Contents lists available at IdNNS

Indonesian Journal of Cyber-AI and Security Intelligence

Volume 1 ; Issue 2

journal homepage: https://journal.idnns.org/index.php/ijcasi

Porting an R-CNN Stop-Sign Detector from MATLAB to PyTorch and TensorFlow

Sheeba Armoogum[a,*]

[a] *University of Mautitius, Mauritius*

ARTICLE INFO

Keywords:
R-CNN
object detection
transfer learning
PyTorch
TensorFlow
CIFAR-10
stop-sign detection

ABSTRACT

Region-based convolutional networks (R-CNN) remain a compact way to teach how transfer learning connects image classification with object detection. This paper documents a Python port of the MATLAB "Train Object Detector Using R-CNN Deep Learning" stop-sign example, implemented in parallel in PyTorch and TensorFlow/Keras on Google Colab. The workflow pre-trains a three-block CNN on CIFAR-10, replaces its head with a two-class (stop sign vs. background) head, fine-tunes it on IoU-labelled sliding-window patches, and detects stop signs by scoring windows. We report the results that the executed run actually produced: the PyTorch CNN, trained for 20 epochs, lowered the cross-entropy loss from 2.3033 to 1.6880 and reached 38.34% top-1 accuracy on the 10,000-image CIFAR-10 test set, and both implementations share 116,906 trainable parameters. We analyse why training initially stalled at the chance-level loss, show analytically that a single-scale 32×32 window can yield positive patches (IoU ≥ 0.5) only for objects about 22.6–45.3 px wide, and list the pitfalls that kept the TensorFlow evaluation and the stop-sign fine-tuning stages from completing. The paper is therefore an implementation report with a corrected evaluation protocol, not a benchmark of detection accuracy.

1. Introduction

Object detection, the task of locating and classifying object instances in an image, is one of the foundational problems of computer vision, and its technical evolution over roughly a quarter century is well documented (Zou et al., 2023). A turning point for deep-learning-based detection was R-CNN, which applied a high-capacity convolutional network to bottom-up region proposals and showed that supervised pre-training on an auxiliary task, followed by domain-specific fine-tuning, gives a large boost when labelled detection data are scarce (Girshick et al., 2014). Later region-based detectors replaced the external proposal step with a learned region proposal network that shares convolutional features with the detector (Ren et al., 2017), and recent transformer-based detectors such as RT-DETR reach real-time speed without non-maximum suppression (Zhao et al., 2024).

Although it has been superseded in accuracy and speed, the original R-CNN recipe remains a useful, compact illustration of transfer learning (Zhuang et al., 2021): a small network is pre-trained on a large classification dataset and then re-purposed for detection with only a few dozen annotated images. MathWorks distributes a widely used example of this recipe that pre-trains a CNN on CIFAR-10 (50,000 training images) and then fine-tunes it for stop-sign detection with just 41 training images (The MathWorks, Inc., 2026a). The same documentation now marks the underlying training function as not recommended in favour of other detector types (The MathWorks, Inc., 2026b), which motivates open, framework-neutral ports of the workflow.

Traffic-sign detection is also an active application area. Recent work includes a survey of approaches and datasets (Lim et al., 2023), multi-task recognition with a YOLOv7-based region-of-interest module (Alawaji et al., 2024), lightweight YOLO variants (Cao et al., 2024; Wu et al., 2025), and a lightweight transformer-based detector (Xie et al., 2025). A minimal R-CNN pipeline is not competitive with these systems, but it is a transparent baseline for teaching and for studying how choices in the proposal stage and in the training recipe affect the outcome.

This paper documents a Python port of the MathWorks stop-sign workflow implemented in parallel in PyTorch (Paszke et al., 2019) and TensorFlow/Keras (Abadi et al., 2016) and run on Google Colab. Our contributions are:

- a side-by-side specification of the two implementations (architecture, parameter count and training configuration), showing that both have 116,906 trainable

*Corresponding author
s.armoogum@uom.ac.mu (S. Armoogum)
ORCID(s):

parameters and identifying the configuration differences that must be controlled before the frameworks can be compared fairly;

- the results actually obtained in the executed run, namely the PyTorch CIFAR-10 pre-training curve and its 38.34% test accuracy, together with an analysis of the initial loss plateau and the learning-rate schedule; and
- an analytical bound showing that a single-scale 32×32 sliding window supplies positive training patches only for a narrow range of object sizes, a list of implementation pitfalls, and a corrected evaluation protocol for completing the study.

We state throughout which stages were executed and which were not; no detection accuracy is claimed.

2. Related Work

2.1. Region-based detection and transfer learning

R-CNN combines class-agnostic region proposals with a CNN classifier (Girshick et al., 2014). In the original work, proposals come from selective search, which merges exhaustive search with segmentation and reaches 99% recall with a mean average best overlap of 0.879 at 10,097 locations (Uijlings et al., 2013). Faster R-CNN removes this bottleneck by sharing full-image convolutional features between a region proposal network and the detector, running at about 5 frames per second on a GPU with a VGG-16 backbone (Ren et al., 2017). Transfer learning, the mechanism behind the pre-train/fine-tune step, is surveyed in Zhuang et al. (2021), and a broader history of detectors, datasets and metrics is given in Zou et al. (2023).

2.2. Traffic-sign detection

Lim et al. (2023) review approaches and datasets for traffic-sign recognition. Alawaji et al. (2024) combine a YOLOv7-based region-of-interest module with multi-task classification and report 99.07% test accuracy for their best soft-sharing model, plus real-time tests on Riyadh highways. Cao et al. (2024) propose YOLOv7-tiny-RCA, which reduces the parameter count from 6.13 M to 4.99 M and reports 81.03% mAP. Wu et al. (2025) describe a lightweight YOLO-based detector deployed in a mobile app, and Xie et al. (2025) adapt RT-DETR to traffic-sign detection with a lighter backbone. These systems rely on large annotated traffic-sign datasets and modern one-stage or transformer detectors; the setting studied here (41 training images and a CNN with about 117 thousand parameters) is deliberately far smaller.

2.3. Comparisons of deep-learning frameworks

PyTorch was introduced as an imperative, high-performance library (Paszke et al., 2019), and TensorFlow as a system for large-scale machine learning (Abadi et al., 2016). PyTorch 2 later added graph compilation through TorchDynamo and TorchInductor while keeping the flexibility of Python (Ansel et al., 2024). Empirical comparisons do not point to a single winner. Novac et al. (2022) trained the same stereo-matching CNN in both libraries and measured PyTorch to be about 25.5% faster in training and 77.7% faster at evaluation, while TensorFlow reached a slightly lower error rate (a 1.16% difference). Georgiou et al. (2022) found that TensorFlow had better energy and run-time performance in training, whereas PyTorch was better in inference in 66% of the cases studied, with comparable model accuracy. Such results depend on the model, the data pipeline and the configuration, so a fair comparison requires matched preprocessing, hyperparameters and seeds. Section 3 examines to what extent this holds for our two implementations.

3. Method

3.1. Overview

Fig. 1 summarises the workflow, which follows the MathWorks example (The MathWorks, Inc., 2026a): (i) pre-train a small CNN on CIFAR-10; (ii) replace the final layers so that the network outputs two classes (stop sign and background); (iii) fine-tune on patches derived from the annotated stop-sign images; and (iv) detect by scoring candidate windows in a test image. Each stage is implemented twice, once per framework, in a single Colab notebook.

3.2. Data

CIFAR-10 (Krizhevsky, 2009) contains 50,000 training and 10,000 test images of size 32×32 with three colour channels in ten classes. The PyTorch branch loads it through torchvision and the TensorFlow branch through `keras.datasets`. The detection stage uses the `stopSignsAndCars.mat` ground-truth table of the MathWorks example, which holds 41 images annotated with stop-sign bounding boxes in $[x, y, w, h]$ format.

3.3. CNN architecture

Both frameworks use the same topology: three Conv–ReLU–MaxPool blocks followed by two fully connected layers (Table 1). With 5×5 kernels, stride 1 and padding 2 (`padding='same'` in Keras), the convolutions preserve spatial size, and the 3×3, stride-2 pooling reduces the 32×32 input to 15×15, 7×7 and finally 3×3, giving $64 \times 3 \times 3 = 576$ features for the first fully connected layer.

The Keras model summary in the notebook reports 116,906 trainable parameters, which matches the analytic count in Table 1 and the layer shapes of the PyTorch model printed in the same notebook (`Linear(576, 64)` and `Linear(64, 10)`). In both ports the first convolutional layer is initialised from $\mathcal{N}(0, 0.0001^2)$; all other layers use their framework's default initialiser, and these defaults are not identical between PyTorch and Keras.

3.4. Pre-training configuration

Table 2 lists the pre-training settings as coded in the notebook. The two configurations are close but not identical. The input scaling and the number of epochs differ; the Keras

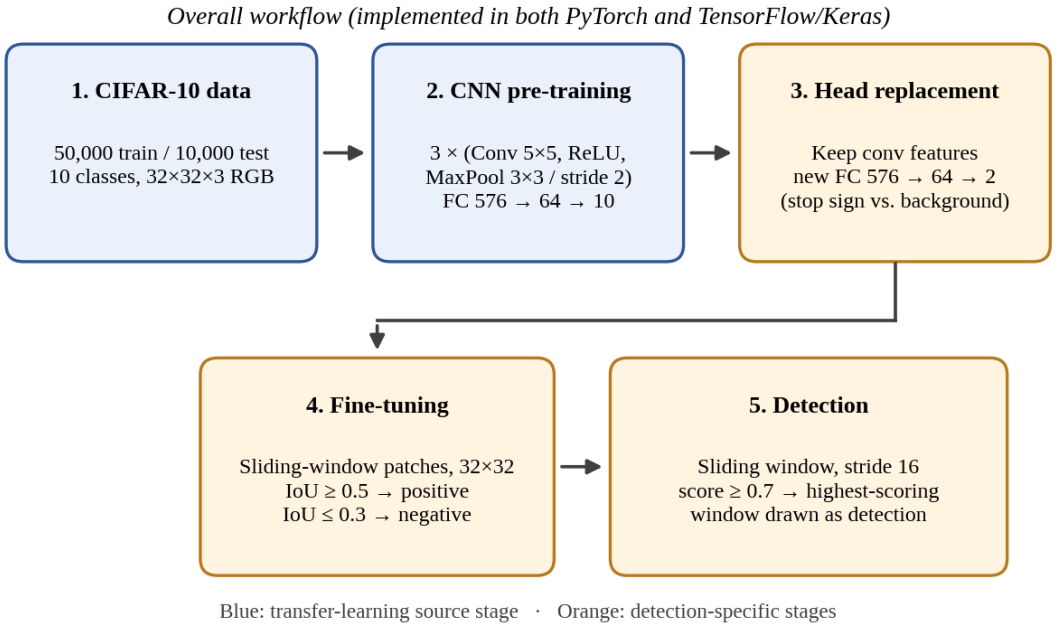


Figure 1: Workflow implemented in the notebook. Stages 1–2 (blue) form the transfer-learning source; stages 3–5 (orange) are specific to stop-sign detection.

Table 1
Layer configuration and parameter count of the CIFAR-10 CNN (identical in both frameworks).

Layer	Configuration	Output size	Parameters
Input	RGB image	$32 \times 32 \times 3$	0
Conv1 + ReLU	32 filters, 5×5, stride 1, padding 2	$32 \times 32 \times 32$	2,432
MaxPool1	3×3, stride 2	$15 \times 15 \times 32$	0
Conv2 + ReLU	32 filters, 5×5, stride 1, padding 2	$15 \times 15 \times 32$	25,632
MaxPool2	3×3, stride 2	$7 \times 7 \times 32$	0
Conv3 + ReLU	64 filters, 5×5, stride 1, padding 2	$7 \times 7 \times 64$	51,264
MaxPool3	3×3, stride 2	$3 \times 3 \times 64$	0
Flatten	—	576	0
FC1 + ReLU	64 units	64	36,928
FC2	10 units (softmax at inference)	10	650
Total			**116,906**

scheduler multiplies the learning rate by 0.1 at the start of the eighth epoch, whereas PyTorch's `StepLR` does so after the eighth epoch has finished, a one-epoch offset; and the weight-decay argument is realised by each framework's own optimiser, so its interaction with momentum should be verified rather than assumed equal. Because only the PyTorch branch was trained in the recorded run, these differences do not affect the reported numbers, but they must be equalised before any accuracy or speed comparison.

3.5. R-CNN fine-tuning and detection

For detection, the convolutional feature extractor is copied from the pre-trained network and a new two-class head (FC 64, then FC 2) is attached. Training patches are produced by sliding a 32×32 window with stride 16 over each annotated image. A window is labelled positive if its maximum IoU with any ground-truth box lies in $[0.5, 1.0]$, negative if it lies in $[0.0, 0.3]$, and is discarded otherwise. Patches are resized to 32×32 and the network is fine-tuned for 100 epochs with SGD (lr 10^{-3}, momentum 0.9, batch size 128); the PyTorch scheduler uses a step of 100 epochs, so the learning rate is effectively constant.

At test time the same window grid is scored by the fine-tuned network. Windows whose softmax probability for the stop-sign class is at least 0.7 are kept and only the highest-scoring window is drawn. There is no multi-scale search, bounding-box regression or non-maximum suppression.

3.6. Proposed evaluation protocol

To complete the study we adopt the following protocol. For CIFAR-10, report top-1 accuracy on the 10,000-image test set as mean ± standard deviation over at least five

Table 2
Pre-training configuration of the two branches, as coded in the notebook.

Setting	PyTorch	TensorFlow / Keras
Input scaling	Mean 0.5, std 0.5 per channel (range $[-1, 1]$)	Division by 255 (range $[0, 1]$)
Output / loss	Logits + `CrossEntropyLoss`	Softmax + categorical cross-entropy (one-hot labels)
Optimiser	SGD, lr 10^{-3}, momentum 0.9, weight decay 0.004	SGD, lr 10^{-3}, momentum 0.9, `weight_decay` 0.004
LR schedule	`StepLR`: $\times 0.1$ after every 8 epochs	`LearningRateScheduler`: $\times 0.1$ at the start of every 8th epoch
Epochs / batch	20 / 128	40 / 128
Run in this study	Yes	No

seeds per framework, with identical input scaling, epochs and schedule timing. For detection, count a prediction as correct if its IoU with an unmatched ground-truth box is at least 0.5, and report precision–recall curves and average precision on a held-out split of the 41 images. The split must be made by image rather than by patch, because windows with stride 16 and width 32 overlap by half and patches from one picture would otherwise leak between training and test sets. Training time and inference throughput should be measured with synchronised GPU timers, stating whether eager or compiled execution (Ansel et al., 2024) was used.

4. Experimental Setup

All experiments ran in a Google Colab notebook. PyTorch reported a CUDA device and TensorFlow listed one GPU. The GPU model and library versions were not recorded in the notebook output, apart from a Python 3.12 runtime visible in a traceback path. Each configuration was run once with a single seed, so every number below is a single-run observation.

5. Results and Discussion

5.1. CIFAR-10 data

Both branches loaded the expected data: the PyTorch loader reported 50,000 training images and the ten class names, and the TensorFlow arrays had shapes $(50000, 32, 32, 3)$ for training and $(10000, 32, 32, 3)$ for testing. Fig. 2 shows the first 25 training images produced by the run.

5.2. PyTorch pre-training

Fig. 3 plots the mean training loss per epoch, re-plotted from the values logged by the notebook, and Table 3 lists selected epochs. The loss stays at about 2.30 for the first four epochs (2.3033, 2.3026, 2.3017 and 2.2990), which equals $\ln 10 \approx 2.3026$, the loss of a classifier that predicts a uniform distribution over ten classes. It then falls quickly between epochs 5 and 8 (2.2779 to 1.8538), and this phase accounts for 73.1% of the total reduction of 0.6153 between epochs 1 and 20. After the learning rate is cut to 10^{-4} for epoch 9, progress slows to 0.0839 between epochs 9 and 20 (1.7719 to 1.6880). The final mean training loss is 1.6880 and the network reaches 38.34% top-1 accuracy on the CIFAR-10 test set, which is 3.8 times the 10% chance level but low for this task.

Table 3
Selected epochs of the PyTorch pre-training run (from the notebook log).

Epoch	Learning rate	Mean training loss
1	10^{-3}	2.3033
4	10^{-3}	2.2990
5	10^{-3}	2.2779
8	10^{-3}	1.8538
9	10^{-4}	1.7719
16	10^{-4}	1.6992
17	10^{-5}	1.6910
20	10^{-5}	1.6880

Two design choices are plausible causes of the slow start and the low final accuracy. First, the first-layer weights have a standard deviation of only 10^{-4}, so initial activations, and therefore the gradients reaching the earlier layers, are very small, and the network needs several epochs to leave the uniform-prediction plateau. Second, the schedule reduces the learning rate tenfold after epoch 8, just after the loss has begun to fall, so 12 of the 20 epochs run at a rate 10 to 100 times smaller. These are hypotheses: separating them requires an ablation over the initial standard deviation, the schedule and the number of epochs. Training accuracy and a validation curve were not recorded, so the high final loss is best read as an under-trained model rather than an overfitted one, and the test accuracy is a single end-of-training measurement without early stopping.

5.3. TensorFlow implementation

The Keras model builds and its summary reports 116,906 trainable parameters, identical to Table 1. The TensorFlow branch was not trained in the recorded run: the training flag was set to `False`, the output shows neither training nor model loading, and the following evaluation call raised an error because the model had not been compiled. We therefore report no TensorFlow accuracy, and no accuracy or speed comparison between the two frameworks can be drawn from this run.

Table 4
Execution status of each stage in the recorded run.

Stage	Coded	Output recorded	Observation
CIFAR-10 loading (both)	Yes	Yes	50,000 / 10,000 images; shapes as expected
PyTorch pre-training	Yes	Yes	Loss 2.3033 → 1.6880; test accuracy 38.34%
TensorFlow pre-training	Yes	No	Training flag `False`; evaluation raised a `compile()` error
Stop-sign data loading	Yes	No	Needs `stopSignsAndCars.mat` and the image files
Patch extraction and fine-tuning	Yes	No	Patch counts and losses not recorded
Detection on a test image	Yes	No	Needs a test image (`stopSignTest.jpg`)

5.4. Limits of a single-scale sliding-window proposal generator

The proposal stage replaces selective search (Uijlings et al., 2013) with a fixed 32×32 window. For a window of area $a = 32^2 = 1024$ px^2 and a ground-truth box of area A, the IoU is bounded by

$$\mathrm{IoU} \le \frac{\min(A, a)}{\max(A, a)}, \tag{1}$$

because the intersection cannot exceed the smaller area and the union cannot be smaller than the larger area. A positive label (IoU ≥ 0.5) therefore requires $512 \le A \le 2048$ px^2, which for a square box means a side of 22.6–45.3 px (Fig. 4). For box sides above $32/\sqrt{0.3} \approx 58.4$ px every window has IoU ≤ 0.3 and is labelled negative, including windows that lie entirely inside the object.

The notebook does not record the box sizes of the stop-sign images or the number of positive and negative patches, so we cannot state how many positives the run would have produced. The bound does imply, however, that for stop signs wider than about 45 px a single-scale grid yields no positive patches and mislabels object interiors as background. Multi-scale windows (an image pyramid or several window sizes) or a proposal method such as selective search avoid this failure mode.

5.5. Status of the remaining stages and implementation pitfalls

Table 4 summarises which stages produced recorded output. The stop-sign stages (data loading, patch extraction, fine-tuning and detection) are implemented but have no recorded output, so no detection result is reported.

Code inspection and the recorded errors reveal the following pitfalls, which should be fixed before the missing stages are run:

- *Head dimension in the PyTorch R-CNN.* The two-class head is defined with $64 \cdot 2 \cdot 2 = 256$ input features, but the copied feature extractor outputs $64 \cdot 3 \cdot 3 = 576$ features for 32×32 inputs (Table 1). The forward pass would fail with a shape error of the same kind as the one recorded in an earlier training cell (128×576 versus 256×64). The head should take 576 inputs.
- *Image paths.* The ground-truth table stores file names that the MathWorks example remaps to the local file system before use (The MathWorks, Inc., 2026a). The notebook passes them directly to OpenCV, which returns no image for a missing file, and the patch-extraction loop skips such images silently, so an empty training set can arise without any error message.
- *Uncompiled Keras model.* A Keras model must be trained, or loaded in compiled form, or compiled explicitly before `evaluate` is called; otherwise the call fails as recorded.
- *Unmatched preprocessing and schedules.* The scaling, epoch and schedule-timing differences in Table 2 confound any comparison between the frameworks.

5.6. What can and cannot be concluded about the frameworks

Because only one branch produced results, we make no claim about which framework is better. The literature reports mixed outcomes (Novac et al., 2022; Georgiou et al., 2022), and the choice between eager and compiled execution (Ansel et al., 2024) can change timing further. The value of the present port lies in making the equivalences and the remaining differences explicit, so that a controlled comparison can be run under the protocol in Section 3.6.

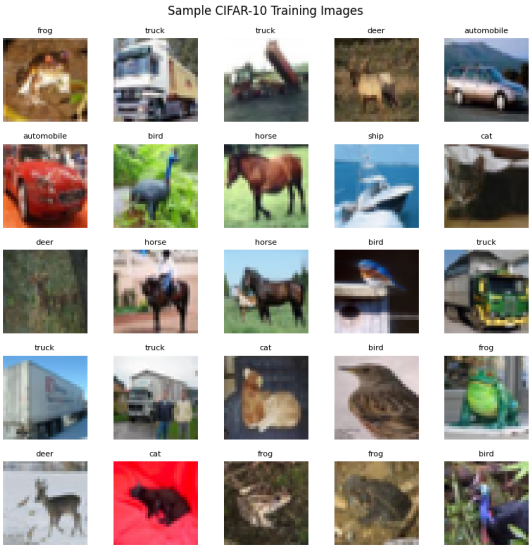


Figure 2: First 25 CIFAR-10 training images and their labels (output of the notebook).

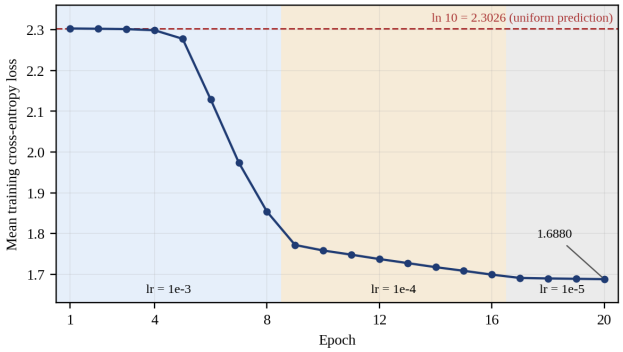


Figure 3: Mean training loss of the PyTorch CNN over 20 epochs, re-plotted from the notebook log. Shading marks the learning-rate phases; the dashed line is $\ln 10$.

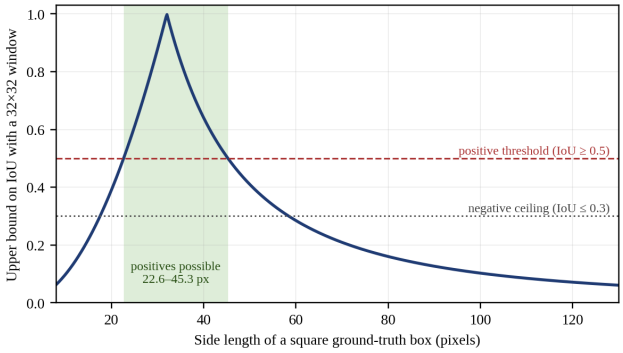


Figure 4: Analytical upper bound on the IoU between a 32×32 window and a square ground-truth box. Positive patches ($\text{IoU} \geq 0.5$) can exist only inside the shaded range.

6. Limitations

- Only the PyTorch CIFAR-10 stage produced quantitative results, from a single seed; no confidence interval can be given.
- No TensorFlow accuracy and no stop-sign detection metrics are available, so the paper does not evaluate R-CNN detection quality.
- The pre-trained network is a small CNN of about 117 thousand parameters trained on CIFAR-10, not an ImageNet-scale model, and the fine-tuning set contains only 41 images.
- Hardware, GPU model and library versions were not recorded, and the two branches differ in preprocessing, epochs and schedule timing.
- The explanations offered for the loss plateau and the low accuracy are hypotheses that have not been tested by ablation.

7. Conclusions and Future Work

We ported the MathWorks R-CNN stop-sign workflow to PyTorch and TensorFlow/Keras and documented it stage by stage. The executed run shows that the CIFAR-10 pre-training stage in PyTorch works but under-trains under the ported settings, reaching 38.34% test accuracy after a loss plateau at ln 10, and that the two implementations are structurally identical at 116,906 parameters. The analysis shows that a single-scale 32×32 window can supply positive patches only for objects about 22.6–45.3 px wide and identifies concrete pitfalls that blocked the remaining stages.

Future work follows directly from the protocol in Section 3.6: repair the pitfalls, equalise the two configurations, run at least five seeds per framework, replace the fixed window with multi-scale proposals or selective search, split the 41 images by image, and report average precision at IoU 0.5 together with training time and inference throughput. Ablations over the first-layer initialisation, the learning-rate schedule and the number of epochs would test the explanations proposed in Section 5.2.

References

Abadi, M., et al., 2016. TensorFlow: A system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA. pp. 265–283.

Alawaji, K., Hedjar, R., Zuair, M., 2024. Traffic sign recognition using multi-task deep learning for self-driving vehicles. Sensors 24, 3282. doi:10.3390/s24113282.

Ansel, J., et al., 2024. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), La Jolla, CA, USA. doi:10.1145/3620665.3640366.

Cao, X., Xu, Y., He, J., Liu, J., Wang, Y., 2024. A lightweight traffic sign detection method with improved YOLOv7-tiny. IEEE Access 12, 105131–105147. doi:10.1109/ACCESS.2024.3435384.

Georgiou, S., Kechagia, M., Sharma, T., Sarro, F., Zou, Y., 2022. Green AI: Do deep learning frameworks have different costs?, in: Proceedings of the 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA. pp. 1082–1094. doi:10.1145/3510003.3510221.

Girshick, R., Donahue, J., Darrell, T., Malik, J., 2014. Rich feature hierarchies for accurate object detection and semantic segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Columbus, OH, USA. pp. 580–587. doi:10.1109/CVPR.2014.81.

Krizhevsky, A., 2009. Learning multiple layers of features from tiny images. Technical Report. Department of Computer Science, University of Toronto. Toronto, ON, Canada.

Lim, X.R., Lee, C.P., Lim, K.M., Ong, T.S., Alqahtani, A., Ali, M., 2023. Recent advances in traffic sign recognition: Approaches and datasets. Sensors 23, 4674. doi:10.3390/s23104674.

Novac, O.C., Chirodea, M.C., Novac, C.M., Bizon, N., Oproescu, M., Stan, O.P., Gordan, C.E., 2022. Analysis of the application efficiency of TensorFlow and PyTorch in convolutional neural network. Sensors 22, 8872. doi:10.3390/s22228872.

Paszke, A., et al., 2019. PyTorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32, Curran Associates, Red Hook, NY, USA. pp. 8024–8035.

Ren, S., He, K., Girshick, R., Sun, J., 2017. Faster R-CNN: Towards real-time object detection with region proposal networks. IEEE Transactions on Pattern Analysis and Machine Intelligence 39, 1137–1149.

The MathWorks, Inc., 2026a. Train object detector using R-CNN deep learning. Computer Vision Toolbox documentation. https://www.mathworks.com/help/vision/ug/object-detection-using-deep-learning.html. Accessed: Sep. 28, 2026.

The MathWorks, Inc., 2026b. trainRCNNObjectDetector. Computer Vision Toolbox documentation. https://www.mathworks.com/help/vision/ref/trainrcnnobjectdetector.html. Accessed: Sep. 28, 2026.

Uijlings, J.R.R., van de Sande, K.E.A., Gevers, T., Smeulders, A.W.M., 2013. Selective search for object recognition. International Journal of Computer Vision 104, 154–171. doi:10.1007/s11263-013-0620-5.

Wu, Y., Zhang, T., Niu, J., Chang, Y., Liu, G., 2025. YOLO-based lightweight traffic sign detection algorithm and mobile deployment. Optoelectronics Letters 21, 249. doi:10.1007/s11801-025-4153-2.

Xie, H., Zhang, X., Wei, P., Cui, C., 2025. Lightweight traffic sign detection based on RT-DETR. Journal of Applied Optics 46, 300. doi:10.5768/JAO202546.0202002.

Zhao, Y., Lv, W., Xu, S., Wei, J., Wang, G., Dang, Q., Liu, Y., Chen, J., 2024. DETRs beat YOLOs on real-time object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 16965–16974. doi:10.1109/CVPR52733.2024.01605.

Zhuang, F., et al., 2021. A comprehensive survey on transfer learning. Proceedings of the IEEE 109, 43–76.

Zou, Z., Chen, K., Shi, Z., Guo, Y., Ye, J., 2023. Object detection in 20 years: A survey. Proceedings of the IEEE 111, 257–276.