Proyecto de Grado

Curso 2020

Redes neuronales de aprendizaje profundo para compresión de imágenes

Facultad de Ingeniería, UDELAR

Autores:
Gabriela Rodríguez - 4.893.011-0
Crhistyan Silva - 5.171.504-2

Docentes:
Álvaro Martín
Gadiel Seroussi

Diciembre, 2020

Índice

1. Introducción

La compresión de imágenes sin pérdida se centra en aprovechar las redundancias en las imágenes para ahorrar bits, y tal como su nombre lo dice no se pierde información en el proceso. Además, dicho proceso es reversible.

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 la distribución.

En este trabajo se presentarán los distintos enfoques existentes para abordar dicha temática así como también una comparación entre ellos.

2. Conceptos

En esta sección se introducen conceptos claves para la correcta comprensión de los algoritmos a presentar.

2.1. Compresión de imágenes

La compresión de imágenes, y de datos en general, es el proceso de codificar la información usando un menor número de bits que la representación inicial. La compresión sin pérdida reduce los bits al identificar y eliminar la redundancia, y tal como su nombre lo dice no se pierde información en el proceso. Además, dicho proceso es reversible.

2.2. BPP y BPSP

En algunos de los trabajos se toman los Bit Bits Pixel (BPP) como medida de evaluación para las imágenes comprimidas, es una medida del promedio de bits necesarios para representar una imagen. Dada una imagen, en donde N es la cantidad de píxeles de la imagen original y S el tamaño en bits de la imagen luego de ser comprimida, los BPP se calculan como:

$$BPP = \frac{S}{N}$$

En otros casos se toma como medida los Bits Per Sub Pixel (BPSP), que también se suele llamar Bits Per Dimension (BPD), esta se calcula de manera similar pero en este caso la fórmula es $BPSP = \frac{S}{NC}$ donde C es la cantidad de canales de la imagen. En este caso se obtiene el promedio de bits necesarios para representar un canal de un píxel.

2.3. Algoritmos de codificación de entropía

Para codificar una compresión s en un *bitstream*, se utilizan los algoritmos de codificación de entropía. La idea de estos algoritmos es utilizar un modelo de distribución de probabilidad p para codificar símbolos de manera eficiente, ya que p permite que el código asigne secuencias de bits más cortas a símbolos más probables.

A continuación se presentan dos algoritmos de codificación de entropía.

2.3.1. Codificación de Huffman

La codificación es un código de prefijo (ninguna palabra del código es prefijo de otra) y de longitud variable que se construye con el fin de minimizar la longitud de código promedio. Es decir, dado un alfabeto $A = a_1, \ldots, a_n$ ponderados por $W = w_1, \ldots, w_n$ el objetivo es encontrar un código $C(A, W) = (c_1, \ldots, c_n)$ con longitud de código:

$$L(C) = \sum_{i=1}^{n} w_i.longitud(c_i)$$

En donde se cumpla que $L(C) < L(T)$ para cualquier código $T(A, W)$.
El algoritmo de construcción del árbol puede resumirse así:

1. Crear un nodo hoja para cada símbolo, asociando un peso según su frecuencia de aparición e insertarlo en la lista ordenada ascendentemente.

2. Mientras haya más de un nodo en la lista:

   a) Eliminar los dos nodos con menor probabilidad de la lista.

   b) Crear un nuevo nodo interno que enlace a los nodos anteriores, asignándole como peso la suma de los pesos de los nodos hijos.

   c) Insertar el nuevo nodo en la lista, (en el lugar que le corresponda según el peso).

3. El nodo que quede es el nodo raíz del árbol.

2.3.2. Codificación aritmética

La Codificación Aritmética (Arithmetic Coding, AC) [1] se fundamenta en el conocimiento *a priori* de la distribución de probabilidades de los símbolos. Esta codificación se basa en la codificación de una secuencia de símbolos de una manera conjunta, es decir, no asocia una codificación específica a cada símbolo individual; por el contrario, se codifica

la secuencia entera en una fracción n dónde $0 \leq n < 1$.

La función de codificación toma un símbolo y un intervalo de frecuencia y da como resultado un nuevo intervalo. Para representar la secuencia de símbolos suele elegirse el límite inferior del intervalo obtenido en la codificación. Mientras que la función de decodificación toma un valor representado como un número fraccionario y los intervalos de frecuencia, calculados en la codificación, y decodifica un símbolo.

Sea $S = (s_1, s_2, ..., s_n)$ la cadena de símbolos de entrada de un alfabeto finito $A = \{a_1, a_2, ..., a_k\}$ de tamaño k. Se parte de una colección de intervalos I_i entre $[0, 1)$, cada uno de los cuales estará asociado a cada uno de los posibles símbolos, con una magnitud proporcional a la probabilidad de símbolo. Esta asignación entre cada símbolo y su intervalo $\{s_i, I_i\}$ se construye de tal forma que $\bigcup_{\forall i} I_i \equiv [0, 1)$ y $\bigcap_{\forall i} I_i \equiv \emptyset$.

Se define $I_i = [x_i, y_i)$, y se parte de un intervalo inicial $[x_{actual}, y_{actual}) = [0, 1)$. Luego, para cada símbolo a codificar, se modifica de la siguiente forma:

$$s_i \Rightarrow [x_{nuevo}, y_{nuevo}) = [x_{actual} + (y_{actual} - x_{actual}) \cdot x_i, x_{actual} + (y_{actual} - x_{actual}) \cdot y_i),$$
$$[x_{actual}, y_{actual}) = [x_{nuevo}, y_{nuevo})$$

El valor final del límite inferior del intervalo $[x_{actual}, y_{actual})$ es suficiente para la recuperación de la secuencia de símbolos.

2.3.3. Sistemas Numéricos Asimétricos

Los Sistemas Numéricos Asimétricos (Asymmetric Numeral Systems, ANS)[6] son una familia de métodos de codificación de entropía. Estos métodos logran ratios de compresión comparables con AC y a su vez velocidades similares a las de Huffman coding[8].

Particularmente, se presentará el método range-ANS (rANS) perteneciente a la familia ANS. Cabe destacar, que este nombre surge debido a su particular similitud con AC.

La idea principal es tomar una secuencia de símbolos y codificarla como un único número natural x.

La función de codificación toma un estado y un símbolo y los codifica en un nuevo número natural. La función de decodificación toma un estado y decodifica un símbolo, mientras produce un nuevo estado, del cual se pueden extraer nuevos símbolos.

Sea $S = (s_1, s_2, ..., s_n)$ la cadena de símbolos de entrada de un alfabeto finito $A = \{a_1, a_2, ..., a_k\}$ de tamaño k. Se asume que la información viene de una distribución caracterizada por la frecuencia de los símbolos, $F = \{F_{a_1}, F_{a_2}, ..., F_{a_k}\}$, se define $M = \sum_{i=1}^{k} F_i$ y la frecuencia acumulada $C_{a_i} = \sum_{j=1}^{i-1} F_{a_j}$.

Finalmente, se define X_t, el estado luego de observar t símbolos, como:

$$X_t = \left\lfloor \frac{X_{t-1}}{F_{s_t}} \right\rfloor \cdot M + C_{s_t} + \mod (X_t, F_{s_t})$$

rANS realiza un seguimiento de la entrada utilizando un solo estado entero, X_t. Como se mencionó anteriormente este método actualiza el estado X_t basado en el estado anterior y en el símbolo actual s_t. La salida es el estado final X_n, representado usando $\lceil \log_2 X_n \rceil$.

X_n junto con la cantidad de símbolo codificados n, se utiliza para decodificar toda la cadena de símbolos inicial.

2.4. Herramientas en redes neuronales

Debido a la gran variación de datos y al problema que puede suponer inicializar los datos de manera incorrecta existen varias técnicas que se aplican en todas las arquitecturas para mejorar el entrenamiento de la red.

2.4.1. Batch Normalization

Batch normalization es una técnica para entrenar redes neuronales que estandariza las entradas a una capa para cada *mini-batch*. Esto tiene el efecto de estabilizar el proceso de aprendizaje y reducir drásticamente el número de *epochs* de entrenamiento necesarias para entrenar las redes.

2.4.2. Dropout

Para entender el dropoout se puede ver su funcinamiento durante la fase de entrenamiento y durante la fase de test.

La idea es "apagar" algunas neuronas en el momento del entrenamiento para hacer que las redes sean diferentes en cada iteración de entrenamiento. Para esto se multiplica cada entrada y_i con una neurona r_i que se construye a partir de una distribución de dos puntos que genera 0 o 1 con distribución de Bernoulli. Sin dropout el paso hacia adelante en la etapa de entrenamiento es:

$$z_i^{(l+1)} = w_i^{(l+1)} y^{(l)} + b_i^{(l+1)} \tag{1}$$

$$y_i^{(l+1)} = f(z_i^{(l+1)}) \tag{2}$$

Con dropout se convierte en:

$$r_i^{(l)} \sim \text{Bernoulli}(p) \tag{3}$$
$$\hat{y}^{(l)} = r^{(l)} y^{(l)} \tag{4}$$
$$z_i^{(l+1)} = w_i^{(l+1)} \hat{y}^{(l)} + b_i^{(l+1)} \tag{5}$$
$$y_i^{(l+1)} = f(z_i^{(l+1)}) \tag{6}$$

2.4.3. Funciones de activación

Una función de activación en una red neuronal define cómo la suma ponderada de la entrada se transforma en una salida de un nodo o varios nodos en una capa de la red.

Rectified Linear Activation (ReLU)

Es común porque es simple de implementar y efectivo para superar las limitaciones de otras funciones de activación previamente populares, como Sigmoid y Tanh. En concreto, es menos susceptible a la desaparición de gradientes que impiden el entrenamiento de modelos profundos, aunque puede sufrir otros problemas como unidades saturadas o "muertas".

La función ReLU se calcula de la siguente manera: $f(x) = \max(0, x)$. Es decir, si el valor es negativo devuelve 0 y sino lo deja como está.

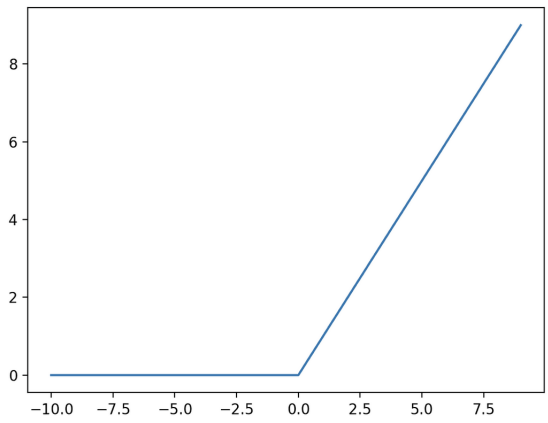


Figura 1: Función de activación relu

2.4.4. Logistic (Sigmoid)

La función de activación sigmoidea también se denomina función logística. Es la misma función utilizada en el algoritmo de clasificación de regresión logística.

La función toma cualquier valor real como valores de entrada y de salida en el rango de 0 a 1. Cuanto mayor sea la entrada, más cercano estará el valor de salida a 1.0, mientras que cuanto menor sea la entrada, más cerca estará la salida de 0.0.

La función de activación sigmoidea se calcula de la siguiente manera: $f(x) = \frac{1,0}{1,0+e^-x}$

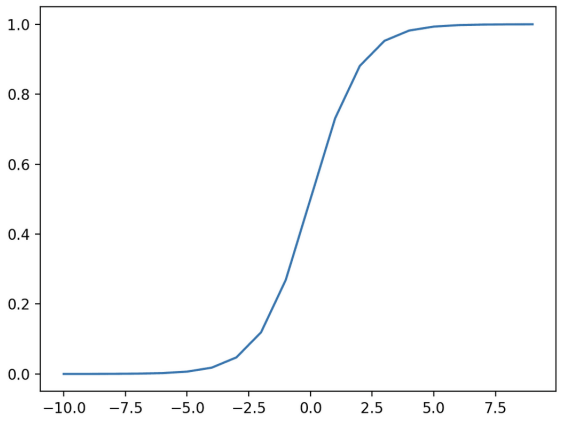


Figura 2: Función de activación sigmoide

2.4.5. Hyperbolic Tangent (Tanh)

La función de activación de la tangente hiperbólica también se denomina simplemente función Tanh (también "tanhz "TanH").

Es muy similar a la función de activación sigmoidea e incluso tiene la misma forma.

La función toma cualquier valor real como entrada y valores de salida en el rango de -1 a 1. Cuanto mayor sea la entrada (más positiva), más cerca estará el valor de salida a 1.0, mientras que cuanto más pequeña sea la entrada (más negativa), más cerca la salida será -1.0.

Esta función de activación se calcula de la siguiente manera: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

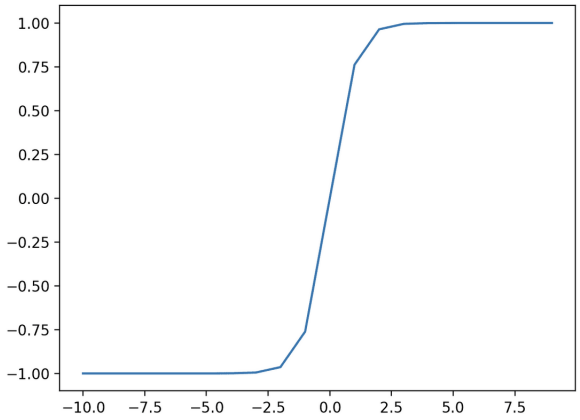


Figura 3: Función de activación tanh

2.5. Arquitecturas de redes neuronales

2.5.1. Redes convolucionales

Una red neuronal convolucional (Convolutional Neural Networks, CNN) típicamente consiste en tres capas. Una capa convolucional, una capa de *pooling* y una capa de activación.[3]

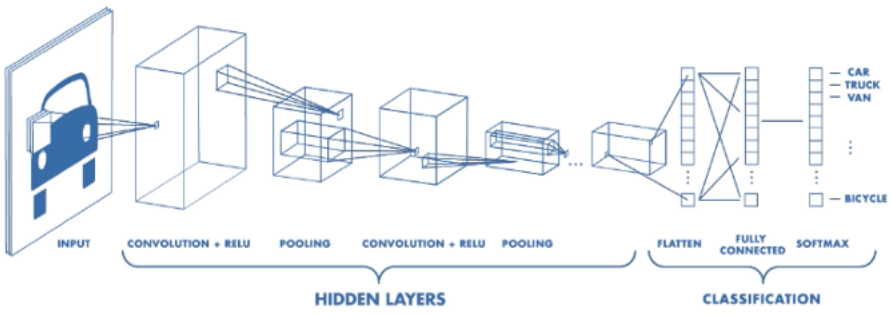


Figura 4: Representación de un autoencoder

La primer primer aplica convoluciones, operaciones entre un píxel y sus vecinos dando, básicamente es un multiplicación de matrices en donde una matriz son parámetros para aprender y la otra es una porción de la imagen de entrada. La segunda capa reduce la dimensionalidad tomando cierta estadística de los valores vecinos, por ejemplo el máximo de cada rectángulo de 2x2, esto reduce en gran medida el costo computacional. La última capa es útil para introducir un aspecto no lineal en el aprendizaje, ya que las convoluciones

son operaciones lineales, esta puede ser cualquiera de las activaciones conocidas, Sigmoide, Tanh, ReLU, etc.

2.5.2. Modelos Generativos

Un Modelo Generativo modela la distribución conjunta $P(X, Y)$ de los datos X con objetivo Y. Al contrario de los modelos discriminativos que buscan la distribución condicional $P(X|Y)$.

Autoencoders

Los autoencoders son un tipo específico de redes neuronales feed-forward en el que la entrada es la misma que la salida. Comprimen la entrada en una representación de menor dimensión, llamada "código latente" y luego reconstruyen la salida a partir de esta representación. [5]

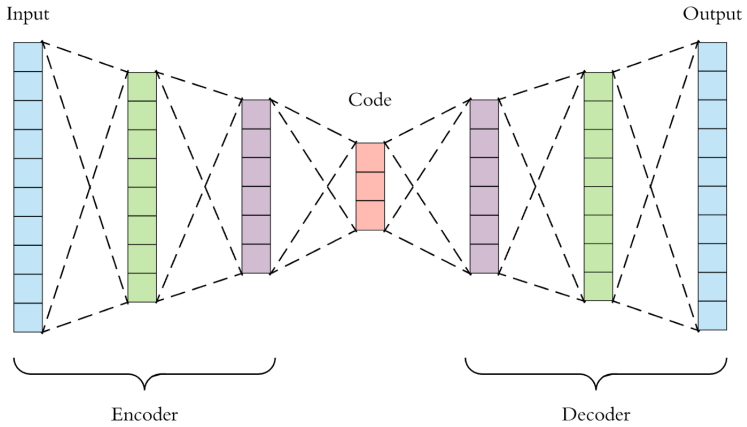


Figura 5: Representación de un autoencoder

Como se ve en la Figura 2 esta arquitectura se compone de dos redes "conectadas" por el código latente, estas redes pueden ser de cualquier tipo mientras se mantenga que la dimensión de entrada y salida de toda la red sea la misma.

Variational Autoencoders (VAE)

Un Variational Autoencoder [14] se puede definir como un autoencoder cuyo entrenamiento se regulariza para evitar el sobreajuste y asegurar que el espacio latente tenga buenas propiedades que permitan el proceso generativo. En lugar de codificar una entrada como un solo punto, se codifica como una distribución sobre el espacio latente

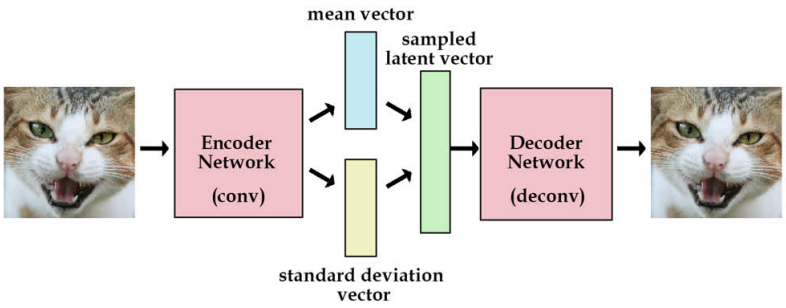


Figura 6: Representación de un VAE

Flow-based Generative Models

Los Flow-based Generative Models se construyen mediante una secuencia de transformaciones invertibles. El modelo aprende explícitamente la distribución de datos $p(x)$ y, por lo tanto, la función de pérdida es simplemente la probabilidad logarítmica negativa.

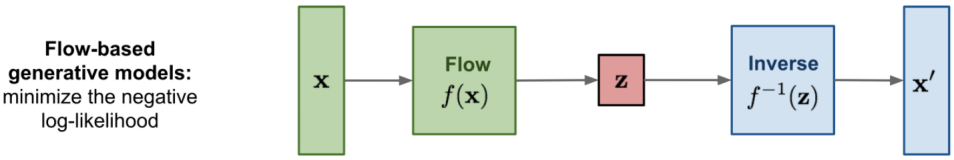


Figura 7: Representación de un Flow-based Generative Model

2.5.3. Redes recurrentes

Las redes neuronales convencionales no manejan el concepto de persistencia. Este tipo de redes puede solucionar esto. Una red neuronal recurrente (Recurrent Neural Networks, RNN) puede considerarse como varias copias de la misma red, cada una pasando un mensaje a un sucesor.

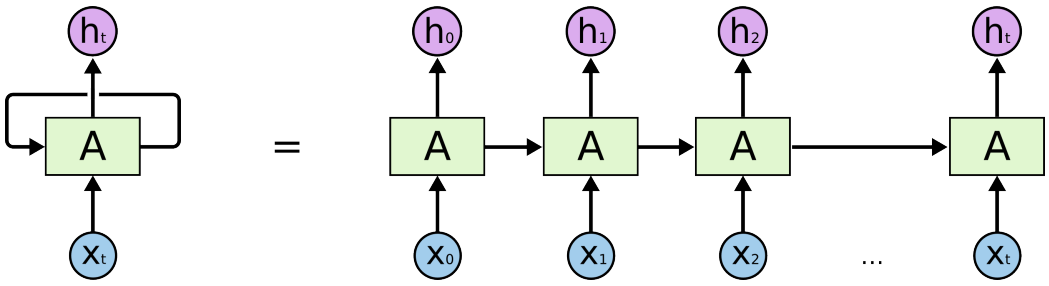


Figura 8: RNN desenrollada

Si bien la idea de tener un contexto pasado que se pueda usar entender o predecir los datos presentes, en la práctica las RNN pueden presentan problemas. Estas redes son buenas prediciendo a partir de un contexto cercano para la tarea que se está desarrollando, pero en la práctica presenta problemas ya que no es capaz de capturar contexto demasiados lejanos, la brecha entre que un contexto se analiza y entre que se necesita puede ser demasiado grande. Una mejora de esto son las redes Long Short-Term Memory (LSTM). [13]

Long Short-Term Memory

Este tipo espacial de RNN se usan para resolver el problema de dependencias a largo plazo. La siguiente imagen muestra un caso simple de un a red neuronal recurrente de una capa:

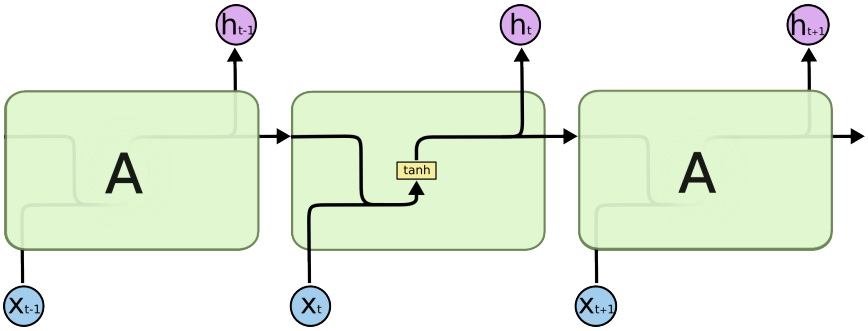


Figura 9: Cadena de una RNN

Para el caso de las LSTM en lugar de estos núcleos en cadena. Pero en lugar de una capa son cuatro que interactúan entre si.

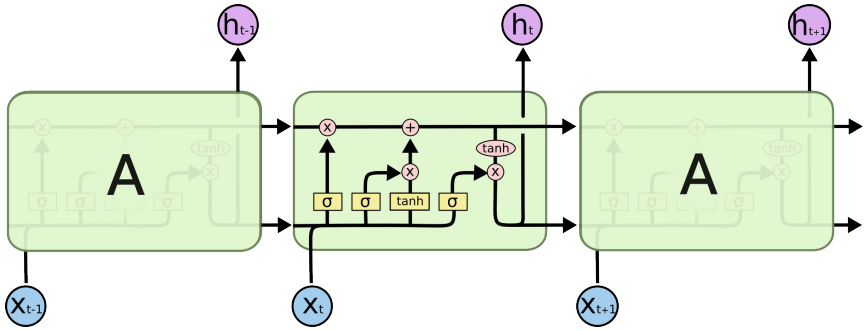


Figura 10: Cadena de una LSTM

2.6. Normalizing flows

Los *Normalizing Flow*[12] usados en los modelos generativos son una aplicación glorificada del cambio de variable. Esto es, dado una variable aleatoria X con densidad $p_X(x)$ y una transformación $Z = f(X)$, X y Z se relacionan de la siguiente manera:

$$p_X(x) = p_Z(f(x))|detDf(x)|$$

en donde:

- $f(X)$ es una función invertible y diferenciable. A la que se le llama *flow*.
- $Df(x)$ es el jacobiano de $f(x)$.

Entonces el objetivo de los normalizing flow es aprender la transformación $f(x)$ para llevar la distribución de datos $p_X(x)$ a $p_Z(z)$. Estas son las dos piezas importantes, la medida base $p_Z(z)$ que usualmente se elige una distribución como $N(z|0, I)$ y la función $f(x)$ que debe ser invertible y diferenciable para poder aplicar el cambio de variable.

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

$$max \sum_{i=1}^{N} \log p_Z(f(x_i|\theta)) + \log |detDf(x_i|\theta)|$$

Donde θ son los parámetros del flow $f(x|\theta)$.

Para lograr esta transformación que potencialmente puede ser compleja se explota el hecho de que las funciones invertibles y diferenciables son cerradas bajo la composición, por lo que el flow f puede construirse a partir de k flows individuales:

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

Linear flows

La primer transformación invertible y diferenciable es una transformación lineal, dada una matriz $A \in \mathbb{R}_{n\times n}$ y un vector $b \in \mathbb{R}^n$ la función f:

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

En este caso, la matriz A debe ser invertible, para que la transformación pueda ser invertible se debe cumplir $f(z)^{-1} = A^{-1} \cdot z - b$ y además $det(Df(x)) = det(A)$. 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 mas complejo que una función lineal, y además el costo de calcular el determinante o la inversa puede ser $O(N^3)$. Por supuesto que para matrices particulares, como diagonales o triangulares, el costo es menor pero el problema de 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 primer parte se deja como está y a la segunda se le aplica otro flow con parámetros dependientes de x^A:

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

O visto de manera gráfica:

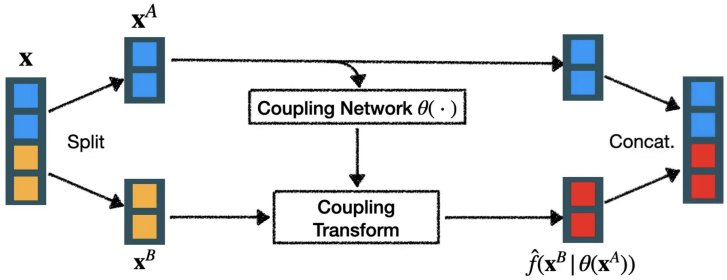


Figura 11: Coupling transform [12]

Para la inversa simplemente se aplican las operaciones en dirección contraria. Una ventaja de estos flows es que si bien el Jacobiano tiene la siguiente forma:

$$\begin{bmatrix} I & 0 \\ \frac{\partial}{\partial_{X^A}} \hat{f}(x^B|\Theta(x^B)) & D\hat{f}(x^B|\theta(x^A)) \end{bmatrix}$$

para la aplicación de los flows se necesita el determinante, no el jacobiano, y en este caso el determinante de toda esa matriz se reduce en $det(D\hat{f}(x^B|\theta(x^A)))$.

Para la aplicación de los flows el calculo de los parámetros $\theta(x^A)$ puede ser tan complejo como se quiera, puede ser una red CNN, RNN, etc; ya que no se necesita su inversa ni el jacobiano una vez computado. Además, es importante cambiar los splits de x^A y x^B para mejorar el entrenamiento, esto se logra haciendo permutaciones sobre la entrada que es básicamente otro flow aplicado al vector de entrada.

Luego, para esa transformación intermedia $\hat{f}$ se tienen varias opciones, los primeros trabajos con normalizing flows [NICE, Dinh et al 2014] proponen una adición $\hat{f}(x|t) = x+t$ pero quedan muy limitados aunque sean simples de aplicar. Una trasnformación mas comúnmente usada es una transformación affin [Real NVP, Dinh et al 2016] $\hat{f}(x|s,t) = s \bigodot x+t$. Trabajos mas recientes proponen operaciones mas complejas como perceptrones multi capa aplicados elemento a elemento [NAF, Huang et al, 2018], mixture de logs CDFs [Flow++, Ho et al, 2019], y el uso de splines [Spline Flow, Durkan et al, 2019]. En 2020 se propone Hierarchical Invertible Neural Transport (HINT) que es una aplicación recursiva de los coupling flows.

Autoregresive models en combinación con flows

Los modelos autoregresivos puede combinarse con los flows, por ejemplo al tomar $p(x_i|x< i) = N(x_i|\mu(x_{<i}), \sigma(x_{<i}))$ y tomando la reparametrización $x_i = \mu(x_{<i})|\sigma(x_{<i})z_i$ donde $z_i \sim N(0,1)$. Entonces cada función f_i es de la forma:

$$f_i(x) = \frac{x_i - \mu(x_{<i})}{\sigma(x_{<i})}$$

Con inversa $f_i^{-1} = \mu(f^{-1}) + \sigma(f^{-1})$ y el determinante del jacobiano es el producto de la inversa de los sigmas de cada función.

2.6.1. Decuantificación

En el caso de aplicación de normalizing flows sobre imágenes aparece el problema de usar el espacio discreto de los píxeles en un modelo que trabaja sobre un espacio continuo. Para resolver esto se usa la decuantificación, que consiste en expresar la distribución de los píxeles $P_Y(y)$ de la siguiente manera:

$$P_Y(y) = \int_{[0,1]^D} p_X(y+u)p_U(u)du$$

Un caso simple puede ser tomar p_U como una uniforme, en 2019 Flow++[7] propone la variante llamada "Variational Dequantization" mirando p_U como una distribución variacional.

3. Algoritmos de compresión clásicos

3.1. LOCO-I/JPEG-LS [4][15]

LOCO-I (LOw COmplexity LOssless COmpression for Images) es un algoritmo para la compresión sin pérdidas y casi sin pérdidas de imágenes de tono continuo. En este al-

goritmo tanto el codificador como el decodificador funcionan de forma sincronizada; y los píxeles de la imagen se recorren de izquierda a derecha y de arriba hacia abajo. Se utiliza un modelo predictivo para la codificación de cada píxel, basado en los píxeles vecinos causales más cercanos, los cuales ya fueron codificados anteriormente, y se los denomina "contexto".

El modelo de este estándar se compone fundamentalmente de tres componentes: bloque de determinación del contexto, bloque de predicción del píxel y error de predicción. El bloque de predicción se compone de un modelo estático de predicción y un componente adaptativo basado en su contexto. Una vez que se conoce la predicción del píxel, el codificador computa a través de un modelo probabilístico el error de predicción como la diferencia entre el valor del píxel y su predicción condicionada por el contexto. Finalmente, para la codificación de la salida final, existen dos modos de operar en función de las operaciones anteriores: modo regular (Golomb Code) o en modo carrera, también conocido como longitud de carrera (Run Length).

La figura 12 muestra el modelo y los principales componentes del estándar LOCO-I.

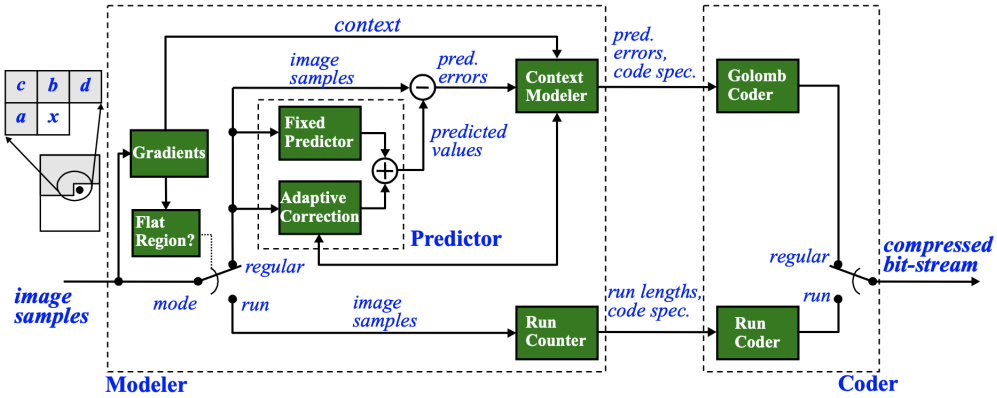


Figura 12: JPEG-LS (LOCO-I Algorithm): Block Diagram

3.1.1. Bloque de predicción

El bloque de predicción del píxel utiliza un predictor fijo que explota la propiedad de vecindad de cuatro 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 se limita a un término aditivo entero, que es dependiente del contexto.

Predictor fijo

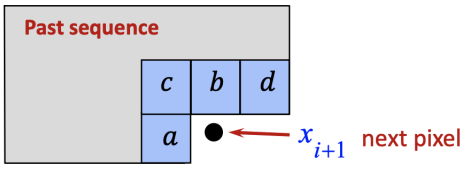


Figura 13

Según se ilustra en la figura, la predicción de un píxel x se basa en el análisis de los píxeles adyacentes a, b, c y d, los cuales determinan el contexto del píxel actual x.

El predictor cambia entre tres predictores simples:

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

Según su definición tiende a elegir b en los casos en que existe un borde vertical a la izquierda de la ubicación actual, a en los casos de un borde horizontal encima de la ubicación actual, o a + b - c si no se detecta ningún borde. La última opción sería el valor de x si la muestra actual perteneciera al "plano" definido por las tres muestras adyacentes con "alturas" a, b y c, lo cual expresa la "suavidad" esperada de la imagen ante la ausencia de bordes.

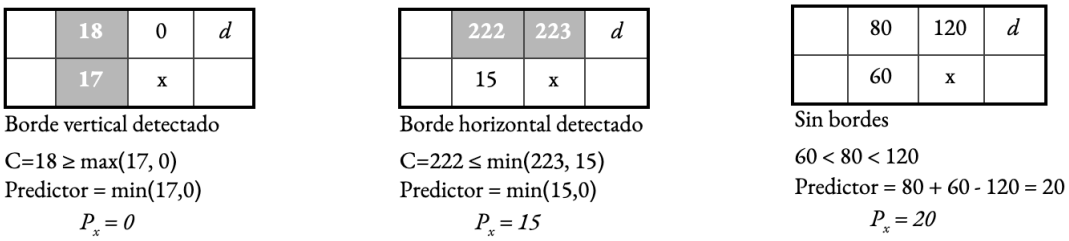


Figura 14: Ejemplos de aplicación del predictor fijo

Corrección adaptativa

El término adaptativo también se conoce como "bias cancellation" (cancelación del sesgo). El mismo es dependiente del contexto por lo que puede interpretarse como parte del bloque de determinación del contexto, lo cual se analiza más adelante, en la sección 3.1.1.

Bloque de determinación del contexto

En un esquema de modelado de contexto reducir el número de parámetros es un objetivo clave. En el modelo, este número depende del número de parámetros libres que definen la distribución de codificación en cada contexto y del número de contextos.

Distribución de codificación

Es una observación aceptada, adoptada por LOCO-I, que el residuo de un predictor fijo sobre una imagen de tonos continuos se puede modelar por una distribución del tipo Two Side Geometric Distribution (TSGD) centrada en cero. En estudios posteriores se demostró que al utilizar un offset con respecto al cero se consiguen resultados más precisos. Este offset se puede dividir en dos partes, una entera y otra fraccional.

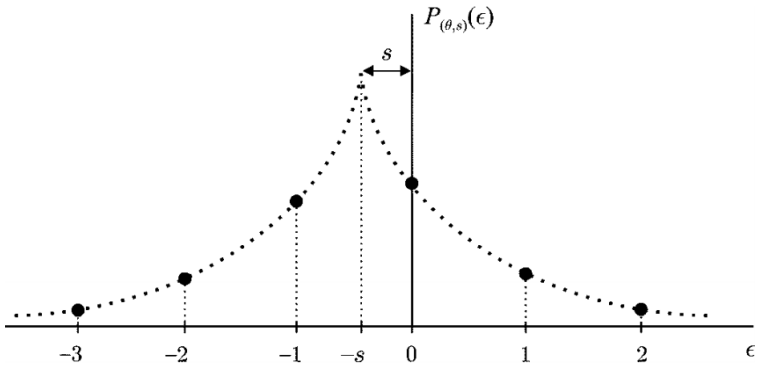


Figura 15: Distribución del tipo TSGD

Entonces, si se toma u como el offset, R como su parte entera (o "bias") y s como la parte fraccional (o "shift"), y se relacionan entre si mediante

$$u = R - s, \text{dónde } 0 \leq \text{s} < 1$$

Luego, según este modelo, la distribución de probabilidad del error para un predictor fijo en cada contexto está dada por

$$P_{(\theta,\mu)}(\epsilon) = C(\theta, s)\theta^{|\epsilon - R + s|}, \epsilon = 0, \pm 1, \pm 2, ..., \tag{7}$$

dónde $C(\theta, s) = (1 - \theta)/(\theta^{1-s} + \theta^s)$, es un factor de normalización.

El parámetro R (o "bias") requiere un término adaptativo entero en el predictor que depende del contexto, se asume que este término esta ajustado para cancelarlo. Se tiene entonces, que el modelo (1) se reduce a

$$P_{(\theta,\mu)}(\epsilon) = C(\theta, s)\theta^{|\epsilon + s|}, \epsilon = 0, \pm 1, \pm 2, ..., \tag{8}$$

dónde $0 < \theta < 1$, $0 \leq \text{s} < 1$.

Determinación del contexto

El contexto que condiciona la codificación del residuo de predicción actual se construye a partir de las siguientes diferencias:

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

Estas diferencias representan el gradiente local, los cuales capturan el nivel de actividad en torno a una muestra. Con nivel de actividad se hace referencia a qué tan cerca de un borde entre tonalidades distintas se encuentra la muestra o cómo se viene degradando una tonalidad.

Para una mayor reducción del tamaño del modelo, cada diferencia $g_i, i = 1, 2, 3$ se cuantifica en un pequeño número de regiones conectadas por un cuantificador $k()$, independiente de i, aproximadamente equiprobables. El objetivo de esta cuantificación es maximizar la información mutua entre el valor de la muestra actual y su contexto, sujeto a que cuantas más regiones se tenga mayor es la información mutua que se obtiene.

Para preservar la simetría, las regiones están indexadas $-T, ..., -1, 0, 1, ..., T$ con $k(g) = -k(-g)$ para un total de $(2T + 1)^3$ contextos diferentes. Al fusionar contextos de "signos opuestos", el número total de contextos se convierte en $((2T + 1)^3 + 1)/2$. Para LOCO-I, se seleccionó $T = 4$, resultando en 365 contextos.

Para terminar la definición de los contextos en LOCO-I, queda por especificar los límites entre las regiones de cuantificación. Para un alfabeto de 8-bit por píxel, las regiones predeterminadas de cuantificación son $0, \pm\{1, 2\}, \pm\{3, 4, 5, 6\}, \pm\{7, ..., 20\}, \pm\{e|e \geq 21\}$. Sin embargo, los límites son parámetros ajustables, a excepción de la región central que debe ser 0. En particular, una elección adecuada para un alfabeto de x-bit por píxel, colapsa las regiones de cuantificación, lo que da como resultado un menor número efectivo de contextos, con aplicaciones para la compresión de imágenes pequeñas.

Cancelación de R("bias")

Como se mencionó anteriormente, la parte adaptativa del predictor se basa en el contexto, y se utiliza para "cancelar" la parte entera (R) del offset debido al predictor fijo.

Para el cálculo del valor del bias, se realiza una estimación basada en el promedio de los errores ocurridos para las muestras previamente analizadas que pertenezcan al mismo contexto de la muestra que se está analizando.

Para realizar dicho cálculo es necesario guardar la siguiente información para cada uno de los contextos existentes:

- N = número de ocurrencias del contexto.
- B = suma acumulativa de los errores de la predicción fija.

Luego, a partir de estos dos valores se calcula el bias como:

$$R = \lceil B/N \rceil$$

Dado que realizar una división es una operación costosa en recursos y tiempo en este estándar se realiza el cálculo del bias utilizando un algoritmo que se basa solamente en sumas y restas. Se definen, como agregación a la definición de B y N, las siguientes variables:

- ϵ: error de la predicción.
- C: hace un seguimiento de $\lceil B/N \rceil$.

A continuación se presenta el algoritmo.

```
B = B + ε;
N = N + 1;
if (B ≤ -N){
  C = C - 1;
  B = B + N;
  if (B ≤ -N) B = -N + 1;
}else if (B > 0){
  C = C + 1;
  B = B - N;
  if (B > 0) B = 0;
}
```

Codificación

Como se mencionó anteriormente, el estándar ejecuta en dos modos según el contexto: modo carrera y modo regular.

Modo carrera

Cuando los gradientes locales (mencionados en la sección 3.1.1) son iguales a 0, el codificador salta el proceso de predicciones y directamente codifica la salida con un código que identifica el número de repeticiones del píxel.

Modo regular

Una vez finalizada la etapa de cálculo de la predicción se genera el residuo de la predicción (ϵ). Este valor es la resta entre el valor real del píxel y su predicción.

Luego, este valor 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 mod 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

Los códigos de Golomb son óptimos para codificar valores positivos que sigan una distribución one-side geometric (OSGD). 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 (ϵ) tiene una distribución TSGD, por lo cual es necesario realizar un mapeo, sin pérdida de información, para llevarlo a una del tipo OSGD. 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. Obteniendo un largo que cumple:

$$largo = k + 1 + \lfloor n/2^k \rfloor$$

Estos códigos son conocidos como Golomb-Rice.

Por último, para estimar k se define:

- A: Suma acumulada de las magnitudes de los errores de predicción
- N: Número de ocurrencias de un contexto

Y se usa la siguiente aproximación:

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

3.2. WebP-lossless

Este formato consiste en transformar la imagen utilizando varias técnicas diferentes. Se elige cual de estas técnicas es aplicada dependiendo del aporte a la compresión, pudiendo aplicarse todas hasta una vez o ninguna. En algunas técnicas, la imagen es dividida en bloques (usualmente de 16x16) y cada bloque de la imagen usa los mismos parámetros de transformación. Esto es seguido de una codificación entrópica en los parámetros de transformación y los datos de imagen transformados. Las transformaciones incluyen:

- Predicción espacial de píxeles
- Transformación del espacio de color con:
  - Indexación de píxeles.
  - Empaquetados de píxel.
  - Substracción de verde.

Para la codificación entrópica se usa una variante de LZ77-Huffman, el cual usa códigos 2D de distancia y valores dispersos compactos.

El formato utiliza imágenes de "subresolución", incrustadas de forma recursiva en el propio formato, para almacenar datos estadísticos sobre las imágenes, como los códigos de entropía utilizados y los datos y parámetros de una transformación. Por ejemplo, si se transforma la imagen usando predicción espacial de píxeles, los datos para revertir esta transformación se ponen como píxeles de la imagen y luego la codificación entrópica se aplica tanto sobre los datos codificados como sobre los parámetros de la transformación.

3.2.1. Transformaciones

Las transformaciones antes mencionadas son reversibles y cada una podría hacer la compresión final mas densa. En cada caso se utiliza un bit para indicar la presencia de una transformación y cada una se hace solo una vez. El codificador usa estas transformaciones para reducir la entropía de Shannon en la imagen residual (la diferencia entre la transformada y la original).

En caso de tener alguna de las 4 transformaciones, se utilizan dos bits para identificar la transformación, seguido de los datos de la misma, que son necesarios para aplicar la transformación inversa dependiendo del tipo que sea. La transformación puede ser de predicción, de color, substracción de verde o de indexación. A continuación, se describe cada uno de los datos de transformación.

Transformación de predicción

Se usa para reducir la entropía explotando el hecho de que píxeles vecinos usualmente están colacionados. Se predice un valor basado en los píxeles ya codificados y solo el valor residual (actual - predicho) es codificado. En esta transformación llamaremos "modo de predicción" a los los píxeles o combinación de píxeles usados para la predicción del valor actual a ser codificado. Estos modos se construyen a partir del contexto de un píxel.

Los tres primeros bits de los datos de predicción definen el ancho y el largo del bloque en número de bits. Suponiendo que n es el número entero que representan estos 3 bits, entonces el largo y ancho del bloque será n+2.

Los datos transformados contienen el modo de predicción, estos modos son tratados como píxeles de una imagen y se codifican usando las mismas técnicas explicadas en la sección **??**.

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

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

Los 14 modos son:

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)

Dónde:

- $Average(a, b) = \frac{a+b}{2}$
- $Select(L, T, TL) = \begin{cases} L & si \quad p_L < p_T \\ T & si \quad p_L \geq p_T \end{cases}$

  Con:

  - $p_L = ||p - L||_1$
  - $p_T = ||p - T||_1$
  - $p = L + T – TL$
- $ClampAddSubstractFull(a, b, c) = Clamp(a + b - c)$
- $ClampAddSubstractHalf(a, b) = Clamp(a + (a - b)/2)$

Para los casos de bordes se manejan ciertas reglas, estas son:

- El valor predicho del píxel de mas 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. 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, según uno de los tres posibles: green_to_red, green_to_blue, red_to_blue, que por simplicidad llamaremos δ_1, δ_2, δ_3 respectivamente.

Para esta transformación se define cada uno de los deltas, y se usan los mismos en cada píxel de un mismo bloque. Estos deltas se suma durante la transformación de color y luego se resta para la transformación inversa.

Dada una función $ColorTransformDelta(a, b) = (a * b) >> 5$, dados δ_1, δ_2, δ_3 para un bloque la Transformación de Color ($CF : \mathbb{R}^3 \to \mathbb{R}^3$) queda definida de la siguiente manera:

$$CF(R, G, B) = (R + ColorTransformDelta(\delta_1, G), G, \\ B + ColorTransformDelta(\delta_2, G) + ColorTransformDelta(\delta_3, R)) \tag{9}$$

Al igual que la anterior los tres primeros bits de los datos de la transformación de color indican el tamaño del bloque y se calculan de la misma manera. El resto de bits contiene las instancias de los deltas correspondiente a cada bloque de la imagen, todos son tratados como píxeles de una imagen y son codificados como se describe en la sección 3.2.2.

Substracción de verde

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

En caso de tener la transformación de color esta es redundante, pero puede ser útil ya que puede extender la dinámica de la transformación de color y no hay bits adicionales, por lo que se puede codificar con menos bits que una transformación de color completa.

Transformación de indexación

Esta transformación mapea las tuplas ARGB a índices de un array. El proceso consiste en contar las tuplas únicas de ARGB en la imagen, si la cantidad es menor a cierto umbral (256) crea un array con esos valores ARGB y luego reemplaza cada píxel por uno de esos valores.

Los datos de la transformación contienen el tamaño del array y las entradas de la tabla de colores, esta es almacenada en el mismo formato que la imagen y puede ser leída como una imagen sin el cabezal de RIFF **??** asumiendo alto uno y ancho el guardado en los datos de la transformación. Sobre esta imagen se usa substracción de verde para reducir la entropía. Los colores de la paleta suele tener menos entropía que los colores originales, la inversa luego se obtiene reemplazando cada píxel por su correspondiente ARGB según el índice.

Si un índice es igual o mayor al tamaño especificado en los datos de la transformación el píxel correspondiente sera negro transparente (0x00000000).

Cuando la tabla tiene 16 o menos colores, varios píxeles son empaquetados en uno solo, reduciendo el tamaño de la imagen. Estos paquetes permiten una codificación entrópica de distribución mas eficiente en los píxeles vecinos y proporciona algunos beneficios como los de la codificación aritmética.

Según el tamaño que tenga la tabla es cuantos píxeles son combinados:

- $color\_table\_size <= 2 \rightarrow width\_bits = 3$
- $2 <= color\_table\_size <= 4 \rightarrow width\_bits = 2$

- $4 <= color\_table\_size <= 16 \rightarrow width\_bits = 1$
- 0 en otro caso

En este caso *width_bits* puede tomar valores de 0 a 4. Si es 0 indica que no hay empaquetados de píxeles. Un valor de uno indica que la combinación es de dos píxeles y cada una está en el rango [0..15]. El valor 2 indica que cuatro píxeles son combinados y cada uno está en el rango [0..3] y el 3 quiere decir que se combinan ocho y los valores son binarios (0 o 1).

3.2.2. Datos de la imagen

Los datos de la imagen se representan por un array de píxeles ordenados en Scanline, se usan para 5 fines distintos:

- Imagen ARGB: guarda el píxel actual de la imagen
- Imagen de entropía: guarda el meta código de Huffman. La componente roja y verde de un bloque identifican el meta código de Huffman para ese bloque.
- Imagen de predicción: guarda la metadata de la transformación de predicción. La componente verde cual de las 14 modos de predicción se usa para un bloque en particular.
- Imagen de transformación de color: almacena los delta de la transformación de color, cada uno es tratado como un píxel con componente alpha en 255. Se almacenan en orden en la componente de color, es decir $RGB = \delta_1\delta_2\delta_3$.
- Imagen de indexación de color: array usado para el mapeo como se almacenado como se describe en la sección 4.6.4.

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 almacenar una codificación entrópica lleva un costo extra este código puede minimizarse si bloques similares usan la misma codificación. Parar esto el codificador puede agrupar bloques agrupando en clusters basado en datos estadísticos o uniendo un par de clusters al azar cuando la cantidad de bits para representar la imagen es menor.

Cada píxel es codificado usando uno de los tres posibles métodos:

- Codificación de Huffman: cada canal usa una codificación entrópica distinta.

- LZ77 backward reference: una secuencia de píxeles es copiada de otra parte de la imagen.
- Color Cache Coding: usando un código de hash corto de un color visto antes.

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

Backward reference son tuplas de longitud y código de distancia. Dónde el largo indica cuántos píxeles van a ser copiados siguiendo el orden de rasterizado. El código de distancia indica la posición del píxel visto previamente, del cual los píxeles van a ser copiados. Estos valores son guardados usando la codificación de prefijo LZ77.

La siguiente imagen muestra LZ77 (en la variante 2D) con el matching de palabras en lugar de píxeles.

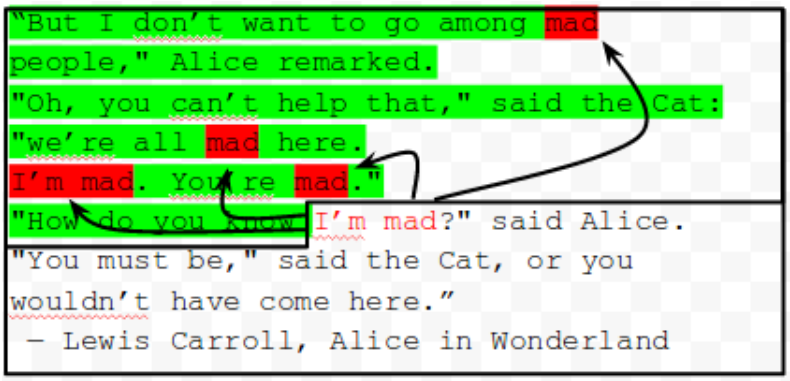


Figura 16

La codificación de prefijo LZ77 divide valores grandes de enteros en dos partes: el código prefijo y los bits extras. El código prefijo se guarda usando codificación entrópica, mientras que los bits extras se guardan sin modificaciones. La idea de esto es reducir los bits extras que son necesarios para almacenar el código. Además, rara vez aparecen valores grandes por lo que muy pocos valores de la imagen usan bits adicionales logrando una mejor compresión en general.

La siguiente tabla muestra los bits extras necesarios para almacenar cada rango de valores, hasta 4096:

Rango	Código de prefijo	Bits extras
1	0	0
2	1	0
3	2	0
4	3	0
5..6	4	1
7..8	5	1
...	...	...
3072..4096	23	10

Para el mapeo de la distancia se diferencia en dos, si el código de distancia es mayor a 120 entonces ese número indica la distancia siguiendo el orden en rasterizado, con un offset de 120.

Las distancias entre [1..120] se representan en coordenadas, indicando cuantos píxeles horizontal y verticalmente respecto al píxel actual hay que moverse para llegar al píxel del cual empezar a copiar. Para almacenar esta distancia se usa un índice que es mapeado a una coordenada guardada en un array.

Por la tanto para obtener la distancia en order a partir de la representación (x, y) de una distancia d se calcula:

$$d = x + y * I_w, \text{ Con } I_w \text{ el ancho de la imagen}$$

Cache de código de color

WebP sin pérdida usa los fragmentos vistos antes para reconstruir nuevos píxeles, también puede usar una paleta local si ningún fragmento se corresponde. La paleta es continuamente actualizada para usar colores recientes. La siguiente imagen muestra la cache de color local que se va actualizando progresivamente con los 32 colores usados recientemente a medida que el escaneo desciende.

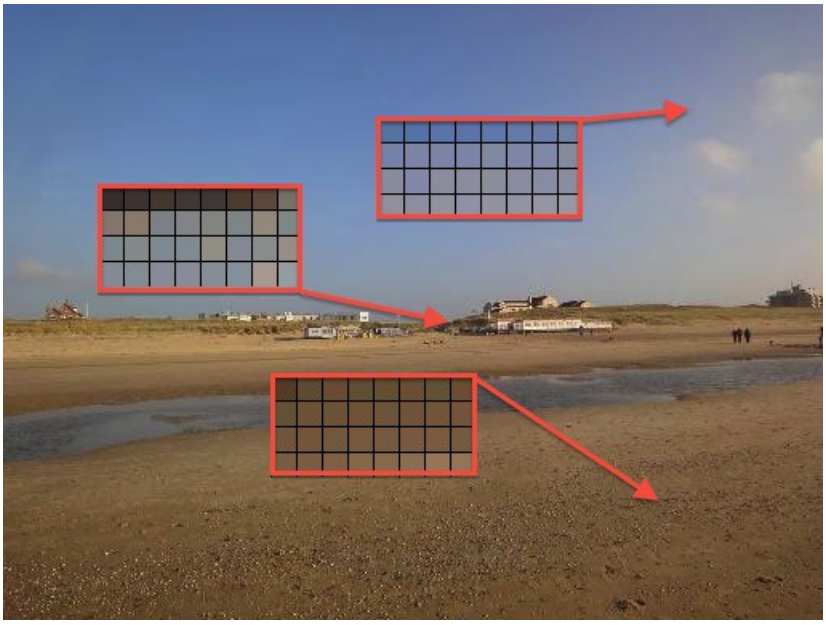


Figura 17: Ejemplo de Color Cache Coding

También sirve para referenciar un conjunto de píxeles anteriormente usado pero a veces es mas conveniente que los otros dos métodos.

Para almacenarlo primero viene un bit indicando si se usa esta codificación, en caso de ser uno, los 4 bits siguientes (s_b) indican el tamaño de la codificación (s) de la siguiente manera:

$$s = 1 << s_b$$

s_b debe estar en el rango [1..11] en caso contrario el decodificador debe indicar un error.

Una cache de color es una matriz de tamaño s. Cada entrada almacena un color ARGB. Los colores se buscan indexándolos por $(0x1E35A7BD * color) >> (32 - s_b)$. Solo se realiza una búsqueda en una caché de color por lo que no hay resolución de conflictos.

Al comienzo de la decodificación o codificación de una imagen, todas las entradas de los valores de caché se ponen en cero. El código de caché de color se convierte a este color en el momento de la decodificación. Cada píxel, ya sea producido por referencias hacia atrás o como literales, se inserta en la caché en el orden en que aparecen en la secuencia.

3.2.4. Codificación

En particular, el formato utiliza codificación Huffman espacialmente variables, es decir, que diferentes bloques de la imagen pueden usar potencialmente diferentes códigos de entropía. Esto proporciona más flexibilidad y potencialmente una mejor compresión.

Los datos de la imagen codificada constan de dos partes: meta codificación de Huffman y código de entropía.

Decodificación de meta códigos de Huffman

Como se señaló anteriormente, el formato permite el uso de diferentes códigos de Huffman para diferentes bloques de la imagen. Los códigos de Meta Huffman son índices que identifican qué códigos de Huffman usar en diferentes partes de la imagen.

Primero un bit el valor 0 indica que se usa el mismo código de meta código de Huffman en toda la imagen. En caso de ser 1 los distintos meta códigos de Huffman se almacenan como una imagen y a esta se le llama *imagen de entropía*[17].

3.3. FLIF: Free Lossless Image Format

FLIF es un algoritmo de compresión de imágenes sin pérdida que utiliza el espacio de color YCoCg, con soporte para el canal alfa[1] en caso que sea necesario. Este algoritmo ofrece dos métodos para recorrer una imagen, uno de entrelazado y el otro no, codificando en ambos métodos las diferencias entre los valores predichos y los reales. Además utiliza una versión generalizada de paleta de colores para agrupar los píxeles. Pero la innovación de este algoritmo se encuentra en la forma de codificar los contextos, utilizando el algoritmo "Meta-Adaptive Near-zero Integer Arithmetic Coding" (MANIAC, o en español "Codificación Aritmética Metaadaptativa de Enteros Cercanos a Cero").

3.3.1. YCoCg

YCoCg es un espacio de color reversible que en lugar de usar tres canales RGB tiene el canal Y o "Luma" que representa la intensidad o luminancia de cada pixel, el canal Co (Croma verde) y Cg (Croma rojo). Un ejemplo de de una imagen descompuesta en cada uno de estos tres canales es el siguiente:

Figura 18: Ejemplo de imagen descompuesto en cada canal de YCoCg

[1]Define la opacidad de un píxel en una imagen.

Para pasar de RGB a YCoCg la transformación es:

$$\begin{bmatrix} Y \\ Co \\ Cg \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{-1}{2} \\ \frac{-1}{4} & \frac{1}{2} & \frac{-1}{4} \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

3.3.2. Color Buckets

FLIF propone un mecanismo llamado Color Buckets, el cual es una alternativa a una transformación existente denominada Palette. Mientras que Palette utiliza un fragmento muy pequeño del espacio de color completo, por ejemplo, $2^{10}, 2^{24}, 2^{32}$ colores posibles; Color Buckets generaliza esta transformación y resulta de utilidad para imágenes que usan relativamente pocos colores diferentes, pero más de lo que cabrían en una paleta de colores.

Esta transformación funciona de la siguiente forma: Para cada valor de Y se hace un seguimiento de los valores de Co que ocurren para ese valor. Siempre que la cantidad de valores de Co distintos para una valor de y dado sea pequeño, se mantiene una lista de valores discretos (un "segmento discreto"); si esa lista se vuelve demasiado grande (de acuerdo con algún umbral arbitrario), entonces se reemplaza por un intervalo, almacenando solo los límites superior e inferior (un "bucket discreto"). A continuación, para cada combinación de Y y Co, se registran los valores de Cg de manera similar. Para mantener un número total de buckets razonable, se utiliza una cuantificación y el mecanismo se deshabilita para imágenes de alta profundidad de bits.

3.3.3. Recorrido de imagen y predicción de píxeles

Como se menciono anteriormente la imagen puede recorrerse en dos modos, el primero, no entrelazado, denominado "Scanline" y utilizado en distintos formatos de imagen, dónde la imagen se escanea línea por línea de arriba a abajo y cada línea se escanea de izquierda a derecha. En este caso, como contexto se toma como predicción la mediana de T (arriba), L (izquierda) y T + L - TL, según se indica en la figura 19.

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

Figura 19: Contexto de un píxel

El otro modo sí es entrelazado, y es una generalización del entrelazado Adam7 usada en PNG. En cada paso de entrelazado, el número de píxeles se duplica. Para el primer paso se toma solo un píxel, el de la esquina superior izquierda. Luego, en cada paso de entrelazado, el número de filas se duplica (un paso horizontal) o el número de columnas se duplica (un paso vertical). El último paso es siempre un paso horizontal, atravesando todas las filas impares de la imagen. Los píxeles indicados en negrita en la Figura 19 se conocen al momento de codificar. En un paso horizontal, también se conoce B, mientras que en un paso vertical, se conoce R. Para este caso se definen tres predictores:

1. El promedio de arriba y abajo (T + B) / 2 (paso horizontal) o izquierda y derecha (L + R) / 2 (paso vertical).

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

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

Para cada caso si se usa una imagen de varios canales se procesa de la siguiente forma: en el recorrido Scanline, simplemente se procesan todos los canales uno por uno, primero el canal alfa (si hay uno), luego el canal Y, luego el canal Co y finalmente el canal Cg. Si el valor de alfa es 0 el resto de los píxeles es irrelevante, por esto se procesa primero. En la otra recorrida, los diferentes canales se intercalan de tal manera que los canales alfa e Y se codifiquen antes con resoluciones más altas que los canales cromáticos.

3.3.4. Codificación de entropía: MANIAC

La codificación aritmética, también conocida como codificación de rango, es una forma de codificación de entropía basada en un modelo de probabilidad para los bits codificados.

En FLIF, a su método de codificación de entropía, lo denominan MANIAC, como mencionamos anteriormente. Esta es una variante de la Codificación Aritmética Binaria

Adaptativa al Contexto (CABAC) 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 este sentido, es metaadaptativo.

En CABAC, es difícil definir un buen modelo de contexto; en particular, para obtener el número de contextos "justo": 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 de contexto terminan en el mismo.
Además, cuando los contextos se definen estáticamente, muchos de estos en realidad no se utilizan, ya que la combinación correspondiente de propiedades simplemente no se produce en la imagen de origen.
MANIAC resuelve este problema mediante la metaadaptación, el cual propone una estructura de datos dinámica como modelo de contexto, permitiendo que el número de contextos sea mas adecuado a la 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, lo que resulta en una mejor compresión.

3.3.5. Árbol de decisión MANIAC

En este árbol se definen contexto que se construyen a partir de la combinación de "propiedades", estas dependen de el recorrido de la imagen. En el caso de Scanline se eligen 5 diferencias tomando en cuenta el contexto del píxel antes mencionado $L - TL$, $TL - T$, $T - TR$, $LL - L$, $TT - T$, además de la misma 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).

Un 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 se construye a partir del "contexto actual", es decir, las combinaciones de propiedades que llevan hasta ese nodo, además de este contexto también se tiene lo que se denomina "contexto virtual" para cada una de las propiedades. Para cada valor codificado se recorre el árbol hasta alcanzar una hoja, inicialmente se usa el *contexto actual* para codificar y además se actualiza el costo estimado (número de bits usados para la salida

comprimida).

Para cada propiedad cada nodo hoja mantiene un promedio de los valores que llegan a esa hoja, luego los dos *contextos virtuales* se usan para almacenar en uno los valores mayores a ese promedio y en el otro los menores. La idea es encontrar en cada hoja las propiedades más relevantes, esto es, aquellas propiedades en donde la suma del costo para ambos *contextos virtuales* es menor que el costo del *contexto actual*, que tanta diferencia se debe alcanzar entre estos depende de un cierto umbral arbitrario. 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, como se puede ver en la figura.

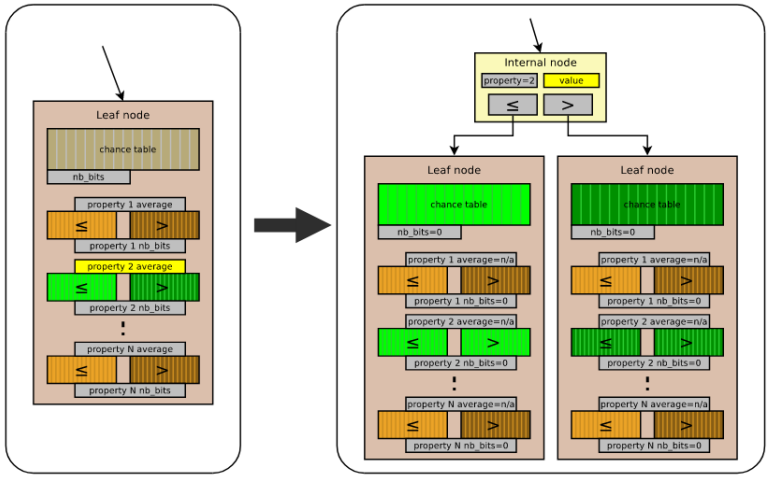


Figura 20: Contexto de un píxel

Al momento de decodificar solo los contexto actuales son necesarios para poder obtener los valores.

4. Estado del arte

Se presenta a continuación los distintos algoritmos de compresión de imágenes sin pérdida basados es redes neuronales.

4.1. L3C

Este sistema se basa en una jerarquía de extractores y predictores de características aprendidas totalmente paralelas que se entrenan conjuntamente para la tarea de compre-

sión.

Para codificar una imagen `x`, se obtienen las predicciones de la distribución de la probabilidad de `x`, así como todas las características auxiliares, en una sola pasada hacia adelante, en paralelo. Estas predicciones se utilizan luego en un codificador aritmético adaptativo para obtener un bitstream comprimido de `x` y las características auxiliares correspondientes. Sin embargo, para el decodificador aritmético se necesitan las predicciones para poder decodificar el bitstream. Entonces, se parte del nivel más bajo de características auxiliares, donde el decodificador obtiene una predicción de la distribución del siguiente conjunto de características auxiliares y, por lo tanto, puede decodificarlas a partir del bitstream. La predicción y la decodificación se alternan hasta que el decodificador aritmético obtiene la imagen `x`.

En la siguiente figura se puede observar una visión general de la arquitectura de L3C. Los *feature extractors* $E^{(s)}$ cuantificados por Q, calculan la representación jerárquica de las características $z^{(1)}, .., z^{(S)}$, cuya distribución conjunta con la imagen x, $p(x, z^{(1)}, .., z^{(S)})$ se modela utilizando predictores no autorregresivos (*non-autoregressive predictors*) $D^{(s)}$. Tanto $E^{(s)}$ como $D^{(s)}$ se modelan como redes neuronales convolucionales. Las características $f^{(s)}$ resumen la información hasta el nivel s.

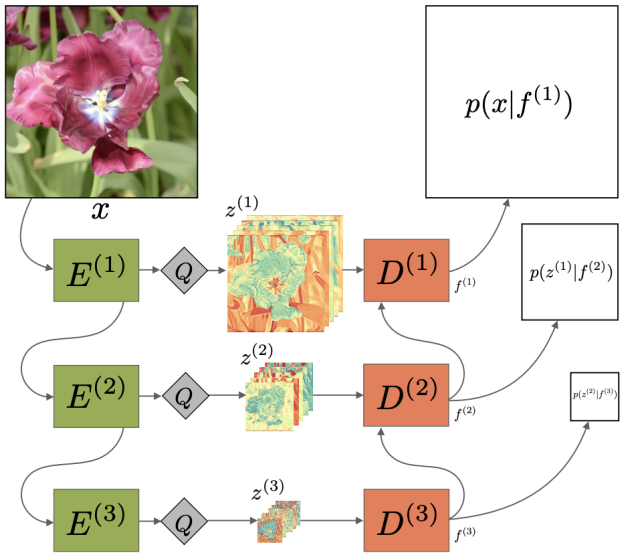


Figura 21: Visión general de la arquitectura de L3C

A diferencia de los modelos autorregresivos, que factorizan la distribución de la imagen de forma autorregresiva sobre subpíxeles, aquí se modelan todos los subpíxeles conjuntamente y además se introduce una jerarquía aprendida de representaciones de característi-

cas auxiliares $z^{(1)}, .., z^{(S)}$. La distribución conjunta de la imagen `x` y la representación de las características $z^{(S)}$ se modela como[16]

$$p(x, z^{(1)}, .., z^{(S)}) = p(x|z^{(1)}, .., z^{(S)}) \sum_{s=1}^{S} p(z^{(s)}|z^{(s+1)}, ..z^{(S)}),$$

dónde $p(z^{(S)})$ es una distribución uniforme.

4.2. IDF

Integer Discete Flow [10] es de los primeros en usar modelos generativo basados en *flows*. Al contrario de los flujos convencionales asumen datos continuos, que pueden conducir a errores de reconstrucción cuando se cuantifican para compresión IDF usa una transformación flexible llamada *integer discrete coupling*, una transformación invertible y cerrada bajo $\mathbb{Z}^d$. Dado $x = (x_a, x_b) \in \mathbb{Z}^d$ la entrada de esta capa, 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 $t()$ una red neuronal.

La arquitectura de IDF se divide en uno o más niveles, dónde cada nivel consta de una operación de *squeeze*, D capas de *integer flow* y una capa de factor de salida (*factor out*).

La siguiente figura muestra la arquitectura de IDF para 2 niveles, la capa de *squeeze* reduce el espacio de dimensiones en dos, mientras que aumenta el número de canales en cuatro. Una capa de *integer flow* consta de una permutación de canal seguido de una capa de *integer discrete coupling*. Se puede observar también que cada nivel consta de D capas de *flow*.

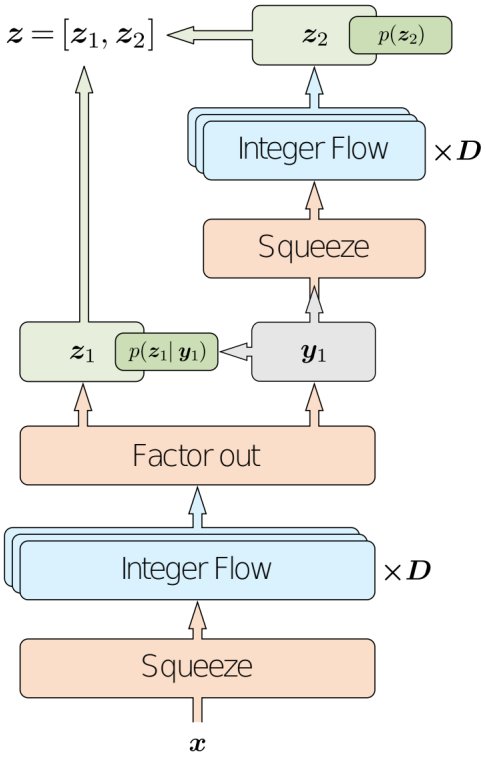


Figura 22: Ejemplo de la arquitectura de IDF en dos niveles

IDF++

IDF++[9] es una variante de IDF que propone algunos cambios en la arquitectura mejorando los radios de compresión pero manteniendo el costo computacional y aplicando la mitad de flows. Principalmente son tres cambios:

- Invertir las permutaciones de los canales después de cada *coupling layer*.
- Adaptación del *rezero trick*
- Alterar cada DenseBlock introduciendo una normalización agrupada y cambiando la activación ReLU por una activación Swish.

4.3. HyperPrior

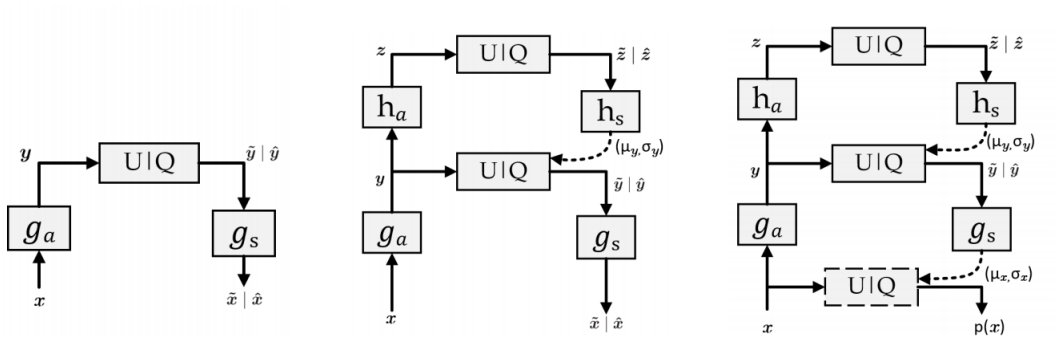
HyperPrior es una técnica eficaz propuesta para la compresión de imágenes con pérdida. Este artículo generaliza el HyperPrior del modelo con pérdidas para la compresión sin pérdidas, y propone un término de norma L2 en la función de pérdida para acelerar el procedimiento de entrenamiento y propone utilizar probabilidades con *gaussian mixture*

para lograr modelos de contexto adaptables y flexibles.

La Figura 14(a) muestra un diagrama operativo para la compresión de imágenes con pérdida, dónde $U|Q$ representa a la cuantificación Q, que durante el entrenamiento es aproximado a un ruido $U(-1/2, 1/2)$.

Mientras que en la Figura 14 (b) se muestra que Hyperprior introduce una información extra, z, que logra capturar las dependencias espaciales entre los elementos de y. La función de h_a y h_s denotan las funciones de análisis y síntesis del codificador auxiliar, dónde ϕ_h y θ_h son parámetros optimizados propios de Hyperprior.

En este artículo lo que se hace es generalizar el diagrama mostrado en la Figura 14 (b) que es para una compresión con pérdida a uno con compresión sin pérdida, el cual se representa en la Figura 14 (c).



(a) Lossy model

(b) Lossy model with a hyperprior

(c) Proposed Lossless model with a hyperprior

Figura 23: Diagramas operativos de compresión de imagen con pérdida y sin pérdida.

4.4. HiLLoC

Lossless Image Compression with Hirerarchical Latent Variable Models (HiLLoC) es una extensión de BB-ANS a modelos jerárquicos de variables latentes. Explota la prioridad de los VAEs puramente convolucionales que entrenados con menor resolución como 32x32 pueden extenderse a imágenes de mayor tamaño, logrando compresión sin pérdida en imágenes de tamaño arbitrario.

4.5. RC

Este algoritmo combina la compresión con pérdida utilizando Better Portable Graphics (BPG, formato de archivo de imagen) y 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 2.3.2), para codificar sin pérdidas el residuo. Luego la imagen se almacena usando la concatenación de bitstream producida por BPG y la codificación del residuo aprendida.

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

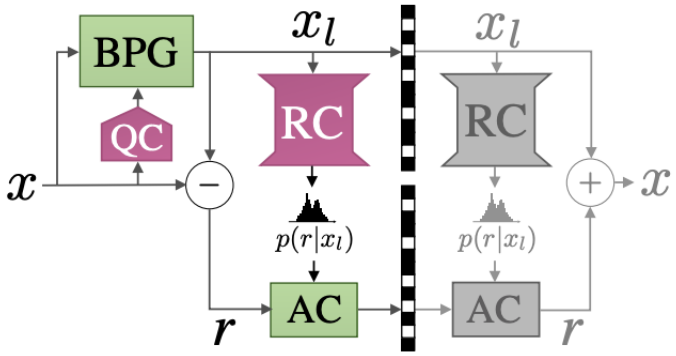


Figura 24: Descripción general de la compresión propuesta por RC

4.6. LBB

Local Bits Back (LBB) es un algoritmo que utiliza flows en combinación con codificación bits-back, esta consiste codificar x usando una distribución de la forma $p(x) = \sum zp(x, z)$, donde $p(x, z) = p(x|z)p(z)$ incluye una variable latente z. Notar que esto puede ser difícil de manejar cuando z se extiende sobre un conjunto exponencialmente grande, lo que hace que sea imposible codificar $p(x)$, aunque manipular la codificación con $p(x|z)$ y $p(z)$ podría ser individualmente manejable. La codificación bits-back introduce una nueva distribución $q(z|x)$ que vuelve esto más manejable, luego se codifica conjuntamente x junto con $z \sim q(z|x)$ mediante estos pasos:

1. Decodificar $z \sim q(z|x)$ de una fuente auxiliar de bits aleatorios.
2. Codificar x usando $p(x|z)$.
3. Codificar z usando $p(z)$.

En esta versión de local bits-back se usan los flows para codificar datos discretizados con alta precisión. Los datos continuos x se discretizan en $\hat{x}$, el cual es el centro de un *bin*

de volumen δ_x. La longitud de código esperada para $\hat{x}$ es la densidad logarítmica negativa del flow f, agregando una constante que depende de la precisión de la discretización:

$$-\log p(\hat{x})\delta_x = -\log p(f(\hat{x})) - \log|\det \mathbb{J}(\hat{x})| - \log \delta_x \tag{10}$$

Siguiendo los pasos descritos arriba las funciones para codificar que se definen en este caso son:

$$\hat{p}(z|x) = N(z; f(x), \sigma^2 \mathbb{J}(x)\mathbb{J}(x)^T) \text{ y } \hat{p}(x|z) = N(x; f^{-1}(z), \sigma^2 \mathbb{I})$$

Por lo que los nuevos paso para la codificación son los siguientes:

1. Decodificar $z \sim \hat{P}(\hat{z}|x) = \int_{B(\hat{z})} \hat{p}(z|x)dz \approx \hat{p}(\hat{z}|x)\delta_z$ de una fuente auxiliar de bits aleatorios.
2. Codificar x usando $\hat{P}(\hat{x}|\hat{z}) = \int_{B(\hat{x})} \hat{p}(x|\hat{z})dz \approx \hat{p}(\hat{x}|\hat{z})\delta_x$.
3. Codificar z usando $P(\hat{z}) = \int_{B(\hat{z})} p(z)dz \approx p(\hat{z})\delta_z$.

Esta codificación se realiza usando rANS, descrita en 2.3.3.

5. Conjunto de datos

En la sección 3, se mencionaron distintos algoritmos de compresión de imágenes, dichos algoritmos presentan informes en base a distintos conjuntos de datos que se presentan a continuación.

5.1. ImageNet

ImageNet es un conjunto de datos de imágenes organizado según la jerarquía de WordNet [2]. 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 1000 imágenes para ilustrar cada synset.

ImageNet32 y ImageNet64 son una variante de ImageNet con la única diferencia de que las imágenes se reducen a 32x32 y 64x64 píxeles por imagen, respectivamente. Contienen aproximadamente 1250000 imágenes de entrenamiento y 50000 imágenes de validación. Las imágenes están en formato PNG.

[2]Base de datos léxica del idioma Inglés.

5.2. CIFAR 10

Este conjunto de datos consta de 60000 imágenes de 32x32 en 10 clases distintas, con 6000 imágenes por clase, dónde las clases son completamente excluyentes entre sí. Hay 50000 imágenes de entrenamiento y 10000 imágenes de validación. El conjunto de datos se divide en cinco batches de entrenamiento y uno de prueba, cada uno con 10000 imágenes. El batch de prueba contiene exactamente 1000 imágenes seleccionadas al azar de cada clase. Los batches de entrenamiento contienen las imágenes restantes en orden aleatorio, pero algunos lotes de entrenamiento pueden contener más imágenes de una clase que de otra. Entre ellos, los batches de entrenamiento contienen exactamente 5000 imágenes de cada clase. 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" [2], están conformados por aproximadamente 2000 imágenes de entrenamiento, dónde CLIC.mobile contiene 61 imágenes de validación tomadas con teléfonos móviles, mientras que CLIC.pro contiene 41 imágenes de validación de DSLR, retocadas por profesionales. Las imágenes tienen distintos tamaños y están en formato PNG.

5.4. Open Images

Open Images es un conjunto de datos de $\sim$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.

Los algoritmos presentados que utilizan este conjunto de datos, usan 300000 imágenes de este conjunto, y usan técnicas para evitar el sobre-ajuste que puede darse debido al formato de las imágenes, el cual no es ideal para la tarea de compresión sin pérdidas. Dichos algoritmos optan por usar igualmente este conjunto de datos debido a que sus características, entre ellas sus imágenes a gran escala.

5.5. DIV2K

El conjunto de datos DIV2K, como su nombre lo dice cuenta con imágenes con resolución 2K, el cual tiene 800 imágenes de entrenamiento y 100 imágenes de validación. Las imágenes están en formato PNG.

6. Experimentos

6.1. Comparación de los distintos algoritmos

Como se mencionó en el estado del arte, hoy en día existen varios algoritmos de compresión sin pérdida, lo cual resulta en la necesidad de poder realizar una comparación entre ellos para ser capaces de detectar el mejor de todos.

Dada la información que se tiene de los algoritmos sin pérdida presentados, se parte de que se pueden comparar entre sí los siguientes: HiLLoC, IDF, IDF++, LBB. Ya que todos presentan la información pertinente para los mismos conjuntos de datos. No así para el caso de RC y HyperPrior, entre ellos sí utilizan conjuntos de datos iguales, pero respecto al resto utilizan distintos. Nuestro objetivo es entonces abordar dicha necesidad para ser capaces de comparar entre sí todos los algoritmos mencionados anteriormente.

El primer paso que se llevo a cabo para cumplir con el objetivo fue filtrar los algoritmos mencionados por los algoritmos que tengan su código *open source*. Este paso fue necesario para ser capaces de probar los algoritmos en distintos conjuntos de datos.

Esto resulto en cuatro algoritmos: HiLLoC, IDF, LBB y RC. Las tablas 1 y 2 muestran las tasas de compresión obtenidas para estos algoritmos.

	CIFAR10	ImageNet32	ImageNet64
HiLLoC	3.56	4.2	3.9
IDF	3.32	4.15	3.9
LBB	**3.12**	**3.88**	**3.7**

Cuadro 1: Rendimiento de compresión en BPD de los métodos HiLLoC, IDF y LBB.

	CLIC.pro	CLIC.mobile	Open Images	DIV2K
RC	2.933	**2.538**	2.791	3.079

Cuadro 2: Rendimiento de compresión en BPD de los métodos RC.

Una vez filtrados los algoritmos ya se podía proceder con las modificaciones correspondientes para probarlos en distintos conjuntos de datos.

Para realizar esto se tienen dos caminos, obtener las tasas de compresión de RC en los conjuntos de datos que usan el resto u obtener las tasas de compresión de todos los

demás algoritmos en los conjuntos de datos que usa RC.

Es evidente que a primera vista la primera opción sería más conveniente, puesto que sólo habría que analizar como se comporta RC en los conjuntos de datos utilizados por el resto de los algoritmos. En un principio se decidió seguir este camino, pero se concluyó que era mejor usar los conjuntos de datos con imágenes mas representativas de la realidad. Dicho esto, lo mejor era evaluar los conjuntos de datos utilizados por RC en el algoritmo que mejor resultados registró del resto, es decir, el algoritmo LBB (ver en tabla 1). Pues si este algoritmo arrojaba mejores resultados que RC, ya tendríamos al mejor de todos.

Para lograr esto, dado que LBB acepta como máximo imágenes de 64x64 y las imágenes pertenecientes a los conjuntos de datos utilizados por RC son mayores o iguales a 512x512 píxeles, se tuvo que proceder con una técnica similar a la que utiliza RC.

Esta técnica se trata de realizar cortes de 64x64 a la imagen original, de tal forma que los cortes x_c de la imagen x no sean superpuestos y la combinación de dichos cortes x_c produzca x. Luego se realiza la compresión de la imagen en b patches de imágenes de 64x64, de tal forma que la cantidad de patches alcance el tamaño de la imagen original. Por ejemplo, si se tiene una imagen de tamaño inicial 512x512, se deben realizar $8 \times 8 = 64$ cortes en total, y por ende 64 patches.

Esta técnica funciona de forma correcta en imágenes cuyos alto y ancho son divisibles entre 64, pero en caso contrario no. Para solucionar este problema se tomo la decisión de cortar previamente la imagen original, de tal forma que se pueda realizar la mayor cantidad de cortes posibles y eliminar el restante de cada imagen. Por ejemplo, si se tiene una imagen de tamaño 2048x1365 se pueden realizar como máximo $32 \times 21 = 672$ cortes, lo cual da como resultado una imagen de 2048x1344 píxeles. Si se quisiera extender esto para usar esos cortes "incompletos" se podrían tomar un *padding* al momento de enviarlo a la red, pero lo anterior era suficiente para lograr nuestro objetivo.

Finalmente se logró obtener para el algoritmo LBB las tasas de compresión para los conjuntos de datos utilizados en RC. La tabla 3 muestra los resultados obtenidos. Dónde se puede ver que LBB supera los resultados alcanzados por RC.

Se agregaron también los resultados para los algoritmos FLIF y WebP, así como también la validación de dichos algoritmos en un nuevo conjunto de datos, denominado Ko-

dak[3].

Es de interés mencionar que todos los resultados de esta tabla son resultados de nuestras propias pruebas.

	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 3: Rendimiento de compresión en BPD de los métodos WebP, FLIFL, LBB y RC.

Es de importancia mencionar que se logró cumplir con el objetivo de lograr comparar todos los algoritmos mencionados anteriormente y se encontró el que mejores resultados arroja, en cuanto a tasas de compresión.

6.2. Entrenamiento con LBB

Dados los resultados obtenidos en el experimento anteriormente mencionado, y que el algoritmo LBB (el que retorno mejores resultados) no ofrece el código para realizar el entrenamiento, se decidió implementar el mismo, a partir del artículo correspondiente a dicho algoritmo y del código de validación disponible en la web.

Los autores del algoritmo ofrecen sus propios modelos, junto con el código *open source* para la evaluación de los mismos, para los conjuntos de datos: CIFAR 10, ImageNet32, ImageNet64. Para la implementación del entrenamiento se decidió utilizar el modelo y la evaluación para ImageNet64, puesto que es el que presenta imágenes de mayor resolución, entre los conjuntos de datos disponibles.

6.2.1. Función de perdida

Como función de pérdida se tomo la función de definida en la ecuación (10), a partir de ahora queda definida de la siguiente forma:

$$total\_logd = dequant\_logd + main\_logd + z\_logp$$

[3]Conjunto de datos con 25 imágenes PNG de tamaño 768 × 512 píxeles publicadas por Kodak Corporation.

6.2.2. Optimizador

Los optimizadores se encargan de actualizar los parámetros de peso para minimizar la función de pérdida. En este experimento se utiliza el optimizador de Adam [11].

6.2.3. Tasa de aprendizaje

La tasa de aprendizaje (*learning rate*) es un hiperparámetro que define que tan rápido se adapta el modelo al problema. Se decidió usar 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 el *learning rate* un factor λ cuando se estanca.

7. Bibliografía

Referencias

[1] *Arithmetic coding*. 1987. URL: https://www.cs.cmu.edu/~aarti/Class/10704/Intro_Arith_coding.pdf.

[2] *CLIC - Workshop and Challenge on Learned Image Compression*. 2021. URL: http://compression.cc/.

[3] *Convolutional Neural Networks, Explained*. 2020. URL: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

[4] *Curso: Aplicaciones de la Teoría de la Información al Procesamiento de Imágenes*. URL: https://eva.fing.edu.uy/course/view.php?id=20.

[5] Aren Dertat. *Applied Deep Learning - Part 3: Autoencoders*. 2017. URL: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

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

[7] *Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design*. 2019. URL: https://arxiv.org/abs/1902.00275.

[8] *Huffman Coding*. 1952. URL: https://www.researchgate.net/publication/304395425_Huffman_coding.

[9] *IDF++: Analyzing and Improving Integer Discrete Flows For Loseless Compression*. 2020. URL: https://arxiv.org/abs/2006.12459.

[10] *Integer Discrete Flows and Lossless Compression*. 2019. URL: https://arxiv.org/abs/1905.07376.

[11] Diederik P. Kingma y Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[12] Carsten Rother Marcus A. Brubaker Ullrich Köthe. *Applied Deep Learning - Part 3: Autoencoders*. 2015. URL: https://mbrubake.github.io/eccv2020-nf_in_cv-tutorial/.

[13] Christopher Olah. *Applied Deep Learning - Part 3: Autoencoders*. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[14] Joseph Rocca. *Understanding Variational Autoencoders (VAEs)*. 2019. URL: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73.

[15] *The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS*. URL: https://eva.fing.edu.uy/pluginfile.php/9130/mod_folder/content/0/2000_loco-i_principles_and_standardization_into_jpeg-ls.pdf?forcedownload=1.

[16] Fabian Mentzer Eirikur Agustsson Michael Tschannen Radu Timofte y Luc Van Gool. "Practical Full Resolution Learned Lossless Image Compression". En: (nov. de 2018). URL: https://arxiv.org/abs/1811.12817.

[17] *WebP Lossless Bitstream Specification*. URL: https://developers.google.com/speed/webp/docs/webp_lossless_bitstream_specification#5_entropy_code.