Capstone Project Proposal - Inventory Monitoring at Distribution Centers

Introduction

Distribution centers have a lot of repetitive processes in their daily operation, nowadays, a lot of these processes have been automated with AI and robotics. A lot of operations using robotics need to feed the robot's logic with information about its surroundings, it can be done using computer vision algorithms such as Convolutional Neural Networks (CNN). This project is about a computer vision solution to help robots successfully achieve their daily tasks.

Robots in a distribution center need to move bins with items from one place to another, for example, robots move items to set up the inventory in warehouses. Usually, these moving operations are achieved using bins, the robot needs to know how many items each bin has to keep count of the number of items moved. It is difficult to equip a moving robot with the necessary technology to make this counting process accurate and affordable, but, considering that we live in the technology revolution, another solution can be to use an external system that does this counting job, maybe with an image of the bins' content.

The most feasible solution to this problem is to equip the robot with a camera that focuses on the bins' content and sends the image to a system that does the job. This outsourced system has to be equipped with the necessary AI to identify how many items a bin has by using only an image of the bin's content. This AI will be CNN which needs to be trained with hundreds or even thousands of labeled images. In this project, we want to train and deploy a model for helping the robots within a distribution center to identify how many items are in a bin's photo.

Dataset

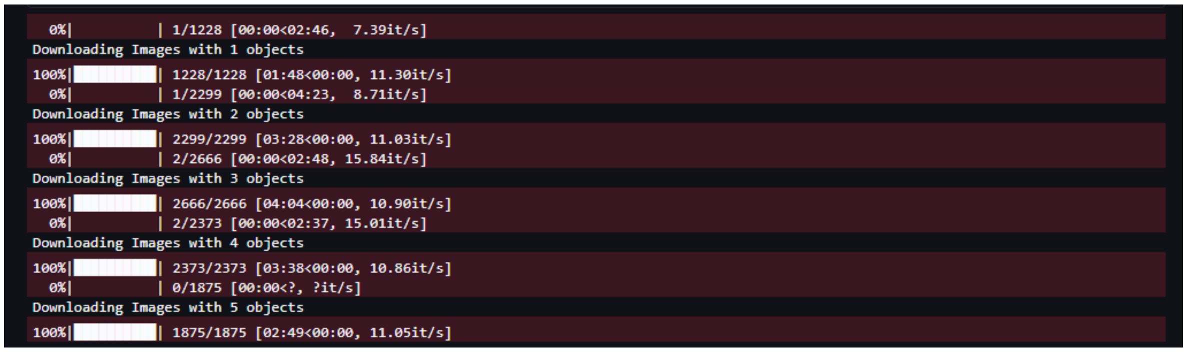
Overview

The dataset for developing this project is the Amazon Bin Image Dataset which contains 500.000 images of bins containing one or more products, it also has a metadata file for each image which contains information about the number of objects in the bin, their dimensions, and type. We need to work with a subset of this dataset to prevent any excess in SageMaker credit usage.

Access

I downloaded the dataset from the S3 bucket that AWS provided to us, I did this process using the function download_and_arrange_data() which was provided within the starter code of this project.

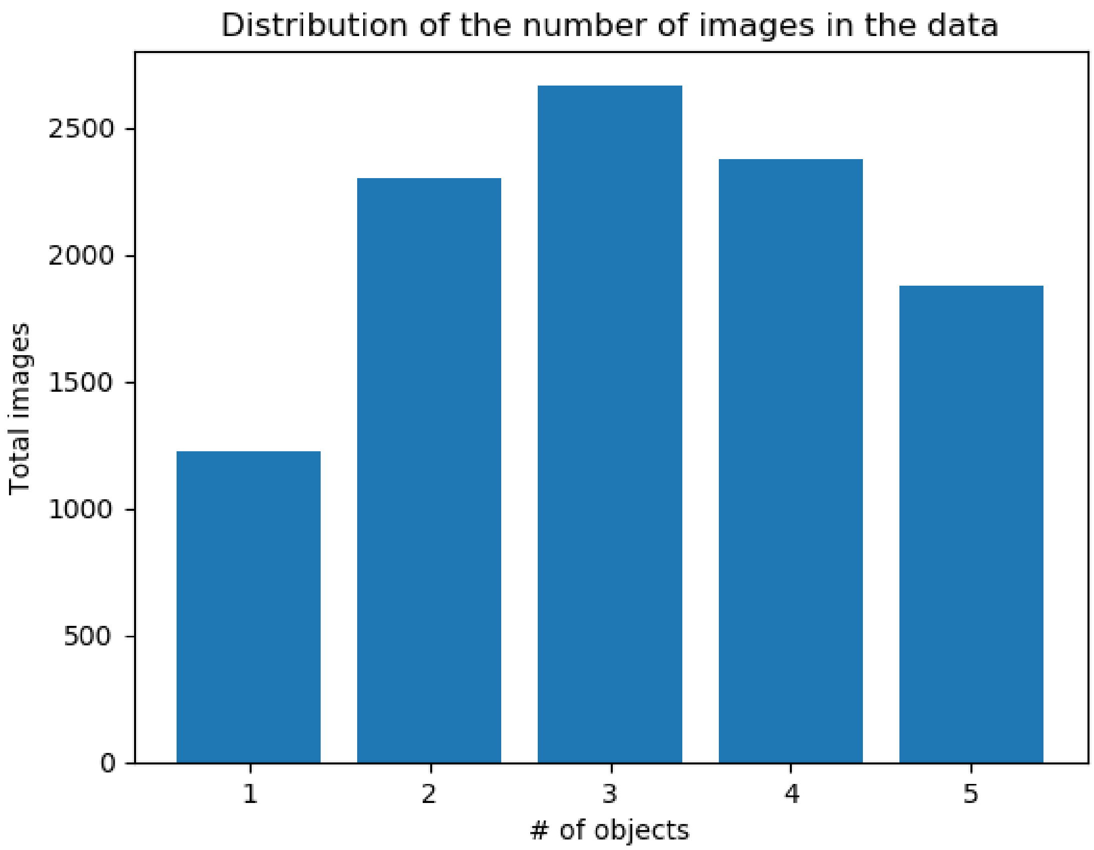
This function downloads a sample of the data and then organizes it within folders (1,2,3,4,5) indicating the number of objects for each image. In this moment, the data was stored in the SageMaker Studio's filesystem. Next, we can see the output of this step in the Notebook.



```
  0%|          | 1/1228 [00:00<02:46,  7.39it/s]
Downloading Images with 1 objects
100%|██████████| 1228/1228 [01:48<00:00, 11.30it/s]
  0%|          | 1/2299 [00:00<04:23,  8.71it/s]
Downloading Images with 2 objects
100%|██████████| 2299/2299 [03:28<00:00, 11.03it/s]
  0%|          | 2/2666 [00:00<02:48, 15.84it/s]
Downloading Images with 3 objects
100%|██████████| 2666/2666 [04:04<00:00, 10.90it/s]
  0%|          | 2/2373 [00:00<02:37, 15.01it/s]
Downloading Images with 4 objects
100%|██████████| 2373/2373 [03:38<00:00, 10.86it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
Downloading Images with 5 objects
100%|██████████| 1875/1875 [02:49<00:00, 11.05it/s]
```

Exploratory data

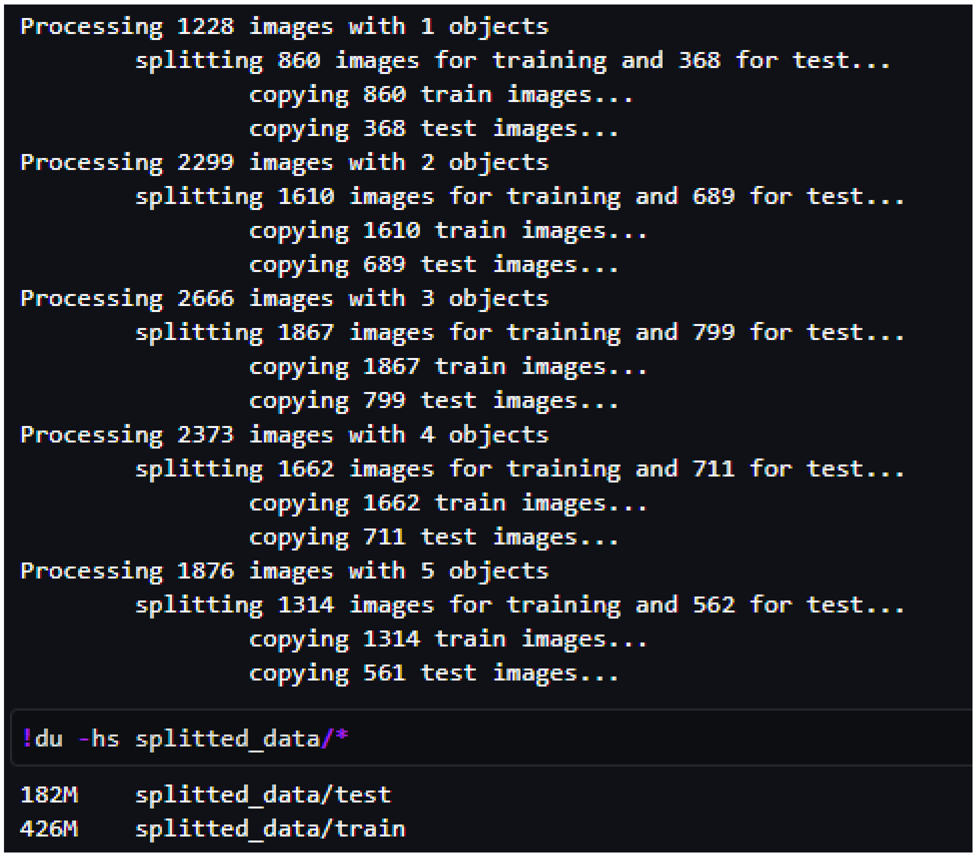
As data is composed of images with different object quantities, we can check the distribution of how many images we have for each possible quantity, i.e., 1 to 5. Next we can see a barplot showing the number of images according to the number of objects in the bin.



Machine Learning Pipeline

Data splitting (train and test)

At this moment, I randomly divided the data into train and test sets (70%, and 30% respectively). For this step, I implemented some code that creates a copy of the downloaded data with the difference that in this case, it is splitted into two folders “train” and “test”. In the next image, we can see how many images for each quantity we have left after this division, this is the output of the cell in charge of the underlying division.



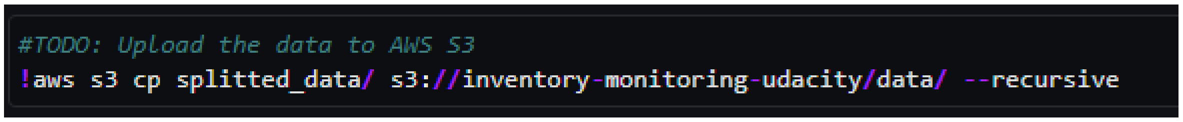
```
Processing 1228 images with 1 objects
        splitting 860 images for training and 368 for test...
                copying 860 train images...
                copying 368 test images...
Processing 2299 images with 2 objects
        splitting 1610 images for training and 689 for test...
                copying 1610 train images...
                copying 689 test images...
Processing 2666 images with 3 objects
        splitting 1867 images for training and 799 for test...
                copying 1867 train images...
                copying 799 test images...
Processing 2373 images with 4 objects
        splitting 1662 images for training and 711 for test...
                copying 1662 train images...
                copying 711 test images...
Processing 1876 images with 5 objects
        splitting 1314 images for training and 562 for test...
                copying 1314 train images...
                copying 561 test images...
```

```
!du -hs splitted_data/*
```

```
182M    splitted_data/test
426M    splitted_data/train
```

Data uploading to S3

In the next step, I uploaded the data to my S3 bucket using the AWS cli with the following command.



```
#TODO: Upload the data to AWS S3
!aws s3 cp splitted_data/ s3://inventory-monitoring-udacity/data/ --recursive
```

Model Training

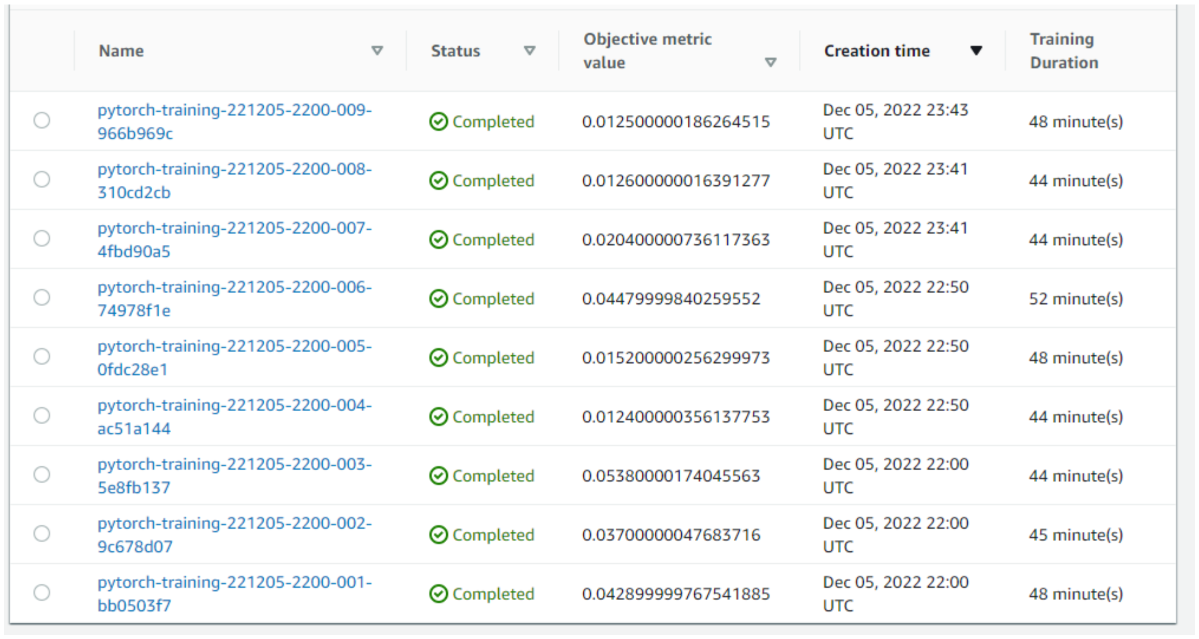
I used a ResNet18 pretrained model to get the advantage of the previously learned features, I add to the end of the NN a fully connected layer with 5 output neurons to use them as predictors for my 5 classes.

Writing training script (train.py)

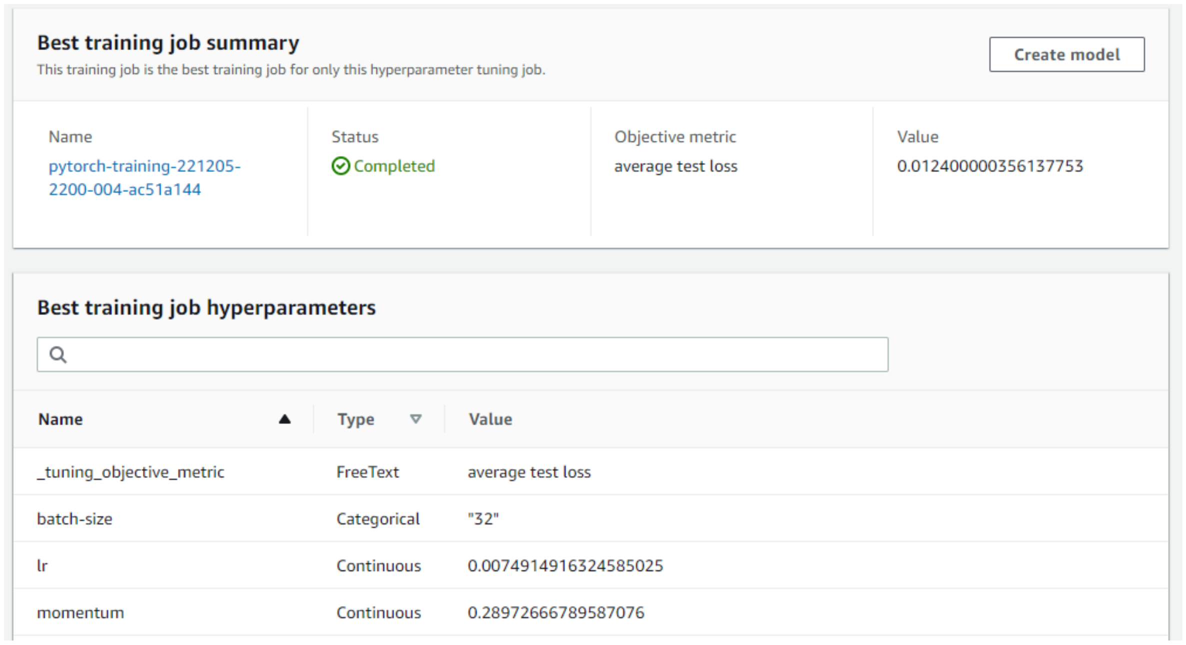
I wrote a PyTorch training code for the training loop, at the end of the script, the model is tested against the test data to know the accuracy of unseen observations, this accuracy is printed in the logs with a specific pattern so then it could be capture by the SageMaker's hyperparameter tuning job. At the end of the script, the model's artifact is saved to the model_dir directory, because of this, any of the trained models would be ready for deployment in the next phase.

Hyperparameter tunning

I executed a hyperparameter tuning job to find the best Learning rate, momentum, and batch-size. As a result, I got a model with 29% accuracy. In the following screenshots, we can see the output of the executed hyperparameter tuning job, the best job's hyperparameters, and the accuracy of the test data in the best model.



	Name	Status	Objective metric value	Creation time	Training Duration
○	pytorch-training-221205-2200-009-966b969c	Completed	0.012500000186264515	Dec 05, 2022 23:43 UTC	48 minute(s)
○	pytorch-training-221205-2200-008-310cd2cb	Completed	0.012600000016391277	Dec 05, 2022 23:41 UTC	44 minute(s)
○	pytorch-training-221205-2200-007-4fbd90a5	Completed	0.020400000736117363	Dec 05, 2022 23:41 UTC	44 minute(s)
○	pytorch-training-221205-2200-006-74978f1e	Completed	0.04479999840259552	Dec 05, 2022 22:50 UTC	52 minute(s)
○	pytorch-training-221205-2200-005-0fdc28e1	Completed	0.015200000256299973	Dec 05, 2022 22:50 UTC	48 minute(s)
○	pytorch-training-221205-2200-004-ac51a144	Completed	0.012400000356137753	Dec 05, 2022 22:50 UTC	44 minute(s)
○	pytorch-training-221205-2200-003-5e8fb137	Completed	0.05380000174045563	Dec 05, 2022 22:00 UTC	44 minute(s)
○	pytorch-training-221205-2200-002-9c678d07	Completed	0.03700000047683716	Dec 05, 2022 22:00 UTC	45 minute(s)
○	pytorch-training-221205-2200-001-bb0503f7	Completed	0.042899999767541885	Dec 05, 2022 22:00 UTC	48 minute(s)

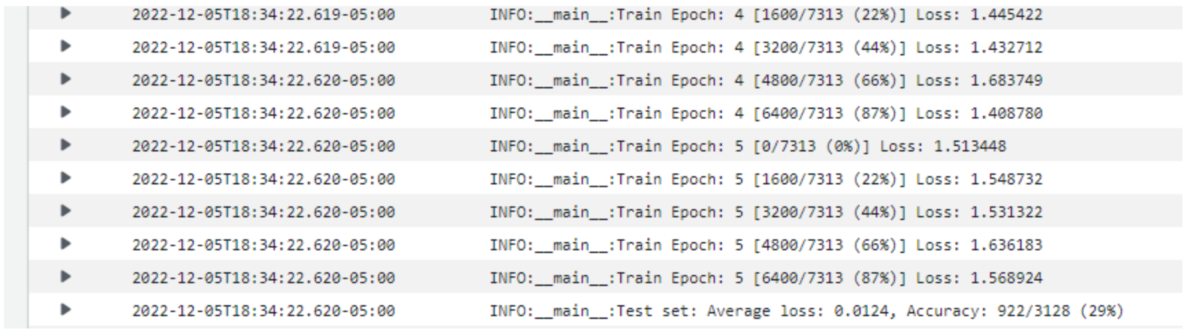

Best training job summary

This training job is the best training job for only this hyperparameter tuning job.

Name	Status	Objective metric	Value
pytorch-training-221205-2200-004-ac51a144	Completed	average test loss	0.012400000356137753

Best training job hyperparameters

Name	Type	Value
_tuning_objective_metric	FreeText	average test loss
batch-size	Categorical	"32"
lr	Continuous	0.0074914916324585025
momentum	Continuous	0.28972666789587076


```
2022-12-05T18:34:22.619-05:00    INFO:__main__:Train Epoch: 4 [1600/7313 (22%)] Loss: 1.445422
2022-12-05T18:34:22.619-05:00    INFO:__main__:Train Epoch: 4 [3200/7313 (44%)] Loss: 1.432712
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 4 [4800/7313 (66%)] Loss: 1.683749
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 4 [6400/7313 (87%)] Loss: 1.408780
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 5 [0/7313 (0%)] Loss: 1.513448
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 5 [1600/7313 (22%)] Loss: 1.548732
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 5 [3200/7313 (44%)] Loss: 1.531322
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 5 [4800/7313 (66%)] Loss: 1.636183
2022-12-05T18:34:22.620-05:00    INFO:__main__:Train Epoch: 5 [6400/7313 (87%)] Loss: 1.568924
2022-12-05T18:34:22.620-05:00    INFO:__main__:Test set: Average loss: 0.0124, Accuracy: 922/3128 (29%)
```

Deploying

Selecting the best model and deploying it

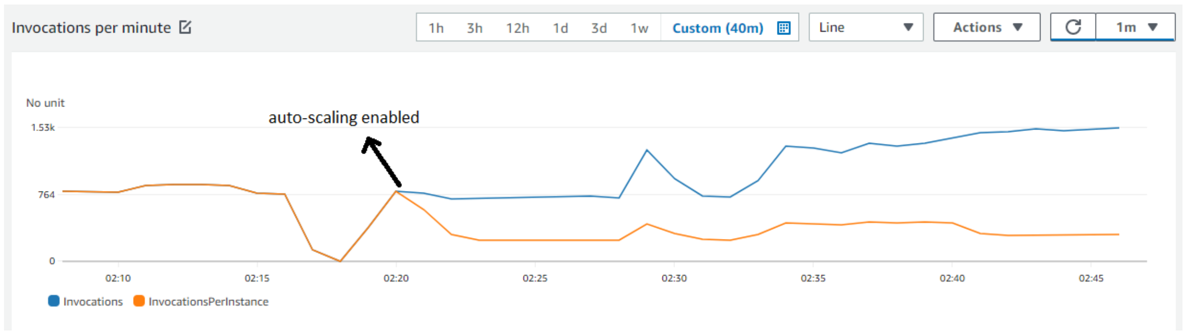
I selected the model with the best hyperparameters we saw in the previous screenshots, then, I deployed it using the sagemaker's PyTorchModel class, for doing this I had to specify the model's artifact S3 path in the parameter model_data. After this, I deployed the model in a ml.m5.large SageMaker instance. In the following sections we can check some tests I did to the deployed endpoint.

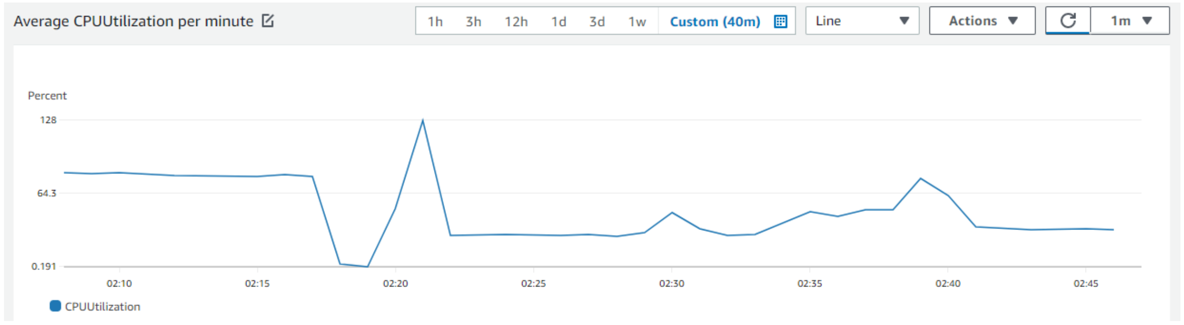
Running multiple predictions to stress the endpoint

I did a few tests to stress the endpoint with a lot of predictions, these tests were made using the first cell in the section "Running multiple predictions to stress the endpoint" and using the Jupyter Notebook "more-predictions.ipynb" from another SageMaker instance.

Calibrating endpoint's autoscaling capabilities

With the multiple sources of requests to the endpoint I started to change the endpoint auto-scaling parameters. I used the default auto-scaling policy “SageMakerVariantInvocationsPerInstance”, I set it to 50, and the maximum number of invoked variants was set to 3. I also used CloudWatch to observe the endpoint performance, the following images are graphs extracted from CloudWatch, in these, we can see how the capabilities to respond to multiple invocations start to grow when auto-scaling is enabled. A single endpoint was able to process about 770 invocations per minute before the auto-scaling was activated, after provisioning multiple variants, the endpoint was able to process about 1,500 invocations per minute, and we can see that the stress for individual instances was dramatically reduced, this kind calibration could reduce endpoint's latency and improve the solution throughput.





Conclusion

In this solution, we can observe a Machine Learning Pipeline including data preprocessing, model training with hyperparameter tuning, model deployment to an endpoint, and auto-scaling calibration capabilities for the endpoint. These are some of the most important steps in a Machine Learning Pipeline, but it is still a lot of improvements that can be done. For example, we can try a different model to improve the model accuracy, or we can pack the solution in a lambda function and explore the concurrency against the previously analyzed model’s auto-scaling capabilities.