**Deploying a Machine Learning Model: Part 3— Google Cloud Vertex AI**

[[Tejas Ekawade](https://medium.com/@TejasEkawade?source=post_page-----5287cf9e45ec--------------------------------)](https://medium.com/@TejasEkawade?source=post_page-----5287cf9e45ec--------------------------------)

[Tejas Ekawade](https://medium.com/@TejasEkawade?source=post_page-----5287cf9e45ec--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F46a3e6c6898d&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40TejasEkawade%2Fdeploying-a-machine-learning-model-part-3-google-cloud-vertex-ai-5287cf9e45ec&user=Tejas+Ekawade&userId=46a3e6c6898d&source=post_page-46a3e6c6898d----5287cf9e45ec---------------------post_header-----------)

9 min read

·

Oct 18, 2023

19

In the previous articles on Machine Learning Model deployment, we have looked at the [***basic concepts of deployment***](https://medium.com/@ekawade1394/deploying-a-machine-learning-model-getting-started-2644392c8953)and then [***deployed a model locally***](https://medium.com/@ekawade1394/deploying-a-machine-learning-model-part-2-local-deployment-1c36d029ce7a).  
Let’s take this a step further and understand how ML models can be deployed on Cloud.

I will be using **Google Cloud Platform — Vertex AI**to demonstrate this procedure. However, AWS and Azure also follow a very similar pattern of deployment. The deployment procedure can be summarized as follows:

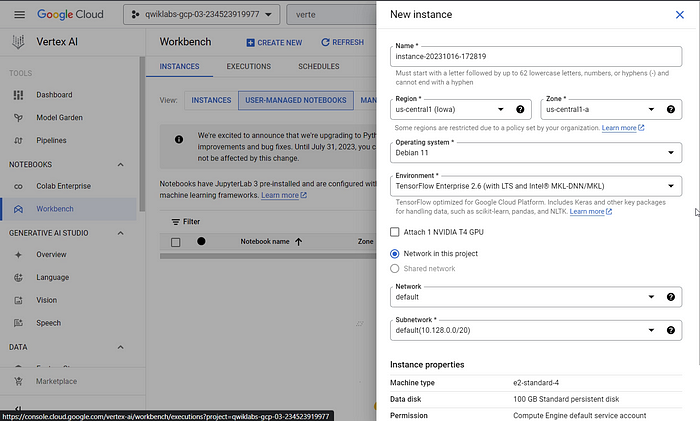
1. Containerizing the training code
2. Submitting the training job
3. Saving the model artifact in the registry
4. Deploying the saved model as an endpoint using a prediction container. (Cloud platforms like Azure support MLFlow as well)
5. Testing the deployed model using the endpoint

For this article, we will have an in-depth example of deployment and testing. I have followed a [Qwiklabs tutorial](https://www.cloudskillsboost.google/course_sessions/5370365/labs/394208" \t "_blank). Qwiklabs provides us with temporary credentials to work on Google Cloud for a limited time.

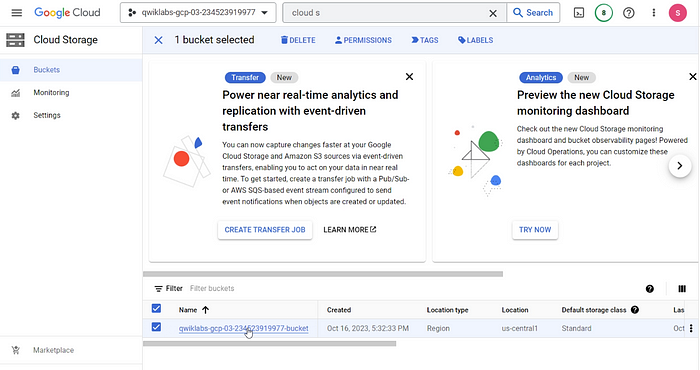
Without further ado, let’s begin the training, deploying, and testing a model on the cloud:

**Step 1: APIs** — Ensure that a project is created in the GCP console and the APIs are enabled for Google Compute Engine, Vertex AI, Cloud Storage, and Container Registry

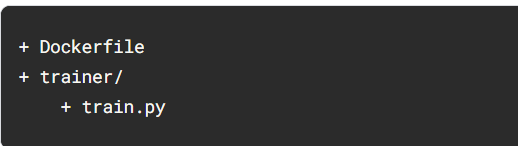
**Step 2: Vertex AI workbench**— Open Vertex AI user-managed Jupyter Notebook Instance. We will be creating notebooks and using the terminal on the Jupyterlab instance.



**Step 3: Create a Google Cloud Storage Bucket** — Here we are going to save our trained model which will be later deployed.



**Step 4: Create a directory** with the following folder structure in the Jupyter instance.



**Step 5: Write a basic Tensorflow training script** called ‘trainer/train.py’. (The script can be read at the end if the goal is to understand the flow).

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://qwiklabs-gcp-03-234523919977-bucket'  
  
### Clean the data  
dataset.isna().sum()  
dataset = dataset.dropna()  
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  
You will use the test set in the final evaluation of your 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.  
### 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()  
"""### Train the model  
"""  
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 6: Create a Dockerfile —**We are creating a Dockerfile using a deep learning base image that consists of the popular ML libraries. This is an example of custom container training. GCP provides another way of submitting training jobs but we will cover that later.

***Why is docker needed?*** I have tried explaining it here: <https://medium.com/@ekawade1394/deploying-a-machine-learning-model-getting-started-2644392c8953>

FROM gcr.io/deeplearning-platform-release/tf2-cpu.2-3  
WORKDIR /root  
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"]

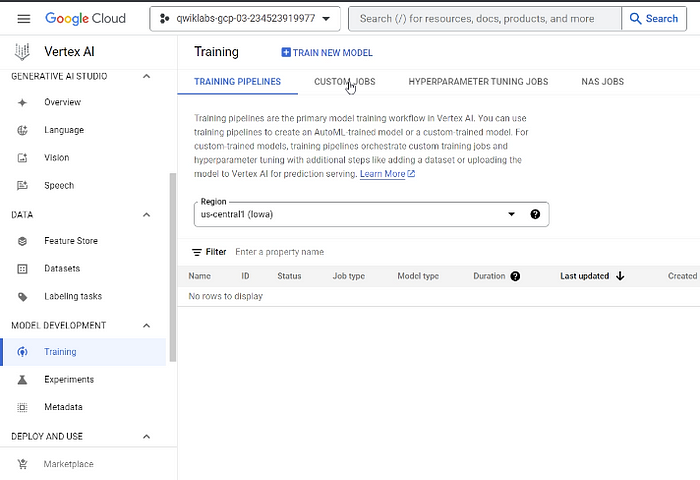
Step 7: **Containerize the training code —**Create a docker image, run it, and push it to Google Container Registry. This can all be done from the terminal of the opened notebook instance.

PROJECT\_ID=qwiklabs-gcp-03-2a0f277674bb  
IMAGE\_URI="gcr.io/$PROJECT\_ID/mpg:v1"  
  
docker build ./ -t $IMAGE\_URI  
docker run $IMAGE\_URI  
docker push $IMAGE\_URI

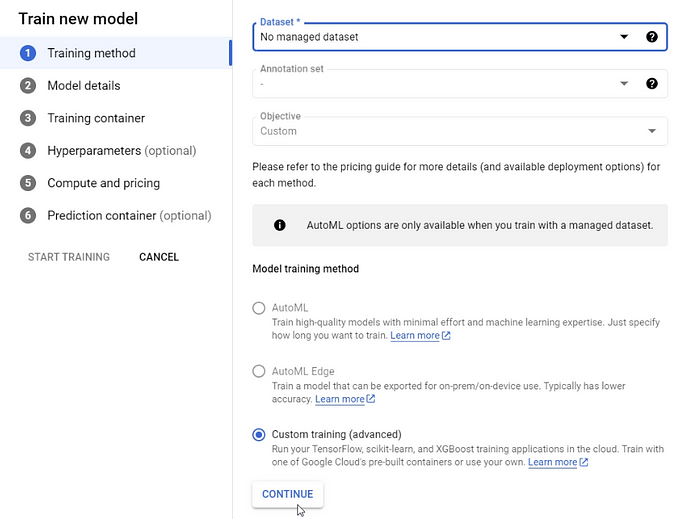


**Step 8: Running a training job** — Our training job will create a Tensorflow model. This model will be saved in the Cloud Storage as specified in the script. Additionally, the script will also save the model in ‘Model Registry’ which will make it easier for deployment purposes.

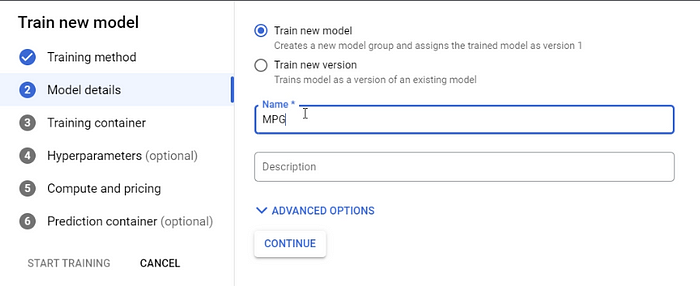
Go to the Training section on the Vertex AI page and submit a job in the following manner:



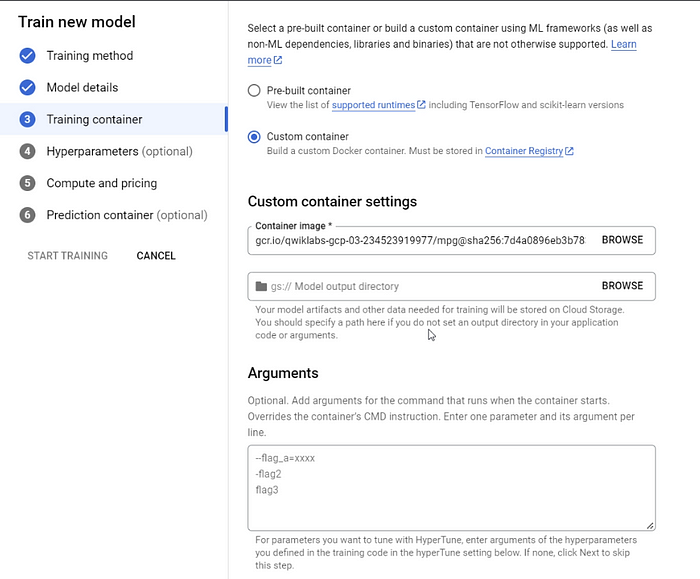
**8.1. Training Method** — We are downloading a dataset in the training script. Thus, we don’t need a managed dataset and we can click on the Custom Training option.



**8.2. Model Details** — Assign a name to the model as follows:

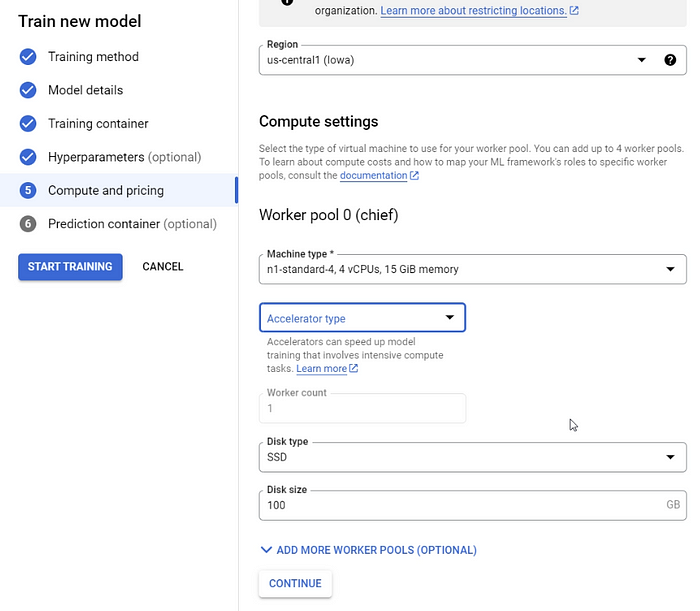


**8.3. Training Container** — Select the Custom Container option and provide a reference to the docker image that was earlier created and saved in the Container registry.

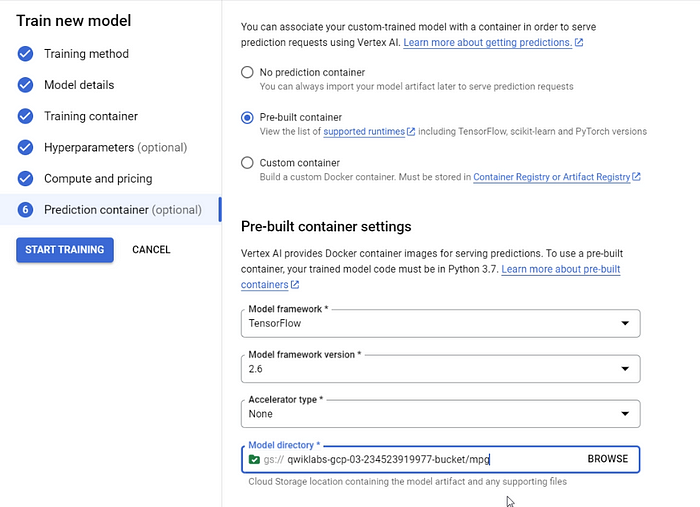


**8.4. Hyperparameters** — We won’t be tuning hyperparameters in this example, hence this part is skipped.

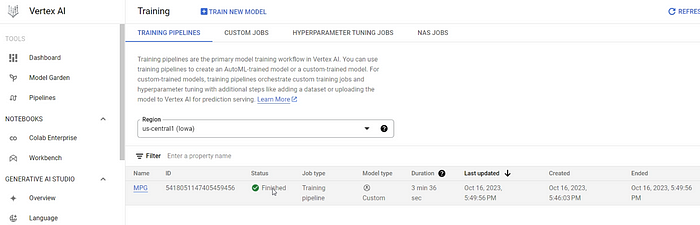
**8.5. Compute and Pricing —**As this is a simple model script, it doesn’t require heavy computing.



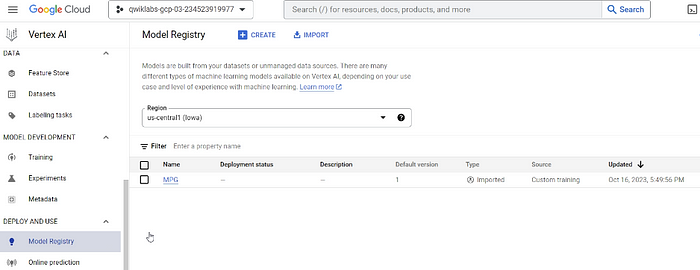
**8.6. Prediction Container —**Just like the training has its own container, so does the prediction part.  
We have seen in the previous articles that model serving requires an HTTP server and we might need to create a Flask/FastAPI application. However, that step is not required here as we can instead choose a pre-built container provided by Vertex AI.



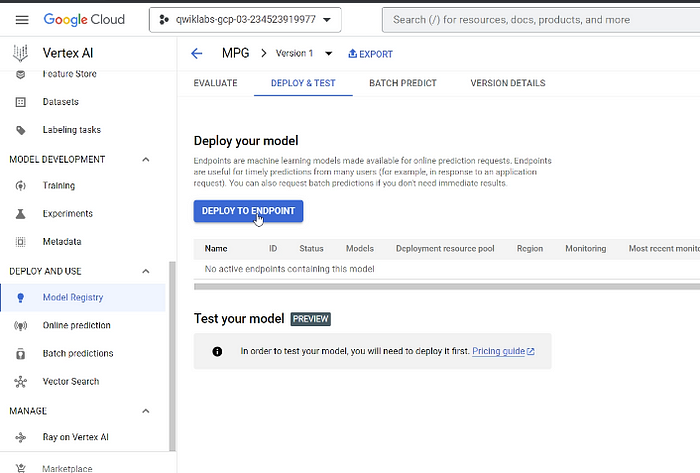
**Step 9: Monitor** and wait for the Training job to finish



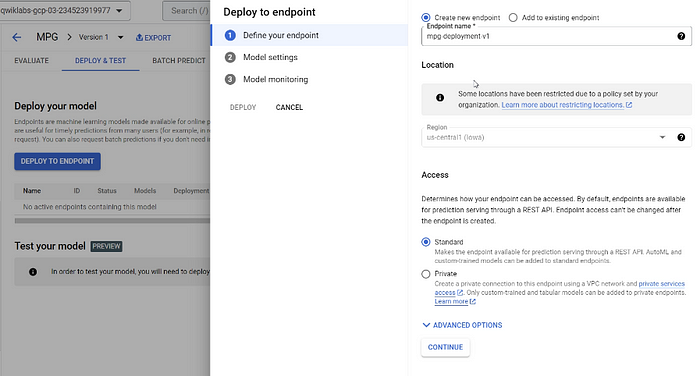
**Step 10: Model Registry** — Check the saved model in the Model Registry



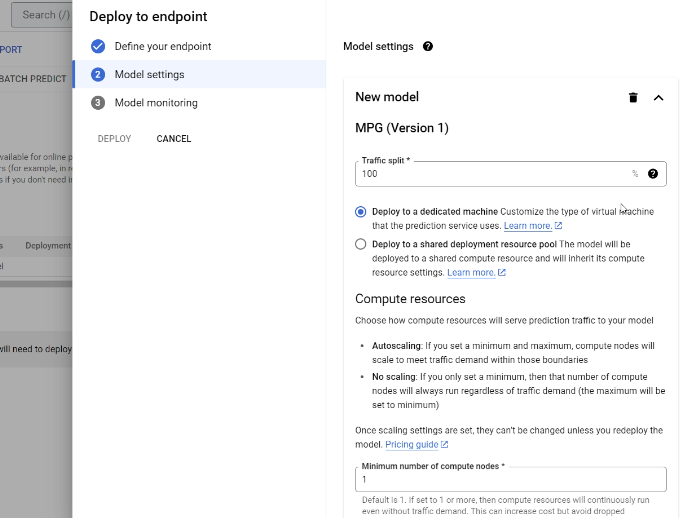
**Step 11: Deploying** **the saved model** — We are going to save our model to an online endpoint. This will allow the end users to query the model using the REST API. But deployment also has a few configurations of its own which we will see now:

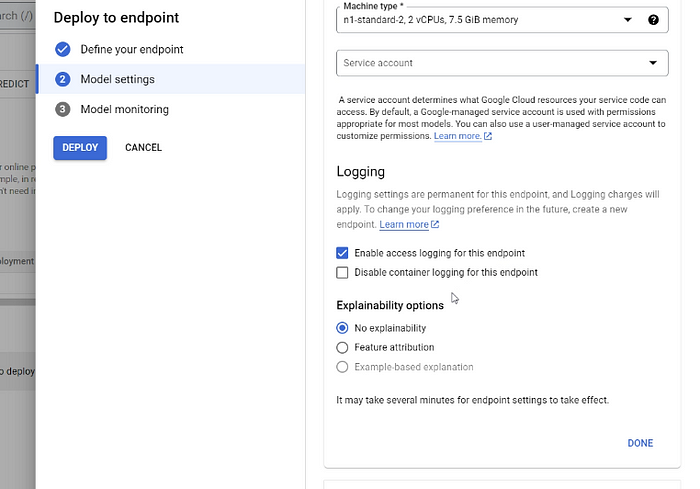


**11. 1. Defining endpoint**: We will name the endpoint and choose the access type.



**11.2. Model Settings:** Deployment also has a set of configurations such as traffic split, compute resources, logging, and explainability. We are going to choose a very standard machine type for this example.



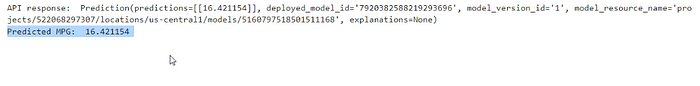


**11.3. Model Monitoring:** There is an option to monitor the model which in our case isn’t switched on.

**Step 12: Testing the online endpoint**  
The endpoint can be tested either using CLI or Google Cloud SDK. We will go with the latter option. We can create a jupyter notebook in the Workbench jupyterlab and test our newly created endpoint.

This is the code for testing:

!pip3 install google-cloud-aiplatform --upgrade --user  
  
from google.cloud import aiplatform  
  
project\_number = 522068297307 # Will be changed as per the user's project  
endpoint\_id = 8169619884003557376 # Will be changed as per the endpoint  
  
endpoint = aiplatform.Endpoint(  
 endpoint\_name=f"projects/{project\_number}/locations/us-central1/endpoints/{endpoint\_id}"  
)  
  
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=[[16.421154]], deployed\_model\_id='7920382588219293696', model\_version\_id='1', model\_resource\_name='projects/522068297307/locations/us-central1/models/5160797518501511168', explanations=None)  
Predicted MPG: 16.421154  
'''



Model response using the endpoint

We are getting the desired response which indicates that our model has been deployed and we are able to get predictions using the endpoint.

Those were a lot of pictures but understanding the concepts of model deployment visually makes it much easier to understand. As mentioned earlier, the deployment procedure is similar for other cloud platforms.

**What’s next?**

* Google Cloud offers two options for both training and prediction containers: pre-built and custom containers. For the training job, we used a custom container by creating our own docker image.  
  However, if we want to use a pre-built container for training, then we need to:  
  1. Create a Python package consisting of the training script and necessary library requirements  
  2. Save the training package in GCS  
  3. Provide the reference for the training job
* The next step after deployment is to create a ***pipeline***using the ***training and deployment components.*** GCP provides Kubeflow for orchestrating pipelines. Here’s an article: [**ML orchestration — Kubeflow**](https://medium.com/@ekawade1394/ml-orchestration-kubeflow-d95e547b3af).  
  These pipelines can be used to automate the ML process for CI/CD/CT (Continuous Integration/ Continuous Deployment/ Continuous Training). The process is generally as follows:  
  1. The pipeline can be scheduled as per the data update to trigger the training component  
  2. The training component will create a model which will then be compared against the existing model  
  3. If the new model ticks the evaluation metrics then the deployment component will be triggered and the new model will be deployed. This covers the CD/CT aspects  
  4. The CI/CD part is similar to traditional software engineering. Suppose there is some code change in our repository like GitHub then the training component can be triggered to create a new model.

This completes the 3-part series of ML Deployment. Hope you found this series useful and informative. Have fun deploying and creating real-time impact with Machine Learning!