Journal of Applied Informatics and Computing (JAIC)
Vol.9, No.6, December 2025, pp. 3057~3067
e-ISSN: 2548-6861

A Two-Stage Braille Recognition System Using YOLOv8 for Detection and CNN for Classification

Tan Valencio Yobert Geraldo Setiawan [1], Eko Hari Rachmawanto [2]*
* Informatics Engineering, Faculty of Computer Science, Universitas Dian Nuswantoro, Semarang, Indonesia
111202213984@mhs.dinus.ac.id [1], eko.hari@dns.dinus.ac.id [2]

Article Info

Article history:

Received 2025-10-11
Revised 2025-10-27
Accepted 2025-11-08

Keyword:

Braille Recognition,
Character Classification,
MobileNetV2,
Object Detection,
YOLOv8.

ABSTRACT

Automatic recognition of Braille characters remains a challenge in the field of computer vision, especially due to variations in shape, size, and lighting conditions in images. This research proposes a two-stage system to detect and recognize Braille letters in real time using a deep learning approach. In the first stage, the YOLOv8 model is used to detect the position of Braille characters within an image. The detected regions are then processed in the second stage using a classification model based on the MobileNetV2 CNN architecture. The dataset used consists of 7,016 Braille character images, collected from a combination of the AEyeAlliance dataset and annotated data from Roboflow. To address the class imbalance problem—particularly for letters T to Z which had fewer samples—oversampling and image augmentation techniques were applied that makes the final combined dataset contained approximately 7,616 images. The system was tested on 1,513 images and achieved strong results, with average precision, recall, and F1-score of 0.98, and an overall accuracy of 98%. This two-stage method effectively separates detection and classification tasks, resulting in an efficient and accurate Braille recognition system suitable for real-time applications.

I. INTRODUCTION

Braille is a tactile writing system designed to assist individuals with visual impairments in reading, writing, and communicating effectively [1]. Developed by Louis Braille, this system uses a rectangular cell structure consisting of six raised dots arranged in a 2×3 matrix, where each combination of dots represents a different letter or symbol. Mastering Braille plays a vital role in promoting literacy and intellectual independence for the blind community [2]. However, for those unfamiliar with Braille, such as the general public, reading or interpreting Braille remains a major barrier in communication [3].

To bridge this gap, there is a growing need for intelligent systems capable of detecting and translating Braille into Latin alphabet text in real-time. Recent advances in computer vision and deep learning, especially convolutional neural networks (CNNs), have made it possible to recognize objects or characters from images or videos [4]. One of the most widely adopted models in object detection is YOLO (You Only Look Once), known for its high detection speed and real-time performance. The latest iteration, YOLOv8, offers improved detection accuracy and model efficiency.

While several studies have been successful in translating Braille characters using image classification, many of them rely on static input or web-based tools that are not optimized for real-time applications. In this study, we propose a real-time Braille recognition system using a two-stage approach. The first stage involves detecting Braille characters in a live camera feed using YOLOv8, while the second stage classifies each detected region using a MobileNetV2-based CNN. This method separates object localization and character recognition, enabling better performance in real-time scenarios [5].

A comprehensive dataset was prepared by combining annotated data from AEyeAlliance and Roboflow, followed by preprocessing, augmentation, and oversampling; particularly for characters with fewer samples (T–Z) [6]. Through this system, users can instantly translate Braille into readable text using only a webcam, making the tool more accessible for non-technical users and practical for everyday use.

Several researchers have explored Braille character recognition using image classification techniques. Ovodov (2023) introduced a robust Braille detection method using an object detection CNN based on the RetinaNet architecture. His model was trained on the Angelina Braille Images Dataset, which consists of 240 smartphone-captured images in natural conditions, including perspective distortion and curved paper. The model achieved a character-level F1 score of up to 0.9981 and demonstrated its capability to detect Braille characters accurately even under challenging visual conditions. However, the system was not designed for real-time interaction, and inference was performed on previously captured images using high-end GPU resources [7]. Other studies used traditional methods like Support Vector Machines (SVM) or K-Nearest Neighbors (KNN) for Braille recognition. While these approaches demonstrated reasonable accuracy in controlled environments, they were often limited in scalability and generalization due to varying lighting conditions, distortions, or image noise [8]. YOLO-based object detection has been widely used in other domains, including vehicle detection, face tracking, and text recognition. Its adoption in Braille recognition, however, remains relatively underexplored. Additionally, lightweight CNN models like MobileNetV2 have proven to be highly efficient for classification tasks on mobile and embedded systems, yet their application in real-time Braille translation systems is still emerging [9]. This research fills the gap by integrating YOLOv8 for real-time Braille localization with MobileNetV2 for lightweight and accurate classification, forming a two-stage pipeline capable of translating Braille characters directly from webcam input—without the need for scanning or uploading images. This approach improves usability and applicability, compared to earlier methods that relied on static images or complex setups.

II. METHODS

This research was conducted to develop a real-time Braille recognition system that can detect and classify Braille characters from camera input and convert them into readable Latin letters. The methodology used in this study follows a two-stage architecture combining object detection and image classification techniques. In the first stage, YOLOv8 (You Only Look Once version 8) is used as the object detection model to localize each Braille character region in an image or video frame. YOLOv8 is known for its fast and accurate performance in detecting multiple objects simultaneously, making it suitable for real-time task [10].

In the second stage, each detected Braille region is cropped and passed into a Convolutional Neural Network (CNN) classifier based on the MobileNetV2 architecture. This model is responsible for identifying the Braille pattern and classifying it into the corresponding alphabet letter from A to Z.

The study was carried out using Python on Jupyter Notebook and Google Colab, which offers a free GPU for computing. Libraries such as OpenCV were used to access camera input and display real-time prediction results. TensorFlow and Keras were used to build and train the CNN classifier, while Ultralytics library was used to implement YOLOv8.

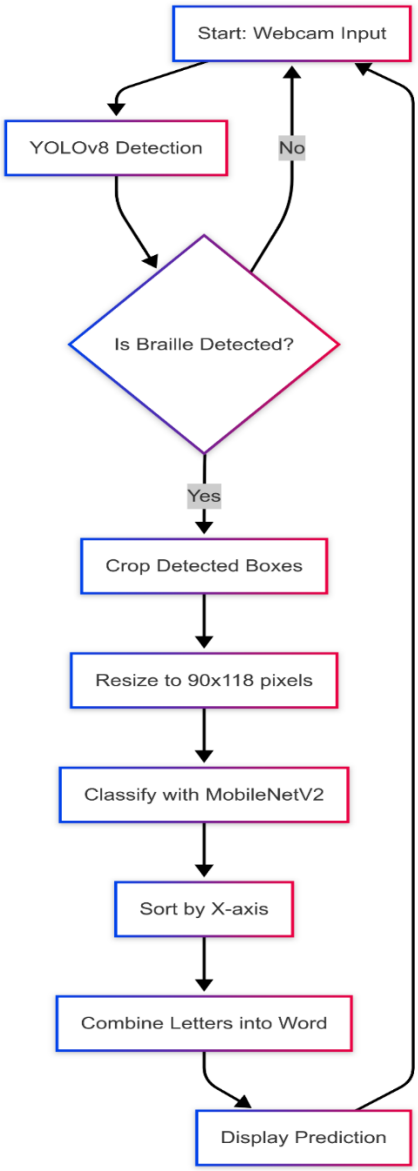

Figure 1 Flowchart of the proposed real-time Braille recognition system combining YOLOv8 detection and MobileNetV2 classification.

To ensure the system runs properly in a real-world scenario, this methodology also involves several other stages such as:

- Dataset preparation and augmentation,
- Model training and evaluation,

- Real-time integration using OpenCV, and
- Output display for user-friendly interaction.

This method was chosen because it separates detection and classification into two different stages, which helps to simplify the architecture and improve the flexibility of each component. The detection model only focuses on finding the Braille objects, while the classification model can be fine-tuned or replaced independently without affecting the detection process.

A. Dataset Collection and Preparation

The dataset used in this research was obtained through the combination of two publicly available sources: the AEyeAlliance Braille Character Dataset and a customized annotated dataset exported from Roboflow. These datasets collectively contain labeled images of Braille characters representing all 26 alphabet letters (A–Z). However, due to imbalance data distribution across classes, we used preprocessing and augmentation to balance the data [11].

1) *AEyeAlliance Dataset*

The AEyeAlliance dataset consists of images collected from various sources, where each image is labeled with a single Braille character. The labels were provided in accompanying .csv files containing the image URLs and character annotations in JSON format. To utilize this dataset efficiently, a script was written to parse and download all image–label pairs automatically. Each image was then sorted into subfolders named after its respective class (A–Z), enabling the use of directory-based image generators for model training. While this dataset offered a clean and labeled collection of images, it exhibited a class imbalance problem. Specifically, classes T to Z had significantly fewer samples (approximately 120–140 images per class), while classes A to S between 250 to 340 images. This imbalance posed a potential risk of bias in the training process.

2) *RoboFlow Dataset*

To enrich the dataset and address class *imbalance*, an additional set of labeled images was collected from RoboFlow. The dataset was structured in YOLOv8 annotation format, with separate directories for images/ and labels/. Each label file contained bounding box coordinates and the class index corresponding to a Braille letter. This dataset was originally intended for object detection training but was adapted for use in classification by cropping each detected Braille region and assigning its label accordingly. After extraction and parsing of the YOLOv8-format annotations, the Braille characters were cropped from their original images and stored in folders by class, similar with the AEyeAlliance format. This procedure not only increased the number of samples per class, but also helped to simulate the appearance of Braille characters in more diverse contexts (e.g., lighting, background, orientation).

In this study, we used a mix of real-world photos and digital-generated images for the Braille dataset to make sure the model learns from different kinds of examples. The AEyeAlliance part mostly has actual photos taken with phone cameras, showing raised Braille dots on things like paper or plastic under normal lighting and angles, which helps handle stuff like shadows or bends in real life. On the other hand, the Roboflow dataset includes synthetic images created on computers to simulate Braille patterns in various setups, like different backgrounds or rotations. Combining them like this lets the model get better at recognizing Braille no matter the situation, and it also fixes issues with not having enough variety in just one type of data.

3) *Data Preprocessing*

Before throwing the images into the models, we did some basic preprocessing to get everything ready and consistent. For YOLOv8, we resized all images to 640x640 pixels because that iswhat the model expects for input, and it helps with faster detection without losing too much detail. We did not do extra pixel normalization manually since the Ultralytics library handles scaling the pixel values automatically during training. For MobileNetV2, we resized to 224x224 pixels to match its standard setup, and we normalized the pixels to a range of 0 to 1 to speed up how the model learns. We skipped turning things to grayscale because keeping RGB colors helps the models pick up on background differences and textures better.

4) *Data Augmentation and Oversampling for CNN*

To further balance the dataset and improve model generalization, some data augmentations were applied for some images. These included:

- Random rotation (±15 degrees),
- Width and height shifting (up to 10%),
- Zooming (up to 20%), and
- Brightness and contrast adjustment (between -7% and +7%)
- Noise: (up to 1.01% of pixels)

Here, oversampling was applied to the underrepresented classes (T to Z) by increasing their total image count through duplication and augmentation until they matched the number of samples in the majority classes. This ensured that the CNN classifier would receive balanced input across all 26 alphabet classes during training [12].

5) *Dataset Splitting*

The final dataset, after applying oversampling and augmentation, included around 7,616 images, evenly spread across the 26 Braille letters (A–Z). These images were divided into training and validation sets with an 80:20 split. The training set was used to teach the CNN classifier, while the validation set helped track overfitting and supported early stopping during training. Separately, 1,513 images were set aside as a test set, kept unseen by the model during

training and validation, to assess how well the system performs on new, real-world data.

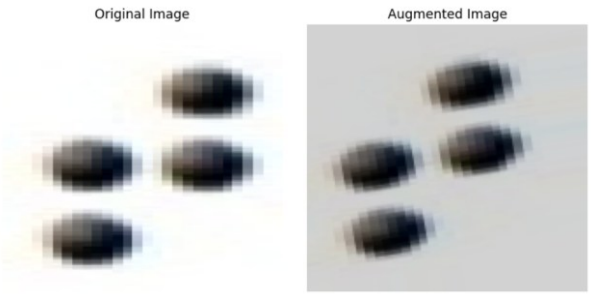

Figure 2 Comparison Between the Original Image and Augmented Image

B. YOLOv8 Data Setup for Detection

To detect and pinpoint Braille characters in images, this study uses YOLOv8, a cutting-edge object detection model created by Ultralytics [13]. YOLOv8 is known for its fast performance, anchor-free design, and excellent accuracy, making it ideal for spotting small objects like Braille, which appear as tight dot patterns in organized layouts [14]. The dataset for this phase consists of Braille character images that have been annotated with bounding boxes in the YOLOv8 format. Each annotation contains a class index corresponding to one of the 26 alphabet letters (A to Z) and bounding box coordinates that are normalized to the image dimensions. The dataset was structured into standard YOLO directories, separating training, validation, and testing data into their respective image and label folders. Instead of training YOLOv8 on a single generic "Braille" class, the model was trained using 26 distinct classes representing each alphabet letter (A–Z). This approach greatly improved the model's ability to distinguish between individual Braille characters during detection. By assigning unique labels for each letter, the YOLOv8 model could directly recognize and localize specific characters in a single inference step, reducing post-processing complexity and improving detection accuracy, especially in real-time scenarios. This approach also enabled the system to provide more informative and structured outputs, aligning well with the end goal of full Braille-to-text translation.

A total of 5,242 images were gathered and split into three groups: 4,154 images for training, about 611 for validation, and the remaining 477 for testing. To boost the model's performance, the training images were apllied with simple augmentation, such as random rotations between –6 and +6 degrees and brightness changes from –8% to +8%. This process expanded the training set to 12,462 images. The YOLOv8 model, specifically the yolov8s variant, was trained on Google Colab with GPU support for 25 epochs. This model balances speed and precision effectively, using a batch size of 16, and input images sized at 640x640 pixels. Throughout training, the model's performance was checked against the validation set, and the best weights were saved as "best.pt".

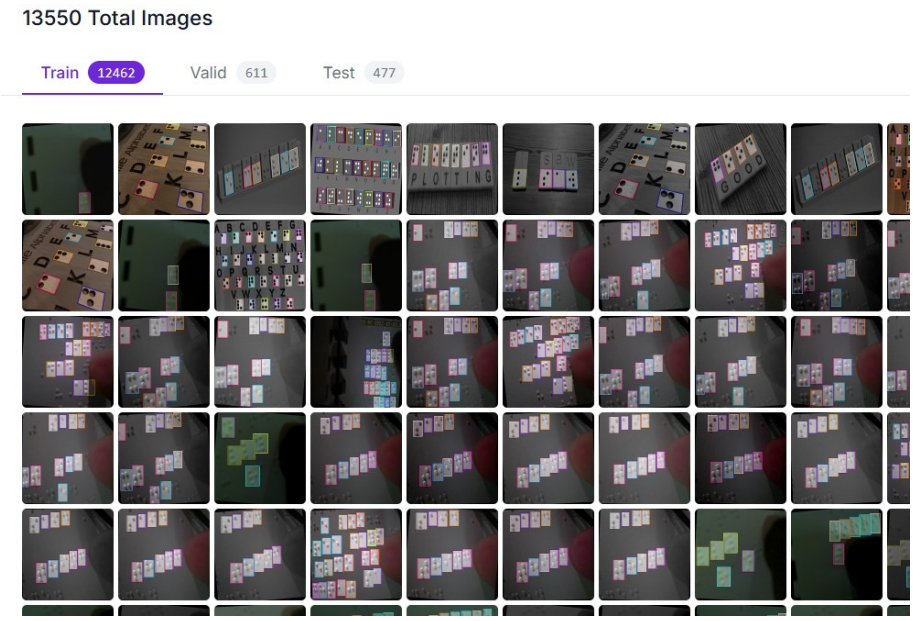

Figure 3 YOLOv8 Braille Detection Dataset

Once trained, the YOLOv8 model was incorporated into a real-time detection system using OpenCV. During operation, webcam frames were fed into the model, which generated bounding boxes, confidence scores, and class labels. Only predictions with confidence scores above 0.5 were kept to ensure reliability. The bounding boxes were arranged by their x-coordinates to maintain a left-to-right reading sequence, and each cropped Braille area was sent to a CNN classifier for letter identification. This two-stage system, which separates object detection and character classification, enables higher flexibility and improves accuracy, especially in real-time applications. YOLOv8 efficiently handles the detection task, while the CNN model focuses exclusively on identifying the correct alphabet for each detected Braille character.

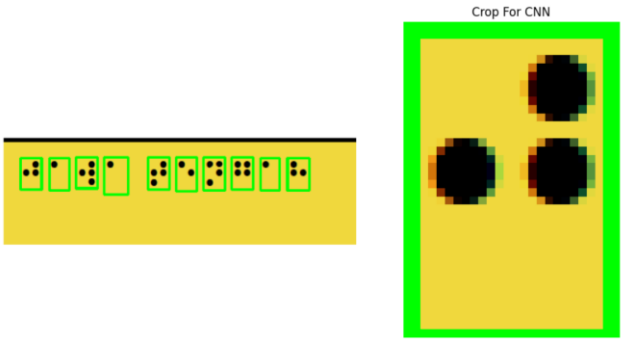

Figure 4 YOLOv8 detection of Braille characters (left) and example of a cropped Braille cell passed to the CNN classifier (right).

C. Character Classification Using CNN

Once the YOLOv8 detector identifies Braille characters, each bounding box is cropped to focus on individual Braille patterns. These cropped images are then processed by a Convolutional Neural Network (CNN) built on the MobileNetV2 framework. MobileNetV2 was selected because it's lightweight and performs well for visual classification, especially in real-time settings [15].

Model: "sequential_1"

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_160 (Functional)	(None, 5, 5, 1280)	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense_2 (Dense)	(None, 256)	327,936
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 26)	6,682

Total params: 2,592,602 (9.89 MB)
Trainable params: 334,618 (1.28 MB)
Non-trainable params: 2,257,984 (8.61 MB)

Figure 5 MobileNetV2-based CNN Model Architecture

The CNN takes preprocessed image patches of individual Braille characters as input, each labeled with its corresponding Latin letter (A–Z). The training dataset for the CNN includes 7,616 labeled images, sourced from the AEyeAlliance dataset and annotated images from Roboflow. To tackle the issue of fewer samples for letters T to Z, we applied data augmentation to the training set, using techniques like rotations between –6° and +6° and brightness tweaks from –8% to +8%. These changes boosted the number of samples and added variety to the dataset. The dataset was split into two parts: a training set with 6,103 images and a test set of 1,513 images. The test set was kept separate during training and used only at the end to check how well the model works on new data.

The MobileNetV2-based CNN was trained in two steps. In the first step, the base model was frozen and trained only the custom top layers for 20 epochs. This transfer learning method helped the model pick up Braille-specific features while keeping the general knowledge it gained from the ImageNet dataset, making the initial training smooth and efficient [16]. In the second step, all layers were unfroze and fine-tuned the model for 40 more epochs with a lower learning rate to carefully adjust the weights for Braille patterns, improving accuracy and flexibility [17]. This fine-tuning stage was carried out with a low learning rate to gradually adapt the pre-trained weights to the specific Braille features, improving accuracy and generalization [18], [19]. This two-step training strategy is commonly used in transfer learning pipelines and was crucial in achieving high performance for real-time Braille classification. This classification step complements the detection results from YOLOv8 and enables accurate, real-time translation of Braille patterns into Latin characters. The modular design of this two-stage system also allows future enhancements, such as integration with text-to-speech systems or deployment on mobile platforms.

D. Post-processing and Sorting

Once the Braille characters are successfully detected and classified, a post-processing step is applied to refine the output sequence. Each prediction from the YOLOv8 detection model includes a bounding box and a corresponding confidence score. However, YOLO's detection results are not inherently sorted in reading order. Therefore, additional processing is required to reconstruct the correct text sequence based on spatial information. In this step, all bounding boxes are first filtered using a confidence threshold (typically above 0.5) to remove false positives. The remaining bounding boxes are then sorted from left to right based on the x-coordinate of their top-left corner. This assumes that the Braille text is arranged in a single horizontal line, which is a reasonable assumption for many real-time use cases such as reading printed Braille labels or books. After sorting, the class labels from the CNN model are compiled into a single string that represents the predicted text. This step ensures that the order of characters reflects the actual reading direction of the original Braille text. The

combination of spatial sorting and CNN classification completes the translation from Braille dots into readable Latin characters. This modular approach—separating detection, classification, and sorting, allows for high flexibility in deployment and easier debugging of errors in individual components.

E. Real-time Visualization via OpenCV

To provide real-time feedback to users, the final system integrates OpenCV for video capture, image processing, and display. Unlike earlier implementations that used Tkinter as the interface layer, OpenCV was selected in this study for its superior performance in handling real-time image frames and its compatibility with camera-based computer vision applications [20]. In the implemented pipeline, the webcam continuously captures frames at a fixed resolution. Each frame is processed using the YOLOv8 model to detect the presence of Braille characters. Once detected, the regions of interest (ROIs) are cropped and passed to the CNN classifier for alphabet prediction. The system then draws bounding boxes around the detected Braille regions, along with the predicted characters and their confidence scores.

OpenCV functions are used to overlay this information directly onto the video stream. The predicted text is also printed at the top of the frame, enabling immediate visual feedback for users. Additionally, the detected character sequence is updated dynamically in real time, giving users an intuitive understanding of the recognition results as they happen. This real-time visualization not only enhances the interactivity and usability of the system but also helps users understand how the Braille input is being interpreted by the model. The simplicity and responsiveness of OpenCV make it an ideal choice for deploying Braille recognition systems in resource-constrained environments, including educational tools or assistive mobile applications.

III. RESULTS AND DISCUSSION

This section presents the results obtained from the implementation of a real-time Braille recognition system using a two-stage deep learning architecture. The discussion includes the performance of the system on the test dataset, visual outputs from real-time testing, and error analysis.

A. Real-time Visualization via OpenCV

To evaluate the performance of the YOLOv8 model in detecting Braille characters, a series of training sessions were conducted and visualized using standard YOLOv8 training logs. The visualization of the training metrics is shown in Figure 6, which summarizes various loss components and evaluation metrics over 25 epochs.

The training loss curves consist of three key components:

- Box Loss: This measures the model's error in predicting bounding box coordinates. As seen in the train/box_loss and val/box_loss plots, both training and validation box loss gradually decreased, indicating that the model improved in localizing Braille character positions.

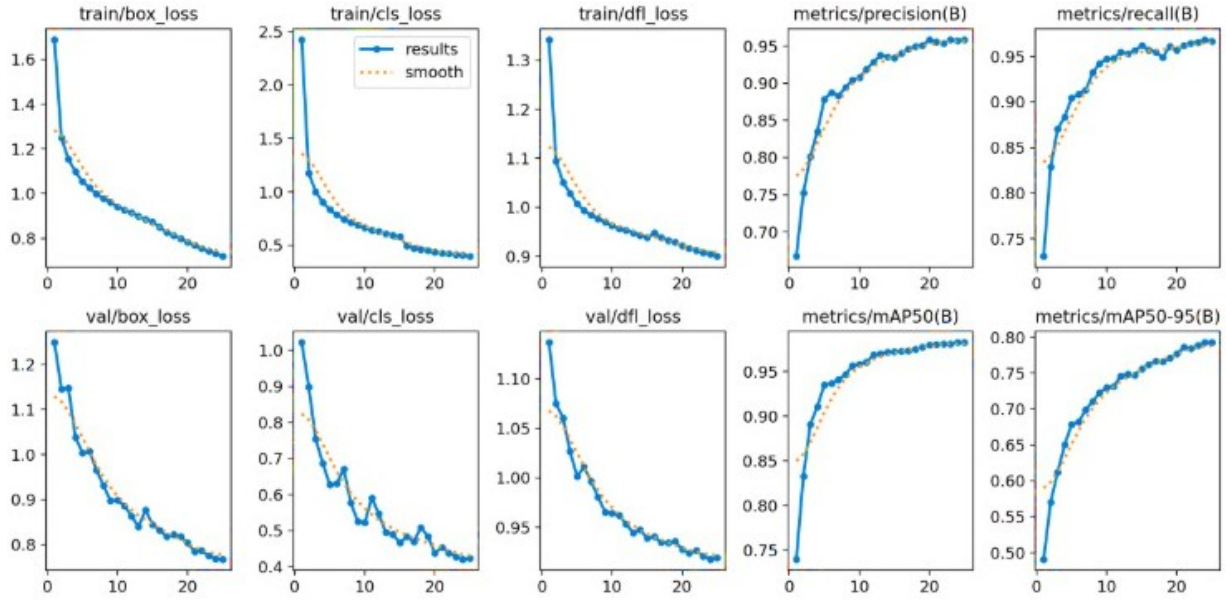

Figure 6 YOLOv8 Braille Detection Training Metrics

- Classification Loss (cls_loss): Shown in train/cls_loss and val/cls_loss, this reflects how accurately the model assigns the correct class (A-Z) to each detected object. The consistent decline in classification loss over epochs demonstrates better learning of class features.

- Distribution Focal Loss (dfl_loss): Displayed in train/dfl_loss and val/dfl_loss, this is a key component in YOLOv8's anchor-free architecture that improves the bounding box quality. The decreasing dfl_loss suggests that the model is producing tighter and more accurate bounding boxes for Braille cells.

To provide a clearer understanding of the model's effectiveness, performance metrics are also plotted for detailed evaluation

- Precision and Recall: metrics/precision(B) and metrics/recall(B) indicate the model's precision and recall across all Braille character classes. Precision increased steadily towards 0.95, while recall reached over 0.96, showing the model's ability to correctly identify most Braille characters while minimizing false positives.
- mAP@0.5 and mAP@0.5:0.95: These are key object detection benchmarks. metrics/mAP50(B) and metrics/mAP50-95(B) show that mean Average Precision (mAP) improved significantly, with mAP@0.5 reaching close to 0.97, and more-strict mAP@0.5:0.95 reaching around 0.85. This indicates that the model performed consistently well across various Intersection over Union (IoU) thresholds, confirming robustness and high detection quality [21].

Overall, the visualization confirms that the YOLOv8 model successfully converged and maintained low validation loss without significant overfitting. This result aligns well with the intended real-time detection goal, enabling accurate localization of Braille characters under diverse image conditions.

B. CNN Classification Results

To evaluate the training process and monitor the model's learning behaviour, the training and validation accuracy and loss were recorded at each epoch. The model was trained using a two-phase strategy consisting of an initial transfer learning stage, where only the classification head was trained, followed by a fine-tuning stage, where the entire MobileNetV2 backbone was unfrozen and optimized with a lower learning rate. The following graphs illustrate how the model's performance evolved during both phases of training.

Figure 7 shows the accuracy and loss trends across 60 epochs of training and fine-tuning for the MobileNetV2-based CNN model. The initial 20 epochs correspond to the transfer learning phase, where the base MobileNetV2 layers were frozen, and only the top layers were trained. During this phase, the model showed steady improvements in both training and validation accuracy, reaching approximately 75% validation accuracy by epoch 20. The validation loss, however, plateaued and showed minor fluctuations, indicating that the model was beginning to saturate in learning capacity with the frozen base layers [22]. The vertical dashed line marks the transition to the fine-tuning phase.

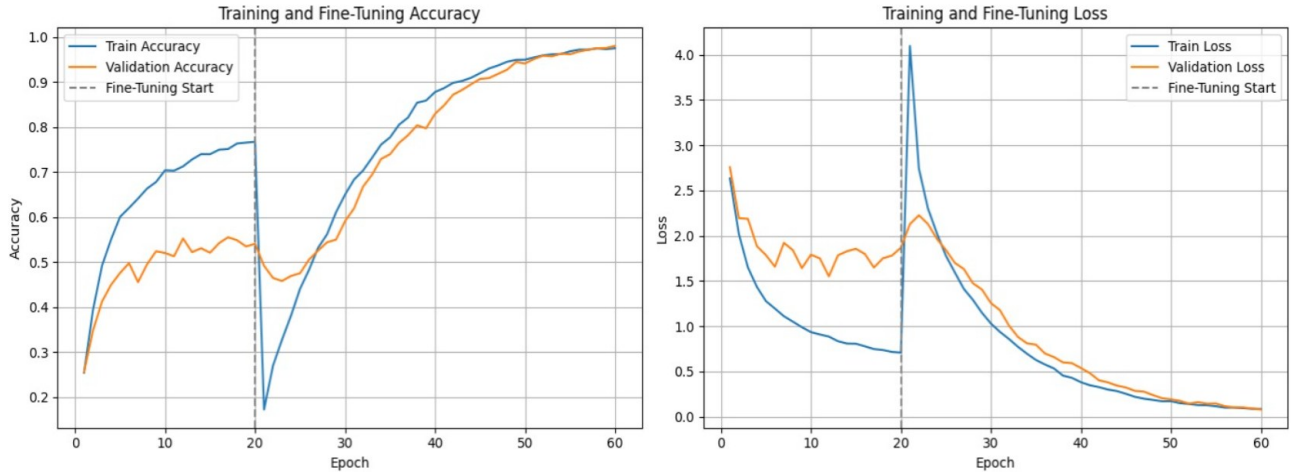

Figure 7 MobileNetV2 Transfer Learning and Fine-Tuning Metrics

Starting from epoch 21, all layers of the base model were unfrozen, and the entire network was fine-tuned with a significantly smaller learning rate. A temporary dip in training accuracy and a spike in training loss can be observed immediately after unfreezing the layers. This is a common phenomenon as the model adjusts to the new training regime. However, the model quickly recovered and continued to improve.

From epoch 25 onwards, the model demonstrated rapid convergence, with training and validation accuracy steadily increasing and loss values consistently decreasing. By the end of training, the model achieved a validation accuracy of over 95%, and both training and validation loss dropped below 0.2, indicating excellent generalization performance and minimal overfitting. This result confirms the effectiveness of a two-phase training approach (transfer

learning followed by fine-tuning) for achieving high-performance Braille character classification. For the classification report, the process was conducted using a total of 7616 images, each representing one of the 26 Braille alphabet characters. These datasets were carefully cleaned, merged, and oversampled particularly for classes T through Z which originally had significantly fewer samples compared to classes A through S.

Table 1 shows the detailed classification metrics for each class. The Braille recognition system demonstrated a high level of accuracy and consistency across all 26 alphabet classes. The trained CNN classifier based on MobileNetV2 achieved a test accuracy of 98%, which is a strong indicator of its generalization ability to classified the majority of the 7616 samples. In this study, accuracy is calculated as the proportion of correct predictions out of all predictions made, using the formula [23]:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

Where:
TP = true positives,
TN = true negatives,
FP = false positives,
FN = false negatives

Precision measures the fraction of correct positive predictions among all positive predictions, expressed as:

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

Recall, sometimes called sensitivity, is the share of correct positive predictions out of all actual positive cases, given by:

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

F1-score is the harmonic average of precision and recall, balancing both metrics, and is calculated as:

$$F1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (4)$$

The macro average and weighted average for precision, recall, and F1-score were all at 0.98, which suggests strong performance not only on well-represented classes but also on minority ones. This consistency highlights the success of the data augmentation and oversampling strategies applied to balance the dataset. Most letters achieved precision and recall scores above 0.95. Notably, several classes like V and K even achieved perfect precision, meaning the model made no false predictions or missed instances in those categories. Nevertheless, the overall evaluation metrics reflect the robustness and reliability of the proposed YOLOv8 and MobileNetV2 pipeline for real-time Braille character recognition.

TABLE I
MOBILENETV2 A-Z CLASSIFICATION REPORT

Class	Precision	Recall	F1-Score	Support
A	0.98	1.00	0.99	286
B	0.98	0.97	0.97	252
C	0.99	0.99	0.99	268
D	0.99	0.99	0.99	304
E	0.99	0.99	0.99	361
F	0.97	0.99	0.98	291
G	1.00	0.98	0.99	302
H	0.97	0.99	0.98	312
I	1.00	0.98	0.99	302
J	0.98	0.96	0.97	278
K	0.98	0.99	0.99	283
L	0.99	0.98	0.99	318
M	0.98	0.98	0.98	295
N	0.98	0.98	0.98	312
O	0.97	0.99	0.98	312
P	0.99	0.99	0.99	307
Q	0.99	1.00	0.99	288
R	0.99	0.97	0.98	328
S	0.99	0.98	0.98	288
T	0.99	0.98	0.98	285
U	0.98	0.99	0.99	279
V	0.99	0.99	0.99	272
W	0.97	0.97	0.97	275
X	1.00	1.00	1.00	272
Y	0.99	0.98	0.99	275
Z	0.99	0.97	0.98	271
Overall Accuracy			0.98	7616
Macro Avg	0.98	0.98	0.98	7616
Weighted Avg	0.98	0.98	0.98	7616

C. *Visualization and Real-Time Implementation*

To evaluate how the system would perform in a real-life environment, a real-time testing setup was developed using OpenCV and a webcam as the input source. YOLOv8 was used to detect Braille cells from each frame, and the detected regions were cropped and resized before being passed to the CNN classifier.

An essential aspect of the visualization pipeline was ensuring that duplicate detections or overlapping boxes were filtered properly. To address this, post-processing included sorting all bounding boxes based on their x-axis coordinates, so that the predicted text matched the left-to-right order of reading. Any overlapping or duplicate predictions were discarded based on a custom-defined IoU threshold and minimum distance rule.

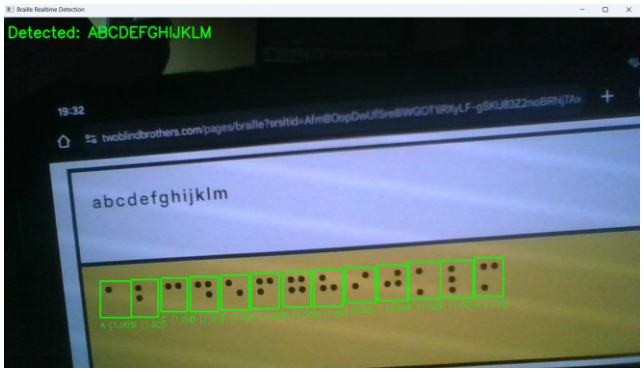

Figure 8 Real Time Braille Recognition A – M

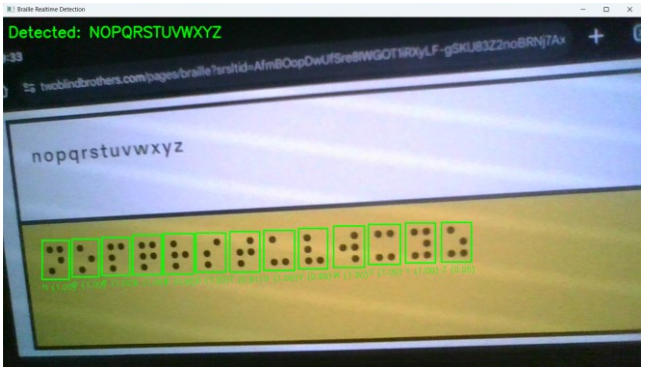

Figure 9 Real Time Braille Recognition N – Z

The system was tested using segmented groups of Braille characters to evaluate its classification performance across the full alphabet. Figure 8 illustrates the detection and recognition results for Braille characters A to M, while Figure 9 shows the corresponding predictions for characters N to Z. In both cases, the system accurately localized each Braille cell and classified it correctly using the MobileNetV2 model, with all confidence scores exceeding 0.95. These results confirm the model's strong generalization across different classes, regardless of shape similarity or character frequency. To further assess real-world usability, a real-time recognition test was conducted using a webcam directed at Braille characters displayed on a screen. As presented in Figure 10, the YOLOv8 model successfully localized each Braille character with bounding boxes, and the cropped regions were passed into the MobileNetV2 classifier. The predictions were then combined and displayed live on the interface as readable Latin letters. In this particulars example, the system correctly recognized the full sequence representing the word "BRAILLERECOGNITION", with each prediction yielding a confidence score greater than 0.95.

This real-time processing capability highlights a major strength of the proposed system compared to previous works, which largely focused on static image uploads or web-based interaction. In contrast, the proposed pipeline operates directly from camera input without requiring preprocessing such as scanning or manual cropping. Furthermore, the system maintained reliable performance even under moderate lighting noise, screen glare, and slight changes in camera angles further validating its suitability for practical assistive technologies. The average frame rate achieved during testing ranged between 5–7 FPS on a mid-range laptop equipped with a GPU, allowing users to move Braille text in front of the camera and receive real-time character predictions almost instantaneously.

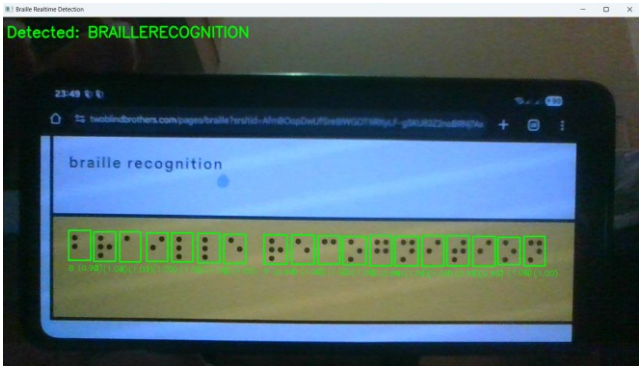

Figure 10 Example of Real Time Braille Recognition

D. Error Analysis and Misclassification Trends

Despite the system achieving high overall accuracy, several misclassifications were still observed during real-time testing. These errors were mostly caused by real-world factors that affected the quality of both detection and classification. One common issue was poor lighting conditions, where low illumination made it difficult for the system to distinguish the raised Braille dots from the background, especially on embossed paper. In such scenarios, the bounding boxes from YOLOv8 tended to shift or miss certain Braille cells. Another major challenge was motion blur, typically caused by the user's hand moving the Braille sheet too quickly in front of the camera. This often resulted in blurry frames, making it harder for the detection model to extract accurate features. Additionally, partial occlusion was a frequent source of error, where fingers or the edge of the camera frame blocked part of a Braille character, leading to incomplete information for classification.

The confusion matrix further revealed common misclassifications between Braille characters with very similar dot patterns, such as between letters J, R and W. This is understandable in dense classification tasks like Braille, where a difference of only one dot can change the character's identity. Lastly, the limitations of the webcam hardware also contributed to detection errors. Low-resolution or low-frame-rate cameras reduced the model's ability to capture fine details. Inconsistent hand stability during usage also

affected detection, especially in handheld or non-mounted scenarios. These limitations point to opportunities for future improvements, such as integrating adaptive lighting correction, improving camera quality, or adding motion stabilization features to enhance system robustness in practical environments.

E. Comparison with Single-Stage Approaches

To highlight why our two-stage method (YOLOv8 for detecting Braille spots and MobileNetV2 for classifying them) stands out, we looked at it next to single-stage options, where one model handles both spotting and labeling in a single pass. For instance, we tried a single-stage version of YOLOv8 that does everything together, and it only hit an mAP@0.5 of about 0.92 in our early tests—decent but not as strong as our 0.97 with the split setup. The issue with single-stage is it can get tripped up on Braille letters that look almost the same, like J, R, and W, which differ by just one dot, leading to more mix-ups in the labeling part. By breaking it into two steps, we can fine-tune each one separately, pushing the overall classification up to 98% on our test data. It's also more adaptable—if we need to swap in a lighter classifier for phones, we can do that without messing with the detection side.

We also compared our work to a previous study by Ovodov (2021) [7], who used a single-stage RetinaNet model to spot and recognize whole Braille characters at once. Their approach worked well on tough photos from phones, like curved pages or angled shots, and scored a super high F1 of 0.9981 at the character level on their Angelina dataset (240 real-world images). That's impressive for handling distortions without extra grid fixing steps. But our two-stage system adds an edge by using MobileNetV2 just for classification after YOLOv8's detection, which helps with similar patterns and gives us room to optimize for speed on everyday hardware. Plus, while their method takes about 0.18 seconds per image on a strong GPU, our setup runs at 5-7 FPS on a mid-range laptop, making it more practical for live webcam use. Overall, the split design in our method offers better flexibility and accuracy for real-time Braille reading, especially when letters are easy to confuse. Studies on YOLO setups [14] support this, noting that two-stage often shines for tasks with detailed, overlapping patterns.

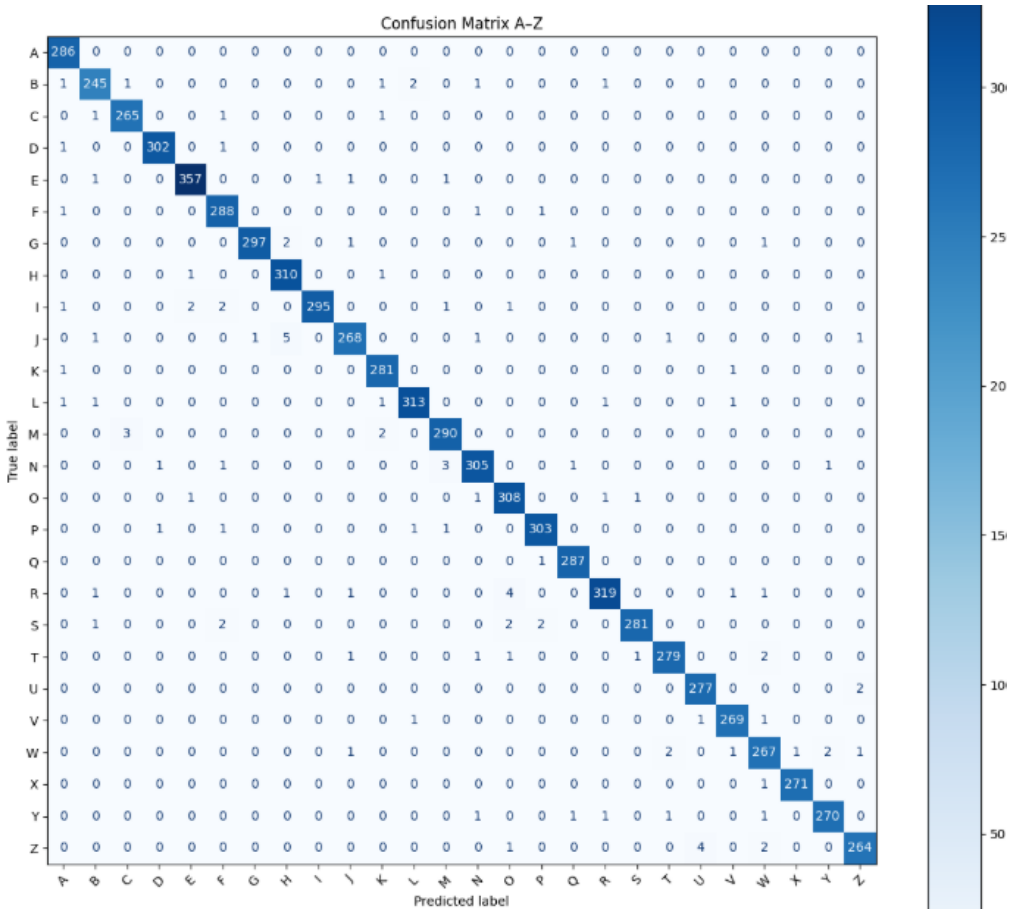

Figure 11 MobileNetV2 A-Z Confusion Matrix

IV. CONCLUSION

This study successfully implemented a two-stage deep learning pipeline for real-time Braille character recognition. The system combined YOLOv8 for efficient object detection of Braille patterns and MobileNetV2 as the CNN-based classifier to identify individual characters from detected regions. Through careful preprocessing, data augmentation, and oversampling, the system was trained to handle imbalanced class distributions and variations in image quality. The experimental results demonstrated that the proposed approach achieved strong classification performance with an overall accuracy of 98% and high precision, recall, and F1-scores across all 26 alphabet classes. The system proved capable of operating in real-time using a webcam, providing a practical solution for translating Braille into Latin characters for assistive technology applications. Although the system achieved high performance, several challenges remain, such as improving robustness under low-light conditions, motion blur, and partial occlusions. Future work could explore integrating more advanced noise handling, testing on more diverse real-world datasets, and optimizing the system for mobile deployment to further enhance accessibility for the visually impaired.

REFERENCES

[1] B. M. Hsu, "Braille recognition for reducing asymmetric communication between the blind and non-blind," *Symmetry Basel*, vol. 12, no. 7, Jul. 2020, doi: 10.3390/SYM12071069.

[2] D. Lee and J. Cho, "Automatic Object Detection Algorithm-Based Braille Image Generation System for the Recognition of Real-Life Obstacles for Visually Impaired People," *Sensors*, vol. 22, no. 4, Feb. 2022, doi: 10.3390/s22041601.

[3] S. Shokat, R. Riaz, S. S. Rizvi, K. Khan, F. Riaz, and S. J. Kwon, "Analysis and Evaluation of Braille to Text Conversion Methods," *Mob. Inf. Syst.*, vol. 2020, 2020, doi: 10.1155/2020/3461651.

[4] G. Dzięgiel-Fivet, J. Plewko, M. Szczerbiński, A. Marchewka, M. Szwed, and K. Jednoróg, "Neural network for Braille reading and the speech-reading convergence in the blind: Similarities and differences to visual reading," *Neuroimage*, vol. 231, May 2021, doi: 10.1016/j.neuroimage.2021.117851.

[5] L. Alzubaidi and others, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *J. Big Data*, vol. 8, no. 1, pp. 1–74, Dec. 2021, doi: 10.1186/s40537-021-00444-8.

[6] S. Bej, N. Davtyan, M. Wolfien, M. Nassar, and O. Wolkenhauer, "LoRAS: an oversampling approach for imbalanced datasets," *Mach. Learn.*, vol. 110, no. 2, pp. 279–301, Feb. 2021, doi: 10.1007/s10994-020-05913-4.

[7] I. G. Ovodov and R. Zelenograd, "Optical Braille Recognition Using Object Detection Neural Network." Oct. 2021. doi: 10.48550/arXiv.2012.12412.

[8] T. Kausar, S. Manzoor, A. Kausar, Y. Lu, M. Wasif, and M. A. Ashraf, "Deep Learning Strategy for Braille Character Recognition," *IEEE Access*, vol. 9, pp. 169357–169371, 2021, doi: 10.1109/ACCESS.2021.3138240.

[9] M. Akay and others, "Deep Learning Classification of Systemic Sclerosis Skin Using the MobileNetV2 Model," *IEEE Open J. Eng. Med. Biol.*, vol. 2, pp. 104–110, 2021, doi: 10.1109/OJEMB.2021.3066097.

[10] A. Sharma, V. Kumar, and L. Longchamps, "Comparative performance of YOLOv8, YOLOv9, YOLOv10, YOLOv11 and Faster R-CNN models for detection of multiple weed species," *Smart Agric. Technol.*, vol. 9, Dec. 2024, doi: 10.1016/j.atech.2024.100648.

[11] K. Maharana, S. Mondal, and B. Nemade, "A review: Data pre-processing and data augmentation techniques," *Glob. Transit. Proc.*, vol. 3, no. 1, pp. 91–99, Jun. 2022, doi: 10.1016/j.gltp.2022.04.020.

[12] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine Learning with Oversampling and Undersampling Techniques: Overview Study and Experimental Results," in *2020 11th International Conference on Information and Communication Systems (ICICS)*, Apr. 2020, pp. 243–248. doi: 10.1109/ICICS49469.2020.239556.

[13] Z. J. Khow, Y. F. Tan, H. A. Karim, and H. A. A. Rashid, "Improved YOLOv8 Model for a Comprehensive Approach to Object Detection and Distance Estimation," *IEEE Access*, vol. 12, pp. 63754–63767, 2024, doi: 10.1109/ACCESS.2024.3396224.

[14] N. Jegham, C. Y. Koh, M. Abdelatti, and A. Hendawi, "YOLO Evolution: A Comprehensive Benchmark and Architectural Review of YOLOv12, YOLO11, and Their Previous Versions," Mar. 17, 2025, *arXiv*: arXiv:2411.00201. doi: 10.48550/arXiv.2411.00201.

[15] R. Indraswari, R. Rokhana, and W. Herulambang, "Melanoma image classification based on MobileNetV2 network," in *Procedia Computer Science*, 2021, pp. 198–207. doi: 10.1016/j.procs.2021.12.132.

[16] F. Zhuang and others, "A Comprehensive Survey on Transfer Learning." Jun. 2020.

[17] Y. Gulzar, "Fruit Image Classification Model Based on MobileNetV2 with Deep Transfer Learning Technique," *Sustain. Switz.*, vol. 15, no. 3, Feb. 2023, doi: 10.3390/su15031906.

[18] Z. Zhu, K. Lin, A. K. Jain, and J. Zhou, "Transfer Learning in Deep Reinforcement Learning: A Survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 11, pp. 13344–13362, Nov. 2023, doi: 10.1109/TPAMI.2023.3292075.

[19] E. C. Too, L. Yujian, S. Njuki, and L. Yingchun, "A comparative study of fine-tuning deep learning models for plant disease identification," *Comput. Electron. Agric.*, vol. 161, pp. 272–279, Jun. 2019, doi: 10.1016/j.compag.2018.03.032.

[20] S. Mishra, M. V. Verma, D. N. Akhtar, S. Chaturvedi, and D. Y. Perwej, "An Intelligent Motion Detection Using OpenCV," *Int. J. Sci. Res. Sci. Eng. Technol.*, pp. 51–63, Mar. 2022, doi: 10.32628/ijsrset22925.

[21] M. Vilar-Andreu, L. Garcia, A. J. Garcia-Sanchez, R. Asorey-Cacheda, and J. Garcia-Haro, "Enhancing Precision Agriculture Pest Control: A Generalized Deep Learning Approach With YOLOv8-Based Insect Detection," *IEEE Access*, vol. 12, pp. 84420–84434, 2024, doi: 10.1109/ACCESS.2024.3413979.

[22] M. Iman, H. R. Arabnia, and K. Rasheed, "A Review of Deep Transfer Learning and Recent Advancements," *Technologies*, Apr. 2023, doi: 10.3390/technologies11020040.

[23] M. Grandini, E. Bagli, and G. Visani, "Metrics for Multi-Class Classification: an Overview." Aug. 2020.