Learning for Compression

Xavier Fernàndez Mellado

Resumen– En este documento se analizan algunos de los modelos basados en *deep learning* para la compresión de imagen sin pérdidas. Dentro de estos modelos existen varias taxonomías que se diferencian entre sí por la arquitectura de los modelos. En este TFG se estudian los modelos basados en auto-regresión llamados L3C, RC, SReC y ICEC; esta arquitectura se basa en la predicción del píxel actual a partir de los anteriores. Finalmente se realiza un *benchmark* que compara L3C y RC con los modelos convencionales y se extraen las conclusiones que los modelos convencionales son más veloces pero consiguen un ratio de compresión menor mientras que L3C y RC tienen una velocidad de dos órdenes de magnitud menor, pero su ratio de compresión es mayor. También se analiza la dificultad de compatibilidad de estos modelos con ciertas librerías necesarias.

Palabras clave– Compresión de imagen, sin pérdidas, machine learning, CNN, modelos auto-regresivos, L3C, RC, SReC, ICEC.

Abstract– In this document, some of the deep learning-based models for lossless image compression are analyzed. Within these models, there are several taxonomies that differ from each other by the architecture of the models. In this TFG, the self-regression-based models called L3C, RC, SReC and ICEC are studied; this architecture is based on predicting the current pixel from the previous ones. Finally, a benchmark is performed comparing L3C and RC with conventional models and the conclusions are that conventional models are faster but achieve a lower compression ratio while L3C and RC have a speed two orders of magnitude lower, but their compression ratio is higher. The compatibility difficulty of these models with certain necessary libraries is also analyzed.

Keywords– Lossless image compression, machine learning, CNN, auto-regressive models, L3C, RC, SReC, ICEC

1 Introducción

Desde que los modelos generativos discretos basados en predicciones auto-regresivas se han empezado a usar en el campo de compresión sin pérdidas en las imágenes, se ha mejorado la capacidad de compresión [5].
Es por este motivo que, en este TFG se indaga dentro de los mecanismos actualmente usados en la compresión sin pérdidas basada en auto-regresión. Para ello, y debido a la necesidad de utilizar mecanismos de *deep learning* para la predicción, se estudiarán también las distintas redes usadas para ese fin.
Dentro del estudio, se incorporará un *benchmark* de los modelos estudiados así como la comparativa con los modelos clásicos. Aprovechando esta comparativa, se implementará una librería que podrá ser descargada desde **Github**.

Los modelos auto-regresivos son modelos que realizan una predicción del píxel actual a partir de información adquirida; dicha información puede ser: los píxeles adyacentes o todos los píxeles previos de la imagen. Los modelos que se estudiarán se basan en realizar una predicción de una lista de probabilidades con todos los valores posibles del píxel actual. Las probabilidades extraídas se utilizarán como valores de entrada en una codificación aritmética condicionada; esta codificación asegurará que la compresión poseerá las propiedades de límites entrópicos de los códigos de Shannon.

Durante el *benchmark* se analiza en qué situaciones se deben utilizar ciertos modelos, basándose en el contexto y si se prefiere una alta compresión o un modelo con unas prestaciones altas a nivel de velocidad. También se discutirá si los modelos basados en *machine learning* ofrecen una solución real y aplicable o, debido a su baja velocidad y escasas

• E-mail de contacto: xavifernandezmellado@gmail.com
• Mención realizada: Tecnologías de la Información
• Trabajo tutorizado por: Joan Serra Sagristà (Ingeniería de la Información y Comunicación)
• Curso 2022/23

compatibilidades, no son una alternativa a los modelos convencionales.

2 PRELIMINAR

La cuestión a tratar se centra en la búsqueda de la optimización sobre la compresión de imágenes.
Dentro del campo de la compresión de imágenes podemos detectar dos campos: la compresión con pérdidas y la compresión sin pérdidas.
La compresión sin pérdidas es aquella que, a partir del estado comprimido, se puede volver al estado inicial obteniendo exactamente los mismos datos.
Por otra parte, la compresión con pérdidas es aquella que el estado inicial y final, después de la compresión y descompresión, son distintos. Aunque los estados difieran, para que la compresión con pérdidas sea considerada como tal, el estado inicial y el final deberán mantener una correlación muy elevada.

La compresión de datos es importante en una gran variedad de campos: desde el almacenamiento de grandes datos hasta la transmisión satelital. Veamos dos casos, siguiendo el ejemplo de almacenamiento y transmisión, donde la compresión de las imágenes nos aporta una ventaja sustancial.

Ejemplo 2.0.1. Imaginemos que debemos gestionar los recursos de una empresa que dispone de una red social con una actividad de unos 20 millones de usuarios diarios. De estos usuarios, supongamos que la mitad suben una imagen cada día. Supongamos también que, de media, las imágenes cargadas pesan 1 MB. Esto significaría que cada día se deben almacenar 10 millones de MB de datos, es decir, 10 TB de información. Se puede ver que mantener una infraestructura que pueda soportar este flujo de almacenaje es muy costoso. Pero, si conseguimos reducir el peso de las imágenes sin perder fidelidad a, por ejemplo, 10 KB, pasaríamos de necesitar almacenar 10 TB de datos diarios a 100 GB; es decir, necesitaríamos 100 días para ocupar el mismo espacio que se habría llenado en 1 solo día.
En este caso, podríamos plantearnos que la compresión con perdidas podría ser mejor opción que sin pérdidas, pues no importa perder la calidad de la imagen, pero sí es importante disminuir al máximo su peso.

Ejemplo 2.0.2. Otro ejemplo de uso real se centra en la transmisión de datos donde el canal tiene unas condiciones muy malas; supongamos que queremos recopilar imágenes de la corteza de Marte con el rover Rosalind Franklin y enviarlas a la Tierra. Si tratamos de modelar el canal sobre el que se deben transmitir las imágenes, nos encontramos que la capacidad del canal será muy baja y, por tanto, que la BER (bit error rate) será enorme; esto supondrá que, para disminuir la BER, debamos aumentar la redundancia del código empleado para transmitir. Todo ello nos comportará a un bit rate muy bajo. Para poder transferir la imagen de manera más rápida, será interesante comprimirla antes de enviarla.
En este caso, si las imágenes serán estudiadas, es necesario no perder nada de información contenida en ellas, pues podrían alterar los resultados del estudio. Por este motivo, es importante usar una compresión sin pérdidas.

En los recientes estudios, se ha empezado a usar modelos basados en *deep learning* que logran mejorar el ratio de compresión de las imágenes; esto lleva a la necesidad, no solo de entender las arquitecturas de *machine learning* sino también de estudiar los modelos que existen actualmente.

3 OBJETIVOS

El objetivo del TFG es el de estudiar algunos de los modelos basados en *deep learning* actuales de compresión de imagen sin pérdidas. Los que se estudiarán se basan en la idea de la compresión a partir del uso de un modelo generativo discreto basado en auto-regresión.
El segundo objetivo es la realización de un *benchmark* para poder comparar cualitativamente los modelos, tanto los clásicos como los basados en *deep learning*.

Para el primer objetivo se estudiarán los *papers* y documentaciones de los modelos implementados y de las técnicas de *deep learning*. También se mirarán los modelos clásicos, aunque no será un punto de gran interés.
Para el segundo objetivo, se realizará una librería en Python que facilitará el *benchmark* de modelos basados en compresión de imagen sin pérdidas. La librería incluirá la opción de elegir distintos *datasets* y distintos métodos de *benchmark* y comparación entre modelos.

4 CNN

La única arquitectura de aprendizaje por máquina que se estudiará en este TFG serán las redes neuronales convolucionales [9], esto se debe a que es la arquitectura implementada en todos los modelos que veremos.
Las Redes Neuronales Convolucionales son aplicadas, generalmente, para reconocimiento en imágenes. Pueden ser usadas como image2text, image2image y image2attribute entre otros. El funcionamiento de las CNN viene formado por 3 pasos que se estudiarán seguidamente.

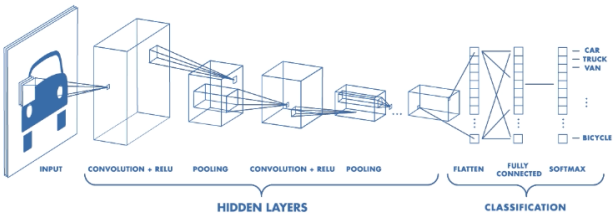

Fig. 1: Arquitectura CNN

4.1. Convolución

El primer paso es la convolución. En esta, se utiliza en la imagen una matriz llamada *kernel* que se irá desplazando sobre la imagen e irá multiplicando cada píxel por su componente respectivo. Esta matriz podrá ser de distintas dimensiones en distintos modelos.

En un amplio número de casos, la matriz imagen entrante habrá sido previamente normalizada a valores entre -1 y +1.

Así pues, las matrices de la DCT –ver apéndice A.5– usadas en los modelos convencionales pueden considerarse como matrices kernel que realizan una convolución sobre una imagen del mismo tamaño.
Es el kernel el que se verá modificado en cada iteración del

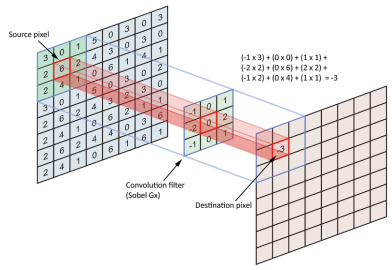

Fig. 2: Convolución en CNN

aprendizaje del modelo para llegar a la predicción deseada. El funcionamiento básico del kernel es el de identificar patrones sobre la imagen: un ejemplo está en la detección de caras. Una cara tendrá ojos, boca y nariz. El kernel terminará por poseer un patrón que se activará cuando en la imagen se detecte un ojo.

Sobre la convolución en imágenes, se utiliza en distintos campos del procesado de imagen. Un ejemplo es el de aplicación de filtros; por ejemplo, el desenfoque Gaussiano. Veamos cómo es la matriz kernel 3x3 en el desenfoque Gaussiano:

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

Y el efecto que tiene al realizar la convolución en una imagen se puede ver en la figura 3 (en la derecha está la imagen original y en la izquierda la filtrada), fíjese en la parte iluminada del sombrero para ver la diferencia.

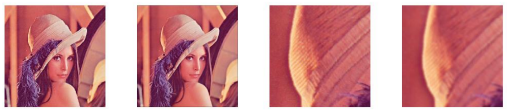

Fig. 3: Efecto desenfoque Gaussiano

4.2. Normalización

Este segundo paso (que muchas veces se fusiona con el tercero u ocurre posteriormente) se basa en aplicar a cada píxel de la imagen convolucionada la función ReLu; esta función devuelve 0 si el valor es menor a 0 y devuelve el mismo valor en los otros casos.

4.3. Pooling

Este paso se centra en disminuir la matriz resultante de la convolución. Esta disminución puede agrupar píxeles de 2x2 o incluso más grandes. La función de agrupación puede ser el promediado, escoger el valor más grande, etc. Una de las causas de la utilización del pooling, a parte de para reducir tiempo de cómputo, es la de poder aplicar imágenes con tamaño variable y finalizar con una imagen de tamaño fijo al final de la red.

Finalmente, estos 3 pasos se pueden repetir más de una vez para hacer la red más profunda y con mayor capacidad de detección.

4.4. Capa final

Como capa final entendemos el proceso final, que es posterior a la convolución, en el que la imagen resultante se transforma a un vector y pasa por una red neuronal clásica. Este paso es utilizado para aplicaciones image2text y image2attribute.

4.5. Aprendizaje

Para el aprendizaje de la red neuronal se aplica un algoritmo de *Back Propagation* que se basa en el descenso del gradiente.
Sin entrar en las matemáticas que aplican, se pretende encontrar cómo el error de predicción varía a partir de modificar los valores del *kernel* y, una vez conocida esta variación, encontrar qué valores son los que más disminuyen el error y aplicarles una corrección (para encontrar el ángulo de mayor variación se utiliza la función gradiente que no deja de ser una derivada).

4.6. resNet

En las redes neuronales convolucionales existe un problema: al aumentar el número de capas esperaríamos mejorar el porcentaje de aciertos en las predicciones, pero no es así. En la práctica, lo que se detecta es que la precisión se estanca por más capas que le añadamos. El motivo de este estancamiento está en la pérdida de la información de la imagen original o de las capas anteriores al paso de las distintas capas. Así pues, una implementación que ayuda a prevenir este problema y consigue mejorar la precisión es la de "recordar" de alguna manera al modelo la imagen anterior. Es en este punto donde entran las capas residuales (resNet [3]). Su funcionamiento es bastante básico: realizamos una suma de la imagen de unas capas anteriores con la capa actual tal y como se ve en la imagen 4.

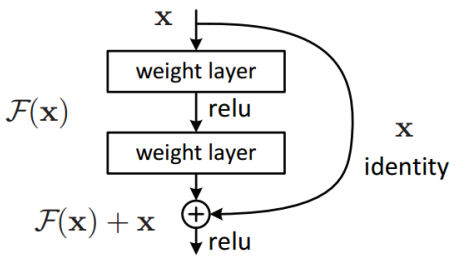

Fig. 4: Bloques Residuales

5 MODELOS

En la actualidad, hay diversos estudios realizados sobre la compresión sin pérdidas a partir del uso de *deep learning*. En este apartado, se realiza el estudio de cuatro modelos que tienen una relación entre sí: son modelos basados en redes convolucionales y con una arquitectura de auto-regresión. Sin embargo, para la selección de los modelos a estudiar, se ha realizado un estado del arte más amplio.

5.1. L3C

L3C [6] es el primer modelo basado en *machine learning* que estudiaremos (todos los estudiados se basarán el la compresión sin pérdidas). Los creadores lo definen como "el primer sistema práctico de compresión sin pérdidas a partir de aprendizaje". La practicidad de la que hablan viene dada por el incremento de la velocidad de compresión en 2 órdenes de magnitud comparado con los modelos de compresión basados en aprendizaje **PixelCNN** –ver apéndice A.8–. En cuanto a la mejoría de compresión se datan de hasta un 34 % en cuanto a bpsp sobre el modelo PNG.
Su diseño se basa en modelos generativos discretos basados en probabilidad condicional. Así pues, su forma de obtener la compresión es a partir de la predicción del píxel actual mediante los píxeles anteriores. Al igual que en PixelCNN se pretenderá calcular la probabilidad de la imagen ($p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i|x_{<i})$). Aun así, este modelo pretende mejorar una ineficiencia de PixelCNN, ya que para cada sub-píxel, el modelo debía realizar todos los cálculos con todos los sub-píxeles previos. Por lo tanto, este modelo pretende calcular la probabilidad de la imagen de manera paralelizable.

El funcionamiento de la compresión de este modelo funciona de dicha manera: primero, el modelo predice un sub-píxel (R,G o B) a partir de los anteriores; esta predicción dota de una probabilidad a cada posible valor de tal manera que la suma de todos sea 1. Una vez la predicción de probabilidad ha sido realizada, en lugar de escoger el píxel de mayor valor y restarle del valor real para quedarse con el error, se utiliza una técnica más inteligente; debido a que esta predicción dota de un conjunto de probabilidades que aportan información ordenada de la frecuencia estimada de cada sub-píxel en esas condiciones, se aplica directamente una codificación aritmética sobre el valor real a partir del conjunto de probabilidades estimado. Nótese entonces, como el código de codificación variará para cada píxel, esto es conocido como codificación aritmética adaptativa.
Esta manera de actuar contiene la ventaja de que realiza una ordenación de los valores más probables, y entonces, si ha acertado en el valor a la primera, la compresión de todo el sub-píxel será de pocos bits; por otro lado, si ha acertado en el valor con la segunda probabilidad más alta, ese sub-píxel estará codificado con los mismos bits que la primera predicción o con pocos más. En cambio, si se realiza una codificación a partir del error, cabe la posibilidad que el valor predicho se separe del valor real una cantidad muy grande, provocando mayor dificultad en la compresión.

Una vez entendida la manera de comprimir, veamos cómo es la arquitectura de manera simplificada. Como

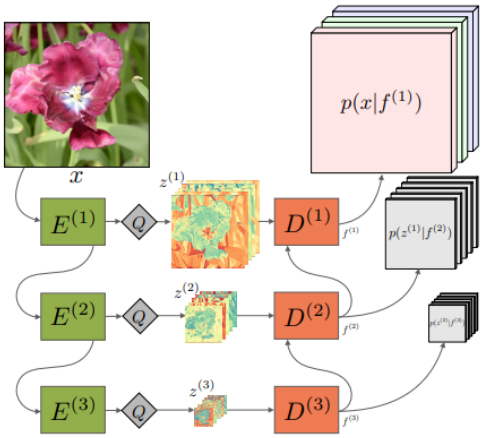

Fig. 5: Visión externa del modelo L3C

podemos ver en la figura 5, el modelo se centra en la paralelización de predicciones a partir de extraer características de la imagen que condicionarán las probabilidades finales en lugar de predecir cada píxel a partir de los anteriores.

Para la compresión, la imagen se incluye en una red CNN llamada *extractor de características* (E en la figura). Dicho extractor de características reducirá la imagen a la mitad tanto vertical como horizontalmente. El segundo paso será enviar las características actuales a un segundo extractor que realizará el mismo proceso hasta finalmente un tercer extractor. Sobre la imagen resultante de cada extractor se realiza una cuantificación (Q en la figura) basada en la asignación del valor más cercano de cada sub-píxel al de los posibles de una lista ($argmin_j||z - l_j||, \{l_1, ..., l_L\} \subset \mathbb{R}$), que en el modelo van de 1 a 25. Después de la cuantifiación, hemos obtenido las imágenes que funcionarán como extractoras de características. Estas imágenes serán enviadas a otro módulo CNN llamado *predictor* (D en la figura) que realizará la función de predecir las probabilidades de los píxeles de la imagen superior a partir de la imagen actual y de la predicción de la imagen actual (excepto en el predictor ubicado a continuación de la imagen más pequeña, que solo recibirá dicha imagen). Finalmente, se enviará en orden: la imagen más pequeña con una codificación aritmética, la imagen mediana con una codificación aritmética adaptativa a partir de la información de la imagen anterior y la imagen mayor con una codificación aritmética adaptativa a partir de la información de las dos imágenes anteriores y la imagen real con una codificación aritmética adaptativa a partir de las predicciones anteriores.

Para la descompresión, se deberá realizar el proceso inverso: primero, al recibir la imagen pequeña se descomprimirá debido al conocimiento de su compresión, después deberá pasar por el predictor menor para encontrar la distribución de probabilidad de cada píxel de la imagen intermedia. Una vez conocida la distribución se podrá descomprimir la imagen intermedia que deberá pasar por el siguiente predictor. Una vez conocida la siguiente distribución se podrá descomprimir la imagen mayor que deberá

pasar por el predictor para terminar descomprimiendo la imagen original.

Podemos ver pues que en este caso, a diferencia de PixelCNN, se deberá almacenar la imagen con las características utilizadas para predecir los sub-píxeles, y que el almacenaje de las características será comprimido recursivamente con la obtención de "características de las características" hasta 3 capas (pero podrían ser más). Esto hace que las obtenciones de las probabilidades de los sub-píxeles se consigan de manera paralelizada.

Veamos la arquitectura interna. En la figura 6 se puede

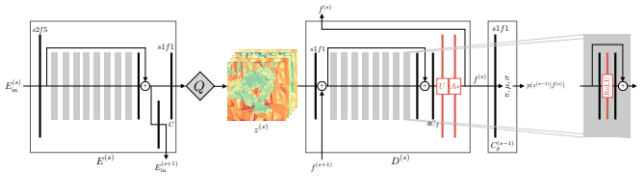

Fig. 6: Visión interna del modelo L3C

apreciar la estructura interna tanto del extractor de características como del predictor. Cada línea vertical es una convolución. Si no se especifica nada, la convolución es a partir de 64 *kernels* de 3x3 con un *stride* de 1 (se mueven un píxel en cada iteración). Por ejemplo, la primera convolución realiza un *stride* de 2 con 5 *kernels* de 3x3. Además, los bloques grises, son bloques residuales. Es importante ver que las imágenes que contienen las características tienen 5 capas. Cabe destacar que la convolución U se realiza para hacer un *upsampling* al tamaño de la imagen de características de entrada y que el bloque A* se realiza para pasar a un mapa de características de 3 capas. Finalmente los valores π, μ y σ serán usados para calcular las probabilidades a partir de conocer f que llevarán a una distribución de probabilidades parecida a sumas de Gaussianas.

5.2. SReC

Como se puede ver en la información de PixelCNN (apéndice A.8), la arquitectura presentada (modelo generativo de imagen a partir de las probabilidades condicionales) se puede usar para superresolución. Este modelo [2] se basa en dicha idea, pero para la compresión de imagen.
Vamos a suponer una imagen $x \in \{0, .., 255\}^{WxHx3}$, donde W es la longitud y H la altura.
Definiremos entonces la función $y = avgpool_2(x)$ cuyo objetivo será disminuir el tamaño de una imagen mediante reducir su altura entre dos y su longitud entre dos. La forma de realizarlo será mediante un promediado de píxeles, que puede ser interpretado como una convolución con un *kernel* de 2x2 y un *stride* de 2, donde el *kernel* tiene estos valores:

$$\frac{1}{4}\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}$$

Además, distinguiremos cuántas veces ha sido disminuida la imagen de esta manera: $x^{(0)}$ significa que no ha sido disminuida, $x^{(1)}$ significa que se ha realizado una disminución, $y^{(1)}$ significará que se ha realizado la disminución sobre la imagen $x^{(0)}$. De otra parte, el proceso de pasar de $y^{(l)}$ a $x^{(l)}$ será un redondeo de los valores resultantes, es decir $x^{(l)} = round(y^{(l)})$. Finalmente, se deberá contemplar los valores de redondeo mediante $r^{(l)} = y^{(l)} - x^{(l)} \in \{-\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}\}$. Cabe destacar que no puede existir $-\frac{1}{2}$ ya que esto significaría que se tendría que haber redondeado hacia el valor superior.

Entonces, la compresión se basará en reducir la imagen tanto como se quiera y tratar de obtener la probabilidad de superresolución. Para ello, se debe obtener la probabilidad condicionada $p(x^{(l)}|y^{(l+1)})$ y, por lo tanto, se debe predecir $p(x_{2i,2j}, x_{2i,2j+1}, x_{2i+1,2j}|y_{i,j})$ ya que $x_{2i+1,2j+1} = 4y_{i,j} - x_{2i,2j} - x_{2i,2j+1} - x_{2i+1,2j} + 4r_{i,j}$. Además, se puede predecir cada píxel de forma autoregresiva (a partir de la información de probabilidad del píxel anterior):

$$p(x_{2i,2j}, x_{2i,2j+1}, x_{2i+1,2j}|y_{i,j} =$$

$$p(x_{2i,2j}|y_{i,j})p(x_{2i,2j+1}|y_{i,j}, x_{2x,2j})$$

$$p(x_{2i+1,2j}|y_{i,j}, x_{2x,2j}, x_{2x,2j+1})$$

Además, sobre cada sub-píxel se podrá predecir a partir de conocer el sub-píxel anterior:

$$p(x_{ij}|z_{ij}) = p(x^R_{ij}|Z_{ij})$$

$$p(x^G_{ij}|Z_{ij}, x^R_{ij})p(x^B_{ij}|Z_{ij}, x^R_{ij}, x^G_{ij})$$

Con esta idea, vayamos a entender el modelo a gran escala.

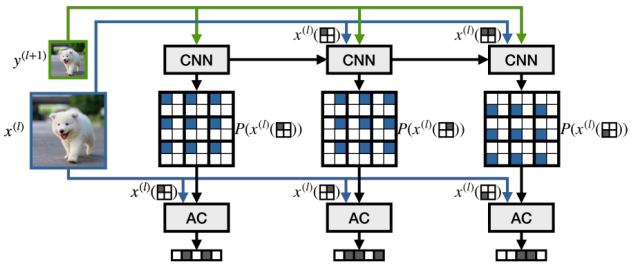

Fig. 7: Visión externa del modelo SReC en compresión

Lo que vemos en la figura 7 es un modelo simple. Para la detección del primer píxel dentro del cuadrado de 4 píxeles se realiza un proceso de CNN sobre la imagen comprimida y se obtiene una predicción sobre la que se realizará una compresión aritmética adaptativa; para el segundo píxel, se realiza un proceso CNN a partir de la imagen comprimida, la predicción anterior y los píxeles de la imagen a comprimir que ya serán conocidos por el descompresor, Posteriormente se realiza la compresión aritmética adaptativa; para el tercer píxel se realizará lo mismo que con el segundo, pero pasándole al CNN la predicción de los píxeles anteriores y el conocimiento de los 2 píxeles reales.
Por tanto, lo que se deberá tener al final comprimido es: la imagen real con la codificación aritmética adaptativa, los valores de redondeo y la imagen pequeña, ya sea comprimida con compresión aritmética o habiendo realizado el mismo proceso de compresión.

En la figura 8 observamos el método de descompresión. A partir de la imagen pequeña obtenemos la predicción de los primeros píxeles que servirán para descodificar los píxeles de la imagen real. Estos píxeles servirán para descodificar los segundos que a su vez servirán para descodificar

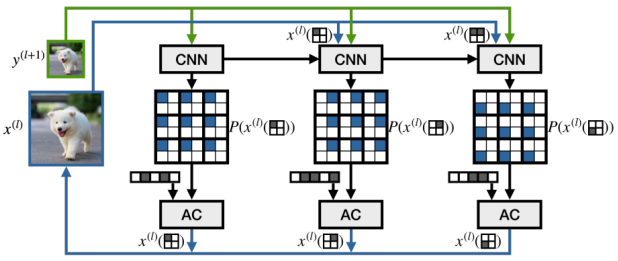

Fig. 8: Visión externa del modelo SReC en descompresión

los terceros. Para los últimos píxeles se utiliza la información de los 3 píxeles, la imagen pequeña y los valores de redondeo.

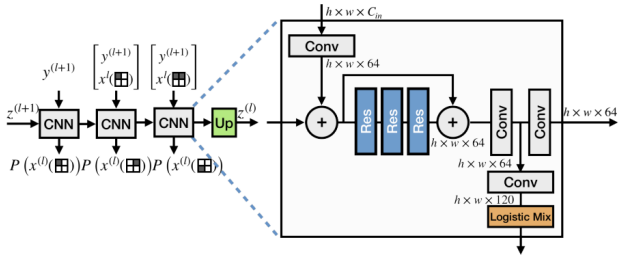

Fig. 9: Visión interna del modelo SReC

En la figura 9 podemos ver cómo es internamente el modelo CNN. Es importante destacar que la predicción no tiene una dimensión con 255 valores sino 120, que serán usados como en PixelCNN++ para calcular las probabilidades, las cuales tendrán la suma de funciones parecidas a Gaussianas.

5.3. RC

Este modelo [7] está ideado con la idea de construir un sistema de compresión sin pérdidas a partir de otro con pérdidas sin aumentar mucho los bpsp. Para ello, pretende predecir los residuos que se generarán en la compresión de BPG (un modelo de compresión con pérdidas basado en algoritmos que tiene una compresión realmente buena). Para ello, y parecido a los modelos anteriormente vistos, realiza un modelo generativo del residuo a partir de las probabilidades condicionales de la imagen con pérdidas creadas por BPG. Es decir, pretende calcular $p(l|x_l)$. Para analizar el sistema, realizaremos los mismos procesos que antes; primero analizaremos la estructura externa y luego la interna.

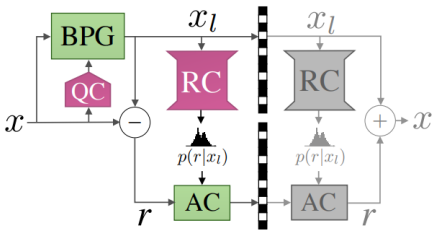

Fig. 10: Visión externa del modelo RC

Como podemos ver en la figura 10, la imagen original pasa por el compresor BPG. También pasa por un módulo llamado QC, que es un clasificador Q basado en CNN que no estudiaremos, pero que básicamente pretende predecir qué cantidad de pérdidas se le debe especificar al BPG para obtener una mayor compresión sin pérdidas. Una vez obtenida la imagen BPG se calcula el residuo y se codifica mediante la distribución de probabilidades obtenida por el predictor RC. Así pues, la imagen comprimida contendrá la imagen BPG y los residuos comprimidos mediante una compresión aritmética adaptativa obtenida por el predictor RC.

Para la descompresión, se deberá pasar la imagen BPG por el predictor RC y se obtendrán las probabilidades que servirán para descomprimir los residuos. Finalmente se sumarán los residuos a la imagen BPG. Sobre la predicción

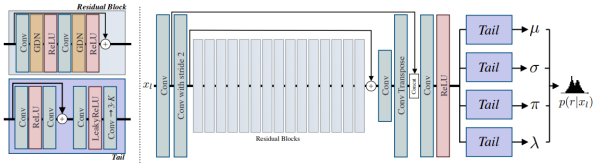

Fig. 11: Visión interna del modelo RC

de los residuos se realizará la misma parametrización que en PixelCNN++ mediante la suma de 5 funciones parecidas a las Gaussianas.
En la figura 11 se puede ver la arquitectura interna del modelo.

5.4. ICEC

Este modelo de compresión sin pérdidas [10] está pensado para ser usado en imágenes volumétricas (por ejemplo, el escaneo cerebral de un paciente). En esas imágenes hay, a demás de las dimensiones RGB, una dimensión espacial que será almacenada con distintas capas (un seguido de imágenes). Por tanto, entre capa y capa hay una gran correlación que puede ser aprovechada.
Así pues, tenemos un espacio tal que $\mathbf{x} = (x_1, x_2, .., x_T)$, donde T son el número de capas.

Para la compresión se sigue la idea que hemos ido viendo en los anteriores modelos: obtener una predicción de las probabilidades de los píxeles condicionada por conocer ciertas características. Así, se pretende minimizar la entropía cruzada entre la predicción y las probabilidades reales. Para ello, en lugar de realizar una predicción a partir de las condiciones de los píxeles anteriores como se hace en PixelCNN, se realiza una extracción de características como se realiza en L3C. Estas características las llamaremos Z_t^s, donde t definirá la capa de procedencia y s la característica dentro de la misma capa.
Aún con este modelo, seguimos teniendo el problema del modelo PixelCNN, y es que si hay una gran cantidad de capas, la cantidad de características que condicionan las probabilidades actuales irán creciendo y se hará incontrolable. Para solucionar esto, se propone utilizar una característica latente que contendrá la información acumulada de las capas anteriores H_{t-1}^s.
De la misma forma, las características de la misma capa dependerán de la cantidad que se deseen calcular, por

lo que podemos encontrarnos con el mismo problema y la solución será la misma: mantener las características anteriores acumuladas H_t^{s+2}. Nótese que s+2 connota que se utilizará Z_t^{s+1} sin acumular a las otras características. Finalmente, la predicción una característica actual Z_t^s vendrá condicionada por las anteriores: $p(Z_t^s|H_{t-1}^s, H_t^{s+2}, Z_t^{s+1})$. Cabe destacar que una capa original, al igual que hacíamos con L3C, se puede considerar como una característica Z_t^0.

Veamos el modelo entonces.

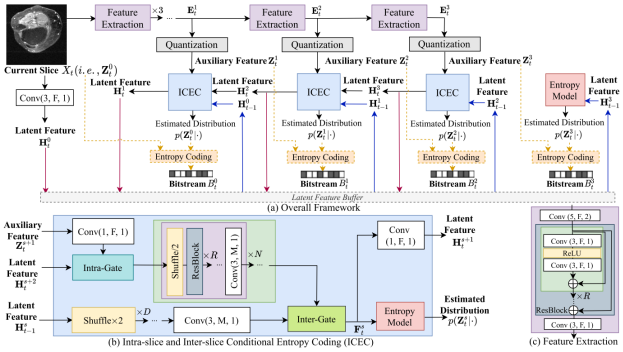

Fig. 12: Visión interna del modelo ICEC

Se puede ver en la figura 12 que el modelo ejemplo tiene 3 extractores por capa, es decir $S = 3$.
Vemos que primero se realiza una extracción de características que vendrá condicionada por la característica anterior. Finalmente podemos ver que en el modelo ICEC se le introduce, para predecir la característica Z_t^s, la característica anterior Z_t^{s+1}, H_t^{s+2} y, de un *buffer*, H_{t-1}^s. Y que, a partir de ello, calcula la distribución estimada y la característica latente que se enviará a la siguiente característica (y se almacenará en el *buffer* para la siguiente capa).

Así pues, en el ejemplo, lo que se almacenará serán las características de cada capa mediante una codificación aritmética adaptativa (esto incluye a la imagen que será vista como la primera característica). También, al igual que ocurría en L3C, la última característica será codificada con una codificación aritmética simple.

6 Benchmark

Sobre los modelos anteriormente estudiados, resulta interesante poder realizar una comparación cualitativa y, de ella, poder sacar conclusiones sobre qué modelos mejoran ciertos aspectos relacionados con la compresión. Para ello, se deben comparar los modelos dentro del mismo entorno, eso significa la utilización del mismo *hardware* en todos los modelos; sin embargo, hay una diferencia en la compresión a partir de los modelos clásicos a los modelos basados en *machine learning*: los primeros utilizan la *CPU* para realizar las operaciones de compresión mientras que los segundos acostumbrarán a realizar la predicción a partir de un modelo que se ejecuta en una *GPU* debido a la mejoría de la velocidad de las operaciones. Sin embargo, cuando se comparan los modelos clásicos con los otros, esta diferencia de *hardware*, actualmente, no supone un problema en los resultados: pues la velocidad de compresión está separada por varios órdenes de magnitud entre ambos grupos, siendo el clásico el grupo que adquiere una velocidad mayor.
Con lo anteriormente dicho, y debido a que el interés del trabajo se centra en los modelos de compresión de imagen sin pérdidas basados en *machine learning*, se realizará el *benchmark* de algunos de los modelos clásicos, pero no de todos. Para la selección de los modelos se tendrá en cuenta la librería de *Python* llamada *Pillow*; dicha librería aporta la posibilidad de almacenar de manera sencilla distintas imágenes en formato *numpy* con distintas extensiones y, por tanto, con su respectiva compresión. Los modelos que esta librería aporta son PNG, WebP, y JPEG LS. Con estos tres modelos será suficiente para poder diferenciar cómo los modelos basados en *machine learning* difieren en cuanto a rendimiento y resultados de los modelos clásicos.
En cuanto a los cuatro modelos basados en IA vistos anteriormente, se deberán realizar el *benchmark* bajo las mismas condiciones. Esto significa el mismo dataset de imágenes, la misma GPU y el mismo nicho de utilización. En cuanto a la última condición, puede ser entendida analizando el modelo ICEC, que trata de comprimir imágenes volumétricas (utilizadas normalmente en campos de la medicina); este nicho es distinto al de los otros modelos debido a su distinto campo de utilización. Finalmente, el modelo RC se basa en la transformación de un modelo clásico basado en compresión con pérdidas a uno con compresión sin pérdidas a partir de la predicción de las pérdidas añadidas; por tanto, este modelo se encuentra entre los dos grupos explicados (clásicos y basados en *machine learning*). Es por este motivo que quedará fuera del *benchmark* ya que se separa del nicho objetivo: un modelo basado en *machine learning* independiente a los modelos clásicos, es decir, modelos independientes al igual que ocurre con los modelos clásicos.

Así pues, el *benchmark* se realizará con los modelos PNG, WebP, JPEG LS, L3C y SReC.

6.1. Llimcobe

Para el *benchmark* se ha creado una librería en *Python* llamada *Llimcobe* –Lossless image compression benchmark–, esta librería calculará el ratio de compresión (en bpsp), la velocidad de compresión y la velocidad de descompresión de los distintos modelos (en MB/s) en su *deployment*. La librería también compara la imagen antes de la compresión y la imagen después de la descompresión para asegurarse que el modelo funciona correctamente y es sin pérdidas.
Sobre cómo opera la librería, se basa en calcular las características anteriores a partir de comprimir las imágenes de un *dataset* en concreto que el usuario decidirá.
El potencial de la librería está en la libertad que el usuario dispondrá. Su función es solo la que se le espera: realizar un *benchmark*; el usuario decidirá qué *dataset* se debe usar y qué modelos se analizarán. El punto negativo de esto es que el usuario deberá especificar cómo se deben realizar ciertas operaciones sobre el modelo.
La manera de programar con la librería se basa en crear una clase que heredará la clase de la librería llamada LCB

(acrónimo de Lossless Compressión Benchmark). La clase LCB, a su vez, está heredada de la clase *Abstract Base Class* y contiene un *decorator* del estilo *abstractmethod* en una función llamada *prepare_dataset*, por lo que obligará al usuario a crear una función con ese nombre en la que deberá preparar el *dataset* y devolver una lista de imágenes en formato NxMx3 *numpy.ndarray*. Por tanto, el usuario deberá realizar el pre-proceso del *dataset* para adaptarlo a un formato estándar. La función explicada será llamada en el constructor de la clase LCB (a demás de otras operaciones), por lo que se deberá llamar al constructor del LCB desde la clase herencia.
Las funciones que esta clase incorpora son varias:

1. La primera es *set_model*: esta se llamará para ir incorporando los modelos sobre los que se realizarán el *benchmark*. Sus argumentos serán el nombre del modelo y unas funciones que se pasarán por parámetro (podrán ser *lambda* –es decir, anónima– o una referencia a una función creada previamente). Estas funciones serán: la de pre-proceso de las imágenes para adaptarlas al modelo, la de guardar la imagen, la de cargar la imagen y la de comparar dos imágenes del modelo. Algunas de estas funciones no serán necesarias dependiendo del contexto.

2. A la función *get_model* se le pasará el nombre del modelo y se devolverá dicho modelo.

3. A la función *del_model* se le pasará el nombre del modelo y se eliminará si existe.

4. A la función *benchmark* se le pasará un valor que representará el número de imágenes del *dataset* sobre el que realizar el *benchmark* y creará unas gráficas estilo *Matlab* a demás de printar algunos valores importantes.

Se puede ver el código de la librería en este repositorio de **GitHub**: `https://github.com/XavierFernandez00/llimcobe` y descargarla desde **pypi** con el comando *pip install llimcobe*.

6.2. Resultados

El *benchmark* se ha realizado a partir de la compresión/descompresión de las imágenes del *dataset* **cifar-10** (`https://www.cs.toronto.edu/~kriz/cifar.html`), se trata de un *dataset* con 60000 imágenes de 32x32 píxeles. La ejecución de los modelos han sido realizados a partir de una *CPU intel core i7 de 9a generación* para los modelos clásicos y en una *GPU NVIDIA GeForce GTX 1660* para las operaciones de los tensores de los modelos basados en *machine learning*.

Primeramente, se analizarán los modelos clásicos mediante un *benchmark* con 50000 imágenes. Se puede ver en la figura 13 el resultado gráficamente.

También se puede ver en la tabla 1 los resultados destilados.

Los ejes de los gráficos de la figura 13 representan dichas medidas: el ratio de compresión (bpsp) mide cuántos bits se requieren de media para almacenar un subpíxel (1 dimensión del píxel), los *throughputs* miden la velocidad

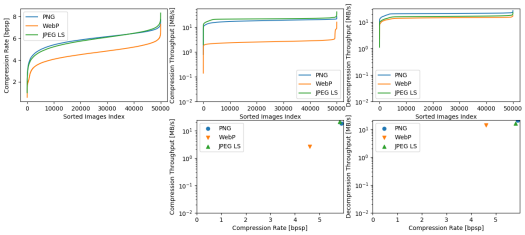

Fig. 13: Benchmark modelos clásicos

Análisis	PNG	WebP	JPEG LS
compresión (bpsp)	5.9	4.6	5.8
compresión (MB/s)	18.0	2.6	21.3
Descompresión (MB/s)	21.2	14.5	16.4

Tabla 1: BENCHMARK MODELOS CLÁSICOS

de *deployment* de los modelos. En ningún caso se tiene en cuenta los tiempos de entrenamiento ya que se utilizan modelos pre-entrenados.

En la figura 13 podemos apreciar, en la primera gráfica, el ratio de compresión de los 3 modelos ordenados de la mayor compresión a la menor compresión; a menor bpsp mayor compresión se habrá conseguido. Se puede apreciar que el modelo WebP obtiene una mayor compresión que los modelos PNG y JPEG LS que se encuentran a la par; en concreto, el modelo WebP tiene una compresión media de 4.6 bpsp mientras que los modelos PNG y JPEG LS tienen una compresión cerca de 5.8 bpsp.
En cuanto a la velocidad de compresión (gráfica 2), el modelo WebP es un orden de magnitud más lento que los modelos PNG y JPEG LS; esto significa que por cada imagen que WebP haya comprimido, los otros modelos habrán comprimido 10.
En cuanto a la descompresión (imagen 3), se puede apreciar que los tres modelos trabajan aproximadamente a la misma velocidad.
En las imágenes 4 y 5 podemos ver una comparación entre la velocidad de compresión/descompresión con el ratio de compresión. Se debe interpretar las gráficas como que a mayor altura y más a la izquierda se encuentren los puntos, de manera más óptima realizarán su operación.

En este caso, podemos hablar de que los modelos PNG y JPEG LS operan de manera equitativa mientras que el modelo WebP adquiere una mayor compresión sacrificando velocidad de compresión. Este resultado es esperado en todos los modelos: es común en la gran mayoría de ámbitos mejorar un aspecto a partir de sacrificar otro. Dependerá de las necesidades del contexto qué modelo se adapta a las necesidades. En este caso dependerá de si se prefiere obtener un 20 % más de compresión o realizar la compresión 10 veces más rápido. De hecho, WebP es un modelo que puede sustituir a los modelos PNG y JPEG LS en esos ámbitos donde sólo es requerida la descompresión (debido a la similitud de velocidades); un ejemplo de este ámbito es en una página web: el cliente solicita la imagen, que será recibida con la compresión WebP realizada y deberá descomprimirla para visualizarla.

Para el siguiente análisis, se bajará la cantidad de imágenes del *benchmark* a 1000 debido a la baja velocidad de compresión de los modelos a estudiar.

Se puede ver en la figura 14 el resultado del *benchmark* del modelo L3C.

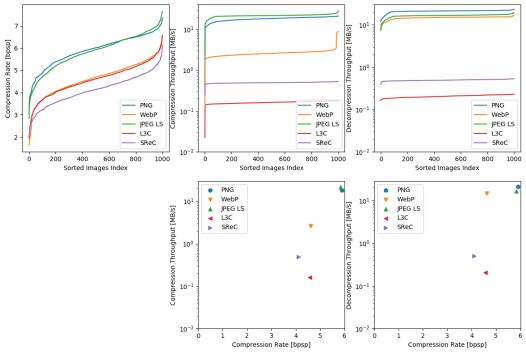

Fig. 14: Benchmark L3C y SReC

También se puede ver en la tabla 2 los resultados destilados.

Análisis	PNG	WebP	JPEG LS	L3C	SReC
compresión (bpsp)	5.9	4.6	5.8	4.5	4.1
compresión (MB/s)	18.0	2.6	21.3	0.2	0.5
descompresión (MB/s)	21.2	14.5	16.4	0.2	0.5

Tabla 2: BENCHMARK L3C Y SREC

En la figura 14 podemos apreciar, en la primera gráfica, que el modelo L3C adquiere unos ratios de compresión muy similares al modelo WebP. Por otro lado, en la segunda imagen vemos que, pese a tener unos ratios muy parecidos, su velocidad disminuye un orden de magnitud; es decir, es diez veces más lento que WebP y cien veces más lento que PNG y JPEG LS. En cuanto a la descompresión es 100 veces más lento que WebP, PNG y JPEG LS.

Con el análisis anteriormente hecho, podemos concluir que el modelo WebP será mejor en todos los ámbitos que el modelo L3C. Por tanto, si se quiere un modelo de alta compresión, será mejor usar WebP que L3C debido a que mejora la velocidad de compresión y descompresión manteniendo la misma bpsp.

En la figura 14 podemos apreciar, en la primera gráfica, que el ratio de compresión del modelo SReC es mayor que el del modelo L3C. En la segunda y tercera gráfica podemos ver que la velocidad de compresión y descompresión es mayor que la del modelo L3C. Por tanto, podemos asegurar con certeza que el modelo SReC es mejor en cualquier caso que el modelo L3C.
Ahora bien, el modelo SReC, pese a conseguir mayor compresión que el modelo WebP, también su tiempo de compresión es 5 veces más lento y el de descompresión 50 veces más lento. Así pues, el dilema actual es el mismo que el analizado con el modelo WebP, pero añadiendo un modelo más: si se requiere de una alta velocida de compresión, el modelo PNG o JPEG LS tendrá mayor peso en la decisión; si se requiere de una mayor velocidad de descompresión el modelo WebP es líder indiscutible debido a que iguala la velocidad con los modelos más veloces analizados y, además, mejora el ratio de compresión; si se quiere un modelo que comprima las imágenes lo máximo posible, el líder será SReC; finalmente, si se requiere de un modelo intermedio, con un muy buen ratio de compresión y una velocidad aceptable, WebP será el modelo a elegir.

7 CONCLUSIONES

A lo largo del documento, se ha realizado una explicación del funcionamiento de los modelos de compresión sin pérdidas basados en *machine learning*. Estos modelos, pueden utilizar distintas técnicas para realizar su cometido; sin embargo, en este documento se ha realizado el estudio de aquellos que se basan en la aplicación de una compresión entrópica condicional a partir de la predicción de distintos pesos obtenidos mediante el entrenamiento de distintos modelos –llamados auto-regresivos–; la elección es debida a que, en la actualidad, los resultados obtenidos con estos modelos son mejores que con los otros tipos de modelos. Todos los modelos de este estilo tienen un módulo de extracción de características y otro de compresión aritmética condicional.
En estos modelos estudiados (L3C, SReC, RC y ICEC) hemos visto algunas ideas interesantes para la extracción de características: En SReC, por ejemplo, se reduce la imagen a un tamaño inferior y, posteriormente, se predicen los 3 píxeles adyacentes a partir de analizar la imagen disminuida dentro de un CNN. Con esta predicción se consiguen unos porcentajes de posibilidad de todos los valores de cada píxel y, con esos valores se realiza la compresión aritmética.

En cuanto al análisis cualitativo de los distintos modelos, vemos que los modelos clásicos siguen siendo los mejores en cuanto a la velocidad de compresión. Sin embargo, en los modelos basados en *machine learninig* se sacrifica mucho el tiempo de compresión y descompresión para obtener una compresión que, actualmente no llega a ser el 20 % mayor que la de, por ejemplo, WebP.
Es en este contexto, en que la aplicación de la compresión influye en la decisión del modelo a utilizar; si se requiere una rápida descompresión sin importar la compresión, es recomendable usar el modelo clásico WebP, pues su ratio de compresión se acerca al de los modelos basados en *machine learning*, pero su velocidad es dos órdenes de magnitud mayor. Por otro lado, si se requiere una gran velocidad de compresión y no hay un problema real de almacenaje o de transmisión, es preferible seguir usando los modelos clásicos y esperar a que se realicen nuevos estudios que mejoren la velocidad de los modelos de *machine learning* o que el mundo del *hardware* evolucione y las velocidades disminuyan. Si se quiere la mayor compresión posible, SReC es acutalmente un modelo muy recomendable; sin embargo, se requerirá de un *hardware* potente y de un tiempo superior. Si se requiere que esté compensado (un tiempo aceptable y una compresión buena) WebP es sin duda el mejor modelo a elegir.

En el proceso de hacer funcionar los modelos se puede

ver una gran facilidad y portabilidad de los modelos clásicos; por otro lado, los modelos basados en *machine learning* requieren muchas dependencias para poder ser ejecutados. Además, dichas dependencias deberán tener unas versiones muy concretas y podrán contener problemas de compatibilidad si se usan ciertas librerías (de *c* o de *Python*) con unas versiones futuras a las esperadas. En el caso de los modelos L3C y SReC, por ejemplo, se utiliza un modelo de compresión entrópica condicional llamada *torchac*, dicho modelo sólo puede ejecutarse si se tiene el módulo *cuda* con la versión 10.1 y el compilador *gcc* con la versión 7.4; en proceso de la ejecución, y debido a la utilización del sistema operativo Ubuntu 22.04, este hecho ha supuesto que se deba descargar manualmente el compilador con la versión 7.4 y todas sus respectivas dependencias con su versión específica –esto es debido a que las versiones de Ubuntu actuales no soportan dichas versiones–. Sin embargo, esto cambia al realizar un *downgrade* del sistema operativo a la 18.04: en este caso, la descarga del compilador a partir del módulo "apt" será suficiente.

AGRADECIMIENTOS

Quiero empezar agradeciendo al coordinador Jordi Pons por recomendarme realizar el TFG con mi tutor Joan Serra. También quiero agradecerle a Joan por ayudarme en este camino de investigación y estudio que, gracias a su comprensión y ayuda, ha resultado gratificante.
Quiero entonar mis mayores gracias a mi madre, que sin ella todo esto no podría haber sido posible.
Quiero dedicarle este pequeño avance en mi vida a Ginebra, Enzo, Lía, Iveth y Sígrid (Quan sigueu grans us ensenyaré això amb il·lusió).

REFERENCIAS

[1] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[2] Sheng Cao, Chao-Yuan Wu, , and Philipp Krähenbühl. Lossless image compression through super-resolution. *arXiv preprint arXiv:2004.02872*, 2020.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[4] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[5] Ning Kang, Shanzhao Qiu, Shifeng Zhang, Zhenguo Li, and Shutao Xia. Pilc: Practical image lossless compression with an end-to-end gpu oriented neural framework. *arXiv preprint arXiv:2206.05279*, 2022.

[6] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[7] Fabian Mentzer, Luc Van Gool, and Michael Tschannen. Learning better lossless compression using lossy compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[8] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[9] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[10] J.D. Pfefferman, P.E. Cingolani, and B. Cernuschi-Frias. An improved search algorithm for fractal image compression. In *ICECS'99. Proceedings of ICECS '99. 6th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.99EX357)*, volume 2, pages 693–696 vol.2, 1999.

[11] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

[12] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

APÉNDICE

Esta sección está pensada para ser completada con información complementaria que se da por sabida durante el *paper*.

A.1. Entropía

En la teoría de la información, la entropía trata de medir el grado de incerteza sobre el posible valor de unas variables. Pongamos un ejemplo: se realiza un juego con un dado de cuatro caras, el juego consiste en intentar acertar, con el menor número de preguntas con respuesta binaria posibles, qué valor ha aparecido. Si el dado no está trucado y la probabilidad de cada cara es de 25 %, entonces se requerirán 2 preguntas siempre (Ver figura 15). La pregunta ahora es, si el dado está trucado y sabemos las probabilidades de cada cara ¿Cuántas preguntas de media se requerirán (con la configuración óptima) para acertar la cara que ha salido? Esto es lo que dice la entropía.

$$H(x) = \sum p(x) \log_2 \left(\frac{1}{p(x)} \right)$$

La gracia aquí está en que primero se preguntarán los valores con más probabilidades de aparecer y se irán dejando los valores menos probables para las preguntas últimas. Es interesante observar como estas preguntas están codificando los distintos posibles valores con secuencias binarias.
El primer teorema de Shannon postula que, sobre un canal que envía palabras con una cierta probabilidad, la longitud mínima de las palabras posibles (con la máxima compresión binaria) será su entropía. Siguiendo el ejemplo del dado, si se quisiera transmitir el valor salido, sería codificar cada valor con el mismo número de bits que preguntas se necesitan para acertar.

Un ejemplo usable en la compresión de imagen está en calcular las probabilidades de cada valor y tratar de buscar una codificación palabra por palabra donde la longitud media de cada palabra sea cercana a la entropía. Para ello hay ciertos algoritmos, como la codificación de Huffman, que lo consiguen. Aún así hay otras codificaciones, no basadas en codificación palabra por palabra, que pueden llegar a aportar una longitud de imagen inferior.

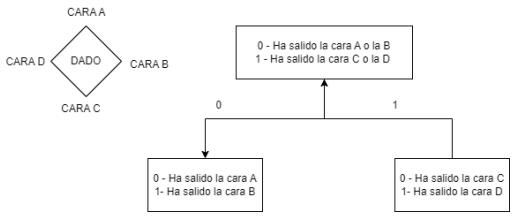

Fig. 15: Juego del dado con p=0.25

A.2. Códigos Shannon

Los códigos de Shannon son códigos instantáneos que te aseguran que la longitud media de palabras se encontrará entre la entropía y la entropía más 1.

$$H(X) \leq L \leq H(X) + 1$$

El primer teorema de Shannon [12] dice que siempre existe un código de Shannon para todos los alfabetos.

Los códigos instantáneos son códigos descodificables de una sola manera y donde la descodificación se puede realizar *on the fly*. Se puede apreciar en la figura 16 la diferencia entre distintos códigos.

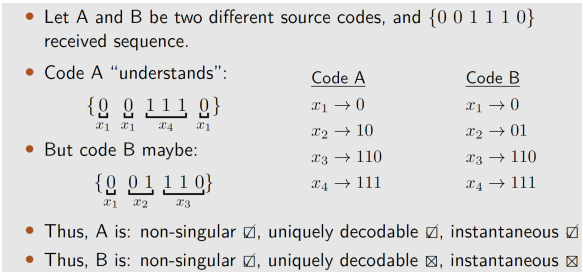

Fig. 16: Códigos Instantáneos

A.3. Codificación Huffman

La codificación Huffman [4] se basa en los códigos de Shannon y, por tanto, la longitud media de las palabras serán inferiores a la entropía más 1; lo que significa que si se pueden agrupar los bits de una secuencia en grupos con una baja entropía, se conseguirá una alta compresión. Además, los códigos Huffman son óptimos y, por tanto, no se encontrará un código más comprimible que el Huffman cuando la codificación es de símbolo a símbolo.
La forma de codificación de Huffman está basada en la probabilidad de aparición de una secuencia (llamémosle palabra): a menor aparición, menos bits le asignaremos y a la inversa. Por tanto, primeramente deberemos crear palabras de una longitud arbitraria y posteriormente calcular su porcentaje de aparición. Con esto ya realizado, la codificación Huffman se realiza como se puede apreciar en la figura 17, donde se van agrupando las palabras con menos probabilidad hasta completar todo el árbol binario. Posteriormente se le da un valor 1 o 0 a cada rama (siempre siguiendo el orden de asignación).

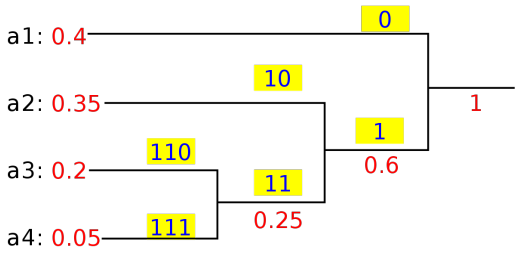

Fig. 17: Codificación Huffman

A.4. LZ77

El algoritmo LZ77 (Lempel-Ziv 1977) [13] se basa en obtener una compresión mediante el remplazo referencial de secuencias de valores repetidas. Este algoritmo itera secuencialmente dentro de una lista de valores mediante una ventana deslizante. Esta ventana deslizante funciona como limitador de búsqueda de reiteración. Cuando una secuencia analizada se ha encontrado dentro de la ventana deslizante,

Paso	Posición	Ventana	Coincidencia	Valor	Output
1	1	-	-	A	(0,0)A
2	2	A	A	-	(1,1)
3	3	AA	-	B	(0,0)B
4	4	AAB	AB	-	(2,2)
5	6	BAB	-	C	(0,0)C
6	7	ABC	B	-	(5,1)
7	8	BCB	-	A	(0,0)A
8	9	CBA	A	-	(8,1)
9	10	BAA	B	-	(7,1)
10	11	AAB	-	C	(0,0)C

Tabla 3: CODIFICACIÓN LZ77

se deberá añadir el siguiente elemento de la lista a la secuencia y analizar si aún se encuentra una secuencia idéntica en la ventana. En caso negativo, la secuencia inicial será referenciada a la posición de la secuencia idéntica de la ventana. Veamos un ejemplo:
Supongamos la secuencia $AABABCBAABC$ con una ventana 3. Se puede ver en la tabla 3 cómo la codificación se efectuaría. Vemos como el *output* puede ser o una referencia a una dirección seguido de qué longitud de cadena referencía o una referencia de (0,0) y el valor que no ha podido referenciar. Vale decir que en este ejemplo de plastelina la compresión es mayor a 1 y, por tanto, es ineficiente, pero en la realidad –y con una ventana mayor– la compresión que se consigue es muy buena. También cabe decir que en el ejemplo se ha usado la dirección absoluta, pero se podría usar la dirección relativa de la ventana (útil ya que la codificación requerirá menos bits).

A.5. Transformada Discreta del Coseno

La DCT [1] es un tipo de transformación aplicada sobre una función discreta. Dicha transformación va del grupo de los reales al grupo de los reales ($\mathbb{R} \to \mathbb{R}$). Algunas de sus propiedades más importantes son que es invertible y que puede ser aplicada en datos multidimensionales. Su definición para 1-D es:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos\left(\frac{(2x+1)u\pi}{2N}\right)$$

Donde:

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \in \mathbb{N} \end{cases}$$

El motivo para que se llame Transformada Discreta del Coseno queda bastante obvio ahora. La inversa de la Transformada es:

$$f(x) = \sum_{u=0}^{N-1} \alpha(u) C(u) \cos\left(\frac{(2x+1)u\pi}{2N}\right)$$

Lo que podemos observar es que se puede construir $u = N$ vectores sobre los cuales su producto escalar con el vector $f(x)$ darán un elemento del vector $C(u)$, en la figura 20 se puede ver, en escala de grises, la matriz que irá mutiplicada por el vector $f(x)$. Se puede apreciar cierto comportamiento frecuencial con distintas frecuenias: siendo la fila 1 un comportamiento de corriente continua, la fila 2 medio coseno, la fila 3 un coseno completo, la fila 4 un coseno y medio... (ver Código 1). Un dato curioso sobre la matriz creada es que demuestra la invertibilidad de la transformación, pues la mutiplicación de la matriz por su transpuesta da como resultado la matriz identidad.

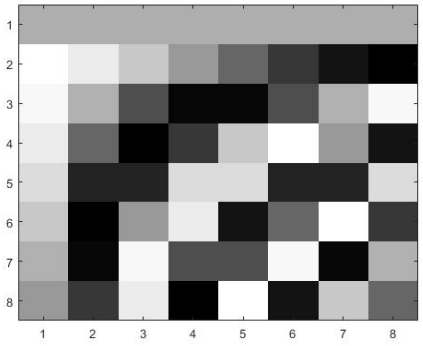

Fig. 18: Matriz de producto escalar DCT 1D.

Aun así, en nuestro caso nos interesa tratar con matrices, por tanto, la DCT que nos interesa es la 2-D. Su formulación es:

$$C(u,v) =$$

$$\alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\left(\frac{(2x+1)u\pi}{2N}\right) \cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

y la inversa:

$$f(x,y) =$$

$$\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \alpha(u)\alpha(v) C(u,v) \cos\left(\frac{(2x+1)u\pi}{2N}\right) \cos\left(\frac{(2y+1)v\pi}{2N}\right)$$

Sobre la formulación ya es fácil deducir cómo sería para N dimensiones. Veamos ahora cómo se verían las distintas matrices que multiplicarían escalarmente (punto a punto y suma de puntos), pues la representación es más interesante que la del caso 1-D. Se puede ver en la figura 19 el comportamiento de la transformada, donde cada matriz va mutiplicada por la matriz imagen (del mismo tamaño) y a la matriz resultante se le suman todos los valores. Cada valor conformará un punto de la matriz resultante. Con la imagen queda muy claro cómo la transformada está obteniendo información de las distintas frecuencias. (ver código 2 si se quiere replicar).

Aunque la versión de DCT introducida es la más usada, hay distintas transformadas, que van desde la DCT-I hasta la DCT-VIII. En este caso, se ha explicado la DCT-II. Aún así, todas siguen el mismo patrón lógico y difieren en el comportamiento del coseno (aumentando las frecuencias usadas o disminuyéndolas).

A.6. Transformación Walsh-Hadamard

La transformación de Walsh-Hadamard es una transformación homóloga a la Transformada Discreta de Fourier, pues resulta de la DFT multidimensional $2 \times 2 \times ... \times 2 \times 2$ normalizada y descompuesta en una matriz 2-D. Lo especialmente curioso de esta transformada es que los valores de su matriz solo serán 1 o -1 (cosa lógica si se piensa en

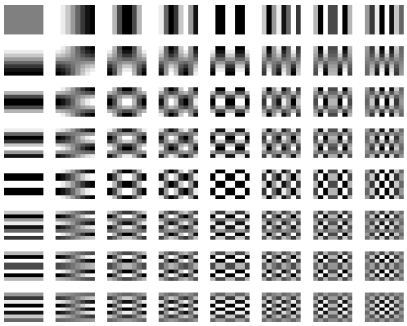

Fig. 19: Matriz de producto escalar DCT 2D.

la DFT de vector de 2). Además, los vectores de la matriz serán todos ortogonales y la matriz será simétrica e invertible. Veamos como se construye la matriz: la construcción tiene un sentido de ser recursivo, pues se construye a partir de la creación de matrices anteriores:

$$H_m = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix}$$

Donde $H_0 = (1)$.
Por ejemplo, la matriz H_2 quedaría tal que:

$$H_2 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

De este método recursivo podemos darnos cuenta que el tamaño de la matriz siempre será $M \times M$, donde M puede tomar cualquier valor potencia de 2.

A.7. Tensor

Un tensor es un objeto algebraico, que se construye como una generalización de los escalares, vectores y matrices; pues un escalar es un tensor de dimensiones 0 y una matriz es un tensor de dimensiones 2. Así pues, un tensor de dimensión 3 sería un objeto con valores en las 3 dimensiones.

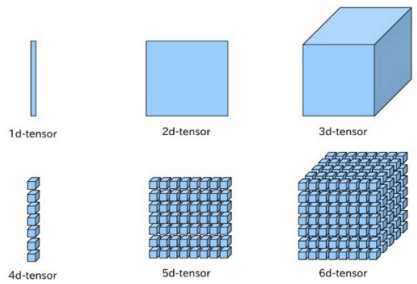

Fig. 20: Ejemplos de Tensores.

A.8. PixelCNN

PixelCNN [8] se trata de un modelo generativo de imágenes basado en la arquitectura CNN. Este modelo se basa en predecir el píxel en una ubicación de la imagen a partir de los píxeles anteriores. El diseño es útil para muchas aplicaciones: reconstrucción de imagen, *impainting*, *deblurring*, superresolución y, el caso que nos concierne, compresión de imagen. El sistema de compresión de imagen se basa en la predicción del píxel actual a partir de los anteriores y la obtención del error. Para entenderlo de forma teórica, se define la probabilidad total de la imagen como:

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1})$$

donde *n* es la longitud de una imagen cuadrada.
La función de probabilidad condicionada se podrá escribir como $p(x_i|\mathbf{x}_{<i})$ o de forma equivalente $\mathcal{L}(\mathbf{x}_{<i}|x_i)$. Así pues, la probabilidad de un píxel en concreto vendrá condicionada por todos los píxeles anteriores.
Por otro lado, cada píxel contendrá 3 valores (RGB), cuyo valor también podrá depender del conocimiento de los otros:

$$p(x_i|\mathbf{x}_{<i}) = p(x_{i,R}|\mathbf{x}_{<i}) p(x_{i,G}|\mathbf{x}_{<i}, x_{i_R}) p(x_{i,B}|\mathbf{x}_{<i}, x_{i,R}, x_{i,G})$$

En la actualidad, este modelo se ha convertido en el estado de arte de muchos modelos basados en CNN que implementan el mismo principio. Se debe interpretar pues, este modelo como un principio teórico para la predicción de píxel y no como un modelo aplicable. Sin embargo, veamos su estructura.

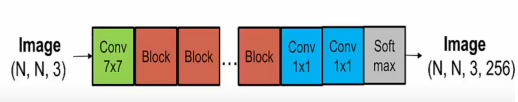

Fig. 21: Arquitectura PixelCNN

En la figura 21 podemos ver una imagen general de la estructura. Primero se realiza una convolución con un *kernel* 7x7 seguida de bloques residuales con convoluciones internas; seguidamente dos bloques de convolución con un *kernel* de 1x1 y finalmente la aplicación de un *softmax* para finalizar con una imagen de dimensiones (N,N,3,256) en la que se especificará la probabilidad en cada sub-píxel de poseer un valor sobre los 256 posibles.

Entrando en la estructura más interna, se realiza el seguido de convoluciones visto en la figura 21 para la dimensión R. Por otro lado, para la dimensión G se incluye en la convolución cierta información de la dimensión R y para la dimensión B se incluye de las R y G.

Otro concepto importante en este modelo es el *kernel mask*. Se trata de utilizar solo la parte del *kernel* anterior al píxel actual para evitar tomar predicciones de píxeles posteriores (cosa que no será viable en la descompresión); para ello, los *kernel* deberán poseer una máscara con una forma parecida a esta pero con unas dimensiones arbitrarias:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Esto posibilitará que siempre se consiga extraer la información de los píxeles anteriores, pero nunca de los futuros.

A.9. PixelCNN++

PixelCNN++ [11] es una re-implementación del modelo PixelCNN realizada por ingenieros de OpenAI. Su principal objetivo está en mejorar ciertos aspectos técnicos de PixelCNN que mejorarán los tiempos de ejecución. Aún así, los principios son los mismos.

B CÓDIGO

```
% $Author: Xavier Fernandez $    $Date:
    2022/10/14 17:23:52 $    $Revision: 0.1 $
% Copyright: Free Use

N = 8;
a_0 = sqrt(1/N);
a_n = sqrt(2/N);

c = cos(((2*[0:N-1]+1).*[0:N-1]')*pi/(2*N));
c(1, :) = c(1, :)*a_0;
c(2:N, :) = c(2:N, :)*a_n;

imagesc(c);
colormap(gray);
```

Código 1: Matriz DCT 1-D

```
% $Author: Xavier Fernandez $    $Date:
    2022/10/14 17:56:08 $    $Revision: 0.1 $
% Copyright: Free Use

N = 8;
a_0 = sqrt(1/N);
a_n = sqrt(2/N);

c = cos(((2*[0:N-1]+1).*[0:N-1]')*pi/(2*N));
c(1, :) = c(1, :)*a_0;
c(2:N, :) = c(2:N, :)*a_n;


for i = 1 : N
    for j = 1 : N
        subplot(N,N, (j-1)*N+i)
        imagesc(c(i, :).*c(j, :)')
        colormap(gray);
        axis off;
    end
end
```

Código 2: Matriz DCT 2-D

```
% $Author: Xavier Fernandez$
% $Date: 2022/10/27 14:59:34$
% $Revision: 0.2$
% $Copyright: Free Use$

I = imread("../img/Lenna.png");
sz = size(I);
sz = [sz(1)+2,sz(2)+2,sz(3)];
I2 = uint8(zeros(sz));

gauss = (1/16).*[1 2 1; 2 4 2; 1 2 1];

I2(:,:,1) = uint8(ceil(conv2(I(:,:,1),gauss)));
I2(:,:,2) = uint8(ceil(conv2(I(:,:,2),gauss)));
I2(:,:,3) = uint8(ceil(conv2(I(:,:,3),gauss)));

subplot(1,4,1), imshow(I)
subplot(1,4,2), imshow(I2)
subplot(1,4,3), imshow(I(120:220,100:200,:))
subplot(1,4,4), imshow(I2(120:220,100:200,:))
```

Código 3: 3x3 Gaussian Blur