Machine Learning Engineer Nanodegree

Capstone Project

Margaret Chan

December 20, 2024

I. Definition

Project Overview

This project is to build an end-to-end ML (Machine Learning) system that counts the number of objects in storage containers, which are commonly moved around by robots in an industrial environment, such as distribution centers; this automation could potentially improve any inventory control process.

While there are a few components in this end-to-end system, the core is a CNN (Convolutional Neural Network) based computer vision model, which is trained with images of containers (or bins), with different counts of objects inside. These images were captured by robots in an Amazon Fulfillment Center and published as a dataset on AWS Data Exchange.

Problem Statement

Distribution centers often use robots to move objects around as a part of their operations, objects are carried in bins which can contain different number of objects. Robots often need human operators to enter data about the contents of the bins, such as the number of objects to start with, which is a slow, mundane, and error prone process.

In this project, I want to improve this process by developing a computer vision model that can identify the number of objects in a bin, and then build an end-to-end system that makes it easy to use the model.

Metrics

Accuracy which measures the ratio of correct predictions of a set of test images will be used a metric to evaluate the performance of our model.

$$Accuracy = \frac{\sum_{i=1}^{T} (p_i = l_i)}{T} \quad where\ p\ is\ predicted,\ l\ is\ label,\ T\ is\ test\ set\ size$$

II. Analysis

Data Exploration

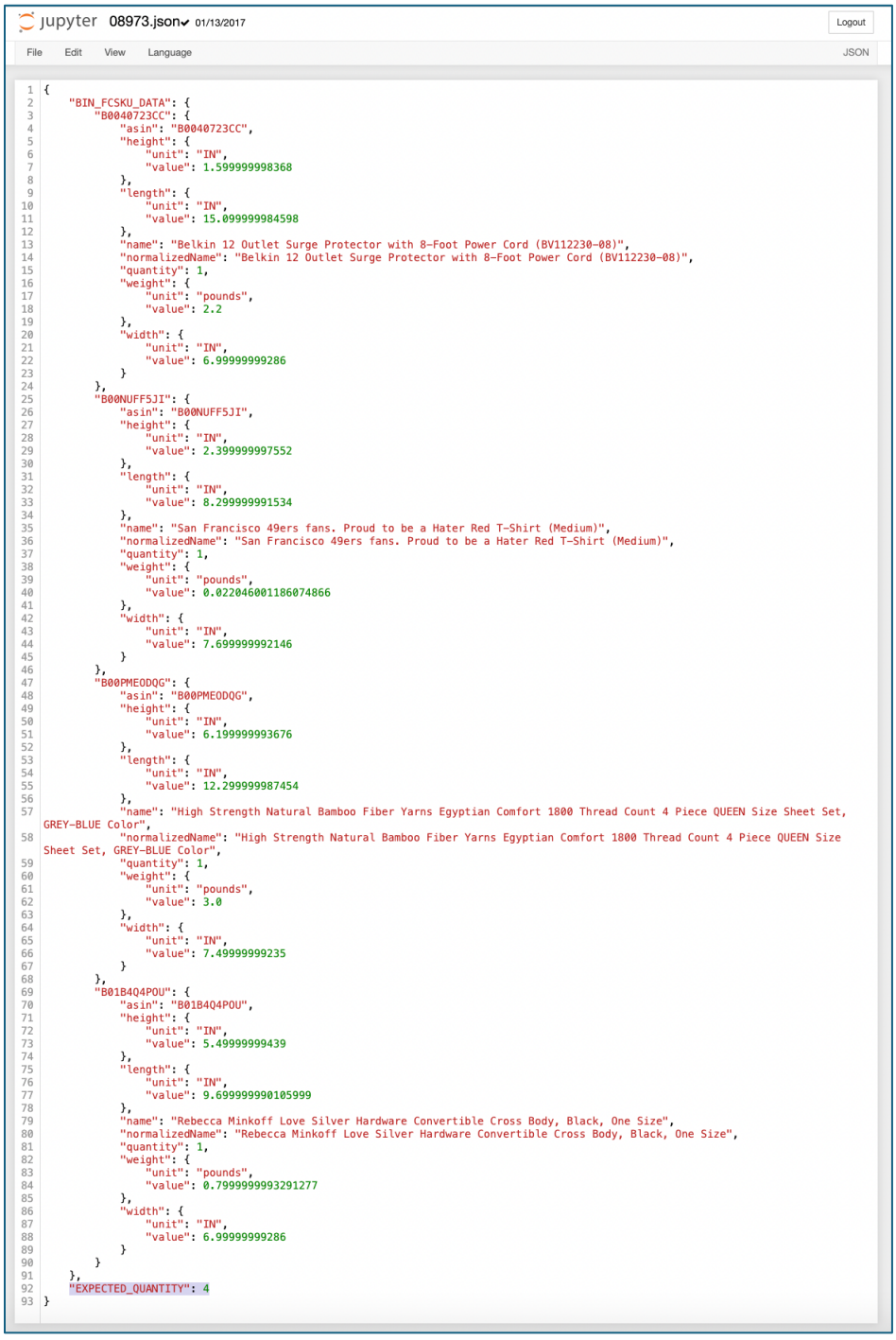
For training images, I am using the 'Amazon Bin Image Dataset' on AWS Data Exchange, documented at https://registry.opendata.aws/amazon-bin-imagery, it provides >500,000 JPEG images of bins with different object counts, and the respective JSON metadata files.

For each JPEG image, there is a corresponding JSON metadata file that describes the contents of the bin, such as object name, and dimensions, although I am only interested in the object count ('EXPECTED_QUANTITY') datapoint.

For example, *bin-images/08973.jpg*:

Figure 1. bin-images/08973.jpg

And the respective *metadata/08973.json*:




```
{
    "BIN_FCSKU_DATA": {
        "B0040723CC": {
            "asin": "B0040723CC",
            "height": {
                "unit": "IN",
                "value": 1.599999998368
            },
            "length": {
                "unit": "IN",
                "value": 15.099999984598
            },
            "name": "Belkin 12 Outlet Surge Protector with 8-Foot Power Cord (BV112230-08)",
            "normalizedName": "Belkin 12 Outlet Surge Protector with 8-Foot Power Cord (BV112230-08)",
            "quantity": 1,
            "weight": {
                "unit": "pounds",
                "value": 2.2
            },
            "width": {
                "unit": "IN",
                "value": 6.99999999286
            }
        },
        "B00NUFF5JI": {
            "asin": "B00NUFF5JI",
            "height": {
                "unit": "IN",
                "value": 2.399999997552
            },
            "length": {
                "unit": "IN",
                "value": 8.299999991534
            },
            "name": "San Francisco 49ers fans. Proud to be a Hater Red T-Shirt (Medium)",
            "normalizedName": "San Francisco 49ers fans. Proud to be a Hater Red T-Shirt (Medium)",
            "quantity": 1,
            "weight": {
                "unit": "pounds",
                "value": 0.022046001186074866
            },
            "width": {
                "unit": "IN",
                "value": 7.699999992146
            }
        },
        "B00PMEODQG": {
            "asin": "B00PMEODQG",
            "height": {
                "unit": "IN",
                "value": 6.199999993676
            },
            "length": {
                "unit": "IN",
                "value": 12.299999987454
            },
            "name": "High Strength Natural Bamboo Fiber Yarns Egyptian Comfort 1800 Thread Count 4 Piece QUEEN Size Sheet Set, GREY-BLUE Color",
            "normalizedName": "High Strength Natural Bamboo Fiber Yarns Egyptian Comfort 1800 Thread Count 4 Piece QUEEN Size Sheet Set, GREY-BLUE Color",
            "quantity": 1,
            "weight": {
                "unit": "pounds",
                "value": 3.0
            },
            "width": {
                "unit": "IN",
                "value": 7.49999999235
            }
        },
        "B01B4Q4POU": {
            "asin": "B01B4Q4POU",
            "height": {
                "unit": "IN",
                "value": 5.49999999439
            },
            "length": {
                "unit": "IN",
                "value": 9.699999990105999
            },
            "name": "Rebecca Minkoff Love Silver Hardware Convertible Cross Body, Black, One Size",
            "normalizedName": "Rebecca Minkoff Love Silver Hardware Convertible Cross Body, Black, One Size",
            "quantity": 1,
            "weight": {
                "unit": "pounds",
                "value": 0.7999999993291277
            },
            "width": {
                "unit": "IN",
                "value": 6.99999999286
            }
        }
    },
    "EXPECTED_QUANTITY": 4
}
```


Figure 2. metadata/08973.json

For project development, I am using files from a Git repo from Udacity, located at https://github.com/udacity/nd009t-capstone-starter, the repo provides the starter Jupyter notebook, python files, and more.

Among the starter files is *file_list.json*, it is a pre-built JSON metadata file, which helps retrieve a subset of 10,441 images from those >500,000 images on AWS Data Exchange. Moreover, it only includes data of object counts from 1 to 5; therefore, limiting the number of prediction classes to 5.


```
{
    "1": ["data/metadata/100313.json", "data/metadata/09915.json", ...],
    "2": ["data/metadata/102489.json", "data/metadata/104982.json", ...],
    "3": ["data/metadata/04031.json", "data/metadata/04749.json", ...],
    "4": ["data/metadata/101042.json", "data/metadata/09753.json", ...],
    "5": ["data/metadata/03858.json", "data/metadata/101598.json", ...],
}
```


Figure 3. file_list.json excerpt

After I downloaded the subset of 10,441 images, I split them into 3 sets (80% training, 10% validation, 10% test), then I checked the distribution of the training set (8351 images), specifically the image count per prediction class and found no anomaly.

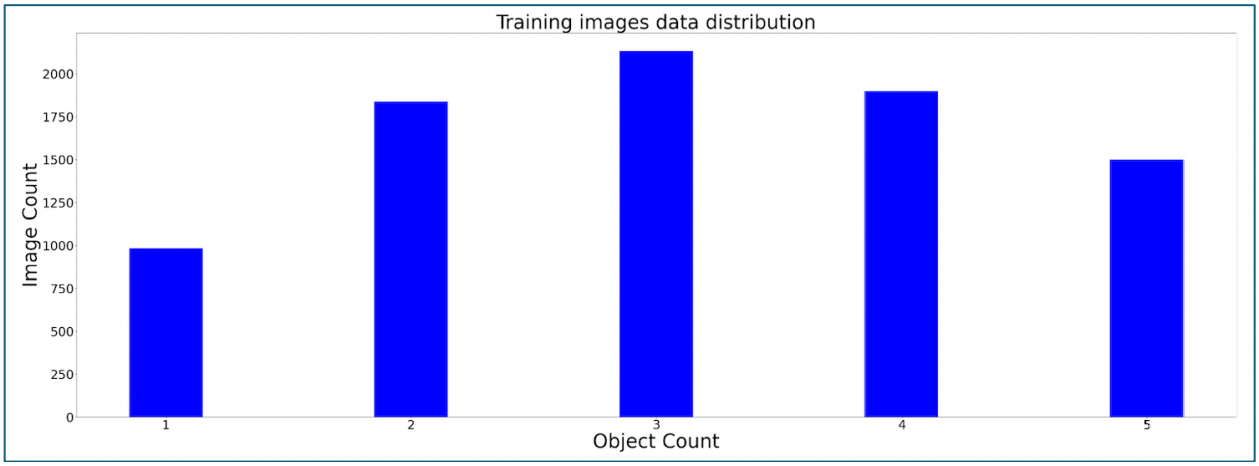


Figure 4. Training images dataset distribution

However, during data exploration, I realize the training images from the source dataset are of extremely low quality, lacking:

- Number of pixels
- Colors
- Contrast

Here are a few examples:

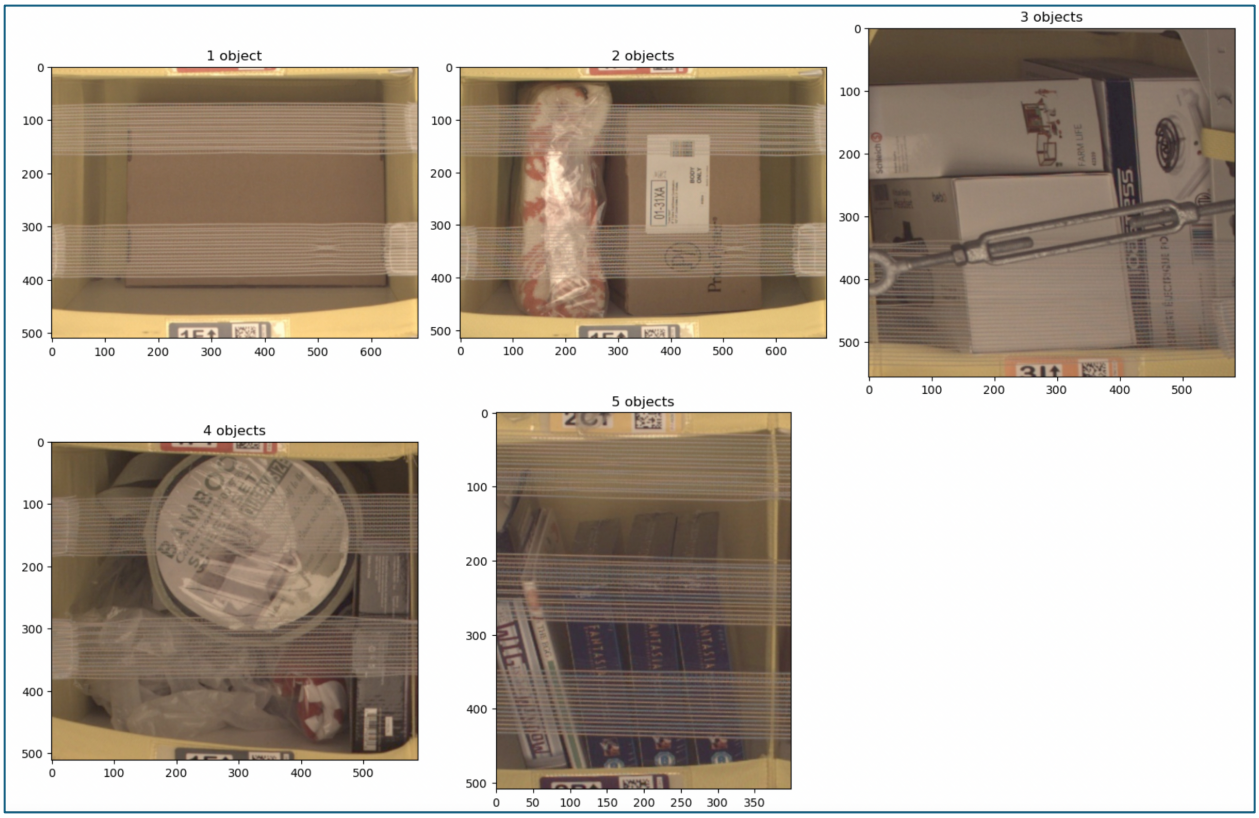


Figure 5. Sample training images

Exploratory Visualization

Example training images in the above section have shown features like edges, shapes, and colors, that are needed to train our computer vision model.

Algorithms and Techniques

CNN is a kind of advanced artificial neural network that is widely used to solve computer vision problems, it is inspired by the visual cortex in humans, it looks for features in images, where features can be simple or complex. One of the benefits of using CNN is its support

for transfer learning, where a pre-trained CNN can be fine-tuned on other tasks. Since there is no need to go back and train all the convolutional and pooling layers, and I only need to train the last FC (Fully Connected) layer with our bespoke (and usually a much smaller set of) training images, it will take significantly less time and still be able to improve the accuracy.

In this project, I will leverage exactly that, by using a pre-trained CNN model called *RESNET34*[2], which is a 34-layer CNN pre-trained on the ImageNet[1] dataset, which would give us a model that has pre-learned a lot of generic features, so all I need to do is to replace its last FC layer with a new one initialized with random weights, and only train the new layer with our set of 10,441 images.

Benchmark

For the evaluation metric of accuracy, of course the higher value is always the better; but for our use-case of inventory control, and with the usual time and cost constraints, I think it would be acceptable to stop improving when it reaches an accuracy of 0.80 or 80%.

III. Methodology

Data Preprocessing

As mentioned in the Data Exploration section, the original images have many problems, one thing I tried was to increase the contrast, by applying the 'UnsharpMask' filter.

Example result:

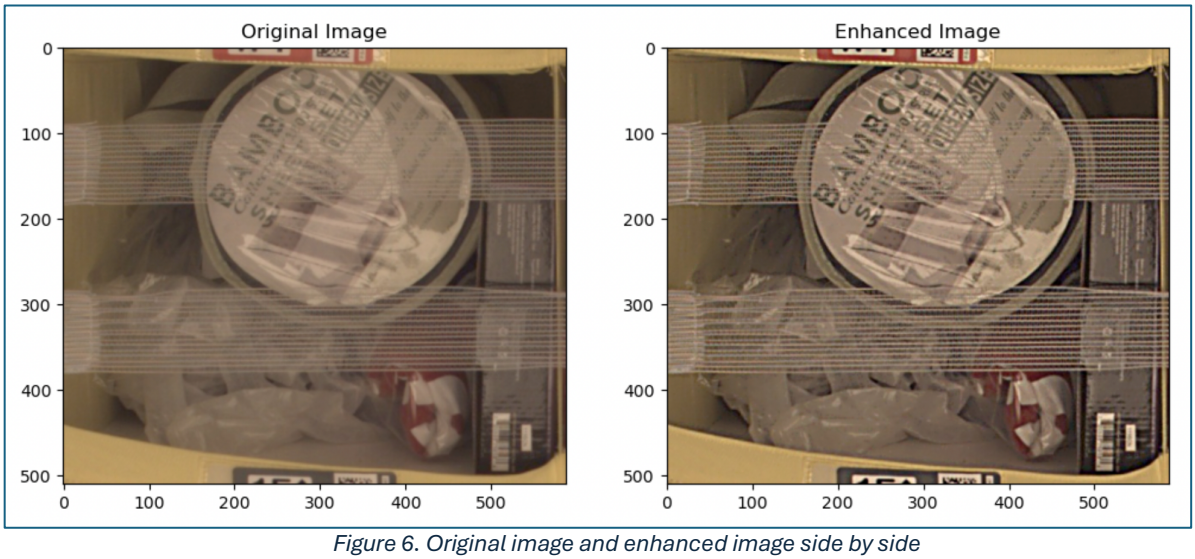


Figure 6. Original image and enhanced image side by side

The next data pre-processing effort is to run the training, validation, and testing datasets through some standard ImageNet transforms[3] before feeding them to the model:

```
    imagenet_mean = [0.485, 0.456, 0.406]
    imagenet_std = [0.229, 0.224, 0.225]

    # for training data, augment and normalize
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224), # crop at a random location, and resize the crop
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(), # convert to PyTorch Tensor
        transforms.Normalize(imagenet_mean, imagenet_std)
    ]),

    # for validation and test data, just normalize
    'val': transforms.Compose([
        transforms.Resize(256), # resize to 256x256
        transforms.CenterCrop(224), # crop at the center
        transforms.ToTensor(), # convert to PyTorch Tensor
        transforms.Normalize(imagenet_mean, imagenet_std)
    ]),
```

Figure 7. training and validation/testing data transforms

Implementation

Before diving into the ML part, there should be a brief overview of the end-to-end system:

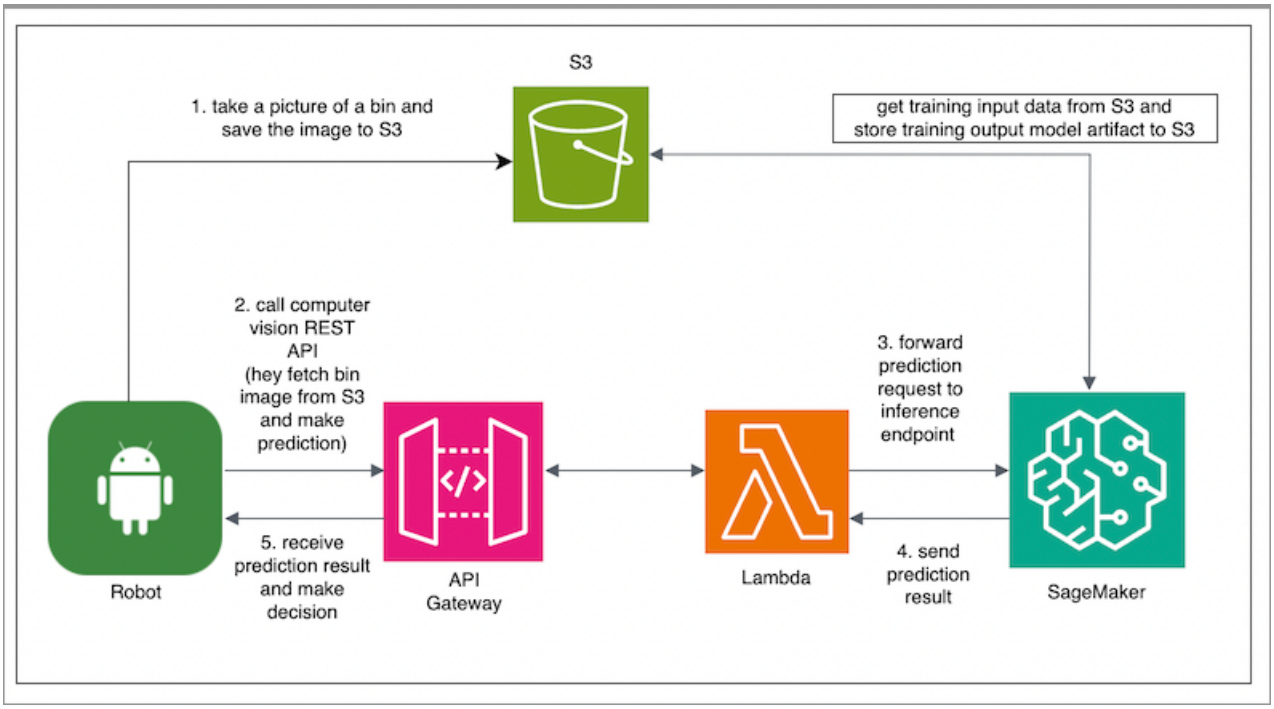


Figure 8. The end-to-end system

As illustrated in the architecture diagram above, I am leveraging a few AWS services:

- SageMaker for training the model, tuning hyperparameters, and deploying the trained model
- Lambda for invoking the deployed model inference endpoint
- API Gateway for provisioning a public endpoint for accessing the system via Lambda
- S3 for storing the training and testing images

For the ML part, I am using SageMaker and SageMaker PyTorch SDK to train the model. With the SDK, I can just specify the instance type, instance count, software versions, training hyperparameters, and the container entry point python script (e.g., *train.py in our project*), and SageMaker can start training.

```
hyperparameters = {
    'learning_rate': 0.001,
    'batch_size': 128,
    'epochs': 10
}

estimator = sagemaker.pytorch.PyTorch(
    entry_point = './train.py',
    base_job_name = 'capstone-train',
    hyperparameters = hyperparameters,
    framework_version = '1.8',
    py_version = 'py36',
    role = sagemaker.get_execution_role(),
    instance_count = 1,
    instance_type = 'ml.g4dn.xlarge' # NVIDIA T4 GPUs
)

estimator.fit({"training": s3_uri}, wait=True)
```

Figure 9. Launch training job from SageMaker

Once it enters the training job container, which is running *train.py*, it parses the hyperparameters, create a model from a pre-trained model, run pre-processing transforms (as discussed in the previous section) on all 3 datasets, then train, validate, test; and lastly, save the model parameters, which are needed when deploy the model:

```python
if __name__=='__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument("--learning_rate", type=float, default=0.001)
    parser.add_argument("--batch_size", type=int, default=128)
    parser.add_argument("--epochs", type=int, default=10)
    parser.add_argument("--data_dir", type=str, default=os.environ['SM_CHANNEL_TRAINING'])
    parser.add_argument("--model_dir", type=str, default=os.environ['SM_MODEL_DIR'])
    parser.add_argument("--output_dir", type=str, default=os.environ['SM_OUTPUT_DATA_DIR'])

    args = parser.parse_args()

    main(args)

def main(args):
    # create a fine-tuned model
    model = net()

    # define loss function and optimizer
    loss_criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.fc.parameters(), lr=args.learning_rate)

    # create data loaders
    data_loaders = create_data_loaders(args.data_dir, int(args.batch_size))

    # train the model
    model = train(model, data_loaders['train'], data_loaders['val'], loss_criterion, optimizer, int(args.epochs))

    # test the model
    test(model, data_loaders['test'], loss_criterion)

    # save best model parameters as checkpoint data
    torch.save(model.state_dict(), os.path.join(args.model_dir, 'model.pt'))
```

Figure 10. train.py - container entry point

```python
def net():
    model = models.resnet34(pretrained=True)

    # do not train the convolutional layers
    for param in model.parameters():
        param.requires_grad = False

    # replace the last FC layer with a new one with random weights, and only train that layer
    in_feature_count = model.fc.in_features
    model.fc = nn.Linear(in_feature_count, 5)

    model = model.to(device)

    print('Created a model fined tuned from RESNET34')

    return model
```

Figure 11. train.py - create a model from RESNET34

```python
def train(model, train_loader, val_loader, criterion, optimizer, epochs):
    best_model = model
    best_accuracy = 0.0

    for epoch in range(epochs):
        for type in ['train', 'val']:
            if type == 'train':
                model.train() # set model to train mode
                data_loader = train_loader
            else:
                model.eval() # set model to evaluate mode
                data_loader = val_loader

            image_count = len(data_loader.dataset)
            batch_count = len(data_loader)
            running_loss = 0
            running_correct = 0

            for batch_id, (inputs, targets) in enumerate(data_loader):

                inputs = inputs.to(device)
                targets = targets.to(device)

                if type == 'train':
                    optimizer.zero_grad()

                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, targets)

                # stats
                loss_count = loss.item() * inputs.size(0)
                running_loss += loss_count
                correct_count = torch.sum(preds == targets.data)
                running_correct += correct_count

                if type == 'train':
                    loss.backward()
                    optimizer.step()

            loss_ratio = running_loss / image_count
            accuracy = running_correct / image_count
            print(f"Pass {epoch}/{epochs} {type} - Average loss: {loss_ratio:.4f}, Accuracy: {accuracy:.4f}")

            if type == 'val':
                if accuracy > best_accuracy:
                    best_model = model
                    best_accuracy = accuracy

    return best_model
```

Figure 12. train.py - train and validate

```
def test(model, test_loader, criterion):
    running_loss = 0
    running_correct = 0
    test_set_size = len(test_loader.dataset)

    model.eval() # set model to evaluation mode

    with torch.no_grad():
        for inputs, targets in test_loader:

            inputs = inputs.to(device)
            targets = targets.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            loss = criterion(outputs, targets)

            # stats
            loss_count = loss.item() * inputs.size(0)
            running_loss += loss_count
            correct_count = torch.sum(preds == targets.data)
            running_correct += correct_count

        loss_ratio = running_loss / test_set_size
        accuracy = running_correct / test_set_size

        print(f"\nTest set: Accuracy: {accuracy:.4f} ({running_correct}/{test_set_size}), Average loss: {loss_ratio:
```

Figure 13. train.py - test

One thing worth mentioning is that 2 other RESNET pre-trained models (RESNET18[4] and RESNET50[5]) were tried before settling with RESNET34, when RESNET34 has performed better than RESNET18, I tried to skip a level to RESNET50, but the performance unexpectedly got worse, so I am sticking to RESNET34.

Refinement

Another SageMaker feature I want to leverage is hyperparameter tuning. To launch a hyperparameter tuning job, all I need is to pass a few parameters to a tuner object, such as what hyperparameters to train, the range of those hyperparameters, what should be the objective (Maximize or Minimize), where the metric data is; and lastly, how many underlying model training jobs it should launch to pick the best from. Notice the code that creates a

PyTorch estimator looks very similar, the only difference is I do not need to pass the hyperparameters, because the tuning job will.

```
hyperparameter_ranges = {
    'learning_rate': ContinuousParameter(0.0005, 0.001),
    'batch_size': CategoricalParameter([128, 256]),
    'epochs': CategoricalParameter([20, 30])
}

objective_metric_name = 'test accuracy'
objective_type = 'Maximize'
metric_definitions = [{"Name": objective_metric_name, "Regex": "Test set: Accuracy: ([0-9\\.]+)"}]

estimator = sagemaker.pytorch.PyTorch(
    entry_point = './train.py',
    base_job_name = 'capstone_train_hpo',
    #hyperparameters = hyperparameters,
    framework_version = '1.8',
    py_version = 'py36',
    role = sagemaker.get_execution_role(),
    instance_count = 1,
    instance_type = 'ml.g4dn.xlarge' # NVIDIA T4 GPUs
)

tuner = sagemaker.tuner.HyperparameterTuner(
    estimator=estimator,
    objective_metric_name=objective_metric_name,
    hyperparameter_ranges=hyperparameter_ranges,
    metric_definitions=metric_definitions,
    max_jobs=4,
    max_parallel_jobs=2,
    objective_type=objective_type,
)

# this will launch a hyperparameter tuning job with 4 training jobs
tuner.fit({"training": s3_uri}, wait=True)
```

Figure 14. Launch hyperparameter tuning job from SageMaker

After the tuning job completes, I can query the best set of hyperparameters:

```
In [116]: # TODO: Find the best hyperparameters

          best_train_job = tuner.best_training_job()

          best_train_job

Out[116]: 'pytorch-training-241219-1837-002-69656183'

In [117]: best_estimator = tuner.best_estimator()

          2024-12-19 18:55:05 Starting - Preparing the instances for training
          2024-12-19 18:55:05 Downloading - Downloading the training image
          2024-12-19 18:55:05 Training - Training image download completed. Training in progress.
          2024-12-19 18:55:05 Uploading - Uploading generated training model
          2024-12-19 18:55:05 Completed - Resource reused by training job: pytorch-training-241219-1837-004-a2ace4e0

In [118]: best_hyperparameters = best_estimator.hyperparameters()

          best_hyperparameters

Out[118]: {'_tuning_objective_metric': '"test accuracy"',
           'batch_size': '"256"',
           'epochs': '"20"',
           'learning_rate': '0.0008015841478983083',
           'sagemaker_container_log_level': '20',
           'sagemaker_estimator_class_name': '"PyTorch"',
           'sagemaker_estimator_module': '"sagemaker.pytorch.estimator"',
           'sagemaker_job_name': '"capstone_train_hpo-2024-12-19-18-37-17-842"',
           'sagemaker_program': '"train.py"',
           'sagemaker_region': '"us-east-1"',
           'sagemaker_submit_directory': '"s3://sagemaker-us-east-1-302337575431/capstone_train_hpo-2024-12-19-18-37-17-842/s
          ource/sourcedir.tar.gz"'}
```

Figure 15. Query best hyperparameters

The hyperparameter tuning job launched the following 4 training jobs and selected the second one because it has the highest accuracy:

Training Job	Epochs	Batch Size	Learning Rate	Accuracy (correct count/ total count)
pytorch-training-241219-1837-001-fab32d4c	20	128	0.0008787037429267698	0.3251 (341/1049)
pytorch-training-241219-1837-002-69656183	20	256	0.0008015841478983083	0.3394 (356/1049)
pytorch-training-241219-1837-003-a16c2a46	20	128	0.0006949646180679423	0.3213 (337/1049)
pytorch-training-241219-1837-004-a2ace4e0	20	256	0.0008902437088970 31	0.3251 (341/1049)

Figure 16. Hyperparameter tuning training jobs

IV. Results

Model Evaluation and Validation

Once I got the best hyperparamters from the tuning job, I went back to re-train the model with the set of best hyperparameters. Then I deployed the trained model to an inference endpoint. Once the endpoint is in service, I made inference calls with images of the entire test set (1049 images), logged the results and plotted the accuracy for each class.

```
Object count '1' has accuracy: 57/124 (0.4597)
Object count '2' has accuracy: 112/231 (0.4848)
Object count '3' has accuracy: 64/268 (0.2388)
Object count '4' has accuracy: 38/238 (0.1597)
Object count '5' has accuracy: 76/188 (0.4043)
```

Figure 17. Test results

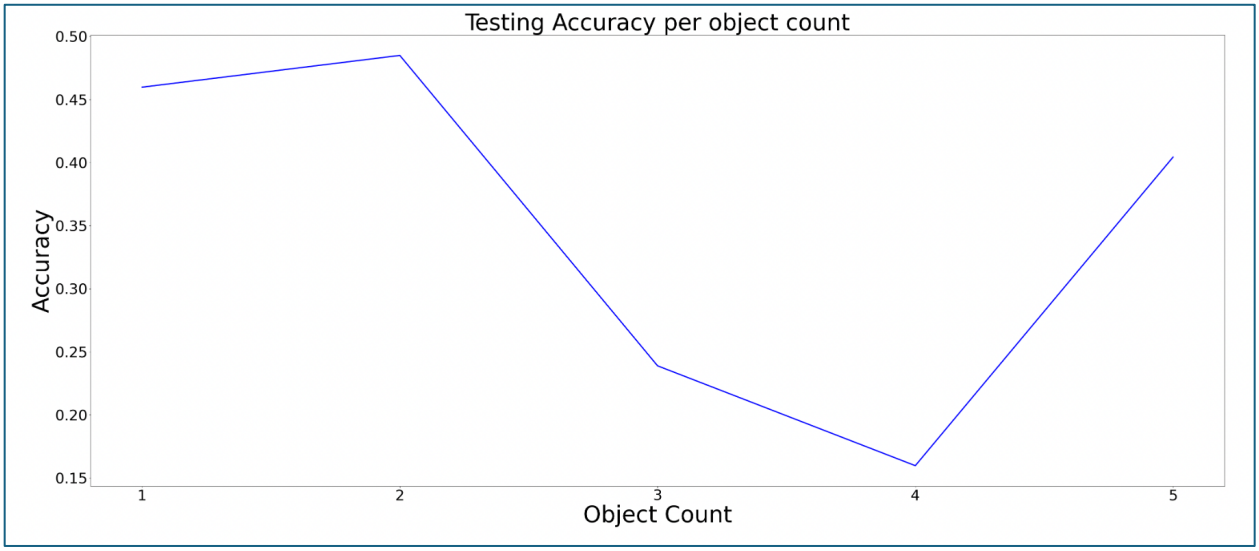


Figure 18. Test accuracy per class

I should also mention our inference endpoint can support prediction requests of both these content types, and return the same (consistent) prediction result:

1. image/jpeg (raw bytes from reading a JPEG file)
2. application/json (S3 URI indicating where the JPEG image is)

```
In [129]: # TODO: Run an prediction on the endpoint

          def run_prediction(input_data, content_type, expected):
              inference = predictor.predict(input_data, initial_args={"ContentType": content_type})

              print('Inference response: ')
              print(inference)

              max_index = np.argmax(inference)
              prediction_class = max_index + 1
              #prediction_probability = inference[0][max_index]

              print(f'Predicted object count: {prediction_class}, actual object count: {expected}')

              return prediction_class

In [130]: # run prediction with a JPEG file

          with open('enhanced_bin_images/test/1/105269.jpg', 'rb') as f:
              payload = f.read()
              run_prediction(payload, 'image/jpeg', 1)

          Inference response:
          [[0.019577527418732643, 0.3969842195510864, 0.16660255193710327, -0.14793714880943298, -0.5739536285400391]]
          Predicted object count: 2, actual object count: 1

In [133]: # run prediction with a S3 uri

          input_data = {
              's3_uri': os.path.join(s3_uri, 'test/1/105269.jpg')
          }

          run_prediction(json.dumps(input_data), 'application/json', 1)

          Inference response:
          [[0.019577527418732643, 0.3969842195510864, 0.16660255193710327, -0.14793714880943298, -0.5739536285400391]]
          Predicted object count: 2, actual object count: 1

Out[133]: 2
```

Figure 19. Inference endpoint supports both content types

I have also provisioned a public internet facing endpoint where I can send prediction requests to and receive the prediction result:

```
(base) chanm@chanm05-mac ~ %
(base) chanm@chanm05-mac ~ %
(base) chanm@chanm05-mac ~ % curl -X POST https://hsx05sm6ca.execute-api.us-east-1.amazonaws.com/inference -H "Content-Type: application/json" -d '{"s3_uri": "s3://mchan-capstone-project/test/1/105269.jpg"}'
{"event_source": "API Gateway", "s3_uri": "s3://mchan-capstone-project/test/1/105269.jpg", "inference_response": [[0.019577527418732643, 0.3969842195510864, 0.16660255193710327, -0.14793714880943298, -0.5739536285
400391]], "prediction_class": 2}%
(base) chanm@chanm05-mac ~ %
(base) chanm@chanm05-mac ~ %
(base) chanm@chanm05-mac ~ % hostname
chanm05-mac
(base) chanm@chanm05-mac ~ %
(base) chanm@chanm05-mac ~ %
(base) chanm@chanm05-mac ~ %
```

Figure 20. Access the ML system from public internet

The model performance has been extremely poor from the get-go, even I have tried many things, such as switching to pre-trained model of different architectures as mentioned previously, or even un-freezing the convolutional layers; the result is either worse, or even when there is improvement, the improvement is often very minimal.

However, I did build an end-to-end ML system leveraging several AWS services successfully, which fits the realm of ML Engineering; after all, improving the model performance is more of the jurisdiction of a Data Scientist and a domain expert (Image Processing Engineer in this use case), than that of a ML Engineer.

Justification

While the ML system I built is completely functional and can potentially add value to any inventory control process, it is not ready for prime time yet because neither the overall accuracy nor the per class accuracy is remotely close to our benchmark.

Overall accuracy	Lowest per class accuracy	Highest per class accuracy	Benchmark accuracy
0.3308 (347/1049)	0.1597	0.4848	0.80

V. Conclusion

Free-Form Visualization

Let us look at more example training images from the class of 'object count = 4', which did especially poorly, bearing in mind these images have already been enhanced, even a far more superior vision system (aka my eyes) could hardly tell the count, how can we expect our ML model to perform well?

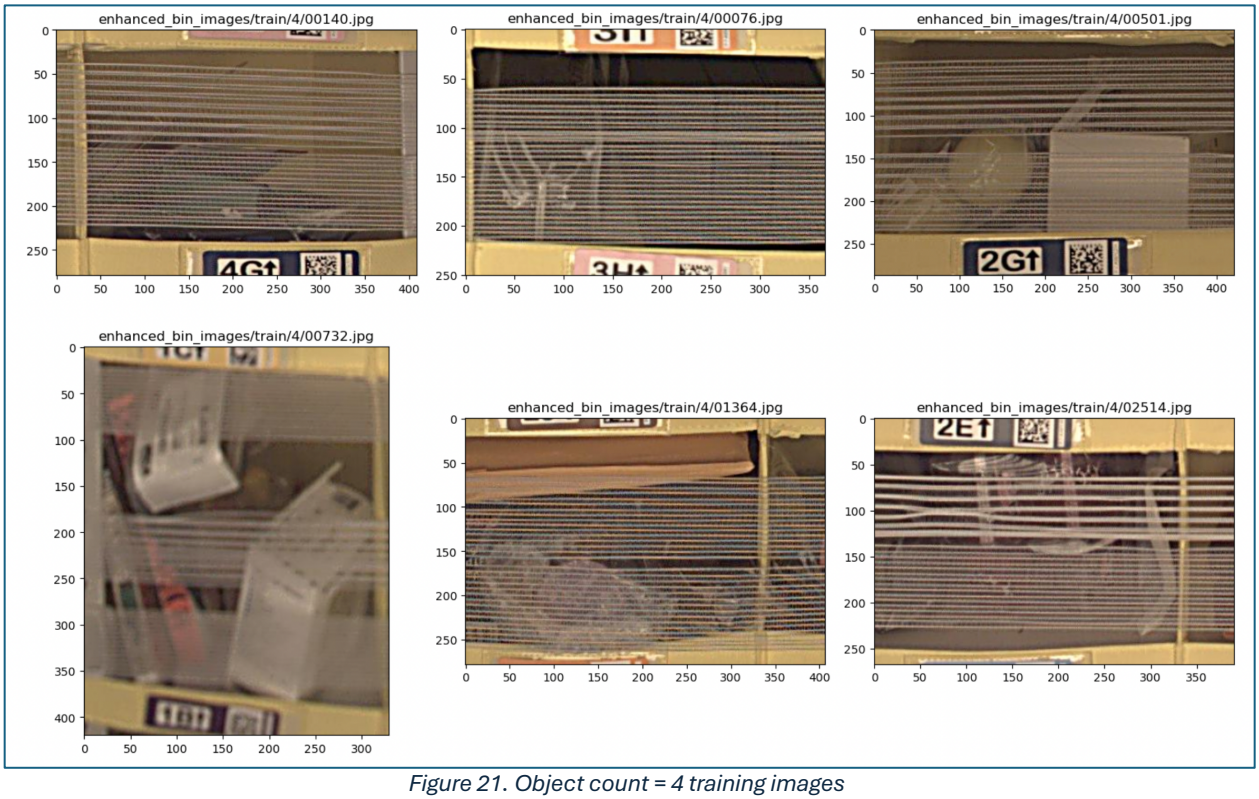


Figure 21. Object count = 4 training images

Reflection

In this project, I am building an end-to-end ML system using AWS SageMaker and other managed services, with a computer vision model at its core, to tell the number of objects in bins, using images captured by robots.

From this experience, I have learned how versatile SageMaker is, and yet how easy it is to launch a job or deploy a model, with an abundant choices of ML frameworks, and hardware architectures. It is also easy to integrate with other AWS services, for example, with just one line of code, I can tell SageMaker to read training data from S3, which is a lower cost storage solution and is especially suitable for large amount of data.

In the ML front, although there are areas I can try to improve the model performance, which will be discussed in the next section, the ROI will likely be low. It is because the fact that this computer vision model performs poorly has showcased the fundamental principle that a ML model is only as good as its training data, or simply put - garbage in, garbage out.

With that being said, I still think our end-to-end system will be able to improve any inventory control process, all it needs are better training data.

Improvement

There are at least a few ideas I can try to improve the model performance:

1. try more pre-trained CNN models
2. if there are no budget and time constraints, do not limit the maximum number of underlying training jobs the hyperparameter tuning job can launch, or set a much higher limit
3. do not freeze the convolutional layers, go back to update those weights, almost like training from scratch
4. consult with a Data Scientist instead of brute force style of trying all possible combinations of 1 to 3
5. consult with an Image Processing Engineer on possible ways to improve the image quality
6. explore different transforms
7. install better cameras on the robots
8. capture images before taping up the bins
9. use flash when capture the images

And approach the ideas in this order:

- 7, 8, 9
- 5, 6
- 4,1,2,3

Reference

1. J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large- scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848. keywords: {Large-scale systems;Image databases;Explosions;Internet;Robustness;Information retrieval;Image retrieval;Multimedia databases;Ontologies;Spine}, https://ieeexplore.ieee.org/document/5206848
2. Resnet34, Torchvision documentation, https://pytorch.org/vision/main/models/generated/torchvision.models.resnet34.html
3. Sasank Chilamkurthy, "Transfer Learning for Computer Vision Tutorial,", PyTorch Tutorials, https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
4. Resnet18, Torchvision documentation, https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html
5. Resnet50, Torchvision documentation, https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html#torchvision.models.resnet50