Report on Training a YOLOv8 Object Detection Model for Rock-Paper-Scissors Recognition

Mehran Bakhtiari

July 12, 2025

Contents

Abstract

This report provides a detailed, end-to-end analysis of training a computer vision model for the object detection task of recognizing Rock, Paper, and Scissors hand gestures. The project leverages the state-of-the-art YOLOv8 (You Only Look Once, version 8) architecture, specifically the lightweight `yolov8n` variant. The entire workflow, from environment setup and data acquisition to model training, hyperparameter configuration, and in-depth results analysis, is documented. The model was trained for 50 epochs on a public dataset from Roboflow, achieving excellent performance metrics, including a mean Average Precision (mAP) at IoU 0.5 of **0.96** and an mAP at IoU 0.5-0.95 of **0.777**. This document dissects the training logs, evaluates performance through loss and accuracy curves, analyzes the confusion matrix, and concludes with recommendations for future improvements and advanced techniques.

1 Initial Setup and Environment

1.1 Framework and Version Selection

The project utilized **YOLOv8**, a state-of-the-art, real-time object detection model developed by Ultralytics. The choice of YOLOv8 was driven by its exceptional balance of high accuracy and impressive inference speed, making it suitable for a wide range of applications. Specifically, the `yolov8n` (nano) pretrained model was selected as the base. This is the smallest and fastest variant, ideal for rapid prototyping, learning, and deployment on edge devices, without a significant compromise in performance for simpler tasks like this one. This approach leverages **transfer learning**, where the model, already trained on the large-scale COCO dataset, can adapt to our specific classes (Rock, Paper, Scissors) much more efficiently than training from scratch.

1.1.1 YOLO Family Comparison

While YOLOv8 is a popular choice, several other versions of YOLO exist, each with its own strengths:

- **YOLOv8 (Ultralytics)**: Currently the most popular version, known for its excellent documentation, ease of use, and strong community support. It serves as a great starting point for both beginners and experts.
- **YOLOv10**: The latest official version, featuring architectural improvements for better performance and efficiency.
- **YOLOv9**: Introduced innovations like Programmable Gradient Information (PGI) to achieve a strong balance of performance and efficiency.
- **YOLOv5**: A widely used and stable predecessor to YOLOv8, still backed by a large community and many resources.

For this project, **YOLOv8** was chosen due to its robust ecosystem and straightforward implementation.

1.1.2 Pre-trained Model Variants

Ultralytics provides several pre-trained YOLOv8 models, scaled for different needs:

- `yolov8n` (nano): The smallest and fastest model, ideal for edge devices and rapid prototyping.
- `yolov8s` (small): A good balance between speed and accuracy.

- `yolov8m` (medium): Offers higher accuracy than the small version with a moderate increase in computational cost.
- `yolov8l` (large): A powerful model for tasks requiring high accuracy.
- `yolov8x` (extra-large): The most accurate model in the series, suitable for high-end applications where inference speed is less critical.

1.2 Execution Environment: Google Colab

Instead of a local machine, **Google Colaboratory (Colab)** was chosen as the execution environment. The primary reason for this choice is the access to powerful, free-of-charge hardware, specifically **NVIDIA GPUs (e.g., Tesla T4)**. Training deep learning models is a computationally intensive task that is significantly accelerated by GPUs. A local setup without a dedicated NVIDIA GPU would have resulted in prohibitively long training times. Colab provides a pre-configured environment with essential libraries and seamless integration with Google Drive for data persistence.

1.3 Library Installation

The following core Python libraries were required and installed in the Colab environment using the `pip` package manager.

```
!pip install ultralytics opencv-python numpy matplotlib torch
    torchvision
```

Listing 1: Installing necessary Python packages.

- **ultralytics**: The official package for the YOLOv8 framework, providing all necessary tools for training, validation, and prediction.
- **opencv-python**: An essential library for computer vision tasks, used internally by Ultralytics for image processing and augmentation.
- **numpy**: A fundamental package for numerical computation in Python.
- **matplotlib**: A widely-used plotting library for visualizing training results, such as loss curves and metric graphs.
- **torch & torchvision**: PyTorch is the deep learning framework upon which YOLOv8 is built.

2 Dataset Acquisition and Preparation

2.1 Dataset Sourcing and Structure

For any object detection task, a properly structured and annotated dataset is paramount. The options were to either create a custom dataset or use a publicly available one. For this project, a pre-existing dataset was chosen to expedite the process. Key considerations include the classes to be detected, image diversity (lighting, angles, backgrounds), and annotation format. YOLO requires bounding box annotations in normalized `.txt` files.

2.1.1 Public Dataset Repositories

While creating a custom dataset is an option, using a pre-existing public dataset can significantly expedite the process. Here are some excellent sources:

- **Roboflow Universe**: A repository of over 100,000 datasets, many of which are already pre-formatted for YOLO. It's an ideal starting point for finding niche datasets and allows for easy downloading and preprocessing. This was the source for the dataset used in this project.
- **COCO (Common Objects in Context)**: A large-scale dataset with 80 common object categories. It is a standard benchmark for general-purpose object detection. Annotations are in JSON format and require conversion scripts (e.g., `coco2yolo`) for YOLO compatibility.
- **Open Images Dataset (OID)**: An even larger dataset from Google with over 600 classes, suitable for training models on a very diverse set of objects.
- **Kaggle**: A platform for data science competitions and datasets. Many domain-specific object detection datasets (e.g., traffic signs, medical imaging) can be found here.
- **Pascal VOC**: A classic dataset with 20 object classes. It's smaller than COCO but still useful for prototyping and smaller-scale tasks.

The dataset was sourced from **Roboflow Universe**, a large repository of public computer vision datasets. The specific dataset used is: Rock Paper Scissors SXSW. This choice was strategic because Roboflow allows users to download datasets pre-formatted for various models, including the specific format required by YOLOv8. This eliminated the need for manual conversion, which can be a time-consuming and error-prone step.

2.1.2 Dataset Splitting

To properly train and evaluate a model, the dataset must be split. A standard practice is to divide the images into three sets:

- **Training Set (70-80%)**: The data the model learns from.
- **Validation Set (10-20%)**: Used to tune hyperparameters and check for overfitting during training.
- **Test Set (10%)**: Used for the final, unbiased evaluation of the trained model's performance.

The Roboflow dataset used in this project was pre-split into training and validation sets.

2.1.3 Required YOLOv8 Dataset Structure

YOLOv8 expects a specific directory structure and annotation format. The dataset directory must contain:

- `/train/images/`: Folder with training images.
- `/train/labels/`: Folder with training annotation files.
- `/valid/images/`: Folder with validation images.
- `/valid/labels/`: Folder with validation annotation files.
- `/test/images/`: (Optional) Folder with test images.
- `/test/labels/`: (Optional) Folder with test annotation files.
- `data.yaml`: A configuration file defining paths and class information.

The `data.yaml` file defines the paths to the data and class information. Its structure is as follows:

```yaml
train: ../train/images
val: ../valid/images
test: ../test/images # Optional

# number of classes
nc: 3

# class names
names: ['Paper', 'Rock', 'Scissors']
```

Listing 2: Example structure of the data.yaml file.

Each image file (e.g., `image1.jpg`) must have a corresponding text file (`image1.txt`) in the `labels` folder. Each line in the text file defines one bounding box in the format:

```
<class_id> <x_center> <y_center> <width> <height>
```

Where coordinates are normalized (from 0 to 1) relative to the image size.

2.1.4 Annotation and Conversion Tools

If a dataset is not in the correct format, several tools can be used for labeling (annotation) or conversion:

- **Annotation Tools**: For creating labels from scratch, tools like **LabelImg** (desktop), **CVAT** (web-based), and **Roboflow's annotation tool** (web-based) are excellent choices. They provide a graphical interface to draw bounding boxes and export them in the YOLO format.
- **Conversion Scripts**: For datasets in other formats (e.g., COCO JSON, Pascal VOC XML), custom Python scripts or Roboflow's platform can be used to convert the annotations into the required YOLO `.txt` format.

2.2 Data Handling in Google Colab

The dataset was downloaded from Roboflow as a `.zip` file.

1. **Upload and Extraction**: The `.zip` file was uploaded to the Colab session and extracted into a directory named `/content/dataset/`.
2. **Path Correction in `data.yaml`**: The default `data.yaml` file provided by Roboflow contained relative paths. These paths were programmatically updated to the absolute paths within the Colab environment (e.g., changing `../train/images` to `/content/dataset/train/images`). This is a critical step to ensure the YOLO trainer can locate the data.
3. **Data Persistence with Google Drive**: Colab runtimes are ephemeral, meaning all uploaded data is lost when a session ends. To prevent re-uploading the dataset for every session, Google Drive was mounted to the Colab environment. The processed dataset was then copied to a folder in Google Drive. In subsequent sessions, the dataset could be quickly copied back from Drive to the Colab runtime, saving significant setup time.

---

3 Training Setup and Configuration

The model training was initiated using the `model.train()` method from the Ultralytics library. The configuration was a combination of default YOLOv8 settings and custom parameters tailored for this task.

```
model = YOLO('yolov8n.pt')
model.train(
    data='/content/dataset/data.yaml',
    epochs=50,
    imgsz=640,
    batch=16,
    device=0,
    name='rock_paper_scissors'
)
```

Listing 3: The training command executed in the notebook.

3.0.1 Key Training Parameters

- **model='yolov8n.pt'**: Specifies the starting model. As discussed, this uses the pre-trained weights of the YOLOv8 nano model, enabling effective transfer learning.
- **data='/content/dataset/data.yaml'**: Points to the YAML file that contains the dataset paths and class information.
- **epochs=50**: The model was trained for 50 full passes over the entire training dataset. This number was chosen as a balance between allowing the model sufficient time to learn and avoiding excessive training time or overfitting.
- **imgsz=640**: All images are resized to 640x640 pixels before being fed into the network. This is a standard resolution for many YOLO models that provides a good trade-off between detail retention and computational load.
- **batch=16**: 16 images are processed together in a single forward/backward pass. This batch size is a good fit for the memory capacity of the Tesla T4 GPU provided by Colab.
- **device=0**: Explicitly assigns the training process to the first available GPU (CUDA device 0).
- **name='rock_paper_scissors'**: Assigns a custom name to the output directory, where all results (weights, plots, logs) are saved. This is good practice for project organization.

3.0.2 Other Notable Hyperparameters (Default and Automatic)

The Ultralytics framework handles many hyperparameters automatically but provides fine-grained control if needed.

- **optimizer='auto'**: A key feature of YOLOv8. Instead of manually setting an optimizer and learning rate, this setting allows the framework to automatically select the best optimizer (in this case, **AdamW**) and an optimal initial learning rate based on the dataset and model architecture. As seen in the logs, it chose `lr=0.001429`. This removes the need for manual tuning of one of the most sensitive hyperparameters.
- **lr0=0.01**, **lrf=0.01**: These define the initial and final learning rates. The final learning rate is `lr0 * lrf`. However, these were ignored because `optimizer='auto'` was active. The learning rate itself follows a schedule, typically with a warmup phase followed by a cosine decay, as seen in the learning rate plot.
- **patience=100**: This parameter controls early stopping. The training would stop if no improvement in a key metric (like mAP50-95) is observed for 100 consecutive epochs. A high value was used to ensure the model completed all 50 epochs.
- **augment=True** (by default): Data augmentation is enabled by default. The logs show that `albumentations` like Blur, MedianBlur, ToGray, and CLAHE were applied. Additionally, YOLOv8 applies its own augmentations like mosaic, which was active for the first 40 epochs (`close_mosaic=10`). Augmentation is crucial for making the model more robust and preventing overfitting.

---

4 Results and Analysis

4.1 Training Process Log Analysis

The console output provides a wealth of information about the training process.

```
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8n.pt...
...
Ultralytics 8.3.165 ... CUDA:0 (Tesla T4, 15095MiB)
...
Overriding model.yaml nc=80 with nc=3
...
Model summary: 129 layers, 3,011,433 parameters, 3,011,417 gradients, 8.2 GFLOPs
```

```
Transferred 319/355 items from pretrained weights
...
albumentations: Blur(p=0.01, blur_limit=(3, 7)), MedianBlur(p=0.01, ...
...
optimizer: 'optimizer=auto' found ... determining best 'optimizer', 'lr0' and 'momentum' automatically...
optimizer: AdamW(lr=0.001429, momentum=0.9) with parameter groups...
...
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       1/50      2.11G      1.294      2.975      1.461         14        640
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95)
                   all        576        400       0.54      0.605      0.549      0.333
...
Closing dataloader mosaic
...
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       41/50      2.57G     0.6185     0.3649     0.9893          5        640
...
50 epochs completed in 1.571 hours.
...
Validating runs/detect/rock_paper_scissors/weights/best.pt...
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95)
                   all        576        400      0.951      0.922       0.96      0.777
                 Paper        132        139      0.962      0.928      0.963      0.786
                  Rock        121        141      0.935      0.929      0.949      0.759
               Scissors        116        120      0.956      0.909      0.967      0.785
```

Listing 4: Key sections of the training log.

Key Observations from the Log

- **Initialization**: The log confirms the use of a Tesla T4 GPU. The pre-trained model, originally configured for 80 COCO classes (`nc=80`), was automatically adapted for our 3 classes (`nc=3`). The model has approximately 3 million parameters.
- **Transfer Learning**: The message `Transferred 319/355 items` indicates that

the majority of weights from the pre-trained model were successfully loaded. The remaining non-transferred weights belong to the final detection layer, which needs to be re-initialized for the new number of classes.

- **Loss Functions**: During training, three types of loss are calculated:
  - **box_loss**: Bounding box regression loss (measures how well the model predicts the location and size of objects).
  - **cls_loss**: Classification loss (measures how well the model predicts the correct class of an object).
  - **dfl_loss**: Distribution Focal Loss, an advancement in regression loss that treats bounding box prediction as a distribution.
- **Mosaic Augmentation**: The log indicates `Closing dataloader mosaic` towards the end. YOLOv8 uses mosaic augmentation (stitching four images together) for the majority of the training but disables it for the final epochs (10, in this case). This allows the model to learn from whole, un-augmented images just before training concludes, which often leads to a final performance boost. This is visible as a sharp drop in loss around epoch 40 in the graphs.
- **Final Validation**: After 50 epochs, the best saved weights (`best.pt`) are automatically validated. The model achieves outstanding final metrics: an overall precision of 0.951, recall of 0.922, mAP@0.5 of 0.96, and mAP@0.5:0.95 of 0.777. The results are also broken down per class, showing strong, balanced performance across Rock, Paper, and Scissors.

4.2 Performance Visualization and Analysis

The training process generates several plots that are crucial for understanding model performance. Additionally, the framework automatically saves images with predicted bounding boxes on a batch of validation data, allowing for direct, qualitative assessment of the model's behavior.

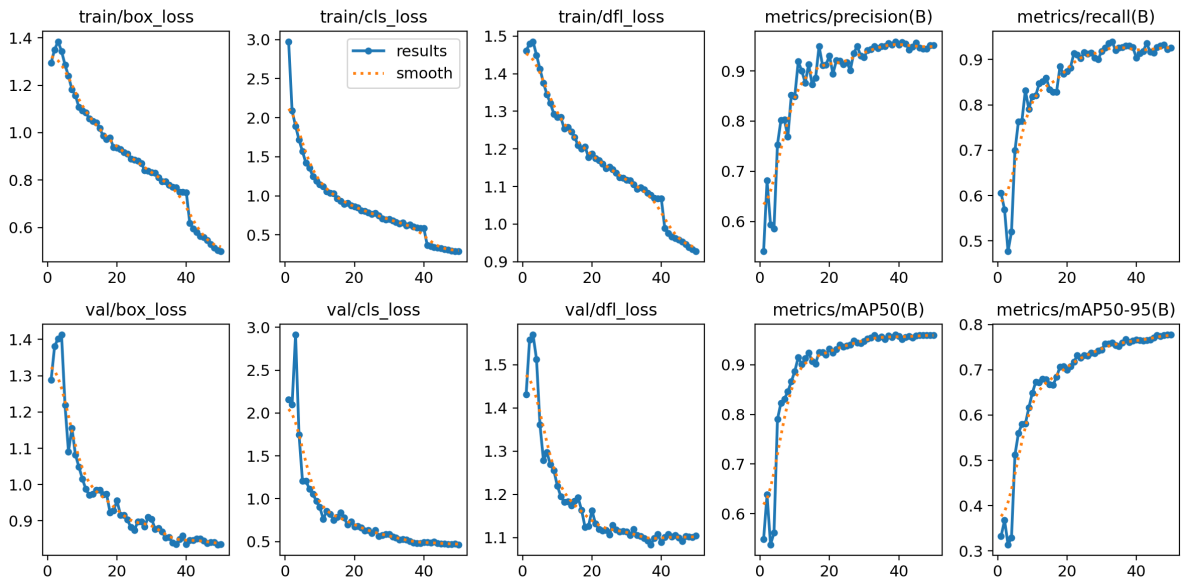

Figure 1: A summary of all training and validation metrics generated by Ultralytics.

4.2.1 Loss Curves

Figure 2 and 3 show the progression of the three loss components over 50 epochs.

- **Overall Trend**: All loss curves (both train and validation) show a consistent downward trend, which is a clear sign that the model is successfully learning and converging.
- **Convergence**: The curves begin to flatten out towards the later epochs, suggesting that the model is approaching its optimal performance under the current configuration. Training for significantly more epochs might yield only marginal gains.
- **Train vs. Validation Loss**: The train and validation loss curves track each other very closely. The gap between them is minimal, which indicates that the model is **not overfitting**. If the validation loss had started to increase while the training loss continued to decrease, it would have been a sign of overfitting.
- **Mosaic Closure Impact**: A distinct "elbow" or drop is visible around epoch 40, particularly in the training loss curves. This directly corresponds to the disabling of the mosaic augmentation, after which the model trains on simpler, non-mosaiced images, leading to a rapid decrease in the loss for that specific data distribution.

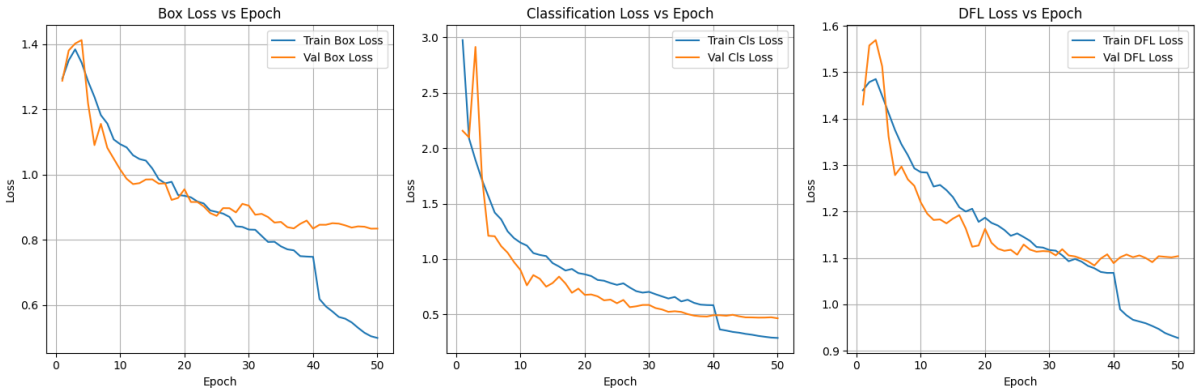

Figure 2: Training and Validation Loss curves for Box, Classification, and DFL losses.

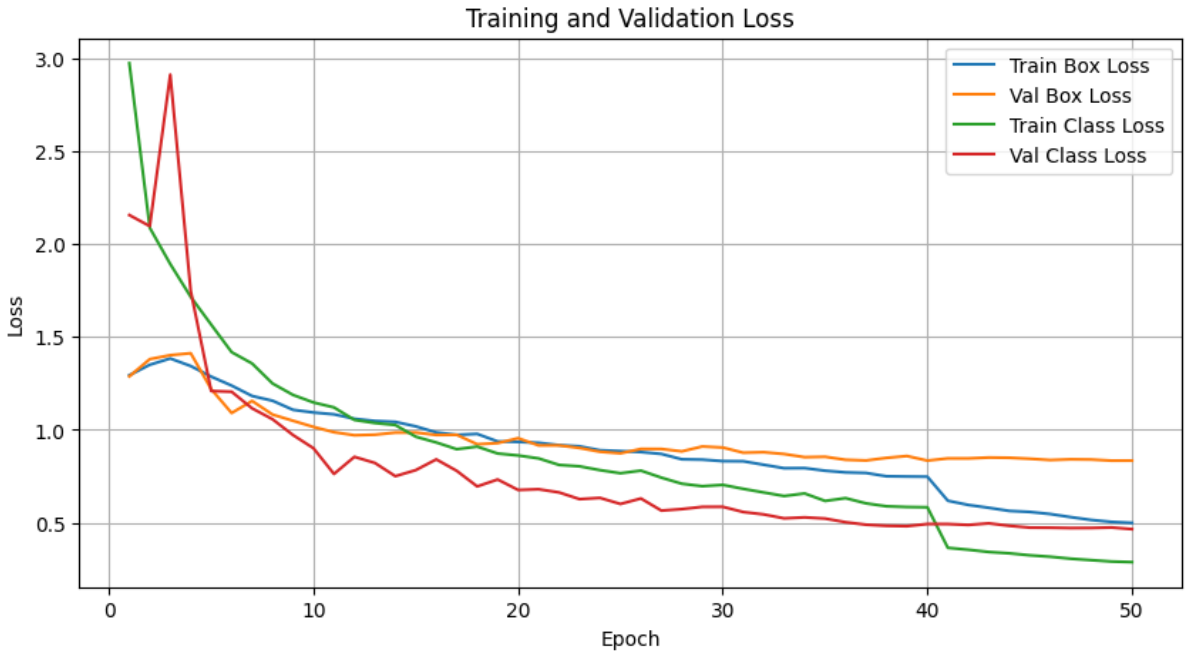

Figure 3: Alternative visualization of combined Training and Validation losses.

4.2.2 Accuracy Metrics Curves

Figure 4 and 5 plot the key performance metrics on the validation set at the end of each epoch.

- **Precision and Recall**: Precision (the accuracy of positive predictions) and Recall (the ability to find all actual positives) both rapidly increase and stabilize at high values (around 0.9-0.95). A good model requires a balance between these two, and the curves show this has been achieved.
- **mAP@0.5**: This is the mean Average Precision calculated at an Intersection over

Union (IoU) threshold of 0.5. It is the primary metric for many object detection challenges. The model shows a very rapid improvement, exceeding 0.9 (90%) by epoch 11 and plateauing around a remarkable **0.96**.

- **mAP@0.5:0.95**: This is a stricter and more comprehensive metric, representing the average mAP over multiple IoU thresholds (from 0.5 to 0.95 with a step of 0.05). It penalizes inaccurate bounding box predictions more heavily. The curve for this metric shows a slower but steady climb, reaching a final value of **0.777**. This is a very strong score and indicates the model is not only classifying objects correctly but is also predicting their bounding boxes with high precision.

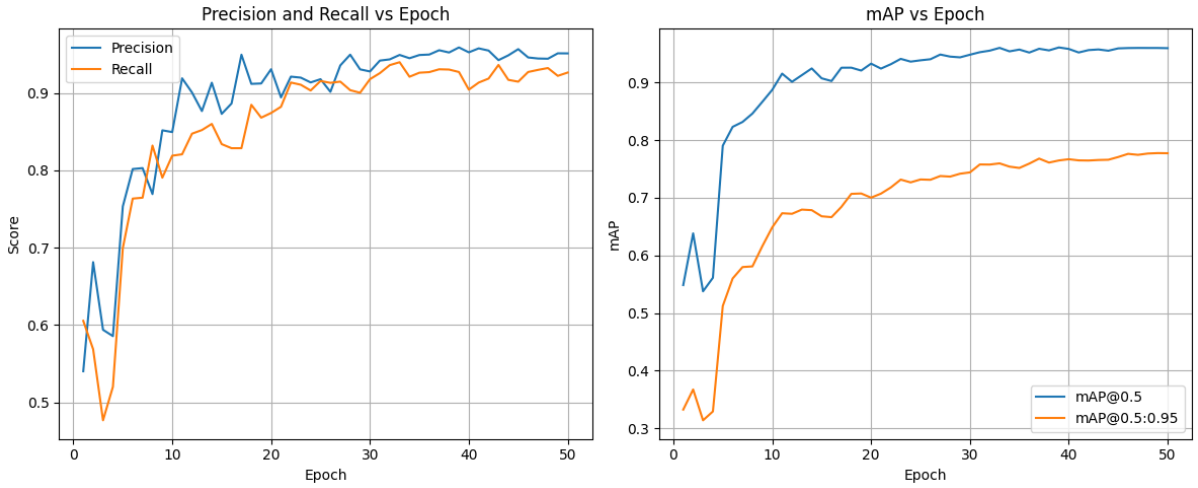

Figure 4: Precision, Recall, and mAP scores over 50 epochs.

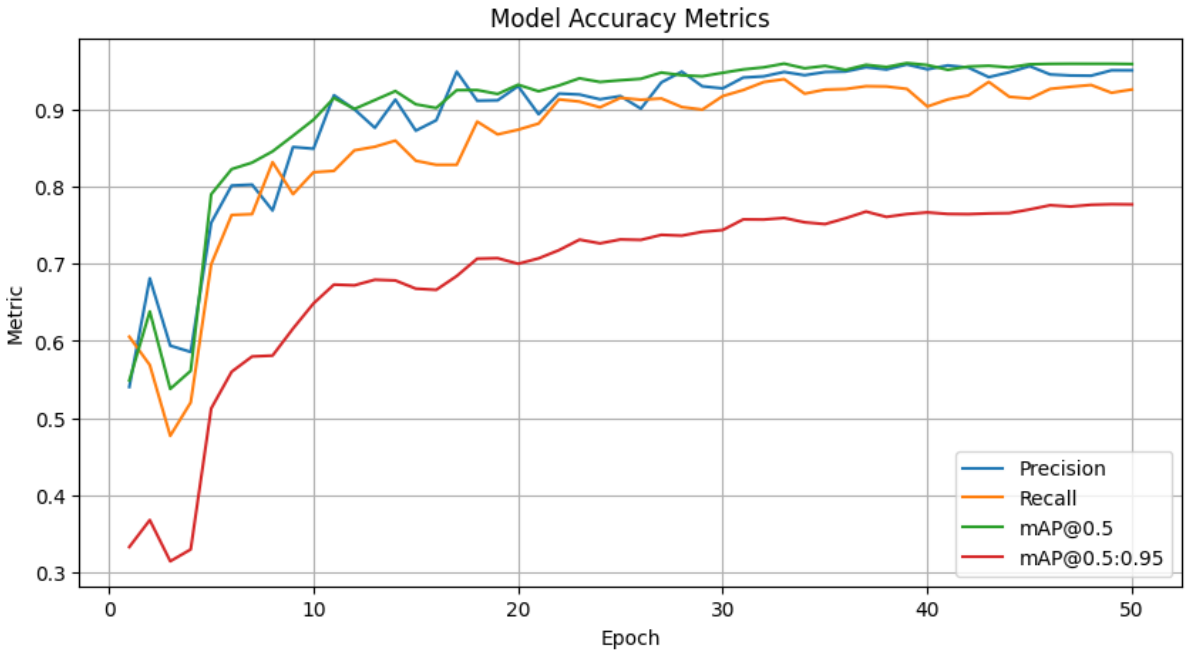

Figure 5: Alternative visualization of all accuracy metrics on a single plot.

4.2.3 Confusion Matrix

The confusion matrix (Figure 6) provides a detailed breakdown of classification accuracy. The rows represent the predicted class, and the columns represent the true class.

- **Diagonal Dominance**: The high values along the main diagonal (131 Paper, 134 Rock, 110 Scissors) show that the vast majority of predictions were correct.
- **Class-to-Class Confusion**: There are very few instances of misclassification between the main classes. For example, only 1 "Scissors" was misclassified as "Paper", and 7 "Rock" as "Paper". This indicates the model can distinguish between the hand gestures very effectively.
- **Background Confusion**: The 'background' column represents false negatives (an object was present but not detected), while the 'background' row represents false positives (the model detected an object where there was none). For example, the model failed to detect 9 "Paper" gestures, and incorrectly identified 7 background patches as "Paper". These numbers are relatively low and are the primary source of error, which is common in object detection.

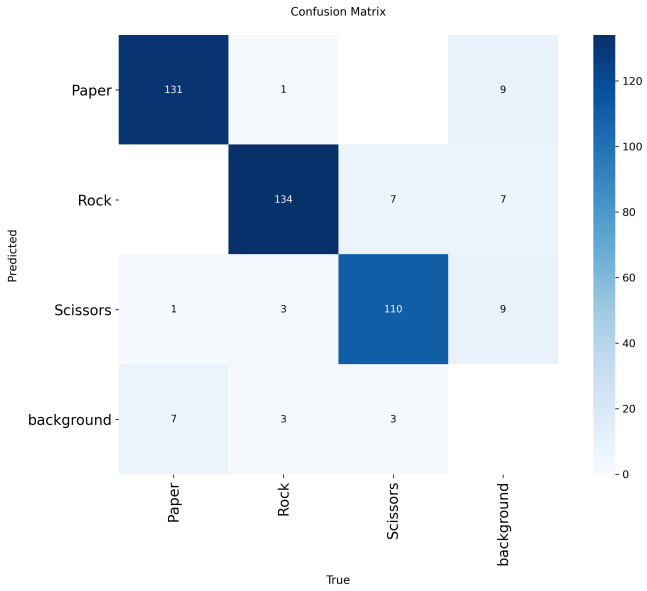

Figure 6: Confusion Matrix on the validation set.

4.2.4 Learning Rate Schedule

Figure 7 shows the learning rate used by the optimizer at each epoch. The schedule consists of two phases:

1. **Warmup (Epochs 0-3)**: The learning rate starts low and linearly increases. This prevents the model from making drastic, destabilizing updates at the very beginning of training when the weights are still random.

2. **Cosine Decay (Epochs 3-50)**: After the warmup, the learning rate follows a cosine curve, gradually decreasing over the remaining epochs. This allows the model to take large steps in the beginning to quickly approach a good solution, and then smaller, finer steps towards the end to settle into a precise minimum in the loss landscape.

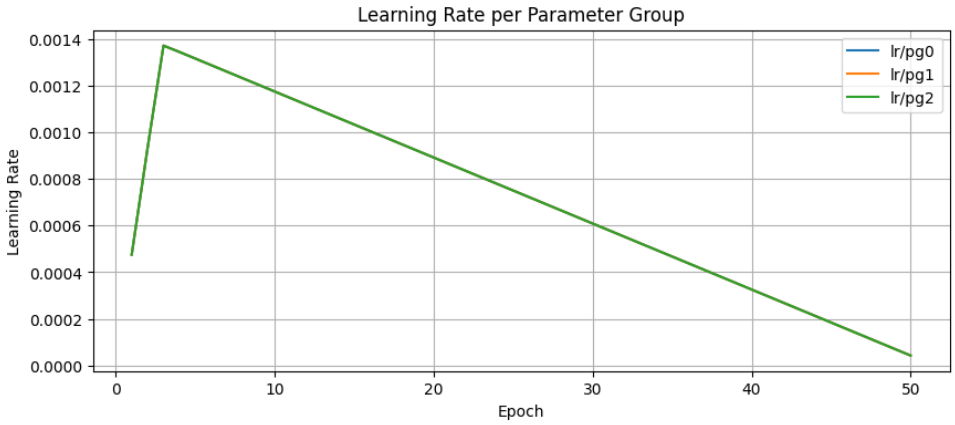

Figure 7: Learning Rate schedule across epochs.

---

5 Future Work and Advanced Techniques

The current model performs exceptionally well, but several techniques could be employed for further enhancement or for adapting it to more challenging scenarios.

5.1 Performance Optimization

- **Data Augmentation**: While default augmentations were used, their parameters (`mosaic`, `mixup`, HSV values) could be adjusted. For instance, increasing the range of HSV (Hue, Saturation, Value) augmentation could make the model more robust to different lighting conditions.
- **Hyperparameter Tuning**: The `model.tune()` function in the Ultralytics library can be used to automatically search for the best set of hyperparameters (like learning rate, weight decay, augmentation settings) over a specified number of trials. This can often squeeze out extra performance.
- **Ensemble Methods**: The predictions of several independently trained models could be averaged. This technique, known as ensembling, often leads to a more robust and accurate final prediction at the cost of increased inference time.
- **Test-Time Augmentation (TTA)**: During inference, predictions can be made on multiple augmented versions of a single test image (e.g., flipped, scaled), and the results can be averaged. This can improve accuracy on challenging images.
- **Analyze and Retrain**: If mAP is low, the best course of action is often to collect more diverse data or adjust augmentations. If performance is satisfactory but

training was interrupted, it can be resumed from the last checkpoint using the `resume=True` argument.

5.2 Addressing Common Issues

- **Overfitting**: Although not an issue in this project, if it were to occur (validation loss increasing), one could reduce model complexity (e.g., using fewer layers), increase regularization (e.g., higher weight decay), or add more aggressive data augmentation.
- **Underfitting**: If the model failed to learn effectively (both train and validation loss remain high), one could increase model capacity (e.g., use a larger model like `yolov8s` or `yolov8m`), train for more epochs, or ensure the dataset is of high quality.
- **Class Imbalance**: If one class had significantly fewer samples than others, techniques like using a weighted loss function or oversampling the minority class could be employed to ensure the model does not become biased.
- **Inference Speed**: If real-time performance is slow, use a smaller model (like `yolov8n`) or export the model to an optimized format like ONNX or TensorRT.

5.3 Advanced Techniques

- **Multi-scale Training**: Training the model with images of varying sizes (`multi_scale=True`) can make it more robust to detecting objects at different scales and distances from the camera.
- **Knowledge Distillation**: A larger, more powerful "teacher" model (e.g., YOLOv8x) could be used to train our smaller "student" model (`yolov8n`). The student model learns to mimic the outputs of the teacher, often resulting in a better-performing small model than one trained alone.
- **Active Learning**: Instead of labeling a large dataset all at once, one can iteratively improve it. Train an initial model, use it to make predictions on unlabeled data, and then have human annotators focus on labeling the images where the model was least confident. This is a highly efficient labeling strategy.

---

6 Model Deployment and Inference

Once the model is trained, it can be used for predictions on new data or deployed in applications.

6.1 Real-Time Inference with a Webcam

The trained model can be easily used for real-time detection using a webcam. The following code snippet demonstrates this process.

```
import cv2
from ultralytics import YOLO

model = YOLO('runs/detect/rock_paper_scissors/weights/best.pt')

cap = cv2.VideoCapture(0)

while True:
    ret, frame = cap.read()
    results = model(frame)
    annotated_frame = results[0].plot()
    cv2.imshow('YOLO Detection', annotated_frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Listing 5: Python code for real-time detection using a webcam.

6.2 Exporting the Model for Deployment

For deployment on various platforms (e.g., web, mobile, edge devices), the PyTorch model should be converted to a standard format like ONNX or an optimized format like TensorRT.

```
yolo export model=runs/detect/rock_paper_scissors/weights/best.pt
   format=onnx
```

Listing 6: Command to export the model to ONNX format.

---

7 Conclusion

This project successfully demonstrated the complete workflow for training a high-performance, real-time object detector using YOLOv8 and Google Colab. By leveraging a well-structured public dataset from Roboflow and the power of transfer learning, a model capable of accurately detecting Rock, Paper, and Scissors gestures was developed in just under 1.6 hours of training on a Tesla T4 GPU.

The final model achieved an outstanding **mAP@0.5 of 96.0%** and a **mAP@0.5:0.95 of 77.7%**, indicating high accuracy in both classification and localization. A detailed analysis of the training logs and performance graphs confirmed healthy convergence with no signs of overfitting. The resulting model is lightweight and fast, making it suitable for deployment in real-time applications. Future work could explore hyperparameter tuning and advanced techniques like knowledge distillation to further refine its performance. For qualitative evaluation, all of the model's predictions on sample data have been visualized and are available for review within the notebook.