

Vertex AI, Training and Serving a Custom Model

Overview

In this experiment, you will use [Vertex AI](#) to train and serve a TensorFlow model using code in a custom container.

While we're using TensorFlow for the model code here, you could easily replace it with another framework.

What you learn

You'll learn how to:

Build and containerize model training code in Vertex Workbench

Submit a custom model training job to Vertex AI

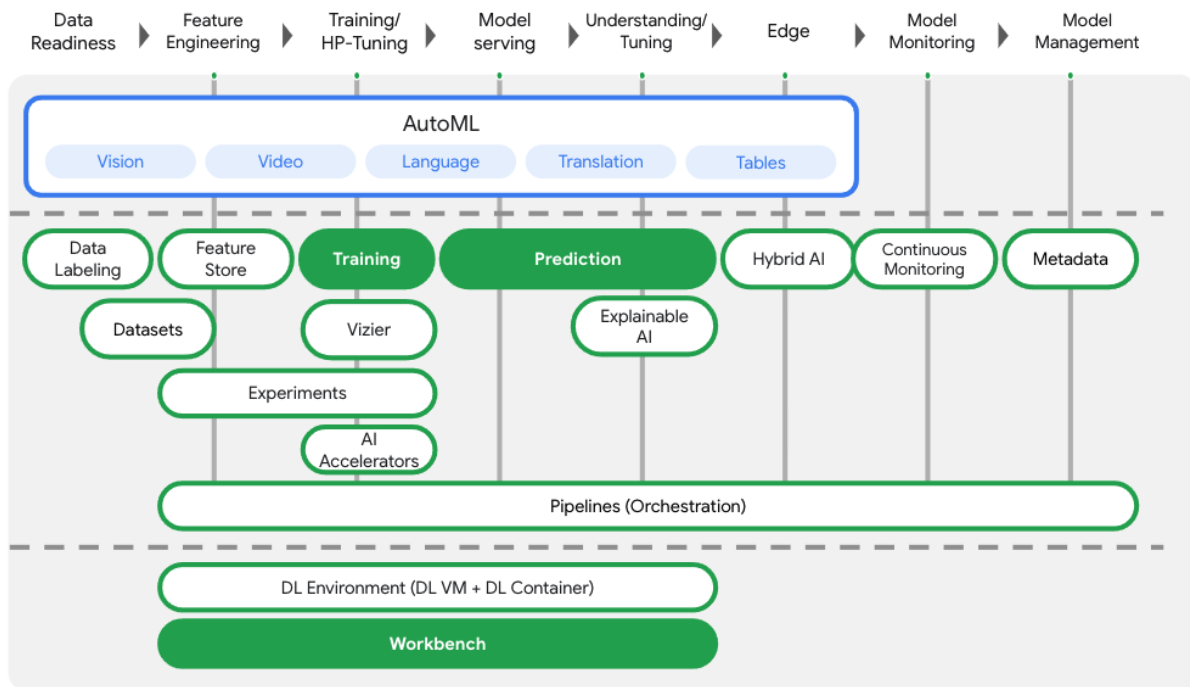
Deploy your trained model to an endpoint, and use that endpoint to get predictions

The total cost to run this lab on Google Cloud is about **\$1**.

Intro to Vertex AI

This experiment uses the newest AI product offering available on Google Cloud. [Vertex AI](#) integrates the ML offerings across Google Cloud into a seamless development experience. Previously, models trained with AutoML and custom models were accessible via separate services. The new offering combines both into a single API, along with other new products. You can also migrate existing projects to Vertex AI.

Vertex AI includes many different products to support end-to-end ML workflows. This lab will focus on the products highlighted below: Training, Prediction, and Workbench.



Setup your environment

You'll need a Google Cloud Platform project with billing enabled to run this experiment. To create a project, follow the [instructions here](#).

Step 1: Enable the Compute Engine API, if not already completed previously

Navigate to Compute Engine and select **Enable** if it isn't already enabled. You'll need this to create your notebook instance.

Step 2: Enable the Vertex AI API, if not already completed previously

Navigate to Vertex AI and select **Enable Vertex AI API** if it isn't already enabled. You'll need this to create your notebook instance.

Step 3: Enable the Container Registry API

Navigate to the Container Registry and select **Enable** if it isn't already. You'll use this to create a container for your custom training job.



Google Container Registry API

Google Enterprise API

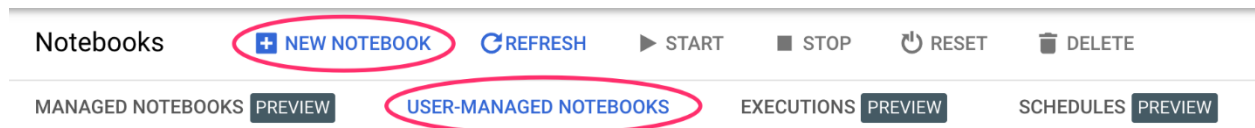
Google Container Registry provides secure, private Docker ima
Google Cloud Platform. ...

ENABLE

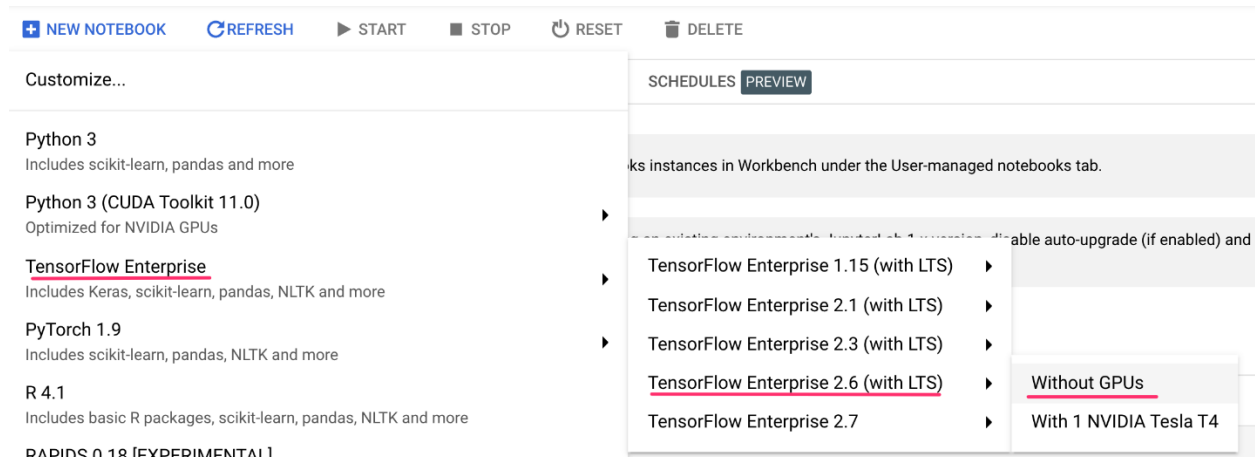
Step 4: Create a Vertex AI Workbench instance

From the Vertex AI section of your Cloud Console, click on Workbench:

From there, within **User-Managed Notebooks**, click **New Notebook**:



Then select the latest version of **TensorFlow Enterprise (with LTS)** instance type **without GPUs**:



Use the default options and then click **Create**.

If you already have a Notebook instance it may display in the **New notebook** dialog for the creation.

oks

NOT

Mig
Plat
To g

ENA

s have
earnin

Enter

New notebook

Notebook name *
tensorflow-2-6-20220110-230309
63-char limit with lowercase letters, digits, or '-' only. Must start with a letter. Cannot end with a '-'.
?

Region *
us-central1 (Iowa)
?

Zone *
us-central1-a
?

Notebook properties

Environment ?	TensorFlow Enterprise 2.6 (with LTS and Intel® MKL-DNN/MKL)
Machine type	4 vCPUs, 15 GB RAM
Boot disk	100 GB Standard persistent disk
Subnetwork	default(10.128.0.0/20)
External IP	Ephemeral(Automatic)
Permission	Compute Engine default service account
Estimated cost ?	\$102.70 monthly, \$0.141 hourly

The model we'll be training and serving in this experiment is built upon Tensorflow notebook instance. The experiment uses the [Auto MPG dataset](#) from Kaggle to predict the fuel efficiency of a vehicle.

Containerize training code

We'll submit this training job to Vertex by putting our training code in a [Docker container](#) and pushing this container to [Google Container Registry](#). Using this approach, we can train a model built with any framework.

To start, from the Launcher menu, open a Terminal window in your notebook instance:



Notebook



Python 3



Python 2



Console



Python 3



Python 2



Other



Terminal



Text File



Markdown File



Show
Contextual Help

Create a new directory called mpg and cd into it:

```
mkdir mpg  
cd mpg
```

Step 1: Create a Dockerfile

Our first step in containerizing our code is to create a Dockerfile. In our Dockerfile we'll include all the commands needed to run our image. It'll install all the libraries we're using and set up the entry point for our training code. From your Terminal, create an empty Dockerfile:

```
touch Dockerfile
```

Open the Dockerfile and copy the following into it:

```
FROM gcr.io/deeplearning-platform-release/tf2-cpu.2-6  
WORKDIR /  
  
# Copies the trainer code to the docker image.
```

```
COPY trainer /trainer
```

```
# Sets up the entry point to invoke the trainer.  
ENTRYPOINT ["python", "-m", "trainer.train"]
```

This Dockerfile uses the [Deep Learning Container TensorFlow Enterprise 2.3 Docker image](#). The Deep Learning Containers on Google Cloud come with many common ML and data science frameworks pre-installed. The one we're using includes TF Enterprise 2.3, Pandas, Scikit-learn, and others. After downloading that image, this Dockerfile sets up the entrypoint for our training code. We haven't created these files yet – in the next step, we'll add the code for training and exporting our model.

Step 2: Create a Cloud Storage bucket

In our training job, we'll export our trained TensorFlow model to a Cloud Storage Bucket. Vertex will use this to read our exported model assets and deploy the model. From your Terminal, run the following to define an env variable for your project, making sure to replace your-cloud-project with the ID of your project:

You can get your project ID by running

```
gcloud config list --format 'value(core.project)'
```

datapipe-20220110-student10xin

in your terminal, update this command to reflect your project id for the PROJECT_ID environment variable definition.

```
PROJECT_ID='datapipe-cloud-project-id'
```

Next, run the following in your Terminal to create a new bucket in your project, if the bucket does not already exist. The -l (location) flag is important since this needs to be in the same region where you deploy a model endpoint later in the tutorial:

```
BUCKET_NAME="gs://${PROJECT_ID}-bucket"  
gsutil mb -l us-central1 $BUCKET_NAME
```

Step 3: Add model training code

From your Terminal, run the following to create a directory for our training code and a Python file where we'll add the code:

```
mkdir trainer  
touch trainer/train.py
```

You should now have the following in your mpg/ directory:

```
+ Dockerfile
+ trainer/
  + train.py
```

Next, open the train.py file you just created and copy the code below.

At the beginning of the file, update the BUCKET variable with the name of the Storage Bucket you created in the previous step:

```
import numpy as np
import pandas as pd
import pathlib
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)

"""## The Auto MPG dataset

The dataset is available from the [UCI Machine Learning
Repository] (https://archive.ics.uci.edu/ml/).

### Get the data
First download the dataset.
"""

dataset_path = keras.utils.get_file("auto-mpg.data",
    "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data")
dataset_path

"""Import it using pandas"""

column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
    'Acceleration', 'Model Year', 'Origin']
dataset = pd.read_csv(dataset_path, names=column_names,
    na_values = "?", comment='\t',
    sep=" ", skipinitialspace=True)

dataset.tail()

# TODO: replace `your-gcs-bucket` with the name of the Storage bucket you
created earlier
BUCKET = 'gs://your-gcs-bucket'

"""### Clean the data

The dataset contains a few unknown values.
"""
```

```

dataset.isna().sum()

"""To keep this initial tutorial simple drop those rows."""

dataset = dataset.dropna()

"""The `Origin` column is really categorical, not numeric. So convert
that to a one-hot:"""

dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3:
'Japan'})

dataset = pd.get_dummies(dataset, prefix='', prefix_sep='')
dataset.tail()

"""### Split the data into train and test

Now split the dataset into a training set and a test set.

We will use the test set in the final evaluation of our model.
"""

train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)

"""### Inspect the data

Have a quick look at the joint distribution of a few pairs of columns from
the training set.

Also look at the overall statistics:
"""

train_stats = train_dataset.describe()
train_stats.pop("MPG")
train_stats = train_stats.transpose()
train_stats

"""### Split features from labels

Separate the target value, or "label", from the features. This label is
the value that you will train the model to predict.
"""

train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')

"""### Normalize the data

Look again at the `train_stats` block above and note how different the
ranges of each feature are.

It is good practice to normalize features that use different scales and
ranges. Although the model might converge without feature normalization,

```


it makes training more difficult, and it makes the resulting model dependent on the choice of units used in the input.

Note: Although we intentionally generate these statistics from only the training dataset, these statistics will also be used to normalize the test dataset. We need to do that to project the test dataset into the same distribution that the model has been trained on.

"""

```
def norm(x):
    return (x - train_stats['mean']) / train_stats['std']
normed_train_data = norm(train_dataset)
normed_test_data = norm(test_dataset)
```

"""This normalized data is what we will use to train the model.

Caution: The statistics used to normalize the inputs here (mean and standard deviation) need to be applied to any other data that is fed to the model, along with the one-hot encoding that we did earlier. That includes the test set as well as live data when the model is used in production.

The model

Build the model

Let's build our model. Here, we'll use a `Sequential` model with two densely connected hidden layers, and an output layer that returns a single, continuous value. The model building steps are wrapped in a function, `build_model`, since we'll create a second model, later on.

"""

```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation='relu',
input_shape=[len(train_dataset.keys())]),
        layers.Dense(64, activation='relu'),
        layers.Dense(1)
    ])

    optimizer = tf.keras.optimizers.RMSprop(0.001)

    model.compile(loss='mse',
                  optimizer=optimizer,
                  metrics=['mae', 'mse'])
    return model
```

```
model = build_model()
```

"""### Inspect the model

Use the `.summary` method to print a simple description of the model

"""

```
model.summary()
```

```
"""Now try out the model. Take a batch of `10` examples from the training
data and call `model.predict` on it.
```

```
It seems to be working, and it produces a result of the expected shape and
type.
```

```
### Train the model
```

```
Train the model for 1000 epochs, and record the training and validation
accuracy in the `history` object.
```

```
Visualize the model's training progress using the stats stored in the
`history` object.
```

```
This graph shows little improvement, or even degradation in the validation
error after about 100 epochs. Let's update the `model.fit` call to
automatically stop training when the validation score doesn't improve.
We'll use an EarlyStopping callback* that tests a training condition
for every epoch. If a set amount of epochs elapses without showing
improvement, then automatically stop the training.
```

```
You can learn more about this callback
```

```
[here] (https://www.tensorflow.org/api\_docs/python/tf/keras/callbacks/Early
Stopping).
```

```
"""
```

```
model = build_model()
```

```
EPOCHS = 1000
```

```
# The patience parameter is the amount of epochs to check for improvement
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss',
patience=10)
```

```
early_history = model.fit(normed_train_data, train_labels,
                           epochs=EPOCHS, validation_split = 0.2,
                           callbacks=[early_stop])
```

```
# Export model and save to GCS
model.save(BUCKET + '/mpg/model')
```

Step 4: Build and test the container locally

From your Terminal, define a variable with the URI of your container image in Google Container Registry:

```
IMAGE_URI="gcr.io/$PROJECT_ID/mpg:v1"
```

Then, build the container by running the following from the root of your mpg directory:

```
docker build ./ -t $IMAGE_URI
```

Run the container within your notebook instance to ensure it's working correctly:

```
docker run $IMAGE_URI
```

The model should finish training in 1-2 minutes with a validation accuracy around 72% (exact accuracy may vary). When you've finished running the container locally, push it to Google Container Registry:

```
docker push $IMAGE_URI
```

With our container pushed to Container Registry, we're now ready to kick off a custom model training job.

Note: Double check to see that you had success and not an error. In the event we missed the step to enable the Container Registry API in setup we would see something similar to below on our **docker push** command.

```
unknown: Service 'containerregistry.googleapis.com' is not enabled for  
consumer 'project:datapipe-20220110-student10xin'.
```

Run a training job on Vertex AI

Vertex AI gives you two options for training models:

- **AutoML:** Train high-quality models with minimal effort and ML expertise.
- **Custom training:** Run your custom training applications in the cloud using one of Google Cloud's pre-built containers or use your own.

In this lab, we're using custom training via our own custom container on Google Container Registry. To start, navigate to the **Models** section in the Vertex section of your Cloud console:



Vertex AI



Dashboard



Datasets



Features



Labelling tasks



Workbench



Pipelines



Training



Experiments



Models



Endpoints



Batch predictions



Metadata

Step 1: Kick off the training job

Click **Create**

Models

 CREATE

 IMPORT

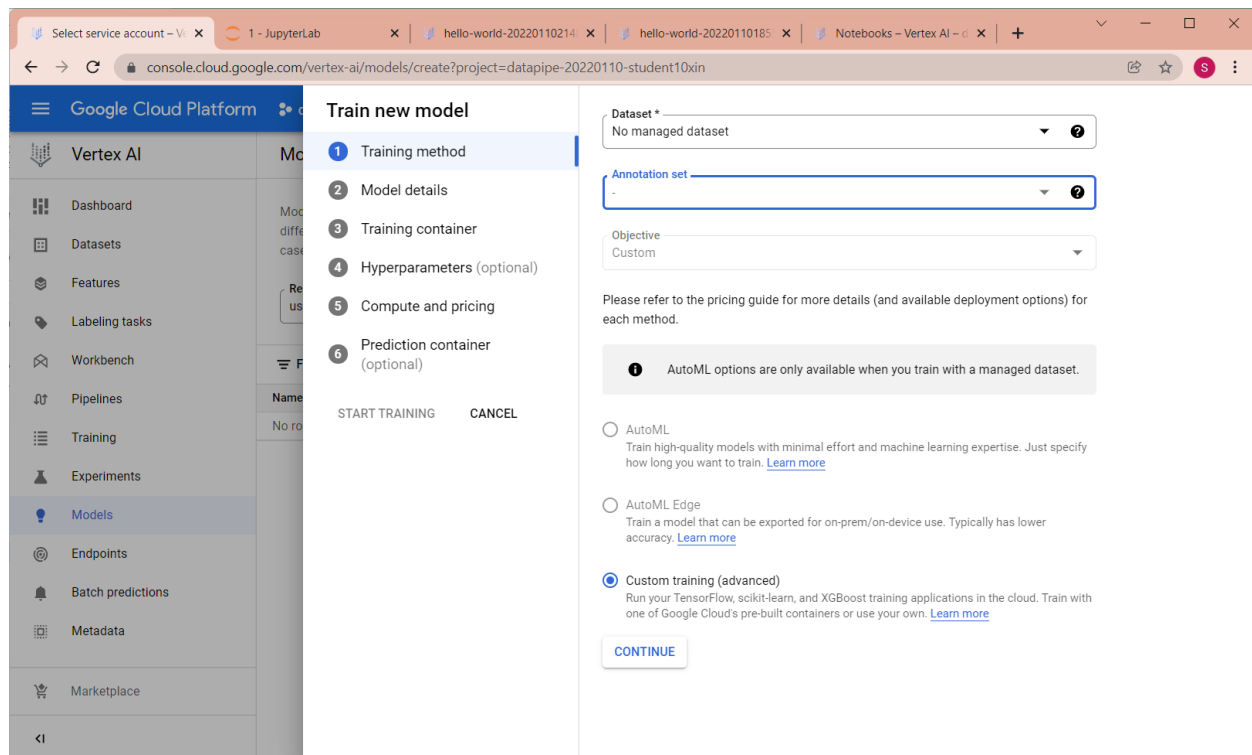
Models are built from your datasets or unmanaged data sources. There are many different types of machine learning models available on Vertex AI, depending on your use case and level of experience with machine learning. [Learn more](#)

Region

us-central1 (Iowa)



Enter the parameters for your training job and deployed model:



- Under **Dataset**, select **No managed dataset**
- Then select **Custom training (advanced)** as your training method.
- Click **Continue**

In the next step, enter **mpg** (or whatever you'd like to call your model) for **Model name**.

Model name *

✓ ADVANCED OPTIONS

Click **Continue**

Now select **Custom container** and Browse for the container we created earlier in this experiment.

Note: If you see this, we missed the step to enable the Container Registry API. We can do that now.

Select container image

CONTAINER REGISTRY

ARTIFACT REGISTRY

Project: datapipe-20220110-student10xin [CHANGE](#)



The [Container Registry API](#) is not enabled for this project

Select a pre-built container or build a custom container using ML frameworks (as well as non-ML dependencies, libraries and binaries) that are not otherwise supported.

[Learn more](#)



Pre-built container

View the list of [supported runtimes](#) including TensorFlow and scikit-learn versions



Custom container

Build a custom Docker container. Must be stored in [Container Registry](#)

In the **Container image** text box, click **Browse** and find the Docker image you just uploaded to Container Registry. Leave the rest of the fields blank and click **Continue**.

Select container image

CONTAINER REGISTRY

ARTIFACT REGISTRY

Project: datapipe-20220110-student10xin [CHANGE](#)



gcr.io/datapipe-20220110-student10xin/mpg

40b3bd6866

v1

SELECT

CANCEL

Custom container settings

Container image *

gcr.io/datapipe-20220110-student10xin/mpg@sha256:40b3bd6866a7b7

BROWSE

gs:// Model output directory

BROWSE

Your model artifacts and other data needed for training will be stored on Cloud Storage. You should specify a path here if you do not set an output directory in your application code or arguments.

Arguments

Optional. Add arguments for the command that runs when the container starts. Overrides the container's CMD instruction. Enter one parameter and its argument per line.

```
--flag_a=xxxx  
-flag2  
flag3
```

For parameters you want to tune with HyperTune, enter arguments of the hyperparameters you defined in the training code in the hyperTune setting below. If none, click Next to skip this step.

CONTINUE

We won't use hyperparameter tuning in this tutorial, so leave the Enable hyperparameter tuning box **unchecked** and click **Continue**.

Hyperparameter tuning optimizes your model through multiple trials but will increase the cost of this job. After training finishes, the best model will be saved to your Model List. [Learn more](#)

☐ Enable hyperparameter tuning

CONTINUE

In **Compute and pricing**, leave the selected region as-is and choose **n1-standard-4** as your machine type:

Filter Type to filter

Standard

High-memory

High-CPU

Custom

High-GPU

Mega-GPU

n1-standard-4

4 vCPUs, 15 GiB memory

n1-standard-8

8 vCPUs, 30 GiB memory

n1-standard-16

16 vCPUs, 60 GiB memory

n1-standard-32

32 vCPUs, 120 GiB memory

CLEAR SELECTION

Worker pool 0

Machine type *

n1-standard-4, 4 vCPUs, 15 GiB memory

Accelerator type

Accelerators can speed up model training that involves intensive compute tasks. [Learn more](#)

Leave the accelerator fields blank and select **Continue**. Because the model in this demo trains quickly, we're using a smaller machine type.

Note: You are welcome to experiment with larger machine types and GPUs if you'd like. If you use GPUs, you'll need to use a GPU-enabled base container image.

Under the **Prediction container** step, select **Pre-built container**

You can associate your custom-trained model with a container in order to serve prediction requests using Vertex AI. [Learn more about getting predictions.](#)

- ☐ No prediction container
You can always import your model artifact later to serve prediction requests
- ☒ Pre-built container
View the list of [supported runtimes](#) including TensorFlow, scikit-learn and PyTorch versions
- ☐ Custom container
Build a custom Docker container. Must be stored in [Container Registry or Artifact Registry](#)

Then select **TensorFlow 2.6**.

Leave the default settings for the pre-built container as is. Under **Model directory**, enter your GCS bucket with the **mpg** subdirectory. This is the path in your model training script where you export your trained model:

Select folder

<

datapipe-20220110-student10xin-bucket ▾

+

Q

mpg/

>

pipeline_root/

>

SELECT

CANCEL

Pre-built container settings

Vertex AI provides Docker container images for serving predictions. To use a pre-built container, your trained model code must be in Python 3.7. [Learn more about pre-built containers](#)

In order to run in a pre-built container, your code needs to be in Python 3.7

Model framework *

TensorFlow

▼

Model framework version *

2.6


▼

Accelerator type *

None

▼

Model directory *

 gs:// datapipe-20220110-student10xin-bucket/mpg/

BROWSE

Cloud Storage location containing the model artifact and any supporting files

We use the folder that we created earlier in this experiment **BUCKET_NAME**. Vertex will look in this location when it deploys your model. Now you're ready for training!

Click **Start training** to kick off the training job. In the Training section of your console, you'll see something like this:

The training job will take about 10-15 minutes to complete.

Deploy a model endpoint

When we set up our training job, we specified where Vertex AI should look for our exported model assets. As part of our training pipeline, Vertex will create a model resource based on this asset path. The model resource itself isn't a deployed model, but once you have a model you're ready to deploy it to an endpoint. To learn more about Models and Endpoints in Vertex AI, check out the [documentation](#).

In this step we'll create an endpoint for our trained model. We can use this to get predictions on our model via the Vertex AI API.

Step 1: Deploy an endpoint

When your training job completes, you should see a model named **mpg** (or whatever you named it) in the **Models** section of your console:

Filter Enter a property name						
Name	ID	Status	Data	Endpoints	Region	Type
mpg	2859262395845443584	✓ Ready	—	0	us-central1	Custom trained Custom training

When your training job ran, Vertex created a model resource for you. In order to use this model, you need to deploy an endpoint. You can have many endpoints per model. Click on the hyperlinked model name (**mpg**) and then click **Deploy to endpoint**.

←

mpg

EXPORT

DEPLOY & TEST

BATCH PREDICTIONS

MODEL PROPERTIES

Deploy your model

Endpoints are machine learning models made available for online prediction requests. are useful for timely predictions from many users (for example, in response to an appli request). You can also request batch predictions if you don't need immediate results.

DEPLOY TO ENDPOINT

Select **Create new endpoint** and give it a name, **v1**. Leave **Standard** selected for Access and then click **Continue**.

Deploy to endpoint

1 Define your endpoint

2 Model settings

3 Model monitoring

DEPLOY

CANCEL

☒ Create new endpoint ☐ Add to existing endpoint

Endpoint name *

v1

Location

Region

us-central1 (Iowa)

Access

Determines how your endpoint can be accessed. By default, endpoints are for prediction serving through a REST API. Endpoint access can't be changed when the endpoint is created.

☒ Standard

Makes the endpoint available for prediction serving through a REST API. Automatically, custom-trained models can be added to standard endpoints.

Leave **Traffic split** at 100 and enter 1 for **Minimum number of compute nodes**. Under **Machine type**, select **n1-standard-2** (or any machine type you'd like). Leave the rest of the defaults selected and then click **Continue**.

Deploy to endpoint

☒ Define your endpoint

2 Model settings

3 Model monitoring

DEPLOY

CANCEL

mpg

Traffic split *

100

%

?

Compute resources

Choose how compute resources will serve prediction traffic to your model

- **Autoscaling:** If you set a minimum and maximum, compute nodes will scale to meet traffic demand within those boundaries
- **No scaling:** If you only set a minimum, then that number of compute nodes will always run regardless of traffic demand (the maximum will be set to minimum)

Once scaling settings are set, they can't be changed unless you redeploy the model. [Pricing guide](#)

Minimum number of compute nodes *

1

Default is 1. If set to 1 or more, then compute resources will continuously run even without traffic demand. This can increase cost but avoid dropped requests due to node initialization.

Maximum number of compute nodes (optional)

Enter a number equal to or greater than the minimum nodes. Can reduce costs but may cause reliability issues for high traffic.

✓ ADVANCED SCALING OPTIONS

Machine type *

n1-standard-2, 2 vCPUs, 7.5 GiB memory

We won't enable monitoring for this model, so next click **Deploy** to kick off the endpoint deployment.

Deploy to endpoint

✓ Define your endpoint

✓ Model settings

3 Model monitoring

DEPLOY

CANCEL

ⓘ

Settings in this step apply to all models deployed to this endpoint

Model monitoring

You can monitor the tabular and custom models deployed to this endpoint for changes in feature drift, training-serving skew and other metrics to help you understand how your model is performing in the real world.

Enable model monitoring for this endpoint

Deploying the endpoint will take 10-15 minutes, and an email will be sent when the deployment completes, although since we're using training accounts we won't receive that email.

←

mpg

↓

EXPORT

DEPLOY & TEST

BATCH PREDICTIONS

MODEL PROPERTIES

Deploy your model

Endpoints are machine learning models made available for online prediction requests. Endpoints are useful for timely predictions from many users (for example, in response to an application request). You can also request batch predictions if you don't need immediate results.

DEPLOY TO ENDPOINT

Name	ID	Status	Models	Region
v1	3189472125945643008	Deploying model	0	us-central1

When the endpoint has finished deploying, you'll see the following, which shows one endpoint deployed under your Model resource:

[←](#)
[EXPORT](#)

[DEPLOY & TEST](#)
[BATCH PREDICTIONS](#)
[MODEL PROPERTIES](#)

Deploy your model

Endpoints are machine learning models made available for online prediction requests. Endpoints are useful for timely predictions from many users (for example, in response to an application request). You can also request batch predictions if you don't need immediate results.

[DEPLOY TO ENDPOINT](#)

Name	ID	Status	Models	Region	Monitoring
v1	3189472125945643008	✓ Active	1	us-central1	Disabled

Step 2: Get predictions on the deployed model

We'll get predictions on our trained model from a Python notebook, using the Vertex Python API.

[Notebooks](#)
[+ NEW NOTEBOOK](#)
[REFRESH](#)
[▶ START](#)
[■ STOP](#)
[🔄 RESI](#)

[MANAGED NOTEBOOKS](#)
[PREVIEW](#)
[USER-MANAGED NOTEBOOKS](#)
[EXECUTIONS](#)
[PREVIEW](#)

Notebooks have JupyterLab 3 pre-installed and are configured with GPU-enabled machine learning frameworks. [Learn more](#)

i You have 1 instances that aren't registered with the new Notebooks API. They won't receive additional functionality until they're registered.

[REGISTER ALL](#)

Filter

Enter property name or value

?

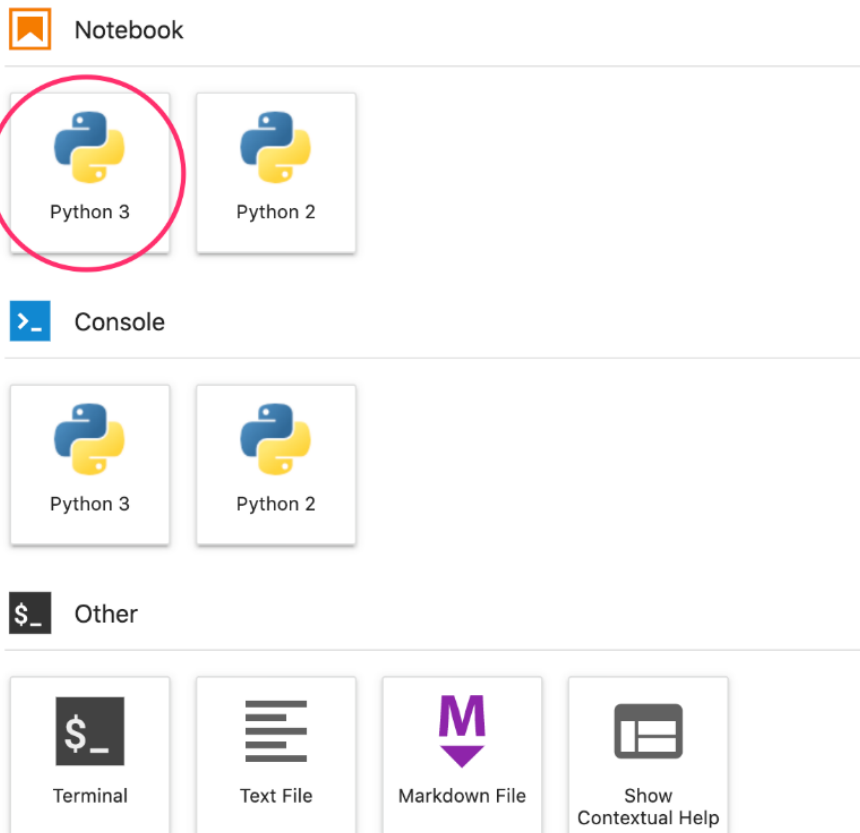
☰

<input type="checkbox"/>	<input checked="" type="radio"/>	Notebook name ↑	Zone	Auto-upgrade
<input type="checkbox"/>	<input checked="" type="radio"/>	tensorflow-2-6-20220110-230309	us-central1-a	—

Click the **REGISTER ALL** in the notification on the notebook. Note the Information Icon next to our Tensorflow 2.6 notebook.

Choose **OPEN JUPYTERLAB**

Create a Python 3 notebook from the Launcher:



In your notebook, run the following in a cell to install the Vertex AI SDK:

```
!pip3 install google-cloud-aiplatform --upgrade --user
```

The image shows a JupyterLab notebook interface. The top bar shows 'Untitled.ipynb' and a toolbar with icons for saving, adding, deleting, and running cells, as well as a 'Code' button. The notebook cell contains the following text:

```
[1]: !pip3 install google-cloud-aiplatform --upgrade --user
```

Below the command, the output shows the installation progress:

```
Requirement already satisfied: google-cloud-aiplatform in /opt,  
Collecting google-cloud-aiplatform  
  Downloading google_cloud_aiplatform-1.9.0-py2.py3-none-any.wl  
  |████████████████████████████████████████| 1.7 MB 5.3 MB/s  
Requirement already satisfied: google-cloud-storage<2.0.0dev,>:
```

Then add a cell in your notebook to import the SDK and create a reference to the endpoint you just deployed:

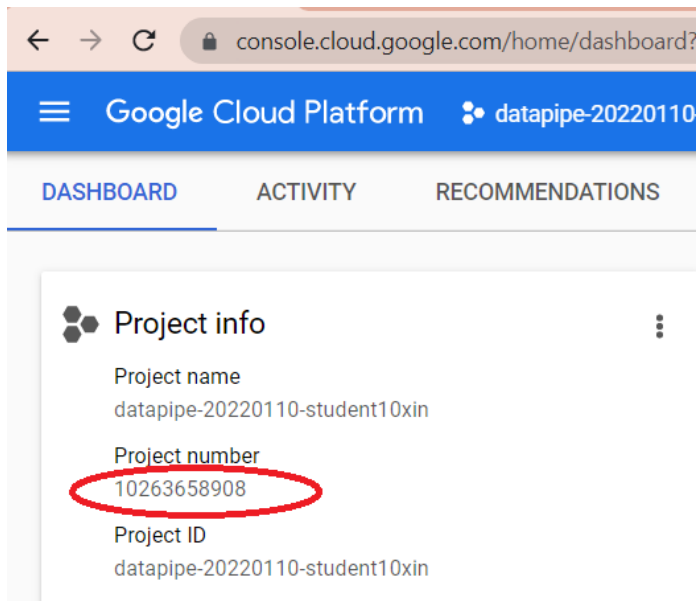

```

from google.cloud import aiplatform

endpoint = aiplatform.Endpoint(
    endpoint_name="projects/YOUR-PROJECT-NUMBER/locations/us-
centrall/endpoints/YOUR-ENDPOINT-ID"
)

```

You'll need to replace two values in the endpoint_name string above with your project number and endpoint. You can find your project number by navigating to your [project dashboard](#) and getting the Project Number value.



You can find your endpoint ID in the endpoints section of the console here:

Name	ID	Status	Models	Region
v1	[REDACTED]	✓ Active	1	us-central1

Finally, make a prediction to your endpoint by copying and running the code below in a new cell:

```

test_mpg = [1.4838871833555929,
1.8659883497083019,
2.234620276849616,
1.0187816540094903,
-2.530890710602246,
-1.6046416850441676,
-0.4651483719733302,
-0.4952254087173721,

```

```
0.7746763768735953]

response = endpoint.predict([test_mpg])

print('API response: ', response)

print('Predicted MPG: ', response.predictions[0][0])
```

This example already has normalized values, which is the format our model is expecting.

Run this cell, and you should see a prediction output around 16 miles per gallon.

```
[3]: test_mpg = [1.4838871833555929,
1.8659883497083019,
2.234620276849616,
1.0187816540094903,
-2.530890710602246,
-1.6046416850441676,
-0.4651483719733302,
-0.4952254087173721,
0.7746763768735953]

response = endpoint.predict([test_mpg])

print('API response: ', response)

print('Predicted MPG: ', response.predictions[0][0])

API response: Prediction(predictions=[[15.8347549]],
Predicted MPG: 15.8347549
```

🎉 Congratulations! 🎉

You've learned how to use Vertex AI to:

- Train a model by providing the training code in a custom container. You used a TensorFlow model in this example, but you can train a model built with any framework using custom containers.
- Deploy a TensorFlow model using a pre-built container as part of the same workflow you used for training.
- Create a model endpoint and generate a prediction.

To learn more about different parts of Vertex, check out the [documentation](#).

Cleanup

If you'd like to continue using the notebook you created in this lab, it is recommended that you turn it off when not in use. From the Workbench UI in your Cloud Console, select the notebook and then select **Stop**.

If you'd like to delete the notebook entirely, click the Delete button in the top right.

To delete the endpoint you deployed, navigate to the **Endpoints** section of your Vertex AI console, click on the endpoint you created, and then select **Undeploy model from endpoint**:

<input checked="" type="checkbox"/>	Model	Status	Most recent alerts	Monitoring	Traffic split	Compute nodes	Type	Created ↑	
<input checked="" type="checkbox"/>	mpg	✓ Ready	—	Disabled	100%	Auto (1 minimum, 1 maximum)	Custom	2 Dec 2021,	<div>Undeploy model from endpoint</div>

DEPLOY ANOTHER MODEL

To delete the Storage Bucket, using the Navigation menu in your Cloud Console, browse to Storage, select your bucket, and click Delete:

datapipe-20220110-student10xin		storage browser
Browser	+ CREATE BUCKET	DELETE REFRESH
Filter Filter buckets		
<input checked="" type="checkbox"/>	Name ↑	Created
<input checked="" type="checkbox"/>	datapipe-20220110-student10xin-buc...	Jan 10, 2022, 2:55:00 PM


Post note: You may have noticed when we enabled the Container Registry API. The default moving forward for containers will be the Artifact Registry. Using the Container Registry currently is fine, but incurs technical debt that will have to be addressed in the future.

datapipe-20220110-student10xin

Search

2

Repositories



Transition to Artifact Registry

Artifact Registry is the recommended service for managing container images. Container Registry is still supported but will only receive critical security fixes. [Learn more about options to transition to Artifact Registry.](#)

[TRY ARTIFACT REGISTRY](#) [LEARN MORE](#)