UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA
UDELAR

# Redes neuronales de aprendizaje profundo para compresión de imágenes sin pérdida

Gabriela Rodríguez
Crhistyan Silva

Proyecto de grado presentado a la Facultad de Ingeniería de la Universidad de la República en cumplimiento parcial de los requerimientos para la obtención del título de Ingeniero en Computación.

Tutores
Álvaro Martín
Gadiel Seroussi

Tribunal
Mercedes Marzoa
Alberto Castro
Gustavo Brown

Montevideo, Uruguay
Noviembre de 2022

# Índice general

# Capítulo 1

# Introducción

## 1.1. Descripción del problema

La compresión de imágenes es un tema importante debido a que muchas de las imágenes utilizan volúmenes de datos muy altos, del orden de mega y giga bytes. Comprimirlas permite almacenar o transmitir los datos de las imágenes reduciendo estos volúmenes, y por consecuencia reduciendo costos.

Informalmente, la *compresión* de imágenes, y de datos en general, consiste en codificar información de forma tal que requiera un menor número de bits que la representación inicial. En *compresión sin pérdida*, no se pierde información en el proceso, de modo que dicho proceso es reversible, es decir, se puede obtener exactamente la misma información que se tenía inicialmente. Por otro lado, en *compresión con pérdida* se utilizan ciertas aproximaciones que no necesariamente permiten reconstruir los datos originales de manera exacta.

La compresión de imágenes, tanto sin pérdida como con pérdida, es un área ampliamente desarrollada. En la actualidad, existen algoritmos de compresión con pérdida tradicionales como JPEG[15], BPG[3], WebP-lossy[23] y algoritmos sin pérdida como PNG[16], LOCO-I[51], WebP-lossless[24], FLIF[44], que logran tasas de compresión competitivas. Sin embargo, estos algoritmos no aprovechan el gran éxito que han tenido las redes neuronales para modelar imágenes, lo cual ha despertado un gran interés en aplicar este tipo de modelos para comprimirlas eficientemente. Por ejemplo, desde 2018 Google patrocina anualmente una competencia internacional para premiar los algoritmos que alcancen el mejor desempeño[11].

En resumen, lo que se necesita para una compresión sin pérdidas es una distribución de probabilidad de los datos que se está tratando de comprimir y un algoritmo de codificación de entropía que transforme los datos en un flujo de bits, utilizando dicha distribución para codificar de manera óptima, es decir, con la menor cantidad posible de bits, por ejemplo, dando una codificación más corta a los datos más probables y una más larga a los menos probables. Los métodos de aprendizaje automático entran en juego para aproximar esa distribución de probabilidad o transformarla para aprovecharla mejor.

El aprendizaje automático en los últimos años ha demostrado gran eficiencia al resolver problemas que a priori podían parecer muy difíciles, en particular el uso de redes neuronales ha abierto puertas en el área de investigación aproximando funciones muy complejas en espacios de muchas dimensiones. Aprovechando el aprendizaje de estas funciones complejas a partir del uso de una gran cantidad de datos, como los que se tiene hoy en día, es que en los últimos años se han desarrollado gran cantidad de algoritmos de compresión con pérdida con el uso de varios tipos de redes, como redes convolucionales [35], autoencoders [45] y redes recurrentes [46]. Además, en los últimos dos años se han desarrollado muchos algoritmos con el objetivo que trabajamos en este proyecto, comprimir imágenes sin pérdida usando redes neuronales, ejemplo de estos son L3C[29], IDF[13], IDF++[4], Hyperprior[5], RC[27], L3C[29], HiLLoC[49] y LBB[12].

En este proyecto nos planteamos explorar la aplicación de redes neuronales para la compresión de imágenes sin pérdida, que es un área de aplicación incipiente. El principal objetivo del proyecto es estudiar detalladamente el estado del arte en este tema y generar una implementación completa que, a partir de este estudio, ofrezca comparativamente una buena relación de compromiso entre nivel de compresión y eficiencia de cómputo.

El primer paso para lograr nuestro objetivo fue el estudio del estado del arte. Luego de confirmar que los algoritmos mencionados anteriormente son realmente sin pérdida de información procedimos a compararlos entre sí. Los algoritmos en cuestión reportan sus resultados sobre base de datos diferentes y por consecuente la comparación entre ellos no pudo hacerse de forma directa. Por un lado teníamos un grupo de algoritmos, HiLLoC, IDF, IDF++ y LBB que usan las bases de datos CIFAR10, ImageNet32 y ImageNet64, bases con imágenes de resolución $32 \times 32$ y $64 \times 64$. Por otro, los algoritmos RC, L3C y HyperPrior

que usan las bases CLIC.pro, CLIC.mobile, Open Images y DIV2K, las cuales tienen una resolución mayor que las bases usadas por los primeros algoritmos mencionados, en un rango de $512 \times 384$ y $2048 \times 2048$.

Para lograr la comparación entre ellos se eligió el algoritmo que mejor resultados daba, en cuanto a tasa de compresión, de cada grupo, LBB y RC, y procedimos a compararlos entre ellos en los conjuntos de datos de mayor resolución. Para obtener dicha comparación tuvimos que adaptar el algoritmo LBB que originalmente funcionaba con las bases de imágenes con menor resolución, usando el código y modelos ya disponibles.

Como resultado de las comparaciones experimentales concluimos que el algoritmo que daba mejores resultados, en este caso, fue el algoritmo LBB, el cual logró un mejor desempeño en todos los conjuntos de datos.

Dado que el algoritmo LBB resulto el más eficiente en cuanto a tasa de compresión, pero a su vez la implementación del entrenamiento de sus modelos no es pública implementamos nuestro propio entrenamiento para el conjunto de datos de mayor resolución dentro de los posibles, ImageNet64.

Experimentamos con dos modelos distintos $LBB'$, un modelo entrenado por nosotros desde cero con imágenes del conjunto de datos ImageNet64, y $LBB''$ una continuación del entrenamiento de $LBB'$ con un conjunto de datos propio denominado Crops, el cual realiza cortes de $64 \times 64$ a imágenes de mayor tamaño. Como resultado, concluimos que el mejor modelo que aproxima al algoritmo original es $LBB'$.

En el trabajo original de LBB [12] los resultados reportados para la tasa de compresión no tienen en cuenta cierta redundancia usada para comprimir. El algoritmo tiene la característica de comprimir secuencia de imágenes, y para lograr esto opera a partir de una cadena de bits iniciales generados de manera aleatoria en la cual se extrae y se agrega bits a medida que se comprime. A su vez, estos datos iniciales se recuperan exactamente del lado del descompresor, por lo que se consideran "información transferida", y no se cuentan en el costo de transmisión de las imágenes. En nuestro trabajo reducimos este problema aprovechando esa información inicial. Para esto, en lugar de tomar una cadena inicial de datos generados de manera aleatoria primero se realiza una etapa de compresión de las imágenes utilizando el algoritmo PNG. Estos datos se usan para la cadena inicial y cuando se obtienen suficientes datos se pasa de nuevo al algoritmo original LBB aprovechando esta información, que una vez que se descomprime con LBB se recupera nuevamente.

Luego, comparando nuestro algoritmo final con los clásicos sobre la base de datos ImageNet64, obtuvimos que a partir de 125 imágenes alcanza las tasas de compresión reportadas por los algoritmos clásicos, y a partir de esta cantidad empieza a mejorarlas, alcanzando una tasa de compresión un 19 % mejor que la reportada por el mejor de los algoritmos clásicos. Además con la mejora propuesta para reducir el costo de los bits iniciales logramos que la tasa mejore al principio y no afecte a la tasa obtenida previamente para grandes conjuntos de imágenes.

## 1.2. Organización del documento

Este trabajo comienza con las definiciones y conceptos en el capítulo 2, dando una introducción a la compresión y aprendizaje automático con redes neuronales para comprender los capítulos siguiente donde analizamos los algoritmos clásico, en el capítulo 3, y los algoritmos con redes neuronales, en el capítulo 4. Luego describimos en detalle los conjuntos de datos utilizados en el capítulo 5 para detallar el procedimiento para la experimentación u obtención de resultados en los capítulos 6 y 7. El código de este trabajo está disponible en github[1].

[1] https://github.com/CrhistyanSilva/proyecto_grado

# Capítulo 2

# Conceptos

Este capítulo tiene como objetivo introducir conceptos clave manejados en compresión de imágenes y aprendizaje automático para la correcta comprensión de los capítulos siguientes.

Comenzamos explicando los conceptos usados en los algoritmos clásicos que no usan redes neuronales, los cuales presentamos en el capítulo 3. Luego, damos una introducción al área de aprendizaje automático, para describir el proceso de entrenamiento y evaluación de una red neuronal. En esta parte se mencionan conceptos principales para ser capaces de abordar nuestro estudio del estado del arte en el capítulo 4. Además, varios de estos conceptos son tratados en los capítulos 6 y 7 por lo que es esencial para comprender correctamente el análisis y evaluación de los algoritmos.

## 2.1. Representación de una imagen

Una posible representación de una imagen es una matriz en la que cada entrada contiene información de la intensidad lumínica o color de un punto de la imagen o *píxel*. Decimos que una imagen es de un *canal* si cada lugar de la matriz tiene un único número, por ejemplo, imágenes en escala de grises donde cada valor de la matriz representa la intensidad lumínica del píxel. Por otro lado, las imágenes de varios canales son aquellas que en cada píxel contienen más de un número para representar, por ejemplo, el color en un formato dado, como una imagen RGB donde se tienen tres números para la intensidad de rojo, verde y azul de ese píxel, respectivamente.

## 2.2. Códigos

Dado un *alfabeto de codificación* $A$, denotamos con $A^*$ el conjunto de cadenas de largo finito sobre A,

$$A^* = \bigcup_{k=1}^{k=\infty} A^k \ ,$$

donde $A^k$ es el conjunto de cadenas de largo $k$ sobre A. Usamos $|\cdot|$ para denotar tanto el tamaño de un conjunto como el largo de una cadena. Dada una variable aleatoria $X$, sobre un *alfabeto fuente* $\chi$, decimos que un *código* $C$ es un mapeo de $\chi$ en $A^*$. Cada símbolo $x$ de $\chi$ tiene asignado una palabra de código $C(x)$, cuyo largo denotamos con $l(x)$, es decir, $l(x) = |C(x)|$. Se define el *largo de código medio* de un código $C$, $L(C)$, para una variable aleatoria $X \sim p_X$, como

$$L(C) = \sum_{x \in \chi} p_X(x) \cdot l(x) \ .$$

Decimos que un código es *no singular* si cada elemento de $X$ se mapea a una palabra de código diferente, es decir, si $x \neq x'$ implica $C(x) \neq C(x')$. Definimos la *extensión* $C^*$ de un código $C$ como el mapeo de secuencias de símbolos de $X$ a secuencias de $A$ definida por

$$C(x_1 x_2 \ldots x_n) = C(x_1)C(x_2)\ldots C(x_n) \ ,$$

donde $C(x_1)C(x_2)\ldots C(x_n)$ es la concatenación de las palabras de código asignadas a $x_1 \ldots x_n$.

Decimos que un código es *unívocamente decodificable* si su extensión es no singular, lo que quiere decir que no hay ambigüedades al momento de decodificar una secuencia. Dentro de esta clasificación tenemos los códigos *instantáneos* o *de prefijo*, en donde ninguna palabra del código es prefijo de otra. La figura 2.1 ilustra esta clasificación.

Para todo código de prefijo se cumple la *desigualdad de Kraft* [6], $\sum_{x \in X} D^{-l(x)} \leq 1$, donde $D = |A|$ y $l(x_1), l(x_2), \ldots, l(x_m)$ son los largos de palabra del código. Además es posible demostrar que dado un conjunto de largos de código que cumple la desigualdad, existe un código de prefijo con esos largos. McMillan [26] demostró que todo código unívocamente decodificable cumple con la desigualdad de Kraft. Como para todo código que cumple dicha desigualdad existe uno de prefijo con los mismos largos, no se pierde nada limitándose a códigos de prefijo.

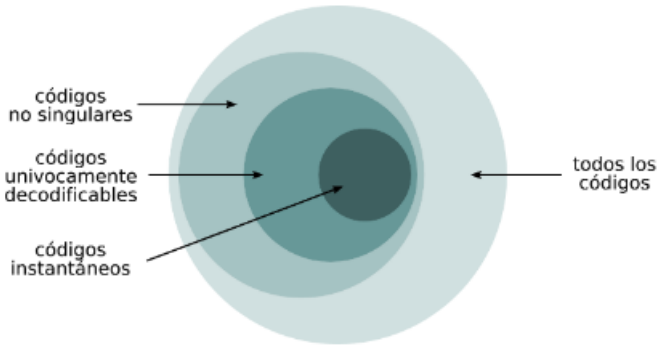



Figura 2.1: Clasificación de códigos [42].

## 2.3. Tasa de compresión

Para medir la eficiencia al comprimir una cadena de símbolos $x_1 \dots x_n$ se usa la *tasa de compresión*, que es el cociente entre el largo de la codificación y el largo de la entrada,

$$t_p(x_1 \dots x_n) = \frac{|C(x_1 \dots x_n)| \log_2 D}{n} ,$$

que se mide en bits por símbolo. Para el caso de una imagen, la unidad de medida para la tasa es *bits por pixel* (BPP). En el caso de imágenes con más de un canal, también se usa como unidad de medida *bits por sub pixel* (BPSP), también llamada *bits por dimensión* (BPD), donde la tasa de compresión se calcula considerando cada componente de cada píxel como un símbolo separado

$$t_d(x_1 \dots x_n) = \frac{|C(x_1 \dots x_{dn})| \log_2 D}{dn} ,$$

donde $d$ es la cantidad de canales y $n$ es la cantidad de píxeles de la imagen.

## 2.4. Algoritmos de codificación de entropía

La idea de estos algoritmos es utilizar un modelo de distribución de probabilidad $p$ sobre los mensajes a codificar asignando códigos más cortos a los mensajes más probables y códigos más largos a los menos probables.

### 2.4.1. Codificación de Huffman

La *codificación de Huffman* [1], propuesto por David A. Huffman en 1952, es un algoritmo para la construcción de un código de prefijo óptimo, es decir, que minimiza el largo de código esperado. Formalmente, dado un alfabeto fuente $\chi$ y una distribución de probabilidad sobre $\chi$, el objetivo es encontrar un código de prefijo cuyo largo de código medio, $L(C)$, cumple que $L(C) \leq L(T)$, para todo código de prefijo $T$ sobre $\chi$.

El algoritmo para la construcción de un código de Huffman para un alfabeto de codificación $A$ con $|A| = D$, consiste en la creación de un árbol $D$-ario que tiene cada uno de los símbolos de $\chi$ como nodos hojas del árbol (nodos que no tienen ningún hijo). Cada una de las $D$ aristas salientes de un nodo interno está etiquetada con un símbolo distinto de A, de manera tal que siguiendo el camino de la raíz a cada una de las hojas se obtiene la palabra de código asociada a cada símbolo.

### 2.4.2. Codificación aritmética

La *codificación aritmética* (arithmetic coding, AC) [14], al igual que Huffman se fundamenta en el conocimiento *a priori* de la distribución de probabilidades de los símbolos a codificar. En este caso, en lugar de codificar cada símbolo por separado, se codifica una secuencia de símbolos, $x_1 \dots x_n$, usando la cadena de dígitos $D$-arios a la derecha de la coma en la representación en base $D$ de un número $y$, $(0 \leq y < 1)$.

Para lograrlo se parte el intervalo entre $[0, 1)$ en subintervalos disjuntos asociados a cada uno de los símbolos a codificar, cada uno con un largo proporcional a su probabilidad. El intervalo correspondiente a $x_1$ se parte nuevamente en subintervalos, cada uno asociado a un símbolo distinto de $\chi$ de largo proporcional a su probabilidad condicional dado que el primer símbolo es $x_1$. Continuando con este procedimiento, luego de $n$ pasos se llega a un intervalo que identifica la secuencia completa $x_1 \dots x_n$. La palabra de código para $x_1 \dots x_n$ se obtiene de la representación $D$-aria de un número $y$ de ese intervalo, con precisión suficiente para distinguir a qué intervalo pertenece.

Se puede demostrar ([34]) que la longitud del código, medida en bits, para una secuencia de símbolos, $x_1 \dots x_n$, con probabilidad $p$, asignada por un codificador aritmético adecuadamente diseñado satisface:

$$L(x_1 \dots x_n) < \log \frac{1}{p(x_1 \dots x_n)} + 2 \ ,$$

por lo que el codificador es óptimo hasta una redundancia de dos bits.

De aquí en más, si no se específica la base del algoritmo, suponemos que es 2.

### 2.4.3. Sistemas Numéricos Asimétricos

Los *sistemas numéricos asimétricos* (asymmetric numeral systems, ANS) [8] son una familia de métodos de codificación de entropía, una nueva variante de los AC, que consigue un desempeño similar al mencionado en la sección 2.4.2. En este caso describimos la variante *range-ANS* (rANS). Esta variante codifica una secuencia de símbolos en un único número natural, que se calcula secuencialmente al procesar cada símbolo.

Dado un alfabeto $\chi = \{x_1, ..., x_k\}$ y una distribución de probabilidad sobre $\chi$, $P = \{p_1, \dots, p_k\}$, se definen naturales $F = \{F_{x_1}, ..., F_{x_k}\}$, y $M = \sum_{i=1}^{k} F_{x_i}$, de forma tal que $\frac{F_{x_i}}{M}$ es una aproximación de $p_i$. Se define también $B_{x_i} = \sum_{j=1}^{i-1} F_{x_j}$. Entonces, para t $= 1 \dots n$, se define el estado $N_t$, a partir del estado anterior $N_{t-1}$ y el símbolo $x_t$, recursivamente como

$$N_t = C_{rANS}(N_{t-1}, x_t) = \left\lfloor \frac{N_{t-1}}{F_{x_t}} \right\rfloor \cdot M + B_{x_t} + \mathrm{mod}(N_{t-1}, F_{x_t}) \ ,$$

donde $\mathrm{mod}(a, b)$ es el resto de dividir $a$ entre $b$. La salida final es $N_n$, siendo $n$ la cantidad de símbolos codificados, y se representa usando $\lceil \log_2 N_n \rceil$ bits.

En la práctica $N_t$ empieza a crecer rápidamente, lo que puede derivar en un problema de precisión numérica para entradas moderadamente grandes. Para evitarlo se define una variante denominada *streaming rANS*, cuyo objetivo es mantener el estado en un rango predefinido. Para ello, los bits menos significativos de $N_t$ son emitidos al stream de salida del codificador a medida que se detecta que dichos bits no van a cambiar en futuras iteraciones, lo cual permite descartarlos de $N_t$ y ajustar su valor para mantenerlo dentro del rango prescrito.

## 2.5. Redes neuronales

El objetivo de esta sección es dar una introducción a los conceptos básicos de las redes neuronales, lo cual será fundamental para la comprensión del análisis de los trabajos realizados hasta la fecha para compresión de imágenes sin pérdida basados en redes neuronales,

presentados en el capítulo 4, y de la experimentación realizada con estos algoritmos en los capítulos 6 y 7.

Nos referimos por algoritmos de aprendizaje automático, a aquellos algoritmos que analizan datos y aprenden de los datos analizados para descubrir patrones de interés. Existen distintas estructuras y formas de desarrollo para estos algoritmos. En este caso nos centraremos en el modelo denominado *redes neuronales*.

### 2.5.1. Estructura

Para comprender la estructura de las redes neuronales, presentamos la estructura más básica que puede tener una red, denominada *unidad sigmoide*.

Dado un vector de entrada $x^T = (x_1, \ldots, x_n) \in \mathbb{R}^n$, la red neuronal más simple que puede construirse es equivalente a la regresión logística. Está compuesta por una primera capa con los atributos de entrada (conocida como *capa de entrada*, la denotamos $a^{(1)}$), y una segunda capa compuesta por una unidad sigmoide, que calcula una combinación lineal de las entradas y le aplica la *función sigmoide* (conocida como *función de activación*, definida en la sección 2.5.6), para obtener una salida real. A partir de esta salida, seremos capaces de tomar una decisión. En la figura 2.2 podemos observar como luce esta estructura.

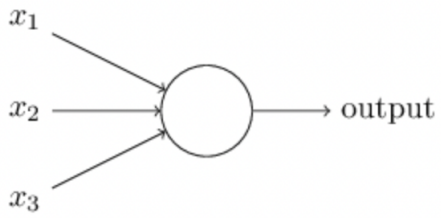



Figura 2.2: Estructura unidad sigmoide [32].

Tenemos entonces la capa de entrada

$$a^{(1)} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix},$$

y para esta red de una sola neurona, la segunda capa es

$$a^{(2)} = g(w^{(1)} \cdot x + b^{(1)}) \ , \tag{2.1}$$

siendo $w^{(1)} = (w_1^{(1)} \dots w_n^{(1)})$ el conjunto de parámetros para la combinación lineal calculados por la neurona de la capa 2 (también conocidos como *pesos*), $b^{(1)} \in \mathbb{R}$ un término independiente de sesgo, y $g(z)$ la función de activación, que como mencionamos anteriormente, para este caso particular es la función sigmoide. En la sección 2.5.6 presentamos otras posibles funciones de activación.

Las redes neuronales son una generalización del ejemplo anterior. En cada una de las capas puede haber más de una neurona, que recibe como entrada los resultados de la capa anterior. Además entre la capa de entrada y la capa de salida pueden existir capas intermedias (conocidas como *capas ocultas*). En la figura 2.3 podemos observar un ejemplo de esta estructura.

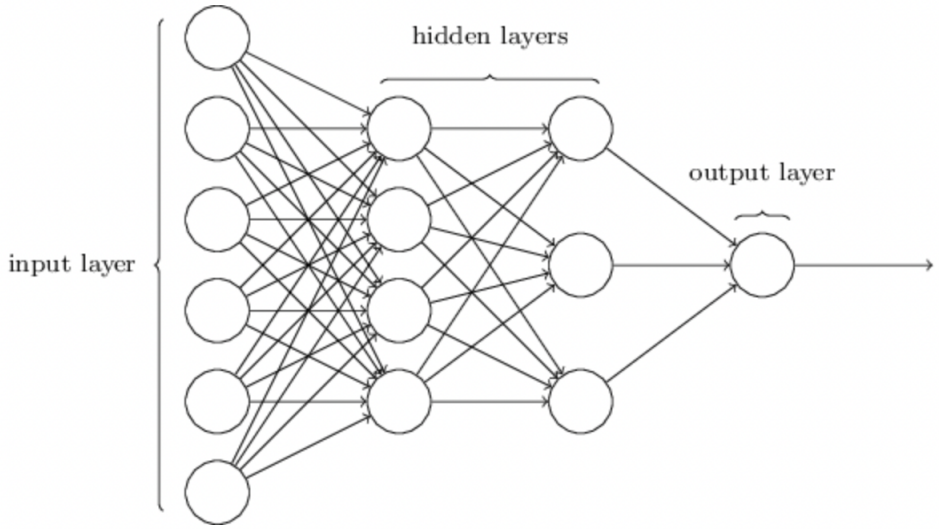



Figura 2.3: Estructura de una red neuronal [32].

Por lo tanto, generalizando el caso anterior tenemos:

$$a^{(1)} \in \mathbb{R}^n, a^{(1)} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$a^{(j)} \in \mathbb{R}^{s_j}, a^{(j)} = \begin{bmatrix} a_1^{(j)} \\ \vdots \\ a_{s_j}^{(j)} \end{bmatrix} = g(W^{(j-1)} \cdot a^{(j-1)} + b^{(j-1)}) \ ,$$

siendo $s_j$ el número de neuronas en la capa $j$, $W^{(j-1)}$ la matriz de pesos que define el mapeo desde la capa $j-1$ a la capa $j$, $b^{(j-1)}$ el vector de sesgo que en el caso mas general incluye un componente por cada neurona de la capa anterior,

$$b^{(j)} \in \mathbb{R}^{s_j}, b^{(j)} = \begin{bmatrix} b_1^{(j)} \\ \vdots \\ b_{s_j}^{(j)} \end{bmatrix} \ .$$

Podemos observar que $W^{(j-1)} \in \mathbb{R}^{s_j} \times \mathbb{R}^{s_{j-1}}$ tiene tantas filas como neuronas hay en la capa $j$, y tantas columnas como neuronas hay en la capa $j-1$. Cada valor $w_{ik}^{(j)}$ de la matriz de pesos se lee como el peso asociado a la i-ésima neurona de la capa $j$, correspondiente a la entrada proveniente de la k-ésima neurona de la capa $j-1$.

### 2.5.2. Etapas en la construcción de una red neuronal

El proceso de crear una red suele dividirse en tres etapas: *entrenamiento*, *validación* y *evaluación*.

La etapa de entrenamiento consiste en que la red aprenda a buscar patrones y generar cierta salida mediante la modificación de sus pesos, a partir de un conjunto de datos específico para el entrenamiento. La etapa de validación tiene como objetivo observar el comportamiento actual de la red sobre datos no vistos en la etapa de entrenamiento, para determinar si la red ya aprendió lo suficiente o es necesario modificarla. Generalmente estas dos etapas se suelen intercalar con el fin de seguir de cerca el proceso de entrenamiento hasta detenerlo.

Finalmente, se cuenta con la etapa de evaluación, donde se pone a prueba la red ante un nuevo conjunto de datos, seleccionado únicamente para evaluar, con el fin de estimar cómo respondería la red ante datos desconocidos.

### 2.5.3. Aprendizaje en redes neuronales

Es importante comprender cómo funciona el aprendizaje en redes neuronales. En este caso, lo que se intenta aprender a partir de los datos de entrenamiento son las matrices de pesos $W^{(j)}$ y $b^{(j)}$ de las diferentes capas.

Utilizamos la siguiente notación:

- $K$ es el número de neuronas en la capa de salida.
- $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ es un conjunto de entrenamiento, de tamaño $m$, donde para cada par $(x, y) = (x^{(i)}, y^{(i)})$, $x \in \mathbb{R}^n$ es un ejemplo de entrada e $y \in \mathbb{R}^K$ es la salida esperada para esa entrada (o etiqueta).
- Para cada $i$, $1 \leq i \leq m$, $a^{(i)}$ es la salida generada por la red para la entrada $x^{(i)}$.

El conjunto de entrenamiento puede ser tomado en cierto orden aleatorio, o en lotes, y además se puede utilizar varias veces para el proceso de aprendizaje.

Para aprender los pesos de las redes neuronales, se aplica descenso por gradiente [37] para minimizar una función de costo.

Un ejemplo de función de costo usada para redes neuronales es una generalización de la función del error cuadrático, pero sumando en todas las unidades de la capa de salida:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} (y^{(i)} - a_k^{(i)})^2 \ . \tag{2.2}$$

### 2.5.4. Forward propagation (Propagación hacia adelante)

El proceso de calcular los valores de salida de cada capa, y utilizarlo como entrada para la siguiente, hasta obtener el valor final de $a^{(L)}$, para una red de L capas, es conocido como *forward propagation*.

### 2.5.5. Algoritmo de backpropagation (Propagación hacia atrás)

El *algoritmo de backpropagation* se utiliza para buscar el mínimo de la función de costo en redes neuronales, calculando sus derivadas parciales respecto a cada parámetro de la red. El objetivo es actualizar cada peso en la red para que la salida real esté más cerca de la salida

objetivo, minimizando así el error para cada neurona de salida y la red como un todo. La base matemática de este algoritmo es la técnica del descenso por gradiente, basada en modificar los pesos en la dirección opuesta al gradiente, esto es $-\frac{\partial J}{\partial w}$, en la dirección que determina el decremento más rápido de la función de costo $J$. A medida que alcance un óptimo el gradiente se aproximará a cero. Una vez que llegue a cero, el valor de los pesos dejará de actualizarse. Si bien esto es generalmente útil ya que da como resultado la convergencia y significa que podemos detener el proceso iterativo, puede plantear un problema cuando el descenso del gradiente encuentra un óptimo local.

Presentamos a continuación la versión que utiliza descenso por gradiente incremental para la función de costo (2.2).

Entradas:

- Una red neuronal con entradas de tamaño $n$ y salida de tamaño $K$, $L$ capas y con función de activación $g$.
- Conjunto de entrenamiento $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$. Para cada ejemplo $(x, y) = (x^{(i)}, y^{(i)})$, $x \in \mathbb{R}^n$ es un ejemplo de entrada e $y \in \mathbb{R}^K$ es la salida esperada para esa entrada (o etiqueta).
- Una tasa de aprendizaje, $\alpha$, que suele tener un valor muy pequeño como 0.01 o 0.001. La tasa de aprendizaje tiene un papel crucial en este algoritmo, ya que controla el tamaño de los cambios de los pesos en cada iteración. A mayor tasa de aprendizaje mayor es la modificación de los pesos en cada iteración del algoritmo, con lo que el aprendizaje será en general más rápido. Aunque, por otro lado, quizá no se logre converger a la solución ya que puede saltarse el mínimo buscado y nunca encontrarlo. Abordaremos este tema en la sección 2.5.7.

1. Inicializar los pesos y los sesgos de la red con valores aleatorios pequeños (e.g. entre -.05 y .05)
2. Mientras no se cumpla la condición de fin
   - 2.1 tomar un nuevo ejemplo $(x, y) = (x^{(i)}, y^{(i)})$ del conjunto de entrenamiento.
   - 2.2 $a^{(1)} \coloneqq x$
   - 2.3 Para cada $l = 2, 3, \ldots, L$ calcular $z^{(l)} = W^{(l-1)}a^{(l-1)} + b^{(l-1)}$ ; $a^{(l)} = g(z^{(l)})$

2.4 Calcular $\delta^{(L)} = \nabla J \bigodot g'(z^{(L)})$ donde $\nabla J$ denota un vector cuyas componentes son las derivadas parciales $\partial J/\partial a^{(L)}$, $g'$ denota la derivada primera de $g$ respecto a $z^{(L)}$ y $\bigodot$ denota la multiplicación elemento a elemento de dos vectores. Para el caso particular en que la función de costo J sea la función del error cuadrático(2.2), se resume en $\delta^{(L)} = (a^{(L)} - y) \bigodot g'(z^{(L)})$.

2.5 Propagar el error hacia atrás: para cada $l = L-1, L-2, \ldots, 2$ calcular $\delta^{(l)} = (W^{(l)})^T \delta^{(l+1)} \bigodot g'(z^{(l)})$.

2.6 Actualizar los pesos de las capas $l = L, L-1, L-2, \ldots, 2$:

$$W^{(l)} = W^{(l)} - \alpha \delta^{l+1} \cdot (a^l)^T \ ,$$
$$b^{(l)} = b^{(l)} - \alpha \delta^l \ .$$

Algunas observaciones a tener presentes

- El orden para calcular los pesos es desde la última capa hacia atrás, hasta llegar a la segunda capa (la primera capa es la capa de entrada, y por lo tanto no tiene error de predicción).
- Algunas posibles condiciones de finalización:
  - Número de iteraciones.
  - Precisión sobre el conjunto de validación.
  - Error en en el conjunto de entrenamiento.

Como podemos observar el algoritmo de backpropagation se ejecuta para cada uno de los vectores del conjunto de entrenamiento, esto da como resultado un entrenamiento muy preciso pero lento. Es por ello que surge la idea de *lotes* (*batches*). Estas estructuras simplemente son bloques de vectores de entrada. Lo que implica que el aprendizaje se acelere, ya que los parámetros se actualizan en función del lote actual y no por vector de entrada. Esto se realiza sobre todos los lotes hasta que se haya visto todo el conjunto de entrenamiento.

Por otro lado, se tienen los *ciclos* (*epochs*) que al contrario que los lotes no son estructuras sino aglomeraciones de iteraciones. Se denomina ciclo al hecho de haber ejecutado en la red todos los vectores de entrada del conjunto de entrenamiento. Esto abre la posibilidad de elegir el número de veces que se desea entrenar la red con la misma base de datos, lo cual implica que en cada ciclo los vectores de entrada se tomen en orden aleatorio.

### 2.5.6. Funciones de activación

La función de activación modifica el valor resultado de cada neurona para pasar a la siguiente. Por lo cual también es conocido como filtro, función limitadora o umbral. Existen distintos tipos y, según el problema a tratar, puede ser mejor usar una u otra. Presentamos las más comunes.

#### Sigmoide (Logística)

La función de activación sigmoide, también denominada función logística, es la función utilizada por ejemplo en el algoritmo de clasificación de regresión logística [54], con el fin de predecir el resultado de una variable categórica (una variable que adopta un valor dentro de un conjunto finito de categorías).

La función toma un valor real y lo transforma a una escala (0, 1). Cuanto mayor sea la entrada, más cercano estará el valor de salida a 1, mientras que cuanto menor sea la entrada, más cerca estará la salida de 0. La definición de la función sigmoide es $f(x) = \frac{1}{1+e^{-x}}$, y se ilustra en la figura 2.4.

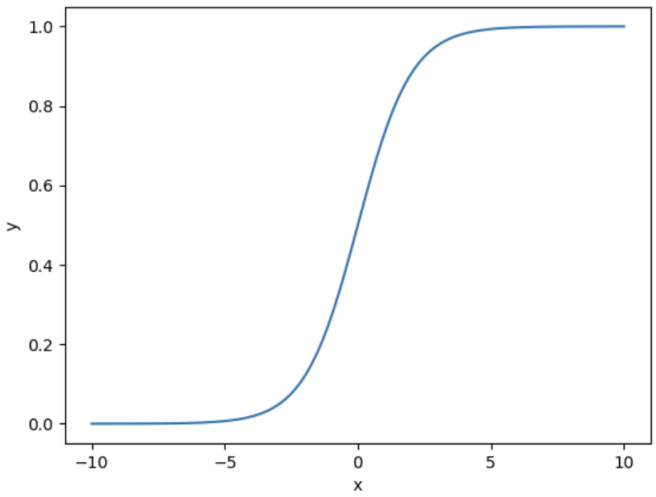



Figura 2.4: Función de activación sigmoide.

#### Tangente hiperbólica (tanh)

La función de activación de la tangente hiperbólica también se denomina simplemente función tanh. Es parecida a la función de activación sigmoide con algunas diferencias. Toma

un valor real y lo transforma a una escala (-1, 1). La figura 2.5 muestra la función tanh, que es calculada como $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

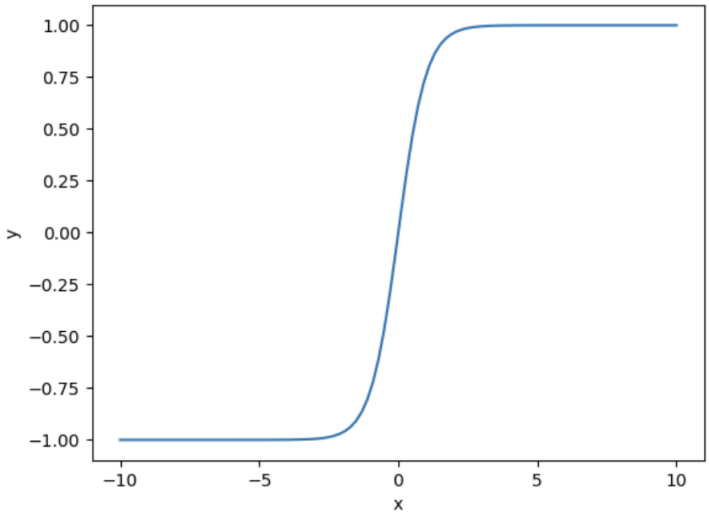



Figura 2.5: Función de activación tanh.

### Unidad Lineal Rectificada (ReLU)

ReLU [31] es la más popular, especialmente en redes muy grandes, porque permite aprender más rápido y es simple de implementar. Devuelve cero para entradas menores o iguales a cero, y la misma entrada si es positiva, por lo cual su rango es $(0, \infty)$. La figura 2.6 muestra la función función ReLU, $f(x) = \max(0, x)$.

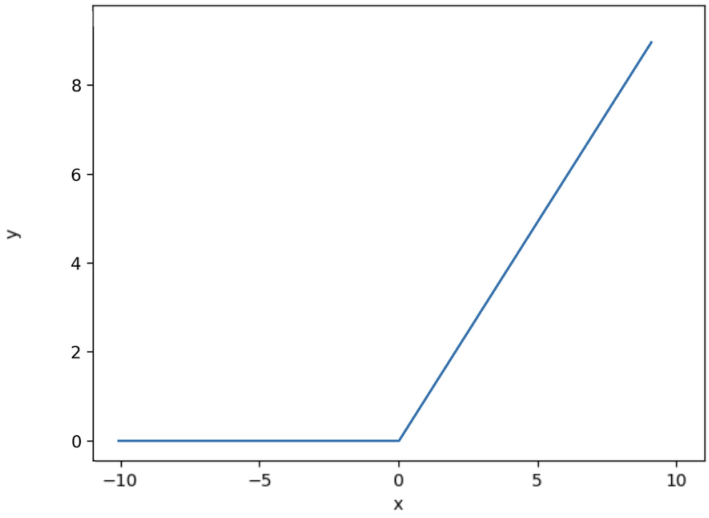



Figura 2.6: Función de activación ReLU.

### Swish

La función de activación Swish[36] se define como $f(x) = x \cdot \sigma(\beta x)$, donde $\sigma(z) = (1 + e^{-z})^{-1}$ es la función sigmoide y $\beta$ es un parámetro. La figura 2.7 muestra ejemplos para algunos valores de $\beta$.

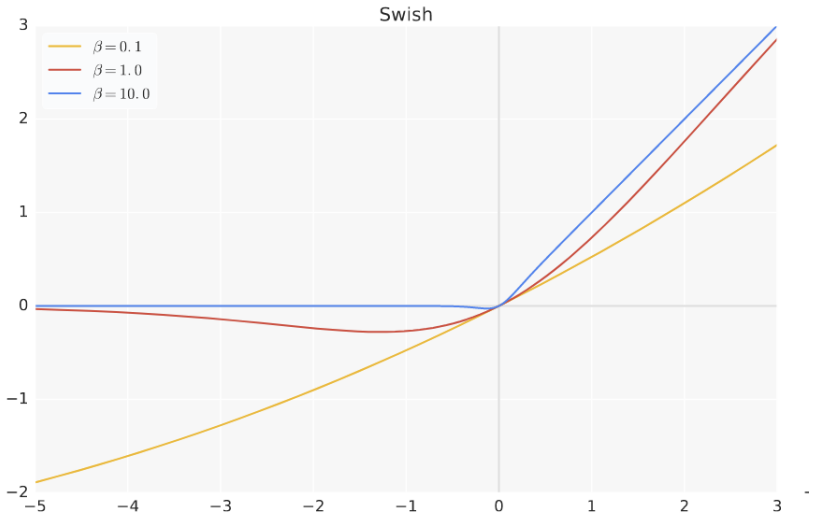



Figura 2.7: Función de activación Swish[36].

### 2.5.7. Hiperparámetros

En el proceso de construcción de una red neuronal, existen parámetros que son ajustables, tales como la cantidad de capas intermedias o la cantidad de veces que el algoritmo entrenará sobre el conjunto de datos de entrenamiento. Dichos parámetros se denominan *hiperparámetros* y su elección puede tener gran impacto en el rendimiento del modelo. Por esta razón es recomendable explorar distintas opciones para elegir lo que mejor se adapta a cada caso. Los hiperparámetros más destacables, presentados a continuación, son la cantidad de *ciclos* (*epochs)*, el *tamaño del lote* (*batch size*) y la *tasa de aprendizaje* (*learning rate*).

Como mencionamos anteriormente en la sección 2.5.5, se conoce como ciclos a la cantidad de veces que el algoritmo entrena sobre el conjunto de datos de entrenamiento. Se puede entrenar al modelo con una cantidad fija de ciclos o una cantidad variable. Para esto último se suele utilizar una técnica llamada *Early Stopping*, donde se detiene el entrenamiento si cierta métrica monitoreada no mejora luego de una determinada cantidad de ciclos.

El tamaño de lote representa la cantidad de ejemplos que son procesados entre ajustes de los parámetros del modelo. En cada ciclo se suele dividir el conjunto de datos en subconjuntos de este tamaño y se dan tantos pasos como subconjuntos haya. Existen tres formas de dividir los datos: *batch gradient descent*, donde el tamaño de lote es igual al tamaño del conjunto de entrenamiento, *stochastic gradient descent*, donde el tamaño de lote es 1, seleccionando de manera aleatoria cada ejemplo que se procesa en lugar de tomar todo el conjunto completo, y *mini-batch gradient descent*, la opción intermedia de las dos anteriores y también la más ampliamente utilizada, donde el tamaño es un valor entre 1 y el tamaño del conjunto de entrenamiento. El tamaño de lote puede afectar tanto a los resultados de la red como también al tiempo de entrenamiento.

Finalmente, la tasa de aprendizaje esta fuertemente relacionada al tamaño de lote. Como mencionamos en la sección 2.5.5, la tasa de aprendizaje conceptualmente corresponde al tamaño de las actualizaciones de los pesos que se da en cada iteración del algoritmo de descenso por gradiente. Se deben evitar dos extremos, si el valor es muy pequeño puede ocasionar una disminución importante en la velocidad de convergencia y la posibilidad de acabar atrapado en un mínimo local; en cambio, un ritmo de aprendizaje demasiado grande puede conducir a inestabilidades en la función de error, lo cual evitará que se produzca

la convergencia debido a que se darán saltos en torno al mínimo global sin alcanzarlo. Es fundamental explorar con distintos valores y buscar el que mejor se adapte al caso de uso en cuestión.

### 2.5.8. Normalización en lotes

Entrenar redes neuronales puede tornarse complejo. Por este motivo en el área se ha invertido un gran esfuerzo en investigar posibles causas de problemas típicos como por ejemplo problemas de convergencia. Algunos de estos problemas son el *overfitting* (*sobreajuste*), cuando el modelo se ajusta demasiado a los datos de entrenamiento y no generaliza bien con los datos de evaluación, y el *underfitting* (*subajuste*), cuando el modelo no alcanza un buen desempeño ni con los datos de entrenamiento ni con los datos evaluación. En este proceso se han desarrollado algunas técnicas relevantes que pueden ser de utilidad para facilitar el entrenamiento. Nos centramos en una de ellas, *normalización en lotes* [17], utilizada en los algoritmos presentados en el capítulo 4.

Suponiendo que los datos de entrada de una red son realizaciones de cierto modelo probabilístico, la distribución de los datos de entrada de cada capa de una red varía a medida que varían los parámetros de las capas anteriores (debido a la técnica de backpropagation). Debido a esto, a medida que la red se hace más profunda, pequeños cambios en dichas capas terminan generando cambios cada vez más grandes en la distribución de un capa en particular. El hecho de que continuamente se produzcan estos cambios hace que cada capa tenga que adaptarse todo el tiempo a nuevas distribuciones, lo que ralentiza el entrenamiento y lleva al uso de tasas de aprendizaje bajas. A este problema se lo conoce como *internal covariate shift*. Una de las técnicas más utilizadas para atacarlo es la normalización en lotes, la cual implica aplicar modificaciones a los datos de entrada de cada capa para así trabajar con datos más homogéneos que permitan un entrenamiento más eficaz y estable, con la ventaja de poder utilizar una tasa de aprendizaje más alta. Dada una capa con entrada $x = x_1, \ldots, x_d$, la normalización por lotes consiste en tomar para cada componente $x_k$ con $k = 1, \ldots, d$ calcular la media $\mu_k$ y la varianza $\sigma_k^2$ sobre el lote y aplicar la transformación:

$$\hat{x_k} = \frac{x_k - \mu_k}{\sqrt{\sigma_k^2}}$$

## 2.6. Arquitecturas de redes neuronales

A partir de la estructura básica de una red neuronal, existen distintas arquitecturas, cada una con distintas estructuras y requerimientos, que pueden desempeñar mejor ciertas tareas.

### 2.6.1. Redes convolucionales

La figura 2.8 muestra una *red neuronal convolucional* (Convolutional Neural Networks, CNN). Típicamente consiste en una secuencia de capas al igual que las redes neuronales regulares, pero en estas redes para la construcción de su arquitectura se usan tres tipos de capas principales: *capa convolucional*, *capa de pooling* y *capa de activación* [22].

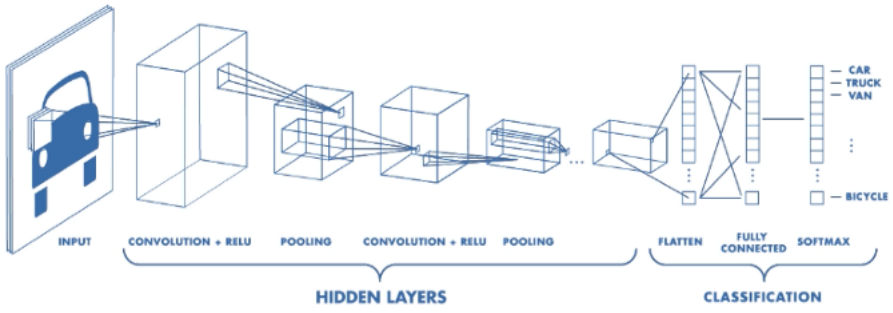



Figura 2.8: Representación de una CNN [39].

La primera capa aplica convoluciones discretas. Una *convolución discreta* es una operación matemática entre dos vectores que devuelve otro vector. En el contexto de procesamiento de imágenes, estos son matrices. La figura 2.9 muestra un ejemplo del proceso de convolución, que consiste en realizar la multiplicación elemento a elemento entre un *filtro* o *kernel* (matriz fija) y una sección del mismo tamaño en la entrada, obteniendo una nueva matriz donde cada elemento se obtiene sumando todos estos resultados para obtener un único valor.

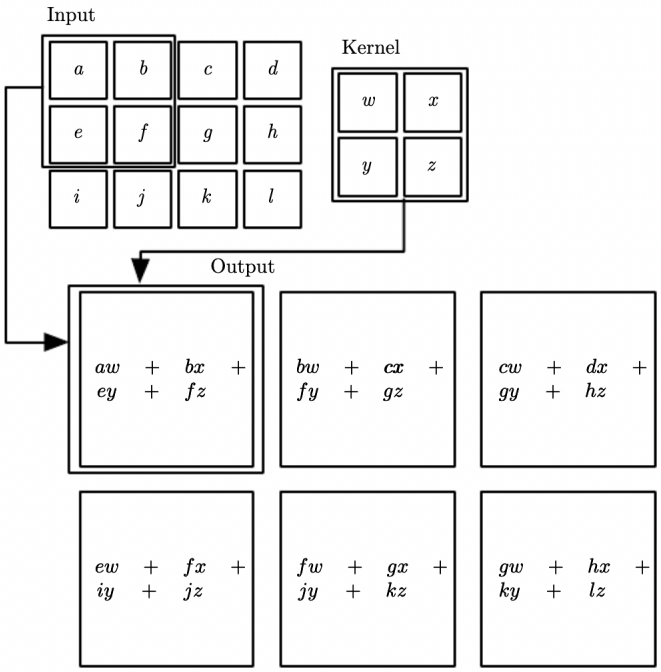



Figura 2.9: Ejemplo del proceso de convolución en 2D [10].

La capa de *pooling* reduce la dimensionalidad tomando cierta estadística de los valores vecinos, por ejemplo el máximo de cada rectángulo de $2 \times 2$ alrededor de un elemento de entrada. Este tipo de operación es útil para evitar que la red aprenda a asociar la presencia de una determinada característica con una ubicación específica en la imagen de entrada, lo que ayuda a generalizar mejor.

La última capa mencionada puede aplicar cualquiera de las activaciones conocidas, algunas de ellas mencionadas en el la sección 2.5.6 (Sigmoide, tanh, ReLU).

### 2.6.2. Redes recurrentes

Estas redes son de utilidad para problemas donde el tamaño de la entrada y/o salida es variable. Una *red neuronal recurrente* (Recurrent Neural Networks, RNN) [43] puede considerarse como varias copias de la misma red, cada una pasando un mensaje a un sucesor como se ve en la figura 2.10.

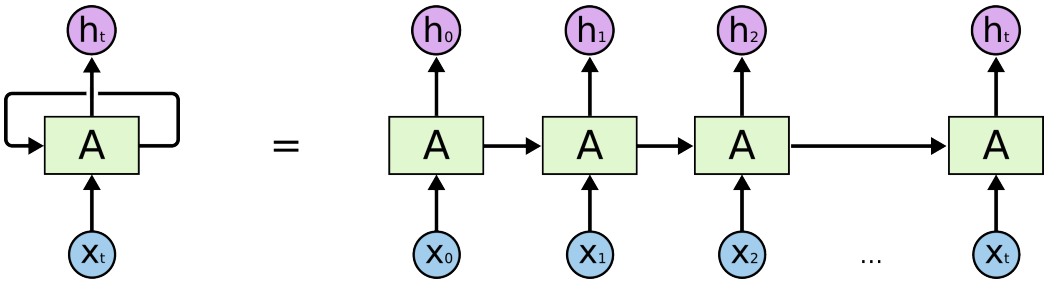



Figura 2.10: RNN desenrollada [33].

En este tipo de redes existe el concepto de *contexto*, que es información pasada o futura sobre la entrada de la red. Por ejemplo, si se está traduciendo un texto palabra por palabra, el contexto puede ser palabras anteriores a la palabra que se quiere traducir. Las redes RNN son buenas prediciendo a partir de un contexto cercano para la tarea que se está desarrollando, pero en la práctica presentan problemas para capturar contexto demasiados lejanos (por ejemplo, palabras que se procesaron hace muchas iteraciones). Una mejora de esto son las redes *long short-term memory* (LSTM) [50], en las cuales los mensajes entre copias de la red se calculan mediante funciones más complejas (ver figura 2.11).

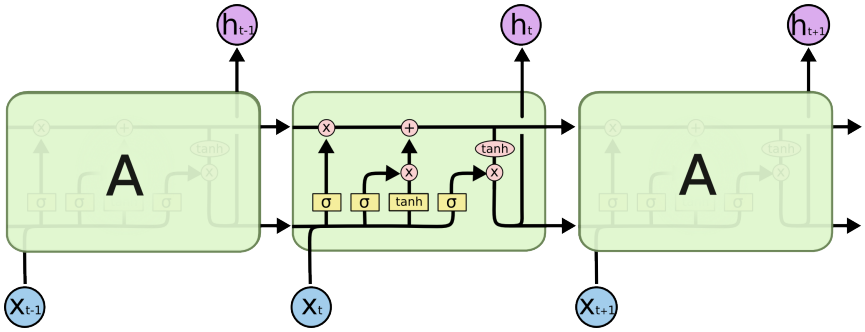



Figura 2.11: Cadena de una LSTM, $\sigma$ representa una función sigmoide [33].

### 2.6.3. Autoencoders

Los *autoencoders* son un tipo de redes neuronales, pero en este caso la red se entrena para que aprenda a generar a la salida el mismo dato de entrada. Para ello consta de un codificador (*encoder*), que transforma la entrada en una representación de menor dimensión (*código latente*), extrayendo las características que permitan su reconstrucción en el decodificador (*decoder*). Generalmente, las capas ocultas tienen menos nodos que las de entrada y salida, de tal forma que se consigue una representación de menor dimensión de los datos de entrada,

lo que también funciona como compresión de la entrada. Podemos observar estos componentes en la figura 2.12.

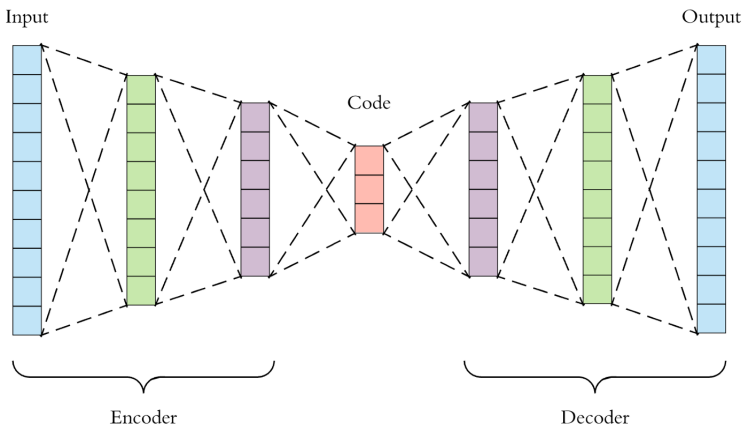



Figura 2.12: Representación de un autoencoder [40].

Los autoencoders se entrenan con las mismas técnicas que una red neuronal tradicional, mediante backpropagation. De forma general, existen tres hiperparámetros a tener en cuenta a la hora de diseñar un autoencoder:

- Número de capas: Las capas ocultas pueden ser tantas como se quiera.
- Número de neuronas por capa: Habitualmente, el número de neuronas va disminuyendo o se mantiene igual conforme se avanza en las capas del codificador, y luego se restablecen en el mismo orden para el decodificador. Para el caso de las neuronas en el código latente cuanto menor tamaño mayor compresión.
- Función de pérdida: Una de las más usadas es el error cuadrático medio. Pero esto depende de la arquitectura de cada autoencoder.

En los autoencoders, el hecho de que el código latente retenga datos relevantes sobre la entrada hace que sean de gran utilidad para la reducción de ruido en imágenes o la clasificación a partir de dicha estructura obtenida. Otra ventaja que se les atribuye, es que son métodos no supervisados, es decir, no requieren que se le especifiquen etiquetas para ser entrenados, sólo se les pasan los datos de entrada.

### 2.6.4. Variational autoencoders (VAE)

Los *variational autoencoders* tienen una propiedad única que los diferencia de los autoencoders estándar y es que el espacio del código latente es continuo, lo que permite una fácil interpolación y muestreo aleatorio. Para lograr esto hace que su codificador no devuelve solo un vector del espacio latente de tamaño $n$ como antes, sino los parámetros de una distribución, por ejemplo, en el caso de trabajar con una distribución gaussiana como es usual, el codificador devuelve dos vectores de tamaño n: un vector de medias ($\mu$) y otro vector de desviaciones estándar ($\sigma$), formando un vector de $n$ variables aleatorias, siendo el $i$-ésimo elemento de $\mu$ y $\sigma$ la media y la desviación estándar de la $i$-ésima variable aleatoria ($X_i$) con distribución gaussiana. Luego, se obtiene el vector del espacio latente a partir del muestreo de estas variables aleatorias como se ilustra en la figura 2.13.

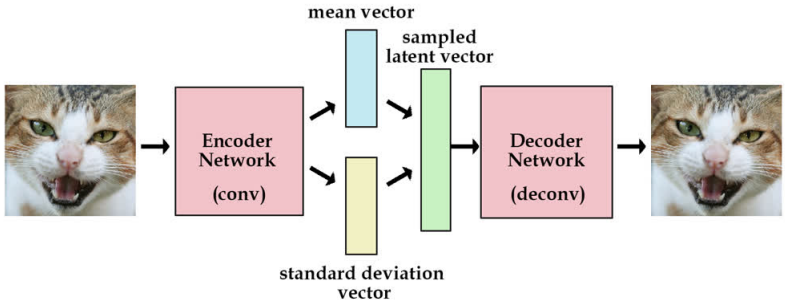



Figura 2.13: Arquitectura a alto nivel de una VAE [30].

Intuitivamente, el vector de medias controla dónde debe centrarse la codificación de una entrada, mientras que la desviación estándar controla el "área" o cuánto puede variar la codificación con respecto a la media.

Idealmente, lo que queremos son codificaciones en el espacio latente que estén lo más cerca posible entre sí sin dejar de ser distintas, lo que permite una interpolación "suave". Para forzar esto, se usa la divergencia de Kullback-Leibler (divergencia KL [20]) en la función de pérdida, que mide cuánto divergen entre sí dos distribuciones de probabilidad. De esta forma se busca optimizar los parámetros de distribución de probabilidad ($\mu$ y $\sigma$) para parecerse a cierta distribución objetivo. La figura 2.14 muestra cada una de las entidades involucradas en la fórmula final para el entrenamiento del codificador y el decodificador.

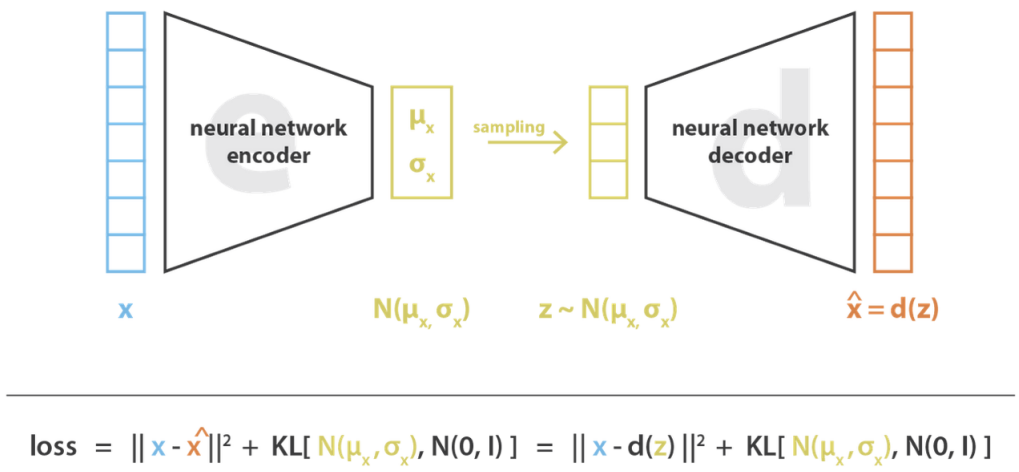



Figura 2.14: Función de pérdida para una VAE [30].

### 2.6.5. Flow-based Models

Los *flow-based models*[53] se construyen mediante una secuencia de transformaciones invertibles denominadas *normalizing-flows* (o simplemente *flows*). La idea principal es llevar la entrada, una variable aleatoria $X$ con distribución desconocida, a una distribución $Z$ con distribución conocida mediante una transformación $f$ invertible, para poder reconstruir nuevamente las muestras $x$ de $X$ como se ilustra en la figura 2.15. Este tipo de arquitecturas tiene aplicación en la compresión gracias a que al saber como se modelan los datos luego de aplicar esta transformación se pueden usar los codificadores usuales ya que se conoce las probabilidades de los datos, por ejemplo, en el caso de los píxeles de una imagen si se transforma a datos con una distribución gaussiana con ciertos parámetros se podría codificar con un codificador aritmético.

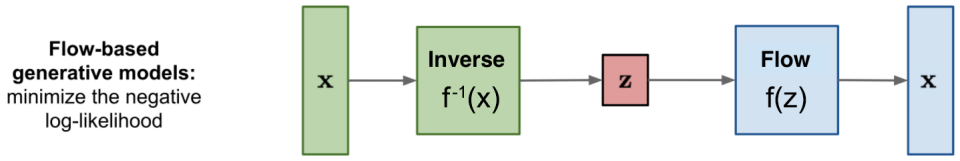



Figura 2.15: Representación de alto nivel de un Flow-based Generative Model [52].

Para lograr la transformación $f$ que potencialmente puede ser compleja, como mostramos en la figura 2.16, se explota el hecho de que las funciones invertibles y diferenciables son

cerradas bajo la composición, por lo que la transformación $f$ puede construirse a partir de $k$ flows individuales:

$$f = f_k \circ \cdots \circ f_1 \ .$$

En la figura 2.16 se observa en detalle lo que corresponde a la parte derecha de la figura 2.15 para la parte izquierda se tendría algo similar aplicando la inversa de cada función comenzando con $x$.

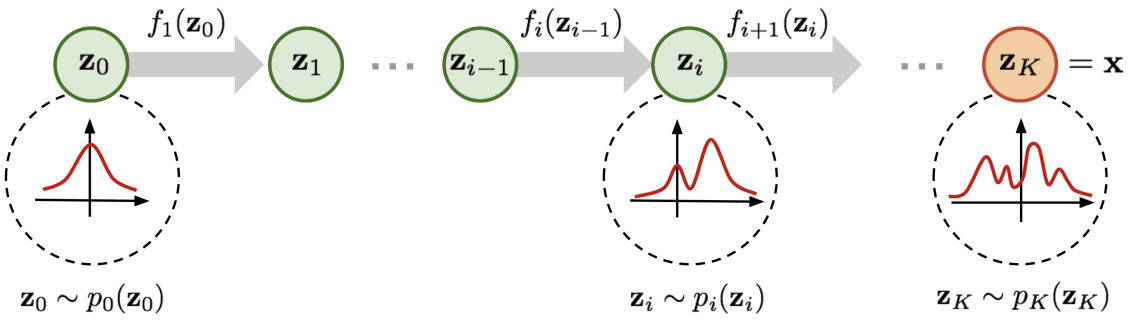



Figura 2.16: Ilustración de un normalizing flow, transformando una distribución simple $p_0(z_0)$ en una compleja $p_k(z_k)$ paso a paso. [52].

Para entender la función de pérdida usada en este tipo de modelos presentamos dos conceptos clave. El primero es la *matriz jacobiana*: dada una función $f : \mathbb{R}^n \to \mathbb{R}^m$ que mapea un vector de entrada $x$ $n-$dimensional a un vector de salida $m-$dimensional, la matriz jacobiana es la matriz de todas las derivadas parciales de primer orden, donde la entrada en la fila i-ésima y la columna j-ésima es $Df(x)_{ij} = \dfrac{\partial f_i}{\partial x_j}$,

$$Df(x) = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix} \ .$$

El otro concepto importante es el cambio de variable: dadas dos variables aleatoria $Z$ y $X$ con densidad de probabilidad $p(x)$ y $p(z)$ respectivamente, que están relacionadas por un mapeo $f : \mathbb{R}^n \to \mathbb{R}^m$ tal que $X = f(Z)$ y $Z = f^{-1}(X)$ se cumple:

$$p_X(x) = p_Z(f(x)) \left|\det Df(x)\right| \ . \tag{2.3}$$

Finalmente, la función de pérdida usada comúnmente para el entrenamiento de estos modelos es la de máxima verosimilitud (logarítmica)

$$\sum_{i=1}^{n} (\ \log p_Z(f(x_i|\theta)) + \log|\det Df(x_i|\theta)|\ )\ , \tag{2.4}$$

donde $\theta$ son los parámetros a ajustar para el flow $f(x|\theta)$.

Dado que el algoritmo final con el que experimentamos en los capítulos 6 y 7 hace uso de flow-based models, definimos a continuación los flows mas usados; en general la función $f$ es una combinación de estos flows.

### Linear flows

Esta transformación es una transformación lineal que dada una matriz $A \in \mathbb{R}_{n\times n}$ y un vector $b \in \mathbb{R}^n$ se calcula como:

$$f(x) = Ax + b\ .$$

Si bien estas transformaciones son fáciles de comprender, tienen el problema de que son cerradas bajo la composición, por lo que una composición de estos flows no puede expresar nada más complejo que una función lineal, y además el costo de calcular el determinante o la inversa puede ser $O(n^3)$. Para matrices particulares, como diagonales o triangulares, el costo es menor, pero el problema de ser cerrados bajo la composición se mantiene.

### Coupling flow

Es un enfoque para construir transformaciones no lineales. En este caso se divide la entrada $x \in \mathbb{R}^n$ en dos partes disjuntas $x = (x^A, x^B)$, la primera parte se deja como está y a la segunda se le aplica otro flow con parámetros dependientes de $x^A$,

$$f(x) = (x^A, \hat{f}(x^B|\theta(x^A)))\ .$$

La figura 2.17 ilustra esta operación.

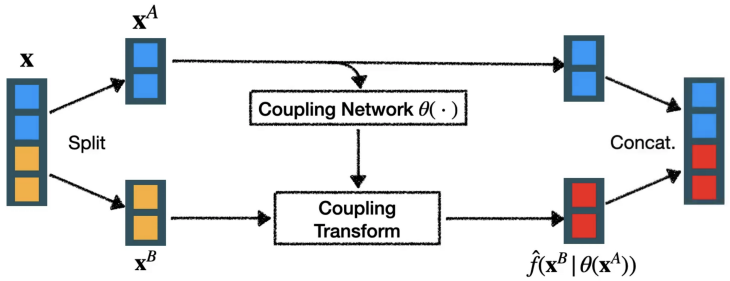



Figura 2.17: Coupling transform [52].

Para la inversa simplemente se aplican las operaciones en dirección contraria. En el caso de este tipo de flows el determinante se reduce a este cálculo det $D\hat{f}(x^B|\theta(x^A))$.

El calculo de $\theta(x^A)$ puede ser complejo, como por ejemplo una red CNN, RNN, etc; ya que no se necesita su inversa ni el jacobiano una vez computado. También se suelen hacer permutaciones de $x^A$ y $x^B$ para mejorar el entrenamiento.

Luego, para esa transformación intermedia $\hat{f}$, se tienen varias opciones. Dados $s$ y $t$ dos parámetros de la misma dimensión que $x^B$ obtenidos como salida de la red $\theta(\cdot)$ tomando como entrada $x^A$, se aplica una transformación sobre $x^B$, a partir de $s$ y $t$ expresada como $f(x^B|\theta(x^A))$ en la figura 2.17. Los primeros trabajos con *flows* proponen esta transformación como una adición $\hat{f}(x|t) = x + t$ pero quedan muy limitados aunque sean simples de aplicar. Una transformación más comúnmente usada es una transformación afín (usada en el modelo NICE [7]), $\hat{f}(x|s,t) = s \odot x + t$, donde recordamos que $\odot$ denota la multiplicación elemento a elemento de dos vectores.

# Capítulo 3

# Algoritmos de compresión clásicos

Este capítulo describe los algoritmos de compresión sin pérdida que no usan redes neuronales. Son muy usados actualmente debido a que su implementación se encuentra disponible en muchos lenguajes, son eficientes, y logran buenas tasas de compresión. Aquí se introducen tres algoritmos que luego usaremos para tomar como referencia para nuestros experimentos detallados en el capítulo 7, donde evaluamos la tasa de compresión lograda con redes neuronales.

## 3.1. LOCO-I/JPEG-LS

En LOCO-I (low complexity lossless compression for images)[42][51] tanto el codificador como el decodificador funcionan de forma sincronizada, recorriendo los píxeles de la imagen de izquierda a derecha y de arriba hacia abajo; este recorrido suele reconocerse bajo el nombre *scan-line*. Para la codificación de cada píxel se utiliza un modelo que toma en cuenta los píxeles vecinos más cercanos que ya fueron codificados anteriormente.

La codificación de cada píxel se realiza en uno de los dos modos posibles: *modo regular* o *modo carrera*. En modo regular consta de los siguientes pasos:

- Determinación de un contexto en el que ocurre un píxel, que es una función de los valores de sus píxeles vecinos.
- Predicción del valor del píxel en función de sus vecinos y estadísticas calculadas sobre píxeles ya codificados.

- Codificación del error de predicción.

La figura 3.1 muestra el modelo y los principales componentes del estándar LOCO-I. A continuación detallamos cada uno de estos pasos que componen el modo regular. El modo carrera se describe posteriormente.

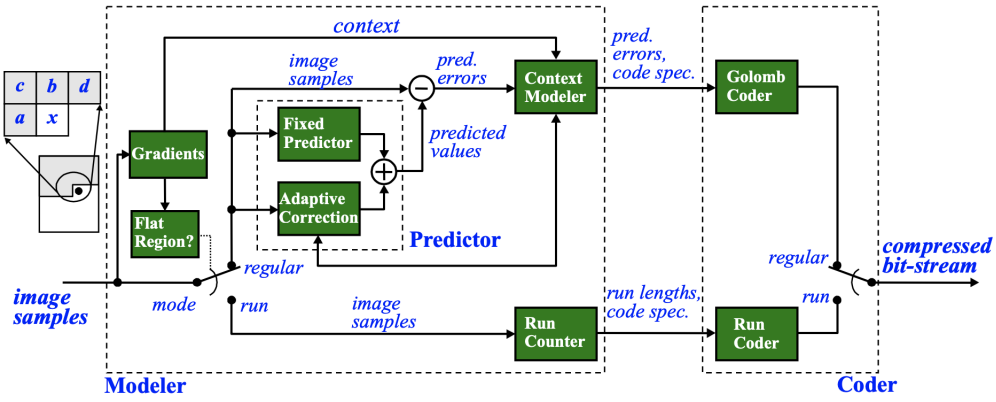



Figura 3.1: JPEG-LS (LOCO-I Algorithm): Block Diagram.

### 3.1.1. Predicción

El bloque de predicción del píxel utiliza un predictor fijo que se basa en tres píxeles procesados anteriormente, aplicando un algoritmo de detección de bordes horizontales o verticales sobre dicho contexto fijo de píxeles. Por otro lado la parte adaptativa suma un número entero, que es una corrección de sesgo dependiente del contexto.

#### Predictor fijo

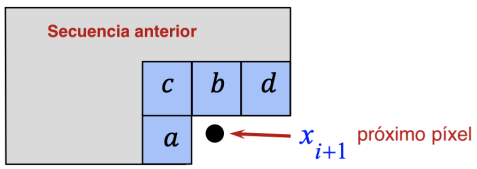



Figura 3.2: Contexto de un píxel.

Según se ilustra en la figura 3.2, la predicción de un píxel x se basa en el análisis de los píxeles adyacentes a, b y c.

El predictor combina tres predicciones simples:

$$\widetilde{x}_{i+1} = \begin{cases} \min(a,b) & \text{si } c \geq \max(a,b), \\ \max(a,b) & \text{si } c \leq \min(a,b), \\ a+b-c & \text{en otro caso.} \end{cases}$$

Los primeros dos casos corresponden a detección de bordes vertical y horizontal respectivamente y el tercero un caso donde no se detecta un borde. La figura 3.3 presenta un ejemplo de cada caso.

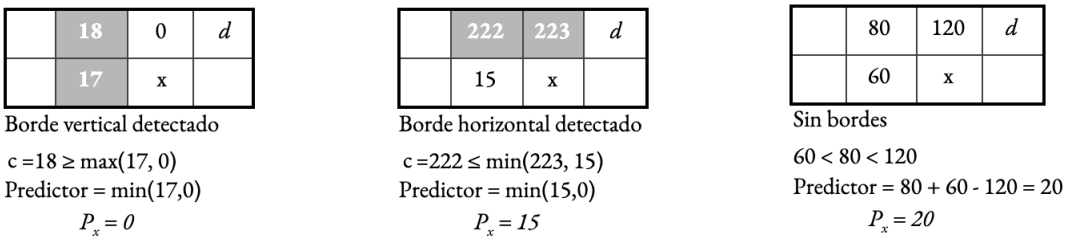



Figura 3.3: Ejemplos de aplicación del predictor fijo.

**Corrección adaptativa**

A la predicción inicial, $\widetilde{x}_{i+1}$, se le suma una corrección de sesgo, $\beta$, para obtener la predicción definitiva $\widehat{x}_{i+1}$. El valor de $\beta$ se determina en función de los errores de predicción cometidos anteriormente en píxeles que ocurrieron en el mismo contexto. Este contexto se determina en función de los píxeles a, b, c y d, como se explica en la sección 3.1.2.

### 3.1.2. Determinación del contexto

El contexto, que determina el código utilizado para el error de predicción y la corrección de sesgo de dicha predicción se construye a partir de las siguientes diferencias:

$$g_1 = d - b, \quad g_2 = b - c, \quad g_3 = c - a$$

Estas diferencias permiten diferenciar diferentes niveles de suavidad o rugosidad en la vecindad de un píxel.

Para reducir el tamaño del modelo, cada diferencia $g_i, i = 1, 2, 3$, se cuantifica en un pequeño número de regiones.

### 3.1.3. Codificación del error de predicción

Una vez calculada la predicción, se determina el error de predicción, $\epsilon$. Este valor es la resta entre el valor real del píxel y su predicción. Luego es codificado utilizando un código de Golomb. Dado un entero positivo $n$, el método de codificación Golomb de orden $m$ ($G_m$), codifica al entero en dos partes:

- Una representación binaria de $m \bmod n$, usando $\lfloor \log m \rfloor$ bits si $n < 2^{\lceil \log m \rceil} - m$, y $\lceil \log m \rceil$ bits en otro caso.
- Una representación unaria de $n/m$, entendiendo representación unaria de un entero $q$ como $unary(q) = \overbrace{00\ldots0}^{q}1$.

Los códigos de Golomb son óptimos para codificar valores no negativos que siguen una distribución geométrica. En estos casos, para cada distribución de este tipo existe un valor de $m$ que genera un código de Golomb $G_m$ que tiene en promedio el largo más corto posible.

El residuo $\epsilon$ puede ser negativo, por lo cual es necesario realizar un mapeo, sin pérdida de información, para llevar estos enteros con signo a enteros no negativos que sigan aproximadamente una distribución geométrica. Para lograrlo, se utiliza la función

$$M(\epsilon) = \begin{cases} 2\epsilon, & \text{si } \epsilon \geq 0, \\ -2\epsilon - 1, & \text{si } \epsilon < 0. \end{cases}$$

Por otro lado, los códigos de Golomb tienen una particularidad cuando el valor de $m$ es potencia de 2 ($m = 2^k$). En estos casos el procedimiento de codificación/decodificación es más simple. El código para un valor $n$ consiste en los $k$ bits menos significativos de $n$ representado en binario, seguidos por el número formado por los restantes bits en representación unaria. Estos códigos son conocidos como Golomb-Rice.

Para estimar el parámetro $k$ del código de Golomb-Rice se define:

- A: Suma acumulada de las magnitudes de los errores de predicción en el contexto del píxel,
- N: Número de ocurrencias de dicho contexto,

y se calcula

$$k = \left\lceil \log \frac{A}{N} \right\rceil .$$

### 3.1.4. Modo carrera

Cuando las diferencias $g_1, g_2, g_3$ (mencionadas en la sección 3.1.2) son iguales a 0, el codificador entra en modo carrera y codifica el número de veces que se repite el mismo valor de píxel hasta que se llega a uno diferente o se alcance el final de la fila.

## 3.2. WebP-lossless

Este formato consiste en transformar la imagen combinando diferentes técnicas tomadas de cierto repertorio preestablecido. Se elige cuáles de estas técnicas son aplicadas dependiendo del aporte a la compresión, pudiendo aplicarse cada una de ellas una vez o ninguna. En algunas técnicas, la imagen es dividida en bloques (usualmente de $16 \times 16$) y cada bloque de la imagen usa los mismos parámetros de transformación. La imagen comprimida se construye codificando los parámetros de cada transformación y los datos de imagen transformados. Las transformaciones pueden ser:

- Predicción espacial de píxeles.
- Transformación del espacio de color con:
  - indexación de píxeles,
  - empaquetado de píxel,
  - substracción de verde.

### 3.2.1. Transformaciones

Todas las transformaciones son reversibles. En cada caso se utiliza un bit para indicar la presencia de una transformación, seguido de los parámetros y datos transformados.

#### Transformación de predicción

Se usa para explotar el hecho de que píxeles vecinos usualmente están correlacionados. De forma similar a LOCO-I, se predice un valor basado en píxeles ya codificados y se codifica el error de predicción. Se le llama *modo de predicción* a la combinación de píxeles usados para la predicción del valor actual a ser codificado.

Hay 14 modos de predicción distintos para un píxel a partir de sus vecinos, que se definen a partir de los vecinos TL (top-left), T (top), TR (top-right) y L (Left) de un píxel P

$$\begin{array}{cccccc} O & O & O & O & O & O \\ O & O & TL & T & TR & O \\ O & O & L & P & X & X \\ X & X & X & X & X & X \end{array}$$

Los 14 modos se presentan en el cuadro 3.1, en el cual utilizamos la siguiente notación

- $Average(a, b) = \frac{a+b}{2}$
- $Select(L, T, TL) = \begin{cases} L & si \quad p_L < p_T, \\ T & si \quad p_L \geq p_T, \end{cases}$
  con:
  - $p_L = ||p - L||_1$
  - $p_T = ||p - T||_1$
  - $p = L + T - TL$
- $ClampAddSubstractFull(a, b, c) = Clamp(a + b - c)$
- $ClampAddSubstractHalf(a, b) = Clamp(a + (a - b)/2)$

| Modo | Valor predicho |
|---|---|
| 0 | 0xFF000000 (Color negro opaco) |
| 1 | L |
| 2 | T |
| 3 | TR |
| 4 | TL |
| 5 | Average(Average(L, TR), T) |
| 6 | Average(L, TL) |
| 7 | Average(L, T) |
| 8 | Average(TL, T) |
| 9 | Average(T, TR) |
| 10 | Average(Average(L, TL), Average(T, TR)) |
| 11 | Select(L, T, TL) |
| 12 | ClampAddSubstractFull(L, T, TL) |
| 13 | ClampAddSubstractHalf(Average(L, T), TL) |

Cuadro 3.1: 14 modos de predicciones para un píxel a partir de sus vecinos.

Para los casos de borde se manejan las siguientes reglas

- El valor predicho del píxel de más arriba a la izquierda es 0xFF000000 (modo 0).
- Para los píxeles de la primera fila es L (modo 1).
- Para los píxeles de la primera columna es T (modo 2).
- Parar los píxeles de la última columna se usa cualquiera de los 14 modos, pero en lugar de TR se toma el primer píxel de la fila de P. Visto de otra forma, si P está en el borde o no, TR siempre es el píxel que en memoria está luego de T.

### Transformación de color

El objetivo es de-correlacionar los valores R, G y B de cada píxel para codificar cada componente por separado. El verde se mantiene igual, el rojo se transforma en función del verde y el azul en función de los otros dos. En este caso la imagen también se divide en bloques y se aplica el mismo modo de transformación para cada píxel en un bloque, que se define mediante tres parámetros, $\delta_1$, $\delta_2$, $\delta_3$, donde $\delta_i$ es un entero de 8 bits con signo que

representa un número de punto fijo con 3 bits para la parte entera y 5 bits para la parte fraccionaria.

Denotando $toDelta(\delta, b) = (\delta * b) >> 5$, siendo $>>$ la operación de desplazamiento de bits hacia la derecha; la transformación de color, $CF$, para un píxel con componentes R, G y B queda definida de la siguiente manera

$$CF(R, G, B) = (R + toDelta(\delta_1, G), G, B + toDelta(\delta_2, G) + toDelta(\delta_3, R)) \ .$$

**Substracción de verde**

Cuando se tiene esta transformación el codificador resta el canal verde tanto al rojo como al azul. No hay parámetros adicionales asociados a esta transformación. Para que el decodificador los recupere solo hace falta sumar el verde.

Esta transformación es un caso particular de transformación de color que no requiere códificar parámetros y por lo tanto puede resultar más económica que una transformación genérica.

**Transformación de indexación**

Esta transformación mapea píxeles a índices de una tabla. El proceso consiste en contar la cantidad de píxeles distintos en la imagen. Si la cantidad es menor a cierto umbral (256), se crea una tabla con esos valores y luego se reemplaza cada píxel por el índice correspondiente.

### 3.2.2. Datos de la imagen

Los datos de la imagen se representan por una matriz de píxeles ordenados en scan-line, al igual que en LOCO-I.

Estos datos se usan para 5 fines distintos:

- Imagen ARGB: píxeles de la imagen.
- Imagen de entropía: representación de un código de Huffman.
- Imagen de predicción: representación de una transformación de predicción.
- Imagen de transformación de color: representación de los parámetros de una transformación de color.

- Imagen de indexación de color: matriz usada para el mapeo como se describe en la sección 3.2.1.

### 3.2.3. Codificación de los datos de la imagen

Cada uno de los bloques de la imagen se almacena con su propia codificación entrópica, aunque varios bloques podrían llevar la misma codificación.

Ya que representar los parámetros de una codificación entrópica lleva un costo extra, este costo puede minimizarse si bloques similares usan la misma codificación. Parar esto el codificador puede agrupar bloques en clusters.

Cada píxel es codificado usando uno de tres métodos posibles:

- Codificación de Huffman.
- LZ77 backward reference.
- Color Cache Coding.

#### Codificación de Huffman

El píxel se almacena como valores codificados por Huffman para cada canal: verde, rojo, azul y alpha (en ese orden).

#### LZ77 backward reference

Se codifica utilizando el algoritmo Lempel-Ziv 1977 (LZ77) [55] mediante tuplas de longitud y código de distancia, donde el largo indica cuántos píxeles van a ser copiados siguiendo el orden de rasterizado y el código de distancia indica la posición de inicio desde donde los píxeles van a ser copiados.

#### Codificación de caché de color

El caché de color almacena un conjunto de colores que se han utilizado recientemente en la imagen. De esta manera, los colores usados recientemente pueden ser referidos de manera más eficiente que usando los dos métodos anteriores.

La figura 3.4 muestra la cache de color local que se va actualizando progresivamente con los 32 colores usados recientemente a medida que el escaneo avanza.

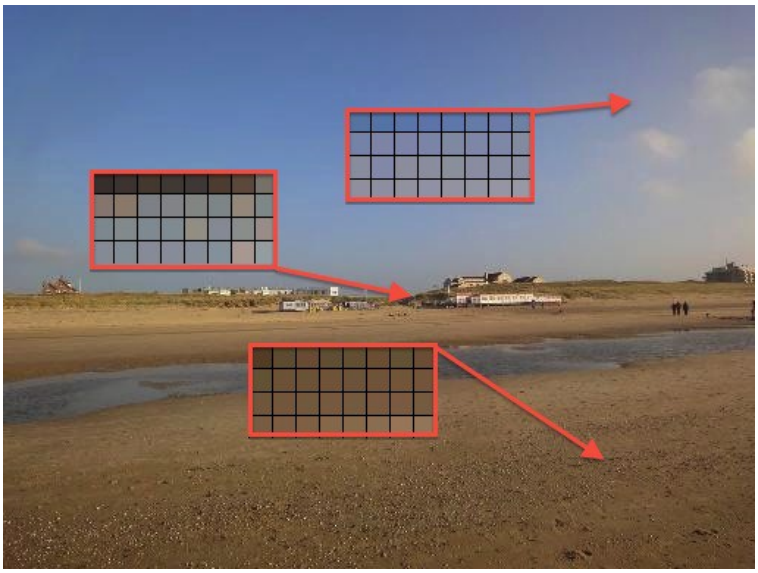


Figura 3.4: Ejemplo de codificación de chaché de color [24].

## 3.3. FLIF: Free Lossless Image Format

FLIF [44] es un algoritmo de compresión de imágenes sin pérdida que utiliza el espacio de color YCoCg, con soporte para el canal alfa en caso que sea necesario. Este algoritmo funciona codificando diferencias entre predicciones de píxeles y sus valores reales, de forma similar a LOCO-I.

### 3.3.1. YCoCg

YCoCg es un espacio de color que en lugar de usar los tres canales RGB tiene el canal Y o *Luma* que representa la intensidad o *luminancia* de cada píxel, el canal Co (Croma rojo) y Cg (Croma verde).

### 3.3.2. Recorrido de imagen y predicción de píxeles

El algoritmo puede trabajar con dos modos de recorrida de la imagen: no entrelazado y entrelazado. El recorrido no entrelazado hace referencia al mismo recorrido de LOCO-I, scan-line. Además, la predicción también se calcula al igual que la predicción de LOCO-I, explicado en la sección 3.1.1.

Para la recorrida entrelazada de una imagen, esta se divide en secciones de $8 \times 8$ y para cada una de estas secciones se repite el mismo proceso. Este proceso consiste en visitar el píxel superior izquierdo de cada sección y luego partir cada sección en dos secciones del mismo tamaño. A continuación se visita el píxel superior izquierdo de cada nueva sección que no haya sido visitado aún, y las secciones se vuelven a partir. Este proceso alternado de partición de secciones y visita de píxeles se repite hasta que todos los píxeles hayan sido visitados. La partición de secciones se realiza por filas y por columnas alternadamente. Entonces, en el primer paso simplemente se visita el píxel en la esquina superior izquierda de cada sección. Esto se ilustra en el paso 1 de la figura 3.5. Luego, las secciones se dividen por columna (*paso vertical*) y se visitan los píxeles representados en azul en el paso 2 de la figura 3.5. A continuación la división se realiza por filas (*paso horizontal*) y se visitan los píxeles en azul en el paso 3 de la figura 3.5. Este proceso se repite hasta completar la recorrida (paso 7 de la figura 3.5).

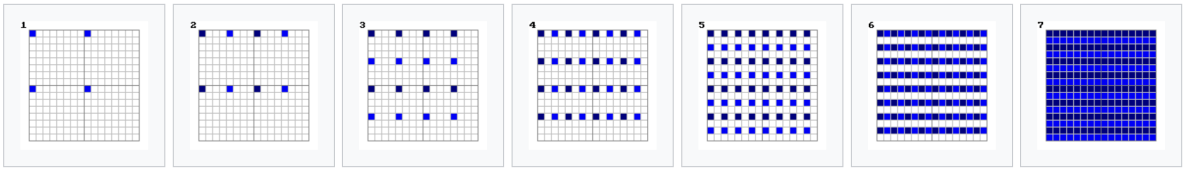



Figura 3.5: La primera muestra el primer paso en donde se divide la imagen de $16 \times 16$ en 4 secciones de $8 \times 8$ y se toma el primer píxel en cada una. Luego se realiza sucesivamente pasos horizontales y verticales hasta llegar a la imagen 7 donde se recorren todas las filas pares de la imagen (numerando las filas desde 1) [44].

Cuando se hace un recorrido entrelazado, si consideramos la subimagen formada por los píxeles superior izquierdo de las secciones en que se encuentra dividida la imagen durante la recorrida, los píxeles indicados en negrita en la figura 3.6 son conocidos tanto por el codificador como por el decodificador al momento de codificar el píxel marcado con "?". En un paso horizontal $B$ también es conocido, mientras que en un paso vertical $R$ es conocido.

En este caso, se definen tres predictores posibles para usar, con la posibilidad de usar uno diferente en cada canal de la imagen:

1. El promedio de arriba y abajo, $(T + B)/2$, si es un paso horizontal. O el promedio de izquierda a derecha, $(L + R)/2$, si es un paso vertical.

2. La mediana de: el promedio anterior, el gradiente superior izquierdo, $T + L - TL$, y el gradiente $L + B - BL$ o $T + R - TR$, según el paso actual.

3. La mediana de tres píxeles vecinos conocidos ($L, T$ y $B$ o $R$).

| | | | |
|---|---|---|---|
| | | $\boldsymbol{TT}$ | |
| | $\boldsymbol{TL}$ | $\boldsymbol{T}$ | $\boldsymbol{TR}$ |
| $\boldsymbol{LL}$ | $\boldsymbol{L}$ | ? | $R$ |
| | $\boldsymbol{BL}$ | $B$ | $\boldsymbol{BR}$ |

Figura 3.6: Contexto de un píxel[44].

### 3.3.3. Árboles de decisión

Un árbol de decisión es un modelo de predicción usado en el área de inteligencia artificial, en el que la predicción se hace recorriendo las ramas en función de la entrada. Cada nodo interno es una condición o pregunta que se hace sobre la entrada, y cada rama representa un resultado; si se cumple se avanza por esa rama hasta llegar a un nodo hoja que representa la predicción final del árbol para esa entrada. En el caso de FLIF, la codificación MANIAC utiliza un árbol de decisión construyendo esas condiciones en base a los posibles contextos.

### 3.3.4. Codificación de entropía: MANIAC

El método de codificación en FLIF se denomina MANIAC. Es una variante de la Codificación Aritmética Binaria Adaptativa al Contexto (CABAC) [25] en la que no solo el modelo de probabilidad es adaptativo (basado en el contexto local), sino que también el modelo de contexto en sí es adaptativo.

En general, es difícil definir un buen modelo de contexto; en particular, usar demasiados contextos daña la compresión porque la adaptación a cada uno es limitada (pocos píxeles por contexto); pero con muy pocos, la compresión también se ve afectada porque los píxeles con diferentes propiedades estadísticas terminan en el mismo contexto. En MANIAC se usa una estructura de datos dinámica como modelo de contexto, permitiendo que el número de

contextos se adapte a cada imagen. Dicha estructura es básicamente un árbol de decisiones, que se aprende dinámicamente en el momento de la codificación. Esto permite que el modelo sea más específico para la imagen.

### 3.3.5. Árbol de decisión MANIAC

En este árbol se definen contextos que se construyen a partir de la combinación de *propiedades*, las cuales dependen del recorrido de la imagen. En el caso en que los datos de la imagen son una matriz de píxeles ordenados en scan-line, se eligen 5 diferencias tomando en cuenta el contexto del píxel: $L - TL$, $TL - T$, $T - TR$, $LL - L$, $TT - T$, además de la predicción del píxel. En el otro caso se usan las mismas excepto que en lugar de $L - TL$ y $TL - T$, se usa $L - \frac{TL+BL}{2}$ y $T - \frac{TL+TR}{2}$, y en lugar de $T - TR$ se usa también $B - \frac{BL+BR}{2}$ (para el paso horizontal) o $R - \frac{TR+BR}{2}$ (para el paso vertical). Adicionalmente se agrega la diferencia entre los dos píxeles adyacentes del paso anterior del entrelazado ($T - B$ para el paso horizontal, $L - R$ para el paso vertical).

Como mencionamos en la sección 3.3.3, cada nodo intermedio del árbol es una desigualdad que compara el valor de alguna de las propiedades del contexto con un valor, para luego ir por una de las dos ramas. Una hoja del árbol constituye el *contexto actual*, es decir, las combinaciones de propiedades que llevan hasta esa hoja determinan el contexto del píxel actual.

Para cada propiedad, cada nodo hoja mantiene un promedio de los valores de esa propiedad para los píxeles que llegan a esa hoja. Además se almacena en una lista los valores mayores a ese promedio y en otra los menores. La idea es encontrar en cada hoja las propiedades más relevantes. En caso de encontrar una propiedad que mejore la compresión el nodo hoja se convierte en un nodo intermedio tomando ese promedio como el valor a considerar para elegir una de las dos ramas.

# Capítulo 4

# Estado del arte en compresión de imágenes con redes neuronales

En este capítulo presentamos siete algoritmos de compresión de imágenes sin pérdida basados en redes neuronales, L3C, IDF, IDF++, HyperPrior, HiLLoC, RC y LBB. Todos estos algoritmos siguen un procedimiento de compresión con las características presentadas en la figura 4.1, que ilustra un codificador. A su vez cada codificador tiene un decodificador que vuelve la imagen comprimida a la imagen original. Para cada uno de los distintos algoritmos describimos su arquitectura con sus transformaciones así como también la codificación de entropía elegida.

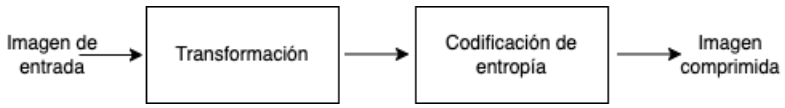



Figura 4.1: Procedimiento de compresión.

## 4.1. L3C

El sistema L3C[29] para codificar una imagen $x$, utiliza los siguientes elementos: $S$ *extractores de características* ($E^{(s)}$), $S$ *predictores* ($D^{(s)}$) y un *cuantificador* ($Q$), con $S = 3$. En la figura 4.2 se puede observar una visión general de la arquitectura de L3C.

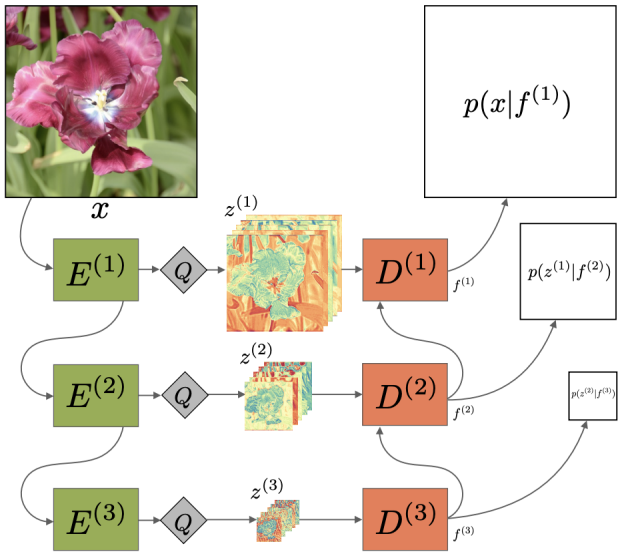



Figura 4.2: Visión general de la arquitectura de L3C[29].

Los extractores de características $E^{(s)}$, tienen como objetivo construir una representación de las características de la imagen. Además, como podemos observar en la figura 4.2, su representación es jerárquica. Las características extraídas por cada bloque son usadas como entrada para el siguiente bloque. Los extractores están cuantificados por $Q$, dando como resultado una característica $z^{(s)}$ de la imagen de entrada. En este caso se usa una función de cuantificación escalar[28].

Por otro lado, se tienen los predictores $D^{(s)}$, que como su nombre lo describe, se encargan de predecir la distribución de probabilidad $p$ de $x$ y de las características $z^{(s)}$. Al igual que los extractores tienen una representación jerárquica, donde cada bloque $D^{(s+1)}$ funciona como entrada para el bloque predictor $D^{(s)}$. Luego estas predicciones se utilizan con la codificación aritmética presentada en el sección 2.4.2, para obtener el *stream* de salida, que define a la imagen comprimida.

Tanto $E^{(s)}$ como $D^{(s)}$ se modelan como redes neuronales convolucionales.

La función de pérdida utilizada en este algoritmo es una generalización de la probabilidad *Discretized Logistic Mixture*, $\mathcal{L}$, presentada en el algoritmo de compresión de imágenes con pérdida PixelCNN++ [41]. Dadas $N$ muestras del conjunto de entrenamiento, $\{x_1, x_2, \ldots, x_N\}$, se define $z_i^{(s)}$ como la representación de las características de la $i$-ésima muestra en el nivel

$s$. Entonces, se busca minimizar

$$\begin{aligned}
&\mathcal{L}(E^{(1)}, \ldots, E^{(S)}, D^{(1)}, \ldots, D^{(S)}) \\
&= -\sum_{i=1}^{N} \log\left(p(x_i)\right) \qquad (4.1a) \\
&= -\sum_{i=1}^{N} \log\left(p(x_i|z_i^{(1)}, \ldots, z_i^{(S)}) \cdot \prod_{s=1}^{S} p(z_i^{(s)}|z_i^{(s+1)}, \ldots, z_i^{(S)})\right) \\
&= -\sum_{i=1}^{N} \left(\log p(x_i|z_i^{(1)}, \ldots, z_i^{(S)}) + \sum_{s=1}^{S} \log p(z_i^{(s)}|z_i^{(s+1)}, \ldots, z_i^{(S)})\right) ,
\end{aligned}$$

donde para $s = S$ el término $p(z_i^{(s)}|z_i^{(s+1)}, \ldots, z_i^{(S)})$ denota la distribución de probabilidad uniforme sobre el espacio de características de nivel $S$.

Esta función de perdida corresponde con la función logarítmica negativa de verosimilitud (*negative log likelihood*), y es aproximadamente igual al largo de código que se obtiene con un codificador aritmético para las imágenes $x_1 \ldots x_N$ cuando se usa la distribución de probabilidad $p(x_i)$ de la ecuación (4.1a).

Para comprimir una imagen $x = z^{(0)}$, se comienza por codificar $z^{(S)}$ con distribución uniforme. Luego para cada $s$, $1 \leq s < S$, se codifica $z^{(s)}$ con una distribución de probabilidad condicionada en $f^{(s+1)}$, $p(z^{(s)}|f^{(s+1)})$, con $f^{(s+1)} = z^{(s+1)}, \ldots, z^{(S)}$. Ver figura 4.2.

## 4.2. IDF

Integer discrete flow [13] es de los primeros algoritmos en usar un modelo basado en flows, ver sección 2.6.5. La idea general y que motiva el uso de flows es alterar la imagen de entrada para llevarla a un espacio con distribución conocida para luego usar algún algoritmo de codificación, por ejemplo, codificación aritmética, aprovechando que se conoce como se distribuyen los datos. Para obtener nuevamente la imagen se decodifica esos datos aplicando la inversa del algoritmo de codificación usado y aplicando la inversa de los flows para obtener nuevamente la imagen.

Como mencionamos en la sección 2.6.5, un coupling flow divide la entrada en $x = (x^A, x^B) \in \mathbb{R}^n$; en el caso de IDF se toma $\frac{3}{4}$ de la entrada para $x^A$ y $\frac{1}{4}$ para $x^B$, la salida será $z = (z^A, z^B)$ con $z^A = x^A$ y $z^B = x^B + \lfloor t(x^A) \rceil$ en donde $\lfloor \cdot \rceil$ denota la operación de

redondeo al entero mas cercano y $t(\cdot)$ una red neuronal.

Posteriormente se desarrolló IDF++[4], una variante de IDF que propone algunos cambios en la arquitectura mejorando las tasas de compresión pero manteniendo el costo computacional.

## 4.3. HyperPrior sin pérdida

HyperPrior [2] es una técnica para compresión de imágenes con pérdida. Este nuevo algoritmo [5] generaliza hyperprior para la compresión sin pérdida.

La figura 4.3 (a) muestra un diagrama operativo para la compresión de imágenes con pérdida en la que se basa este algoritmo. Está definido a partir de

- $y = g_a(x, \phi)$,
- $\hat{y} = Q(y)$,
- $\hat{x} = g_s(\hat{y}, \theta)$,

donde $x$ es la imagen de entrada, $g_a$ representa la función de codificación, $\phi$ representa los parámetros optimizados durante el entrenamiento y $Q$ es un cuantificador uniforme, que da como resultado a $\hat{y}$. Luego, $g_s$ representa la función de decodificación, con los parámetros $\theta$ que son optimizados durante el entrenamiento, y la salida del codificador $\hat{y}$, obteniendo $\hat{x}$, una reconstrucción de la imagen $x$. Tanto el codificador como el decodificador son redes neuronales convolucionales, por lo que hasta aquí se tiene la estructura de un autoencoder.

En la figura 4.3 (b) se muestra un diagrama operativo del algoritmo hyperprior con pérdida, en el cual se introduce una variable extra, $z$, con el objetivo de capturar las dependencias espaciales entre los elementos de la imagen. Podemos resumirlo como

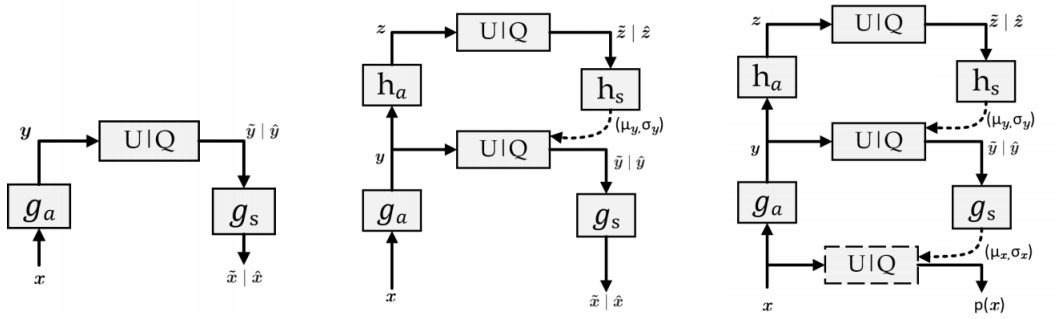
- $z = h_a(y, \phi_h)$,
- $\hat{z} = Q(z)$,
- $(\mu_y, \sigma_y) = h_s(\hat{z}, \theta_h)$,

donde $h_a$ representa una transformación de codificación, $\phi_h$ parámetros optimizados durante el entrenamiento y $Q$ una función de cuantificación, que da como resultado $\hat{z}$. $h_s$ representa al decodificador, que da como resultados los valores reales, $\mu_y$ y $\sigma_y$ que son la media y la desviación estándar para una arquitectura VAE, mencionada en la sección 2.6.4.

En resumen, esta arquitectura se construye a partir de un autoencoder y un variational autoencoder, ambos compuestos por redes convolucionales. El autoencoder original obtiene a partir de la imagen de entrada $x$ un código latente $y$, este código es utilizado como entrada para el VAE que obtiene un código latente $z$. Luego z se cuantifica a partir de la función de cuantificación $Q$, dando como resultado $\hat{z}$, el cual se codifica con un codificador aritmético. Del lado del decodificador, luego de decodificar $\hat{z}$ a partir del decodificador aritmético, se utiliza el decodificador VAE y se obtiene la media ($\mu_y$) y la desviación estándar ($\sigma_y$), lo cual se usa para modelar a $\hat{y}$ como una gaussiana $\mathcal{N}(0, \sigma_y^2)$ (se desestima $\mu_y$). Una vez aproximado $\hat{y} \sim \mathcal{N}(0, \sigma_y^2)$ se utiliza como entrada en $g_s$ para obtener la imagen reconstruida $\hat{x}$.

El algoritmo hyperprior sin pérdida lo que hace es generalizar el diagrama mostrado en la figura 4.3 (b) que es para una compresión con pérdida a uno con compresión sin pérdida, el cual se representa en la figura 4.3 (c).

La modificación que convierte el algoritmo con pérdida a uno sin pérdida es la representación de la imagen de entrada $x$. Anteriormente la imagen $x$ se representaba como $\hat{x} = g_s(\hat{y}, \theta)$, mientras que la versión sin pérdida modela a $x$ como una gaussiana $\mathcal{N}(\mu_x, \sigma_x^2)$. Para lograr esto el autoencoder original se reemplaza por un VAE, que da como resultados los parámetros $\mu_x$ y $\sigma_x$. Conocer la distribución de $x$ permite aplicar una codificación aritmética y así codificar $x$ sin pérdida de información. Otra modificación implementada por esta variación es usar la media $\mu_y$ para modelar $\hat{y}$ como una distribución gaussiana $\mathcal{N}(\mu_y, \sigma_y^2)$ (antes no se usaba $\mu_y$).




(a) Lossy model

(b) Lossy model with a hyperprior

(c) Proposed Lossless model with a hyperprior

Figura 4.3: Diagramas operativos de compresión de imagen con pérdida y sin pérdida [5].

## 4.4. RC

El algoritmo RC [27] combina la compresión con pérdida de Better Portable Graphics (BPG[1]) con redes neuronales convolucionales. Específicamente, la imagen original se descompone en la reconstrucción con pérdida obtenida después de comprimirla con BPG y el residuo correspondiente (QC). Luego se modela la distribución del residuo con un modelo probabilístico convolucional (RC) basado en redes neuronales que está condicionado a la reconstrucción de BPG, y se combina con codificación entrópica, específicamente codificación aritmética (AC, descrita en la sección 2.4.2), para codificar sin pérdida el residuo. Luego la imagen se almacena usando la concatenación de stream producida por BPG y la codificación del residuo aprendida.

En la figura 4.4 se puede observar esto gráficamente. En gris se observa como reconstruir la imagen $x$ y en violeta los componentes aprendidos.

[1]Formato de archivo de imagen [3].

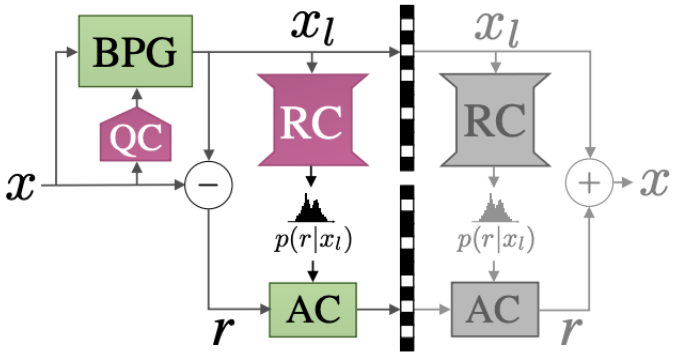



Figura 4.4: Descripción general de la compresión propuesta por RC [27].

## 4.5. BB-ANS

BB-ANS [48] es un algoritmo para realizar compresión sin pérdida con modelos de variables latentes. Su argumento se basa en que los modelos de variables latentes permiten que una imagen $x$ se codifique de manera más eficaz cuando está condicionada a una variable latente $z$.

Este algoritmo funciona para un conjunto de imágenes, dónde cada imagen $x$ y la variable latente $z$ se modelan como variables aleatorias con distribución conjunta conocida, $p_\theta(x, z) = p_\theta(x|z)p_\theta(z)$, y también se supone conocida su distribución posterior $q_\theta(z|x)$, dónde $\theta$ representa los parámetros optimizados durante el entrenamiento. Tanto $p_\theta$ como $q_\theta$ generalmente son aprendidas utilizando modelos de variables latentes, como por ejemplo los VAE, presentados en la sección 2.6.4. Las imágenes son representadas en un tensor de dimensión $3 \times H \times W$, con $H$ la altura y $W$ el ancho de la imagen, con valores en el rango $[0, 255]$.

BB-ANS opera comenzando con un bitstream de $N_{init}$ bits iniciales aleatorios independientes con distribución uniforme. Luego, para codificar $x$, realiza los pasos siguientes, representados también en la figura 4.5:

1. Decodificar $z$ con rANS a partir del bitstream usando $q_\theta(z|x)$, lo cual consume $-\log q_\theta(z|x)$ bits del bitstream.

2. Codificar $x$ con rANS al bitstream usando $p_\theta(x|z)$, lo cual agrega $-\log p_\theta(x|z)$ bits al bitstream.

3. Codificar z con rANS al bitstream usando $p_\theta(z)$, lo cual agrega $-\log p_\theta(z)$ bits al bitstream.

El bitstream resultante tiene un largo igual a $N_{total} = N_{init} + \log q_\theta(z|x) - \log p_\theta(x|z) - \log p_\theta(z)$ bits.

Luego, para la próxima imagen en vez de utilizar bits aleatorios se usa el bitstream resultante de la imagen previamente codificada. Y así sucesivamente para el resto de imágenes del conjunto.

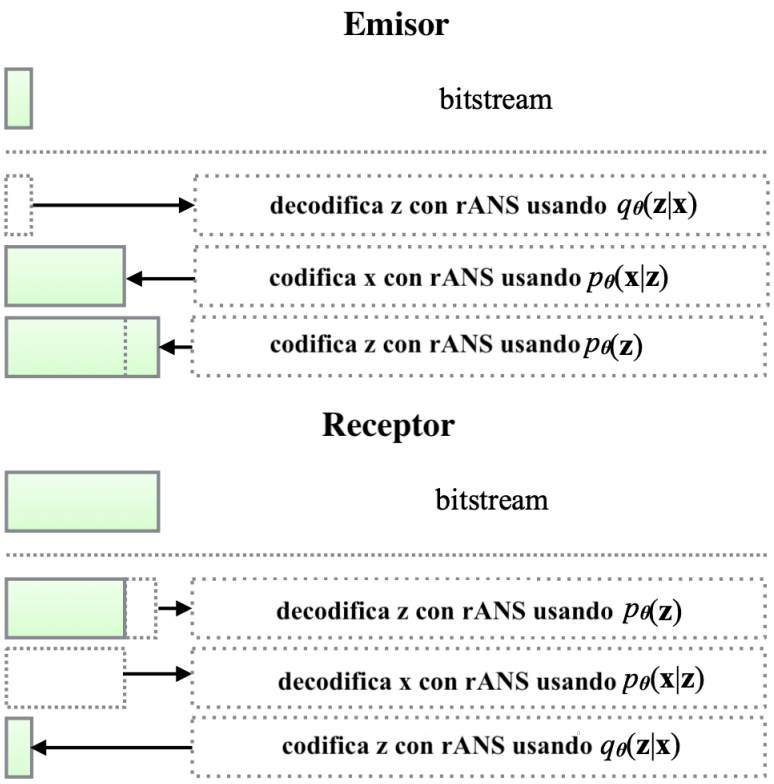



Figura 4.5: BB-ANS: Bits-Back coding con Sistemas Numéricos Asimétricos(ANS).

## 4.6. HiLLoC

Lossless image compression with hirerarchical latent variable models (HiLLoC[49]) es una extensión de BB-ANS con una red de arquitectura VAE compuesta por redes convoluciona-

les entrenadas con menor resolución para luego extenderlas a imágenes de mayor tamaño, logrando compresión sin pérdida en imágenes de tamaño arbitrario.

En resumen, el proceso consiste en pasar la imagen por una red convolucional, luego discretizar el vector resultante (*espacio latente*) para poder codificarlo utilizado codificación ANS en un *stream* previamente inicializado con valores aleatorios.

La figura 4.6 muestra la arquitectura de alto nivel del proceso de codificación.

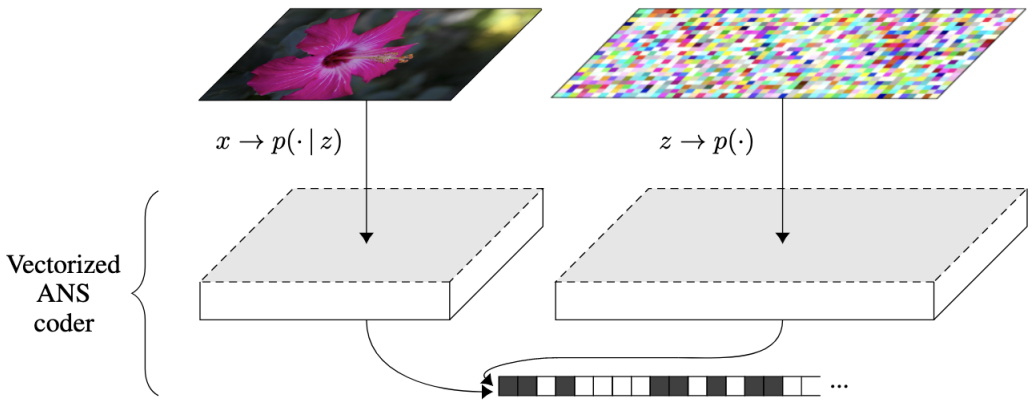



Figura 4.6: Arquitectura de alto nivel del proceso de codificación de HiLLoC luego de procesar la imagen con la red[49].

## 4.7. LBB

*Local bits back* [12] (LBB) es un algoritmo que utiliza flows, presentados en la sección 2.6.5, en combinación con la codificación de BB-ANS.

La idea general del algoritmo, que motiva al uso de flows, consiste en llevar la distribución de todos los píxeles en el espacio de las imágenes (distribución desconocida) a un espacio con distribución conocida (en este caso gaussiana). Al usar flows el pasaje de un espacio a otro se hace con funciones invertibles por lo que no hay pérdida de información en estas transformaciones.

Una vez que se obtienen los parámetros de los flows es posible utilizar un algoritmo de codificación para comprimir los datos transformados aprovechando su distribución conocida. En este caso se utiliza streaming rANS (explicado en la sección 2.4.3). De forma similar a

BB-ANS, el algoritmo parte de una secuencia de bits iniciales a la que llamaremos *stream base*. Cada vez que se codifica una imagen, este stream base se modifica quitando y agrando elementos como explicamos en la sección 4.5.

Finalmente, el proceso de decodificación consiste en obtener de nuevo los datos codificados invirtiendo estas transformaciones sobre el stream base usando la decodificación de streaming rANS y luego aplicar la inversa de todos los flows para obtener de nuevo la imagen original.

Para explicar en mayor detalle el proceso de transformación de la imagen y de entrenamiento de la red, presentamos dos conceptos centrales a continuación.

### Decuantización

Las imágenes de entrada de cada dataset están en un espacio discreto $x \in \mathbb{Z}^n$. Para hacer el modelo basado en flows adecuado para estos datos discretos, es una práctica usual definir una variable aleatoria continua $u \in [0,1)^n$, que se suma a la entrada $x$ antes de pasarla al flow. La distribución de $u$, $q(u|x)$, puede ser simplemente una distribución uniforme o, como en este caso, una función a partir de un modelo basado en flows (que a partir de ahora llamaremos $g$) y es entrenado junto con el modelo principal.

### Codificador y decodificador

Una vez definida la imagen de entrada $x$, el flow principal $f$, la función de decuantización $g$ con la que obtenemos $u$ y el stream base construido con bits aleatorios, el proceso de codificación como se ilustra en la figura 4.7 consiste en

1. Usar la decodificación de rANS a partir del stream para obtener una secuencia de números enteros aleatorios. Notar que este modifica el stream base.
2. Transformar estos datos aplicando $g^{-1}$ para obtener la decuantización $u$.
3. Obtener la imagen decuantizada $x + u$.
4. Aplicar el flow principal $f$ a $x + u$.
5. Cuantizar la salida $f(x + u)$ para tener datos discretos.
6. Codificar el resultado con rANS modificando el stream.

Para la decodificación se aplican las funciones inversas. Primero se decodifica $x+u$. Luego se recupera $x$ discretizando $x+u$ y a continuación se obtiene $u$ restando $x$. Teniendo de nuevo $u$, se puede recuperar el stream base original codificando $g(u)$ con rANS, por lo que es posible aprovechar esta información como mostraremos en el capítulo 6.

**Función de pérdida**

Un modelo basado en flows se entrena con cierta función de pérdida. En este caso necesitamos también minimizar la longitud de código final. Para esto el procedimiento usual es discretizar los datos de $\mathbb{R}^n$ tomando particiones (*bins*) de volumen $\delta_x = 2^{-kn}$, donde $k$ es la precisión de la discretización. Dado $x \in \mathbb{R}^n$, sea $B(x)$ el único bin que contiene a $x$, y sea $\hat{x}$ el centro del bin $B(x)$. Entonces, la distribución $P$ para los datos discretizados es $P(x) = \int_{B(\hat{x})} p(x)\mathrm{d}x$, que para el flow $f$ del modelo, que por construcción es una función suave, se puede aproximar como $P(\hat{x}) \approx p(\hat{x})\delta_x$. El largo de código que se obtiene para esta distribución de probabilidad es $-\log P(x)$. Luego, aprovechando el cambio de variable como en la ecuación (2.3), la función de pérdida para este algoritmo es

$$L_{teorico} = -\log p(\hat{x})\delta_x = -\log p(f(\hat{x})) - \log\left|\det Df(\hat{x})\right| - \log\delta_x \ . \tag{4.2}$$

Además debemos considerar la decuantización inicial para el algoritmo, $u \sim q(u|x)$, usando el flow auxiliar $g$. La función de pérdida considera ambos flows para ajustar los parámetros en conjunto, minimizando la siguiente función

$$\log q(u|x)\delta_x - \log p(x+u)\delta_x \ ,$$

donde $\log q(u|x)\delta_x$ es un valor negativo que corresponde a la cantidad de bits sustraídos del stream base mediante el decodificador rANS para obtener $u$, y $-\log p(x+u)\delta_x$ es el largo de código que se suma al stream base para codificar $x+u$. Dado esto, a la ecuación final de pérdida a partir de 4.2 se le agrega el término de decuantización y se reemplaza $x$ por $x+u$.

$$L_{teorico} = \log q(u|x)\delta_x - \log p(f(\widehat{x+u})) - \log\left|\det Df(\widehat{x+u})\right| - \log\delta_x \ . \tag{4.3}$$

### 4.7.1. Arquitectura

La figura 4.7 muestra la arquitectura a alto nivel de LBB. En esta arquitectura participan los flows f y g descritos anteriormente. Tanto los parámetros de $f$ como los de $g$ son ajustados en el entrenamiento para minimizar el largo de código final.

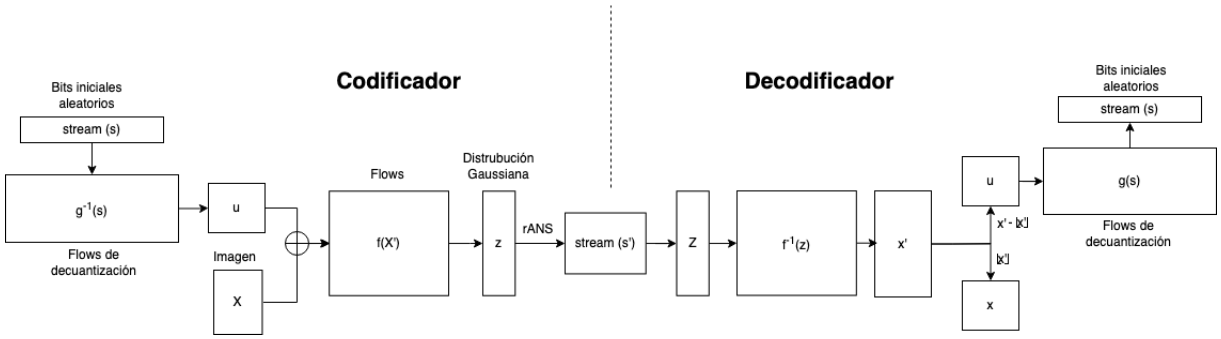



Figura 4.7: Descripción a alto nivel de la arquitectura de LBB.

El proceso de codificación se puede repetir para codificar $N$ imágenes, ya que el stream base modificado luego de codificar una imagen se puede volver a usar como stream base para las siguientes imágenes, como explicamos en la sección 2.4.3. Esto es muy útil para reducir el sobrecosto del stream base; LBB no tiene un buen desempeño para comprimir pocas imágenes debido a que este stream base agrega redundancia, pero a medida que se comprimen más imágenes y este efecto se diluye, el algoritmo alcanza un buen desempeño. Esto lo explicamos con mayor detalle en los capítulos 6 y 7. El decodificador recibe el stream modificado y puede recuperar cada imagen realizando el proceso contrario, aplica la inversa de los flows sobre el stream, y anula la decuantización para obtener nuevamente la imagen. Luego, se repite el proceso hasta obtener todas las imágenes y además recupera el stream base original por lo que este puede ser utilizado para otros propósitos con el fin de mejorar el algoritmo como proponemos en el capítulo 6.

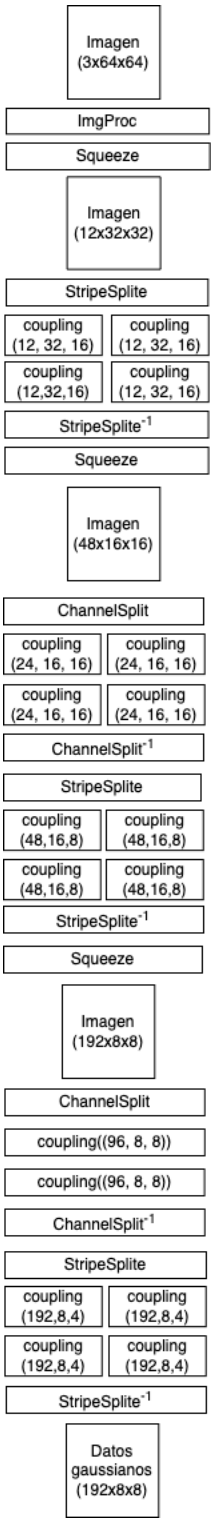



Figura 4.8: Descripción detallada de la composición de flows.

La figura 4.8 muestra de manera más detallada la composición del flow principal (función $f$). Inicialmente los datos de los píxeles que vienen de la imagen de entrada son enteros y mediante la decuantización explicada en la sección 4.7 son pasados a punto flotante para poder pasarlo por el flow correspondiente. Todas las transformaciones desde la primer etapa hasta la última trabajan con los datos en punto flotante, ya sea cambiando de rango los números, cambiando las dimensiones de la entrada o aplicándole alguna función invertible como la sigmoide. La decuantización ayuda de dos maneras a adaptar esta arquitectura para trabajar con píxeles de una imagen. Primero, convierte datos discretos en datos continuos que son lo que necesita como entrada una arquitectura basada en flows. Segundo, agrega precisión extra, lo que ayuda a poder recuperar los valores originales sin perder información por las operaciones en la práctica, lo cual verificamos en nuestros experimentos al momento de codificar y decodificar. En la etapa final, los píxeles se convierten en flotantes con distribución normal estándar que son discretizados como explicamos en la sección 4.7 y luego codificados uno a uno en rANS al igual que explicamos en la sección de BB-ANS 4.5. A continuación explicamos cada componente utilizado en esta arquitectura.

### ImgProc

El método **ImgProc** recibe como entrada una imagen representada como un tensor de dimensión $3 \times 64 \times 64$ (3 canales y 64 de largo y ancho). Este método lleva los valores de la imagen desde el rango $[0, 255]$ al rango $[0{,}05, 0{,}95]$. Le aplica la función sigmoide inversa a la imagen y retorna esta nueva representación de la imagen.

### Squeeze

El método **Squeeze** recibe como entrada una imagen representada como un tensor y mueve parte de los elementos en la dimensión de ancho y largo a la dimensión de los canales. Si una entrada tiene dimensión $C \times H \times W$, donde $C$ es el número de canales, $H$ la altura de la imagen y $W$ el ancho de la imagen, entonces el resultado tendrá dimensiones $4C \times H/2 \times W/2$.

### StripeSplit

**StripeSplit** recibe un tensor y lo divide en dos tensores. La división se hace a lo largo de la dimensión del tensor que representa el ancho de la imagen. En este caso, recibe un tensor de dimensión $12 \times 32 \times 32$ y retorna 2 tensores de dimensión $12 \times 32 \times 16$.

### ChannelSplit

Este método recibe un tensor y lo divide en dos tensores, la división se hace a lo largo de la dimensión del tensor que representa el canal. En este caso, recibe un tensor de dimensión $48 \times 16 \times 16$ y retorna dos tensores de dimensión $24 \times 16 \times 16$.

### Coupling

Esta parte de la red se compone de varias funciones como se ilustra en la figura 4.9, comenzando por una normalización que se ajusta en el entrenamiento. La etapa intermedia transforma la entrada normalizada con una combinación de funciones logísticas, posteriormente con una transformación afín y termina con una convolución 2D. Por último, se divide en dos la salida y se intercambian (flip) esas dos estas partes.

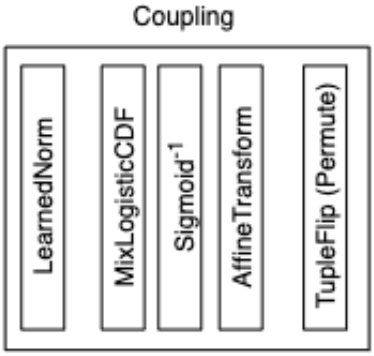



Figura 4.9: Descripción general de la arquitectura de LBB.

# Capítulo 5

# Conjunto de datos

Para cada algoritmo descrito en el capitulo 4 los autores presentan resultados en base a distintos conjuntos de datos, los cuales describimos en este capítulo.

Los algoritmos presentados utilizan estos conjuntos de datos, y en algunos casos usan técnicas para evitar el sobreajuste que puede darse debido al formato de las imágenes, como en el caso de imágenes JPEG con pérdida y el caso de datasets que surgen del submuestreo (o escalado) de imágenes. Estos datos no son los ideales, ya que el modelo podría no generalizar bien para la tarea de compresión sin pérdida.

## 5.1. ImageNet

ImageNet [38] es un conjunto de datos de imágenes organizado según la jerarquía de WordNet [1]. Cada concepto significativo en WordNet, posiblemente descrito por varias palabras o frases de palabras, se denomina *conjunto de sinónimos* o *synset*. Hay más de 100.000 *synsets* en WordNet, la mayoría de ellos son sustantivos (80.000+). En ImageNet, el objetivo es proporcionar un promedio de 1.000 imágenes para ilustrar cada synset.

ImageNet32 e ImageNet64 son variantes de ImageNet con la única diferencia de que las imágenes se reducen a $32 \times 32$ y $64 \times 64$ píxeles por imagen, respectivamente. Contienen aproximadamente 1.250.000 imágenes de entrenamiento y 50.000 imágenes de validación. Las imágenes están en formato PNG.

[1] Base de datos léxica del idioma Inglés.

## 5.2. CIFAR 10

El conjunto de datos CIFAR 10 [19] consta de 60.000 imágenes de $32 \times 32$ en 10 clases distintas, con 6.000 imágenes por clase, cuyas clases son completamente excluyentes entre sí. Hay 50.000 imágenes de entrenamiento y 10.000 imágenes de validación. Las imágenes están en formato PNG.

## 5.3. CLIC.mobile y CLIC.professional

Estos conjuntos de datos, presentados en *Workshop and Challenge on Learned Image Compression* [47], están conformados por aproximadamente 2.000 imágenes de entrenamiento, dónde CLIC.mobile contiene 61 imágenes de validación tomadas con teléfonos móviles, en un rango de $996 \times 756$ y $2016 \times 1512$ píxeles. Mientras que CLIC.pro contiene 41 imágenes de validación de DSLR, retocadas por profesionales, en un rango de $512 \times 384$ y $2048 \times 1370$ píxeles. Todas las imágenes están en formato PNG.

## 5.4. Open Images

Open Images [21] es un conjunto de datos de aproximadamente 9 millones de imágenes anotadas con etiquetas a nivel de imagen, cuadros delimitadores de objetos, máscaras de segmentación de objetos, relaciones visuales y narrativas localizadas. Estas imágenes están en formato JPEG y en un rango de $640 \times 480$ píxeles aproximadamente.

## 5.5. DIV2K

El conjunto de datos DIV2K [9], como su nombre lo dice cuenta con imágenes con resolución 2K, en un rango entre $2040 \times 816$ y $2040 \times 2040$ píxeles. Este dataset cuenta con 800 imágenes de entrenamiento y 100 imágenes de validación. Las imágenes están en formato PNG.

## 5.6. Kodak

El conjunto de datos Kodak, es un conjunto de datos con 25 imágenes PNG de tamaño $768 \times 512$ píxeles publicadas por Kodak Corporation.

## 5.7. Crops

El conjunto de datos Crops, es un conjunto definido por nosotros, creado a partir de cortes de $64 \times 64$ sobre imágenes de mayor tamaño extraídas de DIV2K, Open Images, CLIC.mobile y CLIC.professional, con un total de 20.000 imágenes.

# Capítulo 6

# Desarrollo y evaluación de compresores a partir de LBB

Este capítulo describe los cambios, adaptaciones y detalles clave, para lograr nuestro algoritmo final a partir de la investigación y la experimentación que realizamos.

## 6.1. Adaptación de LBB para ejecutar sobre imágenes grandes

LBB solo es capaz de codificar imágenes de $64 \times 64$ píxeles por la arquitectura de la red. Para evaluar su desempeño con imágenes de mayor tamaño implementamos una técnica que consiste en partir la imagen original en bloques disjuntos de $64 \times 64$. Luego se realiza la compresión de cada bloque por separado. Por ejemplo, si se tiene una imagen de tamaño inicial $512 \times 512$, se comprime $8 \times 8 = 64$ bloques por separado.

A los efectos de la evaluación experimental del algoritmo todas las imágenes son truncadas a largo y ancho que son múltiplos de 64. Por ejemplo, una imagen de tamaño $2048 \times 1365$ se divide en $32 \times 21 = 672$ bloques, truncando la imagen a $2048 \times 1344$ píxeles. Si se quisiera extender esto para usar esos bloques incompletos se podría completar la imagen con ceros alrededor (conocido como *padding*) previo a la codificación, mientras que el decodificador, que conoce el largo y ancho, descarta la parte sobrante.

## 6.2. Entrenamiento con LBB

Los autores del algoritmo LBB ofrecen sus propios modelos, junto con el código para su evaluación sobre los conjuntos de datos CIFAR 10, ImageNet32, ImageNet64. La implementación del entrenamiento correspondiente, sin embargo, no es pública.

Para realizar nuestra propia implementación del entrenamiento utilizamos el modelo y la evaluación para ImageNet64, puesto que es la base de datos que presenta imágenes de mayor resolución entre los conjuntos de datos disponibles.

Usamos la técnica de *learning rate decay*, que aplica modificaciones al hiperparámetro durante el entrenamiento. En particular, se utiliza el método *performance scheduling*, que calcula el error de validación cada N pasos, y reduce la tasa de aprendizaje un factor $\lambda$ cuando se estanca, N y $\lambda$ son hiperparámetros que ajustaremos.

Durante el entrenamiento no se está comprimiendo aún con rANS. El valor de la función de pérdida agregando la decuantización ($L_{teorico}$) dada en la ecuación 4.3, es solo una aproximación al largo de código total. Por lo que definimos la función de costo $f$ como

$$f = \frac{L_{teorico}}{D} \ ,$$

donde $D$ es el tamaño del conjunto de imágenes, $D = W \times H \times C \times N$ y $W, H, C, N$ son el ancho, el alto, la cantidad de canales y el número de imágenes del conjunto respectivamente.

## 6.3. Modelos entrenados

Para el entrenamiento definimos *snapshots* de 10 mil imágenes, es decir, cada 10 mil imágenes escribimos a disco una versión intermedia del modelo para evaluar el entrenamiento Esto resulta apropiado para el ambiente de ejecución de **cluster-uy** en el cual las tareas pueden durar como máximo cinco días.

De nuestro entrenamiento obtuvimos dos modelos:

- $LBB'$: Modelo entrenado por nosotros desde cero con imágenes del conjunto de datos ImageNet64.

- $LBB''$: Continuación del entrenamiento de $LBB'$ luego de 20 ciclos con el conjunto de datos Crops, definido en la sección 5.7.

Es decir, hasta el este momento se tiene un total de tres modelos: $LBB$, que es el original publicado por los autores, $LBB'$ y $LBB''$, que son entrenados por nosotros mismos.

En el cuadro 6.1, presentamos las tasas de compresión, medidas en BPD, obtenidos al comprimir los conjuntos de validación de las bases de datos Imagenet64 y Crops con los modelos LBB, $LBB'$ y $LBB''$.

| | Imagenet64 | Crops |
|---|---|---|
| LBB | 3.55 | 2.51 |
| $LBB'$ | 4.14 | 3.03 |
| $LBB''$ | 4.17 | 3.06 |

Cuadro 6.1: Tasa de compresión en BPD de los modelos LBB, $LBB'$ y $LBB''$ para el conjunto de validación de ImageNet64 y Crops.

Como se puede observar en el cuadro 6.1, el modelo $LBB'$ tiene resultados mejores que $LBB''$. Es por esto, que elegimos descartar el modelo $LBB''$ para los experimentos siguientes.

En la figura 6.1 se presenta la gráfica de costo en el entrenamiento para $LBB'$. También en la figura 6.2, se presenta una gráfica correspondiente al costo de entrenamiento a partir del *snapshot* 50, para visualizar mejor el mínimo obtenido, en este caso 2.89 BPD.

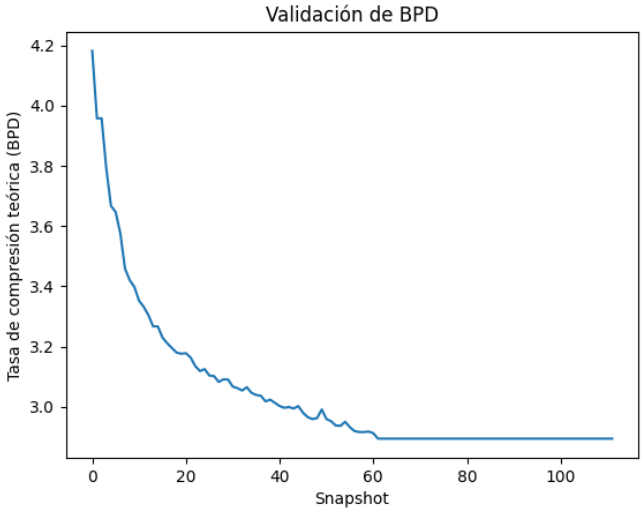



Figura 6.1: Función de costo promedio en BPD sobre el conjunto de validación.

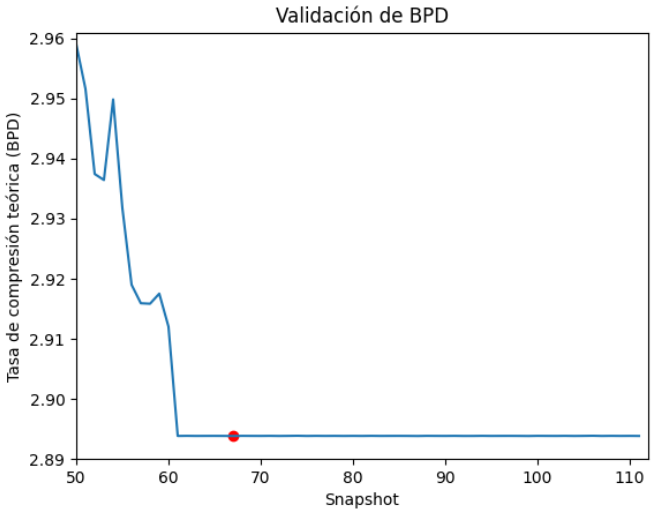



Figura 6.2: Función de costo promedio en BPD sobre el conjunto de validación a partir del *snapshot* 50. El punto rojo representa el mínimo de la gráfica.

En las gráficas se puede observar que el entrenamiento concluyó debido a que su valor se mantiene aproximadamente igual a partir del *snapshot* 60, es decir, a partir del modelo entrenado con 600 mil imágenes y no cambia en las próximas 400 mil imágenes.

## 6.4. Impacto del largo del stream base

Recordamos de la sección 4.7 que este algoritmo utiliza un stream base para la compresión de un conjunto de imágenes, una cadena inicial generada a partir de números pseudoaleatorios. En el trabajo original de LBB [12], este stream base no es considerado en el cálculo de la tasa de compresión. Sin embargo, este stream base es necesario para la compresión y descompresión de las imágenes en la práctica, y es parte del stream que se envía del codificador al decodificador, por lo que debería tomarse en cuenta.

### 6.4.1. Stream base

Este experimento consiste en estudiar el costo del stream base. Sea el tamaño del conjunto de imágenes $D = W \times H \times C \times N$ donde $W, H, C, N$ son el ancho, el alto, la cantidad de canales y el número de imágenes del conjunto respectivamente. Sea $S$ el largo del stream base (usualmente inicializado en 10 millones), $S'$ el largo del stream luego de codificar, y $l_{min}$ el menor largo que llega a tener el stream en el proceso de codificación (usualmente 650.000). Recordemos que en el proceso de codificación, diversas partes del stream agregan y quitan bits del stream, por lo que su largo varía con el tiempo y no necesariamente de manera monótona (ver discusión en sección 4.5). El cálculo de tasa de compresión en el trabajo original de LBB [12] es el siguiente:

$$R_{LBB} = \frac{S' - S}{D} \ ,$$

Notar que esta expresión es la que aproxima la función de costo 4.2 que se busca minimizar durante el entrenamiento del modelo. Aunque el stream base sufre modificaciones hay una parte que no es necesario enviar al decodificador, dado que no interesa recuperar nuevamente todos los bits aleatorios con los que se inicializó. Esos $l_{min}$ bits no fueron "tocados" por el codificador y por lo tanto no son necesarios para decodificar correctamente. Por lo tanto, para calcular la tasa de compresión real, usamos la siguiente fórmula:

$$R_{real} = \frac{S' - l_{min}}{D} \ .$$

En la figura 6.3 se grafican ambas tasas de compresión en función de la cantidad de imágenes comprimidas, tomando el conjunto de validación de ImageNet64. En la figura 6.4 se muestra la diferencia entre estas tasas para el mismo conjunto de imágenes. En el experimento se utilizó un *stream base* de $10^7$ bits, que es el mismo que utilizan los autores [12].

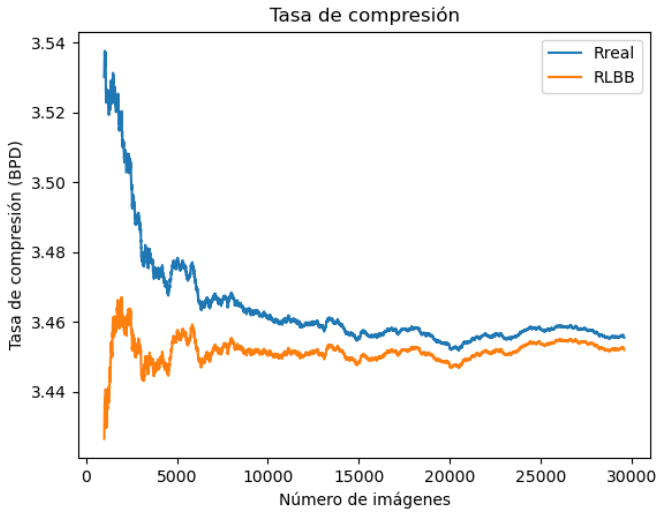



Figura 6.3: Tasa de compresión $R_{LBB}$ y $R_{real}$ en función de la cantidad de imágenes comprimidas en el conjunto de validación de ImageNet64.

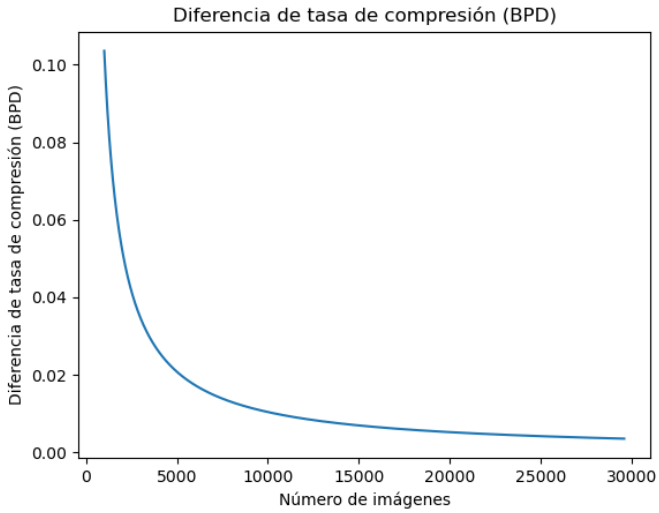



Figura 6.4: Diferencia entre el cálculo de tasa de compresión $R_{LBB}$ y $R_{real}$ en función de la cantidad de imágenes para el conjunto de validación de ImageNet64.

Se puede observar que a medida que el número de imágenes crece, el costo relativo del *stream base* comienza a ser despreciable. Para observar esto con mayor precisión, en las figuras 6.5 y 6.6 se muestra las mismas gráficas a partir de la imagen 10.000.

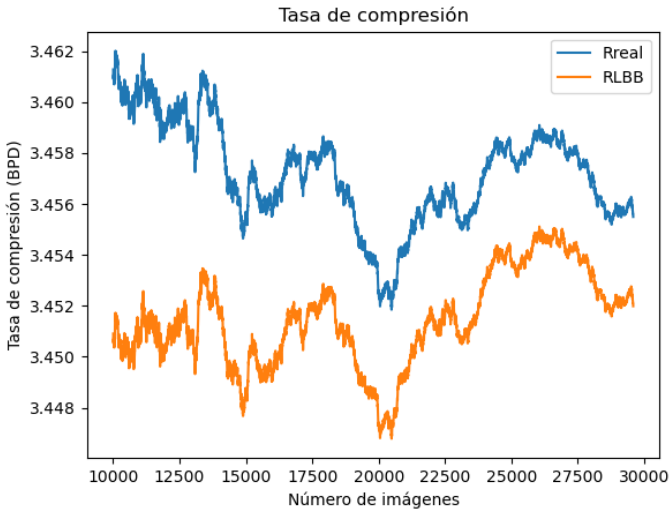



Figura 6.5: Tasa de compresión $R_{LBB}$ y $R_{real}$ en función de la cantidad de imágenes comprimidas para ImageNet64 a partir de la imagen 10.000.

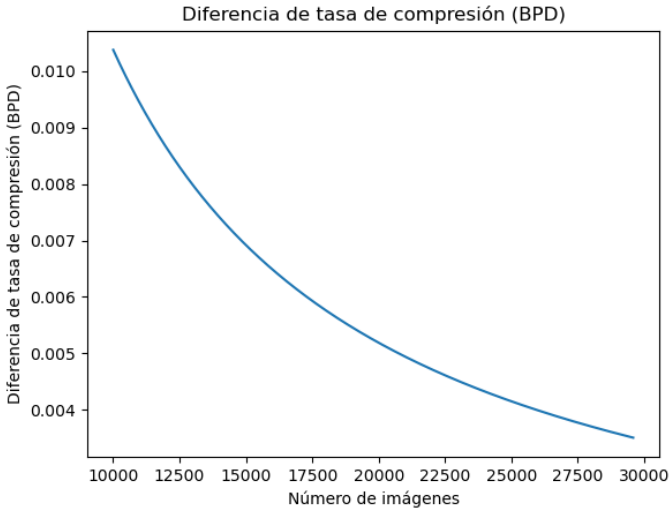



Figura 6.6: Diferencia entre el cálculo de tasa de compresión $R_{LBB}$ y $R_{real}$ en función de la cantidad de imágenes para el conjunto de validación de ImageNet64 a partir de la imagen 10.000.

## 6.5. Explotación del stream base

Con el objetivo de reducir el sobrecosto del stream base se experimentó usando un stream que, en lugar de inicializarse con números aleatorios a partir de una distribución uniforme como en el trabajo original, se parte de una etapa en donde las primeras imágenes del conjunto a comprimir son comprimidas con PNG, hasta alcanzar la cantidad de bits mínima necesaria del stream base para comenzar a utilizar el algoritmo LBB original, que en las pruebas eran entre 5 y 10 imágenes. A este algoritmo lo denominamos $LBB\_PNG$ y el cálculo de la tasa de compresión, dado que ahora todo el stream contiene información útil es:

$$R_{LBB\_PNG} = \frac{S}{D} \ ,$$

donde recordamos que S es el largo del stream base y D es el tamaño del conjunto de imágenes. Notar que sustituir los bits del stream base por imágenes comprimidas con PNG se fundamenta en el hecho de que cuando un compresor funciona bien, como en el caso de PNG, su salida se parece a bits aleatorios. Además al sustituir los números aleatorios por las imágenes en PNG ayuda a que no sea necesario generar números que solo agregan redundancia aumentando la tasa de compresión, dado que el stream original se recupera una vez que se descomprime con LBB estas imágenes se recuperan en la etapa final. La figura 6.7 muestra la tasa de compresión para este nuevo algoritmo comparando con los dos cálculos de tasa de compresión en el algoritmo original, en este caso converge mucho mas rápido al valor dado por la tasa $R_{LBB}$. En la figura 6.8 se puede observar con mayor detalle el comportamiento de las tres tasas para un gran número de imágenes.

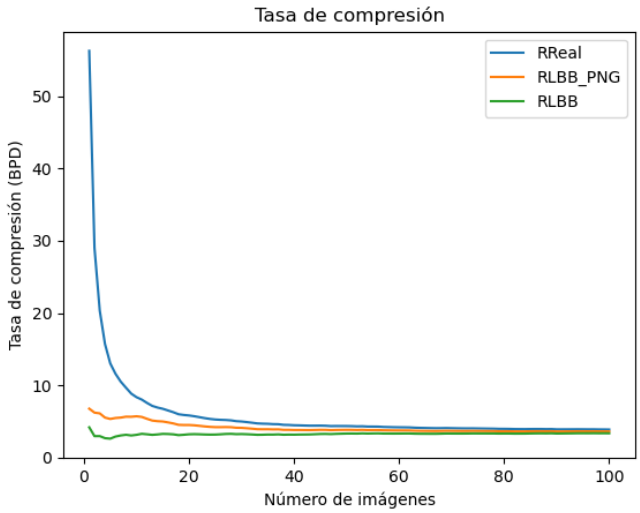



Figura 6.7: Tasas de compresión $R_{LBB}$, $R_{real}$ y $R_{LBB\_PNG}$ en función de la cantidad de imágenes para el conjunto de validación ImageNet64.

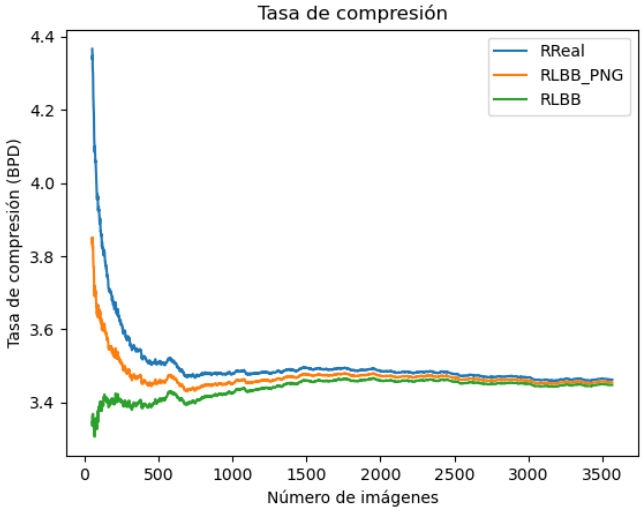



Figura 6.8: Tasas de compresión $R_{LBB}$, $R_{real}$ y $R_{LBB\_PNG}$ en función de la cantidad de imágenes para el conjunto de validación ImageNet64 a partir de la imagen 50.

La cantidad de bits mínima necesaria del *stream base* se calculó experimentalmente con el algoritmo original. El resultado arrojó que un par de imágenes es suficiente para llegar al mínimo.

## 6.6. Formato de archivo usado para representar las imágenes comprimidas

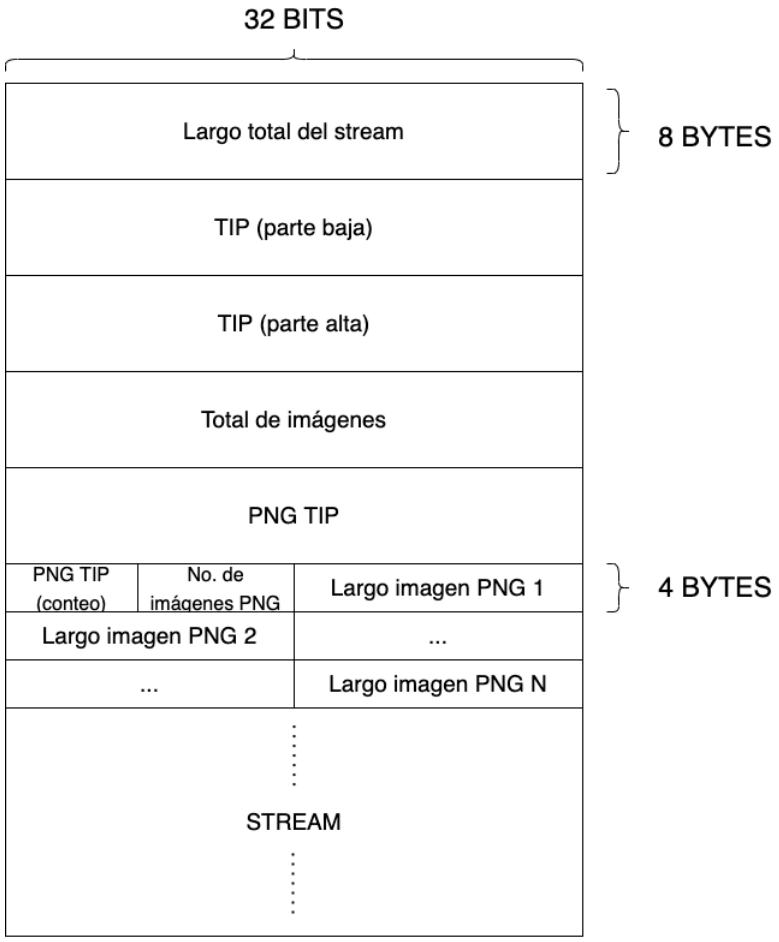



Figura 6.9: Descripción del formato del archivo para representar en disco las imágenes comprimidas con LBB_PNG.

En la figura 6.9 se describe el formato de archivo usado para representar las imágenes comprimidas con LBB_PNG, que consta de tres partes: un cabezal inicial (de largo fijo) que describe cada una de las variables auxiliares usadas por el algoritmo, una lista de largos y el stream.

A continuación se describen las variables del cabezal; siguiendo el orden de la figura se tiene:

- Largo total del stream en bytes.
- TIP es una variable auxiliar que usa el algoritmo LBB original para la codificación con rANS. Se usa para ir guardando el entero codificado con el algoritmo rANS. Es un entero de 128 bits por lo que debe dividirse en dos enteros de 64 bits para su utilización en el lenguaje de programación Python, por eso se tiene parte alta y baja.
- Número total de imágenes comprimidas (tanto con LBB como con PNG).
- PNG TIP es una variable que se utiliza para la inicialización del stream con imágenes comprimidas con PNG.
- PNG TIP (conteo), también se utiliza para la inicialización del stream con imágenes comprimidas con PNG.
- Número total de imágenes comprimidas con PNG.
- Lista con los largos de cada tira de bytes que representa una imagen comprimida con PNG, para poder decodificar cada una de ellas. Cada largo se escribe usando 2 bytes y la cantidad ($N$ en la imagen) está dada por el número de imágenes PNG que se lee antes de la lista (No. de imágenes PNG, ver figura 6.9).

# Capítulo 7

# Comparación experimental de algoritmos de compresión

Como se mencionó en el capítulo 4, hoy en día existen varios algoritmos de compresión sin pérdida. En la bibliografía se puede observar que los resultados reportados para los algoritmos estudiados se reportan sobre bases de datos diferentes, lo cual dificulta la comparación directa de los algoritmos entre sí.

Por un lado, tenemos los resultados para los algoritmos usando los conjuntos de validación de CIFAR10, ImageNet32 e ImageNet64, denominados HiLLoC, IDF, IDF++, LBB. Sus tasas de compresión en cada base se presentan en el cuadro 7.1, donde observamos que el que presenta mejores resultados es LBB, la tasa de compresión utilizada para LBB es $R_{LBB}$.

| | CIFAR10 | ImageNet32 | ImageNet64 |
|---|---|---|---|
| HiLLoC | 3.56 | 4.2 | 3.9 |
| IDF | 3.32 | 4.15 | 3.9 |
| IDF++ | 3.26 | 4.12 | 3.81 |
| LBB | **3.12** | **3.88** | **3.7** |

Cuadro 7.1: Tasa de compresión en BPD de los algoritmos HiLLoC, IDF, IDF++ y LBB para los conjuntos de validación de CIFAR10, ImageNet32 y el conjunto de validación de ImageNet64.

Por otro lado tenemos los algoritmo RC, L3C y HyperPrior, con los resultados correspondientes reportados sobre los conjuntos de validación de CLIC.pro, CLIC.mobile, Open

Images y DIV2K. En general el que mejores resultados presenta es RC, tal como se puede observar en el cuadro 7.2.

| | CLIC.pro | CLIC.mobile | Open Images | DIV2K |
|---|---|---|---|---|
| RC | 2.933 | **2.538** | **2.791** | **3.079** |
| L3C | 2.944 | 2.639 | 2.991 | 3.094 |
| HyperPrior | **2.726** | 2.659 | No registrado | No registrado |

Cuadro 7.2: Tasa de compresión en BPD de los algoritmos RC, L3C y HyperPrior para los conjuntos de validación de CLIC.pro, CLIC.mobile, Open Images, DIV2K.

Teniendo en cuenta estos resultados, a continuación comparamos entre sí el mejor algoritmo de cada grupo, LBB y RC, sobre los conjuntos de validación de CLIC.pro, CLIC.mobile, Open Images, DIV2K y Kodak. Para la comparación también se ejecutaron otros algoritmos que no están basados en redes neuronales, específicamente, JPEG-LS, WebP y FLIF (descritos en el capítulo 3). En el cuadro 7.3 se presentan sus tasas de compresión en cada base y se puede observar que el que presenta mejores resultados es LBB. Estos resultados se obtuvieron de nuestras propias pruebas a partir del código abierto de los algoritmos en cuestión. En particular para LBB, usamos el modelo $LBB$ y la tasa de compresión $R_{LBB}$. Para estos casos, tanto $R_{real}$ como $R_{LBB}$ dan resultados similares luego de un gran conjuntos de imágenes, como mostramos en la figura 6.3, por lo que reportado en las siguientes tablas es equivalente para $R_{real}$.

| | CLIC.pro | CLIC.mobile | Open Images | DIV2K | Kodak |
|---|---|---|---|---|---|
| JPEG-LS | 3.952 | 3.851 | 4.002 | 4.208 | 4.481 |
| WebP | 3.005 | 2.777 | 3.050 | 3.180 | 3.180 |
| FLIF | 3.787 | 2.498 | 2.875 | 2.919 | 2.903 |
| LBB | **2.480** | **2.260** | **2.525** | **2.569** | **2.777** |
| RC | 2.933 | 2.538 | 2.791 | 3.079 | 3.376 |

Cuadro 7.3: Tasa de compresión en BPD de los algoritmos JPEG-LS, WebP, FLIF, LBB y RC para los conjuntos de validación de CLIC.pro, CLIC.mobile, Open Images, DIV2K, Kodak.

En la figura 7.1, se gráfica la tasa de compresión en BPD de los algoritmos LBB (con el modelo $LBB'$ y la tasa de compresión $R_{real}$), Flif, WebP y PNG, en base al conjunto de

validación de ImageNet64, para observar su comportamiento a medida que la cantidad de imágenes a comprimir es mayor.

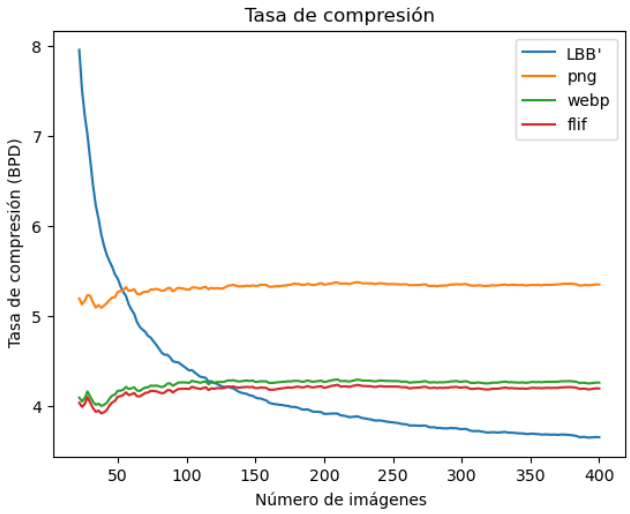



Figura 7.1: Tasa se compresión para los algoritmos $LBB'$, Flif, WebP y PNG, en función de la cantidad de imágenes.

La figura 7.1 muestra que el rendimiento de $LBB'$ es peor al resto para pocas imágenes, pero a partir de aproximadamente 125 imágenes empieza a contrarrestar el sobrecosto del stream base y supera al resto de los algoritmos.

# Capítulo 8

# Conclusiones

Podemos concluir que los objetivos de este proyecto fueron ampliamente cumplidos. Logramos investigar y reportar los aspectos principales del uso de redes neuronales para la compresión de imágenes sin pérdida. Si bien los algoritmos clásicos actuales logran buenas tasas de compresión los algoritmos que utilizan redes neuronales pueden llegar a ser mejores en algunos casos, como vimos para el caso de $LBB'$ cuando se quiere comprimir una gran cantidad de imágenes.

Si bien el uso de redes neuronales para la compresión de imágenes es un área incipiente, pudimos abordar siete algoritmos, que han sido publicados en el correr de los últimos 2 años, lo que no solo demuestra el gran interés por la investigación del uso de redes neuronales en el área de compresión sino que también confirma que las redes neuronales logran modelar las imágenes con éxito. Además estos siete algoritmos nos permitieron evaluar el uso de las distintas arquitecturas de redes neuronales para el problema en cuestión, así como también los distintos usos de algoritmos de codificación de entropía.

El algoritmo LBB tiene un problema con la redundancia de bits para empezar a ser eficiente pero como demostramos en este trabajo al combinar un algoritmo con redes neuronales y un algoritmo clásico se obtiene lo mejor de ambas partes. Podemos decir que el problema en gran parte de las soluciones propuestas con aprendizaje automático está en aproximar correctamente la distribución de la imagen para luego codificarla de la manera más eficiente posible y en este caso las arquitecturas basadas en flows demostraron ser mejores desempeñando esta tarea. Hasta donde sabemos este algoritmo es el estado del arte en la compresión de imágenes sin pérdida. Además, los algoritmos que originalmente fueron estudiados sobre conjunto

de datos de baja resolución como ImageNet64 son fácilmente adaptables para trabajar con imágenes más grandes y pueden lograr igualmente un buen desempeño.

Un problema presente en este tipo de algoritmos con redes, contrario a los clásicos, es que el tiempo de procesamiento puede llegar a ser grande. Por ejemplo, en el caso de, $LBB\_PNG$, para comprimir correctamente una imagen compuesta por 96 bloques de imágenes $64 \times 64$ se tarda siete minutos en el procesamiento utilizando una GPU RTX 2060, mientras que en algoritmos clásicos esto se tarda unos pocos segundos aunque la compresión sea peor. Para mejorar esto ya existen trabajos como por ejemplo DIET-GPU [18] propuesto por facebook que implementa la codificación rANS en GPU, podría ser una mejora a futuro ya que reduce un poco el overhead de no trabajar completamente con la GPU.

Como vimos anteriormente, este algoritmo es eficiente para comprimir grandes cantidades de imágenes, lo cual nos lleva a pensar que su uso puede ser eficiente para almacenar imágenes en la nube reduciendo no solo el ancho de banda al transferirlo sino también su almacenamiento. También podría ser interesante estudiar el desempeño del algoritmo obtenido contra los algoritmos de compresión de vídeo actuales para intentar reducir el almacenamiento necesario, algo muy usado en plataformas de streaming por ejemplo.

Esta línea de investigación tiene muchos puntos para seguir explorando dado que hoy en día el problema de comprimir imágenes se sigue expandiendo, con el manejo de resoluciones más grande y de enormes cantidades de datos como los que se extraen en aplicaciones para el estudio de la población o problemas puntuales en los distintos mercados, creemos que esto es un gran avance en este punto y se puede usar como base para seguir mejorando.

# Anexo

La estructura del repositorio se compone de un repositorio principal que contiene varios submódulo con cada algoritmo estudiado y modificado. En particular el submódulo de LBB contiene todo lo relacionado al algoritmo implementado, dentro de la carpeta *lbb/localbitsback*. Para la compresión y descompresión tenemos un script de prueba que lee una imagen, la recorta en crops de $64 \times 64$ y escribe a disco la compresión correspondiente, también está tiene la posibilidad de recibir un conjunto de imágenes y comprimir todas para luego escribir a disco la estructura descrita en la sección 6.6.

Para ejecutar la prueba de compresión y descompresión con una sola imagen se utiliza el siguiente comando (estando en la carpeta *lbb/localbitsback*):

```
PYTHONPATH=./:compression/ans/build/ python scripts/lbb.py --
   imagenet64_data_path /clusteruy/home03/compresion_imgRN/
   example.png
```

También se puede especificar un pickle con un conjunto de datos en lugar de una sola imagen

```
PYTHONPATH=./:compression/ans/build/ python scripts/lbb.py --
   imagenet64_data_path /clusteruy/home03/compresion_imgRN/val/
   val_64x64.npy
```

En el caso de querer entrenar un modelo para este algoritmo se puede usar el siguiente comando,, especificando el dataset de entrenamiento en el parámetro *imagenet64_data_path* y opcionalmente cambiando los hiperparámetros como los que usamos en este ejemplo:

```
PYTHONPATH=./:compression/ans/build/ python scripts/train.py --
   imagenet64_data_path /clusteruy/home03/compresion_imgRN/
   imagenet64/train/train_64x64.npy -lr 0.0001 --step_size
   1000000 --gamma 0.5 --batch_size 4 --snap_images 10000
```

En caso de querer ejecutar un trabajo en Cluster UY es conveniente definir un batch de ejecución para poder lanzar el entrenamiento varias veces o poder modificar fácilmente los parámetros del trabajo que se lanza en el cluster. Para ello dejamos de ejemplo un batch de ejecución para el entrenamiento de LBB

```
#!/bin/bash
#SBATCH --job-name=lbb_imagenet64
#SBATCH --ntasks=1
#SBATCH --mem=32G
#SBATCH --time=6:00:00
#SBATCH --partition=normal
#SBATCH --qos=gpu
#SBATCH --gres=gpu:0
#SBATCH --mail-type=ALL
#SBATCH --mail-user=user@mail
#SBATCH --output=%x_%j.out

conda activate lbb
cd /home/ubuntu/proyecto_grado/lbb/localbitsback

PYTHONPATH=./:compression/ans/build/ python scripts/
   run_compression_custom.py --mode test --input /clusteruy/
   home03/compresion_imgRN/mobile_valid_cropped/0067.png --
   dataset imagenet64 --single_image --test_output_filename /
   home/ubuntu/lbb_output.json
```

# Bibliografía


[1] David A. «A Method for the Construction of Minimum-Redundancy Codes». En: *Proceedings of the IRE* 40.9 (1952), págs. 1098-1101. DOI: 10.1109/JRPROC.1952.273898.

[2] Johannes Ballé y col. *Variational Image Compression with a Hyperprior*. 2018. arXiv: 1802.01436 [cs.LG].

[3] Fabrice Bellard. *BGP Image Format*. 2014.

[4] Rianne van den Berg y col. *IDF++: Analyzing and Improving Integer Discrete Flows For Loseless Compression*. 2020. arXiv: 2006.12459 [cs.LG].

[5] Zhengxue Cheng y col. *Learned Lossless Image Compression with a HyperPrior and Discretized Gaussian Mixture Likelihoods*. 2020. arXiv: 2002.01657 [cs.LG].

[6] Thomas M. Cover y Joy A. Thomas. «Data Compression». En: *Elements of Information Theory*. 2005. Cap. 5, págs. 103-158.

[7] Laurent Dinh, David Krueger y Yoshua Bengio. *NICE: Non-linear Independent Components Estimation*. 2015. arXiv: 1410.8516 [cs.LG].

[8] Jarek Duda. *Asymmetric numeral systems*. 2009. arXiv: 0902.0271 [cs.IT].

[9] Agustsson Eirikur y Timofte Radu. *NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study*. 2017.

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

[11] Google. *Challenge on Learned Image Compression*. http://compression.cc/.

[12] Jonathan Ho, Evan Lohn y Pieter Abbeel. *Compression with Flows via Local Bits-Back Coding*. 2019. arXiv: 1905.08500 [cs.LG].

[13] Emiel Hoogeboom y col. *Integer Discrete Flows and Lossless Compression*. 2019. arXiv: 1905.07376 [cs.LG].

[14] P.G. Howard y J.S. Vitter. «Arithmetic coding for data compression». En: *Proceedings of the IEEE* 82.6 (1994), págs. 857-865. DOI: 10.1109/5.286189.

[15] *Information technology - Digital compression and coding of continuoustone still images - Requirements and guidelines.* ISO/IEC 10918-1 ITU Recommend. T.81. 1992.

[16] *Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification.* ISO/IEC 15948. 2004.

[17] Sergey Ioffe y Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[18] Jeff Johnson. *DietGPU: GPU-based lossless compression for numerical data*. 2022.

[19] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. 2009.

[20] S. Kullback y R. A. Leibler. «On Information and Sufficiency». En: *The Annals of Mathematical Statistics* 22.1 (1951), págs. 79-86.

[21] Alina Kuznetsova y col. «The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale». En: *arXiv:1811.00982* (2018).

[22] M. Li y col. *Learning Convolutional Networks for Content-weighted Image Compression*. 2017. arXiv: 1703.10553 [cs.LG].

[23] Google LLC. *An image format for the Web*. 2022.

[24] Google LLC. *WebP Lossless Bitstream Specification*. 2022.

[25] D. Marpe, H. Schwarz y T. Wiegand. «Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard». En: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003), págs. 620-636.

[26] B. McMillan. «Two inequalities implied by unique decipherability». En: *IRE Transactions on Information Theory* 2.4 (1956), págs. 115-116. DOI: 10.1109/TIT.1956.1056818.

[27] Fabian Mentzer, Luc Van Gool y Michael Tschannen. *Learning Better Lossless Compression Using Lossy Compression*. 2020. arXiv: 2003.10184 [cs.LG].

[28] Fabian Mentzer y col. «Conditional Probability Models for Deep Image Compression». En: (2018). arXiv: 1801.04260 [cs.LG].

[29] Fabian Mentzer y col. *Practical Full Resolution Learned Lossless Image Compression*. 2018. arXiv: 1811.12817 [cs.LG].

[30] Pablo Musé. *Aprendizaje Profundo para Visión Artificial*. 2021.

[31] Vinod Nair y Geoffrey E. Hinton. *Rectified Linear Units Improve Restricted Boltzmann Machines*. 2010.

[32] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[33] Oinakima. *Understanding LSTM Networks*. 2015.

[34] Richard Clark Pasco. «Source Coding Algorithms for Fast Data Compression.» Tesis doct. 1976.

[35] Matt Poyser, Amir Atapour-Abarghouei y Toby P. Breckon. *On the Impact of Lossy Image and Video Compression on the Performance of Deep Convolutional Neural Network Architectures*. 2020. arXiv: 2007.14314 [cs.CV].

[36] Prajit Ramachandran, Barret Zoph y Quoc V. Le. *Searching for Activation Functions*. 2017. arXiv: 1710.05941v2 [cs.NE].

[37] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[38] Olga Russakovsky y col. *ImageNet Large Scale Visual Recognition Challenge*. 2015.

[39] Sumit Saha. *The American Statistician*. 2018.

[40] Imran us Salam. *Autoencoders — Guide and Code in TensorFlow 2.0*. 2019.

[41] Tim Salimans y col. «PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications». En: (2017). arXiv: 1701.05517 [cs.LG].

[42] Gadiel Seroussi. *Applications of Information Theory in Image Processing*. 2021.

[43] Alex Sherstinsky. *Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network*. 2020. arXiv: 2109.14216 [cs.LG].

[44] Jon Sneyers y Pieter Wuille. *FLIF: Free lossless image format based on MANIAC compression*. 2016. DOI: 10.1109/ICIP.2016.7532320.

[45] Lucas Theis y col. *Lossy Image Compression with Compressive Autoencoders*. 2017. arXiv: 1703.00395 [stat.ML]].

[46] George Toderici y col. *Variable Rate Image Compression with Recurrent Neural Networks*. 2015. arXiv: 1511.06085 [cs.CV]].

[47] George Toderici y col. *Workshop and Challenge on Learned Image Compression (CLIC2020)*. CVPR, 2020.

[48] James Townsend, Tom Bird y David Barber. *Practical lossles compression with latent variables using bits back coding*. 2019. arXiv: 1901.04866 [cs.LG].

[49] James Townsend y col. *HiLLoC: Lossless Image Compression with Hierarchical Latent Variable Models*. 2019. arXiv: 1912.09953 [cs.LG].

[50] Christian Bakke Vennerød, Adrian Kjærran y Erling Stray Bugge. *Long Short-term Memory RNN*. 2021. arXiv: 2105.06756 [cs.LG].

[51] M.J. Weinberger, G. Seroussi y G. Sapiro. «The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS». En: *IEEE Transactions on Image Processing* 9.8 (2000), págs. 1309-1324. DOI: 10.1109/83.855427.

[52] Lilian Weng. *Flow-based Deep Generative Models*. 2018.

[53] Mingtian Zhang y col. *Flow Based Models For Manifold Data*. 2018. arXiv: 2109.14216 [cs.LG].

[54] Tong Zhang. «Solving large scale linear prediction problems using stochastic gradient descent algorithms». En: 2004. DOI: 10.1145/1015330.1015332.

[55] Jacob Ziv y Abraham Lempel. «A Universal Algorithm for Sequential Data Compression». En: *IEEE Transactions on Information Theory* 23.3 (mayo de 1977), págs. 337-343.