Aplicación de técnicas de procesamiento de imágenes en videojuegos

Alex Herrerías Ramírez
Ingeniería Informática
Inteligencia Artificial

Nombre consultor: Jonathan Ferrer Mestres
Nombre PRA: Friman Sánchez Castaño

Fecha de entrega: 12/01/2025


Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Aplicación de técnicas de procesamiento de imágenes en videojuegos*
Nombre del autor:	*Alex Herrerías Ramírez*
Nombre del consultor/a:	*Jonathan Ferrer Mestres*
Nombre del PRA:	*Friman Sánchez Castaño*
Fecha de entrega (mm/aaaa):	10/2024
Titulación::	*Ingeniería informática*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Denoising, Super Resolución, Calidad visual*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El objetivo principal de este proyecto es mejorar la calidad visual de las imágenes en videojuegos utilizando imágenes de videojuegos generadas con el motor gráfico *Unreal Engine*. La aplicación se centra en aplicar técnicas de superresolución y *denoising* para mejorar la calidad visual de las imágenes, aumentando la resolución y eliminando el ruido. Se ha optado por desarrollar este proyecto debido a que la calidad visual es fundamental para la experiencia del jugador.

La metodología utilizada consiste en la creación de un flujo de trabajo para aplicar técnicas de superresolución y posteriormente *denoising* a imágenes de videojuegos en baja resolución (128x128 píxeles). Se implementará una arquitectura *U-Net* modificada, un tipo de red neuronal convolucional utilizada en aplicaciones de procesamiento de imágenes, que se entrenará utilizando una base de datos de imágenes de videojuegos. Se ajustarán las configuraciones para optimizar los resultados, incluyendo la incorporación de capas residuales y la optimización de la función de activación.

Se espera obtener resultados que muestren una mejora significativa en la calidad visual de diferentes videojuegos, con un aumento en la calidad de la imagen y en la reducción de ruido.

Abstract (in English, 250 words or less):

The main objective of this project is to improve the visual quality of images in video games using video game images generated with the Unreal Engine graphics engine. The application focuses on applying super-resolution and denoising techniques to improve the visual quality of images, increasing resolution and removing noise. We have chosen to develop this project because visual quality is essential for the player's experience.

The methodology used consists of creating a workflow to apply super-resolution techniques and subsequently eliminate noise from low-resolution video game images (128x128 pixels). A modified U-Net architecture will be implemented, a type of convolutional neural network used in image processing applications, which will be trained using a database of video game images. Settings will be adjusted to optimize results, including adding residual layers and optimizing the activation function.

It is expected to obtain results that show a significant improvement in the visual quality of different video games, with an increase in image quality and noise reduction.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

En la industria de los videojuegos, la calidad visual desempeña un papel fundamental en la experiencia de usuario. Los jugadores buscan gráficos más realistas y con texturas de gran calidad. Sin embargo, la generación de gráficos de alta calidad requiere una gran cantidad de recursos computacionales, lo que supone un desafío para los desarrolladores.

Muchos usuarios no disponen del *hardware* necesario en sus equipos para renderizar gráficos en alta resolución.

Los desarrolladores se centran en el desarrollo de técnicas de renderizado que permitan optimizar el uso de los recursos computacionales sin sacrificar la calidad visual. Una de estas técnicas es la superresolución, que se aprovecha de generar imágenes de alta resolución (*HR*) a partir de imágenes en baja resolución (*LR*), como el DLSS de NVIDIA[18], que utiliza algoritmos de aprendizaje profundo para escalar imágenes a mayor resolución con una notable mejora en la calidad visual. Si bien existen métodos tradicionales de superresolución, como la interpolación bicúbica, estos a menudo presentan limitaciones en cuanto a la calidad visual final.

En los últimos años, las redes neuronales convolucionales (*CNN*) han demostrado un gran potencial en la superresolución de imágenes. Arquitecturas como la *U-Net*, con una gran capacidad para capturar información contextual a múltiples escalas gracias a sus conexiones de salto, se han utilizado con éxito en diversas aplicaciones de procesamiento de imágenes.

1.2 Objetivos del Trabajo

El presente trabajo se centra en el desarrollo de un modelo de superresolución basado en una arquitectura *U-Net* modificada, denominada *Robust U-Net*. Para lograr este objetivo, se plantean los siguientes objetivos específicos:

- **Implementar una arquitectura *Robust U-Net*:** Incorporar una modificación basada en la arquitectura *U-Net, Robust U-Net* y posteriormente hacer uso de capas residuales para mejorar la capacidad de aprendizaje y la eficiencia del entrenamiento. Estas capas residuales permitirán un flujo más directo del gradiente durante el entrenamiento, facilitando la optimización de redes más profundas y mitigando el problema de la degradación del gradiente.

- **Optimizar la función de activación:** Explorar diferentes funciones de activación en las capas convolucionales para maximizar la capacidad de la red de aprender patrones no lineales.
- **Aplicar técnicas de Denoising:** Someter el modelo *Robust U-Net* una vez ya entrenado a un proceso de *Denoising* para eliminar el ruido presente en las imágenes resultantes de la superresolución y mejorar la nitidez visual.
- **Evaluar el modelo en imágenes de videojuegos:** Aplicar el modelo a imágenes de diferentes videojuegos generadas con *Unreal Engine* para mejorar su calidad visual.

  El objetivo final es obtener un modelo de superresolución que permita generar imágenes de alta calidad a partir de imágenes de baja resolución.

1.3 Enfoque y método seguido

La estrategia usada consiste en el desarrollo de una modificación sobre la arquitectura de red *U-net* original[2], ampliamente utilizada en tareas de procesamiento de imágenes gracias a el uso de las *skip-connections* y a su estructura la cual consiste en el uso de diferentes capas en el codificador y por separado las capas del decodificador. Sin embargo, en lugar de utilizar una *U-Net* estándar, se optó por implementar una variante modificada denominada *Robust U-net*[3]

La decisión de emplear la arquitectura *Robust U-Net* se basa en su capacidad para mejorar la robustez y el rendimiento del modelo en tareas de superresolución. Esta arquitectura introduce modificaciones en la estructura de la *U-Net* que la hacen más eficiente en el aprendizaje de características complejas y en la generación de imágenes de alta calidad.

Para la implementación, se ha seguido la siguiente metodología, la cual dividiremos en tres fases principales:

Fase 1. Implementación de la arquitectura *Robust U-Net*:

- **Adaptación de la *U-Net*:** Se partió de la arquitectura *U-Net* tradicional y se implementaron las modificaciones necesarias para convertirla en una *Robust U-Net*. Estas modificaciones incluyen la incorporación de capas adicionales y la optimización de Hiperpárametros.
- **Entrenamiento inicial:** Se entrenó el modelo *Robust U-Net* con un dataset de imágenes de videojuegos de *Unreal Engine*. Este entrenamiento inicial permitió evaluar el rendimiento de la arquitectura base y ajustar los parámetros del modelo.

Fase 2. Incorporación de capas residuales:

- **Integración de conexiones residuales:** Se añadieron capas residuales a la arquitectura *Robust U-Net*. Las conexiones residuales permiten un flujo más directo del gradiente durante el entrenamiento, lo que facilita la optimización de redes más profundas y mitiga el problema de la degradación del gradiente.

- **Reentrenamiento del modelo:** Se reentrenó el modelo con las capas residuales utilizando el mismo *dataset* de imágenes de videojuegos. Se evaluó el impacto de las capas residuales en el rendimiento del modelo en términos de calidad de imagen y eficiencia del entrenamiento.

Fase 3. Desarrollo de la capa de *Denoising*:

- **Investigación de técnicas de *Denoising*:** Se investigaron diferentes técnicas de *Denoising* aplicables a imágenes de videojuegos. Se seleccionó la técnica más adecuada en función de su eficiencia y su capacidad para eliminar el ruido sin afectar la nitidez de la imagen.
- **Implementación de la capa de *Denoising*:** Se implementó la capa de *Denoising* y se integró en el modelo de superresolución.
- **Evaluación final:** Se evaluó el rendimiento del modelo completo, incluyendo la capa de *Denoising*, utilizando métricas de calidad de imagen y comparaciones visuales.

1.4 Planificación del Trabajo

	tasks:		-	25/Sep	12/Jan
	Buscar propuesta de proyecto	Unassigned	-	25/Sep	07/Oct
2	Reúnir información sobre la propuesta del proyecto	Unassigned	-	25/Sep	07/Oct
3	Analizar la viabilidad del proyecto	Unassigned	-	25/Sep	07/Oct
	Investigación del proyecto y recopilación de información.	Unassigned	-	08/Oct	16/Oct
5	Revisar los requisitos del proyecto	Unassigned	-	08/Oct	16/Oct
6	Establecer el alcance y los objetivos del proyecto	Unassigned	-	08/Oct	16/Oct
	Desarrollo y codificación del proyecto	Unassigned	-	16/Oct	01/Dec
8	Desarrollar la arquitectura del proyecto	Unassigned	-	16/Oct	31/Oct
9	Implementar el modelo principal	Unassigned	-	31/Oct	15/Nov
10	Integrar y Testear los modelos	Unassigned	-	15/Nov	01/Dec
	Evaluación y Ajustes del Proyecto	Unassigned	-	01/Dec	23/Dec
12	Evaluar el desempeño del proyecto	Unassigned	-	01/Dec	15/Dec
13	Implementar los ajustes necesarios	Unassigned	-	15/Dec	23/Dec
	Documentación y presentación del proyecto.	Unassigned	-	24/Dec	12/Jan
15	Redactar la memoria del proyecto	Unassigned	-	24/Dec	05/Jan
16	Preparar la presentación final	Unassigned	-	05/Jan	12/Jan

Figura 1 Planificación de fechas del diagrama de Gantt

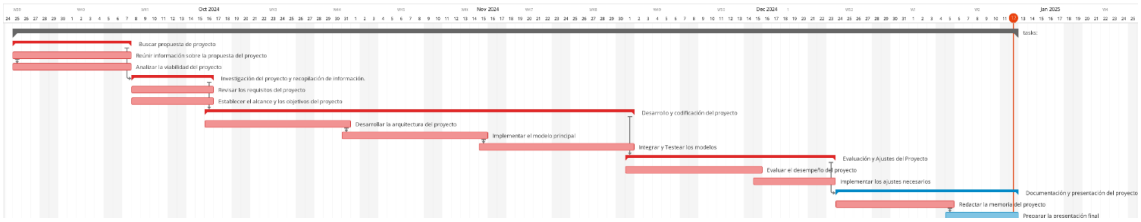

Figura 2 Diagrama de Gantt

1.5 Breve sumario de productos obtenidos

Los principales productos obtenidos en este trabajo son:

- **Implementación de un modelo de superresolución basado en la arquitectura *Robust U-Net*.** Este modelo se ha optimizado para su aplicación en imágenes de videojuegos.
- **Desarrollo de una capa de *Denoising* para la eliminación de ruido en las imágenes superresueltas.**
- **Evaluación del rendimiento del modelo en diferentes datasets de imágenes de videojuegos.** Se han utilizado métricas de calidad de imagen para cuantificar la mejora obtenida.
- **Redacción de una memoria que documenta el trabajo realizado.** La memoria incluye una descripción detallada de la metodología, los resultados y las conclusiones del trabajo.

1.6 Breve descripción de los otros capítulos de la memoria

Capítulo 2 (Marco Teórico): En este capítulo se proporciona los fundamentos teóricos necesarios para comprender el trabajo realizado. Se describen las técnicas de superresolución, las redes neuronales convolucionales, la arquitectura **U-Net** y las métricas de evaluación utilizadas.

Capítulo 3 (Metodología): En este capítulo se detalla la metodología seguida para el desarrollo del modelo de superresolución. Se describe el *dataset* utilizado, la arquitectura del modelo, el proceso de entrenamiento y las técnicas de preprocesamiento de datos.

Capítulo 4 (Resultados): En este capítulo se presentan los resultados obtenidos en la evaluación del modelo. Se muestran las mejoras en la calidad de imagen logradas con la superresolución y el denoising.

Capítulo 5 (Conclusiones): En este capítulo se resumen las conclusiones del trabajo desarrollado, se discuten las limitaciones encontradas y se proponen líneas de trabajo futuro.

2. Marco Teórico

2.1 Framework, software y hardware utilizado

2.1.1 Visual Studio Code

Se ha elegido *Visual Studio Code*[4] como entorno de desarrollo integrado (*IDE*) debido a su amplio soporte para *Python*. *Visual Studio Code* ofrece los recursos necesarios para el desarrollo del proyecto, incluyendo soporte integrado para *Git*, el cual ha sido usado para el control de versiones durante el desarrollo del proyecto y almacenamiento del resultado final del proyecto[1].

2.1.2 Python

La implementación del código y todas sus funciones a lo largo del trabajo se ha realizado mediante el lenguaje de programación *Python*[5]. *Python* es el lenguaje estandarizado para el uso en las aplicaciones de inteligencia artificial y aprendizaje computacional, su disponibilidad de librerías y frameworks orientados a la inteligencia artificial, como *Pytorch*, *Matplotlib*, *Numpy*, *Scikit-learn* usados en este proyecto, permiten el uso de algoritmos escritos en lenguajes de programación diferentes que permiten el uso del lenguaje sin ser afectado por la eficiencia de cómputo en *Python*.

2.1.3 PyTorch

Se ha seleccionado *PyTorch*[6] como el framework principal para este proyecto. *PyTorch* es una librería de código abierto para aprendizaje automático, que ofrece lo necesario para el desarrollo de este proyecto:

- **Facilidad de uso:** *PyTorch* ofrece una sintaxis intuitiva y una *API* fácil de aprender, lo que facilita el desarrollo y la experimentación con modelos de aprendizaje automático.
- **Eficiencia:** *PyTorch* está optimizado para el rendimiento, especialmente en *GPUs*, lo que permite entrenar modelos de forma rápida y eficiente.
- **Flexibilidad:** *PyTorch* soporta grafos de computación dinámicos, lo que permite modificar la estructura del modelo durante el entrenamiento. Esto es especialmente útil para la investigación y el desarrollo de nuevas arquitecturas de modelos.
- ***Eager Execution*:** *PyTorch* utiliza un modo de ejecución *eager*, lo que significa que las operaciones se ejecutan inmediatamente cuando se definen. Esto facilita la depuración y hace que el proceso de desarrollo sea más intuitivo en comparación con los frameworks que utilizan grafos de computación estáticos.

2.1.4 Otras librerías

Además de las librerías de *PyTorch*, se han utilizado las siguientes:

- ***Matplotlib*:** Se utiliza para la creación de gráficos y visualizaciones, lo que permite representar los datos y los resultados del modelo de forma clara y concisa.
- ***Scipy*:** Esta librería proporciona herramientas para la computación científica y técnica, incluyendo algoritmos de optimización, procesamiento de señales y análisis estadístico.

2.1.5 Hardware

Como *Hardware*, se ha usado una tarjeta gráfica *CUDA RTX 3080* con 10GB de *VRAM*, encargada de los cálculos necesarios que se realizaran en el entrenamiento del modelo, acompañada de 32GB de memoria *RAM* y un procesador *AMD 7800X3D*.

2.2 Superresolución

La base y propósito de este proyecto es la superresolución, la superresolución es una técnica de procesamiento de imágenes que busca aumentar la resolución de una imagen digital. En otras palabras, se trata de generar una imagen de alta resolución (*HR*) a partir de una o varias imágenes de baja resolución (*LR*). Este proceso tiene como objetivo recuperar la información de alta frecuencia que se pierde al reducir la resolución de una imagen, lo que se traduce en una mejora en la nitidez, la definición de los detalles y la calidad visual general.

Figura 3 Diferencia de detalles entre dos diferentes resoluciones

Se puede observar en la figura 1, la diferencia de detalles entre una imagen en baja resolución (128x128 pixeles) y una imagen de alta resolución (512x512 pixeles).
Mediante el proceso de superresolución, el objetivo será conseguir una imagen con la mayor semejanza posible partiendo de la imagen de baja resolución.

2.2.1 Definición

Formalmente, podríamos definir la superresolución como un problema inverso en el que se busca encontrar una imagen *HR* que, al ser sometida a un proceso de degradación (como la reducción de la resolución), produzca una imagen *LR* similar a la que se tiene como entrada. Este proceso de degradación se puede modelar como:

$$y = DBx + n$$

Donde y es la imagen de baja resolución observada, D es la imagen de alta resolución que se busca reconstruir, B es un operador de decimación que reduce la resolución de la imagen, x es un operador de *blurring* que introduce desenfoque en la imagen, y n es el ruido presente en la imagen.
El objetivo de la superresolución es encontrar una estimación de x a partir de y, lo que implica invertir el proceso de degradación.

2.2.2 Técnicas tradicionales

Para abordar el problema de la superresolución, antes del aprendizaje profundo, se desarrollaron diversas técnicas, las cuales se pueden clasificar en dos categorías principales:

- Superresolución basada en interpolación[7]: Estos métodos se basan en la interpolación de los píxeles existentes para generar nuevos píxeles y aumentar la resolución de la imagen. Algunos ejemplos comunes son la interpolación bicúbica y la interpolación bilineal. Sin embargo, estas técnicas suelen producir imágenes borrosas o con artefactos, ya que no son capaces de recuperar la información de alta frecuencia que se pierde al reducir la resolución.
- Superresolución basada en múltiples imágenes[8]: Estos métodos utilizan información de varias imágenes *LR* de la misma escena, capturadas desde diferentes ángulos o con ligeros desplazamientos. Al alinear y fusionar estas imágenes, se puede obtener una imagen *HR* con más detalles. Un ejemplo clásico es el método de "desplazamiento y suma" (*shift-and-add*)[9].

2.2.3 Superresolución basada en Aprendizaje Profundo

En los últimos años, el aprendizaje profundo ha revolucionado el campo de la superresolución, ofreciendo resultados significativamente mejores que las técnicas tradicionales mencionadas anteriormente. Las *CNN* han demostrado su capacidad para aprender patrones complejos y generar resultados de alta calidad.

Las *CNNs* para superresolución aprenden una función de mapeo no lineal entre las imágenes *LR* y *HR* a partir de un conjunto de datos de entrenamiento. Durante el entrenamiento, la red ajusta sus pesos para minimizar la diferencia entre las imágenes *HR* generadas y las imágenes *HR* reales, profundizaremos en el tipo de *CNNs* y sus diferencias en el siguiente apartado.

2.3 Redes Neuronales Convolucionales

Las *CNN* se distinguen como un tipo de red neuronal artificial, diseñadas específicamente para el procesamiento de datos con una estructura de grid, como las imágenes. Su arquitectura se inspira en la organización del córtex visual del cerebro, donde las neuronas responden de manera selectiva a estímulos en regiones específicas del del campo visual.

2.3.1 Introducción a las CNNs

Una característica distintiva de las *CNN* es la incorporación de capas convolucionales, las cuales aplican filtros a la entrada para extraer características relevantes. Estos filtros operan mediante un mecanismo de deslizamiento sobre la imagen, realizando operaciones matemáticas con los píxeles. Las características resultantes de este proceso se pasan a capas posteriores para su procesamiento.

Además de las capas convolucionales, las *CNNs* suelen incorporar otros tipos de capas, como las capas de *pooling* que se encargan de reducir la dimensionalidad de los datos, y capas totalmente conectadas, que realizan la clasificación final.

2.3.2 Arquitectura U-net

La arquitectura *U-Net*, propuesta por *O. Ronneberger* en 2015[2], es una *CNN* especialmente efectiva para la segmentación de imágenes médicas. Su nombre proviene de su forma de "U", que refleja su estructura de codificador-decodificador.

El codificador reduce la resolución de la imagen a través de capas convolucionales y de *pooling*, extrayendo características de alto nivel. El decodificador aumenta la resolución de la imagen a través de capas de convolución transpuesta, combinando las características de alto nivel con la información espacial de las capas del codificador.

Una característica clave de la *U-Net* son las conexiones de salto (*skip connections*), que conectan las capas del codificador con las capas del decodificador. Estas conexiones permiten que la red conserve la información espacial de las capas iniciales, lo que es crucial para la segmentación precisa de imágenes.

En la figura 2 se muestra un ejemplo de arquitectura U-Net.

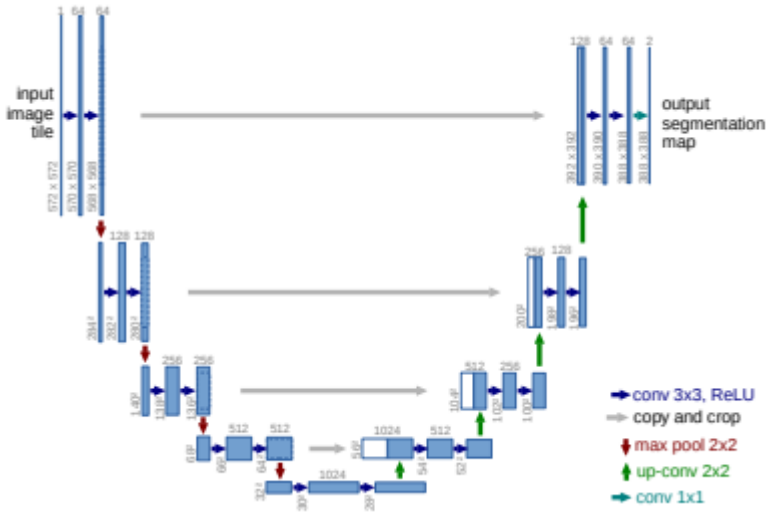

Figura 4 Ejemplo de arquitectura U-net (https://arxiv.org/pdf/1505.04597)

2.3.3 Variante de la U-Net: Robust U-Net (RUNet)

Desde la introducción de la *U-Net* para la segmentación de imágenes médicas[2], numerosas variantes han surgido buscando mejorar su rendimiento o adaptarlas a diferentes aplicaciones, para este proyecto, me he enfocado en la variante llamada *Robust U-Net*, la cual se enfoca en mejorar la robustez y el rendimiento del modelo en tareas de superresolución.

La *RUnet*, como se describe en el paper original, introduce modificaciones clave en la arquitectura *U-Net* estándar para lograr una mejor superresolución[3]:

- Conexiones de largo alcance: La *RUNet* incorpora conexiones de largo alcance para mejorar la capacidad de aprendizaje de la red. Estas conexiones permiten que la información fluya más eficientemente a través de la red, lo que facilita la captura de dependencias a larga distancia en la imagen.
- Modelo de degradación espacial: La *RUNet* utiliza un modelo de degradación basado en degradaciones espacialmente variables. Esto obliga a la red a aprender a manejar degradaciones de imagen que no son estacionarias espacialmente, lo que la hace más robusta a diferentes tipos de degradaciones que se pueden encontrar en imágenes reales.
- Bloques residuales: A diferencia de la *U-Net* convencional, la *RUNet* incorpora bloques residuales en la ruta del codificador. Cada bloque residual consiste en una secuencia de capas convolucionales, capas de normalización por lotes y funciones de activación *ReLU*, seguidas de una operación de suma de tensores. Esta operación de suma permite que la entrada del bloque se alimente hacia adelante al siguiente bloque, lo que facilita el aprendizaje de estructuras más complejas y profundas.

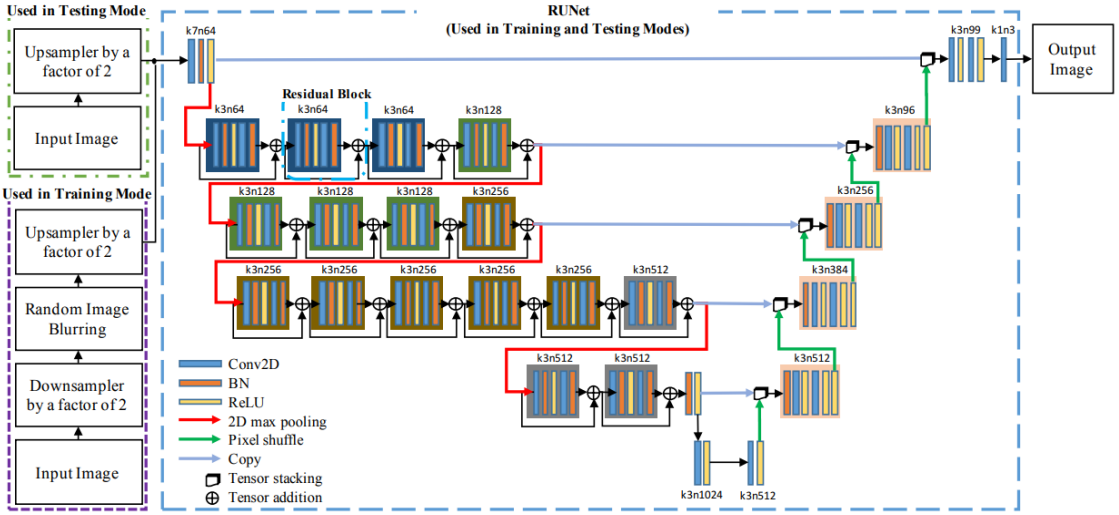

Figura 5 Visualización de la arquitectura Robust UNet (RUNet)

2.3.4 Capas Residuales y Beneficios

Las capas residuales, introducidas por *Kaiming He* en 2015[10], son un tipo de capa que permite el flujo directo del gradiente durante el entrenamiento de redes neuronales profundas. Esto se logra mediante la adición de conexiones de salto que omiten una o más capas.

Las capas residuales ofrecen varios beneficios:

- **Mitigan el problema de la degradación del gradiente:** En redes muy profundas, el gradiente puede desvanecerse o explotar a medida que se propaga hacia atrás, dificultando el entrenamiento. Las capas residuales ayudan a evitar este problema al proporcionar un camino alternativo para el flujo del gradiente.
- **Facilitan el entrenamiento de redes más profundas:** Las capas residuales permiten entrenar redes más profundas sin sufrir una degradación significativa del rendimiento.
- **Mejoran la capacidad de aprendizaje:** Las capas residuales permiten que la red aprenda funciones más complejas.

2.4 Métricas de evaluación

Para poder cuantificar el rendimiento y el desarrollo en modelos de superresolución, es necesario el uso de las métricas de evaluación, las cuales nos permitirán cuantificar el progreso del modelo y la calidad de las imágenes generadas, para posteriormente poder comparar el rendimiento de diferentes modelos. Existen diferentes métricas las cuales proporcionan una medida objetiva de la mejora en la resolución y la fidelidad de las imágenes reconstruidas

2.4.1 PSNR (Peak Signal-to-Noise Ratio).

El *PSNR* mide la relación entre la potencia máxima de una señal y la potencia del ruido que le afecta, en este trabajo, se ha utilizado para evaluar la calidad de imágenes reconstruidas por el modelo de superresolución. Un valor más alto de *PSNR* generalmente indica una mejor calidad de imagen, el cual se calcula con las siguientes fórmulas matemáticas:

$$PSNR = 20\log_{10}\left(\frac{{MAX_I}^2}{\sqrt{MSE}}\right) \quad (1)$$

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \quad (2)$$

Donde *MAX* es el máximo valor posible de la señal. Si se trata de imagen de 8 bits en una escala de grises, el valor de *MAX* = 255. Podemos observar que el *PSNR* es inversamente proporcional al *MSE* (Ec. 1), y el valor final del *PSNR* se da en decibelios [11]

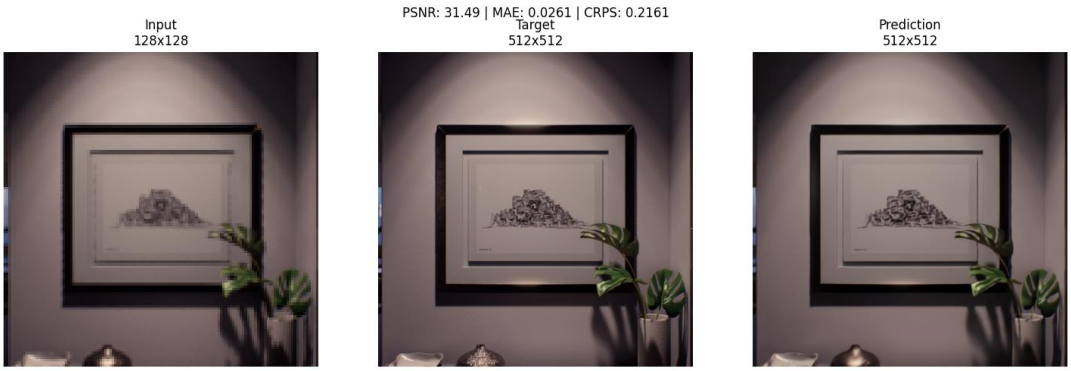

Figura 6 Ejemplo de imagen reconstruida con PSNR 31.49

2.4.2 MSE (Error cuadrático medio).

El *MSE* calcula el promedio del cuadrado de las diferencias entre los píxeles de la imagen generada y la imagen original. Un valor más bajo de *MSE* indica una menor diferencia entre ambas imágenes y, por lo tanto, una mejor calidad.

Su fórmula matemática vista en el apartado 2.4.1 (Ec. 2) es:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \quad (2) \quad [11]$$

Donde:

- $I(i,j)$ es el valor del píxel en la posición (i,j) de la imagen generada
- $X(i,j)$ es el valor del píxel en la posición (i,j) de la imagen original
- m y n son las dimensiones de la imagen (alto y ancho)

2.4.3 MAE (Error absoluto medio).

El *MAE* es similar al *MSE*, solo que calcula el promedio del valor absoluto de las diferencias entre los píxeles, lo cual hace que sea menos sensible a los valores atípicos que el MSE

$$MAE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]$$

2.4.4 Función de perdida SSIM

La función de pérdida *SSIM* (*Structural Similarity Index Measure*) es una métrica que evalúa la similitud entre dos imágenes, considerando la percepción humana. A diferencia de *MSE* o *PSNR*, que se centran en las diferencias pixel a pixel, *SSIM* se basa en la idea de que el sistema visual humano está altamente adaptado para extraer información estructural de la escena visual.

SSIM cuantifica la degradación de la calidad de la imagen como una combinación ponderada de los tres siguientes factores: pérdida de correlación, distorsión de luminancia y distorsión de contraste.

La fórmula matemática para calcular el *SSIM* entre dos imágenes x e y es:

$$SSIM(x,y) = [l(x,y)]^{\alpha} * [c(x,y)]^{\beta} * [s(x,y)]^{\gamma} \quad [12]$$

donde:

- $l(x,y)$ es el componente de luminancia.
- $c(x,y)$ es el componente de contraste.
- $s(x,y)$ es el componente de estructura.
- α, β y γ son parámetros que controlan la importancia relativa de cada componente.

Luminancia

Este componente compara la luminancia (brillo) promedio de las dos imágenes. Se calcula como:

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

donde:

- μ_x y μ_y son las medias de las imágenes x e y, respectivamente.
- C_1 es una constante de estabilización.

La luminancia promedio da una idea general de la diferencia en brillo entre las imágenes. Si las imágenes tienen luminosidades promedio similares, el componente de luminancia tendrá un valor cercano a 1.

Contraste

El componente de contraste compara el contraste de las dos imágenes. Se calcula como:

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

donde:

- σ_x y σ_y son las desviaciones estándar de las imágenes x e y, respectivamente.
- C_2 es una constante de estabilización.

La desviación estándar es una medida de la dispersión de los valores de los píxeles alrededor de la media. Un valor alto de desviación estándar indica un mayor contraste. Si las imágenes tienen contrastes similares, el componente de contraste tendrá un valor cercano a 1.

Estructura

El componente de estructura compara la similitud entre las estructuras de las dos imágenes. Se calcula como:

$$s(x,y) = \frac{\sigma_x y + C_3}{\sigma_x\sigma_y + C_3}$$

donde:

- $\sigma_x y$ es la covarianza entre las imágenes x e y.
- C_3 es una constante de estabilización.

La covarianza mide la relación lineal entre dos variables. En este caso, mide cómo varían los valores de los píxeles en las dos imágenes. Si las imágenes tienen estructuras similares, el componente de estructura tendrá un valor cercano a 1.

Estos componentes se combinan utilizando una función de producto ponderado:

$$SSIM(x,y) = [l(x,y)]^{\alpha} * [c(x,y)]^{\beta} * [s(x,y)]^{\gamma}$$

Si sustituimos las fórmulas de componentes individuales en una ecuación simplificada en la que se asume que $\alpha = \beta$, $\gamma = 1$ obtenemos:

$$SSIM(x,y) = \left[\frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}\right] * \left[\frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}\right] * \left[\frac{\sigma_x y + C_3}{\sigma_x\sigma_y + C_3}\right]$$

Si agrupamos los términos y se asume que C_3 es despreciable en comparación con σ_x σ_y, obtenemos la fórmula final:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_x y + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \; [12]$$

La función de pérdida *SSIM* se puede utilizar en combinación con otras funciones de pérdida, como la *L1 Loss* o la *MSE Loss*, para obtener mejores resultados. En este proyecto, se ha utilizado una función de pérdida híbrida que combina la *L1 Loss* y la *SSIM Loss* en la cual se profundizara más adelante.

3. Metodología

En este capítulo, se describe la metodología empleada para desarrollar el modelo de superresolución. Se detallará el proceso de selección y preparación del conjunto de datos, la arquitectura del modelo, incluyendo las diferentes variantes implementadas, y el proceso de entrenamiento con sus Hiperpárametros y técnicas de optimización.

3.1 Dataset

En este proyecto, la elección del conjunto de datos adecuado ha sido una tarea costosa y necesaria para el éxito del entrenamiento del modelo de superresolución. Inicialmente, se intentó crear un dataset propio a partir de imágenes extraídas del videojuego *Counter-Strike 2*[13]. Para ello, se desarrolló la función *extrac_framesCS.py* incluida en el repositorio *GitHub* del proyecto[1], que se encargaba de procesar videos preprocesados y limpiados del juego para obtener fotogramas individuales. Sin embargo, este enfoque presentó dificultades durante el entrenamiento del modelo, debido a la falta de diversidad y variabilidad en las imágenes y la imposibilidad de ampliar por el coste computacional requerido.
Ante este desafío, se optó por utilizar un dataset público disponible en *Kaggle* [14]. Este conjunto de datos, con un total de 14428 imágenes, se basa en una recopilación de imágenes de diferentes videojuegos creados bajo la arquitectura *Unreal Engine*, este dataset ofrece una mayor amplitud de dominios el cual podría beneficiar la generalización del modelo, y en un futuro, ampliar con más ejemplos sobre el Motor grafico *Unreal Engine*.

Las imágenes del dataset de *Kaggle* se encontraban en una resolución diferente a la requerida para el proyecto (128x128 para baja resolución y 512x512 para alta resolución) debido al coste computacional para procesar imágenes de gran calidad. Para adaptarlas, se utilizó la función *resize_images.py* [1], que reescala las imágenes a las dimensiones necesarias.

Al aplicar esta transformación y reducir la calidad de las imágenes a 128x128, muchas de ellas quedaron con una calidad visual deficiente, lo que podría afectar negativamente el rendimiento del modelo. Para solucionar este problema, se implementó la función *PSNRDel.py* [1], que se encarga de eliminar las imágenes que no alcanzan un valor mínimo de *PSNR* establecido. Este proceso de limpieza, además de asegurar la calidad del dataset, permitió reducir el número de imágenes a 8386, lo cual fue necesario debido a las limitaciones de hardware durante el entrenamiento.

3.2 Modelo

El objetivo de este apartado consiste en explicar en detalle cual ha sido la propuesta como modelo final el cual se encargará de aplicar la superresolución, y posteriormente la reducción de ruido aplicando al modelo entrenado una capa de *Denoising*. Finalmente se compararán diferentes resultados obtenidos comparando las métricas utilizadas en el entrenamiento.

3.2.1 Arquitectura RU-Net

La arquitectura principal implementada se basa en una variante de *U-Net* específicamente adaptada para tareas de superresolución, denominada *RUNet (Robust U-Net).* El modelo sigue una estructura de codificador-decodificador con conexiones residuales y *skip connections*, diseñada para preservar tanto los detalles de alta frecuencia como las características estructurales de las imágenes.

La arquitectura se compone de tres variantes implementadas de complejidad incremental:

1. **SR_Unet**: Implementación base con skip connections tradicionales.
2. **SR_Unet_Residual**: Incorpora bloques residuales para mejorar el flujo de gradientes.
3. **SR_Unet_Residual_Deep**: Versión más profunda con bloques residuales tanto en el decoder como en el encoder.

La arquitectura *SR_Unet_Residual_Deep* representa la versión final del modelo, la cual en nuestro caso conlleva a tener limitaciones debido a su coste computacional, podemos separar el funcionamiento de esta arquitectura en los siguientes componentes:

Capa Inicial de Características

La entrada al modelo comienza con dos capas principales:

```
        self.in_conv1 = FirstFeature(n_channels, 64)
        self.in_conv2 = ResidualBlock(64, 64)
```

La capa FirstFeature realiza la extracción inicial de características:

```
class FirstFeature(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(FirstFeature, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1, 1, 0, bias=False),
            nn.LeakyReLU()
        )

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

En esta capa, se usa una convolución 1x1 para mapear los canales de entrada (RGB) a 64 canales, también se emplea el uso de LeakyReLU[22] como función de activación para evitar problemas con el gradiente en el entrenamiento del modelo.

Bloques de Codificación

La red implementa cinco bloques de codificación con complejidad creciente:

```
        self.enc_1 = ResidualBlock(64, 128, downsample=True)
        self.enc_2 = ResidualBlock(128, 256, downsample=True)
        self.enc_3 = ResidualBlock(256, 512, downsample=True)
        self.enc_4 = ResidualBlock(512, 1024, downsample=True)
        self.enc_5 = ResidualBlock(1024, 2048, downsample=True)
```

En un principio, la red estaba pensada para funcionar con cuatro bloques de codificación, pero tras las primeras pruebas, se añadió una quinta capa para ayudar al modelo con el aprendizaje, conllevando un aumento adicional a la complejidad computacional, limitando nuestra capacidad de aumentar el tamaño del batch a la hora del entrenamiento. Pese a este aumento de la complejidad computacional, se veía una mejora de resultados y en un futuro, si se contase con un *Hardware* diseñado para el entrenamiento de estos modelos, definitivamente sería una mejora considerable.

Cada bloque residual mencionado (*ResidualBlock*) implementa la siguiente estructura:

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample=False):
        super(ResidualBlock, self).__init__()
        stride = 2 if downsample else 1

      #Primera rama convolucional
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, stride=stride,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(inplace=True)
        )

      #Segunda rama convolucional
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, stride=1, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels)
        )

      #Tercera rama convolucional
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_channels)
            )

        self.leaky_relu = nn.LeakyReLU(inplace=True)
```

El funcionamiento de ResidualBlock se divide en tres ramas convolucionales:
En la primera rama Convolucional (self-conv1), se aplica una Convolución 3x3 con stride variable(2 usados en el *downsample*, y 1 para mantener dimensiones), se normaliza en lotes para estabilizar el entrenamiento y se vuelve a usar la función *LeakyReLU* como activación no lineal.

En la segunda rama se vuelve a realizar una Convolución 3x3, pero manteniendo la resolución espacial y el número de canales de entrada y salida son los mismos.

En la tercera rama, se aplica una Conexión residual adaptativa según las dimensiones de entrada o salida, en el caso de ser necesario ajustar dimensiones se aplica una Convolución 1x1 y se vuelve a hacer uso del *BatchNorm* para mantener la consistencia en la normalización.

Bloques de Decodificación

Para la decodificación se sigue un proceso parecido a la codificación, usando bloques residuales especializados

```
        self.dec_0 = DecoderResidual(2048, 1024)
        self.dec_1 = DecoderResidual(1024, 512)
        self.dec_2 = DecoderResidual(512, 256)
        self.dec_3 = DecoderResidual(256, 128)
        self.dec_4 = DecoderResidual(128, 64)
```

En el *DecoderResidual*, se implementa una estructura compleja que combina *upsampling* con procesamiento residual:

```
class DecoderResidual(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderResidual, self).__init__()
       #Rama de upsampling y reducción de canales
       self.conv = nn.Sequential(
            nn.UpsamplingBilinear2d(scale_factor=2),
            nn.Conv2d(in_channels, out_channels, 1, 1, 0, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(),
        )
        self.residual_block = ResidualBlockDeep(out_channels +
out_channels, out_channels)
        self.conv_block = ConvBlock(out_channels + out_channels,
out_channels)
    def forward(self, x, skip):
        x = self.conv(x)
        x_concat = torch.concat([x, skip], dim=1)
        residual = self.residual_block(x_concat)
        x = self.conv_block(x_concat)
        x = x + residual
        return x
```

En el caso del *DecoderResidual*, se utiliza una sola rama de upsampling bilineal que duplica la resolución espacial, luego realiza una convolución 1x1 para reducir canales y hace uso del *BatchNorm* y *LeakyReLU* para normalización y no linealidad.

Una vez aplicado el *upsampling*, procesa las características concatenadas, haciendo uso de las actuales más las proporcionadas por las *skip connection*, después hace uso de una versión más compleja del bloque residual (Solamente disponible en el modelo más avanzado) que mantiene el equilibrio entre capacidad y eficiencia.

Por último, hace uso de un bloque convolucional que procesa de forma paralelo las características y ayuda a completar el procesamiento residual.

Bloque residual profundo

El bloque residual profundo usado en el *DecoderResidual*, es una característica disponible solamente en el modelo más avanzado la cual añade complejidad adicional.

```
class ResidualBlockDeep(nn.Module):
    def __init__(self, in_channels, out_channels, downsample=False):
        super(ResidualBlockDeep, self).__init__()
        stride = 2 if downsample else 1

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, stride=stride,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(inplace=True)
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, stride=1, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, stride=1, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels)
        )

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_channels)
            )

        self.leaky_relu = nn.LeakyReLU(inplace=True)
```

Este bloque residual hace uso de una triple capa convolucional aplicado en la rama principal, hace uso de las activaciones *LeakyReLU* intermedias y normaliza por lotes después de cada convolución, usando siempre la conexión residual para preservar la información.

Capa de Salida

Esta capa se encarga de procesar la imagen a la resolución objetivo

```python
class FinalOutput(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(FinalOutput, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1, 1, 0, bias=False),
            nn.Tanh()
        )

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

Esta capa vuelve a hacer uso de una convolución 1x1 para mapear al espacio RGB, y posteriormente, hace uso de la activación *Tanh* para normalizar de vuelta la salida al rango [-1,1]

3.2.2 Función de Pérdida Personalizada

Se implementó una función de pérdida híbrida que combina dos métricas complementarias:

$$L1SSIMLoss(L1_{weight} = 0.1, SSIM_{weight} = 1.0)$$

Esta función combina:

1. ***L1 Loss***: Mide la diferencia absoluta píxel a píxel
2. ***SSIM Loss***: Evalúa la similitud estructural entre imágenes

La razón de esta combinación es:

- *L1* proporciona convergencia estable y mejora la fidelidad píxel a píxel
- *SSIM* asegura la preservación de estructuras y características perceptuales
- Los pesos (0.1 y 1.0) fueron elegidos para priorizar la calidad perceptual sobre la precisión píxel a píxel tras diferentes pruebas

Podemos observar en la siguiente, imagen, la diferencia entre priorizar *L1* frente *SSIM*, y los pesos usados finalmente para el entrenamiento del modelo

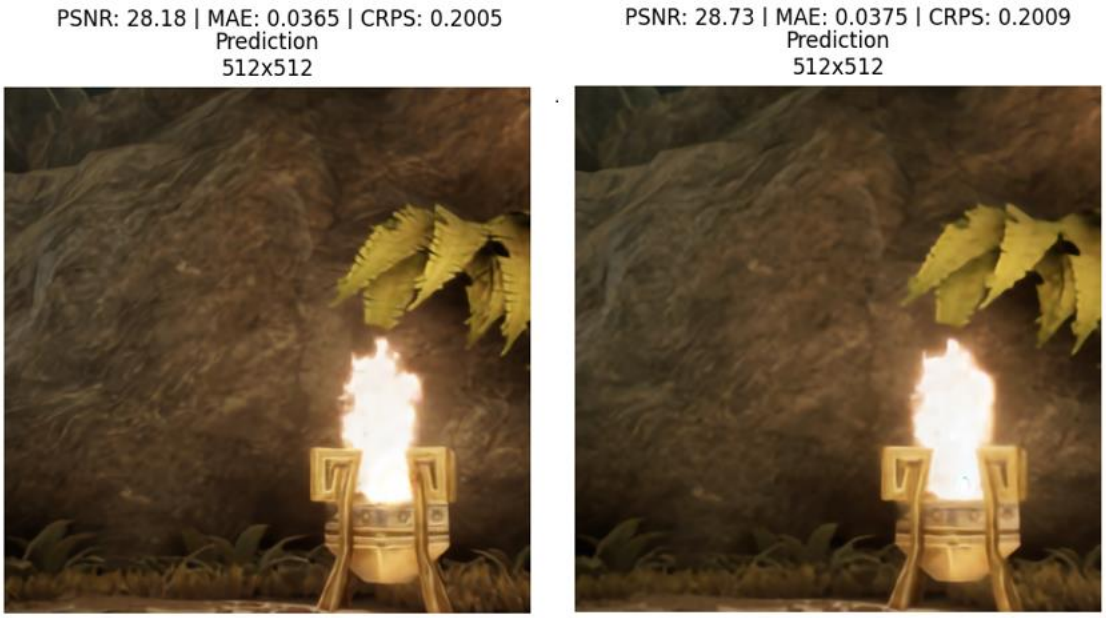

Figura 7 Comparación de prioridad de L1 Loss (2º Imagen) frente a SSIM (1º Imagen)

Podemos observar que pese a priore al priorizar *L1* frente a *SSIM* obtenemos una mayor puntuación en *PSNR*, al priorizar *SSIM* obtenemos mejores resultados dado que *SSIM* asegura la preservación de estructuras, un apartado muy importante a la hora de la superresolución en videojuegos.

3.3 Posprocesamiento con Denoising

Para mejorar aún más la calidad de las imágenes generadas por el modelo de superresolución, se ha añadido una etapa de *denoising* como posprocesamiento. El objetivo es eliminar el ruido presente en las imágenes de salida del modelo, lo que proporcionara una mejora en la nitidez visual y en la calidad general de la imagen.

Para llevar a cabo el *denoising*, se ha implementado un modelo de red neuronal que se adjunta al modelo de superresolución ya entrenado. Este modelo de denoising se basa en una arquitectura que combina la estimación de ruido adaptativa con la preservación de bordes.

3.3.1 Estimación del ruido

Primero, obtenemos una estimación de ruido usando una función adaptativa

```
class AdaptiveNoiseEstimator(nn.Module):
    """Estimador de ruido adaptativo"""
    def __init__(self, channels=3):
        super(AdaptiveNoiseEstimator, self).__init__()
        self.noise_estimation = nn.Sequential(
            nn.Conv2d(channels, 32, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
```

```
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(32, 1, kernel_size=1),
            nn.Sigmoid()
        )

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

Esta clase recibe una imagen como entrada (x) y produce un "mapa de ruido" que indica la intensidad del ruido en diferentes áreas de la imagen.

Primero se realiza convoluciones sobre la imagen de entrada. Se utilizan múltiples capas convolucionales con diferentes números de canales y tamaños de *kernel* para extraer características relevantes de la imagen.
Se hace uso de la función *LeakyReLU* para introducir la no linealidad en el modelo, permitiendo aprender patrones más complejos.
Por último, se escala la salida del modelo a un rango entre 0 y 1, representando la probabilidad de que un píxel contenga ruido.

3.3.2 Preservar los bordes

Para asegurarnos que al aplicar nuestras imágenes los bordes no sufran alteraciones, usamos un *denoiser* que detecta los bordes y se asegura que no son modificados.

```
class EdgePreservingDenoiser(nn.Module):
    """Denoiser que preserva bordes"""
    def __init__(self, channels=3):
        super(EdgePreservingDenoiser, self).__init__()

        self.edge_detector = nn.Sequential(
            nn.Conv2d(channels, 32, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(32, 1, kernel_size=3, padding=1),
            nn.Sigmoid()
        )

        self.smooth_branch = nn.Sequential(
            nn.Conv2d(channels, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, channels, kernel_size=3, padding=1)
        )

        self.detail_branch = nn.Sequential(
            nn.Conv2d(channels, 64, kernel_size=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, channels, kernel_size=1)
        )
```

```
    def forward(self, x):
        edges = self.edge_detector(x)
        smooth = self.smooth_branch(x)
        details = self.detail_branch(x)
        return edges * details + (1 - edges) * smooth
```

Se detectan los bordes en la imagen utilizando capas convolucionales, *LeakyReLU* y una función *Sigmoid*[15] para generar un mapa de bordes, se suaviza las regiones de la imagen que no son bordes utilizando capas convolucionales y *LeakyReLU* y por último se preserva los detalles de los bordes.

En el *forward*, se combinan las salidas de estas tres ramas: multiplica el mapa de bordes (*edges*) por los detalles (*details*) y lo suma al resultado de multiplicar el inverso del mapa de bordes (1 - *edges*) por la imagen suavizada (*smooth*). Esto permite eliminar el ruido en las zonas suaves mientras se mantienen los detalles en los bordes.

3.3.3 Denoising

Esta clase se encarga de combinas las dos clases anteriores (*AdaptiveNoiseEstimator* y *EdgePreservingDenoiser*) para crear el modelo de *denoising* completo.

```
class PostProcessDenoiser(nn.Module):
    def __init__(self, channels=3):
        super(PostProcessDenoiser, self).__init__()

        self.noise_estimator = AdaptiveNoiseEstimator(channels)
        self.edge_denoiser = EdgePreservingDenoiser(channels)

        self.refinement = nn.Sequential(
            nn.Conv2d(channels*2, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, channels, kernel_size=3, padding=1)
        )

        self.residual_weight = nn.Parameter(torch.tensor([0.1]))

    def forward(self, x):
        noise_map = self.noise_estimator(x)
        denoised = self.edge_denoiser(x)
        concat_features = torch.cat([x, denoised], dim=1)
        refined = self.refinement(concat_features)
        return x + self.residual_weight * refined
```

Se crean las instancias de las clases *AdaptiveNoiseEstimator* y *EdgePreservingDenoiser*. Además, se define una capa de refinamiento que consiste en capas convolucionales y

funciones *LeakyReLU* y posteriormente se define un parámetro que controla la influencia de la imagen refinada en la salida final.

Se estima el ruido de la imagen de entrada x utilizando la instancia de *AdaptiveNoiseEstimatory* se aplica el *denoising* que preserva bordes a la imagen x utilizando la instancia de *EdgePreservingDenoiser*.
Se concatenan la imagen original (x) y la imagen *denoised* para crear un tensor con el doble de canales. Esto permite que la capa de refinamiento tenga en cuenta la información de ambas imágenes.
Se aplica la capa de refinamiento a las características concatenadas y se combina la imagen original (x) con la imagen refinada utilizando un peso residual (*self.residual_weight*). Esto permite que el modelo aprenda a ajustar la cantidad de refinamiento que se aplica a la imagen.

3.4 Proceso de Entrenamiento

3.4.1 Hiperpárametros

Los principales hiperparámetros utilizados en el entrenamiento fueron:

```
epochs = 100
batch_size = 2
learning_rate = 0.0001
optimizer = optim.AdamW
loss_function = L1SSIMLoss(l1_weight=0.1, ssim_weight=1)
```

Justificación de la selección:

- **Epochs**: 100 *epochs* proporcionan suficiente tiempo para la convergencia sin sobreajuste.
- **Batch Size**: 2 debido a limitaciones de memoria *GPU* y para mejor generalización, en el caso del entrenamiento más básico del modelo ***SR_Unet*,** este valor se puede ampliar hasta 4.
- **Learning Rate**: 0.0001 permite un aprendizaje estable sin divergencia el cual se regula mediante el Scheduler *ReduceLROnPlateau*
- **Optimizer**: *AdamW* por su capacidad de adaptación y regularización implícita

3.4.2 Scheduler de Tasa de Aprendizaje

Se implementó un *Scheduler ReduceLROnPlateau*[20], Este *Scheduler* monitoriza una métrica específica, en este caso la pérdida en el conjunto de validación, y reduce la tasa de aprendizaje cuando la métrica deja de mejorar.

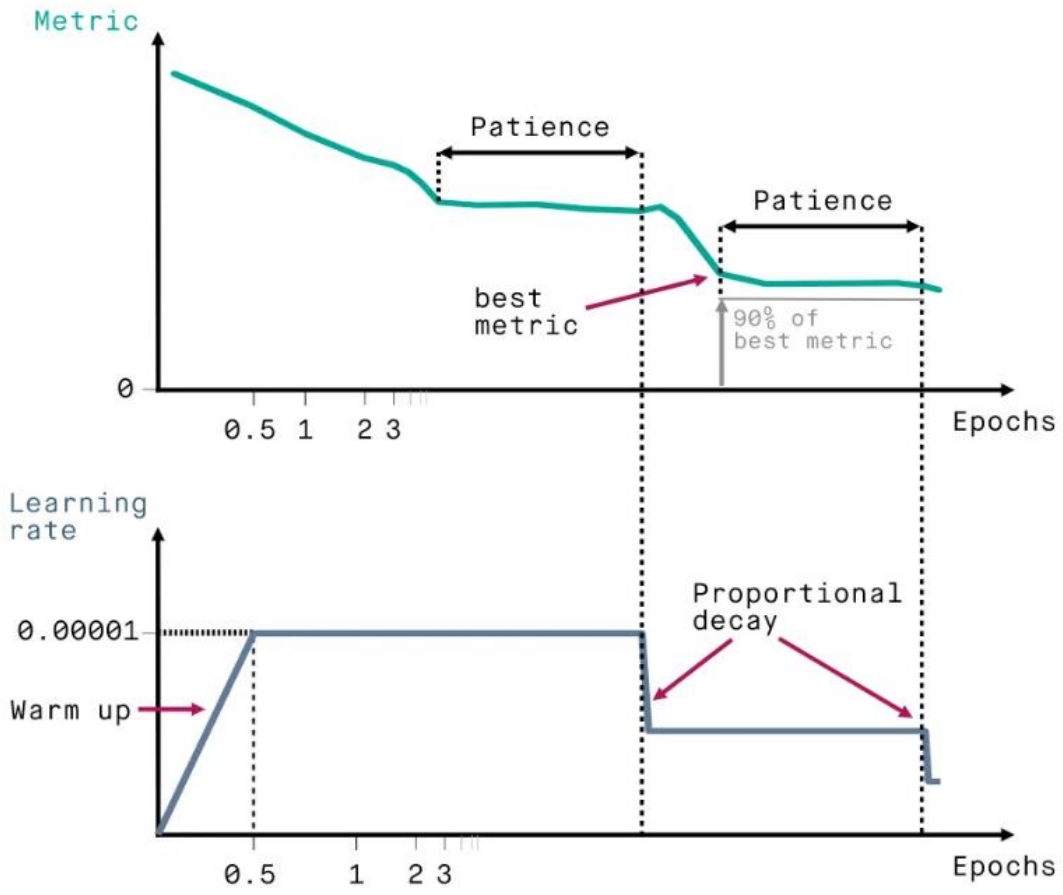

Figura 8 Funcionamiento del Scheduler ReduceLROnPlateau según los parámetros

La Figura 6 ilustra el funcionamiento del *ReduceLROnPlateau*. En la gráfica superior, se observa la evolución de la métrica monitorizada (pérdida de validación) a lo largo de los epochs. El *Scheduler* "espera" un número de *epochs* definido por el parámetro patience (en este caso, 2) para comprobar si la métrica mejora. Si no hay mejora, la tasa de aprendizaje se reduce (gráfica inferior).

En nuestro caso, la implementación se realizó con los siguientes parámetros:

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer,
            mode='min',
            factor=0.3,
            patience=4,
            min_lr=1e-6
        )
```

Con estos parámetros, el *Scheduler*:

- Reduce el *learning rate* cuando la pérdida se estanca
- Factor de reducción de un 30% para ajustes graduales
- Paciencia de 4 *epochs* para reducir el *LR* una vez veamos que el modelo no consigue seguir aprendiendo.
- *LR* mínimo de 1e-6 para mantener cierta capacidad de aprendizaje

3.4.3 Preprocesamiento de Datos

El *dataset* implementa varias técnicas de preprocesamiento:

1. **Normalización**:

```python
    def normalize_basic(self, input_image, target_image):
        """
        Normalización básica al rango [-1, 1]
        """

        input_image = transforms.functional.to_tensor(input_image)
        target_image = transforms.functional.to_tensor(target_image)

        input_image  = input_image*2 - 1
        target_image = target_image*2 - 1

        return input_image, target_image
```

Las imágenes se normalizan al rango [-1,1] para mejorar la convergencia del modelo durante el entrenamiento. Se valoro el implementar una normalización *mín-máx*[21], incluida en el repositorio *GitHub* del proyecto[1] en el archivo *normalization.py*, la cual fue descartada tras no obtener los resultados esperados frente a una normalización básica.

2. **Data Augmentation**:
    - Volteo horizontal (probabilidad 0.5)
    - Rotación aleatoria (-30° a 30°, probabilidad 0.3)

4. Resultados

En esta sección se discuten los diferentes resultados obtenidos durante el entrenamiento y la evaluación del modelo de superresolución. Primero compararemos el entrenamiento entre los diferentes modelos utilizados usando las principales métricas, luego analizaremos los resultados sobre 50 imágenes de las utilizadas para la evaluación, explicando el porqué de algunos valores atípicos dependiendo del dominio de la imagen.

4.1 Análisis del entrenamiento

A lo largo del entrenamiento, se monitorearon dos métricas principales para evaluar en rendimiento del modelo, el ***PSNR (Peak Signal-to-Noise Ratio)***, que nos ayuda a valorar la calidad de la imagen generada, y el ***MAE (Mean Absolute Error)***, que compara el error absoluto medio entre las imágenes generadas y las imágenes originales en alta resolución

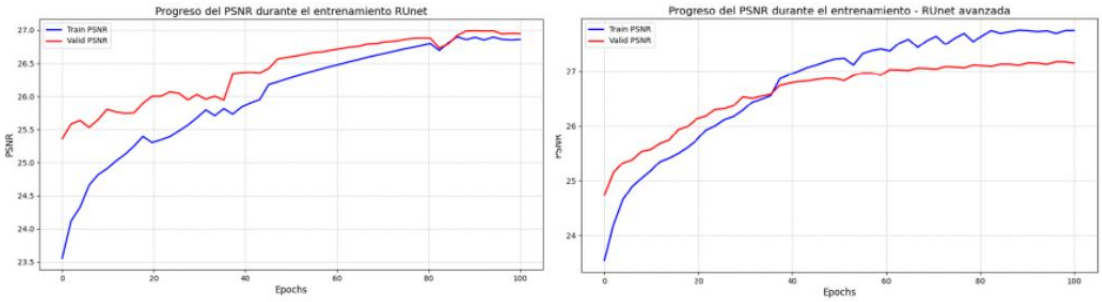

Figura 9 Progreso del entrenamiento de ambos modelos por la métrica *PSNR*

Como se puede observar en la Figura 7, que muestra el progreso del entrenamiento de ambos modelos (*RUnet* y *RUnet* avanzada) a través de la métrica *PSNR*, ambos modelos muestran una tendencia general de mejora en la calidad de la imagen generada a medida que avanza el entrenamiento, dejando de mejorar en ambos casos sobre el epoch 90.

Podemos observar diferencias en ambos entrenamientos, en el entrenamiento de la *RUnet* inicial, se muestra un entrenamiento mas gradual y estable llegando a converger en valores menores de *PSNR*.

Mientras que en el entrenamiento de la *RUnet* avanzada, podemos observar un entrenamiento más rápido del *PSNR*, mostrando fluctuaciones mas pronunciadas debido a la mayor complejidad del modelo y la activación del *Scheduler ReduceLROnPlateau*, que se encarga de ajustar automáticamente la tasa de aprendizaje.

Podemos observar en la siguiente figura, la diferencia visual sobre el rendimiento de ambos modelos sobre la misma imagen, obteniendo una mejor definición en el modelo avanzado gracias a el uso de las capas residuales.

Figura 10 Comparación visual sobre el rendimiento de modelos

Si comparamos la otra métrica principal usada para evaluar el rendimiento del entrenamiento, podemos observar que, debido a la complejidad del modelo, en el caso de la *RUnet* avanzada los valores iniciales son inferiores respecto a su modelo más básico, pero según avanza el entrenamiento la tendencia es favorable y termina con una mejoría notable respecto al modelo inferior.

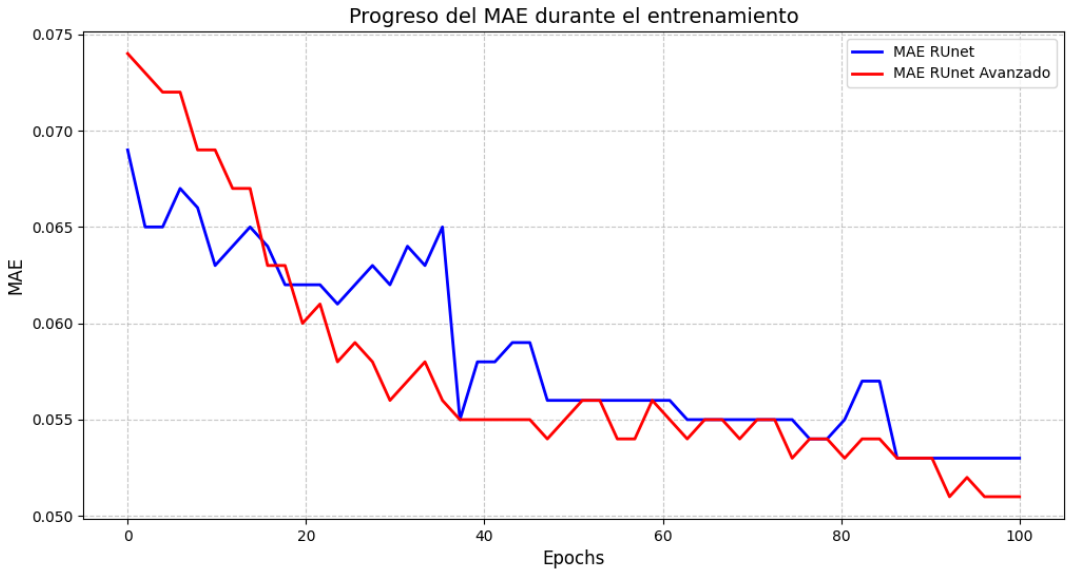

Figura 11 Progreso del entrenamiento de ambos modelos respecto al MAE

4.2 Comparativa visual

Para evaluar de forma visual el rendimiento del modelo final ya entrenado, se han seleccionado una serie de imágenes de prueba del conjunto de datos usadas para la evaluación del modelo, con el objetivo de comparar visualmente la imagen original en baja resolución, con la imagen generada por el modelo. Para cada ejemplo, se incluye información sobre el *PSNR* y el *MAE* obtenidos.

Imagen	PSNR	MAE	Imagen	PSNR	MAE
1	28.36	0.0477	26	26.55	0.0551
2	30.08	0.0390	27	36.54	0.0156
3	28.18	0.0365	28	27.34	0.0323
4	28.68	0.0429	29	33.91	0.0193
5	28.07	0.0493	30	29.36	0.0422
6	29.32	0.0446	31	23.80	0.0753
7	26.23	0.0540	32	26.63	0.0569
8	36.62	0.0164	33	26.86	0.0604
9	23.22	0.0974	34	23.35	0.0915
10	27.33	0.0497	35	29.58	0.0437
11	31.76	0.0295	36	24.42	0.0899
12	31.76	0.0259	37	29.41	0.0386
13	28.11	0.0387	3	25.12	0.0582
14	27.37	0.0557	3	26.95	0.0508
15	26.39	0.0594	40	27.95	0.0511
16	30.56	0.0405	41	29.90	0.0361
17	23.72	0.0754	42	30.27	0.0432
18	28.07	0.0444	43	27.06	0.0572
19	24.61	0.0797	44	28.47	0.0375
20	27.34	0.0516	45	30.75	0.0350
21	31.43	0.0300	46	25.53	0.0582
22	28.99	0.0414	47	27.75	0.0471
23	26.49	0.0629	48	29.07	0.0473
24	30.54	0.0267	49	25.55	0.0680
25	24.97	0.0635	50	30.69	0.0317

Analizando la tabla de comparativa visual, podemos observar una disparidad en los valores de *PSNR* obtenidos para diferentes imágenes. Esta variabilidad se debe principalmente a dos factores:

1. **Complejidad de la escena:** Imágenes con escenas más simples y menos detalles tienden a obtener valores de *PSNR* más altos. Esto se debe a que el modelo tiene menos información que reconstruir, lo que facilita la tarea de superresolución y reduce la posibilidad de errores.

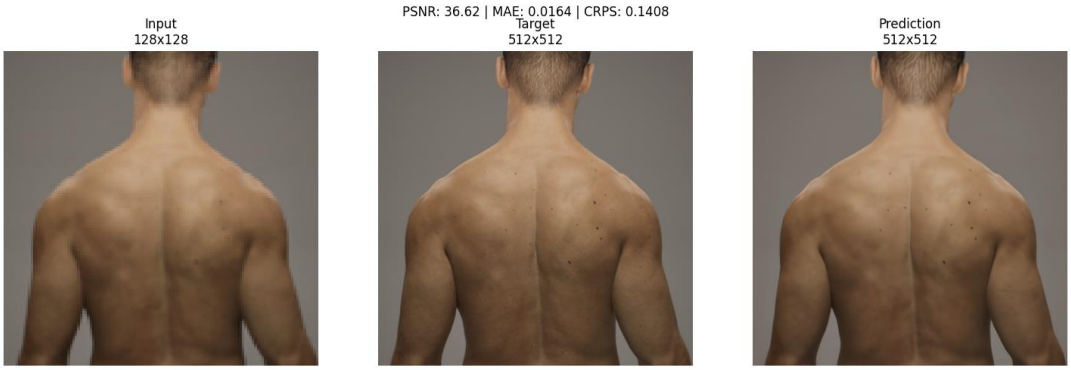

Figura 12 Resultados del modelo sobre una imagen con complejidad baja

Por otro lado, imágenes con escenas complejas, llenas de detalles y texturas, tienden a obtener valores de *PSNR* más bajos. Esto se debe a que el modelo tiene que reconstruir una mayor cantidad de información, lo que aumenta la dificultad de la tarea y relucen las limitaciones computacionales de nuestro modelo actual (Posiblemente, debido al uso de tan solo 5 capas o un batch size limitado)

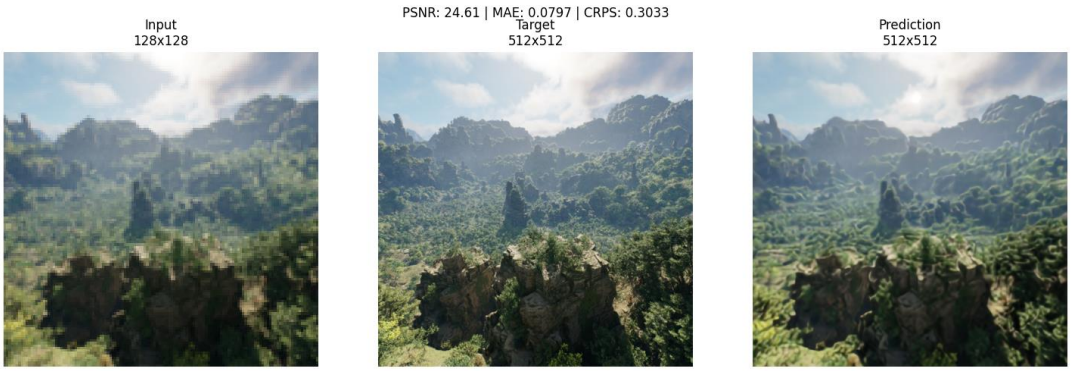

Figura 13 Resultados del modelo sobre una imagen con complejidad elevada

2. **Dominio de la imagen:** El dataset utilizado contiene imágenes de diferentes videojuegos, lo que implica una variedad de estilos artísticos, paletas de colores y tipos de escenas. El modelo puede tener un rendimiento diferente en función del dominio de la imagen, lo que se refleja en los valores de PSNR.

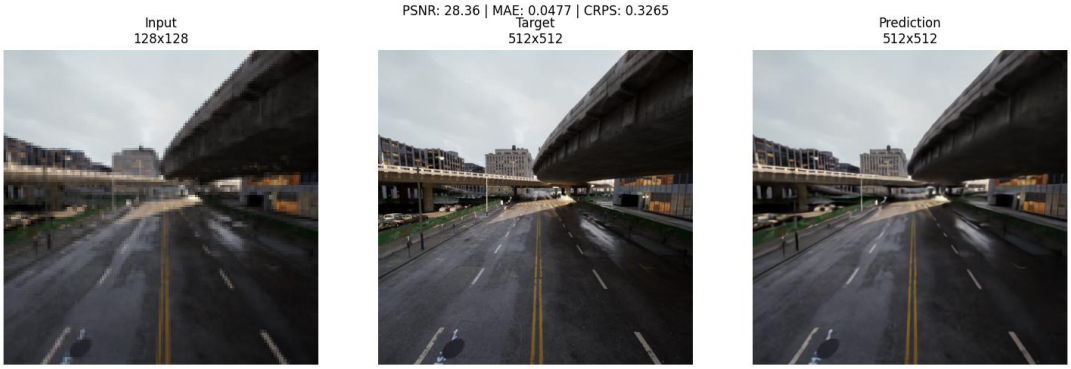

Figura 14 Resultados del modelo sobre una imagen del dominio dominante.

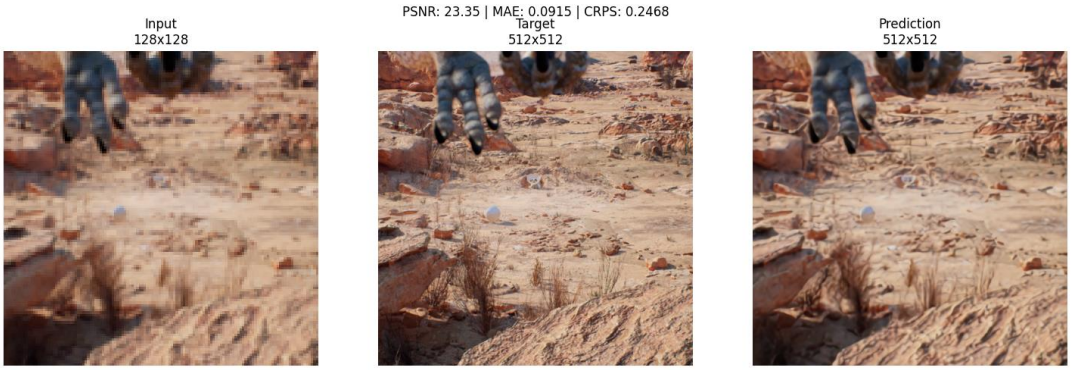

Figura 15 Resultados del modelo sobre una imagen del dominio no dominante.

En general, se observa que el modelo tiende a obtener mejores resultados en imágenes con escenas simples y pertenecientes a dominios en los que ha sido entrenado con mayor cantidad de datos. Por el contrario, el modelo puede tener dificultades para reconstruir imágenes con escenas complejas o pertenecientes a dominios con poca representación en el conjunto de datos de entrenamiento.

5. Conclusiones

Este capítulo presenta las conclusiones extraídas del desarrollo del Trabajo de Fin de Grado, incluyendo una reflexión crítica sobre el proceso, los resultados obtenidos y las limitaciones encontradas.

5.1 Descripción de las conclusiones del trabajo

A lo largo del desarrollo de este proyecto, se han aprendido diversas lecciones, tanto a nivel técnico como de gestión de proyectos:

- **Importancia de la planificación:** La planificación inicial, aunque modificada posteriormente, fue importante para establecer una hoja de ruta y definir los objetivos. Sin embargo, este proyecto ha resaltado la necesidad de una mejor planificación que anticipe posibles dificultades y la necesidad de recursos.
- **Limitaciones del hardware:** El *hardware* disponible ha sido una limitación significativa. La capacidad de procesamiento restringió la complejidad del modelo, el tamaño del *batch size* durante el entrenamiento y la resolución de las imágenes. Esto condicionó los resultados y el alcance del proyecto.
- **Adaptación y flexibilidad:** A pesar de las dificultades, se logró adaptar el proyecto hacia una solución viable. El propuesto inicial con *Counter-Strike 2*, aunque no termino siendo viable, proporcionó un aprendizaje valioso y permitió reenfocar el trabajo hacia un objetivo más alcanzable.
- **Complejidad del desarrollo:** El desarrollo de modelos de superresolución basados en aprendizaje profundo es una tarea compleja que requiere un conocimiento sólido de los fundamentos teóricos y las herramientas de desarrollo.

5.2 Reflexión crítica sobre el logro de los objetivos

Si bien no se han logrado todos los objetivos planteados inicialmente, se puede considerar que el proyecto ha sido exitoso en su adaptación y en la consecución de un modelo funcional de superresolución.

La idea inicial de mejorar la resolución de vídeos de *Counter-Strike 2* tuvo que ser abandonada debido a la complejidad del problema y las limitaciones del hardware. Sin embargo, se logró pivotar hacia un objetivo más realista: la superresolución de imágenes de videojuegos a baja resolución en un *dataset* preestablecido.

En este sentido, se cumplieron los objetivos de implementar una arquitectura *Robust U-Net*, incorporar capas residuales, optimizar la función de activación y aplicar técnicas de *denoising*.

No obstante, la limitación del hardware impidió explorar modelos más complejos y entrenar con un *batch size* mayor, lo que podría haber mejorado los resultados.

5.3 Análisis crítico de la planificación y metodología

La planificación inicial tuvo que ser modificada debido a las dificultades encontradas, principalmente la limitación del hardware y la complejidad del problema inicial.
El cambio de enfoque hacia un objetivo más realista implicó una reorganización de las tareas y la adaptación de la metodología.
A pesar de estos cambios, la metodología general se mantuvo: se investigaron las técnicas de superresolución, se implementó el modelo en *PyTorch*, se entrenó con un *dataset* de imágenes de videojuegos y se evaluó su rendimiento.

5.4 Líneas de trabajo futuro

Este proyecto abre diversas líneas de trabajo futuro que podrían explorarse con mayores recursos y tiempo:

- **Entrenamiento con mayor *batch size*:** Utilizar un *hardware* más potente permitiría entrenar el modelo con un *batch size* mayor, lo que podría mejorar la generalización y el rendimiento del modelo.
- **Aumentar la complejidad del modelo:** Se podrían explorar arquitecturas más complejas con un mayor número de capas y conexiones, para mejorar la capacidad del modelo de capturar detalles y generar imágenes de mayor calidad.
- **Utilizar una resolución mayor:** Trabajar con imágenes de mayor resolución (256x256 o superior) permitiría capturar más detalles y obtener mejores resultados en la superresolución.
- **Experimentar con diferentes optimizadores:** Se podrían realizar pruebas con diferentes optimizadores, como *Adam* o *RMSprop*, para encontrar el que mejor se adapte al modelo y al dataset.
- **Probar más variaciones de la función de pérdida:** Se podrían explorar diferentes combinaciones de *L1 Loss* y *SSIM Loss*, o incluso otras funciones de pérdida, para optimizar el entrenamiento y la calidad de las imágenes generadas.
- **Aplicar el modelo a diferentes tipos de imágenes:** Se podría evaluar el rendimiento del modelo en imágenes de diferentes géneros de videojuegos, o incluso en imágenes reales.

6. Glosario

HR: Alta resolución**.**

LR: Baja resolución.

CNN: Redes neuronales convolucionales.

Skip-connections: Conexiones en una red neuronal que saltan una o más capas, permitiendo que la información fluya más libremente.

Hiperpárametros: Parámetros que controlan el proceso de entrenamiento de una red neuronal, como la tasa de aprendizaje o el tamaño del batch.

Codificador: La parte de una red neuronal que reduce la dimensionalidad de la entrada.

Decodificador: La parte de una red neuronal que aumenta la dimensionalidad de la entrada.

Capas Residuales: Capas que permiten el flujo directo del gradiente durante el entrenamiento, mediante conexiones de salto.

Gradiente: Vector que indica la dirección de mayor ascenso en una función. Se utiliza para actualizar los pesos de la red neuronal.

Dataset: Conjunto de datos utilizado para entrenar la red neuronal.

API: Interfaz de programación de aplicaciones**.** Conjunto de reglas que definen cómo interactúan los programas.

GPU: Unidad de procesamiento gráfico. Hardware especializado para acelerar los cálculos.

Blurring: Desenfoque. Efecto visual que reduce la nitidez.

Downsample: Reducción de la resolución espacial.

Upsampling: Aumento de la resolución espacial.

Convolución: La convolución es una operación matemática que se utiliza en el procesamiento de imágenes para aplicar filtros. Un filtro es una pequeña matriz de números que se desliza sobre la imagen, realizando operaciones matemáticas con los

píxeles de la entrada. Los filtros pueden detectar diferentes características en la imagen, como bordes, esquinas o texturas.

Capas Convolucionales: Las capas convolucionales son el componente principal de las redes neuronales convolucionales (*CNN*). Estas capas aplican múltiples filtros a la imagen de entrada para extraer características relevantes.

Función de Activación LeakyReLU: La función LeakyReLU es una variante de la ReLU que mitiga el problema de las "neuronas muertas". Esto se logra al permitir una pequeña pendiente para valores negativos. Su fórmula matemática es:

$$LeakyReLU(x) = \max(0,x) + \alpha * \min(0,x)$$

Epoch: Un epoch es una pasada completa del conjunto de datos de entrenamiento a través de la red neuronal.

Batch Size: El batch size es el número de muestras de entrenamiento que se utilizan en cada iteración del entrenamiento.

Learning Rate: El learning rate es un hiperparámetro que controla la velocidad a la que la red neuronal aprende.

Optimizer: Un *optimizer* es un algoritmo que se encarga de ajustar los pesos de la red neuronal durante el entrenamiento para minimizar la función de pérdida buscando la configuración ideal que se adapte al entrenamiento.

7. Bibliografía

[1] Repositorio GitHub del proyecto - https://github.com/Alexherrland/app_unet

[2] Olaf Ronneberger, Philipp Fischer, Thomas Brox - U-Net: Convolutional Networks for Biomedical Image Segmentation

[3] Xiaodan Hu, Mohamed A. Naiel, Alexander Wong, Mark Lamm, Paul Fieguth, Vision and Image Processing Lab, University of Waterloo, Waterloo, ON, Canada 2Christie Digital Systems Canada Inc., Kitchener, ON, Canada - RUNet: A Robust UNet Architecture for Image Super-Resolution

[4] Descarga de Visual Studio Code [Citada 08/01/2025] - https://code.visualstudio.com/download

[5] Descarga de Python [Citada 08/01/2025] - https://www.python.org

[6] Guía sobre el uso básico de Pytorch [Citada 08/01/2025] - https://pytorch.org

[7]. R. Y. Tsai and T. S. Huang, "Multiframe image restoration and registration," in Advances in Computer Vision and Image Processing, vol. 1, chapter 7, pp. 317-339, JAI Press, Greenwich, Conn, USA, 1984.

[8]. M. Irani and S. Peleg. 1991, "Super Resolution From Image Sequences" ICPR, 2:115--120, June 1990.

[9] Kassem Al Ismaeil, Djamila Aouada, Bruno Mirbach, and Bjorn Ottersten - Depth Super-Resolution by Enhanced Shift & Add

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep Residual Learning for Image Recognition *10 Dec 2015*

[11] Fernando A. Fardo, Victor H. Conforto, Francisco C. de Oliveira, Paulo S. Rodrigues - A Formal Evaluation of PSNR as Quality Measurement Parameter for Image Segmentation Algorithms

[12] Jim Nilsson, Tomas Akenine-Möller - Understanding SSIM

[13] Información básica sobre *Counter-Strike 2* [Citada 08/01/2025] - *https://www.counter-strike.net/cs2*

[14] Dataset utilizado en el proyecto [Citada 08/01/2025] - *https://www.kaggle.com/competitions/super-resolution-in-video-games/data*

[15] Función Sigmoide [Citada 08/01/2025] - *https://es.wikipedia.org/wiki/Función_sigmoide*

[16] Michael Elad, Bahjat Kawar, Gregory Vaksman, Image Denoising: The Deep Learning Revolution and Beyond -- A Survey Paper –

[17] Transformada de Fourier [Citada 08/01/2025] - https://es.wikipedia.org/wiki/Transformada_de_Fourier

[18] Nvidia DLSS [Citada 08/01/2025] - https://www.nvidia.com/es-la/geforce/technologies/dlss/

[20] Funcionamiento de ReduceLROnPlateau métricas recomendadas [Citada 08/01/2025] - https://wiki.cloudfactory.com/docs/mp-wiki/scheduler/reducelronplateau

[21] Normalización mín-máx [Citada 08/01/2025] https://es.statisticseasily.com/glossario/what-is-min-max-normalization-explained/

[22] Explicación de Leaky ReLU [Citada 08/01/2025] https://paperswithcode.com/method/leaky-relu

[23] Función de activación Tanh [Citada 08/01/2025] https://www.datacamp.com/es/tutorial/introduction-to-activation-functions-in-neural-networks