ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

EVALUACIÓN COMPARATIVA DE MÉTODOS DE DETECCIÓN DE ANOMALÍAS EN IMÁGENES MÉDICAS DE ALTA RESOLUCIÓN

COMPARATIVE EVALUATION OF ANOMALY DETECTION METHODS IN HIGH-RESOLUTION MEDICAL IMAGES

Realizado por

Carlos Pino Padilla

Tutorizado por

Francisco de Asís Fernández Navarro

Departamento

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2025

Resumen

Este Trabajo de Fin de Grado explora la detección de anomalías en radiografías de tórax mediante técnicas de aprendizaje auto-supervisado (SSL). La falta de imágenes médicas etiquetadas de forma manual, debido al alto coste y tiempo que requiere, limita la aplicación de modelos supervisados tradicionales. Ante este reto, se propone el uso de modelos que aprenden directamente a partir de datos no etiquetados, sin necesidad de anotaciones externas.

El objetivo principal del proyecto es analizar, entrenar y evaluar varios enfoques auto-supervisados aplicados a imágenes médicas, comparando sus resultados para extraer conclusiones sobre su rendimiento y utilidad. Para ello, se han implementado tres modelos representativos: AutoEncoder (basado en reconstrucción), SimCLR (aprendizaje contrastivo) y SWSSL (que introduce ventanas deslizantes para conservar la resolución local). Todos han sido entrenados exclusivamente con imágenes normales, simulando un entorno real sin supervisión.

El conjunto de datos utilizado es Chest X-ray, y la evaluación se ha realizado aplicando métricas clínicas estándar como AUC, F1-score, precisión, sensibilidad y exactitud. Los resultados muestran que, aunque ningún modelo alcanza un nivel clínico, todos logran detectar diferencias entre imágenes normales y patológicas. Además, se identifican qué ajustes o estrategias han contribuido a mejorar el rendimiento y qué limitaciones computacionales han influido.

Este trabajo no solo aporta una visión clara del potencial del aprendizaje auto-supervisado en medicina, sino que también ofrece un análisis comparativo útil entre diferentes estrategias. Todo el código ha sido documentado y organizado con vistas a su reutilización.

Palabras clave: aprendizaje auto-supervisado, detección de anomalías, imágenes médicas, entrenamiento de modelos, evaluación comparativa.

Abstract

This Final Degree Project explores anomaly detection in chest X-rays using self-supervised learning (SSL) techniques. The lack of manually labeled medical images, due to the time and expertise required, makes the use of traditional supervised models difficult to scale. To address this challenge, the project proposes training models that learn directly from unlabeled data, without relying on external annotations.

The main objective is to analyze, train, and evaluate several self-supervised approaches applied to medical images, and compare their results to assess performance and practical value. Three representative models have been implemented: AutoEncoder (reconstruction-based), SimCLR (contrastive learning), and SWSSL (which uses sliding windows to preserve local resolution). All models were trained using only normal images, simulating a real unsupervised scenario.

The dataset used is Chest X-ray, and evaluation was carried out using standard clinical metrics such as AUC, F1-score, precision, recall, and accuracy. Results show that, although none of the models reach clinical-grade performance, all of them are able to detect differences between normal and pathological images. The study also identifies what adjustments and strategies have helped improve the models and which technical limitations affected development.

This project provides a clear overview of the potential of self-supervised learning in the medical field and offers a practical and comparative analysis of different methods. All code developed has been documented and structured to support future reuse.

Keywords: self-supervised learning, anomaly detection, medical imaging, model training, comparative evaluation.

Índice

Índice de Figuras

Índice de tablas

1. Introducción

El uso de técnicas de inteligencia artificial en el ámbito médico ha experimentado un crecimiento exponencial en los últimos años. Una de las áreas más prometedoras es la detección automática de anomalías en imágenes médicas, una tarea que resulta crítica en el diagnóstico clínico. La escasez de datos etiquetados ha motivado el desarrollo de enfoques como el aprendizaje auto-supervisado (SSL), que permite entrenar modelos sin necesidad de anotaciones.

Este trabajo propone una evaluación comparativa de distintos métodos SSL aplicados a imágenes de alta resolución, con el objetivo de analizar su desempeño y aplicabilidad. Se toma como base el método SWSSL, complementado con otros enfoques relevantes como AutoEncoder y SimCLR. La comparación se lleva a cabo utilizando el conjunto de datos Chest X-ray, siguiendo un protocolo experimental riguroso y evaluando mediante métricas estandarizadas.

El documento se estructura de la siguiente forma: en el capítulo 2 se expone el glosario de términos y acrónimos, en el capítulo 3 se presentan los fundamentos teóricos y el estado del arte; en el capítulo 4, la metodología empleada; en el capítulo 5, el desarrollo e implementación; en el capítulo 6, los resultados obtenidos; y en el capítulo 7, las conclusiones y líneas futuras.

1.1 Motivación

La detección de anomalías en imágenes médicas constituye una tarea esencial en el diagnóstico clínico asistido por inteligencia artificial. La posibilidad de automatizar este proceso no solo puede mejorar la eficiencia y precisión del diagnóstico, sino también contribuir a la reducción de errores humanos, especialmente en contextos con alta carga asistencial o escasez de especialistas.

Uno de los principales retos en este ámbito es la limitada disponibilidad de datos etiquetados de calidad. La anotación de imágenes médicas requiere la intervención de profesionales altamente cualificados y tiempo considerable, lo que dificulta la escalabilidad de los enfoques supervisados convencionales. Esta problemática ha impulsado el interés en metodologías que no dependan de anotaciones manuales extensas.

Entre estas metodologías, el aprendizaje auto-supervisado (Self-Supervised Learning, SSL) ha emergido como una alternativa prometedora. Esta técnica permite entrenar modelos a partir de datos no etiquetados, utilizando tareas pretextuales para aprender representaciones útiles que posteriormente se pueden aplicar a tareas como clasificación o detección. En el contexto médico, esta capacidad resulta especialmente valiosa por la abundancia de imágenes disponibles sin etiquetar.

Este Trabajo de Fin de Grado parte de esta premisa para explorar y comparar el rendimiento de distintos métodos auto-supervisados aplicados a la detección de anomalías en imágenes médicas de alta resolución, con el objetivo de evaluar su viabilidad técnica y aplicabilidad práctica.

1.2 Objetivos

El objetivo principal de este trabajo es realizar una evaluación comparativa de distintos métodos de aprendizaje auto-supervisado para la detección de anomalías en imágenes médicas de alta resolución, utilizando como referencia el método SWSSL.

Objetivos específicos:

- Estudiar el estado del arte en técnicas auto-supervisadas aplicadas a la detección de anomalías.
- Implementar el método SWSSL y comprender sus componentes clave, como el uso de ventanas deslizantes.
- Integrar e implementar métodos alternativos, como AutoEncoder (basado en reconstrucción) y SimCLR (basado en aprendizaje contrastivo).
- Estandarizar el preprocesamiento y configuración experimental para garantizar comparabilidad entre modelos.
- Evaluar cuantitativamente el rendimiento de cada modelo utilizando métricas como AUC, F1-score, accuracy, sensibilidad y especificidad.

- Analizar los resultados obtenidos e identificar ventajas, limitaciones y potencial de aplicación clínica.

1.3 Metodología

El desarrollo de este Trabajo de Fin de Grado ha seguido el enfoque **CRISP-DM (Cross Industry Standard Process for Data Mining)**, una metodología ampliamente utilizada en proyectos de ciencia de datos e inteligencia artificial por su carácter iterativo, estructurado y adaptable. Esta metodología consta de seis fases: comprensión del negocio, comprensión de los datos, preparación de los datos, modelado, evaluación e implementación. A continuación, se detalla su aplicación concreta al presente trabajo.

En la fase de comprensión del problema, se ha definido el objetivo general del proyecto: evaluar comparativamente distintos métodos de aprendizaje auto-supervisado aplicados a la detección de anomalías en imágenes médicas de alta resolución. Posteriormente, en la fase de comprensión de los datos, se ha explorado el conjunto de datos Chest X-ray, analizando su formato, resolución, características clínicas y equilibrio de clases.

Durante la preparación de los datos, se han aplicado técnicas de preprocesamiento específicas como la normalización, recorte de bordes, inversión de intensidades y eliminación de artefactos, con el fin de adaptar las imágenes al flujo de entrenamiento. La fase de modelado ha incluido la implementación e integración de tres enfoques: SWSSL como propuesta base, AutoEncoder como modelo reconstrucción y SimCLR como método contrastivo. Todos los modelos han sido entrenados con imágenes normales para simular el entorno real de detección de anomalías sin supervisión directa.

En la fase de evaluación, se ha realizado un análisis comparativo de los modelos mediante métricas estándar como AUC, F1-score, accuracy, sensibilidad y especificidad. Finalmente, en la fase de implementación, se han documentado y validado los experimentos, garantizando su reproducibilidad y preparando el código para futuras ampliaciones o aplicaciones clínicas.

El uso de CRISP-DM ha permitido estructurar el proyecto de forma clara, manteniendo una coherencia entre las decisiones técnicas y los objetivos del trabajo, y facilitando el análisis crítico de los resultados.

Fase CRISP-DM	Aplicación en este trabajo
Comprensión del negocio	Definir el objetivo del TFG: evaluar métodos auto-supervisados para la detección de anomalías en imágenes médicas.
Comprensión de los datos	Análisis del dataset Chest X-ray: resolución, tipo de patologías, distribución de clases, formato, calidad de imagen.
Preparación de los datos	Preprocesamiento de las imágenes: normalización, transformaciones, augmentations específicas, limpieza y corte de bordes.
Modelado	Implementación de SWSSL, AutoEncoder y SimCLR. Ajuste de arquitecturas, funciones de pérdida y optimizadores.
Evaluación	Medición de rendimiento con AUC, F1-score, accuracy, sensibilidad y especificidad. Comparación de modelos en igualdad de condiciones.
Implementación	Documentación del código y experimentos. Preparación para reproducibilidad. Reflexión final sobre aplicabilidad y posibles mejoras.

Tabla 1: Aplicación de la metodología CRIPS-DM al flujo de trabajo del TFG

1.4 Estructura del Trabajo

El presente documento se organiza en seis capítulos principales, además del resumen, el abstract, la bibliografía y los apéndices. A continuación, se describe brevemente el contenido de cada uno de ellos:

Capítulo 1. Introducción: se presentan la motivación del trabajo, los objetivos generales y específicos, la metodología empleada y la estructura del documento.

Capítulo 2. Glosario de términos y acrónimos: se recogen los términos técnicos y acrónimos utilizados a lo largo del Trabajo de Fin de Grado, con el fin de facilitar la comprensión del contenido por parte del lector.

Capítulo 3. Fundamentos teóricos y estado del arte: se explican los conceptos clave necesarios para contextualizar el trabajo. Asimismo, se revisan trabajos recientes y enfoques representativos en el ámbito de la visión por computador médica. Se analiza lo que ya existe y se responde a la necesidad de justificar la propuesta.

Capítulo 4. Metodología: se detallan el conjunto de datos utilizado, las técnicas de preprocesamiento aplicadas, las arquitecturas de los modelos implementados y el protocolo de evaluación seguido.

Capítulo 5. Desarrollo e implementación: se describen las etapas de implementación de los distintos modelos, la estructura del código, las decisiones técnicas adoptadas y la configuración del entorno experimental.

Capítulo 6. Resultados y análisis: se presentan los resultados obtenidos mediante la ejecución de los experimentos, se comparan las métricas de rendimiento entre los distintos modelos y se realiza un análisis crítico de los hallazgos.

Capítulo 7. Conclusiones y líneas futuras: se recogen las conclusiones principales del trabajo, se valoran las aportaciones realizadas y se proponen posibles extensiones o mejoras a explorar en el futuro.

Además, el documento incluye una sección de bibliografía con todas las referencias utilizadas y un apéndice final que recogen fragmentos de configuraciones, instrucciones de uso y otros elementos complementarios.

2. Glosario de términos y acrónimos

AutoEncoder (AE): Arquitectura neuronal que aprende a comprimir y reconstruir datos de entrada. Se utiliza para detectar anomalías mediante el error de reconstrucción.

SimCLR: Framework de aprendizaje auto-supervisado basado en contraste entre vistas aumentadas de una imagen.

SWSSL: Sliding Window Self-Supervised Learning. Método basado en ventanas deslizantes y aprendizaje contrastivo para detectar anomalías en imágenes de alta resolución.

SSL (Self-Supervised Learning): Aprendizaje auto-supervisado. Técnica que permite entrenar modelos sin etiquetas mediante tareas pretextuales.

CNN (Convolutional Neural Network): Red neuronal convolucional, especializada en procesar datos con estructura espacial como imágenes.

Latent space: Espacio intermedio donde se codifican las representaciones comprimidas de los datos.

Embedding: Vector que representa de forma densa y numérica una entrada en el espacio latente.

Overfitting (Sobreajuste): Situación en la que un modelo aprende demasiado los datos de entrenamiento, perdiendo capacidad de generalización.

Batch size: Número de muestras procesadas simultáneamente durante una iteración del entrenamiento del modelo.

Augmentation: Técnica que modifica los datos de entrada para crear nuevas variantes y mejorar la robustez del modelo.

Barlow Twins: Función de pérdida contrastiva que reduce la redundancia entre las representaciones aprendidas de dos vistas.

Continuity Preserving Loss (CP Loss): Pérdida adicional usada en SWSSL para garantizar coherencia entre representaciones de parches vecinos.

ROC-AUC: Área bajo la curva ROC. Mide la capacidad del modelo para distinguir entre clases.

Puntuación continua (score): Valor numérico generado por un modelo que refleja la probabilidad, distancia o grado de pertenencia a una clase.

F1-score: Media armónica entre precisión y sensibilidad. Utilizada para evaluar modelos desequilibrados.

Precision: Proporción de verdaderos positivos sobre el total de predicciones positivas.

Recall (Sensibilidad): Proporción de verdaderos positivos detectados sobre el total de positivos reales.

Specificity (Especificidad): Proporción de verdaderos negativos detectados sobre el total de negativos reales.

Accuracy: Proporción total de predicciones correctas sobre el total de muestras evaluadas.

Benchmark: Proceso de evaluación comparativa entre múltiples modelos bajo condiciones controladas.

Checkpoint: Archivo que guarda los pesos entrenados de un modelo neuronal en un punto determinado del entrenamiento.

Sliding window: Técnica que consiste en recorrer una imagen dividiéndola en regiones solapadas (ventanas) para su análisis local.

Preentrenamiento: Fase inicial de entrenamiento de un modelo sobre un conjunto de datos general antes de ajustarlo a una tarea específica.

Fine-tuning: Ajuste fino de un modelo previamente preentrenado, adaptándolo a una tarea concreta con un conjunto de datos específico.

Latencia: Tiempo que transcurre entre la entrada de datos al modelo y la obtención de la salida. Relevante en aplicaciones en tiempo real.

Parche (Patch): Fragmento o región rectangular extraída de una imagen, utilizado frecuentemente en modelos como SWSSL o PatchCore.

Dropout: Técnica de regularización en redes neuronales que consiste en desactivar aleatoriamente un porcentaje de neuronas durante el entrenamiento para prevenir el sobreajuste y mejorar la generalización del modelo.

Curva de pérdida: Representación gráfica de cómo disminuye el error del modelo a lo largo de las épocas de entrenamiento. Refleja la capacidad de aprendizaje del modelo.

Learning Rate (lr): Tasa que controla cuánto se ajustan los pesos del modelo en cada paso del entrenamiento.

MSELoss (Error cuadrático medio): Función de pérdida que mide la diferencia promedio al cuadrado entre la imagen original y la reconstruida.

SGD: Algoritmo de optimización basado en el gradiente estocástico. Puede incorporar momentum para estabilizar la convergencia.

Epoch: Una pasada completa por todo el conjunto de entrenamiento.

Normalización: Proceso de transformación de los valores de entrada para que estén en un rango determinado (normalmente [0,1] o [-1,1]. En redes neuronales, se utiliza para mejorar la

estabilidad del entrenamiento y facilitar la convergencia, reduciendo la variación entre características de la entrada.

Normalización L2: Transformación que ajusta un vector para que tenga longitud 1, útil en espacios métricos como el de embeddings para comparar direcciones más que magnitudes.

Análisis continuo: Evaluación basada en métricas sobre scores numéricos (como AUC), sin necesidad de convertirlos en etiquetas binarias.

Análisis binario: Evaluación que requiere clasificar cada muestra como normal o anómala mediante un umbral, permitiendo calcular métricas como precisión, recall o F1-score.

Tasa de falsos positivos: Métrica derivada que indica cuántas imágenes normales son incorrectamente clasificadas como anómalas.

Batch Normalization: Técnica que normaliza la salida de una capa para estabilizar y acelerar el entrenamiento.

Early Stopping: Técnica de regularización que consiste en detener el entrenamiento de un modelo cuando la pérdida de validación deja de mejorar durante un número determinado de épocas consecutivas. Su objetivo es evitar el sobreajuste.

Validación cruzada: Procedimiento de evaluación que divide el conjunto de datos en varios subconjuntos. El modelo se entrena en alguno de ellos y se valida en los restantes, repitiendo el proceso varias veces. Reduce la dependencia de una única partición de los datos.

Multi-Layer Perceptron (MLP): Red neuronal artificial compuesta por múltiples capas totalmente conectadas.

Projection Head: Bloque final del modelo SimCLR que transforma el vector de características latentes en un nuevo vector más adecuado para la optimización contrastiva.

ResNet18: Arquitectura convolucional residual con 18 capas, utilizada como encoder. Permite un entrenamiento más estable en redees profundas gracias a conexiones residuales que evitan el problema del gradiente desvanecido.

Vector latente: Representación compacta de una imagen generada por una red neuronal. Resume la información semántica relevante del input.

Downstream task: Tarea secundaria o posterior para la que se reutiliza una representación aprendida previamente (por ejemplo, clasificación, segmentación o detección de anomalías).

NT-Xent Loss: Normalized Temperature-scaled Cross Entropy Loss. Función de pérdida contrastiva usada en SimCLR. Acercar en el espacio latente los vectores de ejemplos similares (pares positivos) y alejar los de ejemplos distintos(pares negativos).

Vanishing Gradient (Gradiente desvanecido): Problema que ocurre durante el entrenamiento de redes neuronales profundas, cuando los gradientes que se propagan hacia las capas iniciales tienen a ser extremadamente pequeños. Esto impide que esas capas aprendan de forma efectiva, ralentizando o incluso parando el proceso de optimización.

Continuity Preserving Loss: Función de pérdida propuesta para forzar que representaciones de patches adyacentes sean similares, emulando continuidad sin necesidad de hacer resize.

Twin Loss: Forma general de pérdida contrastiva donde se utilizan pares de imágenes transformadas (vistas) para aprender representaciones robustas e invariantes.

Pares positivos / negativos (aprendizaje contrastivo): Término usado para referirse a relaciones entre imágenes durante el entrenamiento auto-supervisado. Un par positivo está compuesto por dos vistas distintas de una misma imagen. Un par negativo está formado por imágenes diferentes entre sí. El modelo aprende a acercar los positivos y alejar los negativos en el espacio de representaciones.

Umbral de decisión: Valor que separa las clases positivas y negativas al convertir una puntuación continua en una predicción binaria.

Percentil: Medida estadística que indica el valor por debajo del cual se encuentra un porcentaje específico de observaciones.

Umbral experimental: Valor de corte utilizado provisionalmente con fines comparativos en un entorno de evaluación.

3. Fundamentos teóricos y estado del arte

3.1 Conceptos preliminares

3.1.1 Detección de anomalías

La detección de anomalías es una disciplina dentro del aprendizaje automático orientada a identificar patrones, observaciones o regiones que se desvían significativamente del comportamiento normal. Este tipo de análisis resulta esencial en entornos donde las desviaciones pueden estar asociadas a condiciones críticas o riesgosas, como en el ámbito de la seguridad informática, el mantenimiento industrial o la medicina.

En el caso particular de las imágenes médicas, una anomalía puede manifestarse como una región atípica dentro de una radiografía, tomografía o resonancia, lo que puede ser indicativo de una patología. La detección automatizada de estas regiones permite reducir la carga de trabajo de los profesionales clínicos, mejorar la eficiencia diagnóstica y facilitar el cribado masivo en contextos hospitalarios.

Desde un punto de vista técnico, las anomalías pueden clasificarse en distintas categorías:

- **Anomalías puntuales**: son instancias individuales que se alejan del patrón general. En imágenes, podría ser una lesión aislada o una mancha atípica.
- **Anomalías contextuales**: elementos que son normales en ciertos contextos, pero no en otros. Por ejemplo, una estructura anatómica puede ser esperable en una región del cuerpo, pero no en otra.
- **Anomalías colectivas**: un grupo de elementos que, tomados en conjunto, forman un patrón anómalo. Esto puede aplicarse, por ejemplo, a una secuencia de imágenes o a una zona amplia con múltiples características anómalas leves.

La dificultad de esta tarea radica en la alta variabilidad que existe entre lo que se considera "normal" en diferentes pacientes y condiciones clínicas, y en la escasez de ejemplos etiquetados de anomalías, lo que complica el entrenamiento de modelos supervisados. Por ello, se han desarrollado enfoques alternativos que buscan aprender representaciones sólidas de la normalidad y, a partir de ellas, detectar lo que se sale de ese patrón. En este contexto, las técnicas de aprendizaje auto-supervisado juegan un papel fundamental.

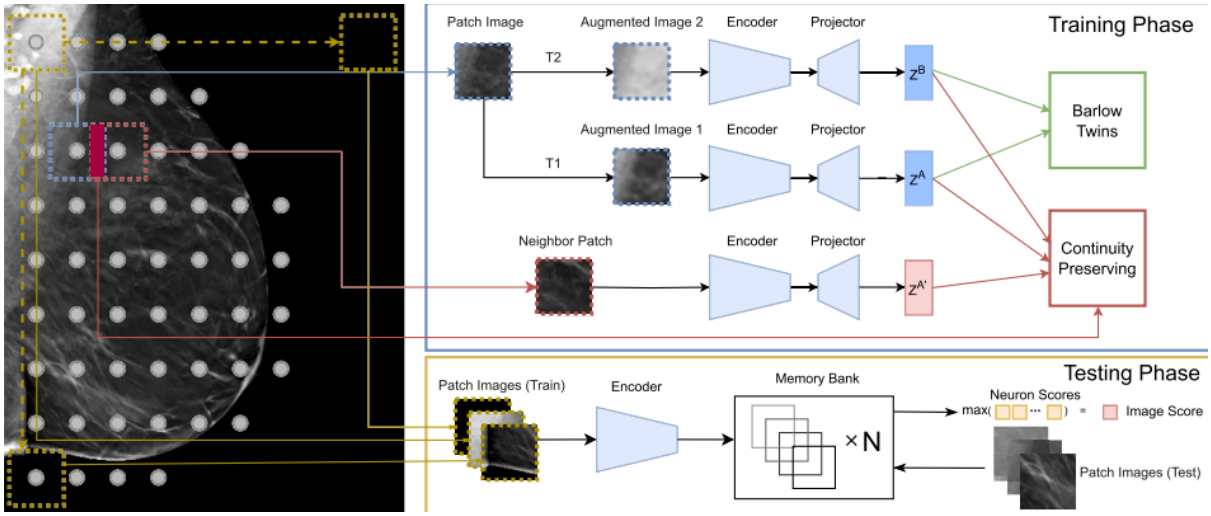

Figura 1: Esquema general del método SWSSL. En la fase de entrenamiento, se generan dos vistas aumentadas de cada patch y se aplica la pérdida Barlow Twins, junto a una pérdida adicional que preserva la coherencia entre parches vecinos. En la fase de test, las activaciones se comparan con una memoria de representaciones normales para detectar anomalías.

Fuente: [1]

3.1.1.1 Métricas de evaluación en el ámbito de detección de anomalías.

A continuación, se describen las métricas más relevantes en este contexto:

AUC (Area Under the ROC Curve): mide la capacidad del modelo para distinguir entre ejemplos normales y anómalos a diferentes umbrales. Un AUC cercano a 1.0 indica una buena separación entre ambas clases. Es especialmente útil cuando las clases están desbalanceadas.

F1-score: representa el equilibrio entre precisión y sensibilidad. Es útil cuando el coste de los falsos positivos y falsos negativos es similar.

Precisión (Precision): indica el porcentaje de predicciones positivas que realmente son anomalías. Una alta precisión implica pocos falsos positivos.

Sensibilidad (Recall o True Positive Rate): refleja la capacidad del modelo para identificar correctamente todas las anomalías reales. Es crítica en entornos clínicos, donde no detectar una anomalía puede tener consecuencias graves.

Especificidad (True Negative Rate): mide la proporción de instancias normales correctamente clasificadas como tales. Es importante para minimizar el número de falsos positivos y evitar alarmas innecesarias.

Accuracy: mide la proporción total de predicciones correctas. Aunque es una métrica común, su utilidad en detección de anomalías es limitada, ya que un modelo que clasifica todo como normal podría obtener una alta accuracy si las anomalías son poco frecuentes.

Estas métricas se utilizarán posteriormente en este trabajo para comparar el rendimiento de los distintos métodos auto-supervisados evaluados, como SWSSL, AutoEncoder y SimCLR.

3.1.2 Aprendizaje Automático

El aprendizaje automático (machine learning) es una rama de la inteligencia artificial que permite a los sistemas aprender patrones a partir de datos sin necesidad de ser programados explícitamente para cada tarea. Este paradigma ha demostrado ser eficaz en una amplia variedad de aplicaciones, incluyendo visión por computador, procesamiento del lenguaje natural, bioinformática y diagnóstico médico.

En su forma más clásica, el aprendizaje automático se divide en tres categorías principales:

- **Aprendizaje supervisado**: el modelo se entrena utilizando un conjunto de datos etiquetado, es decir, donde cada entrada está asociada a una salida conocida. El objetivo es aprender una función que generalice bien sobre nuevos datos. Ejemplo: clasificar radiografías como normales o patológicas a partir de imágenes previamente etiquetadas.
- **Aprendizaje no supervisado**: se emplea cuando los datos no tienen etiquetas. El modelo intenta encontrar patrones, estructuras o agrupaciones dentro del conjunto de datos. Un ejemplo común es el clustering, donde se agrupan instancias similares entre sí sin conocer sus categorías.
- **Aprendizaje auto-supervisado (self-supervised learning, SSL)**: se sitúa entre los dos enfoques anteriores. Consiste en generar tareas auxiliares (pretextuales) que permiten al modelo aprender representaciones útiles de los datos sin necesidad de anotaciones manuales. En estas tareas, parte de la información se oculta o modifica y el modelo debe predecirla a partir del resto.

El aprendizaje supervisado ha demostrado altos niveles de precisión en tareas médicas, pero depende fuertemente de la disponibilidad de datos etiquetados, lo cual es una limitación importante en contextos clínicos. Por su parte, los métodos no supervisados carecen en muchos casos de la capacidad de generar representaciones interpretables o útiles para tareas específicas.

En este sentido, el aprendizaje auto-supervisado surge como una solución intermedia, capaz de aprovechar grandes volúmenes de datos sin etiquetar para entrenar modelos con capacidad generalizable. Esta propiedad es especialmente atractiva en el dominio médico, donde la

obtención de etiquetas precisas y validadas por expertos resulta costosa, lenta y no siempre viable.

3.1.2.1 Aprendizaje auto-supervisado

El aprendizaje auto-supervisado (self-supervised learning, SSL) es un enfoque emergente dentro del aprendizaje automático que permite entrenar modelos sin necesidad de datos etiquetados de forma manual. Se sitúa como un paradigma intermedio entre el aprendizaje supervisado y el no supervisado, ya que utiliza tareas pretextuales (también llamadas tareas auxiliares) para generar señales de supervisión directamente a partir de los datos disponibles.

La idea principal consiste en ocultar o modificar parte de la información de entrada y obligar al modelo a predecir o reconstruir dicha información. A través de este proceso, el modelo aprende representaciones internas significativas, que posteriormente pueden ser utilizadas para resolver otras tareas más complejas, como clasificación, segmentación o detección de anomalías.

Existen distintos tipos de tareas pretextuales que han sido ampliamente exploradas en la literatura, entre ellas:

- **Reconstrucción de entradas**: como en el caso de los AutoEncoders, donde el modelo aprende a comprimir y luego reconstruir la imagen original.
- **Predicción de contexto**: por ejemplo, predecir qué parte de una imagen falta o qué orden tienen los parches que la componen.
- **Contraste entre vistas**: técnica donde se generan dos versiones de la misma entrada aplicando transformaciones distintas, y se entrena al modelo para que sus representaciones internas sean similares (como en SimCLR o Barlow Twins).
- **Enmascarado de regiones**: técnica empleada por modelos como los Masked AutoEncoders (MAE), que consiste en ocultar parte de la imagen y predecir lo que falta.

En el contexto médico, el SSL resulta particularmente útil debido a la abundancia de imágenes no etiquetadas y la escasez de anotaciones clínicas precisas. Además, permite construir modelos generalizables y robustos a partir de datos reales, lo cual es fundamental en entornos donde la variabilidad interpaciente y la ambigüedad diagnóstica son comunes.

Dentro de este enfoque, los métodos utilizados en este trabajo (AutoEncoder, SimCLR y SWSSL) representan tres paradigmas distintos de aprendizaje auto-supervisado: reconstrucción, contraste y continuidad contextual, respectivamente.

3.1.3 Redes neuronales y CNNs

Las redes neuronales son modelos computacionales inspirados en el funcionamiento del cerebro humano, diseñados para procesar datos mediante la propagación de señales a través de capas de nodos interconectados. Estas redes son capaces de aprender representaciones complejas a partir de los datos de entrada, ajustando los pesos de sus conexiones mediante procesos de entrenamiento basados en retro propagación y optimización.

Una red neuronal básica consta de una capa de entrada, una o varias capas ocultas, y una capa de salida. Cada nodo o neurona aplica una función de activación no lineal, lo que permite a la red modelar relaciones no lineales en los datos. La capacidad de aproximar funciones complejas convierte a las redes neuronales en herramientas especialmente útiles en tareas de clasificación, regresión y segmentación.

En el ámbito del procesamiento de imágenes, las redes neuronales convolucionales (Convolutional Neural Networks, CNNs) han demostrado una eficacia notable. Las CNNs introducen operaciones de convolución y pooling, que permiten detectar patrones locales (bordes, texturas, formas) y reducir la dimensión conservando la información relevante. Estas características las hacen especialmente adecuadas para trabajar con datos espaciales, como imágenes médicas.

En este trabajo, todas las arquitecturas implementadas (incluyendo AutoEncoders, SimCLR y SWSSL) se basan en CNNs para la extracción de características visuales. Gracias a su capacidad para aprender jerarquías de representaciones, las redes convolucionales permiten modelar patrones tanto globales como locales, lo que es clave en tareas como la detección de anomalías.

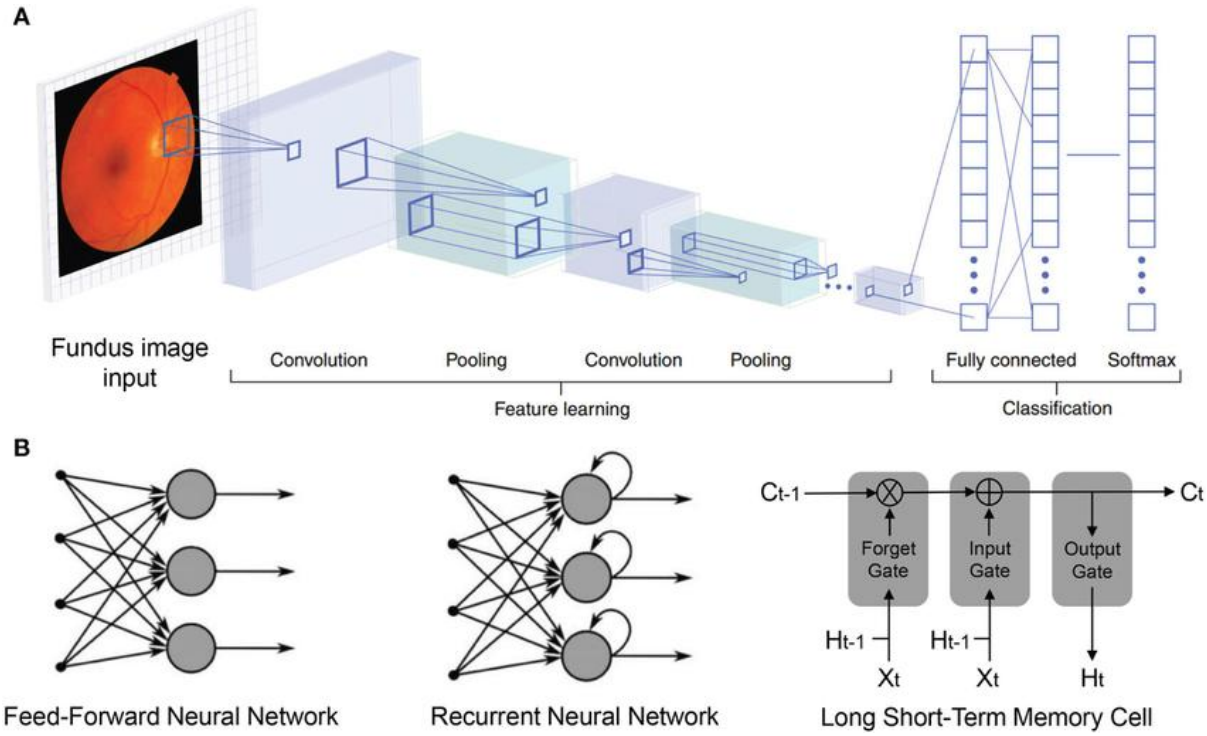

Figura 2: Esquema general de arquitecturas de redes neuronales profundas. (A) Flujo de procesado de una imagen en una red neuronal convolucional (CNN), dónde se incluye la capa de convolución, pooling y clasificación. (B) Diferencias entre arquitecturas comunes: red neuronal feedforward, red neuronal recurrente (RNN) y celda de memoria LSTM. Adaptado de [2].

3.1.4 AutoEncoders

Los AutoEncoders (AEs) [5] son una clase de redes neuronales diseñadas para aprender una representación comprimida (o codificación) de los datos de entrada. Su arquitectura se compone de tres partes principales (ver Figura 2):

- **Un codificador (encoder)**, que transforma la entrada en una representación de menor dimensión. Este módulo puede estar compuesto por capas lineales o convolucionales, según el tipo de datos. En el caso de imágenes, se utilizan capas convolucionales (Conv2D) para extraer patrones locales como bordes, texturas o regiones anatómicas relevantes. En cada capa convolucional, se reduce la resolución espacial mediante stride o pooling, al tiempo que se incrementa la profundidad de los canales. Esto permite al modelo aprender representaciones jerárquicas y progresivamente más difíciles. En resumen, el codificador actúa como un extractor de características que condensa la información esencial de la imagen en un volumen de baja resolución y alta semántica.
- **El espacio latente (bottleneck)** se define como una representación intermedia de dimensión reducida donde se codifica la información esencial de la imagen de entrada.

Esta compresión permite al modelo aprender patrones generales en lugar de memorizar los datos. Su tamaño es un parámetro crucial: si es demasiado reducido, el modelo puede perder detalles importantes; si es demasiado amplio, puede sobre ajustar y reconstruir incluso entradas anómalas. Durante el entrenamiento, las entradas tienden a organizarse de forma estructurada en este espacio, y cada tipo de patrón puede ocupar regiones distintas, aunque no se utilicen etiquetas. Esta propiedad convierte al espacio latente en un recurso útil para tareas posteriores como clustering o clasificación basada en las representaciones aprendidas.

- **Un decodificador (decoder)**, que toma la representación latente y genera una reconstrucción de la imagen original. En los AutoEncoders convolucionales, el decodificador se compone de capas convolucionales transpuestas (ConvTranspose2D), que actúan como "descompresores" espaciales. Estas capas van incrementando la resolución de forma progresiva, utilizando activaciones como ReLU o Sigmoid, hasta recuperar el tamaño original. El decodificador intenta aprender una función inversa del codificador, reconstruyendo los patrones visuales a partir de la información comprimida.

El objetivo del entrenamiento es minimizar la diferencia entre la imagen de entrada y su reconstrucción. Para ello, se utiliza comúnmente la función de pérdida de error cuadrático medio (MSE), aunque también pueden utilizarse métricas perceptuales más avanzadas como el Structural Similarity Index (SSIM) en contextos donde se desea preservar la calidad visual.
Esta arquitectura es especialmente útil para la detección de anomalías. Cuando el modelo se entrena exclusivamente con imágenes normales, aprende a reconstruir fielmente ese tipo de entrada. Sin embargo, cuando recibe una imagen anómala, no dispone de una codificación adecuada en su espacio latente, lo que conduce a una reconstrucción mala. Esta diferencia se traduce en un error mayor, que puede utilizarse como puntuación de anomalía.

En contextos clínicos como la radiología, esta propiedad resulta particularmente útil. El AutoEncoder se entrena únicamente con radiografías normales, y se espera que aquellas que presentan patologías (por ejemplo, neumonía) generen reconstrucciones menos precisas. El error por píxel o por región puede ser cuantificado y visualizado mediante mapas de calor, proporcionando una herramienta útil tanto para detección como para interpretación clínica.

Existen distintas variantes de AutoEncoders, entre ellas:

- **Denoising AutoEncoder (DAE)**: introduce ruido artificial en la entrada y entrena al modelo para reconstruir la versión limpia, lo que mejora la robustez frente a datos ruidosos.
- **Variational AutoEncoder (VAE)**: impone una distribución probabilística sobre el espacio latente, útil en generación de datos sintéticos, compresión y modelado bayesiano.
- **Convolutional AutoEncoder (CAE)**: emplea capas convolucionales y transpuestas, y es especialmente adecuado para el tratamiento de imágenes médicas debido a su capacidad para conservar información espacial. Esta arquitectura permite capturar relaciones locales relevantes (por ejemplo, patrones pulmonares, bordes pleurales o alteraciones morfológicas) sin necesidad de una gran cantidad de parámetros.

En este trabajo se emplea un AutoEncoder convolucional como uno de los modelos base de comparación, siguiendo un enfoque clásico pero robusto para la detección de anomalías mediante el análisis del error de reconstrucción. Esta estrategia proporciona un punto de referencia muy bueno sobre el cual comparar técnicas más recientes como los métodos contrastivos auto-supervisados.

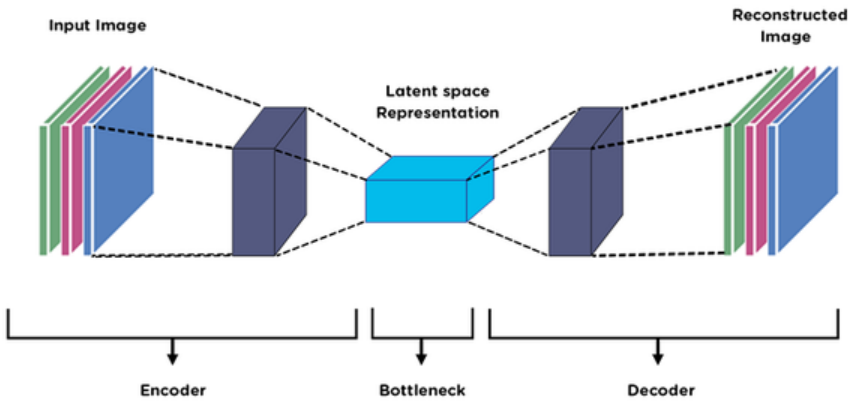

Figura 3: Arquitectura general de un AutoEncoder. El modelo comprime la imagen de entrada en una representación latente a través del encoder, y luego intenta reconstruirla a través del decoder. El entrenamiento se basa en minimizar la diferencia entre la imagen.

3.1.5 SimCLR

SimCLR (Simple Framework for Contrastive Learning of Visual Representations) es un método de aprendizaje auto-supervisado basado en el aprendizaje contrastivo [3]. Este enfoque busca aprender representaciones visuales significativas sin necesidad de etiquetas manuales, optimizando la similitud entre vistas distintas de una misma imagen y maximizando la separación respecto a imágenes distintas.

El procedimiento se basa en los siguientes elementos clave:

- **Generación de vistas aumentadas**: a partir de una misma imagen, se crean dos versiones distintas aplicando transformaciones aleatorias (recortes, rotaciones, cambios de color, etc.). Estas dos vistas forman un par positivo.
- **Extracción de características**: cada vista es procesada por un encoder (típicamente una red ResNet), que produce un vector de características (embedding) que representa la imagen.
- **Proyección al espacio de contraste**: los vectores generados se pasan por una red de proyección (MLP) para llevarlos a un nuevo espacio donde se optimiza la función de contraste.
- **Pérdida contrastiva (NT-Xent Loss)**: esta función penaliza que vistas de imágenes distintas (pares negativos) estén cerca entre sí en el espacio latente, y premia que las vistas de una misma imagen estén lo más próximas posible.

Este mecanismo permite al modelo aprender una representación invariante a las transformaciones aplicadas, capturando aspectos semánticamente relevantes de las imágenes. Posteriormente, estas representaciones pueden utilizarse en tareas de clasificación, detección o segmentación con una pequeña cantidad de datos etiquetados, o incluso para tareas no supervisadas como la detección de anomalías.

En el presente trabajo, SimCLR se implementa como modelo comparativo dentro del conjunto de técnicas auto-supervisadas, actuando como representante de los métodos contrastivos.

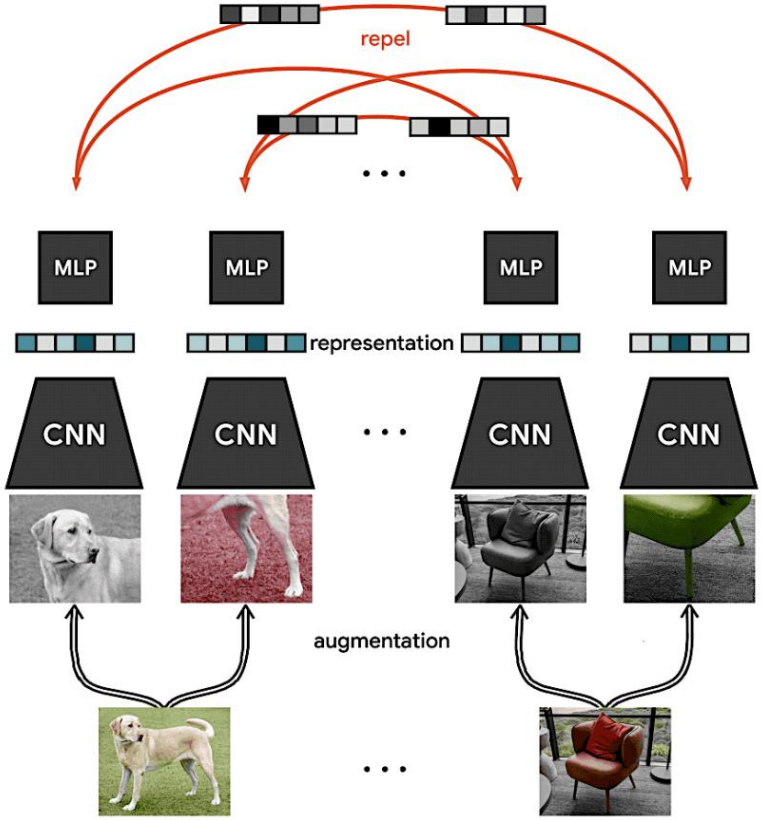

Figura 4: Esquema general del funcionamiento de SimCLR. A partir de una imagen original, se generan vistas aumentadas que se procesan mediante un encoder convolucional (CNN) y una red de proyección (MLP). Adaptado de [3], repositorio oficial de SimCLR [4]

3.1.6 Sliding Window Self-Supervised Learning (SWSSL)

SWSSL (Sliding Window Self-Supervised Learning) es un método de aprendizaje auto-supervisado propuesto por Park et al. [1] para la detección de anomalías en imágenes de alta resolución, como las imágenes médicas. A diferencia de otras técnicas que reducen la imagen para adaptarse a arquitecturas estándar, SWSSL introduce un mecanismo de ventanas deslizantes (*sliding windows*) que permite preservar la resolución original, manteniendo así los detalles clínicamente relevantes.

Durante la fase de entrenamiento, cada imagen se divide en parches mediante un esquema de ventana deslizante. De cada parche, se generan dos vistas diferentes a través de transformaciones de datos (augmentations), como inversión de intensidades o eliminación de bordes. Estas vistas son procesadas por una red convolucional y una red de proyección, entrenadas con la pérdida Barlow Twins, que fuerza a que ambas vistas tengan representaciones similares pero informativas.

Además, SWSSL incorpora una segunda función de pérdida llamada Continuity Preserving Loss, que garantiza que los parches adyacentes dentro de la imagen también tengan representaciones coherentes. Esto es especialmente importante en imágenes médicas, donde el contexto local puede influir fuertemente en la interpretación visual.

En la fase de inferencia, cada imagen se divide nuevamente en parches, que son pasados por el encoder. Las representaciones obtenidas se comparan con una memoria de activaciones normales (obtenida durante el entrenamiento). Las regiones que generan activaciones significativamente diferentes se marcan como anómalas, permitiendo generar mapas de calor o puntuaciones de anomalía por región.

Este enfoque resulta especialmente útil en contextos clínicos, ya que permite detectar áreas patológicas sin necesidad de disponer de imágenes anotadas como “anómalas” durante el entrenamiento. Además, su diseño modular y escalable facilita su aplicación a imágenes de gran tamaño, sin necesidad de pérdida de información por redimensionamiento.

3.1.7 Comparativa de modelos auto-supervisados utilizados en este trabajo

Aspecto	AutoEncoder	SimCLR	SWSSL
Enfoque base	Reconstrucción	Contraste	Contraste con coherencia especial
Arquitectura común	CNN Encoder + Decoder	CNN Encoder + MLP	CNN Encoder + Projector
Tarea pretextual	Reconstrucción de la imagen original	Maximizar similitud entre vistas aumentadas	Vistas aumentadas + coherencia entre parches vecinos
Uso de etiquetas	No	No	No
Ventajas	Simple de implementar, buen rendimiento en estructuras conocidas	Aprende representaciones robustas, generalizables	Preserva resolución, detecta sin etiquetas, contextual
Limitaciones	Sensibilidad al tipo de anomalía, puede sobreajustar	Requiere gran cantidad de augmentations y batch size	Computacionalmente costoso, diseño más complejo
Aplicación en este TFG	Modelo de referencia por reconstrucción	Comparativa contrastiva en imágenes médicas	Método principal evaluado y comparado

Tabla 2: Comparativa de modelos auto-supervisados utilizados

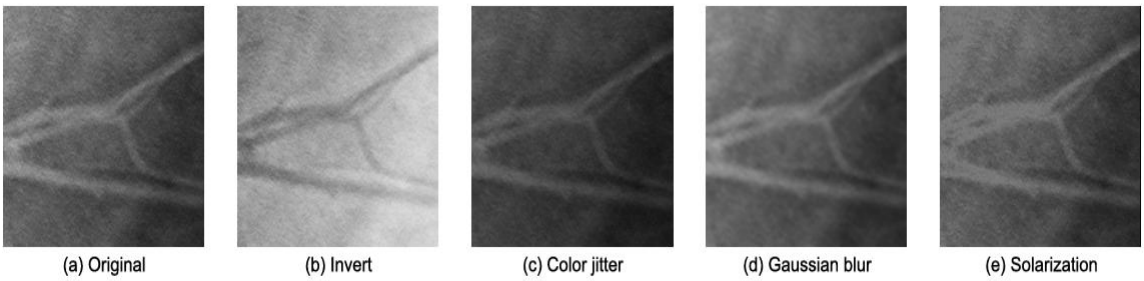

Figura 5: Ejemplos de técnicas de aumento utilizadas en aprendizaje auto-supervisado. Fuente: Adaptado de Park et al., 2022 [1].

3.2 Estado del arte

El estudio del estado del arte permite situar este trabajo dentro del marco actual de investigación en el campo de la detección de anomalías mediante técnicas de aprendizaje automático, con especial atención al aprendizaje auto-supervisado y su aplicación a la visión por computador médica.

A través de la revisión de los enfoques más representativos, se identifican los principales modelos, sus fundamentos teóricos, ventajas y limitaciones, así como su grado de adopción en contextos clínicos. Este análisis permite justificar la selección de métodos incluidos en el estudio comparativo de este TFG, destacando su relevancia técnica y aplicabilidad real.

En las siguientes secciones se exploran las principales técnicas auto-supervisadas utilizadas en visión por computador, su aplicación a imágenes médicas y las razones que motivan la elección de AutoEncoder, SimCLR y SWSSL como objetos de estudio en este trabajo.

3.2.1 Técnicas auto-supervisadas en visión por computador

En los últimos años, el aprendizaje auto-supervisado (SSL) se ha consolidado como una alternativa competitiva al aprendizaje supervisado en tareas de visión por computador. Este avance ha sido impulsado por la necesidad de modelos robustos que puedan entrenarse a partir de grandes volúmenes de datos no etiquetados, especialmente en dominios donde el etiquetado manual resulta costoso o limitado.

Entre las primeras aproximaciones ampliamente adoptadas se encuentran los métodos basados en reconstrucción, como los AutoEncoders, utilizados principalmente en tareas de compresión, reducción de dimensionalidad y detección de anomalías. Aunque su simplicidad los hace interesantes, estudios recientes han señalado su sensibilidad a anomalías sutiles o estructurales, especialmente cuando el espacio latente no está bien regularizado [5].

Los métodos contrastivos, como SimCLR [3], marcaron un hito en el SSL al introducir una estrategia más eficaz de entrenamiento basada en la discriminación entre vistas aumentadas. Su eficacia ha sido demostrada en benchmarks como ImageNet, y han servido de base para arquitecturas más avanzadas como BYOL, MoCo v3 y SwAV. Estos modelos han sido adoptados en escenarios con grandes volúmenes de imágenes no etiquetadas, destacando por su capacidad de generar representaciones generalizables.

Más recientemente, se ha producido una transición hacia arquitecturas más eficientes y contextualmente sensibles, como los Masked AutoEncoders (MAE) o Barlow Twins. Este

último, en particular, ha demostrado ser especialmente útil en situaciones donde la diversidad dentro de las representaciones internas del modelo es crítica, y ha sido integrado como base del método SWSSL [1].

En conjunto, el estado del arte demuestra una progresiva evolución desde enfoques simples y en pequeña escala hacia modelos más estructurados, capaces de capturar relaciones espaciales y semánticas profundas. Esta trayectoria justifica la selección de AutoEncoder, SimCLR y SWSSL como casos representativos dentro de este trabajo, cubriendo distintos paradigmas del aprendizaje auto-supervisado en visión por computador.

3.2.2 Aplicación del aprendizaje auto-supervisado a imágenes médicas

La aplicación del aprendizaje auto-supervisado (SSL) al ámbito médico ha despertado un creciente interés en los últimos años, motivado principalmente por dos factores: la abundancia de imágenes no etiquetadas en entornos clínicos y la dificultad de obtener anotaciones expertas de forma sistemática. A diferencia de otros dominios, como la clasificación generalista o la visión industrial, la medicina presenta particularidades como la alta resolución de las imágenes, la variabilidad anatómica entre pacientes y la ambigüedad diagnóstica incluso entre profesionales.

Numerosos estudios han demostrado que el aprendizaje auto-supervisado puede generar representaciones útiles en tareas médicas, incluso superando en algunos casos a los modelos entrenados de forma supervisada. Por ejemplo, Zhou et al. emplearon contrastive learning para preentrenar modelos de clasificación de enfermedades pulmonares en radiografías de tórax, mostrando mejoras en sensibilidad sin necesidad de etiquetas [7]. Azizi et al. adaptaron técnicas de pretext task sobre imágenes de retina, demostrando que la representación auto-supervisada se transfirió eficazmente a tareas de segmentación patológica [8].

Modelos como Masked AutoEncoders (MAE) han sido aplicados recientemente a datos médicos en tareas de reconstrucción de resonancias magnéticas, y métodos como PatchCore han sido adaptados a contextos clínicos con resultados prometedores en detección de lesiones microscópicas.

Pese a estos avances, la mayoría de trabajos actuales se centran en el preentrenamiento para clasificación o segmentación, mientras que la detección directa de anomalías, especialmente en imágenes sin etiquetas, continúa siendo un reto. En particular, muchos de los modelos

contrastivos empleados en medicina operan sobre imágenes redimensionadas, lo que puede suponer una pérdida de información crítica en imágenes de alta resolución.

Este trabajo aborda dicho vacío mediante la evaluación comparativa de tres enfoques auto-supervisados aplicados directamente a imágenes médicas, manteniendo su resolución original. En concreto, se comparan los modelos sobre el conjunto de datos Chest X-ray, utilizando una metodología similar y métricas clínicas estandarizadas. Este enfoque permite establecer un benchmark específico para tareas de detección de anomalías en contextos clínicos realistas.

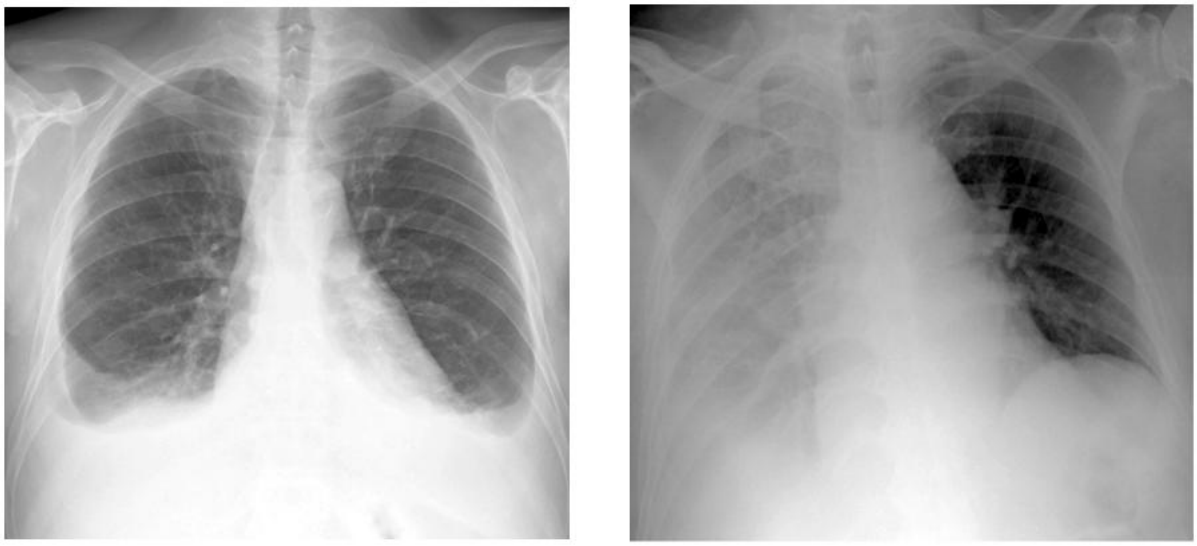

Figura 6: Imagen de radiografía de tórax (PadChest) [6]

3.2.3 Comparación de enfoques existentes y limitaciones detectadas.

Diversos estudios han abordado la detección de anomalías en imágenes médicas mediante aprendizaje auto-supervisado, implementando modelos como AutoEncoders, redes contrastivas (SimCLR, Barlow Twins) o arquitecturas basadas en parches y memorias. No obstante, los enfoques varían ampliamente en términos de condiciones experimentales, lo que dificulta una comparación directa.

Por ejemplo, en el trabajo original de SimCLR se alcanzó una precisión del 94.2% en ImageNet sin etiquetas, pero para tareas médicas su rendimiento ha sido menos sistemáticamente validado. En estudios clínicos, cuando se aplica sobre radiografías de tórax (ChestX-ray14), se ha observado que mejora el AUC en más de un 5% respecto a entrenamiento desde cero con pocas etiquetas [7].

Los AutoEncoders, por su parte, han sido la base de múltiples enfoques de detección de

anomalías. Sin embargo, presentan limitaciones cuando las anomalías son de tipo estructural o difusas, como ocurre en muchos casos clínicos. En el estudio de Dong et al. (2023) [1], los AE aplicados a imágenes mamográficas bajo resolución estándar (256×256) obtuvieron un AUC de 82.13%, mientras que SWSSL superó esta marca alcanzando un AUC del 86.62%, una mejora de más de 4 puntos porcentuales.

Además, ese mismo trabajo mostró que al evaluar sobre el dataset Chest X-ray, el método SWSSL alcanzó una especificidad del 98.29% tras ajustar el umbral a máxima sensibilidad, lo que supone una ventaja significativa en contextos donde es clave minimizar falsos positivos. Por su parte, modelos como PatchCore o PaDiM, que destacan en contextos industriales, no han mostrado un rendimiento superior en dominios médicos. Dong et al. compararon PatchCore y SWSSL con una arquitectura Wide-ResNet-50 bajo condiciones controladas, y observaron que SWSSL superó a PatchCore tanto en precisión como en robustez, especialmente cuando se utilizaba entrada en alta resolución sin necesidad de redimensionamiento previo.

En conjunto, estos resultados sugieren que:

- La mayoría de estudios comparativos no emplean condiciones homogéneas.
- Pocas investigaciones utilizan imágenes médicas en su resolución original.
- No existe un benchmark estandarizado que compare múltiples métodos auto-supervisados en este dominio.

Este TFG pretende contribuir a llenar ese vacío, comparando tres técnicas representativas (AutoEncoder, SimCLR y SWSSL) sobre el dataset Chest X-ray, bajo un entorno experimental unificado. Esto incluye el mismo tamaño de entrada, arquitectura base equivalente (ResNet), protocolo de entrenamiento y métricas clínicas comparables.

3.2.4 Justificación del enfoque adoptado

Haciendo un repaso de los últimos trabajos publicados, se muestra un avance significativo en la aplicación del aprendizaje auto-supervisado a tareas de detección de anomalías en imágenes médicas. Métodos como SimCLR, AutoEncoder y más recientemente SWSSL han sido implementados con éxito en distintos escenarios clínicos además de ser comparados entre sí [1].

No obstante, estas comparativas suelen estar condicionadas por diferencias técnicas en las arquitecturas utilizadas, los protocolos de entrenamiento, el preprocesamiento de los datos o

las resoluciones de imagen aplicadas. En muchos casos, los modelos no son entrenados y evaluados bajo exactamente las mismas condiciones, lo que limita la objetividad de los resultados y su reproducibilidad. Además, no siempre se documentan con suficiente detalle los hiperparámetros, las funciones de pérdida o los criterios de evaluación empleados.

En ese contexto, este Trabajo de Fin de Grado propone una evaluación estructurada y homogénea de tres enfoques representativos donde AutoEncoder, es una técnica basada en reconstrucción; SimCLR es un modelo contrastivo de referencia; y SWSSL, una propuesta reciente que incorpora coherencia espacial y ventanas deslizantes para preservar la resolución original. Estos modelos se entrenan sobre el conjunto de datos Chest X-ray con una configuración común en términos de arquitectura base, preprocesamiento, resolución y métricas clínicas.

La justificación de esta propuesta radica en aportar una evaluación comparativa reproducible y clínicamente realista, que permita valorar las fortalezas y limitaciones de cada enfoque en condiciones controladas. Esta comparación tiene como objetivo no solo cuantificar el rendimiento técnico de los modelos, sino también ofrecer una base objetiva que pueda ser útil en la toma de decisiones para futuros desarrollos de herramientas de apoyo al diagnóstico médico automatizado.

4. Metodología

Este trabajo ha seguido el enfoque estructurado y secuencial propuesto por la metodología **CRISP-DM (Cross-Industry Standard Process for Data Mining)**. CRISP-DM es ampliamente utilizada en proyectos de ciencia de datos por su flexibilidad, enfoque iterativo y clara división en fases. Su aplicación a este proyecto permite mantener la coherencia entre los objetivos del trabajo, el análisis de los datos y la implementación técnica. A continuación, se describe cómo se ha aplicado cada fase al presente TFG:

- **Comprensión del negocio**: Se definió como objetivo principal la evaluación comparativa de modelos auto-supervisados para la detección de anomalías en imágenes médicas de alta resolución.
- **Comprensión de los datos**: Se analizó el conjunto PadChest, su estructura, calidad, distribución de clases, y adecuación al entorno clínico.
- **Preparación de los datos**: Se aplicaron técnicas de preprocesamiento específicas según el modelo (resizing, normalización, recorte, etc.), manteniendo versiones consistentes de las imágenes.
- **Modelado**: Se implementaron tres arquitecturas (AutoEncoder, SimCLR y SWSSL), con su configuración particular de arquitectura, pérdidas y transformaciones.
- **Evaluación**: Se compararon los modelos bajo condiciones experimentales controladas mediante métricas clínicas como AUC, F1-score, sensibilidad y especificidad.
- **Implementación**: Se documentó el código, se visualizó el comportamiento de cada modelo y se preparó el sistema para futuras extensiones o validaciones clínicas.

Esta estructura metodológica permite alinear cada etapa técnica con los objetivos definidos y proporciona una base sólida para la reproducibilidad y análisis crítico de los resultados.

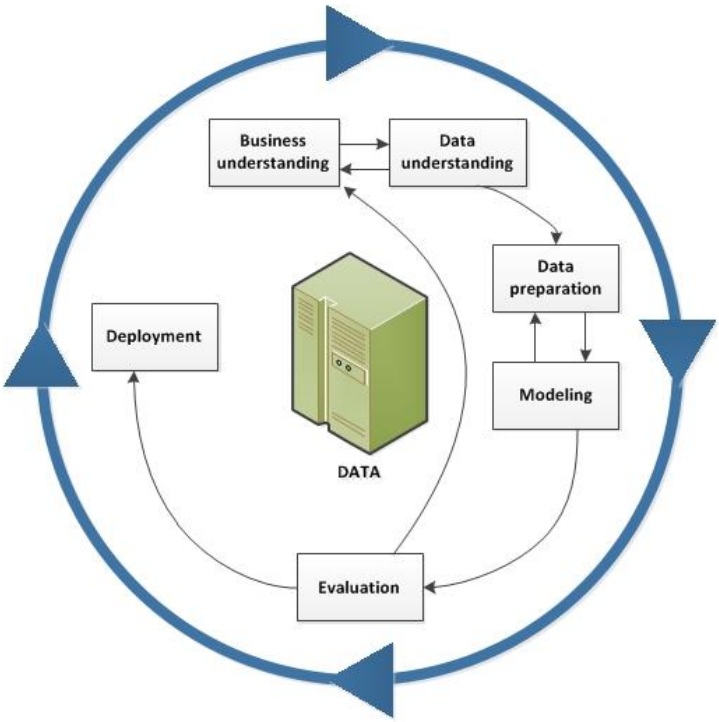

Figura 7: Esquema del ciclo CRISP-DM

4.1 Conjunto de datos utilizado

4.1.1 Descripción general

Para el desarrollo experimental del presente Trabajo de Fin de Grado se ha utilizado el conjunto de datos Chest X-ray (CXR), una colección pública de radiografías de tórax ampliamente utilizada en el ámbito clínico y académico para tareas de clasificación, segmentación y detección de anomalías en imágenes médicas. El dataset fue extraído de la base PadChest, desarrollada por el Hospital Universitario San Juan de Alicante (España) desde 2009 a 2017. Contiene más de 160.000 estudios radiológicos con imágenes en alta resolución y metadatos asociados [9]. Un 27% de los informes se anotaron manualmente por médicos, mientras que el resto, fue etiquetado automáticamente utilizando un método supervisado basado en una RNN (red neuronal recurrente) con mecanismo de atención. Este enfoque híbrido refleja el creciente uso de técnicas de aprendizaje profundo para automatizar tareas clínicas, y es bastante cercano al objetivo de este trabajo; desarrollar modelos auto-supervisados capaces de aprender sin depender de anotaciones humanas.

Este conjunto se adapta perfectamente a los objetivos de este trabajo por las siguientes razones:

- **Volumen y diversidad**: incluye un gran número de imágenes de pacientes reales, con variabilidad clínica, anatómica y técnica.

- **Alta resolución original**: permite evaluar enfoques que trabajan con imágenes sin redimensionar, como SWSSL.
- **Representatividad clínica**: contiene imágenes que reflejan casos normales y patológicos reales, obtenidos en hospitales.
- **Accesibilidad y uso académico**: es de acceso libre bajo licencia para uso no comercial, lo que facilita su uso.

Cabe destacar que, si bien el dataset original incluye anotaciones clínicas detalladas, **este trabajo no hace uso de las etiquetas para el entrenamiento**, ya que todos los modelos implementados siguen un enfoque auto-supervisado. No obstante, la disponibilidad de etiquetas en la partición de test permite realizar una evaluación cuantitativa precisa de los modelos una vez entrenados.

4.1.2 Selección y partición del conjunto de datos

Dado que los modelos auto-supervisados deben aprender la representación de la normalidad a partir de datos no etiquetados (no se sabe que salida esperar), se ha llevado a cabo una selección del subconjunto de imágenes que garantice coherencia.

Conjunto	Nº de imágenes	Tipo	Uso principal
Entrenamiento	1349	Solo normales	Aprendizaje de la representación normal
Test	624 (234 normales + 390 anómalas)	Etiquetadas	Evaluación de capacidad de detección

Tabla 3: Distribución de imágenes en los conjuntos de entrenamiento y test

Las imágenes anómalas del test corresponden a pacientes con diagnóstico confirmado de neumonía.

4.1.3 Consideraciones técnicas del dataset

El conjunto original presenta imágenes en formato DICOM, posteriormente convertidas a escala de grises en formato PNG para su uso directo en los modelos. La resolución nativa varía, pero se han aplicado distintas estrategias de adaptación dependiendo del modelo implementado:

- En modelos como AutoEncoder y SimCLR, se ha optado por un resize a 256×256 px, que facilita el entrenamiento y reduce el coste computacional.
- En el modelo SWSSL, se ha mantenido la resolución alta (1024×1024 px) y se ha utilizado una técnica de ventanas deslizantes (sliding window) para preservar los detalles anatómicos durante el entrenamiento, en línea con la filosofía del paper original [1].

Estas transformaciones no responden a una necesidad de etiquetado, sino a una adecuación estructural de los datos a las limitaciones arquitectónicas y de memoria, así como al objetivo metodológico de evaluar el impacto de la resolución en la detección de anomalías.

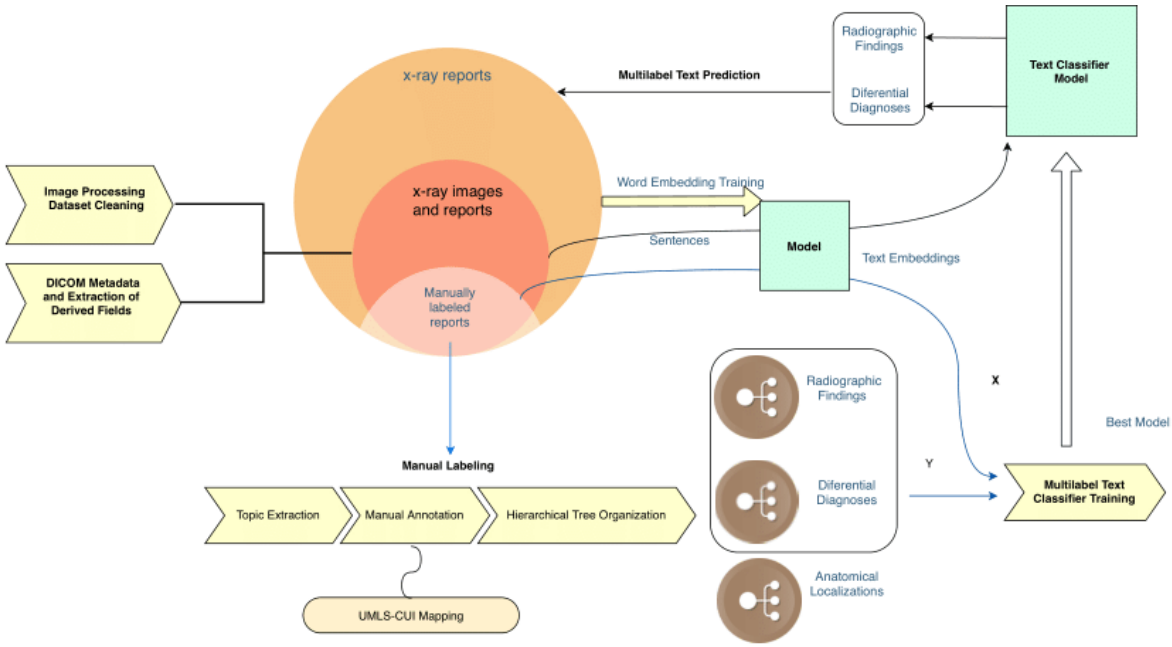

Figura 8: Esquema del proceso de construcción del conjunto de datos PadChest. Se muestran las etapas de limpieza, anotación manual, clasificación automática mediante redes neuronales y estructuración semántica del vocabulario clínico.

[9]

4.2 Técnicas de preprocesamiento aplicadas

El procesamiento de imágenes es una etapa esencial en el flujo de trabajo de cualquier sistema basado en aprendizaje profundo [12]. En el ámbito médico, los datos varían mucho en resolución, intensidad y estructura anatómica. En este trabajo, los objetivos del preprocesamiento son :

- Garantizar la compatibilidad estructural de los datos con las arquitecturas utilizadas.
- Normalizar las condiciones de entrada para obtener resultados comparables entre modelos.

A continuación, se describen las transformaciones aplicadas de forma común y específica según el modelo.

4.2.1 Transformaciones comunes

Todas las imágenes del conjunto de datos han sido sometidas a una serie de pasos previos, independientemente del modelo al que se destinen:

- **Conversión a escala de grises**: todas las imágenes se convierten a un único canal, ya que la información clínica relevante está contenida en la intensidad, no en el color.
- **Normalización de intensidad**: los valores de píxel se escalan al rango , permitiendo una convergencia más estable durante el entrenamiento.
- **Conversión a tensores**: mediante ‘torchvision.transforms.ToTensor’, lo que permite la integración directa en PyTorch.
- **Filtrado de imágenes vacías o corruptas**: se descartan aquellas que presenten artefactos técnicos, ruido excesivo o valores nulos.

Estas transformaciones aseguran una base coherente y limpia desde la que aplicar los pasos específicos por modelo.

4.2.2 Preprocesamiento por modelo

4.2.2.1 AutoEncoder

Para el modelo AutoEncoder se han aplicado las siguientes transformaciones:

- **Conversión a escala de grises**: adecuada para radiografías de tórax, que no contienen información útil en canales de color.
- **Redimensionamiento a 256x256 píxeles**, seguido de **recorte central**, para mantener la proporción anatómica sin distorsiones.

- **Normalización al rango [-1, 1]** para estabilizar el aprendizaje.
- **Conversión a tensor** para su posterior uso por el modelo.

Estas operaciones permiten construir una base coherente para que el AutoEncoder aprenda a reconstruir fielmente imágenes normales. No se aplican técnicas de augmentación, dado que el objetivo del modelo es reconstruir con la mayor fidelidad posible y cualquier distorsión artificial iría en contra de este objetivo. La verificación de estas transformaciones se ha realizado visualmente en una etapa inicial. y reproduce el enfoque clásico de detección de anomalías mediante error de reconstrucción.

4.2.2.2 SimCLR

SimCLR emplea un paradigma de aprendizaje contrastivo. Para ello, se aplican transformaciones que generen vistas distintas de una misma imagen:

- Redimensionamiento a 256×256 píxeles.
- Transformaciones aleatorias (augmentations):
    - Recortes aleatorios (random crop)
    - Volteo horizontal (horizontal flip)
    - Cambios de brillo y contraste
    - Aplicación de ColorJitter y GaussianBlur

Estas operaciones permiten que el modelo aprenda representaciones invariantes, fortaleciendo su capacidad de generalización.

4.2.2.3 SWSSL

El modelo SWSSL requiere un tratamiento especial, ya que trabaja con imágenes de alta resolución:

- Resolución preservada: 1024×1024 píxeles.
- Ventanas deslizantes (sliding window): la imagen se divide en parches de tamaño 128×128 con stride de 32 píxeles. Esta técnica permite mantener el contexto espacial sin sacrificar resolución.
- Augmentations específicas:

  Inversión de intensidades (invert), Gaussian blur, Color jitter.

  Además, se han eliminado técnicas perjudiciales para imágenes médicas, como el cropping + resizing, ya que pueden alterar la densidad de los tejidos.

La combinación de estas transformaciones con la pérdida Barlow Twins y la Continuity Preserving Loss busca que el modelo aprenda representaciones locales coherentes e invariantes.

5. Desarrollo e implementación

Este capítulo describe el proceso de desarrollo técnico de los modelos seleccionados, incluyendo la preparación del entorno, la implementación específica de cada arquitectura, los procesos de entrenamiento y evaluación, así como las particularidades experimentales que han guiado las decisiones tomadas. A diferencia del capítulo metodológico, aquí se presenta una visión operativa y aplicada del trabajo, con apoyo en fragmentos de código y evidencia visual del proceso.

5.1 Entorno de trabajo y librerías utilizadas

El desarrollo de este Trabajo de Fin de Grado se ha llevado a cabo en **Google Colab**, un entorno de ejecución basado en Jupyter Notebooks que proporciona acceso gratuito a GPUs, facilitando así el entrenamiento de modelos de aprendizaje profundo con un coste computacional asumible. La elección de esta plataforma responde a su compatibilidad con librerías modernas, su facilidad para integrar datos desde Google Drive y su entorno.
Los modelos han sido implementados íntegramente en **Python 3.10** y utilizando como base el framework **PyTorch**, por su flexibilidad, dinamismo y uso en el ámbito de la investigación.

Las principales librerías empleadas han sido:

- 'torch' y 'torchvision': para la creación de modelos, transformaciones de imágenes y gestión de datos.
- PIL (Python Imaging Library, su versión moderna conocida como Pillow): para la carga y tratamiento de imágenes individuales.
- 'matplotlib'': para la generación de gráficos, visualización de pérdidas y reconstrucciones.
- 'sklearn.metrics': para el cálculo de métricas de evaluación (AUC, F1, etc.).
- numpy: para operaciones matriciales y manipulación de tensores.

Figura 9: Principales tecnologías y librerías utilizadas en el desarrollo del proyecto.

***Python**: lenguaje de programación principal, ampliamente utilizado en entornos científicos y de aprendizaje automático por su legibilidad, flexibilidad y rica comunidad de desarrollo.*

***PyTorch**: biblioteca de deep learning empleada para definir y entrenar modelos neuronales. Ofrece un control granular del flujo de datos y es muy utilizada en investigación académica.*

***NumPy**: librería fundamental para operaciones matriciales y manipulación de arrays, utilizada como base para muchas operaciones con tensores.*

***Matplotlib**: herramienta de visualización de gráficos utilizada para representar la evolución del entrenamiento, reconstrucciones y comparaciones visuales.*

***Pillow**: versión moderna y mantenida de PIL (Python Imaging Library), utilizada para cargar, convertir y preprocesar imágenes antes de pasarlas a tensores para su procesamiento por modelos neuronales.*

El código se ha estructurado de forma modular dentro del notebook, manteniendo bloques claramente separados para:

- Definición de transformaciones y carga de datos (preparación de dataset inclusive)
- Arquitectura e inicialización del modelo
- Entrenamiento y validación
- Evaluación y visualización de resultados

Esta organización ha permitido realizar pruebas iterativas sobre cada bloque, facilitar el análisis de errores y mantener una trazabilidad clara durante el proceso de desarrollo.

Además, se ha hecho uso de la plataforma Kaggle Datasets como fuente de descarga del conjunto de datos utilizado. Kaggle es una plataforma de ciencia de datos de acceso libre que permite alojar y compartir conjuntos de datos, realizar competiciones de modelado y colaborar en proyectos reproducibles. Esta herramienta facilita la publicación y reutilización de datos en un entorno estructurado y profesional. Su integración con Google Colab permite montar directamente el repositorio de datos con comandos sencillos, sin necesidad de descarga manual.

- Se evita depender de recursos de almacenamiento o conexión propios.
- Se garantiza acceso inmediato al conjunto de datos sin requerir pasos manuales de carga.
- Se incrementa la reutilización del experimento al enlazar directamente con la fuente de datos.

En contraposición a ejecutar el entrenamiento en un entorno local (hardware propio), el uso combinado de Kaggle y Google Colab ofrece ventajas clave como:

- Acceso gratuito a GPU.
- No requerir una tarjeta gráfica dedicada ni configuración de entornos virtuales complejos.
- Evitar problemas de compatibilidad de versiones o librerías. [14]

Este enfoque se alinea con la filosofía de investigación reproducible y accesible, favoreciendo que futuros trabajos puedan replicar o extender los resultados obtenidos.

Figura 10: Kaggle datasets

5.2 Carga de datos y preprocesamiento (AutoEncoder)

El proceso de carga y preparación de los datos para el modelo AutoEncoder se ha diseñado para garantizar la coherencia, simplicidad y eficacia en el entrenamiento. A continuación se describe paso a paso la lógica implementada, basada en el código ejecutado en el entorno de Google Colab.

Antes de aplicar cualquier filtrado o transformación, se importaron las librerías necesarias para el manejo de datos, visualización y construcción de modelos:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import os
```

Figura 11: Librerías importadas en el proyecto

Posteriormente, se conecta Google Colab con la API de Kaggle. Se suble el archivo 'kaggle.json' con las credenciales de la cuenta, movemos e instalamos kaggle.

```
# Subir kaggle.json manualmente cada vez que se ejecute
from google.colab import files
files.upload()  # Subir kaggle.json cuando lo pida (create new token para conectarse a API)

!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

!pip install -q kaggle

!kaggle datasets download -d carlospinopadilla/chestxray
!unzip -q chestxray.zip -d chest_xray
```

Figura 12: Proceso de preparación de dataset usando kaggle

También se realizó una verificación de la estructura de las imágenes en el dataset a modo de prueba. Se seleccionó aleatoriamente una imagen de la carpeta 'chest_xray/chest_xray/train/NORMAL', observando su formato, modo y tamaño:

```
# Se comprueba el formato de las imágenes del dataset
from PIL import Image
import os

# Ruta a una imagen cualquiera
image_path = "chest_xray/chest_xray/train/NORMAL"

# Leer una imagen al azar
sample_file = os.listdir(image_path)[0]
img = Image.open(os.path.join(image_path, sample_file))

# Mostrar formato, modo y tamaño
print("Formato:", img.format)
print("Modo:", img.mode)
print("Tamaño:", img.size)
```

```
Formato: JPEG
Modo: L
Tamaño: (1634, 1183)
```

Figura 13: Proceso de verificación estructura imágenes. Imágenes PNG, en modo 'grayscale', suficientemente grandes para ser redimensionadas a 256 x 256

Se muestra una imagen como ejemplo.

```
import matplotlib.pyplot as plt

plt.imshow(img, cmap='gray')
plt.title(f"{sample_file} - {img.size}")
plt.axis('off')
plt.show()
```

Figura 14: Representación de imagen importando pyplot de matplotlib

5.2.1 Estructura del conjunto de datos y selección de imágenes

Se ha utilizado la clase 'ImageFolder' de 'torchvision.datasets', que organiza automáticamente las imágenes según las subcarpetas del directorio de origen. En este caso, se parte de la ruta chest_xray/chest_xray/train, donde las subcarpetas representan las clases NORMAL y PNEUMONIA.

Dado que el AutoEncoder debe aprender la representación de la normalidad, se filtran únicamente las imágenes de la clase **NORMAL**, siguiendo el enfoque auto-supervisado:

Esta línea elimina todas las muestras etiquetadas con clase distinta de 0, es decir, cualquier imagen con diagnóstico patológico. Este filtrado garantiza que el modelo aprenda un patrón reconstruible únicamente a partir de ejemplos no anómalos.

```
# Solo imágenes normales (clase 0), subcarpeta que se genera automáticamente
train_dataset.samples = [s for s in train_dataset.samples if s[1] == 0]
```

Figura 15: Elección de samples

```
Total imágenes normales para entrenamiento: 1349
```

Figura 16: Total imágenes de entrenamiento

Cabe destacar que la estructura del conjunto de datos y el proceso de selección de imágenes normales descrito en esta sección se mantiene idéntico en los modelos posteriores (SimCLR y SWSSL), con el objetivo de garantizar condiciones iguales durante el entrenamiento y permitir una comparación equitativa.

5.2.2 Transformaciones aplicadas

Para preparar correctamente las imágenes antes de introducirla a la red, se han definido una serie de transformaciones mediante 'transforms.Compose'.

Estas transformaciones realizan las siguientes funciones:

- **Grayscale()**: convierte las imágenes a escala de grises (un solo canal), lo cual es coherente con el tipo de imagen médica analizada.

- **Resize + CenterCrop**: ajusta la imagen a 256x256 píxeles manteniendo su proporción central. Esto asegura homogeneidad sin distorsionar zonas clínicas importantes.
- **ToTensor()**: convierte las imágenes PIL en tensores PyTorch normalizados en el rango [0,1].
- **Normalize((0.5,), (0.5,))**: reescala los valores de píxel al rango [-1, 1], lo que estabiliza el entrenamiento al centrar los datos alrededor de cero.

Este pipeline garantiza que las imágenes entrenadas sean consistentes y adecuadas a la estructura de la red neuronal.

```
transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize(256),
    transforms.CenterCrop(256),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Figura 17: Pipeline de transformaciones aplicadas a las imágenes

5.2.3 Carga mediante DataLoader

Una vez definidas las transformaciones y aplicado el filtrado, se utiliza la clase DataLoader para cargar los datos en memoria por lotes durante el entrenamiento:
Esto permite alimentar al modelo con bloques de imágenes optimizados para entrenamiento en GPU. Además, la opción 'shuffle=True' garantiza que el orden de los datos varíe en cada época, ayudando a evitar sobreajuste.

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Figura 18: DataLoader AutoEncoder

5.2.4 Verificación visual de las transformaciones

Para asegurarse de que las transformaciones se han aplicado correctamente y que las imágenes mantienen la calidad visual necesaria, se ha definido una función de visualización.
Esta verificación cumple dos funciones: confirmar la calidad de los datos y detectar posibles errores de preprocesamiento antes de iniciar el entrenamiento Se observa que las imágenes

mantienen una escala adecuada para el tratamiento clínico de ellas, esto garantiza entradas para el modelo limpias y consistentes.

```python
def show_transformed_images(dataset, n_images=8):
    loader = torch.utils.data.DataLoader(dataset, batch_size=n_images, shuffle=True)
    images, _ = next(iter(loader))
    images = images * 0.5 + 0.5  # Desnormalizar
    grid = make_grid(images, nrow=4)
    plt.figure(figsize=(10, 5))
    plt.imshow(grid.permute(1, 2, 0))
    plt.axis('off')
```

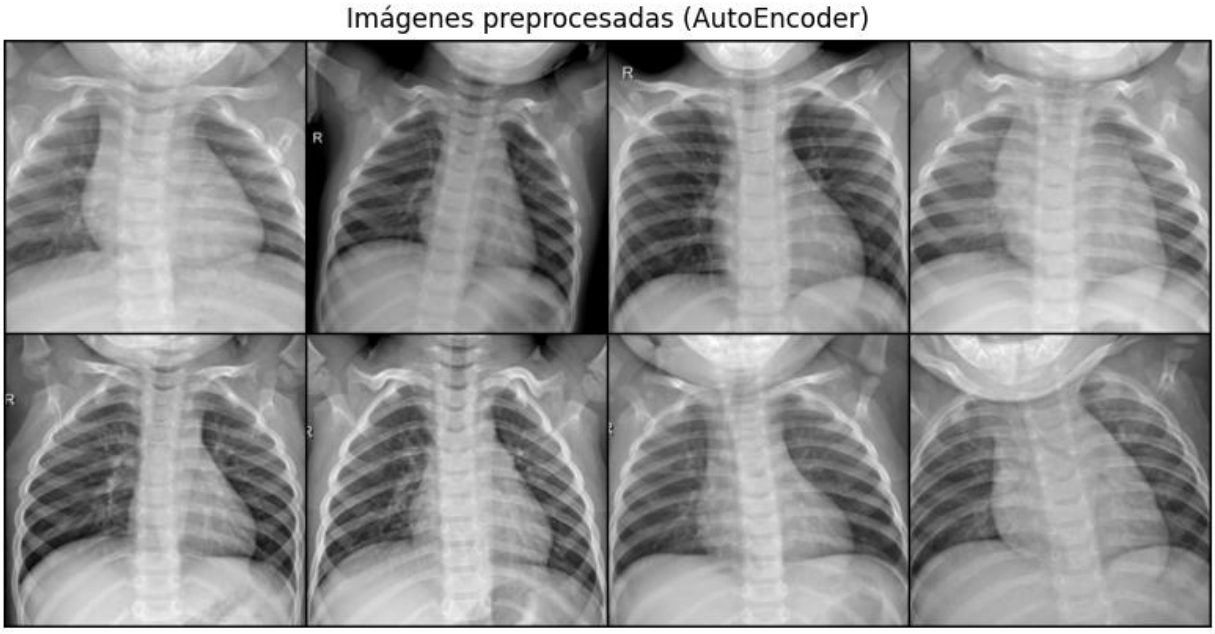

Figura 19: Función de visualización, imágenes de chestxray preprocesadas

5.3 Implementación del modelo AutoEncoder

Una vez definido el conjunto de datos y preparado el preprocesamiento, se procede a la implementación del modelo AutoEncoder, cuya finalidad es reconstruir imágenes normales de tórax. Este tipo de modelo resulta adecuado para tareas de detección de anomalías, ya que cualquier desviación respecto a la normalidad se refleja en un aumento del error de reconstrucción.

5.3.1 Diseño de la arquitectura

Se ha definido un AutoEncoder convolucional simétrico, compuesto por un bloque de codificación (encoder) y otro de decodificación (decoder). El encoder reduce progresivamente la dimensionalidad espacial de la imagen mediante capas convolucionales, mientras que el decoder invierte ese proceso mediante capas deconvolucionales (ConvTranspose2d) para recuperar la imagen original.

La arquitectura implementada en la clase ConvAutoEncoder es la siguiente:

```python
import torch.nn as nn
import torch

class ConvAutoEncoder(nn.Module):
    def __init__(self):
        super(ConvAutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, stride=2, padding=1),  # 128 -> 64
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1), # 64 -> 32
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 128, 3, stride=2, padding=1),# 32 -> 16
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 3, stride=2, padding=1, output_padding=1),  # 16 -> 32
            nn.ReLU(),
            nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1, output_padding=1),   # 32 -> 64
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, 3, stride=2, padding=1, output_padding=1),    # 64 -> 128
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ConvAutoEncoder().to(device)
```

Figura 20: Clase ConvAutoEncoder

5.3.2 Justificación técnica

- Se ha utilizado una arquitectura profunda pero eficiente, adecuada para resolución 256×256.
- Las activaciones ReLU permiten una convergencia más rápida y evitan saturaciones.
- Se ha añadido Batch Normalization en el encoder para estabilizar los gradientes.
- La función de activación final es Sigmoid, apropiada para reconstruir imágenes normalizadas en el rango [0,1].

Este diseño permite capturar patrones locales relevantes en radiografías de tórax, y la estructura simétrica facilita una reconstrucción precisa desde el espacio latente comprimido.

5.4 Entrenamiento del modelo AutoEncoder

Una vez definida la arquitectura del AutoEncoder, se procede a su entrenamiento utilizando el conjunto de datos previamente preprocesado. El objetivo del entrenamiento es minimizar la diferencia entre la imagen de entrada y su reconstrucción, lo que se logra mediante una función de pérdida basada en el error cuadrático medio.

5.4.1 Explicación y configuración del entrenamiento.

Para entrenar el modelo se han utilizado los siguientes componentes:

- **Función de pérdida**: 'nn.MSELoss()' (Mean Squared Error), que penaliza las diferencias entre los valores de los píxeles originales y reconstruidos.
- **Optimizador**: Adam, con una tasa de aprendizaje (lr) de 1e-3 y un término de regularización L2 (weight_decay) de 1e-5.
- **Número de épocas**: inicialmente se han entrenado 10 épocas, con posibilidad de ampliarlo.
- **Tamaño de batch**: 32 imágenes por iteración.

A continuación, se explica el bloque de código de entrenamiento línea por línea.

- Se define la función de pérdida que se usará para medir el error entre la imagen original y su reconstrucción : 'criterion = nn.MSELoss()'.

‘MSELoss’ calcula el error cuadrático medio, lo cual es perfecto para tarea de reconstrucción, ya que penaliza diferencias pixel a pixel.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Figura 21: Mean Square Error

- Se define el optimizador que actualizará los pesos del modelo: ‘optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)’.
  Este optimizador combina lo mejor de AdaGrad y RMSProp, adaptando la tasa de aprendizaje de cada parámetro. Ha sido adoptado en aprendizaje profundo debido a su eficiencia, robustez y adaptación al modelo [10]. Además, el término ‘weight_decay’ actúa como regularización L2 para evitar sobreajuste. Posteriormente, se implementará el optimizador SGD con momentum con fines experimentales.

- Se establece el número de épocas: epochs = 10.
  Indica cuántas veces se recorrerá todo el conjunto de entrenamiento.
- Se activa el modo entrenamiento del modelo: ‘model.train()’.
  Esto es necesario en PyTorch para que se comporten correctamente ciertas capas como BatchNorm (si están presentes).
- Comienza el bucle principal de entrenamiento: ‘for epoch in range(epochs)’:.
  Cada iteración de este bucle corresponde a una época completa.
- Se inicializa la variable que acumulará la pérdida total de la época: ‘running_loss = 0.0’.
- Se recorre el conjunto de entrenamiento por lotes: ‘for imgs, _ in train_loader’:.
  Las imágenes se cargan en imgs, y se ignoran las etiquetas (_) porque el AutoEncoder no las necesita.
- Se pasa el lote de imágenes al dispositivo de entrenamiento (CPU o GPU): ‘imgs = imgs.to(device)’.
- Se obtiene la reconstrucción a partir de las imágenes: outputs = model(imgs).
- Se calcula la pérdida entre la entrada y la reconstrucción: loss = criterion(outputs, imgs).
  Este valor mide qué tan bien el modelo ha reconstruido la entrada.

- Se reinician los gradientes acumulados: ‘optimizer.zero_grad()’.
  Es obligatorio antes de cada paso de retropropagación.
- Se realiza la retropropagación para calcular los gradientes: ‘loss.backward()’.
- Se actualizan los pesos del modelo: ‘optimizer.step()’.
- Se acumula la pérdida de este lote al total de la época:’ running_loss += loss.item()’.
- Al finalizar la época, se imprime la pérdida media.

```python
import torch.optim as optim

criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)  # L2 regularización
epochs = 10

model.train()
for epoch in range(epochs):
    running_loss = 0.0
    for imgs, _ in train_loader:
        imgs = imgs.to(device)
        outputs = model(imgs)
        loss = criterion(outputs, imgs)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Época [{epoch+1}/{epochs}] - Pérdida: {running_loss / len(train_loader):.4f}")
```

Figura 22: Bucle de entrenamiento AutoEncoder

```
Época [1/10] - Pérdida: 0.0993
Época [2/10] - Pérdida: 0.0700
Época [3/10] - Pérdida: 0.0672
Época [4/10] - Pérdida: 0.0665
Época [5/10] - Pérdida: 0.0661
Época [6/10] - Pérdida: 0.0657
Época [7/10] - Pérdida: 0.0654
Época [8/10] - Pérdida: 0.0652
Época [9/10] - Pérdida: 0.0647
Época [10/10] - Pérdida: 0.0657
```

Figura 23: Ejemplo de output de ejecución del entrenamiento

Al finalizar cada época del entrenamiento, se registra el valor de pérdida media y se almacena en una lista (Figura 21.). Esta información se utiliza para generar una curva de pérdida, que representa gráficamente la evolución del error de reconstrucción a lo largo del tiempo. El objetivo es analizar la estabilidad y la velocidad de convergencia del modelo, y comparar configuraciones experimentales para seleccionar la más eficiente, en el siguiente apartado.

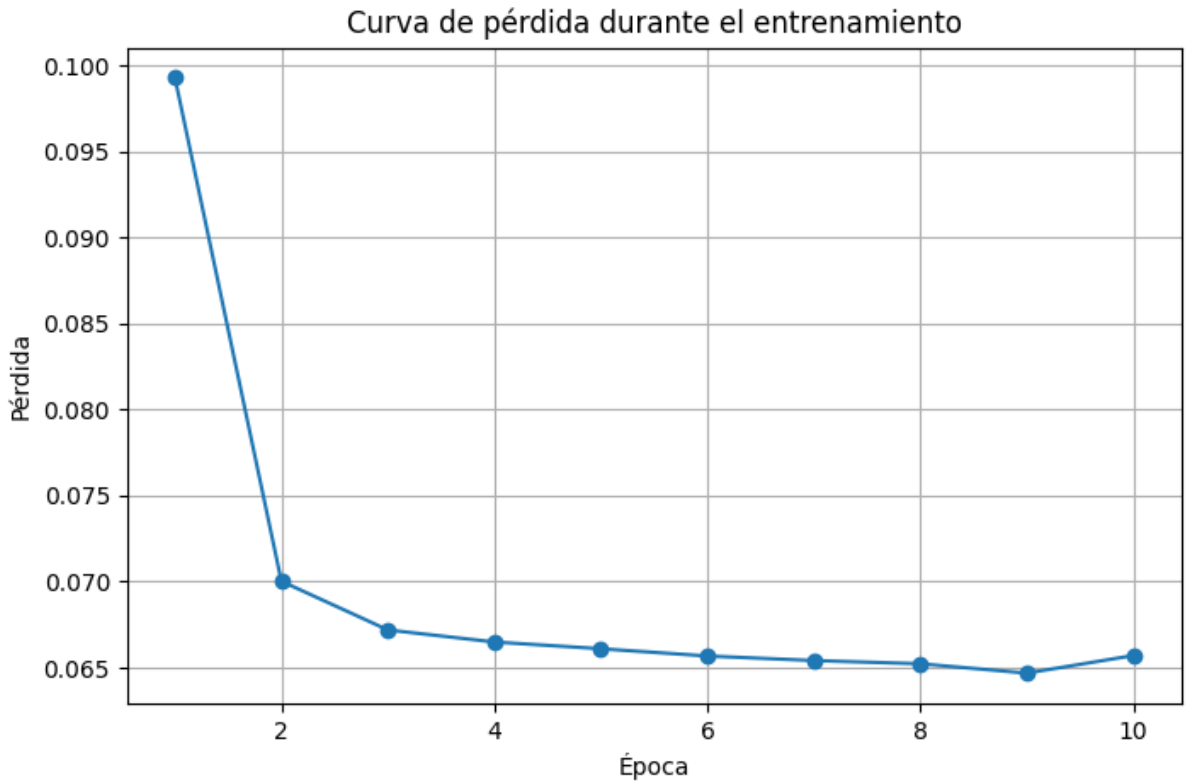

Figura 24: Curva de pérdida AutoEncoder

Al finalizar el entrenamiento, se selecciona aleatoriamente un batch del dataset y se pasa por el AutoEncoder. Se reconstruye las imágenes sin actualizar pesos y se comparan las imágenes originales con las reconstrucciones para evaluar visualmente el rendimiento.
Se introduce un lote de imágenes en modo evaluación (model.eval()), lo que desactiva componentes como Dropout y mantiene constantes las estadísticas de BatchNorm. En este modo, el modelo no actualiza sus pesos, simplemente reconstruye las imágenes con los

parámetros aprendidos. Las reconstrucciones permiten evaluar visualmente la capacidad del modelo para reproducir patrones normales. Si tras muchas épocas de entrenamiento el modelo reconstruye con gran precisión las imágenes de entrenamiento, pero falla al hacerlo con nuevas imágenes normales, esto puede indicar un caso de sobreajuste: el modelo ha memorizado ejemplos concretos en lugar de generalizar los patrones normales.

```python
import torch
import matplotlib.pyplot as plt

def visualizar_reconstrucciones(model, dataset, n=5):
    model.eval()
    loader = torch.utils.data.DataLoader(dataset, batch_size=n, shuffle=True)
    imgs, _ = next(iter(loader))
    imgs = imgs.to(device)
    with torch.no_grad():
        outputs = model(imgs)

    imgs = imgs.cpu() * 0.5 + 0.5  # Desnormalizamos imgs
    outputs = outputs.cpu() * 0.5 + 0.5

    fig, axes = plt.subplots(nrows=2, ncols=n, figsize=(n*2, 4))
    for i in range(n):
        axes[0, i].imshow(imgs[i][0], cmap='gray')
        axes[0, i].set_title('Entrada')
        axes[0, i].axis('off')
        axes[1, i].imshow(outputs[i][0], cmap='gray')
        axes[1, i].set_title('Reconstrucción')
        axes[1, i].axis('off')
    plt.tight_layout()
    plt.show()
visualizar_reconstrucciones(model, train_dataset)
```

Figura 25: Implementación de Código para visualizar reconstrucciones (AutoEncoder)

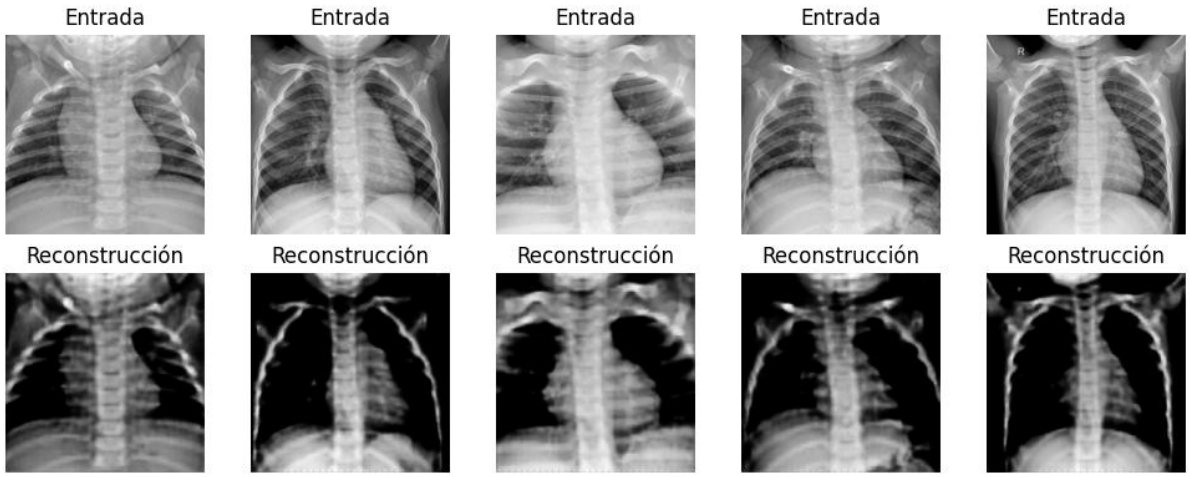

Figura 26: Batch de reconstrucción de Inputs

Una vez que se procede a la reconstrucción de imágenes finalizada la generación del modelo tras el entrenamiento, se puede llegar al siguiente análisis:

- Las reconstrucciones mantienen correctamente la estructura global del tórax: columna vertebral, costillas y campos pulmonares son visibles y bien posicionados.
- Se observa una ligera pérdida de nitidez en los detalles finos (bordes pulmonares, contraste con tejidos blandos), lo cual es esperable al trabajar con una arquitectura comprimida.
- El modelo logra capturar los patrones más relevantes de la imagen sin sobre ajustarse al ruido, lo que sugiere que podría detectar anomalías que se desvíen de estas reconstrucciones, está aprendiendo y no memorizando.

5.4.2 Enfoque experimental y ajuste de hiperparámetros

Dado el carácter experimental del TFG, el entrenamiento del modelo no se limita a una única configuración. Se han explorado distintas variantes con el fin de mejorar el rendimiento:

- Variación del learning rate, se ha probado con valores como 1e-4, 5e-4, 2e-3 para analizar su impacto , ya que un valor menor mejora la estabilidad pero ralentiza la convergencia.

- Ajuste del número de épocas en función del comportamiento de la pérdida. Puede aparecer overfitting. Usando hasta 50 épocas se puede ver si el modelo sigue aprendiendo o si se estabiliza la pérdida.
- Tamaño del batch_size: Valores de 16, 32 y 64 permiten ver cómo afecta al entrenamiento.
- Inclusión o eliminación de Batch Normalization para observar su influencia en la estabilidad del aprendizaje.
- Comparación entre distintos optimizadores: Adam frente a SGD con momentum. Permite ver el impacto sobre la convergencia.

Este proceso iterativo ha permitido no solo validar la arquitectura, sino también afinar los parámetros para mejorar la capacidad de reconstrucción del modelo. Se seguirá esta misma metodología de trabajo con la implementación de los siguientes modelos.

5.4.3 Comparación de configuraciones experimentales (AutoEncoder)

Esta sección recoge los resultados más relevantes de las distintas configuraciones probadas durante el entrenamiento del AutoEncoder. Cada prueba se ha diseñado modificando uno o más hiperparámetros mientras se mantenían constantes el resto, con el fin de aislar su efecto. Aunque en el planteamiento experimental se han considerado muchos hiperparámetros que pueden influir, esta sección recoge las configuraciones probadas y documentadas.

Las configuraciones comparadas incluyen:

- **Base**: lr=1e-3, 10 épocas, batch size 32, optimizador Adam.
- **LR reducido**: lr=5e-4, misma arquitectura, 20 épocas.
- **Más entrenamiento**: lr=1e-3, 50 épocas.
- **Sin Batch Normalization**: arquitectura sin normalización, mismo resto de parámetros.
- **Optimizador SGD**: mismo lr, 10 épocas, con momentum.

Para cada una de estas variantes se ha registrado la pérdida final y la curva completa de entrenamiento, y se han observado las diferencias visuales en la calidad de reconstrucción.

```
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
epochs = 10
```

Figura 27: Fragmento del código a modificar AutoEncoder

A continuación, se muestran las pruebas obtenidas de las distintas configuraciones. En las siguientes gráficas se centran esfuerzos en mostrar la evolución del error de reconstrucción a lo largo del tiempo y la curva de pérdida.

5.4.3.1 LR reducido

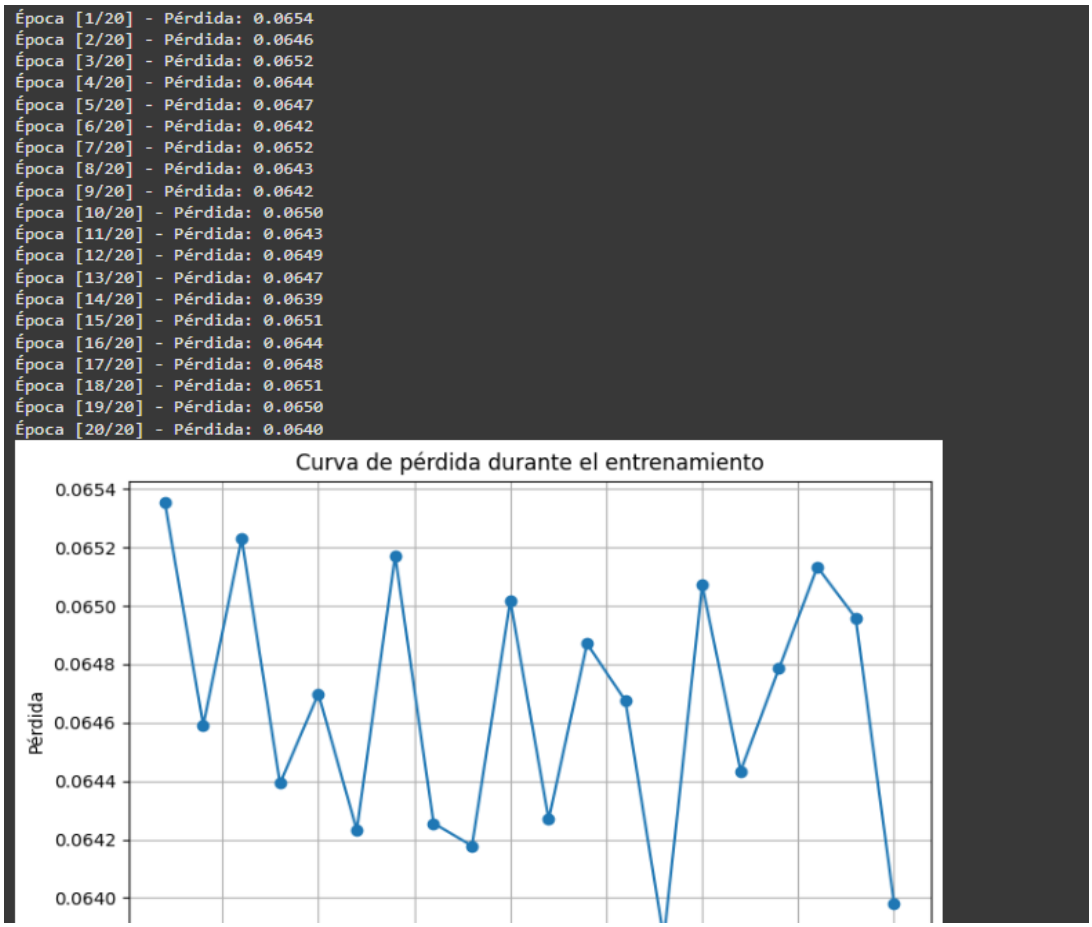

Figura 28: Curva de pérdida entrenamiento AutoEncoder (lr reducido)

5.4.3.2 Más entrenamiento (50 épocas)

```
Época [1/50] - Pérdida: 0.1012
Época [2/50] - Pérdida: 0.0687
Época [3/50] - Pérdida: 0.0664
Época [4/50] - Pérdida: 0.0657
Época [5/50] - Pérdida: 0.0646
Época [6/50] - Pérdida: 0.0652
Época [7/50] - Pérdida: 0.0650
Época [8/50] - Pérdida: 0.0646
Época [9/50] - Pérdida: 0.0648
Época [10/50] - Pérdida: 0.0660
Época [11/50] - Pérdida: 0.0650
Época [12/50] - Pérdida: 0.0650
Época [13/50] - Pérdida: 0.0641
Época [14/50] - Pérdida: 0.0647
Época [15/50] - Pérdida: 0.0648
Época [16/50] - Pérdida: 0.0642
Época [17/50] - Pérdida: 0.0646
Época [18/50] - Pérdida: 0.0647
Época [19/50] - Pérdida: 0.0642
Época [20/50] - Pérdida: 0.0642
Época [21/50] - Pérdida: 0.0652
Época [22/50] - Pérdida: 0.0647
Época [23/50] - Pérdida: 0.0643
Época [24/50] - Pérdida: 0.0639
Época [25/50] - Pérdida: 0.0638
Época [26/50] - Pérdida: 0.0654
Época [27/50] - Pérdida: 0.0638
Época [28/50] - Pérdida: 0.0646
Época [29/50] - Pérdida: 0.0640
Época [30/50] - Pérdida: 0.0644
Época [31/50] - Pérdida: 0.0643
Época [32/50] - Pérdida: 0.0638
Época [33/50] - Pérdida: 0.0642
Época [34/50] - Pérdida: 0.0637
Época [35/50] - Pérdida: 0.0650
Época [36/50] - Pérdida: 0.0642
Época [37/50] - Pérdida: 0.0638
Época [38/50] - Pérdida: 0.0638
Época [39/50] - Pérdida: 0.0646
Época [40/50] - Pérdida: 0.0641
Época [41/50] - Pérdida: 0.0638
Época [42/50] - Pérdida: 0.0639
Época [43/50] - Pérdida: 0.0639
Época [44/50] - Pérdida: 0.0644
Época [45/50] - Pérdida: 0.0638
Época [46/50] - Pérdida: 0.0656
Época [47/50] - Pérdida: 0.0649
Época [48/50] - Pérdida: 0.0634
Época [49/50] - Pérdida: 0.0636
Época [50/50] - Pérdida: 0.0639
```

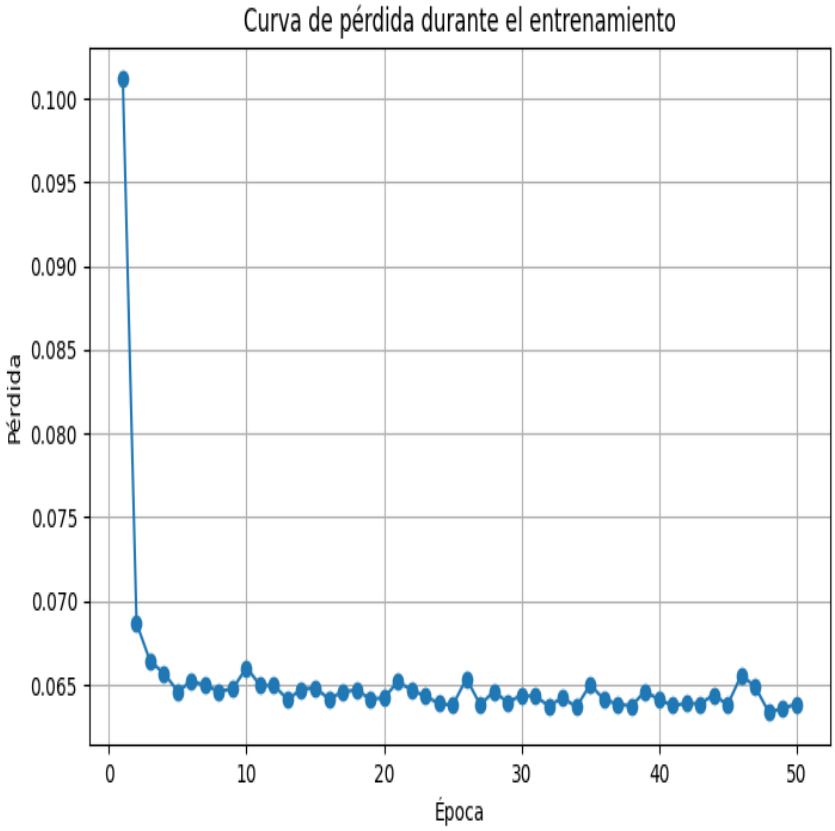

Figura 29: Curva de pérdida entrenamiento AutoEncoder (50 épocas)

5.4.3.3 Sin Batch Normalization

Sin Batch Normalization, en la clase 'ConvAutoEncoder' eliminamos las líneas necesarias

```
class ConvAutoEncoder(nn.Module):
    def __init__(self):
        super(ConvAutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, stride=2, padding=1),  # 128 -> 64
            # nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1), # 64 -> 32
            # nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 128, 3, stride=2, padding=1),# 32 -> 16
            nn.ReLU()
        )
```

Figura 30: Fragmento Código cambio sin Batch Normalization (AutoEncoder)

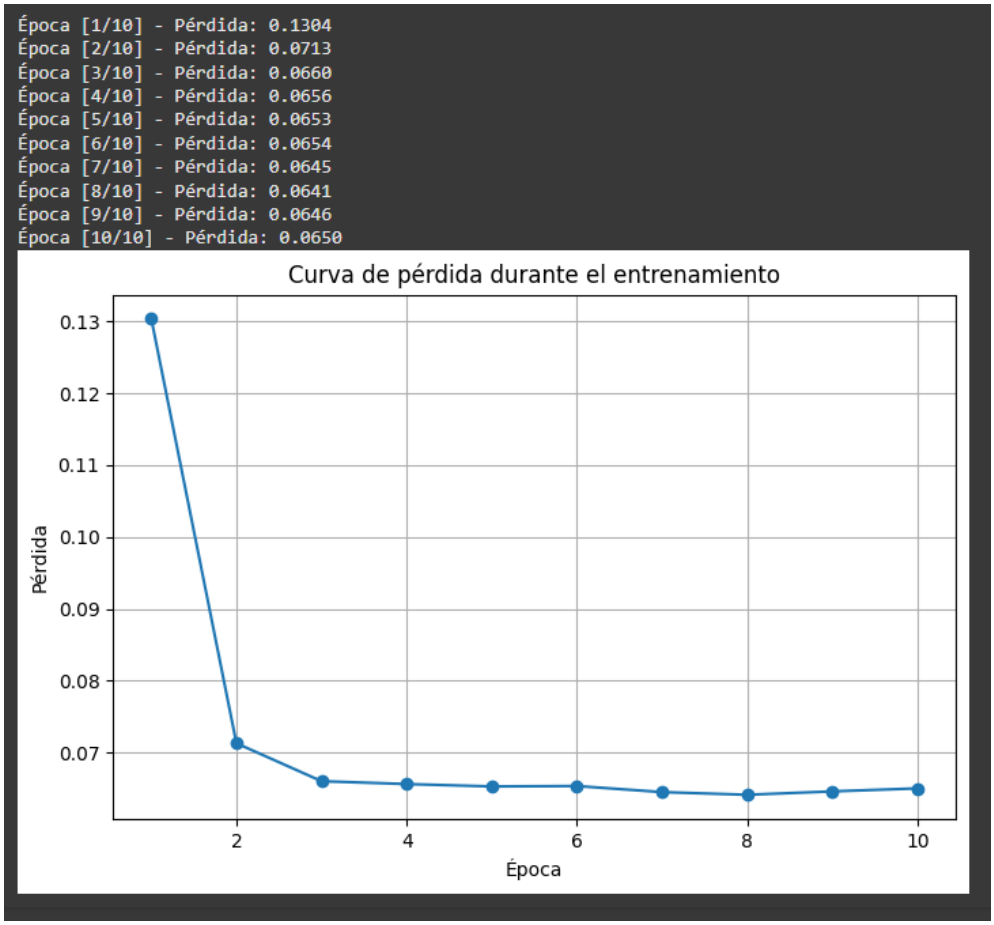

Figura 31: Curva de pérdida entrenamiento sin Batch Normalization (AutoEncoder)

5.4.3.4 Optimizador SGD con momentum = 0.9

Se define el optimizador SGD con momentum = 0.9 para que acelere el gradiente mientras se mantiene estable:

```
optimizer = optim.SGD(model.parameters(), momentum = 0.9, lr=1e-3, weight_decay=1e-5)
```

Figura 32: Fragmento de Código de optimizador SGD (AutoEncoder)

```
Época [1/10] - Pérdida: 0.2899
Época [2/10] - Pérdida: 0.2741
Época [3/10] - Pérdida: 0.2569
Época [4/10] - Pérdida: 0.2436
Época [5/10] - Pérdida: 0.2353
Época [6/10] - Pérdida: 0.2245
Época [7/10] - Pérdida: 0.2159
Época [8/10] - Pérdida: 0.2104
Época [9/10] - Pérdida: 0.2037
Época [10/10] - Pérdida: 0.1989
```

Figura 33: Curva de pérdida entrenamiento optimizador SGD (AutoEncoder)

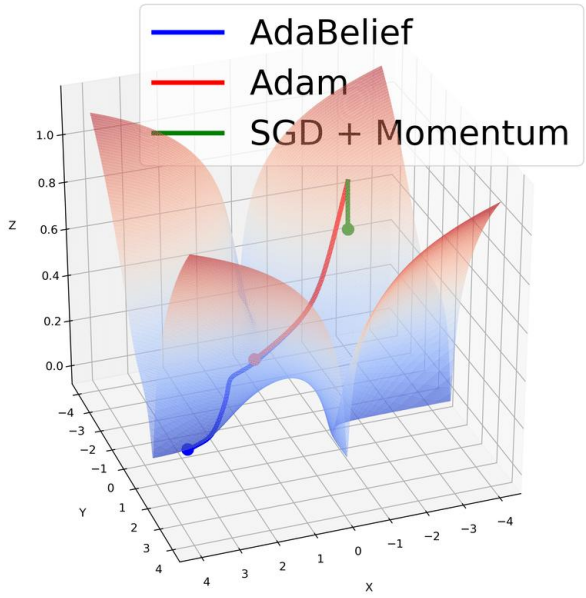

Figura 34: Comparación del comportamiento de optimizadores en una superficie de pérdida

5.4.4 Conclusión de configuraciones experimentales (AutoEncoder)

A continuación, se describen los resultados obtenidos:

- Configuración base **(Adam, 10 épocas, lr=1e-3)**
  Mostró una rápida convergencia en las primeras épocas, con una pérdida final estabilizada en torno a 0.065. Es un buen punto de partida, aunque con solo 10 épocas puede no aprovechar todo el potencial del modelo.
- **Learning rate reducido (Adam, 20 épocas, lr=5e-4)**
  Se observó una pérdida más baja al inicio, pero con más fluctuaciones durante el entrenamiento. Aunque la pérdida media fue similar, el comportamiento no fue tan estable como en otras configuraciones.
- **Entrenamiento extendido (Adam, 50 épocas, lr=1e-3)**
  Esta configuración mantuvo el comportamiento estable del caso base, pero mejoró la suavidad de la curva de pérdida al permitir una convergencia más progresiva. No se aprecian síntomas de sobreajuste hasta la época 50, lo cual valida su uso como configuración óptima en este entorno.
- **Sin Batch Normalization**
  Eliminando las capas de normalización, la pérdida inicial fue significativamente

mayor (~0.13), aunque el modelo logró estabilizarse. Sin embargo, la curva mostró un aprendizaje más lento y menos eficiente. Esto confirma el papel estabilizador de BatchNorm, especialmente en redes profundas con activaciones ReLU.

- **Optimizador SGD con momentum = 0.9**
  Esta configuración presentó la curva más lenta de descenso, con una pérdida final muy superior (~0.198). Aunque SGD puede generalizar mejor en ciertos contextos, en este caso concreto mostró menor capacidad de ajuste y aprendizaje en el mismo número de épocas.

La configuración más equilibrada ha sido con el optimizador Adam con lr=1e-3 y un mayor número de épocas (50). Ofrece una curva de pérdida estable, buen rendimiento final y evita los efectos negativos de la eliminación de BatchNorm o del cambio de optimizador. Esta configuración será utilizada como referencia para comparar los resultados del AutoEncoder frente a los modelos SimCLR y SWSSL.

Por otro lado, durante el experimento con 50 épocas, el cuál era el más propenso a manifestar síntomas de overfitting, no se ha observado indicios de sobreajuste (inversión de la tendencia en la curva de pérdida o aumento del error). En configuraciones más extensas con datasets más pequeños, podría aparecer este fenómeno. En este caso, técnicas como Dropout [11], EarlyStopping o validación cruzada serían recomendables.

5.5 Carga de datos y preprocesamiento SimCLR

SimCLR representa el enfoque contrastivo dentro del conjunto de modelos auto-supervisados implementados en este trabajo. A diferencia del AutoEncoder, que aprende reconstruyendo la entrada a partir de su representación comprimida, SimCLR persigue la obtención de representaciones **invariantes** aplicando una **pérdida contrastiva** sobre pares de vistas distintas generadas a partir de una misma imagen [3].

El objetivo es que estas dos vistas —que mantienen la semántica general pero presentan alteraciones en aspectos de bajo nivel— se proyecten cerca en el espacio latente, mientras que las vistas de imágenes distintas se alejen. Esta idea permite entrenar el modelo sin necesidad de etiquetas manuales, aprovechando exclusivamente la estructura de los datos, algo especialmente útil en entornos médicos con escasez de anotaciones [8].

5.5.1 Estructura del conjunto de datos y organización de vistas

El dataset utilizado es el mismo que en el resto de modelos: PadChest, específicamente filtrado para incluir únicamente radiografías de tórax normales (clase "NORMAL"). Este filtrado busca mantener condiciones homogéneas para todos los modelos y asegurar que las representaciones aprendidas se basan únicamente en anatomía sana, lo cual permite posteriormente detectar desviaciones como anomalías.

La estructura del conjunto de datos se conserva exactamente igual que en el AutoEncoder (ver apartado 5.2.1). A cada imagen del conjunto se le aplican dos veces de forma independiente un conjunto de transformaciones aleatorias, generando así un **par positivo**. El resto de imágenes del batch actúan implícitamente como **pares negativos**.

5.5.2 Transformaciones contrastivas aplicadas

El éxito de SimCLR depende en gran medida del diseño de las transformaciones aleatorias aplicadas (augmentations). Según el estudio original [3], estas transformaciones deben ser lo suficientemente fuertes como para otorgar variación, pero sin destruir la semántica de la imagen. En este trabajo se ha optado por una secuencia contrastiva adaptada al contexto médico, implementada en una clase personalizada de PyTorch denominada SimCLRTransform.

Las transformaciones aplicadas son:

- **RandomResizedCrop (recorte aleatorio con escala):** fuerza al modelo a aprender invariancia frente al encuadre y tamaño del objeto principal.
- **RandomHorizontalFlip (inversión horizontal):** útil en contextos de simetría anatómica, como el tórax.
- **ColorJitter (distorsión de color):** aunque el dataset es en escala de grises, esta transformación añade robustez al modificar contraste y brillo [7].
- **RandomGrayscale:** refuerza la generalización al reducir dependencia de detalles en la tonalidad.
- **GaussianBlur (difuminado):** simula desenfoques o condiciones de imagen más degradadas.
- **ToTensor y Normalize:** convierte a tensores y normaliza valores de píxel a media 0.5 y desviación 0.5, igual que en el AutoEncoder.

Cada transformación se aplica de manera estocástica e independiente a las dos vistas generadas por cada imagen.

```python
class SimCLRTransform:
    def __init__(self):
        self.base_transform = transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.RandomApply([transforms.ColorJitter(0.4, 0.4, 0.4, 0.1)], p=0.8),
            transforms.RandomGrayscale(p=0.2),
            transforms.ToTensor(),
            transforms.Normalize((0.5,), (0.5,))
        ])

    def __call__(self, x):
        return self.base_transform(x), self.base_transform(x)
```

Figura 35: Clase SimCLRTransform

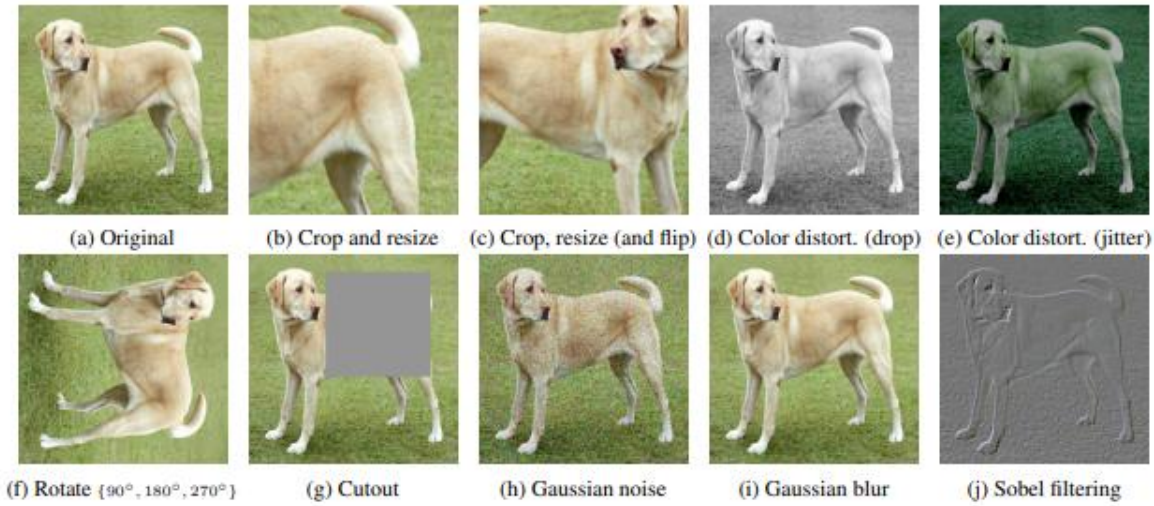

Figura 36: Ejemplos de transformaciones de datos utilizadas en SimCLR

5.5.3 Carga de datos mediante DataLoader

Una vez definidas las transformaciones específicas, se utiliza la clase ImageFolder de PyTorch para organizar el conjunto de datos. En este caso, cada imagen procesada mediante SimCLRTransform produce dos vistas distintas que actuarán como par positivo. El resto de imágenes del batch se consideran pares negativos.

La opción 'shuffle=True' garantiza que las imágenes se mezclen aleatoriamente en cada época, mejorando la generalización del modelo evitando el sobreajuste. Esta organización del dataset permite mantener las mismas condiciones experimentales que el AutoeEncoder, para tener una comparación equitativa entre los modelos.

```
[23] transform = SimCLRTransform()
     #Se escoge el dataset solo con imágenes normales para el entrenamiento
     dataset = datasets.ImageFolder("chest_xray/chest_xray/train", transform=transform)
     dataset.samples = [s for s in dataset.samples if s[1] == 0]

     train_loader = DataLoader(dataset, batch_size = 8, shuffle=True, num_workers=2, pin_memory=True)
```

```
print("Tamaño real del dataset:", len(dataset))
print("Batches por época:", len(train_loader))
```

```
Tamaño real del dataset: 1349
Batches por época: 169
```

Figura 37: Fragmento de código DataLoader y tamaño dataset (SimCLR)

5.6 Implementación del modelo SimCLR

5.6.1 Introducción conceptual

Una vez definido el proceso de carga de datos y las transformaciones contrastivas específicas de SimCLR, se presenta la implementación del modelo, centrada en los elementos estructurales del código, que permiten llevar a cabo el entrenamiento de forma auto-supervisada.

El modelo SimCLR se ha construido mediante dos componentes principales: un encoder convolucional, que actúa como extractor de características latentes, y una cabeza de proyección (projection head), que transforma esas representaciones para optimizar la pérdida contrastiva.

Esta división permite que las salidas del encoder (h) se conserven como representaciones útiles para el futuro, mientras que las proyecciones (z) son utilizadas únicamente durante el entrenamiento contrastivo [3].

La implementación ha seguido los principios de simplicidad y reutilización ya aplicados en el AutoEncoder. De hecho, el encoder utilizado (ResNet18) está preparado para ser reaprovechado como columna vertebral de otros modelos auto-supervisados como posteriormente SWSSL favoreciendo una comparación posterior lo más equitativa posible.

Este diseño modular, no solo se alinea con la propuesta original de SimCLR [3], sino que también facilita la interpretación y análisis del comportamiento del modelo en entornos médicos, donde la interpretabilidad y consistencia son factores clave [13].

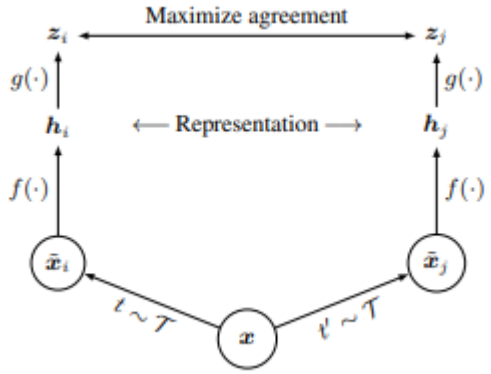

Figura 38: Esquema general del marco de aprendizaje contrastivo de SimCLR

5.6.2 Arquitectura del modelo

El modelo SimCLR implementado se compone de dos bloques funcionales: un **encoder convolucional** basado en ResNet18 y una **cabeza de proyección** formada por una MLP. Esta arquitectura refleja la propuesta original [3], aunque con adaptaciones específicas para ajustarse al entorno computacional disponible y a los objetivos del presente trabajo.

Encoder:

El encoder se ha construido a partir de la arquitectura ResNet18 disponible en torchvision.models. Para su uso como extractor de características, se han eliminado las capas finales de clasificación (fully connected), manteniendo únicamente las capas convolucionales y de pooling. De esta forma, cada imagen se transforma en un vector latente h de 512 dimensiones que codifica la información semántica relevante.

Proyección (projection head):

La salida h del encoder se introduce en una red MLP de dos capas lineales con activación intermedia ReLU, cuya función es proyectar las representaciones en un nuevo espacio latente z. Esta transformación no se utiliza en tareas posteriores, sino únicamente durante el entrenamiento, para aplicar la pérdida contrastiva. La dimensión final de z se ha fijado en 64 (en lugar de 128 como el diseño original) para reducir coste computacional y adaptarse a los recursos disponibles.

Este diseño modular permite reutilizar el encoder en otros modelos del trabajo (como SWSSL) y comparar las representaciones h entre distintos enfoques.

```
[26] class Encoder(nn.Module):
         def __init__(self):
             super().__init__()
             base = models.resnet18(pretrained=False)
             self.encoder = nn.Sequential(*list(base.children())[:-1])

         def forward(self, x):
             return self.encoder(x).squeeze()

     class ProjectionHead(nn.Module):
         def __init__(self, in_dim=512, out_dim=64):
             super().__init__()
             self.proj = nn.Sequential(
                 nn.Linear(in_dim, 512),
                 nn.ReLU(),
                 nn.Linear(512, out_dim)
             )

         def forward(self, x):
             return self.proj(x)

     class SimCLR(nn.Module):
         def __init__(self):
             super().__init__()
             self.encoder = Encoder()
             self.projection = ProjectionHead()

         def forward(self, x):
             h = self.encoder(x)
             z = self.projection(h)
             return z
```

Figura 39: Arquitectura del modelo SimCLR con encoder ResNet18 y cabeza de proyección. El vector h se conserva como representación útil, el vector z se utiliza para la pérdida contrastiva NT-Xent.

5.6.3 Código del modelo SimCLR

A continuación, se presenta en detalle la implementación en código de los componentes que conforman el modelo SimCLR. Si bien la arquitectura general ha sido ya descrita en el apartado anterior, en este bloque se analiza con mayor profundidad la estructura funcional de cada submódulo, mostrando el código específico empleado en la práctica.

La implementación se ha organizado en tres clases diferenciadas, que permiten aislar claramente las responsabilidades de cada parte del modelo:

- Encoder: extrae las representaciones semánticas de las imágenes de entrada.
- ProjectionHead: transforma esas representaciones a un espacio latente optimizado para la pérdida contrastiva.
- SimCLR: integra ambos componentes y gestiona el flujo de datos completo durante el entrenamiento.

Esta separación permite trabajar con una arquitectura más legible, fácilmente modificable y reutilizable en los modelos posteriores. A nivel técnico, todas las clases heredan de 'nn.Module', y están construidas en PyTorch. Ahora se desglosa cada fragmento del modelo.

5.6.3.1 Clase Encoder

- Se importa una ResNet18 sin pesos preentrenados ('pretrained=False') desde 'torchvision.models'.
- Se eliminan las capas de clasificación mediante 'list(resnet.children())[:-1]' que extrae todas las capas excepto la última capa de clasificación (fc). Esto convierte la red en un extractor de características.
- Las capas se agrupan en un 'nn.Sequential', lo que permite recorrerlas en orden como un único bloque.
- En el método forward, se transforma la imagen de entrada x y luego se aplana con '.view(...)' para obtener un vector de 512 dimensiones por muestra, eliminando las dimensiones espaciales [1, 1].

Como resultado se obtiene un vector latente compacto (h) que representa el contenido semántico de la imagen.

```python
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        resnet = models.resnet18(pretrained=False)
        self.encoder = nn.Sequential(*list(resnet.children())[:-1])

    def forward(self, x):
        return self.encoder(x).squeeze()
```

Figura 40: Clase Encoder (SimCLR)

5.6.3.2 Clase Projection Head

- Se define una MLP de dos capas: la primera conserva la dimensión (512 -> 512) y se acompaña de una activación ReLU. La segunda proyecta a un espacio latente reducido (512 ->64).
- Esta estructura sigue la propuesta original de SimCLR (excepto 'out_dim'), donde se comprobó que añadir una capa intermedia mejoraba el rendimiento en el espacio contrastivo.
- No se incluyen capas de normalización (BatchNorm) ni dropout, dado que el objetivo es analizar el comportamiento base sin regularización adicional.

Como resultado se obtiene una representación proyectada (z) lista para calcular la pérdida contrastiva NT-Xent.

Cabe destacar que si se calcula la pérdida contrastiva directamente sobre la salida del encoder (h), el modelo tiene a sobre ajustarse a esa representación.

```python
class ProjectionHead(nn.Module):
    def __init__(self, in_dim=512, out_dim=64):
        super().__init__()
        self.proj = nn.Sequential(
            nn.Linear(in_dim, 512),
            nn.ReLU(),
            nn.Linear(512, out_dim)
        )

    def forward(self, x):
        return self.proj(x)
```

Figura 41: Clase ProjectionHead

5.6.3.3 Clase SimCLR

- La clase SimCLRModel combina ambos bloques en un flujo completo.
- El método forward devuelve tanto el vector de características h como la proyección z
- Esta decisión tiene un valor práctico importante: permite usar h en tareas

```python
class SimCLRModel(nn.Module):
    def __init__(self):
        super(SimCLRModel, self).__init__()
        self.encoder = Encoder()
        self.projection = ProjectionHead()

    def forward(self, x):
        h = self.encoder(x)
        z = self.projection(h)
        return h, z
```

Figura 42: Clase SimCLRModel

5.7 Entrenamiento del modelo SimCLR

Una vez definida la arquitectura del modelo, se procede al entrenamiento bajo un enfoque de aprendizaje contrastivo. El objetivo es que las representaciones generadas por el encoder sean invariantes ante las transformaciones aplicadas a las imágenes de entrada, preservando su semántica incluso cuando cambian aspectos como el brillo, la orientación o el encuadre.

Este proceso se entrena sin necesidad de etiquetas manuales, comparando directamente las salidas generadas por pares de vistas de la misma imagen frente a las del resto del batch. Para ello, se emplea una función de pérdida específica denominada NT-Xent (Normalized Temperature-scaled Cross Entropy Loss).

5.7.1 Función de pérdida NT-Xent

La función de pérdida utilizada para entrenar el modelo SimCLR es la NT-Xent (Normalized Temperature-scaled Cross Entropy Loss). Esta función fue propuesta originalmente por Chen et al. [3] como parte de la arquitectura SimCLR, y se ha consolidado como un estándar en el aprendizaje auto-supervisado basado en contrastes. Su objetivo es

maximizar la similitud entre pares positivos (dos vistas distintas de una misma imagen) y minimizar la similitud con el resto de ejemplos del batch, considerados como negativos.

En esta implementación, la pérdida se calcula siguiendo el siguiente flujo:

- A partir de un batch de N imágenes, se generan dos vistas aumentadas (x_1, x_2) para cada una. Al pasar por el modelo SimCLR, se obtienen los vectores proyectados z_1 y z_2, ambos de dimensión [N, d].

- Estos vectores se concatenan verticalmente en un único tensor z de tamaño [2N, d]. Se aplica una normalización L2 por fila para que el producto escalar entre vectores corresponda a la similitud coseno.

- Se calcula una matriz de similitud entre todos los vectores z mediante un producto escalar entre pares:
  'sim_matrix = torch.matmul(z, z.T) / temperature'
  El resultado es una matriz [2N, 2N] donde cada posición [i, j] representa la similitud entre los vectores z_i y z_j.

- Para evitar que el modelo se "auto-compare", se elimina la diagonal de la matriz (auto-similitudes) con una máscara:
  'mask = torch.eye(2 * batch_size, dtype=torch.bool).to(z.device)'
  'sim_matrix = sim_matrix.masked_fill(mask, float('-inf'))'

- A continuación, se extrae el par positivo de cada muestra mediante las diagonales desplazadas ±N. Por ejemplo, z_0 tiene como positivo a z_0+N, y z_n a z_n–N: 'positives = torch.cat([torch.diag(sim_matrix, batch_size), torch.diag(sim_matrix, -batch_size)])'
- La pérdida NT-Xent se calcula entonces como una entropía cruzada estándar entre estos logits y los índices de las verdaderas parejas:
  'loss = F.cross_entropy(logits, labels)'

Finalmente, se construyen los **logits** para la entropía cruzada:

Cada fila del tensor logits contiene primero el valor de su par positivo, seguido de sus similitudes con el resto del batch.

Las etiquetas (labels) son ceros, indicando que el par positivo está en la primera columna de cada fila. 'logits = torch.cat([positives.unsqueeze(1), sim_matrix], dim=1)'
'labels = torch.zeros(2 * batch_size, dtype=torch.long).to(z.device)'

Esta implementación garantiza un comportamiento más estable en entornos de bajo batch size y ofrece un control más claro sobre la estructura de los pares positivos y negativos en el entrenamiento contrastivo.

```python
import torch
import torch.nn.functional as F

def nt_xent_loss(z1, z2, temperature=0.5):
    batch_size = z1.size(0)
    z = torch.cat([z1, z2], dim=0)
    z = F.normalize(z, dim=1)

    sim_matrix = torch.matmul(z, z.T) / temperature
    labels = torch.arange(batch_size, device=z.device)
    labels = torch.cat([labels + batch_size, labels])

    # Crear máscara para ignorar la diagonal
    mask = torch.eye(2 * batch_size, dtype=torch.bool, device=z.device)
    sim_matrix = sim_matrix.masked_fill(mask, float('-inf'))

    # Calcular la similitud del positivo para cada fila
    positives = torch.cat([
        torch.diag(sim_matrix, batch_size),
        torch.diag(sim_matrix, -batch_size)
    ])

    # Logits: positivos y todo el resto como negativos
    logits = torch.cat([positives.unsqueeze(1), sim_matrix], dim=1)
    labels = torch.zeros(2 * batch_size, dtype=torch.long, device=z.device)

    return F.cross_entropy(logits, labels)
```

Figura 43: Fragmento de código: función NT-Xent

5.7.2 Configuración del entrenamiento

El entrenamiento base del modelo SimCLR se ha realizado utilizando el optimizador Adam, con una tasa de aprendizaje inicial de 1e-3, y un batch size de 8. Esta configuración ha sido seleccionada tras comprobar, en entornos con recursos computacionales limitados como Google Colab, que tamaños de lote mayores provocaban cuellos de botella durante el entrenamiento contrastivo.

La dimensión de salida de la projection head se ha fijado en 64, en lugar de los 128 propuestos en el trabajo original, como medida para reducir el coste computacional sin renunciar a la expresividad del espacio latente. El modelo ha sido entrenado durante 10 épocas, aunque, para reducir tiempos en pruebas exploratorias, el número de batches por época se ha limitado a **25** mediante una condición explícita en el bucle de entrenamiento.

Esta decisión se justifica por la duración media de cada batch (superior a 10 segundos), que hace inviable un entrenamiento completo de más de 150 lotes por época en una fase inicial.

Posteriormente, se plantea un bloque de experimentos donde se evaluarán distintas configuraciones, variando tanto el número de épocas como el número de lotes por época y la dimensionalidad del espacio proyectado.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimCLRModel().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
epochs = 10
temperature = 0.5
```

Figura 44: Hiperparámetros entrenamiento SimCLR

5.7.3 Bucle de entrenamiento

Durante cada época de entrenamiento, el modelo recorre los datos proporcionados por el DataLoader, que en el caso de SimCLR devuelve pares de vistas aumentadas (x_1, x_2) generadas a partir de una misma imagen mediante el pipeline definido en el apartado 5.6.2. Estas vistas se generan de forma independiente mediante transformaciones aleatorias como recortes, inversión horizontal o alteraciones de color, asegurando que cada par comparta la misma semántica aunque visualmente sean distintos.

Cada par (x_1, x_2) actúa como un par positivo, mientras que el resto de elementos del batch actúan implícitamente como pares negativos. Esta estructura permite un entrenamiento completamente auto-supervisado, sin etiquetas manuales.

Ambas vistas se introducen por separado en el modelo SimCLR, que las procesa a través del encoder (ver 5.7.2) para obtener representaciones latentes h_1 y h_2. Estas se proyectan mediante la ProjectionHead, generando los vectores z_1 y z_2. Estos vectores proyectados son los que se utilizan para calcular la pérdida contrastiva NT-Xent, que penaliza la dispersión entre pares positivos y premia su cercanía relativa respecto al resto del batch.

Finalmente, se aplica la retropropagación y se realiza la actualización de pesos, acumulando la pérdida media por época para su posterior análisis. En la sección siguiente se presentan los resultados obtenidos con esta configuración base y se introduce un bloque de experimentación adicional, en el que se comparan distintas configuraciones de entrenamiento para analizar su impacto en la convergencia y la calidad de las representaciones.

```python
import time

for epoch in range(epochs):
    start_epoch = time.time()
    total_loss = 0
    for i, ((x1, x2), _) in enumerate(train_loader):
        print(f"Procesando nuevo batch {i+1}...")
        if i >= 25: break # Para no tener tantos batches por época, si no el entrenamiento tarda mucho
        start = time.time()
        x1, x2 = x1.to(device), x2.to(device)
        z1 = model(x1)
        z2 = model(x2)

        loss = nt_xent_loss(z1, z2)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        print(f"--> Lote {i+1} procesado en {time.time() - start:.2f}s")

    avg_loss = total_loss / len(train_loader)
    print(f"Época [{epoch+1}/{epochs}] - Pérdida: {avg_loss:.4f} - Tiempo total: {time.time() - start_epoch:.2f}s")
```

Figura 45: Bucle entrenamiento SimCLR

```
Época [1/10] - Pérdida: 0.3923 - Tiempo total: 108.24s
Época [5/10] - Pérdida: 0.3944 - Tiempo total: 108.51s
Época [10/10] - Pérdida: 0.3933 - Tiempo total: 110.62s
```

Figura 46: Evolución pérdida épocas configuración base (SimCLR)

5.7.4 Visualización de la pérdida

Al finalizar el entrenamiento, se genera una curva de pérdida para analizar la evolución de la optimización.

```
[ ]  plt.figure(figsize=(8, 5))
     plt.plot(range(1, epochs+1), losses, marker='o')
     plt.title("Curva de pérdida contrastiva durante el entrenamiento")
     plt.xlabel("Época")
     plt.ylabel("Pérdida")
     plt.grid(True)
     plt.show()
```

Figura 47: Fragmento Código para visualizar gráfica curva de pérdida entrenamiento (SimCLR)

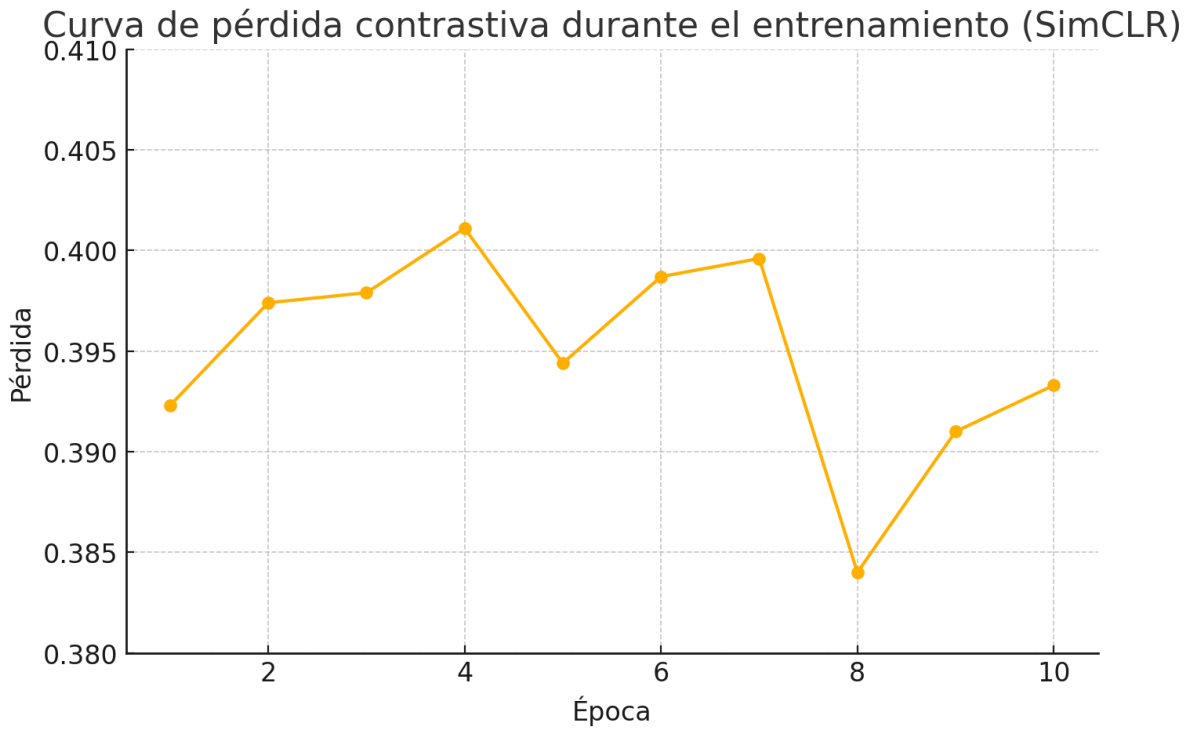

Figura 48: Curva de pérdida contrastiva (SimCLR)

Aunque en la propuesta original de SimCLR se empleaban batch sizes elevados (≥ 128) para maximizar la cantidad de pares negativos por iteración, en este trabajo se ha optado por tamaños reducidos (batch size = 8) debido a las restricciones del entorno de ejecución. Esta decisión puede afectar negativamente a la calidad de la pérdida contrastiva NT-Xent (como se

puede ver en la imagen) dado que un menor número de negativos limita la información sobre las representaciones aprendidas. No obstante, se ha comprobado que incluso en este escenario reducido, el entrenamiento puede completarse y es suficiente para realizar análisis comparativos, que se abordarán en el siguiente apartado.

5.7.5 Comparación de configuraciones experimentales (SimCLR)

Se ha desarrollado una serie de experimentos controlados en los que se modifican tres elementos clave en el entrenamiento de SimCLR con el fin de analizar el impacto de los distintos hiperparámetros. Estos tres elementos son la dimensión del espacio proyectado ('out_dim'), el tamaño del batch ('batch_size)' y el número de épocas de entrenamiento (epochs). Todos los entrenamientos se han realizado con un límite de 25 batches por época, con el fin de garantizar eficiencia computacional en Google Colab, como se justificó en el apartado 5.8.2

Configuración	'out_dim'	'batch_size'	'epochs'	Batches por época
Configuración base del modelo	64	8	10	25
Proyección latente más comprimida	32	8	10	25
Menor número de negativos por batch	64	4	10	25
Doble de épocas (entrenamiento más largo)	64	8	20	25

Tabla 4: Resumen distintas configuraciones experimentales SimCLR

5.7.5.1 Configuración con proyección latente más comprimida

La progresión de la curva de pérdida contrastiva es a la baja respecto la configuración base. La gráfica sugiere una mejor convergencia.

```
Época [1/10] - Pérdida: 0.3839 - Tiempo total: 111.17s
Época [2/10] - Pérdida: 0.3731 - Tiempo total: 109.62s
Época [3/10] - Pérdida: 0.3655 - Tiempo total: 107.92s
Época [4/10] - Pérdida: 0.3527 - Tiempo total: 107.93s
Época [5/10] - Pérdida: 0.3350 - Tiempo total: 107.55s
Época [6/10] - Pérdida: 0.3251 - Tiempo total: 107.91s
Época [7/10] - Pérdida: 0.3163 - Tiempo total: 111.51s
Época [8/10] - Pérdida: 0.3097 - Tiempo total: 107.77s
Época [9/10] - Pérdida: 0.2830 - Tiempo total: 111.74s
Época [10/10] - Pérdida: 0.2928 - Tiempo total: 107.08s
```

Figura 49: Evolución pérdida épocas proyección latente más comprimida (SimCLR)

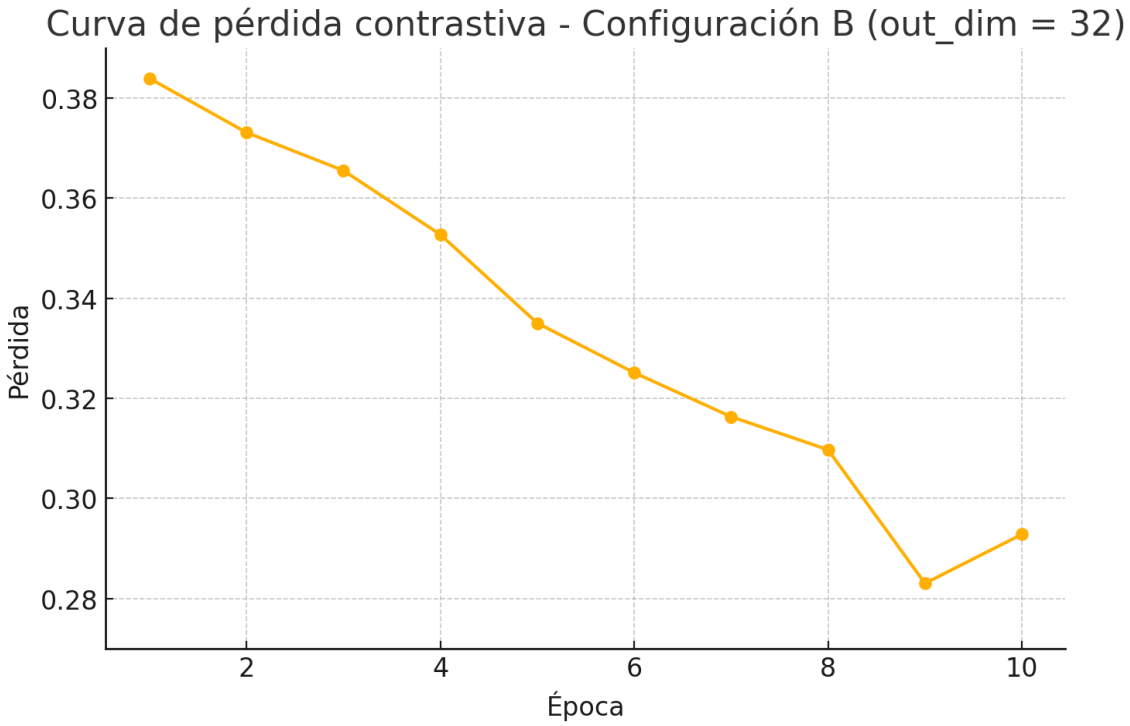

Figura 50: Curva de pérdida contrastiva proyección latente más comprimida (SimCLR)

5.7.5.2 Configuración con menor número de negativos por batch

Mejora respecto a la anterior configuración, el aprendizaje no se ve perjudicado pese al menor número de negativos por batch (probablemente los ejemplos positivos estén mejor aislados).

```
Época [1/10] - Pérdida: 0.1022 - Tiempo total: 58.14s
Época [2/10] - Pérdida: 0.1070 - Tiempo total: 58.05s
Época [3/10] - Pérdida: 0.1101 - Tiempo total: 57.40s
Época [4/10] - Pérdida: 0.0987 - Tiempo total: 57.92s
Época [5/10] - Pérdida: 0.0922 - Tiempo total: 57.07s
Época [6/10] - Pérdida: 0.0994 - Tiempo total: 58.27s
Época [7/10] - Pérdida: 0.0950 - Tiempo total: 57.75s
Época [8/10] - Pérdida: 0.0915 - Tiempo total: 58.78s
Época [9/10] - Pérdida: 0.0907 - Tiempo total: 57.22s
Época [10/10] - Pérdida: 0.0886 - Tiempo total: 58.80s
```

Figura 51: Evolución pérdida épocas menor número de negativos por batch (SimCLR)

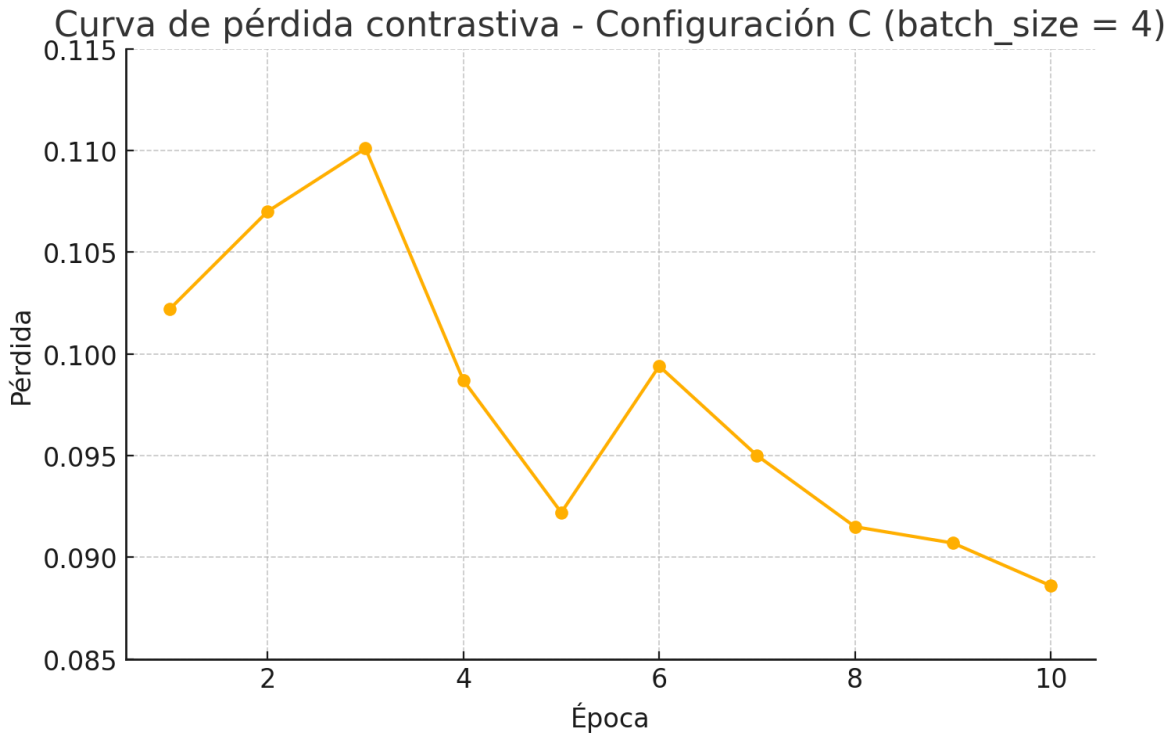

Figura 52: Curva de pérdida contrastiva menor número de negativos por batch (SimCLR)

5.7.5.3 Doble de épocas (entrenamiento más largo)

Demuestra que el modelo puede seguir aprendiendo con más iteraciones, pero tampoco es algo demasiado determinante respecto a otras configuraciones.

```
Época [1/20] - Pérdida: 0.4093 - Tiempo total: 109.73s
Época [2/20] - Pérdida: 0.4044 - Tiempo total: 111.82s
Época [3/20] - Pérdida: 0.4027 - Tiempo total: 108.42s
Época [4/20] - Pérdida: 0.4070 - Tiempo total: 112.22s
Época [5/20] - Pérdida: 0.4031 - Tiempo total: 109.03s
Época [6/20] - Pérdida: 0.4072 - Tiempo total: 110.51s
Época [7/20] - Pérdida: 0.4030 - Tiempo total: 107.83s
Época [8/20] - Pérdida: 0.3998 - Tiempo total: 111.68s
Época [9/20] - Pérdida: 0.3999 - Tiempo total: 108.40s
Época [10/20] - Pérdida: 0.3950 - Tiempo total: 109.41s
Época [11/20] - Pérdida: 0.4097 - Tiempo total: 108.16s
Época [12/20] - Pérdida: 0.4019 - Tiempo total: 109.41s
Época [13/20] - Pérdida: 0.3974 - Tiempo total: 108.87s
Época [14/20] - Pérdida: 0.3989 - Tiempo total: 109.23s
Época [15/20] - Pérdida: 0.3984 - Tiempo total: 108.90s
Época [16/20] - Pérdida: 0.3954 - Tiempo total: 108.59s
Época [17/20] - Pérdida: 0.3945 - Tiempo total: 108.87s
Época [18/20] - Pérdida: 0.3982 - Tiempo total: 108.99s
Época [19/20] - Pérdida: 0.3865 - Tiempo total: 107.88s
Época [20/20] - Pérdida: 0.3676 - Tiempo total: 108.12s
```

Figura 53: Evolución pérdida épocas doble de épocas (SimCLR)

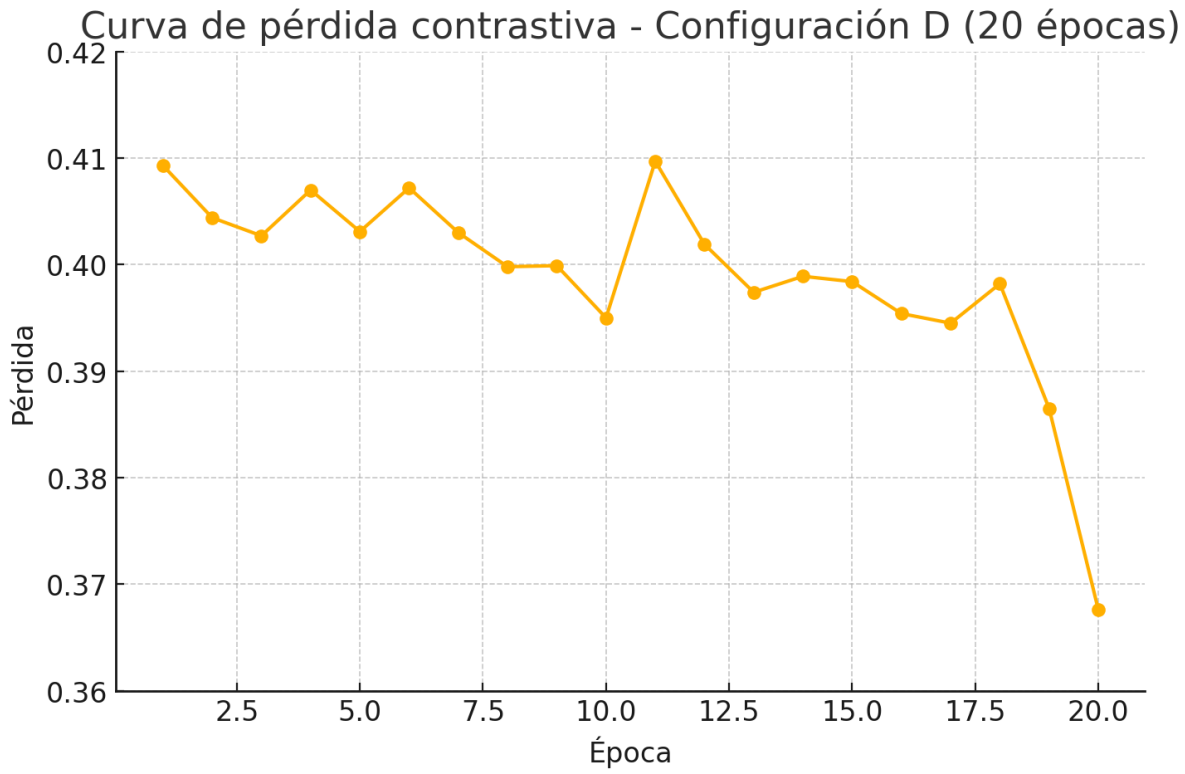

Figura 54: Curva de pérdida contrastiva doble de épocas (SimCLR)

5.7.6 Conclusión de configuraciones experimentales (SimCLR)

Tras comparar diferentes configuraciones del modelo SimCLR, se observa que ajustes como la reducción del 'out_dim', el uso de batch sizes menores o la ampliación del número de épocas afectan de forma significativa a la pérdida. En particular, la configuración con menor número de negativos por batch (batch_size = 4) han mostrado mayor descenso de la pérdida. Cabe destacar que es probable que a cuánto mayor número de épocas mayor capacidad del modelo para discriminar entre ejemplos positivos y negativos, se tendrá en cuenta para la evaluación y comparación de modelos a modo de configuración final.

5.8 Carga de datos y preprocesamiento de SWSSL

El modelo SWSSL (Sliding Window-based Self-Supervised Learning) ha sido implementado en este trabajo a partir de una reimplementación previa desarrollada por el autor y publicada en un repositorio personal de GitHub [15]. Esta versión, inspirada en el código original del paper de Dong et al. [1], adapta la arquitectura al entorno de ejecución utilizado en este TFG (Google Colab) y mantiene la compatibilidad con el resto de modelos (AutoEncoder y SimCLR) para permitir una comparación equitativa.

El repositorio público contiene todos los elementos clave del modelo: encoder convolucional compartido (ResNet18), lógica de ventanas deslizantes, proyección contrastiva, reconstrucción y función de pérdida compuesta. Esta base ha sido empleada y adaptada en el presente trabajo para integrar SWSSL dentro del mismo pipeline de entrenamiento y evaluación.

5.8.1 Introducción conceptual de SWSSL

SWSSL (*Sliding Window-based Self-Supervised Learning*) es un enfoque reciente de aprendizaje auto-supervisado especialmente diseñado para detectar anomalías en imágenes de alta resolución, como radiografías médicas [1]. Su propuesta combina dos estrategias complementarias: el aprendizaje contrastivo local a través de ventanas deslizantes y la reconstrucción de imagen, lo que permite aprovechar tanto la estructura semántica global como la información contextual localizada.

A diferencia de modelos como SimCLR, que tratan cada imagen como una unidad global, SWSSL aplica la lógica contrastiva sobre subregiones obtenidas mediante un esquema de

ventanas deslizantes sobre el mapa de características del encoder. Esto facilita el aprendizaje de patrones locales invariantes, fundamentales en imágenes médicas donde pequeñas anomalías pueden tener gran relevancia clínica.

En este trabajo se ha utilizado como punto de partida una reimplementación pública desarrollada previamente por el autor y disponible en GitHub [14], que adapta fielmente el modelo descrito por Dong et al. [1] a un entorno accesible como Google Colab, y lo integra con el mismo encoder y pipeline de evaluación empleados en el resto de los modelos del estudio (AutoEncoder y SimCLR). Esta consistencia técnica permite realizar una comparación objetiva y reproducible entre los distintos enfoques auto-supervisados implementados.

5.8.2 Estructura del conjunto de datos y organización de vistas

Al igual que en los modelos anteriores, para SWSSL se ha utilizado el conjunto de datos PadChest, filtrado previamente para incluir únicamente imágenes de tórax normales. Este filtrado garantiza que el aprendizaje auto-supervisado se realice exclusivamente sobre ejemplos sanos, lo que permite detectar anomalías como desviaciones respecto a la normalidad aprendida.

Sin embargo, a diferencia de SimCLR o AutoEncoder, SWSSL no trabaja con la imagen completa, sino que aplica un enfoque local basado en ventanas deslizantes sobre el mapa de características generado por el encoder. Este método permite capturar relaciones contextuales entre regiones anatómicas próximas, lo cual es especialmente útil para detectar anomalías sutiles o localizadas.

La estructura del conjunto de datos se mantiene idéntica (ver apartado 5.2.1), pero la forma en que se procesan las imágenes cambia significativamente:

- Para cada imagen, se genera un conjunto de parches mediante un algoritmo de sliding window, definido por dos hiperparámetros clave:
    - Tamaño del parche (patch_size): controla el área cubierta por cada ventana.
    - Paso de desplazamiento (step_size): determina el grado de solapamiento entre parches consecutivos.
- Este esquema garantiza que se cubra toda la imagen con un número adecuado de regiones, facilitando el aprendizaje de representaciones locales.

- Cada parche es procesado por el encoder (ResNet18), y las salidas se utilizan tanto en la proyección contrastiva como en la reconstrucción (según el diseño de la función de pérdida).

Además, para mantener consistencia con el resto de modelos, se han utilizado las mismas herramientas de carga (ImageFolder, DataLoader) adaptadas mediante una clase personalizada (ChestDataset) que implementa la lógica de particionado por parches y generación de pares positivos y negativos necesarios para la optimización contrastiva.

5.8.3 Transformaciones aplicadas y carga mediante DataLoader

El proceso de preprocesamiento y carga de datos en SWSSL difiere de los enfoques anteriores (AutoEncoder y SimCLR) debido al carácter local del aprendizaje y al tratamiento de imágenes mediante parches. Aun así, se mantiene una estructura compatible con el resto del pipeline experimental, reutilizando herramientas como ImageFolder y DataLoader, adaptadas mediante una clase personalizada.

Para preparar adecuadamente los datos antes de su partición en ventanas, se aplica un conjunto reducido y controlado de transformaciones:

- **Grayscale():** convierte la imagen a un solo canal, lo cual es coherente con la naturaleza de las radiografías.
- **Resize(256) + CenterCrop(256):** asegura que todas las imágenes tengan dimensiones consistentes para que puedan dividirse en parches homogéneos.
- **ToTensor():** transforma la imagen PIL en tensor PyTorch, normalizado entre [0,1].
- **Normalize((0.5,), (0.5,)):** reescala los valores de píxel al rango [-1, 1], centrándolos en 0 para facilitar la convergencia.

A diferencia de SimCLR, no se utilizan augmentations aleatorias como rotaciones, flips o jitter de color, ya que SWSSL no busca aprender invariancias globales, sino consistencia local estructural, y en el contexto médico, alteraciones geométricas podrían comprometer la anatomía de los parches.

Estas transformaciones están implementadas en el repositorio del autor [15], dentro de la clase ChestDataset (ubicada en datasets/chest.py), que encapsula también el proceso de extracción de parches mediante sliding window.

```
[ ]  transform = transforms.Compose([
         transforms.Grayscale(),
         transforms.Resize(256),
         transforms.CenterCrop(256),
         transforms.ToTensor(),
         transforms.Normalize((0.5,), (0.5,))
     ])
```

Figura 55: Bloque de Código transformaciones SWSSL

5.8.3.1 Cropping-then-resizing

Una de las decisiones clave que diferencia a SWSSL de otros métodos contrastivos como SimCLR es la eliminación del enfoque tradicional de augmentación basado en "cropping-then-resizing". Este procedimiento, utilizado comúnmente en aprendizaje contrastivo, consiste en recortar aleatoriamente una región de la imagen original y luego redimensionarla a un tamaño fijo, como 224×224 píxeles.

Aunque esta técnica introduce variabilidad visual que puede resultar útil en contextos generales, en imágenes médicas tiene el riesgo de eliminar o distorsionar estructuras anatómicas sutiles y relevantes.

En lugar de ello, SWSSL opta por una extracción sistemática de parches mediante sliding window, preservando la resolución original de cada región de la imagen. De esta forma, se garantiza que:

- No se altera la escala natural de las estructuras clínicas.
- Se mantiene la continuidad espacial entre regiones vecinas.
- Se generan suficientes pares contrastivos sin depender de recortes aleatorios agresivos.

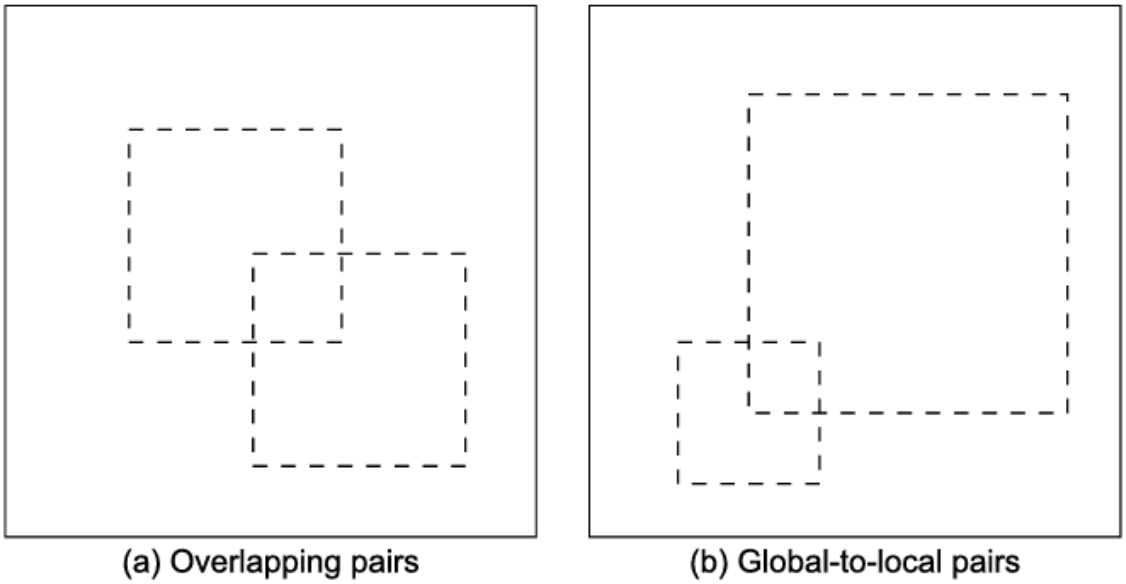

Figura 56: Dos tipos de pares de imágenes utilizados en aprendizaje auto-supervisado. Mientras que los pares global-local requieren operaciones de recorte y redimensionado (cropping then resizing), la pérdida de preservación de continuidad propuesta (Continuity Preserving Loss) permite generar pares solapados (overlapping pairs) sin distorsionar la estructura original de la imagen. Adaptado de [1].

5.8.3.2 Carga con DataLoader y lógica de ventanas

Una vez transformadas, las imágenes se dividen en subregiones mediante un esquema de ventana deslizante (sliding window) definido por dos hiperparámetros:

- patch_size = 128: tamaño de cada región (en píxeles).
- step_size = 32: salto entre parches consecutivos, permitiendo solapamiento.

Este proceso asegura que toda la imagen se cubre por múltiples ventanas parcialmente solapadas, lo que facilita la detección de relaciones de contexto entre regiones cercanas.

La clase ChestDataset se encarga de:

- Cortar cada imagen en parches con la configuración establecida.
- Empaquetar cada parche como una muestra individual.
- Asignar, para cada parche, un "par positivo" (dentro de la misma imagen) y múltiples negativos (del resto del batch).

La carga se realiza con DataLoader, que permite controlar el tamaño de lote (‘batch_size’) y barajar las imágenes (‘shuffle=True’) en cada época. Esto se ajusta dinámicamente según si el modo ‘patch=True’ está activo en la configuración del entrenamiento, como se observa en el archivo train.py del repositorio analizado.

```
[10] class PatchContrastiveDataset(Dataset):
        def __init__(self, root_dir, transform=None, patch_size=128, step_size=64):
            self.root_dir = root_dir
            self.transform = transform
            self.patch_size = patch_size
            self.step_size = step_size
            self.image_paths = [os.path.join(root_dir, f) for f in os.listdir(root_dir) if f.endswith(".jpeg")][:10]

        def extract_patches(self, image):
            _, h, w = image.shape
            patches = []
            for i in range(0, h - self.patch_size + 1, self.step_size):
                for j in range(0, w - self.patch_size + 1, self.step_size):
                    patch = image[:, i:i+self.patch_size, j:j+self.patch_size]
                    patches.append(patch)
            return patches

        def __getitem__(self, idx):
            img = Image.open(self.image_paths[idx]).convert('L')

            if self.transform:
                img = self.transform(img)  # transforma la imagen completa

            patches = self.extract_patches(img)  # ya son tensores

            patch_i = [p.clone() for p in patches]
            patch_j = [p.clone() for p in patches]

            return torch.stack(patch_i), torch.stack(patch_j)

        def __len__(self):
            return len(self.image_paths)
```

```
[11] dataset_path = "/content/chest_xray/chest_xray/train/NORMAL"
     dataset = PatchContrastiveDataset(root_dir=dataset_path, transform=transform)
     dataloader = DataLoader(dataset, batch_size=1, shuffle=True, num_workers=2)
```

Figura 57: Implementación de la clase ChestDataset, utilizada para cargar imágenes y dividirlas en parches mediante sliding window. También se muestra como se instancia el DataLoader

5.9 Implementación del modelo SWSSL

5.9.1 Introducción conceptual

Este apartado presenta la implementación técnica del modelo SWSSL utilizado en este trabajo. A partir de los principios introducidos anteriormente (ver apartado 3.1.6), se ha desarrollado una arquitectura funcional que permite entrenar el modelo en un entorno auto-supervisado sobre parches extraídos de imágenes médicas.

La implementación se ha llevado a cabo en Python utilizando PyTorch como framework principal, y siguiendo una lógica modular que facilita su reutilización y comparación con los modelos anteriores (AutoEncoder y SimCLR). En concreto, se reaprovecha la arquitectura ResNet18 como encoder común, manteniendo así condiciones arquitectónicas homogéneas entre los distintos enfoques.

El código del modelo se estructura en dos bloques: un extractor de características (encoder) y una cabeza de proyección (projection head) utilizada durante el entrenamiento contrastivo. Esta organización se ha programado directamente a partir del repositorio referenciado [15], asegurando una misma línea que la reflejada en la propuesta original [1].

5.9.2 Arquitectura del modelo

El modelo SWSSL implementado se divide en dos componentes principales:

- **Encoder convolucional**: se ha utilizado una ResNet18 truncada, eliminando su capa final de clasificación (fc). Esto permite conservar las características convolucionales profundas y obtener un vector latente de tamaño 512 (h) por cada parche de entrada. Esta misma arquitectura base se reutiliza en AutoEncoder y SimCLR, facilitando la comparación posterior entre modelos.
- **Cabeza de proyección (MLP)**: sobre el vector h extraído por el encoder, se aplica una red neuronal de dos capas (Linear → ReLU → Linear) que proyecta la salida en un espacio latente de 128 dimensiones (z). Esta cabeza de proyección es usada exclusivamente durante el entrenamiento contrastivo.

Este diseño modular está encapsulado en la clase SWSSLModel, definida y reutilizada directamente desde el repositorio.

```python
class SWSSLModel(nn.Module):
    def __init__(self, projection_dim=128):
        super(SWSSLModel, self).__init__()
        resnet = models.resnet18(weights=None)
        resnet.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.encoder = nn.Sequential(*list(resnet.children())[:-1])
        self.projection = nn.Sequential(
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, projection_dim)
        )

    def forward(self, x):
        h = self.encoder(x).squeeze()
        z = self.projection(h)
        return h, z
```

Figura 58: Definición del modelo SWSSLModel, incluyendo tanto el encoder como la cabeza de proyección

5.9.3 Función de pérdida contrastiva

El entrenamiento del modelo SWSSL se basa en una función de pérdida contrastiva denominada 'twin_loss', diseñada para comparar la similitud entre representaciones obtenidas a partir de parches extraídos de una misma imagen o de regiones similares.

Esta función tiene como objetivo que las representaciones proyectadas de parches considerados positivos (similares) estén lo más próximas posible en el espacio latente, mientras que se mantenga una mayor distancia respecto al resto de elementos del lote.

La implementación de 'twin_loss' emplea una estrategia simplificada basada en NT-Xent (ver apartado 5.8.1), similar a la utilizada en SimCLR, aunque aplicada sobre los parches individuales extraídos de una imagen.

```python
def twin_loss(z1, z2, temperature=0.5):
    z1 = F.normalize(z1, dim=1)
    z2 = F.normalize(z2, dim=1)
    logits = torch.matmul(z1, z2.T) / temperature
    labels = torch.arange(z1.size(0)).to(z1.device)
    return F.cross_entropy(logits, labels)
```

Figura 59: Definición de 'twin loss' (SWSSL)

Explicación paso a paso:

- **Normalización** ('F.normalize')

  Los vectores proyectados z1 y z2 se normalizan para que todos tengan norma unitaria. Esto convierte el producto escalar en una medida directa de **similaridad coseno** entre vectores.

- **Producto punto cruzado**

  Se calcula la matriz logits de similitudes entre todos los vectores de z1 frente a todos los de z2. Esta matriz tiene tamaño [N, N], donde N es el número de parches.

- **Etiquetas positivas**

  Se asume que z1[i] y z2[i] son vistas del mismo parche (par positivo), por lo que se construye un vector de etiquetas con los índices [0, 1, ..., N-1].

- **Pérdida final**

  Se aplica entropía cruzada para penalizar que z1[i] no esté más cerca de z2[i] que del resto. Cuanto más baja sea la pérdida, mejor será la agrupación de pares positivos en el espacio latente.

Esta función es sencilla pero efectiva, especialmente en un entorno donde se trabaja con parches locales sin anotaciones.

5.9.4 Entrenamiento del modelo SWSSL

El entrenamiento del modelo se realiza de forma auto-supervisada a partir de las imágenes normales del conjunto de datos Chest X-ray. Para cada imagen del lote, se extraen múltiples parches mediante un esquema de ventana deslizante (ver 5.9.2), y cada uno de estos parches es tratado como una muestra individual.

El objetivo es que el modelo aprenda a generar representaciones consistentes entre parches similares (por ejemplo, zonas del mismo pulmón en distintas imágenes normales). Para ello, se calcula la pérdida contrastiva entre los vectores proyectados obtenidos tras aplicar el encoder y la MLP de proyección a cada parche.

Durante el entrenamiento, se siguen los siguientes pasos:

- **Cargar los datos** con batch_size = 1, lo que implica que en cada lote se procesa una sola imagen base. De ella se extraen todos los parches disponibles, que se convierten en un mini-batch de entrada.

- **Transformar cada parche** con la arquitectura SWSSL, generando los vectores latentes h y sus proyecciones z.

- **Aplicar la función twin_loss(z, z)**, comparando los vectores proyectados entre sí.

- **Realizar la retropropagación** y actualizar los pesos del modelo.

- **Guardar la pérdida media** al final de cada época para monitorizar la convergencia.

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SWSSLModel().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

epochs = 10
losses = []

model.train()
for epoch in range(epochs):
    total_loss = 0
    print(f"\n Época {epoch+1}/{epochs}")

    for (patch_i, patch_j) in dataloader:
        patch_i = patch_i.squeeze(0).to(device)
        patch_j = patch_j.squeeze(0).to(device)

        _, z_i = model(patch_i)
        _, z_j = model(patch_j)

        # twin_loss
        loss = twin_loss(z_i, z_j)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    epoch_loss = total_loss / len(dataloader)
    losses.append(epoch_loss)
    print(" Pérdida promedio: {epoch_loss:.4f}")
```

Figura 60: Entrenamiento del modelo SWSSL

Antes de realizar experimentos con fines comparativos, el modelo SWSSL se ha entrenado inicialmente utilizando una configuración base de hiperparámetros. Esta configuración actúa como referencia para analizar el comportamiento inicial del modelo y sirve como punto de partida para evaluar el impacto de posibles ajustes. Posteriormente, se prueban distintas variables.

La evolución de la curva de pérdida del entrenamiento con la configuración base es el siguiente:

```
Época 1/10
Pérdida promedio: 1.2659

Época 2/10
Pérdida promedio: 0.9964

Época 3/10
Pérdida promedio: 0.9408

Época 4/10
Pérdida promedio: 0.8826

Época 5/10
Pérdida promedio: 0.8113

Época 6/10
Pérdida promedio: 0.7354

Época 7/10
Pérdida promedio: 0.7071

Época 8/10
Pérdida promedio: 0.6981

Época 9/10
Pérdida promedio: 0.6645

Época 10/10
Pérdida promedio: 0.6619
```

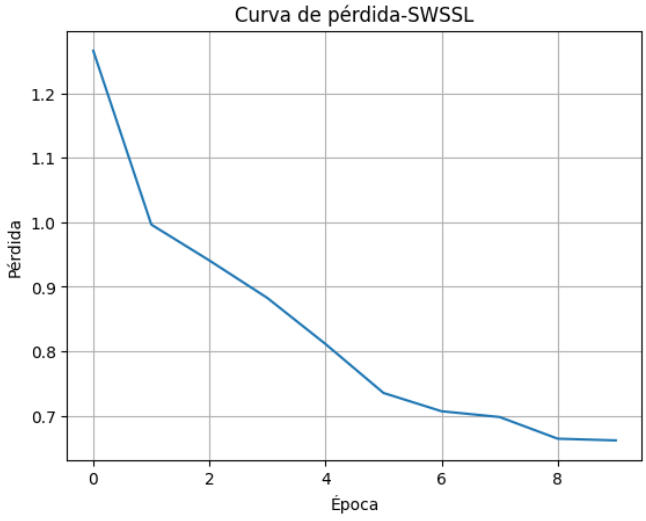

Figura 61: Evolución pérdida épocas configuración base (SWSSL) + Curva de pérdida - SWSSL

5.9.5 Comparación de configuraciones experimentales (SWSSL)

Se ha llevado una serie de experimentos modificando de forma controlada distintos hiperparámetros del modelo SWSSL. El objetivo es observar el impacto que cada modificación tiene en el rendimiento, tanto a nivel de convergencia como de calidad de representación.

Cada configuración, como en pruebas con modelos anteriores, mantiene constante la estructura general del modelo y el conjunto de datos. Al alterar únicamente un aspecto por experimento, se permite identificar de manera más precisa qué elementos son clave en la optimización del modelo.

Las configuraciones experimentales probadas son las siguientes:

Configuración	Descripción	Justificación
Base	Adam, lr = 1e-3, 10 épocas, patch_size = 128, step = 32	Punto de partida común
Parches más pequeños	Patch_size = 96	Evaluar pérdida de contexto
Parches más grandes	Patch_size = 160	Ver si más contexto mejora discriminación
Optimización alternativa	SGD con momentum = 0.9	Comparar con optimizador Adam
Sin normalización L2 (optimizador Adam y SGD)	Se elimina 'F.normalize(z,dim=1)	Evaluar efecto de magnitud no controlada
Temperatura modificada	t = 0.1 y t = 1.0	Analizar sensibilidad al contraste

Tabla 5: Resumen de comparación de configuraciones experimentales (SWSSL)

Para los cambios de parches hay que cambiar los parámetros de este fragmento del código:

```
class PatchContrastiveDataset(Dataset):
    def __init__(self, root_dir, transform=None, patch_size=96, step_size=64):
        self.root_dir = root_dir
        self.transform = transform
```

Figura 62: Bloque de Código PatchContrastiveDataset (definición de hiperparámetros – SWSSL)

5.9.5.1 Parches más pequeños (Patch_size = 96)

```
Época 1/10
Pérdida promedio: 1.1930

Época 2/10
Pérdida promedio: 0.8514

Época 3/10
Pérdida promedio: 0.7772

Época 4/10
Pérdida promedio: 0.7165

Época 5/10
Pérdida promedio: 0.7016

Época 6/10
Pérdida promedio: 0.6681

Época 7/10
Pérdida promedio: 0.6464

Época 8/10
Pérdida promedio: 0.6357

Época 9/10
Pérdida promedio: 0.6312

Época 10/10
Pérdida promedio: 0.6242
```

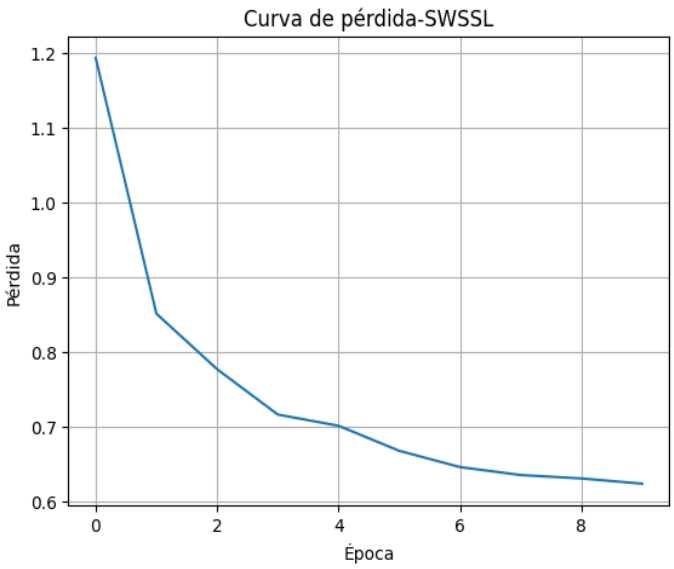

Figura 63: Evolución pérdida épocas parches más pequeños (SWSSL) + Curva de pérdida (parches más pequeños – SWSSL)

5.9.5.2 Parches más grandes (Patch_size = 160)

```
Época 1/10
Pérdida promedio: 1.1903

Época 2/10
Pérdida promedio: 0.9300

Época 3/10
Pérdida promedio: 0.8072

Época 4/10
Pérdida promedio: 0.7513

Época 5/10
Pérdida promedio: 0.7029

Época 6/10
Pérdida promedio: 0.6868

Época 7/10
Pérdida promedio: 0.6591

Época 8/10
Pérdida promedio: 0.6407

Época 9/10
Pérdida promedio: 0.6287

Época 10/10
Pérdida promedio: 0.6202
```

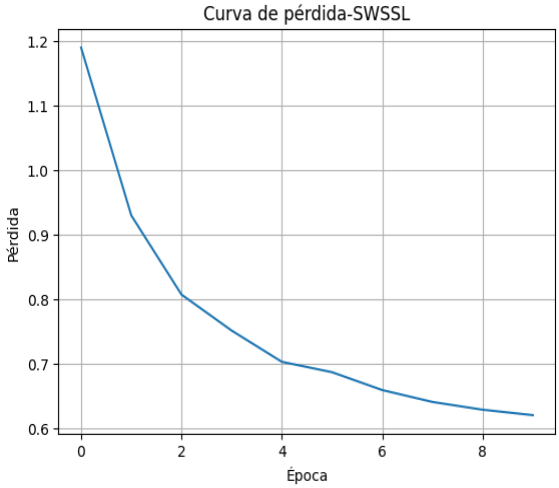

Figura 64: Evolución pérdida épocas parches más grandes (SWSSL) + Curva de pérdida (parches más grandes – SWSSL)

5.9.5.3 Optimización alternativa SGD con momentum = 0.9

Se ha cambiado este fragmento del código :

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Figura 65: Bloque código de optimizador Adam (SWSSL)

Por el siguiente:

```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3, momentum=0.9)
```

Figura 66: Bloque de Código de optimizador SGD con momentum = 0.9 (SWSSL)

```
Época 1/10
Pérdida promedio: 2.0479

Época 2/10
Pérdida promedio: 1.4856

Época 3/10
Pérdida promedio: 0.9435

Época 4/10
Pérdida promedio: 0.7699

Época 5/10
Pérdida promedio: 0.6995

Época 6/10
Pérdida promedio: 0.6683

Época 7/10
Pérdida promedio: 0.6511

Época 8/10
Pérdida promedio: 0.6423

Época 9/10
Pérdida promedio: 0.6349

Época 10/10
Pérdida promedio: 0.6286
```

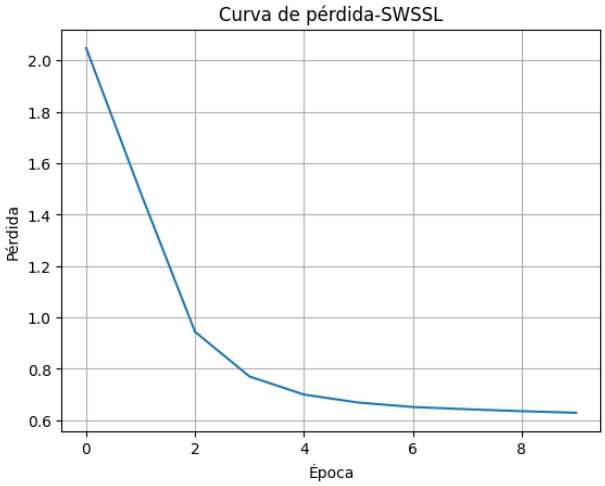

Figura 67: Evolución pérdida épocas optimizador SGD con momentum = 0.9 (SWSSL) + Curva de pérdida (SGD – SWSSL)

5.9.5.4 Sin normalización L2

Se ha modificado 'twin loss':

```python
def twin_loss(z1, z2, temperature=0.5):
  # Eliminamos las siguientes 2 líneas
    z1 = F.normalize(z1, dim=1)
    z2 = F.normalize(z2, dim=1)
    logits = torch.matmul(z1, z2.T) / temperature
    labels = torch.arange(z1.size(0)).to(z1.device)
    return F.cross_entropy(logits, labels)
```

Figura 68: Bloque de código 'twin_loss' (SWSSL)

Por defecto el método normalize realiza normalización L2 a lo largo de la dimensión especificada.

El método se define así : ' torch.nn.functional.normalize(input, p=2.0, dim=1, eps=1e-12, out=None)'

Se cambia por lo siguiente:

```python
def twin_loss(z1, z2, temperature=0.5):

    logits = torch.matmul(z1, z2.T) / temperature
    labels = torch.arange(z1.size(0)).to(z1.device)
    return F.cross_entropy(logits, labels)
```

Figura 69: Bloque de Código 'twin_loss' (sin normalización L2-SWSSL)

Resultado con optimizador SGD y momentum = 0.9

```
Época 1/10
Pérdida promedio: 1.5909

Época 2/10
Pérdida promedio: 1.0720

Época 3/10
Pérdida promedio: 0.6452

Época 4/10
Pérdida promedio: 0.3662

Época 5/10
Pérdida promedio: 0.1590

Época 6/10
Pérdida promedio: 0.0514

Época 7/10
Pérdida promedio: 0.0136

Época 8/10
Pérdida promedio: 0.0049

Época 9/10
Pérdida promedio: 0.0028

Época 10/10
Pérdida promedio: 0.0019
```

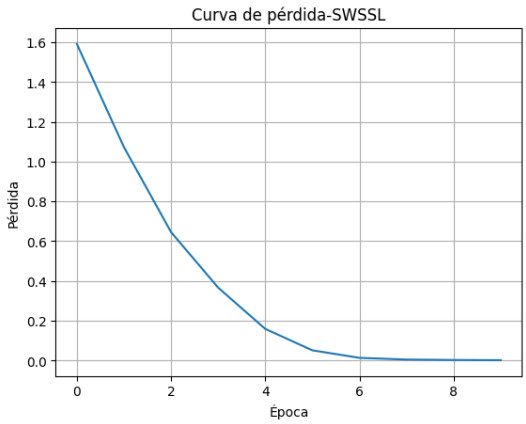

Figura 70: Evolución pérdida épocas optimizador SGD con momentum = 0.9 (sin normalización L2-SWSSL) + Curva de pérdida (SGD – SWSSL)

Resultado con optimizador Adam :

```
Época 1/10
Pérdida promedio: 2.5060

Época 2/10
Pérdida promedio: 0.9519

Época 3/10
Pérdida promedio: 0.1112

Época 4/10
Pérdida promedio: 0.0352

Época 5/10
Pérdida promedio: 0.1187

Época 6/10
Pérdida promedio: 0.5063

Época 7/10
Pérdida promedio: 0.4273

Época 8/10
Pérdida promedio: 0.1572

Época 9/10
Pérdida promedio: 0.0622

Época 10/10
Pérdida promedio: 0.0587
```

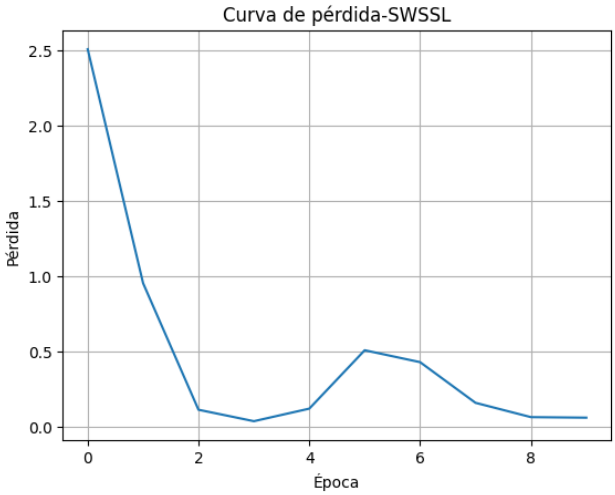

Figura 71: Evolución pérdida épocas optimizador Adam (sin normalización L2-SWSSL) + Curva de pérdida (Adam – SWSSL)

5.9.5.5 Entrenamiento con 50 épocas

El carácter experimental de este trabajo provoca que se vaya trabajando en base a los resultados que se van obteniendo. Se ha visto en el análisis experimental comparando distintos ajustes de hiperparámetros, que sin normalización el entrenamiento es bastante más óptimo. Además, se observa que el optimizador SGD con momentum = 0.9 es significativamente más eficiente con este modelo que el optimizador Adam.

Para probar el entrenamiento con 50 épocas inicialmente se había enfocado en realizarlo con la configuración base (Adam), esta se va a cambiar por el optimizador SGD sin normalización L2.

```
epochs = 50
losses = []
```

Figura 72: Definición épocas (SWSSL)

```
Época 1/50
Pérdida promedio: 1.6097
Época 2/50
Pérdida promedio: 1.1015
Época 3/50
Pérdida promedio: 0.6591
Época 4/50
Pérdida promedio: 0.2799
Época 5/50
Pérdida promedio: 0.1129
Época 6/50
Pérdida promedio: 0.0444
Época 7/50
Pérdida promedio: 0.0214
Época 8/50
Pérdida promedio: 0.0073
Época 9/50
Pérdida promedio: 0.0033
Época 10/50
Pérdida promedio: 0.0020
Época 11/50
Pérdida promedio: 0.0015
Época 12/50
Pérdida promedio: 0.0012
Época 13/50
Pérdida promedio: 0.0010
Época 14/50
Pérdida promedio: 0.0009
Época 15/50
Pérdida promedio: 0.0008
Época 16/50
Pérdida promedio: 0.0008
Época 17/50
Pérdida promedio: 0.0007
Época 18/50
Pérdida promedio: 0.0007
Época 19/50
Pérdida promedio: 0.0006
Época 20/50
Pérdida promedio: 0.0006
Época 21/50
Pérdida promedio: 0.0006
Época 22/50
Pérdida promedio: 0.0005
Época 23/50
Pérdida promedio: 0.0005
Época 24/50
Pérdida promedio: 0.0005
Época 25/50
Pérdida promedio: 0.0005
Época 26/50
Pérdida promedio: 0.0004
Época 27/50
Pérdida promedio: 0.0004
Época 28/50
Pérdida promedio: 0.0004
Época 29/50
Pérdida promedio: 0.0004
Época 30/50
Pérdida promedio: 0.0004
Época 31/50
Pérdida promedio: 0.0004
Época 32/50
Pérdida promedio: 0.0004
Época 33/50
Pérdida promedio: 0.0003
Época 34/50
Pérdida promedio: 0.0003
Época 35/50
Pérdida promedio: 0.0003
Época 36/50
Pérdida promedio: 0.0003
Época 37/50
Pérdida promedio: 0.0003
Época 38/50
Pérdida promedio: 0.0003
Época 39/50
Pérdida promedio: 0.0003
Época 40/50
Pérdida promedio: 0.0003
Época 41/50
Pérdida promedio: 0.0003
Época 42/50
Pérdida promedio: 0.0003
Época 43/50
Pérdida promedio: 0.0003
Época 44/50
Pérdida promedio: 0.0003
Época 45/50
Pérdida promedio: 0.0003
Época 46/50
Pérdida promedio: 0.0002
Época 47/50
Pérdida promedio: 0.0002
Época 48/50
Pérdida promedio: 0.0002
Época 49/50
Pérdida promedio: 0.0002
Época 50/50
Pérdida promedio: 0.0002
```

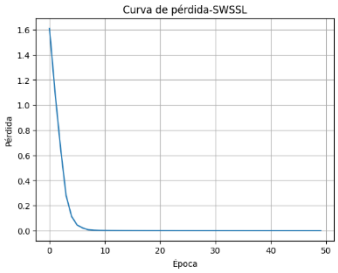

Figura 73: Evolución pérdida épocas optimizador SGD con momentum = 0.9 (50 épocas-SWSSL) + Curva de pérdida (SGD – SWSSL)

5.9.6 Conclusión de configuraciones experimentales (SWSSL)

Durante la fase experimental se realizaron múltiples entrenamientos del modelo SWSSL bajo diferentes configuraciones de hiperparámetros. Uno de los factores más relevantes analizados fue el número de épocas de entrenamiento y la ausencia de normalización L2.

Tal como muestra la curva de pérdida obtenida (Figura 73), la mayor parte del descenso significativo de la función de pérdida ocurre en las primeras 10 a 15 épocas, estabilizándose a partir de ese punto. Si bien los entrenamientos realizados a 50 épocas aseguran una convergencia completa, no se observa una mejora notable en la pérdida a partir de cierto umbral, lo que sugiere un uso de número de épocas razonable, así como 20 o 30.

Esta observación refuerza la idea de que entrenar durante más tiempo no siempre implica un mejor rendimiento, y puede incluso incrementar el riesgo de sobreajuste (overfitting), especialmente si no se evalúan adecuadamente las métricas sobre datos no vistos.

Como conclusión, se destaca la importancia de monitorizar la evolución de la pérdida durante el entrenamiento y considerar técnicas como early stopping o validación cruzada para seleccionar el número óptimo de épocas, evitando entrenamientos innecesariamente largos o poco eficientes, como se ha comentado con modelos previos (AutoEncoder y SimCLR). Se utilizará para la parte de testing SWSSL con un número de épocas en torno a 20-30, sin normalización L2 y con optimizador SGD con momentum = 0.9.

5.10 Función de pérdida según el modelo

Antes de presentar adentrarse a la parte de evaluación una vez finalizada la etapa de entrenamiento, es importante aclarar qué mide exactamente la función de pérdida en cada uno de los modelos evaluados. Esto permite interpretar correctamente las gráficas de entrenamiento y entender por qué los valores de pérdida no pueden compararse directamente entre AutoEncoder, SimCLR y SWSSL. Cada modelo se basa en un enfoque auto-supervisado distinto y, por tanto, emplea una función de pérdida adaptada a su propósito.

- **AutoEncoder**: Utiliza la pérdida de error cuadrático medio (MSELoss), que mide la diferencia entre la imagen original y su reconstrucción. Esta pérdida busca minimizar el error por píxel, lo que implica que el modelo debe aprender a codificar fielmente las estructuras visuales normales.
- **SWSSL**: Se basa en una variante contrastiva de la NT-Xent Loss, aplicada entre vistas aumentadas de parches. Esta función fuerza al modelo a generar representaciones similares para distintas versiones del mismo parche, y disímiles para otras regiones. Se complementa con una pérdida de coherencia espacial que garantiza continuidad entre parches adyacentes.
- **SimCLR**: Emplea la NT-Xent Loss estándar, donde se maximiza la similitud entre vistas aumentadas de una imagen completa y se penaliza la similitud con otras imágenes del batch. Esta pérdida permite al modelo capturar estructuras invariantes a transformaciones, sin necesidad de etiquetas ni reconstrucción.

Dado que estas pérdidas cumplen funciones distintas (reconstrucción frente a aprendizaje contrastivo), no se pueden comparar directamente como indicadores de rendimiento entre

modelos. Su utilidad reside en monitorizar la convergencia durante el entrenamiento. La comparación entre modelos se realiza exclusivamente mediante métricas de test, como AUC, F1-score, precisión o sensibilidad.

6. Resultados y Análisis

Este capítulo presenta los resultados obtenidos tras aplicar los modelos AutoEncoder, SimCLR y SWSSL al conjunto de test, una vez seleccionadas las configuraciones óptimas para cada uno de ellos. El objetivo principal es evaluar y comparar su rendimiento en la tarea de detección de anomalías en imágenes médicas de alta resolución, utilizando un conjunto común de métricas clínicas.

La evaluación se realiza de forma cuantitativa, mediante métricas como AUC, F1-score, precisión, sensibilidad y especificidad, y se complementa con visualizaciones que permiten interpretar el comportamiento de los modelos. Se emplea una metodología uniforme para asegurar la comparabilidad entre enfoques, y se analizan tanto los aciertos como las limitaciones de cada uno.

Los resultados se organizan en función del modelo utilizado y se acompañan de un análisis crítico que permite valorar la utilidad práctica de cada enfoque en el contexto clínico. Finalmente, se recogen las conclusiones generales derivadas de la comparación entre métodos.

6.1 Evaluación del modelo AutoEncoder

6.1.1 Metodología de evaluación

Una vez entrenado el AutoEncoder con la configuración óptima seleccionada en la fase experimental, se procede a su evaluación sobre el conjunto de test, el cual incluye imágenes normales y con anomalías (neumonía). El proceso de evaluación se basa en el cálculo del error de reconstrucción, que mide la diferencia entre la imagen original y la reconstruida por el modelo.

Para cada imagen, se calcula la pérdida MSE (Error Cuadrático Medio), obteniendo un valor escalar que se interpreta como una puntuación de anomalía. Cuanto mayor es el error, más probable es que la imagen contenga anomalías, ya que el modelo fue entrenado exclusivamente con imágenes normales y, por tanto, tiende a reconstruir peor las imágenes patológicas.

Con estos valores se construye un vector de puntuaciones sobre el que se aplican dos estrategias:

- **Evaluación directa con métricas continuas**, como el área bajo la curva ROC (ROC-AUC).
- **Evaluación binaria**, seleccionando un umbral sobre el error para clasificar cada imagen como normal o anómala, y calcular así métricas como F1-score, precisión, sensibilidad, especificidad y accuracy.

Por otro lado, respecto a las métricas utilizadas para cuantificar distintos aspectos del testing del modelo, a continuación, se explica adaptado a AutoEncoder que se espera medir con cada métrica.

- **ROC-AUC (Área bajo la curva ROC)**: mide la capacidad del modelo para asignar scores más altos a imágenes anómalas que a normales. Un valor de 1 indica una separación perfecta, mientras que 0.5 equivale a una predicción aleatoria. Es una métrica robusta frente a desequilibrios de clase.
- **F1-score**: es la media armónica entre la precisión y el recall, y proporciona una visión global del rendimiento del modelo en tareas de clasificación binaria, especialmente útil cuando hay desequilibrio entre clases. Se maximiza cuando ambos valores son altos.
- **Precisión (Precision)**: indica qué proporción de las imágenes clasificadas como anómalas por el modelo realmente lo eran. Evalúa la exactitud de las predicciones positivas.
- **Recall (Sensibilidad)**: mide la capacidad del modelo para detectar verdaderos casos de anomalía. Es decir, qué porcentaje de imágenes anómalas ha sido correctamente identificado.
- **Accuracy**: representa el porcentaje total de predicciones correctas (tanto normales como anómalas) sobre el total de muestras. Aunque es intuitiva, puede ser engañosa en conjuntos de datos desequilibrados.

En el contexto de detección de anomalías mediante AutoEncoders no supervisados, las métricas más representativas son ROC-AUC y recall. El ROC-AUC permite evaluar la capacidad del modelo para asignar scores más altos a imágenes anómalas, independientemente del umbral de clasificación, lo que lo convierte en un indicador robusto incluso en presencia de desequilibrios entre clases. Por su parte, el recall cuantifica la proporción de anomalías correctamente identificadas, lo cual es especialmente crítico en entornos clínicos donde pasar

por alto un caso patológico puede tener consecuencias graves. En cambio, métricas como la accuracy o la precisión pueden resultar engañosas si el conjunto de datos está desbalanceado o si el modelo tiende a ser conservador en sus predicciones. Por ello, el análisis debe centrarse en si el modelo diferencia efectivamente los casos anómalos en el espacio de reconstrucción, más allá del número absoluto de aciertos en la clasificación binaria.

6.1.2 Resultados y mejoras progresivas del modelo AutoEncoder

En una primera evaluación del modelo AutoEncoder entrenado únicamente con imágenes normales y usando pérdida MSE, se observó un rendimiento muy limitado sobre el conjunto de test. Las métricas obtenidas fueron las siguientes:

AUC: 0.2992

F1-score: 0.0711

Precisión: 0.4688

Recall: 0.0385

Accuracy: 0.3718

Estas cifras indicaban que, a pesar de que el modelo reconstruía correctamente las imágenes, no lograba diferenciar de forma clara entre imágenes normales y anómalas. Este comportamiento es característico de un modelo que ha adquirido demasiada capacidad de reconstrucción, incluso sobre entradas que no forman parte de la distribución original de entrenamiento (p. ej., imágenes con neumonía).

Con el objetivo de mejorar esta sensibilidad, se introdujeron varias modificaciones experimentales sobre el modelo original:

Sustitución de la pérdida MSE por L1Loss, más robusta ante valores atípicos y más sensible a errores locales.

Inclusión de ruido gaussiano en las entradas durante el entrenamiento, lo que transforma el modelo en un Denoising AutoEncoder. Esta técnica obliga al modelo a aprender las características esenciales en lugar de memorizar píxel a píxel.

Incorporación de Dropout en las capas del encoder para limitar la capacidad del modelo y prevenir el sobreajuste.

```
# Ruido (Denoising AE)
noisy_imgs = imgs + 0.1 * torch.randn_like(imgs)
noisy_imgs = torch.clamp(noisy_imgs, -1, 1)  # Evita valores fuera de rango

outputs = model(noisy_imgs)
loss = criterion(outputs, imgs)  # Compara contra original
```

Figura 74: Bloque de Código noisy_imgs (evaluación AutoEncoder)

```
criterion = nn.L1Loss()
self.encoder = nn.Sequential(
    nn.Conv2d(1, 32, 3, stride=2, padding=1),  # 128-> 64
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Dropout2d(0.1),
    nn.Conv2d(32, 64, 3, stride=2, padding=1), # 64 -> 32
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Dropout2d(0.2),
    nn.Conv2d(64, 128, 3, stride=2, padding=1),# 32 -> 16
    nn.ReLU()
```

Figura 75: Bloque de Código Encoder (evaluación AutoEncoder)

6.1.2.1 Proceso de evaluación

La evaluación se basa en la diferencia entre la imagen original y la reconstrucción, medida mediante la pérdida L1 en este caso para cada muestra del conjunto de test. A cada imagen se le asigna un score de anomalía: cuanto mayor es el error, mayor probabilidad de que la imagen sea anómala. Este vector de puntuaciones continuas se somete a dos análisis:

- **Evaluación continua:** se calcula el área balo la curva ROC (ROC – AUC), evaluando si las imágenes anómalas tienden a tener scores más altos que las normales
- **Evaluación binaria:** se aplica un umbral experimental sobre los scores (se ha probado con 80 y 95) para dividir las predicciones. Se calculan métricas como F1-score, precisión, recall, especificidad y accuracy.

6.1.2.2 Métricas tras las mejoras

Tras reentrenar el modelo con las modificaciones anteriores durante 10 épocas, se observaron mejoras claras en la capacidad de detección:

AUC: 0.2992

F1-score: 0.0711

Precisión: 0.4688

Recall: 0.0385

Accuracy: 0.3718

Si bien el AUC se mantiene bajo, la mejora en F1-score y recall muestra que el modelo empieza a detectar de forma parcial las anomalías, clasificando correctamente más casos patológicos. Este resultado sugiere que las restricciones introducidas ayudan al modelo a reducir su capacidad de reconstrucción sobre patrones desconocidos.

Este bloque forma parte de un proceso iterativo y progresivo de ajuste y evaluación, en el que se irán probando nuevas técnicas de mejora y visualización para optimizar el rendimiento del AutoEncoder. En los próximos apartados se documentarán dichas pruebas, incluyendo técnicas como cambios estructurales en la arquitectura y la exploración de nuevas funciones de pérdida.

Se reentrena el modelo:

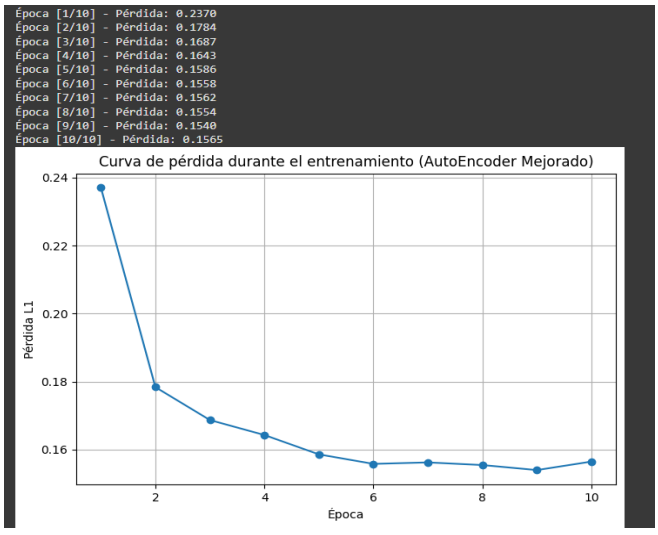

Figura 76: Evolución pérdida épocas + Curva de pérdida durante entrenamiento (AutoEncoder mejorado)

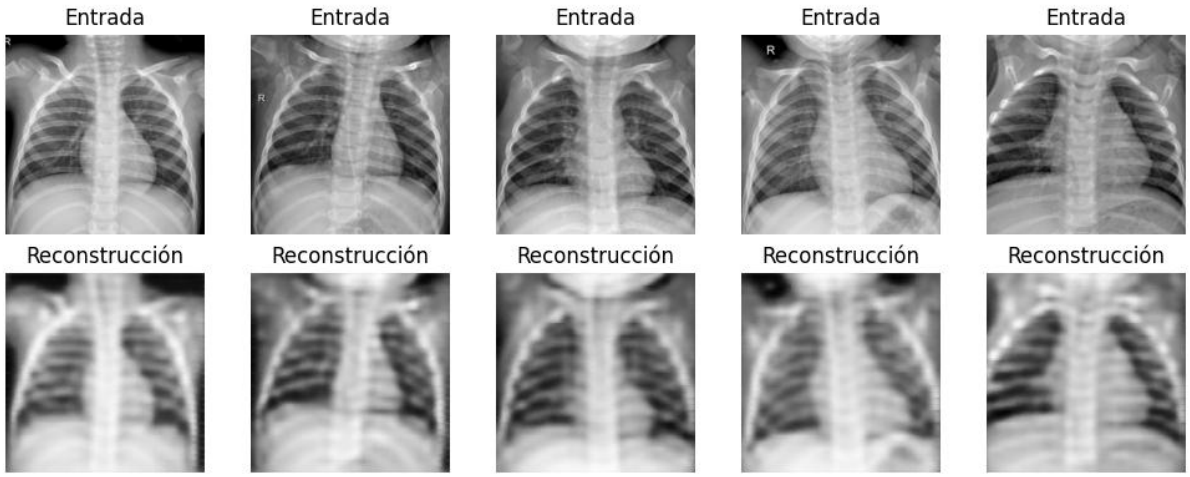

Figura 77: Reconstrucción imágenes chestXRay reentrenamiento AutoEncoder (evaluación)

Posteriormente, se introdujeron nuevas mejoras estructurales y de entrenamiento:

- Compresión más agresiva del espacio latente, añadiendo una capa adicional en el encoder que reduce la resolución intermedia a 8×8. Esta modificación fuerza al modelo a codificar únicamente las características más esenciales.
- Sustitución de la función de activación final del decoder, pasando de Sigmoid a Tanh, para adecuarse mejor a los valores de entrada normalizados en el rango [-1, 1].
- Implementación de una pérdida compuesta, sumando la pérdida L1 estándar con una pérdida de bordes **(**edge loss**)** basada en el operador de Sobel. Esta combinación penaliza no solo errores de intensidad global, sino también fallos en la reconstrucción de estructuras.
- Mantenimiento del esquema de Denoising AutoEncoder, añadiendo ruido gaussiano a las entradas durante el entrenamiento para fomentar la generalización.

Estas estrategias estaban orientadas a limitar la capacidad reconstructiva del modelo y a aumentar su fallo ante patrones que se desvían de la normalidad aprendida. Tras 10 épocas de entrenamiento con estas configuraciones, se obtuvieron los siguientes resultados:

```
Resultados:
AUC: 0.3878
F1-score: 0.1137
Precisión: 0.7500
Recall: 0.0615
Accuracy: 0.4006
```

Tabla 6: Métricas evaluación AutoEncoder

```
nn.ReLU(),
nn.ConvTranspose2d(32, 1, 3, stride=2, padding=1, output_padding=1),    # 64 -> 128
nn.Tanh()
```

```
nn.Conv2d(128, 256, 3, stride=2, padding=1),  # 16 → 8
nn.ReLU()
```

```
# Se define una pérdida diferente edge loss basada en operador sobel
def edge_loss(img1, img2):
    sobel_x = torch.tensor([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=torch.float32, device=img1.device).view(1, 1, 3, 3)
    grad1 = F.conv2d(img1, sobel_x, padding=1)
    grad2 = F.conv2d(img2, sobel_x, padding=1)
    return F.l1_loss(grad1, grad2)
```

```
# Pérdida combinada: L1 + Edge Loss
loss_l1 = criterion(outputs, imgs)
loss_edges = edge_loss(outputs, imgs)
loss = 0.8 * loss_l1 + 0.2 * loss_edges
```

Figura 78: Bloque de Código modificaciones (evaluación AutoEncoder

Comparado con las versiones anteriores, se aprecia una mejora clara en el área bajo la curva ROC y en el F1-score, además de un incremento notable en la precisión. Esto indica que, si bien el modelo aún no detecta la mayoría de los casos anómalos (recall bajo), los que detecta lo hace con gran fiabilidad. Además, el análisis visual de las reconstrucciones confirma que el modelo reconstruye con mayor dificultad ciertas estructuras pulmonares atípicas, lo que respalda la utilidad del enfoque en entornos de pre-filtrado clínico o como apoyo al diagnóstico.

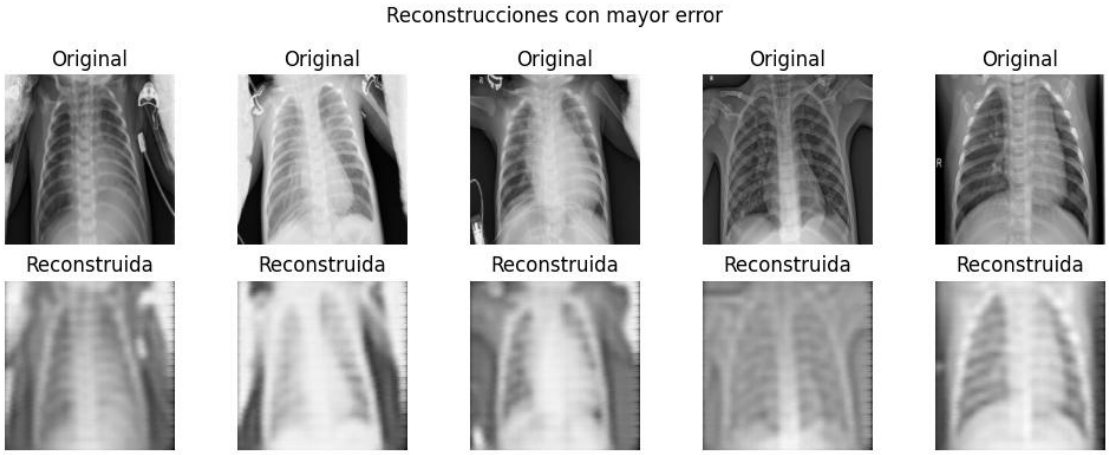

Figura 79: Reconstrucciones correspondientes a las imágenes con mayor error de reconstrucción en el conjunto de test. Se observa pérdida de nitidez y distorsión estructural, lo que sugiere que el modelo penaliza patrones alejados del perfil de normalidad aprendida

6.1.3 Conclusiones del modelo AutoEncoder

A lo largo del desarrollo experimental, el modelo AutoEncoder ha sido sometido a distintas modificaciones estructurales y funcionales con el objetivo de aumentar su sensibilidad a patrones anómalos. Desde una arquitectura inicial con pérdida MSE y reconstrucciones fieles incluso para imágenes patológicas, se ha evolucionado hacia un modelo más restrictivo, basado en:

- Reducción del espacio latente para forzar compresión.
- Uso combinado de pérdida L1 + edge loss para reforzar la penalización de errores estructurales.
- Aplicación de ruido gaussiano a las entradas (Denoising AutoEncoder).
- Cambio en la función de activación final para adaptarse a la normalización de los datos.

Estas transformaciones han permitido observar una mejora progresiva en métricas como el AUC o el F1-score, y especialmente en la precisión, lo cual sugiere que el modelo ha empezado a distinguir ciertos casos de neumonía basándose en su incapacidad para reconstruir correctamente las zonas afectadas. No obstante, el valor de recall sigue siendo bajo, lo que implica que una parte importante de las anomalías no son detectadas por el sistema.

Desde una perspectiva clínica o aplicada, el modelo aún no es lo suficientemente fiable como para ser usado como clasificador autónomo. Sin embargo, puede considerarse útil como herramienta complementaria en sistemas de pre-filtrado, permitiendo resaltar ciertos casos para revisión prioritaria por parte del personal médico.

6.2 Evaluación del modelo SimCLR

6.2.1 Metodología de evaluación

Después de entrenar el modelo SimCLR durante 20 épocas con la pérdida contrastiva NT-Xent (temperatura 0.5), se preparó una fase de evaluación para comprobar si las representaciones aprendidas (los vectores latentes hhh) permiten detectar anomalías en el conjunto de test, sin necesidad de etiquetas durante el entrenamiento.

El modelo está compuesto por un encoder basado en ResNet18 y una proyección adicional (projection head). Durante el entrenamiento, se trabajó solo con imágenes normales y se generaron pares de vistas aumentadas usando transformaciones como recorte, desenfoque, inversión y escala, pensadas para datos médicos.

Para poder evaluar correctamente al modelo, se hicieron varios ajustes respecto a la versión original:

- Se quitó la limitación de procesar solo 25 batches por época, permitiendo entrenar con todos los datos disponibles.
- Se sustituyó la Twin Loss por la NT-Xent Loss, que es la que se usa en el diseño original de SimCLR.
- Se cambió el forward() del modelo para que devuelva tanto h como z, y así poder usar solo h (los embeddings latentes) durante la evaluación.

```
class SimCLR(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.projection = ProjectionHead()

    def forward(self, x):
        h = self.encoder(x)
        z = self.projection(h)
        return h, z
```

Figura 80: Bloque de Código SimCLR (evaluación SimCLR)

- Se activó el modo de evaluación con .eval() y se desactivó el cálculo de gradientes con torch.no_grad(), para que el modelo no actualice parámetros y la inferencia sea más eficiente. Luego se añadió una nueva parte al código para extraer los vectores h tanto como del conjunto de entrenamiento como del test. Para cada imagen del test, se midió la distancia mínima a los embedding normales. Ese valor se interpreta como un score de anomalía, cuanto mayor la distancia, más probable que la imagen sea patológica.

```
train_embeddings = extract_embeddings(model, train_loader)
train_embeddings = F.normalize(train_embeddings, dim=1)
train_embeddings = train_embeddings.to(device)

model.eval()
scores = []
labels = []

with torch.no_grad():
    print("Evaluando sobre el conjunto de test: ")
    for x, label in test_loader:
        x = x.to(device)
        output = model(x)
        h = output[0] if isinstance(output, tuple) else output
        h = F.normalize(h, dim=1)

        distances = F.pairwise_distance(h, train_embeddings)
        score = distances.min().item()

        scores.append(score)
        labels.append(label.item())
```

Figura 81: Bloque de Código evaluación SimCLR (SimCLR evaluación)

Con estos scores se calcularon distintas métricas. Se aplicaron umbrales automáticos usando los percentiles.

```python
from sklearn.metrics import roc_auc_score, f1_score, precision_score, recall_score, accuracy_score
import numpy as np
import pandas as pd

if len(scores) == 0:
    print("No se han generado scores.")
else:
    percentiles = [80, 85, 90, 95]
    metrics_results = []

    roc_auc = roc_auc_score(labels, scores)

    for p in percentiles:
        threshold = np.percentile(scores, p)
        y_pred = [1 if s > threshold else 0 for s in scores]

        f1 = f1_score(labels, y_pred, zero_division=0)
        precision = precision_score(labels, y_pred, zero_division=0)
        recall = recall_score(labels, y_pred, zero_division=0)
        accuracy = accuracy_score(labels, y_pred)

        metrics_results.append({
            "Percentil": p,
            "Umbral": threshold,
            "AUC": roc_auc,
            "F1-score": f1,
            "Precisión": precision,
            "Recall": recall,
            "Accuracy": accuracy
        })

    df_metrics = pd.DataFrame(metrics_results)
    print("Resultados comparativos por umbral:\n")
    print(df_metrics.round(4))
```

Figura 82: Bloque de Código desarrollo métricas SimCLR (evaluación SimCLR)

6.2.2 Resultados y análisis

Una vez extraídos los embeddings latentes de todas las imágenes de test y calculadas sus distancias mínimas respecto al conjunto de entrenamiento, se obtuvo un vector de scores de anomalía. Estos scores fueron evaluados según métricas clínicas, empleando como umbral de decisión los percentiles 80, 85, 90 y 95 de la distribución de scores.

Resultados comparativos por umbral:

	Percentil	Umbral	AUC	F1-score	Precisión	Recall	Accuracy
0	80	0.1228	0.6944	0.4000	0.8240	0.2641	0.5048
1	85	0.1302	0.6944	0.3099	0.7979	0.1923	0.4647
2	90	0.1387	0.6944	0.2340	0.8413	0.1359	0.4439
3	95	0.1516	0.6944	0.1185	0.7812	0.0641	0.4038

Tabla 7: Resumen resultados y análisis métricas evaluación SimCLR

Como puede observarse, el modelo alcanza su mejor F1-score (0.4000) al aplicar un umbral basado en el percentil 80, lo que sugiere que una decisión más laxa mejora el equilibrio entre precisión y sensibilidad. A pesar de que el recall se mantiene relativamente bajo (0.26 en su punto máximo), la precisión es elevada en todos los casos (>0.78), lo que indica que las anomalías que el modelo detecta tienden a ser verdaderas.

El área bajo la curva ROC (AUC) permanece estable en torno a 0.6944 para todos los percentiles, lo cual refleja una capacidad general aceptable del modelo para asignar scores más altos a imágenes anómalas, independientemente del umbral. Este valor se encuentra por encima del azar (0.5) y por encima del obtenido con el AutoEncoder básico.

La siguiente figura representa la evolución de las principales métricas clínicas según el umbral aplicado:

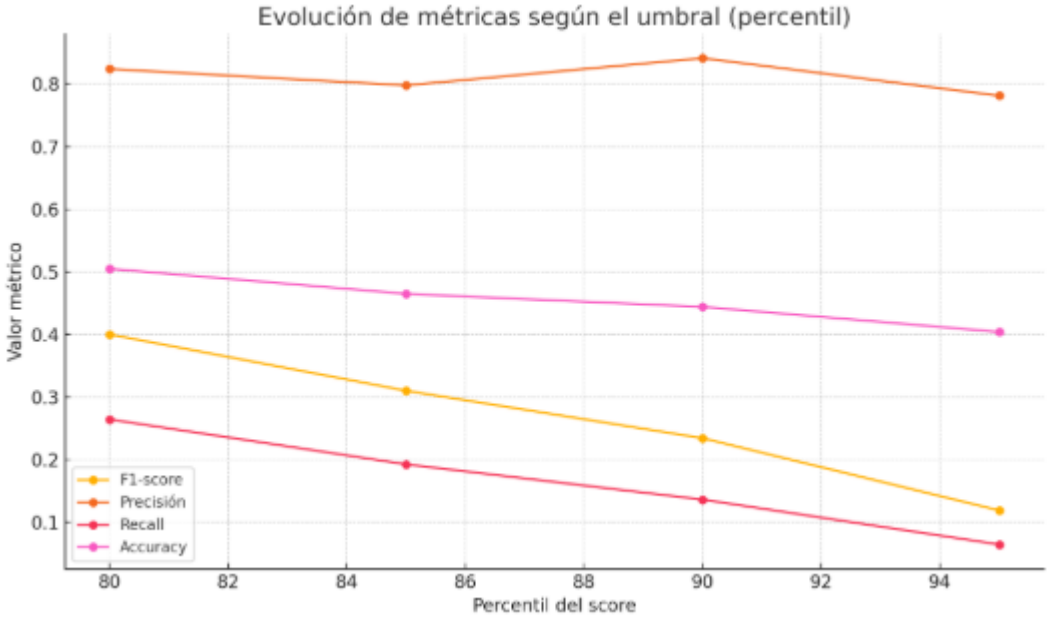

Figura 83: Evolución de métricas (F1-score, Precisión, Recall y Accuracy) en función del percentil de score aplicado como umbral. Se observa que a mayor exigencia en el umbral, la precisión se mantiene alta pero se reduce drásticamente el recall

Estos resultados indican que el modelo SimCLR ha logrado aprender una representación latente estructurada en la que las imágenes anómalas tienden a alejarse de la distribución normal, aunque no con suficiente discriminación para lograr una sensibilidad elevada.

6.3 Evaluación del modelo SWSSL

6.3.1 Metodología de evaluación

Una vez finalizado el entrenamiento del modelo SWSSL con su configuración óptima (optimizador SGD, 20 épocas, sin normalización L2), se procedió a su evaluación sobre el conjunto de test. Esta fase tiene como objetivo cuantificar la capacidad del modelo para detectar imágenes anómalas (casos de neumonía) sin utilizar etiquetas durante el entrenamiento.

Durante la inferencia, se recorrió el conjunto de test (test_loader) y, para cada par de vistas contrastivas (patch_i, patch_j) de una imagen, se calcularon sus proyecciones latentes z_i y z_j utilizando el modelo entrenado (model.eval()). La distancia euclídea media entre estas proyecciones se utilizó como score de anomalía.

Con este error de pérdida, posterior al entrenamiento, se cambió al conjunto de test:

```
Época 1/20                  Época 11/20
Pérdida promedio: 1.0518    Pérdida promedio: 0.0003

Época 2/20                  Época 12/20
Pérdida promedio: 0.6515    Pérdida promedio: 0.0002

Época 3/20                  Época 13/20
Pérdida promedio: 0.2674    Pérdida promedio: 0.0002

Época 4/20                  Época 14/20
Pérdida promedio: 0.0757    Pérdida promedio: 0.0002

Época 5/20                  Época 15/20
Pérdida promedio: 0.0136    Pérdida promedio: 0.0002

Época 6/20                  Época 16/20
Pérdida promedio: 0.0028    Pérdida promedio: 0.0002

Época 7/20                  Época 17/20
Pérdida promedio: 0.0011    Pérdida promedio: 0.0002

Época 8/20                  Época 18/20
Pérdida promedio: 0.0006    Pérdida promedio: 0.0002

Época 9/20                  Época 19/20
Pérdida promedio: 0.0004    Pérdida promedio: 0.0001

Época 10/20                 Época 20/20
Pérdida promedio: 0.0003    Pérdida promedio: 0.0001
```

Figura 84: Evolución pérdida épocas reentrenamiento (evaluación SWSSL)

```
dataset_path = "/content/chest_xray/chest_xray/test"
dataset = PatchContrastiveDataset(root_dir=dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True, num_workers=2)

[ ]  print("Número de muestras en test_dataset:", len(test_dataset))

Número de muestras en test_dataset: 624

/content/chest_xray/chest_xray/test → 0 archivos
/content/chest_xray/chest_xray/test/PNEUMONIA → 390 archivos
/content/chest_xray/chest_xray/test/NORMAL → 234 archivos
```

Figura 85: Cambio path dataset evaluación SWSSL + Número de muestrar en test_dataset (624)

```
import torch.nn.functional as F

model.eval()
scores = []
labels = []

with torch.no_grad():
    print("Evaluando sobre el conjunto de test: ")
    for patch_i, patch_j, label in test_loader:
        patch_i = patch_i.squeeze(0).to(device)
        patch_j = patch_j.squeeze(0).to(device)

        _, z_i = model(patch_i)
        _, z_j = model(patch_j)

        # Score de anomalía: distancia media entre z_i y z_j
        score = F.pairwise_distance(z_i, z_j, p=2).mean().item()
        scores.append(score)
        labels.append(label.item())

print(f"Total de muestras evaluadas: {len(scores)}")
```

Figura 86: Bloque de código evaluación (evaluación SWSSL)

Se evaluó el rendimiento utilizando distintos percentiles (80,85,90,95) como umbrales automáticos. Para cada umbral se calcularon (como se ha descrito en apartados anteriores):

```
Evaluando sobre el conjunto de test...
Total de muestras evaluadas: 624
```

Figura 87: Bloque de Código de evaluación (evaluación SWSSL)

Estas métricas se calcularon mediante el módulo 'sklearn.metrics' sobre los vectores scores y labels:

- **AUC**
- **F1-score**
- **Precisión**
- **Recall**
- **Accuracy**

Por otro lado, cada imagen de los test recibía las puntuaciones continuas de anomalías, scores continuos (por ejemplo, una distancia euclídea entre embeddings). Para evaluar métricas como precisión o recall, se necesita saber si una imagen se clasifica como anómala o no. Se necesita obtener una predicción binaria.

En lugar de elegir un umbral arbitrario, se usaron percentiles de la distribución de scores (por ejemplo, el percentil 90 es el valor por encima del cual se sitúa el 10% de los scores más altos)

Es una forma experimental y no supervisada de establecer un límite de decisión; mientras se observa la sensibilidad del modelo ante distintos escenarios de decisión.

6.3.3 Resultados y análisis

A continuación, se muestran los resultados obtenidos con los distintos umbrales, así como el código implementado para la obtención de los mismos.

```python
from sklearn.metrics import roc_auc_score, f1_score, precision_score, recall_score, accuracy_score
import numpy as np
import pandas as pd

if len(scores) == 0:
    print(" No se han generado scores. ")
else:
    percentiles = [80, 85, 90, 95]
    metrics_results = []

    # AUC no depende de umbral
    roc_auc = roc_auc_score(labels, scores)

    for p in percentiles:
        threshold = np.percentile(scores, p)
        y_pred = [1 if s > threshold else 0 for s in scores]

        f1 = f1_score(labels, y_pred, zero_division=0)
        precision = precision_score(labels, y_pred, zero_division=0)
        recall = recall_score(labels, y_pred, zero_division=0)
        accuracy = accuracy_score(labels, y_pred)

        metrics_results.append({
            "Percentil": p,
            "Umbral": threshold,
            "AUC": roc_auc,
            "F1-score": f1,
            "Precisión": precision,
            "Recall": recall,
            "Accuracy": accuracy
        })

    df_metrics = pd.DataFrame(metrics_results)
    print(" Resultados comparativos por umbral:\n")
    print(df_metrics.round(4))
```

Figura 88: Bloque de código para posterior visualización de métricas

	Percentil	Umbral	AUC	F1-score	Precisión	Recall	Accuracy
0	80	1.6146	0.4072	0.2524	0.5200	0.1667	0.3830
1	85	1.9938	0.4072	0.1818	0.4681	0.1128	0.3654
2	90	2.5304	0.4072	0.1280	0.4603	0.0744	0.3670
3	95	3.1717	0.4072	0.0711	0.4688	0.0385	0.3718

Tabla 8: Representación mediante metrics_results (lista de diccionarios) y método DataFrame de panda (rounded 4)

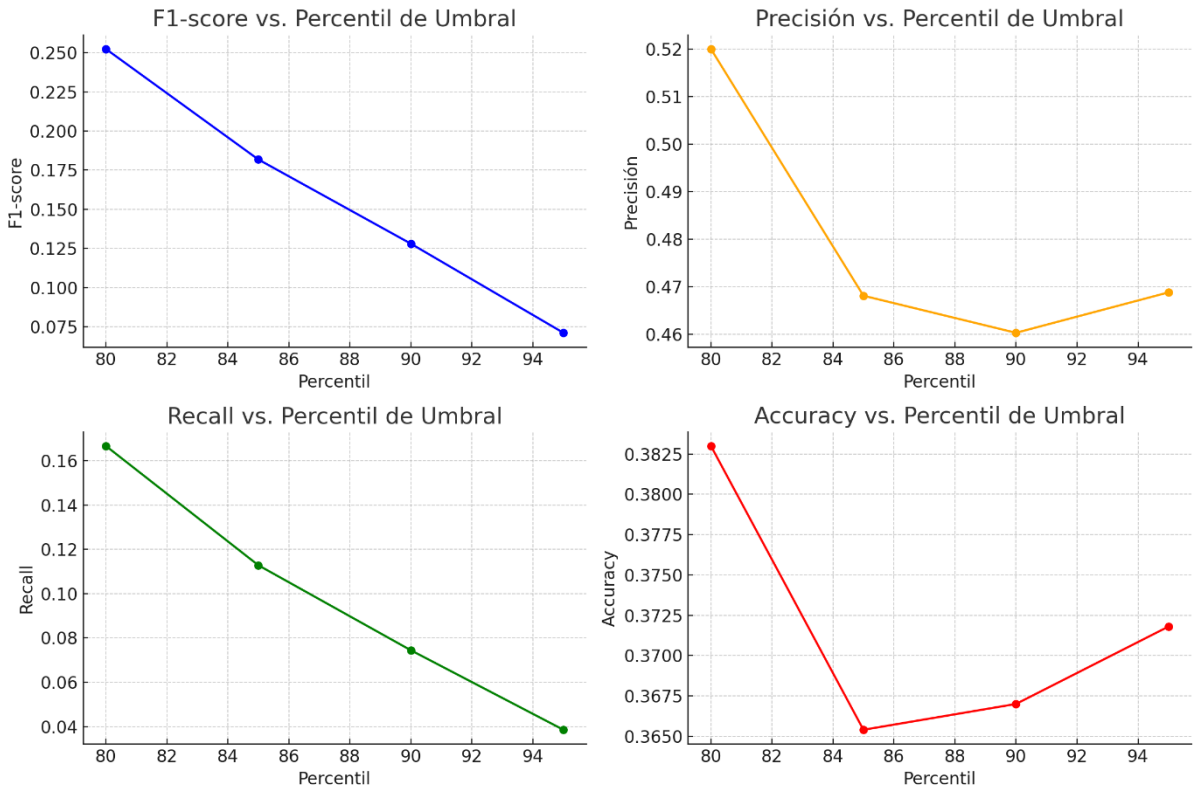

Figura 89: Gráficas resultantes comparación métricas en función del percentil de umbral

Aunque el modelo SWSSL alcanzó una pérdida extremadamente baja durante el entrenamiento (~0.0001), el rendimiento sobre el conjunto de test fue limitado en métricas como AUC, F1-score o precisión. Esta discrepancia revela un fenómeno conocido como colapso de representación, en el que el modelo proyecta de forma casi idéntica todos los ejemplos, independientemente de su clase. Esto suele ocurrir en modelos contrastivos cuando no existen suficientes restricciones estructurales o variabilidad real entre las vistas contrastivas.

En este trabajo, se utilizó inicialmente la Twin Loss como función de entrenamiento, omitiendo la Continuity Preserving Loss (CP Loss) propuesta en el artículo original de SWSSL [1], lo cual pudo haber facilitado dicho colapso. A pesar de ello, el análisis por umbral mostró que el modelo generaba scores con cierta variabilidad, lo que sugiere que sí hubo aprendizaje parcial de estructura en el espacio latente. Esta experiencia permitió identificar limitaciones clave y abre la puerta a propuestas de mejora que se detallan a continuación.

Con el objetivo de subsanar el colapso de representación observado en la primera versión del modelo SWSSL, se intentó la mejora experimental basada en [1]. Esta mejora consistía en incorporar dos modificaciones clave al proceso de entrenamiento:

- La Continuity Preserving Loss (CP Loss), implementada como una pérdida MSE ('torch.nn.functional.mse_loss') entre las salidas intermedias del encoder (h_i, h_j), con el objetivo de preservar la coherencia estructural entre vistas similares del mismo parche.
- La reintroducción de la normalización L2 sobre los vectores proyectados (z_i, z_j) antes del cálculo de la pérdida contrastiva (Twin Loss), siguiendo la formulación común en métodos contrastivos como SimCLR o el propio SWSSL.

Se definió una nueva función 'cp_loss' y se actualizó la función 'twin_loss' para incluir la normalización que por defecto es L2.

```python
def cp_loss(h1, h2):
    return F.mse_loss(h1, h2)


def twin_loss(z1, z2, temperature=0.5):
    z1 = F.normalize(z1, dim=1)
    z2 = F.normalize(z2, dim=1)
    logits = torch.matmul(z1, z2.T) / temperature
    labels = torch.arange(z1.size(0)).to(z1.device)
    return F.cross_entropy(logits, labels)

h_i, z_i = model(patch_i)
h_j, z_j = model(patch_j)

loss_twin = twin_loss(z_i, z_j)
loss_cp = cp_loss(h_i, h_j)
loss = loss_twin + 0.1 * loss_cp  # Ponderación de CP Loss
```

Figura 90: Cambios función pérdida cp_loss (evaluación SWSSL)

A pesar de que estas implementaciones se implementaron correctamente, la ejecución del entrenamiento completo resultó inviable en el entorno gratuito de Google Colab. Este entorno impone limitaciones de recursos computaciones (tiempo de sesión, disponibilidad de la GPU compartida y velocidad de procesamiento). Si se combina con la nueva complejidad del modelo, con dos funciones de pérdida, augmentaciones contrastivas, más vistas por imagen; genera unos tiempos de entrenamiento excesivos e inestables.

A pesar de ello, el código desarrollado queda completamente funcional y preparado para su ejecución en un entorno más potente.

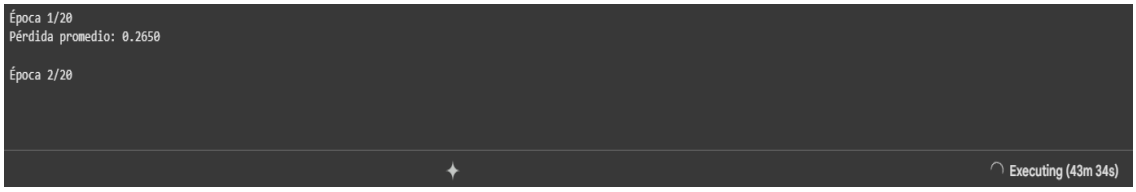

Figura 91: Tiempo de ejecución adaptando el entrenamiento al propuesto en [1] (evaluación SWSSL)

Se estudió la posibilidad de desarrollar las pruebas en el entorno que proporciona kaggle, para trabajar con notebooks con acceso a GPUs bastante eficientes para esta etapa de evaluación. Se consiguieron resultados positivos.

Resultados comparativos por umbral:

	Percentil	Umbral	AUC	F1-score	Precisión	Recall	Accuracy
0	80	0.4128	0.5303	0.3262	0.6720	0.2154	0.4439
1	85	0.4908	0.5303	0.2769	0.7128	0.1718	0.4391
2	90	0.7316	0.5303	0.2163	0.7778	0.1256	0.4311
3	95	0.9625	0.5303	0.1137	0.7500	0.0615	0.4006

Tabla 9: Resumen resultados comparativos por umbral (evaluación SWSSL)

Se puede observar que se evitó el colapso de representaciones y mejorar así la discriminación den el espacio latente. La incorporación de CP Loss ha mejorado la separación de clases en el espacio latente, logrando un mejor equilibrio entre detección de anomalías y reducción de falsos positivos.

6.3.4 Comparación con los resultados del paper original

La implementación propuesta logra aproximarse a los principios fundamentales de SWSSL, pero sus resultados quedan condicionados por restricciones técnicas y simplificaciones necesarias en un entorno académico. Aun así, el análisis por umbrales y la mejora progresiva del rendimiento muestran que el enfoque auto-supervisado es válido y adaptable, abriendo la puerta a futuras mejoras más robustas.

El paper original utiliza datasets de mayor tamaño y variabilidad, lo que mejora la capacidad del modelo para generalizar. En este trabajo, se ha entrenado con un subconjunto

reducido y específico del dataset ChestX-ray. Por otro lado, se entrena el modelo durante 10 veces la cantidad de épocas empleadas en este TFG.

7. Conclusiones y líneas futuras

Este Trabajo de Fin de Grado ha analizado la posibilidad del aprendizaje auto-supervisado como una estrategia viable para detectar anomalías en radiografías de tórax sin necesidad de contar con datos etiquetados. A través de la implementación de tres enfoques distintos se ha llevado a cabo una evaluación comparativa rigurosa, aplicando métricas clínicas estándar y utilizando un proceso experimental igual para todos los modelos.

Los resultados obtenidos, aunque no alcanzan todavía niveles clínicos de precisión, demuestran que los modelos son capaces de detectar diferencias claras entre imágenes normales y con patología a través de las representaciones internas que aprenden. Esta conclusión respalda la idea central del trabajo: es posible entrenar modelos útiles solo con imágenes normales, sin recurrir a etiquetas manuales, y obtener señales válidas para la detección de anomalías.

El AutoEncoder ha mostrado una evolución positiva al introducir cambios como el uso de pérdidas más sensibles (L1 y edge loss), compresión del espacio latente y la introducción de ruido durante el entrenamiento. SWSSL, aunque más complejo, ha permitido explorar mecanismos como el uso de ventanas deslizantes y la combinación de pérdidas contrastivas y de continuidad (CP Loss) que ayudan a que el modelo mantenga cierta coherencia espacial. Por su parte, SimCLR ha demostrado que, incluso con una arquitectura más simple y una única pérdida contrastiva (NT-Xent), puede generar representaciones útiles y diferenciadoras entre lo normal y lo anómalo.

Más allá de los valores exactos obtenidos en las métricas, el proyecto ha servido para identificar qué estrategias tienen más potencial, qué obstáculos surgen al aplicarlas en imágenes médicas, y cómo podrían escalarse o adaptarse mejor. A lo largo del trabajo, se ha comprobado que con un mayor número de datos, más diversidad, y sobre todo con más recursos computacionales, sería posible obtener resultados mucho más robustos. Aunque no se ha llegado al rendimiento que muestran los artículos originales (por ejemplo, en el caso de SWSSL o SimCLR), se han replicado ideas clave y se ha demostrado que el enfoque es prometedor.

Además, se ha construido una base de código modular y reutilizable, que puede adaptarse fácilmente para futuros experimentos o para otros tipos de imágenes médicas. El análisis realizado también deja clara la importancia de combinar varias métricas (no solo la accuracy), y de interpretar los resultados teniendo en cuenta el contexto clínico real.

En definitiva, este trabajo no solo ha permitido evaluar distintas técnicas de forma crítica, sino también aportar una visión más práctica y realista de hasta dónde pueden llegar estas metodologías con los medios disponibles, y qué se podría conseguir si se aplican en entornos más avanzados.

7.1 Líneas futuras

A partir del trabajo realizado, se pueden plantear varias líneas para continuar o ampliar este proyecto:

- **Entrenar durante más épocas y con más datos**: en todos los modelos se han usado recursos limitados, por lo que una ampliación del número de imágenes normales y patológicas, junto con más tiempo de entrenamiento, podría mejorar mucho el rendimiento.

- **Usar modelos preentrenados**: especialmente en el caso de SimCLR y SWSSL, partir de una red entrenada previamente (por ejemplo, con ImageNet o un dataset médico mayor) podría dar lugar a representaciones más potentes desde el principio.

- **Mejorar las estrategias de augmentación**: probar combinaciones más variadas y adaptadas a imágenes médicas podría ayudar a que los modelos aprendan características más discriminativas.

- **Incluir más tipos de anomalías**: el trabajo actual se ha centrado en detectar neumonía, pero los modelos podrían aplicarse a otros tipos de patologías si se cuenta con más variedad en el conjunto de datos.

- **Explorar visualizaciones y explicaciones**: añadir herramientas que permitan visualizar qué zonas de la imagen activan más al modelo o qué diferencias capta podría hacer que estos enfoques fueran más comprensibles y útiles para profesionales clínicos.

- **Aplicar los modelos a otros formatos médicos**: radiografías de otras regiones, tomografías, o imágenes de resonancia también podrían beneficiarse de este enfoque, adaptando el preprocesamiento.

En conjunto, el trabajo realizado sienta una base sólida que demuestra que el aprendizaje auto-supervisado no solo es aplicable al entorno médico, sino que tiene mucho potencial por explotar. Lo que ahora es una aproximación exploratoria, puede convertirse con más recursos y tiempo en una herramienta real de apoyo al diagnóstico.

APÉNDICE

Apéndice A – Descargar del conjunto de datos desde Kaggle según el entorno de trabajo

Para entrenar y evaluar los modelos en este Trabajo de Fin de Grado se ha utilizado el conjunto de datos *ChestX-ray*, alojado en Kaggle en la cuenta del autor. La forma de descargar el dataset varía según el entorno en el que se trabaje, ya que tanto Google Colab como Kaggle Notebooks ofrecen herramientas dedicadas para cargar datos desde la plataforma.

A.1 Descarga desde Google Colabs

En Google Colab, al no estar directamente conectado a la cuenta de Kaggle, es necesario subir manualmente el archivo 'kaggle.json' (token de autenticación), que se obtiene desde el perfil del usuario en 'https://www.kaggle.com/account' al pulsar el botón 'Create New Token'

Una vez subido como en capítulos anteriores se ejecuta este bloque de código:

```
# Subir kaggle.json manualmente cada vez que se ejecute
from google.colab import files
files.upload()  # Subir kaggle.json cuando lo pida (create new token para conectarse a API)

!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

!pip install -q kaggle

!kaggle datasets download -d carlospinopadilla/chestxray
!unzip -q chestxray.zip -d chest_xray
```

Figura 92: Upload kaggle.json desde google colab

Posteriormente hay que descargarse ‘kaggle.json’ para poder conectarse a la API donde está el dataset en cuestión.

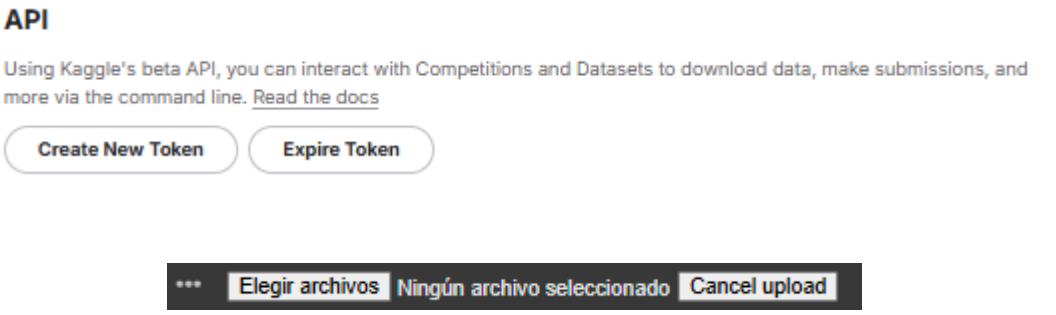

Figura 93: Elegir archivo kaggle.json, generado en ‘Create New Token’

Importante: Este token caduca tras cada sesión, por lo que debe subirse de nuevo si se reinicia el entorno

A.2 Descarga desde Kaggle Notebooks

En notebooks nativos de Kaggle, la descarga del dataset es directa y no requiere token adicional. Basta con usar la librería ‘kagglehub’

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("carlospinopadi
lla/chestxray")

print("Path to dataset files:", path)
```

Figura 94: Código importar dataset kaggle de forma nativa

A.3 Consideraciones sobre los entornos

Tanto Google Colab como Kaggle Notebooks permite ejecutar modelos y visualizar resultados. Sin embargo, Kaggle ofrece mayor tiempo de sesión activa, así como más gráficas disponibles, lo que facilita entrenamientos más largos. Esto hace que Kaggle sea especialmente cómodo para este tipo de pruebas

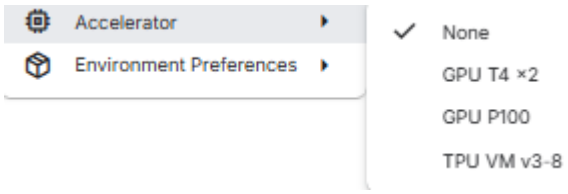

Figura 95: Gráficas dedicadas disponibles en el entorno de Kaggle

Referencias

[1] H. Dong, Y. Zhang, H. Gu, N. Konz, Y. Zhang and M. A. Mazurowski, "SWSSL: Sliding Window-Based Self-Supervised Learning for Anomaly Detection in High-Resolution Images," in IEEE Transactions on Medical Imaging, vol. 42, no. 12, pp. 3860-3870, Dec. 2023. https://ieeexplore.ieee.org/document/10247020

[2] X. Huang, H. Wang, C. She, J. Feng, X. Liu, X. Hu, L. Chen, and Y. Tao, "Artificial intelligence promotes the diagnosis and screening of diabetic retinopathy," Frontiers in Endocrinology, vol. 13, 2022. https://www.frontiersin.org/journals/endocrinology/articles/10.3389/fendo.2022.946915/full

[3] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A Simple Framework for Contrastive Learning of Visual Representations," in *Proceedings of the 37th International Conference on Machine Learning*, 2020. https://proceedings.mlr.press/v119/chen20j/chen20j.pdf

[4] Repositorio oficial de SimCLR: https://github.com/google-research/simclr

[5] G. E. Hinton,R, R. Salakhutdinov, Reducing the Dimensionality of Data with Neural Networks.*Science***313**,504-507(2006).:10.1126/science.1127647

[6] Dataset "PadChest": https://paperswithcode.com/dataset/padchest

[7] T. Zhou et al., "Contrastive learning for chest X-ray disease classification," *Medical Imaging with Deep Learning*, 2021.: https://doi.org/10.1016/j.asoc.2024.112101.

[8] S. Azizi et al., "Big self-supervised models advance medical image classification," *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021: https://doi.org/10.48550/arXiv.2101.05224

[9] Busto, J., et al. (2019). PadChest: A large chest x-ray image dataset with multi-label annotated reports. *Medical Image Analysis*, 66, 101797. https://doi.org/10.1016/j.media.2020.101797

[10] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014. https://arxiv.org/abs/1412.6980

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014. https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

[12] Y. LeCun, Y. Bengio and G. Hinton, “Deep learning,” Nature, vol. 521, no. 7553, pp. 436–444, 2015. https://www.nature.com/articles/nature14539

[13] Y. Zhou, X. Xu, Y. Jiang, et al., “Contrastive learning for chest X-ray disease classification,” Applied Soft Computing, vol. 141, 2024. https://doi.org/10.1016/j.asoc.2024.112101

[14] TDS Editors, “Using Kaggle Datasets in Google Colab,” Towards Data Science, Medium, 2020. https://towardsdatascience.com/using-kaggle-datasets-in-google-colab-b0a8e2ef5d15

[15] carlosspino/SWSSL. GitHub. Disponible en: https://github.com/carlosspino/SWSSL *Adaptado y modificado para su integración en el presente trabajo.*

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga