UNIVERSIDAD EUROPEA DEL ATLÁNTICO

GRADO EN

Ingeniería Informática

Inteligencia Artificial

OCR

Trabajo realizado por :

Brenda Lopes Ventura de Souza

Docente:

José Manuel Breñosa

Introducción

El objetivo de este trabajo es desarrollar un software OCR (Optical Character Recognition) que reciba como entrada una imagen (por ejemplo .jpg o .png) sin texto embebido y devuelve como salida el texto reconocido.

El profesor pide como mínimo:

1. Reconocer texto de imprenta (digital/impreso).
2. Reconocer texto manuscrito.

Además, existe una restricción importante: no se pueden usar librerías OCR ya hechas (Tesseract o similares). Por eso, la solución se construye con:

- **Pre Procesado de imagen** (OpenCV)
  Segmentación (detectar letras en la imagen)
- **Reconocimiento** (red neuronal CNN entrenada desde cero para letras manuscritas)
- Un modo “auto” para elegir el flujo adecuado.

Mi objetivo práctico es que el programa funcione de forma estable en el caso más común: Foto de una hoja con texto centrado, en una sola línea, letra negra y fondo blanco.

Solución Aportada

Estructura final del repositorio

Carpetas principales usadas en la entrega:

- `scripts/`
    - `run_ocr.py` → script principal para ejecutar OCR en una imagen (modo auto / handwritten / print)
    - `compare_models_on_samples.py` → compara modelos manuscritos con un conjunto de samples (para justificar elección)
    - `make_handwritten_dataset.py` → genera el dataset procesado 32×32 (si se quiere reentrenar)
    - `train_handwritten_cnn.py` → entrena la CNN manuscrita (si se quiere reentrenar)
    - `predict_char.py` → prueba rápida de predicción por carácter
- `src/ocr051/`
    - `pipeline.py` → pipeline principal del OCR manuscrito (y llamada a impreso si aplica)
    - `preprocess.py` → preprocesado (binarización, máscara, normalización)
    - `segment.py` → segmentación en cajas por carácter y agrupación por líneas
    - `recognize_handwritten.py` → carga de modelo y predicción por carácter

    - `recognize_printed.py` → OCR impreso basado en plantillas (templates)
    - `io_utils.py` → utilidades de guardado/debug (imágenes intermedias, cajas, etc.)

- `models/`
    - `handwritten_char_cnn/` → **modelo manuscrito elegido** (model.pt + classes.json)
    - `handwritten_char_cnn_AZ09/` → modelo alternativo “MVP” solo mayúsculas y números
    - `print_templates.npz` → plantillas para el modo impreso

- `outputs/`
    - logs, matrices de confusión, y directorios debug generados al ejecutar el OCR

- `data/samples/`
    - imágenes de test y `gt_samples.csv` para comparar modelos

Como se ejecuta:

Instalación:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Ejecución OCR (modo automático recomendado)

```
python3 scripts/run_ocr.py \
  --mode auto \
  --image RUTA_A_LA_IMAGEN.jpg \
  --handwritten-model-dir models/handwritten_char_cnn \
  --min-box-area 120 \
  --debug-dir outputs/debug_examen
```

`--mode auto` intenta decidir si aplicar OCR manuscrito o impreso.

`--min-box-area` sube/baja el filtro contra suciedad (ruido). En mis pruebas 120 funciona bien con hojas limpias.

`--debug-dir` guarda todas las imágenes intermedias para entender errores.

Forzar modo manuscrito

```
python3 scripts/run_ocr.py \
  --mode handwritten \
  --image RUTA_A_LA_IMAGEN.jpg \
  --handwritten-model-dir models/handwritten_char_cnn \
  --min-box-area 120 \
  --debug-dir outputs/debug_hand
```

Forzar modo impreso

```
python3 scripts/run_ocr.py \
  --mode print \
  --image RUTA_A_LA_IMAGEN.jpg \
  --debug-dir outputs/debug_print
```

Requisitos asumidos del input

Para maximizar la estabilidad (especialmente en el examen), defino estas condiciones recomendadas:

- Una línea principal de texto, preferiblemente centrada.
- Letras claras y separadas (no muy pegadas).
- Fondo blanco o claro, tinta oscura.
- Evitar sombras fuertes y reflejos.
- Imagen sin inclinación extrema (ligera inclinación se tolera, pero no rotaciones grandes).

Esto no significa que el sistema no funcione fuera de estas condiciones, pero el rendimiento baja porque la segmentación se vuelve más difícil.

OCR manuscrito (pipeline)

El OCR manuscrito se construye en 4 pasos:

Paso 1: Preprocesado

- Se convierte la imagen a escala de grises.
- Se crea una **máscara binaria** (tinta vs fondo) con umbral robusto.
- Se generan imágenes intermedias para debug (`gray`, `mask`).

Objetivo: que la tinta quede bien separada, porque todo lo demás depende de esto.

Paso 2: Segmentación en caracteres

- Con la máscara se detectan regiones candidatas a letras (cajas: x,y,w,h).
- Se filtran cajas pequeñas con `min_box_area` para eliminar polvo/suciedad.
- Se agrupan las cajas en líneas (por coordenada y).

Paso 3: Normalización de cada carácter a 32×32

Cada carácter detectado se recorta y se normaliza para que la red neuronal reciba un input estándar:

- padding para hacerlo cuadrado
- resize a 32×32
- tinta “negra” sobre fondo “blanco”

Paso 4: Reconocimiento con CNN (clasificador por carácter)

Se usa una CNN simple entrenada desde cero. A nivel conceptual:

- Entrada: imagen 1×32×32
- Varias capas convolucionales para extraer rasgos
- Capa final con **softmax** → probabilidad por clase
- La **confianza por carácter** se toma como la probabilidad máxima

El resultado final es:

- **Texto** (concatenación de caracteres detectados)
- **Confianza global** (media de confianzas por carácter)
- **Detalle por carácter** con su porcentaje

Ejemplo real de salida:

- Texto: `BUENAS`
- Confianza global: `91.4%`
- Detalle: `B(100%) U(100%) E(93%) N(100%) A(100%) s(56%)`

OCR impreso (caracteres digitales)

Para el modo impreso, se implementa una aproximación basada en **plantillas** (“templates”), guardadas en `models/print_templates.npz`. La idea es:

1. Preprocesar y segmentar caracteres como en manuscrito.
2. Comparar cada recorte con plantillas conocidas (matching).
3. Elegir el carácter más similar.

Clases objetivo en impreso: mayúsculas y números.
Limitación principal: si la tipografía o el tamaño cambian mucho, o si el recorte no sale limpio, el matching empeora.

Análisis de rendimiento y justificación de la elección del modelo

Modelos manuscritos probados

Se entrenaron y/o guardaron varias versiones de modelo manuscrito (v1, v2, v3). Para elegir el mejor, hice dos cosas:

1. **Comparación en samples reales** (fotos que yo hice) usando:
    - `scripts/compare_models_on_samples.py`
    - `data/samples/gt_samples.csv`
2. **Revisión de casos difíciles ("hard examples")** para entender errores típicos:
    - confusiones entre caracteres parecidos: `0/O`, `1/I/l`, `S/5`, etc.
    - problemas por sombras o recortes defectuosos (por eso añadí control de calidad en el procesado del dataset)

Resultado de entrenamiento (referencia)

- En el modelo completo de 62 clases (A–Z, a–z, 0–9), el rendimiento de validación llegó aproximadamente a **~69.9%** en el mejor caso, lo cual es razonable para un dataset limitado y con clases parecidas.
- En el MVP AZ09 (A–Z + 0–9), se llegó a una accuracy de validación superior (ejemplo: ~0.7766), porque hay menos clases y menos ambigüedad.

Resultado en mis samples de prueba

Para justificar qué modelo me quedo en `models/handwritten_char_cnn/`, ejecuté una comparación y medí:

- exact match (si predice exactamente el texto)
- CER (Character Error Rate): distancia de edición / longitud real
- confianza global media

En mis pruebas, el modelo que mejor se comportó en promedio fue el que dejé como **modelo principal manuscrito** en: `models/handwritten_char_cnn/`

Además, con el parámetro **`--min-box-area 120`** conseguí mejorar mucho el resultado al eliminar suciedad detectada como "caracteres".

Registro de resultados (logbook)

Aquí voy anotando qué probé, qué pasó y qué decisión tomé. Me ayudó mucho a no perderme y a justificar por qué el proyecto acabó como está ahora.

20.12.2025 — MVP manuscrito (A–Z + 0–9)

Objetivo
Antes de intentar reconocer **62 clases** (A–Z, a–z, 0–9), decidí hacer primero un MVP más "seguro" con **36 clases (A–Z + 0–9)**. La idea era reducir confusiones típicas (por ejemplo `i/I/1/l`, `o/O/0`, `s/S/5`) y asegurar que el OCR manuscrito funcionara en algo realista.

Acción / Prueba

1. Generé un dataset filtrado AZ09 a partir del dataset procesado completo:

- Origen: `data/processed/handwritten_chars/labels.csv`
- Destino: `data/processed/handwritten_AZ09/` (`images/` + `labels.csv`)
- Filtro: etiquetas que cumplan `^[A-Z0-9]$`

  Resultado del filtrado: 1364 imágenes, 36 clases

2. Entrené una CNN simple desde cero para ese caso AZ09.

Entrenamiento (comando)

```
python3 scripts/train_handwritten_cnn.py --data-dir
data/processed/handwritten_AZ09 --model-dir
models/handwritten_char_cnn_AZ09 --out-dir outputs/AZ09 --epochs 20
--batch-size 64
```

Modelo

- Entrada 32×32 en grises
- 3 bloques Conv+ReLU+MaxPool
- Fully connected con dropout
- Aumentación suave (rotación pequeña, traslación y erosión/dilatación ligera)

Resultados

- Device: `mps` (Mac)
- Mejor accuracy de validación: **0.7766**
- Archivos guardados:
  - `models/handwritten_char_cnn_AZ09/model.pt`
  - `models/handwritten_char_cnn_AZ09/classes.json`
  - `outputs/AZ09/train_log.csv`

- `outputs/AZ09/confusion_matrix.png`

Decisión
El MVP fue útil porque me confirmó que el pipeline general (dataset → CNN → predicción) funcionaba. Pero todavía aparecían errores “raros” en algunas letras, y no era solo culpa del modelo: parecía más un problema del **preprocesado**.

20.12.2025 — QA (hard examples) y detección del problema real

Objetivo: Entender *por qué* el modelo fallaba: si era falta de datos, mala segmentación o imágenes mal normalizadas.

Acción / Prueba: Usé una herramienta de QA para ver ejemplos donde el modelo se equivoca pero con mucha confianza (esto suele señalar un error de datos, no del modelo).

Ejemplo de comando: `scripts/qa_hard_examples.py --only-wrong --k 30 --open`

Observaciones

- Encontré muestras normalizadas que salían casi negras, como “bloques” o “barras”.
- Esto indicaba que, en algunos casos, el recorte estaba capturando sombra/borde/fondo como si fuera tinta.
- Confusiones frecuentes detectadas
  - `0/O`
  - `Z/2`
  - `1/I/l`
  - `N/M`
  - `V/W`
  - `6/U/G`

Decisión: Antes de “tocar” más la red, tenía sentido arreglar lo que alimentaba a la red: el preprocesado del dataset. Si entrenas con ejemplos sucios, la red aprende basura.

03.01.2026 — Mejora del preprocesado y reconstrucción del dataset (62 clases)

Objetivo: Conseguir un dataset de 32×32 **realmente limpio y consistente**, donde cada imagen contenga el carácter centrado y no un trozo de sombra o borde.

Cambio aplicado en `make_handwritten_dataset.py`
Hice el preprocesado más robusto en tres ideas:

1. **Binarización más estable**

- probando Otsu en dos polaridades (para evitar que el fondo se convierta en “tinta”)
- eligiendo la mejor opción según el ratio de tinta

2. **Connected Components (componente principal)**

    - en vez de recortar todo lo “blanco” de la máscara, me quedo con el componente conectado más “probable” del carácter
    - así se reduce que una sombra grande domine el recorte

3. **Control de calidad (QC)**

    - rechazo de outliers: demasiada tinta, muy poca tinta o bbox demasiado pequeña
    - opción de guardar rechazados para revisar

Prueba / Ejecución: Reconstruí el dataset con el nuevo preprocesado, guardando los rechazados para poder inspeccionarlos.

Resultado

- Se eliminaron muestras corruptas con QC (por ejemplo: **90 rechazadas** en una de las reconstrucciones).
- Visualmente, el “contact sheet” se veía mucho mejor: caracteres centrados, sin bloques negros raros.

Decisión: Con el dataset ya limpio, tenía sentido volver a entrenar el modelo completo (62 clases), porque ahora el entrenamiento iba a ser más “justo”.

03.01.2026 / 05.01.2026— Entrenamiento modelo completo (62 clases: 0–9, A–Z, a–z)

Objetivo: Tener un modelo manuscrito completo que reconozca:

- dígitos `0–9`
- mayúsculas `A–Z`
- minúsculas `a–z`

Entrenamiento: Se entrenó el modelo final sobre el dataset limpio (32×32).

Resultado (resumen)

- Mejor precisión de validación aproximada: **69.9%** con 62 clases.
- Es un problema más difícil que AZ09, y es normal que baje la precisión: hay muchas clases visualmente parecidas.

Observación importante: Los errores más frecuentes siguen siendo entre caracteres muy parecidos (por ejemplo `1/I/l`, `O/0`, `Z/2`). Esto es coherente con OCR manuscrito y además depende mucho de la caligrafía y la foto.

06.01.2026 / 07.01.2026 — Integración del OCR "de foto" (pipeline completo) y control de ruido

Objetivo: Que el programa funcionara como pide el profesor: Escribir en una hoja y que el OCR lo entienda.

Acción / Mejora clave: Ajusté la segmentación para ignorar suciedad:

- se añadió un filtro por área mínima de caja: `--min-box-area`
- con valores como **120**, se redujo muchísimo el ruido (puntos y manchas detectados como letras)

Resultado
En pruebas reales, el OCR devolvía:

- Texto reconocido
- Confianza global (%)
- Confianza por carácter

Esto me permitió depurar: si un carácter sale con 50–70% ya sé dónde mirar.

10.01.2026 / 11.01.2026 — Comparación de modelos (v1 / v2 / v3) y elección final

Objetivo: Elegir qué modelo manuscrito dejar como "principal" para el examen.

Acción: Probé distintos modelos con un conjunto de samples reales (`data/samples/`) y un CSV de ground truth.

Métrica usada:

- exact match (si acierta todo el texto)
- CER (Character Error Rate)
- confianza global media

Resultado: En mis fotos y condiciones de examen (una línea clara), el modelo que mejor se comportó en promedio fue **v1**, y por eso lo dejé como principal:

Modelo manuscrito final usado por defecto: `models/handwritten_char_cnn/` (model.pt + classes.json)

También mantengo: `models/handwritten_char_cnn_AZ09/` como MVP robusto por si se quiere demostrar versión simplificada

Decisión final para entrega

- Me quedo con un pipeline claro y estable para el manuscrito.
- Mantengo OCR impreso como segundo modo, pero el foco del proyecto y de mis pruebas es el manuscrito desde foto.

Conclusiones y mejoras

Lo que funciona bien

- OCR manuscrito en condiciones controladas (hoja limpia, una línea, letras claras).
- Salida clara para el usuario: **texto + confianza global + detalle por carácter**.
- Sistema de debug completo: guarda máscaras, cajas y crops, lo que ayuda mucho a corregir errores.

Limitaciones actuales

- Confusiones típicas entre caracteres visualmente parecidos (`0/O`, `1/I/l`, `S/5`, etc.).
- Si hay suciedad o sombras fuertes, la segmentación detecta cajas falsas.
- El OCR impreso por plantillas depende bastante de la tipografía y de que el recorte salga limpio.

Posibles mejoras futuras

- Aumentar dataset real (más letras escritas por distintas personas).
- Añadir corrección por “diccionario” o lenguaje (postprocesado) para palabras comunes.
- Mejorar el modo impreso:
    - más plantillas por fuente/tamaño
    - normalización más estricta antes del matching
- Mejorar segmentación cuando hay varias líneas o texto inclinado.