AWS MACHINE LEARNING ENGINEER

INVENTORY MONITORING

Clément Palfroy

1) Definition

Domaine background

User experience and satisfaction are some of the main obsessions of contemporary retail/delivery companies. Any existing delivery company is expected to preserve, or better, to enhance the speed and consistency of their services.

Distribution centers can face large volumes of orders and strict time constraints to fulfill them. Those situations can lead to numerous errors, sometimes hard to notice. As a consequence, distribution centers need to be equipped with multiple means for visual/sensorial quality verification and, when required, restoration.

Due to the advancements in machine learning models and their promising results for labeling tasks, automated quality assessments (or anomaly detections) are getting increasingly adopted in retail/production companies (Redi et al., 2013). Images or weight/length measurements of the product are fed to a model in real-time. If the model sees an anomaly or its predictions are in a mismatch with the expected content, it would create an alarm so that the problem can be properly mitigated.

Problem statement

In this project, we addressed one of the problems Amazon fulfillment centers can face.

Amazon fulfillment centers use robots to carry clients' orders in bins. Each bin can contain multiple items. Occasionally, items are misplaced due to human errors, resulting in a mismatch: the recorded bin inventory, versus its actual content.

One basic error that could easily be noticed is if the count of every object instance in the bin is greater or inferior to the expected number.

In this project, we investigated how can computer vision/machine learning help recognize this error. In other words, we investigated how we can count the number of objects in a bin with a simple picture of its content.

Dataset

To complete this project, we used a subset of the Amazon Bin Image Dataset. This subset contains 10441 bins images of different sizes containing one or more objects. For each image, there is a metadata file containing information about the image like the number of objects, their dimension, and the type of object.

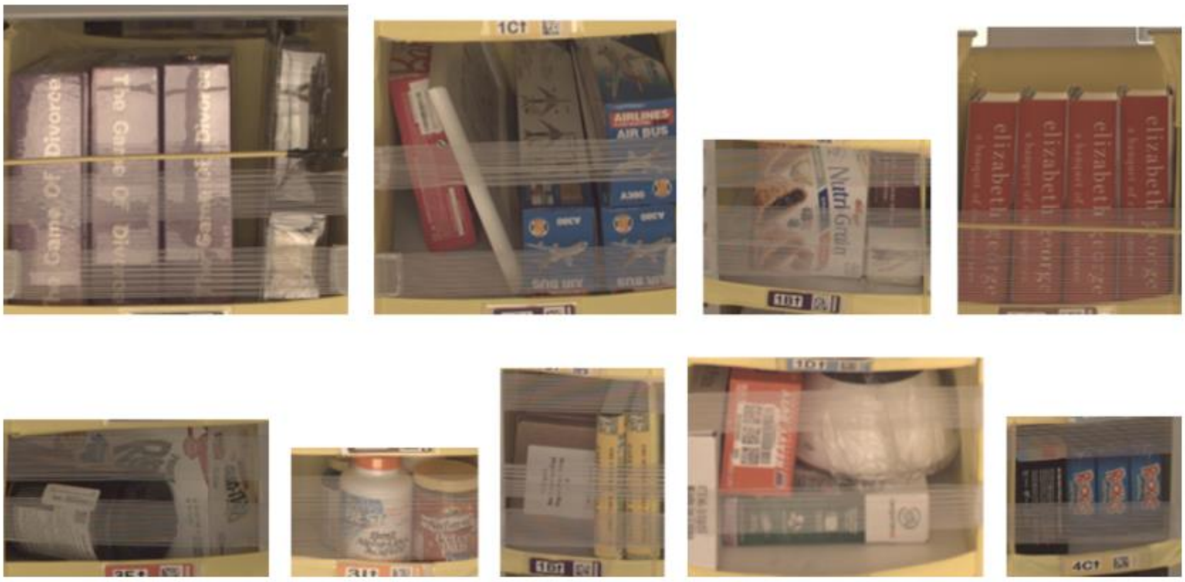


Figure 1: Sample images of the Amazon Bin Image Dataset.

The dataset is distributed within the following classes…

-1 object: 1228 images (~12%)

-2 objects: 2299 images (~22%)

-3 objects: 2666 images (~25%)

-4 objects: 2373 images (~23%)

-5 objects: 1875 images (~0.17%)

…which makes it relatively balanced.

Because our goal was to simply classify the images in one of the five categories, much of the image's metadata won't be used.

To look out and mitigate overfitting, the dataset was be split into train, validation, and test datasets (accounting for respectively 70/20/10%).

Metric

As we're dealing with a classification task that does not give special importance to false-positive or false-negative, we will focus exclusively on accuracy (on the validation or test datasets).

2) Analysis & Methodology

All the following work was performed within SageMaker studio. Different instances and kernel types were used depending on the tasks. Those parameters are specified in each section.

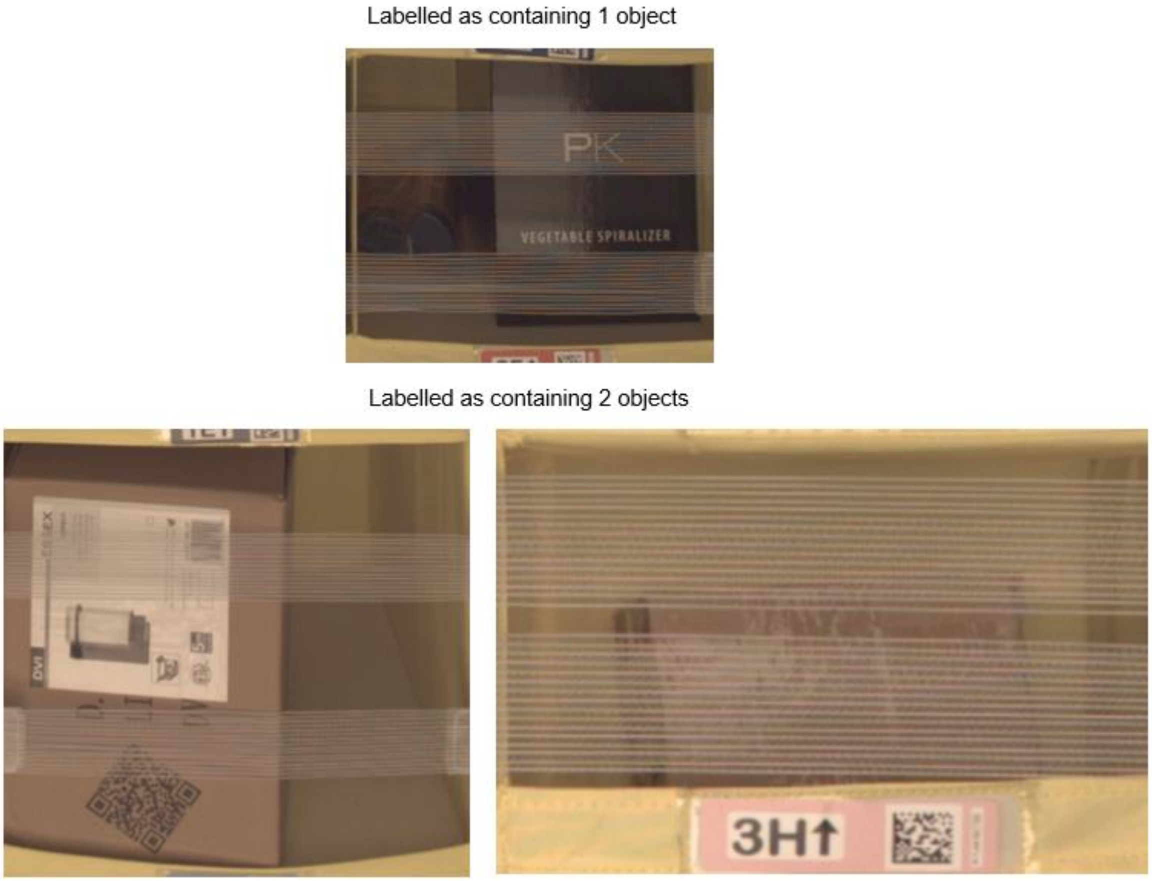
Benchmark

Other researchers/ML engineers already proposed solutions to the problem. In his approach, silverbottlep obtained a 55% accuracy score (on the test set) using a Resnet-34, trained from scratch. Silverbottlep's score will be chosen as a benchmark to assess our model performance.

Data preparation

The images are stored in a public S3 bucket which can be accessed via a list of addresses in a JSON format (provided by Udacity). A script was written in order to download the images using the JSON file while splitting them into a train, validation, and test dataset (respectively accounting for 70, 20, and 10% of the total images).

A quick inspection of the downloaded images was enough to spotlight some limitations. The following figure contains an image which was labeled "1", and two images which were labeled "2".


Labelled as containing 1 object

Labelled as containing 2 objects

Figure 2: Poor training samples.

As we can see, the bin supposed to contain only 1 object contains in reality 3 of them. Similarly, the bins supposed to contain 2 objects, seem to contain only 1. I see two possible explanations:

- The second object is hidden by the one in front.
- Human labeling error.

Either way, it will be extremely difficult (if not impossible) for any vision model to learn valuable insights from those pictures. Similarly, I find that images with 5 objects are extremely difficult to label. The objects tend to be squizzed and end up hiding each other. Because of the bin size and the way the pictures are taken, much important information is lost.

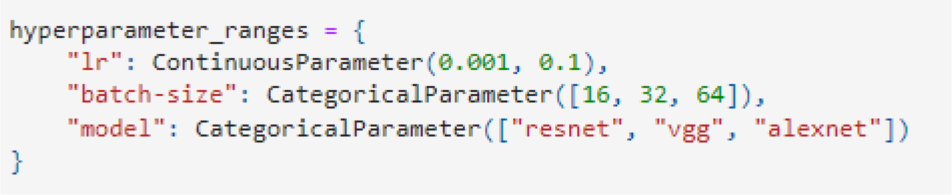
As it would be too laborious to go through every single image and delete those which appear falsely labeled, no data cleaning was performed.

Those limitations already hint at the difficulty to create a highly accurate model for this application.

Ultimately, the images were uploaded to the default S3 bucket associated with the SageMaker session. The code is available inside the "1) Data_preparation" notebook and was executed within an ml.t3.large with a Data science kernel.

Model choice and hyperparameter optimization (hpo)

Next, we investigated what could be the best combination of model architecture, batch size, and learning rate for our application. The following figure shows the ranges to which the parameters were allowed to variate.



```
hyperparameter_ranges = {
    "lr": ContinuousParameter(0.001, 0.1),
    "batch-size": CategoricalParameter([16, 32, 64]),
    "model": CategoricalParameter(["resnet", "vgg", "alexnet"])
}
```

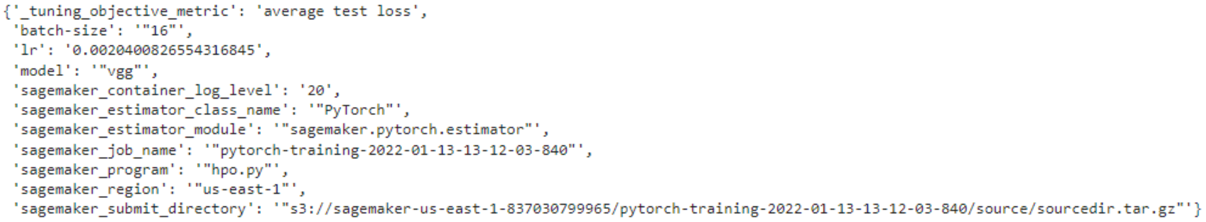
Figure 3: Hyperparameter ranges.

The SageMaker hyperparameter-tuner was used to launch 5 training jobs and search for the best hyperparameters using Bayesian optimization. More specifically, the algorithm was asked to minimize the average validation loss. Each job trained for 5 epochs.

The training jobs were performed using a Pytorch estimator and an ml.g4dn.xlarge instance. As GPU-equipped instances tend to be expensive, I tried to perform the hpo using spot instances (approximately 70% cheaper). However, in addition to a long waiting time, a few training jobs were stopped before completion. I suspect that GPU-equipped instances are in high demand and only a few spot versions are available at a given time. Ultimately, the hpo was performed again using regular instances.

To reduce training time, transfer learning (or finetuning) was used. The weights of the models were frozen except for the modified fully-connected layer with an output size equal to the number of classes.

The following figure shows the hyperparameters of the best training job.



```
{'_tuning_objective_metric': 'average test loss',
 'batch-size': '"16"',
 'lr': '0.0020400826554316845',
 'model': '"vgg"',
 'sagemaker_container_log_level': '20',
 'sagemaker_estimator_class_name': '"PyTorch"',
 'sagemaker_estimator_module': '"sagemaker.pytorch.estimator"',
 'sagemaker_job_name': '"pytorch-training-2022-01-13-13-12-03-840"',
 'sagemaker_program': '"hpo.py"',
 'sagemaker_region': '"us-east-1"',
 'sagemaker_submit_directory': '"s3://sagemaker-us-east-1-837030799965/pytorch-training-2022-01-13-13-12-03-840/source/sourcedir.tar.gz"'}
```

Figure 4: Best hyperparameters.

The lowest validation loss was achieved using a pre-trained Vgg network, and tuned using a 0.002 learning rate and a batch size of 16.

The code is available inside the "2) Model_choice & hpo" notebook and was executed within an ml.t3.medium with a Data science kernel. The hpo.py file contains the training script.

Model training - VGG

Next, the best model (and its associated hyperparameters) was trained for a longer period; 10 epochs.

While the model weights were pre-trained, they were not frozen: the whole model was trained.

The training loop was written to keep track of the best-performing weights and to save them at the end of training. This way, if the model starts overfitting after a certain number of epochs, it won't be a problem.

Sadly, the model did not manage to train and ended up stuck right at the start, at 24-25% accuracy. I see two possible reasons:

- The model architecture was not appropriate to capture the wide range of features that can define the number of objects in a bin.
- The model should have been trained from scratch. To use pre-trained weights might have stuck the model into a local minimum right at the start.

To mitigate those potential issues, another training was performed.

Model training – ResNet34

Silverbottlep obtained a 55% accuracy score using a Resnet-34, trained from scratch (using the full dataset ~500k images). To ensure we did not make any major mistakes in our training script, we trained our Resnet-34 without any pretraining.

At the end of 10 epochs, using the same hyperparameters as before (batch-size = 16, learning-rate = 0.002), the model managed to reach a 33.6% accuracy on the test set, which is way better than the Vgg model.

As a debugger configuration was attached to the Pytorch estimator, we can observe the training evolution closely:

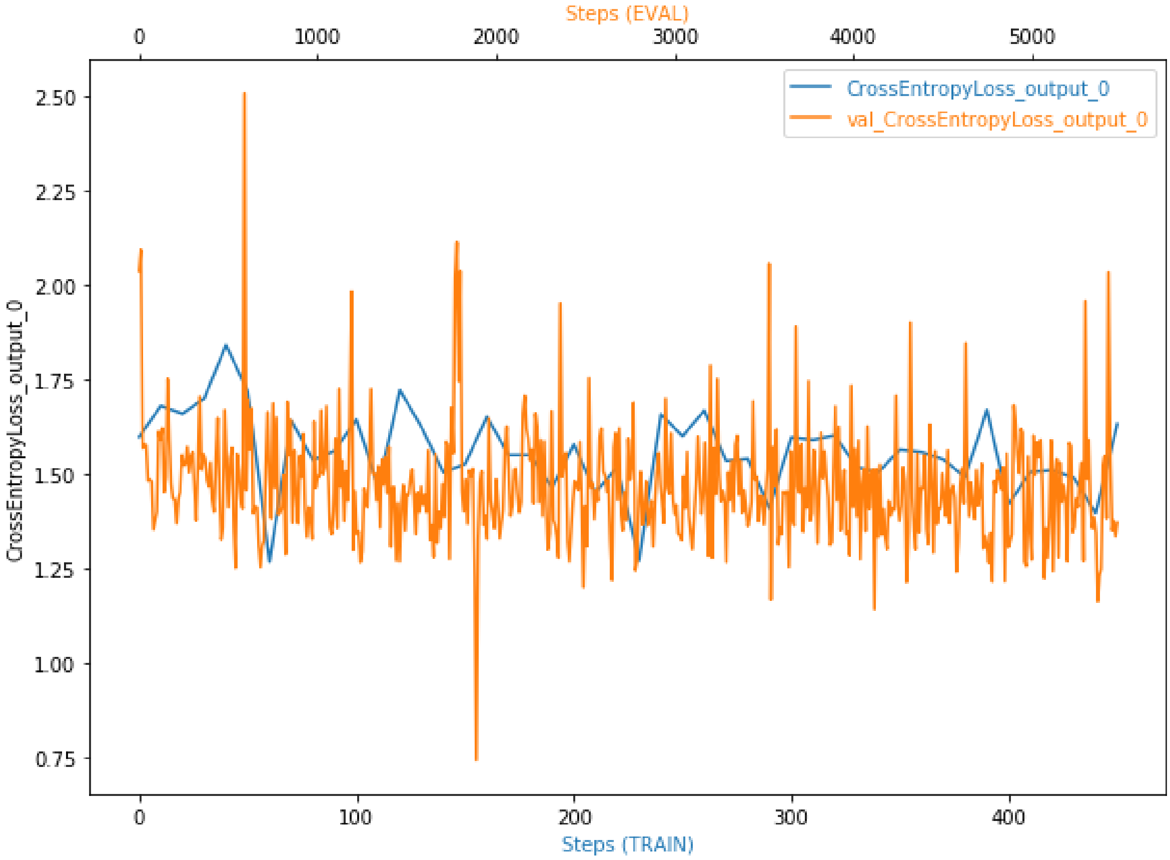


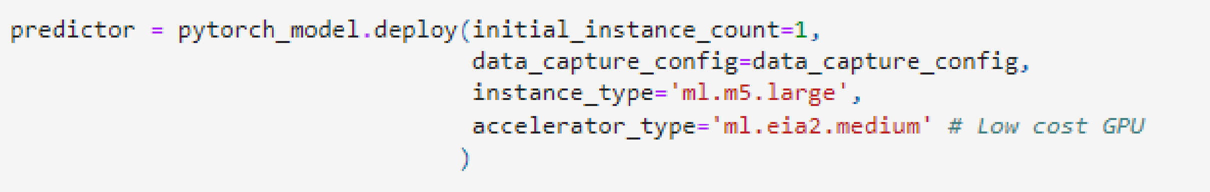
Figure 5: Evolution of the training and validation loss over the training steps.

As we can see the training was extremely noisy and slow. This is probably due to our small batch-size/learning-rate. As reference, silverbottlep, used a batch-size of 128 and a learning rate of 0.1. Given more time and credits, the next step would have probably be to investigate different batch-size/learning-rate and train the model on more epochs.

Model deployment

Next, the model was made accessible via a deployed endpoint.

Vision models tend to contain many weights and be computationally expensive to use. Elastic Inference was used to optimize inference speed cost-effectively. A low-cost GPU-powered acceleration was attached to the deployed EC2 instance. This configuration tends to reduce costs up to 75% compared to traditional GPU instances.



```
predictor = pytorch_model.deploy(initial_instance_count=1,
                                 data_capture_config=data_capture_config,
                                 instance_type='ml.m5.large',
                                 accelerator_type='ml.eia2.medium' # Low cost GPU
                                )
```

Figure 6: Endpoint instance and accelerator types.

A data capture configuration was attached to the model to save inference information (input, output, and other metadata). It helps keep track of the model prediction's confidence over time.

To reduce potential latency, a custom scaling policy was set up. Up to 3 instances can be instantiated to meet demand based on CPU usage. More specifically, if an endpoint has an average CPU utilization of more than 70% for more than 30sc, another endpoint will be deployed. This policy was implemented following this documentation.

A personalized inference script was attached to the endpoint to authorize 2 types of input. The image as bytes, or a link of the S3 location where the image is stored.

For practice purposes, a lambda function was created as an intermediate between the endpoint and the potential API that might need it. It works the following way:

1) We invoke the lambda function and send the S3 URL as payload.

2) The function invokes the endpoint using the URL as input.

3) The endpoint downloads the image, performance inference, and returns the prediction to the lambda function.

4) The prediction is returned by the lambda function.

As a sanity check, predictions were made from the SageMaker notebook (sending the image bytes directly), and by invoking the lambda function. Both worked perfectly!

The model deployment code and prediction tests are inside the "OPTIONAL Model_deployement" notebook.

SageMaker pipeline

Let's now imagine a scenario: Amazon is using our deployed endpoint, and as time goes by, we receive new labeled images inside our training dataset.

It would be nice to be able to automatically train a model with the extra data, and if it performs better than the one currently deployed, update the endpoint with the new model artifact.

A SageMaker pipeline was created to automate model training, assessment, and endpoint update. Also slightly deprecated, this documentation helped a lot. Here is a schematic of the created pipeline.

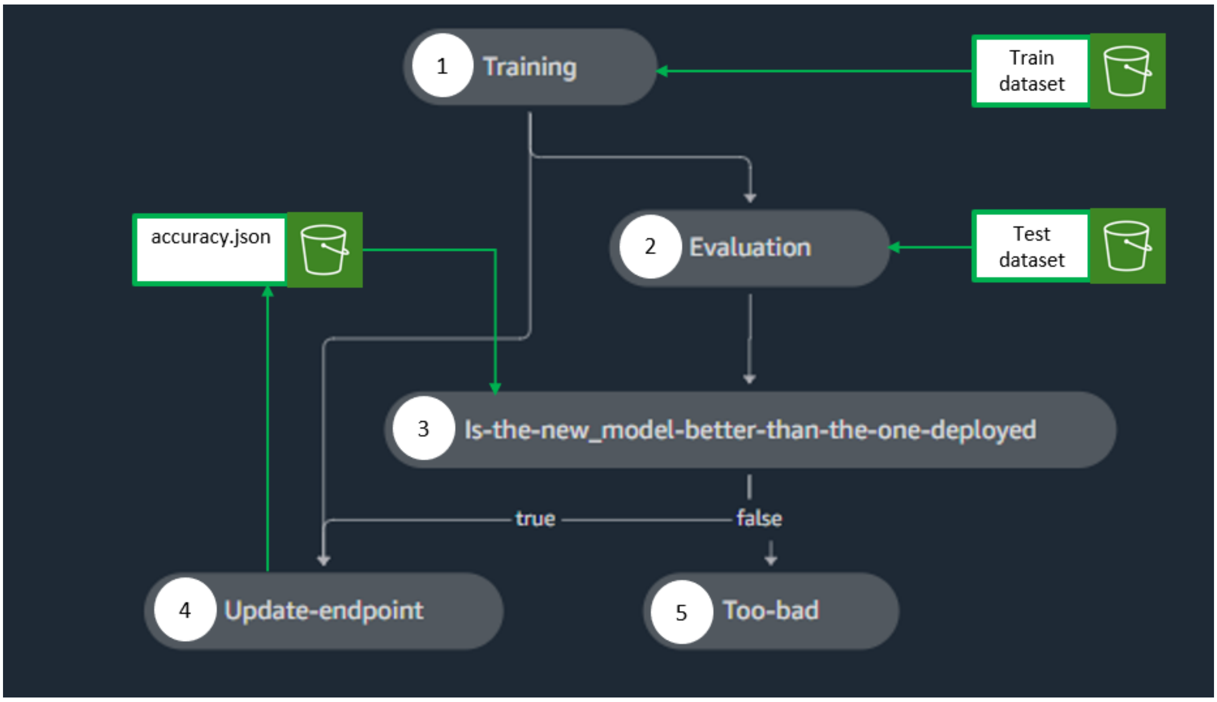


Figure 7: SageMaker pipeline schematic.

Step 1) The pipeline will start with a training step. A resnet34 will be trained with the dataset inside a specific S3 bucket. The model artifact is automatically stored in S3 and its address is passed to the second step.

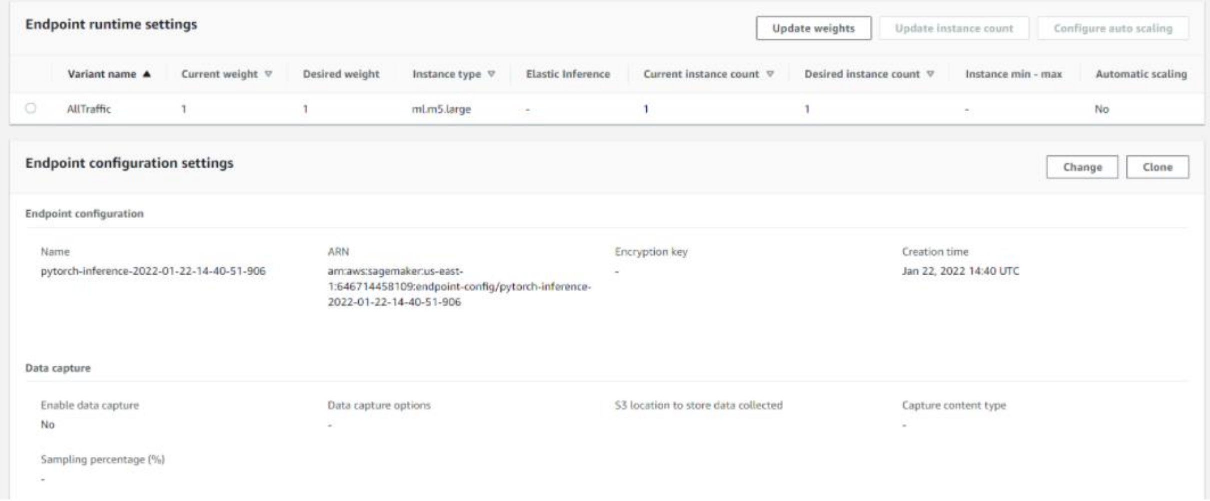
Step 2) Then come the evaluation step, the model artifact and the test dataset are downloaded from S3 to determine the accuracy. The result is stored in a JSON file named "current_accuracy" and directly passed to the next step.

Step 3) A condition step (of type "GreaterThan") will download from S3 the "accuracy.json" file (containing the accuracy of the deployed model) and receive the accuracy previously obtained. If the accuracy of the deployed model is greater, nothing will happen (step 5). If it is lesser, the last step is triggered.

Step 4) Within the last step, 2 main things will happen:

- First, an endpoint configuration will be created. This configuration will point to the new model artifact, use Elastic Inference, and have a data capture configuration. The deployed endpoint is then updated with the new configuration. Updating an endpoint permits to keep its name (which is useful when others services like lambda functions rely on it) and ensures continuity of service.
- Last, the accuracy.json file is updated to contain the score of the new model.

Let's now run a test! As shown in the following figure, I first deployed an old model on an ml.m5.large instance (no Elastic Inference nor data capture).



Endpoint runtime settings

	Variant name ▲	Current weight ▽	Desired weight	Instance type ▽	Elastic Inference	Current instance count ▽	Desired instance count ▽	Instance min - max	Automatic scaling
○	AllTraffic	1	1	ml.m5.large	-	1	1	-	No

Endpoint configuration settings

Endpoint configuration

Name	ARN	Encryption key	Creation time
pytorch-inference-2022-01-22-14-40-51-906	arn:aws:sagemaker:us-east-1:646714458109:endpoint-config/pytorch-inference-2022-01-22-14-40-51-906	-	Jan 22, 2022 14:40 UTC

Data capture

Enable data capture	Data capture options	S3 location to store data collected	Capture content type
No	-		-

Sampling percentage (%)
-

Figure 8: Current endpoint settings.

Then, the pipeline was executed.

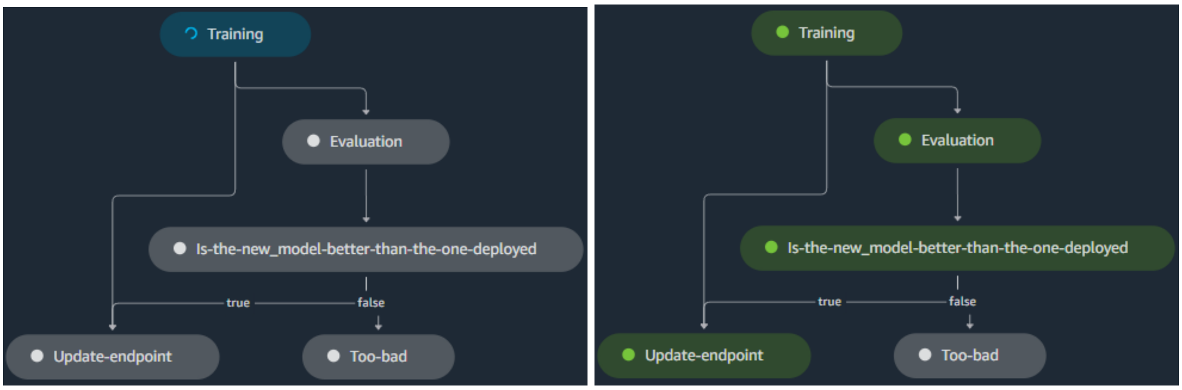
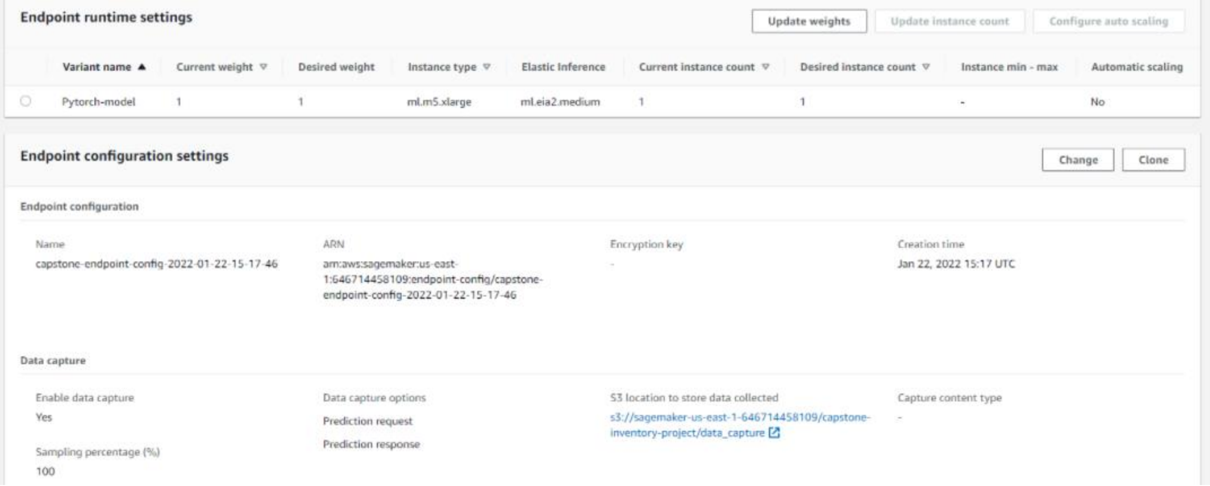


Figure 9: Left: Pipeline start. Right: Pipeline success.

The accuracy.json file was created with an initial value of 0.1, as the trained model reached 32% accuracy, the update-endpoint step was triggered.

As shown in the following figure, the endpoint has been correctly updated.



Endpoint runtime settings

	Variant name ▲	Current weight ▽	Desired weight	Instance type ▽	Elastic Inference	Current instance count ▽	Desired instance count ▽	Instance min - max	Automatic scaling
○	Pytorch-model	1	1	ml.m5.xlarge	ml.eia2.medium	1	1	-	No

Endpoint configuration settings

Endpoint configuration

Name
capstone-endpoint-config-2022-01-22-15-17-46

ARN
arn:aws:sagemaker:us-east-1:646714458109:endpoint-config/capstone-endpoint-config-2022-01-22-15-17-46

Encryption key
-

Creation time
Jan 22, 2022 15:17 UTC

Data capture

Enable data capture
Yes

Sampling percentage (%)
100

Data capture options
Prediction request
Prediction response

S3 location to store data collected
s3://sagemaker-us-east-1-646714458109/capstone-inventory-project/data_capture

Capture content type
-

Figure 10: Updated endpoint settings.

Before executing the pipeline a second time, I manually deleted a few pictures from the training set. Naturally, the trained model performed below 32% accuracy and nothing happened.

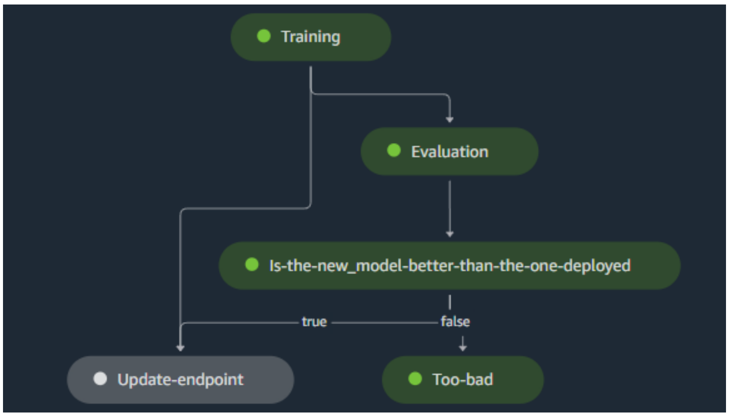


Figure 11: Pipeline fail.

As a last sanity check, the endpoint was invoked via the lambda function again, everything worked fine and a prediction was returned!

Results

To summarize, we obtained a 34% accuracy score when training for 10 epochs with 10k images, and other researchers obtained a 55% accuracy score when training for 25 epochs on 500k images (they started overfitting after that).

Those results are extremely poor and nowhere near what would be required to use them in a production environment.

While a few percentages could be gained by investigating other models and hyperparameters, I am sure it would remain insufficient.

Many 4/5 objects images are impossible to label correctly as the objects are squizzed and end up hiding each other. Another limitation is that the images had different sizes. Every image was reduced to a 224 X 224 format to pass through the neural net, for some, it might have resulted in cropping important features.

It would be fair to assume that in a real situation the camera used to take pictures of the bin will be the same across the preparation centers, or that at least, the image format will be standardized.

However, I do believe that computer vision has very little potential in solving reliably this issue. I would rather consider a precise weighing machine combined with a list of all the product weights (just like in supermarkets).

Nonetheless, it has been a very good opportunity to become pro-efficient with SageMaker and its different tools. Pipelines that can automatically investigate the potential of new model/data and update the deployed endpoint if required are very cost-effective tools for any AI-based application.

Redi, J., Gastaldo, P., & zunino, r. (2013). Supporting visual quality assessment with machine learning. *EURASIP Journal on Image and Video Processing, 2013*, 1-15. https://doi.org/10.1186/1687-5281-2013-54