to access the data needed in the pipeline steps.
- Configure a `Dataset` object to point to persistent data that lives in, or is accessible in, a datastore. Configure an `OutputFileDatasetConfig` object for temporary data passed between pipeline steps.
- Set up the [compute targets](#) on which your pipeline steps will run.

Set up a datastore

A datastore stores the data for the pipeline to access. Each workspace has a default datastore. You can register

more datastores.

When you create your workspace, [Azure Files](#) and [Azure Blob storage](#) are attached to the workspace. A default datastore is registered to connect to the Azure Blob storage. To learn more, see [Deciding when to use Azure Files, Azure Blobs, or Azure Disks](#).

```
# Default datastore
def_data_store = ws.get_default_datastore()

# Get the blob storage associated with the workspace
def_blob_store = Datastore(ws, "workspaceblobstore")

# Get file storage associated with the workspace
def_file_store = Datastore(ws, "workspacefilestore")
```

Steps generally consume data and produce output data. A step can create data such as a model, a directory with model and dependent files, or temporary data. This data is then available for other steps later in the pipeline. To learn more about connecting your pipeline to your data, see the articles [How to Access Data](#) and [How to Register Datasets](#).

Configure data with `Dataset` and `OutputFileDatasetConfig` objects

The preferred way to provide data to a pipeline is a `Dataset` object. The `Dataset` object points to data that lives in or is accessible from a datastore or at a Web URL. The `Dataset` class is abstract, so you'll create an instance of either a `FileDataset` (referring to one or more files) or a `TabularDataset` that's created by from one or more files with delimited columns of data.

You create a `Dataset` using methods like `from_files` or `from_delimited_files`.

```
from azureml.core import Dataset

my_dataset = Dataset.File.from_files([(def_blob_store, 'train-images/')])
```

Intermediate data (or output of a step) is represented by an `OutputFileDatasetConfig` object. `output_data1` is produced as the output of a step. Optionally, this data can be registered as a dataset by calling `register_on_complete`. If you create an `OutputFileDatasetConfig` in one step and use it as an input to another step, that data dependency between steps creates an implicit execution order in the pipeline.

`OutputFileDatasetConfig` objects return a directory, and by default writes output to the default datastore of the workspace.

```
from azureml.data import OutputFileDatasetConfig

output_data1 = OutputFileDatasetConfig(destination = (datastore, 'outputdataset/{run-id}'))
output_data_dataset = output_data1.register_on_complete(name = 'prepared_output_data')
```

IMPORTANT

Intermediate data stored using `OutputFileDatasetConfig` isn't automatically deleted by Azure. You should either programmatically delete intermediate data at the end of a pipeline run, use a datastore with a short data-retention policy, or regularly do manual clean up.

TIP

Only upload files relevant to the job at hand. Any change in files within the data directory will be seen as reason to rerun the step the next time the pipeline is run even if reuse is specified.

Set up a compute target

In Azure Machine Learning, the term **compute** (or **compute target**) refers to the machines or clusters that do the computational steps in your machine learning pipeline. See [compute targets for model training](#) for a full list of compute targets and [Create compute targets](#) for how to create and attach them to your workspace. The process for creating and or attaching a compute target is the same whether you're training a model or running a pipeline step. After you create and attach your compute target, use the `ComputeTarget` object in your [pipeline step](#).

IMPORTANT

Performing management operations on compute targets isn't supported from inside remote jobs. Since machine learning pipelines are submitted as a remote job, do not use management operations on compute targets from inside the pipeline.

Azure Machine Learning compute

You can create an Azure Machine Learning compute for running your steps. The code for other compute targets is similar, with slightly different parameters, depending on the type.

```
from azureml.core.compute import ComputeTarget, AmlCompute

compute_name = "aml-compute"
vm_size = "STANDARD_NC6"
if compute_name in ws.compute_targets:
    compute_target = ws.compute_targets[compute_name]
    if compute_target and type(compute_target) is AmlCompute:
        print('Found compute target: ' + compute_name)
else:
    print('Creating a new compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size=vm_size, # STANDARD_NC6 is GPU-enabled
                                                               min_nodes=0,
                                                               max_nodes=4)
# create the compute target
compute_target = ComputeTarget.create(
    ws, compute_name, provisioning_config)

# Can poll for a minimum number of nodes and for a specific timeout.
# If no min node count is provided it will use the scale settings for the cluster
compute_target.wait_for_completion(
    show_output=True, min_node_count=None, timeout_in_minutes=20)

# For a more detailed view of current cluster status, use the 'status' property
print(compute_target.status.serialize())
```

Configure the training run's environment

The next step is making sure that the remote training run has all the dependencies needed by the training steps. Dependencies and the runtime context are set by creating and configuring a `RunConfiguration` object.

```

from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core import Environment

aml_run_config = RunConfiguration()
# `compute_target` as defined in "Azure Machine Learning compute" section above
aml_run_config.target = compute_target

USE_CURATED_ENV = True
if USE_CURATED_ENV :
    curated_environment = Environment.get(workspace=ws, name="AzureML-sklearn-0.24-ubuntu18.04-py37-cpu")
    aml_run_config.environment = curated_environment
else:
    aml_run_config.environment.python.user_managed_dependencies = False

# Add some packages relied on by data prep step
aml_run_config.environment.python.conda_dependencies = CondaDependencies.create(
    conda_packages=['pandas','scikit-learn'],
    pip_packages=['azureml-sdk', 'azureml-dataset-runtime[fuse,pandas]'],
    pin_sdk_version=False)

```

The code above shows two options for handling dependencies. As presented, with `USE_CURATED_ENV = True`, the configuration is based on a curated environment. Curated environments are "prebaked" with common inter-dependent libraries and can be faster to bring online. Curated environments have prebuilt Docker images in the [Microsoft Container Registry](#). For more information, see [Azure Machine Learning curated environments](#).

The path taken if you change `USE_CURATED_ENV` to `False` shows the pattern for explicitly setting your dependencies. In that scenario, a new custom Docker image will be created and registered in an Azure Container Registry within your resource group (see [Introduction to private Docker container registries in Azure](#)). Building and registering this image can take quite a few minutes.

Construct your pipeline steps

Once you have the compute resource and environment created, you're ready to define your pipeline's steps. There are many built-in steps available via the Azure Machine Learning SDK, as you can see on the [reference documentation for the `azureml.pipeline.steps` package](#). The most flexible class is `PythonScriptStep`, which runs a Python script.

```

from azureml.pipeline.steps import PythonScriptStep
dataprep_source_dir = "./dataprep_src"
entry_point = "prepare.py"
# `my_dataset` as defined above
ds_input = my_dataset.as_named_input('input1')

# `output_data1`, `compute_target`, `aml_run_config` as defined above
data_prep_step = PythonScriptStep(
    script_name=entry_point,
    source_directory=dataprep_source_dir,
    arguments=["--input", ds_input.as_download(), "--output", output_data1],
    compute_target=compute_target,
    runconfig=aml_run_config,
    allow_reuse=True
)

```

The above code shows a typical initial pipeline step. Your data preparation code is in a subdirectory (in this example, `"prepare.py"` in the directory `"./dataprep_src"`). As part of the pipeline creation process, this directory is zipped and uploaded to the `compute_target` and the step runs the script specified as the value for `script_name`.

The `arguments` values specify the inputs and outputs of the step. In the example above, the baseline data is the `my_dataset` dataset. The corresponding data will be downloaded to the compute resource since the code specifies it as `as_download()`. The script `prepare.py` does whatever data-transformation tasks are appropriate to the task at hand and outputs the data to `output_data1`, of type `OutputFileDatasetConfig`. For more information, see [Moving data into and between ML pipeline steps \(Python\)](#). The step will run on the machine defined by `compute_target`, using the configuration `aml_run_config`.

Reuse of previous results (`allow_reuse`) is key when using pipelines in a collaborative environment since eliminating unnecessary reruns offers agility. Reuse is the default behavior when the `script_name`, `inputs`, and the parameters of a step remain the same. When reuse is allowed, results from the previous run are immediately sent to the next step. If `allow_reuse` is set to `False`, a new run will always be generated for this step during pipeline execution.

It's possible to create a pipeline with a single step, but almost always you'll choose to split your overall process into several steps. For instance, you might have steps for data preparation, training, model comparison, and deployment. For instance, one might imagine that after the `data_prep_step` specified above, the next step might be training:

```
train_source_dir = "./train_src"
train_entry_point = "train.py"

training_results = OutputFileDatasetConfig(name = "training_results",
                                           destination = def_blob_store)

train_step = PythonScriptStep(
    script_name=train_entry_point,
    source_directory=train_source_dir,
    arguments=["--prepped_data", output_data1.as_input(), "--training_results", training_results],
    compute_target=compute_target,
    runconfig=aml_run_config,
    allow_reuse=True
)
```

The above code is similar to the code in the data preparation step. The training code is in a directory separate from that of the data preparation code. The `OutputFileDatasetConfig` output of the data preparation step, `output_data1` is used as the *input* to the training step. A new `OutputFileDatasetConfig` object, `training_results` is created to hold the results for a later comparison or deployment step.

For other code examples, see [how to build a two step ML pipeline](#) and [how to write data back to datastores upon run completion](#).

After you define your steps, you build the pipeline by using some or all of those steps.

NOTE

No file or data is uploaded to Azure Machine Learning when you define the steps or build the pipeline. The files are uploaded when you call `Experiment.submit()`.

```
# list of steps to run ('compare_step' definition not shown)
compare_models = [data_prep_step, train_step, compare_step]

from azureml.pipeline.core import Pipeline

# Build the pipeline
pipeline1 = Pipeline(workspace=ws, steps=[compare_models])
```

Use a dataset

Datasets created from Azure Blob storage, Azure Files, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Database for PostgreSQL can be used as input to any pipeline step. You can write output to a [DataTransferStep](#), [DatabricksStep](#), or if you want to write data to a specific datastore use [OutputFileDatasetConfig](#).

IMPORTANT

Writing output data back to a datastore using `OutputFileDatasetConfig` is only supported for Azure Blob, Azure File share, ADLS Gen 1 and Gen 2 datastores.

```
dataset_consumming_step = PythonScriptStep(  
    script_name="iris_train.py",  
    inputs=[iris_tabular_dataset.as_named_input("iris_data")],  
    compute_target=compute_target,  
    source_directory=project_folder  
)
```

You then retrieve the dataset in your pipeline by using the [Run.input_datasets](#) dictionary.

```
# iris_train.py  
from azureml.core import Run, Dataset  
  
run_context = Run.get_context()  
iris_dataset = run_context.input_datasets['iris_data']  
dataframe = iris_dataset.to_pandas_dataframe()
```

The line `Run.get_context()` is worth highlighting. This function retrieves a `Run` representing the current experimental run. In the above sample, we use it to retrieve a registered dataset. Another common use of the `Run` object is to retrieve both the experiment itself and the workspace in which the experiment resides:

```
# Within a PythonScriptStep  
  
ws = Run.get_context().experiment.workspace
```

For more detail, including alternate ways to pass and access data, see [Moving data into and between ML pipeline steps \(Python\)](#).

Caching & reuse

To optimize and customize the behavior of your pipelines, you can do a few things around caching and reuse. For example, you can choose to:

- **Turn off the default reuse of the step run output** by setting `allow_reuse=False` during [step definition](#). Reuse is key when using pipelines in a collaborative environment since eliminating unnecessary runs offers agility. However, you can opt out of reuse.
- **Force output regeneration for all steps in a run** with
`pipeline_run = exp.submit(pipeline, regenerate_outputs=True)`

By default, `allow_reuse` for steps is enabled and the `source_directory` specified in the step definition is hashed. So, if the script for a given step remains the same (`script_name`, inputs, and the parameters), and nothing else in the `source_directory` has changed, the output of a previous step run is reused, the job isn't submitted to the compute, and the results from the previous run are immediately available to the next step instead.

```
step = PythonScriptStep(name="Hello World",
                        script_name="hello_world.py",
                        compute_target=aml_compute,
                        source_directory=source_directory,
                        allow_reuse=False,
                        hash_paths=['hello_world.ipynb'])
```

NOTE

If the names of the data inputs change, the step will rerun, *even if* the underlying data does not change. You must explicitly set the `name` field of input data (`data.as_input(name=...)`). If you do not explicitly set this value, the `name` field will be set to a random guid and the step's results will not be reused.

Submit the pipeline

When you submit the pipeline, Azure Machine Learning checks the dependencies for each step and uploads a snapshot of the source directory you specified. If no source directory is specified, the current local directory is uploaded. The snapshot is also stored as part of the experiment in your workspace.

IMPORTANT

To prevent unnecessary files from being included in the snapshot, make an ignore file (`.gitignore` or `.amignore`) in the directory. Add the files and directories to exclude to this file. For more information on the syntax to use inside this file, see [syntax and patterns for `.gitignore`](#). The `.amignore` file uses the same syntax. *If both files exist, the `.amignore` file is used and the `.gitignore` file is unused.*

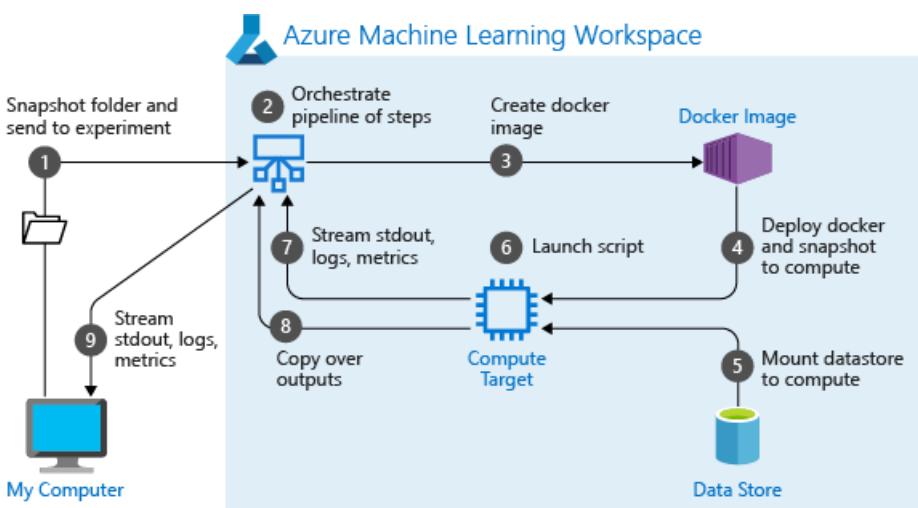
For more information, see [Snapshots](#).

```
from azureml.core import Experiment

# Submit the pipeline to be run
pipeline_run1 = Experiment(ws, 'Compare_Models_Exp').submit(pipeline1)
pipeline_run1.wait_for_completion()
```

When you first run a pipeline, Azure Machine Learning:

- Downloads the project snapshot to the compute target from the Blob storage associated with the workspace.
- Builds a Docker image corresponding to each step in the pipeline.
- Downloads the Docker image for each step to the compute target from the container registry.
- Configures access to `Dataset` and `OutputFileDatasetConfig` objects. For `as_mount()` access mode, FUSE is used to provide virtual access. If mount isn't supported or if the user specified access as `as_upload()`, the data is instead copied to the compute target.
- Runs the step in the compute target specified in the step definition.
- Creates artifacts, such as logs, stdout and stderr, metrics, and output specified by the step. These artifacts are then uploaded and kept in the user's default datastore.



For more information, see the [Experiment class reference](#).

Use pipeline parameters for arguments that change at inference time

Sometimes, the arguments to individual steps within a pipeline relate to the development and training period: things like training rates and momentum, or paths to data or configuration files. When a model is deployed, though, you'll want to dynamically pass the arguments upon which you're inferencing (that is, the query you built the model to answer!). You should make these types of arguments pipeline parameters. To do this in Python, use the `azureml.pipeline.core.PipelineParameter` class, as shown in the following code snippet:

```
from azureml.pipeline.core import PipelineParameter

pipeline_param = PipelineParameter(name="pipeline_arg", default_value="default_val")
train_step = PythonScriptStep(script_name="train.py",
                             arguments=["--param1", pipeline_param],
                             target=compute_target,
                             source_directory=project_folder)
```

How Python environments work with pipeline parameters

As discussed previously in [Configure the training run's environment](#), environment state, and Python library dependencies are specified using an `Environment` object. Generally, you can specify an existing `Environment` by referring to its name and, optionally, a version:

```
aml_run_config = RunConfiguration()
aml_run_config.environment.name = 'MyEnvironment'
aml_run_config.environment.version = '1.0'
```

However, if you choose to use `PipelineParameter` objects to dynamically set variables at runtime for your pipeline steps, you can't use this technique of referring to an existing `Environment`. Instead, if you want to use `PipelineParameter` objects, you must set the `environment` field of the `RunConfiguration` to an `Environment` object. It is your responsibility to ensure that such an `Environment` has its dependencies on external Python packages properly set.

View results of a pipeline

See the list of all your pipelines and their run details in the studio:

1. Sign in to [Azure Machine Learning studio](#).
2. [View your workspace](#).

3. On the left, select **Pipelines** to see all your pipeline runs.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. The left sidebar has a 'Pipelines' item highlighted with a red box. The main content area is titled 'Pipelines' and shows a list of 'Pipeline jobs'. There are four entries, each with a green checkmark indicating 'Completed' status. The columns are 'Display name', 'Experiment', 'Status', and 'Description'. The 'Display name' column lists 'credit_defaults_pipeline' four times. The 'Experiment' column lists 'e2e_registered_components' four times. The 'Status' column shows green checkmarks for all. The 'Description' column shows 'E2E data_perp' for all. A search bar and filter options are at the top of the list.

Display name	Experiment	Status	Description
credit_defaults_pipeline	e2e_registered_components	Completed	E2E data_perp
credit_defaults_pipeline	e2e_registered_components	Completed	E2E data_perp
credit_defaults_pipeline	e2e_registered_components	Completed	E2E data_perp
credit_defaults_pipeline	e2e_registered_components	Completed	E2E data_perp

4. Select a specific pipeline to see the run results.

Git tracking and integration

When you start a training run where the source directory is a local Git repository, information about the repository is stored in the run history. For more information, see [Git integration for Azure Machine Learning](#).

Next steps

- To share your pipeline with colleagues or customers, see [Publish machine learning pipelines](#)
- Use [these Jupyter notebooks on GitHub](#) to explore machine learning pipelines further
- See the SDK reference help for the `azureml-pipelines-core` package and the `azureml-pipelines-steps` package
- See the [how-to](#) for tips on debugging and troubleshooting pipelines=
- Learn how to run notebooks by following the article [Use Jupyter notebooks to explore this service](#).

Moving data into and between ML pipeline steps (Python)

9/21/2022 • 7 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article provides code for importing, transforming, and moving data between steps in an Azure Machine Learning pipeline. For an overview of how data works in Azure Machine Learning, see [Access data in Azure storage services](#). For the benefits and structure of Azure Machine Learning pipelines, see [What are Azure Machine Learning pipelines?](#)

This article will show you how to:

- Use `Dataset` objects for pre-existing data
- Access data within your steps
- Split `Dataset` data into subsets, such as training and validation subsets
- Create `OutputFileDatasetConfig` objects to transfer data to the next pipeline step
- Use `OutputFileDatasetConfig` objects as input to pipeline steps
- Create new `Dataset` objects from `OutputFileDatasetConfig` you wish to persist

Prerequisites

You'll need:

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#).
- The [Azure Machine Learning SDK for Python](#), or access to [Azure Machine Learning studio](#).
- An Azure Machine Learning workspace.

Either [create an Azure Machine Learning workspace](#) or use an existing one via the Python SDK. Import the `Workspace` and `Datastore` class, and load your subscription information from the file `config.json` using the function `from_config()`. This function looks for the JSON file in the current directory by default, but you can also specify a path parameter to point to the file using `from_config(path="your/file/path")`.

```
import azureml.core
from azureml.core import Workspace, Datastore

ws = Workspace.from_config()
```

- Some pre-existing data. This article briefly shows the use of an [Azure blob container](#).
- Optional: An existing machine learning pipeline, such as the one described in [Create and run machine learning pipelines with Azure Machine Learning SDK](#).

Use `Dataset` objects for pre-existing data

The preferred way to ingest data into a pipeline is to use a `Dataset` object. `Dataset` objects represent persistent data available throughout a workspace.

There are many ways to create and register `Dataset` objects. Tabular datasets are for delimited data available in one or more files. File datasets are for binary data (such as images) or for data that you'll parse. The simplest programmatic ways to create `Dataset` objects are to use existing blobs in workspace storage or public URLs:

```
datastore = Datastore.get(workspace, 'training_data')
iris_dataset = Dataset.Tabular.from_delimited_files(DataPath(datastore, 'iris.csv'))

datastore_path = [
    DataPath(datastore, 'animals/dog/1.jpg'),
    DataPath(datastore, 'animals/dog/2.jpg'),
    DataPath(datastore, 'animals/cat/*.jpg')
]
cats_dogs_dataset = Dataset.File.from_files(path=datastore_path)
```

For more options on creating datasets with different options and from different sources, registering them and reviewing them in the Azure Machine Learning UI, understanding how data size interacts with compute capacity, and versioning them, see [Create Azure Machine Learning datasets](#).

Pass datasets to your script

To pass the dataset's path to your script, use the `Dataset` object's `as_named_input()` method. You can either pass the resulting `DatasetConsumptionConfig` object to your script as an argument or, by using the `inputs` argument to your pipeline script, you can retrieve the dataset using `Run.get_context().input_datasets[]`.

Once you've created a named input, you can choose its access mode: `as_mount()` or `as_download()`. If your script processes all the files in your dataset and the disk on your compute resource is large enough for the dataset, the download access mode is the better choice. The download access mode will avoid the overhead of streaming the data at runtime. If your script accesses a subset of the dataset or it's too large for your compute, use the mount access mode. For more information, read [Mount vs. Download](#)

To pass a dataset to your pipeline step:

1. Use `TabularDataset.as_named_input()` or `FileDataset.as_named_input()` (no 's' at end) to create a `DatasetConsumptionConfig` object
2. Use `as_mount()` or `as_download()` to set the access mode
3. Pass the datasets to your pipeline steps using either the `arguments` or the `inputs` argument

The following snippet shows the common pattern of combining these steps within the `PythonScriptStep` constructor:

```
train_step = PythonScriptStep(
    name="train_data",
    script_name="train.py",
    compute_target=cluster,
    inputs=[iris_dataset.as_named_input('iris').as_mount()]
)
```

NOTE

You would need to replace the values for all these arguments (that is, `"train_data"`, `"train.py"`, `cluster`, and `iris_dataset`) with your own data. The above snippet just shows the form of the call and is not part of a Microsoft sample.

You can also use methods such as `random_split()` and `take_sample()` to create multiple inputs or reduce the amount of data passed to your pipeline step:

```

seed = 42 # PRNG seed
smaller_dataset = iris_dataset.take_sample(0.1, seed=seed) # 10%
train, test = smaller_dataset.random_split(percentage=0.8, seed=seed)

train_step = PythonScriptStep(
    name="train_data",
    script_name="train.py",
    compute_target=cluster,
    inputs=[train.as_named_input('train').as_download(), test.as_named_input('test').as_download()]
)

```

Access datasets within your script

Named inputs to your pipeline step script are available as a dictionary within the `Run` object. Retrieve the active `Run` object using `Run.get_context()` and then retrieve the dictionary of named inputs using `input_datasets`. If you passed the `DatasetConsumptionConfig` object using the `arguments` argument rather than the `inputs` argument, access the data using `ArgParser` code. Both techniques are demonstrated in the following snippet.

```

# In pipeline definition script:
# Code for demonstration only: It would be very confusing to split datasets between `arguments` and `inputs`
train_step = PythonScriptStep(
    name="train_data",
    script_name="train.py",
    compute_target=cluster,
    arguments=['--training-folder', train.as_named_input('train').as_download()],
    inputs=[test.as_named_input('test').as_download()]
)

# In pipeline script
parser = argparse.ArgumentParser()
parser.add_argument('--training-folder', type=str, dest='train_folder', help='training data folder mounting point')
args = parser.parse_args()
training_data_folder = args.train_folder

testing_data_folder = Run.get_context().input_datasets['test']

```

The passed value will be the path to the dataset file(s).

It's also possible to access a registered `Dataset` directly. Since registered datasets are persistent and shared across a workspace, you can retrieve them directly:

```

run = Run.get_context()
ws = run.experiment.workspace
ds = Dataset.get_by_name(workspace=ws, name='mnist_opendataset')

```

NOTE

The preceding snippets show the form of the calls and are not part of a Microsoft sample. You must replace the various arguments with values from your own project.

Use `OutputFileDatasetConfig` for intermediate data

While `Dataset` objects represent only persistent data, `OutputFileDatasetConfig` object(s) can be used for temporary data output from pipeline steps and persistent output data. `OutputFileDatasetConfig` supports writing data to blob storage, fileshare, adlsgen1, or adlsgen2. It supports both mount mode and upload mode. In mount mode, files written to the mounted directory are permanently stored when the file is closed. In upload

mode, files written to the output directory are uploaded at the end of the job. If the job fails or is canceled, the output directory will not be uploaded.

`OutputFileDatasetConfig` object's default behavior is to write to the default datastore of the workspace. Pass your `OutputFileDatasetConfig` objects to your `PythonScriptStep` with the `arguments` parameter.

```
from azureml.data import OutputFileDatasetConfig
dataprep_output = OutputFileDatasetConfig()
input_dataset = Dataset.get_by_name(workspace, 'raw_data')

dataprep_step = PythonScriptStep(
    name="prep_data",
    script_name="dataprep.py",
    compute_target=cluster,
    arguments=[input_dataset.as_named_input('raw_data').as_mount(), dataprep_output]
)
```

NOTE

Concurrent writes to a `OutputFileDatasetConfig` will fail. Do not attempt to use a single `OutputFileDatasetConfig` concurrently. Do not share a single `OutputFileDatasetConfig` in a multiprocessing situation, such as when using [distributed training](#).

Use `OutputFileDatasetConfig` as outputs of a training step

Within your pipeline's `PythonScriptStep`, you can retrieve the available output paths using the program's arguments. If this step is the first and will initialize the output data, you must create the directory at the specified path. You can then write whatever files you wish to be contained in the `OutputFileDatasetConfig`.

```
parser = argparse.ArgumentParser()
parser.add_argument('--output_path', dest='output_path', required=True)
args = parser.parse_args()

# Make directory for file
os.makedirs(os.path.dirname(args.output_path), exist_ok=True)
with open(args.output_path, 'w') as f:
    f.write("Step 1's output")
```

Read `OutputFileDatasetConfig` as inputs to non-initial steps

After the initial pipeline step writes some data to the `OutputFileDatasetConfig` path and it becomes an output of that initial step, it can be used as an input to a later step.

In the following code:

- `step1_output_data` indicates that the output of the `PythonScriptStep`, `step1` is written to the ADLS Gen 2 datastore, `my_adlsgen2` in upload access mode. Learn more about how to [set up role permissions](#) in order to write data back to ADLS Gen 2 datastores.
- After `step1` completes and the output is written to the destination indicated by `step1_output_data`, then `step2` is ready to use `step1_output_data` as an input.

```
# get adls gen 2 datastore already registered with the workspace
datastore = workspace.datastores['my_adlsgen2']
step1_output_data = OutputFileDatasetConfig(name="processed_data", destination=(datastore, "mypath/{run-id}/{output-name}")).as_upload()

step1 = PythonScriptStep(
    name="generate_data",
    script_name="step1.py",
    runconfig = aml_run_config,
    arguments = ["--output_path", step1_output_data]
)

step2 = PythonScriptStep(
    name="read_pipeline_data",
    script_name="step2.py",
    compute_target=compute,
    runconfig = aml_run_config,
    arguments = ["--pd", step1_output_data.as_input()]

)

pipeline = Pipeline(workspace=ws, steps=[step1, step2])
```

Register `OutputFileDatasetConfig` objects for reuse

If you'd like to make your `OutputFileDatasetConfig` available for longer than the duration of your experiment, register it to your workspace to share and reuse across experiments.

```
step1_output_ds = step1_output_data.register_on_complete(name='processed_data',
                                                       description = 'files from step1')
```

Delete `OutputFileDatasetConfig` contents when no longer needed

Azure does not automatically delete intermediate data written with `OutputFileDatasetConfig`. To avoid storage charges for large amounts of unneeded data, you should either:

- Programmatically delete intermediate data at the end of a pipeline job, when it is no longer needed
- Use blob storage with a short-term storage policy for intermediate data (see [Optimize costs by automating Azure Blob Storage access tiers](#))
- Regularly review and delete no-longer-needed data

For more information, see [Plan and manage costs for Azure Machine Learning](#).

Next steps

- [Create an Azure machine learning dataset](#)
- [Create and run machine learning pipelines with Azure Machine Learning SDK](#)

Use automated ML in an Azure Machine Learning pipeline in Python

9/21/2022 • 15 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

Azure Machine Learning's automated ML capability helps you discover high-performing models without you reimplementing every possible approach. Combined with Azure Machine Learning pipelines, you can create deployable workflows that can quickly discover the algorithm that works best for your data. This article will show you how to efficiently join a data preparation step to an automated ML step. Automated ML can quickly discover the algorithm that works best for your data, while putting you on the road to MLOps and model lifecycle operationalization with pipelines.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a free account before you begin. Try the [free or paid version of Azure Machine Learning](#) today.
- An Azure Machine Learning workspace. See [Create workspace resources](#).
- Familiarity with Azure's [automated machine learning](#) and [machine learning pipelines](#) facilities and SDK.

Review automated ML's central classes

Automated ML in a pipeline is represented by an `AutoMLStep` object. The `AutoMLStep` class is a subclass of `PipelineStep`. A graph of `PipelineStep` objects defines a `Pipeline`.

There are several subclasses of `PipelineStep`. In addition to the `AutoMLStep`, this article will show a `PythonScriptStep` for data preparation and another for registering the model.

The preferred way to initially move data *into* an ML pipeline is with `Dataset` objects. To move data *between* steps and possibly save data output from runs, the preferred way is with `OutputFileDatasetConfig` and `OutputTabularDatasetConfig` objects. To be used with `AutoMLStep`, the `PipelineData` object must be transformed into a `PipelineOutputTabularDataset` object. For more information, see [Input and output data from ML pipelines](#).

The `AutoMLStep` is configured via an `AutoMLConfig` object. `AutoMLConfig` is a flexible class, as discussed in [Configure automated ML experiments in Python](#).

A `Pipeline` runs in an `Experiment`. The pipeline `Run` has, for each step, a child `StepRun`. The outputs of the automated ML `StepRun` are the training metrics and highest-performing model.

To make things concrete, this article creates a simple pipeline for a classification task. The task is predicting Titanic survival, but we won't be discussing the data or task except in passing.

Get started

Retrieve initial dataset

Often, an ML workflow starts with pre-existing baseline data. This is a good scenario for a registered dataset. Datasets are visible across the workspace, support versioning, and can be interactively explored. There are many ways to create and populate a dataset, as discussed in [Create Azure Machine Learning datasets](#). Since we'll be using the Python SDK to create our pipeline, use the SDK to download baseline data and register it with the

name 'titanic_ds'.

```
from azureml.core import Workspace, Dataset

ws = Workspace.from_config()
if not 'titanic_ds' in ws.datasets.keys() :
    # create a TabularDataset from Titanic training data
    web_paths = ['https://dprepdata.blob.core.windows.net/demo/Titanic.csv',
                 'https://dprepdata.blob.core.windows.net/demo/Titanic2.csv']
    titanic_ds = Dataset.Tabular.from_delimited_files(path=web_paths)

    titanic_ds.register(workspace = ws,
                        name = 'titanic_ds',
                        description = 'Titanic baseline data',
                        create_new_version = True)

titanic_ds = Dataset.get_by_name(ws, 'titanic_ds')
```

The code first logs in to the Azure Machine Learning workspace defined in `config.json` (for an explanation, see [Create a workspace configuration file](#)). If there isn't already a dataset named `'titanic_ds'` registered, then it creates one. The code downloads CSV data from the Web, uses them to instantiate a `TabularDataset` and then registers the dataset with the workspace. Finally, the function `Dataset.get_by_name()` assigns the `Dataset` to `titanic_ds`.

Configure your storage and compute target

Additional resources that the pipeline will need are storage and, generally, Azure Machine Learning compute resources.

```
from azureml.core import Datastore
from azureml.core.compute import AmlCompute, ComputeTarget

datastore = ws.get_default_datastore()

compute_name = 'cpu-cluster'
if not compute_name in ws.compute_targets :
    print('creating a new compute target...')
    provisioning_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                               min_nodes=0,
                                                               max_nodes=1)
    compute_target = ComputeTarget.create(ws, compute_name, provisioning_config)

    compute_target.wait_for_completion(
        show_output=True, min_node_count=None, timeout_in_minutes=20)

    # Show the result
    print(compute_target.get_status().serialize())

compute_target = ws.compute_targets[compute_name]
```

NOTE

You may choose to use [low-priority VMs](#) to run some or all of your workloads. See how to [create a low-priority VM](#).

The intermediate data between the data preparation and the automated ML step can be stored in the workspace's default datastore, so we don't need to do more than call `get_default_datastore()` on the `Workspace` object.

After that, the code checks if the AzureML compute target `'cpu-cluster'` already exists. If not, we specify that we want a small CPU-based compute target. If you plan to use automated ML's deep learning features (for

instance, text featurization with DNN support) you should choose a compute with strong GPU support, as described in [GPU optimized virtual machine sizes](#).

The code blocks until the target is provisioned and then prints some details of the just-created compute target. Finally, the named compute target is retrieved from the workspace and assigned to `compute_target`.

Configure the training run

The runtime context is set by creating and configuring a `RunConfiguration` object. Here we set the compute target.

```
from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies

aml_run_config = RunConfiguration()
# Use just-specified compute target ("cpu-cluster")
aml_run_config.target = compute_target

# Specify CondaDependencies obj, add necessary packages
aml_run_config.environment.python.conda_dependencies = CondaDependencies.create(
    conda_packages=['pandas','scikit-learn'],
    pip_packages=['azureml-sdk[automl]', 'pyarrow'])
```

Prepare data for automated machine learning

Write the data preparation code

The baseline Titanic dataset consists of mixed numerical and text data, with some values missing. To prepare it for automated machine learning, the data preparation pipeline step will:

- Fill missing data with either random data or a category corresponding to "Unknown"
- Transform categorical data to integers
- Drop columns that we don't intend to use
- Split the data into training and testing sets
- Write the transformed data to the `OutputFileDatasetConfig` output paths

```

%%writefile dataprep.py
from azureml.core import Run

import pandas as pd
import numpy as np
import argparse

RANDOM_SEED=42

def prepare_age(df):
    # Fill in missing Age values from distribution of present Age values
    mean = df["Age"].mean()
    std = df["Age"].std()
    is_null = df["Age"].isnull().sum()
    # compute enough (== is_null().sum()) random numbers between the mean, std
    rand_age = np.random.randint(mean - std, mean + std, size = is_null)
    # fill NaN values in Age column with random values generated
    age_slice = df["Age"].copy()
    age_slice[np.isnan(age_slice)] = rand_age
    df["Age"] = age_slice
    df["Age"] = df["Age"].astype(int)

    # Quantize age into 5 classes
    df['Age_Group'] = pd.qcut(df['Age'],5, labels=False)
    df.drop(['Age'], axis=1, inplace=True)
    return df

def prepare_fare(df):
    df['Fare'].fillna(0, inplace=True)
    df['Fare_Group'] = pd.qcut(df['Fare'],5,labels=False)
    df.drop(['Fare'], axis=1, inplace=True)
    return df

def prepare_genders(df):
    genders = {"male": 0, "female": 1, "unknown": 2}
    df['Sex'] = df['Sex'].map(genders)
    df['Sex'].fillna(2, inplace=True)
    df['Sex'] = df['Sex'].astype(int)
    return df

def prepare_embarked(df):
    df['Embarked'].replace('', 'U', inplace=True)
    df['Embarked'].fillna('U', inplace=True)
    ports = {"S": 0, "C": 1, "Q": 2, "U": 3}
    df['Embarked'] = df['Embarked'].map(ports)
    return df

parser = argparse.ArgumentParser()
parser.add_argument('--output_path', dest='output_path', required=True)
args = parser.parse_args()

titanic_ds = Run.get_context().input_datasets['titanic_ds']
df = titanic_ds.to_pandas_dataframe().drop(['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1)
df = prepare_embarked(prepare_genders(prepare_fare(prepare_age(df)))) 

df.to_csv(os.path.join(args.output_path,"prepped_data.csv"))

print(f"Wrote prepped data to {args.output_path}/prepped_data.csv")

```

The above code snippet is a complete, but minimal, example of data preparation for the Titanic data. The snippet starts with a Jupyter "magic command" to output the code to a file. If you aren't using a Jupyter notebook, remove that line and create the file manually.

The various `prepare_` functions in the above snippet modify the relevant column in the input dataset. These functions work on the data once it has been changed into a Pandas `DataFrame` object. In each case, missing data

is either filled with representative random data or categorical data indicating "Unknown." Text-based categorical data is mapped to integers. No-longer-needed columns are overwritten or dropped.

After the code defines the data preparation functions, the code parses the input argument, which is the path to which we want to write our data. (These values will be determined by `OutputFileDatasetConfig` objects that will be discussed in the next step.) The code retrieves the registered `'titanic_cs'` `Dataset`, converts it to a Pandas `DataFrame`, and calls the various data preparation functions.

Since the `output_path` is a directory, the call to `to_csv()` specifies the filename `prepped_data.csv`.

Write the data preparation pipeline step (`PythonScriptStep`)

The data preparation code described above must be associated with a `PythonScriptStep` object to be used with a pipeline. The path to which the CSV output is written is generated by a `OutputFileDatasetConfig` object. The resources prepared earlier, such as the `ComputeTarget`, the `RunConfig`, and the `'titanic_ds'` `Dataset` are used to complete the specification.

```
from azureml.data import OutputFileDatasetConfig
from azureml.pipeline.steps import PythonScriptStep

prepped_data_path = OutputFileDatasetConfig(name="output_path")

dataprep_step = PythonScriptStep(
    name="dataprep",
    script_name="dataprep.py",
    compute_target=compute_target,
    runconfig=aml_run_config,
    arguments=["--output_path", prepped_data_path],
    inputs=[titanic_ds.as_named_input('titanic_ds')],
    allow_reuse=True
)
```

The `prepped_data_path` object is of type `OutputFileDatasetConfig` which points to a directory. Notice that it's specified in the `arguments` parameter. If you review the previous step, you'll see that within the data preparation code, the value of the argument `--output_path` is the directory path at which the CSV file was written.

Train with AutoMLStep

Configuring an automated ML pipeline step is done with the `AutoMLConfig` class. This flexible class is described in [Configure automated ML experiments in Python](#). Data input and output are the only aspects of configuration that require special attention in an ML pipeline. Input and output for `AutoMLConfig` in pipelines is discussed in detail below. Beyond data, an advantage of ML pipelines is the ability to use different compute targets for different steps. You might choose to use a more powerful `ComputeTarget` only for the automated ML process. Doing so is as straightforward as assigning a more powerful `RunConfiguration` to the `AutoMLConfig` object's `run_configuration` parameter.

Send data to `AutoMLStep`

In an ML pipeline, the input data must be a `Dataset` object. The highest-performing way is to provide the input data in the form of `OutputTabularDatasetConfig` objects. You create an object of that type with the `read_delimited_files()` on a `OutputFileDatasetConfig`, such as the `prepped_data_path`, such as the `prepped_data_path` object.

```
# type(prepped_data) == OutputTabularDatasetConfig
prepped_data = prepped_data_path.read_delimited_files()
```

Another option is to use `Dataset` objects registered in the workspace:

```
prepped_data = Dataset.get_by_name(ws, 'Data_prepared')
```

Comparing the two techniques:

TECHNIQUE	BENEFITS AND DRAWBACKS
OutputTabularDatasetConfig	Higher performance Natural route from OutputFileDatasetConfig
	Data isn't persisted after pipeline run
Registered Dataset	Lower performance Can be generated in many ways
	Data persists and is visible throughout workspace
	Notebook showing registered Dataset technique

Specify automated ML outputs

The outputs of the AutoMLStep are the final metric scores of the higher-performing model and that model itself. To use these outputs in further pipeline steps, prepare OutputFileDatasetConfig objects to receive them.

```
from azureml.pipeline.core import TrainingOutput, PipelineData

metrics_data = PipelineData(name='metrics_data',
                            datastore=datastore,
                            pipeline_output_name='metrics_output',
                            training_output=TrainingOutput(type='Metrics'))

model_data = PipelineData(name='best_model_data',
                          datastore=datastore,
                          pipeline_output_name='model_output',
                          training_output=TrainingOutput(type='Model'))
```

The snippet above creates the two PipelineData objects for the metrics and model output. Each is named, assigned to the default datastore retrieved earlier, and associated with the particular type of TrainingOutput from the AutoMLStep. Because we assign pipeline_output_name on these PipelineData objects, their values will be available not just from the individual pipeline step, but from the pipeline as a whole, as will be discussed below in the section "Examine pipeline results."

Configure and create the automated ML pipeline step

Once the inputs and outputs are defined, it's time to create the AutoMLConfig and AutoMLStep. The details of the configuration will depend on your task, as described in [Configure automated ML experiments in Python](#). For the Titanic survival classification task, the following snippet demonstrates a simple configuration.

```

from azureml.train.automl import AutoMLConfig
from azureml.pipeline.steps import AutoMLStep

# Change iterations to a reasonable number (50) to get better accuracy
automl_settings = {
    "iteration_timeout_minutes" : 10,
    "iterations" : 2,
    "experiment_timeout_hours" : 0.25,
    "primary_metric" : 'AUC_weighted'
}

automl_config = AutoMLConfig(task = 'classification',
                             path = '.',
                             debug_log = 'automated_ml_errors.log',
                             compute_target = compute_target,
                             run_configuration = aml_run_config,
                             featurization = 'auto',
                             training_data = prepped_data,
                             label_column_name = 'Survived',
                             **automl_settings)

train_step = AutoMLStep(name='AutoML_Classification',
                       automl_config=automl_config,
                       passthru_automl_config=False,
                       outputs=[metrics_data,model_data],
                       enable_default_model_output=False,
                       enable_default_metrics_output=False,
                       allow_reuse=True)

```

The snippet shows an idiom commonly used with `AutoMLConfig`. Arguments that are more fluid (hyperparameter-ish) are specified in a separate dictionary while the values less likely to change are specified directly in the `AutoMLConfig` constructor. In this case, the `automl_settings` specify a brief run: the run will stop after only 2 iterations or 15 minutes, whichever comes first.

The `automl_settings` dictionary is passed to the `AutoMLConfig` constructor as kwargs. The other parameters aren't complex:

- `task` is set to `classification` for this example. Other valid values are `regression` and `forecasting`
- `path` and `debug_log` describe the path to the project and a local file to which debug information will be written
- `compute_target` is the previously defined `compute_target` that, in this example, is an inexpensive CPU-based machine. If you're using AutoML's Deep Learning facilities, you would want to change the compute target to be GPU-based
- `featurization` is set to `auto`. More details can be found in the [Data Featurization](#) section of the automated ML configuration document
- `label_column_name` indicates which column we are interested in predicting
- `training_data` is set to the `OutputTabularDatasetConfig` objects made from the outputs of the data preparation step

The `AutoMLStep` itself takes the `AutoMLConfig` and has, as outputs, the `PipelineData` objects created to hold the metrics and model data.

IMPORTANT

You must set `enable_default_model_output` and `enable_default_metrics_output` to `True` only if you are using `AutoMLStepRun`.

In this example, the automated ML process will perform cross-validations on the `training_data`. You can control

the number of cross-validations with the `n_cross_validations` argument. If you've already split your training data as part of your data preparation steps, you can set `validation_data` to its own `Dataset`.

You might occasionally see the use `x` for data features and `y` for data labels. This technique is deprecated and you should use `training_data` for input.

Register the model generated by automated ML

The last step in a simple ML pipeline is registering the created model. By adding the model to the workspace's model registry, it will be available in the portal and can be versioned. To register the model, write another `PythonScriptStep` that takes the `model_data` output of the `AutoMLStep`.

Write the code to register the model

A model is registered in a `Workspace`. You're probably familiar with using `Workspace.from_config()` to log on to your workspace on your local machine, but there's another way to get the workspace from within a running ML pipeline. The `Run.get_context()` retrieves the active `Run`. This `run` object provides access to many important objects, including the `Workspace` used here.

```
%%writefile register_model.py
from azureml.core.model import Model, Dataset
from azureml.core.run import Run, _OfflineRun
from azureml.core import Workspace
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--model_name", required=True)
parser.add_argument("--model_path", required=True)
args = parser.parse_args()

print(f"model_name : {args.model_name}")
print(f"model_path: {args.model_path}")

run = Run.get_context()
ws = Workspace.from_config() if type(run) == _OfflineRun else run.experiment.workspace

model = Model.register(workspace=ws,
                      model_path=args.model_path,
                      model_name=args.model_name)

print("Registered version {} of model {}".format(model.version, model.name))
```

Write the PythonScriptStep code

The model-registering `PythonScriptStep` uses a `PipelineParameter` for one of its arguments. Pipeline parameters are arguments to pipelines that can be easily set at run-submission time. Once declared, they're passed as normal arguments.

Create and run your automated ML pipeline

Creating and running a pipeline that contains an `AutoMLStep` is no different than a normal pipeline.

```
from azureml.pipeline.core import Pipeline
from azureml.core import Experiment

pipeline = Pipeline(ws, [dataprep_step, train_step, register_step])

experiment = Experiment(workspace=ws, name='titanic_automl')

run = experiment.submit(pipeline, show_output=True)
run.wait_for_completion()
```

The code above combines the data preparation, automated ML, and model-registering steps into a `Pipeline` object. It then creates an `Experiment` object. The `Experiment` constructor will retrieve the named experiment if it exists or create it if necessary. It submits the `Pipeline` to the `Experiment`, creating a `Run` object that will asynchronously run the pipeline. The `wait_for_completion()` function blocks until the run completes.

Examine pipeline results

Once the `run` completes, you can retrieve `PipelineData` objects that have been assigned a `pipeline_output_name`. You can download the results and load them for further processing.

```
metrics_output_port = run.get_pipeline_output('metrics_output')
model_output_port = run.get_pipeline_output('model_output')

metrics_output_port.download('.', show_progress=True)
model_output_port.download('.', show_progress=True)
```

Downloaded files are written to the subdirectory `azureml/{run.id}/`. The metrics file is JSON-formatted and can be converted into a Pandas dataframe for examination.

For local processing, you may need to install relevant packages, such as Pandas, Pickle, the AzureML SDK, and so forth. For this example, it's likely that the best model found by automated ML will depend on XGBoost.

```
!pip install xgboost==0.90
```

```
import pandas as pd
import json

metrics_filename = metrics_output._path_on_datastore
# metrics_filename = path to downloaded file
with open(metrics_filename) as f:
    metrics_output_result = f.read()

deserialized_metrics_output = json.loads(metrics_output_result)
df = pd.DataFrame(deserialized_metrics_output)
df
```

The code snippet above shows the metrics file being loaded from its location on the Azure datastore. You can also load it from the downloaded file, as shown in the comment. Once you've deserialized it and converted it to a Pandas DataFrame, you can see detailed metrics for each of the iterations of the automated ML step.

The model file can be deserialized into a `Model` object that you can use for inferencing, further metrics analysis, and so forth.

```

import pickle

model_filename = model_output._path_on_datastore
# model_filename = path to downloaded file

with open(model_filename, "rb" ) as f:
    best_model = pickle.load(f)

# ... inferencing code not shown ...

```

For more information on loading and working with existing models, see [Use an existing model with Azure Machine Learning](#).

Download the results of an automated ML run

If you've been following along with the article, you'll have an instantiated `run` object. But you can also retrieve completed `Run` objects from the `Workspace` by way of an `Experiment` object.

The workspace contains a complete record of all your experiments and runs. You can either use the portal to find and download the outputs of experiments or use code. To access the records from a historic run, use Azure Machine Learning to find the ID of the run in which you are interested. With that ID, you can choose the specific `run` by way of the `Workspace` and `Experiment`.

```

# Retrieved from Azure Machine Learning web UI
run_id = 'aaaaaaaa-bbbb-cccc-dddd-0123456789AB'
experiment = ws.experiments['titanic_automl']
run = next(run for run in ex.get_runs() if run.id == run_id)

```

You would have to change the strings in the above code to the specifics of your historical run. The snippet above assumes that you've assigned `ws` to the relevant `Workspace` with the normal `from_config()`. The experiment of interest is directly retrieved and then the code finds the `Run` of interest by matching the `run.id` value.

Once you have a `Run` object, you can download the metrics and model.

```

automl_run = next(r for r in run.get_children() if r.name == 'AutoML_Classification')
outputs = automl_run.get_outputs()
metrics = outputs['default_metrics_AutoML_Classification']
model = outputs['default_model_AutoML_Classification']

metrics.get_port_data_reference().download('.')
model.get_port_data_reference().download('.')

```

Each `Run` object contains `StepRun` objects that contain information about the individual pipeline step run. The `run` is searched for the `stepRun` object for the `AutoMLStep`. The metrics and model are retrieved using their default names, which are available even if you don't pass `PipelineData` objects to the `outputs` parameter of the `AutoMLStep`.

Finally, the actual metrics and model are downloaded to your local machine, as was discussed in the "Examine pipeline results" section above.

Next Steps

- Run this Jupyter notebook showing a [complete example of automated ML in a pipeline](#) that uses regression to predict taxi fares
- [Create automated ML experiments without writing code](#)
- Explore a variety of [Jupyter notebooks demonstrating automated ML](#)

- Read about integrating your pipeline in to [End-to-end MLOps](#) or investigate the [MLOps GitHub repository](#)

Publish and track machine learning pipelines

9/21/2022 • 7 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

This article will show you how to share a machine learning pipeline with your colleagues or customers.

Machine learning pipelines are reusable workflows for machine learning tasks. One benefit of pipelines is increased collaboration. You can also version pipelines, allowing customers to use the current model while you're working on a new version.

Prerequisites

- Create an [Azure Machine Learning workspace](#) to hold all your pipeline resources
- [Configure your development environment](#) to install the Azure Machine Learning SDK, or use an [Azure Machine Learning compute instance](#) with the SDK already installed
- Create and run a machine learning pipeline, such as by following [Tutorial: Build an Azure Machine Learning pipeline for batch scoring](#). For other options, see [Create and run machine learning pipelines with Azure Machine Learning SDK](#)

Publish a pipeline

Once you have a pipeline up and running, you can publish a pipeline so that it runs with different inputs. For the REST endpoint of an already published pipeline to accept parameters, you must configure your pipeline to use `PipelineParameter` objects for the arguments that will vary.

1. To create a pipeline parameter, use a `PipelineParameter` object with a default value.

```
from azureml.pipeline.core.graph import PipelineParameter

pipeline_param = PipelineParameter(
    name="pipeline_arg",
    default_value=10)
```

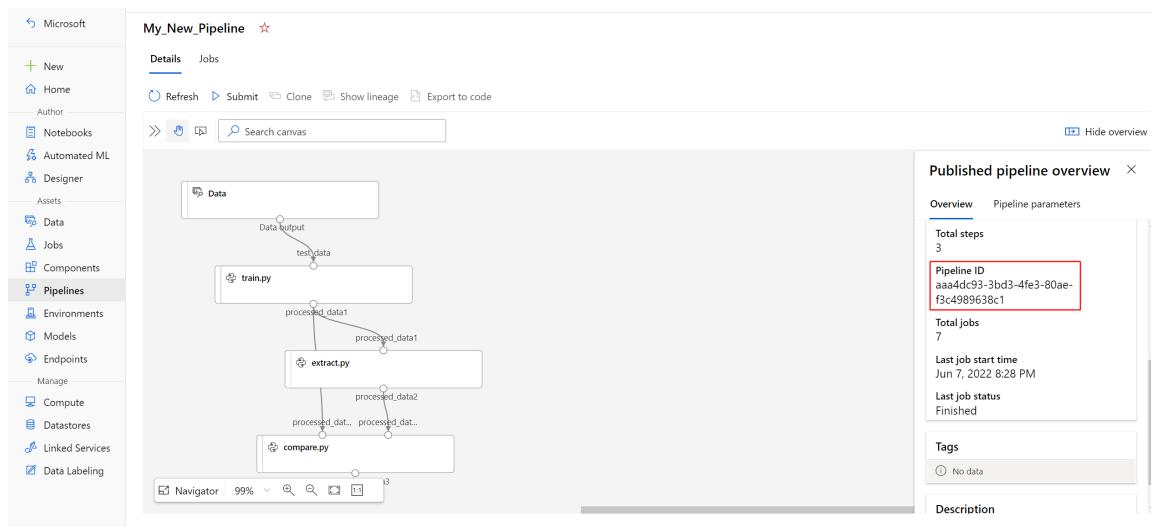
2. Add this `PipelineParameter` object as a parameter to any of the steps in the pipeline as follows:

```
compareStep = PythonScriptStep(
    script_name="compare.py",
    arguments=[ "--comp_data1", comp_data1, "--comp_data2", comp_data2, "--output_data", out_data3, "--param1", pipeline_param],
    inputs=[ comp_data1, comp_data2],
    outputs=[out_data3],
    compute_target=compute_target,
    source_directory=project_folder)
```

3. Publish this pipeline that will accept a parameter when invoked.

```
published_pipeline1 = pipeline_run1.publish_pipeline(
    name="My_Published_Pipeline",
    description="My Published Pipeline Description",
    version="1.0")
```

4. After you publish your pipeline, you can check it in the UI. Pipeline ID is the unique identifier of the published pipeline.



Run a published pipeline

All published pipelines have a REST endpoint. With the pipeline endpoint, you can trigger a run of the pipeline from any external systems, including non-Python clients. This endpoint enables "managed repeatability" in batch scoring and retraining scenarios.

IMPORTANT

If you are using Azure role-based access control (Azure RBAC) to manage access to your pipeline, [set the permissions for your pipeline scenario \(training or scoring\)](#).

To invoke the run of the preceding pipeline, you need an Azure Active Directory authentication header token. Getting such a token is described in the [AzureCliAuthentication class](#) reference and in the [Authentication in Azure Machine Learning](#) notebook.

```
from azureml.pipeline.core import PublishedPipeline
import requests

response = requests.post(published_pipeline1.endpoint,
                         headers=aad_token,
                         json={"ExperimentName": "My_Pipeline",
                               "ParameterAssignments": {"pipeline_arg": 20}})
```

The `json` argument to the POST request must contain, for the `ParameterAssignments` key, a dictionary containing the pipeline parameters and their values. In addition, the `json` argument may contain the following keys:

KEY	DESCRIPTION
<code>ExperimentName</code>	The name of the experiment associated with this endpoint

KEY	DESCRIPTION
Description	Freeform text describing the endpoint
Tags	Freeform key-value pairs that can be used to label and annotate requests
DataSetDefinitionValueAssignments	Dictionary used for changing datasets without retraining (see discussion below)
DataPathAssignments	Dictionary used for changing datapaths without retraining (see discussion below)

Run a published pipeline using C#

The following code shows how to call a pipeline asynchronously from C#. The partial code snippet just shows the call structure and isn't part of a Microsoft sample. It doesn't show complete classes or error handling.

```

[DataContract]
public class SubmitPipelineRunRequest
{
    [DataMember]
    public string ExperimentName { get; set; }

    [DataMember]
    public string Description { get; set; }

    [DataMember(IsRequired = false)]
    public IDictionary<string, string> ParameterAssignments { get; set; }
}

// ... in its own class and method ...
const string RestEndpoint = "your-pipeline-endpoint";

using (HttpClient client = new HttpClient())
{
    var submitPipelineRunRequest = new SubmitPipelineRunRequest()
    {
        ExperimentName = "YourExperimentName",
        Description = "Asynchronous C# REST api call",
        ParameterAssignments = new Dictionary<string, string>
        {
            {
                // Replace with your pipeline parameter keys and values
                "your-pipeline-parameter", "default-value"
            }
        }
    };

    string auth_key = "your-auth-key";
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", auth_key);

    // submit the job
    var requestPayload = JsonConvert.SerializeObject(submitPipelineRunRequest);
    var httpContent = new StringContent(requestPayload, Encoding.UTF8, "application/json");
    var submitResponse = await client.PostAsync(RestEndpoint, httpContent).ConfigureAwait(false);
    if (!submitResponse.IsSuccessStatusCode)
    {
        await WriteFailedResponse(submitResponse); // ... method not shown ...
        return;
    }

    var result = await submitResponse.Content.ReadAsStringAsync().ConfigureAwait(false);
    var obj = JObject.Parse(result);
    // ... use `obj` dictionary to access results
}

```

Run a published pipeline using Java

The following code shows a call to a pipeline that requires authentication (see [Set up authentication for Azure Machine Learning resources and workflows](#)). If your pipeline is deployed publicly, you don't need the calls that produce `authKey`. The partial code snippet doesn't show Java class and exception-handling boilerplate. The code uses `Optional.flatMap` for chaining together functions that may return an empty `Optional`. The use of `flatMap` shortens and clarifies the code, but note that `getRequestBody()` swallows exceptions.

```

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.Optional;
// JSON library
import com.google.gson.Gson;

```

```

String scoringUri = "scoring-endpoint";
String tenantId = "your-tenant-id";
String clientId = "your-client-id";
String clientSecret = "your-client-secret";
String resourceManagerUrl = "https://management.azure.com";
String dataToBeScored = "{ \"ExperimentName\" : \"My_Pipeline\", \"ParameterAssignments\" : {
    \"pipeline_arg\" : \"20\" }}";

HttpClient client = HttpClient.newBuilder().build();
Gson gson = new Gson();

HttpRequest tokenAuthenticationRequest = tokenAuthenticationRequest(tenantId, clientId, clientSecret,
resourceManagerUrl);
Optional<String> authBody = getRequestBody(client, tokenAuthenticationRequest);
Optional<String> authKey = authBody.flatMap(body -> Optional.of(gson.fromJson(body,
AuthenticationBody.class).access_token));
Optional<HttpRequest> scoringRequest = authKey.flatMap(key -> Optional.of(scoringRequest(key, scoringUri,
dataToBeScored)));
Optional<String> scoringResult = scoringRequest.flatMap(req -> getRequestBody(client, req));
// ... etc (`scoringResult.orElse()`) ...

static HttpRequest tokenAuthenticationRequest(String tenantId, String clientId, String clientSecret, String
resourceManagerUrl)
{
    String authUrl = String.format("https://login.microsoftonline.com/%s/oauth2/token", tenantId);
    String clientIdParam = String.format("client_id=%s", clientId);
    String resourceParam = String.format("resource=%s", resourceManagerUrl);
    String clientSecretParam = String.format("client_secret=%s", clientSecret);

    String bodyString = String.format("grant_type=client_credentials&%s&%s&%s", clientIdParam,
resourceParam, clientSecretParam);

    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(authUrl))
        .POST(HttpRequest.BodyPublishers.ofString(bodyString))
        .build();
    return request;
}

static HttpRequest scoringRequest(String authKey, String scoringUri, String dataToBeScored)
{
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(scoringUri))
        .header("Authorization", String.format("Token %s", authKey))
        .POST(HttpRequest.BodyPublishers.ofString(dataToBeScored))
        .build();
    return request;
}

static Optional<String> getRequestBody(HttpClient client, HttpRequest request) {
    try {
        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
        if (response.statusCode() != 200) {
            System.out.println(String.format("Unexpected server response %d", response.statusCode()));
            return Optional.empty();
        }
        return Optional.of(response.body());
    } catch (Exception x) {
        System.out.println(x.toString());
        return Optional.empty();
    }
}

class AuthenticationBody {
    String access_token;
    String token_type;
    int expires_in;
}

```

```
        String scope;
        String refresh_token;
        String id_token;

        AuthenticationBody() {}
    }
```

Changing datasets and datapaths without retraining

You may want to train and inference on different datasets and datapaths. For instance, you may wish to train on a smaller dataset but inference on the complete dataset. You switch datasets with the

`DataSetDefinitionValueAssignments` key in the request's `json` argument. You switch datapaths with `DataPathAssignments`. The technique for both is similar:

1. In your pipeline definition script, create a `PipelineParameter` for the dataset. Create a

`DatasetConsumptionConfig` or `DataPath` from the `PipelineParameter`:

```
tabular_dataset =
Dataset.Tabular.from_delimited_files('https://dprepdata.blob.core.windows.net/demo/Titanic.csv')
tabular_pipeline_param = PipelineParameter(name="tabular_ds_param", default_value=tabular_dataset)
tabular_ds_consumption = DatasetConsumptionConfig("tabular_dataset", tabular_pipeline_param)
```

2. In your ML script, access the dynamically specified dataset using `Run.get_context().input_datasets`:

```
from azureml.core import Run

input_tabular_ds = Run.get_context().input_datasets['tabular_dataset']
dataframe = input_tabular_ds.to_pandas_dataframe()
# ... etc ...
```

Notice that the ML script accesses the value specified for the `DatasetConsumptionConfig` (`tabular_dataset`) and not the value of the `PipelineParameter` (`tabular_ds_param`).

3. In your pipeline definition script, set the `DatasetConsumptionConfig` as a parameter to the

`PipelineScriptStep`:

```
train_step = PythonScriptStep(
    name="train_step",
    script_name="train_with_dataset.py",
    arguments=["--param1", tabular_ds_consumption],
    inputs=[tabular_ds_consumption],
    compute_target=compute_target,
    source_directory=source_directory)

pipeline = Pipeline(workspace=ws, steps=[train_step])
```

4. To switch datasets dynamically in your inferencing REST call, use `DataSetDefinitionValueAssignments`:

```

tabular_ds1 = Dataset.Tabular.from_delimited_files('path_to_training_dataset')
tabular_ds2 = Dataset.Tabular.from_delimited_files('path_to_inference_dataset')
ds1_id = tabular_ds1.id
d22_id = tabular_ds2.id

response = requests.post(rest_endpoint,
                        headers=aad_token,
                        json={
                            "ExperimentName": "MyRestPipeline",
                            "DataSetDefinitionValueAssignments": {
                                "tabular_ds_param": {
                                    "SavedDataSetReference": {"Id": ds1_id #or ds2_id
                                }}}})

```

The notebooks [Showcasing Dataset and PipelineParameter](#) and [Showcasing DataPath and PipelineParameter](#) have complete examples of this technique.

Create a versioned pipeline endpoint

You can create a Pipeline Endpoint with multiple published pipelines behind it. This technique gives you a fixed REST endpoint as you iterate on and update your ML pipelines.

```

from azureml.pipeline.core import PipelineEndpoint

published_pipeline = PublishedPipeline.get(workspace=ws, id="My_Published_Pipeline_id")
pipeline_endpoint = PipelineEndpoint.publish(workspace=ws, name="PipelineEndpointTest",
                                              pipeline=published_pipeline, description="Test description
Notebook")

```

Submit a job to a pipeline endpoint

You can submit a job to the default version of a pipeline endpoint:

```

pipeline_endpoint_by_name = PipelineEndpoint.get(workspace=ws, name="PipelineEndpointTest")
run_id = pipeline_endpoint_by_name.submit("PipelineEndpointExperiment")
print(run_id)

```

You can also submit a job to a specific version:

```

run_id = pipeline_endpoint_by_name.submit("PipelineEndpointExperiment", pipeline_version="0")
print(run_id)

```

The same can be accomplished using the REST API:

```

rest_endpoint = pipeline_endpoint_by_name.endpoint
response = requests.post(rest_endpoint,
                        headers=aad_token,
                        json={"ExperimentName": "PipelineEndpointExperiment",
                              "RunSource": "API",
                              "ParameterAssignments": {"1": "united", "2": "city"}})

```

Use published pipelines in the studio

You can also run a published pipeline from the studio:

1. Sign in to [Azure Machine Learning studio](#).

2. [View your workspace](#).

3. On the left, select **Endpoints**.

4. On the top, select **Pipeline endpoints**.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. The left sidebar has a red box around the 'Endpoints' item under the 'Assets' section. The main content area shows a table titled 'Endpoints' with one row. The row contains the following information:

Name	Description	Modified on	Modified by	Last job submit time	L...	S...
My_New_Pipeline	My Published Pipeline D...	11/11/2019, 3:41:03 PM		11/11/2019, 3:42:41 PM	• Ru	A...

At the top of the main area, there are tabs for 'Real-time endpoints' and 'Pipeline endpoints', with 'Pipeline endpoints' being the active tab and highlighted with a red box. There are also buttons for Refresh, Disable, Enable, and View disabled, along with a search bar.

5. Select a specific pipeline to run, consume, or review results of previous runs of the pipeline endpoint.

Disable a published pipeline

To hide a pipeline from your list of published pipelines, you disable it, either in the studio or from the SDK:

```
# Get the pipeline by using its ID from Azure Machine Learning studio
p = PublishedPipeline.get(ws, id="068f4885-7088-424b-8ce2-eeb9ba5381a6")
p.disable()
```

You can enable it again with `p.enable()`. For more information, see [PublishedPipeline class](#) reference.

Next steps

- Use [these Jupyter notebooks on GitHub](#) to explore machine learning pipelines further.
- See the SDK reference help for the `azureml-pipelines-core` package and the `azureml-pipelines-steps` package.
- See the [how-to](#) for tips on debugging and troubleshooting pipelines.

How to use Apache Spark (powered by Azure Synapse Analytics) in your machine learning pipeline (preview)

9/21/2022 • 8 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement, and it's not recommended for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

In this article, you'll learn how to use Apache Spark pools powered by Azure Synapse Analytics as the compute target for a data preparation step in an Azure Machine Learning pipeline. You'll learn how a single pipeline can use compute resources suited for the specific step, such as data preparation or training. You'll see how data is prepared for the Spark step and how it's passed to the next step.

Prerequisites

- Create an [Azure Machine Learning workspace](#) to hold all your pipeline resources.
- [Configure your development environment](#) to install the Azure Machine Learning SDK, or use an [Azure Machine Learning compute instance](#) with the SDK already installed.
- Create an Azure Synapse Analytics workspace and Apache Spark pool (see [Quickstart: Create a serverless Apache Spark pool using Synapse Studio](#)).

Link your Azure Machine Learning workspace and Azure Synapse Analytics workspace

You create and administer your Apache Spark pools in an Azure Synapse Analytics workspace. To integrate an Apache Spark pool with an Azure Machine Learning workspace, you must [link to the Azure Synapse Analytics workspace](#).

Once your Azure Machine Learning workspace and your Azure Synapse Analytics workspaces are linked, you can attach an Apache Spark pool via

- [Azure Machine Learning studio](#)
- [Python SDK \(as elaborated below\)](#)
- Azure Resource Manager (ARM) template (see this [Example ARM template](#)).
 - You can use the command line to follow the ARM template, add the linked service, and attach the Apache Spark pool with the following code:

```
az deployment group create --name --resource-group <rg_name> --template-file "azuredeploy.json" --parameters @"/azuredetect.deploy.parameters.json"
```

IMPORTANT

To link to the Azure Synapse Analytics workspace successfully, you must have the Owner role in the Azure Synapse Analytics workspace resource. Check your access in the Azure portal.

The linked service will get a system-assigned managed identity (SAI) when you create it. You must assign this link service SAI the "Synapse Apache Spark administrator" role from Synapse Studio so that it can submit the Spark job (see [How to manage Synapse RBAC role assignments in Synapse Studio](#)).

You must also give the user of the Azure Machine Learning workspace the role "Contributor" from Azure portal of resource management.

Retrieve the link between your Azure Synapse Analytics workspace and your Azure Machine Learning workspace

You can retrieve linked services in your workspace with code such as:

```
from azureml.core import Workspace, LinkedService, SynapseWorkspaceLinkedServiceConfiguration

ws = Workspace.from_config()

for service in LinkedService.list(ws) :
    print(f"Service: {service}")

# Retrieve a known linked service
linked_service = LinkedService.get(ws, 'synapseslink1')
```

First, `Workspace.from_config()` accesses your Azure Machine Learning workspace using the configuration in `config.json` (see [Create a workspace configuration file](#)). Then, the code prints all of the linked services available in the Workspace. Finally, `LinkedService.get()` retrieves a linked service named `'synapseslink1'`.

Attach your Apache spark pool as a compute target for Azure Machine Learning

To use your Apache spark pool to power a step in your machine learning pipeline, you must attach it as a `ComputeTarget` for the pipeline step, as shown in the following code.

```
from azureml.core.compute import SynapseCompute, ComputeTarget

attach_config = SynapseCompute.attach_configuration(
    linked_service = linked_service,
    type="SynapseSpark",
    pool_name="spark01") # This name comes from your Synapse workspace

synapse_compute=ComputeTarget.attach(
    workspace=ws,
    name='link1-spark01',
    attach_configuration=attach_config)

synapse_compute.wait_for_completion()
```

The first step is to configure the `SynapseCompute`. The `linked_service` argument is the `LinkedService` object you created or retrieved in the previous step. The `type` argument must be `SynapseSpark`. The `pool_name` argument in `SynapseCompute.attach_configuration()` must match that of an existing pool in your Azure Synapse Analytics workspace. For more information on creating an Apache spark pool in the Azure Synapse Analytics workspace, see [Quickstart: Create a serverless Apache Spark pool using Synapse Studio](#). The type of `attach_config` is

```
ComputeTargetAttachConfiguration .
```

Once the configuration is created, you create a machine learning `computeTarget` by passing in the `Workspace`, `ComputeTargetAttachConfiguration`, and the name by which you'd like to refer to the compute within the machine learning workspace. The call to `ComputeTarget.attach()` is asynchronous, so the sample blocks until the call completes.

Create a `SynapseSparkStep` that uses the linked Apache Spark pool

The sample notebook [Spark job on Apache spark pool](#) defines a simple machine learning pipeline. First, the notebook defines a data preparation step powered by the `synapse_compute` defined in the previous step. Then, the notebook defines a training step powered by a compute target better suited for training. The sample notebook uses the Titanic survival database to demonstrate data input and output; it doesn't actually clean the data or make a predictive model. Since there's no real training in this sample, the training step uses an inexpensive, CPU-based compute resource.

Data flows into a machine learning pipeline by way of `DatasetConsumptionConfig` objects, which can hold tabular data or sets of files. The data often comes from files in blob storage in a workspace's datastore. The following code shows some typical code for creating input for a machine learning pipeline:

```
from azureml.core import Dataset

datastore = ws.get_default_datastore()
file_name = 'Titanic.csv'

titanic_tabular_dataset = Dataset.Tabular.from_delimited_files(path=[(datastore, file_name)])
step1_input1 = titanic_tabular_dataset.as_named_input("tabular_input")

# Example only: it wouldn't make sense to duplicate input data, especially one as tabular and the other as
# files
titanic_file_dataset = Dataset.File.from_files(path=[(datastore, file_name)])
step1_input2 = titanic_file_dataset.as_named_input("file_input").as_hdfs()
```

The above code assumes that the file `Titanic.csv` is in blob storage. The code shows how to read the file as a `TabularDataset` and as a `FileDataset`. This code is for demonstration purposes only, as it would be confusing to duplicate inputs or to interpret a single data source as both a table-containing resource and just as a file.

IMPORTANT

In order to use a `FileDataset` as input, your `azureml-core` version must be at least `1.20.0`. How to specify this using the `Environment` class is discussed below.

When a step completes, you may choose to store output data using code similar to:

```
from azureml.data import HDFSOutputDatasetConfig
step1_output = HDFSOutputDatasetConfig(destination=
(datastore,"test")).register_on_complete(name="registered_dataset")
```

In this case, the data would be stored in the `datastore` in a file called `test` and would be available within the machine learning workspace as a `Dataset` with the name `registered_dataset`.

In addition to data, a pipeline step may have per-step Python dependencies. Individual `SynapseSparkStep` objects can specify their precise Azure Synapse Apache Spark configuration, as well. This is shown in the following code, which specifies that the `azureml-core` package version must be at least `1.20.0`. (As mentioned previously, this requirement for `azureml-core` is needed to use a `FileDataset` as an input.)

```
from azureml.core.environment import Environment
from azureml.pipeline.steps import SynapseSparkStep

env = Environment(name="myenv")
env.python.conda_dependencies.add_pip_package("azureml-core>=1.20.0")

step_1 = SynapseSparkStep(name = 'synapse-spark',
                          file = 'dataprep.py',
                          source_directory=".code",
                          inputs=[step1_input1, step1_input2],
                          outputs=[step1_output],
                          arguments = ["--tabular_input", step1_input1,
                                      "--file_input", step1_input2,
                                      "--output_dir", step1_output],
                          compute_target = 'link1-spark01',
                          driver_memory = "7g",
                          driver_cores = 4,
                          executor_memory = "7g",
                          executor_cores = 2,
                          num_executors = 1,
                          environment = env)
```

The above code specifies a single step in the Azure machine learning pipeline. This step's `environment` specifies a specific `azureml-core` version and could add other conda or pip dependencies as necessary.

The `SynapseSparkStep` will zip and upload from the local computer the subdirectory `./code`. That directory will be recreated on the compute server and the step will run the file `dataprep.py` from that directory. The `inputs` and `outputs` of that step are the `step1_input1`, `step1_input2`, and `step1_output` objects previously discussed. The easiest way to access those values within the `dataprep.py` script is to associate them with named `arguments`.

The next set of arguments to the `SynapseSparkStep` constructor control Apache Spark. The `compute_target` is the `'link1-spark01'` that we attached as a compute target previously. The other parameters specify the memory and cores we'd like to use.

The sample notebook uses the following code for `dataprep.py`:

```

import os
import sys
import azureml.core
from pyspark.sql import SparkSession
from azureml.core import Run, Dataset

print(azureml.core.VERSION)
print(os.environ)

import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--tabular_input")
parser.add_argument("--file_input")
parser.add_argument("--output_dir")
args = parser.parse_args()

# use dataset sdk to read tabular dataset
run_context = Run.get_context()
dataset = Dataset.get_by_id(run_context.experiment.workspace.id=args.tabular_input)
sdf = dataset.to_spark_dataframe()
sdf.show()

# use hdfs path to read file dataset
spark= SparkSession.builder.getOrCreate()
sdf = spark.read.option("header", "true").csv(args.file_input)
sdf.show()

sdf.coalesce(1).write\
.option("header", "true")\
.mode("append")\
.csv(args.output_dir)

```

This "data preparation" script doesn't do any real data transformation, but illustrates how to retrieve data, convert it to a spark dataframe, and how to do some basic Apache Spark manipulation. You can find the output in Azure Machine Learning Studio by opening the child job, choosing the **Outputs + logs** tab, and opening the `logs/azureml/driver/stdout` file, as shown in the following figure.

```

13 Starting the daemon thread to refresh tokens in background for process with pid = 10185
14 +-----+-----+-----+-----+-----+
15 |PassengerId|Survived|Pclass|      Name| Sex| Age|SibSp|Parch|   Ticket| Fare|Cabin|Embarked|
16 +-----+-----+-----+-----+-----+
17 | 1| 1| 0| 3| Braund, Mr. Owen G.| male| 23.0| 1| 0| A/5 21171| 7.25| null| S|
18 | 2| 1| 1| 1| Cumings, Mrs. John S. | female| 38.0| 1| 0| PC 17599| 71.2833| C85| C|
19 | 3| 1| 3| Heikkinen, Miss. Laina | female| 26.0| 0| 0| STON/O2. 3101282| 7.925| null| S|
20 | 4| 1| 1| 1| Futrelle, Mrs. Jaqueline | female| 35.0| 1| 0| 113883| 53.1| C123| S|
21 | 5| 0| 3| Allen, Mr. William C. | male| 35.0| 1| 0| 373450| 8.85| null| S|
22 | 6| 0| 1| 3| de Moussa, Mr. James | male| 26.0| 0| 0| 322112| 8.05| null| Q|
23 | 7| 0| 1| 3| McCarthy, Mrs. Tina H. | female| 54.0| 0| 0| 17463| 51.025| E46| S|
24 | 8| 0| 1| 3| Palsson, Master. Edvard | male| 2.0| 3| 349989| 21.075| null| S|
25 | 9| 1| 3| Johnson, Mrs. Oscar W. | female| 27.0| 0| 2| 347742| 11.1333| null| S|
26 | 10| 1| 2| Nasser, Mrs. Nicholas | female| 14.0| 1| 0| 237736| 30.0788| null| C|
27 | 11| 1| 3| Sandstrom, Miss. Sophie | female| 4.0| 1| 1| PP 95491| 16.7| G6| S|
28 | 12| 1| 1| Bonnell, Miss. Elizabeth | female| 50.0| 0| 0| 113783| 26.55| C103| S|
29 | 13| 0| 3| Sandstrom, Miss. Sophie | female| 4.0| 0| 0| A/5 21251| 16.7| G6| S|
30 | 14| 0| 3| Andersson, Mr. Anselm | male| 39.0| 1| 5| 347082| 31.275| null| S|
31 | 15| 0| 3| Westrom, Miss. Hilda | female| 14.0| 0| 0| 358486| 7.8542| null| S|
32 | 16| 1| 2| Hewlett, Mrs. Mary | female| 55.0| 0| 0| 248706| 16.0| null| S|
33 | 17| 0| 1| Rice, Master. Eugene | male| 2.0| 4| 1| 382652| 29.125| null| Q|
34 | 18| 1| 2| Williams, Mr. Charles | male| null| 0| 0| 244373| 13.0| null| S|
35 | 19| 0| 1| Vanderveen, Mr. George | female| 31.0| 1| 0| 347563| 18.0| null| S|
36 | 20| 1| 3| Hasselman, Mrs. Sophie | female| null| 0| 0| 2649| 7.225| null| C|
37 +-----+
38 only showing top 20 rows
39
40 +-----+-----+-----+-----+-----+
41 |PassengerId|Survived|Pclass|      Name| Sex| Age|SibSp|Parch|   Ticket| Fare|Cabin|Embarked|
42 +-----+-----+-----+-----+-----+
43 | 1| 0| 3| Braund, Mr. Owen G. | male| 22| 1| 0| A/5 21171| 7.25| null| S|
44 | 2| 1| 1| Cumings, Mrs. John S. | female| 38| 1| 0| PC 17599| 71.2833| C85| C|
45 | 3| 1| 3| Heikkinen, Miss. Laina | female| 26| 0| 0| STON/O2. 3101282| 7.925| null| S|
46 | 4| 1| 1| Futrelle, Mrs. Jaqueline | female| 35| 1| 0| 113883| 53.1| C123| S|
47 | 5| 0| 3| Allen, Mr. William C. | male| 35| 1| 0| 373450| 8.85| null| S|
48 | 6| 0| 1| de Moussa, Mr. James | male| null| 0| 0| 322112| 8.05| null| Q|
49 | 7| 0| 1| McCarthy, Mrs. Tina H. | female| 54| 0| 0| 17463| 51.025| E46| S|
50 | 8| 0| 1| Palsson, Master. Edvard | male| 2| 3| 349989| 21.075| null| S|
51 | 9| 1| 3| Johnson, Mrs. Oscar W. | female| 27| 0| 2| 347742| 11.1333| null| S|
52 | 10| 1| 2| Nasser, Mrs. Nicholas | female| 14| 1| 0| 237736| 30.0788| null| C|
53 | 11| 1| 3| Sandstrom, Miss. Sophie | female| 4| 1| 1| PP 95491| 16.7| G6| S|
54 | 12| 1| 1| Bonnell, Miss. Elizabeth | female| 58| 0| 0| 113783| 26.55| C103| S|
55 +-----+
56

```

Use the `synapseSparkStep` in a pipeline

The following example uses the output from the `SynapseSparkStep` created in the [previous section](#). Other steps in the pipeline may have their own unique environments and run on different compute resources appropriate to the task at hand. The sample notebook runs the "training step" on a small CPU cluster:

```
from azureml.core.compute import AmlCompute

cpu_cluster_name = "cpucluster"

if cpu_cluster_name in ws.compute_targets:
    cpu_cluster = ComputeTarget(workspace=ws, name=cpu_cluster_name)
    print('Found existing cluster, use it.')
else:
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2', max_nodes=1)
    cpu_cluster = ComputeTarget.create(ws, cpu_cluster_name, compute_config)
    print('Allocating new CPU compute cluster')

cpu_cluster.wait_for_completion(show_output=True)

step2_input = step1_output.as_input("step2_input").as_download()

step_2 = PythonScriptStep(script_name="train.py",
                         arguments=[step2_input],
                         inputs=[step2_input],
                         compute_target=cpu_cluster_name,
                         source_directory=".code",
                         allow_reuse=False)
```

The code above creates the new compute resource if necessary. Then, the `step1_output` result is converted to input for the training step. The `as_download()` option means that the data will be moved onto the compute resource, resulting in faster access. If the data was so large that it wouldn't fit on the local compute hard drive, you would use the `as_mount()` option to stream the data via the FUSE filesystem. The `compute_target` of this second step is `'cpucluster'`, not the `'link1-spark01'` resource you used in the data preparation step. This step uses a simple program `train.py` instead of the `dataprep.py` you used in the previous step. You can see the details of `train.py` in the sample notebook.

Once you've defined all of your steps, you can create and run your pipeline.

```
from azureml.pipeline.core import Pipeline

pipeline = Pipeline(workspace=ws, steps=[step_1, step_2])
pipeline_run = pipeline.submit('synapse-pipeline', regenerate_outputs=True)
```

The above code creates a pipeline consisting of the data preparation step on Apache Spark pools powered by Azure Synapse Analytics (`step_1`) and the training step (`step_2`). Azure calculates the execution graph by examining the data dependencies between the steps. In this case, there's only a straightforward dependency that `step2_input` necessarily requires `step1_output`.

The call to `pipeline.submit` creates, if necessary, an Experiment called `synapse-pipeline` and asynchronously begins a Job within it. Individual steps within the pipeline are run as Child Jobs of this main job and can be monitored and reviewed in the Experiments page of Studio.

Next steps

- [Publish and track machine learning pipelines](#)
- [Monitor Azure Machine Learning](#)

- Use automated ML in an Azure Machine Learning pipeline in Python

Trigger machine learning pipelines

9/21/2022 • 6 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this article, you'll learn how to programmatically schedule a pipeline to run on Azure. You can create a schedule based on elapsed time or on file-system changes. Time-based schedules can be used to take care of routine tasks, such as monitoring for data drift. Change-based schedules can be used to react to irregular or unpredictable changes, such as new data being uploaded or old data being edited. After learning how to create schedules, you'll learn how to retrieve and deactivate them. Finally, you'll learn how to use other Azure services, Azure Logic App and Azure Data Factory, to run pipelines. An Azure Logic App allows for more complex triggering logic or behavior. Azure Data Factory pipelines allow you to call a machine learning pipeline as part of a larger data orchestration pipeline.

Prerequisites

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#).
- A Python environment in which the Azure Machine Learning SDK for Python is installed. For more information, see [Create and manage reusable environments for training and deployment with Azure Machine Learning](#).
- A Machine Learning workspace with a published pipeline. You can use the one built in [Create and run machine learning pipelines with Azure Machine Learning SDK](#).

Trigger pipelines with Azure Machine Learning SDK for Python

To schedule a pipeline, you'll need a reference to your workspace, the identifier of your published pipeline, and the name of the experiment in which you wish to create the schedule. You can get these values with the following code:

```
import azureml.core
from azureml.core import Workspace
from azureml.pipeline.core import Pipeline, PublishedPipeline
from azureml.core.experiment import Experiment

ws = Workspace.from_config()

experiments = Experiment.list(ws)
for experiment in experiments:
    print(experiment.name)

published_PIPELINES = PublishedPipeline.list(ws)
for published_PIPELINE in published_PIPELINES:
    print(f"{published_PIPELINE.name},{published_PIPELINE.id}")

experiment_name = "MyExperiment"
pipeline_id = "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeee"
```

Create a schedule

To run a pipeline on a recurring basis, you'll create a schedule. A `schedule` associates a pipeline, an experiment, and a trigger. The trigger can either be a `ScheduleRecurrence` that describes the wait between jobs or a Datastore

path that specifies a directory to watch for changes. In either case, you'll need the pipeline identifier and the name of the experiment in which to create the schedule.

At the top of your Python file, import the `Schedule` and `ScheduleRecurrence` classes:

```
from azureml.pipeline.core.schedule import ScheduleRecurrence, Schedule
```

Create a time-based schedule

The `ScheduleRecurrence` constructor has a required `frequency` argument that must be one of the following strings: "Minute", "Hour", "Day", "Week", or "Month". It also requires an integer `interval` argument specifying how many of the `frequency` units should elapse between schedule starts. Optional arguments allow you to be more specific about starting times, as detailed in the [ScheduleRecurrence SDK docs](#).

Create a `Schedule` that begins a job every 15 minutes:

```
recurrence = ScheduleRecurrence(frequency="Minute", interval=15)
recurring_schedule = Schedule.create(ws, name="MyRecurringSchedule",
                                      description="Based on time",
                                      pipeline_id=pipeline_id,
                                      experiment_name=experiment_name,
                                      recurrence=recurrence)
```

Create a change-based schedule

Pipelines that are triggered by file changes may be more efficient than time-based schedules. When you want to do something before a file is changed, or when a new file is added to a data directory, you can preprocess that file. You can monitor any changes to a datastore or changes within a specific directory within the datastore. If you monitor a specific directory, changes within subdirectories of that directory will *not* trigger a job.

NOTE

Change-based schedules only supports monitoring Azure Blob storage.

To create a file-reactive `Schedule`, you must set the `datastore` parameter in the call to `Schedule.create`. To monitor a folder, set the `path_on_datastore` argument.

The `polling_interval` argument allows you to specify, in minutes, the frequency at which the datastore is checked for changes.

If the pipeline was constructed with a [DataPath PipelineParameter](#), you can set that variable to the name of the changed file by setting the `data_path_parameter_name` argument.

```
datastore = Datastore(workspace=ws, name="workspaceblobstore")

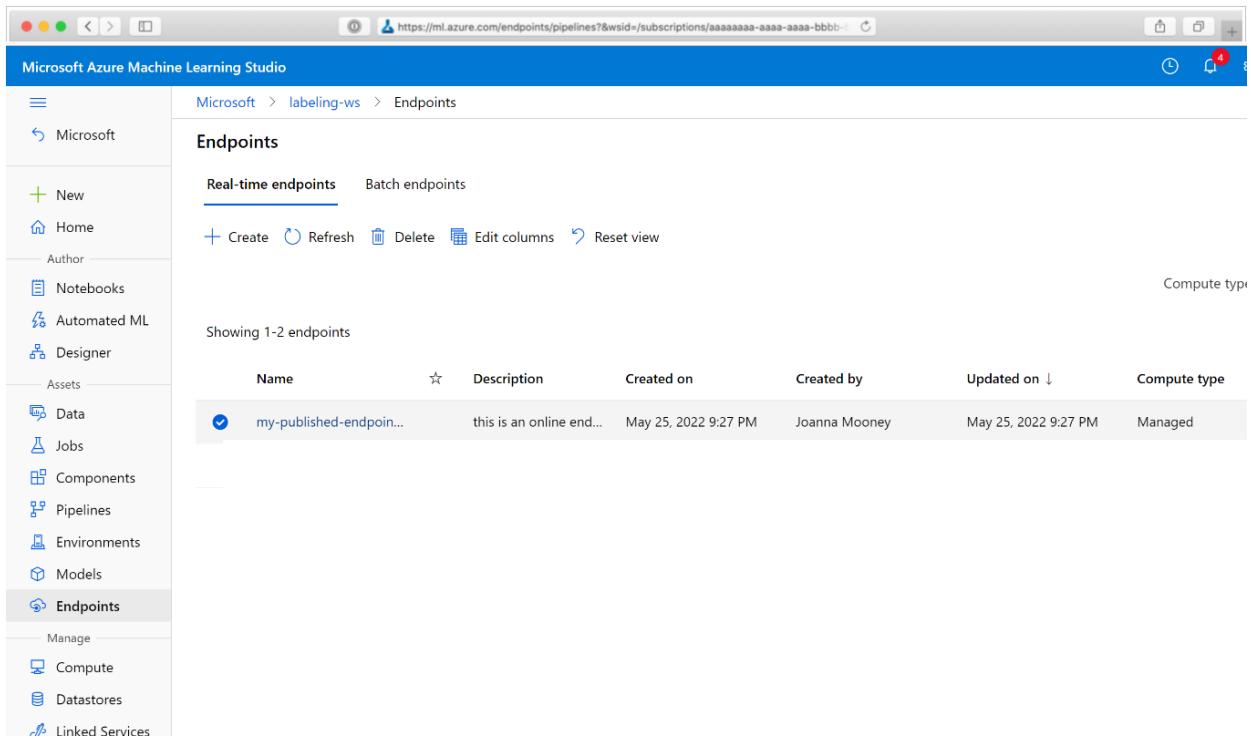
reactive_schedule = Schedule.create(ws, name="MyReactiveSchedule", description="Based on input file
change.", pipeline_id=pipeline_id, experiment_name=experiment_name, datastore=datastore,
data_path_parameter_name="input_data")
```

Optional arguments when creating a schedule

In addition to the arguments discussed previously, you may set the `status` argument to `"Disabled"` to create an inactive schedule. Finally, the `continue_on_step_failure` allows you to pass a Boolean that will override the pipeline's default failure behavior.

View your scheduled pipelines

In your Web browser, navigate to Azure Machine Learning. From the **Endpoints** section of the navigation panel, choose **Pipeline endpoints**. This takes you to a list of the pipelines published in the Workspace.



Name	Description	Created on	Created by	Updated on	Compute type
my-published-endpoint...	this is an online end...	May 25, 2022 9:27 PM	Joanna Mooney	May 25, 2022 9:27 PM	Managed

In this page you can see summary information about all the pipelines in the Workspace: names, descriptions, status, and so forth. Drill in by clicking in your pipeline. On the resulting page, there are more details about your pipeline and you may drill down into individual jobs.

Deactivate the pipeline

If you have a `Pipeline` that is published, but not scheduled, you can disable it with:

```
pipeline = PublishedPipeline.get(ws, id=pipeline_id)
pipeline.disable()
```

If the pipeline is scheduled, you must cancel the schedule first. Retrieve the schedule's identifier from the portal or by running:

```
ss = Schedule.list(ws)
for s in ss:
    print(s)
```

Once you have the `schedule_id` you wish to disable, run:

```
def stop_by_schedule_id(ws, schedule_id):
    s = next(s for s in Schedule.list(ws) if s.id == schedule_id)
    s.disable()
    return s

stop_by_schedule_id(ws, schedule_id)
```

If you then run `Schedule.list(ws)` again, you should get an empty list.

Use Azure Logic Apps for complex triggers

More complex trigger rules or behavior can be created using an [Azure Logic App](#).

To use an Azure Logic App to trigger a Machine Learning pipeline, you'll need the REST endpoint for a published Machine Learning pipeline. [Create and publish your pipeline](#). Then find the REST endpoint of your `PublishedPipeline` by using the pipeline ID:

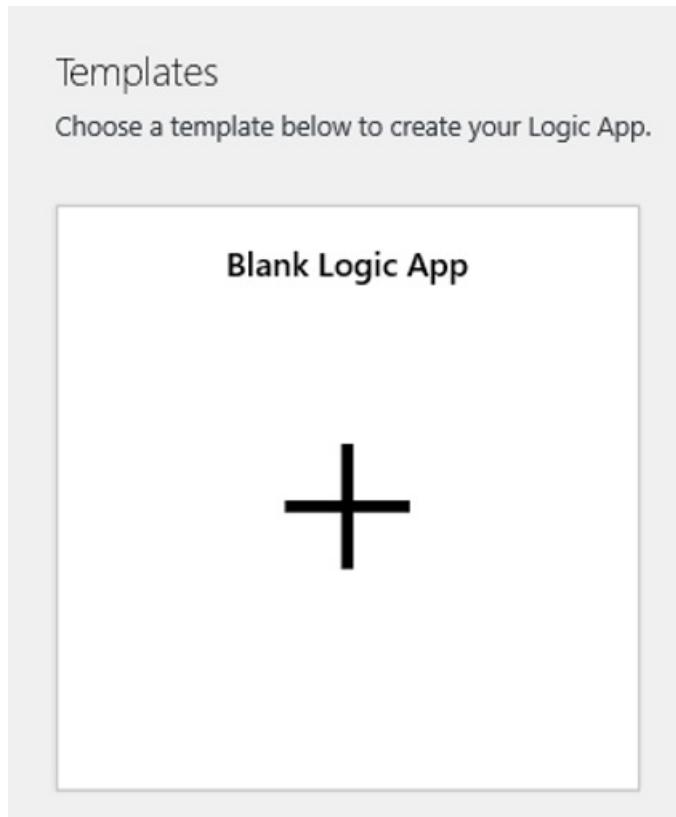
```
# You can find the pipeline ID in Azure Machine Learning studio  
  
published_pipeline = PublishedPipeline.get(ws, id=<pipeline-id-here>)  
published_pipeline.endpoint
```

Create a Logic App

Now create an [Azure Logic App](#) instance. If you wish, [use an integration service environment \(ISE\)](#) and [set up a customer-managed key](#) for use by your Logic App.

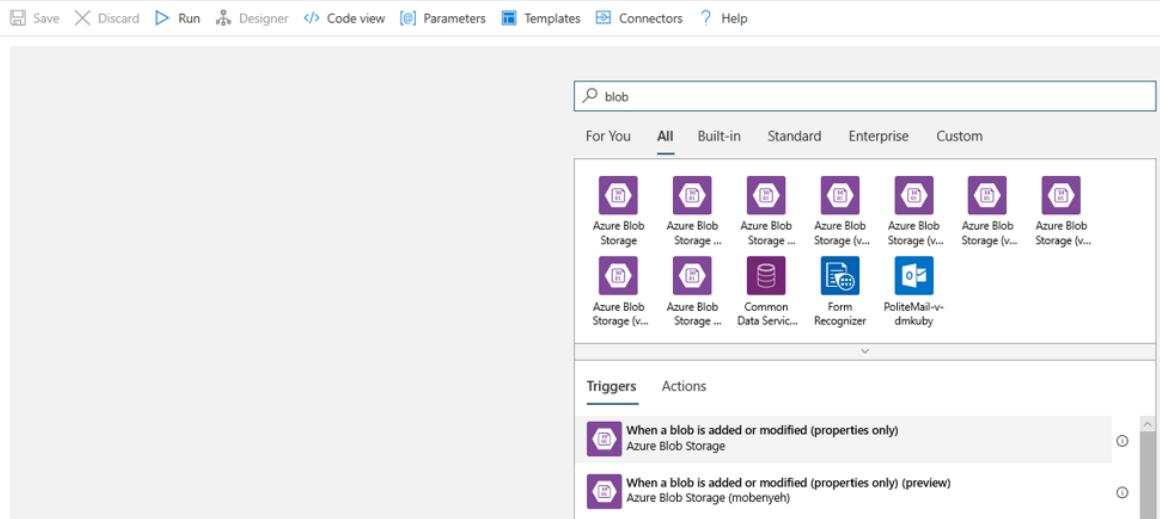
Once your Logic App has been provisioned, use these steps to configure a trigger for your pipeline:

1. [Create a system-assigned managed identity](#) to give the app access to your Azure Machine Learning Workspace.
2. Navigate to the Logic App Designer view and select the Blank Logic App template.



3. In the Designer, search for **blob**. Select the **When a blob is added or modified (properties only)** trigger and add this trigger to your Logic App.

Logic Apps Designer



4. Fill in the connection info for the Blob storage account you wish to monitor for blob additions or modifications. Select the Container to monitor.

Choose the **Interval** and **Frequency** to poll for updates that work for you.

NOTE

This trigger will monitor the selected Container but won't monitor subfolders.

5. Add an HTTP action that will run when a new or modified blob is detected. Select + New Step, then search for and select the HTTP action.

Choose an action

HTTP

For You All Built-in Standard Enterprise Custom

 HTTP	 Office 365 Outlook	 Azure Blob Storage	 Aquaforest PDF	 Azure Blob Storage ...	 Azure Blob Storage ...	 Azure Blob Storage (v...)
----------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------

Triggers Actions See more

-  Extract PDF pages by barcode
Aquaforest PDF
-  Extract PDF pages by text
Aquaforest PDF
-  Get barcode value
Aquaforest PDF
-  Get text from PDF
Aquaforest PDF
-  OCR PDF or images
Aquaforest PDF

Use the following settings to configure your action:

SETTING	VALUE
HTTP action	POST
URI	the endpoint to the published pipeline that you found as a Prerequisite
Authentication mode	Managed Identity

1. Set up your schedule to set the value of any [DataPath PipelineParameters](#) you may have:

```
{
  "DataPathAssignments": {
    "input_datapath": {
      "DataStoreName": "<datastore-name>",
      "RelativePath": "@{triggerBody()?['Name']}"
    }
  },
  "ExperimentName": "MyRestPipeline",
  "ParameterAssignments": {
    "input_string": "sample_string3"
  },
  "RunSource": "SDK"
}
```

Use the `DataStoreName` you added to your workspace as a [Prerequisite](#).

The screenshot shows the Azure Machine Learning studio interface with the 'HTTP' tab selected. The configuration details are as follows:

- * Method:** POST
- * URI:** `https://eastus2.aether.ms/api/v1.0/subscriptions/b8c23406-f9b5-4ccb-8a65-a8cb5dcd6a5a/resourceGroups/aesviennatesteuap/providers/Microsoft.MachineLearningServices/workspaces/elihop-cuseuap/PipelineRuns/PipelineSubmit/7b1817a3-593a-42fe-a3f9-8b149860fe95`
- Headers:** Enter key | Enter value
- Queries:** Enter key | Enter value
- Body:**

```
{
  "DataPathAssignments": {
    "input_datapath": {
      "DataStoreName": "workspaceblobstore",
      "RelativePath": "@{# List of Files Name }"
    }
  },
  "ExperimentName": "MyRestPipeline",
  "ParameterAssignments": {
    "input_string": "sample_string3"
  },
  "RunSource": "SDK"
}
```
- * Authentication:** Managed Identity

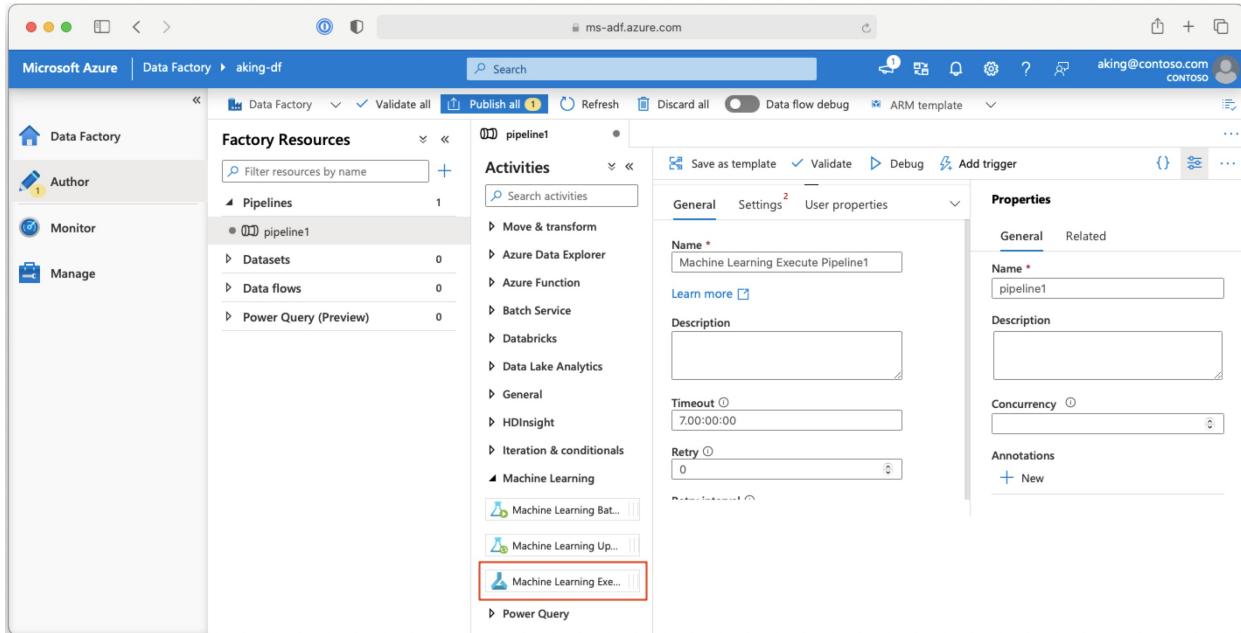
2. Select **Save** and your schedule is now ready.

IMPORTANT

If you are using Azure role-based access control (Azure RBAC) to manage access to your pipeline, [set the permissions for your pipeline scenario \(training or scoring\)](#).

Call machine learning pipelines from Azure Data Factory pipelines

In an Azure Data Factory pipeline, the *Machine Learning Execute Pipeline* activity runs an Azure Machine Learning pipeline. You can find this activity in the Data Factory's authoring page under the *Machine Learning* category:



Next steps

In this article, you used the Azure Machine Learning SDK for Python to schedule a pipeline in two different ways. One schedule recurs based on elapsed clock time. The other schedule jobs if a file is modified on a specified **Datastore** or within a directory on that store. You saw how to use the portal to examine the pipeline and individual jobs. You learned how to disable a schedule so that the pipeline stops running. Finally, you created an Azure Logic App to trigger a pipeline.

For more information, see:

[Use Azure Machine Learning Pipelines for batch scoring](#)

- Learn more about [pipelines](#)
- Learn more about [exploring Azure Machine Learning with Jupyter](#)

Convert ML experiments to production Python code

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO:  [Python SDK azureml v1](#)

In this tutorial, you learn how to convert Jupyter notebooks into Python scripts to make it testing and automation friendly using the MLOpsPython code template and Azure Machine Learning. Typically, this process is used to take experimentation / training code from a Jupyter notebook and convert it into Python scripts. Those scripts can then be used testing and CI/CD automation in your production environment.

A machine learning project requires experimentation where hypotheses are tested with agile tools like Jupyter Notebook using real datasets. Once the model is ready for production, the model code should be placed in a production code repository. In some cases, the model code must be converted to Python scripts to be placed in the production code repository. This tutorial covers a recommended approach on how to export experimentation code to Python scripts.

In this tutorial, you learn how to:

- Clean nonessential code
- Refactor Jupyter Notebook code into functions
- Create Python scripts for related tasks
- Create unit tests

Prerequisites

- Generate the [MLOpsPython template](#) and use the `experimentation/Diabetes Ridge Regression.ipynb` and `experimentation/Diabetes Ridge Regression Scoring.ipynb` notebooks. These notebooks are used as an example of converting from experimentation to production. You can find these notebooks at <https://github.com/microsoft/MLOpsPython/tree/master/experimentation>.
- Install `nbconvert`. Follow only the installation instructions under section [Installing nbconvert](#) on the [Installation](#) page.

Remove all nonessential code

Some code written during experimentation is only intended for exploratory purposes. Therefore, the first step to convert experimental code into production code is to remove this nonessential code. Removing nonessential code will also make the code more maintainable. In this section, you'll remove code from the `experimentation/Diabetes Ridge Regression Training.ipynb` notebook. The statements printing the shape of `x` and `y` and the cell calling `features.describe` are just for data exploration and can be removed. After removing nonessential code, `experimentation/Diabetes Ridge Regression Training.ipynb` should look like the following code without markdown:

```

from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import joblib
import pandas as pd

sample_data = load_diabetes()

df = pd.DataFrame(
    data=sample_data.data,
    columns=sample_data.feature_names)
df['Y'] = sample_data.target

X = df.drop('Y', axis=1).values
y = df['Y'].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=0)
data = {"train": {"X": X_train, "y": y_train},
        "test": {"X": X_test, "y": y_test}}

args = {
    "alpha": 0.5
}

reg_model = Ridge(**args)
reg_model.fit(data["train"]["X"], data["train"]["y"])

preds = reg_model.predict(data["test"]["X"])
mse = mean_squared_error(preds, y_test)
metrics = {"mse": mse}
print(metrics)

model_name = "sklearn_regression_model.pkl"
joblib.dump(value=reg, filename=model_name)

```

Refactor code into functions

Second, the Jupyter code needs to be refactored into functions. Refactoring code into functions makes unit testing easier and makes the code more maintainable. In this section, you'll refactor:

- The Diabetes Ridge Regression Training notebook([experimentation/Diabetes Ridge Regression Training.ipynb](#))
- The Diabetes Ridge Regression Scoring notebook([experimentation/Diabetes Ridge Regression Scoring.ipynb](#))

Refactor Diabetes Ridge Regression Training notebook into functions

In [experimentation/Diabetes Ridge Regression Training.ipynb](#), complete the following steps:

1. Create a function called `split_data` to split the data frame into test and train data. The function should take the dataframe `df` as a parameter, and return a dictionary containing the keys `train` and `test`.

Move the code under the *Split Data into Training and Validation Sets* heading into the `split_data` function and modify it to return the `data` object.

2. Create a function called `train_model`, which takes the parameters `data` and `args` and returns a trained model.

Move the code under the heading *Training Model on Training Set* into the `train_model` function and modify it to return the `reg_model` object. Remove the `args` dictionary, the values will come from the `args` parameter.

3. Create a function called `get_model_metrics`, which takes parameters `reg_model` and `data`, and evaluates the model then returns a dictionary of metrics for the trained model.

Move the code under the *Validate Model on Validation Set* heading into the `get_model_metrics` function and modify it to return the `metrics` object.

The three functions should be as follows:

```
# Split the dataframe into test and train data
def split_data(df):
    X = df.drop('Y', axis=1).values
    y = df['Y'].values

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0)
    data = {"train": {"X": X_train, "y": y_train},
            "test": {"X": X_test, "y": y_test}}
    return data

# Train the model, return the model
def train_model(data, args):
    reg_model = Ridge(**args)
    reg_model.fit(data["train"]["X"], data["train"]["y"])
    return reg_model

# Evaluate the metrics for the model
def get_model_metrics(reg_model, data):
    preds = reg_model.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    metrics = {"mse": mse}
    return metrics
```

Still in `experimentation/Diabetes Ridge Regression Training.ipynb`, complete the following steps:

1. Create a new function called `main`, which takes no parameters and returns nothing.
2. Move the code under the "Load Data" heading into the `main` function.
3. Add invocations for the newly written functions into the `main` function:

```
# Split Data into Training and Validation Sets
data = split_data(df)
```

```
# Train Model on Training Set
args = {
    "alpha": 0.5
}
reg = train_model(data, args)
```

```
# Validate Model on Validation Set
metrics = get_model_metrics(reg, data)
```

4. Move the code under the "Save Model" heading into the `main` function.

The `main` function should look like the following code:

```
def main():
    # Load Data
    sample_data = load_diabetes()

    df = pd.DataFrame(
        data=sample_data.data,
        columns=sample_data.feature_names)
    df['Y'] = sample_data.target

    # Split Data into Training and Validation Sets
    data = split_data(df)

    # Train Model on Training Set
    args = {
        "alpha": 0.5
    }
    reg = train_model(data, args)

    # Validate Model on Validation Set
    metrics = get_model_metrics(reg, data)

    # Save Model
    model_name = "sklearn_regression_model.pkl"

    joblib.dump(value=reg, filename=model_name)
```

At this stage, there should be no code remaining in the notebook that isn't in a function, other than import statements in the first cell.

Add a statement that calls the `main` function.

```
main()
```

After refactoring, `experimentation/Diabetes Ridge Regression Training.ipynb` should look like the following code without the markdown:

```

from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import pandas as pd
import joblib

# Split the dataframe into test and train data
def split_data(df):
    X = df.drop('Y', axis=1).values
    y = df['Y'].values

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0)
    data = {"train": {"X": X_train, "y": y_train},
            "test": {"X": X_test, "y": y_test}}
    return data

# Train the model, return the model
def train_model(data, args):
    reg_model = Ridge(**args)
    reg_model.fit(data["train"]["X"], data["train"]["y"])
    return reg_model

# Evaluate the metrics for the model
def get_model_metrics(reg_model, data):
    preds = reg_model.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    metrics = {"mse": mse}
    return metrics

def main():
    # Load Data
    sample_data = load_diabetes()

    df = pd.DataFrame(
        data=sample_data.data,
        columns=sample_data.feature_names)
    df['Y'] = sample_data.target

    # Split Data into Training and Validation Sets
    data = split_data(df)

    # Train Model on Training Set
    args = {
        "alpha": 0.5
    }
    reg = train_model(data, args)

    # Validate Model on Validation Set
    metrics = get_model_metrics(reg, data)

    # Save Model
    model_name = "sklearn_regression_model.pkl"

    joblib.dump(value=reg, filename=model_name)

main()

```

Refactor Diabetes Ridge Regression Scoring notebook into functions

In `experimentation/Diabetes Ridge Regression Scoring.ipynb`, complete the following steps:

1. Create a new function called `init`, which takes no parameters and return nothing.
2. Copy the code under the "Load Model" heading into the `init` function.

The `init` function should look like the following code:

```
def init():
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)
```

Once the `init` function has been created, replace all the code under the heading "Load Model" with a single call to `init` as follows:

```
init()
```

In `experimentation/Diabetes Ridge Regression Scoring.ipynb`, complete the following steps:

1. Create a new function called `run`, which takes `raw_data` and `request_headers` as parameters and returns a dictionary of results as follows:

```
{"result": result.tolist()}
```

2. Copy the code under the "Prepare Data" and "Score Data" headings into the `run` function.

The `run` function should look like the following code (Remember to remove the statements that set the variables `raw_data` and `request_headers`, which will be used later when the `run` function is called):

```
def run(raw_data, request_headers):
    data = json.loads(raw_data)["data"]
    data = numpy.array(data)
    result = model.predict(data)

    return {"result": result.tolist()}
```

Once the `run` function has been created, replace all the code under the "Prepare Data" and "Score Data" headings with the following code:

```
raw_data = '{"data": [[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}'
request_header = {}
prediction = run(raw_data, request_header)
print("Test result: ", prediction)
```

The previous code sets variables `raw_data` and `request_header`, calls the `run` function with `raw_data` and `request_header`, and prints the predictions.

After refactoring, `experimentation/Diabetes Ridge Regression Scoring.ipynb` should look like the following code without the markdown:

```

import json
import numpy
from azureml.core.model import Model
import joblib

def init():
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)

def run(raw_data, request_headers):
    data = json.loads(raw_data)["data"]
    data = numpy.array(data)
    result = model.predict(data)

    return {"result": result.tolist()}

init()
test_row = '{"data":[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}'
request_header = {}
prediction = run(test_row, {})
print("Test result: ", prediction)

```

Combine related functions in Python files

Third, related functions need to be merged into Python files to better help code reuse. In this section, you'll be creating Python files for the following notebooks:

- The Diabetes Ridge Regression Training notebook([experimentation/Diabetes Ridge Regression Training.ipynb](#))
- The Diabetes Ridge Regression Scoring notebook([experimentation/Diabetes Ridge Regression Scoring.ipynb](#))

Create Python file for the Diabetes Ridge Regression Training notebook

Convert your notebook to an executable script by running the following statement in a command prompt, which uses the `nbconvert` package and the path of `experimentation/Diabetes Ridge Regression Training.ipynb`:

```
jupyter nbconvert "Diabetes Ridge Regression Training.ipynb" --to script --output train
```

Once the notebook has been converted to `train.py`, remove any unwanted comments. Replace the call to `main()` at the end of the file with a conditional invocation like the following code:

```

if __name__ == '__main__':
    main()

```

Your `train.py` file should look like the following code:

```

from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import pandas as pd
import joblib

# Split the dataframe into test and train data
def split_data(df):
    X = df.drop('Y', axis=1).values
    y = df['Y'].values

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=0)
    data = {"train": {"X": X_train, "y": y_train},
            "test": {"X": X_test, "y": y_test}}
    return data

# Train the model, return the model
def train_model(data, args):
    reg_model = Ridge(**args)
    reg_model.fit(data["train"]["X"], data["train"]["y"])
    return reg_model

# Evaluate the metrics for the model
def get_model_metrics(reg_model, data):
    preds = reg_model.predict(data["test"]["X"])
    mse = mean_squared_error(preds, data["test"]["y"])
    metrics = {"mse": mse}
    return metrics

def main():
    # Load Data
    sample_data = load_diabetes()

    df = pd.DataFrame(
        data=sample_data.data,
        columns=sample_data.feature_names)
    df['Y'] = sample_data.target

    # Split Data into Training and Validation Sets
    data = split_data(df)

    # Train Model on Training Set
    args = {
        "alpha": 0.5
    }
    reg = train_model(data, args)

    # Validate Model on Validation Set
    metrics = get_model_metrics(reg, data)

    # Save Model
    model_name = "sklearn_regression_model.pkl"

    joblib.dump(value=reg, filename=model_name)

if __name__ == '__main__':
    main()

```

`train.py` can now be invoked from a terminal by running `python train.py`. The functions from `train.py` can also be called from other files.

The `train_aml.py` file found in the `diabetes_regression/training` directory in the MLOpsPython repository calls the functions defined in `train.py` in the context of an Azure Machine Learning experiment job. The functions can also be called in unit tests, covered later in this guide.

Create Python file for the Diabetes Ridge Regression Scoring notebook

Convert your notebook to an executable script by running the following statement in a command prompt that which uses the `nbconvert` package and the path of `experimentation/Diabetes Ridge Regression Scoring.ipynb`:

```
jupyter nbconvert "Diabetes Ridge Regression Scoring.ipynb" --to script --output score
```

Once the notebook has been converted to `score.py`, remove any unwanted comments. Your `score.py` file should look like the following code:

```
import json
import numpy
from azureml.core.model import Model
import joblib

def init():
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)

def run(raw_data, request_headers):
    data = json.loads(raw_data)["data"]
    data = numpy.array(data)
    result = model.predict(data)

    return {"result": result.tolist()}

init()
test_row = '{"data":[[1,2,3,4,5,6,7,8,9,10],[10,9,8,7,6,5,4,3,2,1]]}'
request_header = {}
prediction = run(test_row, request_header)
print("Test result: ", prediction)
```

The `model` variable needs to be global so that it's visible throughout the script. Add the following statement at the beginning of the `init` function:

```
global model
```

After adding the previous statement, the `init` function should look like the following code:

```
def init():
    global model

    # load the model from file into a global object
    model_path = Model.get_model_path(
        model_name="sklearn_regression_model.pkl")
    model = joblib.load(model_path)
```

Create unit tests for each Python file

Fourth, create unit tests for your Python functions. Unit tests protect code against functional regressions and make it easier to maintain. In this section, you'll be creating unit tests for the functions in `train.py`.

`train.py` contains multiple functions, but we'll only create a single unit test for the `train_model` function using

the Pytest framework in this tutorial. Pytest isn't the only Python unit testing framework, but it's one of the most commonly used. For more information, visit [Pytest](#).

A unit test usually contains three main actions:

- Arrange object - creating and setting up necessary objects
- Act on an object
- Assert what is expected

The unit test will call `train_model` with some hard-coded data and arguments, and validate that `train_model` acted as expected by using the resulting trained model to make a prediction and comparing that prediction to an expected value.

```
import numpy as np
from code.training.train import train_model

def test_train_model():
    # Arrange
    X_train = np.array([1, 2, 3, 4, 5, 6]).reshape(-1, 1)
    y_train = np.array([10, 9, 8, 8, 6, 5])
    data = {"train": {"X": X_train, "y": y_train}}

    # Act
    reg_model = train_model(data, {"alpha": 1.2})

    # Assert
    preds = reg_model.predict([[1], [2]])
    np.testing.assert_almost_equal(preds, [9.939393939394, 9.030303030303])
```

Next steps

Now that you understand how to convert from an experiment to production code, see the following links for more information and next steps:

- [MLOpsPython](#): Build a CI/CD pipeline to train, evaluate and deploy your own model using Azure Pipelines and Azure Machine Learning
- [Monitor Azure ML experiment jobs and metrics](#)
- [Monitor and collect data from ML web service endpoints](#)

Troubleshooting machine learning pipelines

9/21/2022 • 12 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to troubleshoot when you get errors running a [machine learning pipeline](#) in the [Azure Machine Learning SDK](#) and [Azure Machine Learning designer](#).

Troubleshooting tips

The following table contains common problems during pipeline development, with potential solutions.

PROBLEM	POSSIBLE SOLUTION
Unable to pass data to <code>PipelineData</code> directory	Ensure you have created a directory in the script that corresponds to where your pipeline expects the step output data. In most cases, an input argument will define the output directory, and then you create the directory explicitly. Use <code>os.makedirs(args.output_dir, exist_ok=True)</code> to create the output directory. See the tutorial for a scoring script example that shows this design pattern.
Dependency bugs	If you see dependency errors in your remote pipeline that did not occur when locally testing, confirm your remote environment dependencies and versions match those in your test environment. (See Environment building, caching, and reuse)
Ambiguous errors with compute targets	Try deleting and re-creating compute targets. Re-creating compute targets is quick and can solve some transient issues.
Pipeline not reusing steps	Step reuse is enabled by default, but ensure you haven't disabled it in a pipeline step. If reuse is disabled, the <code>allow_reuse</code> parameter in the step will be set to <code>False</code> .
Pipeline is rerunning unnecessarily	To ensure that steps only rerun when their underlying data or scripts change, decouple your source-code directories for each step. If you use the same source directory for multiple steps, you may experience unnecessary reruns. Use the <code>source_directory</code> parameter on a pipeline step object to point to your isolated directory for that step, and ensure you aren't using the same <code>source_directory</code> path for multiple steps.
Step slowing down over training epochs or other looping behavior	Try switching any file writes, including logging, from <code>as_mount()</code> to <code>as_upload()</code> . The mount mode uses a remote virtualized filesystem and uploads the entire file each time it is appended to.

PROBLEM	POSSIBLE SOLUTION
Compute target takes a long time to start	Docker images for compute targets are loaded from Azure Container Registry (ACR). By default, Azure Machine Learning creates an ACR that uses the <i>basic</i> service tier. Changing the ACR for your workspace to standard or premium tier may reduce the time it takes to build and load images. For more information, see Azure Container Registry service tiers .

Authentication errors

If you perform a management operation on a compute target from a remote job, you will receive one of the following errors:

```
{"code": "Unauthorized", "statusCode": 401, "message": "Unauthorized", "details": [{"code": "InvalidOrExpiredToken", "message": "The request token was either invalid or expired. Please try again with a valid token."}]}
```

```
{"error": {"code": "AuthenticationFailed", "message": "Authentication failed."}}
```

For example, you will receive an error if you try to create or attach a compute target from an ML Pipeline that is submitted for remote execution.

Troubleshooting `ParallelRunStep`

The script for a `ParallelRunStep` *must contain* two functions:

- `init()` : Use this function for any costly or common preparation for later inference. For example, use it to load the model into a global object. This function will be called only once at beginning of process.
- `run(mini_batch)` : The function will run for each `mini_batch` instance.
 - `mini_batch` : `ParallelRunStep` will invoke `run` method and pass either a list or pandas `DataFrame` as an argument to the method. Each entry in `mini_batch` will be a file path if input is a `FileDataset` or a pandas `DataFrame` if input is a `TabularDataset`.
 - `response` : `run()` method should return a pandas `DataFrame` or an array. For `append_row` `output_action`, these returned elements are appended into the common output file. For `summary_only`, the contents of the elements are ignored. For all output actions, each returned output element indicates one successful run of input element in the input mini-batch. Make sure that enough data is included in `run` result to map input to run output result. Run output will be written in output file and not guaranteed to be in order, you should use some key in the output to map it to input.

```

%%writefile digit_identification.py
# Snippets from a sample script.
# Refer to the accompanying digit_identification.py
# (https://github.com/Azure/MachineLearningNotebooks/tree/master/how-to-use-azureml/machine-learning-
# pipelines/parallel-run)
# for the implementation script.

import os
import numpy as np
import tensorflow as tf
from PIL import Image
from azureml.core import Model

def init():
    global g_tf_sess

    # Pull down the model from the workspace
    model_path = Model.get_model_path("mnist")

    # Construct a graph to execute
    tf.reset_default_graph()
    saver = tf.train.import_meta_graph(os.path.join(model_path, 'mnist-tf.model.meta'))
    g_tf_sess = tf.Session()
    saver.restore(g_tf_sess, os.path.join(model_path, 'mnist-tf.model'))

def run(mini_batch):
    print(f'run method start: {__file__}, run({mini_batch})')
    resultList = []
    in_tensor = g_tf_sess.graph.get_tensor_by_name("network/X:0")
    output = g_tf_sess.graph.get_tensor_by_name("network/output/MatMul:0")

    for image in mini_batch:
        # Prepare each image
        data = Image.open(image)
        np_im = np.array(data).reshape((1, 784))
        # Perform inference
        inference_result = output.eval(feed_dict={in_tensor: np_im}, session=g_tf_sess)
        # Find the best probability, and add it to the result list
        best_result = np.argmax(inference_result)
        resultList.append("{}: {}".format(os.path.basename(image), best_result))

    return resultList

```

If you have another file or folder in the same directory as your inference script, you can reference it by finding the current working directory.

```

script_dir = os.path.realpath(os.path.join(__file__, '..'))
file_path = os.path.join(script_dir, "<file_name>")

```

Parameters for ParallelRunConfig

`ParallelRunConfig` is the major configuration for `ParallelRunStep` instance within the Azure Machine Learning pipeline. You use it to wrap your script and configure necessary parameters, including all of the following entries:

- `entry_script`: A user script as a local file path that will be run in parallel on multiple nodes. If `source_directory` is present, use a relative path. Otherwise, use any path that's accessible on the machine.
- `mini_batch_size`: The size of the mini-batch passed to a single `run()` call. (optional; the default value is `10` files for `FileDataset` and `1MB` for `TabularDataset`.)
 - For `FileDataset`, it's the number of files with a minimum value of `1`. You can combine multiple files

into one mini-batch.

- For `TabularDataset`, it's the size of data. Example values are `1024`, `1024KB`, `10MB`, and `1GB`. The recommended value is `1MB`. The mini-batch from `TabularDataset` will never cross file boundaries. For example, if you have .csv files with various sizes, the smallest file is 100 KB and the largest is 10 MB. If you set `mini_batch_size = 1MB`, then files with a size smaller than 1 MB will be treated as one mini-batch. Files with a size larger than 1 MB will be split into multiple mini-batches.
- `error_threshold` : The number of record failures for `TabularDataset` and file failures for `FileDataset` that should be ignored during processing. If the error count for the entire input goes above this value, the job will be aborted. The error threshold is for the entire input and not for individual mini-batch sent to the `run()` method. The range is `[-1, int.max]`. The `-1` part indicates ignoring all failures during processing.
- `output_action` : One of the following values indicates how the output will be organized:
 - `summary_only` : The user script will store the output. `ParallelRunStep` will use the output only for the error threshold calculation.
 - `append_row` : For all inputs, only one file will be created in the output folder to append all outputs separated by line.
- `append_row_file_name` : To customize the output file name for `append_row` `output_action` (optional; default value is `parallel_run_step.txt`).
- `source_directory` : Paths to folders that contain all files to execute on the compute target (optional).
- `compute_target` : Only `AmlCompute` is supported.
- `node_count` : The number of compute nodes to be used for running the user script.
- `process_count_per_node` : The number of processes per node. Best practice is to set to the number of GPU or CPU one node has (optional; default value is `1`).
- `environment` : The Python environment definition. You can configure it to use an existing Python environment or to set up a temporary environment. The definition is also responsible for setting the required application dependencies (optional).
- `logging_level` : Log verbosity. Values in increasing verbosity are: `WARNING`, `INFO`, and `DEBUG`. (optional; the default value is `INFO`)
- `run_invocation_timeout` : The `run()` method invocation timeout in seconds. (optional; default value is `60`)
- `run_max_try` : Maximum try count of `run()` for a mini-batch. A `run()` is failed if an exception is thrown, or nothing is returned when `run_invocation_timeout` is reached (optional; default value is `3`).

You can specify `mini_batch_size`, `node_count`, `process_count_per_node`, `logging_level`, `run_invocation_timeout`, and `run_max_try` as `PipelineParameter`, so that when you resubmit a pipeline run, you can fine-tune the parameter values. In this example, you use `PipelineParameter` for `mini_batch_size` and `Process_count_per_node` and you will change these values when resubmit a run later.

Parameters for creating the ParallelRunStep

Create the ParallelRunStep by using the script, environment configuration, and parameters. Specify the compute target that you already attached to your workspace as the target of execution for your inference script. Use

`ParallelRunStep` to create the batch inference pipeline step, which takes all the following parameters:

- `name` : The name of the step, with the following naming restrictions: unique, 3-32 characters, and regex `^[a-z]([-a-z0-9]*[a-z0-9])?$.`
- `parallel_run_config` : A `ParallelRunConfig` object, as defined earlier.
- `inputs` : One or more single-typed Azure Machine Learning datasets to be partitioned for parallel processing.
- `side_inputs` : One or more reference data or datasets used as side inputs without need to be partitioned.
- `output` : An `OutputFileDatasetConfig` object that corresponds to the output directory.
- `arguments` : A list of arguments passed to the user script. Use `unknown_args` to retrieve them in your entry

script (optional).

- `allow_reuse` : Whether the step should reuse previous results when run with the same settings/inputs. If this parameter is `False`, a new run will always be generated for this step during pipeline execution. (optional; the default value is `True`.)

```
from azureml.pipeline.steps import ParallelRunStep

parallelrun_step = ParallelRunStep(
    name="predict-digits-mnist",
    parallel_run_config=parallel_run_config,
    inputs=[input_mnist_ds_consumption],
    output=output_dir,
    allow_reuse=True
)
```

Debugging techniques

There are three major techniques for debugging pipelines:

- Debug individual pipeline steps on your local computer
- Use logging and Application Insights to isolate and diagnose the source of the problem
- Attach a remote debugger to a pipeline running in Azure

Debug scripts locally

One of the most common failures in a pipeline is that the domain script does not run as intended, or contains runtime errors in the remote compute context that are difficult to debug.

Pipelines themselves cannot be run locally, but running the scripts in isolation on your local machine allows you to debug faster because you don't have to wait for the compute and environment build process. Some development work is required to do this:

- If your data is in a cloud datastore, you will need to download data and make it available to your script. Using a small sample of your data is a good way to cut down on runtime and quickly get feedback on script behavior
- If you are attempting to simulate an intermediate pipeline step, you may need to manually build the object types that the particular script is expecting from the prior step
- You will also need to define your own environment, and replicate the dependencies defined in your remote compute environment

Once you have a script setup to run on your local environment, it is much easier to do debugging tasks like:

- Attaching a custom debug configuration
- Pausing execution and inspecting object-state
- Catching type or logical errors that won't be exposed until runtime

TIP

Once you can verify that your script is running as expected, a good next step is running the script in a single-step pipeline before attempting to run it in a pipeline with multiple steps.

Configure, write to, and review pipeline logs

Testing scripts locally is a great way to debug major code fragments and complex logic before you start building a pipeline, but at some point you will likely need to debug scripts during the actual pipeline run itself, especially

when diagnosing behavior that occurs during the interaction between pipeline steps. We recommend liberal use of `print()` statements in your step scripts so that you can see object state and expected values during remote execution, similar to how you would debug JavaScript code.

Logging options and behavior

The table below provides information for different debug options for pipelines. It isn't an exhaustive list, as other options exist besides just the Azure Machine Learning, Python, and OpenCensus ones shown here.

LIBRARY	TYPE	EXAMPLE	DESTINATION	RESOURCES
Azure Machine Learning SDK	Metric	<code>run.log(name, val)</code>	Azure Machine Learning Portal UI	How to track experiments azureml.core.Run class
Python printing/logging	Log	<code>print(val)</code> <code>logging.info(message)</code>	Driver logs, Azure Machine Learning designer	How to track experiments Python logging
OpenCensus Python	Log	<code>logger.addHandler(AzureLogHandler)</code> <code>logging.log(message)</code>	Application Insights - traces	Debug pipelines in Application Insights OpenCensus Azure Monitor Exporters Python logging cookbook

Logging options example

```
import logging

from azureml.core.run import Run
from opencensus.ext.azure.log_exporter import AzureLogHandler

run = Run.get_context()

# Azure ML Scalar value logging
run.log("scalar_value", 0.95)

# Python print statement
print("I am a python print statement, I will be sent to the driver logs.")

# Initialize Python logger
logger = logging.getLogger(__name__)
logger.setLevel(args.log_level)

# Plain Python logging statements
logger.debug("I am a plain debug statement, I will be sent to the driver logs.")
logger.info("I am a plain info statement, I will be sent to the driver logs.")

handler = AzureLogHandler(connection_string='<connection string>')
logger.addHandler(handler)

# Python logging with OpenCensus AzureLogHandler
logger.warning("I am an OpenCensus warning statement, find me in Application Insights!")
logger.error("I am an OpenCensus error statement with custom dimensions", {'step_id': run.id})
```

Azure Machine Learning designer

For pipelines created in the designer, you can find the `70_driver_log` file in either the authoring page, or in the

pipeline run detail page.

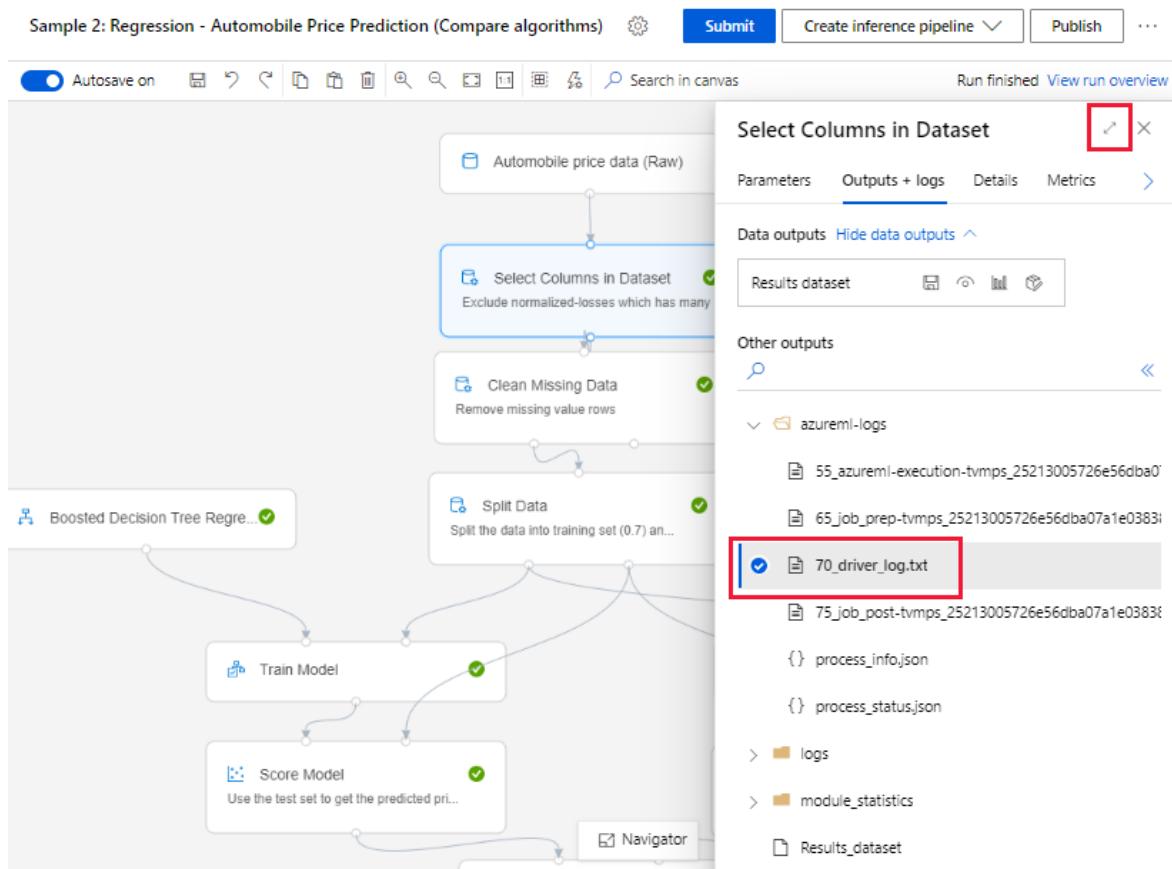
Enable logging for real-time endpoints

In order to troubleshoot and debug real-time endpoints in the designer, you must enable Application Insight logging using the SDK. Logging lets you troubleshoot and debug model deployment and usage issues. For more information, see [Logging for deployed models](#).

Get logs from the authoring page

When you submit a pipeline run and stay in the authoring page, you can find the log files generated for each component as each component finishes running.

1. Select a component that has finished running in the authoring canvas.
2. In the right pane of the component, go to the **Outputs + logs** tab.
3. Expand the right pane, and select the **70_driver_log.txt** to view the file in browser. You can also download logs locally.



Get logs from pipeline runs

You can also find the log files for specific runs in the pipeline run detail page, which can be found in either the **Pipelines** or **Experiments** section of the studio.

1. Select a pipeline run created in the designer.

The screenshot shows the Azure Machine Learning Pipelines interface. On the left is a sidebar with icons for creating new pipelines, cloning, deleting, and viewing pipeline details. The main area is titled 'Pipelines' and contains a table of 'Pipeline runs'. The columns are: Run, Created time, Duration, Status, Description, Experiment, Submitted by, and Tags. There are four rows in the table, each representing a pipeline run. The 'Tags' column for each row contains specific tags: 'azureml.Designer: true', 'azureml.pipelineComponent: pi...', 'azureml.Designer: true', and 'azureml.pipelineComponent: pi...'. A search bar at the top right allows filtering items.

Run	Created time	Duration	Status	Description	Experiment	Submitted by	Tags
Run 1	12/12/2019, 2:41:00 PM	10m 37s	Completed	Sample 2: Regression - Automo...	Sample2	Blanca Li	azureml.Designer: true, azure...
Run 1	12/12/2019, 12:57:12 PM	18m 55s	Failed	NYCTaxi_Tutorial_Pipelines	NYCTaxi_Tutorial_Pipelines	Blanca Li	azureml.pipelineComponent: pi...
Run 1	12/11/2019, 5:23:15 PM	13m 7s	Completed	Sample 1: Regression - Automo...	Sample1	Blanca Li	azureml.Designer: true, azure...
Run 1	12/11/2019, 5:00:21 PM	3m 47s	Completed	has updated train step	helloworld	Blanca Li	azureml.pipelineComponent: pi...

2. Select a component in the preview pane.
3. In the right pane of the component, go to the **Outputs + logs** tab.
4. Expand the right pane to view the `std_log.txt` file in browser, or select the file to download the logs locally.

IMPORTANT

To update a pipeline from the pipeline run details page, you must **clone** the pipeline run to a new pipeline draft. A pipeline run is a snapshot of the pipeline. It's similar to a log file, and cannot be altered.

Application Insights

For more information on using the OpenCensus Python library in this manner, see this guide: [Debug and troubleshoot machine learning pipelines in Application Insights](#)

Interactive debugging with Visual Studio Code

In some cases, you may need to interactively debug the Python code used in your ML pipeline. By using Visual Studio Code (VS Code) and debugpy, you can attach to the code as it runs in the training environment. For more information, visit the [interactive debugging in VS Code guide](#).

Next steps

- For a complete tutorial using `ParallelRunStep`, see [Tutorial: Build an Azure Machine Learning pipeline for batch scoring](#).
- For a complete example showing automated machine learning in ML pipelines, see [Use automated ML in an Azure Machine Learning pipeline in Python](#).
- See the SDK reference for help with the `azureml-pipelines-core` package and the `azureml-pipelines-steps` package.
- See the list of [designer exceptions and error codes](#).

Collect machine learning pipeline log files in Application Insights for alerts and debugging

9/21/2022 • 4 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

The [OpenCensus](#) Python library can be used to route logs to Application Insights from your scripts. Aggregating logs from pipeline runs in one place allows you to build queries and diagnose issues. Using Application Insights will allow you to track logs over time and compare pipeline logs across runs.

Having your logs in once place will provide a history of exceptions and error messages. Since Application Insights integrates with Azure Alerts, you can also create alerts based on Application Insights queries.

Prerequisites

- Follow the steps to create an [Azure Machine Learning workspace](#) and [create your first pipeline](#)
- [Configure your development environment](#) to install the Azure Machine Learning SDK.
- Install the [OpenCensus Azure Monitor Exporter](#) package locally:

```
pip install opencensus-ext-azure
```

- Create an [Application Insights instance](#) (this doc also contains information on getting the connection string for the resource)

Getting Started

This section is an introduction specific to using OpenCensus from an Azure Machine Learning pipeline. For a detailed tutorial, see [OpenCensus Azure Monitor Exporters](#)

Add a PythonScriptStep to your Azure ML Pipeline. Configure your [RunConfiguration](#) with the dependency on `opencensus-ext-azure`. Configure the `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable.

```

from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import RunConfiguration
from azureml.pipeline.core import Pipeline
from azureml.pipeline.steps import PythonScriptStep

# Connecting to the workspace and compute target not shown

# Add pip dependency on OpenCensus
dependencies = CondaDependencies()
dependencies.add_pip_package("opencensus-ext-azure>=1.0.1")
run_config = RunConfiguration(conda_dependencies=dependencies)

# Add environment variable with Application Insights Connection String
# Replace the value with your own connection string
run_config.environment.environment_variables = {
    "APPLICATIONINSIGHTS_CONNECTION_STRING": 'InstrumentationKey=00000000-0000-0000-0000-000000000000'
}

# Configure step with runconfig
sample_step = PythonScriptStep(
    script_name="sample_step.py",
    compute_target=compute_target,
    runconfig=run_config
)

# Submit new pipeline run
pipeline = Pipeline(workspace=ws, steps=[sample_step])
pipeline.submit(experiment_name="Logging_Experiment")

```

Create a file called `sample_step.py`. Import the AzureLogHandler class to route logs to Application Insights. You'll also need to import the Python Logging library.

```

from opencensus.ext.azure.log_exporter import AzureLogHandler
import logging

```

Next, add the AzureLogHandler to the Python logger.

```

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)
logger.addHandler(logging.StreamHandler())

# Assumes the environment variable APPLICATIONINSIGHTS_CONNECTION_STRING is already set
logger.addHandler(AzureLogHandler())
logger.warning("I will be sent to Application Insights")

```

Logging with Custom Dimensions

By default, logs forwarded to Application Insights won't have enough context to trace back to the run or experiment. To make the logs actionable for diagnosing issues, more fields are needed.

To add these fields, Custom Dimensions can be added to provide context to a log message. One example is when someone wants to view logs across multiple steps in the same pipeline run.

Custom Dimensions make up a dictionary of key-value (stored as string, string) pairs. The dictionary is then sent to Application Insights and displayed as a column in the query results. Its individual dimensions can be used as [query parameters](#).

Helpful Context to include

FIELD	REASONING/EXAMPLE
parent_run_id	Can query logs for ones with the same parent_run_id to see logs over time for all steps, instead of having to dive into each individual step
step_id	Can query logs for ones with the same step_id to see where an issue occurred with a narrow scope to just the individual step
step_name	Can query logs to see step performance over time. Also helps to find a step_id for recent runs without diving into the portal UI
experiment_name	Can query across logs to see experiment performance over time. Also helps find a parent_run_id or step_id for recent runs without diving into the portal UI
run_url	Can provide a link directly back to the run for investigation.

Other helpful fields

These fields may require extra code instrumentation, and aren't provided by the run context.

FIELD	REASONING/EXAMPLE
build_url/build_version	If using CI/CD to deploy, this field can correlate logs to the code version that provided the step and pipeline logic. This link can further help to diagnose issues, or identify models with specific traits (log/metric values)
run_type	Can differentiate between different model types, or training vs. scoring runs

Creating a Custom Dimensions dictionary

```
from azureml.core import Run

run = Run.get_context(allow_offline=False)

custom_dimensions = {
    "parent_run_id": run.parent.id,
    "step_id": run.id,
    "step_name": run.name,
    "experiment_name": run.experiment.name,
    "run_url": run.parent.get_portal_url(),
    "run_type": "training"
}

# Assumes AzureLogHandler was already registered above
logger.info("I will be sent to Application Insights with Custom Dimensions", extra={
    "custom_dimensions":custom_dimensions})
```

OpenCensus Python logging considerations

The OpenCensus AzureLogHandler is used to route Python logs to Application Insights. As a result, Python logging nuances should be considered. When a logger is created, it has a default log level and will show logs greater than or equal to that level. A good reference for using Python logging features is the [Logging Cookbook](#).

The `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable is needed for the OpenCensus library. We recommend setting this environment variable instead of passing it in as a pipeline parameter to avoid passing around plaintext connection strings.

Querying logs in Application Insights

The logs routed to Application Insights will show up under 'traces' or 'exceptions'. Be sure to adjust your time window to include your pipeline run.

The screenshot shows the Azure Application Insights Log Analytics interface. At the top, there are buttons for 'Run', 'Time range : Last 24 hours', 'Save', 'Copy link', 'New alert rule', 'Export', 'Pin to dashboard', and 'Prettify query'. Below this is a search bar with the word 'traces'. The main area displays a table of log results. The table has columns: timestamp [UTC], message, severityLevel, itemType, and customDimensions. A dropdown menu is open over the 'customDimensions' column, showing a dictionary entry: {"lineNumber": "67", "fileName": "..."}. Underneath this, the table lists several log entries with various properties like build_id, build_url, experiment_name, fileName, level, lineNumber, module, parent_run_id, process, run_type, step_id, and step_name. The table has a dark background with light-colored rows and columns.

The result in Application Insights will show the log message and level, file path, and code line number. It will also show any custom dimensions included. In this image, the `customDimensions` dictionary shows the key/value pairs from the previous [code sample](#).

Other helpful queries

Some of the queries below use 'customDimensions.Level'. These severity levels correspond to the level the Python log was originally sent with. For more query information, see [Azure Monitor Log Queries](#).

USE CASE	QUERY
Log results for specific custom dimension, for example 'parent_run_id'	<pre>traces where customDimensions.parent_run_id == '931024c2-3720-11ea-b247-c49deda841c1'</pre>
Log results for all training runs over the last seven days	<pre>traces where timestamp > ago(7d) and customDimensions.run_type == 'training'</pre>

USE CASE	QUERY
Log results with severityLevel Error from the last seven days	<pre>traces where timestamp > ago(7d) and customDimensions.Level == 'ERROR'</pre>
Count of log results with severityLevel Error over the last seven days	<pre>traces where timestamp > ago(7d) and customDimensions.Level == 'ERROR' summarize count()</pre>

Next Steps

Once you have logs in your Application Insights instance, they can be used to set [Azure Monitor alerts](#) based on query results.

You can also add results from queries to an [Azure Dashboard](#) for more insights.

Troubleshooting the ParallelRunStep

9/21/2022 • 16 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

In this article, you learn how to troubleshoot when you get errors using the [ParallelRunStep](#) class from the [Azure Machine Learning SDK](#).

For general tips on troubleshooting a pipeline, see [Troubleshooting machine learning pipelines](#).

Testing scripts locally

Your ParallelRunStep runs as a step in ML pipelines. You may want to [test your scripts locally](#) as a first step.

Entry script requirements

The entry script for a `ParallelRunStep` *must contain* a `run()` function and optionally contains an `init()` function:

- `init()` : Use this function for any costly or common preparation for later processing. For example, use it to load the model into a global object. This function will be called only once at beginning of process.

NOTE

If your `init` method creates an output directory, specify that `parents=True` and `exist_ok=True`. The `init` method is called from each worker process on every node on which the job is running.

- `run(mini_batch)` : The function will run for each `mini_batch` instance.
 - `mini_batch` : `ParallelRunStep` will invoke `run` method and pass either a list or pandas `DataFrame` as an argument to the method. Each entry in `mini_batch` will be a file path if input is a `FileDataset` or a pandas `DataFrame` if input is a `TabularDataset` .
 - `response` : `run()` method should return a pandas `DataFrame` or an array. For `append_row` `output_action`, these returned elements are appended into the common output file. For `summary_only`, the contents of the elements are ignored. For all output actions, each returned output element indicates one successful run of input element in the input mini-batch. Make sure that enough data is included in `run` result to map input to `run` output result. Run output will be written in output file and not guaranteed to be in order, you should use some key in the output to map it to input.

NOTE

One output element is expected for one input element.

```

%%writefile digit_identification.py
# Snippets from a sample script.
# Refer to the accompanying digit_identification.py
# (https://github.com/Azure/MachineLearningNotebooks/tree/master/how-to-use-azureml/machine-learning-
pipelines/parallel-run)
# for the implementation script.

import os
import numpy as np
import tensorflow as tf
from PIL import Image
from azureml.core import Model

def init():
    global g_tf_sess

    # Pull down the model from the workspace
    model_path = Model.get_model_path("mnist")

    # Construct a graph to execute
    tf.reset_default_graph()
    saver = tf.train.import_meta_graph(os.path.join(model_path, 'mnist-tf.model.meta'))
    g_tf_sess = tf.Session()
    saver.restore(g_tf_sess, os.path.join(model_path, 'mnist-tf.model'))

def run(mini_batch):
    print(f'run method start: {__file__}, run({mini_batch})')
    resultList = []
    in_tensor = g_tf_sess.graph.get_tensor_by_name("network/X:0")
    output = g_tf_sess.graph.get_tensor_by_name("network/output/MatMul:0")

    for image in mini_batch:
        # Prepare each image
        data = Image.open(image)
        np_im = np.array(data).reshape((1, 784))
        # Perform inference
        inference_result = output.eval(feed_dict={in_tensor: np_im}, session=g_tf_sess)
        # Find the best probability, and add it to the result list
        best_result = np.argmax(inference_result)
        resultList.append("{}: {}".format(os.path.basename(image), best_result))

    return resultList

```

If you have another file or folder in the same directory as your inference script, you can reference it by finding the current working directory. If you want to import your packages, you can also append your package folder to `sys.path`.

```

script_dir = os.path.realpath(os.path.join(__file__, '..'))
file_path = os.path.join(script_dir, "<file_name>")

packages_dir = os.path.join(file_path, '<your_package_folder>')
if packages_dir not in sys.path:
    sys.path.append(packages_dir)
from <your_package> import <your_class>

```

Parameters for ParallelRunConfig

`ParallelRunConfig` is the major configuration for `ParallelRunStep` instance within the Azure Machine Learning pipeline. You use it to wrap your script and configure necessary parameters, including all of the following entries:

- `entry_script` : A user script as a local file path that will be run in parallel on multiple nodes. If `source_directory` is present, use a relative path. Otherwise, use any path that's accessible on the machine.
- `mini_batch_size` : The size of the mini-batch passed to a single `run()` call. (optional; the default value is `10` files for `FileDataset` and `1MB` for `TabularDataset` .)
 - For `FileDataset`, it's the number of files with a minimum value of `1`. You can combine multiple files into one mini-batch.
 - For `TabularDataset`, it's the size of data. Example values are `1024`, `1024KB`, `10MB`, and `1GB`. The recommended value is `1MB`. The mini-batch from `TabularDataset` will never cross file boundaries. For example, if you have .csv files with various sizes, the smallest file is 100 KB and the largest is 10 MB. If you set `mini_batch_size = 1MB`, then files with a size smaller than 1 MB will be treated as one mini-batch. Files with a size larger than 1 MB will be split into multiple mini-batches.

NOTE

TabularDatasets backed by SQL cannot be partitioned. TabularDatasets from a single parquet file and single row group cannot be partitioned.

- `error_threshold` : The number of record failures for `TabularDataset` and file failures for `FileDataset` that should be ignored during processing. If the error count for the entire input goes above this value, the job will be aborted. The error threshold is for the entire input and not for individual mini-batch sent to the `run()` method. The range is `[-1, int.max]`. The `-1` part indicates ignoring all failures during processing.
- `output_action` : One of the following values indicates how the output will be organized:
 - `summary_only` : The user script will store the output. `ParallelRunStep` will use the output only for the error threshold calculation.
 - `append_row` : For all inputs, only one file will be created in the output folder to append all outputs separated by line.
- `append_row_file_name` : To customize the output file name for `append_row` `output_action` (optional; default value is `parallel_run_step.txt`).
- `source_directory` : Paths to folders that contain all files to execute on the compute target (optional).
- `compute_target` : Only `AmlCompute` is supported.
- `node_count` : The number of compute nodes to be used for running the user script.
- `process_count_per_node` : The number of worker processes per node to run the entry script in parallel. For a GPU machine, the default value is 1. For a CPU machine, the default value is the number of cores per node. A worker process will call `run()` repeatedly by passing the mini batch it gets. The total number of worker processes in your job is `process_count_per_node * node_count`, which decides the max number of `run()` to execute in parallel.
- `environment` : The Python environment definition. You can configure it to use an existing Python environment or to set up a temporary environment. The definition is also responsible for setting the required application dependencies (optional).
- `logging_level` : Log verbosity. Values in increasing verbosity are: `WARNING`, `INFO`, and `DEBUG`. (optional; the default value is `INFO`)
- `run_invocation_timeout` : The `run()` method invocation timeout in seconds. (optional; default value is `60`)

)

- `run_max_try` : Maximum try count of `run()` for a mini-batch. A `run()` is failed if an exception is thrown, or nothing is returned when `run_invocation_timeout` is reached (optional; default value is `3`).

You can specify `mini_batch_size`, `node_count`, `process_count_per_node`, `logging_level`, `run_invocation_timeout`, and `run_max_try` as `PipelineParameter`, so that when you resubmit a pipeline run, you can fine-tune the parameter values. In this example, you use `PipelineParameter` for `mini_batch_size` and `Process_count_per_node` and you will change these values when you resubmit another run.

CUDA devices visibility

For compute targets equipped with GPUs, the environment variable `CUDA_VISIBLE_DEVICES` will be set in worker processes. In AmlCompute, you can find the total number of GPU devices in the environment variable `AZ_BATCHAI_GPU_COUNT_FOUND`, which is set automatically. If you want each worker process to have a dedicated GPU, set `process_count_per_node` equal to the number of GPU devices on a machine. Each worker process will assign a unique index to `CUDA_VISIBLE_DEVICES`. If a worker process stops for any reason, the next started worker process will use the released GPU index.

If the total number of GPU devices is less than `process_count_per_node`, the worker processes will be assigned GPU index until all have been used.

Given the total GPU devices is 2 and `process_count_per_node = 4` as an example, process 0 and process 1 will have index 0 and 1. Process 2 and 3 won't have an environment variable. For a library using this environment variable for GPU assignment, process 2 and 3 won't have GPUs and won't try to acquire GPU devices. If process 0 stops, it will release GPU index 0. The next process, which is process 4, will have GPU index 0 assigned.

For more information, see [CUDA Pro Tip: Control GPU Visibility with CUDA_VISIBLE_DEVICES](#).

Parameters for creating the ParallelRunStep

Create the ParallelRunStep by using the script, environment configuration, and parameters. Specify the compute target that you already attached to your workspace as the target of execution for your inference script. Use `ParallelRunStep` to create the batch inference pipeline step, which takes all the following parameters:

- `name` : The name of the step, with the following naming restrictions: unique, 3-32 characters, and regex `^([a-z]([-a-z0-9]*[a-z0-9])?$.)`
- `parallel_run_config` : A `ParallelRunConfig` object, as defined earlier.
- `inputs` : One or more single-typed Azure Machine Learning datasets to be partitioned for parallel processing.
- `side_inputs` : One or more reference data or datasets used as side inputs without need to be partitioned.
- `output` : An `OutputFileDatasetConfig` object that represents the directory path at which the output data will be stored.
- `arguments` : A list of arguments passed to the user script. Use `unknown_args` to retrieve them in your entry script (optional).
- `allow_reuse` : Whether the step should reuse previous results when run with the same settings/inputs. If this parameter is `False`, a new run will always be generated for this step during pipeline execution. (optional; the default value is `True`.)

```

from azureml.pipeline.steps import ParallelRunStep

parallelrun_step = ParallelRunStep(
    name="predict-digits-mnist",
    parallel_run_config=parallel_run_config,
    inputs=[input_mnist_ds_consumption],
    output=output_dir,
    allow_reuse=True
)

```

Debugging scripts from remote context

The transition from debugging a scoring script locally to debugging a scoring script in an actual pipeline can be a difficult leap. For information on finding your logs in the portal, see [machine learning pipelines section on debugging scripts from a remote context](#). The information in that section also applies to a ParallelRunStep.

For example, the log file `70_driver_log.txt` contains information from the controller that launches the ParallelRunStep code.

Because of the distributed nature of ParallelRunStep jobs, there are logs from several different sources. However, two consolidated files are created that provide high-level information:

- `~/logs/job_progress_overview.txt` : This file provides a high-level info about the number of mini-batches (also known as tasks) created so far and number of mini-batches processed so far. At this end, it shows the result of the job. If the job failed, it will show the error message and where to start the troubleshooting.
- `~/logs/sys/master_role.txt` : This file provides the principal node (also known as the orchestrator) view of the running job. Includes task creation, progress monitoring, the run result.

Logs generated from entry script using EntryScript helper and print statements will be found in following files:

- `~/logs/user/entry_script_log/<node_id>/<process_name>.log.txt` : These files are the logs written from entry_script using EntryScript helper.
- `~/logs/user/stdout/<node_id>/<process_name>.stdout.txt` : These files are the logs from stdout (for example, print statement) of entry_script.
- `~/logs/user/stderr/<node_id>/<process_name>.stderr.txt` : These files are the logs from stderr of entry_script.

For a concise understanding of errors in your script there is:

- `~/logs/user/error.txt` : This file will try to summarize the errors in your script.

For more information on errors in your script, there is:

- `~/logs/user/error/` : Contains full stack traces of exceptions thrown while loading and running entry script.

When you need a full understanding of how each node executed the score script, look at the individual process logs for each node. The process logs can be found in the `sys/node` folder, grouped by worker nodes:

- `~/logs/sys/node/<node_id>/<process_name>.txt` : This file provides detailed info about each mini-batch as it's picked up or completed by a worker. For each mini-batch, this file includes:
 - The IP address and the PID of the worker process.
 - The total number of items, successfully processed items count, and failed item count.
 - The start time, duration, process time and run method time.

You can also view the results of periodical checks of the resource usage for each node. The log files and setup files are in this folder:

- `~/logs/perf` : Set `--resource_monitor_interval` to change the checking interval in seconds. The default interval is `600`, which is approximately 10 minutes. To stop the monitoring, set the value to `0`. Each `<node_id>` folder includes:
 - `os/` : Information about all running processes in the node. One check runs an operating system command and saves the result to a file. On Linux, the command is `ps`. On Windows, use `tasklist`.
 - `%Y%m%d%H` : The sub folder name is the time to hour.
 - `processes_%M` : The file ends with the minute of the checking time.
 - `node_disk_usage.csv` : Detailed disk usage of the node.
 - `node_resource_usage.csv` : Resource usage overview of the node.
 - `processes_resource_usage.csv` : Resource usage overview of each process.

How do I log from my user script from a remote context?

ParallelRunStep may run multiple processes on one node based on `process_count_per_node`. In order to organize logs from each process on node and combine print and log statement, we recommend using ParallelRunStep logger as shown below. You get a logger from EntryScript and make the logs show up in `logs/user` folder in the portal.

A sample entry script using the logger:

```
from azureml_user.parallel_run import EntryScript

def init():
    """Init once in a worker process."""
    entry_script = EntryScript()
    logger = entry_script.logger
    logger.info("This will show up in files under logs/user on the Azure portal.")

def run(mini_batch):
    """Call once for a mini batch. Accept and return the list back."""
    # This class is in singleton pattern and will return same instance as the one in init()
    entry_script = EntryScript()
    logger = entry_script.logger
    logger.info(f"__file__: {mini_batch}.")
    ...

    return mini_batch
```

Where does the message from Python `logging` sink to?

ParallelRunStep sets a handler on the root logger, which sinks the message to

`logs/user/stdout/<node_id>/processNNN.stdout.txt`.

`logging` defaults to `INFO` level. By default, levels below `INFO` won't show up, such as `DEBUG`.

How could I write to a file to show up in the portal?

Files in `logs` folder will be uploaded and show up in the portal. You can get the folder `logs/user/entry_script_log/<node_id>` like below and compose your file path to write:

```

from pathlib import Path
from azureml_user.parallel_run import EntryScript

def init():
    """Init once in a worker process."""
    entry_script = EntryScript()
    log_dir = entry_script.log_dir
    log_dir = Path(entry_script.log_dir) # logs/user/entry_script_log/<node_id>/
    log_dir.mkdir(parents=True, exist_ok=True) # Create the folder if not existing.

    proc_name = entry_script.agent_name # The process name in pattern "processNNN".
    fil_path = log_dir / f"{proc_name}_<file_name>" # Avoid conflicting among worker processes with
    proc_name.

```

How to handle log in new processes?

You can spawn new processes in your entry script with `subprocess` module, connect to their input/output/error pipes and obtain their return codes.

The recommended approach is to use the `run()` function with `capture_output=True`. Errors will show up in `logs/user/error/<node_id>/<process_name>.txt`.

If you want to use `Popen()`, you should redirect stdout/stderr to files, like:

```

from pathlib import Path
from subprocess import Popen

from azureml_user.parallel_run import EntryScript

def init():
    """Show how to redirect stdout/stderr to files in logs/user/entry_script_log/<node_id>/. """
    entry_script = EntryScript()
    proc_name = entry_script.agent_name # The process name in pattern "processNNN".
    log_dir = Path(entry_script.log_dir) # logs/user/entry_script_log/<node_id>/
    log_dir.mkdir(parents=True, exist_ok=True) # Create the folder if not existing.
    stdout_file = str(log_dir / f"{proc_name}_demo_stdout.txt")
    stderr_file = str(log_dir / f"{proc_name}_demo_stderr.txt")
    proc = Popen(
        ["..."],
        stdout=open(stdout_file, "w"),
        stderr=open(stderr_file, "w"),
        # ...
    )

```

NOTE

A worker process runs "system" code and the entry script code in the same process.

If no `stdout` or `stderr` specified, a subprocess created with `Popen()` in your entry script will inherit the setting of the worker process.

`stdout` will write to `logs/sys/node/<node_id>/processNNN.stdout.txt` and `stderr` to `logs/sys/node/<node_id>/processNNN.stderr.txt`.

How do I write a file to the output directory, and then view it in the portal?

You can get the output directory from the `EntryScript` class and write to it. To view the written files, in the step Run view in the Azure Machine Learning portal, select the **Outputs + logs** tab. Select the **Data outputs** link, and then complete the steps that are described in the dialog.

Use `EntryScript` in your entry script like in this example:

```
from pathlib import Path
from azureml_user.parallel_run import EntryScript

def run(mini_batch):
    output_dir = Path(entry_script.output_dir)
    (Path(output_dir) / res1).write...
    (Path(output_dir) / res2).write...
```

How can I pass a side input such as, a file or file(s) containing a lookup table, to all my workers?

User can pass reference data to script using `side_inputs` parameter of `ParallelRunStep`. All datasets provided as `side_inputs` will be mounted on each worker node. User can get the location of mount by passing argument.

Construct a [Dataset](#) containing the reference data, specify a local mount path and register it with your workspace. Pass it to the `side_inputs` parameter of your `ParallelRunStep`. Additionally, you can add its path in the `arguments` section to easily access its mounted path.

NOTE

Use FileDatasets only for `side_inputs`.

```
local_path = "/tmp/{}".format(str(uuid.uuid4()))
label_config = label_ds.as_named_input("labels_input").as_mount(local_path)
batch_score_step = ParallelRunStep(
    name=parallel_step_name,
    inputs=[input_images.as_named_input("input_images")],
    output=output_dir,
    arguments=["--labels_dir", label_config],
    side_inputs=[label_config],
    parallel_run_config=parallel_run_config,
)
```

After that you can access it in your inference script (for example, in your `init()` method) as follows:

```
parser = argparse.ArgumentParser()
parser.add_argument('--labels_dir', dest="labels_dir", required=True)
args, _ = parser.parse_known_args()

labels_path = args.labels_dir
```

How to use input datasets with service principal authentication?

User can pass input datasets with service principal authentication used in workspace. Using such dataset in `ParallelRunStep` requires that dataset to be registered for it to construct `ParallelRunStep` configuration.

```

service_principal = ServicePrincipalAuthentication(
    tenant_id="***",
    service_principal_id="***",
    service_principal_password="***")

ws = Workspace(
    subscription_id="***",
    resource_group="***",
    workspace_name="***",
    auth=service_principal
)

default_blob_store = ws.get_default_datastore() # or Datastore(ws, '***datastore-name***')
ds = Dataset.File.from_files(default_blob_store, '**path**')
registered_ds = ds.register(ws, '***dataset-name***', create_new_version=True)

```

How to Check Progress and Analyze it

This section is about how to check the progress of a ParallelRunStep job and check the cause of unexpected behavior.

How to check job progress?

Besides looking at the overall status of the StepRun, the count of scheduled/processed mini-batches and the progress of generating output can be viewed in `~/logs/job_progress_overview.<timestamp>.txt`. The file rotates on daily basis, you can check the one with the largest timestamp for the latest information.

What should I check if there is no progress for a while?

You can go into `~/logs/sys/error` to see if there's any exception. If there is none, it's likely that your entry script is taking a long time, you can print out progress information in your code to locate the time-consuming part, or add `--profiling_module`, `"cProfile"` to the `arguments` of `ParallelRunStep` to generate a profile file named as `<process_name>.profile` under `~/logs/sys/node/<node_id>` folder.

When will a job stop?

if not canceled, the job will stop with status:

- Completed. If all mini-batches have been processed and output has been generated for `append_row` mode.
- Failed. If `error_threshold` in `Parameters for ParallelRunConfig` is exceeded, or system error occurred during the job.

Where to find the root cause of failure?

You can follow the lead in `~/logs/job_result.txt` to find the cause and detailed error log.

Will node failure impact the job result?

Not if there are other available nodes in the designated compute cluster. The orchestrator will start a new node as replacement, and ParallelRunStep is resilient to such operation.

What happens if `init` function in entry script fails?

ParallelRunStep has mechanism to retry for a certain times to give chance for recovery from transient issues without delaying the job failure for too long, the mechanism is as follows:

- If after a node starts, `init` on all agents keeps failing, we will stop trying after `3 * process_count_per_node` failures.
- If after job starts, `init` on all agents of all nodes keeps failing, we will stop trying if job runs more than 2 minutes and there're `2 * node_count * process_count_per_node` failures.
- If all agents are stuck on `init` for more than `3 * run_invocation_timeout + 30` seconds, the job would fail

because of no progress for too long.

What will happen on OutOfMemory? How can I check the cause?

ParallelRunStep will set the current attempt to process the mini-batch to failure status and try to restart the failed process. You can check `~logs/perf/<node_id>` to find the memory-consuming process.

Why do I have a lot of processNNN files?

ParallelRunStep will start new worker processes in replace of the ones exited abnormally, and each process will generate a `processNNN` file as log. However, if the process failed because of exception during the `init` function of user script, and that the error repeated continuously for `3 * process_count_per_node` times, no new worker process will be started.

Next steps

- See these [Jupyter notebooks demonstrating Azure Machine Learning pipelines](#)
- See the SDK reference for help with the [azureml-pipeline-steps](#) package.
- View reference [documentation](#) for ParallelRunConfig class and [documentation](#) for ParallelRunStep class.
- Follow the [advanced tutorial](#) on using pipelines with ParallelRunStep. The tutorial shows how to pass another file as a side input.

CLI (v1) pipeline job YAML schema

9/21/2022 • 11 minutes to read • [Edit Online](#)

APPLIES TO:  Azure CLI ml extension v1

NOTE

The YAML syntax detailed in this document is based on the JSON schema for the v1 version of the ML CLI extension. This syntax is guaranteed only to work with the ML CLI v1 extension. Switch to the [v2 \(current version\)](#) for the syntax for ML CLI v2.

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning.

Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

Define your machine learning pipelines in [YAML](#). When using the machine learning extension for the [Azure CLI v1](#), many of the pipeline-related commands expect a YAML file that defines the pipeline.

The following table lists what is and is not currently supported when defining a pipeline in YAML for use with CLI v1:

STEP TYPE	SUPPORTED?
PythonScriptStep	Yes
ParallelRunStep	Yes
AdlaStep	Yes
AzureBatchStep	Yes
DatabricksStep	Yes
DataTransferStep	Yes
AutoMLStep	No
HyperDriveStep	No
ModuleStep	Yes
MPIStep	No
EstimatorStep	No

Pipeline definition

A pipeline definition uses the following keys, which correspond to the [Pipelines](#) class:

YAML KEY	DESCRIPTION
<code>name</code>	The description of the pipeline.
<code>parameters</code>	Parameter(s) to the pipeline.
<code>data_reference</code>	Defines how and where data should be made available in a run.
<code>default_compute</code>	Default compute target where all steps in the pipeline run.
<code>steps</code>	The steps used in the pipeline.

Parameters

The `parameters` section uses the following keys, which correspond to the [PipelineParameter](#) class:

YAML KEY	DESCRIPTION
<code>type</code>	The value type of the parameter. Valid types are <code>string</code> , <code>int</code> , <code>float</code> , <code>bool</code> , or <code>datapath</code> .
<code>default</code>	The default value.

Each parameter is named. For example, the following YAML snippet defines three parameters named

`NumIterationsParameter`, `DataPathParameter`, and `NodeCountParameter`:

```
pipeline:  
  name: SamplePipelineFromYaml  
  parameters:  
    NumIterationsParameter:  
      type: int  
      default: 40  
    DataPathParameter:  
      type: datapath  
      default:  
        datastore: workspaceblobstore  
        path_on_datastore: sample2.txt  
    NodeCountParameter:  
      type: int  
      default: 4
```

Data reference

The `data_references` section uses the following keys, which correspond to the [DataReference](#):

YAML KEY	DESCRIPTION
<code>datastore</code>	The datastore to reference.

YAML KEY	DESCRIPTION
<code>path_on_datastore</code>	The relative path in the backing storage for the data reference.

Each data reference is contained in a key. For example, the following YAML snippet defines a data reference stored in the key named `employee_data`:

```
pipeline:
  name: SamplePipelineFromYaml
  parameters:
    PipelineParam1:
      type: int
      default: 3
  data_references:
    employee_data:
      datastore: adftestadla
      path_on_datastore: "adla_sample/sample_input.csv"
```

Steps

Steps define a computational environment, along with the files to run on the environment. To define the type of a step, use the `type` key:

STEP TYPE	DESCRIPTION
<code>AdlaStep</code>	Runs a U-SQL script with Azure Data Lake Analytics. Corresponds to the AdlaStep class.
<code>AzureBatchStep</code>	Runs jobs using Azure Batch. Corresponds to the AzureBatchStep class.
<code>DatabricksStep</code>	Adds a Databricks notebook, Python script, or JAR. Corresponds to the DatabricksStep class.
<code>DataTransferStep</code>	Transfers data between storage options. Corresponds to the DataTransferStep class.
<code>PythonScriptStep</code>	Runs a Python script. Corresponds to the PythonScriptStep class.
<code>ParallelRunStep</code>	Runs a Python script to process large amounts of data asynchronously and in parallel. Corresponds to the ParallelRunStep class.

ADLA step

YAML KEY	DESCRIPTION
<code>script_name</code>	The name of the U-SQL script (relative to the <code>source_directory</code>).
<code>compute</code>	The Azure Data Lake compute target to use for this step.
<code>parameters</code>	Parameters to the pipeline.

YAML KEY	DESCRIPTION
<code>inputs</code>	Inputs can be InputPortBinding , DataReference , PortDataReference , PipelineData , Dataset , DatasetDefinition , or PipelineDataset .
<code>outputs</code>	Outputs can be either PipelineData or OutputPortBinding .
<code>source_directory</code>	Directory that contains the script, assemblies, etc.
<code>priority</code>	The priority value to use for the current job.
<code>params</code>	Dictionary of name-value pairs.
<code>degree_of_parallelism</code>	The degree of parallelism to use for this job.
<code>runtime_version</code>	The runtime version of the Data Lake Analytics engine.
<code>allow_reuse</code>	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains an ADLA Step definition:

```

pipeline:
  name: SamplePipelineFromYaml
  parameters:
    PipelineParam1:
      type: int
      default: 3
  data_references:
    employee_data:
      datastore: adftestadla
      path_on_datastore: "adla_sample/sample_input.csv"
  default_compute: adlacomp
  steps:
    Step1:
      runconfig: "D:\\\\Yaml\\\\default_runconfig.yml"
      parameters:
        NUM_ITERATIONS_2:
          source: PipelineParam1
        NUM_ITERATIONS_1: 7
      type: "AdlaStep"
      name: "MyAdlaStep"
      script_name: "sample_script.usql"
      source_directory: "D:\\\\scripts\\\\Adla"
      inputs:
        employee_data:
          source: employee_data
      outputs:
        OutputData:
          destination: Output4
          datastore: adftestadla
          bind_mode: mount

```

Azure Batch step

YAML KEY	DESCRIPTION
<code>compute</code>	The Azure Batch compute target to use for this step.
<code>inputs</code>	Inputs can be InputPortBinding , DataReference , PortDataReference , PipelineData , Dataset , DatasetDefinition , or PipelineDataset .
<code>outputs</code>	Outputs can be either PipelineData or OutputPortBinding .
<code>source_directory</code>	Directory that contains the module binaries, executable, assemblies, etc.
<code>executable</code>	Name of the command/executable that will be ran as part of this job.
<code>create_pool</code>	Boolean flag to indicate whether to create the pool before running the job.
<code>delete_batch_job_after_finish</code>	Boolean flag to indicate whether to delete the job from the Batch account after it's finished.
<code>delete_batch_pool_after_finish</code>	Boolean flag to indicate whether to delete the pool after the job finishes.
<code>is_positive_exit_code_failure</code>	Boolean flag to indicate if the job fails if the task exits with a positive code.
<code>vm_image_urn</code>	If <code>create_pool</code> is <code>True</code> , and VM uses <code>VirtualMachineConfiguration</code> .
<code>pool_id</code>	The ID of the pool where the job will run.
<code>allow_reuse</code>	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains an Azure Batch step definition:

```

pipeline:
  name: SamplePipelineFromYaml
  parameters:
    PipelineParam1:
      type: int
      default: 3
  data_references:
    input:
      datastore: workspaceblobstore
      path_on_datastore: "input.txt"
  default_compute: testbatch
  steps:
    Step1:
      runconfig: "D:\\\\Yaml\\\\default_runconfig.yml"
      parameters:
        NUM_ITERATIONS_2:
          source: PipelineParam1
        NUM_ITERATIONS_1: 7
      type: "AzureBatchStep"
      name: "MyAzureBatchStep"
      pool_id: "MyPoolName"
      create_pool: true
      executable: "azurebatch.cmd"
      source_directory: "D:\\\\scripts\\\\AureBatch"
      allow_reuse: false
      inputs:
        input:
          source: input
      outputs:
        output:
          destination: output
          datastore: workspaceblobstore

```

Databricks step

YAML KEY	DESCRIPTION
<code>compute</code>	The Azure Databricks compute target to use for this step.
<code>inputs</code>	Inputs can be InputPortBinding , DataReference , PortDataReference , PipelineData , Dataset , DatasetDefinition , or PipelineDataset .
<code>outputs</code>	Outputs can be either PipelineData or OutputPortBinding .
<code>run_name</code>	The name in Databricks for this run.
<code>source_directory</code>	Directory that contains the script and other files.
<code>num_workers</code>	The static number of workers for the Databricks run cluster.
<code>runconfig</code>	The path to a <code>.runconfig</code> file. This file is a YAML representation of the RunConfiguration class. For more information on the structure of this file, see runconfigsschema.json .
<code>allow_reuse</code>	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a Databricks step:

```

pipeline:
  name: SamplePipelineFromYaml
  parameters:
    PipelineParam1:
      type: int
      default: 3
  data_references:
    adls_test_data:
      datastore: adftestadla
      path_on_datastore: "testdata"
    blob_test_data:
      datastore: workspaceblobstore
      path_on_datastore: "dbtest"
  default_compute: mydatabricks
  steps:
    Step1:
      runconfig: "D:\\Yaml\\default_runconfig.yml"
      parameters:
        NUM_ITERATIONS_2:
          source: PipelineParam1
        NUM_ITERATIONS_1: 7
      type: "DatabricksStep"
      name: "MyDatabrickStep"
      run_name: "DatabricksRun"
      python_script_name: "train-db-local.py"
      source_directory: "D:\\scripts\\Databricks"
      num_workers: 1
      allow_reuse: true
      inputs:
        blob_test_data:
          source: blob_test_data
      outputs:
        OutputData:
          destination: Output4
          datastore: workspaceblobstore
          bind_mode: mount

```

Data transfer step

YAML KEY	DESCRIPTION
<code>compute</code>	The Azure Data Factory compute target to use for this step.
<code>source_data_reference</code>	Input connection that serves as the source of data transfer operations. Supported values are InputPortBinding , DataReference , PortDataReference , PipelineData , Dataset , DatasetDefinition , or PipelineDataset .
<code>destination_data_reference</code>	Input connection that serves as the destination of data transfer operations. Supported values are PipelineData and OutputPortBinding .
<code>allow_reuse</code>	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a data transfer step:

```

pipeline:
  name: SamplePipelineFromYaml
  parameters:
    PipelineParam1:
      type: int
      default: 3
  data_references:
    adls_test_data:
      datastore: adftestadla
      path_on_datastore: "testdata"
    blob_test_data:
      datastore: workspaceblobstore
      path_on_datastore: "testdata"
  default_compute: adftest
  steps:
    Step1:
      runconfig: "D:\\\\Yaml\\\\default_runconfig.yml"
      parameters:
        NUM_ITERATIONS_2:
          source: PipelineParam1
        NUM_ITERATIONS_1: 7
        type: "DataTransferStep"
        name: "MyDataTransferStep"
        adla_compute_name: adftest
        source_data_reference:
          adls_test_data:
            source: adls_test_data
        destination_data_reference:
          blob_test_data:
            source: blob_test_data

```

Python script step

YAML KEY	DESCRIPTION
<code>inputs</code>	Inputs can be InputPortBinding , DataReference , PortDataReference , PipelineData , Dataset , DatasetDefinition , or PipelineDataset .
<code>outputs</code>	Outputs can be either PipelineData or OutputPortBinding .
<code>script_name</code>	The name of the Python script (relative to source_directory).
<code>source_directory</code>	Directory that contains the script, Conda environment, etc.
<code>runconfig</code>	The path to a <code>.runconfig</code> file. This file is a YAML representation of the RunConfiguration class. For more information on the structure of this file, see runconfig.json .
<code>allow_reuse</code>	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a Python script step:

```

pipeline:
  name: SamplePipelineFromYaml
  parameters:
    PipelineParam1:
      type: int
      default: 3
  data_references:
    DataReference1:
      datastore: workspaceblobstore
      path_on_datastore: testfolder/sample.txt
  default_compute: cpu-cluster
  steps:
    Step1:
      runconfig: "D:\\\\Yaml\\\\default_runconfig.yml"
      parameters:
        NUM_ITERATIONS_2:
          source: PipelineParam1
        NUM_ITERATIONS_1: 7
      type: "PythonScriptStep"
      name: "MyPythonScriptStep"
      script_name: "train.py"
      allow_reuse: True
      source_directory: "D:\\\\scripts\\\\PythonScript"
      inputs:
        InputData:
          source: DataReference1
      outputs:
        OutputData:
          destination: Output4
          datastore: workspaceblobstore
          bind_mode: mount

```

Parallel run step

YAML KEY	DESCRIPTION
<code>inputs</code>	Inputs can be Dataset , DatasetDefinition , or PipelineDataset .
<code>outputs</code>	Outputs can be either PipelineData or OutputPortBinding .
<code>script_name</code>	The name of the Python script (relative to <code>source_directory</code>).
<code>source_directory</code>	Directory that contains the script, Conda environment, etc.
<code>parallel_run_config</code>	The path to a <code>parallel_run_config.yml</code> file. This file is a YAML representation of the ParallelRunConfig class.
<code>allow_reuse</code>	Determines whether the step should reuse previous results when run again with the same settings.

The following example contains a Parallel run step:

```

pipeline:
  description: SamplePipelineFromYaml
  default_compute: cpu-cluster
  data_references:
    MyMinistInput:
      dataset_name: mnist_sample_data
  parameters:
    PipelineParamTimeout:
      type: int
      default: 600
  steps:
    Step1:
      parallel_run_config: "yaml/parallel_run_config.yml"
      type: "ParallelRunStep"
      name: "parallel-run-step-1"
      allow_reuse: True
      arguments:
        - "--progress_update_timeout"
        - parameter:timeout_parameter
        - "--side_input"
        - side_input:SideInputData
      parameters:
        timeout_parameter:
          source: PipelineParamTimeout
    inputs:
      InputData:
        source: MyMinistInput
      side_inputs:
        SideInputData:
          source: Output4
          bind_mode: mount
    outputs:
      OutputDataStep2:
        destination: Output5
        datastore: workspaceblobstore
        bind_mode: mount

```

Pipeline with multiple steps

YAML KEY	DESCRIPTION
steps	<p>Sequence of one or more PipelineStep definitions. Note that the <code>destination</code> keys of one step's <code>outputs</code> become the <code>source</code> keys to the <code>inputs</code> of the next step.</p>

```

pipeline:
  name: SamplePipelineFromYAML
  description: Sample multistep YAML pipeline
  data_references:
    TitanicDS:
      dataset_name: 'titanic_ds'
      bind_mode: download
  default_compute: cpu-cluster
  steps:
    Dataprep:
      type: "PythonScriptStep"
      name: "DataPrep Step"
      compute: cpu-cluster
      runconfig: ".\\default_runconfig.yml"
      script_name: "prep.py"
      arguments:
        - '--train_path'
        - output:train_path
        - '--test_path'
        - output:test_path
      allow_reuse: True
      inputs:
        titanic_ds:
          source: TitanicDS
          bind_mode: download
      outputs:
        train_path:
          destination: train_csv
          datastore: workspaceblobstore
        test_path:
          destination: test_csv
    Training:
      type: "PythonScriptStep"
      name: "Training Step"
      compute: cpu-cluster
      runconfig: ".\\default_runconfig.yml"
      script_name: "train.py"
      arguments:
        - "--train_path"
        - input:train_path
        - "--test_path"
        - input:test_path
      inputs:
        train_path:
          source: train_csv
          bind_mode: download
        test_path:
          source: test_csv
          bind_mode: download

```

Schedules

When defining the schedule for a pipeline, it can be either datastore-triggered or recurring based on a time interval. The following are the keys used to define a schedule:

YAML KEY	DESCRIPTION
description	A description of the schedule.
recurrence	Contains recurrence settings, if the schedule is recurring.

YAML KEY	DESCRIPTION
<code>pipeline_parameters</code>	Any parameters that are required by the pipeline.
<code>wait_for_provisioning</code>	Whether to wait for provisioning of the schedule to complete.
<code>wait_timeout</code>	The number of seconds to wait before timing out.
<code>datastore_name</code>	The datastore to monitor for modified/added blobs.
<code>polling_interval</code>	How long, in minutes, between polling for modified/added blobs. Default value: 5 minutes. Only supported for datastore schedules.
<code>data_path_parameter_name</code>	The name of the data path pipeline parameter to set with the changed blob path. Only supported for datastore schedules.
<code>continue_on_step_failure</code>	Whether to continue execution of other steps in the submitted PipelineRun if a step fails. If provided, will override the <code>continue_on_step_failure</code> setting of the pipeline.
<code>path_on_datastore</code>	Optional. The path on the datastore to monitor for modified/added blobs. The path is under the container for the datastore, so the actual path the schedule monitors is <code>container/<code>path_on_datastore</code></code> . If none, the datastore container is monitored. Additions/modifications made in a subfolder of the <code>path_on_datastore</code> are not monitored. Only supported for datastore schedules.

The following example contains the definition for a datastore-triggered schedule:

```

Schedule:
  description: "Test create with datastore"
  recurrence: ~
  pipeline_parameters: {}
  wait_for_provisioning: True
  wait_timeout: 3600
  datastore_name: "workspaceblobstore"
  polling_interval: 5
  data_path_parameter_name: "input_data"
  continue_on_step_failure: None
  path_on_datastore: "file/path"

```

When defining a **recurring schedule**, use the following keys under `recurrence` :

YAML KEY	DESCRIPTION
<code>frequency</code>	How often the schedule recurs. Valid values are <code>"Minute"</code> , <code>"Hour"</code> , <code>"Day"</code> , <code>"Week"</code> , or <code>"Month"</code> .
<code>interval</code>	How often the schedule fires. The integer value is the number of time units to wait until the schedule fires again.

YAML KEY	DESCRIPTION
<code>start_time</code>	The start time for the schedule. The string format of the value is <code>YYYY-MM-DDThh:mm:ss</code> . If no start time is provided, the first workload is run instantly and future workloads are run based on the schedule. If the start time is in the past, the first workload is run at the next calculated run time.
<code>time_zone</code>	The time zone for the start time. If no time zone is provided, UTC is used.
<code>hours</code>	If <code>frequency</code> is "Day" or "Week", you can specify one or more integers from 0 to 23, separated by commas, as the hours of the day when the pipeline should run. Only <code>time_of_day</code> or <code>hours</code> and <code>minutes</code> can be used.
<code>minutes</code>	If <code>frequency</code> is "Day" or "Week", you can specify one or more integers from 0 to 59, separated by commas, as the minutes of the hour when the pipeline should run. Only <code>time_of_day</code> or <code>hours</code> and <code>minutes</code> can be used.
<code>time_of_day</code>	If <code>frequency</code> is "Day" or "Week", you can specify a time of day for the schedule to run. The string format of the value is <code>hh:mm</code> . Only <code>time_of_day</code> or <code>hours</code> and <code>minutes</code> can be used.
<code>week_days</code>	If <code>frequency</code> is "Week", you can specify one or more days, separated by commas, when the schedule should run. Valid values are "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", and "Sunday".

The following example contains the definition for a recurring schedule:

```

Schedule:
  description: "Test create with recurrence"
  recurrence:
    frequency: Week # Can be "Minute", "Hour", "Day", "Week", or "Month".
    interval: 1 # how often fires
    start_time: 2019-06-07T10:50:00
    time_zone: UTC
    hours:
      - 1
    minutes:
      - 0
    time_of_day: null
    week_days:
      - Friday
  pipeline_parameters:
    'a': 1
  wait_for_provisioning: True
  wait_timeout: 3600
  datastore_name: ~
  polling_interval: ~
  data_path_parameter_name: ~
  continue_on_step_failure: None
  path_on_datastore: ~

```

Next steps

Learn how to [use the CLI extension for Azure Machine Learning](#).

Hyperparameters for computer vision tasks in automated machine learning (v1)

9/21/2022 • 7 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

Learn which hyperparameters are available specifically for computer vision tasks in automated ML experiments.

With support for computer vision tasks, you can control the model algorithm and sweep hyperparameters. These model algorithms and hyperparameters are passed in as the parameter space for the sweep. While many of the hyperparameters exposed are model-agnostic, there are instances where hyperparameters are model-specific or task-specific.

Model-specific hyperparameters

This table summarizes hyperparameters specific to the `yolov5` algorithm.

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>validation_metric_type</code>	Metric computation method to use for validation metrics. Must be <code>none</code> , <code>coco</code> , <code>voc</code> , or <code>coco_voc</code> .	<code>voc</code>
<code>validation_iou_threshold</code>	IOU threshold for box matching when computing validation metrics. Must be a float in the range [0.1, 1].	0.5
<code>img_size</code>	Image size for train and validation. Must be a positive integer. <i>Note: training run may get into CUDA OOM if the size is too big.</i>	640
<code>model_size</code>	Model size. Must be <code>small</code> , <code>medium</code> , <code>large</code> , or <code>xlarge</code> . <i>Note: training run may get into CUDA OOM if the model size is too big.</i>	<code>medium</code>
<code>multi_scale</code>	Enable multi-scale image by varying image size by +/- 50% Must be 0 or 1. <i>Note: training run may get into CUDA OOM if no sufficient GPU memory.</i>	0

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>box_score_thresh</code>	During inference, only return proposals with a score greater than <code>box_score_thresh</code> . The score is the multiplication of the objectness score and classification probability. Must be a float in the range [0, 1].	0.1
<code>nms_iou_thresh</code>	IOU threshold used during inference in non-maximum suppression post processing. Must be a float in the range [0, 1].	0.5

This table summarizes hyperparameters specific to the `maskrcnn_*` for instance segmentation during inference.

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>mask_pixel_score_threshold</code>	Score cutoff for considering a pixel as part of the mask of an object.	0.5
<code>max_number_of_polygon_points</code>	Maximum number of (x, y) coordinate pairs in polygon after converting from a mask.	100
<code>export_as_image</code>	Export masks as images.	False
<code>image_type</code>	Type of image to export mask as (options are jpg, png, bmp).	JPG

Model agnostic hyperparameters

The following table describes the hyperparameters that are model agnostic.

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>number_of_epochs</code>	Number of training epochs. Must be a positive integer.	15 (except <code>yolov5</code> : 30)
<code>training_batch_size</code>	Training batch size. Must be a positive integer.	Multi-class/multi-label: 78 (except <code>vit-variants</code> : <code>vits16r224</code> : 128 <code>vitb16r224</code> : 48 <code>vitl16r224</code> : 10) Object detection: 2 (except <code>yolov5</code> : 16) Instance segmentation: 2 <i>Note: The defaults are largest batch size that can be used on 12 GiB GPU memory.</i>

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>validation_batch_size</code>	Validation batch size. Must be a positive integer.	Multi-class/multi-label: 78 (except <i>vit-variants</i> : <code>vits16r224</code> : 128 <code>vitb16r224</code> : 48 <code>vitl16r224</code> : 10) Object detection: 1 (except <code>yolov5</code> : 16) Instance segmentation: 1 <i>Note: The defaults are largest batch size that can be used on 12 GiB GPU memory.</i>
<code>grad_accumulation_step</code>	Gradient accumulation means running a configured number of <code>grad_accumulation_step</code> without updating the model weights while accumulating the gradients of those steps, and then using the accumulated gradients to compute the weight updates. Must be a positive integer.	1
<code>early_stopping</code>	Enable early stopping logic during training. Must be 0 or 1.	1
<code>early_stopping_patience</code>	Minimum number of epochs or validation evaluations with no primary metric improvement before the run is stopped. Must be a positive integer.	5
<code>early_stopping_delay</code>	Minimum number of epochs or validation evaluations to wait before primary metric improvement is tracked for early stopping. Must be a positive integer.	5
<code>learning_rate</code>	Initial learning rate. Must be a float in the range [0, 1].	Multi-class: 0.01 (except <i>vit-variants</i> : <code>vits16r224</code> : 0.0125 <code>vitb16r224</code> : 0.0125 <code>vitl16r224</code> : 0.001) Multi-label: 0.035 (except <i>vit-variants</i> : <code>vits16r224</code> : 0.025 <code>vitb16r224</code> : 0.025 <code>vitl16r224</code> : 0.002) Object detection: 0.005 (except <code>yolov5</code> : 0.01) Instance segmentation: 0.005

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>lr_scheduler</code>	Type of learning rate scheduler. Must be <code>warmup_cosine</code> or <code>step</code> .	<code>warmup_cosine</code>
<code>step_lr_gamma</code>	Value of gamma when learning rate scheduler is <code>step</code> . Must be a float in the range [0, 1].	0.5
<code>step_lr_step_size</code>	Value of step size when learning rate scheduler is <code>step</code> . Must be a positive integer.	5
<code>warmup_cosine_lr_cycles</code>	Value of cosine cycle when learning rate scheduler is <code>warmup_cosine</code> . Must be a float in the range [0, 1].	0.45
<code>warmup_cosine_lr_warmup_epochs</code>	Value of warmup epochs when learning rate scheduler is <code>warmup_cosine</code> . Must be a positive integer.	2
<code>optimizer</code>	Type of optimizer. Must be either <code>sgd</code> , <code>adam</code> , <code>adamw</code> .	<code>sgd</code>
<code>momentum</code>	Value of momentum when optimizer is <code>sgd</code> . Must be a float in the range [0, 1].	0.9
<code>weight_decay</code>	Value of weight decay when optimizer is <code>sgd</code> , <code>adam</code> , or <code>adamw</code> . Must be a float in the range [0, 1].	1e-4
<code>nesterov</code>	Enable <code>nesterov</code> when optimizer is <code>sgd</code> . Must be 0 or 1.	1
<code>beta1</code>	Value of <code>beta1</code> when optimizer is <code>adam</code> or <code>adamw</code> . Must be a float in the range [0, 1].	0.9
<code>beta2</code>	Value of <code>beta2</code> when optimizer is <code>adam</code> or <code>adamw</code> . Must be a float in the range [0, 1].	0.999
<code>amsgrad</code>	Enable <code>amsgrad</code> when optimizer is <code>adam</code> or <code>adamw</code> . Must be 0 or 1.	0
<code>evaluation_frequency</code>	Frequency to evaluate validation dataset to get metric scores. Must be a positive integer.	1
<code>checkpoint_frequency</code>	Frequency to store model checkpoints. Must be a positive integer.	Checkpoint at epoch with best primary metric on validation.

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>checkpoint_run_id</code>	The run ID of the experiment that has a pretrained checkpoint for incremental training.	no default
<code>checkpoint_dataset_id</code>	FileDataset ID containing pretrained checkpoint(s) for incremental training. Make sure to pass <code>checkpoint_filename</code> along with <code>checkpoint_dataset_id</code> .	no default
<code>checkpoint_filename</code>	The pretrained checkpoint filename in FileDataset for incremental training. Make sure to pass <code>checkpoint_dataset_id</code> along with <code>checkpoint_filename</code> .	no default
<code>layers_to_freeze</code>	How many layers to freeze for your model. For instance, passing 2 as value for <code>seresnext</code> means freezing layer0 and layer1 referring to the below supported model layer info. Must be a positive integer. <pre>'resnet': [('conv1.', 'bn1.'), 'layer1.', 'layer2.', 'layer3.', 'layer4.'], 'mobilenetv2': ['features.0.', 'features.1.', 'features.2.', 'features.3.', 'features.4.', 'features.5.', 'features.6.', 'features.7.', 'features.8.', 'features.9.', 'features.10.', 'features.11.', 'features.12.', 'features.13.', 'features.14.', 'features.15.', 'features.16.', 'features.17.', 'features.18.'], 'seresnext': ['layer0.', 'layer1.', 'layer2.', 'layer3.', 'layer4.'], 'vit': ['patch_embed', 'blocks.0.', 'blocks.1.', 'blocks.2.', 'blocks.3.', 'blocks.4.', 'blocks.5.', 'blocks.6.', 'blocks.7.', 'blocks.8.', 'blocks.9.', 'blocks.10.', 'blocks.11.'], 'yolov5_backbone': ['model.0.', 'model.1.', 'model.2.', 'model.3.', 'model.4.', 'model.5.', 'model.6.', 'model.7.', 'model.8.', 'model.9.'], 'resnet_backbone': ['backbone.body.conv1.', 'backbone.body.layer1.', 'backbone.body.layer2.', 'backbone.body.layer3.', 'backbone.body.layer4.']}</pre>	no default

Image classification (multi-class and multi-label) specific hyperparameters

The following table summarizes hyperparameters for image classification (multi-class and multi-label) tasks.

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>weighted_loss</code>	0 for no weighted loss. 1 for weighted loss with sqrt. (<code>class_weights</code>) 2 for weighted loss with <code>class_weights</code> . Must be 0 or 1 or 2.	0
<code>valid_resize_size</code>	<ul style="list-style-type: none"> Image size to which to resize before cropping for validation dataset. Must be a positive integer. <p><i>Notes:</i></p> <ul style="list-style-type: none"> <code>seresnext</code> doesn't take an arbitrary size. Training run may get into CUDA OOM if the size is too big 	256
<code>valid_crop_size</code>	<ul style="list-style-type: none"> Image crop size that's input to your neural network for validation dataset. Must be a positive integer. <p><i>Notes:</i></p> <ul style="list-style-type: none"> <code>seresnext</code> doesn't take an arbitrary size. ViT-variants should have the same <code>valid_crop_size</code> and <code>train_crop_size</code>. Training run may get into CUDA OOM if the size is too big 	224
<code>train_crop_size</code>	<ul style="list-style-type: none"> Image crop size that's input to your neural network for train dataset. Must be a positive integer. <p><i>Notes:</i></p> <ul style="list-style-type: none"> <code>seresnext</code> doesn't take an arbitrary size. ViT-variants should have the same <code>valid_crop_size</code> and <code>train_crop_size</code>. Training run may get into CUDA OOM if the size is too big 	224

Object detection and instance segmentation task specific hyperparameters

The following hyperparameters are for object detection and instance segmentation tasks.

WARNING

These parameters are not supported with the `yolov5` algorithm. See the [model specific hyperparameters](#) section for `yolov5` supported hyperparameters.

PARAMETER NAME	DESCRIPTION	DEFAULT
<code>validation_metric_type</code>	Metric computation method to use for validation metrics. Must be <code>none</code> , <code>coco</code> , <code>voc</code> , or <code>coco_voc</code> .	<code>voc</code>
<code>validation_iou_threshold</code>	IOU threshold for box matching when computing validation metrics. Must be a float in the range [0.1, 1].	0.5
<code>min_size</code>	Minimum size of the image to be rescaled before feeding it to the backbone. Must be a positive integer. <i>Note: training run may get into CUDA OOM if the size is too big.</i>	600
<code>max_size</code>	Maximum size of the image to be rescaled before feeding it to the backbone. Must be a positive integer. <i>Note: training run may get into CUDA OOM if the size is too big.</i>	1333
<code>box_score_thresh</code>	During inference, only return proposals with a classification score greater than <code>box_score_thresh</code> . Must be a float in the range [0, 1].	0.3
<code>nms_iou_thresh</code>	IOU (intersection over union) threshold used in non-maximum suppression (NMS) for the prediction head. Used during inference. Must be a float in the range [0, 1].	0.5
<code>box_detections_per_img</code>	Maximum number of detections per image, for all classes. Must be a positive integer.	100
<code>tile_grid_size</code>	The grid size to use for tiling each image. <i>Note: tile_grid_size must not be None to enable small object detection logic</i> A tuple of two integers passed as a string. Example: --tile_grid_size "(3, 2)"	No Default
<code>tile_overlap_ratio</code>	Overlap ratio between adjacent tiles in each dimension. Must be float in the range of [0, 1)	0.25
<code>tile_predictions_nms_thresh</code>	The IOU threshold to use to perform NMS while merging predictions from tiles and image. Used in validation/inference. Must be float in the range of [0, 1]	0.25

Next steps

- Learn how to [Set up AutoML to train computer vision models with Python \(preview\)](#).
- [Tutorial: Train an object detection model \(preview\) with AutoML and Python.](#)

Data schemas to train computer vision models with automated machine learning (v1)

9/21/2022 • 10 minutes to read • [Edit Online](#)

APPLIES TO:  Python SDK azureml v1

IMPORTANT

The Azure CLI commands in this article require the `azure-cli-ml`, or v1, extension for Azure Machine Learning. Support for the v1 extension will end on September 30, 2025. You will be able to install and use the v1 extension until that date.

We recommend that you transition to the `ml`, or v2, extension before September 30, 2025. For more information on the v2 extension, see [Azure ML CLI extension and Python SDK v2](#).

IMPORTANT

This feature is currently in public preview. This preview version is provided without a service-level agreement. Certain features might not be supported or might have constrained capabilities. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Learn how to format your JSONL files for data consumption in automated ML experiments for computer vision tasks during training and inference.

Data schema for training

Azure Machine Learning AutoML for Images requires input image data to be prepared in **JSONL** (JSON Lines) format. This section describes input data formats or schema for image classification multi-class, image classification multi-label, object detection, and instance segmentation. We'll also provide a sample of final training or validation JSON Lines file.

Image classification (binary/multi-class)

Input data format/schema in each JSON Line:

```
{  
    "image_url": "AmlDatastore://data_directory/..../Image_name.image_format",  
    "image_details": {  
        "format": "image_format",  
        "width": "image_width",  
        "height": "image_height"  
    },  
    "label": "class_name",  
}
```

KEY	DESCRIPTION	EXAMPLE
<code>image_url</code>	Image location in AzureML datastore <code>Required, String</code>	"AmlDatastore://data_directory/Image_01.jpg"

KEY	DESCRIPTION	EXAMPLE
image_details	Image details Optional, Dictionary	{"image_details":{"format": "jpg", "width": "400px", "height": "258px"}}
format	Image type (all the available Image formats in Pillow library are supported) Optional, String from {"jpg", "jpeg", "png", "jpe", "jfif", "bmp", "tif", "tiff"}	"jpg" or "jpeg" or "png" or "jpe" or "jfif" or "bmp" or "tif" or "tiff"
width	Width of the image Optional, String or Positive Integer	"400px" or 400
height	Height of the image Optional, String or Positive Integer	"200px" or 200
label	Class/label of the image Required, String	"cat"

Example of a JSONL file for multi-class image classification:

```
{"image_url": "AmlDatastore://image_data/Image_01.jpg", "image_details": {"format": "jpg", "width": "400px", "height": "258px"}, "label": "can"}
{"image_url": "AmlDatastore://image_data/Image_02.jpg", "image_details": {"format": "jpg", "width": "397px", "height": "296px"}, "label": "milk_bottle"}
.
.
.
{"image_url": "AmlDatastore://image_data/Image_n.jpg", "image_details": {"format": "jpg", "width": "1024px", "height": "768px"}, "label": "water_bottle"}
```



Image classification multi-label

The following is an example of input data format/schema in each JSON Line for image classification.

```
{
  "image_url": "AmlDatastore://data_directory/../{Image_name}.image_format",
  "image_details": {
    "format": "image_format",
    "width": "image_width",
    "height": "image_height"
  },
  "label": [
    "class_name_1",
    "class_name_2",
    "class_name_3",
    "...",
    "class_name_n"
  ]
}
```

KEY	DESCRIPTION	EXAMPLE
<code>image_url</code>	Image location in AzureML datastore Required, String	<code>"AmlDatastore://data_directory/Image_01.jpg"</code>
<code>image_details</code>	Image details Optional, Dictionary	<code>"image_details": {"format": "jpg", "width": "400px", "height": "258px"}</code>
<code>format</code>	Image type (all the Image formats available in <code>Pillow</code> library are supported) Optional, String from {"jpg", "jpeg", "png", "jpe", "jfif", "bmp", "tif", "tiff"}	<code>"jpg" or "jpeg" or "png" or "jpe" or "jfif" or "bmp" or "tif" or "tiff"</code>
<code>width</code>	Width of the image Optional, String or Positive Integer	<code>"400px" or 400</code>
<code>height</code>	Height of the image Optional, String or Positive Integer	<code>"200px" or 200</code>
<code>label</code>	List of classes/labels in the image Required, List of Strings	<code>["cat", "dog"]</code>

Example of a JSONL file for Image Classification Multi-label:

```
{"image_url": "AmlDatastore://image_data/Image_01.jpg", "image_details": {"format": "jpg", "width": "400px", "height": "258px"}, "label": ["can"]}
{"image_url": "AmlDatastore://image_data/Image_02.jpg", "image_details": {"format": "jpg", "width": "397px", "height": "296px"}, "label": ["can", "milk_bottle"]}
.
.
.

{"image_url": "AmlDatastore://image_data/Image_n.jpg", "image_details": {"format": "jpg", "width": "1024px", "height": "768px"}, "label": ["carton", "milk_bottle", "water_bottle"]}
```



Object detection

The following is an example JSONL file for object detection.

```
{
  "image_url": "AmlDatastore://data_directory/.../Image_name.image_format",
  "image_details": {
    "format": "image_format",
    "width": "image_width",
    "height": "image_height"
  },
  "label": [
    {
      "label": "class_name_1",
      "topX": "xmin/width",
      "topY": "ymin/height",
      "bottomX": "xmax/width",
      "bottomY": "ymax/height",
      "isCrowd": "isCrowd"
    },
    {
      "label": "class_name_2",
      "topX": "xmin/width",
      "topY": "ymin/height",
      "bottomX": "xmax/width",
      "bottomY": "ymax/height",
      "isCrowd": "isCrowd"
    },
    ...
  ]
}
```

Here,

- `xmin` = x coordinate of top-left corner of bounding box
- `ymin` = y coordinate of top-left corner of bounding box
- `xmax` = x coordinate of bottom-right corner of bounding box
- `ymax` = y coordinate of bottom-right corner of bounding box

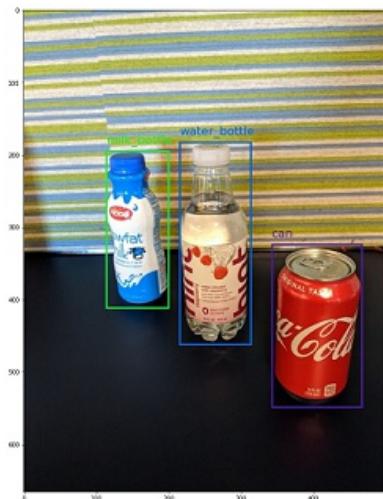
KEY	DESCRIPTION	EXAMPLE
<code>image_url</code>	Image location in AzureML datastore Required, String	"AmlDatastore://data_directory/Image_01.jpg"

KEY	DESCRIPTION	EXAMPLE
<code>image_details</code>	<p>Image details Optional, Dictionary</p>	<code>"image_details": {"format": "jpg", "width": "400px", "height": "258px"}</code>
<code>format</code>	<p>Image type (all the Image formats available in <code>Pillow</code> library are supported. But for YOLO only image formats allowed by <code>opencv</code> are supported) Optional, String from {"jpg", "jpeg", "png", "jpe", "jfif", "bmp", "tif", "tiff"}</p>	<code>"jpg" or "jpeg" or "png" or "jpe" or "jfif" or "bmp" or "tif" or "tiff"</code>
<code>width</code>	<p>Width of the image Optional, String or Positive Integer</p>	<code>"499px" or 499</code>
<code>height</code>	<p>Height of the image Optional, String or Positive Integer</p>	<code>"665px" or 665</code>
<code>label</code> (outer key)	<p>List of bounding boxes, where each box is a dictionary of <code>label</code>, <code>topX</code>, <code>topY</code>, <code>bottomX</code>, <code>bottomY</code>, <code>isCrowd</code> their top-left and bottom-right coordinates Required, List of dictionaries</p>	<code>[{"label": "cat", "topX": 0.260, "topY": 0.406, "bottomX": 0.735, "bottomY": 0.701, "isCrowd": 0}]</code>
<code>label</code> (inner key)	<p>Class/label of the object in the bounding box Required, String</p>	<code>"cat"</code>
<code>topX</code>	<p>Ratio of x coordinate of top-left corner of the bounding box and width of the image Required, Float in the range [0,1]</p>	<code>0.260</code>
<code>topY</code>	<p>Ratio of y coordinate of top-left corner of the bounding box and height of the image Required, Float in the range [0,1]</p>	<code>0.406</code>
<code>bottomX</code>	<p>Ratio of x coordinate of bottom-right corner of the bounding box and width of the image Required, Float in the range [0,1]</p>	<code>0.735</code>
<code>bottomY</code>	<p>Ratio of y coordinate of bottom-right corner of the bounding box and height of the image Required, Float in the range [0,1]</p>	<code>0.701</code>

KEY	DESCRIPTION	EXAMPLE
isCrowd	Indicates whether the bounding box is around the crowd of objects. If this special flag is set, we skip this particular bounding box when calculating the metric. Optional, Bool	0

Example of a JSONL file for object detection:

```
{"image_url": "AmlDatastore://image_data/Image_01.jpg", "image_details": {"format": "jpg", "width": "499px", "height": "666px"}, "label": [{"label": "can", "topX": 0.260, "topY": 0.406, "bottomX": 0.735, "bottomY": 0.701, "isCrowd": 0}]}
{"image_url": "AmlDatastore://image_data/Image_02.jpg", "image_details": {"format": "jpg", "width": "499px", "height": "666px"}, "label": [{"label": "carton", "topX": 0.172, "topY": 0.153, "bottomX": 0.432, "bottomY": 0.659, "isCrowd": 0}, {"label": "milk_bottle", "topX": 0.300, "topY": 0.566, "bottomX": 0.891, "bottomY": 0.735, "isCrowd": 0}]}
.
.
.
{"image_url": "AmlDatastore://image_data/Image_n.jpg", "image_details": {"format": "jpg", "width": "499px", "height": "666px"}, "label": [{"label": "carton", "topX": 0.0180, "topY": 0.297, "bottomX": 0.380, "bottomY": 0.836, "isCrowd": 0}, {"label": "milk_bottle", "topX": 0.454, "topY": 0.348, "bottomX": 0.613, "bottomY": 0.683, "isCrowd": 0}, {"label": "water_bottle", "topX": 0.667, "topY": 0.279, "bottomX": 0.841, "bottomY": 0.615, "isCrowd": 0}]}
```



Instance segmentation

For instance segmentation, automated ML only support polygon as input and output, no masks.

The following is an example JSONL file for instance segmentation.

```
{
    "image_url": "AmlDatastore://data_directory/..../Image_name.image_format",
    "image_details": {
        "format": "image_format",
        "width": "image_width",
        "height": "image_height"
    },
    "label": [
        {
            "label": "class_name",
            "isCrowd": "isCrowd",
            "polygon": [[{"x1": "x1", "y1": "y1", "x2": "x2", "y2": "y2", "x3": "x3", "y3": "y3", "...": "...", "xn": "xn", "yn": "yn"}]]
        }
    ]
}
```

KEY	DESCRIPTION	EXAMPLE
<code>image_url</code>	Image location in AzureML datastore Required, String	"AmlDatastore://data_directory/Image_01.jpg"
<code>image_details</code>	Image details Optional, Dictionary	"image_details": {"format": "jpg", "width": "400px", "height": "258px"}
<code>format</code>	Image type Optional, String from {"jpg", "jpeg", "png", "jpe", "jfif", "bmp", "tif", "tiff" }	"jpg" or "jpeg" or "png" or "jpe" or "jfif" or "bmp" or "tif" or "tiff"
<code>width</code>	Width of the image Optional, String or Positive Integer	"499px" or 499
<code>height</code>	Height of the image Optional, String or Positive Integer	"665px" or 665
<code>label</code> (outer key)	List of masks, where each mask is a dictionary of label, isCrowd, polygon coordinates Required, List of dictionaries	[{"label": "can", "isCrowd": 0, "polygon": [[0.577, 0.689, 0.562, 0.681, 0.559, 0.686]]}]
<code>label</code> (inner key)	Class/label of the object in the mask Required, String	"cat"
<code>isCrowd</code>	Indicates whether the mask is around the crowd of objects Optional, Bool	0
<code>polygon</code>	Polygon coordinates for the object Required, List of list for multiple segments of the same instance. Float values in the range [0,1]	[[0.577, 0.689, 0.567, 0.689, 0.559, 0.686]]]

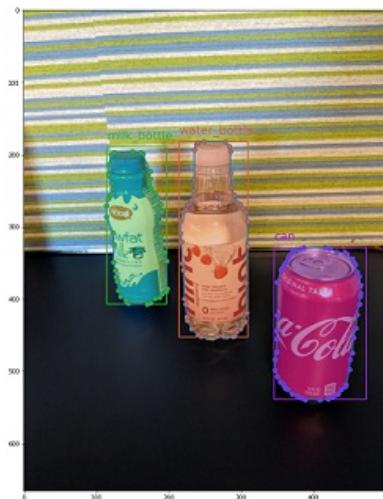
Example of a JSONL file for Instance Segmentation:

```

{"image_url": "AmlDatastore://image_data/Image_01.jpg", "image_details": {"format": "jpg", "width": "499px", "height": "666px"}, "label": [{"label": "can", "isCrowd": 0, "polygon": [[0.577, 0.689, 0.567, 0.689, 0.559, 0.686, 0.380, 0.593, 0.304, 0.555, 0.294, 0.545, 0.290, 0.534, 0.274, 0.512, 0.2705, 0.496, 0.270, 0.478, 0.284, 0.453, 0.308, 0.432, 0.326, 0.423, 0.356, 0.415, 0.418, 0.417, 0.635, 0.493, 0.683, 0.507, 0.701, 0.518, 0.709, 0.528, 0.713, 0.545, 0.719, 0.554, 0.719, 0.579, 0.713, 0.597, 0.697, 0.621, 0.695, 0.629, 0.631, 0.678, 0.619, 0.683, 0.595, 0.683, 0.577, 0.689]]}]}
{"image_url": "AmlDatastore://image_data/Image_02.jpg", "image_details": {"format": "jpg", "width": "499px", "height": "666px"}, "label": [{"label": "carton", "isCrowd": 0, "polygon": [[0.240, 0.65, 0.234, 0.654, 0.230, 0.647, 0.210, 0.512, 0.202, 0.403, 0.182, 0.267, 0.184, 0.243, 0.180, 0.166, 0.186, 0.159, 0.198, 0.156, 0.396, 0.162, 0.408, 0.169, 0.406, 0.217, 0.414, 0.249, 0.422, 0.262, 0.422, 0.569, 0.342, 0.569, 0.334, 0.572, 0.320, 0.585, 0.308, 0.624, 0.306, 0.648, 0.240, 0.657]]}, {"label": "milk_bottle", "isCrowd": 0, "polygon": [[0.675, 0.732, 0.635, 0.731, 0.621, 0.725, 0.573, 0.717, 0.516, 0.717, 0.505, 0.720, 0.462, 0.722, 0.438, 0.719, 0.396, 0.719, 0.358, 0.714, 0.334, 0.714, 0.322, 0.711, 0.312, 0.701, 0.306, 0.687, 0.304, 0.663, 0.308, 0.630, 0.320, 0.596, 0.32, 0.588, 0.326, 0.579]]}]}
.
.
.

{"image_url": "AmlDatastore://image_data/Image_n.jpg", "image_details": {"format": "jpg", "width": "499px", "height": "666px"}, "label": [{"label": "water_bottle", "isCrowd": 0, "polygon": [[0.334, 0.626, 0.304, 0.621, 0.254, 0.603, 0.164, 0.605, 0.158, 0.602, 0.146, 0.602, 0.142, 0.608, 0.094, 0.612, 0.084, 0.599, 0.080, 0.585, 0.080, 0.539, 0.082, 0.536, 0.092, 0.533, 0.126, 0.530, 0.132, 0.533, 0.144, 0.533, 0.162, 0.525, 0.172, 0.525, 0.186, 0.521, 0.196, 0.521]]}, {"label": "milk_bottle", "isCrowd": 0, "polygon": [[0.392, 0.773, 0.380, 0.732, 0.379, 0.767, 0.367, 0.755, 0.362, 0.735, 0.362, 0.714, 0.352, 0.644, 0.352, 0.611, 0.362, 0.597, 0.40, 0.593, 0.444, 0.494, 0.588, 0.515, 0.585, 0.621, 0.588, 0.671, 0.582, 0.713, 0.572, 0.753]]}]}

```



Data format for inference

In this section, we document the input data format required to make predictions when using a deployed model. Any aforementioned image format is accepted with content type `application/octet-stream`.

Input format

The following is the input format needed to generate predictions on any task using task-specific model endpoint. After we [deploy the model](#), we can use the following code snippet to get predictions for all tasks.

```

# input image for inference
sample_image = './test_image.jpg'
# load image data
data = open(sample_image, 'rb').read()
# set the content type
headers = {'Content-Type': 'application/octet-stream'}
# if authentication is enabled, set the authorization header
headers['Authorization'] = f'Bearer {key}'
# make the request and display the response
response = requests.post(scoring_uri, data, headers=headers)

```

Output format

Predictions made on model endpoints follow different structure depending on the task type. This section explores the output data formats for multi-class, multi-label image classification, object detection, and instance segmentation tasks.

Image classification

Endpoint for image classification returns all the labels in the dataset and their probability scores for the input image in the following format.

```
{  
  "filename": "/tmp/tmpppjr4et28",  
  "probs": [  
    2.098e-06,  
    4.783e-08,  
    0.999,  
    8.637e-06  
  ],  
  "labels": [  
    "can",  
    "carton",  
    "milk_bottle",  
    "water_bottle"  
  ]  
}
```

Image classification multi-label

For image classification multi-label, model endpoint returns labels and their probabilities.

```
{  
  "filename": "/tmp/tmpsdzx1mlm",  
  "probs": [  
    0.997,  
    0.960,  
    0.982,  
    0.025  
  ],  
  "labels": [  
    "can",  
    "carton",  
    "milk_bottle",  
    "water_bottle"  
  ]  
}
```

Object detection

Object detection model returns multiple boxes with their scaled top-left and bottom-right coordinates along with box label and confidence score.

```
{  
    "filename": "/tmp/tmpdkg2wky",  
    "boxes": [  
        {  
            "box": {  
                "topX": 0.224,  
                "topY": 0.285,  
                "bottomX": 0.399,  
                "bottomY": 0.620  
            },  
            "label": "milk_bottle",  
            "score": 0.937  
        },  
        {  
            "box": {  
                "topX": 0.664,  
                "topY": 0.484,  
                "bottomX": 0.959,  
                "bottomY": 0.812  
            },  
            "label": "can",  
            "score": 0.891  
        },  
        {  
            "box": {  
                "topX": 0.423,  
                "topY": 0.253,  
                "bottomX": 0.632,  
                "bottomY": 0.725  
            },  
            "label": "water_bottle",  
            "score": 0.876  
        }  
    ]  
}
```

Instance segmentation

In instance segmentation, output consists of multiple boxes with their scaled top-left and bottom-right coordinates, labels, confidence scores, and polygons (not masks). Here, the polygon values are in the same format that we discussed in the schema section.

```
{
  "filename": "/tmp/tmpi8604s0h",
  "boxes": [
    {
      "box": {
        "topX": 0.679,
        "topY": 0.491,
        "bottomX": 0.926,
        "bottomY": 0.810
      },
      "label": "can",
      "score": 0.992,
      "polygon": [
        [
          0.82, 0.811, 0.771, 0.810, 0.758, 0.805, 0.741, 0.797, 0.735, 0.791, 0.718, 0.785, 0.715, 0.778, 0.706, 0.775, 0.696, 0.758, 0.695, 0.717, 0.698, 0.567, 0.705, 0.552, 0.706, 0.540, 0.725, 0.520, 0.735, 0.505, 0.745, 0.502, 0.755, 0.493
        ]
      ]
    },
    {
      "box": {
        "topX": 0.220,
        "topY": 0.298,
        "bottomX": 0.397,
        "bottomY": 0.601
      },
      "label": "milk_bottle",
      "score": 0.989,
      "polygon": [
        [
          0.365, 0.602, 0.273, 0.602, 0.26, 0.595, 0.263, 0.588, 0.251, 0.546, 0.248, 0.501, 0.25, 0.485, 0.246, 0.478, 0.245, 0.463, 0.233, 0.442, 0.231, 0.43, 0.226, 0.423, 0.226, 0.408, 0.234, 0.385, 0.241, 0.371, 0.238, 0.345, 0.234, 0.335, 0.233, 0.325, 0.24, 0.305, 0.586, 0.38, 0.592, 0.375, 0.598, 0.365
        ]
      ]
    },
    {
      "box": {
        "topX": 0.433,
        "topY": 0.280,
        "bottomX": 0.621,
        "bottomY": 0.679
      },
      "label": "water_bottle",
      "score": 0.988,
      "polygon": [
        [
          0.576, 0.680, 0.501, 0.680, 0.475, 0.675, 0.460, 0.625, 0.445, 0.630, 0.443, 0.572, 0.440, 0.560, 0.435, 0.515, 0.431, 0.501, 0.431, 0.433, 0.433, 0.426, 0.445, 0.417, 0.456, 0.407, 0.465, 0.381, 0.468, 0.327, 0.471, 0.318
        ]
      ]
    }
  ]
}
```

NOTE

The images used in this article are from the Fridge Objects dataset, copyright © Microsoft Corporation and available at [computervision-recipes/01_training_introduction.ipynb](#) under the [MIT License](#).

Next steps

- Learn how to [Prepare data for training computer vision models with automated ML](#).

- [Set up computer vision tasks in AutoML](#)
- [Tutorial: Train an object detection model \(preview\) with AutoML and Python.](#)