DATASET:

The dataset of interest, Blood Cell Detection Dataset has 100 images of annotated red blood cells (RBCs) and white blood cells (WBCs) as seen under a light microscope. Each image has a size of 256x256 pixels, with 3 RGB channels and the coordinates of the bounding box around each blood cell is given in the 'annotations.csv' file.

Upon further exploration of the data, we note that there are 4 duplicate annotations in 4 different images (image-1.png, image-100.png, image-104.png, image-114.png). Upon further observation of the images in Figure 1, we can see that for image-1, the WBC has been mistakenly labelled an extra time as RBC while for the rest of the images, one of the RBCs in them has been mistakenly labelled an extra time as WBC. We remove these duplicates, change the labels ('RBC' to 1, 'WBC' to 2) and save the new data as 'clean_anno.csv'.

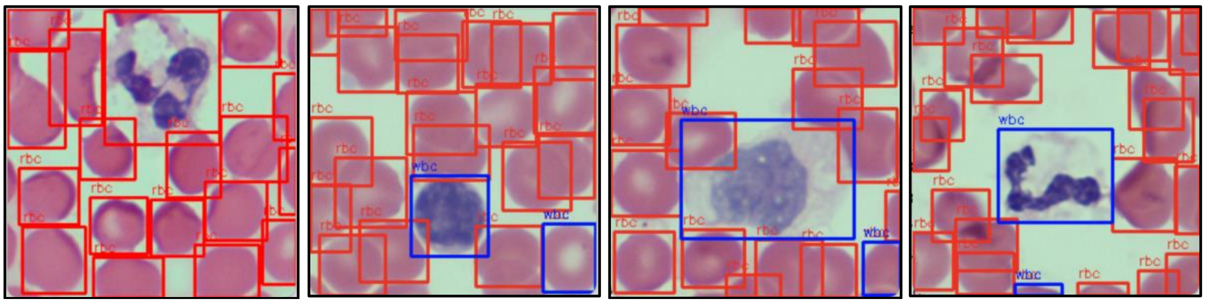

Figure 1: From left to right, image-1, image-100, image-104, image-114

We also observe from Figure 2 that there is significant class imbalance in each image, where WBCs make up 10% or less of the annotations in each image. This may pose a significant problem as features pertaining to the WBCs may not be well-learnt by the model during training, with the model overfitting to the RBCs instead [1].

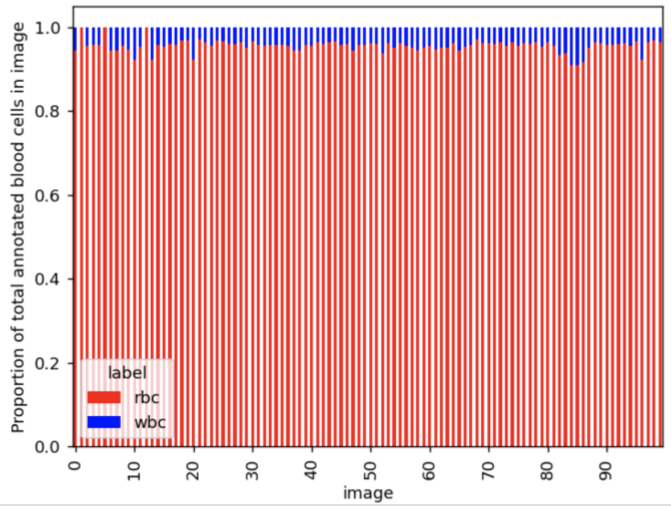

Figure 2: Class distribution in each image

Another potential issue is the dense nature of each image, where there are many objects (RBCs and WBCs) near each other. This results in problems such as overlaps between the ground truth bounding boxes and significant difficulty in identifying the start and end positions of these adjacent objects [2].

Some potential methods we can use to address these issues include the use of generative models to increase the diversity of the dataset and further refining the region proposal process [1] [2]. However, due to time and resource constraints, I will regrettably be unable to implement them in this assessment.

OBJECTIVE:

Generally speaking, deep learning models for object detection try to identify regions-of-interest (ROIs) or anchor boxes as precursors to the bounding boxes and classify these ROIs as either the background or one of the object classes, though the way they do it may differ (e.g. 2-stage models such as R-CNN vs single-stage models like RetinaNet).

For a model to perform well in this object detection task, we require the model to:

1. Generate bounding boxes (of k-th class) that have significant overlap with ground truth boxes (of k-th class) [Maximise true positives, minimise false negative]
2. Generate little to no bounding boxes that have little to no overlap with ground truth boxes. [Minimise false positives]

To check whether a bounding box is a positive classification for an object near it, we can determine the extent of overlap it has with the ground truth box using the Intersection-of-Union (IoU), which is depicted in Figure 1. An IoU value of 1 means the predicted bounding box fits the ground truth box perfectly (the two boxes are the same), while an IoU value of 0 means that the predicted bounding box does not overlap with the ground truth box at all.

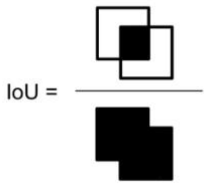

Figure 3: Illustration of IoU (shaded regions represent values taken)

We take a detection to be a true positive (TP) classification if it has IoU greater than α with a ground truth box, where α can be tuned as a hyperparameter during the training process. Conversely, if a detection has IoU value less than α or the ground truth box it overlaps already has been detected, then it is a false positive (FP). A false negative (FN) will be a ground truth bounding box for which no detection has an IoU greater than α with.

From these, we can calculate the precision [TP / (TP + FP)], which tells us the proportion of our detections that are actually blood cells. We can also calculate the recall [TP / (TP + FN)], which tells us what proportion of the blood cells we managed to detect. Ideally, we want to maximise both precision and recall. Average precision (AP) is given by the total area under the Precision-Recall Curve. It ranges from [0, 1] and a high AP value means both precision and recall are high.

For each object class that we have, we are able to calculate an AP value. This leads us to our last metric, the mean average precision (mAP), which is the average of all AP values across all classes. The mAP is a suitable quantitative metric to gauge the performances of their models because it accounts for both the presence of false positives and negatives across all classes. In other words, maximising mAP does not result in a bias towards positive and negative samples, nor does it result severe bias towards any particular classes (assuming the dataset is balanced). The mAP metric is commonly computed with the IoU threshold α being set to 0.5. This may result in an overestimation of model performance, as a detection with an IoU of 0.5 is considered similar to another detection with an IoU of 0.99.

Another variant of mAP, also known as the COCO-mAP, overcomes this by averaging the mAP of all classes over 10 IoU thresholds from 0.5 to 0.95, with step size of 0.05. This means that a model which has all n detections with IoU = 0.5 will have a lower COCO-mAP score than a model which has all n detections with IoU = 0.95.

EXPERIMENTAL WORKFLOW:

As our dataset is limited in size (only 100 images), training a base model from scratch is not feasible. Therefore, we will use transfer learning with pre-trained models. For most object detection models, there is a “backbone” network that is responsible for extracting features from the input image, which is usually trained on large image datasets. After this feature extraction network, there is also a “head” network which resizes the proposed regions or anchor boxes into the final bounding boxes, classifies them into the predicted classes and outputs the regression and classification loss for backpropagation during training. Figure 4 shows a general overview of object detection deep learning models.

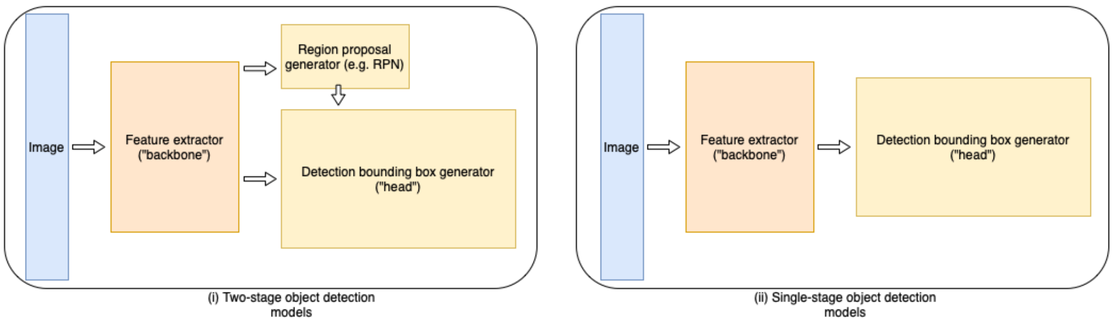

Image
Feature extractor ("backbone")
Region proposal generator (e.g. RPN)
Detection bounding box generator ("head")
(i) Two-stage object detection models
Image
Feature extractor ("backbone")
Detection bounding box generator ("head")
(ii) Single-stage object detection models

Figure 4: Overview of architectures of object detection models

The three pre-trained deep learning models that are implemented in this report are:
1. Faster R-CNN (Two-stage, Backbone network=ResNet50)
2. RetinaNet (Single-stage, Backbone network = ResNet50)
3. Single-Shot Multibox Detector (Single-stage, Backbone network = VGG-16)

In the experiment, we will perform finetuning by retraining the entire “head” network and weights of the final layer of the “backbone” network for all three models. All other layers will retain their original, pre-trained weights.

For the data, we first do a train-test split into a training set containing 80 images and a test set containing 20 images. To prevent data leakage, the test set will remain untouched until the end of all training and hyperparameter tuning processes. As the training data is limited, we will also perform data augmentation during the training process to increase its diversity.

The following processing steps are then done for the data:
1. Normalize the range of image pixel values from [0, 255] to [0, 1].
2. Randomly perform the following data augmentation during training (*p* denotes the probability of the augmentation being applied to each image):
    a. Horizontally flip the image and bounding boxes (*p* = 0.4)
    b. Vertically flip the image and bounding boxes (*p* = 0.4)
3. Further resize and standardise the images using the means and standard deviations of the large image datasets the “backbone” networks were trained on (this step is performed by the PyTorch implementations of the pre-trained models)

For the rest of the experiment setup, they have been summarised in Table 1 below.

Algorithm / Parameter	Value(s)	Reasoning
Stochastic Gradient Descent	Momentum = 0.9	SGD tends to generalize better than adaptive optimizers, although it is slower in convergence [3]
Initial learning rate (divided by 10 every 3 epochs)	Choose from [0.1, 0.05, 0.01, …], till model performance stops improving	Decaying learning rate to help SGD find a good minimum loss

Number of epochs	1 to 30	Initial experimentation shows that model does not improve much beyond 30 epochs
Batch size[1]	2	Since our dataset is limited, we set small batch size to allow for sufficient update steps during optimization.

Table 1: Experimental Setup

We then perform 5-fold cross validation on the training dataset and obtain the mean and standard deviations of the COCO-mAP scores to tune the learning rate and number of epochs.

EXPERIMENTAL RESULTS:

Figure 5 shows the plots of the best validation results obtained for each model.

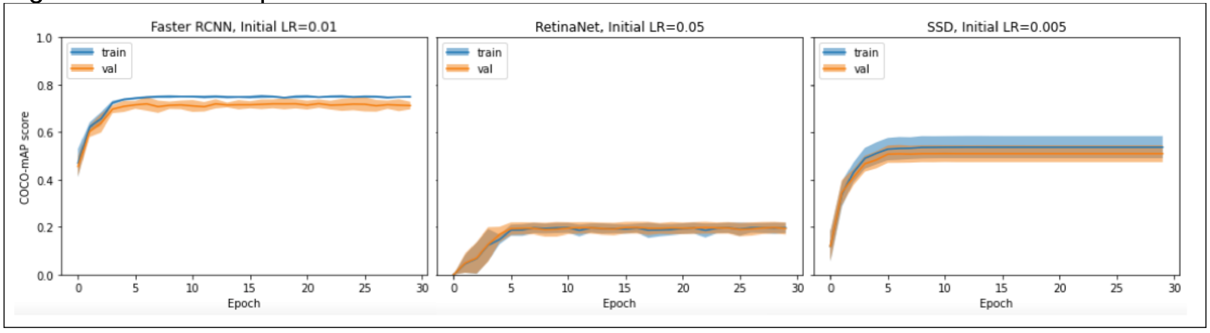

Figure 5: 5-fold CV results for Faster R-CNN, RetinaNet and SSD

Based on the plots, we then select the best set of hyperparameters to use for each model, which is summarized in Table 2. The models are then trained on the full training dataset to obtain our final three models.

Model	Initial learning rate	Number of epochs	Batch size
Faster R-CNN	0.01	15	2
RetinaNet	0.05		
SSD	0.005		

Table 2: Hyperparameters used for final training

All three models are evaluated using the test dataset and we obtain the metrics as shown in Table 3. mAP-50 and mAP-75 represents the mAP value obtained if we set the IoU threshold to be 0.50 and 0.75 respectively.

Model	COCO-mAP	mAP-50	mAP-75
Faster R-CNN	0.7415	0.9882	0.9404
RetinaNet	0.0160	0.0606	0.0023
SSD	0.3981	0.8310	0.3393

Table 3: Test metrics for final models

From Table 3, we observe that across all three metrics, Faster R-CNN has the best performance, followed by SSD, with RetinaNet performing the worst. An interesting observation is that Faster R-CNN has a high mAP-75 value, which suggests that many of its detections overlap significantly with the ground truth boxes. The models used to derive these scores are saved under the “saved_models” folder. An image from the test dataset, along with its ground truth and predicted boxes from each model is shown in Figure 6. It should be noted

[1] I usually perform tuning for batch size as well, but due to time constraints and Colab restricting my GPU access, I am unable to do so here, apologies!

that non-maximum suppression has been applied to the predicted bounding boxes in Figure 6, which helps to reduce the number of predicted bounding boxes that overlap significantly.

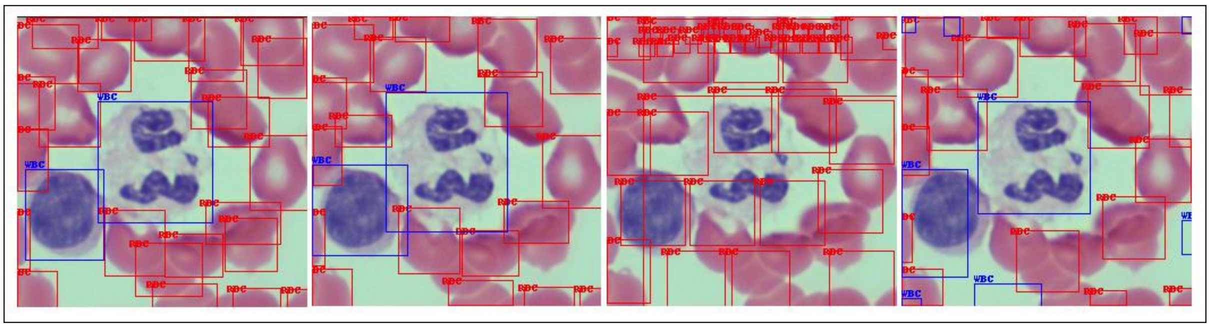

Figure 6: From left to right, ground truth, Faster R-CNN, RetinaNet, SSD predictions

The strong performance of the Faster R-CNN is due to its two-stage approach, where it has a Region Proposal Network (RPN) to learn and generate proposed regions likely to contain the blood cells. Single stage detectors such as SSD and RetinaNet do not include this additional network, resulting in worse performance. However, they are much faster in prediction, thus making them more suitable for real-time use cases [4].

CODE WALKTHROUGH:

Majority of the code has been written in a modular fashion so any complicated but necessary code chunks have been packed away in Python functions or classes. This makes it easier for users to carry out repeated experiments without having to rewrite a lot of the code. The code is also formulated to suit the PyTorch framework and uses many of PyTorch's classes and functions (e.g. PyTorch Dataset, torchvision's transforms). The table below provides a rough overview of the structure of the code and the main functions to take note of:

Folder/Module	Description	Key classes/functions
reference_detect	Contains torchvision modules necessary to run the pre-trained models	-
data	Contains the images and data regarding bounding boxes.	-
saved_models	Contains .pth files that hold the state_dict of trained models	-
img_utils.py	Module with functions for displaying the images with their bounding boxes on them.	• plot_img_w_box
data_utils.py	Module containing the custom PyTorch Dataset for this data, custom data augmentation functions and data visualization functions.	• BloodCellDataset • get_transform • get_duplicates • get_class_distribution
train_eval_utils.py	Module with functions for cross-validation, training and evaluating the models. Also contains helper functions to save and load models.	• train_model, eval_model • hocv_model, kfcv_model • plot_metrics_per_epoch • model_predict • save_model, load_model
bcnet.py	Module with a function to return a modified, pre-trained model that is ready for fine-tuning.	• get_bcnet

There is also a Jupyter Notebook ("main.ipynb") that provides a brief walkthrough on the various stages of the experiment process, such as data exploration, hyperparameter tuning, model training and predicting using trained models, while using the written code.

References

[1] K. Oksuz, B. C. Cam, S. Kalkan and E. Akbas, "Imbalance Problems in Object Detection: A Review," *IEEE Transactions on Pattern Analysis & Machine Intelligence,* vol. 43, no. 10, pp. 3388-3415, 2021.

[2] C. Xu, Y. Zheng, Y. Zhang, G. Li and Y. Wang, "A method for detecting objects in dense scenes," *Open Computer Science,* vol. 12, no. 1, pp. 75-82, 2022.

[3] P. Zhou, J. Feng, C. Ma, C. Xiong, S. Hoi and W. Ee, "Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning," arXiv, 2020.

[4] P. Soviany and R. T. Ionescu, "Optimizing the Trade-Off between Single-Stage and Two-Stage Deep Object Detectors using Image Difficulty Prediction," in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2018.