Capstone Project Proposal for
Inventory Monitoring at Distribution Centers

1. Domain Background

Inventory monitoring at distribution centers is a crucial task for businesses that deal with many products. It involves tracking the number of items in each bin and ensuring that the correct number of items are delivered to customers. This task is often performed using robots that move objects in bins. Machine learning can be used to automate this process and make it more efficient. In this project, we will apply what we learned and use AWS SageMaker and good machine learning engineering practices to build a model that can count the number of objects in each bin.

2. Problem Statement

Inventory monitoring at distribution centers is currently facing significant challenges due to outdated monitoring systems and lack of real-time data. These issues have led to an average increase of 15% in warehousing costs and a 20% decrease in order fulfillment.

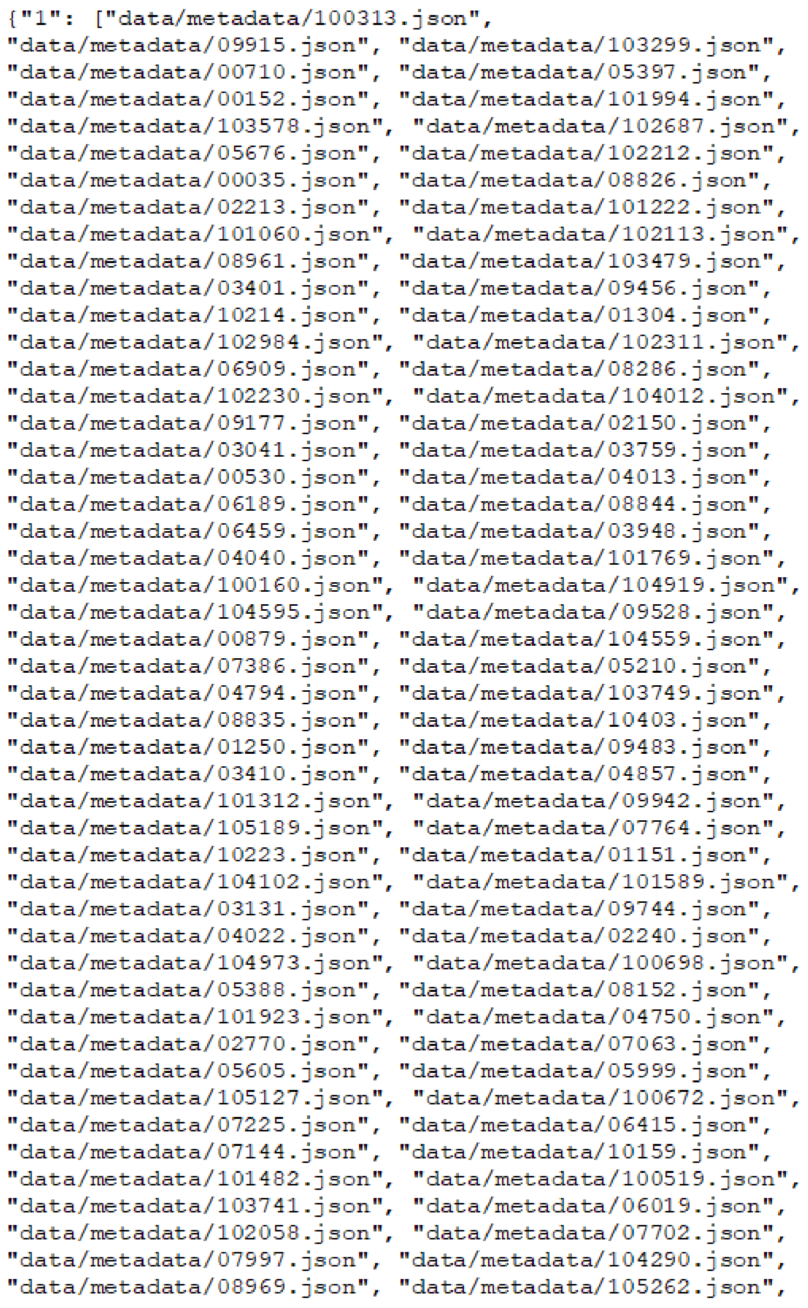
Distribution centers often use robots to move objects as a part of their operations. Objects are carried in bins which can contain multiple objects. In this project, we will have to build a model that can count the number of objects in each bin. A system like this can be used to track inventory and make sure that delivery consignments have the correct number of items.

3. Datasets and Inputs

The dataset we will use is the Amazon Bin Image Dataset which contains over 500,000 images and metadata from bins of a pod in an operating Amazon Fulfillment

Center [1]. The bin images in this dataset are captured as robot units carrying pods as part of normal Amazon Fulfillment Center operations.

Since the dataset is large, we were provided a subset of it to use in this project. Our data are 5 folders named 1 to 5, each representing the number of objects in each image in that folder.



```
{"1": ["data/metadata/100313.json",
"data/metadata/09915.json", "data/metadata/103299.json",
"data/metadata/00710.json", "data/metadata/05397.json",
"data/metadata/00152.json", "data/metadata/101994.json",
"data/metadata/103578.json", "data/metadata/102687.json",
"data/metadata/05676.json", "data/metadata/102212.json",
"data/metadata/00035.json", "data/metadata/08826.json",
"data/metadata/02213.json", "data/metadata/101222.json",
"data/metadata/101060.json", "data/metadata/102113.json",
"data/metadata/08961.json", "data/metadata/103479.json",
"data/metadata/03401.json", "data/metadata/09456.json",
"data/metadata/10214.json", "data/metadata/01304.json",
"data/metadata/102984.json", "data/metadata/102311.json",
"data/metadata/06909.json", "data/metadata/08286.json",
"data/metadata/102230.json", "data/metadata/104012.json",
"data/metadata/09177.json", "data/metadata/02150.json",
"data/metadata/03041.json", "data/metadata/03759.json",
"data/metadata/00530.json", "data/metadata/04013.json",
"data/metadata/06189.json", "data/metadata/08844.json",
"data/metadata/06459.json", "data/metadata/03948.json",
"data/metadata/04040.json", "data/metadata/101769.json",
"data/metadata/100160.json", "data/metadata/104919.json",
"data/metadata/104595.json", "data/metadata/09528.json",
"data/metadata/00879.json", "data/metadata/104559.json",
"data/metadata/07386.json", "data/metadata/05210.json",
"data/metadata/04794.json", "data/metadata/103749.json",
"data/metadata/08835.json", "data/metadata/10403.json",
"data/metadata/01250.json", "data/metadata/09483.json",
"data/metadata/03410.json", "data/metadata/04857.json",
"data/metadata/101312.json", "data/metadata/09942.json",
"data/metadata/105189.json", "data/metadata/07764.json",
"data/metadata/10223.json", "data/metadata/01151.json",
"data/metadata/104102.json", "data/metadata/101589.json",
"data/metadata/03131.json", "data/metadata/09744.json",
"data/metadata/04022.json", "data/metadata/02240.json",
"data/metadata/104973.json", "data/metadata/100698.json",
"data/metadata/05388.json", "data/metadata/08152.json",
"data/metadata/101923.json", "data/metadata/04750.json",
"data/metadata/02770.json", "data/metadata/07063.json",
"data/metadata/05605.json", "data/metadata/05999.json",
"data/metadata/105127.json", "data/metadata/100672.json",
"data/metadata/07225.json", "data/metadata/06415.json",
"data/metadata/07144.json", "data/metadata/10159.json",
"data/metadata/101482.json", "data/metadata/100519.json",
"data/metadata/103741.json", "data/metadata/06019.json",
"data/metadata/102058.json", "data/metadata/07702.json",
"data/metadata/07997.json", "data/metadata/104290.json",
"data/metadata/08969.json", "data/metadata/105262.json",
```

file_list.json

4. Solution Statement

To come up with a solution for our problem, I will utilize the State-of-the-Art techniques of Computer Vision tasks to do the multi-class image classification. I will use transfer learning and a pre-trained convolutional neural network to solve our problem.

I chose a pre-trained Resnet CNN and used an Adam optimizer for the training job.

5. Benchmark Model

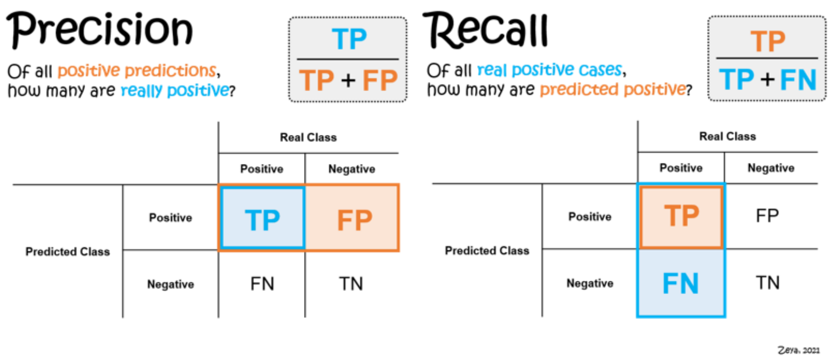
I first trained a pre-trained Resnet as a benchmark model on random weights and batch_size = 64 and got an accuracy of 16%

6. Evaluation Metrics

I used accuracy metrics to evaluate the overall model performance.

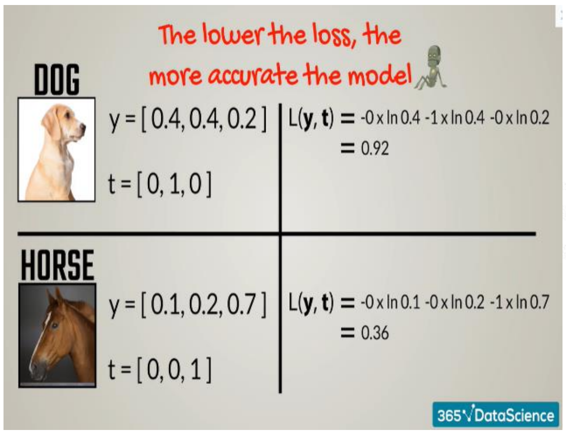
In a future update, I am going to use classification report or confusion matrix to best describe the performance of our classification model.

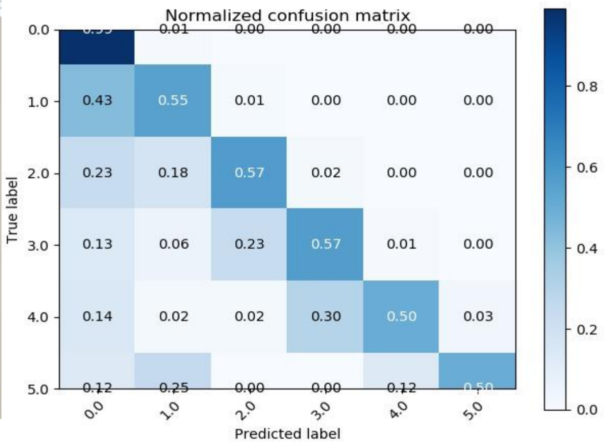
As the report includes f1-score, precision and recall and the mean overall accuracy of the model, we can evaluate our model accordingly.

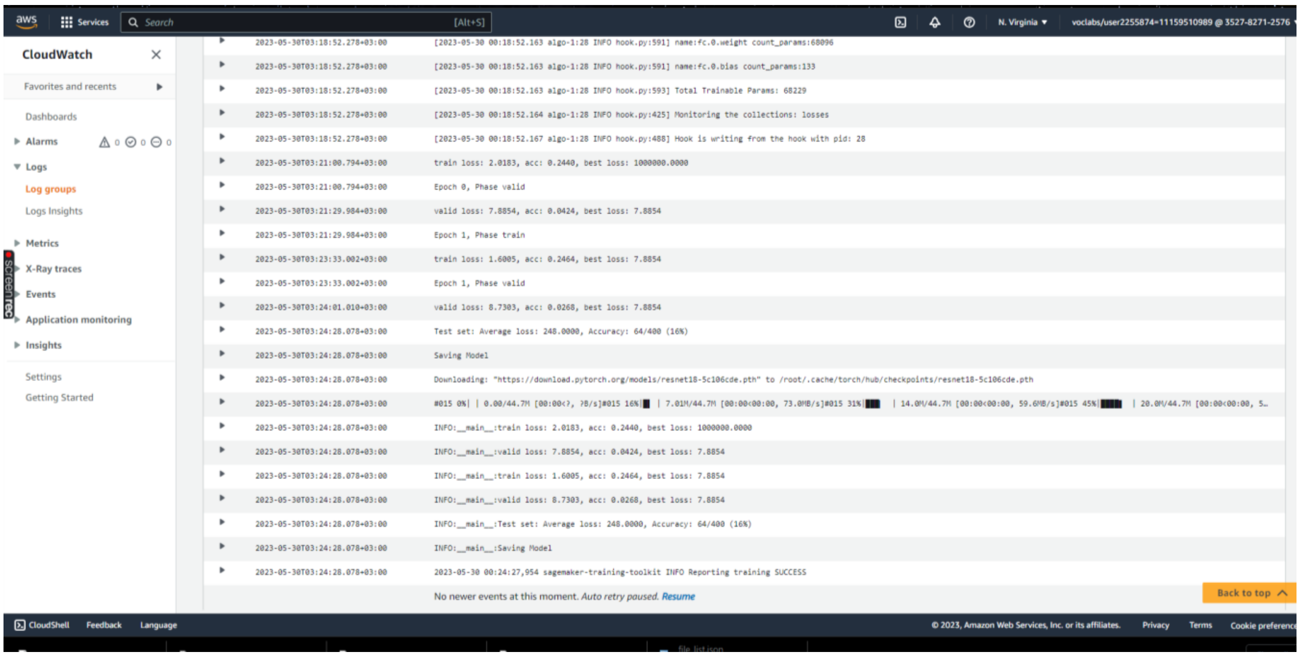


A confusion matrix is a table that is used to define the performance of a classification algorithm. A confusion matrix visualizes and summarizes the performance of a classification algorithm.

After training my ML model I tried to use the standard *Cross Entropy Loss function* as my metric to study my training process results along with hyperparameter tuning. The reason for this is that *Cross-Entropy* is to take the output probabilities (P) and measure the distance from the truth values (as shown in Figure below).


The lower the loss, the more accurate the model
DOG
$y = [0.4, 0.4, 0.2]$
$t = [0, 1, 0]$
$L(\mathbf{y}, \mathbf{t}) = -0 \times \ln 0.4 - 1 \times \ln 0.4 - 0 \times \ln 0.2 = 0.92$
HORSE
$y = [0.1, 0.2, 0.7]$
$t = [0, 0, 1]$
$L(\mathbf{y}, \mathbf{t}) = -0 \times \ln 0.1 - 0 \times \ln 0.2 - 1 \times \ln 0.7 = 0.36$
365√DataScience





```
2023-05-30T03:18:52.278+03:00    [2023-05-30 00:18:52.163 algo-1:28 INFO hook.py:591] name:fc.0.weight count_params:68096
2023-05-30T03:18:52.278+03:00    [2023-05-30 00:18:52.163 algo-1:28 INFO hook.py:591] name:fc.0.bias count_params:133
2023-05-30T03:18:52.278+03:00    [2023-05-30 00:18:52.163 algo-1:28 INFO hook.py:593] Total Trainable Params: 68229
2023-05-30T03:18:52.278+03:00    [2023-05-30 00:18:52.164 algo-1:28 INFO hook.py:425] Monitoring the collections: losses
2023-05-30T03:18:52.278+03:00    [2023-05-30 00:18:52.167 algo-1:28 INFO hook.py:488] Hook is writing from the hook with pid: 28
2023-05-30T03:21:00.794+03:00    train loss: 2.0183, acc: 0.2440, best loss: 1000000.0000
2023-05-30T03:21:00.794+03:00    Epoch 0, Phase valid
2023-05-30T03:21:29.984+03:00    valid loss: 7.8854, acc: 0.0424, best loss: 7.8854
2023-05-30T03:21:29.984+03:00    Epoch 1, Phase train
2023-05-30T03:23:33.002+03:00    train loss: 1.6005, acc: 0.2464, best loss: 7.8854
2023-05-30T03:23:33.002+03:00    Epoch 1, Phase valid
2023-05-30T03:24:01.010+03:00    valid loss: 8.7303, acc: 0.0268, best loss: 7.8854
2023-05-30T03:24:28.078+03:00    Test set: Average loss: 248.0000, Accuracy: 64/400 (16%)
2023-05-30T03:24:28.078+03:00    Saving Model
2023-05-30T03:24:28.078+03:00    Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.cache/torch/hub/checkpoints/resnet18-5c106cde.pth
2023-05-30T03:24:28.078+03:00    #015 0%| | 0.00/44.7M [00:00<?, ?B/s]#015 16%|█ | 7.01M/44.7M [00:00<00:00, 73.0MB/s]#015 31%|███  | 14.0M/44.7M [00:00<00:00, 59.6MB/s]#015 45%|████ | 20.0M/44.7M [00:00<00:00, 5…
2023-05-30T03:24:28.078+03:00    INFO:__main__:train loss: 2.0183, acc: 0.2440, best loss: 1000000.0000
2023-05-30T03:24:28.078+03:00    INFO:__main__:valid loss: 7.8854, acc: 0.0424, best loss: 7.8854
2023-05-30T03:24:28.078+03:00    INFO:__main__:train loss: 1.6005, acc: 0.2464, best loss: 7.8854
2023-05-30T03:24:28.078+03:00    INFO:__main__:valid loss: 8.7303, acc: 0.0268, best loss: 7.8854
2023-05-30T03:24:28.078+03:00    INFO:__main__:Test set: Average loss: 248.0000, Accuracy: 64/400 (16%)
2023-05-30T03:24:28.078+03:00    INFO:__main__:Saving Model
2023-05-30T03:24:28.078+03:00    2023-05-30 00:24:27,954 sagemaker-training-toolkit INFO Reporting training SUCCESS
                                 No newer events at this moment. Auto retry paused. Resume
```

Logs and model evaluation metrics

7. Implementation

I thought of many ways to tackle this problem. The first notebook I worked on was 'sagemaker.ipynb' I did all the data extraction and saving there. Then, built the first benchmark model.

Then after getting my project review I different things in another notebook 'EDA_and_evaluation.ipynb'

i. Data preparation:

- Download a subset of the data.
- Preprocess and clean the data.
- Visualization and EDA
- Upload train, test, and validation files to an S3 bucket so SageMaker use them for training.

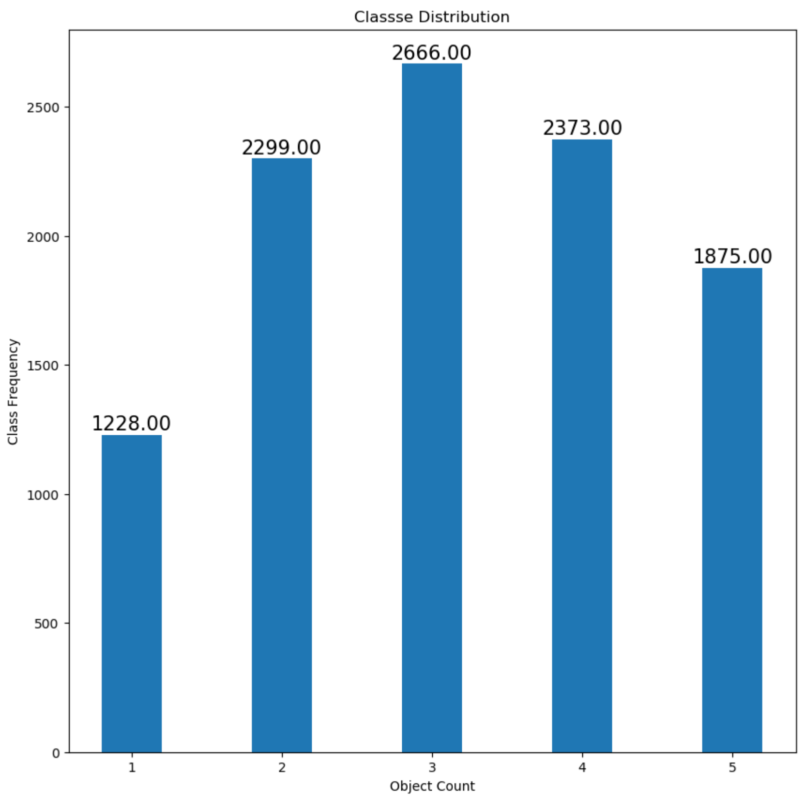
After downloading the dataset subset, the main folder was the train_data folder which included 5 folders for each class. To train the model, I needed to do some Extract-Transform-Load processes to split the data into train, validation and test data folders which I uploaded into S3 for SageMaker to use in the training job.

- From this subset, I took a Train-Test-Validation Split as follows:
- 1. I randomly sample 80 Images from each class as Test Data
- 2. For the remaining data, I did a 80-20 Train-Validation Stratified Split.
- Stratification is important to preserve the same distributions in both sets.

Visualizing and exploring the data

This is a random visualization of our data which shows how the images of the bin containers are not recognizable enough even to us humans which will really affect the model.

Also, another important visualization is the classes distribution which shows the big imbalance in data classes which might cause the model to have a bias for the third class.

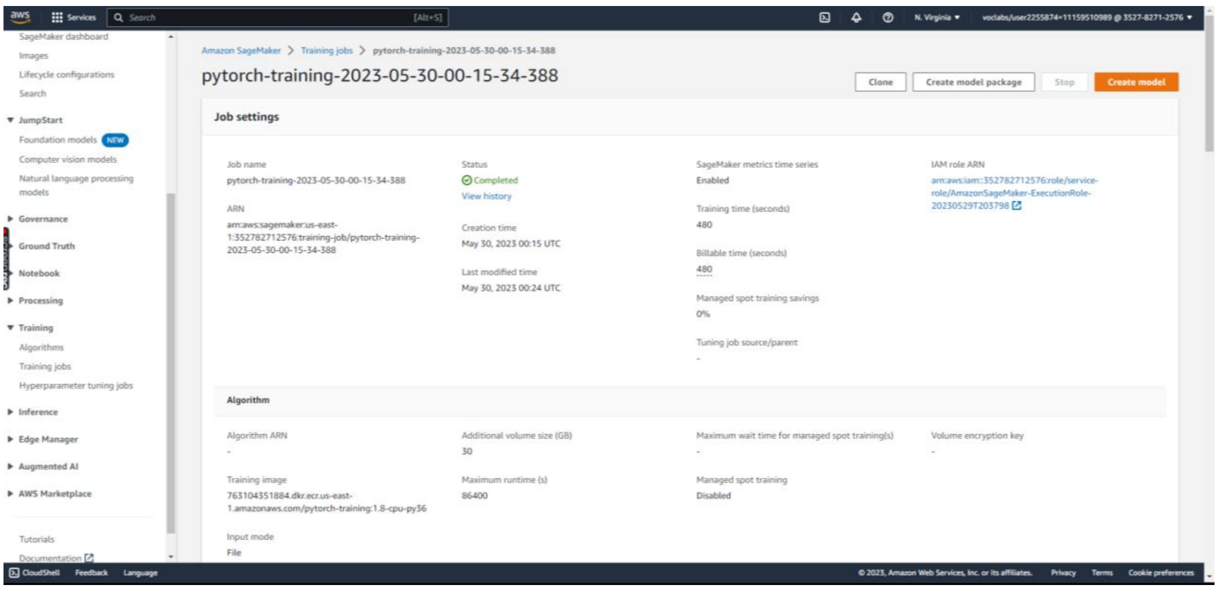


ii. Write Model Training script:

- Read, load, and preprocess our training, testing and validation data.
- Choosing the pre-trained model, optimizer and loss function we will be using.

iii. Train our CNN Model with SageMaker:

- Define our hyperparameters, instances type and count for training.


pytorch-training-2023-05-30-00-15-34-388

Job settings

Job name	pytorch-training-2023-05-30-00-15-34-388
ARN	arn:aws:sagemaker:us-east-1:352782712576:training-job/pytorch-training-2023-05-30-00-15-34-388
Status	Completed
Creation time	May 30, 2023 00:15 UTC
Last modified time	May 30, 2023 00:24 UTC
SageMaker metrics time series	Enabled
Training time (seconds)	480
Billable time (seconds)	480
Managed spot training savings	0%
Tuning job source/parent	-
IAM role ARN	arn:aws:iam::352782712576:role/service-role/AmazonSageMaker-ExecutionRole-20230529T203798

Algorithm

Algorithm ARN	-
Training image	763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.8-cpu-py36
Input mode	File
Additional volume size (GB)	30
Maximum runtime (s)	86400
Maximum wait time for managed spot training(s)	-
Managed spot training	Disabled
Volume encryption key	-

SageMaker Training Job parameters

iv. ***Profiler-Report***

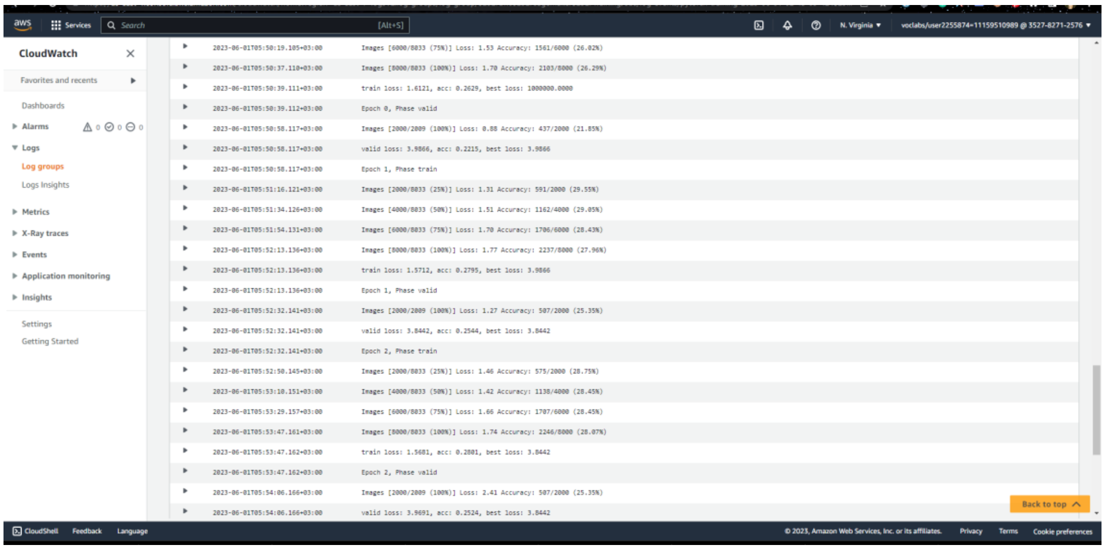
Rules summary

The following table shows a profiling summary of the Debugger built-in rules. The table is sorted by the rules that triggered the most frequently. During your training job, the StepOutlier rule was the most frequently triggered. It processed 1850 datapoints and was triggered 1 times.

	Description	Recommendation	Number of times rule triggered	Number of datapoints	Rule parameters
StepOutlier	Detects outliers in step duration. The step duration for forward and backward pass should be roughly the same throughout the training. If there are significant outliers, it may indicate a system stall or bottleneck issues.	Check if there are any bottlenecks (CPU, I/O) correlated to the step outliers.	1	1850	threshold:3 mode:None n_outliers:10 stddev:3
LowGPUUtilization	Checks if the GPU utilization is low or fluctuating. This can happen due to bottlenecks, blocking calls for synchronizations, or a small batch size.	Check if there are bottlenecks, minimize blocking calls, change distributed training strategy, or increase the batch size.	1	1115	threshold_p95:70 threshold_p5:10 window:500 patience:1000
BatchSize	Checks if GPUs are underutilized because the batch size is too small. To detect this problem, the rule analyzes the average GPU memory footprint, the CPU and the GPU utilization.	The batch size is too small, and GPUs are underutilized. Consider running on a smaller instance type or increasing the batch size.	1	1114	cpu_threshold_p95:70 gpu_threshold_p95:70 gpu_memory_threshold_p95:70 patience:1000 window:500
Dataloader	Checks how many data loaders are running in parallel and whether the total number is equal the number of available CPU cores. The rule triggers if number is much smaller or larger than the number of available cores. If too small, it might lead to low GPU utilization. If too large, it might impact other compute intensive operations on CPU.	Change the number of data loader processes.	0	10	min_threshold:70 max_threshold:200
MaxInitializationTime	Checks if the time spent on initialization exceeds a threshold percent of the total training time. The rule waits until the first step of training loop starts. The initialization can take longer if downloading the entire dataset from Amazon S3 in File mode. The default threshold is 20 minutes.	Initialization takes too long. If using File mode, consider switching to Pipe mode in case you are using TensorFlow framework.	0	1850	threshold:20
IOBottleneck	Checks if the data I/O wait time is high and the GPU utilization is low. It might indicate IO bottlenecks where GPU is waiting for data to arrive from storage. The rule evaluates the I/O and GPU utilization rates and triggers the issue if the time spent on the IO bottlenecks exceeds a threshold percent of the total training time. The default threshold is 50 percent.	Pre-fetch data or choose different file formats, such as binary formats that improve I/O performance.	0	1120	threshold:50 io_threshold:50 gpu_threshold:10 patience:1000
LoadBalancing	Detects workload balancing issues across GPUs. Workload imbalance can occur in training jobs with data parallelism. The gradients are accumulated on a primary GPU, and this GPU might be overused with regard to other GPUs, resulting in reducing the efficiency of data parallelization.	Choose a different distributed training strategy or a different distributed training framework.	0	1115	threshold:0.2 patience:1000
CPUBottleneck	Checks if the CPU utilization is high and the GPU utilization is low. It might indicate CPU bottlenecks, where the GPUs are waiting for data to arrive from the CPUs. The rule evaluates the CPU and GPU utilization rates, and triggers the issue if the time spent on the CPU bottlenecks exceeds a threshold percent of the total training time. The default threshold is 50 percent.	Consider increasing the number of data loaders or applying data pre-fetching.	0	1120	threshold:50 cpu_threshold:90 gpu_threshold:10 patience:1000

v. *Evaluation*

Rather than model's accuracy, I tried to use the standard *Cross Entropy Loss function* as my metric to study my training process results along with hyperparameter tuning and optimization for quality and improvements. The reason for this is that *Cross-Entropy* is to take the output probabilities (P) and measure the distance from the truth.



```
2023-06-01T05:50:19.105+03:00    Images [6000/8033 (75%)] Loss: 1.53 Accuracy: 1561/6000 (26.02%)
2023-06-01T05:50:37.110+03:00    Images [8000/8033 (100%)] Loss: 1.70 Accuracy: 2103/8000 (26.29%)
2023-06-01T05:50:39.111+03:00    train loss: 1.6121, acc: 0.2629, best loss: 1000000.0000
2023-06-01T05:50:39.112+03:00    Epoch 0, Phase valid
2023-06-01T05:50:58.117+03:00    Images [2000/2009 (100%)] Loss: 0.88 Accuracy: 437/2000 (21.85%)
2023-06-01T05:50:58.117+03:00    valid loss: 3.9866, acc: 0.2215, best loss: 3.9866
2023-06-01T05:50:58.117+03:00    Epoch 1, Phase train
2023-06-01T05:51:16.121+03:00    Images [2000/8033 (25%)] Loss: 1.31 Accuracy: 591/2000 (29.55%)
2023-06-01T05:51:34.126+03:00    Images [4000/8033 (50%)] Loss: 1.51 Accuracy: 1162/4000 (29.05%)
2023-06-01T05:51:54.131+03:00    Images [6000/8033 (75%)] Loss: 1.70 Accuracy: 1706/6000 (28.43%)
2023-06-01T05:52:13.136+03:00    Images [8000/8033 (100%)] Loss: 1.77 Accuracy: 2237/8000 (27.96%)
2023-06-01T05:52:13.136+03:00    train loss: 1.5712, acc: 0.2795, best loss: 3.9866
2023-06-01T05:52:13.136+03:00    Epoch 1, Phase valid
2023-06-01T05:52:32.141+03:00    Images [2000/2009 (100%)] Loss: 1.27 Accuracy: 507/2000 (25.35%)
2023-06-01T05:52:32.141+03:00    valid loss: 3.8442, acc: 0.2544, best loss: 3.8442
2023-06-01T05:52:32.141+03:00    Epoch 2, Phase train
2023-06-01T05:52:50.145+03:00    Images [2000/8033 (25%)] Loss: 1.46 Accuracy: 575/2000 (28.75%)
2023-06-01T05:53:10.151+03:00    Images [4000/8033 (50%)] Loss: 1.42 Accuracy: 1138/4000 (28.45%)
2023-06-01T05:53:29.157+03:00    Images [6000/8033 (75%)] Loss: 1.66 Accuracy: 1707/6000 (28.45%)
2023-06-01T05:53:47.161+03:00    Images [8000/8033 (100%)] Loss: 1.74 Accuracy: 2246/8000 (28.07%)
2023-06-01T05:53:47.162+03:00    train loss: 1.5681, acc: 0.2801, best loss: 3.8442
2023-06-01T05:53:47.162+03:00    Epoch 2, Phase valid
2023-06-01T05:54:06.166+03:00    Images [2000/2009 (100%)] Loss: 2.41 Accuracy: 507/2000 (25.35%)
2023-06-01T05:54:06.166+03:00    valid loss: 3.9691, acc: 0.2524, best loss: 3.8442
```

8. Algorithms and techniques

Fine-tuned a Resnet50 CNN as a pre-trained image classification network.

ResNet-50 is a convolutional neural network that is 50 layers deep. You can load a pretrained version of the neural network trained on more than a million images from the ImageNet database

PyTorch

PyTorch is a fully featured framework for building deep learning models, which is a type of machine learning that's commonly used in applications like image recognition and language processing. Written in Python, it's relatively easy for most machine learning developers to learn and use.

AWS SageMaker

Amazon SageMaker is a managed service in the Amazon Web Services (AWS) public cloud. It provides the tools to build, train and deploy machine learning (ML) models for predictive analytics applications. The platform automates the tedious work of building a production-ready artificial intelligence (AI) pipeline.

9. Result:

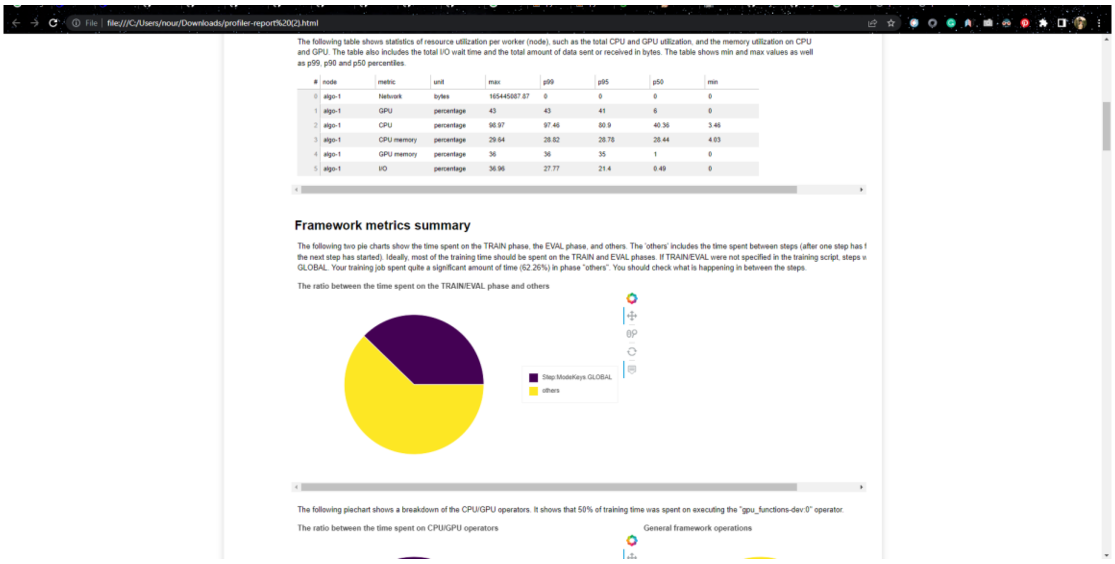
My first model had hyperparameters a batch_size of 64 and 5 epochs and got accuracy of 16 to 22%

So, I tried to change the hyperparameters a bit and added learning rate to the hyper parameters. I got 24 to 29% for batch_size = 16 and 25 epochs (did 7 and stopped early). No real dealbraker results as my algorithm still needs much debugging and improvements but I tried and searched for so many approaches. Such as: imbalanced_data_sampler, SMOTE and much other techniques but encountered so many odd errors and library version issues I spent all my time trying to solve.

10. Reflections:

- Debug Case

The last model stopped after only 7 epochs as it detected that the loss was not decreasing. I then decided to print a debugger-profiler output report and found this out. And mostly all rules on the rules summary didn't even trigger.



file:///C:/Users/nour/Downloads/profiler-report%20(2).html

The following table shows statistics of resource utilization per worker (node), such as the total CPU and GPU utilization, and the memory utilization on CPU and GPU. The table also includes the total I/O wait time and the total amount of data sent or received in bytes. The table shows min and max values as well as p99, p90 and p50 percentiles.

#	node	metric	unit	max	p99	p95	p50	min
0	algo-1	Network	bytes	165445087.87	0	0	0	0
1	algo-1	GPU	percentage	43	43	41	6	0
2	algo-1	CPU	percentage	98.97	97.46	80.9	40.36	3.46
3	algo-1	CPU memory	percentage	29.64	28.82	28.78	28.44	4.03
4	algo-1	GPU memory	percentage	36	36	35	1	0
5	algo-1	I/O	percentage	36.96	27.77	21.4	0.49	0

Framework metrics summary

The following two pie charts show the time spent on the TRAIN phase, the EVAL phase, and others. The 'others' includes the time spent between steps (after one step has f the next step has started). Ideally, most of the training time should be spent on the TRAIN and EVAL phases. If TRAIN/EVAL were not specified in the training script, steps w GLOBAL. Your training job spent quite a significant amount of time (62.26%) in phase "others". You should check what is happening in between the steps.

The ratio between the time spent on the TRAIN/EVAL phase and others

The following piechart shows a breakdown of the CPU/GPU operators. It shows that 50% of training time was spent on executing the "gpu_functions-dev:0" operator.

The ratio between the time spent on CPU/GPU operators

General framework operations

My conclusion is there might have been a vanishing gradient problem that I need to figure out first before tuning hyperparameters.

- Thoughts on improvement

Over-Sampling Data / Data Augmentation

Hyperparameter Tuning

11. References

[1] https://github.com/awslabs/open-data-docs/tree/main/docs/aft-vbi-pds

[2] https://github.com/udacity/nd009t-capstone-starter

[3] https://www.arxiv-vanity.com/papers/2008.05756/