Deep Learning Methods for ArUco Marker Detection and Classification Under Challenging Distortions

Filip Hanuš,
School of Engineering, College of Art, Technology and Environment,
University of the West of England, Bristol, UK
Email: filip2.hanus@uwe.ac.uk

***Abstract*—This project investigates deep learning methodologies for detecting and classifying ArUco markers in the presence of severe distortions such as blur, noise, and rotations. The project addresses two primary challenges: marker classification under varying distortion levels and accurate localisation in images. For classification, several architectures were evaluated, with GoogLeNet achieving 100% accuracy on weakly distorted markers and 97.48% under severe distortions. For detection, RetinaNet-ResNet50 showed performance at 89.00% IoU in standard conditions and 85.64% under severe distortions.**

***Index Terms*—ArUco Markers, Deep Learning, Computer Vision, Object Detection, Image Classification, Fiducial Markers**

I. Introduction

ArUco markers (Figure 1) are square visual codes designed for efficient detection and identification using computer vision algorithms. They are widely used in applications such as robotics, virtual reality, and industrial automation for positioning, tracking, and navigation tasks. Examples of these markers in use are shown in Figure 2.

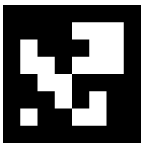
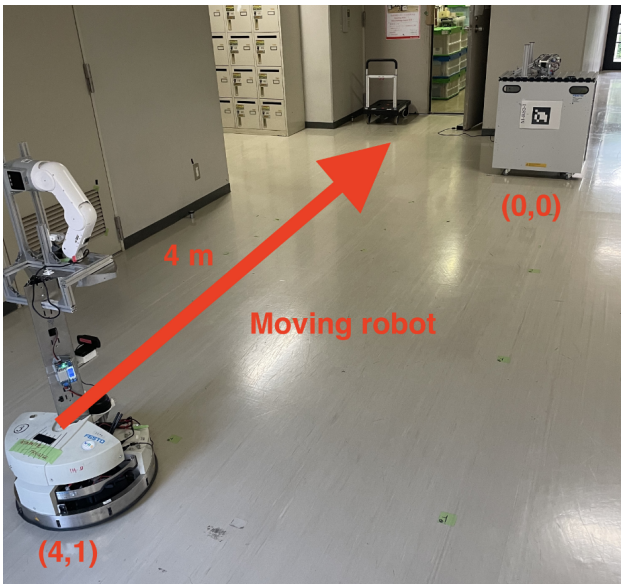


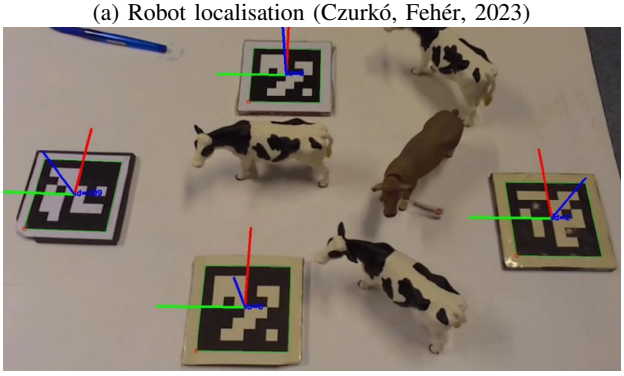
Fig. 1: Basic ArUco marker example

However, as noted by Zakiev et al. (2020) and Romero-Ramirez, Muñoz-Salinas, Medina-Carnicer (2021), traditional detection methods perform well under controlled conditions but struggle with real-world challenges such as rotation, blur, and noise. This project investigates deep learning approaches to address these limitations. The key objectives are:

1) Classifying ArUco markers under challenging conditions (blur, noise, and rotation).
2) Detecting and tracking marker positions in noisy or distorted environments.



(a) Robot localisation (Czurkó, Fehér, 2023)



(b) Drone localisation (Nakajima et al., 2024)

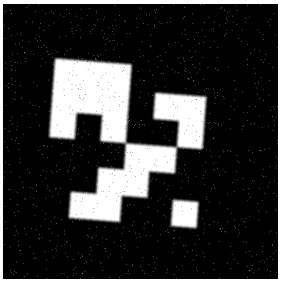
Fig. 2: Examples of ArUco markers in different conditions

II. Classification

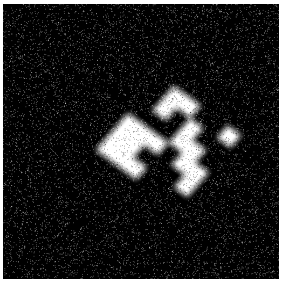
A. Classification Methodology

The classification dataset examples are visible in Figure 3. The dataset contains 100 variations/classes of the ArUco marker. Developing a classification model was divided into the following steps:

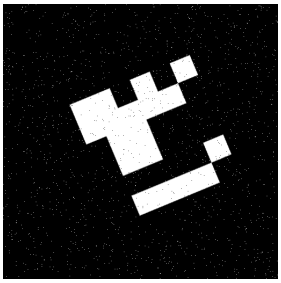
1) Create a data visualisation tool for datasets and results.
2) Develop a data augmentation pipeline to expand the dataset.
3) Prepare and implement suitable classification architectures.



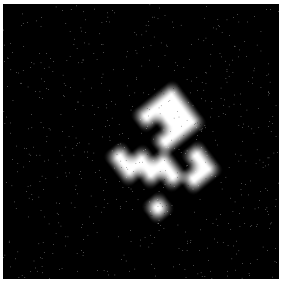
(a) Class 0 (out of 100) marker example 1



(b) Class 0 (out of 100) marker example 2



(c) Class 1 (out of 100) marker example 1



(d) Class 1 (out of 100) marker example 2

Fig. 3: Examples of ArUco marker classification under different conditions (from the provided dataset)

4) Design an experiment to evaluate architectures and parameters.
5) Train and test the chosen classification model.

1) Data Preparation: Firstly, a dataset browser was created to understand the data. This tool provided insight into image characteristics, pixel distribution, and basic dataset statistics. It was also designed to visualise both datasets and results, as demonstrated in Figure 4.

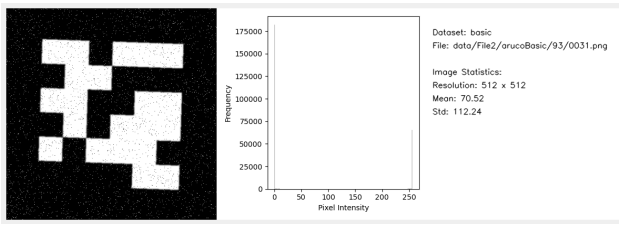


Fig. 4: Dataset browser used with the File2 dataset

A data augmentation pipeline was developed to generate a larger, more robust dataset. Through trial and error, augmentation parameters were optimised to mimic the distortions observed in File3. The pipeline expanded the original dataset (100 images in File1, each representing a different ArUco marker) to 1,000 images per class at a resolution of 512x512 pixels. Augmentations, including rotation, blurring, and noise, were configured via a central configuration file, allowing for flexibility without needing to modify the codebase. The pipeline workflow is summarised below:

Algorithm 1 Data Augmentation Pipeline

List all of the images in the File1 dataset
while size of augmentations list $<$ target dataset size **do**
  Select random rotation, between 0 and 360 degrees
  Select random scaling, between 50% and 100% of original size
  Select random blur, between 0 and 14 sigma
  Select random noise, between 0% and 10% of pixels
  Append the set of augmentations to the list
end while
for each image in File1 **do**
  Create folder named after the image's number (0-99)
  Read the raw image
  for each set in the list of augmentations **do**
    Apply the augmentations to the image
    Save the augmented image to the new folder
  end for
end for

Fig. 5: Classification training workflow

2) Network Architectures: Several classification architectures were implemented using PyTorch, starting with a basic convolutional neural network (CNN) to establish baseline performance. Following this, pre-trained and non-pre-trained variants of AlexNet were tested. ResNet18 and GoogLeNet were also evaluated, leveraging PyTorch documentation and guidelines PyTorch (2024d). The codebase was modular, allowing seamless integration of new models into experiments.

The following architectures were selected for initial experimentation:

- **MinimalCNN**: A lightweight custom architecture with:
  - Three convolutional blocks (32→64→128 features)
  - Batch normalisation and dropout for regularisation
  - Global average pooling for dimension reduction
- **AlexNet Variants**:
  - Clean implementation without pre-training
  - Version with pre-trained features
- **ResNet18**: Pre-trained implementation
- **GoogLeNet**: Pre-trained implementation

3) Training Configuration: The experiments tested combinations of the above architectures with varied parameters:

- **Datasets**: File3 (100 images per class, significant distortions).
- **Models**: MinimalCNN, AlexNet (clean), AlexNet, ResNet18, GoogLeNet.
- **Batch Sizes**: 32, 64.
- **Training Transformations**: None, random rotations, and combinations of rotations, blur, and noise.

This resulted in 30 parameter/model combinations, trained over nearly 8 hours on a single GPU. Customisation required minimal effort, achieved via a central configuration file. Adjustments could be made to also trial and compare

other combinations with datasets, batch sizes, learning rates, transformations, and models, although most of these were not tested due to GPU memory constraints. The selected learning rate and training duration were chosen during the initial experiments throught the development, this can be seen in some of the past GitHub commits.

- **Loss Function**: Cross-entropy
- **Optimizer**: Adam with learning rate 3e-4
- **Training Duration**: 50 epochs with early stopping at 96% training accuracy

Once training was complete, results were plotted and analysed. Full results are detailed in the appendices and results directory. The final classification model was trained on the custom dataset with the following optimised parameters:

- **Datasets**: Custom dataset (1000 images per class)
- **Model**: GoogLeNet
- **Batch Sizes**: 32
- **Additional Random Training Transformations**: None

4) Method Implementation: The experiment implementation followed this algorithmic workflow:

Algorithm 2 Classification Experiment Pipeline

```
Load configuration for dataset, model, batch size, and transformations
for each combination in configurations do
    Load dataset
    Setup results directory
    Initialise model and transforms
    Create data loaders
    Initialise loss function, optimiser
    Train and validate model for number of epochs
    Plot the training progress
    Run evaluation on test set
    Plot classification metrics
    Save all results
end for
```

Fig. 6: Classification experiment workflow

The model training loop is shown below:

Algorithm 3 Classification Training and Validation Pipeline

```
for each epoch in total number of epochs do
    Set model to train mode
    for each batch in train loader do
        Reset gradients
        Forward pass through network
        Calculate cross-entropy loss
        Backpropagate and update weights
        Add loss and accuracy to totals
    end for
    Calculate average training loss and accuracy
    Set model to evaluation mode
    for each batch in validation loader do
        Forward pass through network
        Calculate cross-entropy loss
        Add loss and accuracy to totals
    end for
    Calculate average validation loss and accuracy
    Save model if the best validation accuracy was achieved
    Check for early stopping criteria
end for
```

Fig. 7: Classification training workflow

After training, the model was tested using the original datasets (File2 and File3) to evaluate its generalisation capabilities without seeing the originally provided data. Test-only data loaders were created for this purpose, and the model performance on these datasets was separately recorded.

B. Classification Results

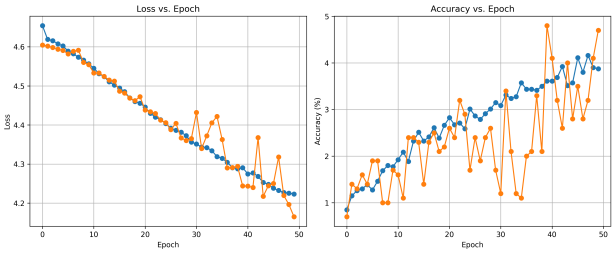
The classification experiments were conducted in two phases. Initially, multiple architectures and configurations were tested on the File3 dataset to determine the optimal approach. Subsequently, the best performing configuration was trained on a custom dataset and was evaluated against separately File2 and File3 datasets.

1) Initial Experiments: Various architectures with different batch sizes and data augmentation strategies were explored. Each configuration was tested on 1000 samples from File3, with the following (best of) results. The full results of each combination can be seen in the repository.

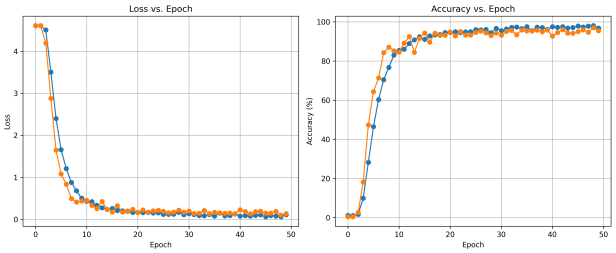
- **MinimalCNN**:
  - Batch size 32, full (training time) augmentation: 5.20% accuracy (F1: 3.62%)
- **AlexNet (Clean Implementation)**:
  - Batch size 64, rotation (training time) augmentation: 98.60% accuracy (F1: 98.51%)
- **AlexNet (Pre-trained)**:
  - Batch size 64, full (training time) augmentation: 73.70% accuracy (F1: 72.15%)
- **ResNet18**:

  - Batch size 32, no (training time) augmentation: 100.00% accuracy (F1: 100.00%)
- **GoogLeNet**:
  - Batch size 32, no (training time) augmentation: 99.90% accuracy (F1: 99.87%)

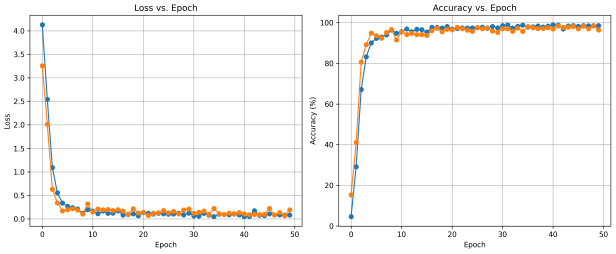
The experiment results are shown in the figure 8.



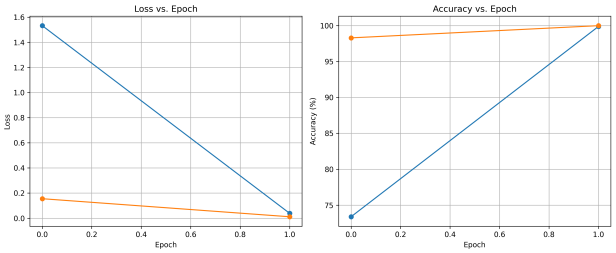
(a) MinimalCNN training progress



(b) AlexNet (clean) training progress



(c) AlexNet (pre-trained) training progress



(d) ResNet18 training progress

Fig. 8: Training progress for different model architectures showing training and validation accuracy over epochs (orange plot is validation accuracy, blue is training accuracy)

The progress of models trained with a batch size of 32 and no data augmentation for comparability is shown in all training plots. The minimal CNN was observed to behave as anticipated, although better performance would likely be achieved with additional epochs.

AlexNet with no pre-trained weights was observed to perform very well, though it converged more slowly than its pre-trained counterpart. ResNet18 and GoogLeNet were found to have performed almost perfectly and were noticeably better without additional training transforms. Out of GoogLeNet and ResNet18, the ResNet plot is shown, as the results were essentially identical. Early stopping was triggered for both models once 99% validation accuracy was reached.

For further insights, training logs and detailed analysis are provided in the appendices, which also contain additional plots, results, and data.

2) Final Model Performance: Based on the previous findings, GoogLeNet was chosen as the final model because of its consistent performance across different parameter settings. Training was conducted on FileCustom1, a synthetic dataset created to approximate conditions in Files 2 and 3. The training set size was thereby increased from 9,000 to 95,000 samples.

The following results were provided by the final evaluation:

- **Training Evaluation** (2,500 samples):
  - Accuracy: 100.00%
  - Precision: 100.00%
  - Recall: 100.00%
  - F1-score: 100.00%
- **File2 Evaluation** (10,000 samples):
  - Accuracy: 100.00%
  - Precision: 100.00%
  - Recall: 100.00%
  - F1-score: 100.00%
- **File3 Evaluation** (10,000 samples):
  - Accuracy: 97.48%
  - Precision: 97.68%
  - Recall: 97.48%
  - F1-score: 97.51%

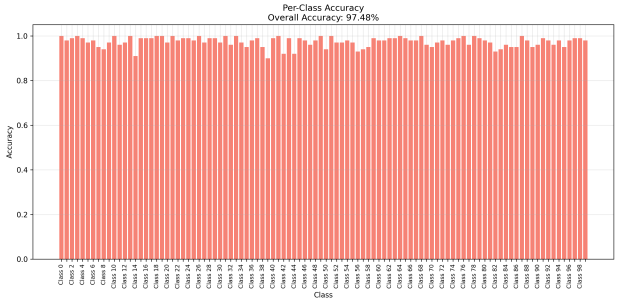
Near-perfect performance (as shown in Figure 9) under various distortion conditions is indicated by these results. Perfect scores were achieved on File2, suggesting that weak distortions were handled effectively, whereas a strong outcome on File3 (only 252 misclassifications out of 10,000 samples) demonstrates good resilience in more challenging scenarios. Based on the per-class accuracy, precision, and recall plots, marker 33 was identified as the most difficult class, as its precision was lowest, while markers 14 and 39 were missed most often because of the lowest recall. A clearer depiction of these findings is provided by the confusion matrix in the appendices.

III. Detection

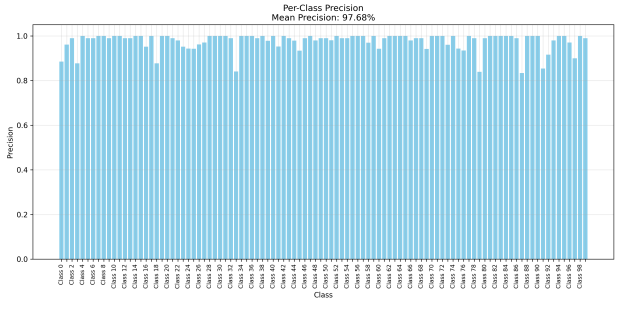
A. Detection Methodology

Figure 10 shows detection dataset examples. One hundred ArUco placements are included, alongside a CSV listing each marker's position.

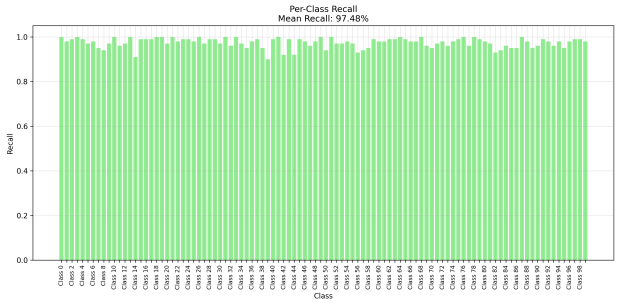
The detection problem was approached similarly to classification with these steps:



(a) Per-class accuracy analysis



(b) Per-class precision analysis



(c) Per-class recall analysis

Fig. 9: Final model performance analysis on File3 dataset showing per-class metrics

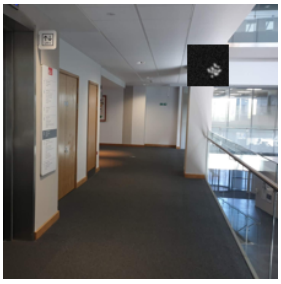
1) The data plotter was adapted for detection data and results.
2) A data augmentation script was created for office images with ArUco markers.
3) Detection network architectures were prepared.
4) An experiment was designed to compare detection architectures and settings.
5) The final detection model was trained and tested.

1) Data Preparation: Firstly, the dataset browser was adapted for detection data viewing and results. It displays the image, bounding box, pixel distribution, and basic dataset statistics. This is visible in Figure 11.

After the data were analysed, a data augmentation script was developed to expand the training dataset, as the provided one was insufficient compared to classification. The data browser was used to identify an optimal approach.

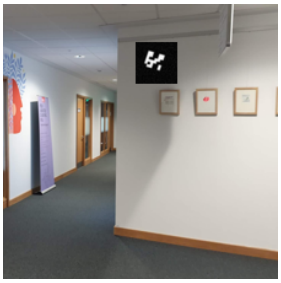
The data augmentation pipeline is outlined below:

(a) Marker in office placement example 1



(b) Marker in office placement example 2

(c) Marker in office placement example 3



(d) Marker in office placement example 4

Fig. 10: Examples of ArUco marker placements (from the provided dataset)

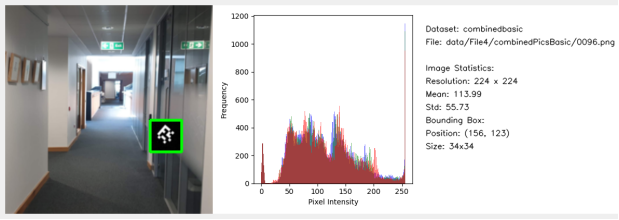


Fig. 11: Dataset browser used with the File4 dataset

Algorithm 4 Detection Data Preparation Pipeline

```
Load office images and ArUco tags
for each office image do
    Get random 1000x1000 crop from original image
    Resize cropped image to 224x224
    for each ArUco class do
        Select random tag from custom dataset 1
        Resize tag to 34x34
        Calculate valid random position
        Place tag at position
        Save combined image
        Store bounding box coordinates
    end for
end for
Save bounding box annotations to CSV
```

Fig. 12: Detection data preparation workflow

2) Network Architectures: Multiple pre-trained PyTorch architectures were implemented. RetinaNet and MobileNet worked out of the box, whereas others were problematic (YOLOv3) and later abandoned. Final models were based on PyTorch (2024c), PyTorch (2024a), and PyTorch (2024b), chosen for performance and easy implementation.

- **RetinaNet with ResNet50 backbone**: ∼40M parameters
  - Pre-trained on ImageNet dataset for general object recognition (PyTorch, 2024c)
- **Faster R-CNN with ResNet50 backbone**: ∼38M parameters
  - Pre-trained on ImageNet dataset for general object recognition
  - Two-step detection: first finds regions of interest, then classifies them (PyTorch, 2024a)
- **MobileNetV3-Large with FPN**: ∼20M parameters
  - Pre-trained on ImageNet dataset for general object recognition
  - Originally optimised for mobile/edge devices with fewer parameters (PyTorch, 2024b)

3) Training Configuration: Similarly to classification, the detection training pipeline included:

- **Datasets**: File5 (provided dataset with 100 images of the office with tags with strong distortions to test the detection models)
- **Models**: RetinaNet-ResNet50, FasterRCNN-ResNet50, MobileNetV3-Large-FPN
- **Batch Sizes**: 4
- **Additional Random Training Transformations**: None, this is not to distort the office images, since the tags are already distorted

Only three distinct networks and result sets were produced. With one configuration file change, the entire experiment can be altered: the custom dataset can be included (increasing epoch time), different batch sizes can be used (potentially exceeding GPU memory), and varied models can be tested.

Additional settings and parameters were used solely to reduce experiment runtime:

- **Optimizer**: Adam with learning rate 3e-4
- **Training Duration**: 100 epochs with early stopping at 98% training accuracy

After the experiments, results were plotted and analysed. They are presented in the following section, with full results in the results directory and appendices.

The final model was trained on the custom dataset using parameters and settings derived from the best-performing initial experiments:

- Batch size of 8 (GPU memory constrained)
- 50 training epochs
- Early stopping at 99.6% IoU threshold
- Separate evaluation on File4 and File5

The implementation is identical to the classification approach. The Intersection over Union (IoU) metric was used to track progress, as it measures bounding box overlap. Only one bounding box and one class exist per image; mean IoU and mean Mean Absolute Error (MAE) were used for evaluation.

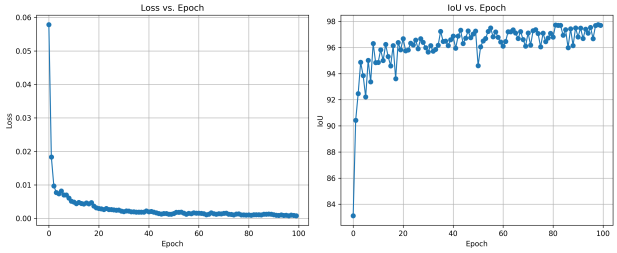
B. Detection Results

Detection experiments were conducted in two phases, mirroring the classification procedure. Multiple architectures were tested on File5 to determine the optimal approach. The best-performing configuration was then trained on a custom dataset and evaluated against both File4 and File5.

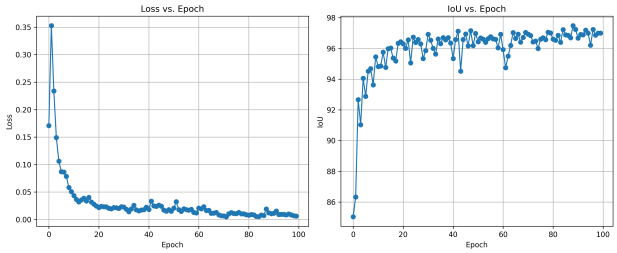
1) Initial Experiments: The selected architectures were used with a batch size of four due to GPU memory constraints. Each configuration was tested on 1000 samples from File5, outputting the following results:

- **FasterRCNN-ResNet50**:
  - Batch size 4: Mean IoU: 97.80%
- **MobileNetV3-Large-FPN**:
  - Batch size 4: Mean IoU: 95.93%
- **RetinaNet-ResNet50**:
  - Batch size 4: Mean IoU: 98.05%

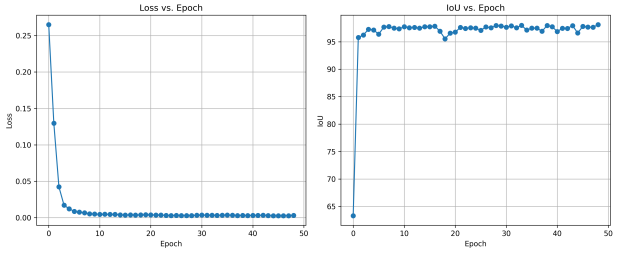
The results are shown in Figure 13.



(a) FasterRCNN training progress



(b) MobileNet (pre-trained) training progress



(c) RetinaNet (pre-trained) training progress

Fig. 13: Training progress for different model architectures showing training loss and IoU over epochs

The RetinaNet model was observed to perform the best and converge the fastest, exhibiting the highest IoU and lowest loss.

2) Final Model Performance: RetinaNet with a ResNet50 backbone was selected as the final model due to its performance. It was trained on FileCustom2, a synthetic dataset approximating conditions in Files 4 and 5. A set of 10,000 images was used, but this is adjustable in the configuration file. This was done to conserve time and resources, as the dataset was large enough, and augmentation plus each epoch were lengthy.

The final evaluation produced these results:

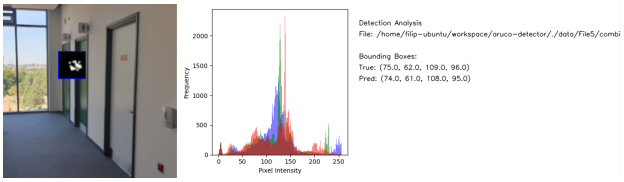
- **Training Evaluation** (1,000 testing samples):
  - Mean IoU: 99.70%
- **File4 Evaluation** (100 of the provided samples):
  - Mean IoU: 89.00%
- **File5 Evaluation** (100 of the provided samples):
  - Mean IoU: 85.64%

Strong performance was demonstrated across various distortion conditions. A perfect detection rate on File4 (IoUs over 50%) indicated effective handling of weak distortions. On File5, only three out of 100 markers were missed, showing resilience to more demanding conditions. The missed markers (and other results) can be seen in Figure 14.

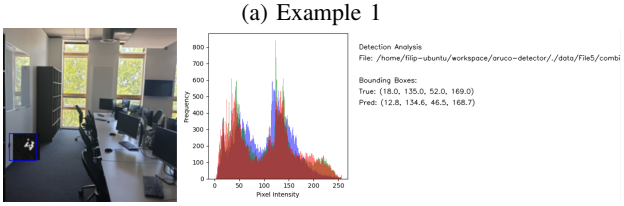
IV. Conclusion

Deep learning approaches for ArUco marker detection and classification under challenging conditions were explored. Several key findings emerged:

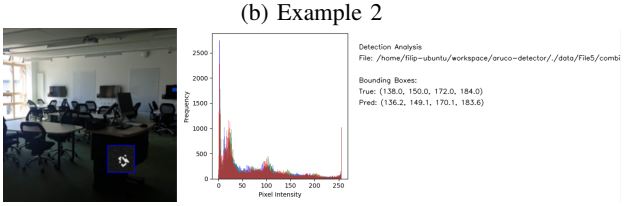
- **Classification Performance**: GoogLeNet was observed to achieve near-perfect accuracy (100%) on weakly distorted markers and to maintain robust performance (97.48%) even under severe distortions, thereby confirming that transfer learning is effective. The custom model was found to perform better after extended training, as verified individually in the GitHub commit d9fd8ad.
- **Detection Performance**: Excellent performance in localising markers was shown by RetinaNet with ResNet50 backbone, achieving 89.00% IoU in basic office scenarios and 85.64% IoU under challenging conditions. Occasional (0,0) predictions highlighted the need for longer training.
- **Architecture Insights**: Pre-trained models consistently outperformed custom architectures.
- **Technical Limitations**: GPU memory constraints restricted batch sizes and training duration, with each detection-model epoch requiring 20 minutes at the current state. With greater compute, the batch size should be increased and the dataset expanded through the configuration file to improve detection outcomes. More settings and parameters could then easily be tested and compared.
- **Augmentation Mistake**: Additional data augmentation on pre-augmented images proved counterproductive for the reason that the images were already augmented.



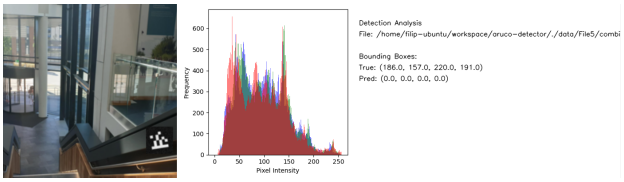
(a) Example 1



(b) Example 2



(c) Example 3



(d) Example 4

Fig. 14: Final detection model prediction examples

Future improvements that could yield interesting results include:

- Expanding synthetic dataset generation with perspective transformations, partial occlusion, and other distortions
- Implementing larger batch sizes and extended training (more compute required)
- Collecting real-world data to validate synthetic training results
- Testing more pre-trained detection models given sufficient computational resources

These outcomes show the viability of deep learning for marker detection, although systematic improvements, resource expansion, and thorough validation remain highly essential.

References

Czurkó, D., Fehér, G. (2023). AI-Assisted Drone Localization of Arbitrary Objects using Aruco Markers. In: *1st Workshop on Intelligent Infocommunication Networks, Systems and Services*. DOI: 10.3311/WINS2023-003.

Nakajima, K., Nagashima, T., Komori, Y., Yasuda, S., Tanabe, R., Uemura, W. (2024). About an Error Correcting Method of ArUco Markers Considering Row Data for a Mobile Robot. In: *2024 IEEE 13th Global Conference on Consumer*

Electronics (GCCE), pp. 273–275. Available from: https://api.semanticscholar.org/CorpusID:274371639.

PyTorch (2024a) *Faster R-CNN Implementation*. Available from: https://pytorch.org/vision/0.12/_modules/torchvision/models/detection/faster_rcnn.html [Accessed Mar. 20, 2024].

PyTorch (2024b) *FasterRCNN MobileNetV3 Implementation*. Available from: https://pytorch.org/vision/main/models/generated/torchvision.models.detection.fasterrcnn_mobilenet_v3_large_fpn.html [Accessed Mar. 20, 2024].

PyTorch (2024c) *RetinaNet Implementation*. Available from: https://pytorch.org/vision/0.20/_modules/torchvision/models/detection/retinanet.html [Accessed Mar. 20, 2024].

PyTorch (2024d) *torchvision.models Documentation*. Available from: https://pytorch.org/vision/main/models.html [Accessed Mar. 20, 2024].

Romero-Ramirez, F. J., Muñoz-Salinas, R., Medina-Carnicer, R. (2021). Tracking fiducial markers with discriminative correlation filters. In: *Image and Vision Computing*. 107, p. 104094. ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2020.104094. Available from: https://www.sciencedirect.com/science/article/pii/S0262885620302262.

Zakiev, A., Tsoy, T., Shabalina, K., Magid, E., Saha, S. K. (2020). Virtual Experiments on ArUco and AprilTag Systems Comparison for Fiducial Marker Rotation Resistance under Noisy Sensory Data. In: *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–6. DOI: 10.1109/IJCNN48605.2020.9207701.

APPENDIX A
GITHUB REPOSITORY

The GitHub repository for this project can be found at the following link: https://github.com/FHanus/aruco-detector.

The repository contains the following key components that should be viewed in order to more thoroughly understand the project:

1) **Results, classification, per trained model:**
   - Per class accuracy plots
   - Per class confusion matrices
   - Per class precision plots
   - Per class recall plots
   - Training progress plots
   - CSVs with evaluation predictions and data
2) **Results, detection, per trained model:**
   - IoU distribution plots
   - Training progress plots
   - CSVs with evaluation predictions and data
3) **Data:**
   - Original datasets (File1-File5)
   - Custom synthetic datasets (Custom1-Custom2)
4) **Docs:**
   - This LaTeX paper
   - README explaining the repository more thoroughly
   - Training log file
5) **Scripts:**
   - All of the code

This repository represents a complete research project on ArUco marker detection and classification, including code implementation, experimental results, and detailed documentation.