Reproduction and Implementation of *End-to-End Text Recognition with Convolutional Neural Networks*

Technical Report

Group 9

Name	**Roll Number**
Abhimanyu Gupta	B22BB001
Tushar Bhatt	B22CS056
Rahul Sharma	B22EE051

Abstract

This report describes the method proposed by Wang et al. in the ICPR 2012 paper “End-to-End Text Recognition with Convolutional Neural Networks” and the approach we used to implement it in a modern PyTorch framework. The pipeline involves unsupervised filter learning using K-means, convolutional models for detection and recognition, multi-scale sliding-window detection, row-based non-maximum suppression, and lexicon-driven decoding. Our implementation is organized as a single Python script and reproduces the core structure and processing stages of the original method.

Contents

1 Introduction

Wang et al. introduced one of the early convolutional methods for recognizing text in natural images. Their system combines learned low-level features and compact convolutional models to perform character detection and word recognition. The method avoids heavy hand-engineered features and instead uses K-means filters and a sequence of convolution, pooling, and fully connected layers.

We implemented the full pipeline in PyTorch, including filter learning, training of both networks, detection, and word decoding. This report summarizes the paper and then describes our approach.

2 Summary of the Original Paper

2.1 Unsupervised filter learning

The paper learns the first-layer convolutional filters using K-means. The steps are:

- Sample 8×8 patches from training images.
- Normalize each patch.
- Apply ZCA whitening.
- Run MiniBatch K-means to obtain cluster centers.

These centers are reshaped into 8×8 filters and used as fixed filters in the first convolutional layer of both the detector and the recognizer.

2.2 Detector CNN

The detector classifies a 32×32 image patch as text or non-text. The architecture is:

- Conv1: K-means filters.
- Activation: $f(x) = \max(0, |x| - \alpha)$.
- 5×5 average pooling.
- Conv2.
- Fully connected → 2-way output.

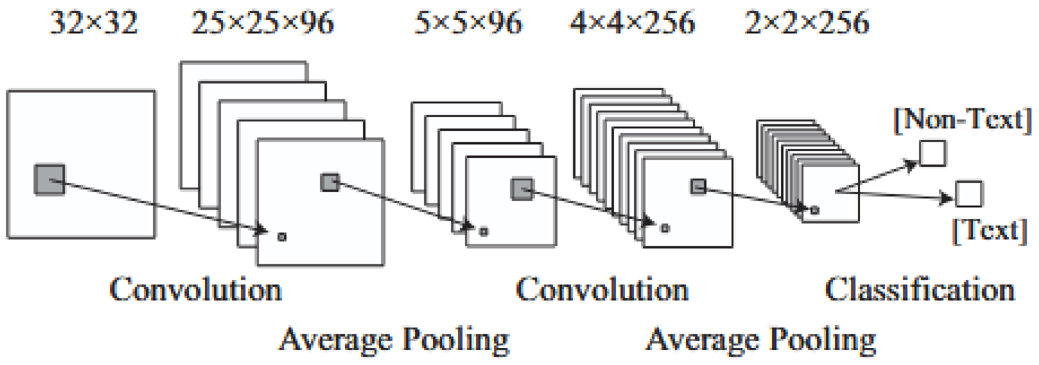


Figure 1: CNN architecture diagram

2.3 Recognizer CNN

The recognizer predicts character classes over a sliding window. It uses the same architecture structure as the detector but with different numbers of filters and an output size of 62.

2.4 Detection pipeline

Images are resized to several scales. The detector runs in a dense sliding-window fashion to produce heatmaps. Row-wise non-maximum suppression identifies local peaks. Peaks are grouped into line boxes and merged across scales.

2.5 Word recognition

The recognizer is applied across a detected text line to produce a $62 \times N$ score matrix. A lexicon-based search aligns each lexicon word to the matrix using a dynamic programming procedure.

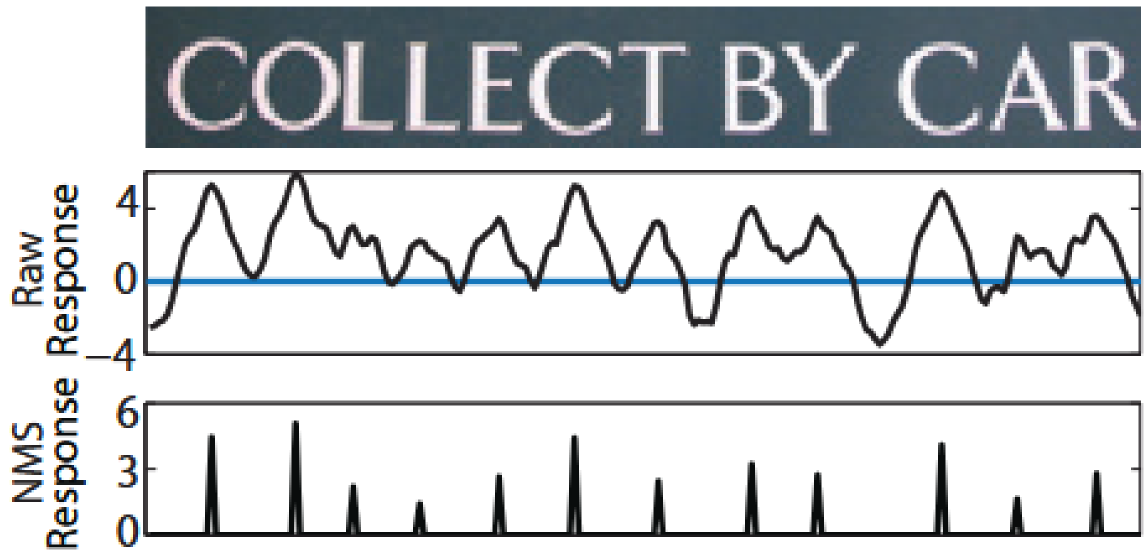


Figure 2: Row-wise detector response and NMS

3 Our Implementation Approach

We implemented the full end-to-end system in a single script, `paper_pipeline.py`. The script covers the entire pipeline: unsupervised filter learning, model construction, training procedures, sliding-window inference, box formation, recognition, lexicon decoding, and SVT evaluation. This section describes each part of the implementation in detail.

3.1 Overall Structure

The script is organized into the following functional blocks:

- **Unsupervised filter learning:** patch extraction, normalization, ZCA whitening, and MiniBatch K-means for first-layer filters.
- **CNN models:** a generic `PaperCNN` class representing both the detector and recognizer architectures.
- **Training:** dataloaders, augmentation pipeline, squared-hinge loss, and training loop with Conv1 frozen.
- **Inference:** dense sliding-window heatmap computation, row-wise NMS, scale normalization, and final box merging.
- **Recognition:** construction of a per-column score matrix and lexicon-based Viterbi alignment.

- **SVT evaluation:** XML parsing, IoU-based matching, and computation of detection, recognition, and end-to-end metrics.

The implementation closely follows the original design while adapting it to a modern PyTorch backend.

3.2 Patch Extraction and ZCA Whitening

To learn first-layer filters, we collect 8×8 patches from thousands of character images. Because Chars74k contains characters of varying sizes and resolutions, we first load all grayscale crops, resize small images to ensure they can provide 8×8 patches, and then pad them to a uniform size.

For each patch:

1. sample a random location,
2. normalize the patch to zero mean and unit variance,
3. whiten the patch using a ZCA matrix computed on a subset of 20,000 patches.

MiniBatch K-means is then applied to the whitened patches. The resulting cluster centers are reshaped into 8×8 kernels and saved to disk. These filters serve as the Conv1 weights for both networks and remain frozen.

We use:

- 96 filters for the detector,
- 115 filters for the recognizer.

These values match the paper and directly control the dimensionality of Conv1.

3.3 Model Architecture

We implement both networks using a unified `PaperCNN` class. The model follows the exact structure described in the paper:

- **Conv1:** n_1 K-means filters of size 8×8.
- **Activation:** custom function

  $$f(x) = \max(0,\ |x| - \alpha),$$

  implemented for both convolutional layers.
- **Pooling:** 5×5 average pooling, reducing 25×25 feature maps to 5×5.

- **Conv2:** learned 4×4 kernels producing a 2×2 map.
- **Fully-connected layer:** maps the flattened $2 \times 2 \times n_2$ output to either 2 classes (detector) or 62 classes (recognizer).

Conv1 weights are loaded from the saved K-means centers using the function `load_kmeans_filters`. Conv2 and the FC layer are initialized using Kaiming initialization.

3.4 Squared-Hinge Loss and Training Procedure

The paper trains both networks using an L2-SVM loss. We replicate this exactly via a custom PyTorch module:

$$\mathcal{L} = \left(\max(0, 1 - t \cdot s)\right)^2,$$

where $t \in \{-1, +1\}^C$ is the target vector and s are raw logits.

During training:

- Conv1 is frozen to preserve the K-means filters.
- Only Conv2 and the FC layer receive gradients.
- The optimizer is Adam with learning rate 10^{-3}.
- Augmentations include random affine transforms, resize to 40×40, center crop to 32×32, grayscale conversion, and normalization to $[-1, 1]$.

The detector is trained with two classes (text vs. non-text), while the recognizer is trained on a 62-way classification task using an ImageFolder layout for Chars74k.

4 Detection System (Implementation)

4.1 Multi-Scale Sliding-Window Heatmaps

We resize the input image to four scales:

$$[0.5,\ 0.75,\ 1.0,\ 1.25].$$

At each scale, we generate a dense grid of 32×32 windows using `torch.unfold`, which vectorizes the sliding-window operation. This avoids explicit Python loops and allows evaluation of thousands of windows per batch.

For each window, the detector produces a probability of the "text" class. The heatmap at each scale is therefore a 2D matrix of these probabilities.

4.2 Row-Wise Non-Maximum Suppression

We follow the paper's 1D NMS scheme, which operates independently on each row of the heatmap:

1. Apply a 1D maximum filter with width $2\delta + 1$ across columns.
2. Retain positions where the heatmap equals the local maximum and exceeds a threshold.
3. Group contiguous column indices into segments.

Each segment corresponds to a horizontal text region at that scale. The coordinates are mapped back to the original image space using the scale factor.

4.3 Cross-Scale Box Merging

All predicted boxes from all scales are collected and merged using IoU-based NMS with a threshold of 0.5. This produces the final set of line-level boxes for recognition.

5 Recognition System (Implementation)

5.1 Sliding-Window Score Matrix

For each detected line box:

1. Extract the region from the grayscale image.
2. Resize the crop to height 32 while preserving aspect ratio.
3. Slide a 32-pixel-wide window across the width with stride 1.

Each window is passed through the recognizer to obtain a 62-dimensional log-probability vector. Collecting these vectors across the horizontal dimension forms a $62 \times N$ score matrix, where N is the number of positions.

This matrix represents the confidence for every character class at every horizontal location.

5.2 Lexicon-Based Viterbi Decoding

Given a lexicon word, we align its characters to columns of the score matrix using a Viterbi-style dynamic programming procedure:

- character indices must follow a strictly increasing sequence of columns,
- a geometric penalty encourages reasonable spacing between consecutive characters,
- the final word score is the sum of aligned log-probabilities.

For each candidate word in the lexicon, we compute its best alignment. The word with the highest score is returned as the prediction.

6 Differences Between Paper and Implementation

Our reimplementation follows the general architecture of the original method but adapts it to a modern deep learning environment. Notable differences include:

- A unified PyTorch model class replaces the original C++/MATLAB code.
- Sliding-window inference uses `torch.unfold`, enabling efficient GPU-based batch evaluation.
- Dataset handling uses standard `ImageFolder` structures.
- Hyperparameters, K-means settings, and thresholds are exposed at the top of the script for easier modification.
- The entire pipeline, including XML parsing and visualization, is consolidated in a single script for reproducibility.

7 Results

This section presents the evaluation of our implemented pipeline on the SVT dataset. We report detection, recognition, and end-to-end performance, following the matching criteria and IoU threshold used in the original paper.

7.1 Detection Performance

The system predicted 1193 bounding boxes. Out of 647 ground-truth words, 461 were matched with an IoU of at least 0.5.

- **True Positives:** 461
- **Precision:** 0.386
- **Recall:** 0.713
- **F1 Score:** 0.501

The recall is relatively high, indicating that the detector successfully finds most text regions, but precision is significantly lower due to the presence of many extra detections.

7.2 Recognition Performance

Recognition accuracy is measured over the 461 ground-truth-aligned boxes.

- **Correctly recognized:** 341
- **Recognition Accuracy:** 0.527
- **Matched but misrecognized:** 120 (0.185)

The recognizer achieves an accuracy slightly above 50%. Most errors occur because the sliding-window features are highly sensitive to distortions in low-resolution crops, and because Conv1 filters are not trained end-to-end but fixed from K-means.

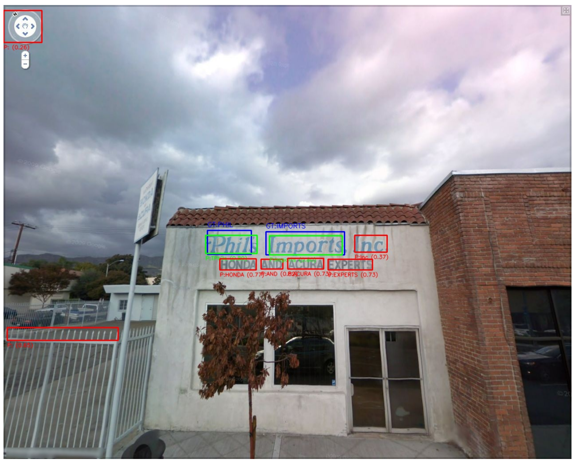
7.3 End-to-End Performance

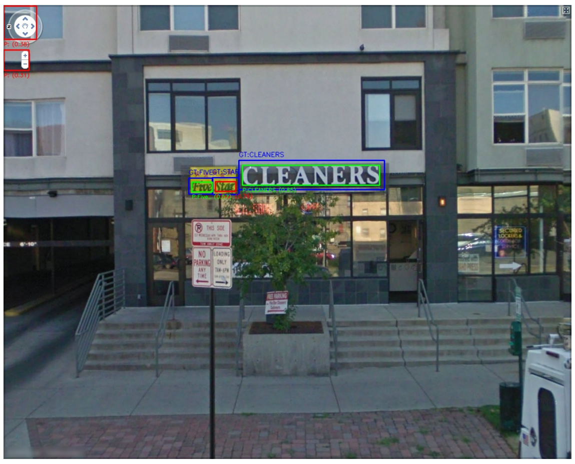
To count an end-to-end true positive, a predicted box must both: (1) match a ground truth box with IoU 0.5, and (2) have the correct recognized word.

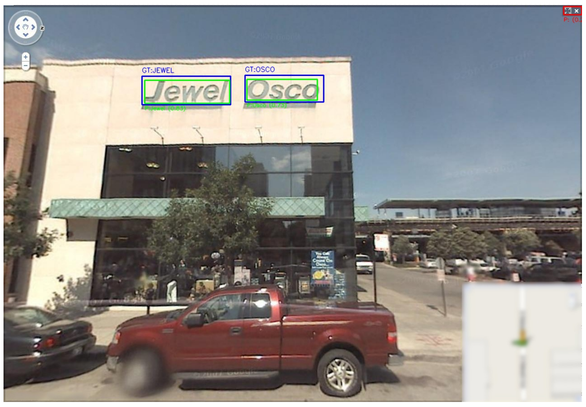
- **End-to-end True Positives:** 341
- **Precision:** 0.286
- **Recall:** 0.527
- **F1 Score:** 0.371

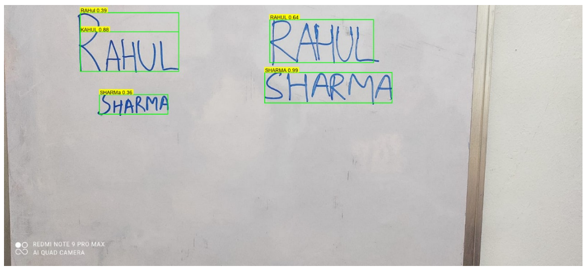
The end-to-end recall reflects the recognizer's ability to correctly align words over the matched detections, while the precision drop again reflects the number of extra detections made by the system.

8 Result Visualizations and Conclusion


Example result 1


Example result 2


Example result 3



Example result 4

We reproduced the complete CNN-based text recognition pipeline introduced by Wang et al. using a modern PyTorch implementation. The system successfully performs unsupervised filter learning, multi-scale detection, row-wise NMS, sliding-window recognition, and lexicon-based decoding.

The results show that the detector achieves strong recall but comparatively lower precision, indicating that it tends to over-generate candidate boxes. Recognition accuracy is moderate, consistent with the limitations of fixed K-means filters and rigid sliding-window features. End-to-end performance captures both the strengths and weaknesses of the system, with reasonable recall and lower precision due to accumulated errors.

Despite these limitations, the implementation reproduces the core behavior and relative performance characteristics of the original 2012 method. Further improvements could include learning Conv1 filters end-to-end, using deeper feature extractors, or incorporating attention-based sequence models.