AWS Machine Learning Engineer Capstone Report

- INVENTORY MONITORING AT DISTRIBUTION CENTERS

- by Raphael Blankson

Domain Background

In recent times, Artificial Intelligence (AI) has come roaring out of high-tech labs to become something that people use every day without even realizing it. AI has brought great benefits to all industries, including supply chain and logistics. The volume of data in supply chain and logistics is ever-growing daily and thus more sophisticated solutions are urgently needed.

One of the main engines of supply-chain and logistics is distribution centers. A distribution center is a specialized warehouse that serves as a hub to strategically store finished goods, streamline the picking and packing process, and ship goods out to another location or final destination [1]. Monitoring inventory is critical for the success of all distribution centers. This is because without the proper monitoring, there will be shortage in inventory which can lead to backorders and customers not receiving the expected number of products or no product at all. To achieve this, companies usually use robots to move items in bins.

This project focuses on building a machine learning (computer vision) model that can be used in the inventory monitoring process (counting the number of items in the bins robots carry). Using the model for this process speeds up the distribution process and also keeps customers happy.

Problem Statement

Amazon fulfillment centers are bustling hubs of innovation the company to deliver millions of products all over the world. These products are randomly placed in bins which are carried by robots. There are often issues of misplaced items which result in mismatch. The recorded bin inventory differs from its actual content [6].
The goal of this project is to create a deep learning model that can identify the content of the items in the bin and provide the item count. This can be achieved by attaching a camera to the robot and feeding the images from the camera to the deep learning model which will then identify the content and provide the item count.
The tasks involved may be summarized as follows:

- Download and preprocess the Amazon Bin Image Dataset [2].
- Upload data to S3.
- Train model that can identify and show item count.
- Deploy model to sagemaker endpoint.
- Use hyper-parameter tuning to select the best hyperparameters.
- Train model with spot instances to reduce cost.
- Use multi-instance training to distribute training across multiple instances.

By providing the correct count of items using artificial intelligence will help serve its customers better.

The final model is expected to output the number of items present in an image of a bin.

Evaluation Metrics

We will use accuracy as the evaluation metrics for this project. The formula for accuracy is given by:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Dataset and Inputs

The original Amazon Bin Image Dataset [2] contains 500,000 images (jpg format) from bins (containing one or more objects) of a pod in an operating Amazon Fulfillment Center. The bin images in this dataset are captured as robot units carry pods as part of normal Amazon Fulfillment Center operations. For each image there is a metadata file (json format) containing information about the image like the number of objects, it's dimension and the type of object. For this task, only a subset of the dataset used to classify the number of objects in each bin in order to avoid excess cost and the item count from the metadata is used as the label to train the model.

Images are located in the bin-images directory, and metadata for each image is located in the metadata directory. Images and their associated metadata share simple numerical unique identifiers.
Below is a table showing the total count in both original dataset and our sample:

Description	Total size	Train size	Test size
Project Image	10,441	8,352	2,089
Original Datasource	535,234	481,711	53,523

Below is an example image and its associated metadata:

fig 1: sample image

```
{
    "BIN_FCSKU_DATA": {
        "B000A8C5QE": {
            "asin": "B000A8C5QE",
            "height": {
                "unit": "IN",
                "value": 4.200000000000001
            },
            "length": {
                "unit": "IN",
                "value": 4.7
            },
            "name": "MSR PocketRocket Stove",
            "quantity": 1,
            "weight": {
                "unit": "pounds",
                "value": 0.45
            },
            "width": {
                "unit": "IN",
                "value": 4.4
            }
        },
        "B0064LIWVS": {
            "asin": "B0064LIWVS",
            "height": {
                "unit": "IN",
                "value": 1.2
            },
            "length": {
                "unit": "IN",
                "value": 5.799999999999999
            },
            "name": "Applied Nutrition Liquid Collagen Skin Revitalization, 10 Count 3.35 Fl Ounce",
            "quantity": 1,
            "weight": {
                "unit": "pounds",
                "value": 0.3499999999999999
            },
            "width": {
                "unit": "IN",
                "value": 4.7
            }
        }
    },
    "EXPECTED_QUANTITY": 2,
    "image_fname": "523.jpg"
}
```


fig 2: sample metadata.json

As can be seen from the image, there are tapes in front of the bin that prevent items from falling and sometimes can make the objects in the bin unclear. From the metadata, we can see that this particular bin shown in the image contains 2 different object categories as shown in `"EXPECTED_QUANTITY": 2` and for each category, the `quantity` field is 1. Each object category has a unique id called `asin`.

For this task we used only a subset of the original dataset. A file_list.json file was provided as part of the starter files for the project. Before downloading the images, the file_list.json file was randomly shuffled and splitted into train and test data json files without losing the structure of the file_list.json. The total size of the subset data was 10,441. 80% was used for training (8,352 items) and 20% for test (2,089 items).The dataset was uploaded to S3 to be used later in training.

From the distribution plots below, it can be observed that for both train and test datasets, bin images that contain 3 objects have the highest frequency whilst those with only one item occurred the least. The plot shows a normal distribution of the count plots.

```
[9]: <AxesSubplot:xlabel='col_names'>
```

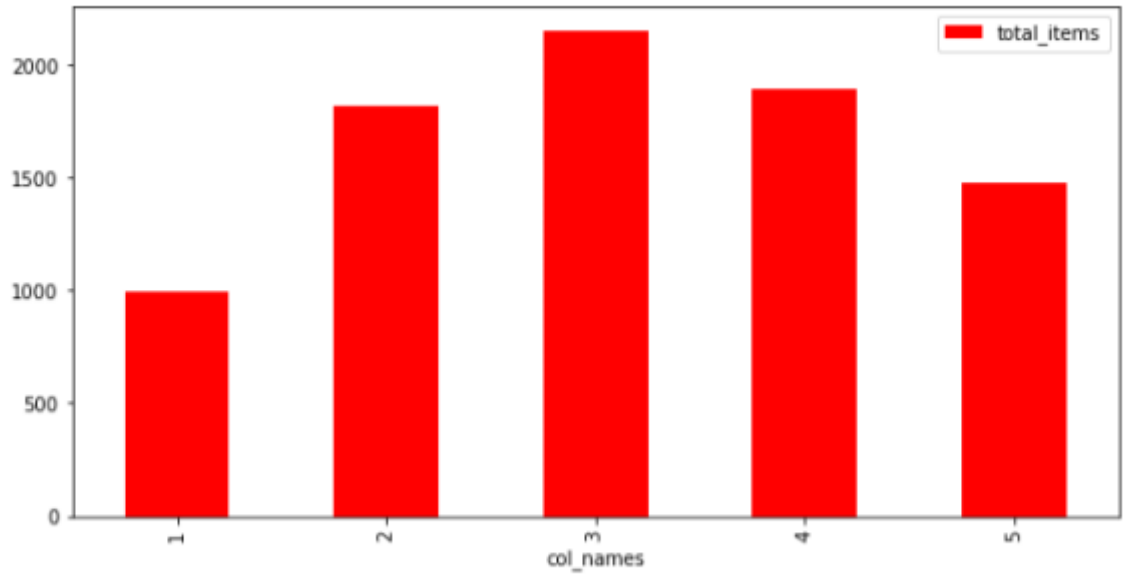

Fig 3: distribution plot for train data

```
[12]: <AxesSubplot:xlabel='col_names'>
```

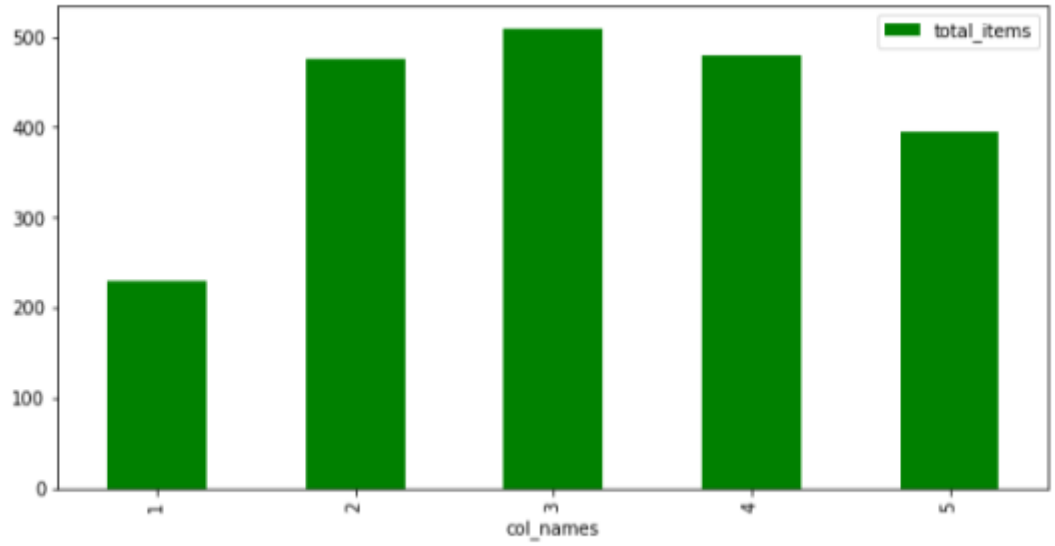

Fig 4: Distribution plot for test data

Algorithms and Techniques

Since the task at hand is an image classification task, the ideal model to choose is a convolutional neural network. Training a convolutional neural network from scratch is expensive and thus a method called **transfer learning** which is the improvement of learning a new task through the transfer of knowledge from a related task that has already been learned [4]. Convolutional neural networks are data-hungry, requiring large amounts of training data in order to perform very well. Since only a small subset of the dataset is used in this project (as proof-of-concept), we do not expect our model to beat the benchmark model which was trained on the entire dataset.

A pre-trained convolutional neural network called ResNet [3] will be used. One advantage of using the ResNet architecture is that it solves the vanishing gradient problem. The pre-trained network as a feature extractor and then replace its final layer with a fully connected neural network for fine-tuning on our task at hand. The final layer outputs 5 numbers which correspond to each of our labels for the dataset. The model will access the data from S3 bucket.

The solution will be implemented in a sagemaker notebook instance. I will perform hyperparameter tuning to search for the best hyperparameters for training the model. The following hyperparameters can be tuned to optimize the model:

- Learning rates (how fast should the model learn)
- Number of Epochs (training lengths)
- Batch size (how many images to look at once during a single training)

The model will be deployed to a sagemaker endpoint and also create a lambda function to test how the model can be accessed by external clients. The model will also be tested on an EC2 instance to check how we can reduce the cost of running our model in the cloud.

Finally, multi-instance training will be done. Accuracy will be the metric used in accessing the model.

Below is the image of the model architecture used for transfer learning:

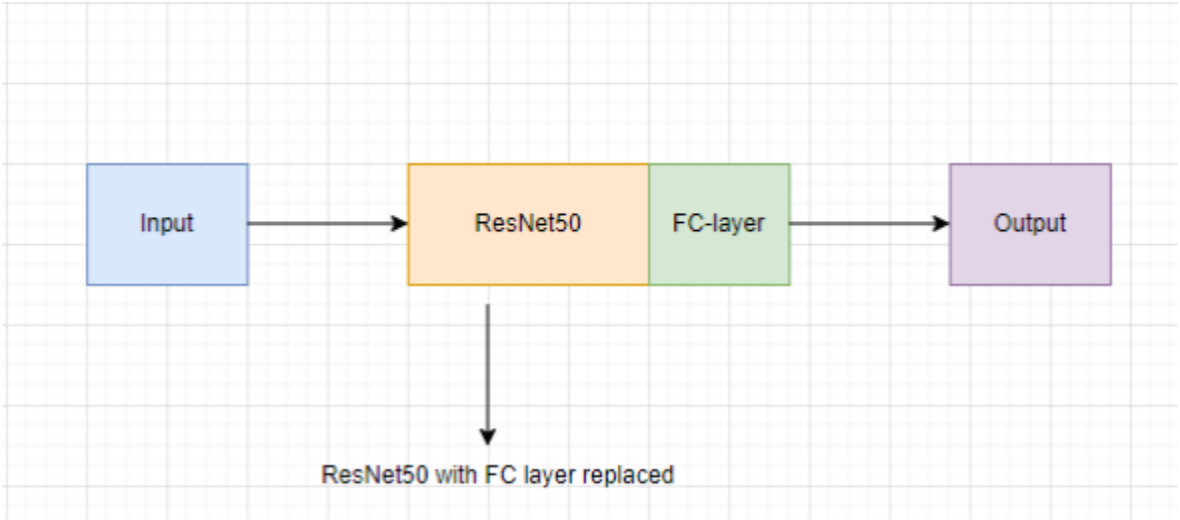

Fig 5: transfer-learning architecture

BenchMark Model

The bench model is taken from [5]. The model from [5] achieved an overall accuracy of 55.67%. We will have to improve the accuracy of this model and also train using sagemaker best practices as learnt from the nanodegree. However, since we are using only a subset of the dataset whilst the benchmark was trained on the entire dataset, the model is expected to achieve an accuracy of about 25%.

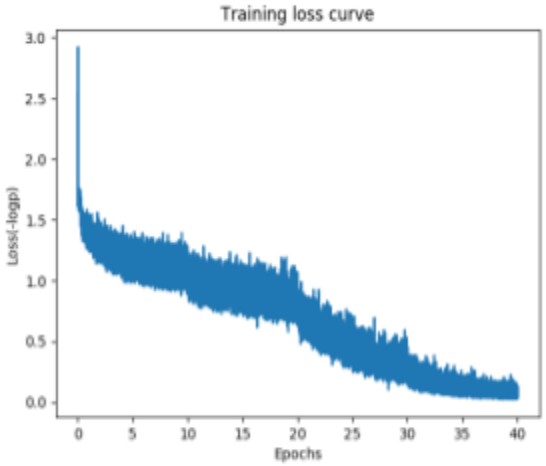

Fig 6: Benchmark loss

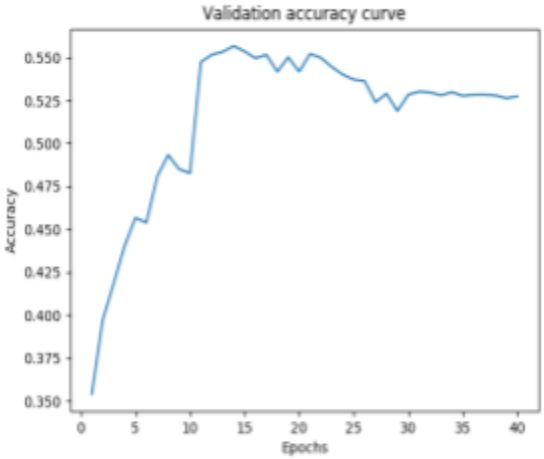

Fig 7: Benchmark accuracy

Project Design

The project will be run in sagemaker, thus we have to create a sagemaker instance. The main steps involved in the project design are listed below:

- Download the dataset.
- Perform data preprocessing if necessary.
- Upload data to an S3 bucket.
- Perform hyperparameter tuning to select the best hyperparameters.
- Create and train the model using sagemaker estimator.
- Use model debugging and profiling to monitor and debug the training job in sagemaker.
- Deploy the model to a sagemaker endpoint and make predictions on the endpoint.
- Perform cheaper training using EC2 instance.
- Train the model on multiple instances.

Data Preprocessing

Using the json file provided, the file was divided into train and test json files that were used to download the actual images into train and test folders respectively.

Since the pre-trained model was trained on imagenet dataset, the images were resized to shape (224 X 224) to be used as input to the model. The images were horizontally flipped , normalized and converted to pytorch tensors when defining the pytorch dataloaders.

Project Implementation

External training scripts are called in the main jupyter notebook called **sagemaker.ipynb** for training the model. The training scripts are implemented in the following steps:

1. Implement helper functions:
    a. **Create_data_loaders:** for creating pytorch dataloaders from both train and test datasets while applying data transformation.
    b. **net:** for defining the model or network architecture (pretrained ResNet50)
    c. **train:** for defining the training process and training parameters
    d. **test:** for defining the model testing process for evaluation
2. Define the loss function (Cross Entropy loss) and optimizer (Adam optimizer)
3. Train the network by calling the helper methods, logging both training and validation loss as well as validation accuracy.

Note: As described earlier, the final fully connected layer of the ResNet50 model used is replaced by a customized fully connected layer which outputs 5 numbers that correspond to each of the class labels (number of items in a bin).

We use similar training scripts for both hyperparameter tuning, training on EC2 instance and sagemaker estimator.

Hyperparameter Tuning job

Hyperparameters are the settings that can be tuned before running a training job in order to control the behaviour of a machine learning model. They can impact the training time, model convergence and the accuracy of the model. To select the best hyperparameter, I used the following 3 parameters in my hyperparameter search space:

- **learning_rate** - determines the step size at each epoch whilst moving toward the minimum of a loss function.Learning rate influences the extent to which a model can acquire new information. Generally, during training, the model starts at some random point and samples different weights. It is extremely crucial to set the right learning rate for any training job because a larger learning rate may cause the model to overshoot the optimal solution and smaller learning rate will result in longer training time to find the optimal solution.
- **batch_size** - shows the number of training samples that will be used in each epoch. This is necessary because smaller batch sizes help to easily get out "local minima" and also utilize less resource power whilst larger batch sizes require more resource power and may get stuck in the wrong solution.
- **epochs** - refers to one complete pass of the training dataset through the model. It is important to set the right number of epochs because this can influence overfitting. For a small dataset as used in this project, setting a very large epoch may cause the model to memorize the training data instead of generalizing and this may cause overfitting and setting it very small may also result in the model not learning.

For the hyperparameter tuning job, I used the sagemaker tuner. This searches within the range provided for the hyperparameters listed for the values that gives the best outcome of the model, ie. the minimum loss function. The batch_size and epochs ranges are categorical whilst that of the learning rate is continuous. I used the **hpo.py** script as an entry point, single instance and the **ml.g4dn.xlarge** instance type for faster training since it has a **GPU.** I run the 2 jobs in parallel and set the maximum jobs to 2 as well. Below are images of the hyperparameter tuning jobs:

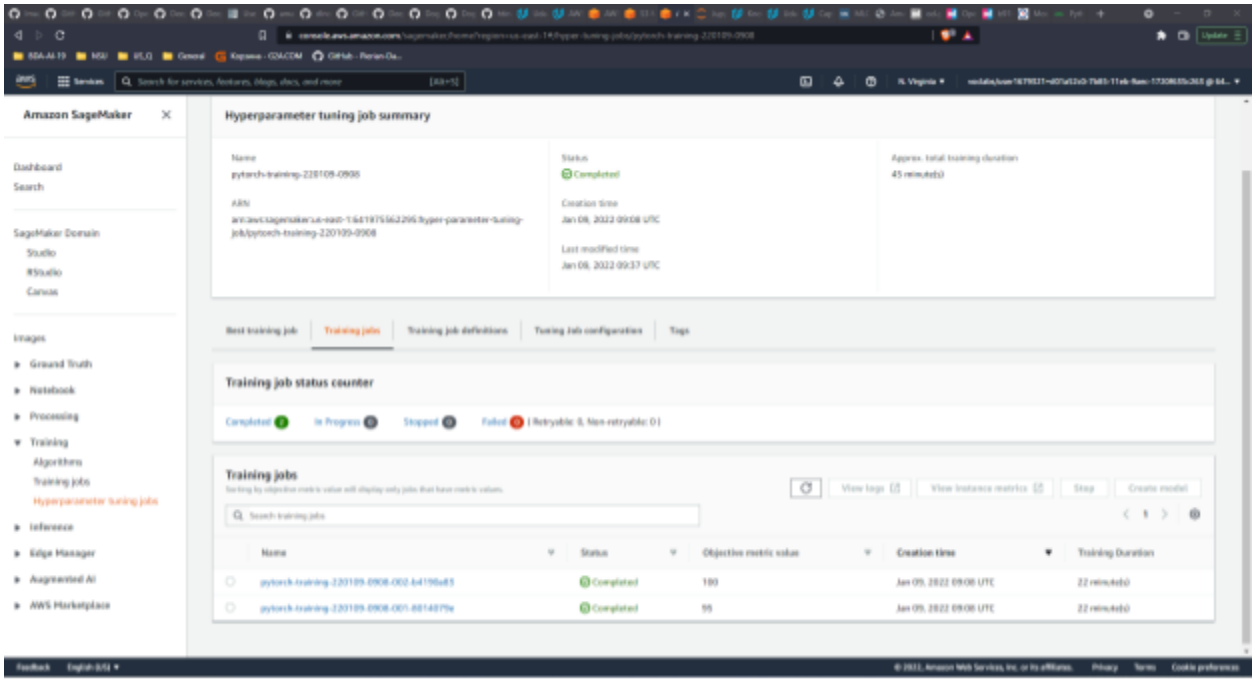

Fig 8: Hyperparameter tuning jobs

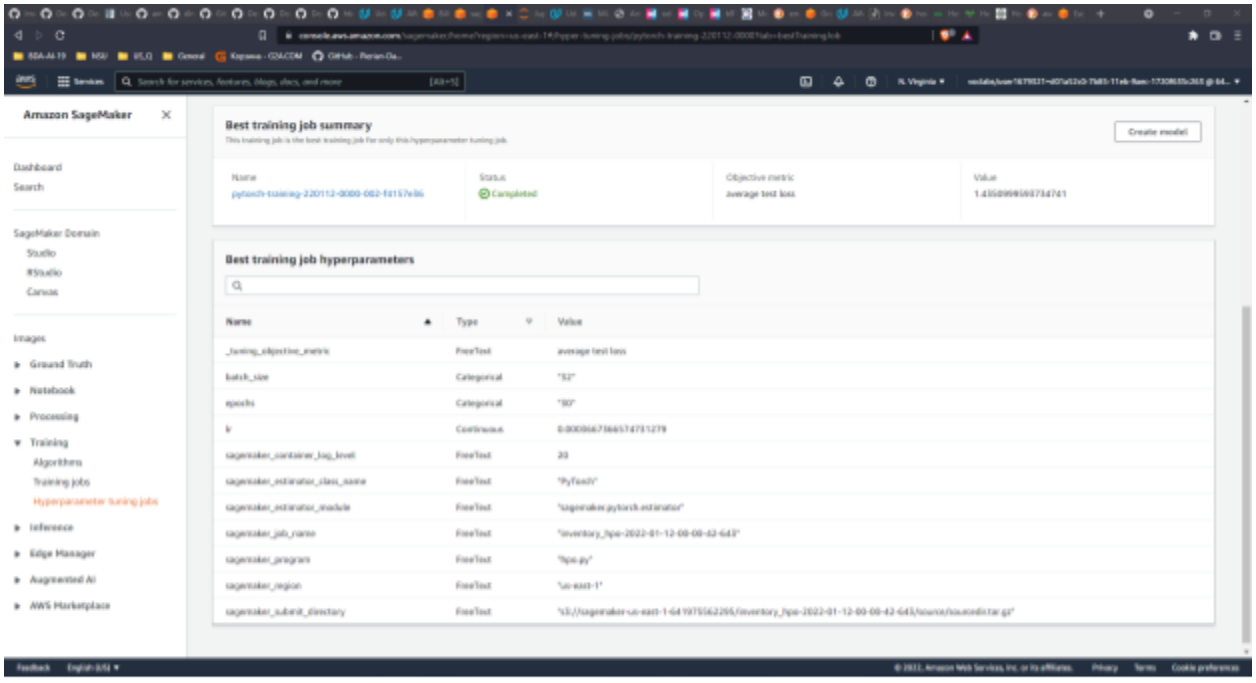

Fig 9: Best Training job

Training on AWS Sagemaker

Using the best hyperparameters obtained from the hyperparameter tuning job (described earlier), the sagemaker estimator is used to train the model. I used the `ml.g4dn.xlarge` instance type because it has **4 vCPUs**, **16 GiB** of memory and also includes a **GPU** for faster and accelerated training. I used a single instance count for the training and used the file **train.py** as entry point.
For the estimator to have access to the training and test data stored in the S3 bucket, we set up variables that map input channel and output storage to point the estimator to the data sources. The environment variables are listed below:

- SM_CHANNEL_TRAINING: points to the training data location in S3 bucket
- SM_CHANNEL_TESTING: points to the test data location in the S3 bucket
- SM_CHANNEL_MODEL_DIR: the location in S3 where model will be saved
- SM_OUTPUT_DATA_DIR: the location for storing output data

The hook for debugging is set to record the loss in both training and testing. The profiler report can be found in the Profiler report folder. Below is the loss plot from the debugger output: however, the plot shows only the training loss because I set an unreasonable count thus I plotted the validation loss separately.

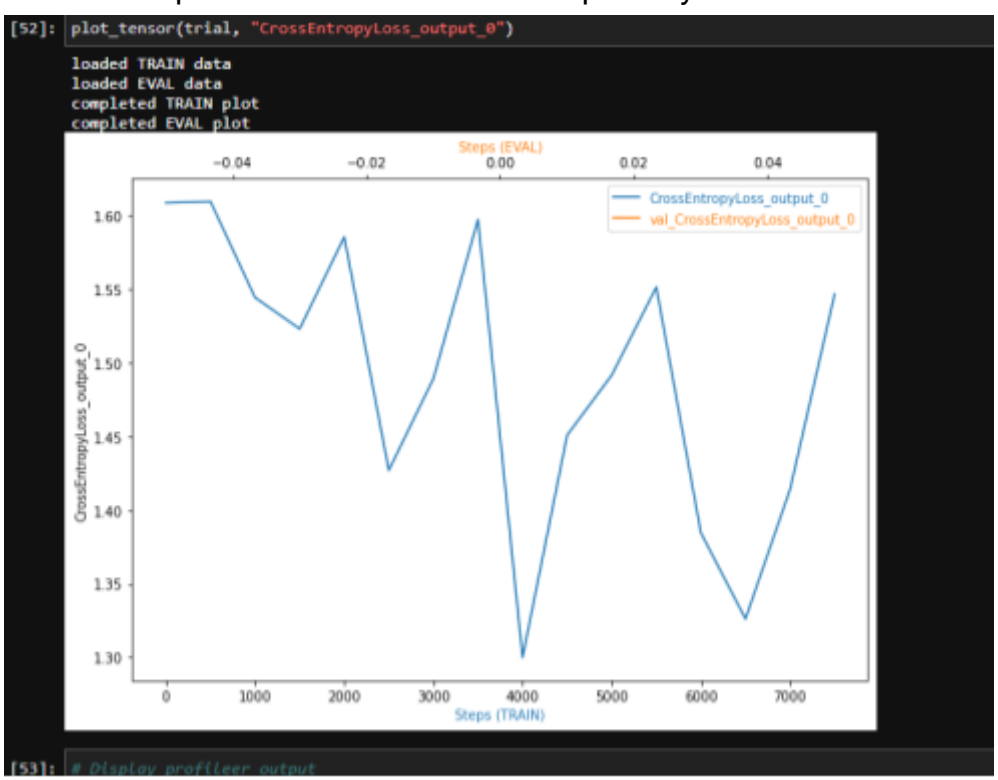

Fig 10: hook loss plot

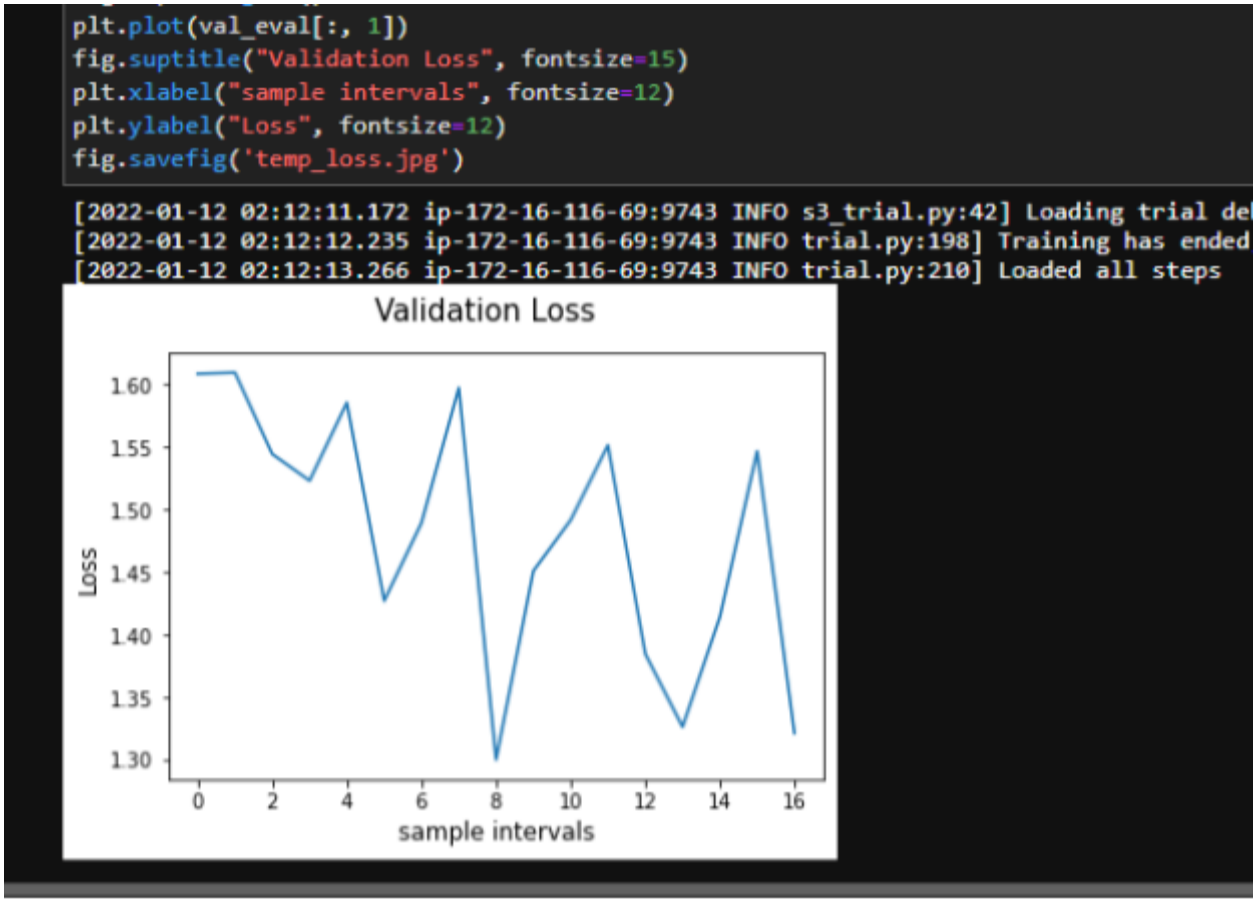

Fig 11: Validation loss plot

Model Deployment to an Endpoint

During training the model was saved to S3. We first get the saved model from S3 using `estimator.model_location`. We pass in the model_location, the sagemaker role, and use the script `inference.py` as entry point. We created a class called `ImagePredictor` that inherits the sagemaker predictor class that deserializes the image because we read the image as a json object, which is what the sagemaker endpoint expects as input. We use the class as the `predictor_cls` parameter for the sagemaker pytorch model. We then deploy to the sagemaker endpoint using a single instance count and the `ml.m5.large` instance type. Below is a sample code used to define the predictor and also deploy it.

```
pytorch_model = PyTorchModel(model_data=model_location,
role=role, entry_point='inference.py',py_version='py3',
framework_version='1.4', predictor_cls=ImagePredictor)

predictor = pytorch_model.deploy(initial_instance_count=1,
instance_type='ml.m5.large')
```

To query the endpoint, we first use the python request library to download an image from the original datasource which is not part of our training and test dataset. We use the predict function from the sagemaker pytorch model to predict how many items are in the given image. The output is an array of numbers. Thus we use `numpy.argmax` to get the actual number of items (single number) in the given image. Below is an image of the created endpoint in sagemaker and an output inference.

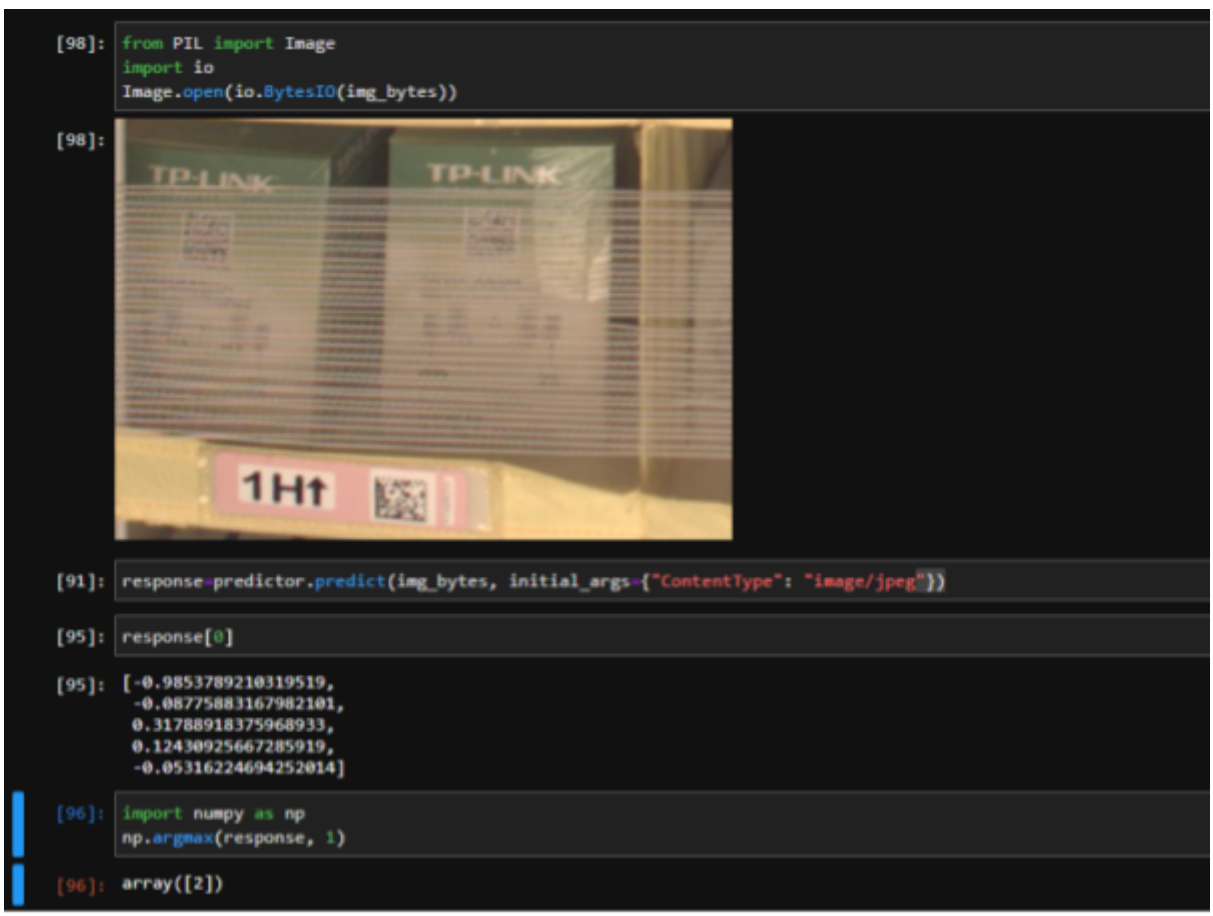

Fig 12: endpoint inference

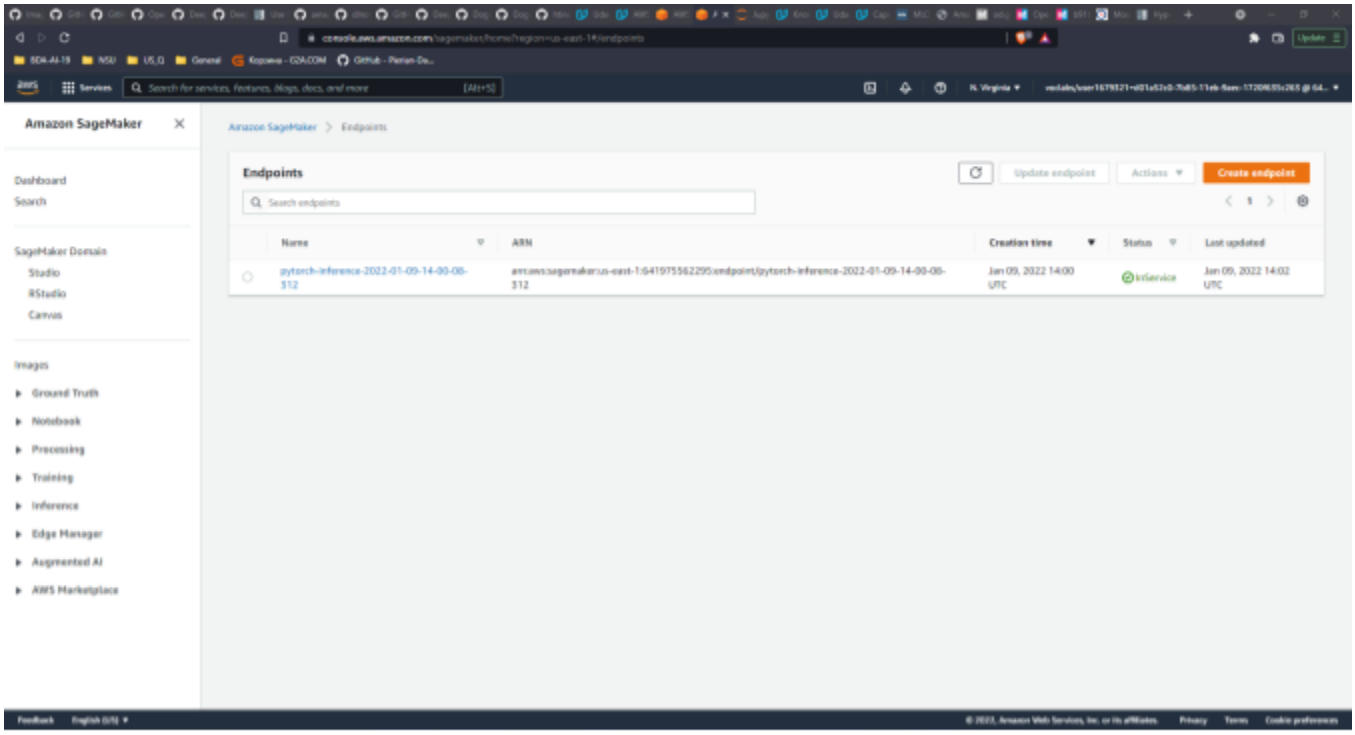

Fig 13: active endpoint

Training on EC2 instance

To reduce the cost of training, I used an EC2 spot instance for my training. I chose the **t2.medium** as an instance type. This instance costs $0.0464 with **2 vCPUs** and **4 GB** of memory and it has low to moderate performance. After creating and launching the spot instance, I activated the latest pytorch model entering `source activate pytorch_latest_p37` in the launched bash shell. In the terminal I used vim to create a file called **train.py** using `vim train.py`. In the opened vim editor, I used the command `:set paste` (this is done to keep the formatting of any pasted code) and also entered `i` to set it to insert mode. I pasted code from my **train.py** script and used the command `:wq!` to save and close the vim editor. To easily have access to my training data, I connected my ec2 instance to my s3 bucket by first creating an IAM role for my ec2 instance and attaching the AmazonS3FullAccess role (although this is not the most secured option but for the purpose of this project) to allow the ec2 instance to read from S3. First I installed the aws cli and verified if my ec2 instance has access to S3 using `aws s3 ls s3://<name-of-s3-bucket>`.

I used the command `aws s3 sync s3:// <name-of-s3-bucket> <my-local-directory>` to get my training and test dataset into my ec2 instance. To train the model, I run the script using `python train.py.` The ec2 spot instance training cost is lower than the sagemaker cost. For about 7 hours, it only cost $0.35 whilst the total cost of training on sagemaker is about $9.83 dollars. Even though sagemaker has a lot of different services, the main cost of running a sagemaker notebook is still more expensive than that of ec2 instance. Another reason that also accounted for the high in using sagemaker is because I used the instance type ml.g4dn.xlarge which has a GPU.

Multi-instance training

For multi-instance training, I used the `ml.g4dn.xlarge` instance. I selected this because it has **4 vCPUs**, **16 GiB** of memory and also includes a **GPU** for faster and accelerated training. For the actual training I used the **train.py** as entry point and used an instance count of 2. This will speed up training by training two instances at the same time. Below is the image showing the multi-instance jobs created in sagemaker.

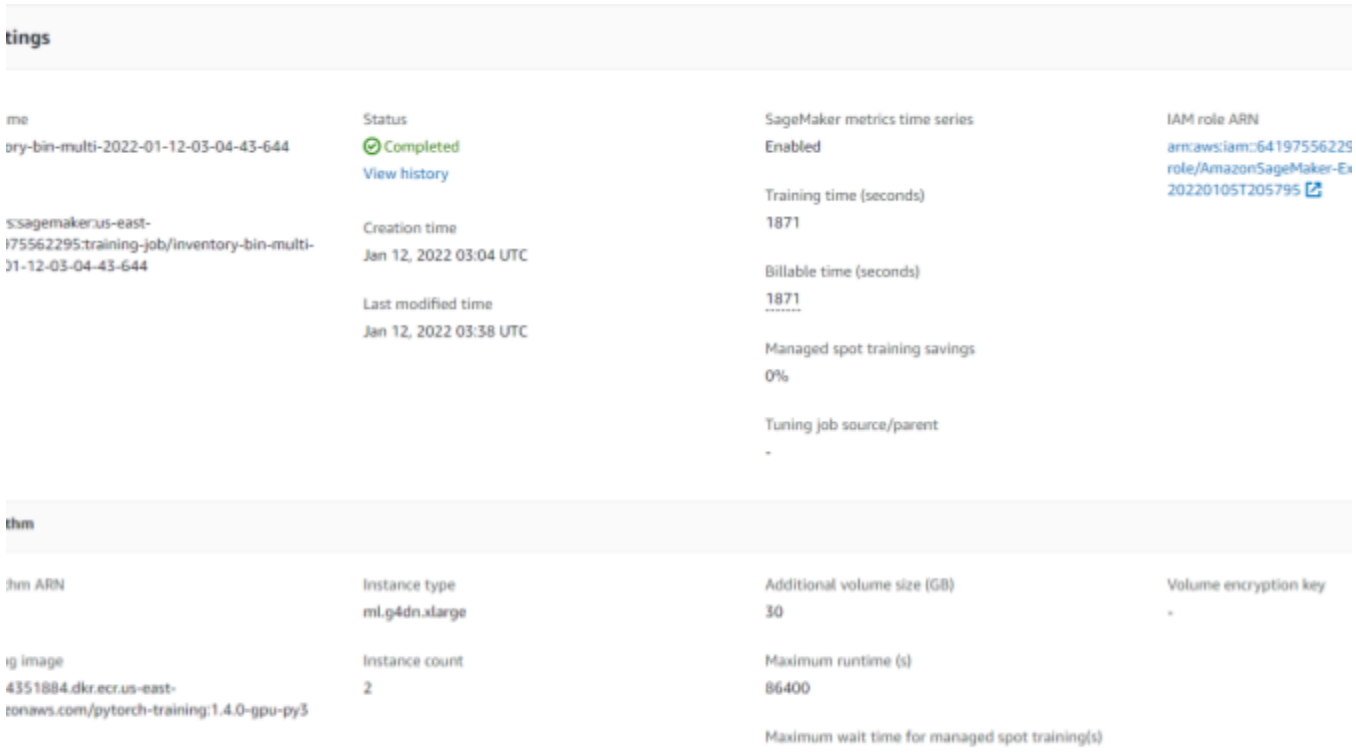

Fig 14: multi-instance training details

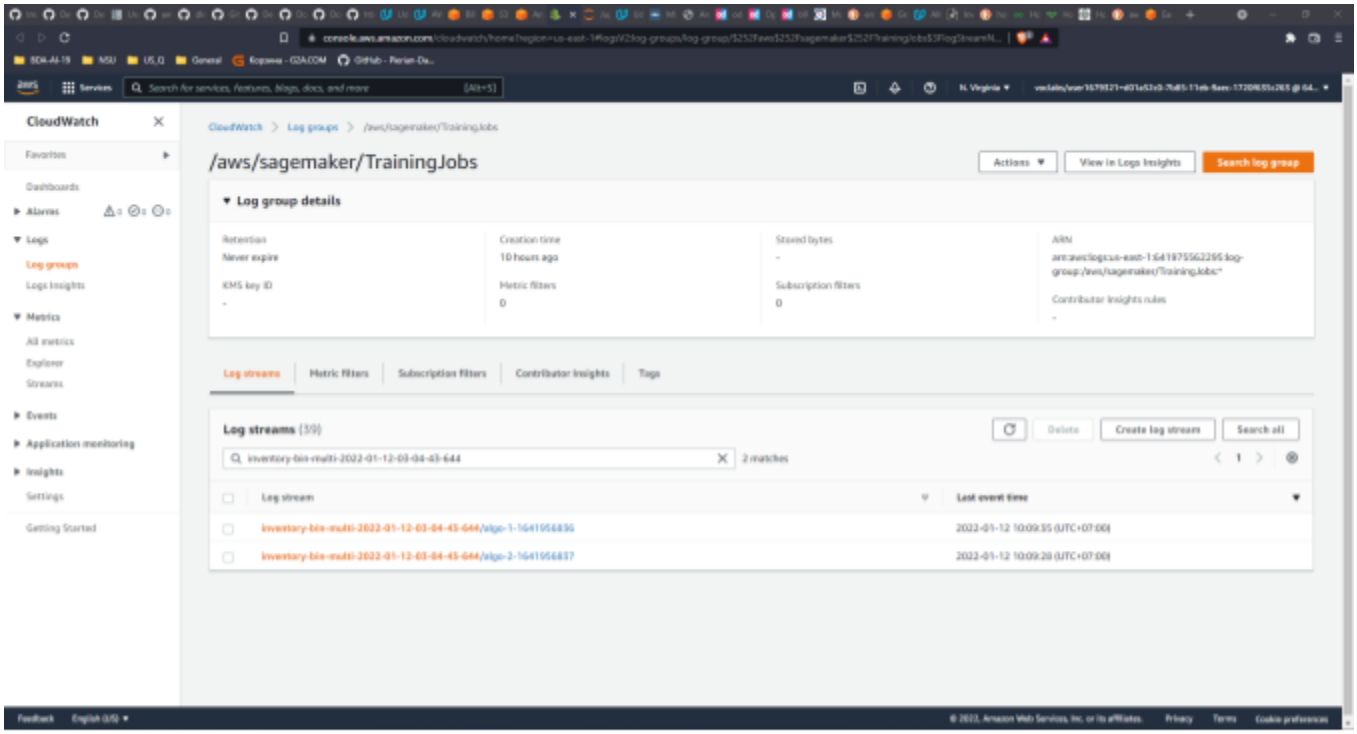

Fig 15: multi-instance training jobs

Results

The dataset used for this project was small as such, no validation set was used but only train set and test set was used. The hyperparameter tuning job provided us with the best hyperparameters which were used to train the model. The loss (cross entropy) decreased as expected. However, due to the small size of our dataset, the model accuracy could not beat the accuracy of the benchmark model.
Overall, our model achieved an accuracy of about 31.78% as compared to 55.67% of the benchmark model. However, our model beat the target accuracy we set (25%) because of the size of our dataset.
In future work, the augmentation technique called **mixup** as described in [7] can be used to generate extra data in order to improve the performance of our model even with our small dataset. Below is a table of the results compared with the benchmark model and their respective data sizes.

Description	Total size	Train size	Test size	Accuracy
Project image	10,441	8,352	2,089	31.78%
Benchmark images	535,234	481,711	53,523	55.67%

Refinement

In order to boost the performance of the model, a few augmentation techniques were used. Some of the augmentation techniques used include random rotations, color jitter which adjusts brightness of the image randomly, random vertical flip and random grayscale with probability of 0.2.
I also applied the mixup technique, another augmentation technique as described in [7] in order to improve the model performance.
Mixup is a data augmentation technique that generates weighted combinations of random image pairs from the training data. Given two images and their ground truth labels: (x_i, y_i) , (x_j, y_j) a synthetic training example $(\hat{x}, \hat{y})$ is generated as:

$$\hat{x} = \lambda x_i + (1 - \lambda) x_j$$
$$\hat{y} = \lambda y_i + (1 - \lambda) y_j$$

Where λ is independently sampled for each augmented example.
The mixup was applied to the batches obtained from the dataloaders during the training process. There was about 2% increase in the test accuracy results when the new techniques were applied. To have a comparison for the model against the benchmark results, I would use all the dataset provided for this task for training, apply another version of mixup technique called manifold mixup [8], different network models like vgg and increase the hyperparameter search space as well.
The jupyter notebook called `mixup.ipynb` contains local training results and the `mixup.py` file is what was used to train on an EC2 instance.

Below is a table comparing improvement in accuracy to the model described earlier with fewer augmentations.

Description	Accuracy
Initial model	31.78%
Benchmark model	55.67%
Refined model	33.41

** All images are stored in the screenshot folder

References

1. https://www.shipbob.com/blog/distribution-center/
2. Amazon Bin Image Dataset was accessed on 03-01-2022 from https://registry.opendata.aws/amazon-bin-imagery.
3. He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
4. Torrey, L. and Shavlik, J., 2010. Transfer learning. In Handbook of research on machine learning applications and trends: algorithms, methods, and techniques (pp. 242-264). IGI global.
5. Amazon Bin Image Dataset Challenge by silverbottlep
6. Amazon Inventory Reconciliation using AI by Pablo Rodriguez Bertorello, Sravan Sripada, Nutchapol Dendumrongsup.
7. Zhang, H., Cisse, M., Dauphin, Y.N. and Lopez-Paz, D., 2017. mixup: Beyond empirical risk minimization. *arXiv preprint* ***arXiv:1710.09412***.
8. Verma, V., Lamb, A., Beckham, C., Najafi, A., Mitliagkas, I., Lopez-Paz, D. and Bengio, Y., 2019, May. Manifold mixup: Better representations by interpolating hidden states. In *International Conference on Machine Learning* (pp. 6438-6447). PMLR.