Object Detection in an Urban Environment Project

Project overview

The objective of this project was to utilize a convolutional neural network (CNN) on data from Waymo to detect and classify objects, namely cyclists, pedestrians, and vehicles. This task is important within the context of self-driving car systems because said systems need to be able to construct a model of the surrounding environment (i.e., perceive) so as to be able to move about safely and effectively in the world.

Set up

The procedure taken in this project were those given in the *Project Instructions (Workspace)* sub-section of the final ***Object Detection in an Urban Environment*** section.

Dataset

Dataset analysis

The *Exploratory Data Analysis* notebook was used to write code (Figure 1) to display the images. These images showed objects during the day and at night, of different sizes, sometimes close together, and occasionally blurred/hazy (Figure 2). These observations were used to inform the data augmentation strategy.

```
def display_images(batch):
    batch_size = 10
    figure_cols = 2
    figure_rows = (batch_size + 1) // figure_cols
    f, ax = plt.subplots(figure_rows, figure_cols, figsize=(figure_rows*4, figure_cols*24))
    colormap = {1: [1, 0, 0], 2: [0, 1, 0], 4: [0, 0, 1]}

    for idx, image_data in zip(range(batch_size), batch.shuffle(buffer_size = 100).take(batch_size)):
        x = idx % figure_rows
        y = idx % figure_cols

        img = image_data['image'].numpy()
        ax[x,y].imshow(img)

        bboxes = image_data['groundtruth_boxes'].numpy().tolist()
        classes = image_data['groundtruth_classes'].numpy().tolist()
        width, height = image_data['original_image_spatial_shape'].numpy().tolist()

        for cl, bb in zip(classes, bboxes):
            ymin, xmin, ymax, xmax = bb

            # de-normalise
            xmin *= width
            xmax *= width
            ymin *= height
            ymax *= height

            rec = patches.Rectangle((xmin, ymin), xmax- xmin, ymax-ymin, facecolor='none', edgecolor=colormap[cl])
            ax[x, y].add_patch(rec)

        ax[x ,y].axis('off')

    plt.tight_layout()
    plt.show()
```

Figure 1: Code to display a subset of images.

Figure 2: Subset of images.

Further exploratory analysis was performed in the form of a distribution plot of the classes because, if the validation and test sets possess distributions that significantly deviate from that of the training set, then the model derived from the training data will not adequately perform on the unseen data. Figures 3 and 4 show the code and corresponding plot, which illustrates that the distribution of the three split sets are similar.

```
def display_density_plots():
    import seaborn as sns
    locations = {'train': "data/train/*.tfrecord", 'val':  "data/val/*.tfrecord", 'test': "data/test/*.tfrecord"}

    plt.figure(figsize=(10, 6))
    for split_type, location in locations.items():
        dataset = get_dataset(location)
        all_classes_list = []
        for batch in dataset.take(100):
            batch_classes = batch['groundtruth_classes'].numpy().tolist()

            all_classes_list.extend(batch_classes)

        # Draw the density plot
        sns.distplot(all_classes_list, hist = False, kde = True,
                     kde_kws = {'shade': True, 'linewidth': 3},
                     label = split_type)

    # Plot formatting
    plt.legend(prop={'size': 16}, title = 'DataSet')
    plt.title('Density Plot of Classes')
    plt.xlabel('Category')
    plt.ylabel('Density')
plt.show()
```

Figure 3: Code to display distributions of the split sets.

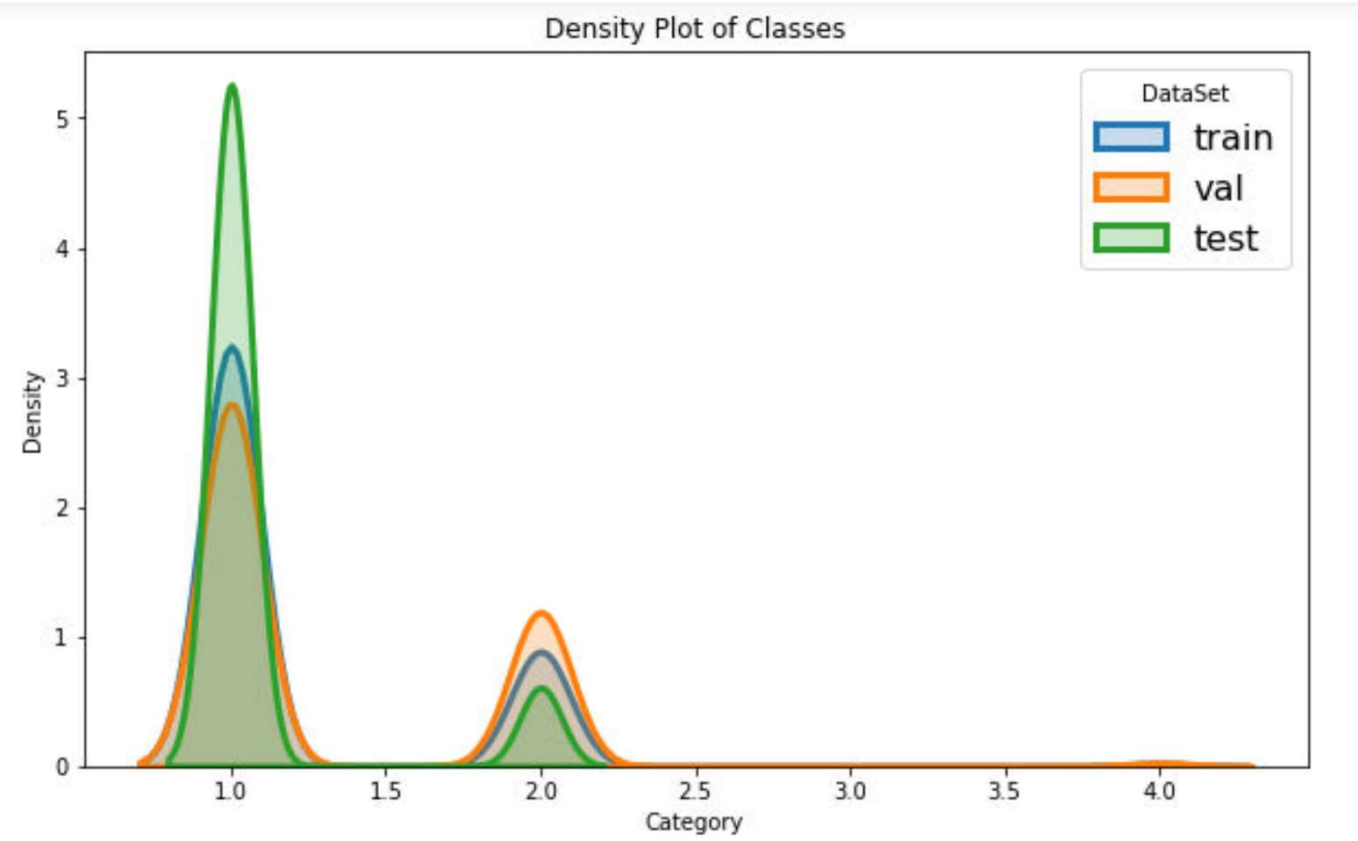

Figure 4: Distributions of the split sets.

Within the *Explore Augmentations* notebook, various options were employed, including rotation, horizontal flipping, cropping, adjusting brightness or contrast (Figure 5).

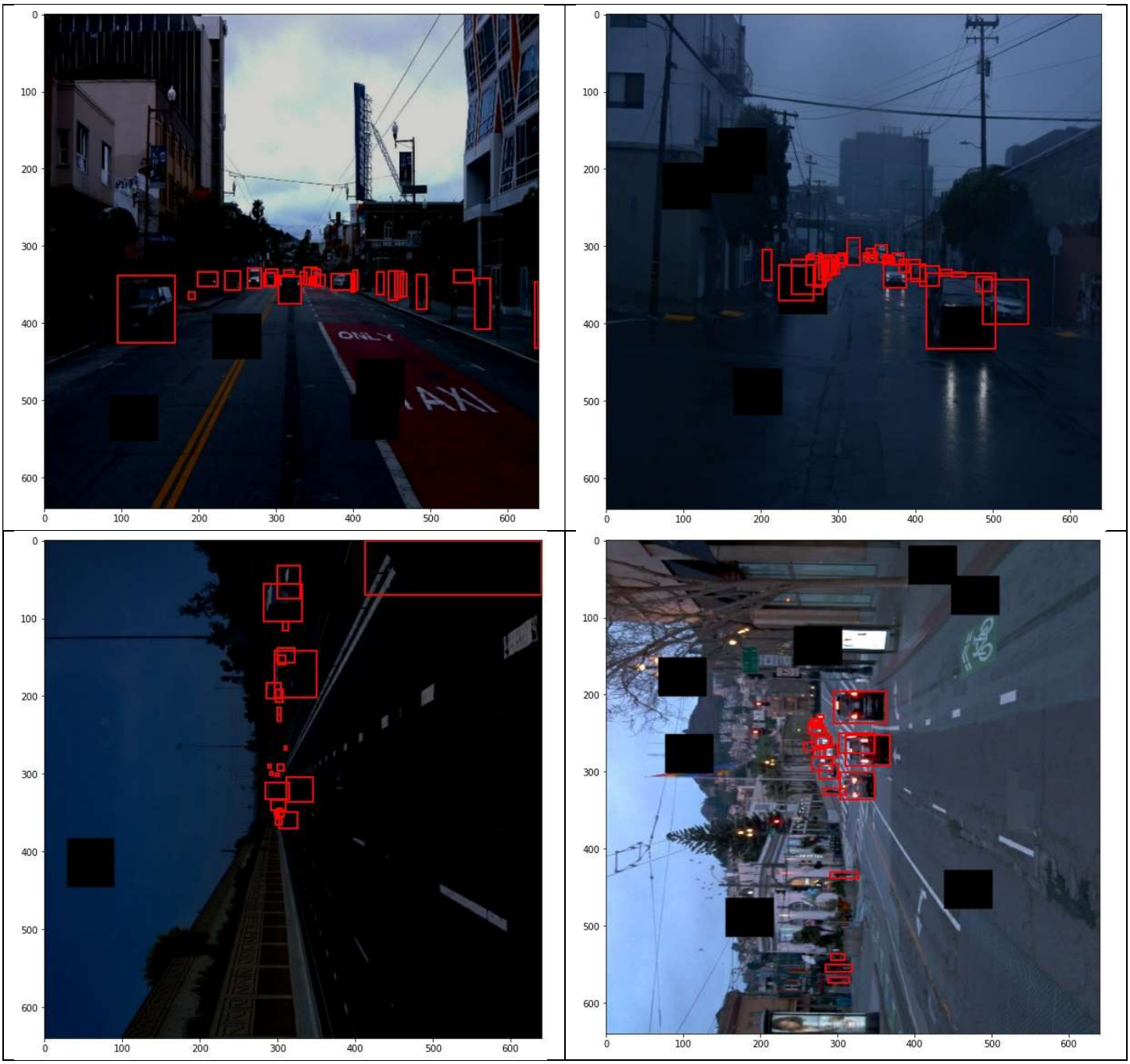

Figure 5: Results of various augmentation operations applied to some images.

Cross validation

The hold-out cross-validation approach was employed splitting the 100 records into training, validation and testing sets in the ratio of 87:10:3. This means that at no point does the training model have access to the validation and test data. Moreover, given the time constraints of this exercise, hold-out validation has the advantage, in general, of only needing to train once. Other cross validation methods exist, such as leave one out (LOO) or k-fold cross validation. However, they are generally more expensive and time-consuming to train. Given the already heavy computational load of Deep Learning algorithms they are less suited.

Training

Reference experiment

Following the instructions outlined, a reference model was trained. As shown in Figures 6a-c, the loss was around 4.5, with mean average precision and recall values very close to 0.

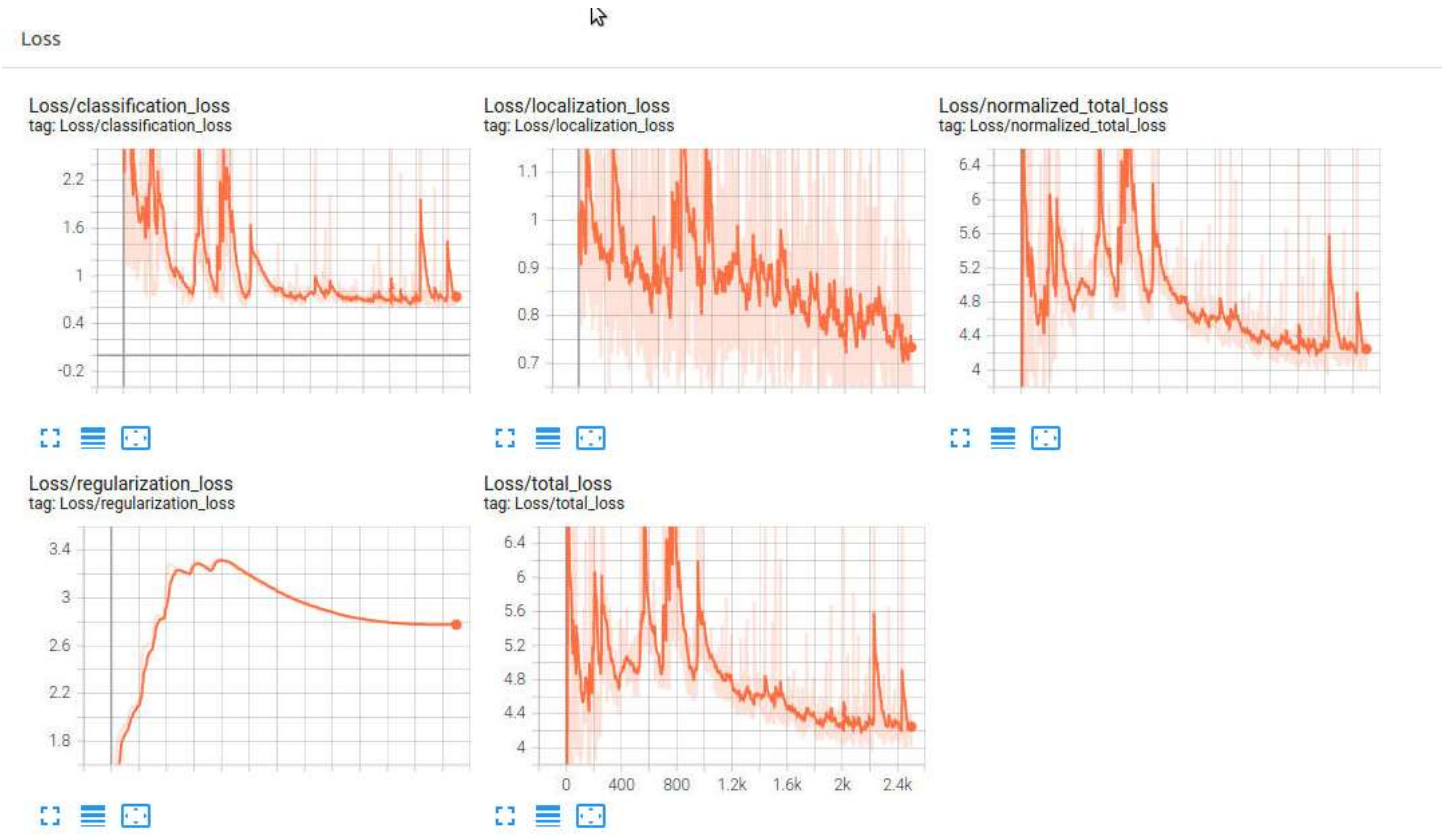

Figure 6a: Training loss of reference model

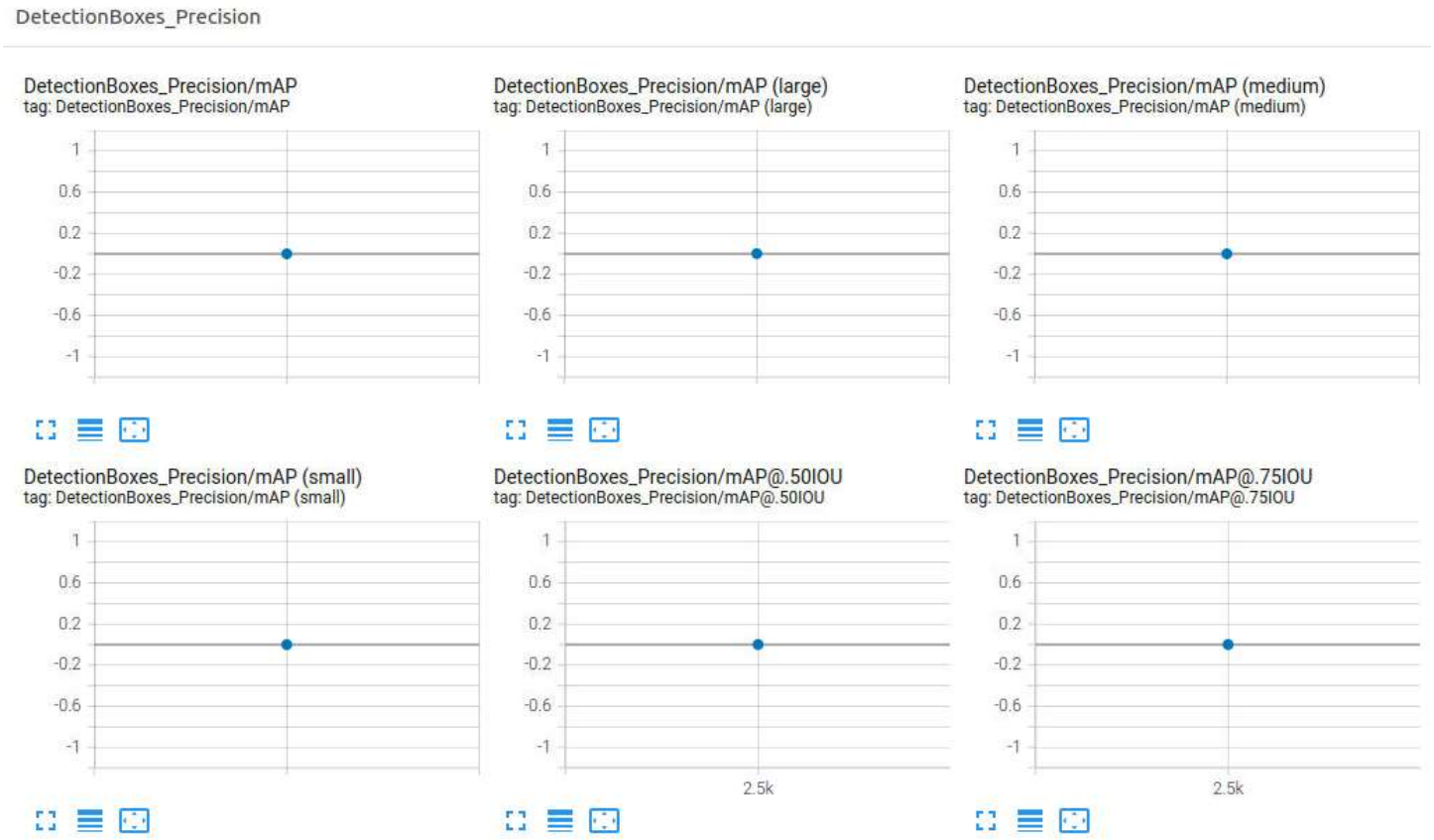

Figure 6b: Mean Average Precision of reference model

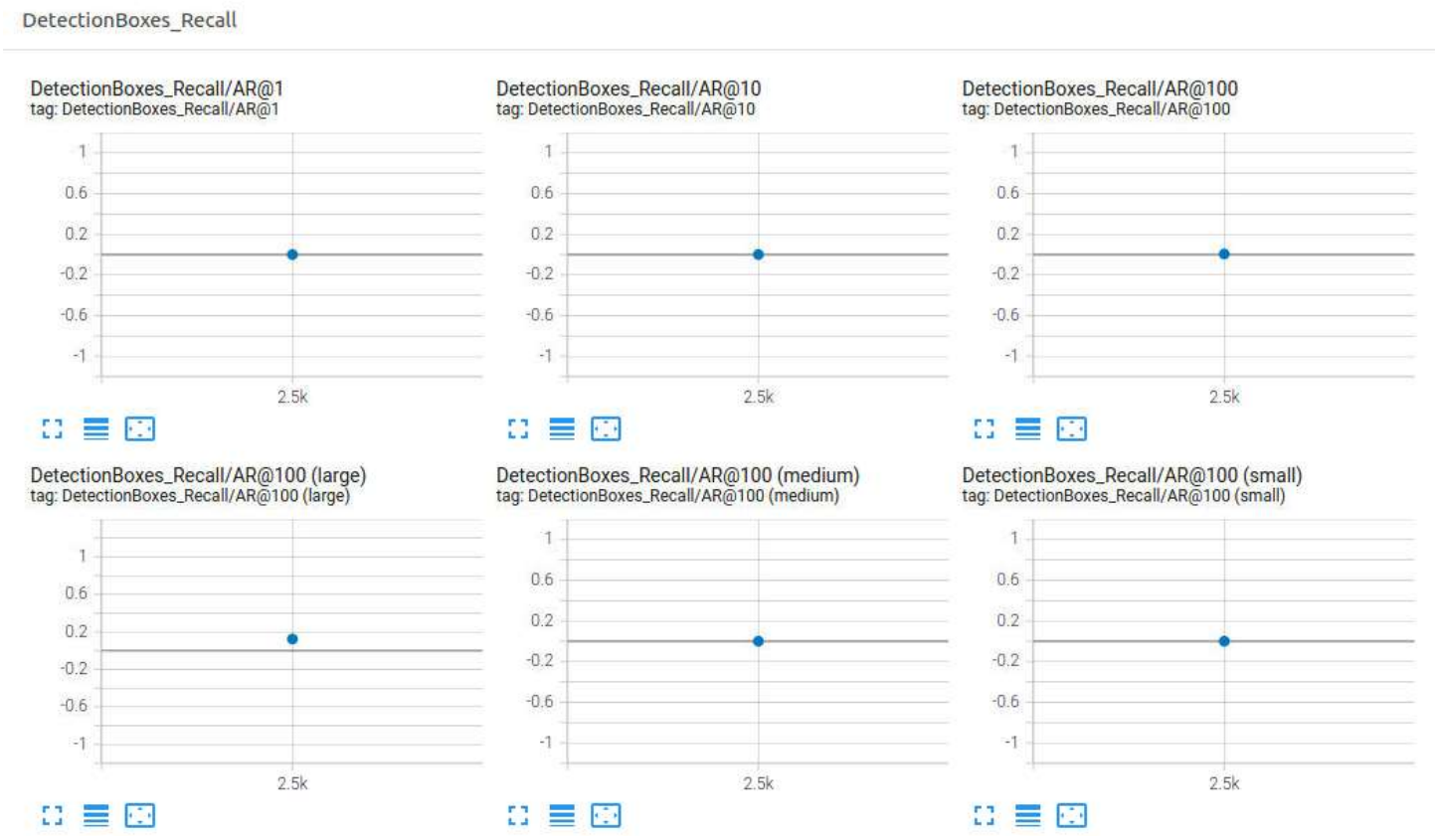

Figure 6c: Recall of reference model

Improve on the reference

The first attempt at improving the performance of the model was to employ augmentations. The following were utilized:

(i) random_horizontal_flip, given that an object could appear on either side of the frame and facing in opposite directions.
(ii) random_adjust_brightness, given that objects could be viewed at different times of the day or night.
(iii) random_crop_image, as this has been reported to improve the performance when detecting small objects.

As shown in Figures 7a-c, the loss was only slightly lower and the precision and recall about the same or slightly higher.

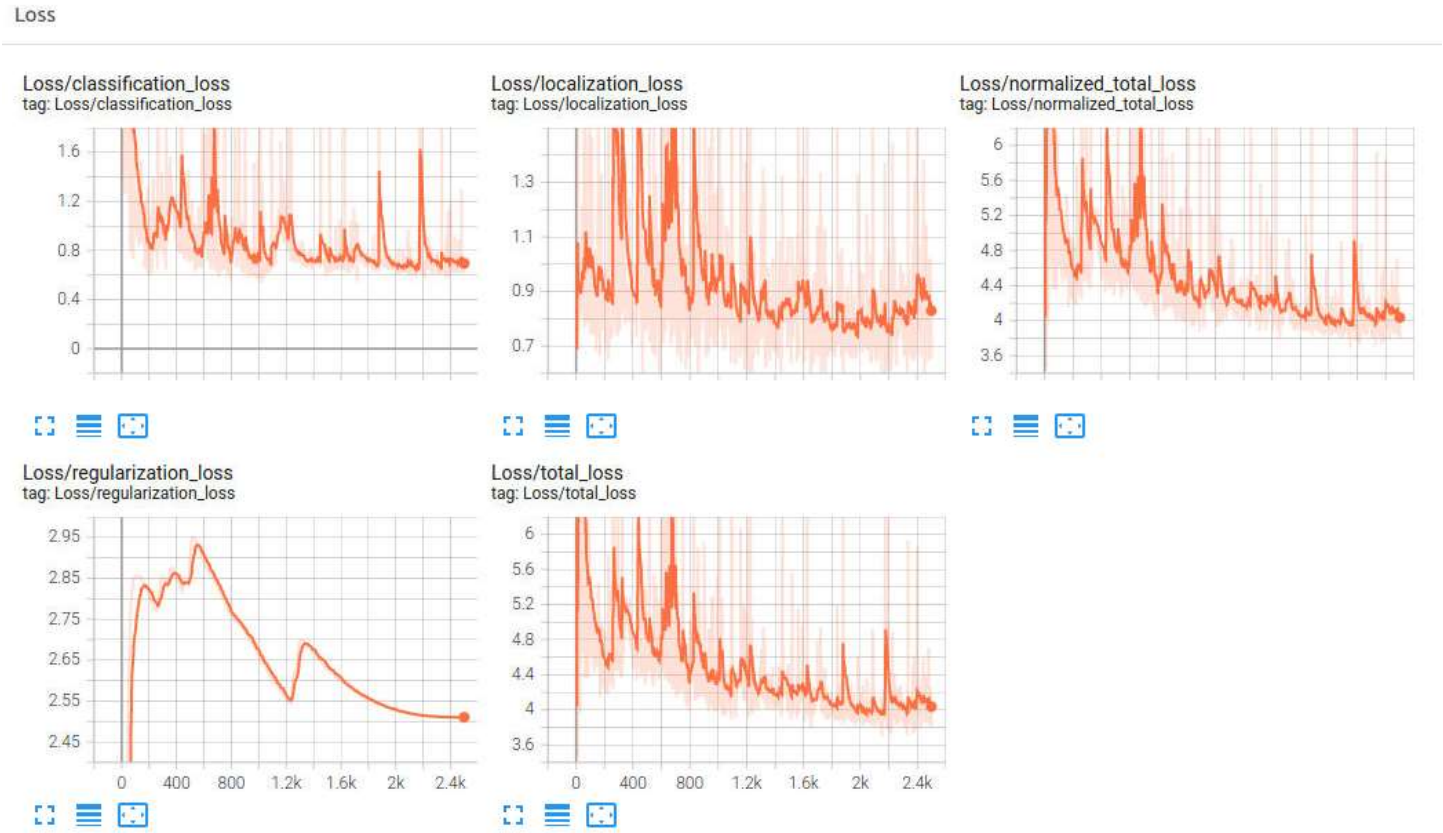

Figure 7a: Training loss of augmented model

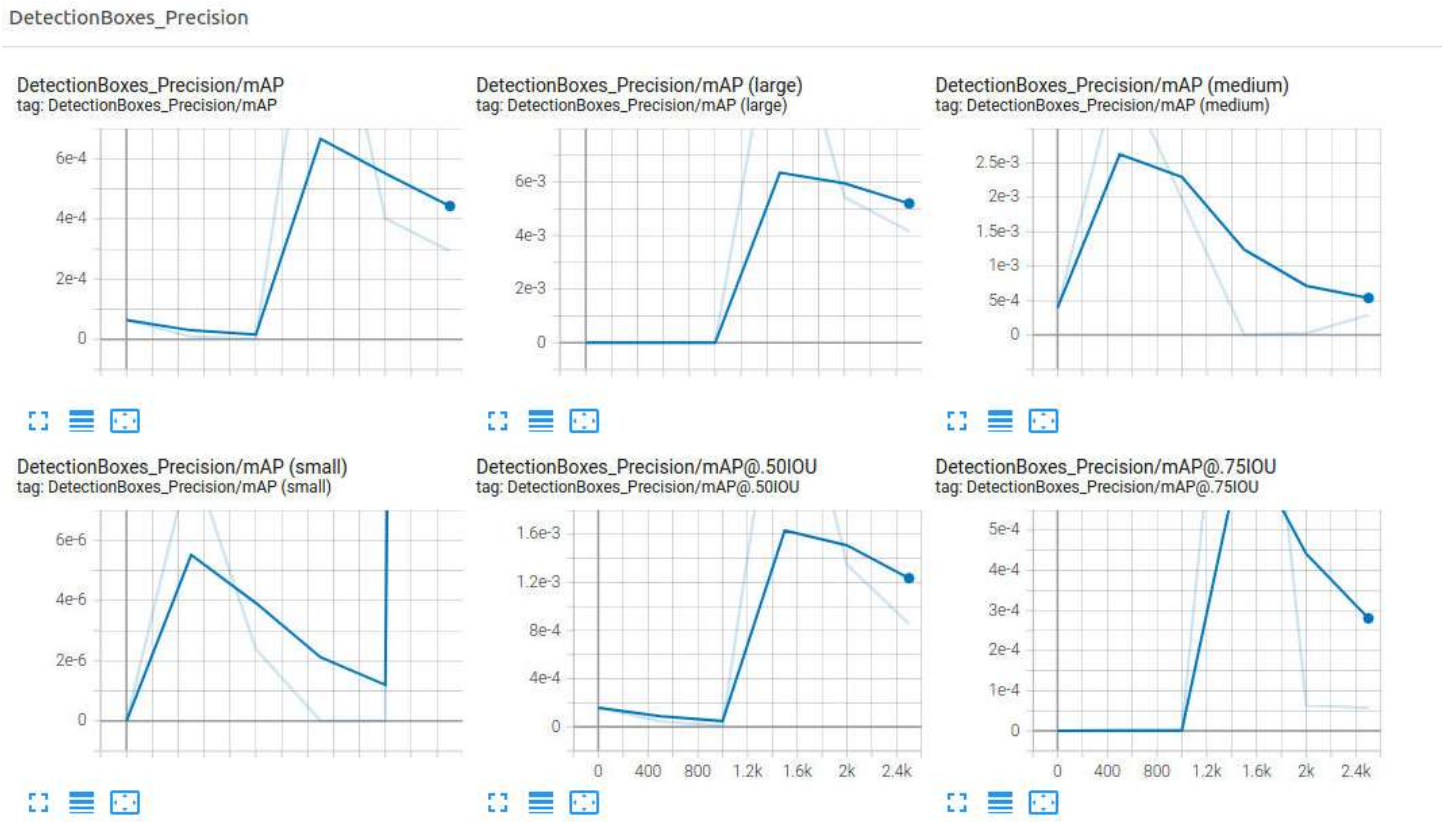

Figure 7b: Mean Average Precision of augmented model

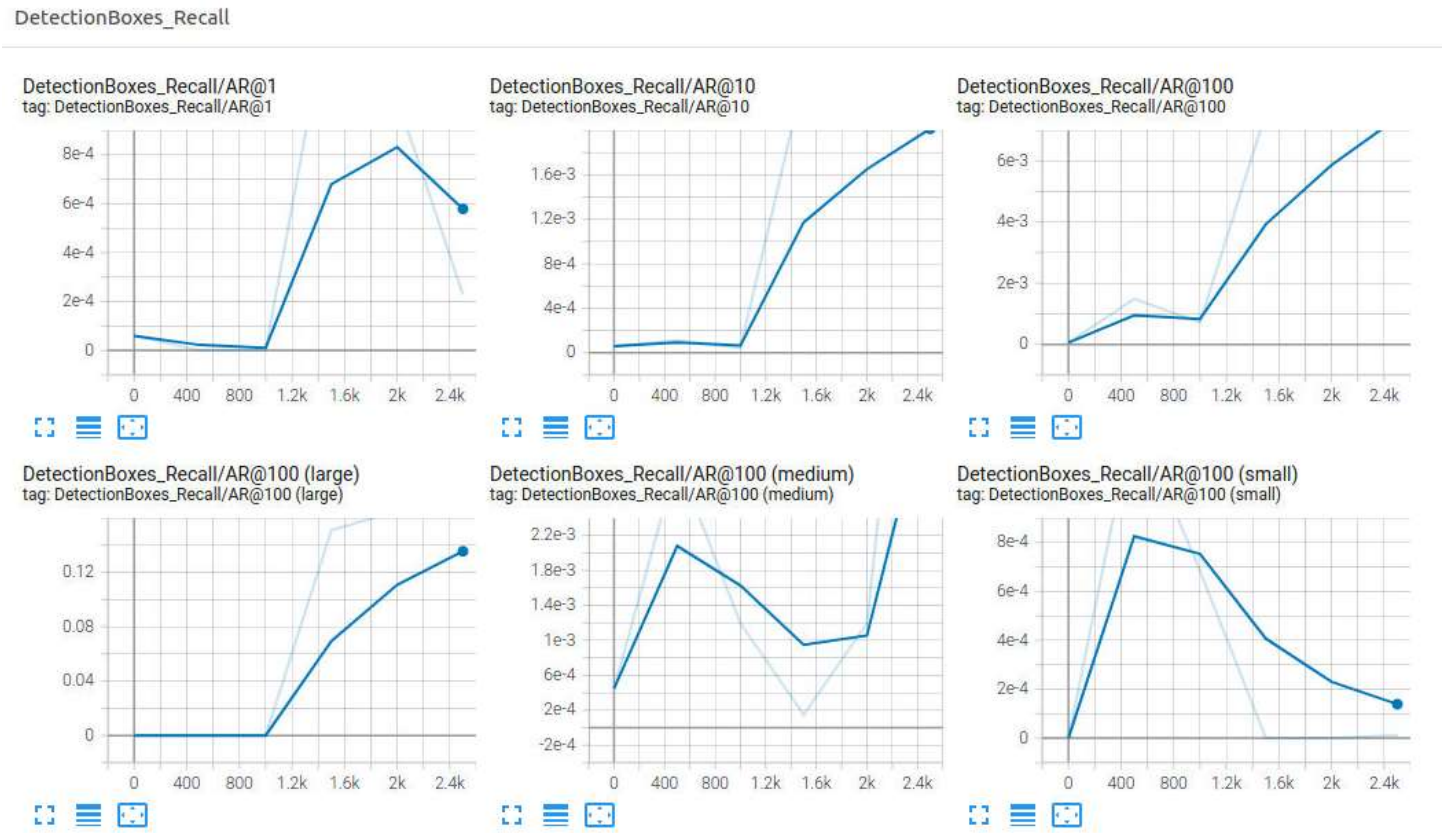

Figure 7c: Recall of reference model

The second change sought to address the high variability in the losses during training. This indicated a lot of jumping around during gradient descent. To mitigate this, the batch size was increased from 2 to 4. In this case, there was a marked improvement, with almost a 1 point drop in the loss, and an increase in some of the precision and recall metrics.

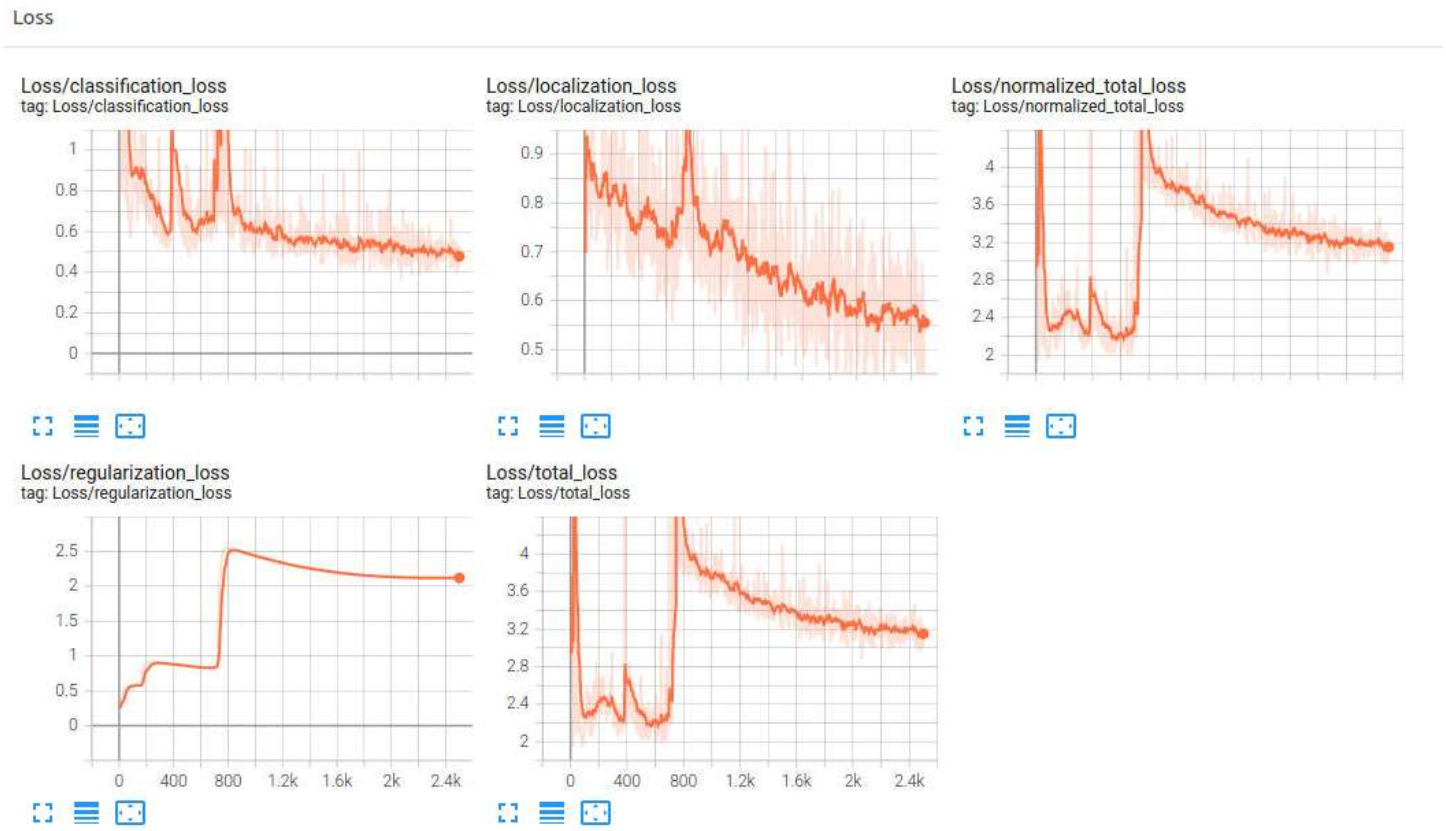

Figure 8a: Training loss of augmented model with increased batching

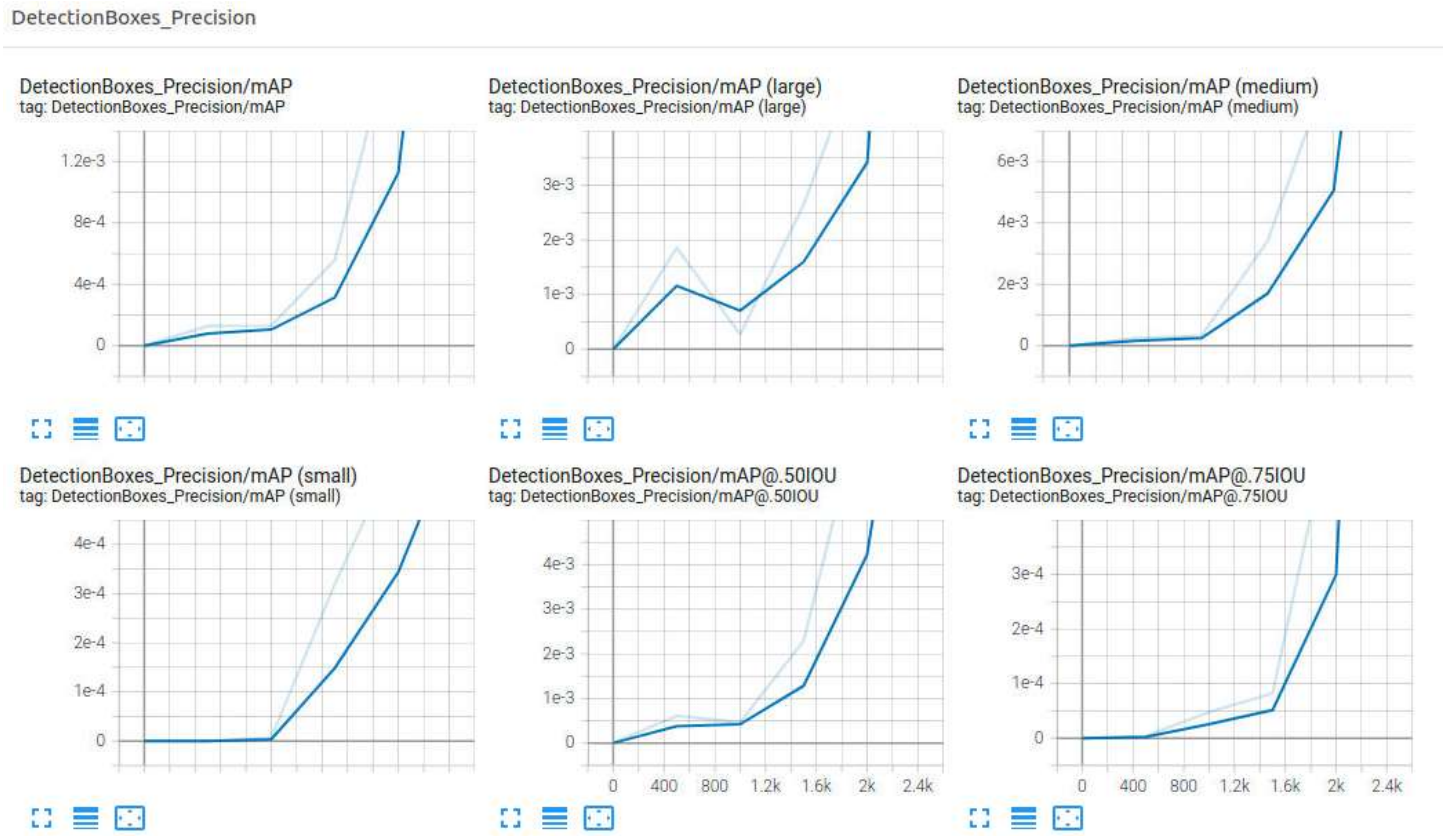

Figure 8b: Mean Average Precision of augmented model with increased batching

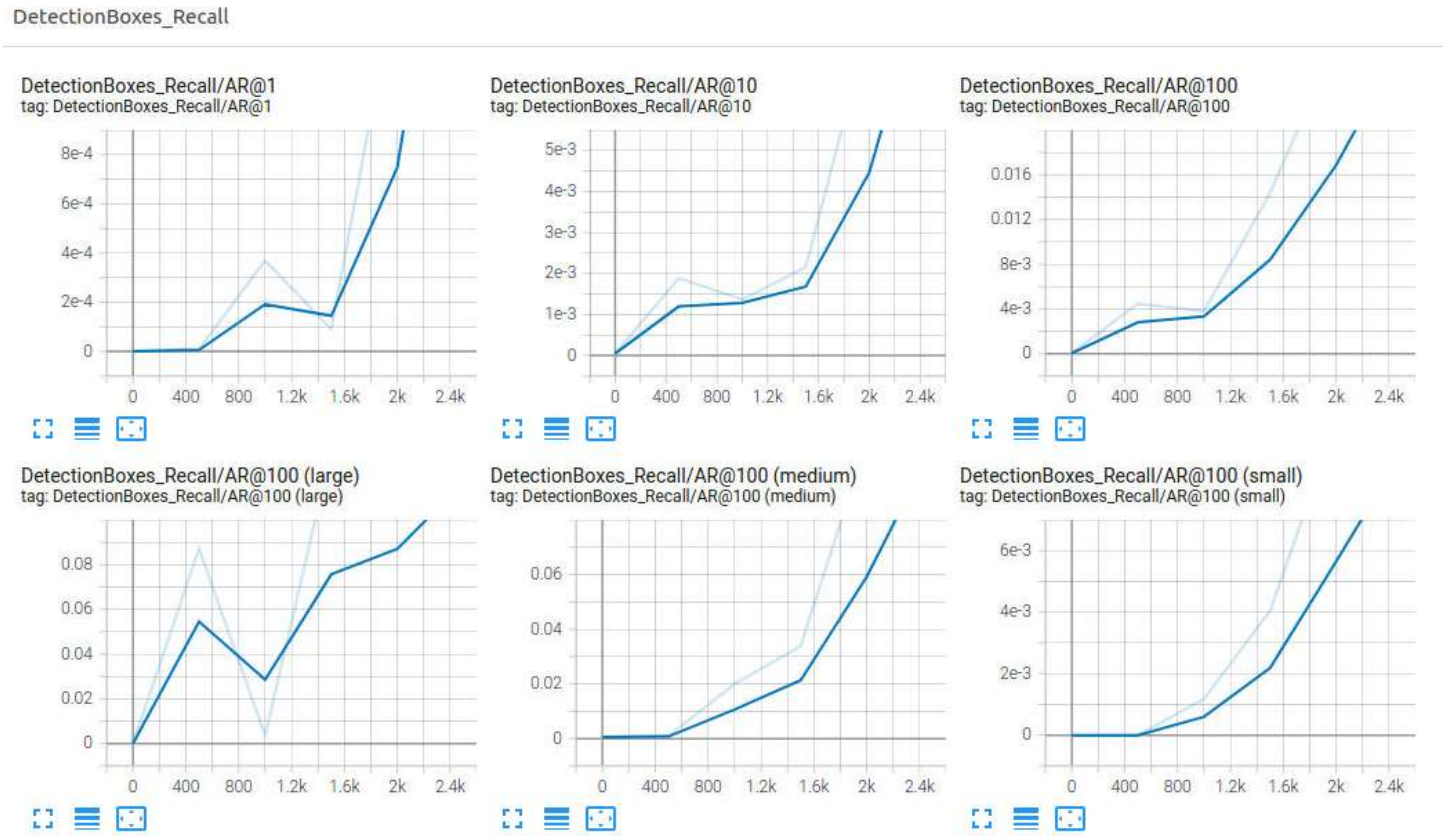

Figure 8c: Recall of augmented model with increased batching

The third change was to determine if using the popular (owing to its excellent performance on a wide range of tasks) ADAM optimizer rather than that of the momentum would yield better results. Figure 9 showed that this was not the case as the loss was reaching a plateau well above the previous model at above 5.

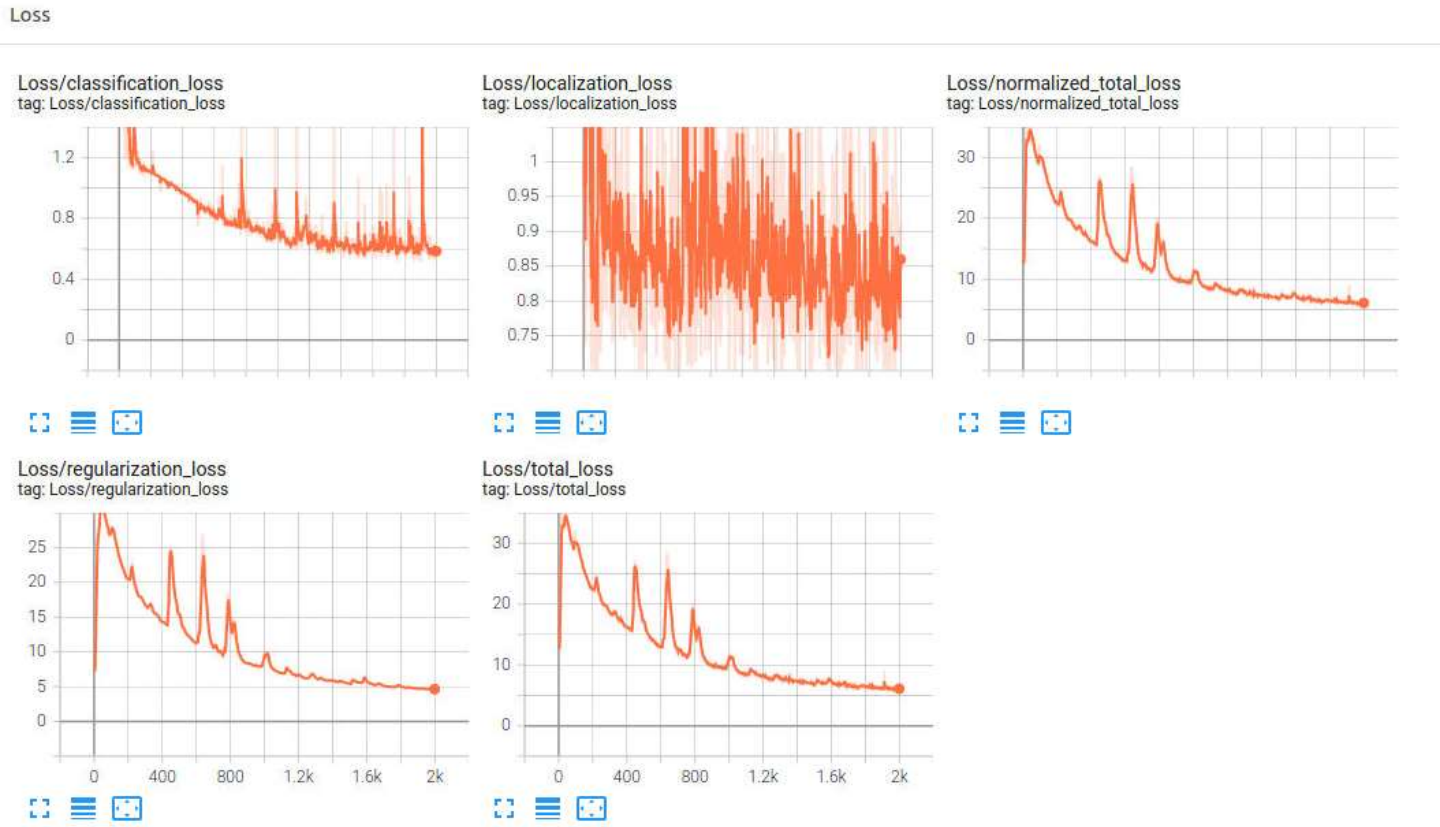

Figure 4: Training loss of an Adam-optimised augmented model with increased batching

Judging that the poorer performance of ADAM might have been due to the high initial loss and a slow drop-off, a change was made to the learning parameters in an attempt to drive the loss down more quickly. Specifically, the base learning rate was increased from 0.04 to 0.1 and the number of warmup steps was increased to 500. The idea was that the model would drive the loss function down faster over a longer initial period, hopefully enough to allow the loss to approach that of the current best performing momentum with augmentation and increased batching. This was achieved, as demonstrated in Figure 10a. However, this did not carry over to the evaluation as performance deteriorated across all measures (Figure 10b and c).

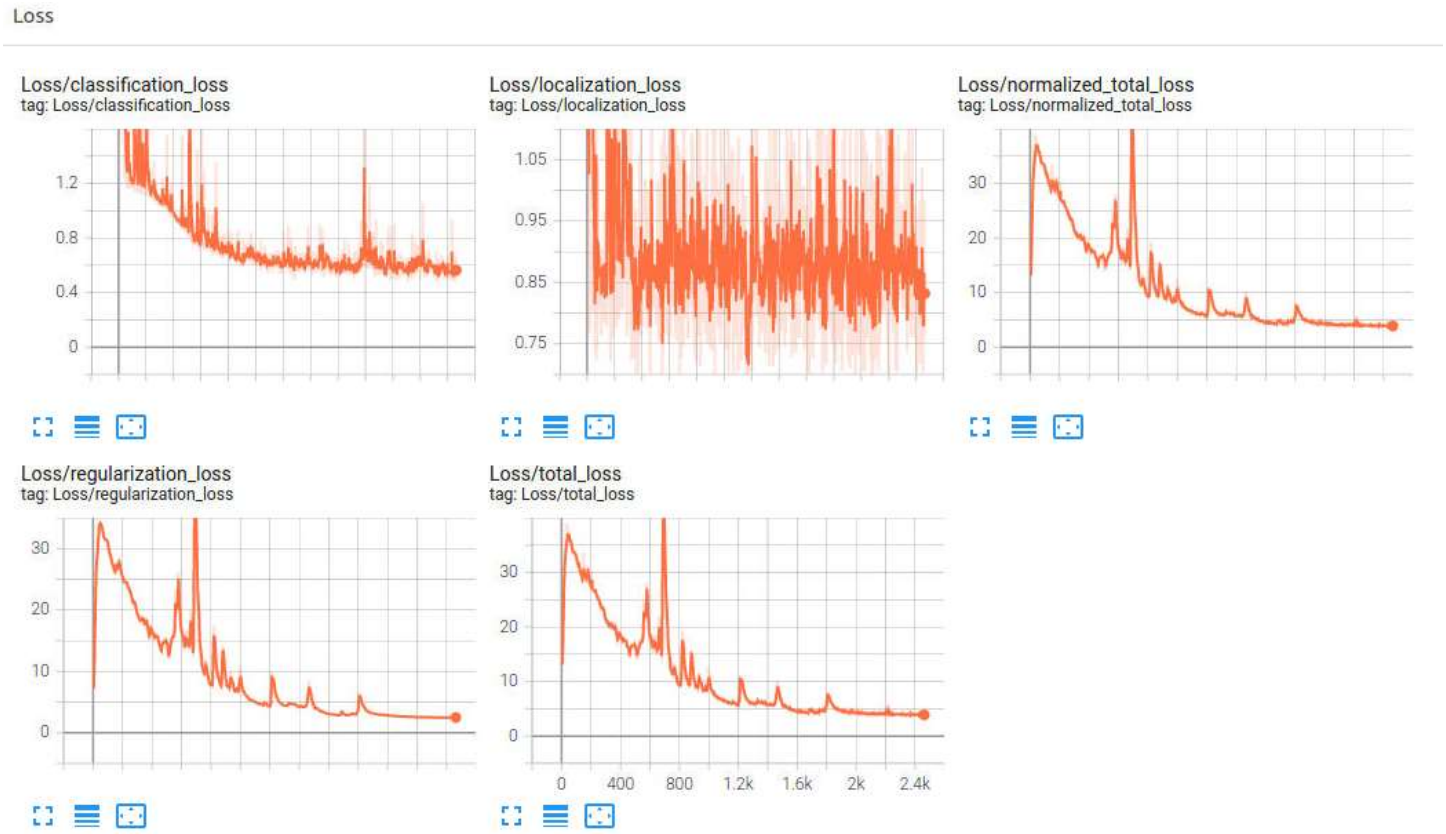

Figure 10a: Training loss of an Adam-optimised augmented model with modified learning parameters and increased batching.

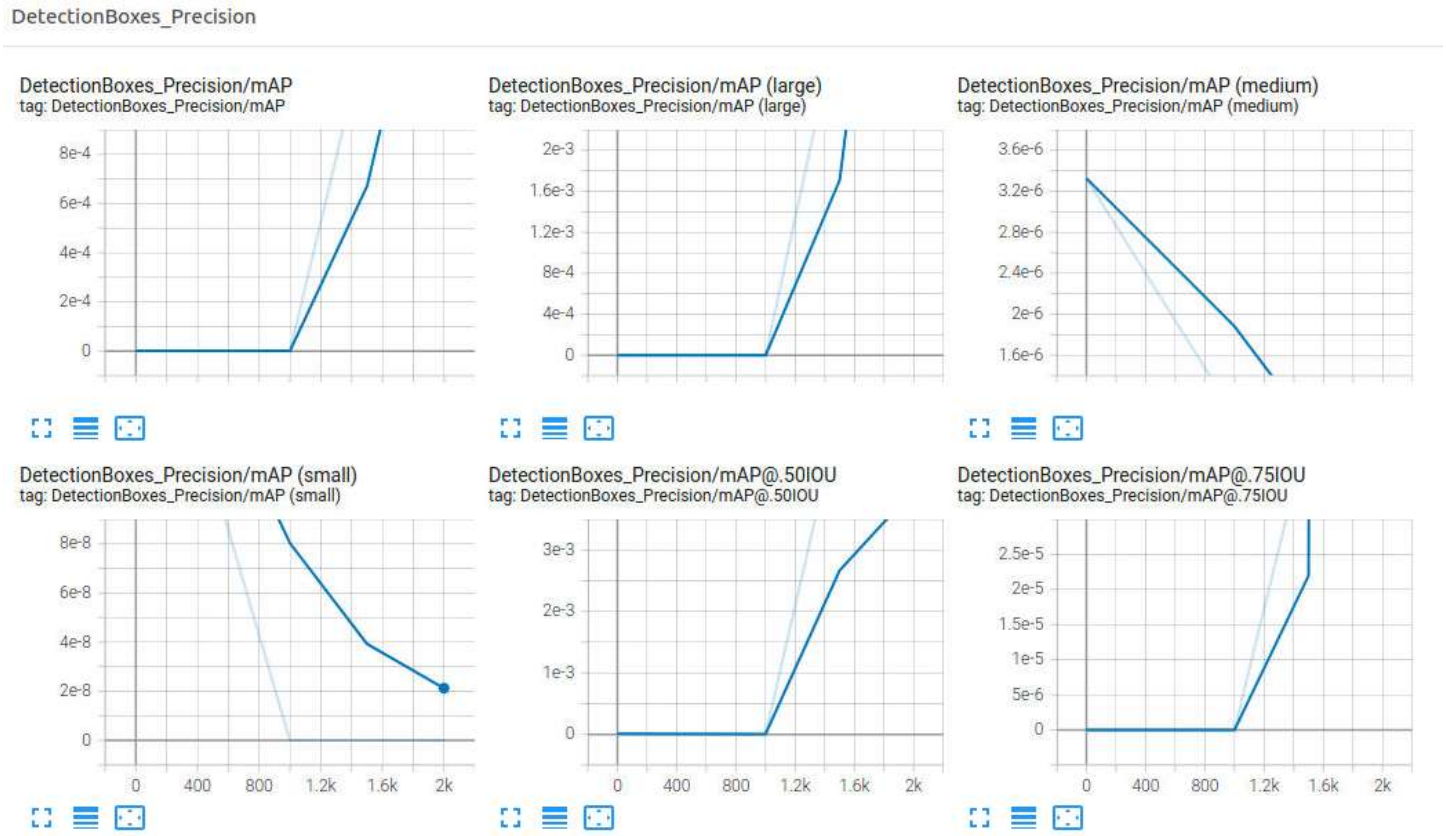

Figure 10b: Mean Average Precision of an Adam-optimised augmented model with modified learning parameters and increased batching.

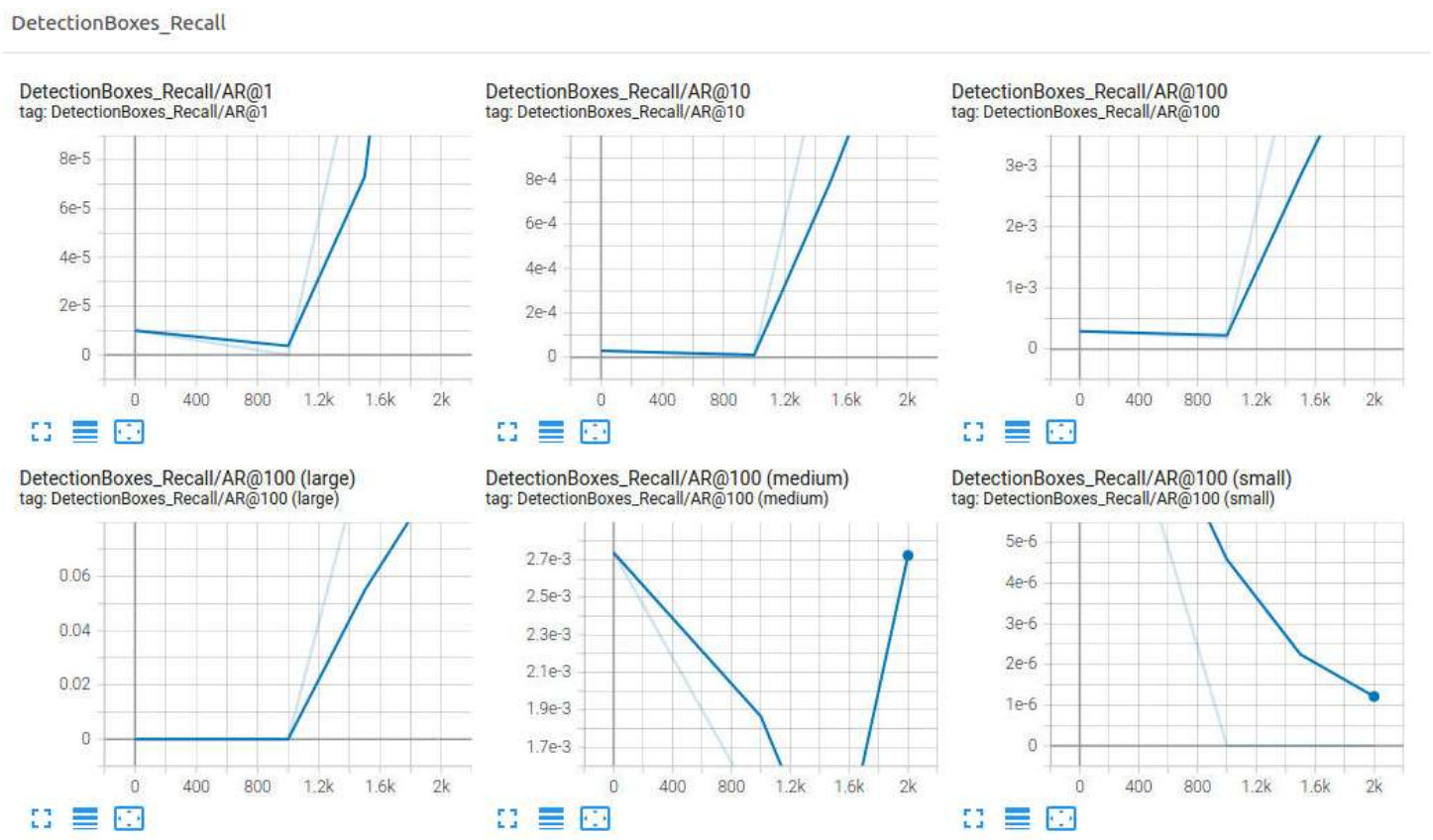

Figure c: Recall of an Adam-optimised augmented model with modified learning parameters and increased batching.