Inventory Monitoring at Distribution Centers

AWS Machine Learning Nanodegree – Capstone Project Report

Author: Narmina Yadullayeva

Project Overview

The Amazon Fulfillment Centers are highly active centers for innovation, enabling Amazon to distribute hundreds of products every day. These products are randomly placed in bins, which are transported by robots. However, it is not uncommon for products to be misplaced during handling, resulting in discrepancies between the recorded inventory of a bin and its actual contents.

Figure 1: Amazon Robotics Fulfillment Center located in an industrial part of Troutdale near the Columbia River. [Source]

To build this project I will be using AWS SageMaker and good machine learning engineering practices to fetch data from a database, preprocess it, and then train a machine learning model. This project will serve as a demonstration of end-to-end machine learning engineering skills that I have learned as a part of this nanodegree.

Problem Statement

The objective of this project is to develop a deployable ML solution that can accurately count the number of objects in each bin. Such a system would be useful for inventory tracking and ensuring the correct number of items are included in delivery shipments.

There are several approaches that can be taken to solve this problem, for example one can:

- Build classification model where each image has an assigned class reflecting number of items in the bin (class 1 = 1 item, class 2 = 2 items and so on).
- Build a solution that uses object detection model to detect distinct objects on the image and then uses detection count as a prediction.

In this capstone project I've used image classification approach to this problem.

Evaluation Metrics

For classification problem I've used accuracy as a main performance metrics to track. For loss function, I've used cross-entropy which is one of the loss functions commonly used for multi-class classification problems.

Project Design

The goal of this project is to design a ML solution that will allow us to identify the number of objects in each image of bin. In order to build and deploy this ML model, I will be using transfer learning approach i.e. I will fine-tune pre-trained convolutional neural network on our dataset. One of the common pre-trained models used for image classification is ResNet models family. In the process of experimentation, I've used ResNet18 and ResNet50 models with the latter one yielding in relatively higher performance than the first one thus it has been used in final setup.

The framework of choice is Pytorch. It PyTorch is designed to provide good flexibility and high speeds for deep neural network implementation. My personal preference in PyTorch is because its design patterns are fairly simple to understand.

In this project AWS platform was extensively used on every stage of ML cycle:

S3 was used for training / test data storage
- Sagemaker Notebook Instance (t2.medium) was used for EDA, scripts submission and querying deployed endpoint.
- Sagemaker Training jobs were used for training multiple models using various configurations;
- Sagemaker Tuning job was used for hyperparameter optimization, Inference was used for model deployment to an endpoint.

The steps I will be following:
1. Download data (images and metadata) from Amazon S3 public storage (this will utilize 'file_list.json' file).
2. Put the data in an appropriate folder according to the number of objects contained in every image.
3. Split the dataset into training and test subsets and upload to S3.

4. Define pre-processing steps (transformations such as scaling and normalization) and visualize sample data prior and after preprocessing is applied.
5. Create a train_model.py script in order to train the model. In this script, I will be loading a pre-trained ResNet50 model and will modify FC in order to tune it to our specific dataset. Framework of choice: Pytorch.
6. Launch the training locally first and then using Sagemaker Training jobs. Evaluate results.
7. Use hyperparameter tuning to obtain the set of hyperparameters combination with the best model performance.
8. Train and deploy the model to an endpoint so we can start using it to identify the number of objects in new images. To do that, we will require a separate inference.py script to attach to deployed estimator that will allow it to load trained model artifact and obtain model predictions.

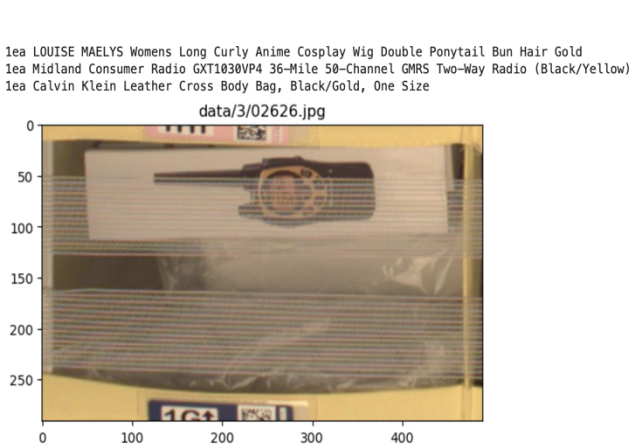
Data Exploration

To complete this project I will be using the Amazon Bin Image Dataset.

The Amazon Bin Image Dataset contains images and metadata from bins 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.

Amazon Fulfillment Technologies has made the bin images available free of charge on Amazon S3 to encourage recognition research in a variety of areas, including counting generic items and learning from weakly-tagged data. One can download and find the details at here.

Over 500,000 bin JPEG images and corresponding JSON metadata files describing items in the bin are available in the aft-vbi-pds S3 bucket in the us-east-1 AWS Region. 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. For example, the metadata for the image at https://aft-vbi-pds.s3.amazonaws.com/bin-images/523.jpg is found at https://aft-vbi-pds.s3.amazonaws.com/metadata/523.json.



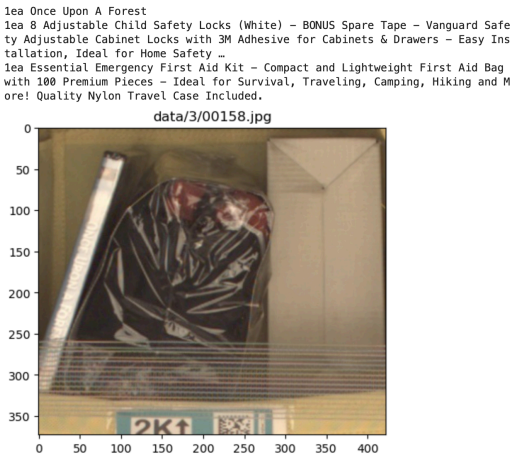


Figure 2: Example of images from the dataset

Since this is a large dataset, starter repo includes the list of small subset of the dataset (~10 000 out of 500 000 images). I will be using this subset to prevent any excess SageMaker credit usage.

To start with data exploration, I've downloaded both raw images as well as corresponding metadata and saved json object contents into the dataframe.

The json object has two main keys. The `BIN_FCSKU_DATA` key returns the data about the image like height, width, weight etc. of the objects in the image while the `EXPECTED_QUANTITY` contains how many items are contained in the bin.

```
meta_df = pd.read_csv("metadata/metadata.csv")
meta_df
```

	BIN_FCSKU_DATA	EXPECTED_QUANTITY	filename	filepath
0	{'B00O0WRO08': {'asin': 'B00O0WRO08', 'height'...	1	100313.jpg	data/1/100313.jpg
1	{'B00UO0UNOE': {'asin': 'B00UO0UNOE', 'height'...	1	09915.jpg	data/1/09915.jpg
2	{'B003JKT3DS': {'asin': 'B003JKT3DS', 'height'...	1	103299.jpg	data/1/103299.jpg
3	{'B00NG87OIY': {'asin': 'B00NG87OIY', 'height'...	1	00710.jpg	data/1/00710.jpg
4	{'B00U2UZ40Y': {'asin': 'B00U2UZ40Y', 'height'...	1	05397.jpg	data/1/05397.jpg
...	...	...	...	...
10436	{'0783811691': {'asin': '0783811691', 'height'...	5	100598.jpg	data/5/100598.jpg
10437	{'0061782661': {'asin': '0061782661', 'height'...	5	1042.jpg	data/5/1042.jpg
10438	{'B0161TCR9A': {'asin': 'B0161TCR9A', 'height'...	5	07628.jpg	data/5/07628.jpg
10439	{'B00DVQRJEK': {'asin': 'B00DVQRJEK', 'height'...	5	103667.jpg	data/5/103667.jpg
10440	{'B004ED815A': {'asin': 'B004ED815A', 'height'...	5	104002.jpg	data/5/104002.jpg

10441 rows × 4 columns

Let's look at metadata for a sample image:

```
image_meta
```

```
{'B005KDEIZ0': {'asin': 'B005KDEIZ0',
  'height': {'unit': 'IN', 'value': 1.6535433054},
  'length': {'unit': 'IN', 'value': 6.2204724346},
  'name': 'Rubbermaid Commercial Oven Thermometer, Stainless Steel, FGTH0550',
  'normalizedName': 'Rubbermaid Commercial Oven Thermometer, Stainless Steel, FGTH0550',
  'quantity': 5,
  'weight': {'unit': 'pounds', 'value': 0.13227600711644918},
  'width': {'unit': 'IN', 'value': 4.5669291292}}}
```

Moderate class imbalance can be observed when looking at labels distribution:

```
meta_df['EXPECTED_QUANTITY'].value_counts().sort_values()
```

```
1    1228
5    1875
2    2299
4    2373
3    2666
Name: EXPECTED_QUANTITY, dtype: int64
```

Next step was to split data into training and test datasets. To do that, I've created a function called `split_and_save_data` which uses stratified sampling technique with 80:20 split and fixed seed to sample datapoints for train/test subsets. Function iterates through initial folder and creates `train_data` folder with train and test subfolders which I've downloaded to S3.

```
def split_and_save_data(meta_df, output_path ="train_data", test_size = 0.20):

    train, test = train_test_split(meta_df, test_size=test_size, stratify=meta_df['EXPECTED_QUANTITY'], random_state = 0)

    for old_file_path, file_name, qty in train[['filepath','filename',"EXPECTED_QUANTITY"]].values:
        new_file_path = os.path.join(output_path,'train', str(qty))
        if not os.path.exists(new_file_path):
            os.makedirs(new_file_path)
        shutil.copyfile(old_file_path, os.path.join(new_file_path,file_name))

    for old_file_path, file_name, qty in test[['filepath','filename',"EXPECTED_QUANTITY"]].values:
        new_file_path = os.path.join(output_path,'test', str(qty))
        if not os.path.exists(new_file_path):
            os.makedirs(new_file_path)
        shutil.copyfile(old_file_path, os.path.join(new_file_path,file_name))

    train["new_filepath"] = train["filepath"].apply(lambda x: 'train_data/train/'+''.join(x.split("/")[1:]))
    test["new_filepath"] = test["filepath"].apply(lambda x: 'train_data/test/'+'/'.join(x.split("/")[1:]))

    train.to_csv(os.path.join(output_path,'train_meta.csv'), index=False)
    test.to_csv(os.path.join(output_path,'test_meta.csv'), index=False)

    return train, test
```

```
train, test = split_and_save_data(meta_df, output_path ="train_data", test_size = 0.20)
```

The last thing to do to complete preprocessing setup was to define required transformations for ResNet50 Convolutional Neural Network.

Note: default image size used in the beginning for scaling purposes was 224, but later I've used it as an additional hyperparameter to tune.

First, I've created a dummy data loader and calculated mean and standard deviation of training dataset. This allowed me to find statistics for normalization transformation to be applied for our dataset. Similar piece of code was included into train_model.py script to calculate mean and std based on any given training dataset thus making training script universal i.e. not tied up to our specific dataset.

```
train_transform = transforms.Compose([
        transforms.Resize((image_size,image_size)),
        transforms.ToTensor()
])

train_dataset = torchvision.datasets.ImageFolder(root=training_data_path, transform=train_transform)
train_dataset
```

```
Dataset ImageFolder
    Number of datapoints: 8352
    Root location: ./train_data/train/
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)
               ToTensor()
           )
```

```
# calculating approximation for mean and std
def get_mean_and_std(loader):
    mean = 0.
    std = 0.
    total_images_count = 0

    for images, _ in loader:
        image_count_in_a_batch = images.size(0)
        images = images.view(image_count_in_a_batch, images.size(1), -1)
        mean += images.mean(2).sum(0)
        std += images.std(2).sum(0)
        total_images_count += image_count_in_a_batch

    mean /= total_images_count
    std /= total_images_count

    return mean, std
```

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=False)

train_mean, train_std = get_mean_and_std(train_loader)
train_mean, train_std
```

```
(tensor([0.5232, 0.4421, 0.3526]), tensor([0.1355, 0.1196, 0.0899]))
```

```
# loading dataset into tv dataset object

train_transform = transforms.Compose([
        transforms.Resize((image_size,image_size)),
        transforms.RandomHorizontalFlip(),
        # transforms.RandomRotation(10),
        transforms.ToTensor(),
        transforms.Normalize(torch.Tensor(train_mean), torch.Tensor(train_std))
    ])

test_transform = transforms.Compose([
        transforms.Resize((image_size,image_size)),
        transforms.ToTensor(),
        transforms.Normalize(torch.Tensor(train_mean), torch.Tensor(train_std))
    ])

train_dataset = torchvision.datasets.ImageFolder(root=training_data_path, transform=train_transform)
test_dataset = torchvision.datasets.ImageFolder(root=test_data_path, transform=test_transform)

train_dataset
```

```
Dataset ImageFolder
    Number of datapoints: 8352
    Root location: ./train_data/train/
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)
               RandomHorizontalFlip(p=0.5)
               ToTensor()
               Normalize(mean=tensor([0.5232, 0.4421, 0.3526]), std=tensor([0.1355, 0.1196, 0.0899]))
           )
```

I've also included visualization of processed sample data – as you can see, images look completely different and might not seem to become clearer from person's perspective, but in reality, normalization is one of the most crucial preprocessing steps for model training to be more effective.

Benchmark Model

After doing some research, I have found a couple of GitHub projects associated with the same problem and dataset:

- https://github.com/pablo-tech/Image-Inventory-Reconciliation-with-SVM-and-CNN
- https://github.com/silverbottlep/abid_challenge

In both of the projects, overall accuracy on the test set was ~55% and overall RMSE ~0.90.

In this project I will be aiming to reproduce similar results. However, as I will be using only a fraction of dataset from provided file list (10 000 = ~ 2% of 500 000), it is likely that I won't be able to achieve same accuracy.

Experimentation

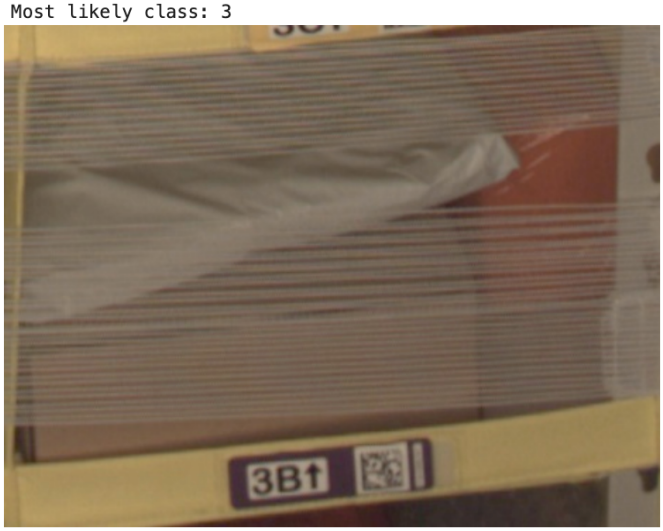
- ### Testing training script locally

Train_model.py script contains all necessary functions to conduct data loading / preprocessing and model training. First, I've trained the model locally verifying code integrity and tested saved model by writing auxiliary functions for inference:

```
def model_fn(model_dir):
    model = torch.load(os.path.join(model_dir, "model.pth"), map_location=device)
    model.eval()
    model.to(device)
    return model

def input_fn(iobytes):
    img = Image.open(iobytes)

    preprocess = transforms.Compose(
        [
            transforms.Resize((image_size,image_size)),
            transforms.ToTensor(),
            transforms.Normalize(train_mean, train_std),
        ]
    )
    input_tensor = preprocess(img)
    input_batch = input_tensor.unsqueeze(0)  # create a mini-batch as expected by the model
    return input_batch.to(device)


def predict_fn(image, model):
    image = input_fn(image)
    return model(image)
```

```
sample_test_image = test.sample()['new_filepath'].values[0]
print(sample_test_image)

f = open(sample_test_image, "rb")

prediction = predict_fn(f,predictor).cpu().detach().numpy()
print(prediction)
most_likely_class_num = np.argmax(prediction)
most_likely_class = get_keys_from_value(train_dataset.class_to_idx,most_likely_class_num)[0]

print("Most likely class: {}".format(most_likely_class))

image = Image.open(sample_test_image)
image
```

```
train_data/test/3/04382.jpg
[[-0.58978057  0.61877054  0.6425688   0.34781188 -0.33548206]]
Most likely class: 3
```



After verifying all works well, training and test data was uploaded into a separate S3 bucket:

Downloading dataset to S3 bucket

```
sagemaker_session = sagemaker.Session()
bucket = sagemaker_session.default_bucket()
role = sagemaker.get_execution_role()
```

```
prefix = "capstone_project/train_data"
data_path = "train_data"

## this lines are commented out as I've downloaded this to s3 already
inputs = sagemaker_session.upload_data(path=data_path, bucket=bucket, key_prefix=prefix)
print("input spec (in this case, just an S3 path): {}".format(inputs))
```

```
input spec (in this case, just an S3 path): s3://sagemaker-us-east-1-976414713425/capstone_project/train_data
```

```
## saved it here
inputs = "s3://sagemaker-us-east-1-976414713425/capstone_project/train_data"
```

- Submitting Training Job to AWS

First training job was completed after 10 epochs and achieved 30% accuracy on test dataset.

Running training job

```
# Declaring model training hyperparameter.
hyperparameters = {"epochs": "10", "batch-size": "64", "lr": "0.01", "momentum": "0.5", "image-size": "224"}
```

```
#TODO: Create your training estimator
from sagemaker.pytorch import PyTorch
from sagemaker import get_execution_role

estimator = PyTorch(
    entry_point="train_model.py",
    base_job_name="sagemaker-script-mode",
    role=get_execution_role(),
    instance_count=1,
    instance_type = "ml.g4dn.xlarge",
    hyperparameters=hyperparameters,
    framework_version="1.8",
    py_version="py36",
)
```

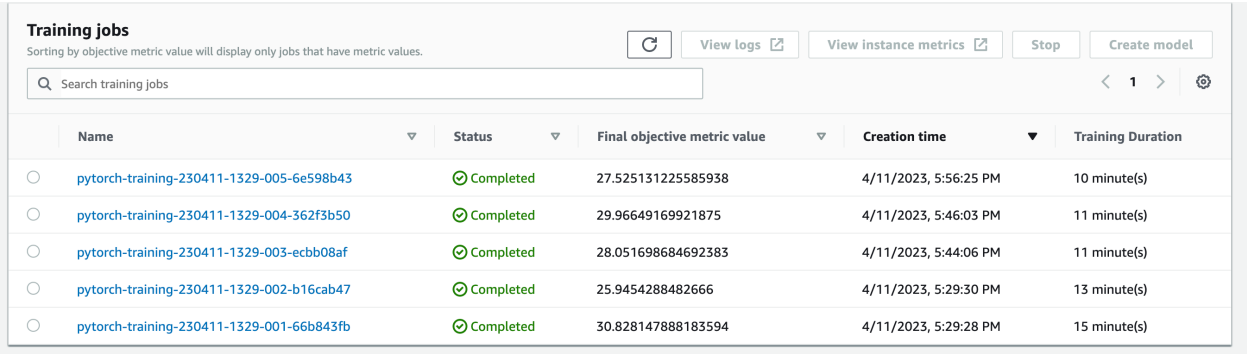
```
# TODO: Fit your estimator
estimator.fit({"training": inputs}, wait=True)
```

```
.............
INFO:__main__:Images [1280/8352 (15%)] Loss: 1.4211 Accuracy: 467/1280 (36.4844%)
INFO:__main__:Images [2560/8352 (31%)] Loss: 1.3718 Accuracy: 913/2560 (35.6641%)
INFO:__main__:Images [3840/8352 (46%)] Loss: 1.4533 Accuracy: 1360/3840 (35.4167%)
INFO:__main__:Images [5120/8352 (61%)] Loss: 1.3719 Accuracy: 1792/5120 (35.0000%)
INFO:__main__:Images [6400/8352 (77%)] Loss: 1.6136 Accuracy: 2243/6400 (35.0469%)
INFO:__main__:Images [7680/8352 (92%)] Loss: 1.3113 Accuracy: 2695/7680 (35.0911%)
INFO:__main__:Epoch Training Loss: 1.4312; Epoch Training Accuracy: 35.285
INFO:__main__:Epoch 9, Phase valid
INFO:__main__:Images [1280/2089 (61%)] Loss: 1.6116 Accuracy: 395/1280 (30.8594%)
INFO:__main__:Epoch Test Loss: 1.4964; Epoch Test Accuracy: 30.0144
INFO:__main__:--- Testing model ---
INFO:__main__:--- Testing model on the whole test dataset ---
INFO:__main__:Test set accuracy: 30.014360938247965, Test set average loss: 1.4964461016164208
INFO:__main__:--- Saving model to /opt/ml/model ---
2023-04-09 12:26:00,819 sagemaker-training-toolkit INFO     Reporting training SUCCESS

2023-04-09 12:26:21 Uploading - Uploading generated training model
2023-04-09 12:26:21 Completed - Training job completed
Training seconds: 1101
Billable seconds: 1101
```

- Hyperparameter Tuning

Next step was to run hyperparameter tuning to find an optimum values to use for learning rate, batch size, momentum, and image size. I've created a separate script called hpo.py to be used for tuning. Number of epochs was reduced to 5 in order to reduce cost of training jobs ("ml.g4dn.xlarge" instance type was used).

"lr": ContinuousParameter(0.001, 0.1),
"batch-size": CategoricalParameter([32, 64, 128, 256, 512]),
"momentum": ContinuousParameter(0.5, 0.9),
"image-size": CategoricalParameter([128, 224, 256]),



The best set of hyperparameters from limited number of training jobs was the one with ~30.83% test accuracy achieved after 5 epochs:

Model-type: Resnet50

'batch-size': '"128"',
'image-size': '"256"',
'lr': '0.004656335826920995',
'momentum': '0.6998010571771176',

- Profiler & Debugger

Best hyperparameter configuration was used for training final model with Profiler and Debugger analysis included. Profiler report can be found in ProfilerReport folder. Some of the key recommendations from Debugger can be found below:

Error name	Description	Recommendation
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.
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.
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	Change the number of data loader processes.

Error name	Description	Recommendation
	GPU utilization. If too large, it might impact other compute intensive operations on CPU.	
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.
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.
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.
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.
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.
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.

Model Deployment and Testing

Final model was deployed to inference endpoint and tested on a couple of test samples:

Amazon SageMaker > Endpoints > pytorch-inference-2023-04-10-16-35-59-072

pytorch-inference-2023-04-10-16-35-59-072

Endpoint settings

Name	Status	Type	URL
pytorch-inference-2023-04-10-16-35-59-072	InService	Real-time	https://runtime.sagemaker.us-east-1.amazonaws.com/endpoints/pytorch-inference-2023-04-10-16-35-59-072/invocations
ARN	**Creation time**	**Last updated**	Learn more about the API
arn:aws:sagemaker:us-east-1:976414713425:endpoint/pytorch-inference-2023-04-10-16-35-59-072	Mon Apr 10 2023 20:35:59 GMT+0400 (Azerbaijan Standard Time)	Mon Apr 10 2023 20:40:00 GMT+0400 (Azerbaijan Standard Time)	

```
train_data/test/4/104332.jpg
1ea Avery Durable Binder with 3-Inch Slant Ring, Holds 8.5 x 11-Inch Paper, Burgundy, 1 Binder (27652)
2ea Half-Strip Electric Stapler
1ea Umbro Women's Seamless Bikini Panties 3 Pack - Fuchsia/Lavender Assorted - Large
[[-1.51156998  0.3350271   1.27345514  0.57719386  0.20296389]]
Most likely class: 3
```

Figure 3 Example of correct prediction

```
train_data/test/5/102503.jpg
4ea Sony MDRAS200 Active Sports Headphones (White)
1ea Louis Garneau Women's 2015 Equipe Long Sleeve Cycling Jersey (Black/White, X-Large)
[[-2.05087328 -0.25515145  0.7236805   1.07355046  1.11097896]]
Most likely class: 5
```

Figure 4 Example of correct prediction

```
train_data/test/2/105038.jpg
1ea Soft Scrub 1734778 Toilet Care, Sapphire Waters (Pack of 10)
1ea Egiant- Macbook Pro 13/13.3 Inch (A1425/A1502) New Case - Rubberized Hard Shell Protective Case with Soft Keyboard Cover Skin For Macbook Pro 13/13.3 Inch
With Retina Display (Wine)
[[-0.64114732  0.71583951  0.27874669  0.16952883 -0.07929881]]
Most likely class: 2
```

Figure 5 Example of incorrect prediction

Results

During the experimentation phase, multiple configurations were tested and compared based on performance metrics. Below table contains experimentation summary which includes several model types and different hyperparameter configurations:

- First stage was to experiment with different sizes of models from Resnet family (Resnet18, Resnet50, Resnet101) having rest of HP fixed -> Resnet50 showed moderately better results amongst the others and was used as a base model for HPO.

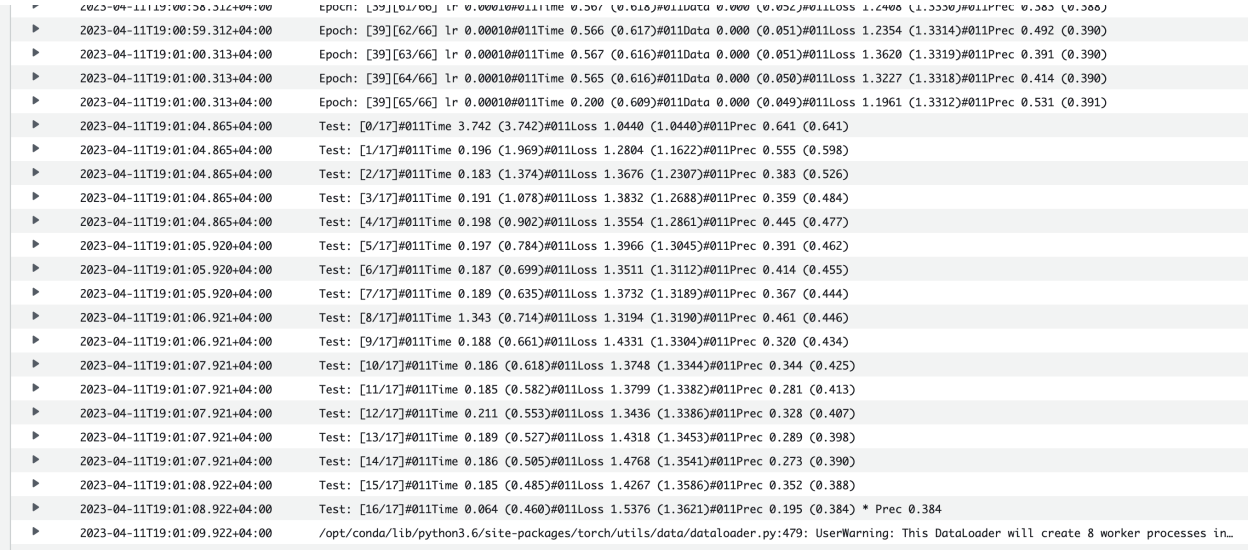
- Next stage was to run hyperparameter optimization -> Exp8_hpo showed comparatively better results and its HP configuration was used for final model.
- Last stage was to run final model training for 40 epochs and compare with benchmark model.

Table 1 Experimentation summary

Name	Model used	Batch size	Epochs	Image size	Learning rate	Momentum	Accuracy	Time (min)
Initial experiments with 3 architectures								
Exp1	ResNet18	64	10	224	0.01	0.5	30.014360938247965	24
Exp2	ResNet50	64	10	224	0.01	0.5	31.64193393968406	25
Exp3	ResNet101	64	10	224	0.01	0.5	30.876017233125896	27
Hyperparameter tuning using Resnet50 for 5 epochs only								
Exp4_hpo	ResNet50	128	5	224	0.04215797346056457	0.6621556854645394	27.525131225585938	10
Exp5_hpo	ResNet50	64	5	256	0.032824686350117864	0.8612426863599609	29.96649169921875	11
Exp6_hpo	ResNet50	512	5	256	0.002423255249470192 4	0.8006698202730027	28.051698684692383	11
Exp7_hpo	ResNet50	512	5	224	0.001676763425190747	0.5549096783618431	25.9454288482666	13
Exp8_hpo	ResNet50	128	5	256	0.0046563358269209 95	0.6998010571771176	30.828147888183594	15
Final Model								
Exp9_best	ResNet50	128	40	256	0.004656335826920995	0.6998010571771176	0.314026	40
Benchmark	-	-	40	-	-	-	0.384	40

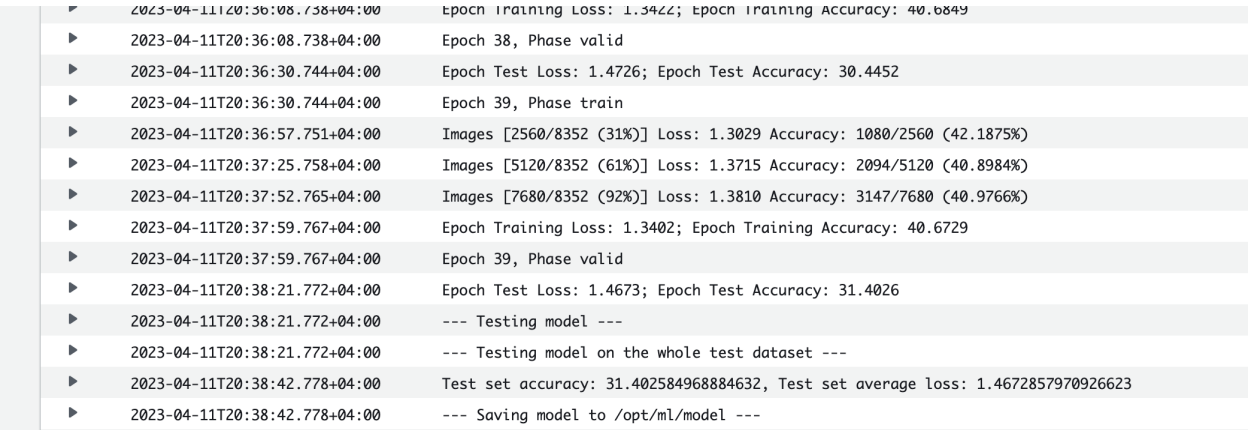
When it comes to comparing best model to the benchmark (Pablo et al), more thorough analysis need to be performed as training / test datasets used for benchmark model training and evaluation were different from the one I am working with in the context of this capstone project. To make comparison fair, I took the training script from Github repo (source) and run training of the benchmark architecture using our small dataset – details can be found in the script 'train_benchmark.py':

- 31.5% accuracy score was achieved by final model after training for 40 epochs;
- 38.4% accuracy score was achieved by benchmark model after training for 40 epochs – see screenshot below:



2023-04-11T19:00:59.312+04:00	Epoch: [39][62/66] lr 0.00010#011Time 0.566 (0.617)#011Data 0.000 (0.051)#011Loss 2.2354 (1.3314)#011Prec 0.492 (0.390)
2023-04-11T19:01:00.313+04:00	Epoch: [39][63/66] lr 0.00010#011Time 0.567 (0.616)#011Data 0.000 (0.051)#011Loss 1.3620 (1.3319)#011Prec 0.391 (0.390)
2023-04-11T19:01:00.313+04:00	Epoch: [39][64/66] lr 0.00010#011Time 0.565 (0.616)#011Data 0.000 (0.050)#011Loss 1.3227 (1.3318)#011Prec 0.414 (0.390)
2023-04-11T19:01:00.313+04:00	Epoch: [39][65/66] lr 0.00010#011Time 0.200 (0.609)#011Data 0.000 (0.049)#011Loss 1.1961 (1.3312)#011Prec 0.531 (0.391)
2023-04-11T19:01:04.865+04:00	Test: [0/17]#011Time 3.742 (3.742)#011Loss 1.0440 (1.0440)#011Prec 0.641 (0.641)
2023-04-11T19:01:04.865+04:00	Test: [1/17]#011Time 0.196 (1.969)#011Loss 1.2804 (1.1622)#011Prec 0.555 (0.598)
2023-04-11T19:01:04.865+04:00	Test: [2/17]#011Time 0.183 (1.374)#011Loss 1.3676 (1.2307)#011Prec 0.383 (0.526)
2023-04-11T19:01:04.865+04:00	Test: [3/17]#011Time 0.191 (1.078)#011Loss 1.3832 (1.2688)#011Prec 0.359 (0.484)
2023-04-11T19:01:04.865+04:00	Test: [4/17]#011Time 0.198 (0.902)#011Loss 1.3554 (1.2861)#011Prec 0.445 (0.477)
2023-04-11T19:01:05.920+04:00	Test: [5/17]#011Time 0.197 (0.784)#011Loss 1.3966 (1.3045)#011Prec 0.391 (0.462)
2023-04-11T19:01:05.920+04:00	Test: [6/17]#011Time 0.187 (0.699)#011Loss 1.3511 (1.3112)#011Prec 0.414 (0.455)
2023-04-11T19:01:05.920+04:00	Test: [7/17]#011Time 0.189 (0.635)#011Loss 1.3732 (1.3189)#011Prec 0.367 (0.444)
2023-04-11T19:01:06.921+04:00	Test: [8/17]#011Time 1.343 (0.714)#011Loss 1.3194 (1.3190)#011Prec 0.461 (0.446)
2023-04-11T19:01:06.921+04:00	Test: [9/17]#011Time 0.188 (0.661)#011Loss 1.4331 (1.3304)#011Prec 0.320 (0.434)
2023-04-11T19:01:07.921+04:00	Test: [10/17]#011Time 0.186 (0.618)#011Loss 1.3748 (1.3344)#011Prec 0.344 (0.425)
2023-04-11T19:01:07.921+04:00	Test: [11/17]#011Time 0.185 (0.582)#011Loss 1.3799 (1.3382)#011Prec 0.281 (0.413)
2023-04-11T19:01:07.921+04:00	Test: [12/17]#011Time 0.211 (0.553)#011Loss 1.3436 (1.3386)#011Prec 0.328 (0.407)
2023-04-11T19:01:07.921+04:00	Test: [13/17]#011Time 0.189 (0.527)#011Loss 1.4318 (1.3453)#011Prec 0.289 (0.398)
2023-04-11T19:01:07.921+04:00	Test: [14/17]#011Time 0.186 (0.505)#011Loss 1.4768 (1.3541)#011Prec 0.273 (0.390)
2023-04-11T19:01:08.922+04:00	Test: [15/17]#011Time 0.185 (0.485)#011Loss 1.4267 (1.3586)#011Prec 0.352 (0.388)
2023-04-11T19:01:08.922+04:00	Test: [16/17]#011Time 0.064 (0.460)#011Loss 1.5376 (1.3621)#011Prec 0.195 (0.384) * Prec 0.384
2023-04-11T19:01:09.922+04:00	/opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py:479: UserWarning: This DataLoader will create 8 worker processes in…

Figure 6 Final logs from benchmark model training



2023-04-11T20:36:08.738+04:00	Epoch Training Loss: 1.3422; Epoch Training Accuracy: 40.6849
2023-04-11T20:36:08.738+04:00	Epoch 38, Phase valid
2023-04-11T20:36:30.744+04:00	Epoch Test Loss: 1.4726; Epoch Test Accuracy: 30.4452
2023-04-11T20:36:30.744+04:00	Epoch 39, Phase train
2023-04-11T20:36:57.751+04:00	Images [2560/8352 (31%)] Loss: 1.3029 Accuracy: 1080/2560 (42.1875%)
2023-04-11T20:37:25.758+04:00	Images [5120/8352 (61%)] Loss: 1.3715 Accuracy: 2094/5120 (40.8984%)
2023-04-11T20:37:52.765+04:00	Images [7680/8352 (92%)] Loss: 1.3810 Accuracy: 3147/7680 (40.9766%)
2023-04-11T20:37:59.767+04:00	Epoch Training Loss: 1.3402; Epoch Training Accuracy: 40.6729
2023-04-11T20:37:59.767+04:00	Epoch 39, Phase valid
2023-04-11T20:38:21.772+04:00	Epoch Test Loss: 1.4673; Epoch Test Accuracy: 31.4026
2023-04-11T20:38:21.772+04:00	--- Testing model ---
2023-04-11T20:38:21.772+04:00	--- Testing model on the whole test dataset ---
2023-04-11T20:38:42.778+04:00	Test set accuracy: 31.402584968884632, Test set average loss: 1.4672857970926623
2023-04-11T20:38:42.778+04:00	--- Saving model to /opt/ml/model ---

Figure 7 Training logs from final model

Given the project's credit and time limitations I found achieved accuracy metrics acceptable (31.5% vs 38.4%).