Capstone Report for project: Inventory Monitoring at Distribution Centers

1. Definition

Project Overview

In this project, we will build, train and deploy a deep learning model for image classification to count the number of objects from Amazon Bin Image Dataset, which contains over 500,000 images and metadata from bins of a pod in an operating Amazon Fulfillment Center, by using AWS Sagemaker. For the purpose of demonstrating how we build end-to-end machine learning pipelines, Udacity provides 10.4K images with a json metadata file which contains upto 5 objects in the bins. The solution can be used to solve real-world problems, such as efficient inventory management where the distribution centers can quickly and accurately count and record millions of delivered and received objects and enable real-time or quasi-real-time updates of inventory data.

Domain Background

Amazon is the largest eCommerce company in the world. Amazon ships approximately 1.6 million packages a day[1] and it offers order delivery within two days at no additional cost to its Prime members. Handling the logistics of such a large number of packages and ensuring that items can be delivered to customers in time to ensure the best user experience is a huge challenge for Amazon.
Amazon has responded to this challenge by building a large and efficient supply chain to ensure the best user experience. According to Amazon's official website, the eCommerce giant has more than 175 fulfillment centers, occupying approximately 150 million sq feet of space, spread across the world[2]. In addition, Amazon is estimated to operate roughly 500 warehouses worldwide[3]. Distribution centers play an important role in the process of delivering goods to customers in a timely manner, and the intelligence and automation is key to optimize operations and speed up delivery .

Problem Statement

Inventory management, a critical element of the supply chain, is the tracking of inventory from manufacturers to warehouses and from these facilities to a point of sale. Quickly synchronizing data between orders and available inventory quantities is the key to quickly and efficiently meeting consumer demand. This requires distribution centers to be able to quickly and accurately count and record the number of objects delivered and received, which can often be in the millions.
However, trying to count millions of objects manually is not enough to meet the demand for real-time or quasi-real-time inventory data updates, and can also bring huge expenses to the company. This requires the introduction of artificial intelligence, such as intelligent robots, to help companies accomplish these tasks efficiently.
Robots are primarily used to move objects from point A to point B in order to facilitate inventory management. Objects are carried in bins which can contain multiple objects. We can train the robots to identify the number of objects inside each bin and record this data for automation and intelligence. Since the images of the objects inside each bin have been collected, we will use the Image Classification method to count the objects. We will use AWS Sagemaker, a fully managed service for machine learning solution by Amazon, to train and deploy image classification model. For the model, we will use the pre-trained model Resnet50 to train our image classification model to identify the number of objects inside the bin.

Evaluation Metrics

The accuracy will be the evaluation metric. It can be used to evaluate exactly how accurately our model counts the number of objects in bins.

2. Analysis

Data Exploration

In this project, Udacity provides 10.4K images of Amazon Bin Image Dataset with a json metadata file which contains upto 5 objects in the bins for the demo purpose. We will create a folder called train_data, and download training data and arranges it in subfolders according to the json file provided by Udacity. Each of these subfolders contain images where the number of objects is equal to the name of the folder. For instance, all images in folder `1` has images with 1 object in them. Finally, we will divide the train_data into training, test and validation sets with 80-10-10.

The dataset we use has a total of 10,441 images, and their corresponding label distribution is shown in Fig 1

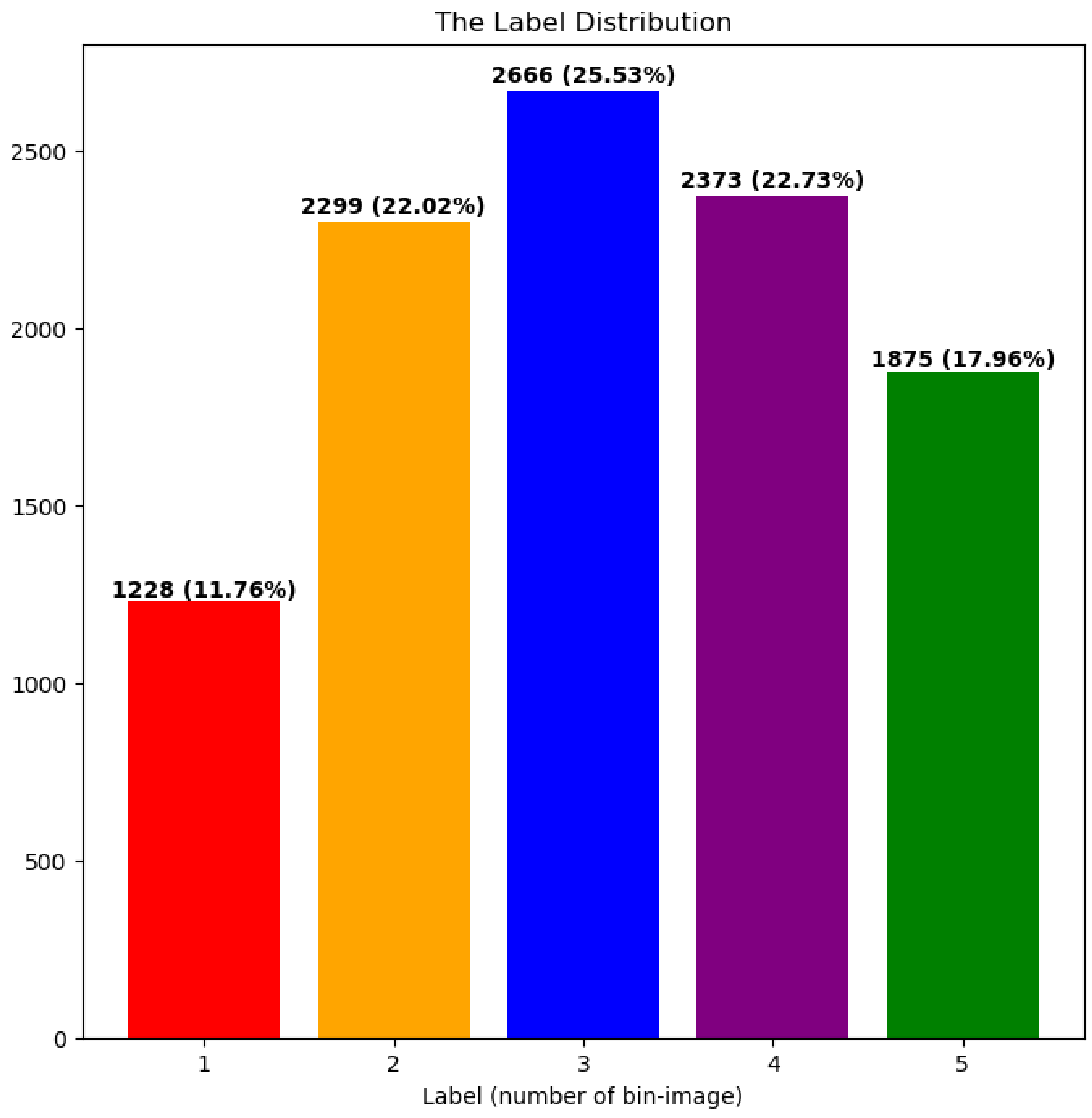


Fig 1. The Label distribution

We can see that the distribution of label in the dataset provided by Udaciy has a slight imbalance, with label 1 making up the least percentage(11.76%), while label 3 makes up the largest percentage (25.53%). We can extract additional data from Amazon Bin Image Dataset to make sure the number of labels is the same. But we will not make any changes in this project for demo purpose.

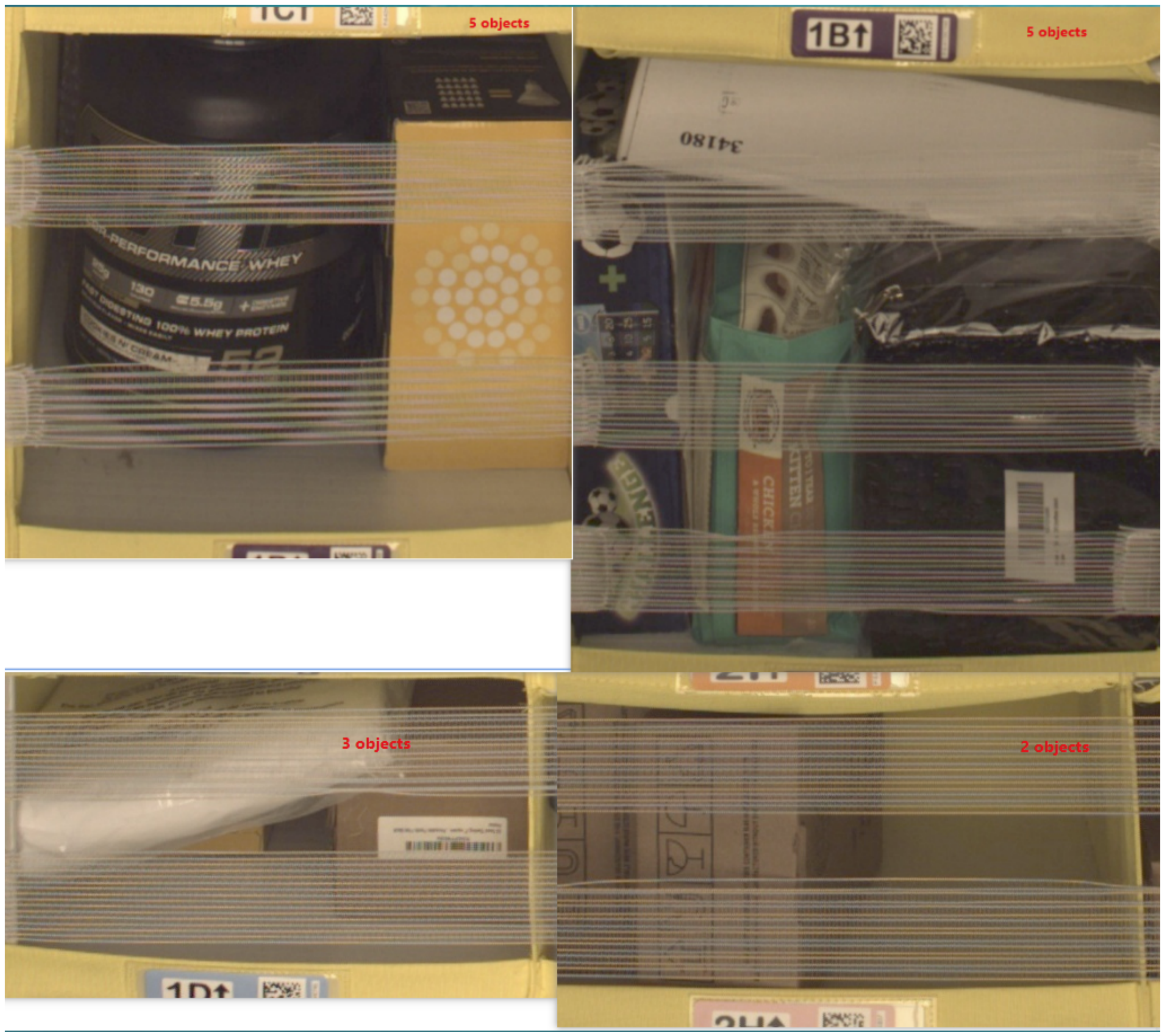


Fig 2. Sample Images with different number of objects in bins

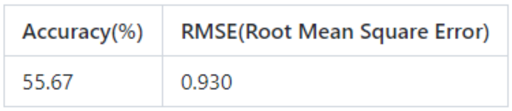
By randomly viewing the images, as shown in Fig 2, we can see that the images are of different sizes each, and the packaging has many bundles of tape on the objects that obscure the view, making it difficult for us humans to determine the number of objects in bins. We will resize the image as transform to ensure that the input image size of the model is the same. Although there is a lot of research into how to remove an image noise like tape, etc., it is actually a data quality issue when tape on objects hinders human or machine recognition. Therefore, we will not do data augmentation, data noise removal, etc. for the time being in this project.

Algorithms and Techniques

Convolutional Neural Networks(CNNs) are commonly used to power computer vision applications. **We will use a more advanced CNN, Resnet50, an pre-trained image classification model based on millions of images whose .** ResNet stands for Residual Network and is a specific type of convolutional neural network (CNN) introduced in the 2015 paper “Deep Residual Learning for Image Recognition” by He Kaiming, Zhang Xiangyu, Ren Shaoqing, and Sun Jian[5]. ResNet-50 is a 50-layer convolutional neural network (48 convolutional layers, one MaxPool layer, and one average pool layer)[6]. In this project, I will freeze the layers of the pre-trained model ResNet50 except for the last linear layer, which will be replaced with our own output layer with an output size of 5.
To make the deployment of the model easier, we will use AWS Sagemaker, a fully managed service for machine learning solution by Amazon, to train and deploy image classification model.

Benchmark

The Amazon Bin Image Dataset (ABID) Challenge provides a benchmark for moderate task that counts upto 5 objects, and the accuracy is one of the evaluation metric. Considering the same prediction target and metric, I will use the 55.67% of accuracy as the benchmark. It run 40 epochs, and every 10 epochs learning rate will decay by a factor of 0.1. Batch size is 128. Benchmark accuracy can be seen in below.



Accuracy(%)	RMSE(Root Mean Square Error)
55.67	0.930

3. Methodology

Data Preprocessing

As we mentioned in the Data Exploration section, the images of the dataset we use are not of the same size, so the data preprocessing we need is to resize the size of the original images. Then we convert them to Tensor and finally normalize these tensors. This transform operation is applied to the train, test and validation datasets. In each train job and endpoint invoke, this transform operation will be executed first.

Implementation

1. Data Preprocessing

We will use the transforms of the `torchvision` library to implement the data preprocessing operations described above.`

```
import torchvision.transforms as transforms

transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0,0,0), std=(1,1,1))
    ])
```

2. Model

We will use the models under the `torchvision` library as our pre-train model and then freeze all the model parameters and finally replace the last linear layer with our own linear layer with output size of 5. The trained model will be save at the specified model_dir.

```
def net(num_classes=5):
    # load the pre-trained model - resnet50
    model = models.resnet50(pretrained=True)
    # freeze part of  model except the last linear layer
    for param in model.parameters():
        param.requires_grad = False
    # input size for last linear layer
    num_features = model.fc.in_features
    # update the last linear layer
    model.fc = nn.Linear(num_features, num_classes)

    return model

def save_model(model, model_dir):
    logger.info("Saving the model...")
    path = os.path.join(model_dir, "model.pth")
    torch.save(model.cpu().state_dict(), path)
```

3. Training Job in Sagemaker

For the initiate training which is implemented by PyTorch built in sagemaker with `"ml.g4dn.12xlarge"` instance, we set up the following parameters for the training job :

```
    "batch_size": 64,
    "num_classes": 5,
    "lr": 0.01,
    "epochs": 2,
```

The initial accuracy for the testing data during training based on the above parameter0.3072.

4. Refinement - Hyperparameter Tuning

Amazon Sagemaker supports to look for the best model automatically by focusing on the most promising combinations of hyperparameter values within the ranges that we specify. To get good results, we need to choose the right ranges to explore. We select `lr` and `batch_size` as the hyperparameters:

```
hyperparameter_ranges = {
    "lr": ContinuousParameter(0.001, 0.1),
    "batch_size": CategoricalParameter([32, 64, 128]),
}
```

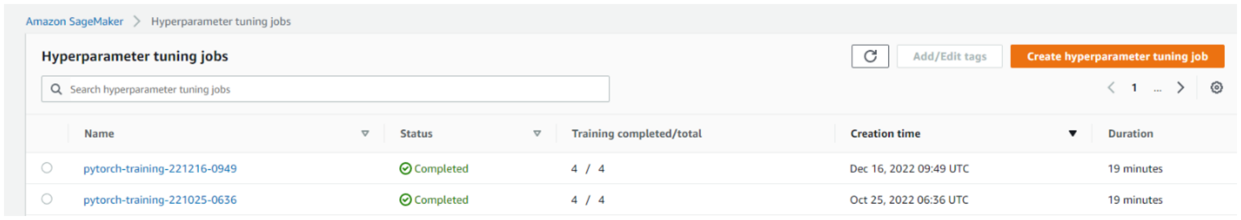
Sagemaker also support to define an objective metric for the tuning job to check the best ones. In this tuning job, we define the metric definition as below:

```
objective_metric_name = "average test loss"
objective_type = "Minimize"
metric_definitions = [{"Name": "average test loss", "Regex": "Testing Loss: ([0-9\\.]+
```

The minimum test loss will be the basis for finding the best parameters. The best hyperparameters are:

```
 'batch_size': '64',
 'lr': '0.0018788785870785117',
```

Note that the instance for the tuning job is ‘`ml.g4dn.12xlarge` to accelerate the hyperparameter tuning task. But for the cost saving, we could use `ml.g4dn.xlarge`


Amazon SageMaker > Hyperparameter tuning jobs
Add/Edit tags
Create hyperparameter tuning job
Search hyperparameter tuning jobs

Hyperparameter tuning jobs

	Name	Status	Training completed/total	Creation time	Duration
○	pytorch-training-221216-0949	Completed	4 / 4	Dec 16, 2022 09:49 UTC	19 minutes
○	pytorch-training-221025-0636	Completed	4 / 4	Oct 25, 2022 06:36 UTC	19 minutes

Fig 3. Hyperparameter Tuner Jobs

5. Debugging/Profiling Job

After geting the best hyperparameters with hyperparameter tuning, we will use model debugging and profiling to better monitor and debug the model training job.

SageMaker Debugger auto generated a SageMaker Debugger Profiling Report for the training jobs. The report provides summary of training job, system resource usage statistics, framework metrics, rules summary, and detailed analysis from each rule. To implement Sagemaker Debugger/Profiler, we need to add hooks to train and test the model after importing necessary libraries. Creating hook and registering the model will give us ability to select rules such as overfit, poor_weight_initialization, and profiler_report.

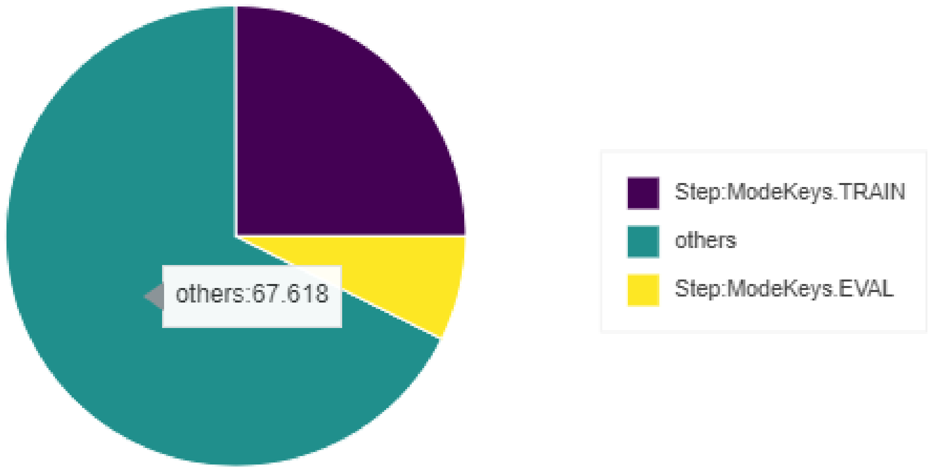
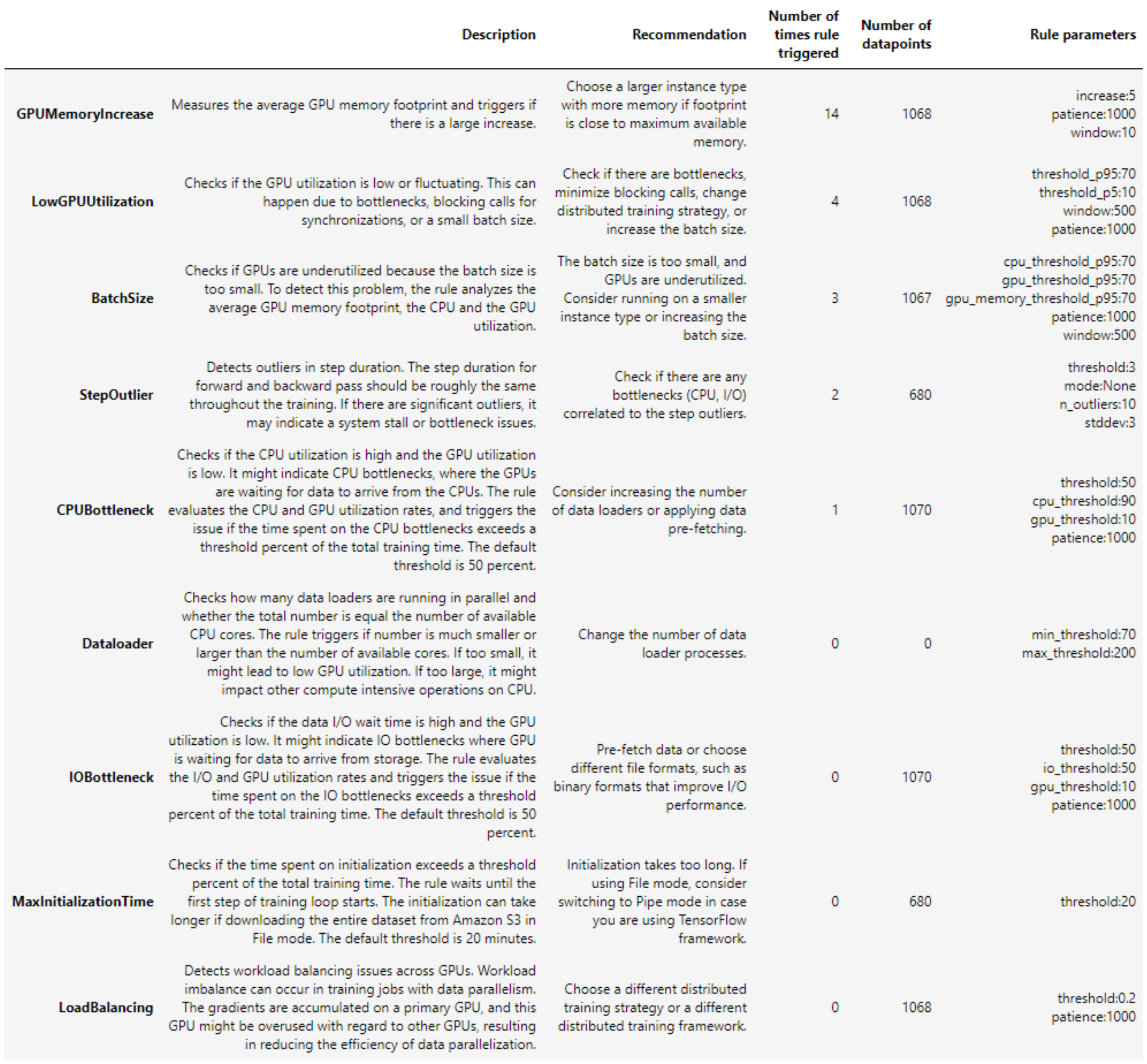


Fig 4. Debugger Framework metrics summary

Fig 4 shows the time spent on the TRAIN phase, the EVAL phase, and others. Our training job spent quite a significant amount of time (67.62%) in phase "others". The main reason is that it takes more time to download the data from S3 and to download the ResNet50 model.



	Description	Recommendation	Number of times rule triggered	Number of datapoints	Rule parameters
GPUMemoryIncrease	Measures the average GPU memory footprint and triggers if there is a large increase.	Choose a larger instance type with more memory if footprint is close to maximum available memory.	14	1068	increase:5 patience:1000 window:10
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.	4	1068	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.	3	1067	cpu_threshold_p95:70 gpu_threshold_p95:70 gpu_memory_threshold_p95:70 patience:1000 window:500
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.	2	680	threshold:3 mode:None n_outliers:10 stddev:3
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.	1	1070	threshold:50 cpu_threshold:90 gpu_threshold:10 patience:1000
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	0	min_threshold:70 max_threshold:200
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	1070	threshold:50 io_threshold:50 gpu_threshold:10 patience:1000
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	680	threshold:20
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	1068	threshold:0.2 patience:1000

Fig 5. Rule Summary

Fig 5 shows a profiling summary of the Debugger built-in rules. The table is sorted by the rules that triggered the most frequently. During our training job, the GPUMemoryIncrease rule was the most frequently triggered. It processed 1068 datapoints and was triggered 14 times. Hence we can consider choosing a larger instance with more memory. The 2nd LowGPUUtilization shows our training job is underutilizing the instance(`ml.g4dn.12xlarge`). We may want to consider to either switch to a smaller instance type or to increase the batch size.
and 3rd BatchSize recommend that we need to increase the batch size.

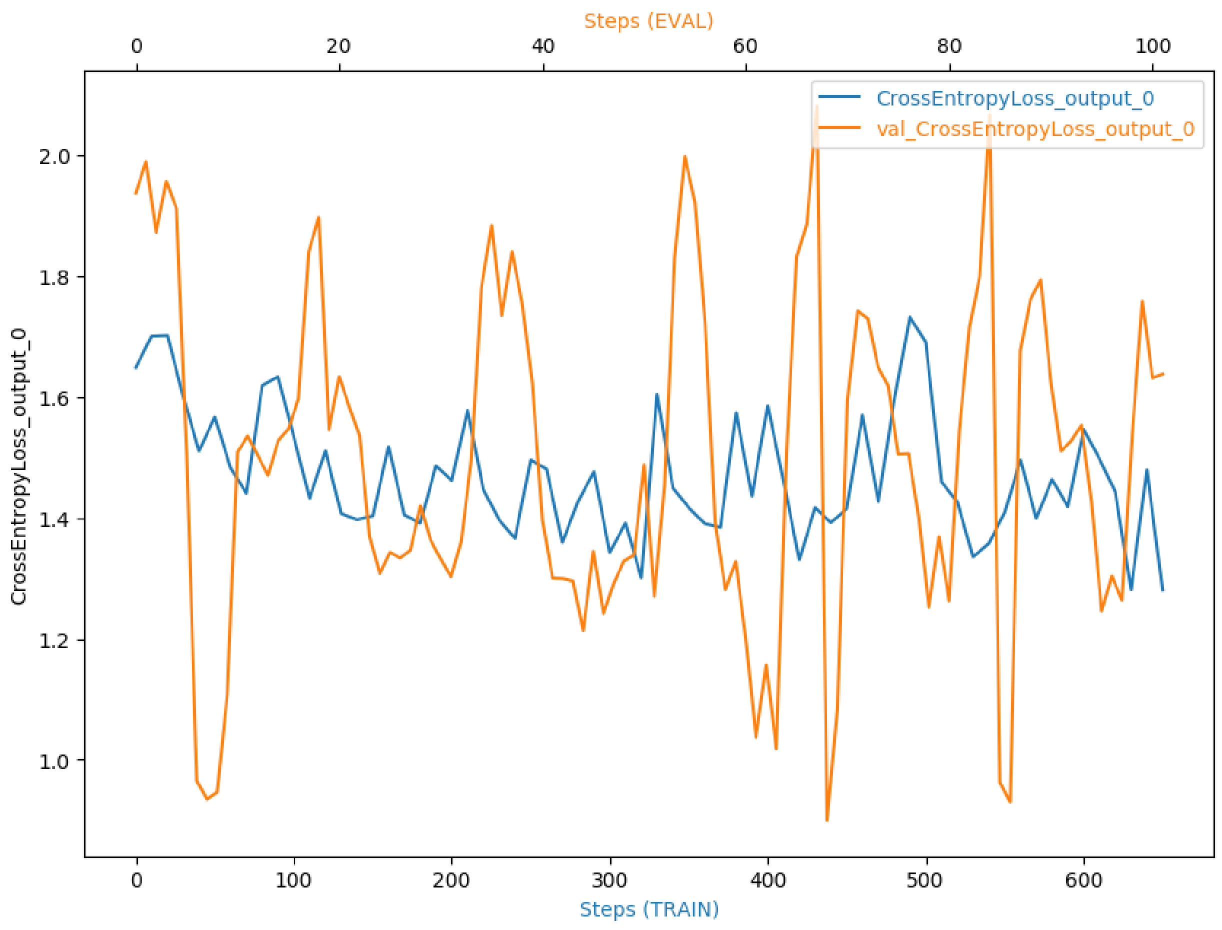


Fig 6. CrossEntropyLoss Output per step

I set save_interval up as 10 steps interval for training and 1 for validation as we want to record more data. We see that the cross-entropy loss in the validation set fluctuates widely, which may be related to the large difference between the images in the validation set and the training set.

6. Deployment

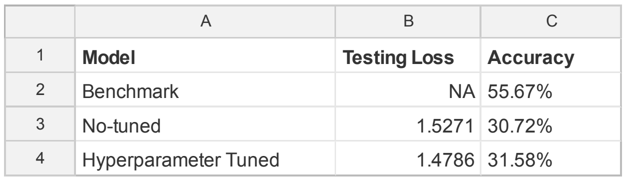
The deployment of this project is mainly implemented through Sagemaker's endpoint service. After training and evaluating and tuning the best model, we need to write a separate script `inference.py` to achieve successful deployment of endpoint, in which we need to implement the net, model_fn, input_fn and predict_fn functions. In addition to this, we need to define predictor_cls, which is a function to call to create a predictor. Our ImagePredictor class will collect endpoint name, serializer, deserializer and also Sagemaker session. We use the `PyTorchModel` in Sagemaker to create a SageMaker PyTorchModel object that can be deployed to an `Endpoint` and then invoke deploy method to create an endpoint in Sagemaker.

```
def net(num_classes=5):
    # load the pre-trained model - resnet50
    model = models.resnet50(pretrained=True)
    # freeze part of  model except the last linear layer
    for param in model.parameters():
        param.requires_grad = False
    # input size for last linear layer
    num_features = model.fc.in_features
    # update the last linear layer
    model.fc = nn.Linear(num_features, num_classes)

    return model

def model_fn(model_dir):
    model = net()
    model_path = os.path.join(model_dir, 'model.pth')
    device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
    if (device == torch.device("cpu")) or (device=="cpu"):
                model.load_state_dict(
            torch.load(model_path, map_location=torch.device('cpu')))
    else:
        model.load_state_dict(
            torch.load(model_path))

    # with open(os.path.join(model_dir, "model.bin"), "rb") as f:
    #     model.load_state_dict(torch.load(f))
    logger.info('Successfully loaded the model')
    return model.to(device)

def input_fn(request_body, content_type='image/jpeg'):
    logger.info('Deserializing the input data.')
    # process an image uploaded to the endpoint

    logger.debug(f'Request body CONTENT-TYPE is: {content_type}')
    logger.debug(f'Request body TYPE is: {type(request_body)}')

    if content_type == 'image/jpeg':
        return Image.open(io.BytesIO(request_body))

    elif content_type == 'application/json':
        #img_request = requests.get(url)
        logger.debug(f'Request body is: {request_body}')
        request = json.loads(request_body)
        logger.debug(f'Loaded JSON object: {request}')
        url = request['url']
        img_content = requests.get(url).content
        return Image.open(io.BytesIO(img_content))
    else:
        raise Exception('Requested unsupported ContentType in content_type: {}'.format

# inference
def predict_fn(input_object, model):

    logger.info('In predict fn')
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0,0,0), std=(1,1,1))
    ])

    logger.info("transforming input")
    input_object=transform(input_object)
    device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
    input_object = input_object.to(device)
    with torch.no_grad():
        logger.info("Calling model")
        prediction = model(input_object.unsqueeze(0))
    return prediction
```

Results

Model Evaluation and Validation

In this project, we trained and tuned the model based on our dataset with the following results, including benchmark.



	A	B	C
1	**Model**	**Testing Loss**	**Accuracy**
2	Benchmark	NA	55.67%
3	No-tuned	1.5271	30.72%
4	Hyperparameter Tuned	1.4786	31.58%

Table 1 Model Evaluation Results

Justificatiion

Compared with benchmark's 55.67% of accuracy, the accuracy of our tuned model is 31.58%, which is 24.09% lower than benchmark. This is mainly due to the fact that we are using a small dataset, which accounts for 2% of the total data in the Amazon Bin Image Dataset.
In future work, in order to get better performance from this model, we can start by collecting more data.

Reference

[1] https://landingcube.com/amazon-statistics/
[2] https://www.aboutamazon.co.uk/amazon-fulfilment/fulfilment-in-our-buildings
[3] https://feedvisor.com/resources/amazon-shipping-fba/3-key-elements-to-amazons-supply-chain-strategy/
[4] https://en.wikipedia.org/wiki/Amazon_SageMaker
[5] Deep Residual Learning for Image Recognition
[6] ResNet-50: The Basics and a Quick Tutorial