**Resumen del trabajo**

**1) Preparación del dataset**

* **Fuente**: carpeta única con imágenes y .json (los .json tienen el mismo nombre que su imagen defectuosa).
* **Limpieza y organización**:
  + Separamos en normales/ (0) y defectuosas/ (1) usando la presencia/ausencia del .json.
  + Generamos labels.csv con pares filename,label.
* **Splits (one-class para entrenar)**:
  + train/: **solo normales** (70% de las normales).
  + val/: normales + **parte** de defectuosas (15% normales + ~40% defectuosas).
  + test/: normales + **resto** de defectuosas (15% normales + ~60% defectuosas).
  + Cada split incluye su labels.csv.

**2) Modelos de IA probados**

**A) Autoencoder (AE) — aprendizaje de “normalidad”**

* **Idea**: aprender a reconstruir piezas **sanas**. Las anomalías dan mayor **error de reconstrucción**.
* **Arquitectura**: Conv-AE ligero (encoder 3 capas conv, decoder 3 capas deconv).
* **Entrenamiento**:
  + Datos: **solo normales** de train/.
  + Tamaño imagen: **256×256** (también probamos 500×500, más costoso).
  + Pérdida: MSE (con soporte opcional a MS-SSIM).
  + Trucos: augmentación ligera + **Top-K error** (promedia el 2% de los píxeles con mayor error → mejora defectos locales).
  + **Umbral**: calibrado en val/ maximizando **F1** (PR-curve).
* **Resultados (ejemplos en test/)**:
  + Versión base: **ROC-AUC ≈ 0.68, F1 ≈ 0.64**.
  + Versión mejorada (Top-K + tweaks): **AUC ≈ 0.74, F1 ≈ 0.67**.
* **Conclusión**: funciona, pero quedó **por debajo** de PatchCore en este dataset.

**B) PatchCore (SOTA en anomalías sin supervisión)**

* **Idea**: extraer **features** de una red preentrenada y modelar la “normalidad” con un **banco de parches**.
* **Backbone**: ResNet18 (capas layer2 + layer3), **IMG\_SIZE=256**.
* **Banco de memoria**:
  + Extraemos parches normalizados L2 de **normales (train)**.
  + **Coreset 5%** (proyección aleatoria + k-center greedy) para hacerlo **ligero y rápido**.
* **Detección**:
  + Para cada imagen: KNN (k=3–5) sobre el banco; mapa de distancias → **score** (máximo del mapa).
  + **Umbrales** desde val/:
    - **p95 de normales** ≈ **0.488** (muy bajo FP).
    - **TPR≈95%** ≈ **0.438** (más sensible).
* **Resultados (test)**:
  + Con calibración del notebook: **ROC-AUC ≈ 0.95**, **F1 ≈ 0.92**  
    (CM: [[23 1],[3 24]] con umbral ~0.49/0.44).
  + En una variante, con umbral 0.488: **~98% accuracy** ([[24 0],[1 26]]).
* **Conclusión**: **mejor desempeño** que AE en precisión/robustez con muy poco entrenamiento.

**3) Backend (API) para inferencia**

* **Stack**: FastAPI + Uvicorn (CUDA cuando está disponible).
* **Artefactos**:
  + models/patchcore/memory\_bank\_core.npz (banco coreset).
  + models/patchcore/config.json (umbrales de validación).
* **Config por variables de entorno** (o .env):
  + IMG\_SIZE=256 (coherente con el banco).
  + KNN\_K (típico 3–5; 3 más sensible).
  + THRESHOLD (p.ej. **0.44** en producción para balance P/R).
  + PERCENT y AREA\_MIN (solo visual: filtran máscaras y polígonos).
  + SAVE\_VIS=1 (guarda overlay y heatmap en static/).
* **Endpoints**:
  + POST /predict: recibe imagen, devuelve:
  + {
  + "score": <float>,
  + "threshold": <float>,
  + "is\_anomaly": true/false,
  + "polygons": [...], // solo si is\_anomaly=true
  + "overlay\_url": "/static/..."// PNG con overlay
  + }

Parámetros opcionales:

* + - thr=<num> para umbral puntual
    - mode=sensitive|strict (0.8× / 1.2× del umbral base)
  + GET /health: devuelve configuración activa (umbral, KNN, img\_size, etc.).
* **Calibración práctica**:
  + Elegimos **THRESHOLD ≈ 0.44** tras probar normales (scores ~0.36–0.38) y defectuosas (scores ~0.43–0.45).
  + Si priorizas menos FPs: **0.48–0.50**; si priorizas recall: **0.42–0.44** o mode=sensitive.

**4) ¿Por qué elegimos PatchCore para producción?**

* **Rinde mejor** en este dataset (AUC≈0.95, F1≈0.92 vs AE ~0.64–0.67 F1).
* **Sin entrenamiento pesado**: solo extrae features + KNN; el **coreset** lo hace ágil.
* **Calibración simple** y **umbral interpretable** (score de distancia).
* **Explainability** útil (heatmap + polígonos) para inspección visual.

**5) Notas de sostenibilidad (breve)**

* PatchCore evita “entrenamientos profundos” largos → **menos energía**.
* **Coreset 5%** reduce memoria y cómputo en inferencia → **eficiencia**.
* **Umbral ajustable** reduce rechazos falsos (menos retrabajos y desperdicio).
* El backend permite **contenerizar** y ejecutar en GPU/CPU según disponibilidad.