AMERICAN UNIVERSITY OF BEIRUT

Automatic number plate recognition with Python, Yolov8 and EasyOCR

Submitted to Dr. Yasser Mohanna

Submitted by Nourhan Salam, Omar Ramadan, & Reem Arnaout

EECE 490 - Introduction to Machine Learning

Spring 2023-2024

April 22nd, 2024

Introduction:

Automatic Number-Plate Recognition (ANPR) systems have garnered significant attention due to their widespread applications in various fields including law enforcement, traffic management, and parking enforcement. These systems employ advanced machine learning techniques to automatically detect and recognize license plate numbers from images or video streams.

In this report, we will delve into the implementation and workings of an ANPR system developed using Python, YOLOv8 (You Only Look Once version 8), and EasyOCR. YOLOv8 is a real-time object detection and image segmentation model made by Utltralytics, and it serves as the backbone for accurately detecting license plates within images or video frames. Complementing this, EasyOCR is a lightweight and efficient optical character recognition (OCR) tool employed for precisely extracting alphanumeric characters from the detected license plates.

Data Acquisition & Collection:

Initially, we embarked on creating a custom dataset by manually annotating images using computer vision annotation tools. However, the manual annotation process proved to be time-consuming, labor-intensive and inefficient. Consequently, we transitioned to utilizing Roboflow, which is a platform that specializes in automating data management tasks, including dataset creation, augmentation, and annotation. It also offers a large variety of pre-annotated and pre-labelled datasets, hence accelerating the data collection step in order to quickly delve into training a robust object detection model. Consequently, we managed to gather a size-efficient dataset (≈21MB) that is suitable for our ANPR model training. Hence, the following link is a reference to the number plate dataset we have worked with, called “License Plate Detector Computer Vision Dataset”:

https://universe.roboflow.com/mochoye/license-plate-detector-ogxxg

Data Extraction and Installation:

We will use the following Python functions to extract the dataset and install it in Google Colab accordingly:

```
!wget -O dataset.zip
"https://universe.roboflow.com/ds/8zKUTFPXs2?key=2MnMHmdWSP"
!unzip dataset.zip
```

Note that the dataset contains 395 car images, and has been split into training set (70%), valid set (21%) and test set (9%). Auto-orientation was applied and all images have been scaled to 640x640. While the size of our dataset might be smaller than usual, we aimed to find a dataset that is uploadable while yielding satisfactory results, and this dataset reflected both of these qualities.

Environment Setup:

Upon obtaining the dataset, we configured the project environment to ensure that all required dependencies and libraries were properly installed. We leveraged the requirements.txt file, which contains a list of Python packages essential for our project's functionality. These packages include new libraries other than the ones we have previously seen in Machine Learning assignments (matplotlib, numpy, scipy, pandas, seaborn, etc.), notably:

- **Ultralytics:** Provides utilities specifically tailored for YOLO models, such as data-loaders, loss functions, and model architectures.
- **PyYAML:** Enables parsing and handling of YAML (Yet Another Markup Language) configuration files, which facilitates customization and management of model configurations.
- **OpenCV:** Offers comprehensive computer vision functionalities, including image processing, feature extraction, and visualization, essential for data preprocessing and post-processing tasks.
- **PyTorch and TorchVision:** Serve as the backbone for our deep learning operations, providing robust support for neural network architectures, optimization algorithms, and GPU acceleration.

Installing these dependencies ensured that our project environment was adequately configured, enabling smooth execution of our training and prediction scripts without encountering compatibility issues or missing required methods for basic operation. The installation step occurred using this following function:

```
!pip install -r requirements.txt
```

Model Training:

The train.py script establishes key components for training, such as data loaders, model configuration, loss functions, and utilities to support the training process. The script uses Hydra for flexible configuration management, allowing dynamic settings and parameter overrides. It also contains a “DetectionTrainer” class, which is responsible for managing the training process, as well as the “Loss” class responsible for calculating training losses. Finally, the “train” function serves as the entry point for training. We will delve into the functionality of each in order to better understand the importance of each method implemented.

DetectionTrainer Class:

The "DetectionTrainer" class extends the BaseTrainer from Ultralytics YOLO, and contains the following methods:

- **get_dataloader**: Creates a data loader for the training and validation datasets. It supports various data loading configurations, including batch size, augmentation, caching, and parallel data loading through workers.
- **preprocess_batch**: Prepares a batch for training by moving it to the correct device (such as a GPU) and normalizing image pixel values.
- **set_model_attributes**: Configures attributes of the YOLO model, such as the number of classes, class names, and specific hyperparameters based on model layers and image size.
- **get_model**: Initializes a detection model with a given configuration, and optionally loads pre-trained weights.
- **get_validator**: Sets up a validator to evaluate the model's performance during training, useful for checking progress and calculating validation metrics.
- **criterion**: Computes the training loss. It uses the Loss class, which calculates various components of loss, such as bounding box loss and classification loss.
- **label_loss_items**: Generates a dictionary of loss items for logging and visualization purposes.
- **progress_string**: Returns a formatted string showing progress information during training epochs, including epoch number, GPU memory usage, and loss metrics.
- **plot_training_samples**: Plots examples of training images with annotations, useful for visualization and monitoring the training process.
- **plot_metrics**: Plots training metrics to visualize training outcomes, allowing users to analyze the results graphically.

Loss Class:

Responsible for calculating training losses, it includes the following methods:

- **init**: Initializes key components for loss calculation, including BCEWithLogitsLoss, BboxLoss, and configurations related to model stride and the number of detection layers.
- **preprocess**: Prepares target data for loss calculation, converting bounding boxes to a compatible format.
- **bbox_decode**: Converts prediction distributions into bounding box coordinates.
- **call**: Computes the total loss, including classification loss, bounding box loss, and distribution-focused loss.

Finally, the script's execution begins with the **Main Execution Block** (__main__), where the "train" function is triggered, hence starting the YOLO training process. The latter function consists of the following tasks:

- **Configuration Setup:** where the training configuration is loaded, such as the model architecture (e.g. YOLOv8), the dataset details (e.g., data sources, labels, train/validation split), the training hyperparameters (e.g., learning rate, epochs, batch size), and other related settings.
- **Model Initialization:** This creates a YOLO model with the specified configuration.
- **Training Execution:** This calls the "train" method on the YOLO model to start the training process.

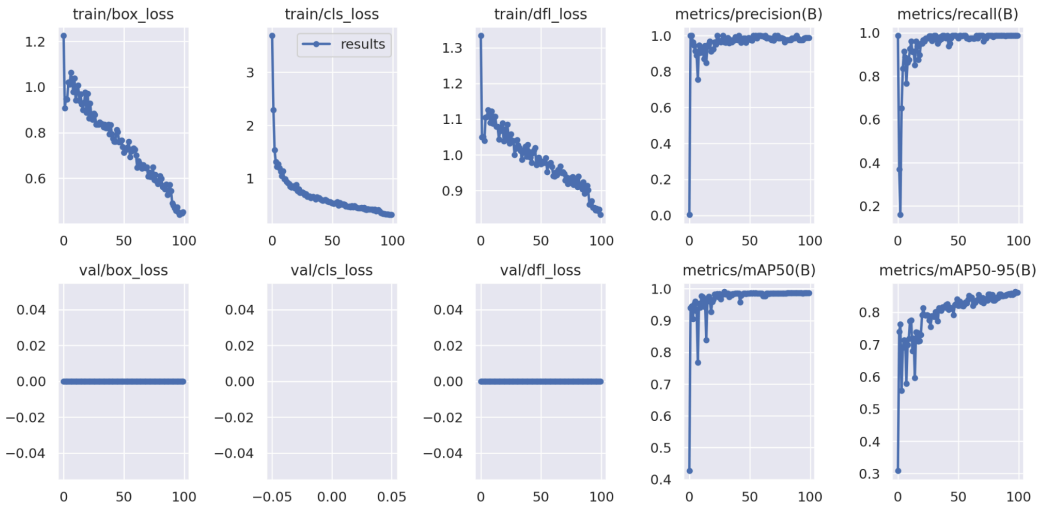
Hence, after explaining the usability of each method, we will use the YOLOv8 basic model (model=yolov8.pt) to train a model that optimally fits the object detection purpose of our car licenses database. This model (best.pt) will be trained for 100 epochs to compensate for the small size of our dataset and thus better estimate its weights and biases for more accurate results. The model training should take around 20 minutes of processing before conceiving the best.pt model, and this is done by executing the following python codes:

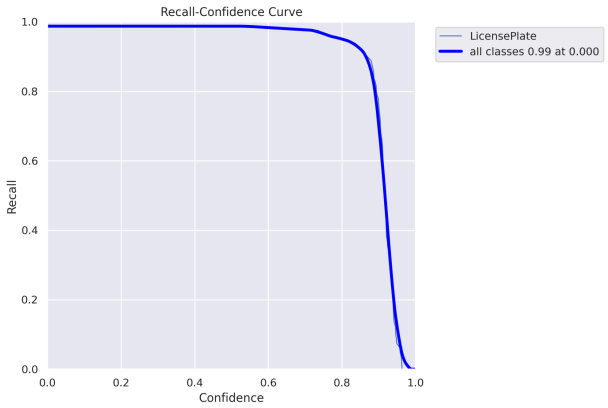
```
!python /content/train.py model=yolov8n.pt data=/content/data.yaml epochs=100
```

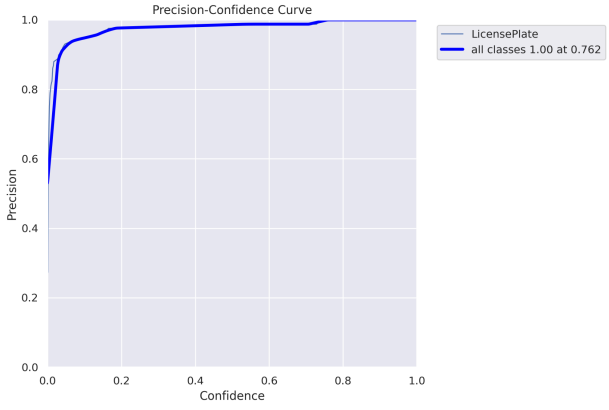
After the model training is done, "best.pt" is usually located in the following Google Colab directory: /content/runs/detect/train/weights, where another model ("last.pt") is created. We will use the former one in our detection and recognition steps, because it represents the best-performing checkpoint based on some specified metric (usually validation loss metric or specific performance metric like Mean Average Precision), hence yielding adequate results upon testing the model on photographic (.jpg, .jpeg, .png, etc.) or videographic (.mp4, .mov, ect.) data.

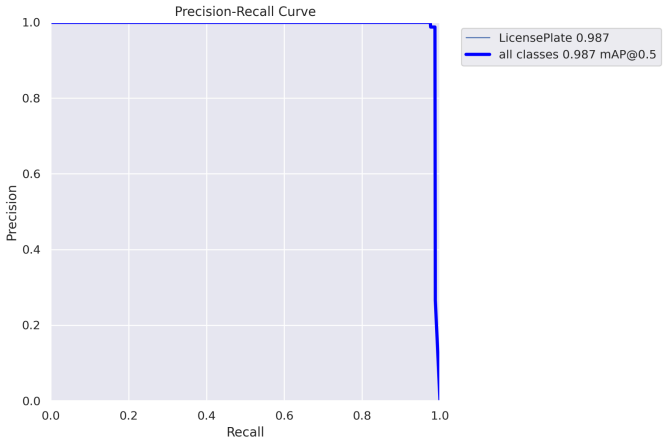
Model Output Showcasing & Analysis:

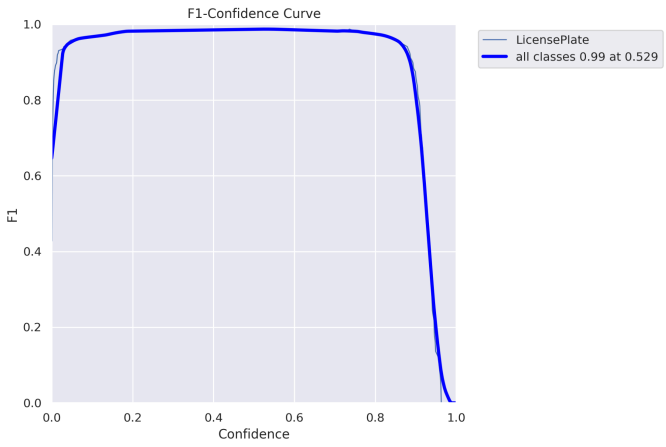
Alongside the above-mentioned two models obtained from the training we did using YOLOv8, several batches of numerical and graphical data gets stored in the following directory: /content/runs/detect/train/predictions. We will present some of them to validate the efficiency of number plate detection on the provided dataset.


train/box_loss
train/cls_loss
results
train/dfl_loss
metrics/precision(B)
metrics/recall(B)
val/box_loss
val/cls_loss
val/dfl_loss
metrics/mAP50(B)
metrics/mAP50-95(B)


Recall-Confidence Curve
LicensePlate
all classes 0.99 at 0.000
Recall
Confidence


Precision-Confidence Curve
LicensePlate
all classes 1.00 at 0.762
Precision
Confidence





Before commenting on the above figures, it is a good practice to recall the following information:

- Precision: emphasizes on the accuracy in identifying true positives, and is defined by $\frac{True\ Positives}{True\ Positives + False\ Positives}$.
- Recall: focuses on the model's ability to identify as many positives as possible, and is defined by $\frac{True\ Positives}{True\ Positives + False\ Negatives}$.
- Confidence: refers to the model's estimated probability or certainty of its prediction.
- F1-Score: is the harmonic mean of precision and recall, and is defined by $2 \times \frac{Precision \times Recall}{Precision + Recall}$

Also, it is a good practice to define the following terms derived from the Loss class in “train.py”:

- **Box Loss (bbox_loss):** Measures the discrepancy between the predicted bounding box coordinates and the ground-truth coordinates, penalizing inaccurate localization predictions.
- **Classification Loss (cls_loss):** Evaluates the difference between the predicted class probabilities and the true labels, guiding the model's ability to correctly classify objects into their respective categories.
- **Distribution Focal Loss (dfl_loss):** Integrates focal loss with distribution focal loss to address class imbalance issues and enhance detection accuracy, particularly for datasets with uneven class.

Hence, from the above provided visual outputs, we can assert that our model’s performance (best.pt) is highly effective, as the patch images show that all instances of number plates are being correctly detected, with the confidence rates ranging between 80% and reaching 100% in some pictures. In addition, the graphical results show promising evolution of the model

per-epoch, as all of the loss values for the training data (train/bbox_loss, train/cls_loss, train/dfl_loss) have progressively decreased as we approach the target epoch=100 iteration. Moreover, notice that the validation data have negligible losses (val/bbox_loss, val/cls_loss, val/dfl_loss), which further showcases a correct execution of the trained model on testing data. Also, notice that the plots reflect high statistical metrics for our model, as it performs with very high precision, recall, confidence and F1 scores.

Note that more result data was returned by the YOLOv8 training. These are all provided in the "prediction" folder of this report's submission.

Model Prediction:

We can now use the obtained model (best.pt) in order to perform inference on a given input source, such as images or videos. We have decided to demonstrate the functioning of our model by separating the checkpoints of prediction into two functionalities: either invoking methods (predict.py) to only detect the number plate, or invoking more-developed methods (predictWithOCR.py) that both detect the number plate and recognize what's written inside them.

Hence, if one wishes to test the model for detection only, then the following python code allows for such verification:

```
!python /content/predict.py model='/content/{best.pt path}'
source='/content/{sample video or image path}'
```

And if testing the model for both number plates detection and recognition is desired, then the following python cell should be executed:

```
!python /content/predictWithOCR.py model='/content/{best.pt path}'
source='/content/{sample video or image path}'
```

Model Detection:

The predict.py script implements a prediction pipeline for object detection using YOLOv8, where the "DetectionPredictor" class runs inference the "best.pt" model and then performs post-processing on the predictions to include bounding boxes for the presumably detected number plate.

DetectionPredictor Class:

This main class inherits from "BasePredictor" and overrides specific methods to handle the object detection workflow. It consists of the following pipelined operations:

- **get_annotator**: create annotator object to draw bounding boxes and labels on images.
- **preprocess**: converts an inputted image (presumably of type Numpy array) to a "torch.Tensor". This conversion allows for the image to be processed by PyTorch models, which is essential for deep learning computations. It also adjusts the data type of the tensor in case it is not of type FP32 (32-bit floating-point), which is the standard data type used in most computations. It also applies normalization methods to constraint the pixel values in a 0-1 range.
- **postprocess**: applies Non-Maximum Suppression (NMS) to remove redundant bounding boxes based on the specified confidence and IoU (Intersection over Union) thresholds (defined mathematically by $IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union}$ to evaluate how well the predicted bounding box aligns with the ground truth bounding box). It also rescales the bounding boxes to match the original image dimensions.
- **write_results**: saves the prediction results. It creates text files with bounding box coordinates, class labels (LicensePlate in our case), and confidence scores. It also annotates the image with bounding boxes and labels.

Hence, the "predict" function initializes a DetectionPredictor instance and uses it to run predictions on the specified source of visual data (image or video). This function is called in the "__main__" block, where the detection pipeline is executed.

As an illustration, here is an example of a number plate image before and after running "predict.py" on it using "best.pt":

Figure 1 and 2: Example of a number plate image detection. Left: unmodified image (input); Right: same image with LicencePlate correctly detected with confidence rate of 0.91.

We will provide more instances in our submission to showcase the detection of the number plates in images or videos. Hence, please refer to our submission for more sample outputs.

Model Detection and Recognition:

The predictWithOCR.py script combines object detection using the YOLOv8 model with Optical Character Recognition (OCR) using EasyOCR. Object detection, bounding boxes extraction, and application of OCR for text-recognition are key operations performed here, and thus all the methods included in the predict.py file (explained and elaborated in the previous section) are present in this Python script, in addition for extra methods that allow for OCR integration and writing the recognized text from the detected box region.

Hence the "DetectionPredictor" class functions very similarly to its previous implementation in "predict.py", but also calls the "GetOCR" function to recognize text from the bounding boxes derived from object detection. Therefore, annotations are made on the image with bounding boxes and the OCR-derived text.

Similarly, the "predict" function initializes a DetectionPredictor instance and uses it to run OCR-based predictions on the specified source of visual data (image or video). This function is called in the "__main__" block, where the detection and recognition pipeline is executed.

As an illustration, here is another example of a number plate image before and after running "predictUsingOCR.py" on it using "best.pt":

Figure 3 and 4: Example of a number plate image detection and detection. Left: unmodified image (input); Right: same image with LicencePlate correctly detected, as well as the derived OCR text correctly matching the plate.

Again, we will provide more instances in our submission to showcase the detection and recognition of the number plates in images or videos. Hence, please refer to our submission for more sample outputs.

Model Imperfection:

Nevertheless, just like any other machine learning application, our model is prone to errors and vulnerabilities. Hence, best.pt occasionally performs poorly, where either number plates are detected in incorrect positions, are incorrectly recognized (non-identical reading of original image text) or not read at all. Keep in mind that these are rare cases out of the images or videos we have applied predict.py or predictUsingOCR.py on.

As an illustration, we will present a few examples where the trained model did not yield to the perfect outcome we are striving to achieve:

Figure 5 and 6: Examples where the best.pt model have returned slightly erroneous outputs, or have not detected distant car plates. This error-prone behavior is observed in any machine learning model, notably ours.

Conclusion:

In this project, we have successfully integrated advanced object detection techniques with Optical Character Recognition (OCR) to create a robust solution for automatic number plate detection and recognition. Using Ultralytics YOLOv8 as the underlying object detection framework, we were able to detect number plates with high accuracy across a range of conditions. The inclusion of EasyOCR allowed for seamless extraction and recognition of text from detected regions, providing a reliable method to extract vehicle license plate numbers. Our approach demonstrated adaptability to various image qualities and scenarios, thanks to comprehensive pre- and post-processing techniques. Through extensive testing, we validated that the combination of object detection and OCR offers a powerful tool for automated recognition tasks, with potential applications in traffic monitoring, law enforcement, and parking management.

References:

Vehicle number plate detection and recognition using yolo- V3 and OCR Method. (December 2021). https://www.researchgate.net/publication/358184872_Vehicle_Number_Plate_Detection_and_Recognition_using_YOLO-_V3_and_OCR_Method

Ultralytics. (2024, April 2). *Predict*. Predict - Ultralytics YOLOv8 Docs. https://docs.ultralytics.com/modes/predict/#streaming-source-for-loop

Ultralytics GitHub Repository for YOLO model. *Ultralytics/ultralytics/models/YOLO at main · ultralytics/ultralytics*. GitHub. https://github.com/ultralytics/ultralytics/tree/main/ultralytics/models/yolo