Inventory Monitoring at Distribution Centers

1. Project Overview:

   The project focuses on creating an automated inventory monitoring system for distribution centers. These centers often use robots to transport bins filled with multiple items as part of their logistics operations. The objective is to develop a model that accurately counts the number of objects within each bin. This will aid in inventory management and ensure the correct number of items are in each delivery consignment. The project will leverage AWS SageMaker and robust machine learning engineering practices to process data from a database, and then train a machine learning model. The project serves as a comprehensive showcase of learned end-to-end machine learning engineering skills from the nanodegree.

2. Problem Statement:

   The main challenge to address is designing a model that can accurately count the number of objects in each bin based on bin images. This task will be achieved by utilizing the Amazon Bin Image Dataset, which contains 500,000 images of bins with varying numbers of objects. Metadata files associated with each image provide details such as object count, object type, and image dimensions. The model could be built using a pre-existing convolutional neural network, or by designing a unique neural network architecture. However, the model training must be performed using AWS SageMaker.

3. Metrics:
   The performance of the model will be evaluated using three metrics: accuracy, and sparse categorical accuracy.

a. Accuracy: This is a primary metric that calculates the proportion of correct predictions made by the model. Given that the task is a classification problem (classifying bins based on the number of items they contain), accuracy serves as a direct and easily understandable measure of performance.

b. Sparse Categorical Accuracy: This metric is particularly suitable for multiclass classification problems where the labels are integers and not one-hot encoded. In this scenario, each bin can be considered as belonging to one of several classes corresponding to the number of items in the bin. The sparse categorical accuracy compares the predicted class (the one with the highest probability in the model's output) directly with the true class. If they match, it's considered a correct prediction, otherwise, it's not. This gives us the proportion of images for which the model accurately predicted the number of items in the bin.

These metrics were chosen because they offer different perspectives on the model's performance, ranging from a general overview (accuracy).

2. Analysis:

Each class is represented by a subfolder in the 'train_data' folder, and the name of each subfolder (or class) corresponds to the number of objects visible in the images it contains. There are five classes in total, ranging from 1 to 5.

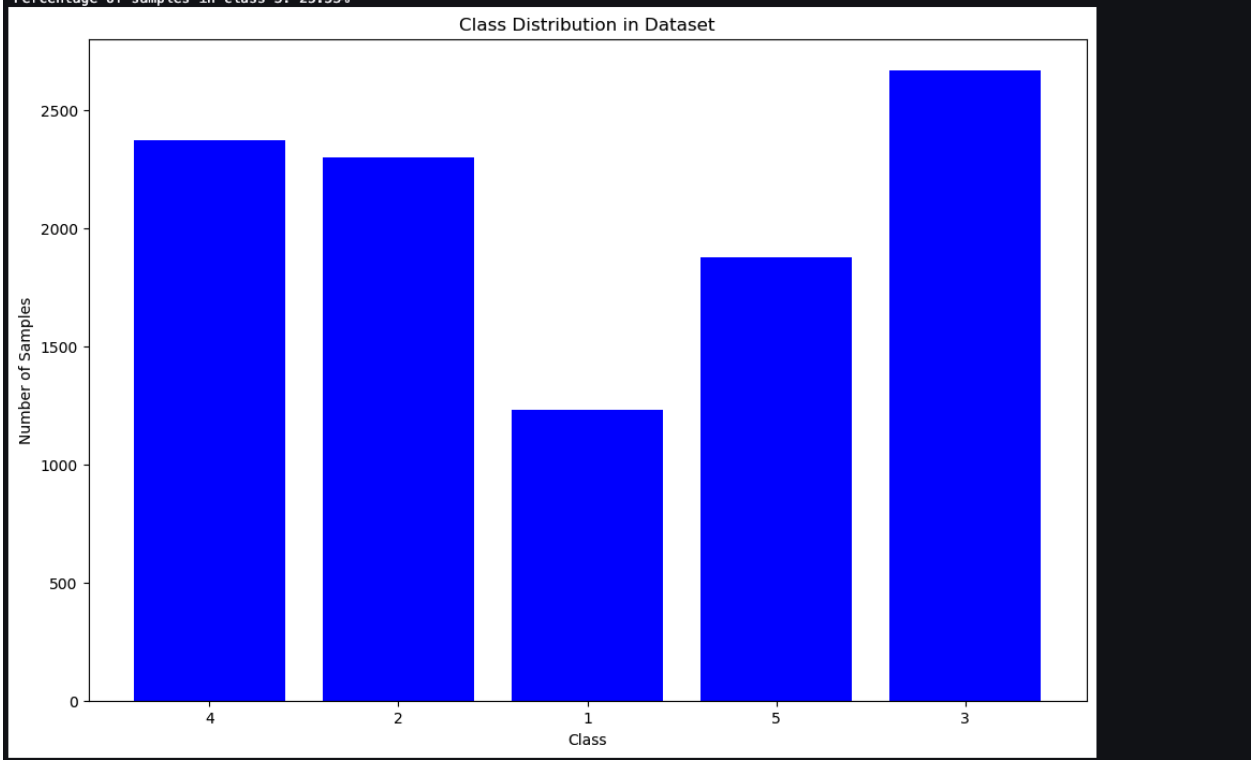
The detailed distribution of the dataset is as follows:

1. **Class 1:** Contains 1,228 samples, which makes up approximately 11.76% of the total data.
2. **Class 2:** Contains 2,299 samples, comprising roughly 22.02% of the total data.
3. **Class 3:** Contains 2,666 samples, accounting for around 25.53% of the total data.
4. **Class 4:** Contains 2,373 samples, which represents about 22.73% of the total data.
5. **Class 5:** Contains 1,875 samples, which is approximately 17.96% of the total data.

In total, there are 10,441 samples in this dataset.

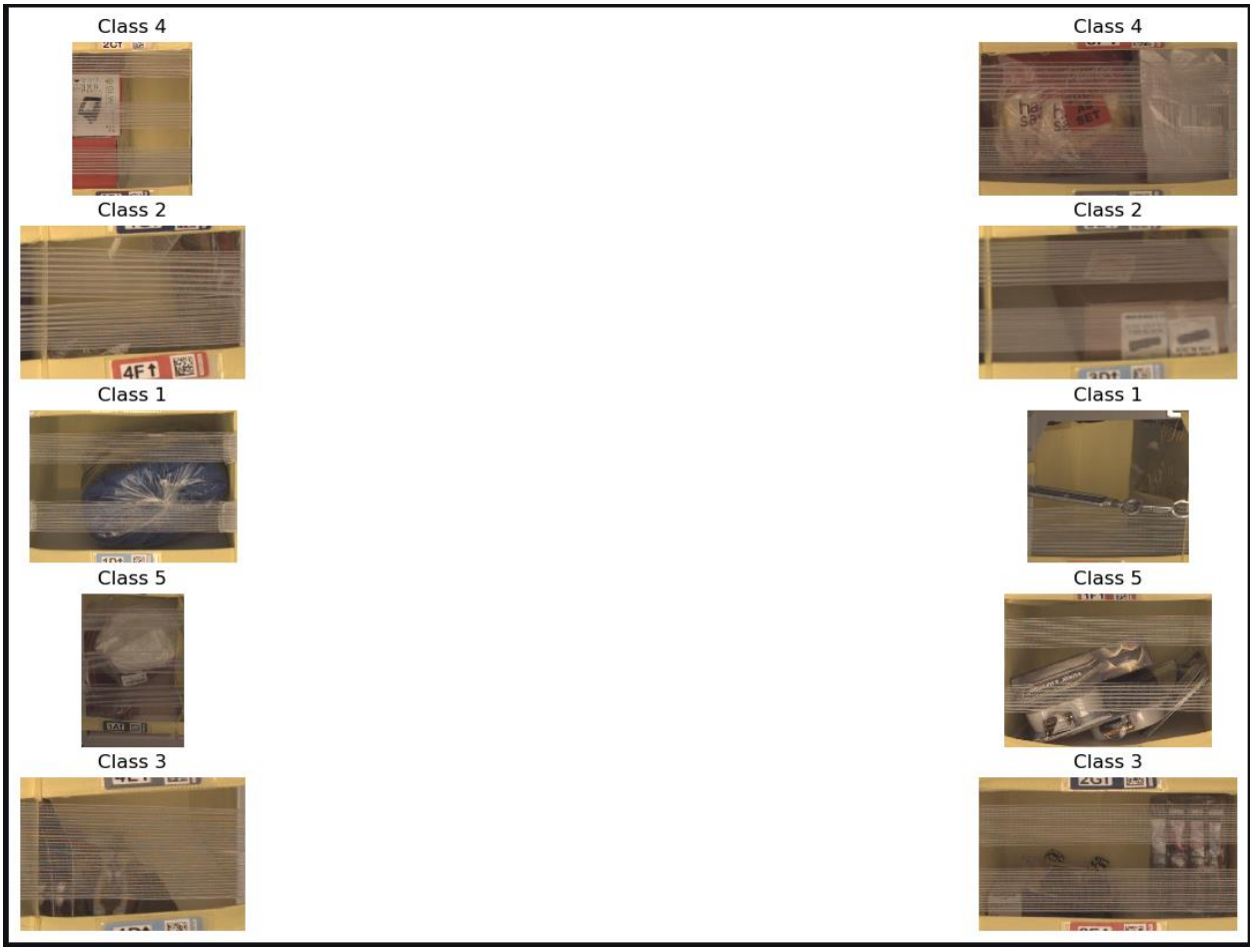
The distribution of samples across these classes is imbalanced. Classes 2, 3, and 4 have over 2,000 samples each, while classes 1 and 5 have fewer than 2,000 samples. Class 3 contains the most samples, and Class 1 contains the least. This imbalance could potentially influence the performance of a machine learning model trained on this dataset.

```
Class 4 has 2373 samples.
Class 2 has 2299 samples.
Class 1 has 1228 samples.
Class 5 has 1875 samples.
Class 3 has 2666 samples.
Total samples in dataset: 10441
Percentage of samples in class 4: 22.73%
Percentage of samples in class 2: 22.02%
Percentage of samples in class 1: 11.76%
Percentage of samples in class 5: 17.96%
Percentage of samples in class 3: 25.53%
```



In conclusion, an effective pipeline will need to take into consideration the imbalanced nature of this dataset. Techniques like data augmentation for underrepresented classes or different handling strategies for imbalanced data can be used.

Example Images:



Report on Chosen Techniques and Benchmark

In this image classification project, I have chosen to use the ResNet50 Convolutional Neural Network (CNN). This decision is based on ResNet50's proven capability to effectively train deep networks through its unique "skip connections," making it suitable for the multiclass classification task at hand.

I've opted for the Sparse Categorical Crossentropy loss function due to the nature of the problem. Given that the classification task involves mutually exclusive classes represented as integers, this loss function provides computational efficiency without compromising on performance.

For benchmarking, a simple CNN model serves as the reference. The goal is to illustrate that the ResNet50 model surpasses this basic benchmark in terms of accuracy, justifying its use despite the increased complexity. By comparing the performance of ResNet50 to this benchmark, the value of a more complex network architecture in this context is demonstrated.

4. Methodology

The primary methodology for this project involved the preprocessing of data to create a more structured dataset, particularly suitable for training a deep learning model. The initial dataset was a single folder, 'train_data', containing subfolders for each class of images. The aim was to reorganize this data into three distinct categories - training, validation, and testing.

```
def create_data_splits(source_folder='train_data', target_folder='binDataset', train_size=0.7, valid_size=0.15):
    # Ensure the target directory exists
    if not os.path.exists(target_folder):
        os.makedirs(target_folder)

    classes = [folder for folder in os.listdir(source_folder)]
    for class_ in classes:
        print(f"Processing class: {class_}")

        # Create target subdirectories
        os.makedirs(os.path.join(target_folder, 'train', class_), exist_ok=True)
        os.makedirs(os.path.join(target_folder, 'valid', class_), exist_ok=True)
        os.makedirs(os.path.join(target_folder, 'test', class_), exist_ok=True)

        # List all images in this class
        image_files = os.listdir(os.path.join(source_folder, class_))

        # Shuffle and split data
        np.random.shuffle(image_files)
        train, valid, test = np.split(image_files, [int(train_size*len(image_files)), int((train_size+valid_size)*len(image_files))])

        # Copy files into respective splits
        for image_file in train:
            shutil.copy(os.path.join(source_folder, class_, image_file), os.path.join(target_folder, 'train', class_, image_file))
        for image_file in valid:
            shutil.copy(os.path.join(source_folder, class_, image_file), os.path.join(target_folder, 'valid', class_, image_file))
        for image_file in test:
            shutil.copy(os.path.join(source_folder, class_, image_file), os.path.join(target_folder, 'test', class_, image_file))

create_data_splits()
```

Implementation:

The implementation phase of this project involved creating a custom ResNet50 model and defining functions for training and testing the model.

The **CustomNet** class defines the modified ResNet50 model. This class utilizes the pretrained ResNet50 model provided by PyTorch's torchvision package, modifies its last fully connected layer to match the number of output classes (5, in this case).

The **train** function is used to train the model on the training dataset. It takes as input the model, the training DataLoader, the criterion (loss function), the optimizer, the device to train on (CPU or GPU), and the number of epochs for which to train. In each epoch, the function performs a forward pass on the data, calculates the loss, backpropagates the gradients, and updates the model's weights.

The **test** function is used to evaluate the model's performance on the validation dataset. This function loops over the validation data, makes predictions using the trained model, and compares these predictions with the actual labels to compute the overall accuracy.

The **create_data_loaders** function prepares DataLoader objects for the training and validation datasets. The function applies necessary transformations to the images, including resizing them to 224x224 pixels (required by the ResNet50 model), converting them into tensors, and normalizing them.

The **main** function ties all these components together. It initializes the model, the loss function, and the optimizer. It also prepares the DataLoader objects and then calls the **train** and **test** functions to train the model and evaluate its performance.
Lastly, the command-line argument parser allows for dynamic specification of training parameters such as the number of epochs, learning rate, and batch size. This makes the code more flexible and easier to fine-tune.

5. Results:

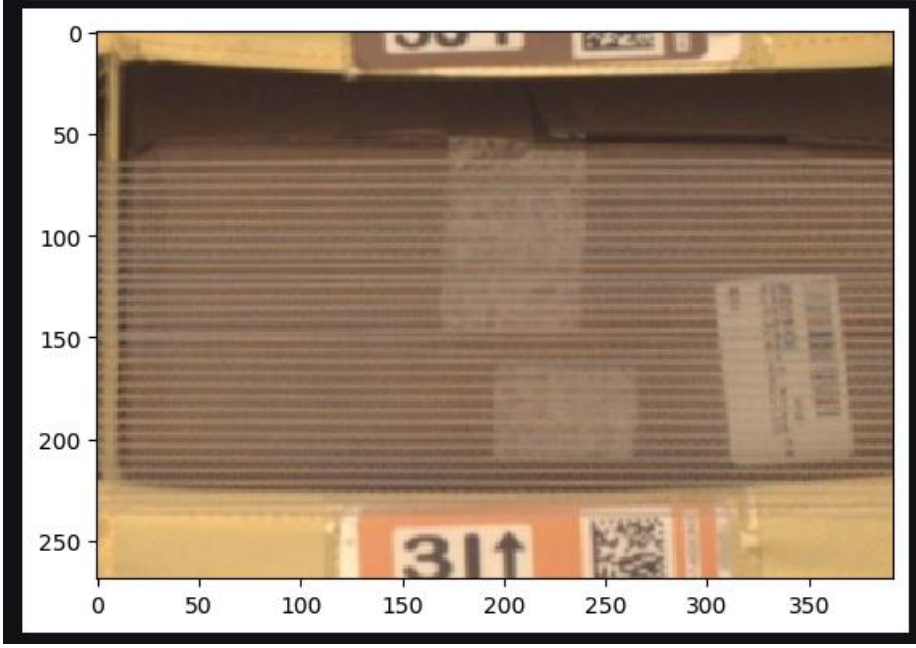
   The deployment and prediction process has been successfully completed using Amazon SageMaker. The deployed model is a PyTorch model, and the instance used for deployment is 'ml.g4dn.xlarge'.

   During deployment, the model artifact was repacked along with the inference script and any dependencies into a single tar.gz file. This was then uploaded to a specified S3 bucket. Once the model was successfully repacked and uploaded, an endpoint was created for model serving.

   For testing the endpoint, an image located at './binDataset/test/1/00048.jpg' was read as bytes and sent to the endpoint for prediction. The ContentType used for the prediction request was 'application/x-image' which indicates that the data being sent to the endpoint is an image file.

   The response received from the model prediction indicates that the class predicted for the provided image is '1'.

```
# Display the image
img = mpimg.imread(image_path)
imgplot = plt.imshow(img)
plt.show()
```



In conclusion, the deployment process was successful and the model is making predictions as expected. The next steps could be a more extensive testing of the model's prediction capabilities and possibly setting up auto-scaling rules to handle varied traffic to the endpoint. Additionally, monitoring the endpoint's performance to ensure continued performance and functionality is also recommended.

Project Links:

Github: https://github.com/Its-suLav-D/Inventory-Monitoring-At-Distribution-Centers/tree/master/starter