Car License Plate Recognition System Documentation

Rostislav Lán (xlanro00), Ondřej Mach (xmacho12)

December 23, 2024

Contents

Car License Plate Recognition System Documentation

Introduction

The Car License Plate Recognition System detects and extracts text from license plates using deep learning methods. The system serves applications in traffic monitoring, automated toll collection, and parking management. Our implementation combines efficient license plate detection with optical character recognition in a streamlined pipeline.

Modern traffic systems increasingly rely on automated license plate recognition for various applications. Our solution addresses this need through a two-stage approach that separates the detection and recognition tasks, allowing for independent optimization of each component.

System Overview

The system operates in two stages: license plate detection and text recognition. We focused our development on the detection component, implementing a custom convolutional neural network for accurate plate localization. For text recognition, we integrated Microsoft's pre-trained TrOCR model as specified in the project requirements.

The separation of detection and recognition stages offers several advantages. It allows for independent optimization of each component, enables easier debugging and performance analysis, and provides flexibility for future improvements or replacements of individual components.

Neural Network Architecture

Input Processing

Our detection network processes 416x416 RGB images through three convolutional stages with increasing channel depths (32, 64, 128). The input size was chosen based on experimentation to balance computational requirements with detection accuracy. Images are automatically resized and normalized during preprocessing.

Network Layers

Each convolutional stage combines multiple operations:

1. Convolution with 3x3 kernels and appropriate padding
2. Batch normalization for training stability
3. ReLU activation for non-linearity
4. Max pooling with 2x2 windows for spatial reduction

The flattened features pass through two fully connected layers (256 units and 4 outputs), with dropout regularization in the penultimate layer. This architecture was chosen after experimenting with various configurations including different layer counts and filter sizes.

Output Processing

The network outputs normalized bounding box coordinates (xmin, ymin, xmax, ymax). Post-processing includes an adjustable enhancement factor to optimize the detected region for subsequent OCR processing. The enhancement step can expand or contract the predicted box based on confidence levels and empirical performance metrics.

Training Process

Hardware and Environment

The model training utilized:

- Hardware: NVIDIA RTX 3070 GPU
- CUDA Toolkit: 12.7
- PyTorch: 1.10.0
- Dataset: Car Plate Detection dataset (Kaggle)
- Split: 80% training, 10% validation, 10% testing
- Loss: SmoothL1Loss for bounding box regression
- Optimizer: Adam
- Training time: 30-45 minutes per optimization trial

Hyperparameter Optimization

We employed Optuna for automated hyperparameter tuning, exploring:

- Learning rates: 1e-7 to 1e-2
- Batch sizes: 10-24
- Dropout rates: 0.001-0.05
- Epochs: 20-100

Data Augmentation

To improve model robustness, we implemented several augmentation techniques:

- Random horizontal flips
- Brightness and contrast adjustments
- Gaussian noise injection
- Random rotations (±15 degrees)

Implementation Details

Project Structure

The implementation spans seven Python modules with distinct responsibilities:

- `detector.py` manages model training and hyperparameter optimization, implementing the core training loop and validation process. The module handles batch processing, loss computation, and model updates during the training phase.
- `net_model.py` defines the CNN architecture, encapsulating layer definitions and forward pass logic.
- `parser.py` handles dataset acquisition and preprocessing, implementing efficient data loading and augmentation pipelines. The module manages dataset downloads and annotation parsing.
- `predictor.py` provides inference functionality, including bounding box prediction and visualization. It supports both single-image and batch processing modes with configurable output options.
- `utils.py` contains supporting functions for data management and model persistence, including file operations and performance metrics calculations.
- `batch_predict.py` enables bulk processing of image folders making testing easier.
- `recognition.py` implements the OCR stage using TrOCR, managing text extraction and result formatting.

Data Processing Pipeline

The system processes images through several stages:

1. Input normalization and resizing to constant input size
2. CNN-based plate detection
3. Coordinate denormalization
4. Enhancement and cropping of the region of interest
5. OCR text extraction

Performance and Limitations

The system effectively detects single license plates under various conditions but has limitations:

- Reduced accuracy with multiple plates
- Sensitivity to severe occlusions
- Performance degradation in extreme lighting conditions
- Limited robustness to severe perspective distortions

The pre-trained OCR component introduces additional constraints:

- Reduced accuracy with non-standard fonts
- Sensitivity to image quality
- Limited support for multi-line plates
- Language-specific limitations

Future Development

Potential immediate enhancements include:

- Multi-plate detection capability through architecture modifications
- License plate-specific OCR model fine-tuning
- Optimization for batch processing
- Expanded dataset coverage for edge cases

Future research directions and improvements:

- Real-time processing capabilities
- Multi-language support
- Advanced post-processing techniques

Usage Instructions

The system offers both single-image and batch processing modes. Results include visualizations of detected regions and extracted text output. Results are also saved into a log file under `output/ocr_pred.txt`.

```
python detector.py <max_epochs_for_hyperparameter_optimization>
```

```
python predictor.py <path_to_saved_model> <path_to_input_image> -t 1
```

```
python batch_predict.py
```

```
python recognition.py
```

Performance Optimization

For optimal performance, we recommend:

- GPU acceleration when available
- Appropriate batch sizes for hardware
- Regular model checkpointing
- Monitoring of system resources