Object counting using the Amazon Bin Image Dataset (ABID)

Distribution centres and warehouses require careful inventory management to avoid over- or understocking, which can negatively affect customer experience, hinder efficiency and increase costs. Amazon ships around 1.6 million packages per day, and it is therefore most desirable to have a modern, accurate system for tracking inventory. One possible component of such a system could be to monitor the number of items in each package being processed. Images of packages are captured as Amazon warehouse worker robots perform Fulfilment Centre operations and these would form the input data to be analysed for inventory management.

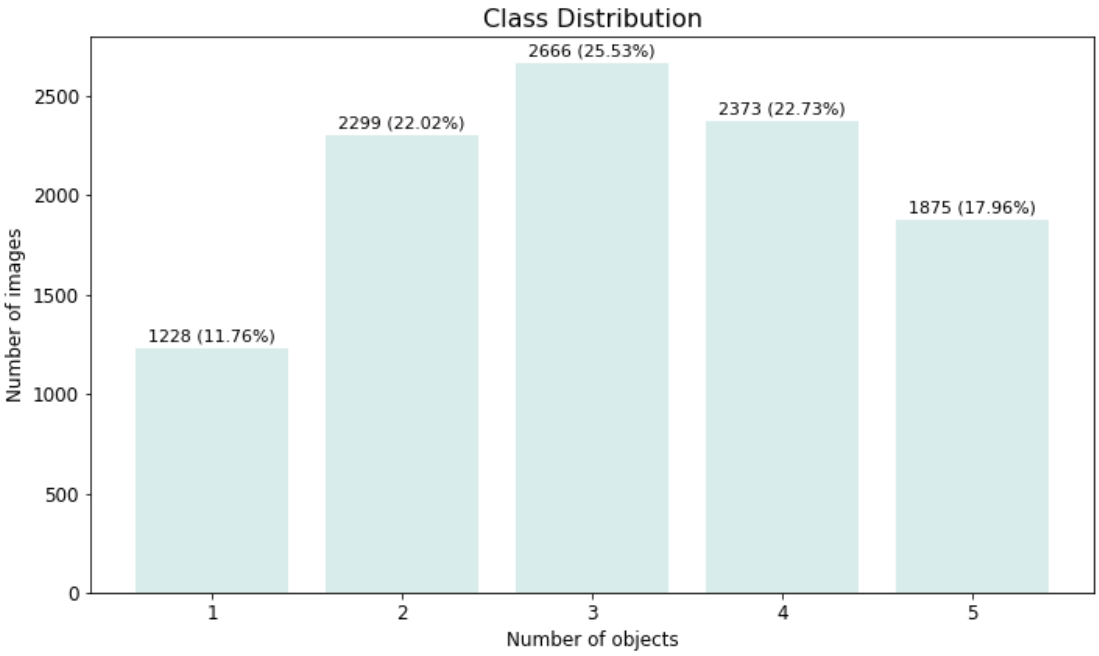
Problem statement

This project uses AWS Sagemaker to train an ML model on a subset of the ABID to count the number of objects in each bin. Although in reality we would want to optimise accuracy as much as possible in line with business priorities, in our case the focus was on building an ML pipeline that utilises best practices and could in principle be deployed to production. That is, we assume that necessary initial work regarding thorough EDA, data quality and prototyping considerations have already been carried out and we are concerned with taking a model, training it (including a hyperparameter optimisation step) on Sagemaker, and preparing and testing it for inference rather than optimising the performance of the model itself.

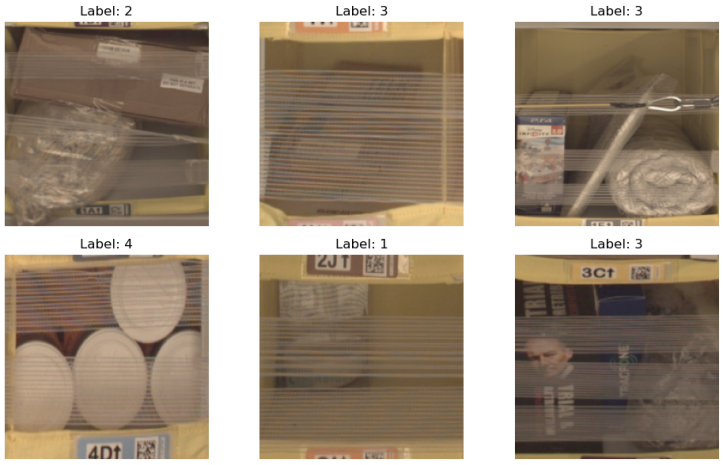
Dataset

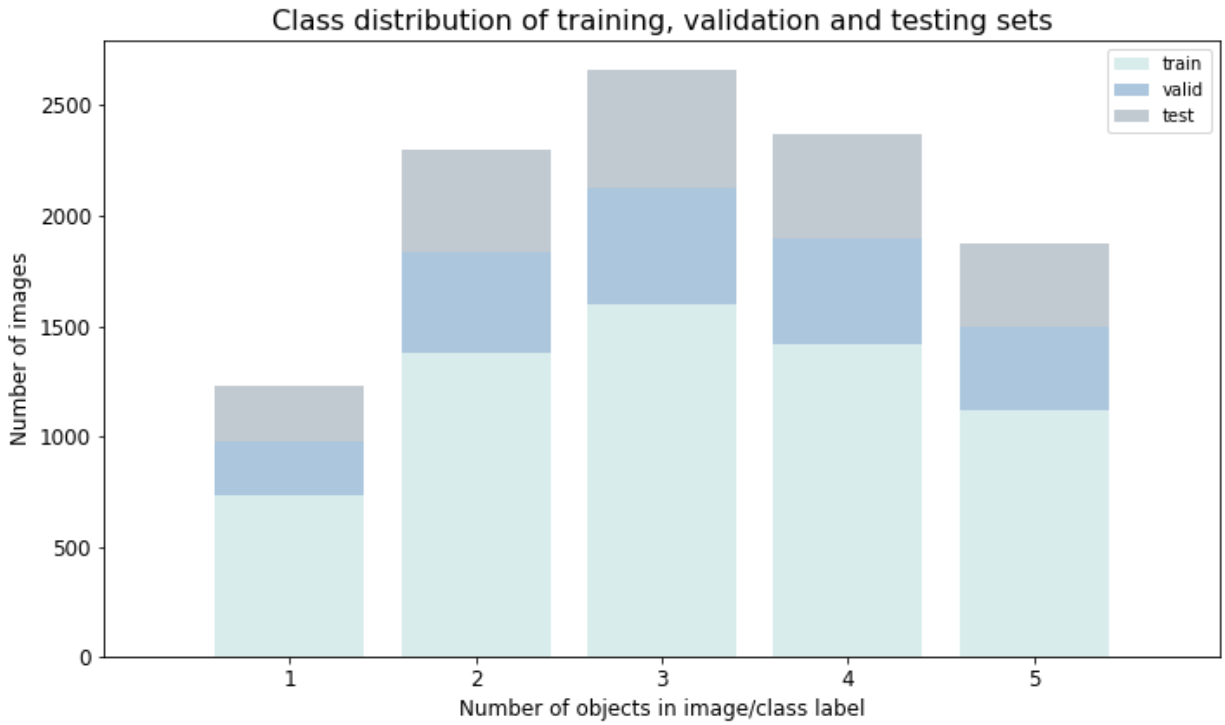
The full ABID of ~500,000 images can be found here, but we use a subset provided by Udacity; below is a visualisation of the class distribution of the latter.

The classes are imbalanced, with a relative majority of images (~26%) containing three images and a minority (~12%) containing just one image. That said, given that the most prevalent class is only a little over twice as common as the rarest, we assume that this will not affect learning significantly and therefore omit a preprocessing step wherein we balance the classes somehow, e.g. via over- or undersampling. Once the pipeline is built and assessed, we could take a look at how well the model is classifying each class and add this in later to optimise performance if necessary.



As part of the initial data preparation we will split the data into sets according to a 60:20:20 training:validation:testing split. This will leave about 700 images in the training set of the least-represented class, which should be enough to get okay results given that we are using transfer learning. The bar chart on the next page visualises the class distribution after the three-way split, and below we have some sample images from the training set. We can see from the images that due to contrast, resolution and occlusion, in at least some of the cases it is not entirely straightforward for a human to see exactly how many objects there are; we thus anticipate that this could potentially be a difficult task to obtain good accuracy on.





Preprocessing

It is general good practice in machine learning to normalise the training data so that the input data are on the same scale as one another. The network is trying to learn how important particular pixels are, and it does so via gradient calculations; outlier pixels in the image can skew the gradients causing issues with learning and convergence.

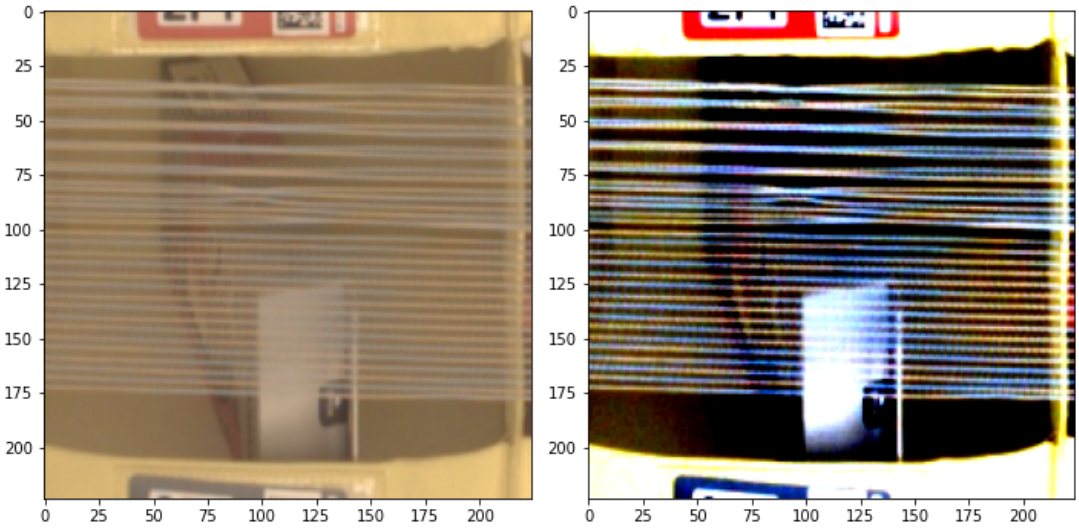
Here we used standard scaling to normalise the image set, that is, we calculated the mean and standard deviation for each channel and then normalised such that it has mean = 0 and standard deviation = 1, i.e. for each channel we calculated

$$output = \frac{input - mean}{std}$$

In the equation above, the mean and standard deviation are for the individual channels. For details of the code see normalize_local.py. The values outputted from there were then taken as arguments for a normalisation transform in the training and hyperparameter scripts (see train.py and hpo.py).

Additional preprocessing was also applied on some of the training runs. This can be used to create additional data but in our case we simply use it to add some noise to the dataset to try to mitigate overfitting. Below is an example of an image pre- and post-standardisation along with some examples of data augmentation utilised in the project.

```
        transforms.RandomResizedCrop((224, 224)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomGrayscale(p=0.15),
        transforms.RandomAdjustSharpness(2, p=0.2),
        transforms.RandomAutocontrast(p=0.1),
        transforms.RandomApply([
            transforms.ColorJitter(0.5, 0.5, 0.5, 0.5)
        ], p=0.15),
        transforms.RandomApply([
            transforms.RandomChoice([
                transforms.RandomRotation((90, 90)),
                transforms.RandomRotation((-90, -90)),
            ])
        ], p=0.1),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.4758, 0.4253, 0.3760],
                             std=[0.1616, 0.1491, 0.1424])
])
```



Evaluation metric, performance, class imbalance and benchmark

The ABID Challenge was proposed in order to attract researchers in academia and industry to apply themselves to various computer vision tasks based on the dataset. One of the challenges outlined was to count the number of objects in each bin, i.e. the same task as here, and they provided an initial accuracy score of ~55% on a subset of the data using bins with 0–5 objects in them. The specific ML task to be carried out here is multi-class classification, and there are various metrics available with which to judge model performance. One metric given in the ABID challenge to provide a benchmark was accuracy, simply defined as follows:

$$Accuracy = \frac{\#\ Correct\ predictions}{\#\ predictions}$$

This is straightforward to interpret: if we make 100 predictions and 80 of these are correct, our accuracy is 0.8. While we may be tempted to take the ABID challenge's 55% as a performance benchmark, we note that t may not be reasonable to do this: for example, if there was a large bias in their dataset towards class 0 and their model predicted most of that class correctly, then the comparison is not a fair one since that will be where most of their score comes from. Given we don't have a direct comparison, we instead take our first simple model as a benchmark and see if we can make improvements from there.

In general, the choice of metric to evaluate model performance is specific to a given use case and dataset; in particular, accuracy may be a good metric for a balanced dataset but may well not be the best for an unbalanced dataset, and certainly not if the imbalance is very severe. By using accuracy here we make two assumptions: (1) that the class imbalance noted earlier will not have any effect on performance and (2) that we value all misclassifications equally, i.e. a misclassification of class 1 as class 5 is just as unfavourable as one of class 1 as class 2. In reality, both of these should be checked (and point (2) is particularly unlikely to hold in this case), the first by checking the per class accuracies and the second by the business context. We note here that their accuracy varies somewhat from class to class, so it may be that a better score could be obtained if the class imbalance mentioned earlier was indeed mitigated ahead of training.

Model iteration and improvement

Model training

We chose a pretrained ResNet18 CNN to fine-tune for this project as it is known to give good results on CV problems and doesn't have as many parameters as some of the other larger models, meaning that training time would be relatively short.

The project consists of the following steps.

1. Download and arrange data.
2. Train a model on Sagemaker and conduct hyperparameter optimisation.
3. Train model on the full dataset with best hyperparameters, including debugging and profiling on Sagemaker.
4. Deploy trained model to an endpoint and confirm that we can make an inference successfully.
5. Use the model to demonstrate a batch transform on Sagemaker.

Note that there is iteration between steps 2 and 3 to refine and improve the algorithm, as outlined below.

Initial Model (M0)
The initial model was very simple, with a single fully connected layer of 128 neurons added to the pre-trained ResNet18 layers. In this case we added no extra transformations or preprocessing before training and for speed tuned the hyperparameters (batch size, learning rate, α, and β_1) using 0.3 of the training set.

First refinement (M1)
For the first refinement, we maintained the same model head architecture but added a preprocessing step to normalise the images before training. Because low GPU usage was flagged by the debugger on the first trial (and in fact for each refinement thereafter), the batch size wasn't tuned and instead increased to 128, then 256, 512 for the third iteration. As overtraining was also flagged, we tuned weight decay here and in subsequent runs, and also incorporated early stopping into the training code, starting with a patience of 8 on this run. Finally, the proportion of the training set that the tuner was fit on was increased from 0.3 to 0.5.

Second refinement (M2)
Here we kept all of the above refinements but also added another extra preprocessing step by adding some random augmentations to the training images and also decreased the early stopping patience to five epochs.

Third refinement
I was surprised that the accuracy had actually decreased when the extra preprocessing had been added in the last step and then realised that the mean and standard deviation would have changed and so they should have been recalculated for the transformed dataset. Unfortunately at this point I ran out of time.

A summary of the optimised parameters for each model are given below.

Model	Batch size	β_1	α	Regulariz.	Early stopping (patience)	Epochs trained	Train proportion
M0	128	0.99	0.0010	-	-	5	0.3
M1	256	0.99	0.0011	0.002	Yes (8)	35	0.5
M2	512	0.97	0.0017	0.036	Yes (5)	15	0.5

Results

Overall accuracy and per-class accuracies for each iteration of model training are given in the following table. Values for per class accuracies were derived from logging values post-training; see, e.g., the screenshot below for M2.

```
Epoch 16, Phase valid
Images [256/2090 (12%)] Phase: valid Loss: 2.13 Accuracy: 11/256 (4.30%) Time: Tue
Sep 12 20:23:45 2023
Loss not improved for 5 epochs
Loss not improved for 5 epochs: early stopping.
Loading model before testing.
Testing model on whole testing dataset
Testing Accuracy: 29.842180774748922%; Testing Loss: 1.4820582209335795
Classification report:
              precision    recall  f1-score   support

           1       0.70      0.06      0.10       247
           2       0.30      0.64      0.41       460
           3       0.29      0.32      0.30       534
           4       0.29      0.30      0.29       475
           5       0.38      0.01      0.03       375

    accuracy                           0.30      2091
   macro avg       0.39      0.26      0.23      2091
weighted avg       0.36      0.30      0.25      2091
Confusion matrix:
[[ 14 189  36   8   0]
 [  5 295 109  51   0]
 [  0 241 169 122   2]
 [  1 180 147 141   6]
 [  0  88 117 165   5]]
Saving model...
Model saved at /opt/ml/model
```

Model	Accuracy	1	2	3	4	5
M0	0.280	0.358	0.598	0.049	0.413	0.005
M1	0.305	0.285	0.265	0.354	0.297	0.328
M2	0.298	0.057	0.641	0.316	0.297	0.013

It's interesting to note that the accuracy of the simple baseline model, M0, is relatively near to the more sophisticated versions. This could be due to numerous things, for example, we may simply not have enough data to train with, or the preprocessing transformations added for M1 and M2 may not be the optimum ones to encourage the network to learn. It may also be that the head architecture of the network is too simple to allow learning of the pertinent features of the bins. What is certainly the case is that for M2 I made a mistake in standardising using the mean and standard deviation from the untransformed training set, so I'd imagine that is what caused the drop off in accuracy in that case.

Finally, note that in the case of M1, where the data was preprocessed simply but correctly using standardisation, the variation in per-class accuracies is not present as in the other two cases (respective standard deviations for M0, M1 and M2 are 0.23, 0.03 and 0.22), which may imply that the class imbalance is not the main obstacle to obtaining high accuracy on this dataset.

We do see improvement via iteration, but it is modest at only 2–3% and it may be that we need more data to improve the score further.

Improvements, given time, would include:

- Re-running M2 using re-calculated mean and std of transformed dataset for standardisation.
- Tuning hyperparameters on a larger portion of the training set, and in series rather than some runs being in parallel.
- Adding another layer to the head architecture to see how that affects the accuracy.
- Analysing the transformations applied to gain some intuition about which should be kept.

Conclusion

A full solution to the problem of counting the objects in bins at distribution centres might work as follows. Once all items have been added to a package, images are fed into the trained ML model for inference. The output is then fed back into the inventory database, which is integrated with the rest of the management system. This project demonstrates a solution to the central part of that problem, i.e. a working ML pipeline, from data ingestion through model training, hyperparameter optimisation, debugging and profiling, to batch inference. For real-time inference, the model could be easily deployed to an endpoint invoked by an AWS Lambda function that outputs predictions and passes them to the rest of the management system; the Lambda function could be configured for concurrency to deal with peak demand, and batch processing could be used for non-time-sensitive inference.

In this case we only managed a modest 2–3% increase over the initial baseline score; given more time, some of the steps mentioned previously could increase it further. Although there is plenty still to do in terms of model performance, the main structure of the project could be easily adapted to another use case with a different dataset, and given the focus was on implementing a working ML pipeline rather than accuracy, this solution satisfies the aims set out at the beginning of the project.