Object Counting on Amazon Bin Image Dataset

1. Background

Supply chain management is essential in economics as it allows enterprises to:

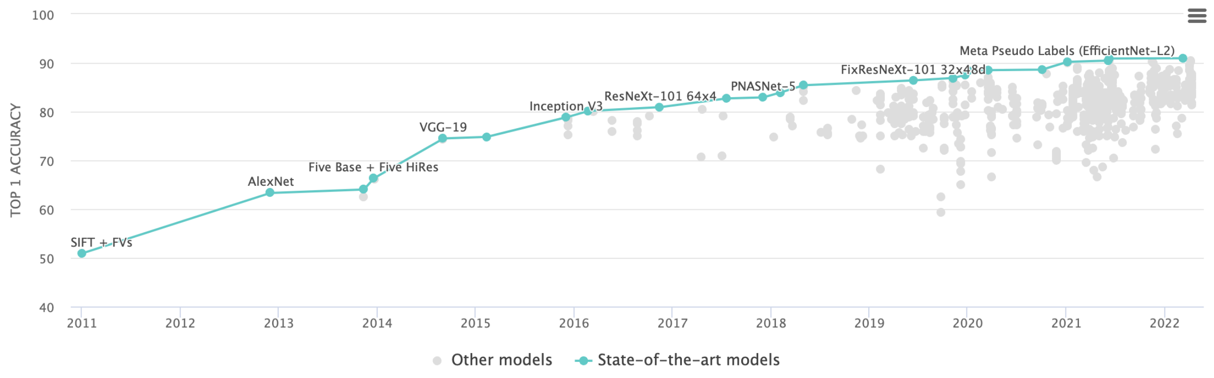
1. Achieve better resource monitoring and planning
2. Eliminate redundant transportation costs
3. Satisfy customers to give a trace-ability for their delivery

Since the invention of the Barcode and its patent in 1951, it has been used for inventory management and is still useful as we experience in our daily lives like in-store checkouts. However, reading barcodes printed on the products is time-consuming because we need to find the exact location of the barcodes. At every checkpoint, reading barcodes costs lots of human labor.

As a technically better solution, the first ancestor of modern RFID was patented in 1973. Like FasTrak on highways, RFID tags can be read automatically without human intervention. However, RFID tags are more expensive than printed barcodes so they increase the final product cost. In addition, RFID tags have environmental side effects because they are microchips that produce electronic waste. Due to the limitation, they are often attached manually to the assets during the tracking period and detached to be reused for other assets later.

During the last decade, image classification using machine learning has significantly improved a lot from 50% to 90% in terms of classification accuracy.

https://paperswithcode.com/sota/image-classification-on-imagenet



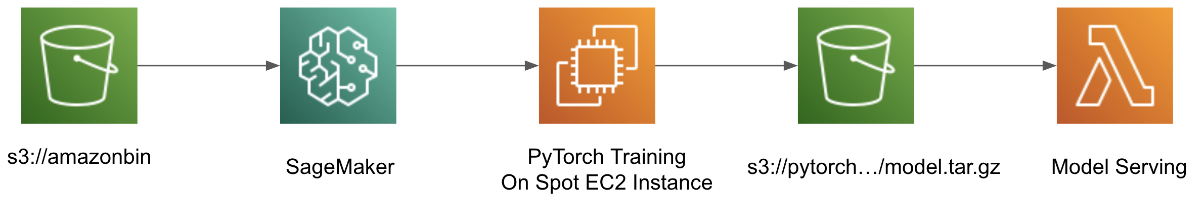
The improved vision-based machine learning can disrupt the existing supply chain operating practices by introducing new ways to save costs.

2. Problem Statement

Although there are many published pre-trained models based on ImageNet, it is not easy to apply to real-world problems due to the lack of real-domain data or computing resources. To achieve a reasonably good accuracy within the given project timeline in the real-world environment, we need to narrow down the focus.

Amazon provides a good real-world image data set for object recognition and counting problem. As one of the simplified versions of object recognition, this project focuses on the Object Counting problem on the Amazon Bin Image Dataset instead of identifying the objects by reading barcodes or RFID.

I propose to build an object counting service on AWS using a pre-trained Continuous Neural Network (CNN) model. The following is the overall architecture.



- **Input**: Amazon Bin Image Dataset is used as the training data.
- **Machine Learning**: Transfer learning based on pre-trained CNN Models from the PyTorch library.
- **Training**: SageMaker is used to train PyTorch models from the training data on S3
- **Model**: S3 is used to store the trained model
- **Serving**: Lambda is used to serve the trained model as a service
- **Output**: Service will count objects from 0 to 12. According to the EDA result, we are focusing on 95% of image data which belongs to 0 ~ 12.

The performance of my model will be determined by its accuracy.

3. Datasets: Amazon Bin Image Dataset

The Amazon Bin Image Dataset contains over 500,000 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 carrying pods as part of normal Amazon Fulfillment Center operations. This dataset has many images and the corresponding metadata.

3.1 Documentation

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

3.2 Download

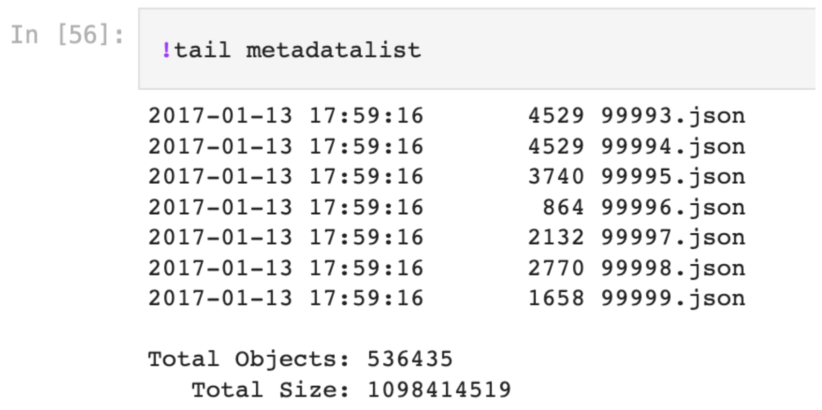
- https://registry.opendata.aws/amazon-bin-imagery/
- https://github.com/awslabs/open-data-registry/blob/main/datasets/amazon-bin-imagery.yaml

3.3 License

- Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States (CC BY-NC-SA 3.0 US) https://creativecommons.org/licenses/by-nc-sa/3.0/us/

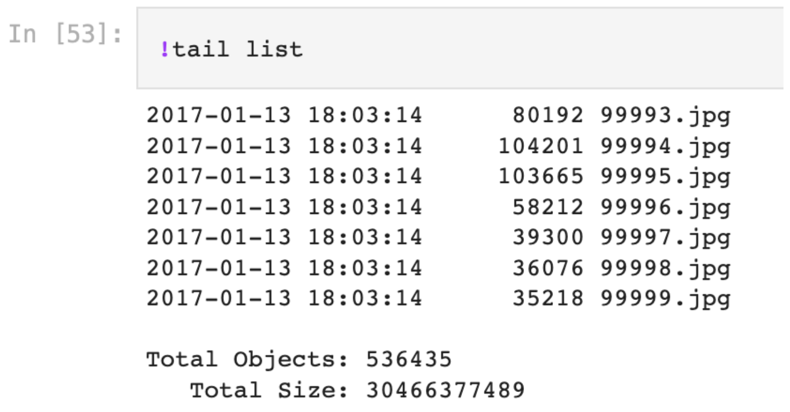
3.4 Metadata

Metadata files are JSON files containing image_fname and EXPECTED_QUANTITY fields that can be useful in this counting capstone project.



```
In [56]: !tail metadatalist

2017-01-13 17:59:16       4529 99993.json
2017-01-13 17:59:16       4529 99994.json
2017-01-13 17:59:16       3740 99995.json
2017-01-13 17:59:16        864 99996.json
2017-01-13 17:59:16       2132 99997.json
2017-01-13 17:59:16       2770 99998.json
2017-01-13 17:59:16       1658 99999.json

Total Objects: 536435
   Total Size: 1098414519
```

There are a total of 536,435 JSON metadata files, and an equal amount of jpg files to match.



```
In [53]: !tail list

2017-01-13 18:03:14      80192 99993.jpg
2017-01-13 18:03:14     104201 99994.jpg
2017-01-13 18:03:14     103665 99995.jpg
2017-01-13 18:03:14      58212 99996.jpg
2017-01-13 18:03:14      39300 99997.jpg
2017-01-13 18:03:14      36076 99998.jpg
2017-01-13 18:03:14      35218 99999.jpg

Total Objects: 536435
   Total Size: 30466377489
```

3.5 File Naming Conventions

The dataset considers `1.jpg` and `00001.jpg` differently like the following.

```
!aws s3 ls --no-sign-request s3://aft-vbi-pds/bin-images/1.jpg
```

```
2016-06-17 17:30:24      56301 1.jpg
```

```
!aws s3 ls --no-sign-request s3://aft-vbi-pds/bin-images/00001.jpg
```

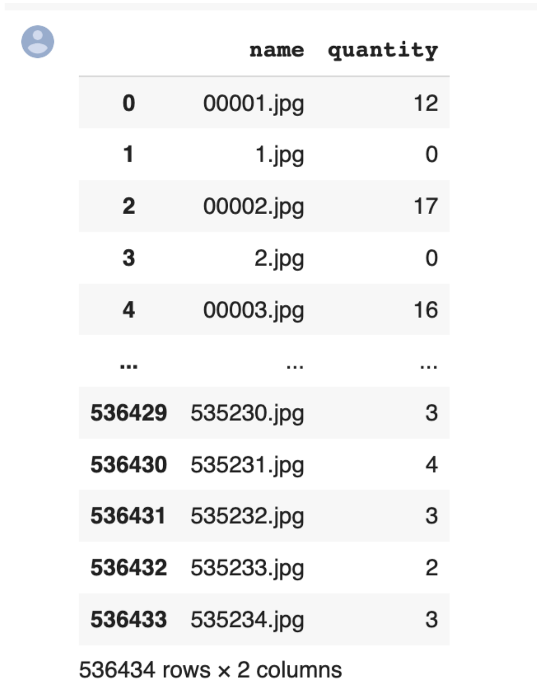
```
2017-01-13 22:47:53      45769 00001.jpg
```

In total, it has 536,434 images:

- **1~4 digit**: `1.jpg` ~ `1200.jpg` : 1200
- **5-digit**: `00001.jpg` ~ `99999.jpg` : 99,999
- **6-digit**: `100000.jpg` ~ `535234.jpg` : 435,235

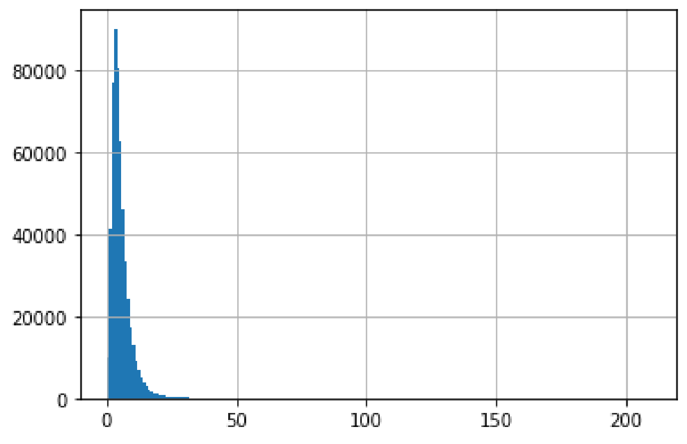
In addtion, in `Udacity Capston preparation chapter`, `Project Overview: Inventory Monitoring at Distribution Centers` provides a selected subset of files, `file_list.json`.

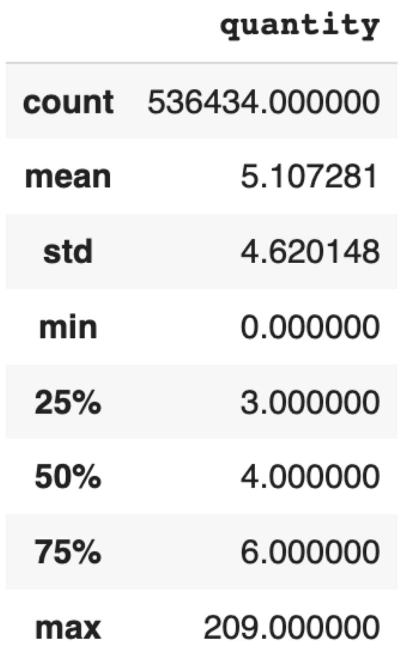
4. Exploratory Data Analysis



	name	quantity
0	00001.jpg	12
1	1.jpg	0
2	00002.jpg	17
3	2.jpg	0
4	00003.jpg	16
...	...	...
536429	535230.jpg	3
536430	535231.jpg	4
536431	535232.jpg	3
536432	535233.jpg	2
536433	535234.jpg	3

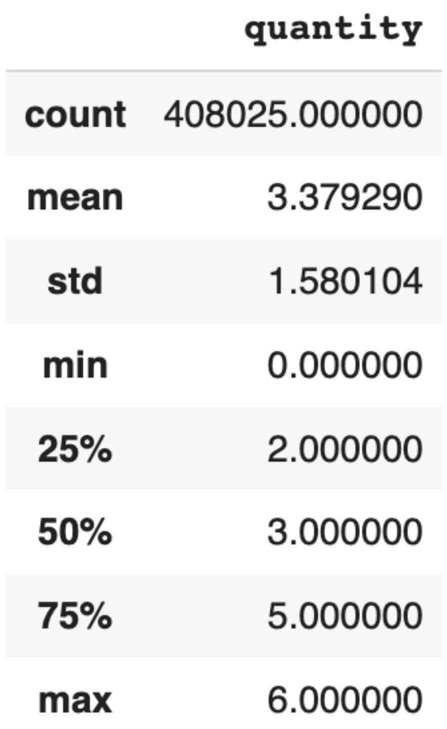
536434 rows × 2 columns



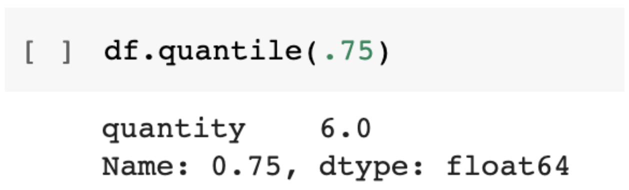


	quantity
count	536434.000000
mean	5.107281
std	4.620148
min	0.000000
25%	3.000000
50%	4.000000
75%	6.000000
max	209.000000

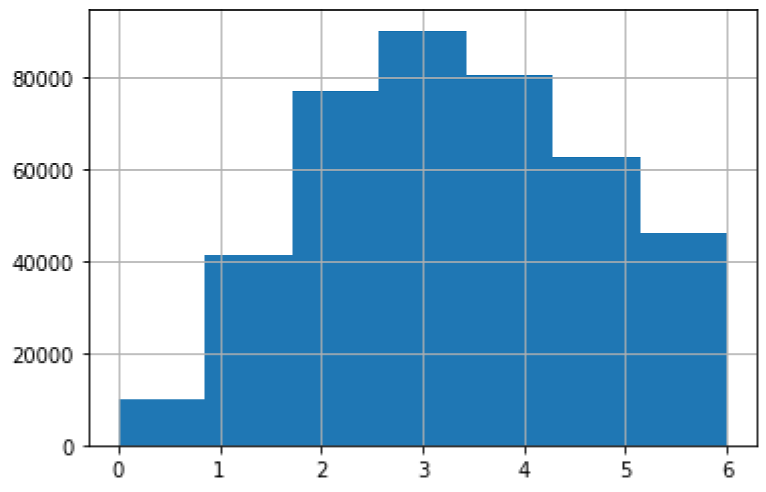
Since the data is skewed and decreases in frequency significantly after 6, I decided to narrow down the dataset from 0 to 6 objects, which is 75% of the data.



	quantity
count	408025.000000
mean	3.379290
std	1.580104
min	0.000000
25%	2.000000
50%	3.000000
75%	5.000000
max	6.000000



```
[ ]  df.quantile(.75)

     quantity    6.0
     Name: 0.75, dtype: float64
```



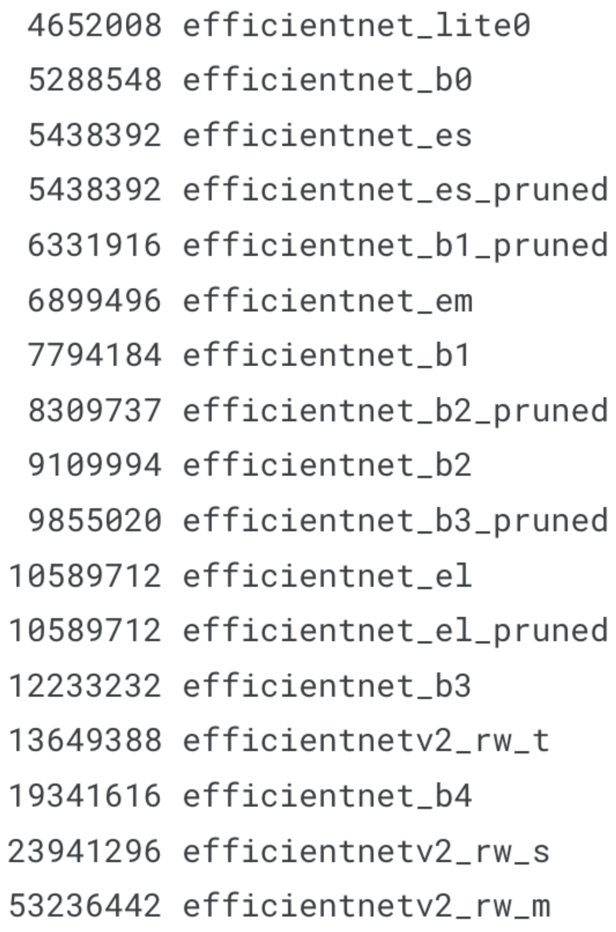
5. Choosing a Pre-trained Model

According to paperswithcode, Efficientnet had the most accurate model, therefore I decided to use an Efficientnet pre-trained model.
I first listed all of the pre-trained models from Efficientnet.

```
models = timm.list_models('efficientnet*', pretrained=True)
models
```

```
['efficientnet_b0',
 'efficientnet_b1',
 'efficientnet_b1_pruned',
 'efficientnet_b2',
 'efficientnet_b2_pruned',
 'efficientnet_b3',
 'efficientnet_b3_pruned',
 'efficientnet_b4',
 'efficientnet_el',
 'efficientnet_el_pruned',
 'efficientnet_em',
 'efficientnet_es',
 'efficientnet_es_pruned',
 'efficientnet_lite0',
 'efficientnetv2_rw_m',
 'efficientnetv2_rw_s',
 'efficientnetv2_rw_t']
```

Then I ordered the models by the number of parameters for each pre-trained model.

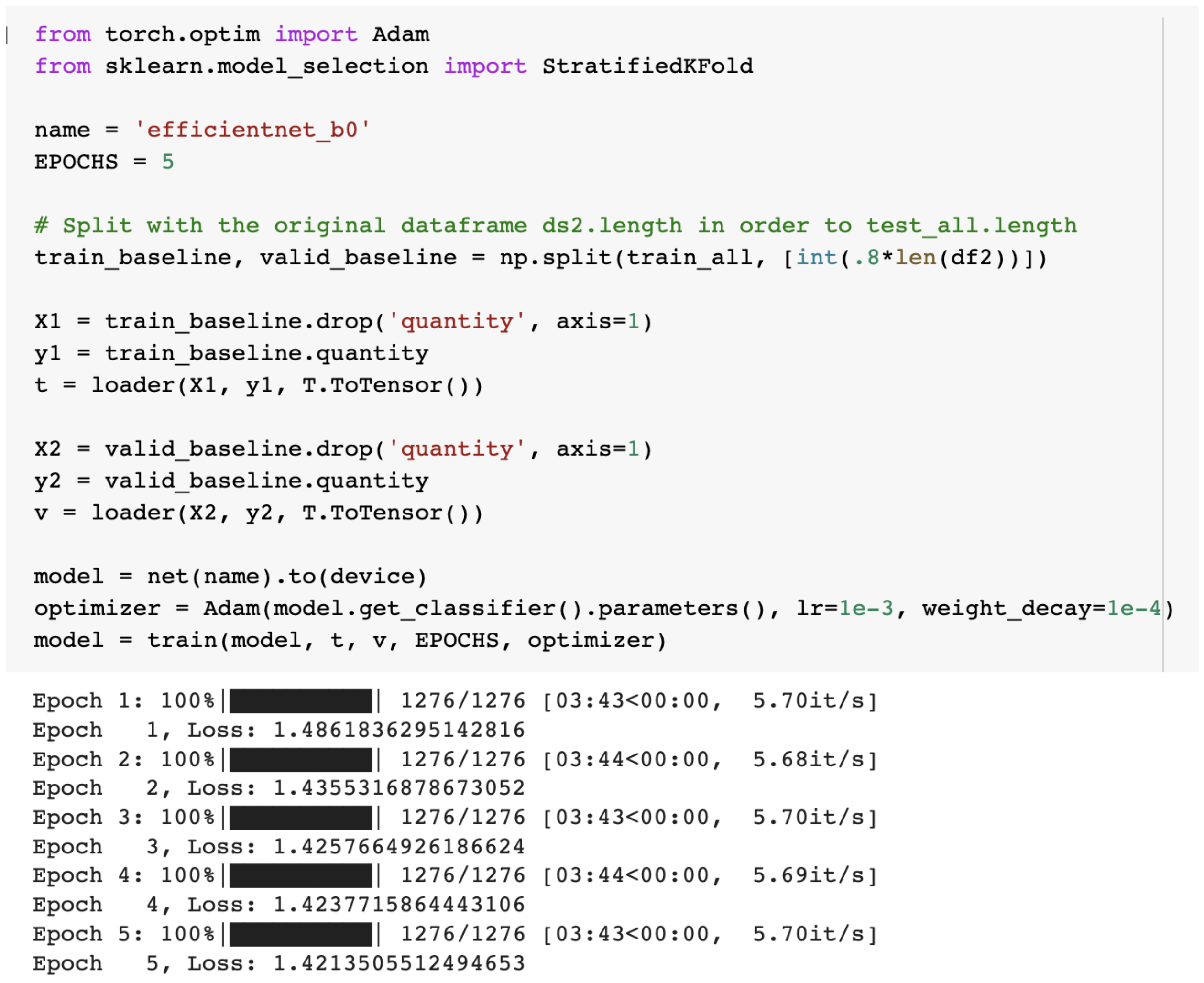


```
 4652008 efficientnet_lite0
 5288548 efficientnet_b0
 5438392 efficientnet_es
 5438392 efficientnet_es_pruned
 6331916 efficientnet_b1_pruned
 6899496 efficientnet_em
 7794184 efficientnet_b1
 8309737 efficientnet_b2_pruned
 9109994 efficientnet_b2
 9855020 efficientnet_b3_pruned
10589712 efficientnet_el
10589712 efficientnet_el_pruned
12233232 efficientnet_b3
13649388 efficientnetv2_rw_t
19341616 efficientnet_b4
23941296 efficientnetv2_rw_s
53236442 efficientnetv2_rw_m
```

Due to cost restrictions and time constraints, I wanted the smallest CNN so I could achieve maximum performance within the budget. Unfortunately, Efficientnet_lite0 was not documented by Mingxing Tan and Quoc V. Le in their research, "EfficientNetV2: Smaller Models and Faster Training" (https://arxiv.org/abs/2104.00298), therefore I chose the second-smallest option, Efficientnet_b0.

6. Benchmark Model

```
[ ] import timm
    import torch
    import torch.nn as nn
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(device)

    def net(name, pretrained=True):
        model = timm.create_model(name, pretrained)
        for param in model.parameters():
            param.requires_grad = False

        layer = nn.Sequential(
            nn.Linear(model.get_classifier().in_features, 7, bias=False),
        )
        if name.startswith('resnet'):
            model.fc = layer
        else: # efficientnet
            model.classifier = layer
        return model
```



```
from torch.optim import Adam
from sklearn.model_selection import StratifiedKFold

name = 'efficientnet_b0'
EPOCHS = 5

# Split with the original dataframe ds2.length in order to test_all.length
train_baseline, valid_baseline = np.split(train_all, [int(.8*len(df2))])

X1 = train_baseline.drop('quantity', axis=1)
y1 = train_baseline.quantity
t = loader(X1, y1, T.ToTensor())

X2 = valid_baseline.drop('quantity', axis=1)
y2 = valid_baseline.quantity
v = loader(X2, y2, T.ToTensor())

model = net(name).to(device)
optimizer = Adam(model.get_classifier().parameters(), lr=1e-3, weight_decay=1e-4)
model = train(model, t, v, EPOCHS, optimizer)
```

```
Epoch 1: 100%|██████████| 1276/1276 [03:43<00:00,  5.70it/s]
Epoch   1, Loss: 1.4861836295142816
Epoch 2: 100%|██████████| 1276/1276 [03:44<00:00,  5.68it/s]
Epoch   2, Loss: 1.4355316878673052
Epoch 3: 100%|██████████| 1276/1276 [03:43<00:00,  5.70it/s]
Epoch   3, Loss: 1.4257664926186624
Epoch 4: 100%|██████████| 1276/1276 [03:44<00:00,  5.69it/s]
Epoch   4, Loss: 1.4237715864443106
Epoch 5: 100%|██████████| 1276/1276 [03:43<00:00,  5.70it/s]
Epoch   5, Loss: 1.4213505512494653
```

I ran a benchmark model with EfficientNet_b0, 5 epochs, and an optimizer with a learning rate of 0.001 and a weight decay of 0.0001. Its accuracy came out to be 26.417175%, and this will serve as a benchmark for improvements.

```python
X = test_all.drop('quantity', axis=1)
y = test_all.quantity

test_dataset = MyDataset(X, y, 'bin-images-224x224/bin-images-224x224/', valid_transforms)
test_loader  = DataLoader(test_dataset, batch_size=256, num_workers=6)

model.eval()

running_corrects=0
l, p = [], []
for (inputs, labels) in tqdm(test_loader, 'Test', file=sys.stdout):
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    _, preds = torch.max(outputs, 1)
    running_corrects += torch.sum(preds == labels.data).item()
    l.extend(labels.tolist())
    p.extend(preds.tolist())
accuracy = 100.0 * running_corrects / len(test_loader.dataset)

print("Average accuracy: %f" % accuracy)
```

```
Test: 100%|██████████| 160/160 [00:26<00:00,  6.07it/s]
Average accuracy: 26.417175
```

7. Improvements

7.1 Data Preprocessing

I manipulated the data to fit the EfficientNet model by resizing each image to 224x224. I then uploaded the dataset to Kaggle to simplify the process.

- https://www.kaggle.com/datasets/williamhyun/amazon-bin-image-dataset-536434-images-224x224

7.2 Data Augmentation

To improve the given dataset, I chose to augment pre-existing images to create new, possible versions of a bin image. However, conventional data augmentation methods such as cropping and blur did not provide much success due to the fact that a bin image with 5 objects, if cropped, can appear to have only 2-3 objects. I encountered a similar issue with blur since the image gets lower in resolution, making two or more objects blend together to make one big object. In addition, I attempted to use AutoAugment, provided by PyTorch, but it proved to be a challenge since AutoAugment was hard to control in the context of a bin image, therefore I used the transformations listed below.

```
import torchvision
from torchvision import transforms as T

train_transforms = T.Compose([
    T.RandomHorizontalFlip(p=0.1),
    T.RandomGrayscale(p=0.1),
    T.RandomAdjustSharpness(2, p=0.1),
    T.RandomAutocontrast(p=0.1),
    T.RandomApply([T.ColorJitter(0.5, 0.5, 0.5, 0.5)], p=0.1),
    T.RandomApply([T.RandomAffine(degrees=0, translate=(0, 0.02), scale=(0.95, 0.9
9))], p=0.1),
    T.RandomApply([T.RandomChoice([
        T.RandomRotation((90, 90)),
        T.RandomRotation((-90, -90)),
    ])], p=0.1),
    T.ToTensor(),
])
```

7.3 Weighted Random Sampling

After running the initial testing with the improvements listed above, I wanted to improve the accuracy of detecting a bin with no items in it. I found that the problem stems from the fact that there is a lack of data for empty bins. I then applied weights to the number of objects in an item bin, where a higher weight means its more likely to be sampled.

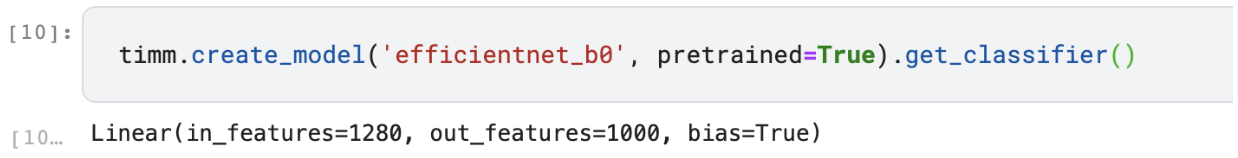
```
CLASS_WEIGHTS = df2.groupby(['quantity'])['quantity'].count().apply(lambda x: 1/x)
CLASS_WEIGHTS
```

```
quantity
0    0.000101
1    0.000024
2    0.000013
3    0.000011
4    0.000012
5    0.000016
6    0.000022
Name: quantity, dtype: float64
```

```
sample_weights = [0] * len(y)
for idx, label in enumerate(y):
    sample_weights[idx] = CLASS_WEIGHTS[label]
sampler = WeightedRandomSampler(sample_weights, num_samples=len(sample_weights))
```

7.4 Refine Model

Whereas the benchmark model replaces the following classifier’s 1000 out_features



```
[10]: timm.create_model('efficientnet_b0', pretrained=True).get_classifier()

[10… Linear(in_features=1280, out_features=1000, bias=True)
```

with 7 out_features:

```
layer = nn.Sequential(
    nn.Linear(model.get_classifier().in_features, 7, bias=False),
)
```

I chose to add another layer of 512 out_features before reducing it to 7:

```
layer = nn.Sequential(
    nn.BatchNorm1d(model.get_classifier().in_features),
    nn.Linear(model.get_classifier().in_features, 512, bias=False),
    nn.ReLU(),
    nn.BatchNorm1d(512),
    nn.Dropout(p=0.5, inplace=True),
    nn.Linear(512, 7, bias=False))
```

In addition, before the final classifier, I also added a Dropout function with ReLU and a Batch normalization for 512 nodes to reduce overfitting.

7.5 Multi-Resolution Training

Testing with these new improvements still did not yield satisfactory results, therefore I turned to multi-resolution training to improve my model. Multi-resolution training suggests that the model be trained with lower resolution images first, then with a higher resolution image after, building on the previous framework trained by low-resolution images. In order to test this method, I created and trained a model for detecting empty bins. Using 10 epochs each, I trained using 56x56, then 112x112, then 224x224 pixel images and achieved a 98.26%, which was an improvement above the base model where I used the exact same settings without multi-resolution training and ensemble models.

8. Final Solution

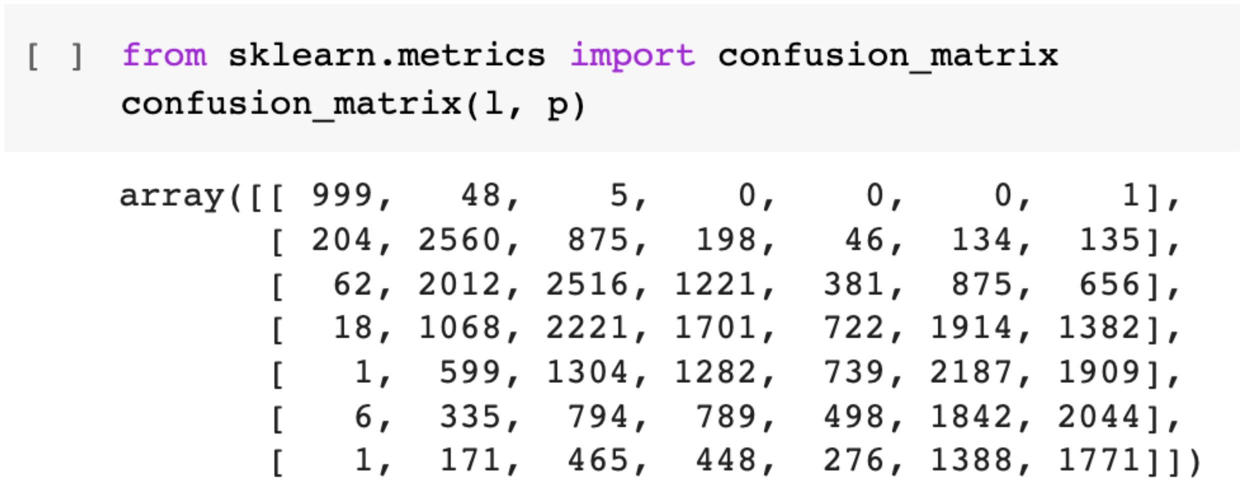
```python
from torch.optim import Adam
from sklearn.model_selection import StratifiedKFold

name = 'efficientnet_b0'
EPOCHS = 10
IMAGE_SIZES = [112, 224]

# Split with the original dataframe ds2.length in order to test_all.length
train_baseline, valid_baseline = np.split(train_all, [int(.8*len(df2))])

X1 = train_baseline.drop('quantity', axis=1)
y1 = train_baseline.quantity
X2 = valid_baseline.drop('quantity', axis=1)
y2 = valid_baseline.quantity

model = net(name).to(device)
for s in IMAGE_SIZES:
    t = loader(X1, y1, T.Compose([T.Resize(s), train_transforms]))
    v = loader(X2, y2, T.ToTensor())
    optimizer = Adam(model.get_classifier().parameters(), lr=1e-3, weight_decay=1e-4)
    model = train(model, t, v, EPOCHS, optimizer)
```

The final solution utilizes EfficientNet_b0 and multi-resolution training with 112x112 and 224x224 pixels with 10 epochs respectively. I kept the same optimizer settings since there were not enough resources for hyperparameter tuning.

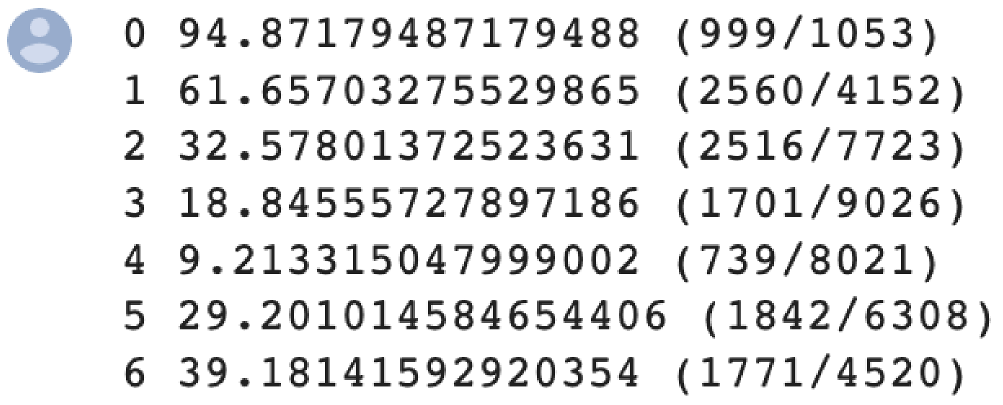
```python
X = test_all.drop('quantity', axis=1)
y = test_all.quantity

test_dataset = MyDataset(X, y, 'bin-images-224x224/bin-images-224x224/', valid_transforms)
test_loader  = DataLoader(test_dataset, batch_size=256, num_workers=6)

model.eval()

running_corrects=0
l, p = [], []
for (inputs, labels) in tqdm(test_loader, 'Test', file=sys.stdout):
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    _, preds = torch.max(outputs, 1)
    running_corrects += torch.sum(preds == labels.data).item()
    l.extend(labels.tolist())
    p.extend(preds.tolist())
accuracy = 100.0 * running_corrects / len(test_loader.dataset)

print("Average accuracy: %f" % accuracy)
```

```
Test: 100%|██████████| 160/160 [00:27<00:00,  5.90it/s]
Average accuracy: 29.723305
```

We can see a total change of of 3.30613% or a 12.5151% increase.



```
[ ]  from sklearn.metrics import confusion_matrix
     confusion_matrix(l, p)

     array([[ 999,   48,    5,    0,    0,    0,    1],
            [ 204, 2560,  875,  198,   46,  134,  135],
            [  62, 2012, 2516, 1221,  381,  875,  656],
            [  18, 1068, 2221, 1701,  722, 1914, 1382],
            [   1,  599, 1304, 1282,  739, 2187, 1909],
            [   6,  335,  794,  789,  498, 1842, 2044],
            [   1,  171,  465,  448,  276, 1388, 1771]])
```

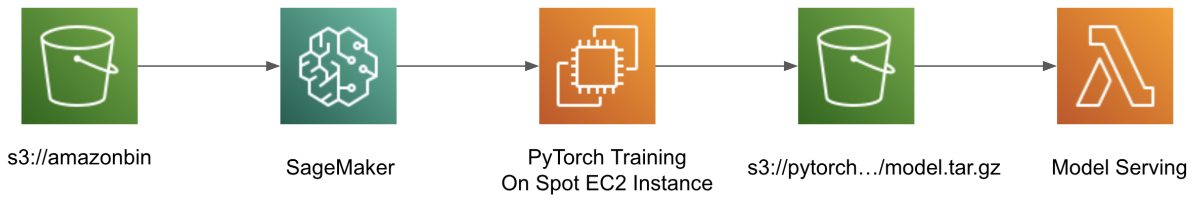


```
0 94.87179487179488 (999/1053)
1 61.65703275529865 (2560/4152)
2 32.57801372523631 (2516/7723)
3 18.84555727897186 (1701/9026)
4 9.213315047999002 (739/8021)
5 29.201014584654406 (1842/6308)
6 39.18141592920354 (1771/4520)
```

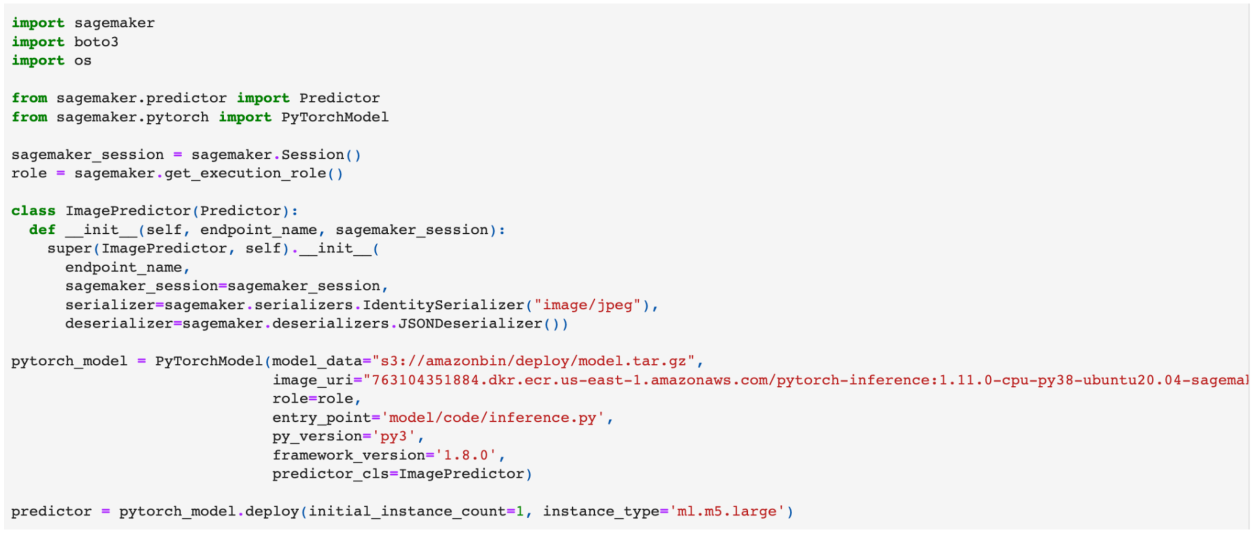
The confusion matrix demonstrates that my methods improves the accuracy for predicting empty bins in addition to improving accuracy for the majority of the categories.

9. Model Serving

I chose to serve the trained model according to the pipeline described above in section 2.

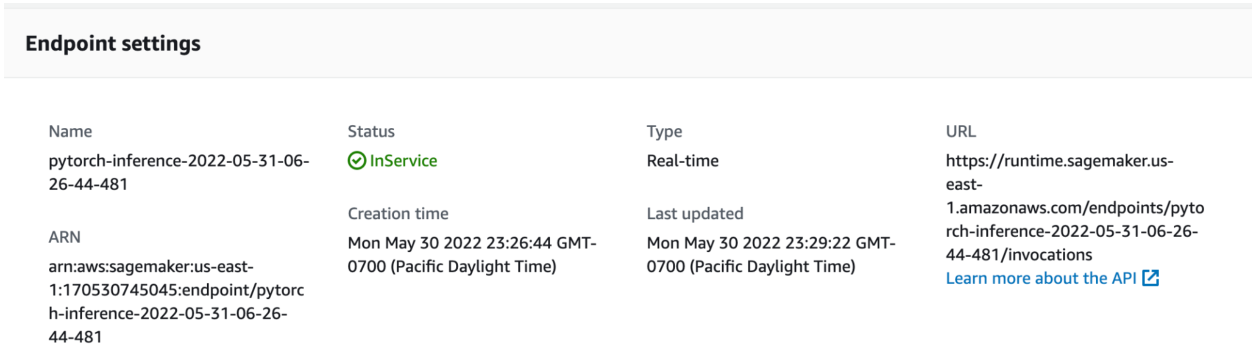


I first uploaded the model to s3://amazonbin/deploy/model.tar.gz then deployed that model on Python 3.8. In that process, SageMaker copies the model to the serving location, s3://sagemaker-us-east-1-170530745045/pytorch-inference-2022-05-31-06-26-41-811/model.tar.gz.



```python
import sagemaker
import boto3
import os

from sagemaker.predictor import Predictor
from sagemaker.pytorch import PyTorchModel

sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()

class ImagePredictor(Predictor):
  def __init__(self, endpoint_name, sagemaker_session):
    super(ImagePredictor, self).__init__(
      endpoint_name,
      sagemaker_session=sagemaker_session,
      serializer=sagemaker.serializers.IdentitySerializer("image/jpeg"),
      deserializer=sagemaker.deserializers.JSONDeserializer())

pytorch_model = PyTorchModel(model_data="s3://amazonbin/deploy/model.tar.gz",
                             image_uri="763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:1.11.0-cpu-py38-ubuntu20.04-sagema
                             role=role,
                             entry_point='model/code/inference.py',
                             py_version='py3',
                             framework_version='1.8.0',
                             predictor_cls=ImagePredictor)

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

This real-time endpoint with my model can be seen here: https://runtime.sagemaker.us-east-1.amazonaws.com/endpoints/pytorch-inference-2022-05-31-06-26-44-481/invocations



Endpoint settings

Name	Status	Type	URL
pytorch-inference-2022-05-31-06-26-44-481	✓ InService	Real-time	https://runtime.sagemaker.us-east-1.amazonaws.com/endpoints/pytorch-inference-2022-05-31-06-26-44-481/invocations Learn more about the API
ARN	**Creation time**	**Last updated**	
arn:aws:sagemaker:us-east-1:170530745045:endpoint/pytorch-inference-2022-05-31-06-26-44-481	Mon May 30 2022 23:26:44 GMT-0700 (Pacific Daylight Time)	Mon May 30 2022 23:29:22 GMT-0700 (Pacific Daylight Time)	

I then invoked the deployed endpoint with the following image, which contains one object.

```python
from PIL import Image
import io
buf = io.BytesIO()
Image.open("00024.jpg").save(buf, format="JPEG")

response = predictor.predict(buf.getvalue())
```

```python
response
```

```
[[-2.8799102306365967,
  2.4312524795532227,
  1.433470606803894,
  0.49086323380470276,
  -0.32889169454574585,
  -0.8075268864631653,
  -1.3364081382751465]]
```

The service responded correctly by stating that it is most likely to be an image with one object.

10. Conclusion

Throughout the project, I have done EDA, chose the best pre-trained model for my budget, and done research to find new methods of improvement. I made best use of the given dataset with data augmentation where I created new, possible versions of pre-existing data, sampled the data according to my weighted random sampling, and refined the model through multi-resolution training and using the Dropout function. I then deployed the trained model according to the pipeline I provided and tested it so that this can be consumed by users.