CNN Project: Inventory Monitoring

Sylvanus Quarm

February 16, 2022

Contents

1 Project Overview

Inventory monitoring, also called stock monitoring, is the process of ensuring the right amount of supply is available in an organization. With the appropriate internal and production controls, like ensuring that bins have the correct number of products or items, this practice ensures the company can meet customer demand and deliver financial elasticity.

Inventory monitoring enables the maximum amount of profit from the least amount of investment in inventory without affecting customer satisfaction. Done right, it can help avoid problems, such as out-of-stock (stockout) events.

Since distribution warehouses store and sell large quantities of goods, typically, house goods from multiple manufacturers, the manual processes are gradually fading out with the replacement of digital processes with some level of human intervention to help in the enumeration of stocks.

However, advancement in machine learning and deep neural networks can be a great turning point in the business of supply chain management (SCM).

N. K. Verma, T. Sharma, S. D. Rajurkar and A. Salour, "Object identification for inventory management using convolutional neural network," 2016 IEEE Applied Imagery Pattern Recognition Workshop (AIPR), 2016, pp. 1-6, doi: 10.1109/AIPR.2016.8010578. 5

2 Problem Statement

With the invent of automation, warehouses are becoming more flexible through the use of robotics. A typical operation of a working robot is to convey objects in a container called a bin. A system like this can be used to track inventory through bins scanning and make sure that delivery consignments have the correct number of items.

Classifying the content of an image by the count of objects present is a typical computer vision problem. To build this project we will use AWS SageMaker, Amazon Bin Image Dataset (ABIN), and good machine learning engineering practices to fetch data from a database, preprocess it, and then train a machine learning model.

ImageNet-based models like ResNet predicts the class of one object in an image on a simple background but it does not provide a count of multiple instances of either the same or different type.

Luckily, the metadata of the images in the ABIN provides a quantity attribute. Hence our task reduces to algorithmically matching the supplied image to the right image in the ABIN, and then extracting the count attribute as the prediction result. That is the reason why we will need to build one with our own CNN in this project, and overlay it on top of well-known CNN model architecture, e.g., VGG16, ResNet50, etc., pre-trained on the ImageNet dataset and update its top (head) layers for transfer learning. We then compare the accuracy of using pretrained and [not pretrained] CNN model. We again modify some layers of our chosen model.

3 Metrics

Similar to other classification problems, the CrossEntropyLoss function provided by PyTorch will be used to evaluate the train, validation and test losses of the CNN model. The overall accuracy of the classification cannot be used to evaluate the performance of the trained model, since the dataset is **imbalanced (All classes do not contain the same number of images).**

4 Data Exploration

The link to Amazon Bin Image Dataset (ABID) is provided by Udacity. Amazon uses a random storage scheme where items are placed into accessible bins with available space, so the contents of each bin are random, rather than organized by specific product types. Thus, each bin image may show only one type of product or a diverse range of products. Occasionally, items are misplaced while being handled, so the contents of some bin images may not match the recorded inventory of that bin. Creating a similar dataset like this will require a camera taking shots of the product catalogue at fixed angles.

After unzipping the ABID dataset, we can see one folder containing 50,000 images, and each one is identified by an image_id along with its metadata. Fig. 1 shows an example of an image in the ABID datasets whiles Fig. 2 shows the metadata. The corresponding metadata exists for each bin image and it includes the object category identification (Amazon Standard Identification Number, ASIN), quantity, size of objects, weights, normalizedName, quantity of each item, and so on. The size of bins varies depending on the size of objects in it. The Images are extracted from the source which are available in JPEG format and the target or label data is extracted from the corresponding JSON file. We can inspect the properties of these datasets, and double check that- Number of training samples: 6680, Number of testing samples: 836, Number of bin classes: 5. A sample metadata can be found here. https://aft-vbi-pds.s3.amazonaws.com/metadata/523.json

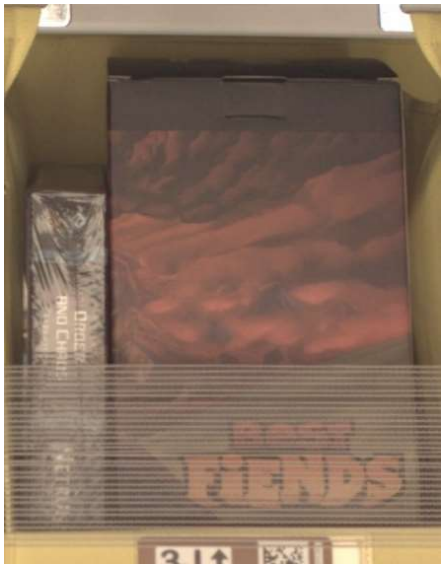

Figure 1: Example of an image from the dataset

img_id	data
Image ID	Metadata JSON object of the corresponding image
50000 unique values	Metrics not available
231573	{"BIN_FCSKU_DATA": {"9966338926": {"asin":"9966338926","height": {"unit":"IN","value":0.799999999184},"...

Figure 2: Example row in the AMID datasets

5 Data Preprocessing

Since the input images can be of different shapes of RGB channel values between 0 and 255, we need to normalize them for our CNN models. All pre-trained CNN models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 × H × W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

As the last step, since PyTorch models can only accept torch.tensor data type as inputs, all the preprocessed images along with their training labels need to be converted to tensors. We will then load images from the test folder as the input to our CNN model to test our model. This can also be done by invoking the url endpoint https://aft-vbi-pds.s3.amazonaws.com/bin-images/523.jpg .

In order to load batches of images (as well as their labels, if needed) from the built PyTorch datasets, I also need to build PyTorch dataloaders, which can be used as iterators in the training process. Fig. 4 shows the implementation codes I used to build the corresponding dataloaders.

```
parser.add_argument('--train', type=str, default=os.environ['SM_CHANNEL_TRAIN'])
parser.add_argument('--test', type=str, default=os.environ['SM_CHANNEL_TEST'])
args = parser.parse_args()
```

```
train_loader, test_loader = create_data_loaders(args.train, args.test, args.batch_size)
```

Figure 3: Generate dataloaders from create_data_loaders function

```
def create_data_loaders(data_train, data_test, batch_size):
    train_transform = transforms.Compose([
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.Resize(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

    test_transform = transforms.Compose([
        transforms.Resize(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

    trainset = datasets.ImageFolder(root=data_train, transform=train_transform)
    testset = datasets.ImageFolder(root=data_test, transform=test_transform)

    return (
        torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True),
        torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False))
```

Figure 4: Data augmentation and preprocessing

6 Algorithms and Techniques

Since, the number of classes do not match and the images are not the same as those found in the ImageNet dataset, we need to design our own CNN models. We cannot utilize a CNN model already pretrained on ImageNet since that will obviously lead to a low accuracy.

But first, the natural idea is to make classes match. Fig. 3 shows a summary of the neural network which is a customized version of the resnet-50 model.

```
import os
import json
import boto3
from tqdm import tqdm

def download_and_arrange_data():
    s3_client = boto3.client('s3')

    with open('file_list.json', 'r') as f:
        d=json.load(f)
        temp = d
    for data_path in ["train","test"]:
        for k, v in d.items():
            d.update({k:v[0:1024]}) if (data_path == "train") else d.update({k:v[512:-1]})
            print(f"Downloading Images with {k} objects")
            directory=os.path.join('data_path', k)
            if not os.path.exists(directory):
                os.makedirs(directory)
            for file_path in tqdm(v):
                file_name=os.path.basename(file_path).split('.')[0]+'.jpg'
                s3_client.download_file('aft-vbi-pds', os.path.join('bin-images', file_name),
                                        os.path.join(directory, file_name))
        d = temp

download_and_arrange_data()
```

Figure 6: Code to create test and train folder directories

My deep neural network consists of a few combinations of Conv2D layers, BatchNorm2d layers,ReLU activation functions, and MaxPooling2D layers for multiscale feature extraction, and thena global average pooling operation to get the averages of all filtered feature maps, whereafter the values are 1D tensors. Then I add two fully-connected layers with a dropout layer between themthat prevents overfitting. The first fully-connected layer is a hidden layer with the ReLU activation function followed by the dropout layer with probability of an element to be zeroed as 0.5 and the second one is the output layer that provides the probability results, if a softmax function follows. That is within the range of [1, 5]. For simplicity, an image of bin_type1, contains 1 item; bin_type2 contains 2 items, bin_type3 contains 3 items, bin_type4 contains 4 items, bin_type5 contains 5 items and so on.

```python
import torch
import torch.nn as nn

class custom_resnet(nn.Module):
  def __init__(self, resnet):
    super(custom_resnet, self).__init__()
    self.resnet = resnet
    self.classifier = nn.Sequential(
      nn.Dropout(),
      nn.Linear(1024, 4096),
      nn.ReLU(inplace=True),
      nn.Dropout(),
      nn.Linear(4096, 4096),
      nn.ReLU(inplace=True),
      nn.Linear(4096, 5),
      nn.Sigmoid()
    )

  def forward(self, x1, x2):
    x1 = self.resnet.conv1(x1)
    x1 = self.resnet.bn1(x1)
    x1 = self.resnet.relu(x1)
    x1 = self.resnet.maxpool(x1)

    x1 = self.resnet.layer1(x1)
    x1 = self.resnet.layer2(x1)
    x1 = self.resnet.layer3(x1)
    x1 = self.resnet.layer4(x1)

    x1 = self.resnet.avgpool(x1)
    x1 = x1.view(x1.size(0), -1)

    x2 = self.resnet.conv1(x2)
    x2 = self.resnet.bn1(x2)
    x2 = self.resnet.relu(x2)
    x2 = self.resnet.maxpool(x2)

    x2 = self.resnet.layer1(x2)
    x2 = self.resnet.layer2(x2)
    x2 = self.resnet.layer3(x2)
    x2 = self.resnet.layer4(x2)

    x2 = self.resnet.avgpool(x2)
    x2 = x2.view(x2.size(0), -1)

    concat = torch.cat([x1,x2],1)
    p = self.classifier(concat)
    return p
```

Figure 7: Implementation of the CNN model from scratch silverbottlep

After training for 20 epochs with ResNet **pretrained**, the accuracy of such model reaches **11%** i.e., 56 bin images out of 512 testing images are correctly classified. It has reached the expected goal that the test accuracy is greater than 10%.

Still, **11%** accuracy is a good starting point but not the end of the game. We would like to further improve the classification accuracy on the test dataset.

In the second phase, I used ResNet50 (**not pretrained**) model as the base model with pretrained parameter set to false, and modified the inner layers. Since we are working on a more specific image dataset, i.e., a ABID with 5 different bin classes, we set the final linear layer output to 5.

After training for another 20 epochs, I noticed that the train and validation losses are much lower and keeps decreasing after every epoch. That implies that such a model can achieve much better classification accuracy on the test dataset. After performing the test task, we see a **58%** accuracy. Of course, this is also slightly lower than the expected 60%accuracy on the test set as compared to silverbottlep award-winning results.

```
def net():
    model = custmodel.custom_resnet(models.resnet50())

    num_features = model.fc.in_features
    model.fc = nn.Sequential(
        nn.Linear(num_features, 5)
    )

    return model
```

Figure 8: Custom Resnet50 model based on ResNet50 model layers

7 Implementation and Refinement

Fig. 7 shows the PyTorch implementation of the CNN model from scratch. Since we utilize GPU(s) to train the model, we can move the entire model, i.e., all the model weights to the GPU(s).

Since it is a multiclass classification problem, cross entropy loss needs to be used. I also chose the Adam optimizer to update the network parameters (weights), as shown in Fig. 9.

The training process is designed in a standard way as a train function: for each training epoch,I load the training and validation data batch by batch from the corresponding dataloaders, move them to GPU if applicable, then calculate the losses and update the model parameters accordingly.For each batch operation, as a PyTorch standard, I need to clear all optimized variables before calculating the model prediction. After calculating the training loss by applying the loss function withrespect to the prediction and the loss, I perform backward propagation and update the parameters by one step. I save the model if validation loss has decreased and print out log messages includinglosses and whether the model has been saved to the console output. The train function returns the trained model as its output.

Since the train function is quite long I have not placed the codes here, please refer to the link https://github.com/vanusquarm/Inventory-Monitoring-A-Deep-Learning-Approach for full details. The test function is also designed similarly. We load the test dataset batch by batch from the built dataloaders, infer the model outputs and then get the element index that has a largest value in the 2-dimension classifier output vector. We can query the class name with such index.

```
loss_criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=args.lr)
```

Figure 9: My choices of loss function and optimizer

```
# Create training estimator
estimator = PyTorch(
    entry_point = "train.py",
    base_job_name = "bin-imgclassification",
    role = role,
    instance_count = 1,
    instance_type = "ml.g4dn.2xlarge",
    hyperparameters = hyperparameters,
    framework_version = "1.8",
    py_version = "py36",
    output_path = f"s3://{bucket}/output/"
)
```

```
# Fit estimator
train_data = f's3://{bucket}/train/'
test_data = f's3://{bucket}/test/'
estimator.fit({'train': train_data, 'test':test_data}, wait = True)
```

```
from sagemaker.pytorch import PyTorchModel

pytorch_model = PyTorchModel(model_data=estimator.model_data,
                             role=role,
                             entry_point='inference.py',
                             py_version='py36',
                             framework_version='1.8')


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

```
from sagemaker.serializers import IdentitySerializer
predictor.serializer = IdentitySerializer("image/jpeg")
```

```
def identify_bin_type(image_url):
    img_bytes = requests.get(image_url).content  #get image bytes in any format
    image = Image.open(io.BytesIO(img_bytes)) #convert bytes to file object and open

    buf = io.BytesIO()
    image.save(buf, format="JPEG") #convert file to jpeg BytesIO object
    response = predictor.predict(buf.getvalue()) #get the bytes from the object and pass it to the predict function

    return image, response
```

```
import numpy as np
bin_index = np.argmax(response, 1).item()
per_acc = response[0][bin_index]*100

print(f"The image contains {bin_index+1} items with a prediction accuracy of {per_acc:.2f}%")
```

Figure 10: Top-level algorithm Implementation in Sagemaker

8 Model Evaluation, Validation, and Justification

After training the CNN models we can evaluate their performances from the metrics of the running epochs. The accuracy of the model reaches **58%** at the 14th epoch. The accuracy is better than the project target of 60%.

This indeed proves that our custom model can be used as a good feature extractor to obtain all possible features from product images, like bin images in this project. Such a featureextractor are way better than using an already pretrained ImageNet model like ResNet-50 (pretrained =True). If the bin image does not match any of the classes, it reports a message indicating such information, or better still produces an prediction probability of less than 10%. Fig. 9 summarizes such a top-level algorithm.

I also did some home-made tests by invoking the deployed endpoint with some of my own images and other images I found online, as shown in Fig. 10. All the classification results of bin imagēs are correct, while some classification resultsthat look for the mostly resembled bin-type. Also, for curiosity, I placed a plain monochromic image, and the prediction failed with zero accuracy.

9 Conclusion

In this project, I implemented an image classifier which will tell the type(class) of bin an input image is. A bin may contain from 1 to 5 items depending on its size measured by its length, width and height.

In order to classify 5 different classes, I have built custom untrained ResNet50 model with some layers removed, some optimized and modified, and others newly created. I finally imported the custom-resnet model from the custmodel module, and passed in the untrained ResNet50 model to classify bins. This led to a classification accuracy of 86%.

Such a bin classifier algorithm also provides quite satisfactory results on the images thatI additionally provide (humper). The algorithm can classify the bin quite well that it reports an error when itsees an image containing more than 5 items.

I think there are still rooms to improve the CNN models. Below are some of my thoughts that could potentially further improve the performance.

- I could use a better model pretrained on unit images of the products found in amazon fulfillment center as the base model for transfer learning.
- I could employ the state-of-the-art Auto Machine Learning (AutoML) technique to find the best neural network architecture for such problem. Though it would take longer time to find best architecture given a metric and train it.
- I could apply real-time data augmentation technique to increase the size of the training set.
- I could train more epochs, along with a learning rate scheduler or early stopping mechanism to get better prediction accuracy.

Projecting into future, we can also use the classification in the [1-5] classes to predict images containing 6 or more items. This can potentially outperform models that choses object detection algorithm over image classification for object counting.

10 Acknowledgement

I would like to thank Udacity for providing such opportunity to let me work on these real-life projects. Also, I would like to thank the project reviewers for their time and efforts in reviewing my submissions.