Inventory Monitoring at Distribution Centers (Amazon Bin Image counting)

Mohsen Mahmoodzadeh
NLP Engineer at Part AI Research Center
1mohsen.smm@gmail.com

Abstract

Distribution centers often use robots to move objects as a part of their operations. Objects are carried in bins which can contain multiple objects. Sometimes, objects are misplaced while being handled, resulting in a mismatch between the recorded bin inventory and its actual content. A system that can facilitate inventory tracking and ensure complete delivery of consignments by counting the number of items in each bin will be so valuable. In this work, we present a solution which can handle the mentioned challenge. The solution is based on one of the extensions of EfficientNet, a proposed architecture that yields better results while using less computation resources w.r.t the previous counterparts. Our focus is on some subset of the dataset which compromises between amount and balance of the data. Our experiment shows that our model has a defensible and competitive performance with previous works on this particular subset of the dataset.

Definition

Project Overview and Problem Statement

Distribution centers often use robots to move objects as a part of their operations. Objects are carried in bins which can contain multiple objects. Sometimes, objects are misplaced while being handled, resulting in a mismatch between the recorded bin inventory and its actual content.

A system that can facilitate inventory tracking and ensure complete delivery of consignments by counting the number of items in each bin will be so valuable.

Specifically and with more scientific and precise literature, our mentioned application is an object counting task. The goal of the object counting task is to count the number of object instances in a single image or video sequence. It has many real-world applications such as traffic flow monitoring, crowdedness estimation, and product counting.
We count individual instances separately, which means if there are two same objects in the bin, we count them as two.

In Figure 1, we can see an example of the input and output of this task:

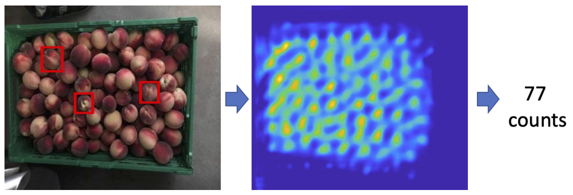


Figure 1: An example object counting task

Analysis

Data Exploration

For the mentioned above task, we use Amazon Image Bin Dataset. The dataset contains 536,435 bin JPEG 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. We apply an EDA on the dataset to know it better.

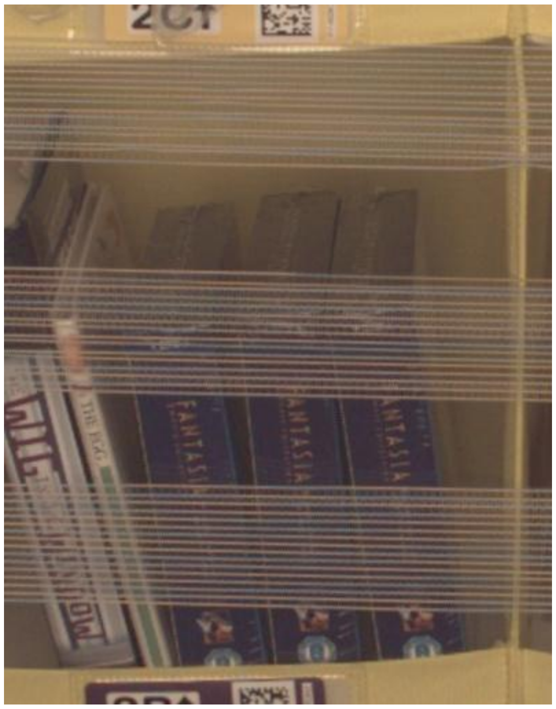


Figure 2: A sample of the dataset with five objects

The dataset has 536,435 JPEG files in total in alignment with metadata files. Due to file naming rules, they can be divided into three categories:

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

Furthermore, the capstone repository provides us with a chosen subset of the dataset, file_list.json. We'll investigate that later.

1. 1200 files with 1~4 digits in their file names(1.jpg to 1200.jpg)

Table 1: The main statistics of Group 1 of dataset

count	mean	std	min	max
1200	2.5	1.708	0	5

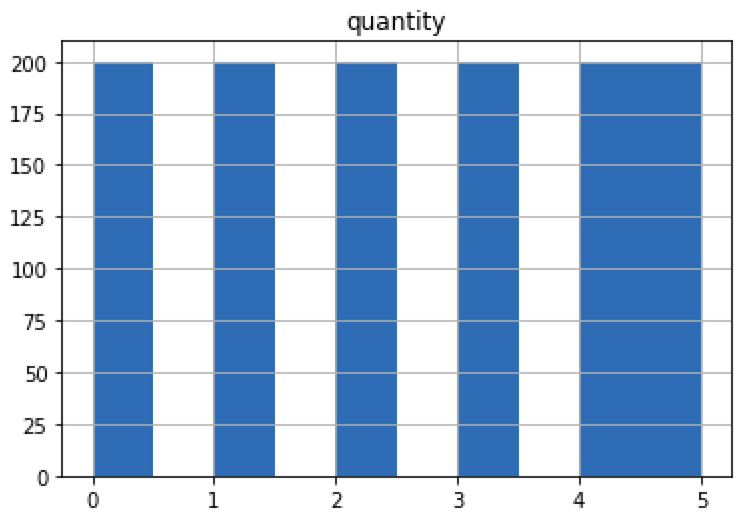


Figure 3: Class distribution of Group 1 of dataset

As we can see, Group 1 is distributed evenly and has no imbalance between different quantities.

2. **99,999 files with 5 digits in their file names(00001.jpg to 99999.jpg)**

Table 2: The main statistics of Group 2 of dataset

count	mean	std	min	max
99,999	5.123	4.758	0	209

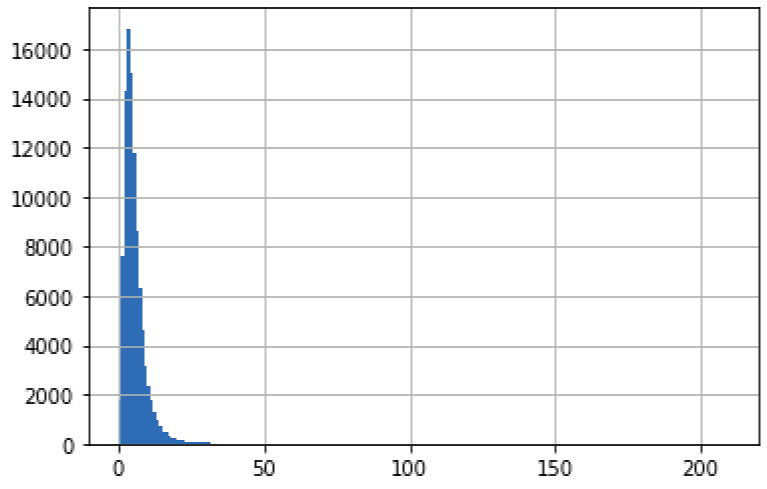


Figure 4: Class distribution of Group 2 of dataset

As we can see, Group 2 has a skewed distribution and has an intense imbalance between different quantities.

3. 10,441 files in file_list.json

Table 3: The main statistics of Group 3 of dataset

count	mean	std	min	max
10441	3.131	1.272	1	5

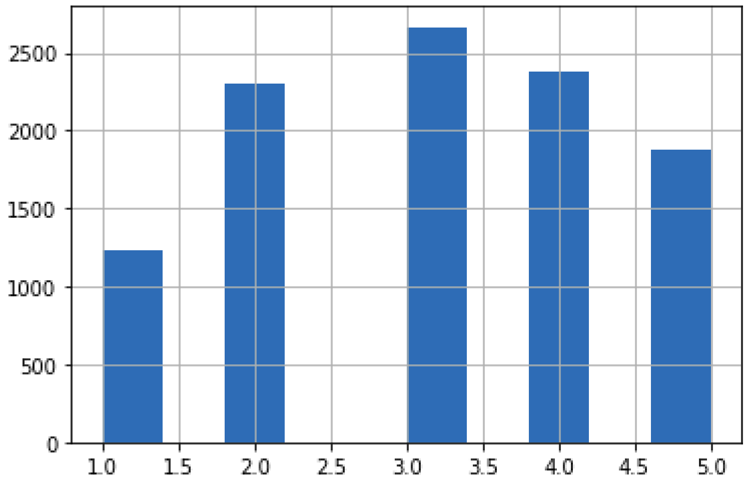
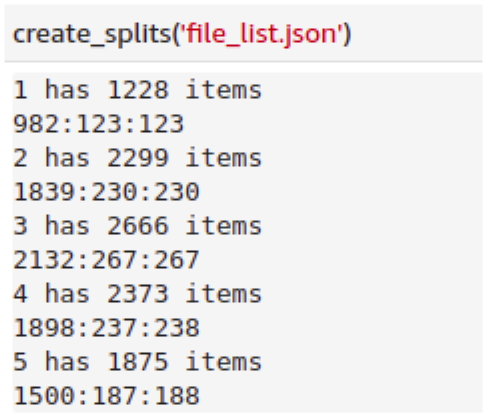


Figure 5: Class distribution of Group 3 of dataset

As we can see, Group 3 is somewhere in between Group 1 and Group 2; It has a less skewed distribution than Group 2 and has a bigger frequency than Group 1. So these properties make file_list.json a good start point to train a model.

We then split file_list.json with 80-10-10 ratios for train, validation and test respectively.



```
create_splits('file_list.json')
```

```
1 has 1228 items
982:123:123
2 has 2299 items
1839:230:230
3 has 2666 items
2132:267:267
4 has 2373 items
1898:237:238
5 has 1875 items
1500:187:188
```

Figure 6: The statistics of splits of file_list.json

Algorithms and Techniques

In this section, we present our solution which tries to integrate the pros of previous solutions and yet reduce the cost of computation.

In general, the steps we did can be listed as follows:

1. Use transfer learning to fine-tune a pre-trained EfficientNet_B4 model on our dataset.
2. Hyperparameter tuning
3. Training and evaluation

For model evaluation, the following metrics will be calculated:

❖ Accuracy

$$accuracy = \frac{1}{N}\sum_{i=1}^{N}\frac{p_i}{g_i}$$

Where:

- N is the number of data
- p_i is the number of counted objects(at the i th image) by the algorithm
- g_i is the number of total objects

❖ RMSE

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (p_i - g_i)^2}$$

Where:

- N is the number of data
- p_i is the number of counted objects(at the ith image) by the algorithm
- g_i is the ground truth or the number of counted objects(at the ith image) by the algorithm()

❖ F1 Score

$$F1\,Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Where:

- $Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$
- $Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$

Benchmark

Convolutional Neural Networks (ConvNets) are commonly developed at a fixed resource budget, and then scaled up for better accuracy if more resources are available. Compound scaling (Tan et al.) is a kind of model scaling which uniformly scales all dimensions of depth/width/resolution using a simple yet highly effective compound coefficient.

Based on this idea, a new baseline network can be designed using neural architecture search(NAS) and scaled up to obtain a family of models, called EfficientNet, which achieve much better accuracy and efficiency than previous ConvNets.

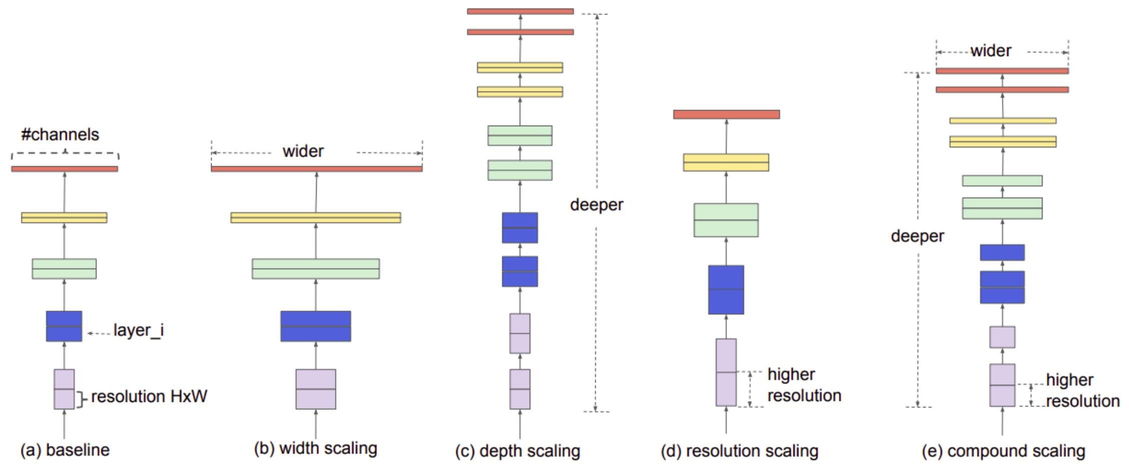


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

Figure 7: Model scaling methods w.r.t EfficientNet paper

In Table 4, we can see the performance statistics of the models which were used in the previous works (except EfficientNet_B4 which is our candidate model).

Table 4: Performance statistics(on ImageNet-1K) of the candidate models

	Acc@1	Params
ResNet18	69.758	11.7M
ResNet34	73.314	21.8M
ResNet50	80.858	26.6M
EfficientNet_B0	77.692	**5.3M**
EfficientNet_B4	**83.384**	19.3M

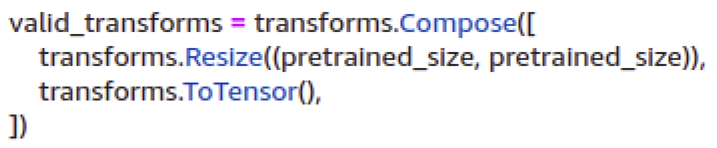
Due to Table 4, we consider the pretrained EfficientNet_B4 model as our baseline/benchmark model. EfficientNet_B4 is not the model with the least number of parameters; However, the goal is to find a balance between the accuracy of the model and the number of its parameters. Considering this condition, EfficientNet_B4 is a reasonable and economical choice between our candidates.

Methodology

In this section, we introduce the experiment details including the data preprocessing, implementation procedure, improvements and refinements.

Data Preprocessing

For benchmark, we just resize the image by 224*224 for each split and then convert it to a tensor to be prepared for the model to process.

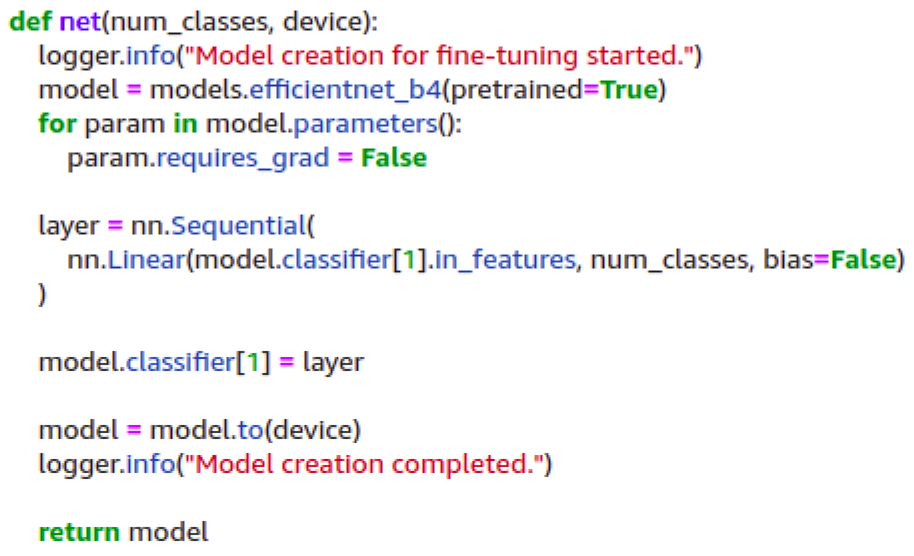


```
valid_transforms = transforms.Compose([
    transforms.Resize((pretrained_size, pretrained_size)),
    transforms.ToTensor(),
])
```

Figure 8: An example of transformer for benchmark model

Implementation

The benchmark model is implemented as shown in the following snippet code:

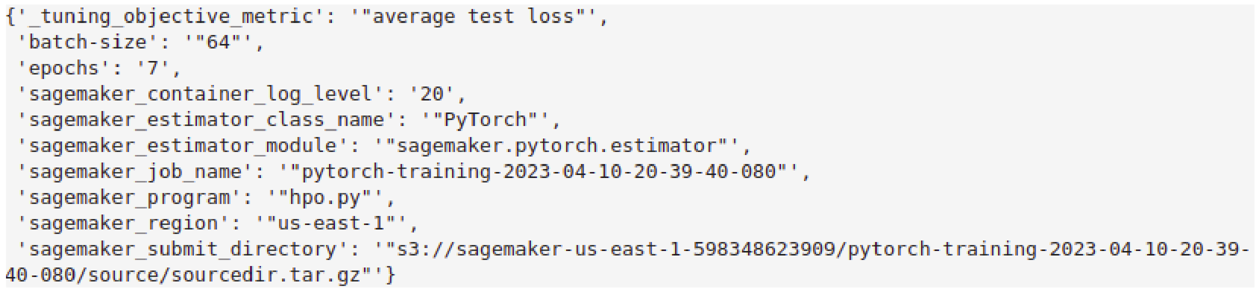


```
def net(num_classes, device):
    logger.info("Model creation for fine-tuning started.")
    model = models.efficientnet_b4(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False

    layer = nn.Sequential(
        nn.Linear(model.classifier[1].in_features, num_classes, bias=False)
    )

    model.classifier[1] = layer

    model = model.to(device)
    logger.info("Model creation completed.")

    return model
```

Figure 9: The main architecture of benchmark model

It’s worth noting that we freezed our base model parameters to reduce the computation and time cost. Certainly the training on the whole model may yield better performance but for now, we ignore it for simplicity.

The benchmark model is hyperparameter-tuned(on “batch size” and “epochs” hyperparameters) and trained which yields the following configuration as the best one:



```
{'_tuning_objective_metric': '"average test loss"',
 'batch-size': '"64"',
 'epochs': '7',
 'sagemaker_container_log_level': '20',
 'sagemaker_estimator_class_name': '"PyTorch"',
 'sagemaker_estimator_module': '"sagemaker.pytorch.estimator"',
 'sagemaker_job_name': '"pytorch-training-2023-04-10-20-39-40-080"',
 'sagemaker_program': '"hpo.py"',
 'sagemaker_region': '"us-east-1"',
 'sagemaker_submit_directory': '"s3://sagemaker-us-east-1-598348623909/pytorch-training-2023-04-10-20-39-
40-080/source/sourcedir.tar.gz"'}
```

Figure 10: The best hyperparameters for benchmark model

We also prepared the possibility of using debugger and profiler to have a comprehensive look at computations on our machines.

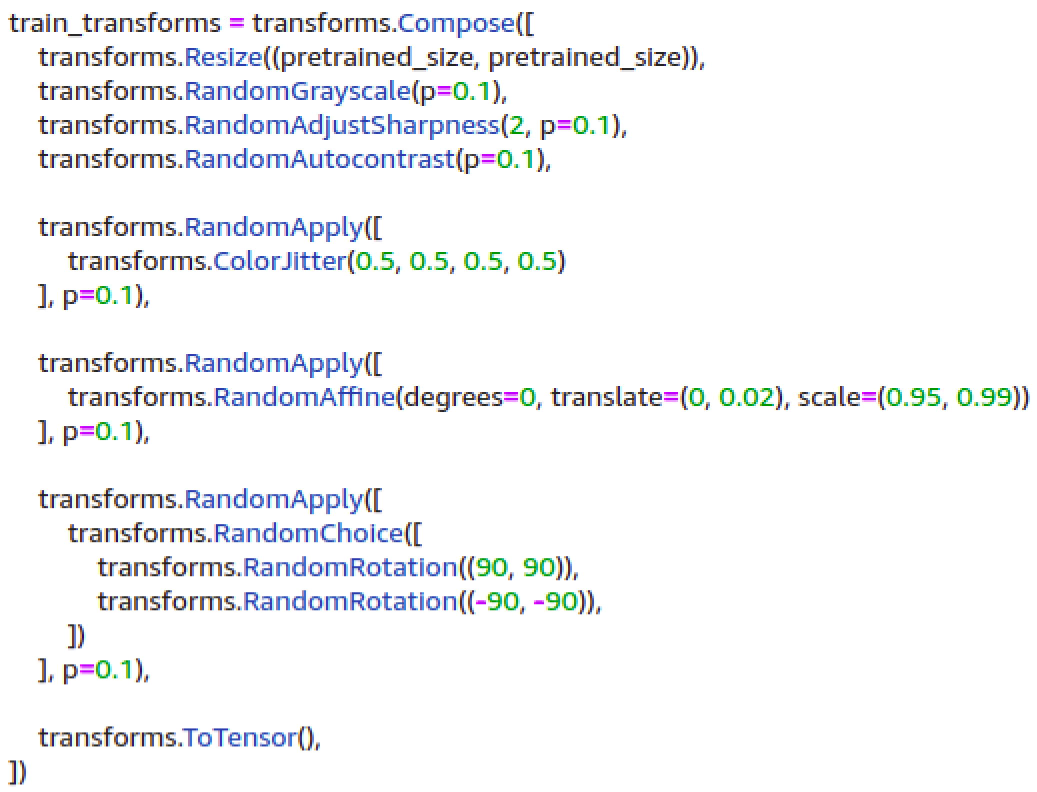
These implementations are also done for our improved model which we'll talk about later.

Refinement

Due to the performance of the benchmark model on our data, We found it valuable to improve the model and re-train and re-evaluate it. The improvements and refinements are as follows:

Data augmentation

For improvement step, we augment train data by the following configuration:



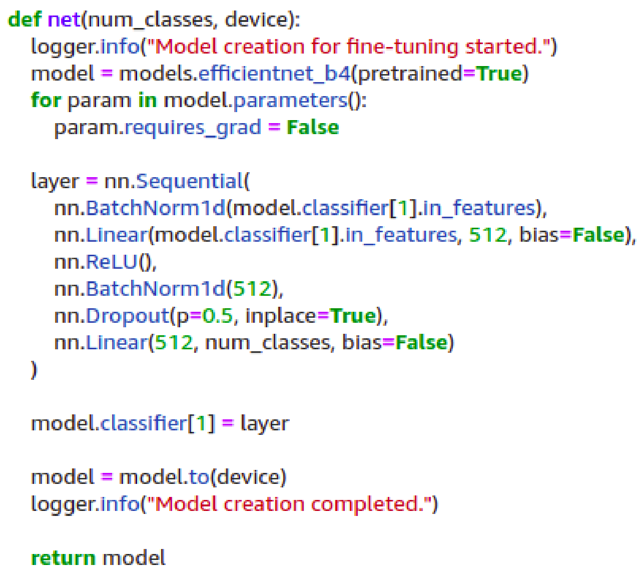
```
train_transforms = transforms.Compose([
    transforms.Resize((pretrained_size, pretrained_size)),
    transforms.RandomGrayscale(p=0.1),
    transforms.RandomAdjustSharpness(2, p=0.1),
    transforms.RandomAutocontrast(p=0.1),

    transforms.RandomApply([
        transforms.ColorJitter(0.5, 0.5, 0.5, 0.5)
    ], p=0.1),

    transforms.RandomApply([
        transforms.RandomAffine(degrees=0, translate=(0, 0.02), scale=(0.95, 0.99))
    ], p=0.1),

    transforms.RandomApply([
        transforms.RandomChoice([
            transforms.RandomRotation((90, 90)),
            transforms.RandomRotation((-90, -90)),
        ])
    ], p=0.1),

    transforms.ToTensor(),
])
```

Figure 11: An example of transformer for improved model

The transformations for validation and test splits are the same as we did for benchmark.

Model head architecture changing

We also change the model head architecture as show below:



```python
def net(num_classes, device):
    logger.info("Model creation for fine-tuning started.")
    model = models.efficientnet_b4(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False

    layer = nn.Sequential(
        nn.BatchNorm1d(model.classifier[1].in_features),
        nn.Linear(model.classifier[1].in_features, 512, bias=False),
        nn.ReLU(),
        nn.BatchNorm1d(512),
        nn.Dropout(p=0.5, inplace=True),
        nn.Linear(512, num_classes, bias=False)
    )

    model.classifier[1] = layer

    model = model.to(device)
    logger.info("Model creation completed.")

    return model
```

Figure 12: The main architecture of improved model

The best hyperparameters for the improved model are as follows:

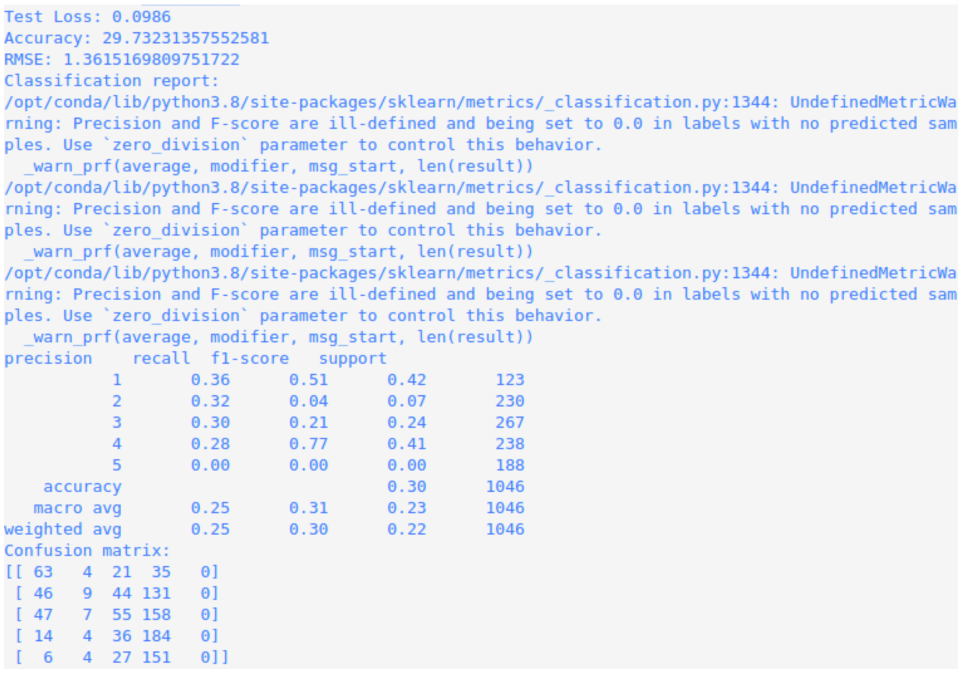
```
{'_tuning_objective_metric': '"average test loss"',
 'batch-size': '"16"',
 'epochs': '11',
 'sagemaker_container_log_level': '20',
 'sagemaker_estimator_class_name': '"PyTorch"',
 'sagemaker_estimator_module': '"sagemaker.pytorch.estimator"',
 'sagemaker_job_name': '"pytorch-training-2023-04-10-21-53-19-610"',
 'sagemaker_program': '"hpo_improved.py"',
 'sagemaker_region': '"us-east-1"',
 'sagemaker_submit_directory': '"s3://sagemaker-us-east-1-598348623909/pytorch-training-2023-04-1
0-21-53-19-610/source/sourcedir.tar.gz"'}
```

Figure 13: The best hyperparameters for benchmark model

Results

In this section, we analyze the performances of our benchmark and refined model in two parts.

Model Evaluation and Validation



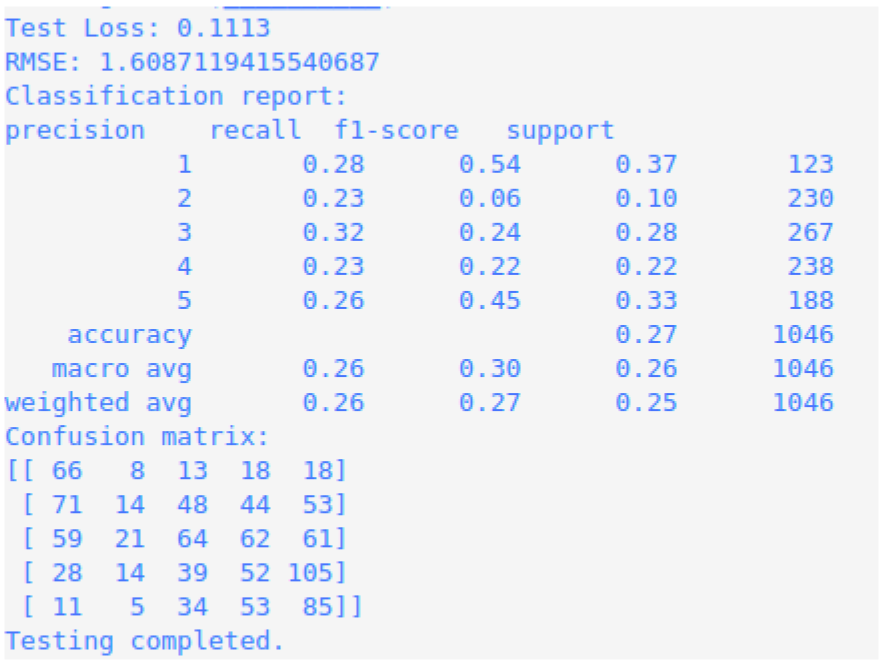
```
Test Loss: 0.0986
Accuracy: 29.73231357552581
RMSE: 1.3615169809751722
Classification report:
/opt/conda/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWa
rning: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted sam
ples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWa
rning: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted sam
ples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWa
rning: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted sam
ples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
precision    recall  f1-score   support
           1       0.36      0.51      0.42       123
           2       0.32      0.04      0.07       230
           3       0.30      0.21      0.24       267
           4       0.28      0.77      0.41       238
           5       0.00      0.00      0.00       188
    accuracy                           0.30      1046
   macro avg       0.25      0.31      0.23      1046
weighted avg       0.25      0.30      0.22      1046
Confusion matrix:
[[ 63   4  21  35   0]
 [ 46   9  44 131   0]
 [ 47   7  55 158   0]
 [ 14   4  36 184   0]
 [  6   4  27 151   0]]
```

Figure 14: The results of the benchmark model

Table 5: Result metrics for benchmark and improved models

	Accuracy	RMSE	Macro F1 score	Weighted F1 score
Benchmark model	0.27	1.61	0.26	0.25
Improved model	0.30	1.36	0.23	0.22

Justification



```
Test Loss: 0.1113
RMSE: 1.6087119415540687
Classification report:
precision    recall  f1-score   support
           1       0.28      0.54      0.37       123
           2       0.23      0.06      0.10       230
           3       0.32      0.24      0.28       267
           4       0.23      0.22      0.22       238
           5       0.26      0.45      0.33       188
    accuracy                           0.27      1046
   macro avg       0.26      0.30      0.26      1046
weighted avg       0.26      0.27      0.25      1046
Confusion matrix:
[[ 66   8  13  18  18]
 [ 71  14  48  44  53]
 [ 59  21  64  62  61]
 [ 28  14  39  52 105]
 [ 11   5  34  53  85]]
Testing completed.
```

Figure 14: The results of the improved model

Table 6: Result accuracy metric for benchmark and improved models per classes

	1	2	3	4	5
Benchmark model	0.54	0.06	0.24	0.22	0.45
Improved model	0.51	0.04	0.21	0.77	0

Table 7: Result F1 score metric for benchmark and improved models per classes

	1	2	3	4	5
Benchmark model	0.37	0.10	0.28	0.22	0.33
Improved model	0.42	0.07	0.24	0.41	0

Conclusion

Throughout this work, we’ve done EDA, chosen the most suitable pre-trained model for our work, and done research to find new methods of improvement.

We made best use of the given dataset with data augmentation where we created new, possible versions of pre-existing data, and refined the model through using the Dropout function. Since our model did improve upon the benchmark version on accuracy and RMSE metrics, our implementation adequately meets the goals set for this project. However the weakness of data imbalance will remain to cure for future works.