Text Detection System

Harsh Kumar (220428)
Manisha Kaushal (220623)
Manvi Verma (220631)

Abstract

The digitization of physical documents using cameras is often hindered by poor image quality, notably motion/defocus blur and perspective distortions, which severely degrade the performance of Optical Character Recognition (OCR) systems. This project introduces "Document Quality Restoration", a multi-stage pipeline to create a robust document reader for real-world scenarios. Our primary contribution is an adaptive pre-processing approach that first classifies the input document's quality. We employ a fine-tuned MobileNetV2 classifier to detect the presence of blur. If blur is detected, an adaptive sharpening filter is applied to restore textual details. This restoration stage is followed by a robust perspective correction module that isolates the document and rectifies geometric distortions. The cleaned and flattened document is then passed to a standard OCR engine that can detect the underlying text. Our experiments demonstrate that the blur classifier achieves an accuracy of 87% on a diverse dataset of clean and blurred images.

1 Introduction

The transition from physical to digital documents is a cornerstone of modern information management. Standard OCR engines, such as Tesseract or EasyOCR, perform poorly on blurry inputs, leading to unusable text output.

In this project, we introduce a pre-processing pipeline that incorporates quality assessment and restoration stage prior to standard geometric correction and OCR. The core novelty of our work lies in this conditional pre-processing pipeline:

1. We first classify the document image as either "clean" or "blur" using a lightweight, fine-tuned deep learning model (MobileNetV2).

2. Only if the image is classified as "blur" do we apply an adaptive sharpening filter. This prevents artifact introduction on high-quality images while restoring crucial character details on degraded ones.

3. The quality-restored image is then passed through our perspective correction and OCR modules.

This adaptive methodology ensures that each document receives the appropriate treatment, leading to a more robust and accurate digitization system overall. Our experiments validate this approach, showing high accuracy in blur classification and a better OCR output.

2 Proposed Method

Our proposed pipeline consists of four main stages. Each stage is designed to address a common issue in camera-captured document images.

2.1 Stage 1: Document Quality Classification

The first and most critical stage is to assess the quality of the input image, specifically for blur.

Model Architecture: We use a MobileNetV2 model, pre-trained on ImageNet, as our feature extractor. We chose MobileNetV2 for its excellent balance of accuracy and computational efficiency, making it suitable for potential mobile deployment. The top classification layers of the pre-trained model are removed, and we freeze the convolutional base to leverage its powerful learned features.

Custom Classifier Head: On top of the frozen base, we add a Global Average Pooling 2D layer, followed by a Dense layer with 128 neurons and a ReLU activation function. A Dropout layer with a rate of 0.3 is added for regularization to prevent overfitting. The final output is a Dense layer with 2 neurons (for "clean" and "blur" classes) and a softmax activation.

Training: The model is fine-tuned on a dataset comprising two classes: "clean" and "blur". We use the Adam optimizer and categorical cross-entropy loss function. Data augmentation, including minor rotations and zooms, is applied during training to improve robustness.

2.2 Stage 2: Adaptive Sharpening

This stage is executed conditionally based on the output of the blur classifier. If the image is predicted as "blur," we

apply a simple yet effective sharpening kernel using a 2D filter operation. The kernel used is:

$$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This kernel enhances edges and details by emphasizing the difference between a pixel and its neighbors. Applying this step only when necessary is the key to our adaptive approach.

2.3 Stage 3: Perspective Correction

To handle geometric distortions, we implement a four-point perspective transform module. This process involves:

1. Converting the image (sharpened or original) to grayscale.
2. Applying a Gaussian blur to reduce noise and Canny edge detection to find prominent edges.
3. Identifying contours in the edge map and sorting them by area to find the largest candidates.
4. Approximating the contour shape to find a four-cornered polygon that represents the document's boundary.
5. Computing a perspective transform matrix from these four corners to map them to a rectangle.
6. Applying the warp perspective transformation to obtain a flattened, top-down view of the document.

Finally, contrast is enhanced on the warped image using Contrast Limited Adaptive Histogram Equalization (CLAHE) to handle uneven lighting.

2.4 Stage 4: Text Recognition and Assembly

The clean, perspective-corrected image is fed into the OCR engine.

OCR Engine: We utilize the EasyOCR library, which is based on a CRNN (Convolutional Recurrent Neural Network) architecture with CTC (Connectionist Temporal Classification) loss. It is robust and provides good performance on a variety of fonts.

Text Assembly: EasyOCR returns a list of detected text blocks with their bounding box coordinates. To reconstruct the document's original reading order, we sort these blocks based on the vertical (y-coordinate) position of their top-left corner. The sorted text fragments are then joined together to form a single, coherent block of text.

3 Experiments and Results

3.1 Dataset and Implementation

Blur Classification Dataset: For training and validating our blur classifier, we used the "blur-dataset" available on Kaggle. We combined the 'defocused_blurred' and 'motion_blurred' directories into a single 'blur' class and used the 'sharp' directory as our 'clean' class. This resulted in 840 training images and 210 validation images after an 80/20 split.

Implementation Details: The pipeline was implemented in Python. The blur classifier was built and trained using TensorFlow/Keras. Image processing relied on OpenCV. OCR was performed using EasyOCR. All experiments were conducted in a Kaggle notebook environment.

3.2 Blur Classification Performance

The blur classification model was trained for 6 epochs. The model achieved a final validation accuracy of 85.71%. The detailed performance is shown in the classification report (Table 1) and the confusion matrix (Figure 1).

Class	Precision	Recall	F1-Score	Support
blur	0.90	0.90	0.90	140
clean	0.80	0.80	0.80	70
Accuracy			**0.87**	**210**
Macro Avg	0.85	0.85	0.85	210
Weighted Avg	0.87	0.87	0.87	210

Table 1: Classification report for the blur detection model on the validation set.

The model shows strong performance, particularly in identifying blurred images (90% precision and recall). The high accuracy validates our approach of using an automated classifier to decide whether to apply the sharpening preprocessing step.

3.3 Qualitative OCR Results

We tested the full pipeline on a sample document image with moderate perspective skew.

Image Quality Prediction: The blur classifier correctly predicted the image as "clean" with high confidence. Consequently, the adaptive sharpening step was skipped, as intended.

```
[INFO] Document quality predicted as: clean
[ACTION] Proceeding without sharpening.
```

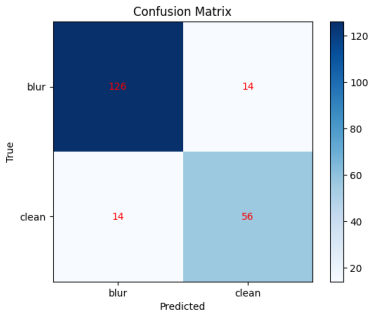

Figure 1: Confusion matrix for the blur classifier. The model correctly identified 126/140 blurred images and 56/70 clean images.

Perspective Correction: The perspective correction module successfully identified the document's four corners and produced a flattened, high-contrast grayscale image for the OCR engine, as seen in Figure 2.

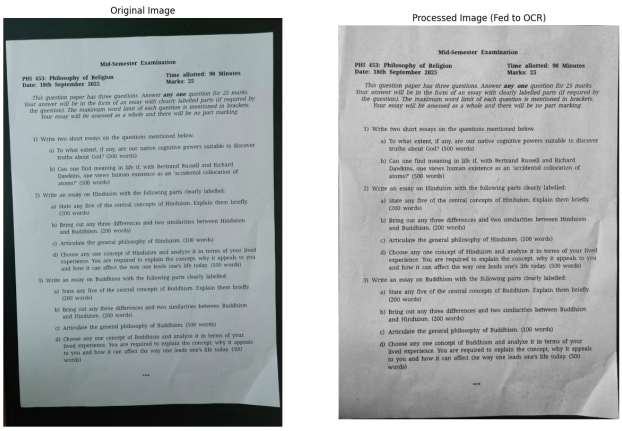

Figure 2: Qualitative results of the pre-processing stages. (Left) The original input image with perspective skew. (Right) The corrected, flattened, and contrast-enhanced image fed to the OCR engine.

Final Assembled Text: The final text output demonstrates the effectiveness of the pipeline. The text is coherent and accurately transcribed, with only minor errors on characters with low confidence scores.

In contrast, running OCR on the raw, uncorrected image would result in significant errors and fragmented text due to the skewed lines and inconsistent lighting. The structured and clean output is a clear demonstration of the pipeline's success.

4 Conclusion

We have presented an intelligent document reader that employs a novel adaptive pre-processing pipeline. By first classifying the input image quality and conditionally applying a sharpening filter, our system avoids straightforward approaches. This, combined with perspective correction and text assembly, results in a system capable of digitizing real-world, camera-captured documents. Future work could involve extending the quality classifier to detect shadows or wrinkles and incorporating a more advanced document layout analysis module to handle complex, multi-column formats.

A Appendix

A.1 Links to Resources

- **Project GitHub Repository:** https://github.com/manisha-1112/EE604_Project/tree/main
- **MobileNetV2 Paper (arXiv):** https://arxiv.org/abs/1801.04381
- **Blur Dataset:** https://www.kaggle.com/datasets/balabaskar/blur-dataset
- **EasyOCR GitHub:** https://github.com/JaidedAI/EasyOCR

A.2 Complete Project Code

The entire project was implemented within a single Jupyter Notebook. The code from each cell is provided below.

A.2.1 Cell 1 & 2: Dataset Preparation

```
import os

# Create dataset folders
base = "/kaggle/input/blur-detection"
root = "/kaggle/working/doc_quality_dataset"
os.makedirs(root, exist_ok=True)

# Create sub-directories for clean and blur
    images
for cls in ["clean", "blur"]:
    os.makedirs(f"{root}/{cls}", exist_ok=True)

print("Dataset folders created!")

import shutil
import glob

# Copy images from source dataset
source_base = "/kaggle/input/blur-dataset"

# Clean images
for f in glob.glob(f"{source_base}/sharp/*"):
    shutil.copy(f, f"{root}/clean/")
```

```

# Blurred images = motion + defocus
for f in glob.glob(f"{source_base}/motion_blurred/*"):
    shutil.copy(f, f"{root}/blur/")

for f in glob.glob(f"{source_base}/defocused_blurred/*"):
    shutil.copy(f, f"{root}/blur/")

print("Images copied successfully!")
print("Clean count:", len(os.listdir(f"{root}/clean")))
print("Blur count:", len(os.listdir(f"{root}/blur")))
```

A.2.2 Cell 3: Model Training (Blur Classifier)

```
!pip install tensorflow -q

import tensorflow as tf
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt

IMG_SIZE = (224,224)
BATCH = 32

datagen = ImageDataGenerator(
    rescale=1/255.,
    validation_split=0.2,
    rotation_range=2,
    zoom_range=0.05,
)

train_gen = datagen.flow_from_directory(
    root, target_size=IMG_SIZE, batch_size=BATCH,
    class_mode='categorical', subset='training', shuffle=True
)

val_gen = datagen.flow_from_directory(
    root, target_size=IMG_SIZE, batch_size=BATCH,
    class_mode='categorical', subset='validation', shuffle=False
)

base = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224,224,3))
for layer in base.layers:
    layer.trainable = False

x = GlobalAveragePooling2D()(base.output)
x = Dense(128, activation='relu')(x)
x = Dropout(0.3)(x)
out = Dense(2, activation='softmax')(x)

model = Model(inputs=base.input, outputs=out)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_gen, validation_data=val_gen, epochs=6)

model.save("/kaggle/working/blur_classifier.h5")
print("Model saved!")
```

A.2.3 Cell 4: Evaluation of Classifier

```
import numpy as np
from sklearn.metrics import confusion_matrix, classification_report
import itertools
import matplotlib.pyplot as plt

val_gen.reset()
preds = model.predict(val_gen)
y_pred = np.argmax(preds, axis=1)
y_true = val_gen.classes
labels = list(val_gen.class_indices.keys())

print(classification_report(y_true, y_pred, target_names=labels))

cm = confusion_matrix(y_true, y_pred)
plt.imshow(cm, cmap='Blues')
plt.title("Confusion Matrix")
plt.colorbar()
plt.xticks(range(2), labels)
plt.yticks(range(2), labels)
plt.xlabel("Predicted"); plt.ylabel("True")

for i in range(2):
    for j in range(2):
        plt.text(j, i, cm[i,j], ha='center', va='center', color='red')

plt.show()
```

A.2.4 Cell 5: Prediction Function

```
from tensorflow.keras.models import load_model
classifier = load_model("/kaggle/working/blur_classifier.h5")

label_map = {v:k for k,v in train_gen.class_indices.items()}

def predict_blur(img):
    img = cv2.resize(img, (224,224)).astype('float32')/255.
    pred = classifier.predict(np.expand_dims(img,0))[0]
    label = label_map[np.argmax(pred)]
    return label, pred
```

A.2.5 Cell 6: Pre-processing Module

```
import cv2
import numpy as np

def preprocess_and_correct_image(image):
    """
    This function finds a document in an image, corrects its perspective,
```

```
    and enhances its quality for better OCR results.
    """
    print("[INFO] Part 1: Pre-processing and correcting image perspective...")
    if image is None:
        raise FileNotFoundError(f"Image not found at path: {image_path}")

    orig_image = image.copy()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Use Gaussian blur and Canny edge detection to find document edges
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    edged = cv2.Canny(blurred, 75, 200)

    # Find the largest contours
    contours, _ = cv2.findContours(edged.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    contours = sorted(contours, key=cv2.contourArea, reverse=True)[:5]

    # Find the 4-point contour of the document
    screen_cnt = None
    for c in contours:
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.02 * peri, True)
        if len(approx) == 4:
            screen_cnt = approx
            break

    # If a 4-point contour is found, apply perspective transform
    if screen_cnt is not None:
        print("[INFO] Document contour detected. Applying perspective correction.")
        pts = screen_cnt.reshape(4, 2)
        rect = np.zeros((4, 2), dtype="float32")
        s = pts.sum(axis=1)
        rect[0] = pts[np.argmin(s)]
        rect[2] = pts[np.argmax(s)]
        diff = np.diff(pts, axis=1)
        rect[1] = pts[np.argmin(diff)]
        rect[3] = pts[np.argmax(diff)]
        (tl, tr, br, bl) = rect

        width_a = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
        width_b = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
        max_width = max(int(width_a), int(width_b))

        height_a = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
        height_b = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
        max_height = max(int(height_a), int(height_b))

        dst = np.array([[0, 0], [max_width-1, 0], [max_width-1, max_height-1], [0, max_height-1]], dtype="float32")
        M = cv2.getPerspectiveTransform(rect, dst)
        warped = cv2.warpPerspective(orig_image, M, (max_width, max_height))
    else:
        # Fallback if no contour is found
        print("[WARN] No document contour found. Using original image.")
        warped = orig_image

    # Final contrast enhancement on the corrected image
    final_gray = cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY)
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    final_image = clahe.apply(final_gray)

    return orig_image, final_image
```

A.2.6 Cell 7: Text Assembly Module

```
def assemble_text_from_results(ocr_results):
    """
    Sorts OCR results based on vertical position and joins them
    into a single coherent block of text.
    """
    print("\n[INFO] Part 2: Assembling text into correct reading order...")
    # Sort the text blocks by their vertical position (y-coordinate of the top-left corner)
    sorted_results = sorted(ocr_results, key=lambda r: r[0][0][1])

    # Join the sorted text blocks into a single string
    full_text = " ".join([text for bbox, text, prob in sorted_results])

    return full_text
```

A.2.7 Cell 8: Main Execution Pipeline

```
import easyocr
import matplotlib.pyplot as plt

# --- Configuration ---
IMAGE_PATH = '/kaggle/input/test-doc/test_doc.jpg'
image = cv2.imread(IMAGE_PATH)
label, prob = predict_blur(image)

print(f"[INFO] Document quality predicted as: {label}")

if label == "clean":
    print("[ACTION] Proceeding without sharpening.")
else:
    print("[ACTION] Applying sharpening filter...")
    image = cv2.filter2D(image, -1, np.array([[0, -1, 0],
                                              [-1, 5, -1],
                                              [0, -1, 0]]))
# --- Step 1: Pre-process the image ---
```

```
print("[ACTION] Preprocessing")
original_image, processed_image =
    preprocess_and_correct_image(image)

# --- Step 2: Run the OCR Engine on the CLEAN
    image ---
print("\n[INFO] Running EasyOCR on the processed
    image...")
reader = easyocr.Reader(['en'], gpu=False)
# Use paragraph=False to get individual text
    blocks, which gives us more control
results = reader.readtext(processed_image,
    paragraph=False)

# --- Step 3: Assemble the text (Novelty 2) ---
full_extracted_text = assemble_text_from_results(
    results)

# --- Step 4: Display the results for comparison
    ---
print("\n" + "="*50)
print("      INTELLIGENT DOCUMENT READER RESULTS"
    )
print("="*50)

print("\n--- RAW DETECTED FRAGMENTS (FOR
    DEBUGGING) ---\n")
for (bbox, text, prob) in results:
    print(f"Text: '{text}', Confidence: {prob:.4f
    }")

print("\n" + "="*50)
print("\n--- FINAL ASSEMBLED TEXT ---\n")
print(full_extracted_text)
print("\n" + "="*50)

# Display the original vs. processed image to see
     the improvement
fig, axes = plt.subplots(1, 2, figsize=(15, 10))
axes[0].imshow(cv2.cvtColor(original_image, cv2.
    COLOR_BGR2RGB))
axes[0].set_title('Original Image')
axes[0].axis('off')

axes[1].imshow(processed_image, cmap='gray')
axes[1].set_title('Processed Image (Fed to OCR)')
axes[1].axis('off')

plt.show()
```