UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

# Uso de autoencoders en la compresión de imágenes

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Rodríguez Ferrero, Ignacio

Tutor/a: Paredes Palacios, Roberto

CURSO ACADÉMICO: 2021/2022

# Resum

S'estudia la viabilitat del ús d'*autoencoders* en la compressió d'imatges en escenaris de vida real fent ús d'un metod de divisió per cuadrícula de l'imatge original. Es construeixen diversos models, utilizant diferents tècniques d'entrenament, i es compara la seua qualitat de la reconstrucció amb l'obtinguda per altres formats de compressió amb pèrdua establits en la indústria, com *JPEG*, en la seua sumititud amb la imatge original.

Per a això, es desenrotllen diversos compressors basats en *autoencoders* amb el seus corresponents descompressors. La validació es fa amb 41 imatges, que es comprimixen i reconstruixen, i de les quals s'extraixen les métriques de similitud amb les imatges originals *mse*, *psnr* i *ssim*. L'objectiu és optimitzar les mètriques de similitud obtingudes pel compressor desenvolupat i comparar-les amb les mètriques obtingudes per altres formats de compressió establits en la indústria.

**Paraules clau:** compressió d'imatges, decompressió, xarxa neuronal, autoencoder

# Resumen

Se estudia la viabilidad del uso de *autoencoders* en la compresión de imágenes para escenarios de la vida real usando un método de división por cuadrícula de la imagen original. Se construyen varios modelos, utilizando diferentes técnicas de entrenamiento, y se compara su calidad de reconstrucción con obtenida por otros formatos de compresión con pérdida establecidos en la industria, como *JPEG*, en su similitud con la imagen original.

Para ello, se desarrollan compresores basados en un *autoencoder* con su correspondientes descompresores. La validación se hace con 41 imágenes de una base de datos de imágenes públicas, las cuales se comprimen y reconstruyen, para entonces extraer las métricas de similitud con las imágenes originales *mse*, *psnr* y *ssim*. El objetivo es optimizar las métricas obtenidas por el compresor desarrollado y comprarlas con las métricas obtenidas por otros formatos de compresión establecidos en la industria.

**Palabras clave:** compresión de imagen; descompresión; red neuronal; autoencoder

# Abstract

In this work the main focus will be to study viability of the use of autencoders in the task of image compression, making use of a grid division of the image. The performance of the autoencoder-based compressor will be evaluated and compared with other industry-stablished formats, such as *JPEG*, designing neuronal network models and training them with various techniques.

To achieve this an autoencoder-based compressor is developed, with its corresponding decompressor. The validation of its performance is performed with 41 images with different content types, all of them retrieved from a public image database, with which the reconstrucction fidelity is evaluated using the *mse*, *psnr* and *ssim* metrics. The objective is to optimize optimize these metrics for the autoencoder-based compressor, while keeping the compression ratio as low as possible. Then, the results are compared with the results produced by industry-stablished image compression formats.

**Key words:** image compression, image decompression, neuronal net, autoencoder

# Índice general

# Índice de figuras

# Índice de tablas

# CAPÍTULO 1
# Introducción

En esta sección se explican los diferentes motivos que motivan el desarrollo del trabajo, los objetivos que finalmente se pretenden conseguir y cuál es la estructura de la memoria en su conjunto.

## 1.1 Motivación

La compresión de imágenes es una de las tareas más importantes en la actualidad. Tanto para el almacenamiento de estas como en su transmisión por internet, es necesario encontrar formas nuevas de comprimir esta forma de datos de formas más eficientes mientras que la pérdida de información se mantenga al mínimo.

Por ejemplo, en un día cualquiera, se suben a la plataforma *Youtube* 500 horas de vídeo al segundo y todos esos datos toman espacio en uno de los 23 centros de datos que a fecha de mayo de 2022 tiene Google repartidos por el mundo [13]. Por otro lado, uno de los elementos que más espacio ocupa en los dispositivos móviles son las imágenes, que suelen ser lo primero que recurrimos a eliminar cuando nos quedamos sin espacio de almacenamiento.

En general, almacenar y transmitir imágenes es muy caro y reducir su tamaño mediante métodos de compresión es una forma en la que podemos reducir los costes. Los algoritmos de compresión de imágenes actuales (*JPEG*, *JPEG2000*) se apoyan en el uso de algoritmos prefabricados. Estos usan transformaciones predeterminadas y elegidas a mano, como Transformadas Discretas de Coseno, *Wavelet Transform*, cuantización y codificadores de entropía. Sin embargo, no están diseñadas para ser una solución general y no se amoldan a todos los tipos de contenido o formato. El aprendizaje profundo (*Deep Learning*) se ha aplicado en muchos problemas de la visión por computador y es posible que tenga el potencial de mejorar la eficiencia de la compresión de imágenes, gracias a su flexibilidad y su capacidad de adaptarse a casos específicos.

## 1.2 Objetivos

Este trabajo pretende, como principal objetivo, estudiar la viabilidad del uso de los *autoencoders* como motor en un compresor de imágenes. Como objetivos concretos para llevar a cabo el objetivo principal se determinan los siguientes pasos:

1. Familiarizarse con librerías de desarrollo de redes neuronales, el estado de la cuestión actual y los métodos usados para el desarrollo de *autoencoders*.

2. Construir un compresor de imágenes basado en un *autoencoder* capaz de diferentes niveles de compresión. Al final del trabajo se dispondrán de diferentes compresores de imágenes, cada uno de ellos basado en un *autoencoder* o que hacen uso de uno, capaces de comprimir imágenes con diferentes porcentajes de compresión.
3. Medir la calidad de reconstrucción obtenida por cada uno de los niveles de compresión usando las métricas *MSE*, *PSNR* y *SSIM*.
4. Comparar los resultados con los obtenidos por *JPEG*, determinando así si el uso de *autoencoders* como motor para los compresores es viable

## 1.3 Estructura de la memoria

La memoria de este trabajo está dividida en siete capítulos, junto con un anexo.

1. **Introducción**. En este apartado es en el que nos encontramos ahora mismo. En él se introduce el tema que se va a tratar, se fijan los objetivos y describe la estructura de la memoria.
2. **Estado de la cuestión**. Aquí se describen los métodos de compresión de imágenes más usados hoy en día tanto en compresión con como sin pérdida de información y se exponen los resultados que estos obtienen.
3. **Marco Teórico**. En este apartado se explican los fundamentos de conocimientos que han sido necesarios para llevar a cabo este trabajo.
4. **Metodología Empleada**. Aquí se describen las herramientas utilizadas para el desarrollo, las herramientas que se han creado para facilitar el trabajo y los diseños que se han implementado, especificando también cómo se han implementado.
5. **Experimentación y Resultados**. En este capítulo se describen los diferentes experimentos que se han llevado a cabo para encontrar el mejor modelo posible. Además, se exponen los resultados de cada uno de estos experimentos.
6. **Conclusiones**. Explica las conclusiones finales de este trabajo y mejoras que se podrían aplicar en futuros trabajos.
7. **Bibliografía**. Enumera todas las fuentes externas utilizadas en este trabajo.
8. **Anexo ODS**. Explica el grado de relación de este trabajo con los objetivos de desarrollo sostenible de las Naciones Unidas.

# CAPÍTULO 2
# Estado de la cuestión

El objetivo de este trabajo es obtener un algoritmo de compresión alternativo a los métodos de compresión actuales. Es decir, este es un trabajo enfocado en la imagen digital y en las herramientas que se pueden utilizar para el desarrollo de nuevos tratamientos de imagen digital.

En este capítulo se investigan cuales son los algoritmos de compresión de imágenes, con y sin pérdida, más utilizados hoy en día. También se investigan cuales son los porcentajes de compresión que se obtienen y la precisión de reconstrucción que tienen los métodos con pérdida.

Los algoritmos de compresión de datos suelen tener dos componentes, el compresor y el descompresor, los cuales encuentran una representación de menor espacio en memoria a partir de los datos originales y reconstruyen los datos originales a partir de esa representación de menor espacio. Además, la compresión de datos se puede clasificar en dos categorías: con pérdida y sin pérdida. Estos dos tipos de compresión intentan resolver problemas diferentes y se utilizan en diferentes aplicaciones.

## 2.1 Algoritmos de compresión sin pérdida

La compresión con pérdida es aquella que, a partir de la representación comprimida, es capaz de reconstruir la información con total fidelidad, es decir sin desviación alguna de los datos originales. En algunas aplicaciones esta propiedad es necesaria. Por ejemplo, la compresión de texto debe ser sin pérdida, ya que si el resultado de la reconstrucción tuviera errores, el texto resultante no sería legible o tendría un mensaje erróneo.

En algunas aplicaciones de la compresión de imágenes es también necesario que la reconstrucción sea exactamente igual que la imagen original, como en algunas aplicaciones médicas, donde se busca tener la mayor claridad de imagen posible. Algunos de los algoritmos de compresión de datos sin pérdida más importantes son *Lampel-Ziv compression* [21] y *Predicton by partial matching* [30].

En cuanto a la compresión de imágenes sin pérdida, los formatos más populares en la práctica son PNG y TIFF. Ambos algoritmos son muy parecidos, siendo utilizados en imágenes de trama, no vectoriales, y utilizando algoritmos de compresión muy similares. Además, PNG cuenta con la capacidad de comprobar errores con una suma de control, lo cual lo hace perfecto para la transmisión a través de internet.

En el artículo [31] se estudia cual es la proporción de compresión de los formatos de compresión de imágenes sin pérdida más utilizados hoy en día. En dicho estudio se

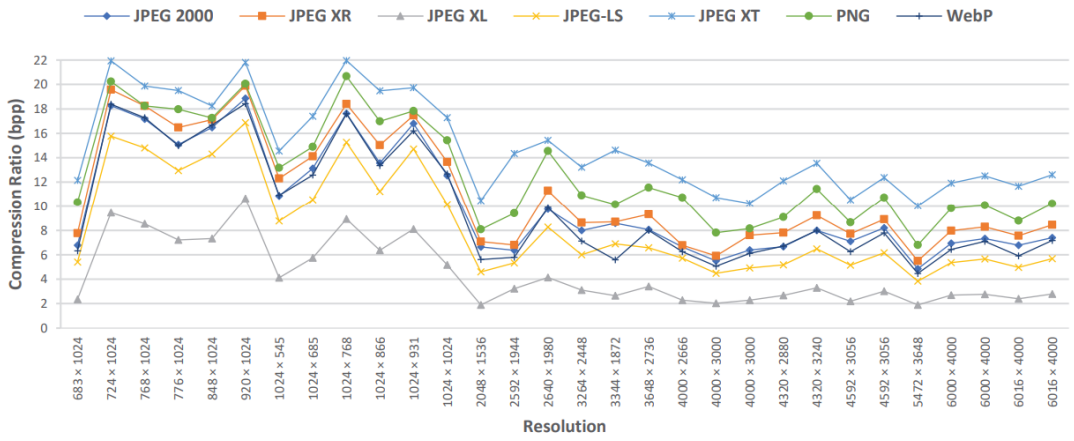



**Figura 2.1:** Proporciones de compresión de algoritmos sin pérdida, incluyendo PNG. Imagen extraída de [31].

comprimen imágenes con contenidos variados y tamaños diferentes con los compresores que se estudian.

Los resultados de este estudio se pueden ver en la figura 2.1. En dicha figura se comparan la proporción de compresión de JPEG 2000, JPEG XR, JPEG XL, JPEG-LS, JPEG-XT, PNG y WebP en imágenes de tamaños entre 683x1024 y 6016x4000. Se puede observar que el algoritmo de compresión JPEG XL es el que obtiene mejores resultados de forma consistente, mientras que PNG obtiene los peores resultados.

## 2.2 Algoritmos de compresión con pérdida

La compresión con pérdida es aquella que, a partir de la representación comprimida, reconstruye los datos originales con una fidelidad muy alta, pero sin ser exactamente igual a la original. Estos algoritmos de compresión se suelen utilizar para aplicaciones en las que la representación totalmente exacta no es una necesidad y una buena aproximación es totalmente válida. Debe tenerse en cuenta que, al perder algo de información de los datos originales, estos formatos de compresión alcanzan proporciones de compresión muchísimo más altas y en cuanto más información se elimine mayor será el porcentaje de compresión.

Este tipo de algoritmos de compresión son especialmente útiles en la compresión de imágenes, ya que el ojo humano es capaz de intuir formas y colores generales, ignorando detalles concretos que puedan ser incorrectos. Concretamente, la compresión de imágenes con pérdida es usada en la web, ya que estas imágenes normalmente se quieren únicamente por su estética y un detalle muy alto en ellas no es necesario.

En el artículo [11] se estudia cual es la proporción de compresión de los algoritmos de compresión JPEG y JPEG 2000, dos de los formatos de compresión con pérdida más utilizados en la práctica hoy en día. Para realizar el estudio se comprimen las mismas imágenes con ambos compresores para obtener diferentes proporciones de compresión. Entonces estas imágenes se descomprimen y se mide la diferencia de reconstrucción media para cada nivel de compresión. Esto se mide con una métrica llamada PSNR, donde un PSNR mayor indica una mejor reconstrucción de la imagen original.

Los resultados de este estudio se pueden ver en la figura 2.2. En la gráfica se observan los resultados de ambos algoritmos, con JPEG en blanco y JPEG 2000 en negro y utilizan-

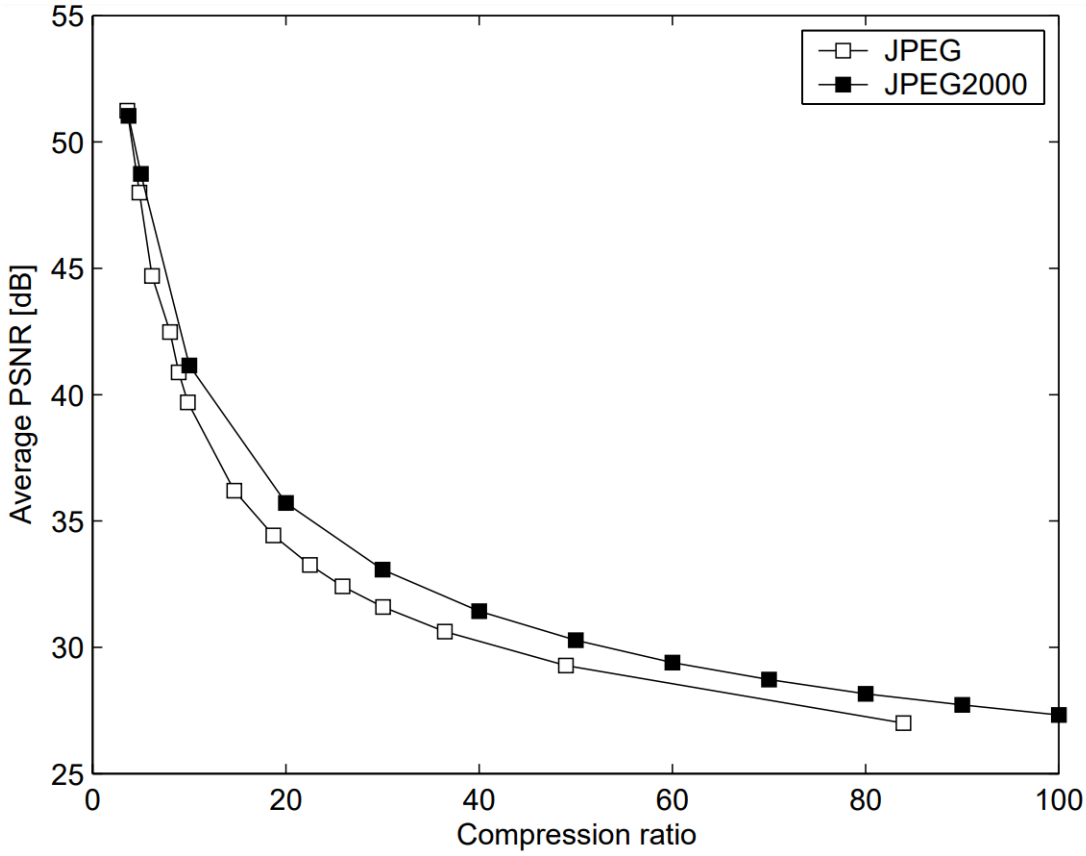



**Figura 2.2:** PNSR medio contra proporción de compresión de JPEG y JPEG200 [11]

do en el eje $x$ una proporción de compresión en vez de bits por píxel (bpp). En todos los casos JPEG 2000 obtiene los mejores resultados, con una puntuación de PNSR mayor.

# CAPÍTULO 3
# Marco Teórico

En esta sección se introducen todos los conocimientos teóricos que han sido necesarios para el desarrollo de este proyecto. Entre ellos se encuentran las redes neuronales artificiales, que son la forma en la que crean los *autoencoders*, cómo se entrenan y todos los componentes que las crean. Además, se explican otros conocimientos necesarios para el tratamiento de imágenes, como los modelos de representación del color, los algoritmos de compresión de PNG y JPEG y las métricas de similitud entre imágenes. Finalmente, se hace una introducción a la reducción de dimensionalidad y se explica cómo funciona en un *autoencoder*.

## 3.1 Redes neuronales artificiales

Las redes neuronales artificiales, o *ANN* de sus siglas en inglés, son un método de aprendizaje inspirado en el funcionamiento de las neuronas de un cerebro biológico. Estas funcionan como una función matemática muy compleja, que es capaz de adaptarse a los datos que se le proporcionan.

La unidad básica de una red neuronal es la neurona. Esta se representa por un nodo con un sesgo ($B$) y un peso ($W$). Estas neuronas se organizan en capas, que recogen valores ($X$) y utilizan su parámetros para calcular nuevas representaciones ($Y$). Estas capas pueden entenderse como operaciones matriciales con la siguiente forma:

$$Y = X \times W + B$$

Estas capas a su vez pueden ser ordenadas de forma que el resultado de una alimente a las siguientes, proporcionándole una estructura de red, muy similar a las redes neuro-

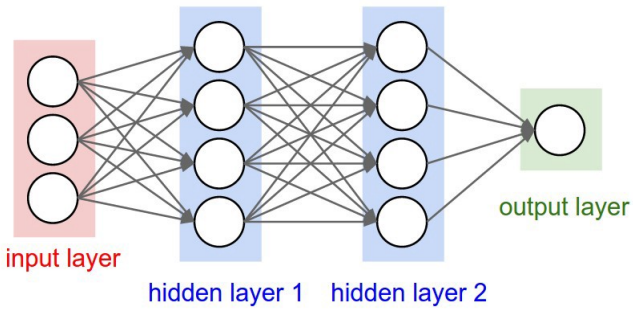



**Figura 3.1:** Ejemplo de red neuronal artificial [10].

nales biológicas. Estas redes pueden ser de diferentes profundidades, dependiendo del número de capas que la formen, y cada una de estas capas pueden tener diferentes anchuras. Del uso de redes neuronales profundas es de donde recibe el nombre el «Aprendizaje Profundo» o *Deep Learning*. La estructura de red se puede ver fácilmente cuando estas funciones matemáticas son representadas como grafos, como en la Figura 3.1.

Sin embargo, combinar capas de esta forma, con operaciones lineales una tras otra, resulta en otra operación lineal. Aunque una operación lineal se puede usar para clasificar, esta no nos proporciona una expresividad suficiente. Para incrementar esta expresividad y conseguir que las redes neuronales no sean funciones lineales se intercalan las capas lineales con funciones no lineales, llamadas funciones de activación.

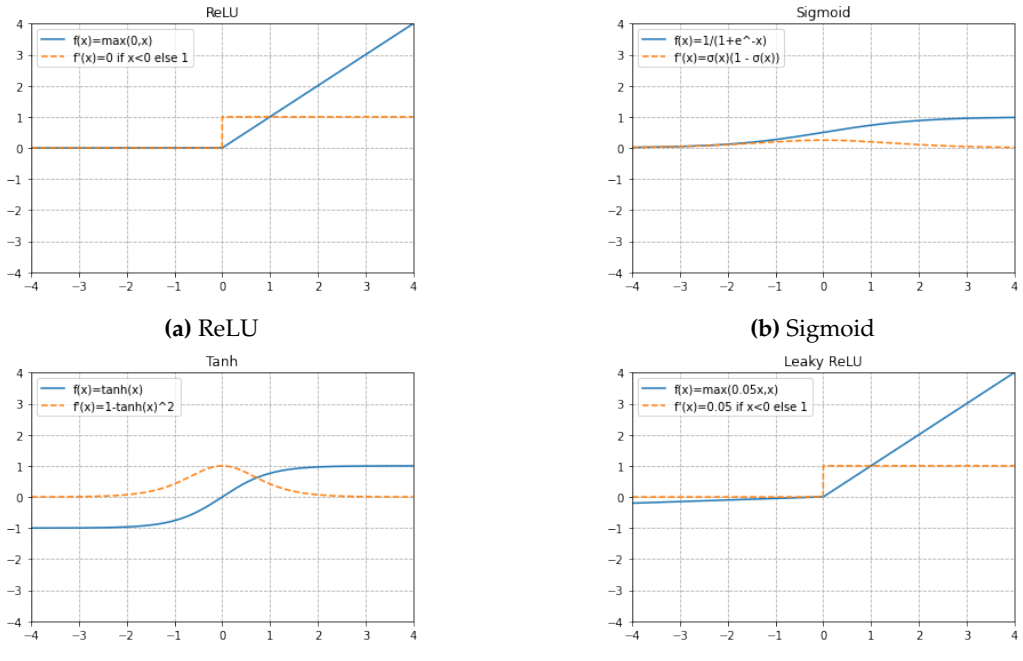
Finalmente, en la capa de salida, se recogen los resultados de esta función. Dependiendo del problema el número de neuronas que tiene esta capa de salida, es decir su ancho, puede variar. En problemas de clasificación suele usarse como ancho de salida el número de clases entre las que se quiera clasificar, mientras que en problemas de regresión se utilizan tantas dimensiones como el problema requiera.

### 3.1.1. Funciones de activación

Las funciones de activación son el secreto que hace que las redes neuronales sean capaces de resolver problemas complejos. Estas se añaden después de cada capa lineal, para evitar el comportamiento lineal en todos los pasos. Además de no ser operaciones lineales, las funciones de activación deben tener, preferiblemente, una primera derivada que sea fácilmente computable. La razón para esto quedará evidente en la sección 3.1.3.

Existen muchas funciones de activación, cada una con sus propiedades. En esta sección se explicarán algunas de ellas.

- *Sigmoid* es otra función de activación que era muy utilizada en los primeros modelos de redes neuronales. Una gráfica de *Sigmoid* se puede ver en la figura 3.2b. Esta




**(a)** ReLU **(b)** Sigmoid

**(c)** Tanh **(d)** Leaky ReLU

**Figura 3.2:** Gráficas de funciones de activación y sus derivadas

función tiene un rango de $(0, 1)$, es derivable en todo su dominio y su derivada es muy fácil de computar, ya que tiene la siguiente propiedad:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$
$$f'(x) = \sigma(x)(1 - \sigma(x))$$

Sin embargo, *Sigmoid* sufre de un problema de saturación. Solo un intervalo muy estrecho del dominio de su derivada resulta en un valor suficientemente diferente a cero, esto hace que cuando sus valores se encuentran en estas zonas planas resulte muy difícil aprender. Para evitar esto se introdujo *ReLU*.

- *ReLU*, *Rectified Liear Unit* en inglés, es una función de activación no lineal que pone todos los valores inferiores a cero como cero. De esta forma muchas de las neuronas terminan con un valor nulo, «desactivándolas». Durante el entrenamiento, esta desactivación la convierte en una función muy eficiente, ya que no todas las neuronas se activan a la vez, en cambio solo una fracción de ellas se activa. Además, hace que, durante la fase de aprendizaje, a veces los valores de peso y sesgo no se actualicen. Se puede ver una gráfica de *ReLU* en la figura 3.2a.
- *Tanh* es una función relativamente similar a *Sigmoid*, pero su rango es $(-1, 1)$ en vez de $(0, 1)$. Esto resulta en diferentes signos de valores que se alimentarán a la siguiente capa. Además, *tanh* se puede definir en términos de *Sigmoid* de la siguiente forma $tanh(x) = 2\sigma(2x) - 1$, por lo que es similar en cuanto a coste computacional. Sin embargo, sufre de los mismos problemas de saturación que *Sigmoid*, pero con un intervalo más reducido incluso, ya que su pendiente es mayor. Se puede ver una gráfica de *tanh* en la figura 3.2c.
- *Leaky ReLU* es una versión de *ReLU* en la que los valores inferiores a cero no toman un valor nulo. En cambio estos toman como valor una fracción de la entrada. Se puede ver una gráfica de *Leaky ReLU* en la figura 3.2d.

El campo de las funciones de activación es uno que todavía está abierto, sobre el que se sigue investigando y publicando. Otras funciones que se utilizan a menudo, y que obtienen diferentes resultados, son *Exponential Linear Unit*, *Swish Function*, *Softmax*, *Parametrized ReLU* y más.

### 3.1.2. Función de pérdida

Las funciones de pérdida son la forma de medir cómo de cerca está la predicción hecha por la red neuronal del valor objetivo. En cierta forma es la manera de calcular la distancia que hay entre estos dos valores. Intuitivamente, una mayor pérdida indica peores predicciones o un peor rendimiento; de esa forma sabemos como el modelo que se está entrenando debe mejorar. Muchas veces el tipo de función de pérdida que se use depende del tipo del problema. Algunas de las funciones de pérdida son:

#### Problemas de regresión

- Error medio cuadrático o *MSE*. Mide el error cuadrático entre la predicción y el valor objetivo. Es una función de error muy popular en los problemas de regresión ya que su pendiente es variable y disminuye con el error.

$$L = \Sigma_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- Error absoluto o Error L1. Mide el error absoluto entre el valor de la predicción y el objetivo. A diferencia de *MSE* no es tan susceptible a valores atípicos, ya que crece linealmente. Sin embargo, la pendiente de L1 en el valor objetivo, es decir en el valor del dominio cero, no es nula, ya que esta es constante a ambos lados de este punto en el dominio. En otras palabras, L1 no es derivable en todo su dominio, ya que no tiene pendiente en el cero.

$$L = \Sigma_{i=1}^{N} |y_i - \hat{y}_i|$$

**Problemas de clasificación**

- *Cross Entropy Loss* es una función de pérdida para clasificación de múltiples clases. Esta utiliza la propiedad de la teoría de la información que dice que algo más probable de conseguir proporciona menos información para definir la siguiente fórmula:

$$L = -\Sigma_{i=1}^{n} \Sigma_{k=1}^{k} (y_{ik} \cdot \log(\hat{y}_i))$$

Igual que con las funciones de activación, las funciones de pérdida son un campo de investigación y del que todavía se publican artículos. Otras funciones de pérdida para problemas de clasificación son *Exponential Loss*, *Hinge Loss*, *Pinball Loss* y *Rescaled Hinge Loss*, mientras que para problemas de regresión existen *Huber Loss*, *Log-cosh Loss* y *Quantile Loss* [28]

### 3.1.3. Entrenamiento de redes

Una vez la arquitectura de una red neuronal ha sido decidida, las funciones de activación están en su lugar y la función de pérdida preparada, solo queda entrenarla. El entrenamiento de una red nueronal es, en esencia, la optimización de su función matemática respecto a la función de pérdida. Sin embargo, calcular en qué punto la función de pérdida tiene un valor menor es muy complicado de manera analítica, ya que esta tiene innumerables mínimos locales. Para ello, el entrenamiento se hace a partir de una algoritmo de descenso de gradiente.

Para llevar a cabo el descenso de gradiente es necesaria una gran cantidad de datos, que se separan en tres conjuntos: el conjunto de entrenamiento, validación y *testing*. El entrenamiento se realiza únicamente a partir del conjunto de entrenamiento, y es del único de los tres conjuntos del que aprenderá una red neuronal. Este conjunto suele tomar entre el 60 y 80 por ciento de los datos, aunque depende mucho de la cantidad de datos que se tenga.

El objetivo es conseguir que los parámetros de una red neuronal, los pesos y sesgos de cada una de las neuronas, encuentren el valor óptimo para que la función de pérdida dé el valor mínimo posible, en el mayor número de casos posibles. Además, estos parámetros no deberán ajustarse únicamente al conjunto de entrenamiento, sino que deberán hacerlo también al conjunto de validación.

Para conseguir esto, se necesita de una cantidad de datos en cuanto más extensa posible mejor. Además, se pueden usar técnicas de regularización, como añadir un término extra a la función de pérdida, para que la red neuronal sea capaz de extrapolar sus aprendizajes al mayor número de casos no previamente contemplados como sea posible.

La parte más importante durante el entrenamiento, sin embargo, son dos fases llamadas «propagaciones». Estas reciben sus nombre por la propagación de los resultados

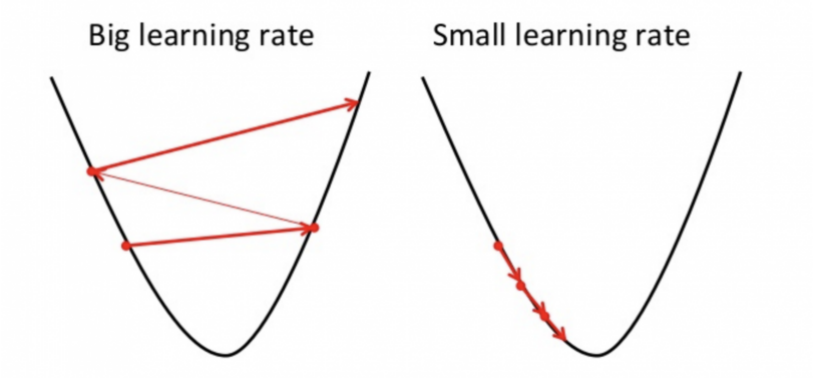



**Figura 3.3:** Una tasa de aprendizaje muy grande (izquierda) puede hacer que el descenso de gradiente nunca converja, mientras que una tasa de aprendizaje muy pequeña (derecha) puede hacer que esta converja muy lentamente [9]

primero hacia delante, de entrada a salida, y luego hacia atrás, del error hacia la entrada, pasando por todos los parámetros. Estos dos procesos se llaman:

1. Propagación hacia delante (*Forward pass/propagation*). Este paso es el cálculo de los resultados de cada neurona haciendo uso de los parámetros que tiene en ese momento. Estos resultados se pasan de capa en capa, hasta que llegan al final, donde se calcula el error con la función de pérdida, dando fin a esta fase.

2. Propagación hacia atrás (*Back-propagation*). En esta fase se calcula la pendiente de la función de pérdida respecto a cada uno de los parámetros. Para ello se hace uso de la regla de la cadena del cálculo [12]. Esta dice lo siguiente:

$$\frac{dx}{dy} = \frac{dx}{dt}\frac{dt}{dy}$$

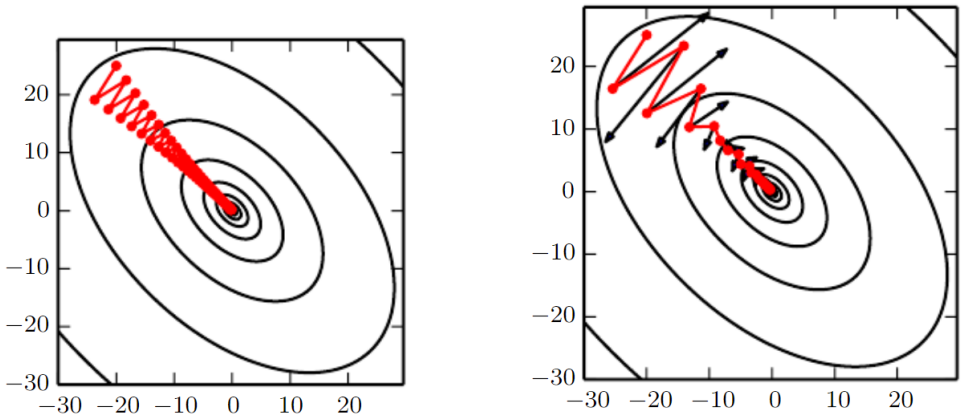
   Usando la regla de la cadena, se puede derivar una expresión algebraica para la pendiente de cualquier parámetro de la red neuronal de forma sencilla. De esta forma, es posible calcular la pendiente de todos los parámetros respecto a la pérdida.

### 3.1.4. Optimizadores

La fase de *Back-propagation* nos indica cuál es la pendiente respecto a la función de pérdida de cada uno de los parámetros. Esto indica cómo debe de afectar a cada parámetro la actualización durante la fase de descenso de gradiente. Sin embargo, saber en qué medida se debe actualizar cada uno de estos parámetros para que converja no es tan sencillo como parece. Algunos problemas que pueden surgir durante la fase de descenso de gradiente se pueden ver en la Figura 3.3.

Una forma de solucionar estos problemas es la introducción del parámetro de Tasa de Aprendizaje o *Learning Rate*. Este valor es multiplicado por la pendiente de cada uno de los parámetros, haciendo que el aprendizaje sea más rápido o lento según se necesite.

Sin embargo, únicamente introduciendo este parámetro no se resuelven todos los problemas, ya que, por ejemplo, en cada fase del aprendizaje la Tasa de Aprendizaje debe ser diferente, con saltos cada vez más pequeños para ajustar más finamente los parámetros conforme avanza el entrenamiento en el tiempo, o que cada parámetro puede necesitar una Tasa de aprendizaje diferente. Para solucionar todos estos problemas se hace uso de algoritmos de optimización, también conocidos como Optimizadores.




**(a)** Descenso de pendiente. ([12] Capítulo 4.3)

**(b)** Descenso de pendiente estocástico más momento. ([12] Capítulo 8.3)

**Figura 3.4:** Se muestra cómo diferentes algoritmos de optimización se acercan al mínimo global.

### Descenso de pendiente

El descenso de pendiente es un método de aprendizaje muy utilizado en el ámbito del *Machine Learning*. Para ello primero se calcula la pendiente respecto la función de pérdida ($\Delta_\theta L$), entonces se actualizan los pesos del modelo ($\theta$) restándoles esta pendiente multiplicada por la Tasa de Aprendizaje ($\alpha$).

$$\theta^{k+1} = \theta^k - \alpha \cdot \Delta_\theta L$$

#### Descenso de pendiente estocástico - *Stochastic Gradient Descent*

En el descenso de gradiente, dadas $n$ muestras de entrenamiento se debe calcular la pendiente de cada una de esas muestras, lo cual tiene una complejidad temporal de $O(N)$. El error usado para calcular la pendiente es el valor medio del error de todas las muestras, en otras palabras es el valor esperado. El descenso de pendiente estocástico propone que, si en vez de coger todo el conjunto de datos de entrenamiento se coge una sola fracción, estadísticamente el valor esperado de error es el mismo.

Para ello, se entrena el modelo usando *mini-batches* o mini-lotes en vez del conjunto entero de entrenamiento. Esto implica que en el primer mini-lote aprenderá a partir de esas muestras y luego aplicará lo aprendido para el siguiente mini-lote, pero cada fase de aprendizaje tiene una complejidad de $O(\frac{n}{i})$, donde $i$ es el número de mini-batches en los que se quiera dividir el conjunto de entrenamiento. Al final, este algoritmo converge en una solución, igual que si se usara el conjunto de entrenamiento entero, pero más rápido.

Sin embargo, el descenso estocástico de pendiente sigue teniendo algunos problemas, como que no puede escalar independientemente en diferentes direcciones o que la Tasa de aprendizaje debe ser conservadora para evitar divergencia. A esto se le suma el problema de que el error calculado por cada mini-lote varía de la media del lote, otro problema que se debe aplacar.

#### *Gradient descent* + Momento

Este método se usa sobre el descenso de pendiente estocástico, incluyendo una variable de momento. La introducción del momento es una forma de disminuir las escalas que oscilan a lo largo de una dimensión y, sobre todo, aumentar las escalas que siempre van en la misma dirección. Para esto se le añade un parámetro extra, para decidir la tasa

de acumulación de momento ($\beta$). Este algoritmo acelera el entrenamiento acelerando el aprendizaje en las direcciones que se repiten.

$$v^{k+1} = \beta \cdot v^k - \alpha \cdot \Delta_\theta L(\theta^k)$$
$$\theta^{k+1} = \theta^k + v^{k+1}$$

**Raíz cuadrada del valor medio de los cuadrados - *RMSProp***

*RMSProp* construye sobre el Descenso de Pendiente Estocástico, dividiendo la tasa de aprendizaje por una media de las pendientes cuadradas, que disminuye exponencialmente. De esta forma se amortiguan las oscilaciones en las direcciones con variaciones muy altas, acelerando la velocidad de entrenamiento, ya que es más difícil que diverja. Para este algoritmo se introducen dos parámetros, una constante, que estabiliza las divisiones por números muy bajos ($\epsilon$), y un factor de decaimiento, que indica cómo se acumulan los cuadrados de las pendientes ($\mu$).

$$s^{k+1} = \mu \cdot s^k + (1-\mu) \cdot [\Delta_\theta L(\theta^k) \circ \Delta_\theta L(\theta^k)]$$
$$\theta^{k+1} = \theta^k + \alpha \cdot \frac{\Delta_\theta L(\theta^k)}{\sqrt{s^{k+1}} + \epsilon}$$

**Estimación adaptativa de momento - *ADAM***

*ADAM* busca combinar los métodos de Momento y *RMSProp*. Para ello, se calcula la media de las pendientes y la variación de las pendientes para aplicar los dos algoritmos anteriores a la vez.

$$v^{k+1} = \beta \cdot v^k + (1-\beta) \cdot \Delta_\theta L(\theta^k)$$
$$s^{k+1} = \mu \cdot s^k + (1-\mu) \cdot [\Delta_\theta L(\theta^k) \circ \Delta_\theta L(\theta^k)]$$
$$\theta^{k+1} = \theta^k + \alpha \cdot \frac{v^{k+1}}{\sqrt{s^{k+1}} + \epsilon}$$

### 3.1.5. Topologías de Redes Neuronales

Las redes neuronales artificiales ayudan a resolver una variedad problemas muy diferentes y cada vez se buscan nuevas formas de expandir la utilidad de estas herramientas. Sin embargo, diseñar modelos para que se ajusten lo mejor posible a nuestro problema es una tarea difícil. Aquí se explican algunas de los tipos de redes neuronales que se utilizan o se han utilizado para resolver diferentes problemas.

***Feed Forward Networks - FFN***

Las *FFN* son el tipo de red neuronal más clásicas. Estas, como se explica en la sección 3.1, se construyen apilando capas ocultas de neuronas comúnmente llamadas capas «Totalmente conectadas», o *fully connected*, e intercalándolas con capas de activación. Las capas *fully connected* pasarán todos sus resultados a cada una de las neuronas de la siguiente capa. Apilando más y más capas se consiguen construir modelos más complejos, que le dan una mayor potencia de clasificación o regresión a los modelos. Una ilustración de una red *FNN* se puede ver en la Figura 3.1.

***Residual Networks - ResNET***

Es lógico llegar a la conclusión de que una red neuronal más profunda siempre será mejor, pero esto sería incorrecto, ya que unos problemas muy comunes en las redes neuronales demasiado profundas son los de los gradientes que se desvanecen y gradientes explosivos. Para aplacar estos problemas, se introducen las conexiones de salto. La idea de estas conexiones es darle un camino más directo a los datos, realimentando las capas más profundas con datos de capas anteriores. De esta forma, el proceso de entrenamiento es más rápido y efectivo.

***Recurrent Networks - RNN***

Las *RNN* son una familia de redes neuronales para procesar datos secuenciales. Estas guardan los resultados intermedios de pases anteriores para calcular el resultado del pase actual. De esta forma, pueden usar lo que han aprendido en los pases anteriores y aplicarlo en el actual. Una arquitectura en concreto de esta familia son los *Long Short Term Memory* o *LSTM*, que recogen los datos de múltiples pases anteriores en vez de solo uno, como hacen las *RNN* convencionales. Esto es, en esencia, otorgarle una memoria a corto y largo plazo a la red neuronal, en vez de solo corto plazo como en las *RNN*.

***Autoencoder Networks***

Los *autoencoders* pueden verse como un caso especial de *FFN*, que es entrenada copiando su entrada a la salida. De entre sus capas ocultas, una de ellas describe un código que representa la entrada. Históricamente los *autoencoders* se han usado la reducción de dimensionalidad, obteniendo un código $h$ que ocupa menos espacio que los datos originales, aplicación que se le da en este trabajo también.

Este código $h$ es también llamado un espacio latente. El espacio latente contiene una representación comprimida de la entrada, recogiendo determinada información, que será la única información que el decodificador podrá usar para reconstruir la entrada en la salida con la mayor lealtad posible. Más información sobre esta familia de redes neuronales en la sección 3.2.

**Redes neuronales convolucionales - *CNN***

Las redes neuronales convolucionales son un tipo de *FFN* que aprovecha la estructura de los datos aplicando operaciones matemáticas llamadas convoluciones. Esta técnica se usa satisfactoriamente en problemas de reconocimiento de formas en imágenes, procesamiento del lenguaje natural y otros procesos de análisis de imágenes. Una ventaja muy importante de las *CNN* es que, al reemplazar las capas totalmente conectadas de las *FFN* por capas convolucionales, se reduce sustancialmente el número de parámetros que se debe entrenar.

***Generative adversarial networks - GAN***

Un *GAN* cuenta con dos partes un generador y un discriminador. Estas dos partes trabajan como adversarios, los generadores son entrenados para que sean capaces de simular datos originales, mientras que los discriminadores son entrenados para diferenciar entre datos originales y simulados. Esta familia de redes neuronales entran dentro de la categoría de aprendizaje no supervisado, ya que nuevos datos pueden ser generados a partir de los datos de entrada.

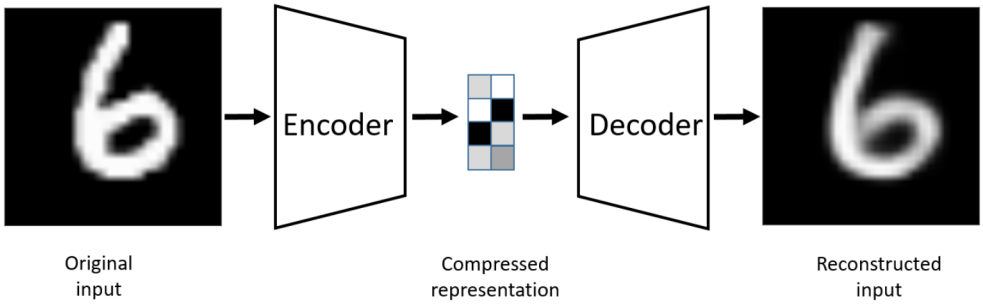



**Figura 3.5:** Estructura visual de un *autoencoder* para la compresión de imágenes. Imagen obtenida en [5].

## 3.2 *Autoencoders* para la reducción de dimensionalidad

De entre las aplicaciones de los *autoencoders*, una de las más comunes es la reducción de la dimensionalidad de los datos. Normalmente copiar la entrada en la salida puede parecer algo bastante inútil; sin embargo, con este proceso se consigue que las capas intermedias sean capaces de recoger información interesante sobre la muestra. Estas capas intermedias, también llamadas espacios latentes, son diferentes representaciones de los datos que se introducen. Una manera de recoger esta información es teniendo una dimensión inferior en las capas intermedias a la de la entrada, usando un cuello de botella.

Una reducción de la dimensionalidad implica que se pierde información por el camino, que estos datos ya no están completos. Por ello, este tipo de *autoencoders* recibe el apellido de «Incompletos». El proceso que sigue un *autoencoder* es muy similar al de otros algoritmos de reducción de dimensionalidad, como PCA. De hecho, si utilizáramos una sola capa oculta y *MSE* como función de pérdida, sería totalmente equivalente a PCA y aprendería a proyectar el conjunto de entrenamiento al mismo subespacio.

Sin embargo, gracias a la arquitectura modular de una red neuronal *FFN*, de la que deriva la arquitectura *autoencoder*, podemos hacer uso de funciones no lineales, intercalándolas entre las capas lineales. Así se consigue una transformación generalizada no lineal más potente que la de PCA, que se ajusta mejor a los datos de entrenamiento. Esto, a su vez, puede resultar en un problema ya que un *autoencoder* con demasiada libertad puede aprender que un código $i$ representa una salida $x^{(i)}$, sin haber extraido información relevante, y obtener malos resultados en casos no vistos durante el entrenamiento.

Al final, un *autoencoder* se compone de dos partes: una función codificadora $f$ y una función decodificadora $g$, como se puede ver en la figura 3.5. El codificador extrae la información relevante, aplicando su función no lineal a los datos, y con esa información la función decodificadora proyecta al espacio original, tomando la entrada del codificador y aplicando una nueva proyección no lineal sobre ella. Estas funciones deben usarse siempre juntas.

En el entrenamiento, como se quiere minimizar el error de reconstrucción, el objetivo es minimizar la función de pérdida que, dada una entrada $x$, se define de la siguiente forma:

$$L(x, g(f(x)))$$

donde la función de pérdida es una medida de disimilitud entre la entrada $x$ y la reconstrucción $g(f(x))$. Así, el codificador aprende a recoger las características importantes

de la entrada en un espacio latente. En cuanto mayor sea el ancho del cuello de botella y más dimensiones tenga para representar sus propiedades, mayor será la capacidad de recoger información del codificador. En otras palabras, un espacio latente de mayor dimensionalidad será capaz de recoger más información de la muestra y, por lo tanto, el decodificador podrá reconstruir la muestra más fielmente.

Además, durante el entrenamiento, el decodificador aprende a reconocer la información que recibe del codificador y a reconstruir los datos originales. Este lenguaje en el que se comunican es único entre estas dos partes, ya que no todos los codificadores obtendrán el mismo resultado y ningún otro decodificador será capaz de entender el espacio latente resultante de este codificador, a no ser que sea entrenado expresamente con este.

### 3.2.1. Otras aplicaciones de los *autoencoders*

La capacidad de reducción de dimensionalidad de un *autoencoder* puede ser usada de diversas formas y, aunque históricamente su objetivo final ha sido ese, emergen otras aplicaciones a partir de los espacios latentes. En esta sección simplemente se mencionarán estas aplicaciones, sin hacer especial hincapié en ellas.

Una representación latente puede hacer muchas tareas más eficientes. Al tener que tratar con menos datos, los cálculos son mucho más rápidos y el espacio de almacenamiento necesario es también mucho menor. Un proceso que se agiliza mucho mediante la reducción de dimensionalidad es la búsqueda de información. Al tener menos parámetros entre los que buscar, encontrar la información necesaria es mucho más eficiente, sobre todo si estos códigos son del tipo *binarios*.

#### *Denoising autoencoders*

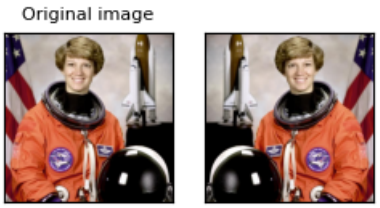
Otra tarea que pueden cumplir los *autoencoders* es la de la eliminación o reducción de ruido. Para entrenar estos modelos se usa de entrada una versión de la salida con un ruido añadido. En la salida, se compara con los datos originales, pero esta vez sin el ruido. El *autoencoder* aprende a eliminar el ruido de las muestras de entrada, aprendiendo solo las características importantes en un espacio latente, e ignorando el ruido. Además, aprende a reconstruir los valores originales a partir del resto de datos.

Este tipo de modelos pueden ser entrenados también para mejorar la capacidad de generalización a datos no vistos durante el entrenamiento. Este proceso se llama «aumento de datos» y se utiliza en muchos problemas de visión por computador, ya que de esta forma las redes entrenadas con este método aprenden a utilizar más datos de la entrada para obtener las conclusiones. En el caso de los *autoencoders*, estas conclusiones son las características extraídas y la reconstrucción.

## 3.3 Técnicas de aumento de datos en imágenes

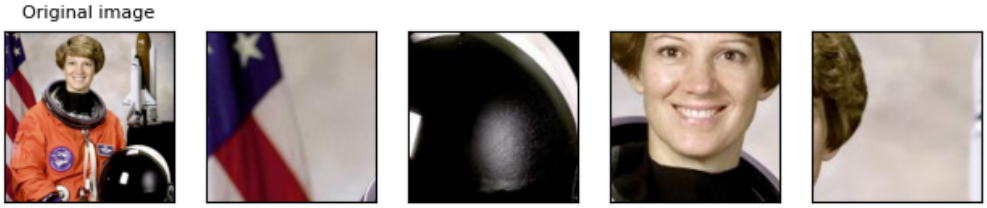
Obtener los datos apropiados es una de las partes más importantes del proceso de entrenamiento. Esto es en ocasiones complicado. Obtener suficientes datos para que las redes que se entrenan sean capaces de generalizar a casos nuevos, que no han sido vistos previamente durante el entrenamiento, es en ocasiones difícil e incluso imposible. Para superar estos problemas se han desarrollado diferentes métodos de «aumento de datos».

El aumento de datos es una forma de, a partir de un conjunto de datos de entrenamiento, conseguir un conjunto de datos mayor, es decir aumentar la cantidad de datos




**(a)** Reflejo de forma horizontal


**(b)** Reflejo de forma vertical




**(c)** *Crops* aleatorios

**Figura 3.6:** Ejemplos de aumento de datos usando *crop* y reflejos extraídos de [19]

de la que se dispone. Estos procesos aplican muchas veces modificaciones a los datos de entrenamiento, de diversas formas, para obtener nuevos datos basados en los originales.

En el caso del aumento de datos para imágenes, estas técnicas pueden tener muchas formas, pero, en todos los casos, tratan de obtener imágenes nuevas a partir de las imágenes de entrenamiento originales. Estas técnicas pueden ser aplicadas individualmente, en conjunto y en diversas proporciones para obtener diferentes resultados.

### 3.3.1. Tipos de aumentos de datos en imágenes

Algunos de los métodos de aumento de datos que se aplican en imágenes se han tomado de [23], y se explican en esta sección. Además de una corta explicación de los procesos, se incluyen ejemplos de cómo estas técnicas se verían en la práctica.

#### Modificaciones geométricas

Las transformaciones geométricas son muy útiles cuando existe un sesgo en la geometría de los datos de entrenamiento. Los sesgos en el conjunto de entrenamiento pueden aparecer de muchas formas, pero si este este viene de forma geométrica, como cuando el objeto buscado está siempre en la misma postura o en la misma posición relativa dentro de la imagen, las transformaciones geométricas pueden ser una manera muy buena de obtener mejores generalizaciones.

Un ejemplo de transformación geométrica pude ser el reflejo. Reflejar una imagen es bastante fácil de entender. A partir de la imagen original se obtiene una nueva en la que se invierte uno de los ejes. De esta forma, dependiendo de las orientaciones en las que se aplique el reflejo, se pueden obtener hasta cuatro versiones, que, a efectos prácticos, es multiplicar por cuatro el tamaño del conjunto de entrenamiento. Se pueden ver ejemplos de cómo se vería esto en una imagen en las figuras 3.6a y 3.6b.


**(a)** Imagen original y versiones con ruido gausiano y *speckle*


**(b)** Imágenes con ruido del tipo *Salt and Pepper*, Oculsión, y *Poisson*

**Figura 3.7:** Ejemplos de aumento de datos usando inyección de ruido aleatorio extraídos de [3]

### *Cropping* o recortes

El *cropping* consiste en, a partir de una imagen de dimensiones HxW, obtener una o varias imágenes recortando los píxeles de la imagen original. En todos los casos la imagen resultante tiene una dimensión AxB que es inferior o igual a la de la imagen original.

Este tipo de aumento de datos se puede utilizar para sustituir un cambio de resolución de la imagen original, una alternativa que se habría utilizado para reducir el tamaño de la entrada. Sin embargo, a diferencia de un cambio de resolución que solo tiene un posible resultado, el *cropping* puede obtener diferentes resultados, dependiendo de cómo se configure.

Algunos métodos establecidos son el de recortar cinco piezas de forma que se recortan las esquinas y el centro o recortar de forma aleatoria. Todos estos métodos son alternativas que se usan en la práctica y con los que se han visto resultados positivos [22]. Un ejemplo de cómo se vería el recorte de la imágenes de forma aleatoria se puede ver en la figura 3.6c.

### Inclusión de ruido aleatorio

Cuando el sesgo que existe en los datos de entrenamiento no es del tipo geométrico y es más complejo que este, la inclusión de ruido en las imágenes es una manera de hacer que el modelo aprenda a generalizar de forma muy efectiva. Este ruido se puede aplicar

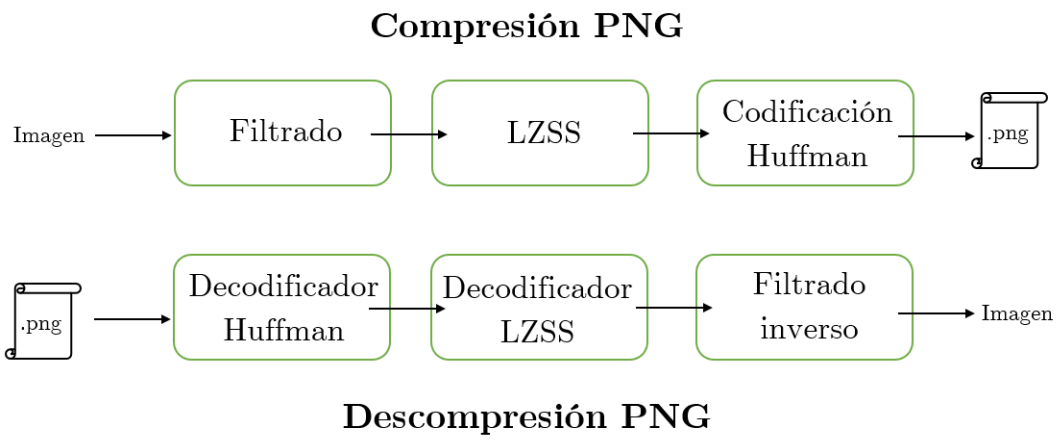



**Figura 3.8:** Esquema del algoritmo PNG

de diferentes formas. Alguno métodos hacen uso de aspectos perceptivos de la imagen, como emborronarla, mientras que otros métodos de ruido añaden o restan valores arbitrarios a los píxeles de la imagen. Algunos tipos de ruido que se pueden añadir a una imagen se puede ver en la figura 3.7.

Todos estos métodos son aleatorios, por lo que cada vez que los apliques serán diferentes. Sin embargo, cuando el problema de generalización es del tipo geométrico el ruido puede no ser efectivo, por lo que no es una panacea que resulte siempre en la mejor forma de aumentar el conjunto de entrenamiento.

## 3.4 Compresión PNG

El formato de imagen *Portable Network Graphics* PNG es un estándar en cuanto a la compresión de imágenes sin pérdida. Es el formato más utilizado y se puede ver en todos ámbitos de la informática, desde el desarrollo de aplicaciones web al de videojuegos. Este formato de compresión es muy potente, ya que, a pesar de no perder nada de los datos originales, es capaz de conseguir proporciones de compresión muy altas, como se ve en la figura 2.1.

Cabe destacar que el formato PNG es una especificación y que no detalla como debe implementarse. Por lo tanto, diferentes implementaciones pueden dar diferentes tamaños de compresión, pero al final cualquier imagen en formato PNG puede ser descomprimida por cualquier otra implementación que se ajuste al estándar definido por el World Wide Web Consortium [2]. Todas las implementaciones seguirán el esquema presentado en la figura 3.8.

El proceso de compresión que utiliza PNG tiene dos partes. La primera parte es un preproceso de los datos llamado filtrado, mientras que en la segunda parte es donde se hace la compresión de los datos en sí.

### Filtrado o Predicción

En la fase de filtrado de los datos lo que se pretende es conseguir una representación de los píxeles a partir de los que los que los rodean. Este tipo de proceso puede entenderse como, a partir de una serie de valores, conseguir una serie de las diferencias entre estos

| Tipo | Proceso |
|---|---|
| Ninguno | Devuelve los valores igual |
| Resta | Devuelve la diferencia entre cada byte y el byte del píxel a la izquierda |
| Arriba | Como en la resta pero con el píxel superior en vez de el de la izquierda |
| Media | Usa la media de los píxeles superior y anterior para calcular la diferencia |
| Paeth | Utiliza los tres píxeles más cercanos para calcular una función lineal |

**Tabla 3.1:** Tipos de filtros usados por PNG en la fase de filtrado como son explicados en la recomendación del World Wide Web Consortium [2]

valores. Un ejemplo simplificado sería el siguiente

$$v = [20, 23, 26, 29, 30, 31, 32, 33]$$
$$\hat{v} = [20 - 0, 23 - 20, 26 - 23, 29 - 26, 30 - 29, 31 - 30, 32 - 31, 33 - 32]$$
$$\hat{v} = [20, 3, 3, 3, 1, 1, 1, 1]$$

donde los datos originales $v$ se representan como la diferencia entre los valores consecutivos en $\hat{v}$.

El objetivo de este proceso es conseguir el mayor número de valores duplicados como sea posible, ya que este tipo de datos es mucho más fácil de comprimir. Es decir, se espera que, a partir de los datos de los canales de las imágenes, se encuentre alguna correlación lineal que incremente de forma constante, para que luego el compresor tenga una mayor eficiencia. Este proceso se aplica por canal de color, lo que permite al formato PNG amoldarse a diferentes espacios de color como la inclusión del canal *alpha* para la transparencia de manera sencilla.

En el formato PNG, el primer paso del filtrado consiste en seccionar la imagen por filas, a las que se les aplicará uno de los cinco filtros que este tiene en su especificación. El motivo por el que PNG tiene diferentes filtros es para conseguir el mayor número de datos repetidos. Algunos de ellos hacen uso de los datos de la capa inmediatamente superior para conseguir desvelar correlaciones lineales más complejas. Los cinco tipos de filtros se pueden ver en la tabla 3.1.

El filtro que se elija para cada fila se aplicará a todos los canales de la misma forma, pero a cada canal individualmente. De esta forma, la nueva representación de cada fila vendrá dada por el tipo de filtro que se haya utilizado en esa fila, representado con un valor de 0 a 4, seguido por los datos filtrados de cada uno de los canales.

La forma en la que se elije el filtro que se usa en cada una de las filas es un proceso complicado. Aunque podría parecer que probar todos los filtros y escoger el que da mejores resultados tras la compresión sería la mejor forma, este proceso de fuerza bruta es demasiado costoso. En vez de esto muchas implementaciones de PNG utilizan unas reglas predefinidas, que intentan pronosticar cuál de los filtros es el óptimo. Estas reglas se construyen a partir de la experimentación con muchos tipos de imagen.

### Compresión *DEFLATE*

El siguiente paso, una vez los datos han sido filtrados, es el de la compresión. Para este proceso se utiliza un algoritmo parecido al algoritmo de compresión *Lempel-Ziv compression*, llamado *DEFLATE*. Este se puede dividir en dos partes: una que elimina cadenas duplicadas y otra que hace uso de códigos de Huffman para sustituir los símbolos más utilizados por representaciones más cortas y los menos utilizados por representaciones más largas.

El proceso de eliminación de duplicados es el algoritmo de compresión *LZSS*. La idea básica de este algoritmo son las referencias inversas. Esto consiste en sustituir cadenas duplicadas por una referencia a la posición donde esta se ha encontrado previamente, especificando la distancia a la que se encuentra y la longitud de esta cadena. Un ejemplo de cómo esto se haría en una cadena de texto se puede ver aplicado aquí (texto extraído de [20]):

Volvía a ser de noche. En la posada Roca de Guía reinaba el silencio, un silencio triple.

↓

Volvía a ser de noche. En la posada Roca de Guía reinaba el silencio, un(14,9) triple.

donde la cadena « silencio» puede ser sustituida por una representación más corta, especificando el número de caracteres que se debe retroceder (14) y el número de caracteres que constituyen esta cadenas (9). En el caso de PNG, como este trabaja con bytes en vez de caracteres, se cuenta el número de bytes que se debe retroceder con la distancia y el número de bytes que componen la cadena.

Para señalizar la referencia, PNG utiliza 8 bits para la longitud y 15 para la distancia. En total eso hace que se pueda apuntar a una cadena de píxeles a una distancia de, como máximo, 32,768 bytes y con una longitud mínima de tres bytes y 258 como máximo. Este algoritmo es realmente la parte que más cuesta de computar de la compresión, sin embargo, es posible reducir la ventana en la que se comparan las cadenas duplicadas para reducir este tiempo a cambio de una menor compresión, aunque igualmente compatible. PNG utiliza una ventana máxima de 32768 bytes [2].

Una propiedad interesante, aunque no intuitiva, de este proceso es que representar varias repeticiones de una misma cadena es muy sencillo. Para ello, se puede utilizar una referencia con una longitud más larga que la distancia de la referencia. Así se vería un ejemplo de una repetición en unos versos de una canción [18]:

*When are these colonies gonna rise up? When are these colonies gonna rise up? When are these colonies gonna rise up? When are these colonies gonna rise up?*

↓

*When are these colonies gonna rise up?* (39, 117)

El segundo paso de la compresión es reemplazar los símbolos más utilizados por representaciones más cortas de estos. A efectos prácticos este proceso construye un árbol binario, donde las hojas del árbol de menor altura son los símbolos más usados y los caminos a dichas hojas son los códigos usados en la versión comprimida. De esta forma los símbolos más usados obtienen una representación menor. El resultado final es una cadena de los códigos y un árbol de Huffman para descifrar dicha cadena.

#### Descomprimir PNG

Para la descompresión de la imagen se aplican los procesos opuestos, ya que son completamente reversibles. Primero, a partir de la cadena de códigos Huffman y su correspondiente árbol, se reconstruye la cadena resultante de la compresión *LZSS*. Se utilizan los códigos de referencias para reconstruir las filas por canal resultante del filtrado. Como el filtrado añade un código para indicar qué tipo de filtrado se aplicó a dicha cadena, se puede aplicar el proceso inverso para recuperar los datos originales, reconstruyendo así la imagen original.

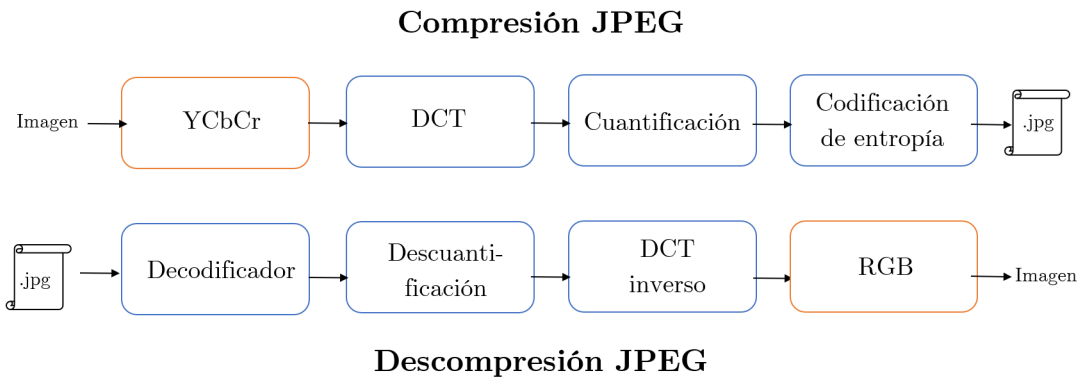



**Figura 3.9:** Esquema del algoritmo de compresión JPEG

### Proporción de compresión de PNG

PNG es el formato de compresión sin pérdida más utilizado en el mercado actualmente. Su alta velocidad de compresión y descompresión van acompañados de una comprobación de suma de control lo hace perfecto para su uso en aplicaciones web. Además, la proporción de compresión que obtiene PNG es considerablemente buena, aunque no es la mejor de los métodos de compresión sin pérdida de imágenes, como se puede ver en la figura 2.1.

## 3.5 Compresión JPEG

El desarrollo del formato de compresión JPEG se inspira en la forma de percibir del ojo humano, que es menos sensible al color que a la intensidad de la luz y también menos sensible a frecuencias más altas. JPEG utiliza estas propiedades para proporcionar un algoritmo de compresión que minimiza el cambio de estas características entre la imagen original y la resultante tras la compresión, mientras que se reduce el tamaño del archivo enormemente.

### Transformación del espacio de color

Para conseguir este propósito JPEG utiliza un espacio de color diferente a RGB, $YC_bC_r$. Al utilizar este espacio de color se puede separar la intensidad de la luz de los componentes de croma (color), permitiendo aplicarle un tratamiento diferente a cada uno de los canales. Para encontrar más información acerca de espacios de color y de cómo se transforma de RGB a $YC_bC_r$ se puede leer la sección 3.6.

### Reducción de resolución

El primer paso de compresión que se aplica es una reducción de resolución a los canales de croma $C_b$ y $C_r$. El nivel de la reducción de resolución puede ser cambiado para producir diferentes resultados y existen diferentes proporciones: por ejemplo 4:4:4, que no aplica ninguna reducción de resolución, 4:2:2, que aplica la reducción con dos píxeles adyacentes horizontalmente, y 4:2:0, que reduce la resolución vertical y horizontalmente. Reduciendo la resolución de esta forma es posible reducir el tamaño de los datos de manera muy efectiva mientras que se pierde relativamente poca información. Por ejemplo,

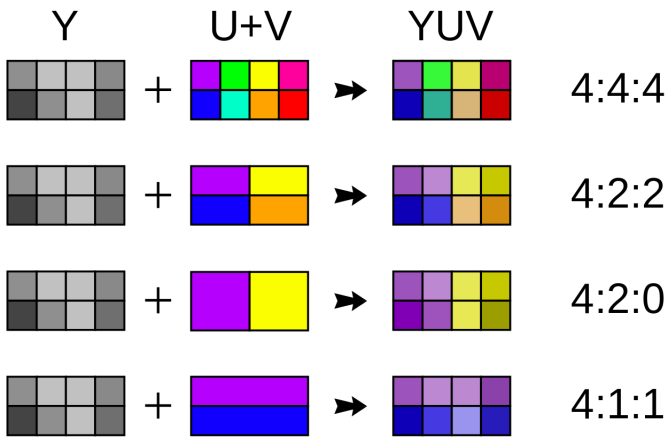



**Figura 3.10:** Reducción de resolución usando *subsampling* por [7]

la configuración de compresión más utilizada, que es 4:2:0, consigue una reducción del 50 % mientras que se pierde relativamente poca información, al menos par el ojo humano.

Además, cabe tener en cuenta que esta reducción de resolución se puede aplicar de diferentes formas. Las formas más comunes son *downsampling*, el cual escoge el color del píxel haciendo la media de los elegidos, y *subsampling*, que, como se muestra en la figura 3.10, escoge el valor del píxel en la esquina superior izquierda.

Cabe destacar, que a partir de este paso todos los demás también se aplican de forma separada a cada uno de los canales de color. Además, la imagen también se separa en una cuadrícula de bloques de 8x8, a los que se les aplica los siguientes pasos independientemente. Cuando una de las dimensiones de la imagen, o ambas, no es divisible por este factor de ocho, el resto de la cuadrícula se rellena con datos arbitrarios, aunque una práctica común es la de repetir los píxeles, para evitar producir imperfecciones en el resultado final.

### Transformada Discreta de Coseno

En este apartado se tratan los bloques de imagen como señales. Al aplicar DCT se convierte cada bloque de una serie de amplitudes a una serie de frecuencias de funciones de coseno. Para ello, primero se centran los datos a cero, de forma que en vez de tomar valores en el rango [0,255] estén en el rango [-128,127]. A la matriz resultante se le aplica la transformación de DCT de dos dimensiones adaptada a los bloques 8x8:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{u\pi}{8}\left(x+\frac{1}{2}\right)\right]\cos\left[\frac{v\pi}{8}\left(y+\frac{1}{2}\right)\right]$$

donde

- $u$ es la frecuencia espacial horizontal, con enteros en el rango [0,8)
- $v$ es la frecuencia espacial vertical, con enteros en el rango [0,8)
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{si } u\text{=0} \\ 1, & \text{en otro caso} \end{cases}$
- $g_{x,y}$ es el valor del píxel en las coordenadas $(x, y)$

- $G_{u,v}$ es el coeficiente DCT en las coordenadas $(u, v)$

Tras esta transformación la esquina superior izquierda de la matriz $G_{u,v}$ tendrá siempre un valor mucho más alto que los demás, mientras que los otros valores serán mucho más pequeños, algunos muy cercanos al cero. Esto es debido a la tendencia de DCT de trasladar la mayoría de los pesos a esa esquina. Luego el proceso de cuantificación acentuará más este efecto.

**Cuantificación digital**

Tras el paso de DCT, la matriz resultante cuenta con un 64 números de coma flotante. Representar estos datos requiere más espacio de lo que se usaba antes para representar 64 enteros en el rango [0,255]. Para solucionar eso lo que se hace ahora es cuantificar los valores, dividiendo cada uno de ellos por una constante y convirtiéndolos a enteros.

El bloque resultante de DCT es dividido elemento por elemento con una matriz de cuantificación $Q$, que puede ser cambiada para obtener calidades de compresión diferentes. Como la mayoría de los valores con magnitud grande están agrupados en la esquina superior izquierda de la matriz $G_{u,v}$, la matriz de cuantificación toma valores más pequeños en esa misma esquina, para reducir menos su magnitud y valores más grandes en el resto de esquinas para obtener valores más cercanos a cero.

Finalmente, todos los valores de la matriz resultante son redondeados al entero más cercano, obteniendo así la matriz $C_{u,v}$. En esencia, la operación de cuantificación se puede resumir en la siguiente expresión:

$$C_{u,v} = \left\lfloor \frac{G_{u,v}}{Q_{u,v}} \right\rceil$$

**Codificación de entropía**

El paso final de la compresión JPEG es la aplicación de un compresor de entropía. Este compresor tiene dos partes: una codificación *Run-Length Encoding*, especial para JPEG, y, entonces, una codificación con códigos Huffman.

Para ello, primero se reordenan los valores de la matriz cuantificada $C$ en zigzag, empezando por la esquina superior izquierda y terminando por la inferior derecha, maximizando así la probabilidad de ordenar una gran cantidad de ceros seguidos. Entonces, dada esta nueva representación como cadena de enteros, se aplica la codificación *Run-Length Encoding*. En su forma normal esta toma cadenas de valores repetidos y los comprime como una pareja, representado el valor y la cantidad de veces que se repite. Sin embargo, en JPEG esto se amplia añadiendo un tercer elemento que indica cuantos ceros hay antes de dicho valor y utilizando la tupla (0,0) para indicar que el resto de valores de este bloque son ceros. Un ejemplo de cómo esto se vería en una serie de enteros es este:

-26, 0,0,0, -2, -6, 0, 0, 0, 0, 1, 0, 0, 0
↓
-26,[3,2,-2],[0,3,-6],[4,1,1], [0,0]

donde el primer valor, -26 en este caso, no se comprime de la misma forma y el resto de tuplas tienen tres valores [*RUNLENGTH*,TAMAÑO,AMPLITUD]. Estos tres valores tienen los siguientes significados:

- *RUNLENGTH*: es el número de ceros antes de este valor
- TAMAÑO: es el número de bits necesarios para representar el valor $x$
- AMPLITUD: la representación del valor $x$

Entonces, igual que en PNG, JPEG codifica el patrón generado usando codificación de Huffman. El estándar de JPEG proporciona una tablas de Huffman para construir los árboles, aunque el codificador puede elegir construir sus propias tablas a partir de los patrones obtenidos. Este sería el resultado final del archivo JPEG y lo que se guardaría en memoria.

**Descompresión JPEG**

Para descomprimir un archivo JPEG se tienen que seguir los pasos de forma inversa, como se muestra en la figura 3.9. La codificación de entropía es totalmente sin pérdida, por lo que se puede recuperar la representación anterior sin problema. Con ello, se reconstruye el bloque y se deshace la cuantificación, multiplicando los valores por los coeficientes de cuantificación.

El siguiente paso es deshacer el DCT con una DCT inversa bidimensional para recuperar las amplitudes originales en el rango [-128,127]. Esto se hace aplicando la siguiente ecuación:

$$g_{x,y} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}\alpha(u)\alpha(v)G_{u,v}\cos\left[\frac{u\pi}{8}\left(x+\frac{1}{2}\right)\right]\cos\left[\frac{v\pi}{8}\left(y+\frac{1}{2}\right)\right]$$

- $x$ es la fila del píxel, como enteros en el rango [0,8)
- $y$ es la columna del píxel, como enteros en el rango [0,8)
- $v$ es la frecuencia espacial vertical, con enteros en el rango [0,8)
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{si } u\text{=0} \\ 1, & \text{en otro caso} \end{cases}$
- $g_{u,v}$ es el valor del pixel en las coordenadas $(x, y)$
- $G_{u,v}$ es el coeficiente DCT en las coordenadas $(u, v)$

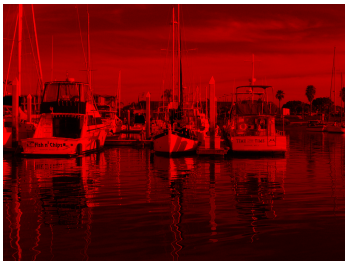
a las que luego se les suma 128 para recuperar el rango de valores [0,255]. Finalmente, se aumenta la resolución de los canales de croma que hayan pasado por una reducción de resolución y se recupera el espacio de color RGB.

**Proporción de compresión de JPEG**

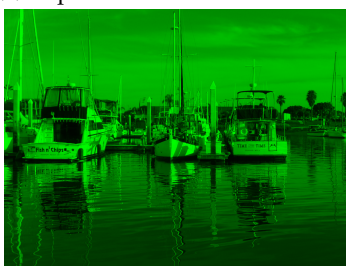
La proporción de compresión de JPEG viene condicionada por la calidad de reconstrucción que se desee. Al utilizar matrices de cuantificación con valores más altos se obtienen más ceros al final de la cuantificación, pero a su vez reconstruir las amplitudes del DCT es más inexacto.

La proporción de compresión que consigue el algoritmo JPEG estándar en diferentes calidades de reconstrucción se puede ver en la figura 2.2. En esta figura se puede ver cómo la proporción de compresión afecta a la calidad de la imagen, siendo un PSNR mayor una mejor calidad de imagen.

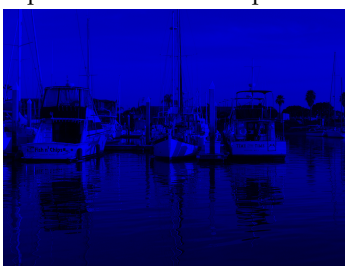
**(a)** Representación RGB mezclada



**(b)** Representación del componente rojo



**(c)** Representación del componente verde



**(d)** Representación del componente azul

**Figura 3.11:** Reproducción de una imagen digital como es originalmente y separada en sus componentes según el modelo RGB

## 3.6 Modelos de representación del color

Una imagen es una representación ordenada de píxeles, donde cada uno de estos píxeles tiene un color asociado. Determinar qué color tiene cada píxel se hace mediante un modelo de representación de color. Un modelo de representación del color es un modelo matemático que describe la forma en la que un color se representa a partir de una tupla de números de tres o cuatro valores. Además, cada modelo especifica como cada uno de estos valores debe ser interpretado, creando lo que se llama un espacio de color. Estos modelos suelen tener en cuenta cuál es la forma en la que los humanos perciben el color y cuál es la mejor forma de imitar los procesos biológicos.

### 3.6.1. RGB

El modelo RGB es un modelo de representación del color aditivo. En este, tres componentes (rojo, verde, azul) son representados en una tupla, de forma que la mezcla de las intensidades de estos tres colores representa un único color. Se puede ver un desglose de una imagen separada en sus componentes en la figura 3.11. La forma en la que se representa cada uno de estos componentes puede variar, aunque en el ámbito de la representación de imágenes digitales se suele usar un *byte* por componente, con un total de 256 valores posibles diferentes, lo que resulta en un total de $2^{24}$ colores.

Una variación de RGB en la representación de imágenes introduce un cuarto componente llamado *alpha*, que representa la transparencia de dicho color. La adición de este nuevo componente resulta en el espacio de color RGBA, comúnmente usado en la representación de imágenes digitales.

Un problema que tiene el modelo de color RGB es que, aunque define el espacio de color, no se define la forma en la que cada valor debe reproducirse. Esto implica que cada dispositivo puede mostrar colores ligeramente diferentes, convirtiendo al modelo RGB en un modelo dependiente del dispositivo usado.

### 3.6.2. YCbCr

Este es otro modelo de representación de color aditivo, pero que intenta acercarse un poco más a la forma en la que los humanos vemos, dándole uno de los tres parámetros de su tupla a la intensidad de la luz (Y). La razón de que se use este formato de representación del color es que al comprimir las imágenes se puede aplicar un proceso diferente a cada uno de los canales y, como el ojo humano es más sensible a la luz que al color, se le puede aplicar una mayor compresión a los dos componentes de color (CbCr).

YCbCr se define a partir de una transformación lineal al espacio de color RGB. La transformación es la siguiente:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0{,}2126 & 0{,}7152 & 0{,}0722 \\ -0{,}1146 & -0{,}3854 & 0{,}5 \\ 0{,}5 & -0{,}4542 & -0{,}0458 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1{,}5748 \\ 1 & -0{,}1873 & -0{,}4681 \\ 1 & 1{,}8556 & 0 \end{bmatrix} \begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix}$$

## 3.7 Métricas de calidad de imágenes

Una imagen tiene muchas propiedades que se pueden tener en cuenta a la hora de determinar cómo de similares son dos de ellas. Tamaño, histograma de colores y formas son algunas de las propiedades que se pueden usar en la comparación de dichas imágenes. Sin embargo, métricas tan generales son una mala aproximación de su similitud.

Otra cosa a tener en cuenta de la imágenes es que estas tienen una estructura ordenada. Una imagen es una matriz de tuplas de un espacio de color y la posición dentro de esta matriz es igual o más importante, en cuanto a la similitud, que el color exacto que se represente. En las métricas de similitud de imágenes más utilizadas en la comparación de algoritmos de compresión de imágenes con pérdida esta propiedad se explota en mayor o menor medida, pero siempre se tiene en cuenta.

Existen también métodos de similitud de imágenes que utilizan el contenido de las imágenes para determinar cuan parecidas son. Estas se pueden usar para determinar qué dos camisetas se parecen más dadas las imágenes de ellas, entre otras cosas. Estas medidas de similitud pueden ser conseguidas utilizando técnicas de extracción de información explicadas en las secciones 3.1 y 3.2, pero no es el enfoque que se le da en este estudio.

Existen muchos tipos de métricas de calidad de imagen. Algunas, como *MSE* y *PSNR* pertenecen a las métricas de fidelidad objetiva, mientras que otras, basadas en el sistema de percepción humano como *SSIM*, pertenecen a la categoría de métricas de fidelidad subjetiva [25]. En total existen muchas más métricas, pero a día de hoy ninguna se ha considerado óptima.

### 3.7.1. Error medio cuadrático - *MSE*

El error medio cuadrático o *Mean Square Error* en inglés, calcula la distancia euclídea media de todos los píxeles de la imagen. Esta métrica es muy simple y fácil de calcular, reduciendo la carga computacional. Un valor menor de *MSE* se considera mejor, ya que

indica una menor distancia entre los valores de la imagen original y la distorsionada. Dadas dos imágenes $A$ y $B$ de dimensiones $m \times n$ esta métrica se calcula con la siguiente formula:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [a_{ij} - b_{ij}]^2$$

### 3.7.2. Proporción máxima de señal a ruido - *PSNR*

Cuando el objetivo es encontrar cual es la diferencia entre la potencia máxima de la señal y la del ruido distorsionador que afecta a la lealtad de la reconstrucción, una métrica muy común es *Peak Signal-to-Noise Ratio* (*PSNR*). Esta métrica construye sobre el *MSE* y tiene en cuenta también la escala de representación. Dado que la escala de valores de la señal suele tener un rango muy amplio, se utiliza una transformación logarítmica para esta. En el caso de una imagen que usa el espacio RGB con 24 bits por muestra, el valor máximo que puede toma su señal sería de $2^{24}$.

Dadas dos imágenes $A$ y $B$, para calcular el *PSNR* es primero necesario calcular el *MSE*. Además, se necesita también el valor máximo de señal $MAX$. La formula que define el valor *PSNR* es la siguiente:

$$\begin{aligned} PSNR &= 10 \log_{10}(\frac{MAX^2}{MSE}) \\ &= 20 \log_{10}(MAX) - 10 \log_{10}(MSE) \end{aligned}$$

A diferencia de *MSE*, a la hora de comparar el rendimiento de dos compresores diferentes sobre las mismas muestras, un valor mayor de *PSNR* es siempre mejor. Esto se debe a que se calcula la diferencia entre el error y la potencia de la señal. Un mayor error *MSE* dará una menor diferencia con la potencia de la señal y, por lo tanto, un menor valor de *PSNR*.

### 3.7.3. Similitud estructural - *SSIM*

El índice de similitud estructural o *Structural similarity* es un sistema basado en la visión humana que se considera una mejora respecto a las métricas de *MSE* y *PSNR* [29]. Utiliza procesos más complejos de procesamiento de imágenes y se calcula a través de diferentes ventanas de las imágenes de la siguiente forma:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 - c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

Donde las regiones $x$ y $y$ de las imágenes tienen una media de valores $\mu_x$ y $\mu_y$, una varianza $\sigma_x^2$ y $\sigma_y^2$ y una covarianza $\sigma_{xy}$. Además, la constante $c_1$ se define como $c_1 = (k_1 L)^2$ y la constante $c_2$ se define como $c_2 = (k_2 L)^2$, donde $L$ es el rango de valores de un píxel (255 por defecto), $k_1 = 0{,}01$ y $k_2 = 0{,}03$ (ambos por defecto también) [29].

# CAPÍTULO 4
# Metodología empleada

En esta sección se explica cuales son los métodos aplicados en el trabajo para llevarlo a cabo. Se hace una descripción de todos los materiales usados, lenguajes de programación, librerías, el editor de código y otros programas auxiliares. Seguidamente, se explican las herramientas usadas para el entrenamiento de los modelos de *autoencoder* usados en los compresores y el significado de cada uno de los parámetros usados en la configuración de dichos *autoencoders*.

## 4.1 Herramientas empleadas

Para el desarrollo de este proyecto se ha utilizado una serie de herramientas, todas necesarias para que este sea completado. En esta sección se detallan todas las herramientas que se utilizan, las razones por las que se han elegido y los usos que se les ha dado.

El lenguaje de programación elegido para el desarrollo es *Python*, en concreto la versión 3.9.13 de *Pyhton*. *Python* es un lenguaje sencillo de aprender, de usar para el desarrollo y con innumerables librerías dedicadas. Además, existe una gran comunidad, con lo que es fácil encontrar ayuda sobre problemas o dudas que se puedan tener, porque probablemente alguien ya haya tenido ese problema alguna vez.

Una de las razones concretas por las que se ha usado *Python*, y la de más peso, es la existencia de librerías específicas para *Machine Learning*. Existen varias librerías que se pueden usar para el desarrollo en *Deep Learning*, como *Keras* y *TensorFlow*, pero en concreto en este proyecto se hace uso de *PyTorch*.

Una vez el lenguaje de programación ha sido elegido es necesario instalar el intérprete, que nos permitirá ejecutar los programas escritos en nuestro sistema Windows, aunque también existen versiones para Linux y MacOs. Esto se puede hacer de varias formas: desde la tienda de *Microsoft Store*, descargando el instalador de fuera de línea desde la página web oficial o usando un entorno como Anaconda, por ejemplo. Existen otras formas de utilizar *Python*, pero en este caso se hizo uso de la tienda de *Microsoft Store* para descargar la última versión disponible de *Python* 3.9.

Para el entrenamiento de los modelos es necesaria una gran cantidad de ejemplos. Un gran *dataset* nos dará una mayor cantidad de ejemplos desde donde el modelo puede aprender, adquiriendo mayor precisión de reconstrucción. Un *dataset* muy común en la comunidad científica para la clasificación de imágenes es *ImageNet* [8].

Sin embargo, debido a su gran tamaño, más de 14 millones de imágenes a fecha de Junio de 2022, se opta por usar una versión reducida de esta *ImageNet-r*. Este es un conjunto de imágenes etiquetadas con etiquetas *ImageNet*, obtenidas mediante la recopilación de

arte, dibujos animados, devianart, graffiti, bordados, gráficos, origami, pintura, patrones, objetos de plástico, peluches, esculturas, bocetos tatuajes, juguetes y videojuegos de las clases de *ImageNet*. En total tiene representaciones para 200 clases de *ImageNet*, resultando en 30.000 imágenes [15].

El editor elegido para escribir los *scripts* ha sido Visual Studio Code, *VS Code* para abreviar. Este editor se ha elegido por su capacidad de instalar paquetes que, entre otras cosas, resaltan la sintaxis de *Python*, ayudan completando automáticamente pequeñas partes de código como variables o nombres de funciones y su integración con las libretas interactivas de *Jupyter Notebooks* y el kit de herramienas *TensorBoard*.

*Jupyter Notebooks* es un formato de documento de texto basado en JSON, capaz de incluir código, texto narrativo, ecuaciones y texto enriquecido. Además, su diseño flexible nos permite configurar y reorganizar la forma de trabajo y está pensando para ser usado, entre otras cosas, en aplicaciones de *Machine Learning*. Por otro lado, *VS Code* cuenta con un paquete para el uso y visualización de *TensorBoard*, que también se usará. Este es un kit de herramientas de visualización originalmente de *Tensorflow*, pero que tiene integración con *PyTorch*.

Todas las pruebas y el desarrollo del trabajo se han hecho desde un equipo Windows 10 con un procesador Intel i7 con velocidad base de 2.30 GHz y 4 núcleos, 16 GB de RAM y una GPU NVIDIA Quadro P520. Esta GPU cuenta con la plataforma de computación CUDA, que acelera la velocidad de cálculo paralelizando partes del proceso de entrenamiento.

CUDA son las siglas en inglés de Arquitectura Unificada de Dispositivos de Cómputo, que hace referencia a una plataforma de computación en paralelo, incluyendo un compilador y un conjunto de herramientas de desarrollo creadas por Nvidia, que permiten a los programadores usar una variación del lenguaje de programación C (CUDA C) para codificar algoritmos en GPU de Nvidia[1]. A través de *wrappers* es posible utilizar estas librerías en *Python*, por lo que librerías como *PyTorch* pueden hacer uso de CUDA cuando está disponible.

### 4.1.1. Librerías de Python empleadas

Seguidamente se detallan las librerías de *Python* que se han utilizado, cómo instalarlas y el uso que se les ha dado. Todas estas librerías han sido necesarias para el desarrollo del trabajo y sin ellas habría sido difícil o imposible completar alguna de sus partes. Sin embargo, en muchos casos *Python* cuenta con alternativas que tendrán la misma funcionalidad y podrían reemplazarlas.

**NumPy**

*NumPy* es una librería de Python para el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos [14, 4]. En este trabajo se usa para la manipulación de la imágenes una vez han sido cargadas de disco, troceando la imagen una cuadrícula y en el proceso inverso. La librería se instala con el siguiente comando.

```
pip3 install numpy
```

**PIL: Python Image Library**

*PIL* es una librería desarrollada con el objetivo de incluir toda la funcionalidad relevante en cuanto a imágenes en *Python* [26]. Esta librería se usa en el proyecto únicamente para cargar las imágenes desde disco y para guardarlas.

```
pip3 install PIL
```

**PyTorch**

*PyTorch* es la librería que se usa para el diseño y entrenamiento de los *autoencoders*. Esta proporciona un marco de desarrollo para *Machine Learning*, con un entorno de desarrollo rápido y flexible [19]. Como se ha explicado en la sección 4.1 existen otras librerías que pueden cumplir las mismas funcionalidades con la misma eficiencia. Además, se hace uso también de la librería *Torchvision*, una parte del proyecto de *PyTorch*, que nos da acceso a una base de datos de *datasets*, arquitecturas de modelos y transformaciones comunes para imágenes.

Instalar la versión correcta de estos paquete es muy importante, en este caso se usó la versión 1.10.0+cu113 de *PyTorch* y la versión 0.11.1+cu113 de *Torchvision*. Estas versiones cuentan con la capacidad de usar el kit de herramientas de CUDA 11.3. Para instalarlas se ejecuta el siguiente comando en la terminal *PowerShell*:

```
pip3 install torch==1.10.0+cu113 torchvision==0.11.1+cu113 -f https://download.pytorch.org/whl/torch_stable.html
```

**os: Operating System**

Esta es una librería nativa de *Python* que nos permitirá interactuar con el sistema operativo. En este proyecto se utiliza para hacer las listas de las rutas de las imágenes utilizadas durante el entrenamiento y para manipular las rutas de las imágenes que se quieren comprimir. Concretamente estas funciones son `listdir`, `join` y `isdir`. Como es nativa de *Python* no hace falta instalarla.

**Matplotlib**

*Matplotlib* es una librería para crear visualizaciones estáticas, dinámicas e interactivas desde *Python*[16]. Esta librería nos permite crear gráficas como en las Figuras 4.2 y 4.3. Además, cuenta con integración dentro de las Libretas Jupyter, con lo que se pueden insertar sus figuras en el propio archivo. La instalación de esta librería se puede hacer con el siguiente comando de terminal:

```
pip3 install matplotlib
```

**scikit-image**

*Scikit-Image* es un librería de algoritmos para el procesamiento de imágenes en Python [27]. Esta librería será usada para el preprocesamiento de las imágenes en algunos de los experimentos. La instalación de la librería desde la terminal se hace de la siguiente forma:

```
pip3 install scikit-image
```

mientras que para usarla en un *script* de Python se importa de la siguiente manera:

```
import skimage
```

## 4.2 Algoritmo propuesto

El algoritmo propuesto en este trabajo toma una estructura parecida a la de JPEG, sustituyendo la parte de la transformada de coseno (DCT) por una transformación no lineal, determinada por el compresor de un *autoencoder*. El objetivo de este intercambio es introducir una transformación que esté mejor adaptada a la estructura de una imagen. Un esquema de los pasos del algoritmo propuesto se puede ver en la figura 4.1.

Inspirado en el algoritmo de JPEG se trocean las imágenes en bloques de 8x8 y se les aplica la transformación con el codificador de un *autoencoder* a cada uno de esos bloques.

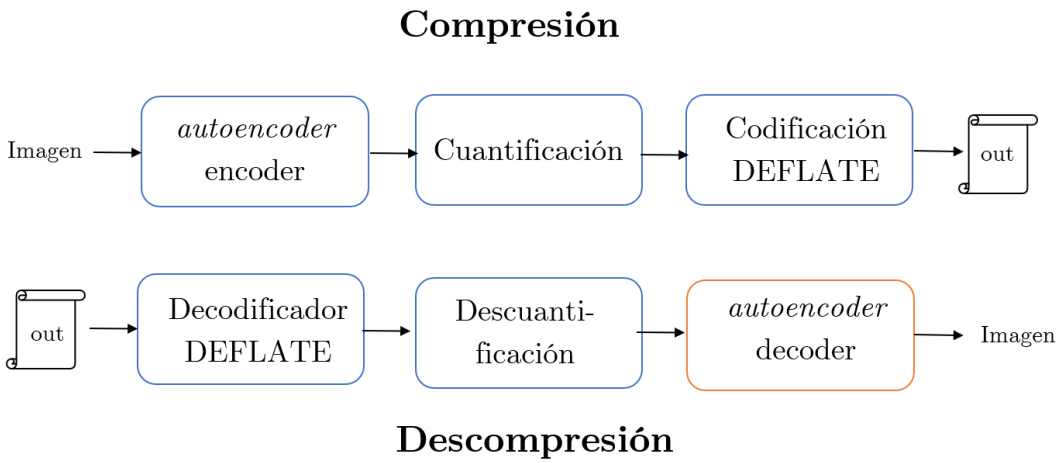



**Figura 4.1:** Esquema del algoritmo propuesto

Por lo tanto el resultado de la codificación de uno de estos bloques sería una cadena de valores enteros. Seguidamente, se cuantifican estos valores, obteniendo el rango máximo que estos ocupan para reconstruirlos más tarde y escalándolos a enteros en el rango [0,255], para que puedan ser almacenados en un byte cada uno.

Finalmente, estas cadenas de valores se comprimen con el algoritmo de compresión sin pérdida DEFLATE. Este paso está inspirado en algoritmo PNG, y se aplica a forma de recoger los patrones que se repiten en los datos y reducir el tamaño de su representación.

Para descomprimir estos datos se hacen los pasos inversos. Primero, se aplica la descompresión de DEFLATE, obteniendo las cadenas que representan cada uno de los bloques. Dichas cadenas, entonces, se introducen en la parte decidificadora del *autoencoder*, recuperando los bloques en una representación similar a la original. Finalmente, dichos bloques se reestructuran para formar la reconstrucción de la imagen original.

## 4.3 Otras herramientas creadas para el trabajo

Además de las herramientas de terceros, se han creado otras herramientas para llevar a cabo el desarrollo del trabajo. Estas herramientas están basadas en las librerías mencionadas en las sección 4.1 y ayudan en el desarrollo del trabajo simplificando tareas.

Estas herramientas se usan específicamente para la creación de los lotes o *batches* de bloques de las imágenes, usados durante el entrenamiento, y para el entrenamiento en sí de los modelos. Además, conforme las necesidades de los experimentos cambian se han desarrollado alternativas que también se explican en esta sección.

### 4.3.1. Generación de lotes de bloques para el entrenamiento

Para entrenar cada uno de los *autoencoders* usados en el compresor, se extraen de las imágenes bloques cuadrados del tamaño deseado en cada caso. Para ello, se usan las imágenes del conjunto de entrenamiento de *ImageNet-r*.

Primero, se genera una lista de todos las rutas de las imágenes contenidas en el conjunto de entrenamiento de *ImageNet-r*. Para ello, se itera por todas las clases contenidas dentro del conjunto de entrenamiento, ya que la clase de cada imagen no es realmente importante, solo su contenido. Esta lista de rutas se guarda con el nombre `image_path_list`.

**Figura 4.2:** Ocho bloques de tamaño 8x8 píxeles tomados de 8 imágenes aleatorias del *dataset Imagenet-r*

Haciendo uso de `image_path_list`, creamos nuestro propio generador de lotes con `get_tile_batch`, el cual es capaz de generar el número de bloques que se pida de imágenes aleatorias de esa lista. Esto se consigue creando un generador de Python, un tipo especial de función que funciona con un bucle infinito, que se reanuda cada vez que la función es llamada. La clave de estos generadores reside en la palabra clave `yield`, capaz de hacer lo anteriormente descrito, finalizar la ejecución de la función y devolver un resultado, pero guardando el estado en el que quedó para ser usado en siguientes llamadas.

La función `get_tile_batch` nos devuelve el generador de bloques y toma los siguientes parámetros:

1. `image_list`: la lista de rutass de las imágenes del *dataset*. Normalmente se le pasa `image_path_list`.

2. `tile_size`: tamaño del lado del cuadrado que tienen los bloques a extraer. Por defecto este tamaño es de ocho píxeles.

3. image_number: determina el número de imágenes de nuestro *dataset* que se cargan aleatoriamente de disco para extraer sus bloques

4. `num_tiles`: Número de bloques que se extraerá de cada una de las imágenes. Por defecto, se extraen 100 bloques aleatorios, un tamaño razonable, teniendo en cuenta que las imágenes son todas más grandes de 256 por 256 píxeles y que los bloques cuadrados extraídos tienen un tamaño de ocho píxeles de lado.

De esta forma, el generador toma tantas imágenes como indique image_number de la lista de imágenes `image_list` y de cada una extrae tantos bloques como `num_tiles` indique, de forma aleatoria. Por ejemplo, si el generador se inicializa de la siguiente forma:

```
1 batch_maker = get_tile_batch(image_path_list, image_number=8, num_tiles=1)
```

**Listing 4.1:** Cómo crear un iterador

Cada vez que se llama a la función `next(batch_maker)`, esta devuelve una lista de ocho bloques, cada uno de una imagen diferente, ya que `num_tiles` está configurado a uno. Podemos ver un ejemplo de los bloques que esta función devuelve en la figura 4.2. Una propiedad muy interesante que tienen los generadores en *Python* es su capacidad de ser iterados por un bucle, inicialmente infinito. Esta propiedad es lo que se usa en la siguiente sección para entrenar los modelos.

Durante la experimentación, también se hará uso de una versión de `get_tile_batch` capaz de obtener dos copias de los bloques. Esta función ligeramente modificada, llamada `get_noisy_tile_batch`, se usa de la misma forma que `get_tile_batch` y devuelve, junto a los bloques de las imágenes, una versión que tiene ruido, para entrenar los modelos usando estos bloques.

### 4.3.2. Entrenamiento de modelos

Para el entrenamiento de los modelos se hace uso de la función `train_autoencoder`, que toma los siguientes parámetros:

1. `epoch_size`: determina cada cuantos lotes que se imprime información acerca del error de entrenamiento.
2. `train_loader`: es el generador de bloques que se usa en el entrenamiento. Normalmente usaremos el generador devuelto por `batch_maker`.
3. `optimizer`: el optimizador del modelo.
4. `model`: el modelo del *autoencoder* a entrenar.
5. `criterion`: la función de pérdida utilizada para calcular el error de reconstrucción.
6. `max_epochs`: el número máximo de lotes que se suarán para el entrenamiento.

La función `train_autoencoder` itera sobre el generador `train_loader`, usando la función de Python `enumerate` para llevar la cuenta de la iteración, es decir el *batch* en el que se encuentra. Dentro del bucle siguen los siguientes pasos:

```
1 for batch_number,tiles in enumerate(train_loader):
```

**Listing 4.2:** Cabecera del bucle

La cabecera del bucle carga un nuevo lote de bloques (*tiles*) que se usan para el entrenamiento. Además, se obtiene el número del tole en el que nos encontramos en `batch_number`.

```
1     tiles = tiles.to(device)
```

**Listing 4.3:** Se mandan los bloques a CUDA

Entonces, se mandan los bloques al dispositivo indicado, siendo este la GPU para una mayor velocidad de computación. Si la GPU no está disponible se mandará a la CPU.

```
1     optimizer.zero_grad()
```

**Listing 4.4:** Se reinician las pendientes

Además, se reinician las pendientes de los parámetros del modelo a cero. Esto se hace porque *PyTorch* acumula estas pendientes en sucesivos pases hacia atrás.

```
1     outputs = model(tiles)
2     train_loss = criterion(outputs, tiles.view(-1, 3 * tile_size * tile_size)
3     train_loss.backward()
4     optimizer.step()
5     loss += train_loss.item()
```

**Listing 4.5:** Pases hacia delante y hacia atrás del entrenamiento

Seguidamente, se hace el pase hacia delante, dando como resultado `outputs`, que serían los bloques reconstruidos. Se calcula el error de reconstrucción con la función de pérdida `criterion`. Antes de ello, sin embargo, debemos transformar la matriz de datos de los bloques para que tenga la misma estructura que en la salida. Para ello, hacemos uso de la función `tiles.view(-1,3 * tile_size * tile_size)`, que reorganiza los datos para que tengan una estructura de (`image_number` $\times$ `num_tiles`) por $(3 \times 8 \times 8)$, es decir, lado por lado por canales de la imagen RGB.

Una vez `train_loss` ha sido calculado, se hace el pase hacia atrás con `backward()`, que calcula las pendientes acumuladas. Con las pendientes calculadas, `optimizer.step()` actualiza los parámetros del modelo.

Finalmente, se añade la pérdida al contador `loss`, por motivos de registro. Una vez cada tantas iteraciones como indique `epoch_size` se guarda la media de la pérdida en las últimas iteraciones, para ser representadas en una gráfica de pérdida en entrenamiento, se reinicia el valor de `loss` a cero y, si se ha superado el número de iteraciones máximas indicadas al principio en `max_epochs` se finaliza el entrenamiento.

Este proceso es ligeramente diferente cuando se utiliza el generador de bloques para bloques con ruido `get_noisy_tile_batch`. Este, a diferencia de `get_tile_batch`, devuelve dos elementos cada vez que se usa su iterador. Para adaptar la función de entrenamiento, primero se modifica la línea de código de 4.2 por la siguiente:

```
1    for batch_number,(tiles,noisy_tiles) in enumerate(train_loader):
```

**Listing 4.6:** Cabecera del bucle para iterador con ruido

Tanto `tiles` como `noisy_tiles` se tienen que mandar al dispositivo CUDA si está. Finalmente, también se modifica el código de 4.5 para adaptarlo a las necesidades de la siguiente forma:

```
1    outputs = model(noisy_tiles)
2    train_loss = criterion(outputs, tiles.view(-1, 3 * tile_size * tile_size)
3    train_loss.backward()
4    optimizer.step()
5    loss += train_loss.item()
```

**Listing 4.7:** Pases hacia delante y hacia atrás para entrenar con ruido

### 4.3.3. Diseño del compresor

Para automatizar el proceso de prueba se crea una clase de *Python* para que se encargue de los pasos de separación en bloques, cálculo de la compresión del *autoencoder* y cuantificación. La estructura del compresor está formada por dos partes: la función de compresión y la de descompresión. Cada una de estas partes toma una mitad del *autoencoder* entrenado, el compresor toma el codificador y el descompresor toma el decodificador. Dichos codificadores y decodificadores deberán provenir del mismo *autoencoder*, ya que estos han sido entrenados en tándem, y los parámetros dependen el uno del otro.

El compresor es un objeto del tipo `Compressor_Decompressor`, nombre recibido por su naturaleza dual. Este objeto se inicializa con los siguientes parámetros:

1. `model_path`: ruta del *autoencoder* que se desea usar en la compresión y descompresión.

2. `model_type`: clase del modelo de *autoencoder* que se ha usado. Diferentes clases tienen diferentes estructuras que se adaptan a los parámetros cargados del modelo.

3. `chunk_size` (opcional): tamaño del lado del bloque que se usará en la segmentación de la imagen. Por defecto `chunk_size` es ocho.

4. `compression_out` (opcional): determina el número de bytes que se usan para representar cada bloque al final de la compresión. Este también debe asociarse correctamente al modelo cargado. Por defecto este se configura a ocho, pero se cambia para cada *autoencoder*.

5. `color_format` (opcional): determina el formato en el que se debe cargar la imagen para codificarla en el compresor. Por defecto este valor es RGB, pero se acepta también $YC_bC_r$

Al inicializarse el objeto `Compressor_Decompressor` cargará el modelo de memoria y lo mandará al dispositivo *GPU*, si este está disponible. Si no está disponible se quedará en la memoria de la *CPU*. Además, el formato en el que se debe cargar la imagen, pasado como parámetro en `color_format`, se guarda para usarlo tanto en la carga de las imágenes como en el guardado, para poder convertirla al espacio de color que use el *autoencoder* y que el resultado de la descompresión sea siempre una imagen en formato RGB.

Una vez el compresor está inicializado y su modelo cargado, es posible comenzar a comprimir las imágenes. Para ello hacemos uso del método `compress_image`, que toma los siguientes parámetros:

1. `image_path`: ruta de la imagen que se desea comprimir.
2. `image_np` (opcional): imagen como objeto de `ndarray` de *NumPy*. Por si se quiere comprimir una imagen que ya está en memoria en vez de cargarlo otra vez. Por defecto este está configurado como `None`
3. `apply_scale`: Determina si el compresor aplicará un nivel de compresión extra escalando el resultado con la función `scale_array` que se explicará más adelante.

```
if not isinstance(image_np,np.ndarray):
    image_np = self.load_image_as_np(image_path)
```

**Listing 4.8:** Cargar la imagen de memoria

El primer paso que toma el compresor es determinar si debe usar una imagen que ya esté cargada en memoria o si la debe cargar él. Si el parámetro `image_np` no es del tipo `ndarray` usará `image_path` para cargar la imagen. Esta será cargada en un *array* de *NumPy*, con tres canales de color *RGB* por defecto.

```
image_size             = image_np.shape
tile_list_np           = self.segment_image(image_np,pad_type='reflect')
tile_list_tensor       = self.make_tensor(tile_list_np)
tile_list_tensor_cuda  = self.send_image_to_device(tile_list_tensor)
compressed_image_tensor = self.apply_compress_function(tile_list_tensor_cuda)
clean_c                = compressed_image_tensor.detach().cpu()
```

**Listing 4.9:** Pasos seguidos por el compresor

Seguidamente, se guarda el tamaño de la imagen original, necesaria para la reconstrucción. Esta tiene un tamaño de la forma HxWxC, siendo C siempre 3.

El paso `self.segment_image` es el encargado de separar la imagen original en bloques, utilizando una cuadrícula. Cabe destacar que, cuando la altura o anchura de la imagen no es divisible por el tamaño que indica `chunk_size`, esta imagen debe ser expandida para acomodar perfectamente la cuadrícula. Esto se consigue usando la función de *NumPy* `pad`, que expande la matriz usando la configuración que se introduce en `pad_type`, la cual configuramos a 'reflect' por defecto. Esta función transformará la matriz de su formato HxWxC a $(H \times W/(\texttt{chunk\_size} \times \texttt{chunk\_size})) \times \texttt{chunk\_size} \times \texttt{chunk\_size} \times C$. De esta forma tenemos una lista de $H \times W/(\texttt{chunk\_size} \times \texttt{chunk\_size})$ bloques de tamaño $\texttt{chunk\_size} \times \texttt{chunk\_size} \times C$, que es el formato que acepta el codificador.

Otro paso previo necesario para usar el *autoencoder* es transformar la matriz de *NumPy* a un objeto `Tensor` de *Pytorch*. Esto se hace en la función `make_tensor`, que cambia el orden

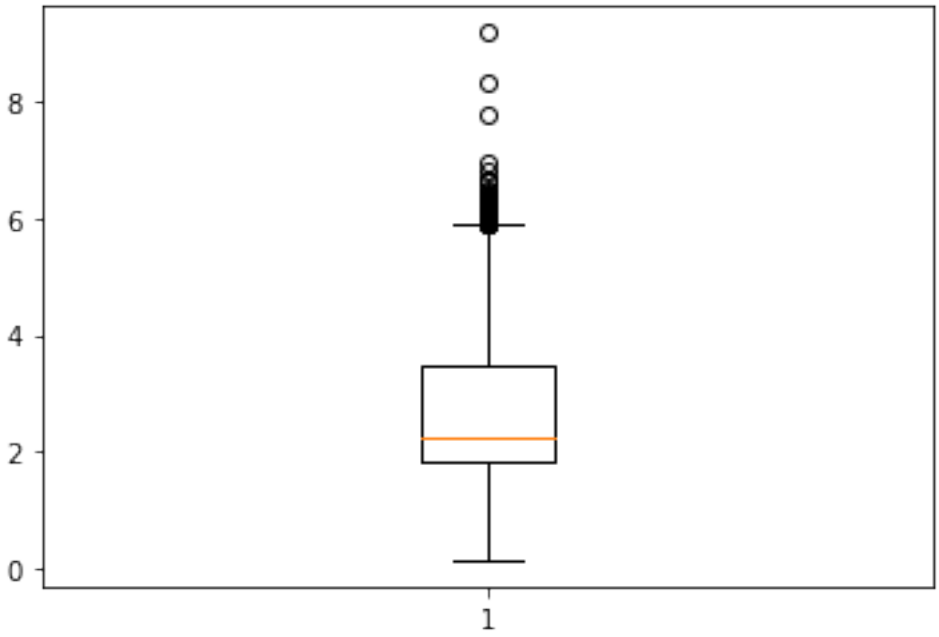



**Figura 4.3:** Representación en un gráfico de caja y bigotes de los valores del espacio latente una vez ha sido comprimida una imagen

de las coordenadas usadas al formato usado por `Tensor`, $C \times$ `chunk_size` $\times$ `chunk_size`. Finalmente, los valores de la matriz pasan de ser del tipo *int8*, enteros de ocho bits, a números de puntos flotante de 32 bits, en el rango de valores de cero a uno.

El ultimo paso previo antes de pasarlo por el codificador del *autoencoder* es mandarlo al dispositivo *GPU* si este está disponible. Entonces, finalmente, se le aplica la compresión en `self.apply_compress_function`, cuya única función es devolver una nueva representación del tamaño indicado en `compression_out` para cada bloque.

```
if not apply_scale:
    return clean_c, image_size
scaled_array, interval  = Compressor_Decompressor.scale_array(clean_c)
return scaled_array, interval, image_size
```

**Listing 4.10:** Devolver el resultado cuantificado si es necesario

Como se ha explicado en la enumeración de los parámetros de la función de compresión `compress_image`, existe un parámetro llamado `apply_scale`. Este parámetro determina si se aplica una transformación cuantificación a los resultados de la compresión con *autoencoder*. El resultado de la compresión del *autoencoder* es una matriz de valores del tipo *float32* con valores mayores a cero. Esto quiere decir que, si utilizamos ocho valores para su representación estaríamos usando un total de 32 *bytes*. Esto resulta en un nivel de compresión de cuatro bits por píxel, es decir un 16,7 % del tamaño original.

Al habilitar el escalado final se le aplica una cuantificación a la señal para representarla con 8 bits. En esencia esta cuantificación es una transformación del estado latente, interpolando estos valores de 0 a 255 y transformándolos a enteros. De esta forma conseguimos una compresión cuatro veces mayor a cambio de una diferencia de reconstrucción muy baja. En la figura 4.3 podemos ver que los valores, además de ser todos superiores a cero, se encuentran en valores inferiores a 10. Esto supone que, una vez se descuatifican, los errores de reconstrucción son del orden de $10^{-2}$, como se puede ver en la figura 4.4.

Finalmente, el método `compress_image` devuelve la representación comprimida como una tupla triple. Una matriz de las dimensiones $(H \times W/(\texttt{chunk\_size} \times \texttt{chunk\_size})) \times$

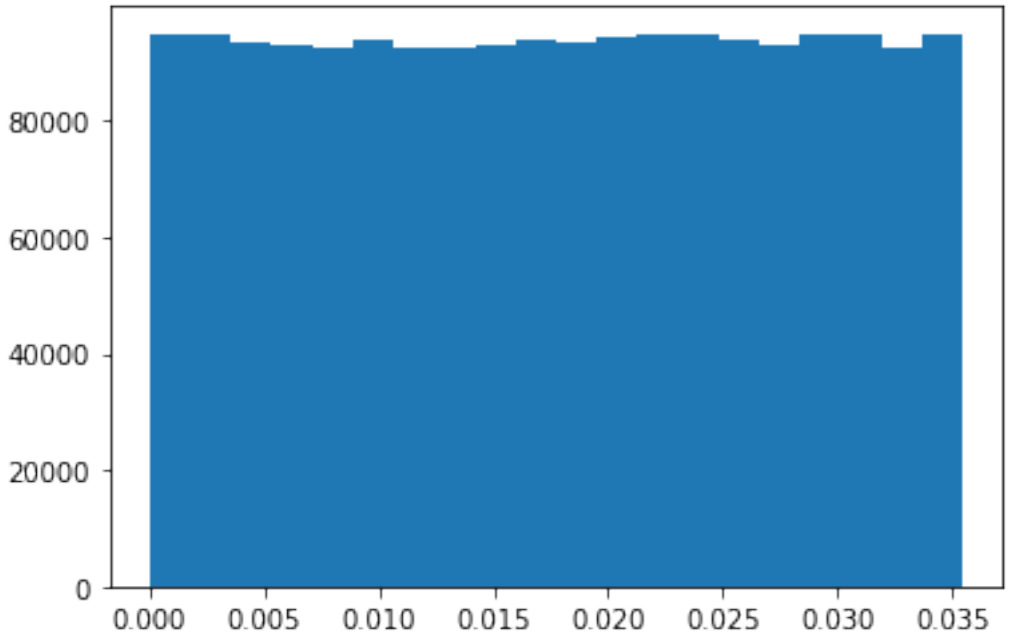



**Figura 4.4:** Representación en un histograma con 20 grupos de mismo tamaño del error de reconstrucción del espacio latente tras la decuantifiación

`compression_out` que contiene la representación latente, el intervalo de valores original antes del escalado (en formato *float* x *float*) y el tamaño de la imagen original, en el formato *int* x *int*. Esto suma ocho *bytes* extra por los dos *int* más 16 *bytes* por los dos *floats* del intervalo, haciendo un total de 24 *bytes* extra, una pequeña cantidad constante que se puede menospreciar en comparación a la representación latente de la imagen en sí.

### 4.3.4. Diseño del descompresor

El descompresor, en esencia, hace el proceso inverso que el compresor. Este se usa con el mismo *autoencoder* que en el compresor, porque estos están entrenado para trabajar en conjunto. La descompresión se hace con la función `decompress_image`, que toma los siguientes parámetros:

1. `compressed_image`: recibe la matriz con la representación comprimida de la imagen a reconstruir.
2. `interval`: intervalo de valores en la representación latente antes del escalado.
3. `destination_path`: ruta de la imagen una vez descomprimida para guardarla en disco.
4. `image_size`: tamaño original de la imagen.
5. `return_image` (opcional): determina si la imagen se guardará en disco o si, en cambio, se devolverá como una matriz de *NumPy*. Por defecto este parámetro tiene el valor *False* y se devolverá la imagen como `ndarray` de *NumPy*.

```
1    compressed_image = descale_array(compressed_image, interval)
```

**Listing 4.11:** Descuantificación

El primer paso es la descuantifiación del espacio latente. En esencia esto reconstruir la representación latente original utilizando el intervalo de valores obtenido durante la compresión. Para ello se sigue el proceso inverso que en la cuantificación del compresor, transformando los valores de de la matriz, de enteros de ocho *bits* en el rango $[0, 255]$ a números de punto flotante en el rango de `interval`.

```
compressed_image_cuda = self.send_image_to_device(compressed_image)
decompressed_tensor   = self.apply_decompress_function(compressed_image_cuda)
decompresssed_image   = self.retrieve_array(decompressed_image_tensor)
end_image             = self.rebuild_image(decompresssed_image, image_size)
```

**Listing 4.12:** Pasos seguidos por el descompresor

En estos pasos se manda la matriz al dispositivo *GPU* si está disponible y se pasa por decodificador del *autoencoder*. En este momento tenemos los valores de la imagen en bloques cuadrados con una longitud de lado indicado por `chunk_size`. Para recuperar la imagen primero los convertimos a `ndarray` de *NumPy* con `retrieve_array()` y, entonces, se reestructura para recuperar el orden de los valores que tenía en la imagen original. Adicionalmente, en `rebuild_image`, se recorta la imagen para eliminar el *padding* que hemos usado en la compresión para conseguir la cuadrícula perfectamente divisible por el valor de `chunk_size`. Finalmente, si se ha indicado con `return_image` se guarda la imagen en disco en la ruta especificada por `destination_path`, si no se guarda en disco.

### 4.3.5. Estructura de un *autoencoder*

La clase `AutoEncoder` es la clase con la cual crearemos los modelos a entrenar. Esta clase hereda de la clase `Module` de la librería `torch.nn`. Para ello, primero se importa la librería en la cabecera. Además, se importa también el módulo `torch`, para acceder a más funcionalidades que este aporta.

```
import torch.nn as nn
import torch

class AutoEncoder(nn.Module):
```

**Listing 4.13:** Cabecera de la clase *AutoEncoder*

Al heredar de la clase `Module`, `AutoEncoder` se convierte en un módulo de red neuronal de *PyTorch*, por lo cual se puede usar como componente en otras redes que se creen en el futuro. Para inicializar el `AutoEncoder` se define la función `__init__`. Esta tiene los siguientes parámetros:

1. `activation`(opcional): define la función de activación que usará la red entre cada capa lineal. Al tenerla como parámetro nos permite cambiarla de manera sencilla cuando se hacen los prototipos. Por defecto este parámetro es un módulo del tipo `ReLU`, y cualquier otra función de activación que se indique también debería ser un módulo de `torch.nn`.

2. `input_size` (opcional): determina el tamaño del bloque que se va a usar como entrada y por lo tanto el de salida. Este debe tener en cuenta el alto, ancho y número de canales. Por defecto `input_size` es $8 \times 8 \times 3$.

3. `hidden_sizes` (opcional): define el número de nodos que utiliza cada una de las capas del*autoencoder*. Si se quiere hacer un modelo con mayor profundidad se le puede dar una lista de mayor profundidad. Por defecto `hidden_sizes` es asignada una lista con los valores $[96, 8, 96]$. Esta lista debe tener una longitud impar y siempre se

toma el valor central como el tamaño de salida una vez se calcule su representación comprimida, mientras el parámetro `compressed_size` no esté habilitado.

4. `compressed_size` (opcional): es una forma rápida de definir el número de valores que se usarán como representación comprimida. Sobrescribe el valor de salida definido por `hidden_sizes`.

Durante la inicialización, `AutoEncoder` inicializa las capas lineales para que tengan las dimensiones que se han indicado en `hidden_sizes` y `compressed_size`. Además, se asigna la función de activación. Esto se hace desde la función `init_layers`, de forma que es sencillo de cambiar por cualquier otro modelo que extienda este, mejorando la modularidad. Por último, si CUDA está disponible, el modelo se manda al dispositivo.

```
    def __init__(self, activation=nn.ReLU(), input_size=3*8*8,
        hidden_sizes=[32*3,8,32*3],compressed_size=None):

        super(AutoEncoder, self).__init__()
        self.input_size = input_size
        if compressed_size != None:
            hidden_sizes[int(len(hidden_sizes)/2)] = compressed_size

        self.init_layers(activation,hidden_sizes)

        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.to(device)
```

**Listing 4.14:** Inicialización del *AutoEncoder*

Durante el entrenamiento la función del modelo que se llama primer es la función `forward`. Esta función es la que llama, en orden, las capas de codificación y decodificación. A su vez, estas capas se agrupan en dos métodos, llamadas `encode` y `decode`, que llaman las capas lineales y de activación en orden. Esto, otra vez, ha sido decidido así por motivos de modularidad, para ayudar en la creación de prototipos y, además, porque permite, una vez entrenado el modelo, aplicar la compresión de manera sencilla llamando a `encode` y luego descomprimir llamando únicamente a `decode`.

Una vez el resultado de la función definida por el modelo ha sido calculado, se calcula el error mediante la función de pérdida, como es explicado en la sección 4.3.2 y luego se hace el pase hacia atrás, para calcular los gradientes. En *PyTorch*, no es necesario definir un método de pase hacia atrás, ya que el propio marco lo infiere a partir del pase hacia delante, la definido en `formward`.

Otros métodos que se han creado para el manejo más sencillo de estos modelos son `save_model` y `load_model`, que son simples funciones para guardar y cargar de disco los modelos. `save_model` es principalmente usado una vez el modelo ha sido entrenado, mientras que `load_model` es usado por `Compressor_Decompressor` para luego aplicar las funciones de compresión y descompresión de dicho modelo en las imágenes.

```
    def save_model(self,PATH):
        torch.save(self.state_dict(),PATH)

    def load_model(PATH, compression_out):
        model = AutoEncoder(hidden_sizes=[32*3,compression_out,32*3])
        model.load_state_dict(torch.load(PATH))
        model.eval()
        return model
```

**Listing 4.15:** Métodos *save_model* y *load_model*

# CAPÍTULO 5
# Experimentación y resultados

Para encontrar cuales son las características que hacen que un *autoencoder* comprima y reconstruya una imagen con el menor error posible, se han hecho una serie de experimentos. En esta sección se detallan todos los experimentos realizados y cuál es su implementación.

Cada experimento consiste en cambiar un parámetro o característica de la red, y se quiere medir cuánto afecta este cambio a la calidad de reconstrucción. Para mantener una rigurosidad lo mayor posible, en todos los experimentos se usan algunas configuraciones que no se cambiaron en ningún experimento, salvo cuando se especifica. Las configuraciones que se mantienen constantes son las siguientes:

1. El optimizador usado durante el entrenamiento de las redes siempre es el algoritmo *ADAM*. La razón por la que se ha usado este optimizador es porque consigue una convergencia mucho más rápida que otros optimizadores. De esta forma, el tiempo de entrenamiento es menor, consiguiendo resultados similares. Se utiliza una tasa de aprendizaje de $10^{-3}$ y un momento de 0,9, como se propone en [6].

2. La función de pérdida usada es el error medio cuadrático. El problema de optimización de un *autoencoder* es un problema de regresión, ya que se quiere, a partir de unos atributos extraídos de la imagen original, inferir unas características de mayor dimensionalidad.

3. Se utilizan *batches* de 2000 bloques de imagen en cada iteración.

4. En total se entrenan los *autoencoders* en 200.000 iteraciones, divididas en 20.000 *epochs* en las que se registra el error calculado, aunque estas converjan antes.

5. La imagen se utiliza en todos casos con el modelo de color RGB y los canales de color no son independientes, estando conectados entre sí.

6. Cada uno de los modelos se prueba finalmente con las mismas imágenes, una colección de 41 imágenes de todo tipo extraídas de la base de datos del SIPI [24], incluyendo retratos, imágenes satélite, paisajes, escenas de películas en blanco y negro y más. Para cada modelo se calcula el *MSE*, *PSNR* y *SSIM* con cada imagen, con lo que luego se calcula una media.

7. Los resultados a veces se comparan con los resultados de JPEG obtenidos calculando las mismas métricas que las utilizadas sobre imágenes comprimidas con el compresor JPEG de la aplicación web https://squoosh.app. Los resultados obtenidos de dichas pruebas pueden verse en la tabla 5.1.

| bpp | PNSR (dB) | MSE | SSIM |
|---|---|---|---|
| 0,078 | 33,26 | 33,26 | 0,844 |
| 0,125 | 34,48 | 34,48 | 0,884 |
| 0,156 | 35,13 | 35,13 | 0,900 |
| 0,187 | 35,80 | 35,80 | 0,912 |
| 0,218 | 36,35 | 36,35 | 0,921 |

**Tabla 5.1:** Resultados de la compresión de JPEG

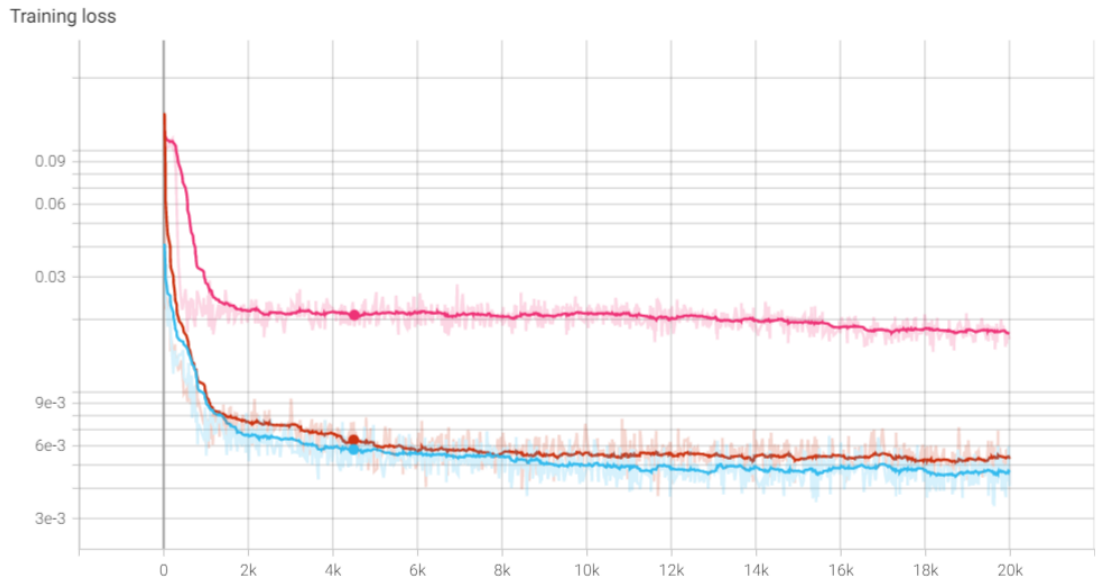



**Figura 5.1:** Error de entrenamiento de los modelos usando *Leaky ReLU*, *ReLU* y *Sigmoid*

## 5.1 Funciones de activación

En esta sección se describe el procedimiento de experimentación en el que se elige el mejor tipo de función de activación para el problema. Para ejecutar este experimento se configura el *autoencoder*, al iniciarlo, con la función de activación que se pretende usar. Esto se consigue simplemente dándole la función deseada al parámetro `activation` del modelo de los *autoencoders*. Esto se prueba únicamente en las arquitecturas de ocho capas ocultas `AutoEncoder_8H`, dejando el resto de parámetros como son por defecto.

En total se comparan tres funciones de activación diferentes: *ReLU*, *Sigmoid*, *Leaky ReLU*. La función de activación *Tanh* no se utiliza por su similitud con la función *Sigmoid*, aunque a diferencia de *Sigmoid*, que se sature en el lado negativo a cero, esta se satura a -1.

Tras entrenar estos modelos se prueban con imágenes no vistas previamente y se calculan los valores de MSE y PSNR de las reconstrucciones. Los resultados se pueden ver en la figura 5.2. En estos resultados se puede ver cómo el mejor resultado se obtiene con *Leaky ReLU*, seguido muy de cerca por *ReLU*.

Utilizando esta información como guía para elegir la función de activación más apropiada, se decide utilizar en el resto de experimentos la función *ReLU*. Aunque *Leaky ReLU* da mejores resultados, estos son muy similares y el tiempo de entrenamiento es algo menor, ya que cálculo de los pases es más rápido al haber menos cálculos que hacer.

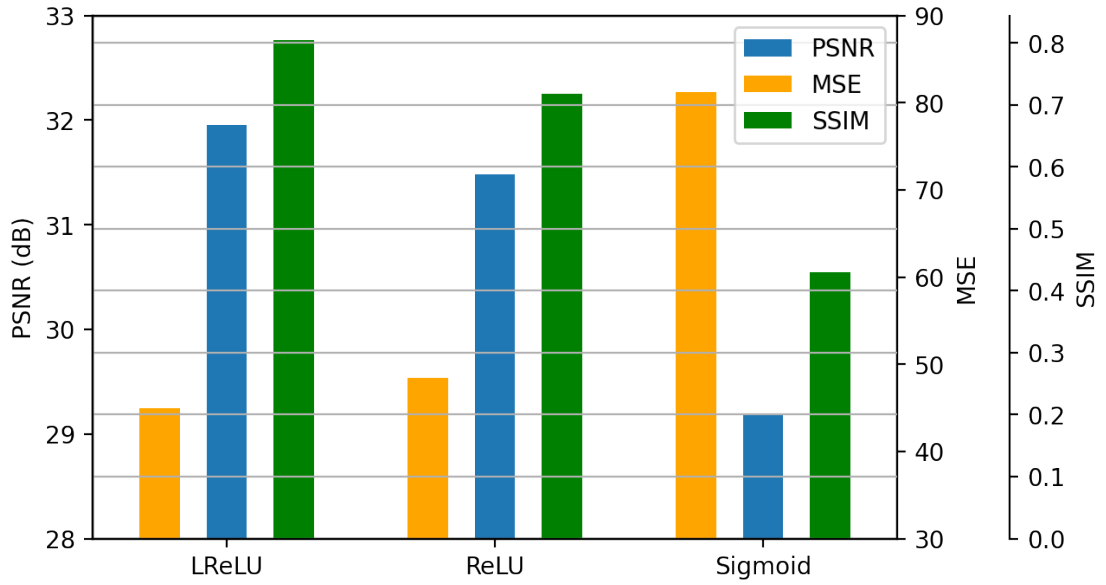



**Figura 5.2:** Valores de MSE, PSNR y SSIM por cada función de activación

## 5.2 Variación de la profundidad y del cuello de botella en el *autoencoder*

Como se explica en la sección 3.1, una profundidad mayor en las redes neuronales del tipo *FNN* otorga una mayor expresividad y una mayor capacidad de aprendizaje de características. En este experimento, se pretende estudiar cual es el impacto de este cambio de profundidad en la calidad de compresión y reconstrucción.

Para ello, se diseñaron cuatro *autoencoders* con diferentes profundidades, de entre 4 y 8 capas ocultas. La profundidad de dichos *autoencoders* se mide en el número de capas ocultas que este tiene, siendo estas las capas que tienen parámetros ajustables. En este experimento se utilizan únicamente capas totalmente conectadas, intercaladas con capas de activación *ReLU*.

La implementación de los modelos se hace mediante un modelo base, con el nombre de clase `AutoEncoder`, preparado para alojar cuatro capas lineales, inicializadas estas en la función `init_layers` que, como se explica en la sección 4.3.5, es llamada en la inicialización de una instancia nueva del modelo. La inicialización de las capas se hace de la siguiente forma:

```
1     self.activation = activation
2     #Encode
3     self.fc1 = nn.Linear(self.input_size, hidden_sizes[0])
4     self.fc2 = nn.Linear(hidden_sizes[0], hidden_sizes[1])
5     #Decode
6     self.fc3 = nn.Linear(hidden_sizes[1], hidden_sizes[2])
7     self.fc4 = nn.Linear(hidden_sizes[2], self.input_size)
```

**Listing 5.1:** Inicializar capas lineales y no lineales de *AutoEncoder*

Luego, durante la evaluación del modelo, la función `forward` llama a las funciones `encode` y `decode`, que utilizan las capas lineales inicializadas bajo el comentario de *Encode* y *Decode* respectivamente. También se intercalan capas de activación del tipo *ReLU*.

Este mismo proceso de inicialización se usa en el resto de modelos `AutoEncoder_6H`, `AutoEncoder_8H`, `AutoEncoder_12H`. Sin embargo, en el resto se inicializan, en vez de cuatro

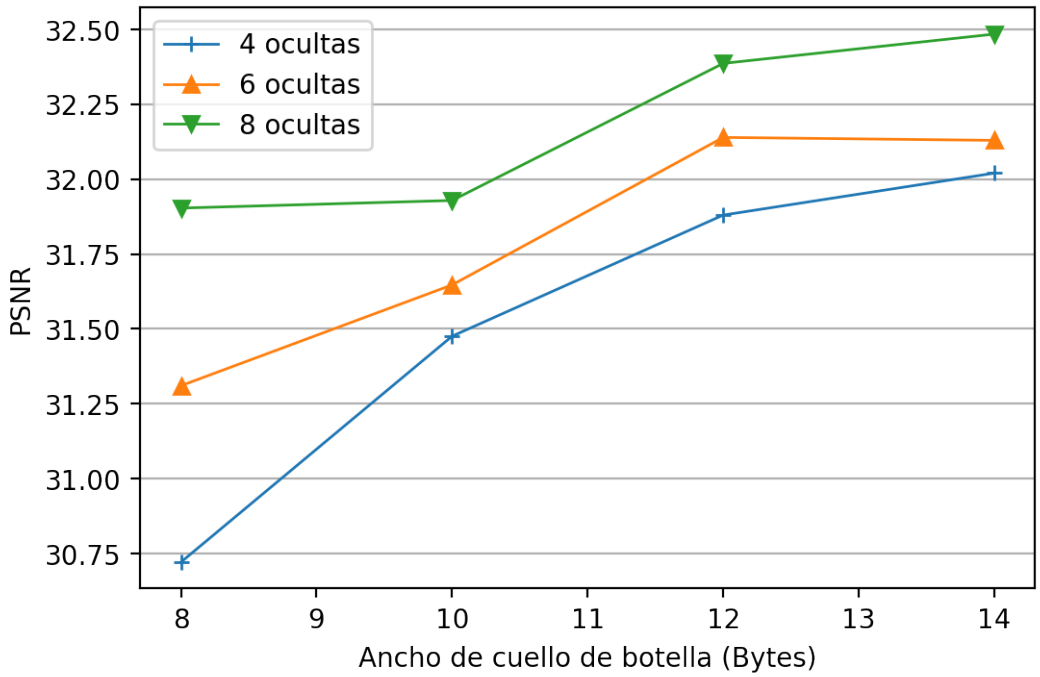



**Figura 5.3:** Comparación de cómo diferentes profundidades de modelo y anchos de representaciones afectan al PNSR de la reconstrucción

**Tabla 5.2:** Tabla de los anchos de las capas utilizadas y el número de capas ocultas en cada una de las arquitecturas de *autoencoder* con las que se experimentó.

| Nombre | Nº capas ocultas | Anchos por defecto |
|---|---|---|
| AutoEncoder | 4 | $[96, 8, 96, 192]$ |
| AutoEncoder_6H | 6 | $[96, 24, 8, 24, 96, 192]$ |
| AutoEncoder_8H | 8 | $[96, 48, 24, 8, 24, 48, 96, 192]$ |
| AutoEncoder_12H | 12 | $[160, 128, 96, 48, 24, 8, 24, 48, 96, 128, 160, 192]$ |

capas, como en la clase `AutoEncoder`; 6, 8 y 12 respectivamente, como se indica en sus nombres de clase. Cada una de sus estructuras pueden ser vistas en forma de grafo en la figura 5.4.

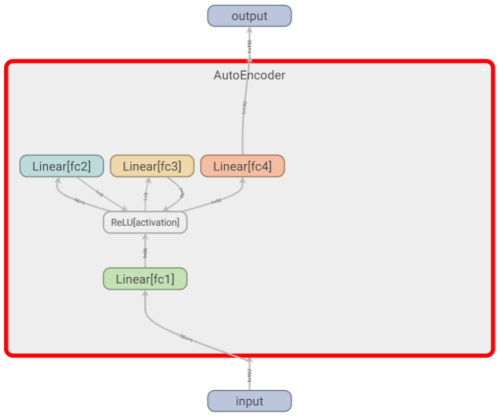
Cabe mencionar que los parámetros de inicialización para los diferentes modelos son exactamente los mismos: se usa *ReLU* como activación y entrada de $8 \times 8 \times 3 = 192$. Sin embargo, el parámetro `hidden_sizes`, que es una lista de enteros, tiene una longitud diferente. Las capas ocultas de los diferentes modelos y sus anchos pueden verse en la tabla 5.2.

Además, en este experimento se estudia cómo afecta la proporción de compresión, calculada en bits por píxel o *bpp*, a la calidad de recuperación, medida con las métricas de *MSE*, *PSNR* y *SSIM*. La teoría dice que aumentar la dimensionalidad de la salida del compresor aumenta también la capacidad de este de expresar diferentes características de la imagen. Durante este experimento también, se mide en qué medida se capta más información de la imagen al incrementar el tamaño del espacio latente.

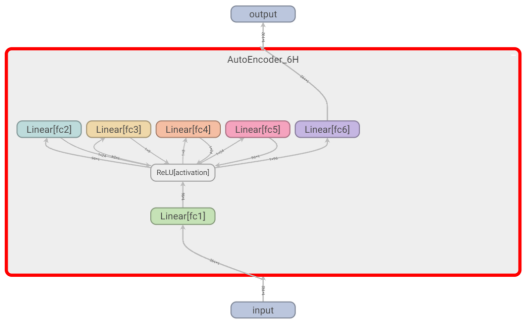
Esto se repite con todos los modelos mencionados, captando así cual es la progresión de mejora de calidad en las diferentes profundidades. Para ello, se entrenan 4 diferentes *autoencoders* para cada una de las diferentes arquitecturas.

Cambiando el parámetro `compressed_size` al iniciar cada uno de los modelos, se crean *autoencoders* con anchos de cuello de botella de 8, 10, 12 y 14. El resto de parámetros del *autoencoder*, como el ancho del resto de las capas, la función de activación y el tamaño de entrada, se mantienen con los valores por defecto.

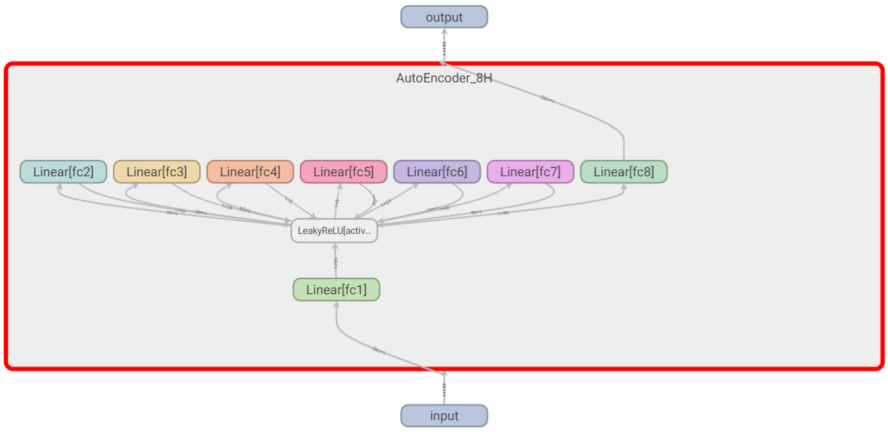
Tras entrenar todos los modelos, estos son puestos en práctica para comprimir las imágenes de prueba. Tras su compresión y consecuente descompresión las puntuaciones de PSNR obtenidas son las observadas en la figura 5.3. En esta gráfica se puede ver cómo las diferentes arquitecturas puntúan en la métrica de PNSR y cómo esta cambia para diferentes anchos de cuello de botella.



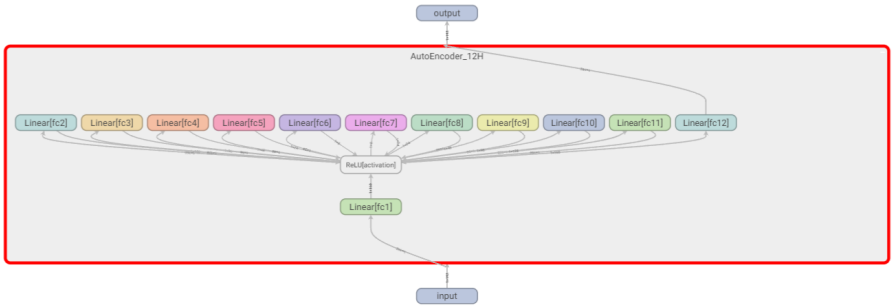

**(a)** Autoencoder




**(b)** Autoencoder_6H

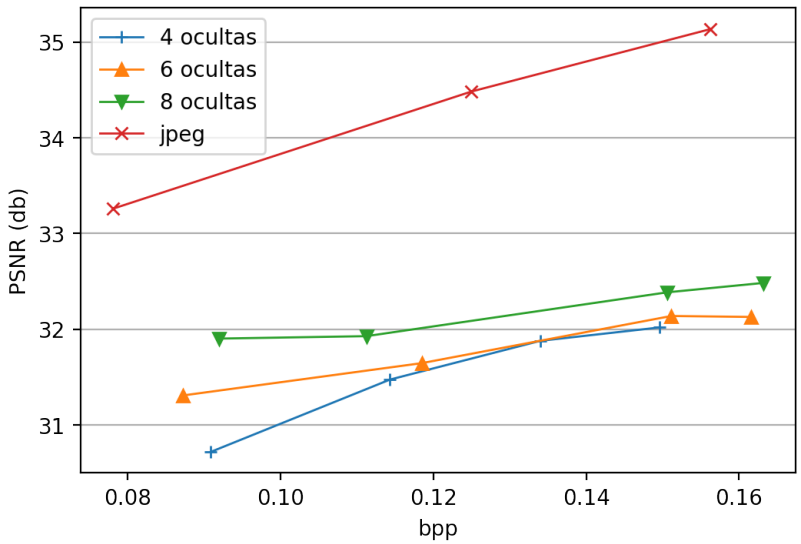



**(c)** Autoencoder_8H

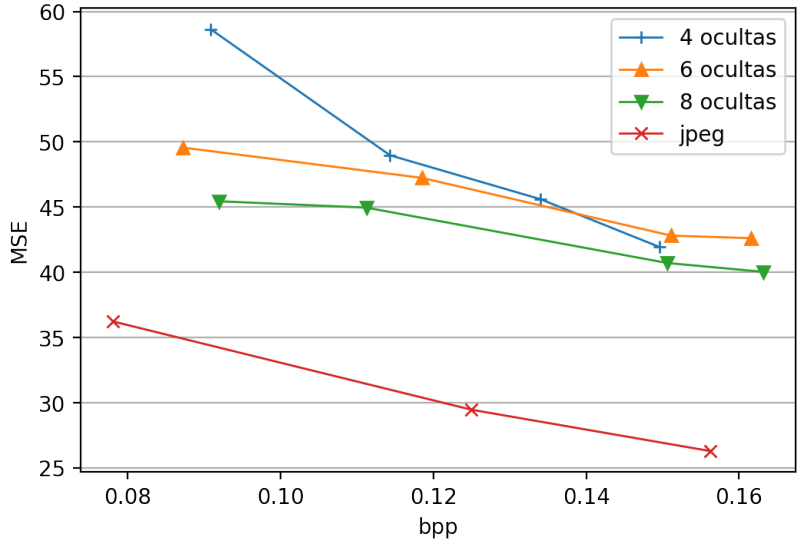



**(d)** Autoencoder_12H

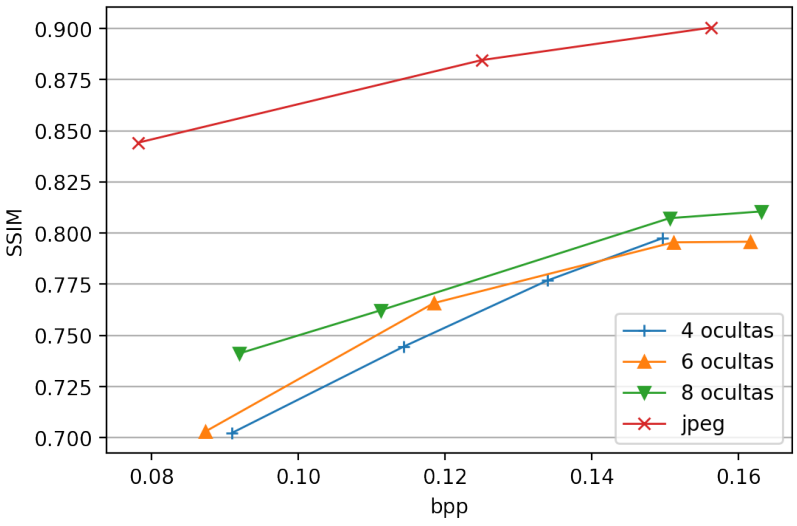
**Figura 5.4:** Modelos representados como grafos




**(a)** PSNR




**(b)** MSE




**(c)** SSIM

**Figura 5.5:** MSE, PSNR y SSIM de los modelos entrenados comparados con JPEG

| D | H | Tamaño (MB) | DEFLATE (MB) | bpp | PNSR (dB) | MSE | SSIM |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 2,42 | 1,76 | 0,091 | 30,72 | 58.62 | 0.702 |
| | 10 | 3,02 | 2,21 | 0,114 | 31,47 | 48.97 | 0.744 |
| | 12 | 3,61 | 2,58 | 0,134 | 31,88 | 45.61 | 0.776 |
| | 14 | 4,21 | 2,88 | 0,149 | 32,02 | 41.93 | 0.797 |
| 6 | 8 | 2,42 | 1,69 | 0,087 | 31,31 | 49.56 | 0.703 |
| | 10 | 3,02 | 2,29 | 0,118 | 31,64 | 47.24 | 0.765 |
| | 12 | 3,61 | 2,91 | 0,151 | 32,13 | 42.81 | 0.795 |
| | 14 | 4,21 | 3,11 | 0,161 | 32,12 | 42.62 | 0.795 |
| 8 | 8 | 2,42 | 1,78 | 0,091 | 31,90 | 45.44 | 0.741 |
| | 10 | 3,02 | 2,15 | 0,111 | 31,92 | 44.96 | 0.762 |
| | 12 | 3,61 | 2,90 | 0,150 | 32,38 | 40.71 | 0.807 |
| | 14 | 4,21 | 3,14 | 0,163 | 32,48 | 40.03 | 0.810 |

**Tabla 5.3:** D: profundidad en número de capas ocultas, H: ancho en bytes, Tamaño: tamaño total tras compresión en *autoencoder*, DEFLATE: tamaño tras compresión con DEFLATE

Para completar el experimento se comprimen con DEFLATE los vectores generados por los *autoencoders* para ver cuál es la proporción de compresión final. Los resultados de este experimento se pueden ver en la figura 5.5, donde también se comparan con el PSNR de JPEG para representaciones de tamaño parecido.

Gracias a estos datos podemos ver que una mayor profundidad es evidentemente beneficiosa para la capacidad de reconstrucción de los modelos. El modelo con más capas, `AutoEncoder_8H`, obtiene métricas mejores que el resto de modelos de forma consistente, mientras que el modelo `AutoEncoder`, el modelo menos profundo, obtiene las peores, con el último modelo `AutoEncoder_6H` en medio de forma consistente también.

Cabe mencionar que también se experimentó con modelos más profundos, como el modelo `AutoEncoder_12H`. Sin embargo, dichos modelos no convergían de manera satisfactoria incluso dejándolos entrenar con más lotes. Por ello, estos resultados no se tuvieron en cuenta y no se incluyen en el resultado final. También se propuso experimentar con anchos de cuello de botella mayores, pero los datos recabados con los modelos obtenidos parecían lo suficientemente ilustrativos.

## 5.3 *Denoising autoencoders*

Este experimento pretende entrenar modelos utilizando como entrada una versión del bloque que deben reconstruir a la que se le ha añadido ruido. La intención es entrenar modelos más robustos, que sean capaces de generalizar mejor a casos de imágenes no vistas durante el entrenamiento.

Para completar este experimento, se utiliza ruido del tipo *Salt and Pepper* ya que se ha encontrado que funciona bien en la aumento de datos para otros problemas de visión por computador [17]. De esta forma, durante el entrenamiento el modelo debe intentar

| H | Tamaño (MB) | DEFLATE (MB) | bpp | PNSR (dB) | MSE | SSIM |
|---|---|---|---|---|---|---|
| 8 | 2,42 | 1,75 | 0,086 | 30,74 | 57,57 | 0,6467 |
| 12 | 3,61 | 2,92 | 0,144 | 31,34 | 52,26 | 0,6986 |
| 14 | 4,21 | 3,26 | 0,160 | 30,94 | 55,39 | 0,6936 |

**Tabla 5.4:** H: ancho en bytes, Tamaño: tamaño total tras compresión en un *autoencoder* entrenado con ruido, DEFLATE: tamaño tras compresión con DEFLATE

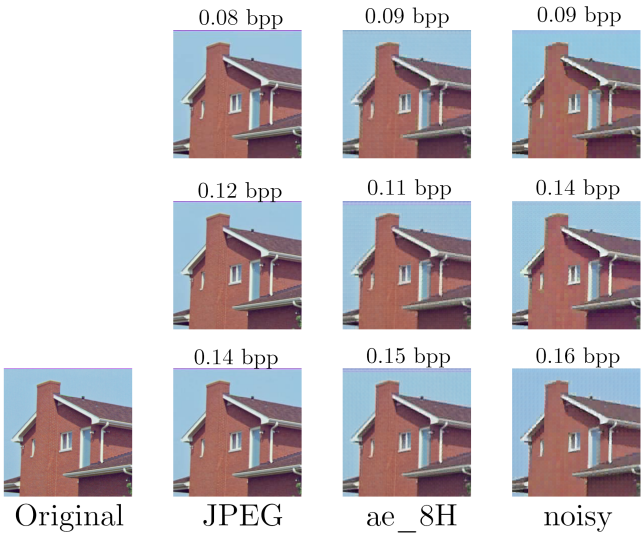



**Figura 5.6:** Resultados de aplicar las diferentes técnicas y JPEG como comparación visual

reconstruir el bloque sin el ruido que se da como entrada. Al incluir este nuevo ruido a la entrada, la función de pérdida se define como

$$L = \sum_{i=1}^{N} (g(f(x_i + SnP)) - x_i)^2$$

donde $SnP$ indica el ruido que se añade a cada uno de los bloques.

Para implementar este ruido en la entrada se hace uso de la función de *scikit-image* `random_noise`. Esta función se encuentra en el apartado de `skimage.utils` y se usa de la siguiente forma:

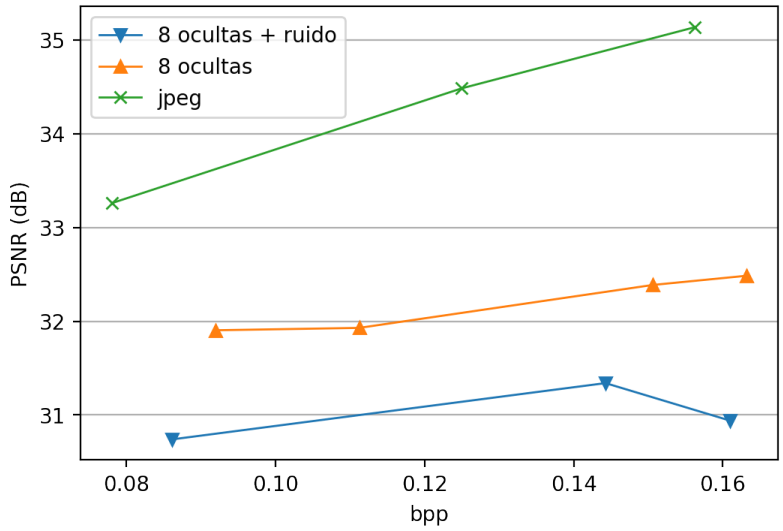
```
1    noisy_image = random_noise(image_np,mode="s&p",amount=0.3)
```

**Listing 5.2:** Cómo incluir ruido a los bloques

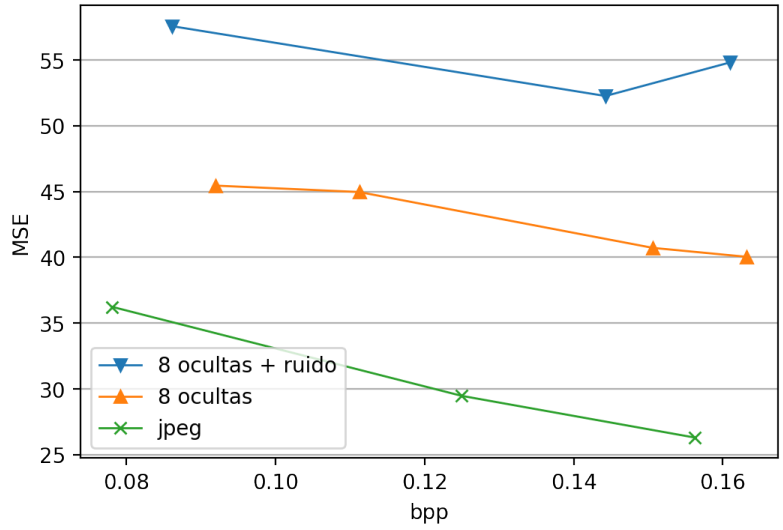
De esta forma se indica que se quiere añadir ruido aleatorio del tipo *Salt and Pepper*, `mode="s&p"`; con una probabilidad del 30 por ciento, `amount=0.3`; en la imagen `image_np`. Esta nos dará una imagen igual que la introducida, pero con un ruido añadido. Seguidamente, se extraen de las mismas coordenadas bloques aleatorios en ambas imágenes, como se explica en la sección 4.3.1.

Con esta nueva forma de construir los datos de entrenamiento se entrenan tres modelos diferentes con diferentes anchos de cuello de botella. Para los tres modelos se usa una arquitectura con ocho capas ocultas de la clase `AutoEncoder_8H` con *ReLU* como función de activación. Estos tres modelos tienen anchos de 8, 12 y 14 bytes. Una vez son entrenados se prueban en el conjunto de imágenes de prueba y se obtienen los resultados observados en la tabla 5.4 y la figura 5.7, además de un ejemplo visual en la figura 5.6.

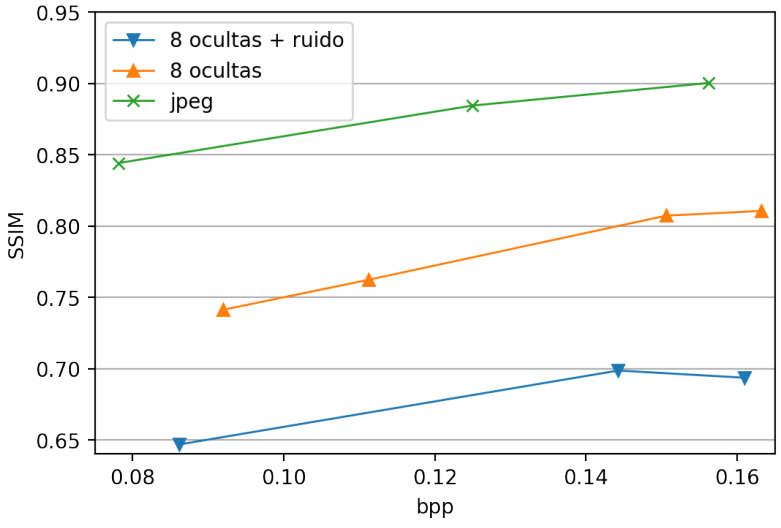
Los resultados muestran que, a pesar de que se esperaba que añadirle ruido a la entrada fuera a dar mejores resultados, la adición de ruido no es del todo efectivo. Es posible que no se haya añadido suficiente ruido o que, en cambio, se haya añadido demasiado. En cualquier caso, el modelo entrenado sin ruido tiene mejores métricas y JPEG todavía mejores.




**(a)** PSNR




**(b)** MSE




**(c)** SSIM

**Figura 5.7:** MSE, PSNR y SSIM de los modelos entrenados con ruido comparados con JPEG y los modelos entrenados sin ruido

# CAPÍTULO 6
# Conclusión

En la sección de experimentación se ha visto cómo el uso de *autoencoders* en técnicas de compresión de imágenes es viable, aunque se ha obtenido resultados peores que JPEG, el formato de compresión estandarizado en la industria. En esta sección se explican las conclusiones y lecciones que se extraen de los experimentos y sus resultados, además de mejoras que se pueden aplicar en futuros trabajos.

Primero, en el experimento 5.1 se obtuvo mejores resultados para el uso de la función de activación *Leaky ReLU*; sin embargo, en el resto de experimentos se utiliza *ReLU* por la facilidad de su uso y la mayor velocidad de entrenamiento, aunque marginal. Podría ser una buena idea probar el rendimiento de otras funciones de activación como *Parametric ReLU* y compararlo con los resultados actuales, pero en cualquier caso utilizar *Leaky ReLU* parece una mejor alternativa que la usada en los demás experimentos y debería probarse también en ellos.

Por otro lado, en el experimento 5.2 se observó que la tendencia era que los resultados de reconstrucción mejoraran al aumentar el número de capas utilizadas y también al aumentar el ancho del cuello de botella, ya que se incrementa la expresividad de la representación latente. Un problema que se observó también era que para redes demasiado profundas el entrenamiento no convergía satisfactoriamente, por lo que es posible que se necesite usar otras técnicas, como convoluciones, para mejorar estos resultados.

En el último experimento 5.3 se utilizaron *autoencoders* para la eliminación de ruido durante el entrenamiento. A pesar de que se esperaba que esta técnica de entrenamiento mejorara la generalización del modelo, se obtuvieron peores resultados durante este experimento. Esto puede ser a causa del tipo de ruido utilizado (*Salt and Pepper*), la cantidad de ruido o al conjunto de entrenamiento utilizado, aunque también puede ser causado por una falta de entrenamiento. La técnica de eliminación de ruido usada favorece la generalización de los resultados del entrenamiento, sin embargo, ya que el número de bloques utilizados durante el entrenamiento fueron los mismos para todos los experimentos, esta generalización puede causar un empeoramiento en los casos específicos, y, al usar esta técnica de entrenamiento, tal vez usar un mayor número de iteraciones puede ser necesario para sanear esos casos específicos.

En cuanto al trabajo futuro, se ha identificado tres formas en las que este trabajo podría mejorarse. Con estas técnicas se pretende mejorar la capacidad de los *autoencoders* para reconstruir las imágenes de forma más fiable.

Uno de los problemas más obvios que se ha observado en los resultados es la distorsión que se genera entre bloques. Esta sección tan evidente puede ser mejorada utilizando bloques más grandes o incluyendo un factor proporcional a la métrica de SSIM en la función de pérdida. Esto último puede mejorar los resultados de la compresión respecto a la

percepción humana de la forma de la imagen, ya que al entrenar el modelo únicamente con el MSE esto no se tiene en cuenta y SSIM está especialmente pensado para ello.

Otra forma en la que se podría mejorar este proyecto podría ser dividiendo la imagen por los canales de color. De esta forma se podría aplicar el proceso de reducción de resolución a los canales de croma como hace JPEG, aumentando incluso más la compresión pero sin afectar gravemente a la percepción de la forma, ya que el canal de luz mantendría una mayor resolución.

Finalmente, una técnica que se podría aplicar para mejorar significativamente la precisión de reconstrucción podría ser el uso de convoluciones. Usando convoluciones es posible aumentar mucho más la profundidad de los modelos, una característica que se ha visto que mejora significativamente la precisión de reconstrucción en la sección de experimentación. Además, el uso de convoluciones bidimensionales permite a la red neuronal aprender más sobre la estructura y geometría de cada bloque, que es una característica realmente importante a la hora de reconstruir la imagen.

# Bibliografía


[1] Fedy Abi-Chahla. Nvidia's cuda: The end of the cpu? https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html, 2008. [Internet; descargado 3-junio-2022].

[2] Mark Adler. Png specification. https://www.w3.org/TR/PNG/, 1996. [Internet; descargado 21-junio-2022].

[3] Murtaza Eren Akbiyik. Data augmentation in training cnns: Injecting noise to images. *ICLR*, 2019.

[4] Alfredo Sánchez Alberca. La librería numpy. https://aprendeconalf.es/docencia/python/manual/numpy/, May 2022.

[5] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders, 2020.

[6] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Deep convolutional autoencoder-based lossy image compression. *CoRR*, abs/1804.09535, 2018.

[7] Contribuidor de *Wikimedia Commons* Stevo-88. File:common chroma subsampling ratios.svg, 2020. [Internet; descargado 24-junio-2022].

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] Niklas Donges. Gradient descent: An introduction to 1 of machine learning's most popular algorithms. https://builtin.com/data-science/gradient-descent, Aug 2021. [Internet; descargado 9-junio-2022].

[10] Luke Dormehl. What is an artificial neural network? https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/, Jan 2019. [Internet; descargado 7-junio-2022].

[11] Farzad Ebrahimi, Matthieu Chamik, and Stefan Winkler. Jpeg vs. jpeg 2000: an objective comparison of image encoding quality. In *Applications of Digital Image Processing XXVII*, volume 5558, pages 300–308. SPIE, 2004.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Google. Conoce las ubicaciones de nuestros centros de datos. https://www.google.com/about/datacenters/locations. [Internet; descargado 3-junio-2022].

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew

Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[15] Dan Hendrycks, Steven Basart, Norman Mu, Saurav Kadavath, Frank Wang, Evan Dorundo, Rahul Desai, Tyler Zhu, Samyak Parajuli, Mike Guo, Dawn Song, Jacob Steinhardt, and Justin Gilmer. The many faces of robustness: A critical analysis of out-of-distribution generalization. *ICCV*, 2021.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[17] Xiaogeng Liu, Haoyu Wang, Yechao Zhang, Fangzhou Wu, and Shengshan Hu. Towards efficient data-centric robust machine learning with noise-based augmentation, 2022.

[18] Lin-Manuel Miranda. My shot, 2015.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[20] Patrick Rothfuss. El nombre del viento, 2015.

[21] C Saravanan and M Surender. Enhancing efficiency of huffman coding using lempel ziv coding for image compression. *International Journal of Soft Computing and Engineering*, 2013.

[22] Jia Shijie, Wang Ping, Jia Peiyi, and Hu Siping. Research on data augmentation for image classification based on convolution neural networks. In *2017 Chinese Automation Congress (CAC)*, pages 4165–4170, 2017.

[23] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1), 2019.

[24] *USC-SIPI* image database. Disponible en https://sipi.usc.edu/database/.

[25] Samajdar Tina and Md. Iqbal Quraishi. Analysis and evaluation of image quality metrics. *Springer*, 340:369–378, Jan 2015. [Internet; descargado 11 Junio 2022].

[26] P Umesh. Image processing in python. *CSI Communications*, 23, 2012.

[27] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[28] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, 2020.

[29] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[30] Yong Zhang and Donald A. Adjeroh. Prediction by partial approximate matching for lossless image compression. *IEEE Transactions on Image Processing*, 17(6):924–935, 2008.

[31] Emir Öztürk and Altan Mesut. Performance evaluation of jpeg standards, webp and png in terms of compression ratio and time for lossless encoding. In *2021 6th International Conference on Computer Science and Engineering (UBMK)*, pages 15–20, 2021.

# ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

| **Objetivos de Desarrollo Sostenible** | **Alto** | **Medio** | **Bajo** | **No procede** |
|---|---|---|---|---|
| ODS 1. **Fin de la pobreza.** | | | | X |
| ODS 2. **Hambre cero.** | | | | X |
| ODS 3. **Salud y bienestar.** | | | X | |
| ODS 4. **Educación de calidad.** | | | X | |
| ODS 5. **Igualdad de género.** | | | | X |
| ODS 6. **Agua limpia y saneamiento.** | | | | X |
| ODS 7. **Energía asequible y no contaminante.** | | X | | |
| ODS 8. **Trabajo decente y crecimiento económico.** | | | | X |
| ODS 9. **Industria, innovación e infraestructuras.** | X | | | |
| ODS 10. **Reducción de las desigualdades.** | | | | X |
| ODS 11. **Ciudades y comunidades sostenibles.** | | X | | |
| ODS 12. **Producción y consumo responsables.** | X | | | |
| ODS 13. **Acción por el clima.** | | | X | |
| ODS 14. **Vida submarina.** | | | | X |
| ODS 15. **Vida de ecosistemas terrestres.** | | | | X |
| ODS 16. **Paz, justicia e instituciones sólidas.** | | | | X |
| ODS 17. **Alianzas para lograr objetivos.** | | | X | |

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Comprimir imágenes es una técnica que hace que toda tarea que tenga que ver con ellas sea más eficiente. Tomar un menor espacio en disco y reducir los tiempos de transmisión por internet tiene efectos positivos relacionados con varios ODS. Estos son los ODS relacionados más relevantes:

- **Industria, innovación e infraestructuras.** Este trabajo promueve el desarrollo de nuevas técnicas de compresión para imágenes que pueden ser utilizadas en procesos industriales. A su vez esto mejora la infraestructura y la hace más sostenible, aumenta la investigación científica en este tema y facilita el acceso a la información mediante internet.

  Proporcionar nuevas técnicas de compresión de imágenes ayuda a modernizar la infraestructura del software de las industrias para que sean más sostenibles ya que utilizan menos energía para mantener en funcionamiento los servicios de almacenaje. Además, utilizan los recursos con mayor eficacia, ya que utilizar menos servidores implica también que se tienen que fabricar menos componentes electrónicos para estos, reduciendo así los recursos utilizados. Como último apunte respecto a la modernización de la infraestructura, con este trabajo se promueve la adopción de tecnologías limpias y ambientalmente razonables ya que al reducir las facturas de energía la industria puede optar por utilizar proveedores que generen su energía mediante procesos renovables en vez de elegir los más baratos siempre.

  Por otro lado, este trabajo intenta aumentar la investigación científica en nuevas técnicas de compresión, mejorando la capacidad tecnológica de los sectores industriales y fomentando la innovación en estos. Proponer nuevas técnicas de compresión para imágenes es muy importante para encontrar el siguiente formato que tome el mundo como estándar en compresión de imágenes, siendo estos formatos hoy en día PNG y JPEG. La imagen y el vídeo son dos de los tipos de datos que más espacio ocupan y cada día son más utilizados, por lo que la industria, además de las instituciones académicas, deben invertir recursos en reducir el espacio que toman en los dispositivos, tanto móviles como bases de datos centralizadas.

  Cuando se trata de ayudar a las industrias emergentes, especialmente en países en desarrollo, dar nuevas y más eficientes opciones de compresión de imágenes es muy importante para proporcionar el apoyo tecnológico que necesitan. Estos desarrollos ayudarán a desarrollar infraestructuras más sostenibles y resilientes, que a su vez ayuden en el desarrollo de estas industrias, proporcionando procesos más asequibles y sostenibles.

  Finalmente, gracias a tiempos de transmisión más cortos usando nuevos formatos de imagen se espera que se incremente el acceso a la tecnología de la información y las comunicaciones, otorgando a los usuarios un acceso a internet más universal y asequible. Esto se puede conseguir gracias a que reducir la cantidad de datos que se tienen que transmitir implica que se necesita una ancho de banda inferior para recibir una imagen más comprimida con el mismo tiempo de espera.

- **Consumo y producción sostenibles.** Conseguir nuevas técnicas de compresión de imágenes puede mejorar la forma en la consumimos y producimos reduciendo tiempos de espera y espacios de almacenaje en bases de datos. De esta forma se hace un uso más

eficiente de los recursos naturales, se reduce la generación de desechos y se alienta a las empresas a que tomen prácticas sostenibles.

Se espera que mejorar los procesos de compresión de imágenes permita a la industria lograr una gestión sostenible de los recursos naturales, haciendo un uso más eficiente de ellos, especialmente de la energía. Reducir el consumo de energía por la industrias implicaría que esta se volvería más accesible y que sería más fácil de utilizar por los ciudadanos y que estos serían capaces de aplicar más prácticas de consumo sostenible.

Por otro lado, se espera que al reducir la cantidad de servidores dedicados a bases de datos que almacenan imágenes se pueda reducir el número de componentes electrónicos que se fabrican, o al menos reducir la velocidad en la que este número aumenta. De esta forma se podría reducir el número de desechos que se generan, haciendo la forma en la que se consume más sostenible y concienciada con el medio ambiente.

Finalmente, se espera que una vez las empresas adopten nuevas técnicas de compresión de imágenes sean capaces de ver otras prácticas de consumo más sostenibles. Si ven los resultados positivos de adoptar técnicas de compresión de imágenes pueden ser más susceptibles de adoptar otras prácticas que también tengan efectos similares.

- Otros objetivos de desarrollo sostenible que están relacionados con este trabajo, aunque en menor medida, serían:
  - **Salud y bienestar.** Al proporcionar nuevas técnicas de compresión de imágenes se espera que se puedan aplicar estos procesos a tratamientos médicos que hacen uso de imágenes, como la radiografía.
  - **Educación de calidad.** ¡¡Una imagen cuenta más que mil palabras¿¿ y es por eso que es importante utilizar las imágenes como apoyo en la educación. Por lo tanto es importante garantizar el acceso a ellas de la forma más sencilla posible, reduciendo tiempos de espera en las transmisiones por internet y el espacio que ocupan en los dispositivos de los estudiantes.
  - **Energía asequible y no contaminante.** Reducir la energía que se consume para el mantenimiento de bases de datos y el procesamiento de imágenes es una forma de garantizar que esta es más asequible y que se puede optar por por el uso de energías renovables.
  - **Ciudades y comunidades sostenibles.** Proporcionar formas más eficientes de comunicación y almacenamiento de datos es un paso muy importante en el desarrollo de las comunidades sostenibles, ya que reduce el impacto que estas tienen sobre el entorno.
  - **Acción por el clima.** Reducir el impacto que tienen las industrias en el clima dándoles procesos que consumen menos energía y también dándoles la oportunidad de elegir proveedores de energía limpia es la primera parte para hacer las paces con el clima como sociedad.
  - **Alianzas para lograr objetivos.** Facilitando la comunicación se espera que los grupos de trabajo que se generen sean eficientes y puedan alcanzar sus objetivos de forma satisfactoria.