UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DETECCIÓN DE SEGMENTOS DE VIDEOS DUPLICADOS EN UNA SERIE DE ANIMÉ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

CRISTÓBAL ADRIÁN MUÑOZ VILDÓSOLA

PROFESOR GUÍA:
JUAN MANUEL BARRIOS NUÑEZ

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
MARISA ERNST ELIZALDE

SANTIAGO DE CHILE
2020

Resumen

La detección de segmentos de videos duplicados es un problema que se ha abordado por varios años, buscando solucionar problemas como la detección automática de infracciones de derechos de autor, o la eliminación de contenido duplicado. El aumento exponencial en la cantidad de videos en el Internet empuja la investigación a desarrollar nuevas y mejores técnicas constantemente, presentando nuevos algoritmos todos los años.

Considerando la gran cantidad de segmentos duplicados encontrados en algunas series de animé en la forma de recapitulaciones, esta memoria busca aplicar y comparar técnicas de recuperación de información multimedia para detectar segmentos duplicados en la serie *Naruto Shippuden*.

Para abordar el problema este se dividió en 4 partes: selección de fotogramas clave, extracción de características, búsqueda de vecinos cercanos, y detección de duplicados. Para los primeros tres pasos se buscaron e implementaron técnicas con distintos enfoques, buscando tanto soluciones estándar y ampliamente probadas como algoritmos más novedosos en el estado del arte. Para el último paso se diseñó un algoritmo exhaustivo para detectar la mayor cantidad de duplicados posibles y evaluar efectivamente los métodos probados.

En específico, se probaron los siguientes métodos en cada paso. Para la selección de fotogramas clave se utilizaron los métodos de **reducción de FPS** y la selección por máxima **diferencia de histogramas** local. En la extracción de características se utilizó el descriptor **color layout**, y un descriptor entrenado obtenido a través de una red neuronal profunda **autoencoder**. Finalmente para la búsqueda de vecinos cercanos se utilizaron los índices de búsqueda aproximada **random k-d trees**, y **locality sensitive hashing** utilizando las funciones de hash **stable distributions** y **scalable graph hashing**.

Estas soluciones se evaluaron manualmente utilizando capítulos de la serie analizada, encontrando diferencias importantes tanto en la eficiencia como la efectividad de los métodos en algunos pasos. Especialmente, destacó el índice de búsqueda **random k-d trees** que logró la mejor eficiencia de todos los índices de búsqueda y entregando un muy alto recall (96 % en promedio) en la detección de segmentos duplicados.

En este trabajo se logró desarrollar un framework que permite comparar distintas técnicas propias de la recuperación de información multimedia involucradas en la detección de segmentos de videos duplicados. Esto implica que al incorporarse nuevos algoritmos al estado del arte, será posible comparar su rendimiento contra soluciones existentes, logrando así encontrar rápidamente una combinación eficiente de métodos. Es posible además tomar este trabajo como la base de una herramienta o aplicación que dado un conjunto de videos, reporte los segmentos duplicados encontrados en ellos.

A Naruto, por acompañarme en mi camino ninja.

Agradecimientos

A la Mati, por ayudarme a concentrarme, a redactar, y a seguir trabajando incluso cuando parecía que nada funcionaba, y por escuchar cada vez que le decía cosas que no tenían ningún sentido para ella.

A mi mamá y papá por apoyarme durante toda la carrera y enseñarme todas las cosas nerds que me formaron.

A todos los frens que me apoyaron cuando Naruto me perseguía hasta las profundidades del infierno.

Tabla de Contenido

Capítulo 1

Introducción

1.1. Antecedentes

Día a día se transmiten y generan inmensas cantidades de datos en forma de videos, tan solo en la plataforma de streaming de videos Youtube [15] se suben más de 300 horas de video cada minuto y de acuerdo a reportes creados por la compañía de tecnología Cisco [1], más de un 80 % de todo el tráfico en Internet corresponde a videos (figura 1.1).

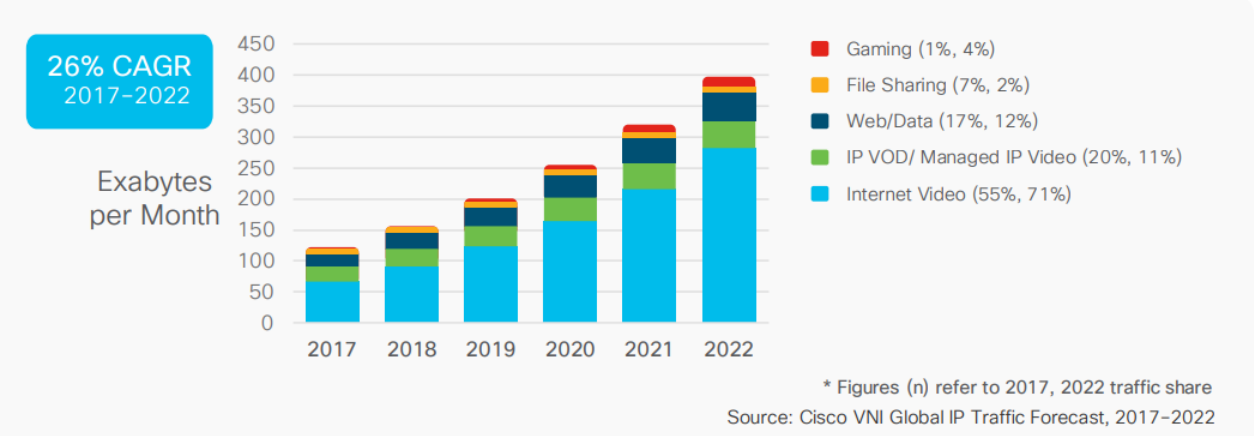


Figura 1.1: Reporte de Cisco indicando la cantidad de datos transferidos por Internet según su categoría.

En este contexto, la investigación académica acerca de videos se ha vuelto cada vez más relevante y se ha desarrollado por ya varias décadas. Sin embargo, el exponencial crecimiento en la cantidad de datos a procesar exige que se desarrollen nuevas y mejores técnicas constantemente, por lo que la vanguardia de la tecnología está constantemente cambiando. Siguen existiendo muchos problemas abiertos para los cuáles no se ha llegado a un resultado satisfactorio, que podrían impactar positivamente la experiencia de los usuarios.

Un problema importante en esta área es la detección de videos duplicados, teniendo múl-

tiples aplicaciones como la detección de infracciones de derechos de autor [31], reducir la redundancia de los resultados de un motor de búsqueda para mejorar su calidad [36], la asignación automática de etiquetas en base a su contenido o simplemente eliminar contenido duplicado de un disco para ahorrar espacio.

Hoy en día, gracias a la gran disponibilidad de herramientas para editar y combinar videos, prácticamente cualquier usuario puede subir videos que sean un duplicado de otro, o que contengan un segmento que pertenece a otro video. Este último caso es de gran relevancia en el problema de los derechos de autor. Para detectar una infracción ya no basta simplemente con buscar videos duplicados, sino que también hay que realizar un análisis más fino buscando segmentos duplicados.

El área de la computación que aborda este problemas es la Recuperación de Información Multimedia (RIM), y se enfoca en buscar y recuperar información semántica de contenido multimedia. Se entiende por contenido multimedia como cualquier objeto que pueda almacenar información, incluyendo imágenes, audio, texto, videos, etc.

A grandes rasgos, el funcionamiento de una aplicación RIM consiste en 2 partes: Primero, resumir el contenido de los documentos a analizar. Luego buscar (o recuperar) documentos relevantes al problema utilizando métodos de búsqueda eficientes. Dentro de estos dos pasos existen cientos de técnicas distintas para llevar a cabo la solución, y un paso se puede dividir en varias sub-etapas para simplificar la resolución de sub-problemas.

1.2. Motivación

Considerando la gran cantidad de técnicas distintas para la detección de segmentos de videos duplicados, el objetivo de este trabajo es evaluar y comparar distintos métodos para determinar cuáles son más adecuados. Para probar los distintos métodos se decidió utilizar la serie de animé *Naruto Shippuden* [9].

El animé es un estilo de animación digital originado en Japón, conocido por su estilo de arte característico. Existe un gran número animé de diversos géneros y cada año aparecen nuevas series, expandiendo constantemente la cantidad de horas de video en el internet.

Una técnica común en este género es la recapitulación de escenas a través de *flashbacks*. Esto consiste en reproducir uno o más segmentos de capítulos anteriores en nuevos episodios con el objetivo de recordar a la audiencia información que ayuda a contextualizar ciertos momentos clave. De ahora en adelante llamaremos a esta técnica simplemente recapitulación. Esta práctica puede ser molesta para la audiencia cuando es abusada, ya que se repite la misma escena reiteradas veces sin agregar contenido a la serie.

Naruto Shippuden es una serie de animé muy popular (posicionada número 20 en popularidad en la base de datos comunitaria de animé MyAnimeList [8] y número 8 en popularidad en la base de datos online de películas y series IMDB [6]) producida por el studio Pierrot que fue emitida entre los años 2007 y 2017.

La razón por la que se escogió la serie *Naruto Shippuden* es que es muy popular, contiene

una gran cantidad de capítulos, y un gran porcentaje de los capítulos tienen recapitulaciones. La serie cuenta con 500 capítulos de aproximadamente 23 minutos cada uno, un total de 191 horas con más de 16 millones de fotogramas. Esto representa un desafío que obliga a utilizar técnicas eficientes para procesar la gran cantidad de datos en un tiempo razonable.

En la figura 1.2 se muestra un ejemplo de una recapitulación entre el capítulo 417 y 179, es claro que esta corresponde a dos segmentos de videos duplicados, con pequeñas modificaciones en el color del video y los subtítulos. Múltiples recapitulaciones se pueden encontrar a lo largo de toda la serie.

Figura 1.2: Ejemplo de una recapitulación en la serie de animé *Naruto Shippuden*. Ambas escenas son idénticas a excepción de un pequeño cambio en el color.

Luego, si se pudieran encontrar todas las recapitulaciones en la serie de forma automática, sería posible permitir al usuario "saltar" estas recapitulaciones si el considera que no es necesario verlas de nuevo, similar a la característica del servicio de streaming de series y películas Netflix que permite saltar la introducción en una serie.

1.3. Objetivo

El principal objetivo de este trabajo es implementar un programa que permita comparar métodos de detección de segmentos de videos duplicados, y utilizarlo para detectar todos los duplicados dentro de un conjunto de videos. Las técnicas implementadas se pueden utilizar para detectar recapitulaciones en cualquier animé, como caso de estudio se analizarán capítulos de la serie *Naruto Shippuden*.

La herramienta implementada para llevar a cabo esta tarea debe ser lo más genérica posible, y no estar ajustada a la detección de duplicados en la serie analizada para entregar un resultado que sea generalizable a otros conjuntos de videos.

1.3.1. Objetivos Específicos

Para lograr el objetivo general se necesitará completar los siguientes puntos:

1. Implementar y evaluar los métodos de extracción de fotogramas clave a usar, **reducción de FPS** y otro método que analice el contenido de los fotogramas.
2. Implementar y evaluar los descriptores visuales a usar, **color layout** y un descriptor obtenido por un **autoencoder**.
3. Implementar y evaluar los índices de búsqueda a usar, uno basado en árboles de búsqueda, uno en base a **locality sensitive hashing** y otro en base a **learning based hashing**.
4. Detectar segmentos duplicados de videos y comparar la efectividad de todos los métodos probados en cada paso.

1.4. Metodología

Para llevar a cabo el objetivo, el trabajo se dividió en tres etapas, las cuáles son descritas a continuación.

1. **Investigación**: En esta etapa se estudiaron diversas técnicas de RIM, su funcionamiento, fortalezas y debilidades.
2. **Solución**: Esta parte consistió en diseñar e implementar la solución al problema. Esta fue la etapa más larga del proceso, en la cual se realizó toda la programación necesaria para la detección de duplicados. Este paso también incluyó la búsqueda de librerías que implementaran los métodos de RIM a utilizar.
3. **Resultados y Análisis**: Finalmente, se graficaron y analizaron los resultados obtenidos comparando los distintos métodos utilizados.

Por otra parte, para simplificar el problema de la detección de duplicados, este se separó en 4 pasos que se describen a continuación. Esto facilitó la comparación entre métodos y permitió organizar de mejor manera el trabajo.

1. **Selección de fotogramas clave:** El objetivo de este paso es resumir los videos a almacenar. Un video actualmente tiene como mínimo 20 fotogramas por segundo (FPS), llegando hasta más de 100 FPS en algunos casos, lo que representa una gran cantidad de datos. Sin embargo, no es necesario almacenarlos todos para detectar copias, y puede llegar a empeorar los resultados al haber demasiados fotogramas parecidos en algunos casos. El ideal es obtener el mínimo número de fotogramas que describen de manera efectiva el contenido del video sin una pérdida de información significativa.
2. **Extracción de características:** En esta etapa se busca resumir cada fotograma en un descriptor de tamaño corto (e.g. pasar de una imagen de 100x100 a un descriptor de 100 valores). Esta etapa no solo busca reducir la cantidad de datos, si no que resumir de manera general la información de la imagen para que sea más fácil compararla con otras imágenes similares (e.g. que las imágenes de perros tengan descriptores similares).
3. **Búsqueda de vecinos más cercanos:** Este paso se encarga de buscar los vecinos más cercanos de los fotogramas en el espacio de características. Los descriptores que son similares entre sí debiesen representar fotogramas similares, lo que indicaría la existencia de fotogramas duplicados.
4. **Detección de duplicados:** Finalmente, teniendo una lista de vecinos más cercanos

para cada fotograma, se buscan secuencias continuas de calces para determinar la existencia de segmentos duplicados.

1.5. Estructura de Documento

El resto del documento se estructura como sigue: En el capítulo 2 se detallan todos los métodos revisados durante la investigación. Luego, en el capítulo 3 se detalla la solución propuesta en este trabajo y la implementación asociada. En el capítulo 4 se exponen y analizan los resultados obtenidos. Finalmente, en el capítulo 5 se expone la conclusión del trabajo realizado y se proponen posibles mejoras y caminos futuros.

Capítulo 2

Marco Teórico

2.1. Procesamiento de Imágenes

2.1.1. Definición de Imagen

En este trabajo consideraremos una imagen como una matriz bidimensional de vectores I con un ancho y alto determinados, a cada punto en esta matriz se le llama *píxel*.

Una imagen blanco y negro posee un valor para cada coordenada $I_{x,y}$ que corresponde a la intensidad de ese píxel. Por otra parte, una imagen a color almacena una tupla de valores para cada coordenada, con cada valor representando el valor de un canal en el modelo de color de la imagen.

Un modelo de color es un modelo matemático que permite representar color en un espacio vectorial. A cada una de las dimensiones del vector se le llama *canal*. Algunos de los modelos más conocidos son RGB, HSV, o CMYK, donde cada letra en el nombre representa un canal de color. Entonces, si se utilizara el modelo de color RGB, una imagen de color tendría tres canales.

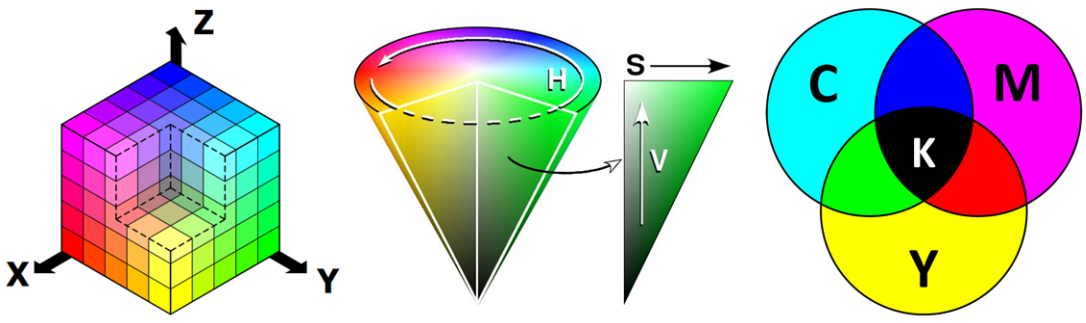


Figura 2.1: Distintos modelos de color, de izquierda a derecha: RGB, HSV, y CMYK.

2.1.2. Operaciones Punto a Punto

Las operaciones punto a punto se utilizan para aplicar una función f a cada uno de los píxeles de una imagen independientemente. Algunos ejemplos comunes de operaciones punto a punto son las conversiones entre modelos de color, pasar de una imagen de color a una blanco y negro, o cambiar el contraste y brillo de una imagen. Lo importante en estas operaciones es que el valor de un píxel no afecta el resultado del resto.

2.1.3. Convolución

La convolución es un operador lineal (comúnmente representado por el operador $*$) que consiste en aplicar un filtro a una imagen. Dada una imagen I en blanco y negro de tamaño $W \times H$ y un filtro w de tamaño $a \times b$, la convolución de $I * w$ está dada por la siguiente fórmula:

$$G(x,y) = (I * w)(x,y) = \sum_{\mathrm{i}=0}^{a-1} \sum_{j=0}^{b-1} I(x - \lfloor \frac{a}{2} \rfloor - \mathrm{i}, y - \lfloor \frac{b}{2} \rfloor - j) \times w(\mathrm{i}, j)$$

Cada nuevo píxel $G_{x,y}$ corresponde a la suma del producto de una "ventana" de la imagen con el filtro aplicado. En la figura 2.2 se muestra el procedimiento para calcular el resultado de una convolución al utilizar un filtro. En general se utilizan filtro de tamaño impar para evitar problemas con la definición del centro del filtro.

Para decidir qué hacer con los píxeles en los bordes de la imagen (en los cuáles el filtro se "sale" de la imagen) hay dos métodos populares. El primero consiste en ignorar estos píxeles, reduciendo el tamaño de la imagen resultante. El segundo es agregar *zero-paddings* al borde de la imagen, y la convolución se aplica en los píxeles del borde de la imagen tomando todo lo que esté fuera igual a cero.

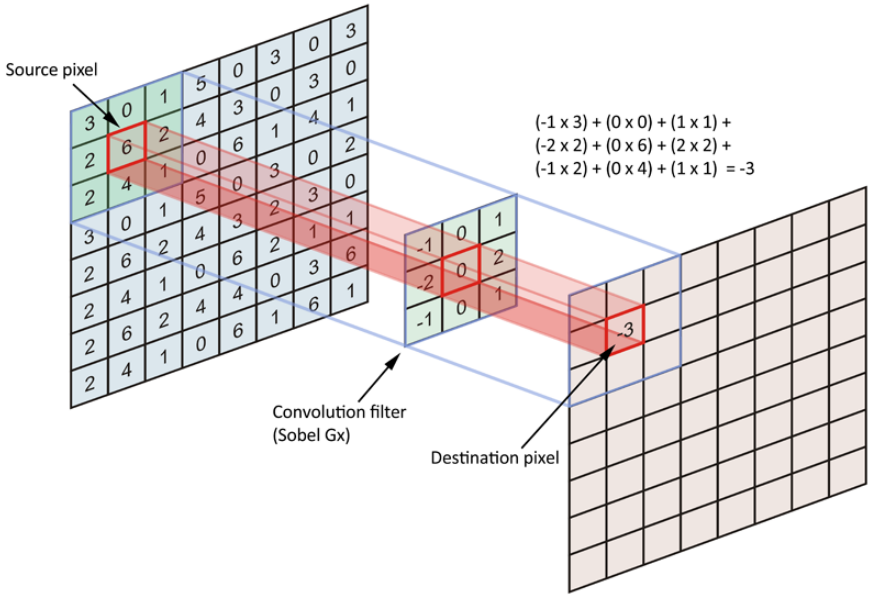


Figura 2.2: Procedimiento para aplicar una convolución de tamaño 3×3 sobre una imagen [24].

Dependiendo del filtro utilizado en la convolución se pueden lograr diversos efectos sobre una imagen, alguno de los cuáles se listan a continuación.

Blur

Los filtros de blur o difuminados, son filtro que hacen que una imagen se vea más borrosa o "suave". Se caracterizan por ser filtros simétricos, de suma uno (significando que no afectan la intensidad total en la imagen) y ser decrecientes hacia los extremos del filtro.

El caso más simple de estos filtros es llamado *box blur*, que consiste en un filtro de $n \times n$ celdas, y todos los valores son $\frac{1}{n \times n}$. Un ejemplo de la aplicación de un filtro de *box blur* de 5×5 se puede ver en la figura 2.3.

Figura 2.3: Filtro *box blur* de tamaño 5×5.

Otro filtro de blur muy común es el *gaussian blur*. Este filtro se basa en crear una gaussiana bidimensional de suma 1 y utilizarla como filtro. Es un poco más sofisticado que el filtro de *box blur*, además de permitir ajustar el parámetro σ, la desviación estándar de la gaussiana usada.

0.075	0.123	0.075
0.123	0.204	0.123
0.075	0.123	0.075

Tabla 2.1: Filtro gaussiano de tamaño 3×3 con $\sigma = 1$

Detección de Bordes

Una utilidad muy grande de los filtros convolucionales es la de tener la capacidad de detectar bordes en una imagen. Existen múltiples técnicas para detectarlos que combinan distintos filtros para obtener los bordes. Algunas de ellas se describen a continuación.

- **Gradiente**: Tomando el gradiente en cada dimensión de una imagen se puede calcular la magnitud y orientación del gradiente usando las siguientes fórmulas:

$$|\nabla L| = \sqrt{L_x^2 + L_y^2}$$
$$\theta = \arctan(L_y, L_x).$$

Luego, se debe utilizar un filtro de convolución que aproxime el gradiente en cada píxel de la imagen. Para ello existen diversos métodos que realizan aproximaciones discretas del gradiente, el más simple utiliza la diferencia central:

$$\nabla_x L(x, y) = -L(x-1, y) + 0 \cdot L(x, y) + L(x+1, y)$$

Con esta fórmula se pueden crear filtros para calcular el gradiente en cada punto. Por ejemplo, en la siguiente tabla se muestra un filtro comúnmente usado para la detección de bordes, llamado filtro de Sobel.

1	0	-1
2	0	-2
1	0	-1

1	2	1
0	0	0
-1	-2	-1

Tabla 2.2: Filtros de sobel para calcular el gradiente en una imagen. El filtro de la derecha calcula el gradiente en el eje X, y el de la izquierda el del eje Y.

- **Diferencia de Gaussianas**: Otra técnica para detectar bordes corresponde a la diferencia de gaussianas (o *DoG*). Este método consiste en aplicar dos filtros de blur gaussiano con distinto σ y luego restar un resultado al otro para detectar los lugares que son más afectados por el blur, que corresponden a píxeles de borde. En la figura 2.4 se puede ver el resultado de aplicar este filtro.

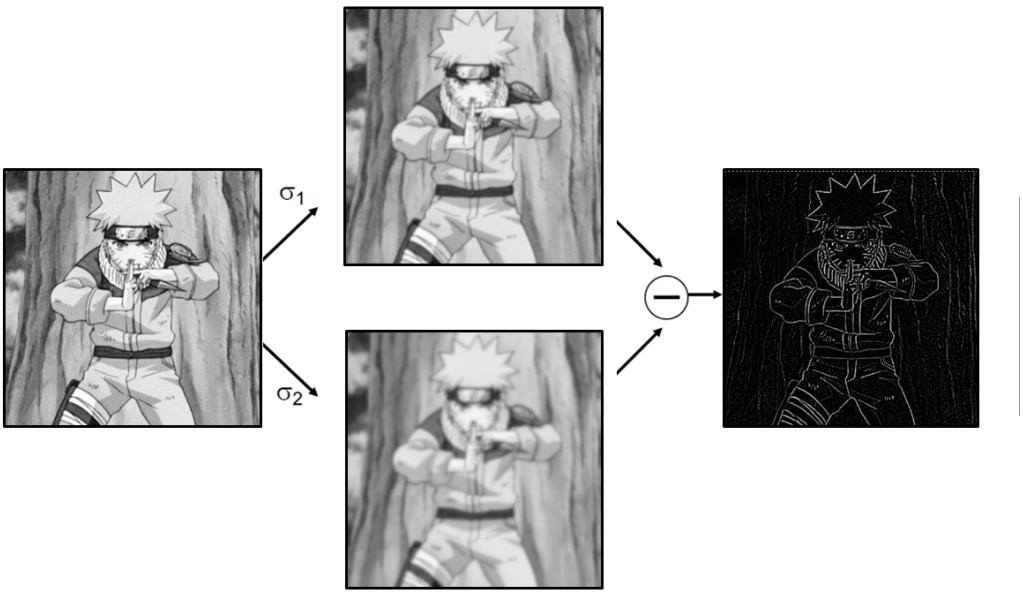


Figura 2.4: Resultado de aplicar el filtro diferencia de gaussianas sobre una imagen.

2.2. Redes Neuronales y Deep Learning

Las redes neuronales artificiales son un modelo computacional inspirado en el funcionamiento del cerebro animal, compuesto por muchas sub-unidades llamadas neuronas artificiales

conectadas entre sí. Su objetivo es aprender y mejorar su comportamiento en base a ejemplos, sin conocimiento previo de la tarea revisada ni programación específica para ella.

El ejemplo más simple de una red neuronal es un *Multi Layer Perceptron* que consiste de 3 capas: una capa de entrada, una capa oculta o de procesamiento y una capa de salida. Las conexiones en esta red solo se hacen hacia "adelante" (*feed-forward*) y de manera completa (*fully-connected*): todas las neuronas de una capa envían información a todas las neuronas de la siguiente capa.

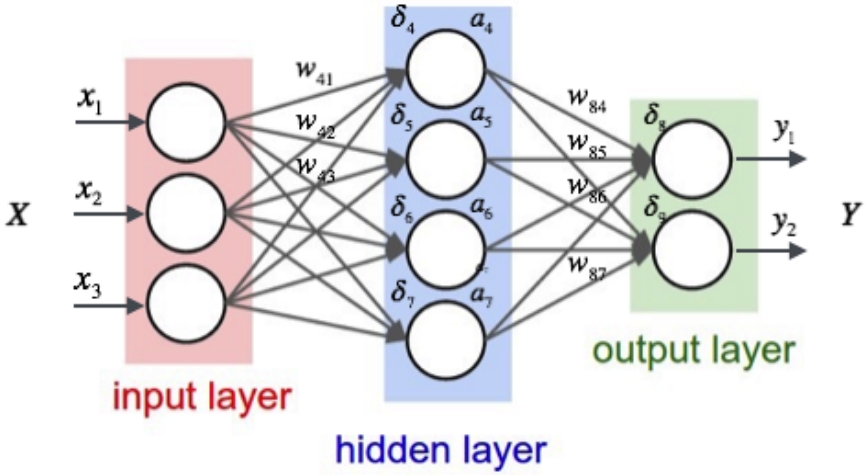


Figura 2.5: Red neuronal *feed-forward* con solo 3 capas de neuronas *fully-connected*. [28]

Algunas de las capacidades generales de las redes neuronales se listan a continuación:

- Aproximación de funciones y análisis de regresiones.
- Clasificación, incluyendo el reconocimiento de patrones o secuencias y la toma de decisiones.
- Procesamiento de datos, incluyendo filtrado, clustering y compresión.

Debido a su gran capacidad de modelar y reproducir procesos no lineares, estas redes se han utilizado en múltiples disciplinas para resolver problemas con una gran certeza. Desde filtrar spam en el correo hasta detectar pacientes con diabetes, las redes neuronales pueden ofrecer solución a muchos problemas de distinta índole.

2.2.1. Deep Learning

El aprendizaje profundo, o *deep learning*, es una rama del aprendizaje de máquinas utilizando redes neuronales. Las principales diferencias entre las redes neuronales tradicionales y las redes neuronales profundas o *Deep Neural Networks* (DNN) son dos:

1. La cantidad de capas/neuronas en la red debe ser muy grande. Debido a que la capacidad computacional aumenta constantemente, este valor no es fijo, si no que se va adaptando a medida que pasa el tiempo. Las redes neuronales profundas en el estado del arte actual tienen más de 100 capas.

2. Los datos de entrada de la red corresponden a los datos "en bruto". En las redes neuronales tradicionales, independiente del problema a analizar, siempre se preprocesan los datos antes de entregárselos a la red (e.g. para clasificar imágenes, se le entregan a la red los descriptores de las imágenes). En cambio, en una DNN el cálculo del descriptor forma parte de la red (e.g. para clasificar imágenes se le entrega la imagen completa).

Existen 2 grandes ramas de DNN, las DNN recurrentes (R-DNN) y las DNN de convolución (C-DNN). Las R-DNN son redes en las que existen conexiones entre neuronas "hacia atrás", lo que le permite a la red almacenar información de las entradas anteriores al procesar una nueva. Esta "memoria" hace que estas redes sean especialmente útiles al trabajar con datos que tienen un orden temporal, como lo es una cadena de texto. Por otra parte, las C-DNN utilizan la convolución para procesar los datos de entrada, lo que hace que destaquen en la resolución de problemas relacionados con imágenes y audio. Para este trabajo son más relevante las C-DNN, por lo que profundizaremos en ellas.

El recibir los datos de entrada "en bruto" imposibilita el uso de las capas *fully-connected* de las redes neuronales tradicionales debido a la gran cantidad de parámetros que se necesitarían. En una red *fully-connected*, si la entrada es una imagen de 100×100 en blanco y negro, y la primera capa oculta de la red tiene 1000 neuronas, se necesitarían $100 \times 100 \times 1000 = 10{,}000{,}000$ de parámetros solo en la primera capa. Entrenar una red neuronal con estas características requeriría mucho tiempo y muchos ejemplos.

Para resolver este problema, las C-DNN utilizan 2 herramientas que reducen considerablemente el poder computacional necesario para procesar la entrada de la red. Estas son las capas de convolución y las capas de *pooling*.

Convolución

Una capa de convolución es una capa de la red neuronal que aplica uno o más filtros de convolución sobre los datos de entrada. Los filtros aplicados deben tener la misma cantidad que canales que la entrada, por lo que en general son filtro tridimensionales. Por ejemplo, un filtro de convolución de tamaño 5×5 para una entrada con 8 canales será una matriz de tamaño $5 \times 5 \times 8$. En la figura 2.6 se ilustra una convolución de tamaño 3×3 aplicado a una entrada de n capas.

La mayor ventaja de utilizar filtros convolucionales es que el número de parámetros se reduce drásticamente, pasando de millones de parámetros en una capa *fully-connected* a unas pocas decenas en una capa convolucional. Esto reduce el tiempo necesario para entrenar la red y procesar los datos multimedia "en bruto", permitiéndole a la red aprender qué características extraer de los datos.

Pooling

Las capas de *pooling* son capas en las que el principal objetivo es reducir el tamaño de la entrada. Para ello se toman ventanas de un tamaño definido en cada canal de la entrada, con un espaciado o *stride* entre ventanas (en general sin solaparse) y se reduce la ventana a un solo valor utilizando algún criterio establecido. El *pooling* más comúnmente utilizado debido a los buenos resultados que entrega en la práctica es el *max-pooling*, que toma solo el

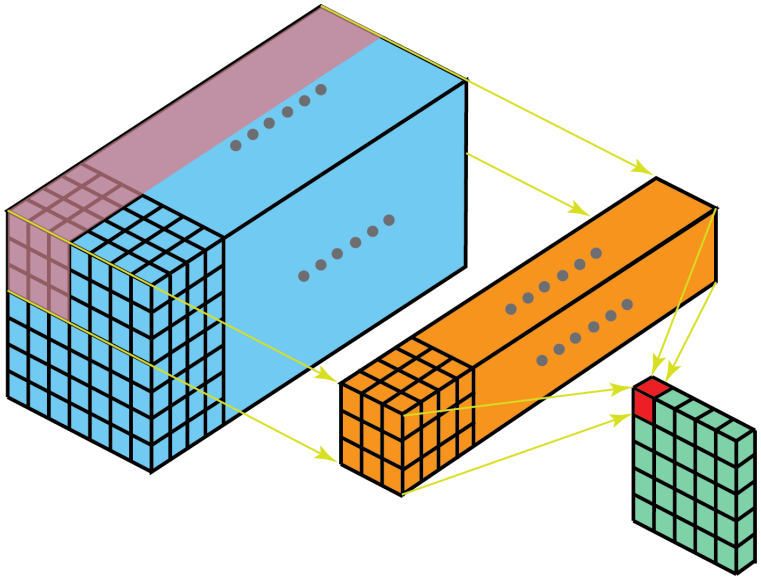


Figura 2.6: Capa de convolución de tamaño 3×3 en una red neuronal *deep*. El filtro se extiende por todos los canales de la entrada. [17]

máximo dentro de cada ventana. Un ejemplo de *max-pooling* se muestra en la figura 2.7, en este caso, se eligieron ventanas de tamaño 2×2 con stride 2 (las ventanas no se solapan).

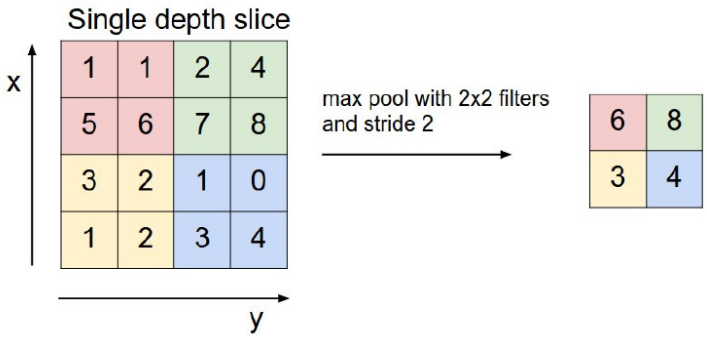


Figura 2.7: *Max-pooling* de tamaño 2×2 aplicado a una. [27]

Ejemplos

Las C-DNN vieron una explosión de popularidad cuando entre 2011 y 2012 redes de este tipo ganaron múltiples competencias de reconocimiento de imágenes, alcanzando por primera vez una efectividad mayor a la efectividad humana. Desde entonces, nuevas redes convolucionales se han desarrollado para diversos problemas.

Una de las redes más famosas es *AlexNet* [30]. Esta fue una de las primeras redes convolucionales en ganar competencias de visión por computación, y llamó mucho la atención debido a la gran diferencia que obtuvo con respecto a los otros competidores (AlexNet obtuvo un error de 15 %, mientras que el segundo lugar tuvo un error de 25 %). Esta red posee 5 capas convolucionales y 3 capas de *max-pooling* que realizan la extracción de características de la

imagen de entrada y 3 capas *fully-connected* que realizan la clasificación final, en la figura 2.8 se puede apreciar la arquitectura completa.

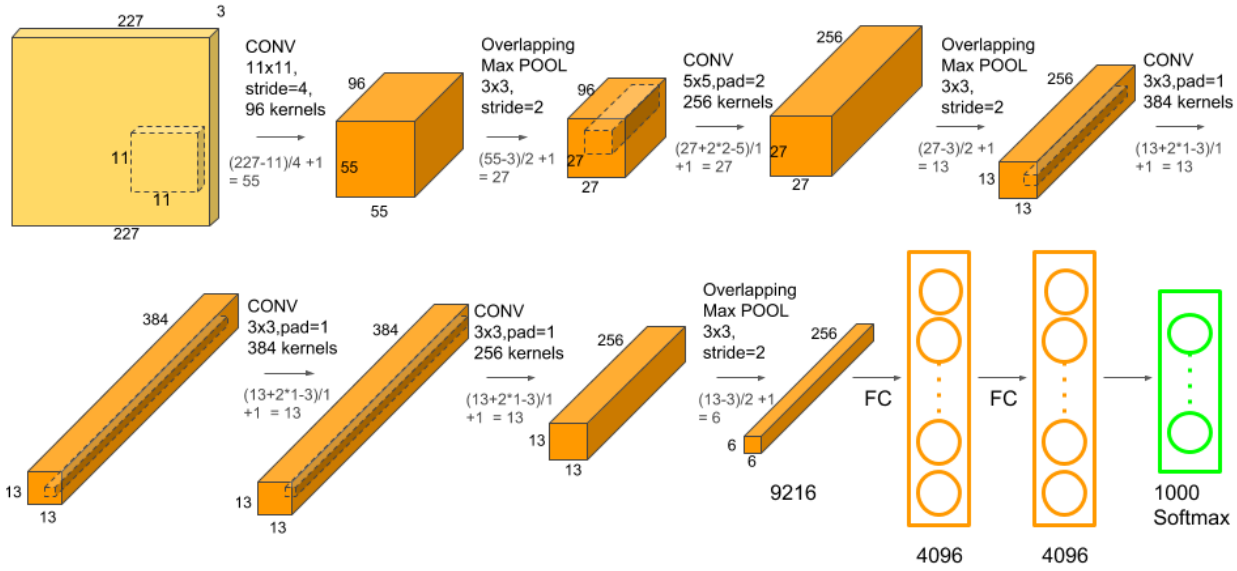


Figura 2.8: Arquitectura de la C-DNN *AlexNet*. [32]

2.3. Descripción de Características

La descripción de características consiste en calcular un descriptor (también llamado vector de características o simplemente características) para representar características visuales de una imagen. Estos descriptores generalmente corresponden a vectores multidimensionales o secuencias binarias, pero también puede ser un título o un conjunto de categorías a las que pertenece un documento. Estos descriptores se clasifican en dos categorías principalmente:

- Descriptores de alto nivel: Son descriptores semánticos asignados por humanos, directa o indirectamente. e.g. etiquetas, títulos, descripciones.
- Descriptores de bajo nivel: Corresponden a descriptores que analizan características del contenido mismo, extraídos automáticamente a través de algún algoritmo. e.g. histogramas de color, promedios de intensidad, orientaciones de borde.

La mayoría de los sistemas grandes de RIM (e.g. Youtube) utilizan principalmente descriptores de alto nivel para analizar y organizar los documentos. Un descriptor de alto nivel representa información semántica del documento almacenada típicamente en la forma de metadata, como los son etiquetas, títulos y descripciones. Esta información se obtiene utilizando criterio humano para describir el contenido a través de etiquetas, títulos, descripciones, entre otros. Sin embargo, los descriptores de alto nivel necesitan de humanos que los escriban, lo cual no es sostenible con la cantidad de contenido multimedia que se genera a diario [22].

Además, la calidad de la metadata de un documento es variable debido a los errores humanos asociados (términos ambiguos, errores de ortografía, etc).

Sin embargo, en los últimos años y cada vez con más fuerza se han empezado a utilizar descriptores de bajo nivel. Estos representan información del contenido mismo del documento, en general en la forma de patrones o datos estadísticos. Aunque estos descriptores tienden a no representar de manera tan efectiva la información semántica del documento (problema conocido como *semantic gap*) [34], tienen la ventaja de poder ser generados automáticamente para conjuntos de documentos grandes. Un ejemplo común de su uso es el reconocimiento facial para etiquetar automáticamente a personas en fotos. De ahora en adelante nos referiremos a estos descriptores como vectores de características o simplemente características.

Existen 2 tipos de descriptores de bajo nivel, los descriptores globales y los descriptores locales. Como su nombre lo indica, los descriptores globales resumen todo el contenido de un documento en un solo vector de características y su principal función es encontrar documentos similares en su totalidad, por ejemplo, encontrar todas las fotos de perros en una colección. Por otra parte, los descriptores locales buscan representar zonas o puntos importantes del documento y permiten realizar búsquedas parciales dentro de otros documentos, por ejemplo, buscar la ocurrencia del logo de Coca Cola en una foto.

2.3.1. Descriptores Globales

A continuación se describen algunos descriptores globales comunes para imágenes.

Vector de Intensidades

El vector de intensidades de una imagen es una descripción global sencilla que consiste en representar una imagen por la intensidad de sus pixeles. Para esto primero se escalan las imágenes de un conjunto a un tamaño fijo deseado, $W \times H$. Luego cada imagen se convierte a escala de grises y se crea su vector de intensidades. Este vector es de tamaño $W \times H$ y guarda la intensidad de cada partición correspondiente de la imagen, como se ilustra en la figura 2.9.

Este descriptor permite la comparación de imágenes pues los vectores resultantes son fácilmente comparables usando la distancia euclidiana. Además, como el vector de tamaño fijo, es invariable a cambios en la resolución de una imagen. Otro beneficio de este descriptor es que es relativamente rápido de calcular y solo utiliza el espacio necesario para guardar los $W \times H$ valores del vector de intensidades.

Histograma de Intensidades

Siguiendo por esta misma línea, otro descriptor global comúnmente usado es el histograma de intensidades de una imagen. Este consiste en crear un histograma que contabiliza las intensidades de una imagen, sin considerar su ubicación en ésta. De este mismo modo se puede describir una imagen utilizando su histograma de color, contabilizando la ocurrencia de cada color dentro de una celda de la imagen.

Figura 2.9: Vector de intensidades: la imagen original se convierte a escala de grises y luego se escala a un tamaño $W \times H$, en este caso, 8x8.

Color Layout

Este descriptor busca resumir el contenido de una imagen según los patrones que se encuentran en sus colores básicos [29]. El proceso para obtener este descriptor es el siguiente:

1. División de imagen: La imagen se particiona en una grilla de 8×8 para garantizar que no varíe la escala o la resolución.
2. Elección de color representativo: Se calcula un color representativo para cada celda y se traduce de RGB (Rojo Verde Azul) a YCbCr (Luminancia, crominancia azul, crominancia roja), se recomienda utilizar el promedio para elegir el color representativo.
3. Transformación: Sobre cada matriz de colores representativos se calcula la transformada discreta del coseno de 8×8. Esto representa la detección de patrones con distintas frecuencias en cada eje.
4. Scan Zigzag: Se cuantifican los coeficientes a valores enteros utilizando una tabla de cuantificación para disminuir el espacio de almacenamiento y facilitar la compresión, y se recorren los coeficientes en "zigzag" para agrupar las frecuencias bajas en el principio del arreglo generado.

Este descriptor tiene la ventaja de usar poco espacio (192 valores) manteniendo un gran porcentaje de información de la imagen, y de ser fácil de comparar utilizando las normas L2 (distancia euclidiana) o L1 (distancia Manhattan).

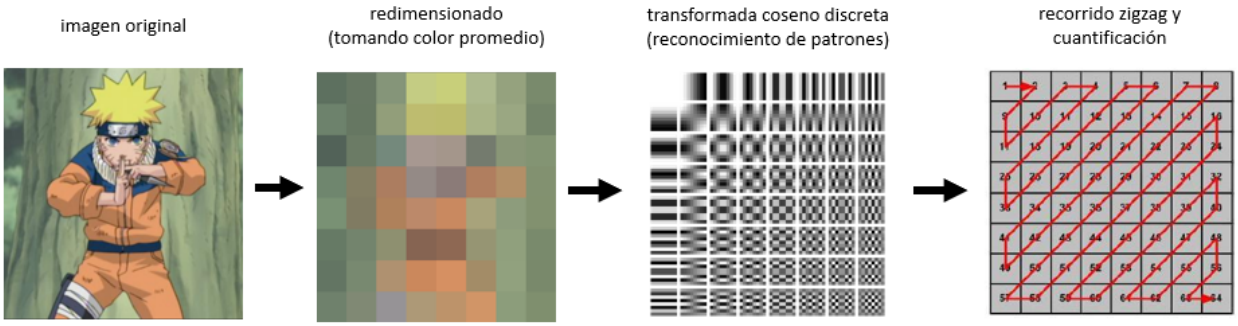


Figura 2.10: Ejemplo del proceso realizado al extraer el descriptor *color layout*.

Deep Features

Una técnica que derivó del *Deep Learning* es el aprendizaje de características *deep* o también llamado *deep features*. Esta técnica consiste en entrenar una red neuronal profunda, usualmente en algún problema de clasificación, y luego utilizar la salida de la red en una de las capas ocultas como descriptor global de las imágenes ingresadas.

Por ejemplo, en AlexNet (figura 2.8) se ha utilizado la salida de la penúltima capa *fully-connected* para generar un descriptor global de 4096 valores, y con él resolver otros problemas de clasificación de imágenes y reconocimiento de objetos.

Estos *deep features* han probado entregar buenos resultados siempre y cuando el problema a resolver sea similar al problema que resolvía originalmente la C-DNN usada.

Autoencoder

Un *autoencoder* es una C-DNN que genera descriptores de características de forma no supervisada. Su funcionamiento se divide en dos, primero se reduce el tamaño de la entrada a través del uso de capas convolucionales y *max-pooling* hasta llegar a una capa del tamaño de código deseado, y luego se reconstruye la entrada a partir del código generado buscando lograr un resultado lo más cercano posible, de ello su nombre.

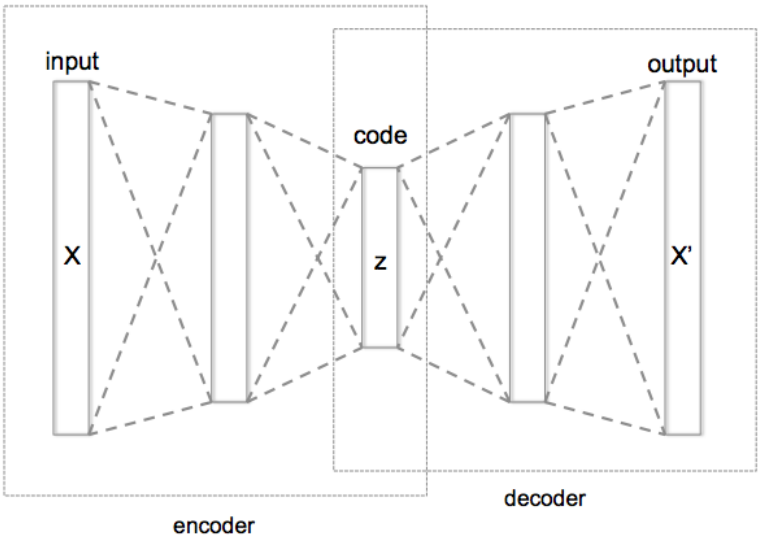


Figura 2.11: Estructura general de un autoencoder. [21]

Para entrenar la red, la entrada y la salida esperada son las mismas, obligando al *auto-encoder* a comprimir la entrada en la primera parte, para luego reconstruirla minimizando los cambios generados. Este entrenamiento busca encontrar una representación eficiente del conjunto de entrada, detectando patrones comunes en ellos. Esto también se puede ver como una reducción de dimensionalidad de los datos.

La salida de la capa intermedia corresponde al descriptor de la imagen. Después del entrenamiento solo es necesario pasar una imagen por la primera mitad de la red (también conocido como el *encoder*) para obtener su descriptor. El desafío para obtener un buen descriptor es que el conjunto de entrenamiento represente fielmente las imágenes que se procesarán en el futuro.

2.4. Fotogramas Clave

Teniendo un método para extraer características de una imagen, este se puede generalizar fácilmente a la extracción de características de un video tomando los descriptores de todos sus fotogramas y concatenándolos en orden. Sin embargo, si se realizara esto, muchos de los descriptores serían casi idénticos debido a que 2 fotogramas consecutivos tienden a presentar cambios despreciables.

La solución a este problema es realizar un paso previo a la extracción de características conocido como selección de fotogramas clave (o *keyframes*, en inglés). Este paso consiste en seleccionar un conjunto de fotogramas que represente de la mejor manera el video disminuyendo lo más posible la redundancia. Un keyframe es un fotograma de un segmento de video que puede representarlo sin que haya pérdida de información significativa. El largo de los segmentos puede ser fijo o variable dependiendo del método de selección usado.

2.4.1. Reducción de FPS

El método más sencillo para seleccionar keyframes es hacer simplemente una reducción de fotogramas por segundo (FPS) a la cantidad deseada. En la figura 2.12 se diagrama el proceso de selección que consiste en tomar 1 de cada k fotogramas, reduciendo la cantidad de datos almacenados a un k-ésimo del total.

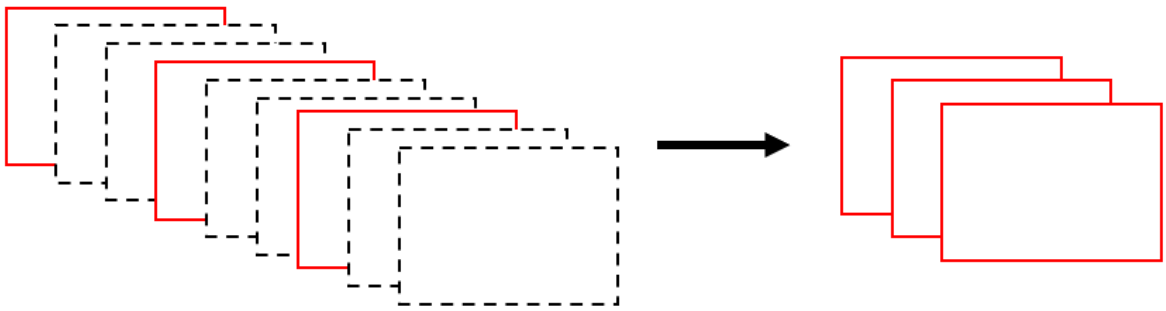


Figura 2.12: Selección de keyframes mediante reducción de FPS. En este ejemplo, se toma 1 de cada 3 fotogramas.

2.4.2. Análisis de Contenido

Otro enfoque para seleccionar keyframes consiste en analizar el contenido del video para decidir bajo algún criterio establecido qué fotogramas tomar. Las formas de analizar el contenido varían desde herramientas muy simples como comparar vectores de intensidades, hasta métodos complejos como seguimiento de movimiento. Los criterios para elegir qué fotogramas tomar en general se enfocan en detectar cambios que superan un límite definido (criterio global), o tomar los cambios máximos dentro de una ventana (criterio local). Los criterios globales pueden ser útiles para detectar cambios de escena, mientras que los criterios locales ayudan a segmentar de manera uniforme el video.

2.5. Búsqueda por Similitud

Teniendo los descriptores de los keyframes de un video el siguiente paso es buscar descriptores similares entre sí, que deberían corresponder a fotogramas similares si es que el descriptor es bueno. La Búsqueda de Videos Basada en Contenidos (BVBC) es el subárea de RIM que se especializa en la búsqueda de videos utilizando principalmente descriptores audiovisuales de bajo nivel. Esto permite utilizar RIM en conjuntos muy grandes de documentos que no contienen descriptores de alto nivel apropiados y no es posible pedirle a un grupo humano que los genere.

La forma más común de buscar descriptores similares es la búsqueda del vecino más cercano al vector de consulta dada una función de distancia. Esto se define de la siguiente forma: Dado un objeto de consulta q, un conjunto de objetos $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ y una función de distancia $\mathrm{d}(q, x)$, el objetivo es encontrar al objeto $NN(q)$ (del inglés, nearest

neighbour) que cumpla $NN(q) = \arg\, min_{x \in \mathcal{X}}\ \mathrm{d}(q, x)$. Esto se puede generalizar fácilmente a la búsqueda de los k vecinos más cercanos.

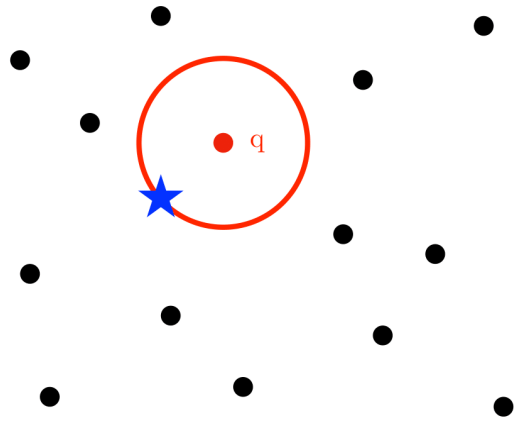


Figura 2.13: Ejemplo de una búsqueda de vecino más cercano. El conjunto de puntos de referencia $\mathcal{X}$ corresponden a los puntos negros y la estrella, la estrella es el vecino más cercano del punto de consulta q. [33]

El algoritmo ingenuo para encontrar a los vecinos más cercanos consiste en calcular la distancia desde un punto a todos los otros y luego tomar los k más cercanos. Utilizando un *minHeap* este algoritmo tiene un costo $O(n)$, con n el número de vectores del conjunto, lo cuál se vuelve rápidamente infactible de calcular en un tiempo aceptable a medida que n crece. Es por ello que nace la necesidad de usar estructuras de datos para optimizar las búsquedas a cambio de una pérdida en la precisión. Este nuevo problema se llama búsqueda aproximada de vecinos más cercanos, para el cuál existen diversas soluciones en el estado del arte. A continuación se describen algunas de ellas.

2.5.1. Árboles de Búsqueda

Una estructura común para indexar los datos son los árboles de búsqueda. Generalmente se usan árboles balanceados, lo que significa que todos sus nodos terminales (hojas) están a una distancia similar del nodo principal (raíz). Existen muchas formas de construirlos, pero su funcionamiento general es que dividen los datos en ramas según su posición recursivamente, hasta llegar a un tamaño deseado en los nodos finales (hojas), su estructura general se puede ver en la figura 2.14. Los puntos que se encuentran en una misma hoja siempre serán cercanos entre sí (aunque no necesariamente son los más cercanos).

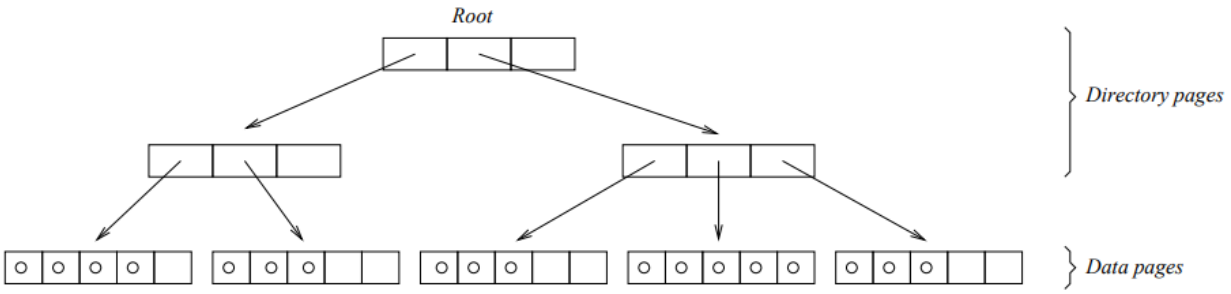


Figura 2.14: Estructura general de un árbol de búsqueda balanceado. Los nodos internos definen regiones espaciales jerárquicas mientras que las hojas almacenan punteros a los vectores. [20]

El algoritmo de búsqueda exhaustivo aprovecha la topología del árbol para recorrer los puntos de manera inteligente, recorriendo solo los nodos "cercanos" entre sí para encontrar los vecinos más cercanos a un punto. Esto entrega un tiempo promedio de $O(log(n))$, pero un pero caso de $O(n)$ al igual que el algoritmo ingenuo. La verdadera ventaja de los árboles es que se puede relajar la búsqueda para disminuir el tiempo de búsqueda pero no necesariamente obteniendo los vectores **más** cercanos al punto de consulta. Por ejemplo, se puede limitar el número de hojas a examinar, con lo que el tiempo de búsqueda se vuelve constante.

k-d Tree

Un k-d Tree es una estructura de datos que particiona el espacio con el fin de organizar puntos en un espacio k-dimensional [18]. Este tipo de estructuras resultan útiles para distintas aplicaciones, como búsquedas por rango o búsquedas de vecinos más cercanos.

Esta estructura puede verse como un árbol binario en el cual cada hoja es un punto k-dimensional. A su vez, cada nodo interno puede interpretarse como un hiperplano que divide el espacio en dos partes. Los puntos a la izquierda de este hiperplano son representados por el subárbol izquierdo del nodo, mientras que los puntos a la derecha del hiperplano son representados por el subárbol derecho.

La dirección del hiperplano se determina asociando a cada nodo del árbol con una de las k-dimensiones. El hiperplano será ortogonal al eje de esa dimensión. De este modo si a una división se le asigna un eje, todos los puntos en el subárbol cuyo valor en ese eje es menor al del nodo, se asignarán al subárbol izquierdo. En caso contrario, se repartirán al subárbol derecho.

Una manera de construir un k-d es eligiendo la dirección del hiperplano como el eje perpendicular al eje de mayor varianza en los datos. Luego se selecciona la mediana de los puntos a insertar respecto al eje divisor. Los puntos insertados se dividen entonces según si su valor en el eje divisor es menor o mayor a la mediana. Este proceso se repite recursivamente hasta distribuir todos los puntos.

Este método de construcción asegura que el árbol resultante sea balanceado y que los

puntos se encuentren distribuidos de la mejor manera posible.

2.5.2. Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) fue introducido en 1998 por *Indyk et al* [25] como un sistema de hashing aleatorio para realizar búsquedas aproximadas de vecino más cercano de manera eficiente. La técnica se basa en la definición de la familia de LSH $\mathcal{H}$, un conjunto de funciones de hash que mapea vectores similares al mismo código de hash con mayor probabilidad que vectores disimilares. LSH tiene la ventaja de poder realizar búsquedas aproximadas de vecinos más cercano en tiempo sub-linear. Basta con buscar todos los vectores pertenecientes al mismo *bucket* que el vector a buscar y luego realizar una búsqueda secuencial sobre estos candidatos.

Bit Sampling

Bit sampling es uno de los primeros métodos de LSH creados [25] y se caracteriza por su simplicidad. Teniendo un conjunto de entradas de n bits, se seleccionan l posiciones al azar en $\{1, 2, .., n\}$ y el código de cada elemento corresponde a seleccionar los l bits en las posiciones seleccionadas.

Random Projections

Esta función de hash mapea puntos desde $\mathbb{R}^{\mathrm{d}}$ a $\{-1, 1\}$ [16]. Para ello, genera un hiper-plano (definido por un vector normal unitario r) de d dimensiones y luego calcula a qué lado del plano se encuentra el punto de entrada v. Esto es:

$$h(v) = signo(v \cdot r) = \pm 1$$

Concatenando p proyecciones se obtiene un hashing de largo p, y para obtener mejores probabilidades de encontrar los vecinos más cercanos se pueden utilizar L tablas de hashing separadas.

Esta función de hash es útil para aproximar la distancia angular entre 2 puntos.

Stable Distributions

Esta función de hash $h_{a,b,r}(v) : \mathbb{R}^{\mathrm{d}} \longrightarrow \mathbb{N}$ mapea un vector de d dimensiones al espacio de los números enteros [23]. Cada función está definida por un valor r que representa el ancho de cada celda, un vector aleatorio de d dimensiones a que define un hiper-plano para particionar el espacio, y un valor aleatorio de offset b.

Dado el número de dimensiones d y un ancho de celda r, se genera a tomando d valores de una distribución estable (de ello su nombre) y b se escoge del rango $[0, r)$. La función de hash corresponde a:

$$h_{a,b}(v) = \left\lfloor \frac{v \cdot a + b}{r} \right\rfloor$$

Concatenando k proyecciones se obtiene un código de hash compuesto por k enteros, lo que entrega una partición más fina del espacio. En la figura 2.15 se puede ver un ejemplo de un espacio en $\mathbb{R}^2$ particionado usando 2 proyecciones.

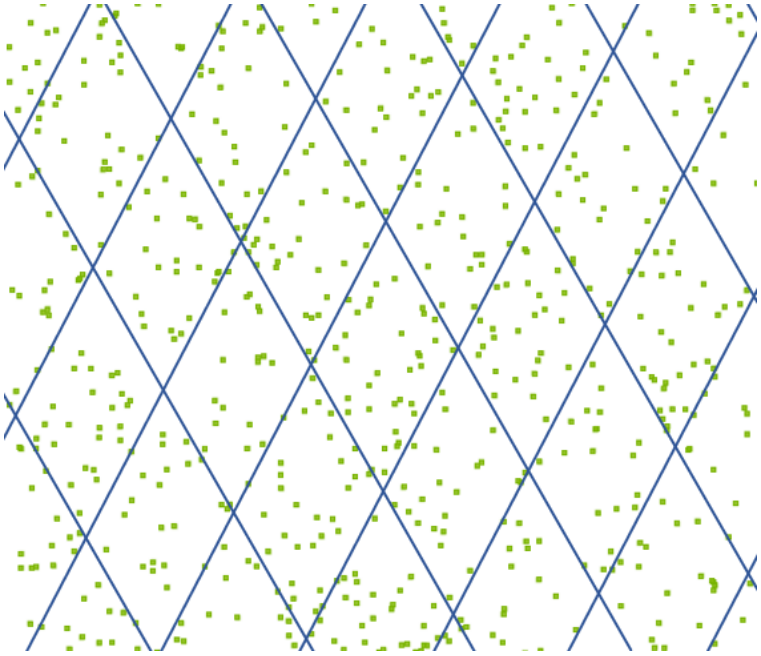


Figura 2.15: División del espacio utilizando 2 proyecciones aleatorias. Los puntos dentro de la misma celda tiene el mismo código de hash.

La intuición es que todos los puntos dentro de un mismo bucket están cerca entre sí, aunque no necesariamente se encontrará el vecino más cercano de cada uno. Para abordar este problema se pueden crear L tablas de hash para obtener más de una partición como se muestra en la figura 2.16.

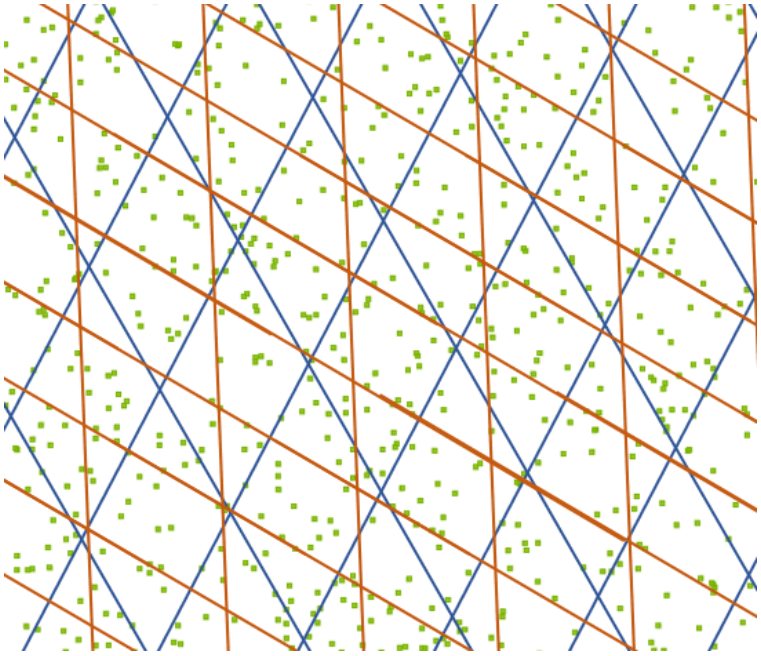


Figura 2.16: División del espacio utilizando 2 tablas de 2 proyecciones aleatorias.

2.5.3. Learning Based Hashing

Una desventaja de LSH es que está limitado debido a que es un algoritmo probabilístico y no toma en cuenta la distribución de los datos, lo que puede llevar a una función de hash con mala distribución. Un enfoque basado en LSH pero que intenta superar estas desventajas es *Learning-Based Hashing* (LBS). Estos métodos intentan aprender una función de hash que se ajuste de mejor manera a la distribución de los datos. El objetivo es encontrar una función de hash $y = h(x)$ tal que la búsqueda de vecinos más cercanos en el espacio codificado sea eficiente y aproxime de manera efectiva a los vecinos más cercanos en el espacio original.

Spectral Hashing

Spectral Hashing [35] fue el primer método de hashing basado en aprendizaje. Su objetivo es encontrar una función de hash tal que objetos similares se mapeen a códigos similares, y que se usen pocos bits en el proceso. El segundo objetivo se traduce en 2 requerimientos, bits balanceados y bits no relacionados. El primero representa que todos los bits tienen un 50 % de probabilidades de ser 1, y el segundo que no haya correlación entre los bits. Estos requerimientos aseguran que todos los bits se aprovechen al máximo.

En vez de solucionar este problema los autores llegan a una aproximación de la solución asumiendo una distribución uniforme de los datos. El algoritmo propuesto utiliza *Principal Component Analysis* y funciones propias para llegar a la solución aproximada.

Scalable Graph Hashing

Scalable Graph Hashing [26] (SGH) es un método que busca aprender códigos de hashing que reflejen la matriz (o grafo) de similitud S. La matriz de similitud de un conjunto de n

elementos es una matriz de tamaño $n \times n$ donde cada elemento está dado por $S_{\mathrm{i},j} = \mathrm{sim}(x_\mathrm{i}, x_j)$, con sim una función de similitud entre 2 vectores con $\mathrm{sim}(x, y) \in \{0, 1\}$. En este trabajo, se utiliza la función de similitud $S_{\mathrm{i},j} = \mathrm{e}^{-\frac{||x_\mathrm{i} - x_j||^2}{\rho}}$, con ρ un parámetro de la función.

Tomando códigos de hashing $b \in \{-1, +1\}^c$ con c el largo del código, el objetivo de SGH es aproximar la matriz de similitud con los códigos aprendidos. Esto se expresa a través de la siguiente función objetivo:

$$\min_{\{b_l\}_{l=1}^n} \sum_{\mathrm{i},j=1}^{n} (\tilde{S}_{\mathrm{i},j} - \frac{1}{c} b_\mathrm{i}^T b_j)^2$$
$$\tilde{S}_{\mathrm{i},j} = 2 \cdot S_{\mathrm{i},j} - 1 \in \{-1, +1\}$$

Realizando algunos cambios de variables y utilizando un *kernel trick* esta función se transforma en:

$$\min_{W} ||c\tilde{S} - \mathrm{sgn}(K(X)W^T)\,\mathrm{sgn}(K(X)W^T)^T||^2$$
$$K(x) = [\phi(x, x_1) - \sum_{\mathrm{i}=1}^{n} \phi(x_\mathrm{i}, , x_1)/n, ..., \phi(x, x_m) - \sum_{\mathrm{i}=1}^{n} \phi(x_\mathrm{i}, , x_m)/n]$$

donde $\phi(x, y)$ es el kernel usado de m bases, en este caso un kernel RBF (gaussiano), y W corresponde a la matriz de pesos para cada uno de los kernel. Para obtener la matriz de pesos W se aplica un algoritmo de aprendizaje iterativo que aprende un bit a la vez, y captura el residuo generado por los bits anteriores en el aprendizaje de los bits siguientes. Este problema residual se relaja para obtener el siguiente problema generalizado de valores propios:

$$K(X)^T R_t K(X) w_t = \lambda K(X)^T K(X) w_t$$
$$R_t = c\tilde{S} - \sum_{\mathrm{i}=1}^{t-1} \mathrm{sgn}(K(X)w_\mathrm{i})\,\mathrm{sgn}(K(X)w_\mathrm{i})^T$$

Finalmente se aplica la siguiente transformación de características para aproximar la matriz de similitud:

$$P(x) = [\sqrt{\frac{2(\mathrm{e}^2-1)}{\mathrm{e}\rho}}\mathrm{e}^{-\frac{||x||^2}{\rho}}x; \sqrt{\frac{\mathrm{e}^2+1}{\mathrm{e}}}\mathrm{e}^{-\frac{||x||^2}{\rho}}; 1]$$
$$Q(x) = [\sqrt{\frac{2(\mathrm{e}^2-1)}{\mathrm{e}\rho}}\mathrm{e}^{-\frac{||x||^2}{\rho}}x; \sqrt{\frac{\mathrm{e}^2+1}{\mathrm{e}}}\mathrm{e}^{-\frac{||x||^2}{\rho}}; -1]$$

$$\text{tomando } \mathrm{e}^x \approx \frac{\mathrm{e}^2-1}{2\mathrm{e}}x + \frac{\mathrm{e}^2+1}{2\mathrm{e}}$$
$$P(x_\mathrm{i})^T Q(x_j) \approx \tilde{S}_{\mathrm{i},j}$$

y luego reemplazando en la parte derecha del problema de valores propios se obtiene:

$$\begin{aligned} cK(X)^T\tilde{S}K(X) &\approx cK(X)^T \underbrace{P(x_\mathrm{i})^T Q(x_j)}_{\tilde{S}} K(X) \\ &= c[K(X)^T P(x_\mathrm{i})^T][Q(x_j)K(X)] \end{aligned}$$

Esta última ecuación es el punto clave de SGH, donde aplicando el kernel usado y la transformación de características se evita calcular explícitamente la matriz de similitud $\tilde{S}$, pero aún así se incluye toda su información de manera implícita en el aprendizaje de los códigos de hashing.

Esta optimización logra que tanto el espacio como el costo asociados al entrenamiento se reduzcan de $O(n^2)$ a $O(n)$, lo que permite utilizar este método en grandes conjuntos de datos con un entrenamiento eficiente y una alta efectividad en los resultados.

Capítulo 3

Solución Propuesta

En este capítulo se describirá en detalle la solución propuesta para el problema de detección de segmentos de videos duplicados y su implementación. Esta se divide en 4 pasos principales: selección de fotogramas clave; extracción de características; búsqueda de vecinos más cercanos; y detección de duplicados.

3.1. Estructura General

El sistema tiene una estructura que ayuda a separar de la mejor manera posible cada paso. Esto permite que sea fácil reemplazar un método dentro de un paso sin afectar el funcionamiento del resto. Además, debido a la gran cantidad de datos, la ejecución de todos los pasos toma varias horas, por lo que se decidió guardar el resultado de cada paso en disco y utilizar eso como la entrada del siguiente paso. De esta manera se puede separar la ejecución en más de un periodo y no perder el progreso cuando ocurre un error.

En la figura 3.1 se muestra el esquema general del sistema. El único paso del cuál no se guarda el resultado parcial es el de selección de fotogramas clave, debido a que es un proceso rápido y cuya salida se puede utilizar inmediatamente para la extracción de características. Para procesar un video, se seleccionan los fotogramas clave y luego se extraen las características de los fotogramas seleccionados inmediatamente.

Todo el código se encuentra disponible en Github [3]. El lenguaje de programación utilizado fue Python 3 [13], y para la mayoría de los métodos se utilizaron librerías como base que serán descritas más adelante. Una librería utilizada transversalmente para el manejo de los datos es NumPy [11], que permite manejar arreglos multi dimensionales de manera eficiente y simple.

Para cada uno de los pasos se creó una clase abstracta que define las funciones que debe tener cada método. Esto permite cambiar el método usado en cada paso sin tener que modificar nada más en el programa principal para que funcione, lo que facilita la ejecución y automatización de cada paso.

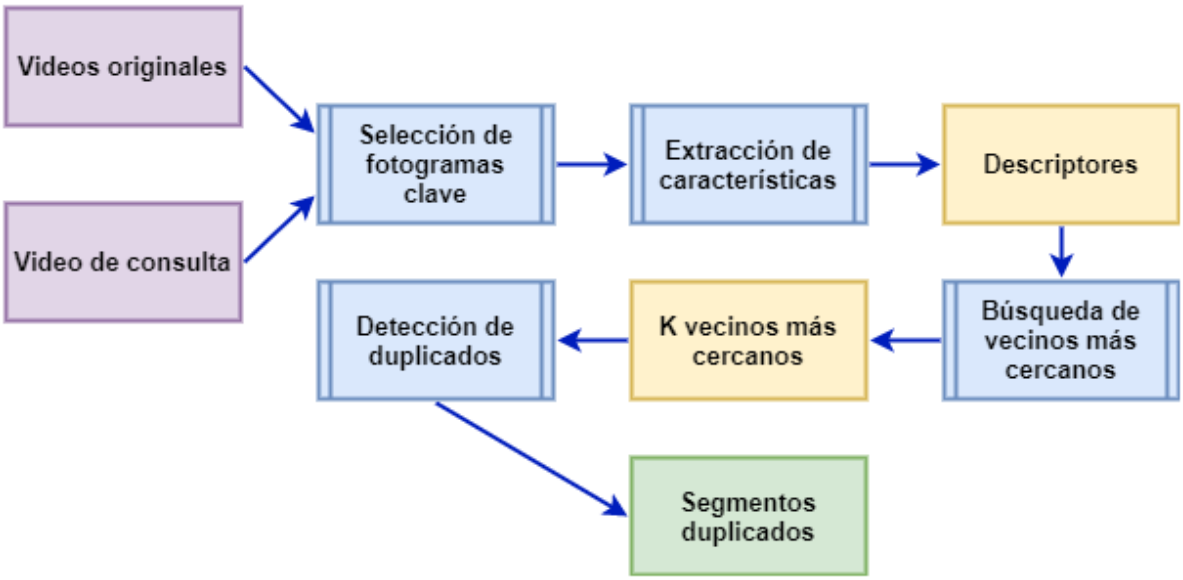


Figura 3.1: Estructura del sistema a desarrollar. Los rectángulos morados son la entrada, los celestes corresponden a las operaciones, los amarillos son resultados parciales que son almacenados en disco y el rectángulo verde es el resultado final.

3.2. Selección de Fotogramas Clave

La selección de fotogramas clave es el proceso mediante el cuál se resume el contenido de un video en un conjunto menor de fotogramas, buscando minimizar la pérdida de información. En este trabajo se tomaron 2 métodos para la selección: reducción de FPS y selección por diferencia de histogramas.

La implementación de estos métodos se realizó utilizando la librería OpenCV [12], que entrega múltiples herramientas de *computer vision* para procesar imágenes y videos.

La estructura de las clases utilizadas en este paso se pueden ver en la figura 3.2. El método `select_keyframes(filename)` es el que se encarga de realizar la selección. Este recibe un nombre de archivo, el video del cuál seleccionar fotogramas claves, y retorna una lista que contiene todos los fotogramas seleccionados. El método `name()` solo se utiliza para automatizar la separación de los datos en carpetas dependiendo del método utilizado.

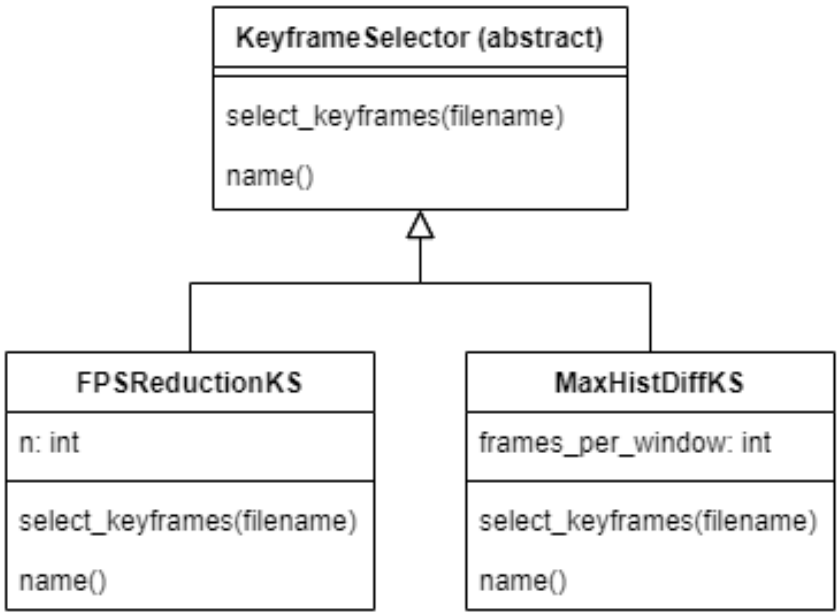


Figura 3.2: Estructura de las clases utilizadas para la selección de fotogramas clave.

3.2.1. Reducción de FPS

La reducción de FPS consiste en la selección de K fotogramas por segundo, distribuidos uniformemente. Con esto no se busca detectar cambios importantes ni momentos claves dentro del video, si no que simplemente reduce la cantidad de datos. La ventaja de este método es que es muy simple de implementar, además de ser rápido en la ejecución.

El único parámetro de este método es n, el número fotogramas que se desea seleccionar por segundo. El método de selección de fotogramas se implementó de manera muy simple, seleccionando 1 de cada s fotogramas del video, con $s = \text{FPS}/n$ siendo FPS la cantidad de fotogramas por segundo del video.

Para este trabajo se eligió reducir a 3 FPS todos los videos, lo que representa un total de 2,065,844 fotogramas en todos los videos, un 12.4 % del total. A este conjunto se le llamará *reduction* de ahora en adelante.

3.2.2. Selección por Diferencia de Histogramas

Este método intenta detectar cambios entre fotogramas continuos para determinar cuáles son los más importantes y así resumir de manera efectiva el video. El algoritmo elegido consiste en tomar una ventana de fotogramas del video, calcular el histograma de color de cada uno, luego calcular la diferencia de histogramas entre todos los fotogramas contiguos y finalmente seleccionar los k fotogramas que producen la mayor diferencia. La figura 3.3 muestra el proceso de selección para $k = 1$.

Para poder detectar movimientos dentro del fotograma, se dividió la imagen en 9, se tomó el histograma de color de cada parte y se concatenaron todos en un solo vector. Además, para

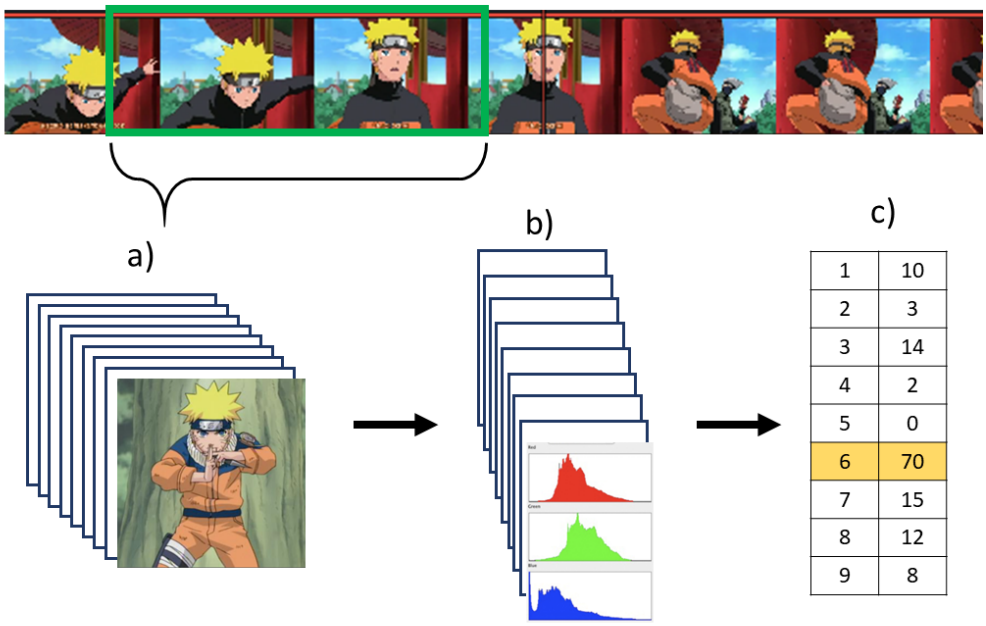


Figura 3.3: Selección de fotogramas clave por diferencia de histogramas. a) Primero se toma una ventana del video. b) Se calculan los histogramas de color de cada fotograma. c) Se calcula la diferencia entre histogramas consecutivos y se toma el fotograma que genere el mayor cambio.

evitar que los cambios en la intensidad de la imagen (luz/sombra) afectaran la selección se normalizaron los vectores. En la figura 3.4 se muestra el proceso mediante el cuál se calcula el descriptor de cada fotograma.

Figura 3.4: Proceso para extraer los histogramas de color de un fotograma para la selección de características. Primero se divide la imagen en 9 partes, luego se extrae un histograma de color de cada una, y finalmente se concatenan todos los histogramas en un solo vector y se normaliza a un vector unitario.

El único parámetro de la clase es `frames_per_window`, que corresponde al número de fotogramas que seleccionar en cada ventana. Cada ventana contiene los fotogramas de 1 segundo de video, y entre ventanas hay 0.5 segundos de separación.

Para calcular los histogramas de los fotogramas se utilizó la función `calcHist` de OpenCV. Esta función entrega un histograma tridimensional que particiona el espacio de color RGB en el número de cubos deseados. Primero se divide la imagen en una grilla de 3x3 y luego se calcula el histograma de color de cada celda usando una partición de 5x5x5, por lo que una imagen es descrita por 3x3 x 5x5x5=1125 valores.

Para seleccionar los fotogramas clave se lee del video y se van ingresando los fotogramas en una lista hasta completar una ventana. Luego usando esa ventana se calcula la diferencia de histogramas de todas las imágenes para seleccionar los máximos locales. Finalmente, se elimina la primera mitad de la lista (esto corresponde a avanzar medio segundo de video) y se sigue leyendo hasta volver a llenarla o que termine el video.

Debido a que los FPS de cada video pueden variar se eligieron ventanas de 1 segundo de largo con espaciados de 0.5 segundos entre ventanas. Se decidió tomar solo 1 fotograma por ventana, lo que representa aproximadamente 2 fotogramas por segundo. Con este método se seleccionaron 1,376,529 fotogramas, lo que equivale a un 8.3 % del total. A este conjunto se le llamará *histDiff* de ahora en adelante.

3.3. Extracción de Características

Corresponde al paso que busca resumir la información de los fotogramas seleccionados en un descriptor simple. Para esto se decidió utilizar los métodos color layout y autoencoder, comparando un método estándar ampliamente utilizado en sistemas RIM con un método más reciente que busca ajustarse a los datos utilizando *deep learning*.

Para implementar los métodos de este paso se utilizó la estructura de clases mostrada en la figura 3.5. El método `extract_features(frames)` es el que realiza el trabajo de este paso. Este recibe como entrada la lista de fotogramas clave seleccionados y retorna una lista de descriptores, uno por cada fotograma en el mismo orden que se recibieron. El método `descriptor_size()` retorna el tamaño del descriptor y se utiliza para realizar operaciones con los descriptores, y `name()` se utiliza para separar los datos en carpetas según el método utilizado.

3.3.1. Color Layout

Color layout es un descriptor simple y rápido de extraer que intenta resumir la información de la distribución del color en una imagen utilizando la transformada coseno discreta.

La única decisión que se tuvo que tomar fue qué tablas de cuantificación utilizar. Se decidió optar por las tablas entregadas en el estándar de JPEG *libjpeg* de 1998 que han sido ampliamente usadas y aprobadas. Estas se pueden ver en la figura 3.6.

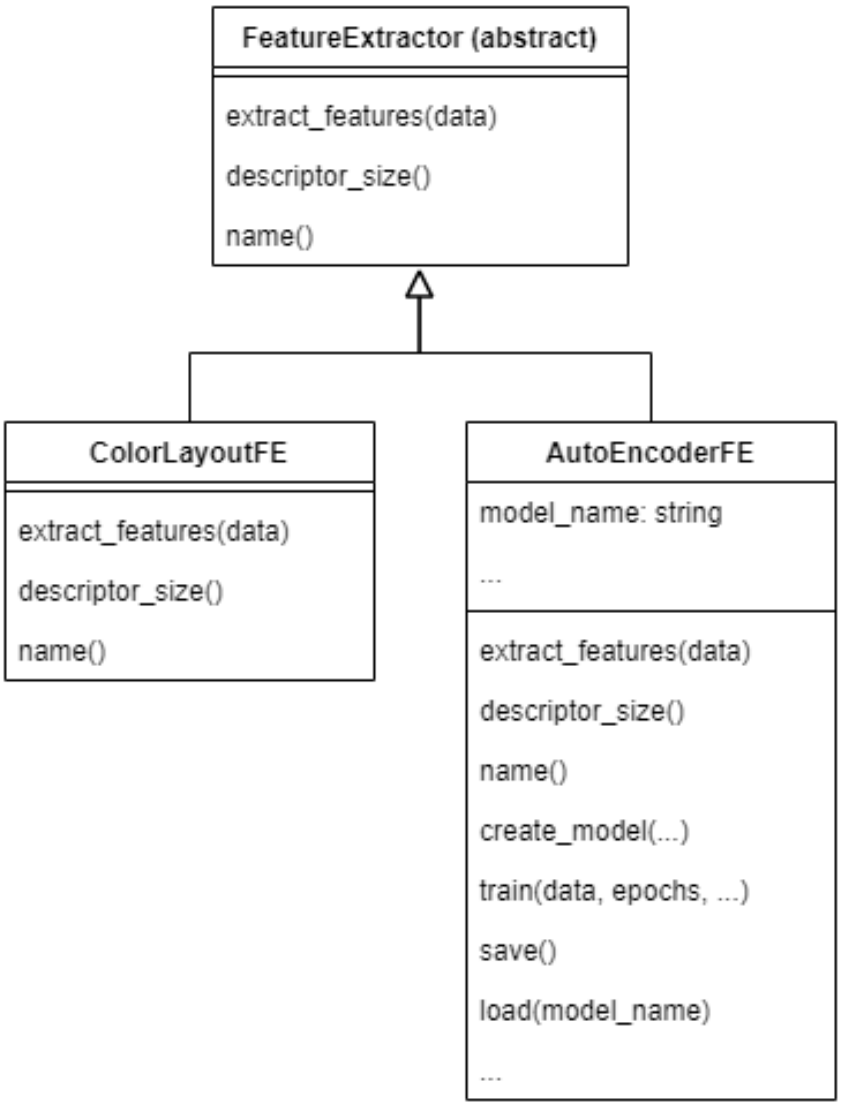


Figura 3.5: Estructura de las clases utilizadas en la extracción de características.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figura 3.6: Tablas de cuantificación entregadas en el estándar de JPEG en libjpeg, 1998. A la izquierda la tabla utilizada para cuantificar la luminancia y a la derecha la de la crominancia. Se puede apreciar que la tabla de la crominancia presenta una mayor compresión, debido a que el ojo humano es más sensible a cambios en la luminosidad de una imagen, se aplica una mayor compresión a los colores.

Para la extracción del descriptor *color layout* se utilizó nuevamente la librería OpenCV, que incluye funciones para escalar imágenes, convertir el modelo de color y calcular la transformada coseno discreta. Teniendo estos métodos, la implementación consistió en aplicar estas funciones siguiendo el esquema estándar, explicado en el marco teórico en la sección 2.3.1, cuidando el formato de los datos para cada función.

El descriptor obtenido corresponde a los 192 coeficientes de la transformada coseno discreta cuantificados utilizando las tablas elegidas, los cuáles se almacenaron en enteros de 8 bits (-128 a 127).

Se extrajeron características de los 2 conjuntos de fotogramas clave y se almacenó el resultado en archivos, obteniendo 614 MB para el conjunto *reduction* (de aquí en adelante *reduction_ CL*) y 409 MB para el conjunto *histDiff* (de aquí en adelante *histDiff_ CL*).

3.3.2. Autoencoder

Un autoencoder es una red neuronal profunda convolucional que busca aprender códigos compactos de un conjunto de imágenes.

Para decidir la arquitectura del autoencoder se realizaron experimentos tomando distintas configuraciones. La estructura general fue utilizar "células" de reducción de tamaño (convoluciones con max-pool) y de aumento de tamaño (convoluciones con up-sampling). Los parámetros de cada célula son:

- Número de convoluciones a usar.
- Tamaño del kernel para la convolución.
- Número de filtros de la convolución.
- Función de activación de la convolución.
- Tamaño del max-pool o up-sampling.

Luego, la red se compone de C células de reducción y C de aumento. Agregando la condición de que la imagen de entrada tenga un tamaño que sea múltiplo de 2^N entonces esta red siempre terminará en una salida del mismo tamaño que su entrada. En la figura 3.7 se muestra una arquitectura utilizando este formato.

Para la implementación se utilizó la librería Keras [7], con Tensorflow [14] como backend. Tensorflow es una librería que permite implementar múltiples algoritmos de aprendizaje de máquinas, incluyendo las redes neuronales convolucionales necesarias en el autoencoder. Keras es una librería que facilita la creación, entrenamiento y prueba de redes neuronales, y que por debajo llama a la funciones de Tensorflow, simplificando todo el proceso para el usuario.

La clase AutoEncoderFE recibe todos los argumentos listados anteriormente para crear el modelo de la red neuronal usando el método `create_model(...)`. Luego, el modelo completo se entrena con el método `train(data, epochs, ...)` que recibe los datos de entrenamiento (data), la cantidad de épocas (epochs), junto a otros parámetros cómo el porcentaje de datos reservados para la validación y el tamaño de cada *batch* de entrenamiento.

Una vez entrenado el modelo se pueden extraer características de forma normal. Para

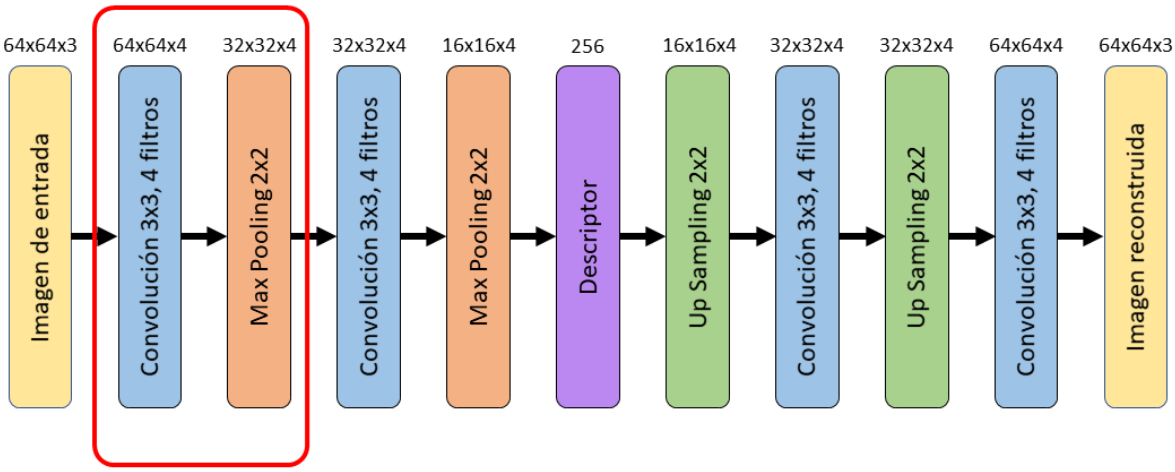


Figura 3.7: Ejemplo de una arquitectura generada utilizando 2 células por lado (la primera célula está encerrada en rojo). Cada célula posee una capa de convolución usando 4 filtros de 3x3 y una capa de max-pool/up-sampling de 2x2.

poder mantener un modelo entre ejecuciones se agregaron los métodos para guardar y cargar un modelo desde archivos, lo que además permite separar el entrenamiento de la red de la extracción de características.

Para evaluar el desempeño de la red al entrenar se comprobó visualmente que tan buenas eran las reconstrucciones de imágenes al pasarlas por la red completa. Se realizaron distintas pruebas de configuración variando los parámetros del modelo hasta obtener resultados satisfactorios. En la figura 3.8 se muestra una reconstrucción lograda por el autoencoder que se utilizó para la extracción de características. En el Anexo A se pueden encontrar más ejemplos de imágenes reconstruidas.

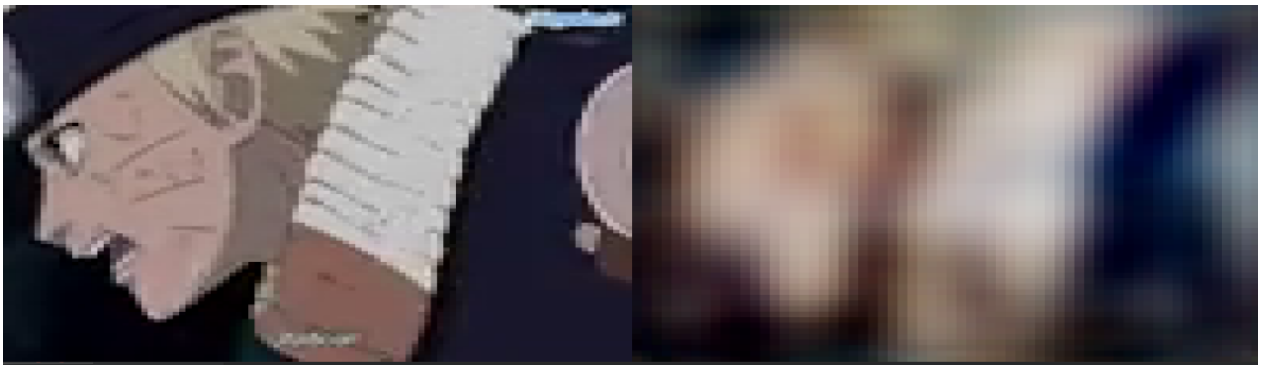


Figura 3.8: Ejemplo de la reconstrucción conseguida por el autoencoder.

En la figura 3.9 se muestra la arquitectura obtenida después de experimentar, por espacio se muestra solo la primera mitad de la red, la segunda mitad es exactamente la misma secuencia pero invertida. La red obtenida está compuesta por 4 células por lado, cada célula

está a su vez compuesta por 2 capas convolucionales de 16 filtros de 3x3, stride 1 y función de activación tangente hiperbólica (no cambian el tamaño de la entrada) y una capa de max pooling o up sampling de tamaño 2x2 (el tamaño se reduce a la mitad o se duplica). La entrada es una imagen RGB de tamaño 64x64, la cual se codifica en un descriptor de 256 variables en la mitad de la red.

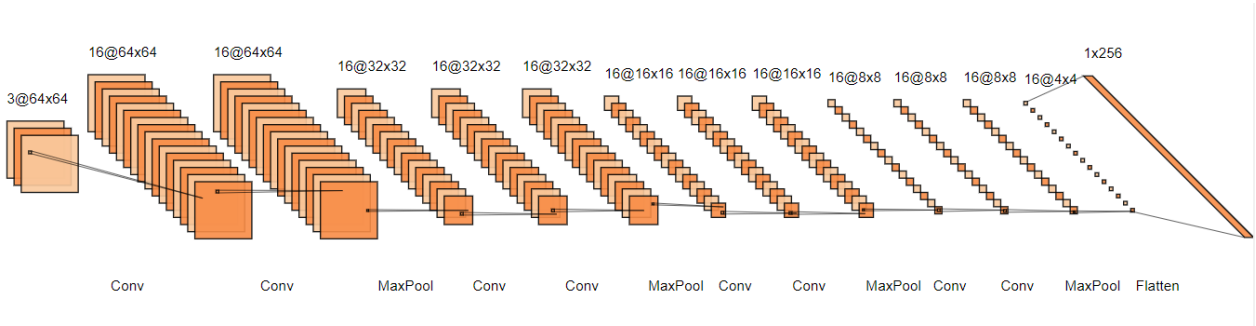


Figura 3.9: Arquitectura del autoencoder utilizado en la extracción de características (primera mitad).

El descriptor obtenido corresponde a 256 números de punto flotante entre -1 y 1, los cuáles se cuantificaron en el rango -128 a 127 para almacenarlos en enteros de 8 bits y así disminuir el uso de memoria.

Se extrajeron características de los 2 conjuntos de fotogramas clave y se almacenó el resultado en archivos, obteniendo 740 MB para el conjunto *reduction* (de aquí en adelante *reduction_AE*) y 493 MB para el conjunto *histDiff* (de aquí en adelante *histDiff_AE*).

3.4. Búsqueda de Vecinos más Cercanos

Se decidió utilizar tres métodos de búsqueda de vecinos cercanos: random k-d trees, y *Locality Sensitive Hashing* utilizando *Random Projections* y *Scalable Graph Hashing*.

Para la implementación se creo la estructura de clases mostrada en la figura 3.10. La función más importante es `search(vector)` que retorna los k vecinos más cercanos del vector de consulta. Las otras 2 funciones se utilizan para la separación de los resultados en carpetas según el nombre del método usado.

Para este paso se utilizaron principalmente 2 librerías: FLANN [2] (Fast Library for Aproximate Nearest Neighbour) y NearPy [4]. FLANN es una librería en C++ con *bindings* para ser usada desde Python, está enfocada en ser muy eficiente, e implementa los árboles de búsqueda aproximada *random k-d trees* y *k-means tree*. NearPy es una librería de *locality sensitive hashing* que permite indexar y buscar vectores utilizando distintas funciones de hashing, además de permitir configurar distintas funciones de distancia y filtros de vectores.

Se optó por extraer 100 vecinos cercanos por fotograma clave, considerando la gran cantidad de datos en cada conjunto.

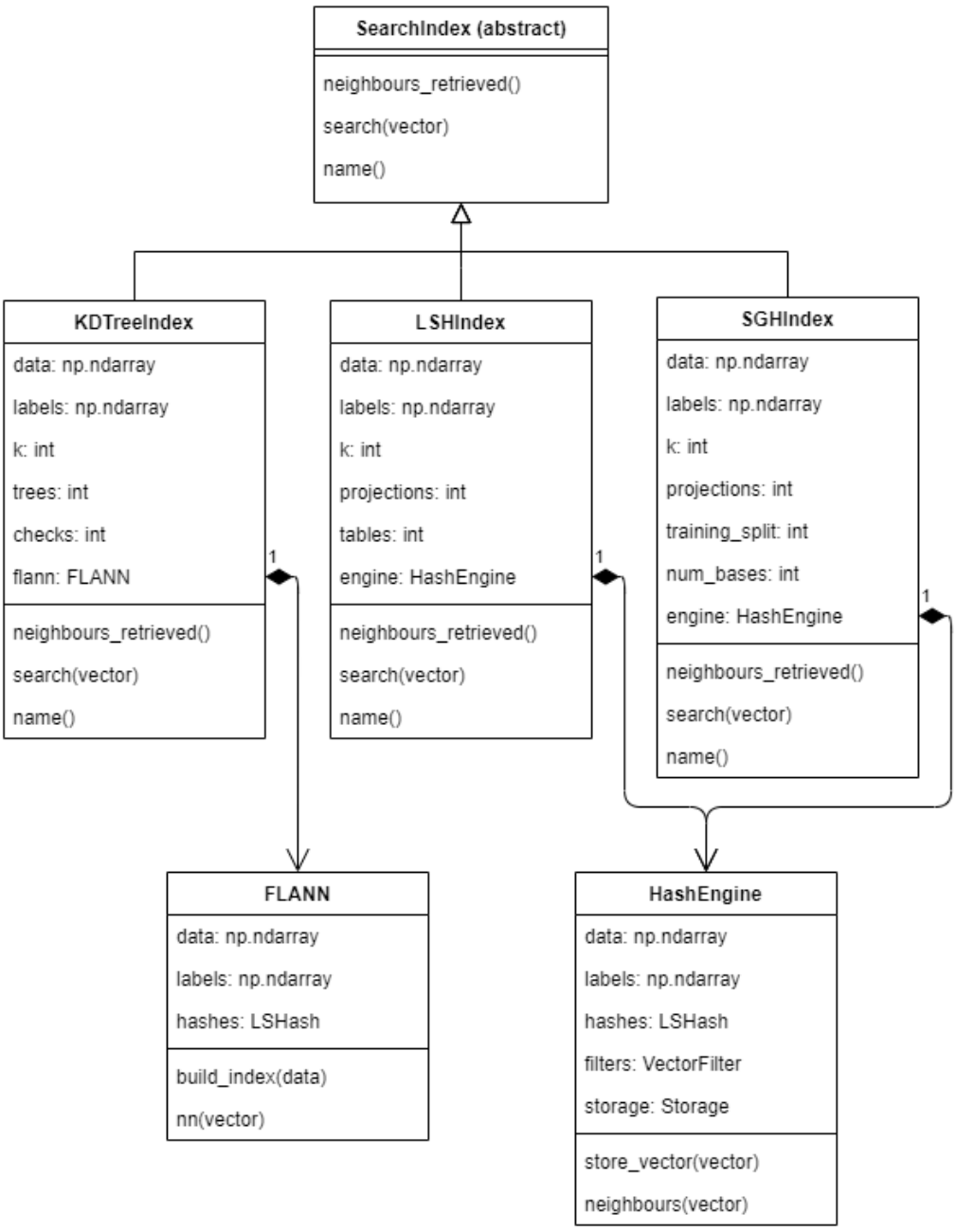


Figura 3.10: Estructura de clases de los índices de búsqueda de vecinos cercanos.

3.4.1. Random k-d Trees

Un k-d tree es una estructura de particionamiento espacial que permite realizar búsquedas de vecinos cercanos sobre un conjunto de puntos multidimensionales de manera eficiente. Su nombre viene de que puede organizar un espacio de k dimensiones. Su funcionamiento en términos generales es: calcular la varianza en cada dimensión del conjunto; tomar la dimensión de mayor varianza; separar el conjunto según la mediana de esa dimensión; y luego iterar recursivamente sobre los nuevos conjuntos creados hasta llegar a un tamaño de hoja deseado. El tomar la mediana de cada dimensión asegura que este sea un árbol balanceado de búsqueda.

El algoritmo de búsqueda de vecinos cercanos utiliza el orden espacial del árbol para recorrer solo las hojas necesarias, entregando un tiempo promedio de búsqueda $O(log(n))$. Sin embargo, a medida que el número de dimensiones aumenta las búsquedas se acercan cada vez más a una búsqueda exhaustiva debido al fenómeno conocido como la *maldición de la dimensión* [19].

Para solucionar este problema, se utiliza un algoritmo de búsqueda aproximado que limita la cantidad de hojas que se revisan (número de chequeos), y se ha comprobado que se obtienen mejores resultados si se utiliza más de un árbol aleatorizando levemente el progreso (por ejemplo, eligiendo al azar entre las 5 dimensiones de mayor varianza en cada nodo).

Se utilizó el índice de búsqueda random k-d Trees como base de comparación. Este método es simple y además entrega buen control sobre el balance entre eficiencia y efectividad. Tomando en cuenta experimentación previa realizada con esta estructura se utilizaron 5 árboles y 1000 chequeos para la búsqueda.

La clase que implementa el índice es simple, ya que la librería FLANN incluye los métodos para crear y realizar búsquedas usando este índice, solo fue necesario crear un adaptador de las funciones de la librería FLANN. Se puede apreciar en el diagrama de clases que el `KDTreeIndex` almacena un objeto de la clase `FLANN`, que es el que se encarga de crear el índice elegido (con `build_index(data, algorithm='kdtree', trees=5)` y luego realizar las búsquedas con `nn(vector, checks)` (de *nearest neighbor*).

3.4.2. Locality Sensitive Hashing

Locality Sensitive Hashing es un método de hashing que permite realizar búsquedas de cercanía en tiempo constante o sub-linear. La base de este método se haya en un conjunto de familias de hashing conocidas como *locality preserving hashes*, estas funciones se caracterizan por aumentar la probabilidad de choque del hash de 2 puntos cuando esos puntos son cercanos. Visto en términos de distancia, tomando 3 puntos en $\mathbb{R}^{\mathrm{d}}$: a, b y c, y una función de hash $h(v) : \mathbb{R}^{\mathrm{d}} \to \mathbb{R}^k$, si $|a-b| \leq |a-c|$ entonces $|h(a)-h(b)| \leq |h(a)-h(c)|$ donde $|x|$ corresponde a la norma euclidiana.

Teniendo una tabla de hash utilizando una de estas funciones, el método de búsqueda es muy simple y consiste en buscar las casillas en las cuales caería el vector de consulta q y tomar todos los puntos encontrados como candidatos a vecinos cercanos. Luego, se puede realizar una búsqueda de fuerza bruta sobre este conjunto para encontrar los vecinos más cercanos. Si es que se eligió bien la función de hashing y la cantidad de bits en el código entonces los elementos deberían quedar bien distribuidos en todas las casillas y realizar la búsqueda final por fuerza bruta no afecta de manera importante el rendimiento.

Existen 2 principales métodos de hashing: *data-independent* y *data-dependent*. Se decidió comparar un método de cada categoría, *Random Projections* como método independiente de los datos y *Scalable Graph Hashing* como método *data-dependent*. Ambos métodos aproximan la distancia euclidiana de los datos, por lo que se consideraron adecuados para realizar las búsquedas.

Para implementar los algoritmos de LSH se utilizó la librería NearPy. Esta incluye todo

lo necesario para almacenar un conjunto de vectores y realizar búsquedas de acuerdo a las funciones de hash definidas.

Sin embargo, al empezar a experimentar con ella se encontró que era muy ineficiente en el almacenamiento de los vectores, ya que guardaba una copia de cada vector en todas las tablas de hash utilizadas. Esto es, si se utilizaban 5 tablas de hash entonces cada vector estaba siendo almacenado 6 veces, debido a la gran cantidad de datos esto impactaba notoriamente el desempeño del algoritmo.

Para solucionar este problema se re-implementó el motor principal de la librería para guardar una copia de los datos y en las tablas de hash guardar solamente el índice de los vectores. De esta forma, se minimiza el espacio ocupado cambiando solo algunos métodos de la clase, manteniendo el resto de la estructura que permite elegir funciones de hashing, distancia utilizada y filtros.

Stable Distributions

Para probar la eficiencia de *Locality Sensitive Hashing* usando un método simple y estándar se utilizó la familia de hashing *stable distributions* (también conocido como random discretized projections).

Esta función de hash es fácil de configurar para variar el número de vectores dentro de cada casilla variando el ancho de las particiones r, y al usar más de un conjunto de proyecciones aleatorias es más probable encontrar los vectores que se encuentran cerca del vector de consulta.

Para la implementación se utilizó la clase que viene incluída con la librería NearPy, por lo que solo fue necesario crear un *wrapper* del `HashEngine` para poder ajustarse a la interfaz de funciones definidas para este paso.

Al experimentar utilizando este método se encontró que la distribución de los descriptores de ColorLayout está muy cargada a ciertas dimensiones, en la figura 3.11 se puede apreciar que las dimensiones correspondientes al primer coeficiente de cada transformada coseno discreta presentan la mayor varianza, disminuyendo a medida que se avanza a los coeficientes de frecuencias más altas (llegando a tener varianza 0 para algunas dimensiones).

Esto produce que la distribución de los vectores en la tabla de hash sea desbalanceada, quedando unas pocas casillas con la mayoría de los vectores mientras que el resto almacena solo unos poco vectores (menos de 10 en promedio). Finalmente se decidió usar 10 tablas con 5 proyecciones y ancho de celda 20, esto retorna en promedio entre 10.000 y 20.000 candidatos por búsqueda y un máximo de hasta 100.000, pero aún así mantiene el mínimo de candidatos bajo los 100. Al intentar mejorar este mínimo se llegó a realizar búsquedas secuenciales sobre más de 100.000 datos, por lo que dejaba de tener sentido utilizar un índice en vez de el método fuerza bruta.

Por otra parte, los descriptores extraídos por el autoencoder presentan una distribución más uniforme, como se puede apreciar en la figura 3.12. Para este conjunto se utilizaron 10 tablas de hashing, cada una con 3 proyecciones con ancho de celda 20. Esta configuración

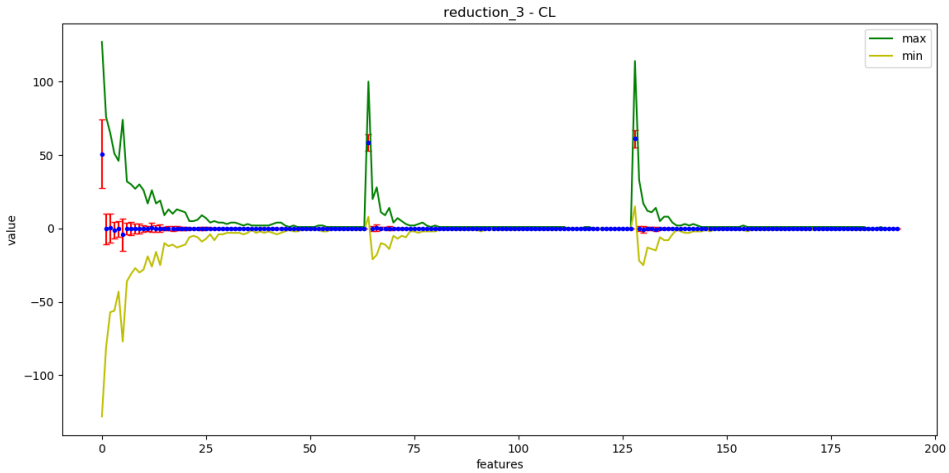


Figura 3.11: Distribución de los descriptores obtenidos por el método color layout en el conjunto *reduction_3*. En verde y amarillo se muestra el máximo y mínimo valor en cada dimensión respectivamente, en azul se muestra el promedio y en rojo la desviación estándar. los 3 valores con alta varianza corresponden al primer coeficiente de cada transformada coseno discreta.

entregó entre 1.000 y 3.000 candidatos por búsqueda en promedio y un máximo de 10.000, lo que hizo las búsquedas considerablemente más rápidas que en los descriptores de color layout.

Scalable Graph Hashing

Scalable Graph Hashing es un método que utiliza una transformación de características para crear una aproximación de la matriz de similitud de los puntos de entrenamiento, y utilizarlo para aprender códigos de hashing ajustados a los datos. La principal fortaleza de este método es que en ningún momento se calcula explícitamente la matriz de similitud, lo que reduce el tiempo y espacio requeridos de $O(n^2)$ a $O(n)$. De ello que el método es escalable para conjuntos grandes de datos manteniendo la efectividad de hashing basado en el grafo de similitud.

Para la implementación de Scalable Graph Hashing se utilizó un código de ejemplo entregado por los autores [5]. Se tuvo que adaptar para poder ser utilizado con el `HashEngine` utilizado, por lo que se separaron las fases de entrenamiento y la de predicción. La implementación utiliza NumPy para la mayoría de los cálculos matriciales realizados, y SciPy para resolver los problemas de vectores propios.

Siguiendo la recomendación de los autores se utilizaron 300 bases en el *kernel* para la transformación de características. La experimentación inicial demostró que los descriptores se distribuían de una manera casi uniforme sobre los códigos de hashing, lo que permitió utilizar código muy compactos sin impactar notoriamente la velocidad de búsqueda.

Se decidió usar 10 proyecciones, lo que entrega 1024 casillas con 1000 a 2000 vectores en cada una en promedio. Esto representa una gran mejora con respecto al hashing realizado

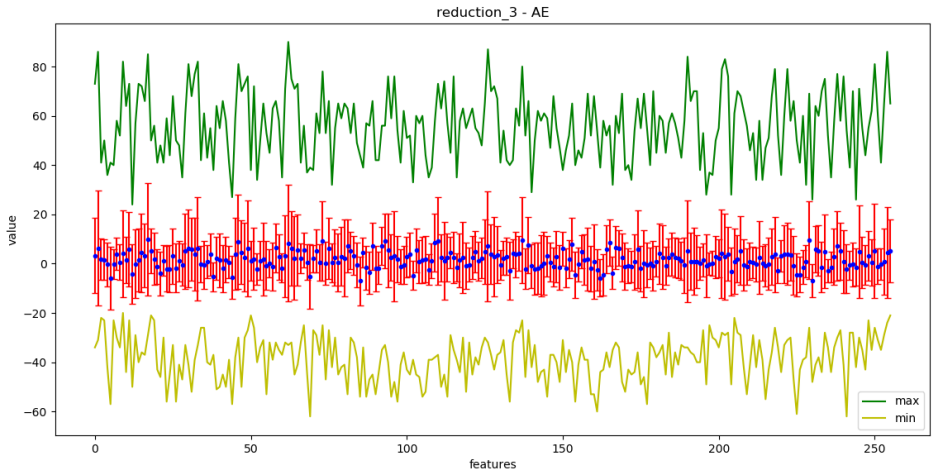


Figura 3.12: Distribución de los descriptores obtenidos por el métodos autoencoder en el conjunto *reduction_3*. En verde y amarillo se muestra el máximo y mínimo valor en cada dimensión respectivamente, en azul se muestra el promedio y en rojo la desviación estándar.

con *stable distributions*, en el cual fue más difícil encontrar un buen balance entre número de candidatos y tiempo de búsqueda.

3.5. Detección de Duplicados

En este trabajo cada fotograma se identifica por una tupla que contiene el video al que corresponde y su posición con relación al resto de los fotogramas clave del video, i.e. el fotograma (417, 46) corresponde al 46-avo fotograma clave seleccionado en el video 417.

Una lista de vecinos cercanos es un fotograma junto con su lista de los fotogramas más cercanos obtenidos en la búsqueda. A continuación se muestra un ejemplo de la lista de vecinos cercanos del fotograma (12, 1).

$$(12,\ 1) \longrightarrow [(15,\ 5),\ (29,\ 7),\ (15,\ 1),\ ...]$$

Teniendo las listas de vecinos cercanos de todos los fotogramas, la detección de segmentos duplicados se reduce a buscar secuencias temporalmente consistentes que tengan una alta cantidad de calces.

Por ejemplo, si se tuviera la lista de vecinos cercanos mostrada en la Tabla 3.1, se puede notar que existe una secuencia continua que calza entre los fotogramas (12, 1) a (12, 6) y (15, 5) a (15, 10), lo que representaría un duplicado entre el video 12 y el 15.

fotograma	vecinos cercanos			
(12, 1)	(15, 5)	(29, 7)	(15, 1)	...
(12, 2)	(13, 2)	(24, 6)	(15, 6)	...
(12, 3)	(18, 4)	(15, 7)	(25, 23)	...
(12, 4)	(01, 1)	(15, 8)	(04, 6)	...
(12, 5)	(15, 9	(05, 8)	(09, 11)	...
(12, 6)	(10, 9)	(10, 4)	(15, 10)	...
...	...			

Tabla 3.1: Lista de vecinos cercanos hipotética para un video 12. Es fácil notar que existe una secuencia temporalmente consistente entre (12, 1) a (12, 6) y (15, 5) a (15, 10), lo que indicaría que estos segmentos son duplicados.

En la práctica, no se encuentran estas secuencias completas, si no que existen segmentos vacíos o desfasados. Además, no siempre los duplicados van a estar al inicio del video, si no que pueden ocurrir en cualquier momento y ser de un largo arbitrario.

Para poder asegurar una buena detección de duplicados se decidió realizar una búsqueda exhaustiva sobre las listas de vecinos cercanos, tomando cada fotograma en la lista como el inicio de un candidato de duplicado. El procedimiento para detectar segmentos duplicados se describe a continuación:

1. La entrada del algoritmo es una lista de todas las listas de vecinos cercanos de un video, ordenadas cronológicamente
2. Inicialmente se tiene una lista de candidatos y una lista de duplicados vacía.
3. Para cada **fotograma_actual** junto con su **lista de vecinos cercanos** se realizan los siguientes pasos:
   (a) Para cada candidato en la lista se chequea si es que su siguiente fotograma esperado se encuentra en la **lista de vecinos cercanos** actual. Si es que se encuentra, se aumenta la cantidad de fotogramas encontrados y la duración del candidato, y se reinicia la racha de fotogramas no encontrados a 0. Si no se encuentra, solamente se aumenta la cantidad de fotogramas no encontrados y la racha de no encontrados actual.
   (b) Para cada candidato se chequea si es que este se debe dar por terminado o no. El criterio para decidir esto es si es que no se ha encontrado el fotograma esperado en las últimas k listas de vecinos cercanos, o sea que su racha de no encontrados es k. Los candidatos determinados como terminados se eliminan de la lista de candidatos y entran en otra lista temporal.
   (c) Para cada candidato terminado, se chequea si la duración de este es mayor a una duración mínima y que la relación entre fotogramas encontrados y no encontrados es mayor a un límite definido para decidir si se considera como un duplicado o no. Los candidatos que pasan este filtro se agregan a la lista de duplicados.
   (d) Finalmente, para cada **fotograma_cercano** en la **lista de vecinos cercanos** actual se chequea si es que este fue el último fotograma encontrado en alguno de los candidatos actuales. Si es que no lo es, entonces se agrega un nuevo candidato como

un duplicado entre el **fotograma_actual** y **fotograma_cercano** con duración 0.

4. Al finalizar este proceso se tiene una lista de segmentos duplicados, sin embargo esta contiene ocurrencias que se encuentran repetidas o que fueron detectadas en más de un segmento separado. Por esto, se realiza un procesamiento final sobre la lista que combina los duplicados que son cercanos entre sí y elimina todas las ocurrencias repetidas.

Un candidato de duplicado posee los siguientes atributos:

- El video que se está revisando actualmente.
- El índice del fotograma clave donde comienza el duplicado en el video actual.
- El video original del duplicado.
- El índice del fotograma clave donde comienza el duplicado en el video original.
- La duración del segmento duplicado.
- El índice del próximo fotograma del video original que se espera ver.
- La cantidad de fotogramas encontrados y no encontrados que calzan con este candidato.
- La racha de fotogramas no encontrados actual, i.e. si es que no se han encontrado calces de los últimos 5 fotogramas esperados, este valor es 5.

En la figura 3.13 se muestra un ejemplo del proceso de detección en un candidato. El candidato comienza en la segunda fila como una duplicado entre los fotograma (417, 002) y (420, 116). El fotograma esperado en la siguiente fila es el (420, 117) que no está, y luego los siguientes 2 fotogramas si se encuentran en las listas de vecinos cercanos. El estado actual del candidato sería el siguiente:

- video actual: 417
- fotograma inicio: 116

- video original: 420
- fotograma inicio original: 002

- duración: 4
- encontrados: 3
- no encontrados: 1
- racha no encontrados: 0

El candidato se declara terminado cuando su racha de no encontrados es mayor a 6 y se considera un duplicado solo si su duración es mayor a 5 segundos. Para convertir la duración del candidato de fotogramas a segundos se almacenan las marcas de tiempo de los fotogramas clave.

Este proceso se repite iniciando un candidato cada vez que se encuentra un fotograma que no pertenece a un candidato actual, lo que entregaría las secuencias resaltadas en la figura 3.14. El resto de los fotogramas no logra formar cadenas lo suficientemente largas para considerarse duplicados.

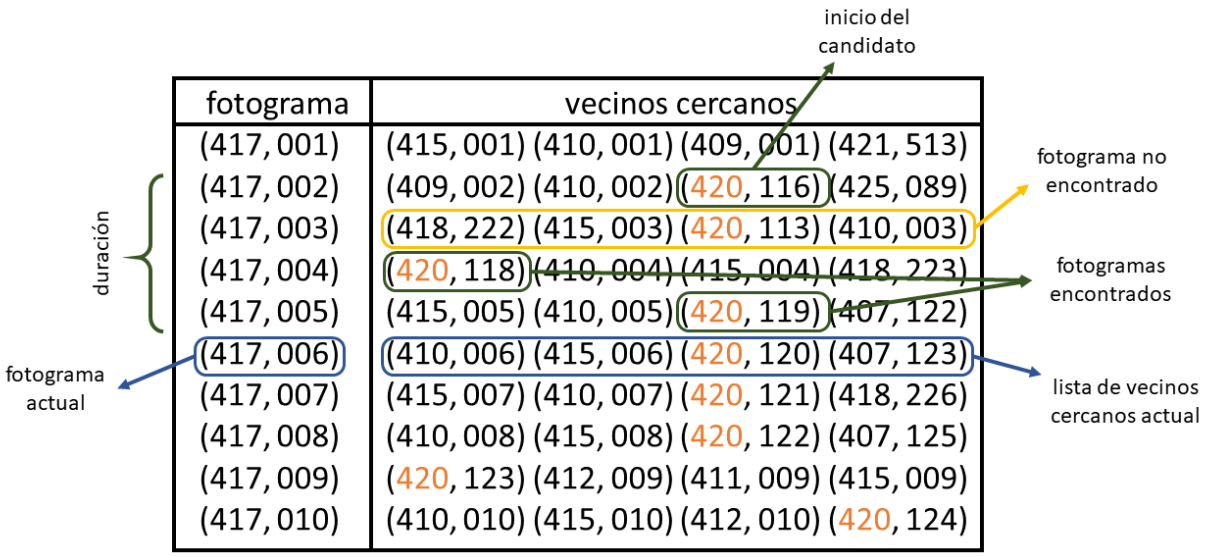


Figura 3.13: Ejemplo del procedimiento para detectar un candidato de segmentos duplicados.

fotograma	vecinos cercanos
(417, 001)	(415, 001) (410, 001) (409, 001) (421, 513)
(417, 002)	(409, 002) (410, 002) (420, 116) (425, 089)
(417, 003)	(418, 222) (415, 003) (420, 113) (410, 003)
(417, 004)	(420, 118) (410, 004) (415, 004) (418, 223)
(417, 005)	(415, 005) (410, 005) (420, 119) (407, 122)
(417, 006)	(410, 006) (415, 006) (420, 120) (407, 123)
(417, 007)	(415, 007) (410, 007) (420, 121) (418, 226)
(417, 008)	(410, 008) (415, 008) (420, 122) (407, 125)
(417, 009)	(420, 123) (412, 009) (411, 009) (415, 009)
(417, 010)	(410, 010) (415, 010) (412, 010) (420, 124)

Figura 3.14: Segmentos duplicados al finalizar el procedimiento de detección en el ejemplo anterior.

La razón por la que existen tantos duplicados en este ejemplo es porque corresponde a la introducción del capítulo (o *opening*), que se reutiliza por varios capítulos generando estos duplicados. Para detectar correctamente todas sus ocurrencias se utilizaron listas de vecinos más grandes en el algoritmo real.

Para facilitar la implementación del algoritmo se creó la clase `Candidate`, que almacena toda la información de un candidato e incluye métodos para manejarlos. Los atributos de la clase son los que se acaban de enumerar y se pueden ver en la figura 3.15.

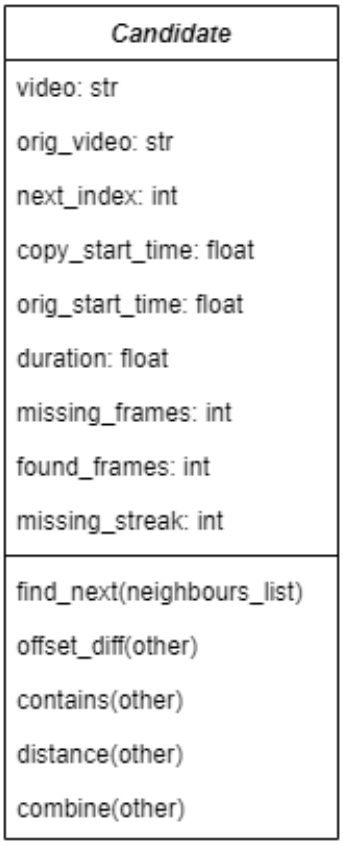


Figura 3.15: Clase utilizada para la detección de duplicados en la fase final del trabajo.

El método `find_next(neighbours)` recibe como entrada la lista de vecinos más cercanos del fotograma actual y busca el siguiente fotograma esperado por su índice (`next_index`). Si lo encuentra reinicia la racha de no encontrados (`missing_streak`) y aumenta en 1 la cantidad de fotogramas encontrados (`found_frames`). Si no lo encuentra aumenta en 1 la cantidad de fotogramas no encontrados (`missing_frames`) y la racha de no encontrados.

Los métodos `offset_diff(other)`, `contains(other)` y `distance(other)` permiten comparar 2 candidatos de duplicados para decidir qué hacer con ellos.

La función `contains(other)` determina si un candidato está completamente contenido dentro de otro, lo que indicaría que es innecesario conservarlo. En la figura 3.16 se muestra un ejemplo en el que un duplicado queda completamente contenido dentro de otro.

La función `distance(other)` retorna la distancia (en segundos) entre los rangos de los candidatos en el video en el que ocurre la copia. En la figura 3.17 se muestran ejemplos del cálculo de distancia, si es que la distancia es 0, entonces los duplicados tienen por lo menos un fragmento superpuesto en el video de la copia. Si es mayor a 0 entonces están separados.

La función `offset_diff(other)` calcula cuál es la diferencia absoluta entre los *offsets* de los candidatos. El *offset* de un candidato representa la diferencia entre el tiempo de inicio en el video de copia y el original. Por ejemplo, si un candidato empieza en el segundo 157 del video 1 y en el 189 del video 2, su offset es de 32 segundos. Si otro candidato tiene un offset similar, entonces su `offset_diff` es cercano a 0 y esto significa que ambos calzan temporalmente. Si además de eso tienen una distancia cercana a 0, entonces estos 2 duplicados se pueden combinar en solo 1.

La función `combine(other)` se encarga de combinar 2 candidatos en 1 solo que abarque el rango de los 2 juntos. Para ello se toma el comienzo mínimo entre los 2 y se toma la duración combinada de los 2 (sin contar la duración superpuesta que puedan tener). En la figura 3.19 se muestra el resultado de combinar los candidatos 1 y 2 de la figura 3.18.

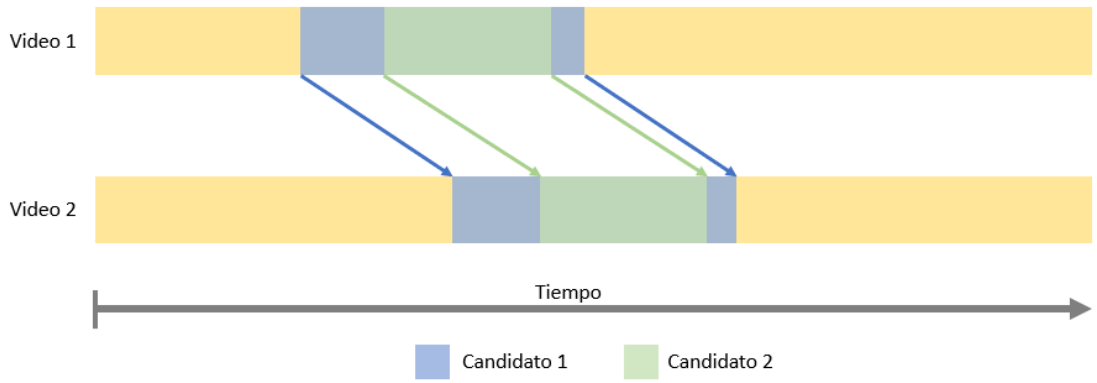


Figura 3.16: Ejemplo de 1 duplicado contenido completamente dentro de otro. En este caso, el candidato 2 queda contenido dentro del candidato 1, por lo que se elimina de las copias reportadas.

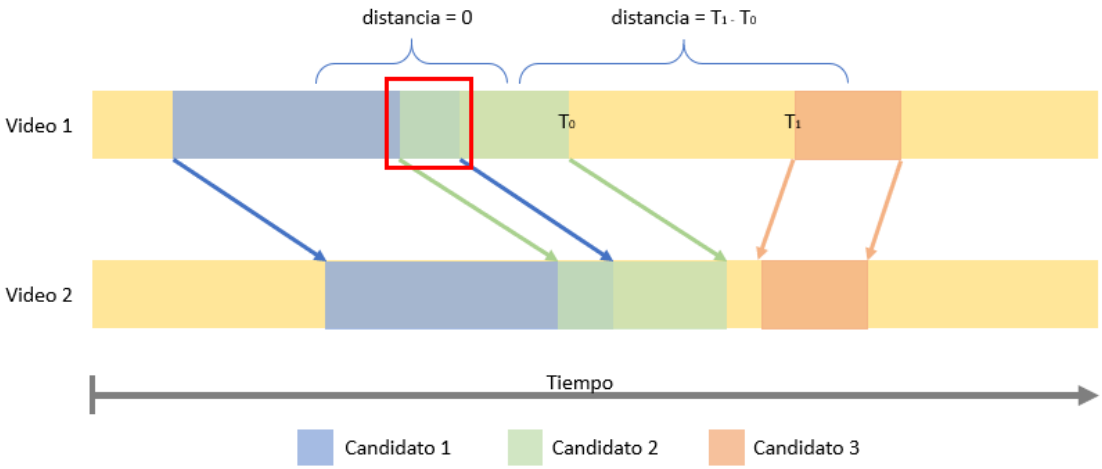


Figura 3.17: Cálculos de distancia entre candidatos. Si 2 candidatos están superpuestos, como los candidatos 1 y 2, su distancia es 0. De lo contrario se calcula como la distancia (en segundos) entre los rangos.

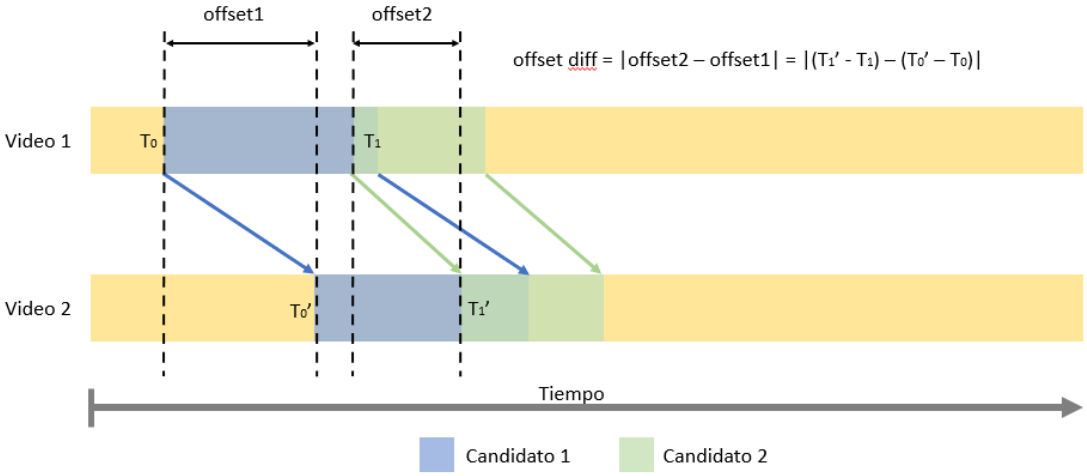


Figura 3.18: visualización de la diferencia de offset entre candidatos. Cuando la distancia y la diferencia de offset entre 2 candidatos son cercanas a 0, entonces estos candidatos se pueden combinar para obtener un duplicado de mayor duración.

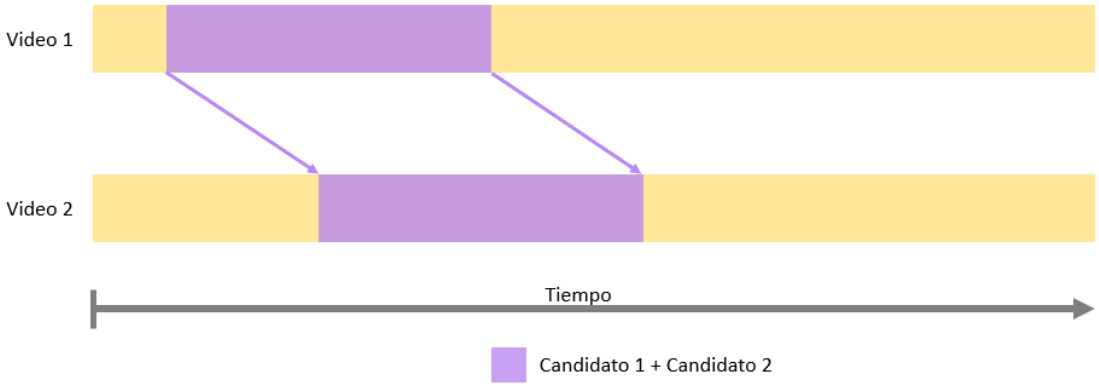


Figura 3.19: Resultado de combinar los candidatos 1 y 2 de la figura 3.18.

Capítulo 4

Resultados y Análisis

En la siguientes secciones se muestran primero los resultados de eficiencia y luego los resultados de efectividad obtenidos con los métodos probados. Finalmente se analizan estos resultados y algunos casos especiales encontrados.

4.1. Eficiencia

Para cada uno de los métodos utilizados se midió el tiempo que tomaba cada acción, a continuación se muestra la eficiencia de cada uno separada por el paso al que corresponden.

Para calcular la eficiencia de cada método se promedió el tiempo de procesamiento en todos los conjuntos de datos utilizados. Por ejemplo, para el caso de un índice, se promedió el tiempo de búsqueda y construcción en los 4 conjuntos de características extraidos. (`reduction_3_CL`, `reduction_3_AE`, `histDiff_1_CL`, `histDiff_1_AE`).

4.1.1. Selección de Fotogramas Clave

En la figura 4.1 se muestra la eficiencia los distintos métodos de selección de fotogramas clave. De las 2 técnicas, la reducción de fps es notoriamente más rápida que la selección por diferencia de histogramas. Esto se debe a que en la diferencia de histogramas se deben realizar múltiples operaciones para calcular el descriptor de histogramas de cada imagen y luego obtener el que produce la máxima diferencia. Aún así, el método por diferencia de histogramas sigue siendo muy eficiente, procesando un video de 23 minutos con 24 FPS en menos de 10 segundos. En el caso del método de reducción de fps, procesar un video de este tamaño le toma 2.5 segundos en promedio.

4.1.2. Extracción de Características

En la figura 4.2 se muestra la eficiencia los distintos métodos de extracción de características. En la extracción de características el descriptor color layout fue el más eficiente, extrayendo características un 60 % más rápido que el autoencoder. Esto se debe a lo profundo que se implementó el autoencoder, considerando que para codificar una imagen esta se

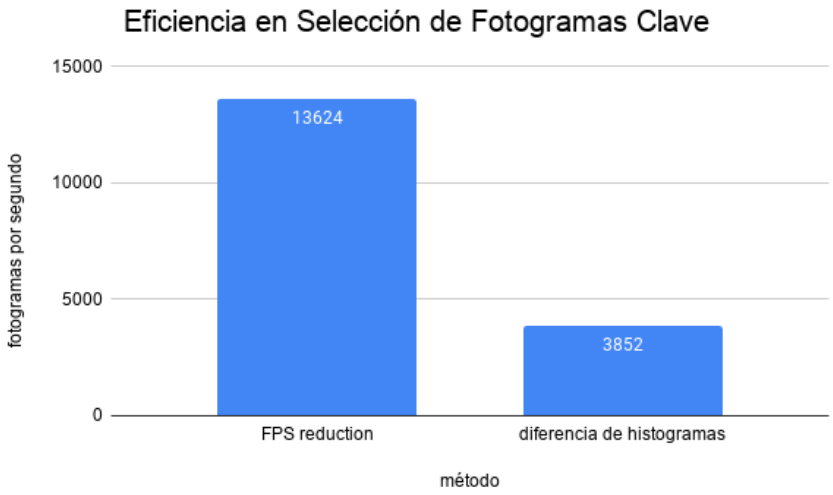


Figura 4.1: Fotogramas promedio procesados por segundo por los selectores de fotogramas clave.

pasaba por 12 capas de convoluciones y max pooling, pero se podría buscar una implementación más rápida tratando de mantener los resultados obtenidos.. Ambos pueden procesar videos de forma rápida, considerando un video de 23 minutos seleccionando 3 fotogramas por segundo, al método color layout le toma 4.5 segundos extraer sus características, mientras que al autoencoder le toma 7.3 segundos.

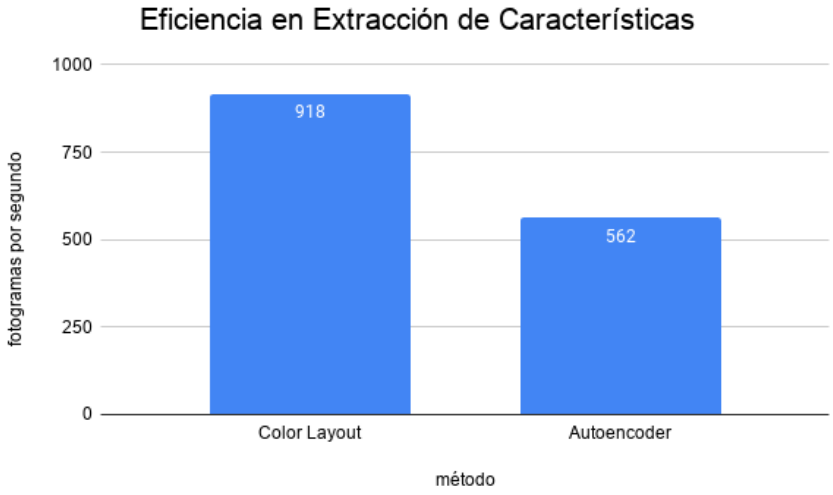


Figura 4.2: Fotogramas promedio procesados por segundo por los métodos de extracción de características.

Cabe notar que en el tiempo de extracción de características del autoencoder no se consideró el tiempo de entrenamiento de este. Esto debido a que este paso se realizó de forma autónoma solo una vez, y después solo era necesario cargar el encoder para utilizarlo. El entrenamiento de la red tomó 2 horas y 13 minutos utilizando una CPU y 50 épocas, esto

podría hacerse de manera más rápida en GPU, pero no se contaba con el hardware necesario para hacerlo.

Considerando la selección de fotogramas clave y la extracción de características juntas, cada capítulo era procesado en menos de 20 segundos. Tomando los métodos más eficientes toma solo 7 segundos por capítulo en promedio, lo que significa extraer características de todos los capítulos de Naruto Shippuden en aproximadamente 1 hora.

4.1.3. Búsqueda de Vecinos más Cercanos

Al comparar los tiempos de indexado de vectores el índice k-d tree fue el más rápido por una gran diferencia, por otra parte, los índices locality sensitive hashing (con stable distributions) y scalable graph hashing tuvieron velocidades similares, con SGH siendo el más lento de todos. Uno de los factores que influye en la diferencia de tiempos es el hecho de que el k-d tree está programado en C++, siendo este lenguaje de programación conocido por ser más eficiente. Sería necesario realizar una implementación de todos los métodos en el mismo lenguaje para poder comparar efectivamente su eficiencia, pero no se pudo hacer con el tiempo disponible para este trabajo. Por otro lado, el índice SGH es el más lento debido a la gran cantidad de operaciones matriciales que se realizan para calcular el hash de cada vector, debido a la *feature transformation* aplicada a cada punto.

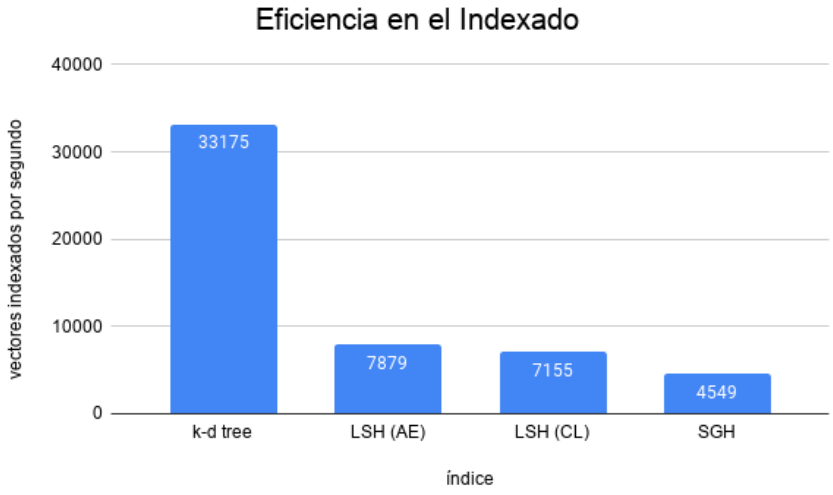


Figura 4.3: Velocidad de indexado promedio en los distintos índices de búsqueda utilizados.

Al comparar la velocidad de búsqueda, el índice k-d tree se repite como el índice más rápido con una ventaja considerable, sin embargo, el índice más lento es el de locality sensitive hashing que se utilizó en los conjuntos de características extraídos utilizando color layout (LSH-CL).

Nuevamente se atribuye el rápido funcionamiento del índice k-d tree a estar implementado en C++, que fue pensado para ser eficiente. En el otro extremo, el mal rendimiento del índice LSH-CL se atribuye a la desbalanceada distribución de los datos, que causa que algunas de

las proyecciones aleatoriamente elegidas puedan no crear una buena división de los datos, empeorando la calidad de la función de hashing.

Finalmente, cabe notar que a pesar de ser más lento en su construcción, el índice SGH es más rápido que el LSH sobre los conjuntos de características extraídas por el autoenconder (LSH-AE). Esto gracias a que tener una distribución balanceada sobre la tabla de hash permite regular de muy buena manera la cantidad de candidatos por búsqueda, que finalmente reduce el tamaño de las búsquedas secuenciales realizadas y con ello el tiempo de búsqueda.

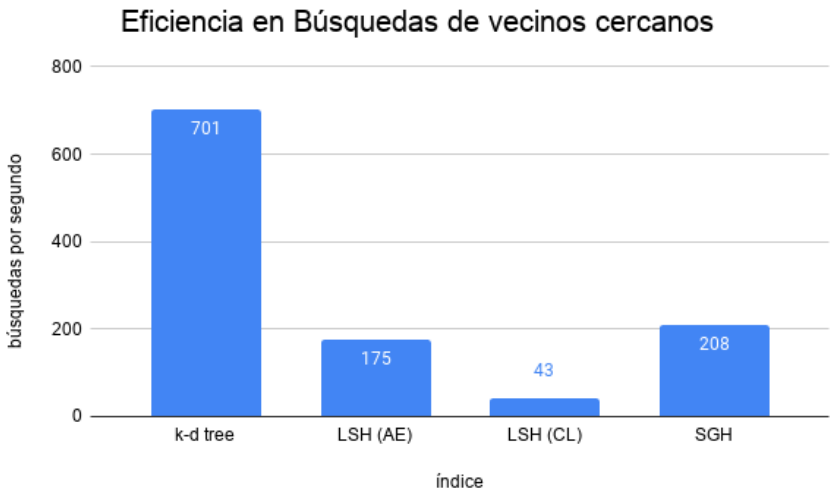


Figura 4.4: Velocidad de búsqueda promedio en los distintos índices de búsqueda utilizados.

El tiempo promedio para procesar un capítulo fue de 5.9 segundos al utilizar random k-d trees, 19.9 segundos con SGH, 23.6 segundos con LSH-AE y 95.6 segundos con LSH-CL.

4.1.4. Detección de Duplicados

El método de detección de duplicados procesó 60 listas de vecinos cercanos por segundo, lo que es equivalente a demorarse 70 segundos en un capítulo extrayendo 3 fotogramas clave por segundo. Este valor variaba levemente dependiendo de la cantidad de fotogramas claves extraídos en el capítulo y la cantidad de recapitulaciones en él, llegando a un mínimo de 20 listas por segundo y un máximo de 90.

4.1.5. Proceso Completo

Considerando el proceso completa de detección de duplicados, comparando el tiempo dedicado en cada paso según el método en la figura 4.5, se puede ver que el paso más costoso es la detección de duplicados. Esto podría acelerarse más utilizando un método que no realice una búsqueda de manera tan exhaustiva, pero debido a que el objetivo era evaluar las técnicas de RIM se buscó obtener los mejores resultados posibles para cada parte.

El único método que supera a la detección en tiempo de procesamiento es LSH sobre el conjunto de datos extraídos usando color layout, que se debe a la mala distribución de los

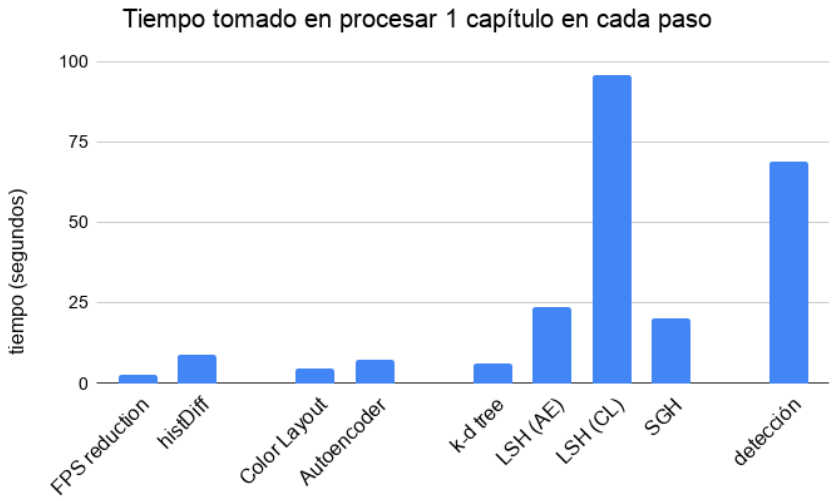


Figura 4.5: Tiempo que toma cada método en procesar un capítulo, ordenados por pasos, asumiendo videos de 23 minutos con una extracción de 3 fotogramas por segundo.

datos mencionada anteriormente.

Tomando en cuenta el método más eficiente en los primeros 3 pasos, esto es, utilizando FPS reduction para la selección de keyframes, color layout para la extracción de características y el índice de búsqueda random k-d trees, el tiempo total necesario para encontrar los vecinos cercanos de un capítulo sería de 13 segundos. Esto sumaría un total de menos de 2 horas para procesar todos los capítulos de *Naruto Shippuden*, faltando solo el paso de la detección de segmentos duplicados, para lo cuál se podría buscar un método más eficiente.

4.2. Efectividad

Debido a la gran cantidad de capítulos (500 capítulos de 23 minutos cada uno) no se pudieron obtener manualmente los resultados esperados para los experimentos. Por ello, no se podrá determinar qué tan exacto es realmente cada método, pero sí se podrán comparar los métodos entre sí.

Para sortear este problema se decidió comparar manualmente los segmentos de cada predicción y determinar cuáles eran duplicados y cuáles no. Luego, se juntaron las predicciones correctas de todas las configuraciones para crear un conjunto que se utilizó como *ground truth*. Utilizando esto se evaluaron tres métricas en cada configuración:

- **Precisión**: Corresponde al porcentaje de los resultados que correspondían realmente a duplicados. Por ejemplo, si una configuración predecía 3 segmentos duplicados, pero solo 2 de ellos lo eran realmente, entonces la precisión es $2/3 = 0{,}66$. Cuando este valor es cercano a 1, eso significa que los resultados obtenidos son altamente confiables.
- **Recall**: Corresponde al porcentaje de los segmentos duplicados encontrados. Si un video tiene 10 segmentos duplicados en total de los cuáles se reportaron 7, entonces el recall es $7/10 = 0{,}7$. Mientras más se acerque este valor a 1, mayor es la probabilidad

de obtener todos los segmentos duplicados de un video.

- **Intersection over Union** (IoU): También conocido como índice de Jaccard, esta métrica se utiliza para ver la similitud entre los resultados obtenidos y el *ground truth*. Se define de la siguiente forma:

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

  Esta métrica permite ver qué tan exactos fueron los resultados, más allá de si un segmento fue detectado o no.

Se obtuvieron estas 3 métricas para cada una de las configuraciones probadas. Para realizar los experimentos se realizó la búsqueda de duplicados sobre 4 videos, lo que implicó evaluar las predicciones de 52 conjuntos de resultados. Tres de los videos fueron elegidos debido a que en experimentación previa se habían encontrado recapitulaciones en ellos, y además se agregó un cuarto elegido al azar.

Los resultados de cada configuración se muestran en la tabla B.1 del apéndice B, donde también se pueden encontrar resultados específicos. A continuación se muestran los resultados promediados por método.

4.2.1. Selección de Fotogramas Clave

Entre los métodos de selección de fotogramas claves, la reducción de FPS obtuvo resultados un poco mejores que la selección por diferencia de histogramas, pero cabe recordar que el primer método entregaba más fotogramas por segundo que el segundo. Esta diferencia de "densidad" de fotogramas claves influye en el desempeño de los métodos, por lo que el hecho de que la diferencia de histogramas obtenga resultados tan cercanos a la reducción de FPS usando menos fotogramas es una señal de que se escogieron buenos fotogramas clave. En la figura 4.6 se muestra la efectividad de los métodos de selección de fotogramas clave usados.

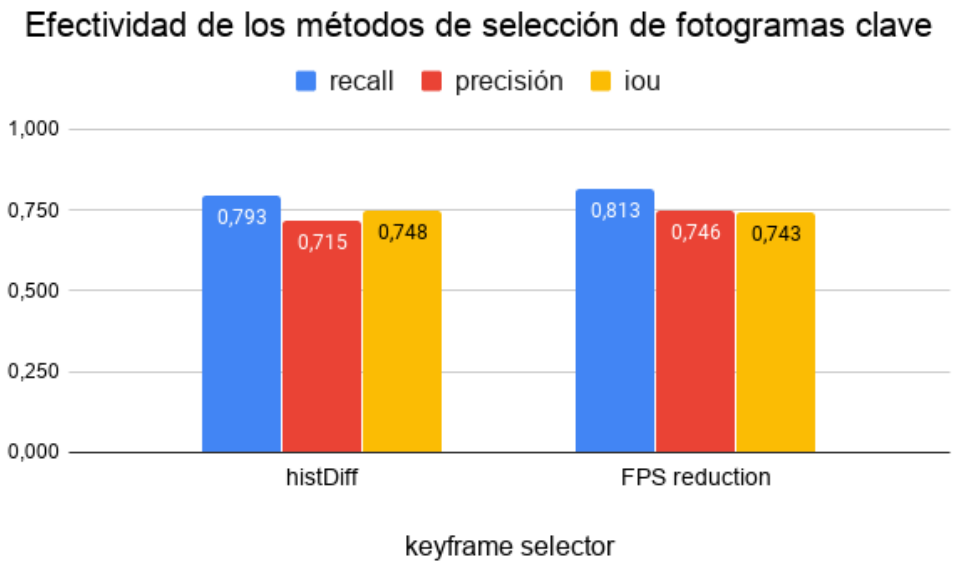


Figura 4.6: Efectividad promedio de los distintos selectores de fotogramas clave probados.

4.2.2. Extracción de Características

En los métodos de extracción de características se puede apreciar un intercambio entre recall y precisión, donde el autoencoder entregó una mejor precisión, mientras que el descriptor color layout entregó mejor recall e IoU. En la figura 4.7 se muestra la efectividad de los métodos de extracción de características usados.

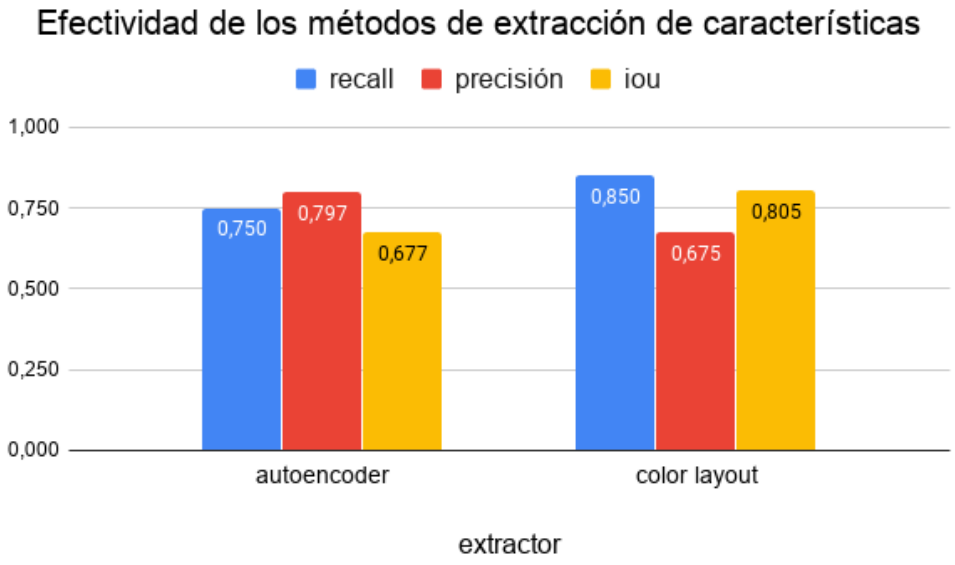


Figura 4.7: Efectividad promedio de los distintos extractores de características.

4.2.3. Búsqueda de Vecinos más Cercanos

Finalmente, al comparar los índices de búsqueda se encuentra una gran diferencia entre la precisión y recall de cada método. El índice random k-d trees obtuvo el mayor recall e IoU, ambos sobre el 96 %, sin embargo su precisión es la más baja con un 56 %. Por otro lado, LSH y SGH obtuvieron una mejor precisión, sobre 80 %, a cambio de un menor recall e IoU. Cabe destacar que aunque LSH y SGH son ambos métodos de búsqueda basados en hashing, SGH presentó un peor desempeño en general que LSH, a pesar de haber sido entrenado para ajustarse a los datos. En la figura 4.8 se muestra la efectividad de los métodos de búsqueda de vecinos cercanos usados.

En específico el resultado de SGH fue inesperado ya que durante la etapa de búsquedas las tablas de hash generadas por SGH estaban muy bien balanceadas, lo que se creía era un buen indicador de la función de hash. Es posible que no se haya entrenado con suficientes datos, o que se requiera una transformación de características con más bases que las recomendadas por los autores.

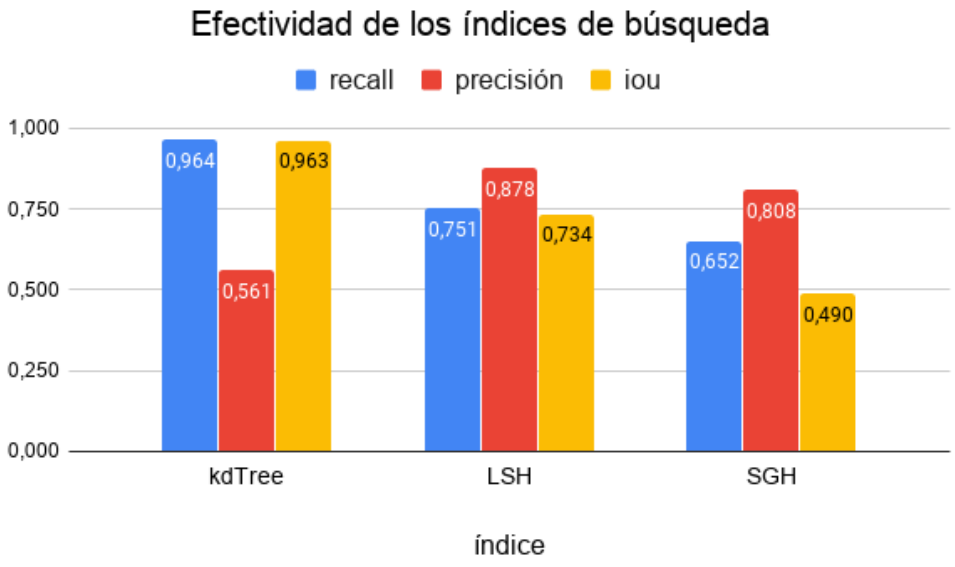


Figura 4.8: Efectividad promedio de los distintos índices de búsqueda utilizados.

4.3. Análisis

4.3.1. Desempeño de Métodos "Inteligentes"

Los resultados de eficiencia muestran que los algoritmos "inteligentes" (esto es, que utilizan los datos para mejorar los resultados, como el autoencoder o SGH) agregan un costo extra que impacta negativamente el tiempo requerido para cada acción. Estos métodos buscan tomar decisiones en base a los datos para obtener mejores resultados, lo que los hace más costosos que otras alternativas pre-definidas como la reducción de fps y color layout.

Sin embargo, a pesar de tener este procesamiento extra, no se logra apreciar una diferencia notoria en la efectividad de los métodos utilizados. Posibles causas para esto es que se eligieron malos parámetros para el método o que simplemente no era el adecuado para el problema. No se pudo determinar cuál era la causa exacta debido a que no se podía realizar una búsqueda exhaustiva sobre los parámetros de cada método con el tiempo disponible. Esta es la desventaja de estos métodos "inteligentes" contra otros métodos estandarizados: se debe invertir tiempo en realizar pruebas para encontrar la configuración adecuada, si es que existe una.

4.3.2. Índices de Búsqueda

Al comparar los índices de búsqueda, se puede ver que random k-d trees resultó ser mucho más eficiente que el resto tanto en búsquedas como en indexado, realizando las búsquedas de vecinos cercanos 3.5 veces más rápido que el índice SGH que le sigue en eficiencia.

Además de eso, este índice obtuvo un recall sorprendentemente alto en comparación al resto. A pesar de presentar una precisión baja (56 %), su buen recall permitiría aplicarlo como primer filtro para detectar todos los segmentos duplicados, acompañado por un segundo filtro

más exigente para comprobar qué predicciones son correctas y cuáles erróneas.

Respecto al mal desempeño de SGH, que se esperaba que fuera por lo menos superior a LSH, se tendrían que hacer más pruebas para determinar cuál fue el problema. Aumentar el número de bases utilizadas en el kernel de transformación de características debería entregar una mejor aproximación del grafo de similitud, lo que podría mejorar los resultados. También se podría utilizar más de una tabla, escogiendo bases aleatorias distintas para cada una para obtener candidatos más variados de la misma forma que en LSH.

4.3.3. Duplicados Visuales

Un problema que se encontró al evaluar las predicciones de todos los métodos fue que se encontraron muchos candidatos que eran visualmente similares, casi copias exactas, pero que correspondían a escenas distintas. La mayoría de estos casos se debía a escenas prolongadas de personajes sobre fondo homogéneo, como el ejemplos de la figura 4.9. Ambas escenas reportadas muestran a los mismos personajes en el mismo fondo, pero corresponden a escenas distintas.

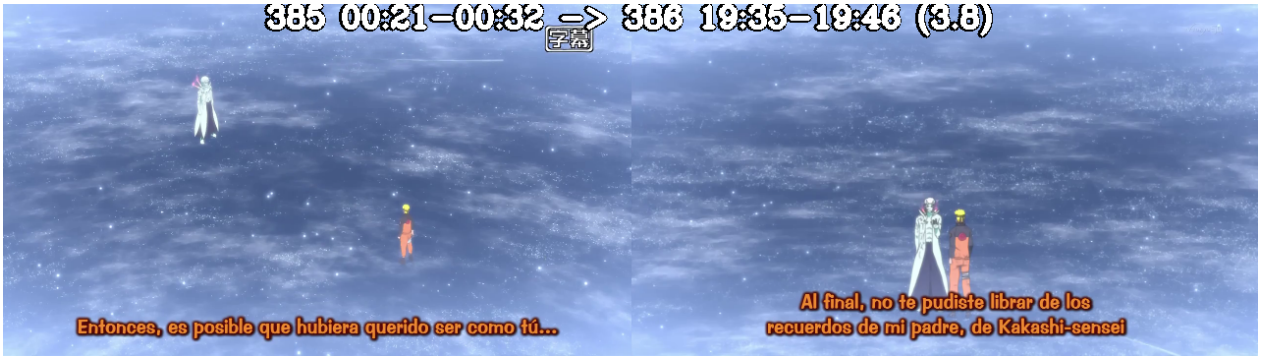


Figura 4.9: Ejemplo de una predicción errónea causada por escenas con el mismo fondo.

Este mismo problema se encontró en otras escenas en las que se reutilizaban recursos. No solo fondos si no que también movimientos e interacciones entre personajes. En las imágenes de la figura 4.10 se muestra un ejemplo de 2 escenas con movimientos de cámara e interacciones idénticas, pero que no son duplicados.

4.3.4. Introducciones y Cierres

Un resultado que se obtuvo como consecuencia de la búsqueda de segmentos duplicados es que se detectó con una gran precisión la introducción y el cierre de los capítulos, duplicados varias veces en los capítulos cercanos al capítulo analizado. En la tabla B.2 se puede apreciar que se detectaron múltiple introducciones y cierres que se entregan como duplicados de la introducción y cierre del capítulo analizado.

De acuerdo a la siguiente fuente [10], las detecciones registradas calzan con el rango real de la introducción y cierre del capítulo, siendo la introducción la canción *Sign* entre los capítulos 129 y 153, y el cierre la canción *For you* entre los capítulos 142 y 153, lo cuál también fue

Figura 4.10: Ejemplo de una predicción errónea causada por escenas que utilizan la misma base y muestran interacciones extremadamente similares. Se puede apreciar claramente la reutilización de una animación para crear una nueva escena.

comprobado manualmente. Cabe notar que ambos se detectaron por más de 88 segundos por capítulos, lo que significa que todas las ocurrencias se detectaron casi en su totalidad.

Capítulo 5

Conclusión

En este trabajo se compararon diversos métodos de RIM usados para abordar el problema de detección de segmentos de videos duplicados, aplicado al caso específico de buscar escenas repetidas en la serie de animé *Naruto Shippuden*. Para ello se dividió la tarea en 4 pasos principales y se tomaron técnicas con enfoques distintos en cada uno.

Según todo lo descrito a lo largo del documento se puede concluir que los objetivos específicos de este trabajo fueron logrados.

1. Se implementaron y evaluaron los métodos de selección de fotogramas clave **reducción de FPS** y **diferencia de histogramas**. Se concluye que a cambio de un mayor tiempo de procesamiento en **diferencia de histogramas**, se lograron obtener resultados similares a **reducción de FPS** utilizando un 33 % menos de fotogramas clave, lo que hace que los siguientes pasos sean más eficientes.
2. Se compararon implementaciones de los métodos de extracción de características **color layout** y **autoencoder**. El descriptor **color layout** obtuvo un mayor recall acompañado de una precisión menor, mientras que el **autoencoder** obtuvo mayor precisión a cambio de un menor recall. Luego, el descriptor a usar depende del objetivo buscado. Considerando el objetivo d
3. En cuanto a búsqueda de vecinos cercanos, se implementaron y evaluaron los métodos **random k-d trees**, y **locality sensitive hashing** usando **stable distributions** y **scalable graph hashing**. El índice **random k-d trees** de la librería FLANN presentó los mejores resultados en eficiencia y en recall, detectando más de un 96 % de los duplicados en promedio. A pesar de que su precisión es baja (56 %), el hecho de detectar la mayoría de los duplicados en un tiempo menor a los otros métodos lo convierte en una buena opción, considerando agregar una segunda etapa de filtrado para mejorar la precisión de los resultados. También cabe destacar que se reimplementó el motor principal de NearPy para optimizar los métodos de LSH utilizados, de otra forma los resultados obtenidos habrían sido peores.
4. Finalmente, se logró implementar un algoritmo de detección de duplicados utilizando las listas de vecinos cercanos obtenidas para evaluar el desempeño de todos los métodos.

Dado todo lo anterior, se concluye que se consiguió cumplir el objetivo general propuesto.

Se implementaron y utilizaron diversos métodos de recuperación de información multimedia para resolver el problema de detección de segmentos de videos duplicados, comparando tanto eficiencia como eficacia en el proceso, y logrando detectar recapitulaciones dentro de la serie analizada.

A pesar de que esta herramienta solo fue evaluada utilizando *Naruto Shippuden*, los métodos probados deberían ser igual de útiles a la hora de detectar segmentos duplicados en cualquier conjunto de videos, debido a que no se realizó ningún ajuste orientado a mejorar los resultados específicamente en los videos utilizados. Además de ello, el resultado de este trabajo no es solo una comparación entre los métodos usados, sino que también una base de código que permite probar nuevas técnicas para cualquiera de las etapas de la detección de forma rápida y sencilla.

Uno de los principales desafíos que se tuvo que abordar a lo largo del trabajo fue manejar la gran cantidad de información generada. Por ejemplo, por la cantidad de métodos probados, se tenían 13 respuestas distintas para cada video (uno por cada configuración). Este desafío trajo grandes problemas sobre todo a la hora de hacer cambios, ya que si se modificaba la estructura en la cual se organizaban los archivos había que asegurarse de que todo estuviera bien actualizado para evitar errores.

5.1. Trabajo Futuro

Como trabajo a futuro se podrían tomar varias rutas posibles. Primero, tomando los resultados obtenidos se propone construir una aplicación que permita a un usuario buscar segmentos repetidos dentro de un conjunto de videos definido, utilizando los métodos que obtuvieron mejor eficiencia y/o efectividad en las pruebas realizadas en este trabajo. Esto ya no tendría que estar limitado solamente a animé, e incluiría desafíos de UI/UX y desarrollo de software en vez de los desafíos técnicos encontrados durante este proyecto.

Por otra parte, se podrían implementar y probar nuevas técnicas de RIM con el objetivo de mejorar los resultados obtenidos. Una posibilidad que no se exploró en este trabajo es el uso de descriptores basados en audio para detectar los duplicados. Si se utilizara un descriptor mixto basado en imágenes y audio se podrían evitar problemas como el expuesto en las figuras 4.9 y 4.10.

Bibliografía

[1] Cisco homepage. https://www.cisco.com/. [Online; ultimo acceso 15-mayo-2020].

[2] github - flann. https://github.com/mariusmuja/flann. [Online; ultimo acceso 15-mayo-2020].

[3] github - flashback detector. https://github.com/cristobal-vildosola/flashback-detector. [Online; ultimo acceso 15-mayo-2020].

[4] github - nearpy. https://github.com/pixelogik/NearPy. [Online; ultimo acceso 15-mayo-2020].

[5] github - scalable graph hashing. https://github.com/jiangqy/SGH-IJCAI2015. [Online; ultimo acceso 15-mayo-2020].

[6] Imdb - top rated. https://www.imdb.com/list/ls021457150/?sort=moviemeter,asc&st_dt=&mode=detail&page=1. [Online; ultimo acceso 15-mayo-2020].

[7] Keras homepage. https://keras.io/. [Online; ultimo acceso 15-mayo-2020].

[8] Myanimelist - most popular. https://myanimelist.net/topanime.php?type=bypopularity. [Online; ultimo acceso 15-mayo-2020].

[9] Myanimelist - naruto: Shippuuden. https://myanimelist.net/anime/1735/Naruto__Shippuuden. [Online; ultimo acceso 15-mayo-2020].

[10] Narutopedia - music. https://naruto.fandom.com/wiki/Music. [Online; ultimo acceso 15-mayo-2020].

[11] Numpy homepage. https://numpy.org/. [Online; ultimo acceso 15-mayo-2020].

[12] Opencv homepage. https://opencv.org/. [Online; ultimo acceso 15-mayo-2020].

[13] Python homepage. https://www.python.org/. [Online; ultimo acceso 15-mayo-2020].

[14] Tensorflow homepage. https://www.tensorflow.org/. [Online; ultimo acceso 15-mayo-2020].

[15] Youtube homepage. https://www.youtube.com/. [Online; ultimo acceso 15-mayo-

2020].

[16] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51:117–122, 01 2008.

[17] Kunlun Bai. 3d convolution. https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-2019. [Online; ultimo acceso 15-mayo-2020].

[18] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *1997 Conference on Computer Vision and Pattern Recognition (CVPR '97), June 17-19, 1997, San Juan, Puerto Rico*, pages 1000–1006, 1997.

[19] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[20] Benjamin Bustos. *Index structures for similarity search in multimedia databases*. PhD thesis, Department of Computer and Information Science, University of Konstanz, June 2006.

[21] Chervinskii. autoencoder structure. https://commons.wikimedia.org/wiki/File:Autoencoder_structure.png, 2015. [Online; ultimo acceso 15-mayo-2020].

[22] Cisco. Cisco visual networking index: Forecast and methodology, 2016–2021. White paper, Cisco Systems Inc., 2017.

[23] Mayur Datar, Piotr Indyk, Nicole Immorlica, and Vahab Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, 01 2004.

[24] Tim Hartley. 3x3 convolution. https://community.arm.com/developer/tools-software/graphics/b/blog/posts/when-parallelism-gets-tricky-accelerating-floyd-steinberg-on-the-mali-gpu, 2014. [Online; ultimo acceso 15-mayo-2020].

[25] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, 604-613:604–613, 1998.

[26] Qing-Yuan Jiang and Wu-Jun Li. Scalable graph hashing with feature transformation. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 2248–2254. AAAI Press, 2015.

[27] Andrej Karpathy. 2x2 max pooling. https://cs231n.github.io/convolutional-networks/#pool, 2015. [Online; ultimo acceso 15-mayo-2020].

[28] Andrej Karpathy. neural network basic architecture. https://cs231n.github.io/neural-networks-1/#layers, 2015. [Online; ultimo acceso 15-mayo-2020].

[29] Eiji Kasutani and Akio Yamada. The mpeg-7 color layout descriptor: a compact image feature description for high-speed image/video segment retrieval. In *Proceedings of the 2001 International Conference on Image Processing, ICIP 2001, Thessaloniki, Greece, October 7-10*, volume 1, pages 674–677, 2001.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[31] Julien Law-To, Valérie Gouet-Brunet, Olivier Buisson, and Nozha Boujemaa. Video copy detection on the internet: The challenges of copyright and multiplicity. In *Proceedings of the 2007 IEEE International Conference on Multimedia and Expo, ICME 2007, July 2-5, 2007, Beijing, China*, pages 2082–2085, 2007.

[32] Sunita Nayak. alexnet architecture. `https://www.learnopencv.com/understanding-alexnet/`, 2018. [Online; ultimo acceso 15-mayo-2020].

[33] Steven Skiena. vecino más cercano. `http://algorist.com/problems/Nearest_Neighbor_Search.html`, 2019. [Online; ultimo acceso 15-mayo-2020].

[34] Arnold W. M. Smeulders, Marcel Worring, Simone Santini, Amarnath Gupta, and Ramesh C. Jain. Content-based image retrieval at the end of the early years. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(12):1349–1380, 2000.

[35] Yair Weiss, Antonio Torralba, and Robert Fergus. Spectral hashing. In *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, pages 1753–1760, 2008.

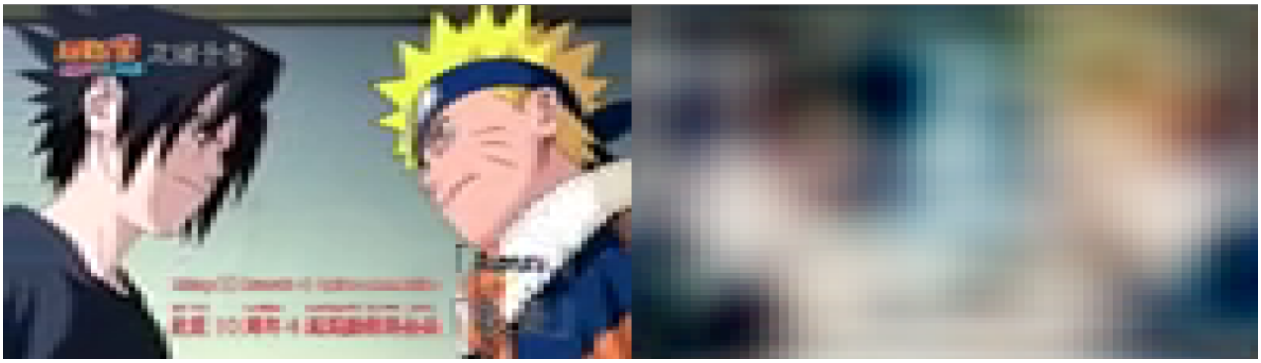
[36] Xiao Wu, Alexander Hauptmann, and Chong-Wah Ngo. Practical elimination of near-duplicates from web video search. In *Proceedings of the 15th International Conference on Multimedia 2007, Augsburg, Germany, September 24-29*, pages 218–227, 2007.

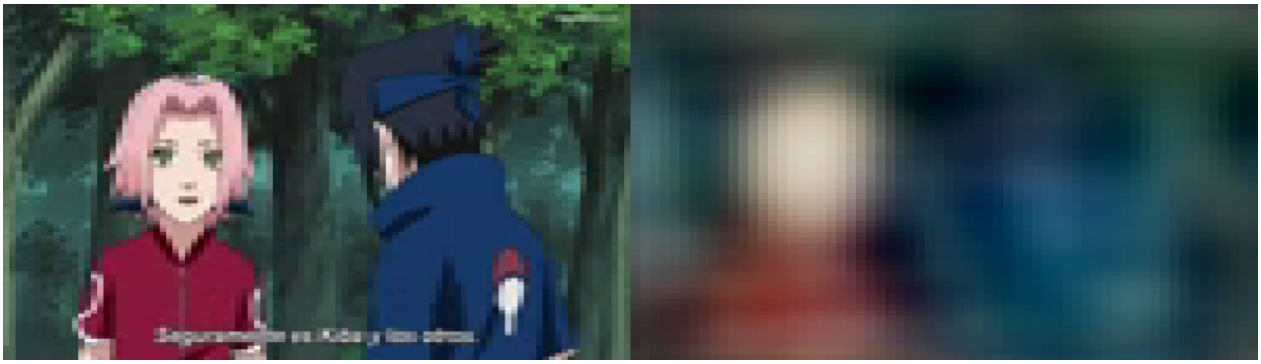
Apéndice A

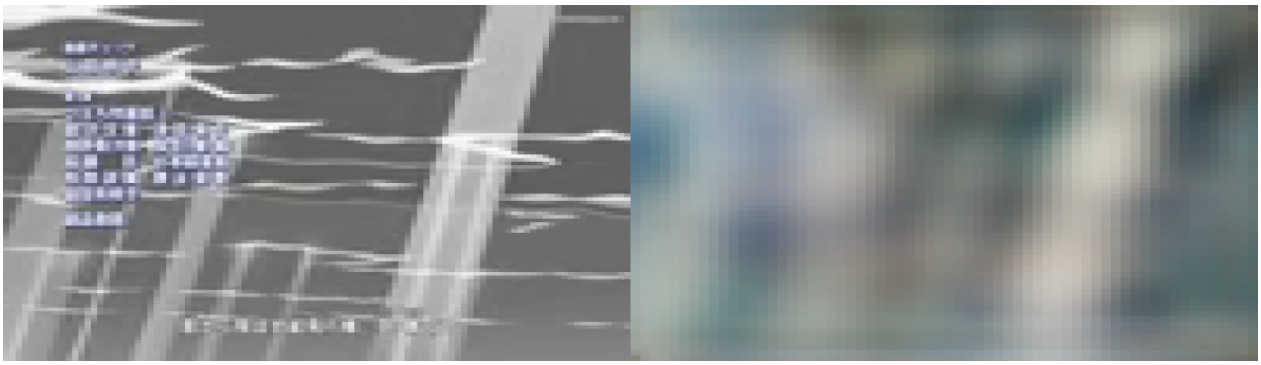
Imágenes reconstruidas por el autoencoder

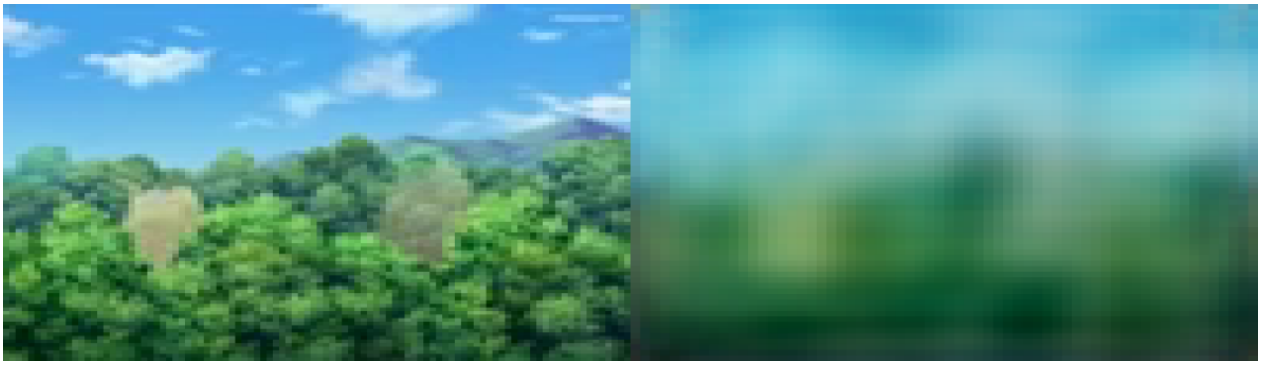
En este apéndice se muestran ejemplos de la reconstrucción conseguida por el autoencoder utilizado para la extracción de características. Cada ejemplo muestra a la derecha la imagen de entrada y a la izquierda la imagen reconstruida. Ambas imágenes son de 64x64 píxeles para comparar adecuadamente la entrada con la salida del autoencoder, y no la imagen original con la reconstrucción (el autoencoder no fue entrenado para reconstruir imágenes con mayor definición que la imagen de entrada).

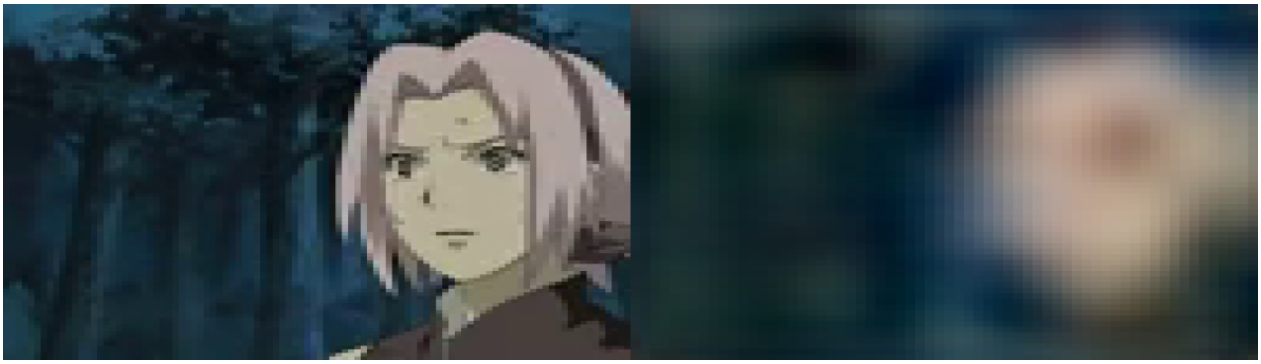
Se puede apreciar una buena reconstrucción en la mayoría de los casos, probablemente el autoencoder aprovecha el hecho de que unos pocos colores y formas se usan repetidamente en la serie, como la alta aparición del verde y celeste por ser utilizado como fondo consistentemente.

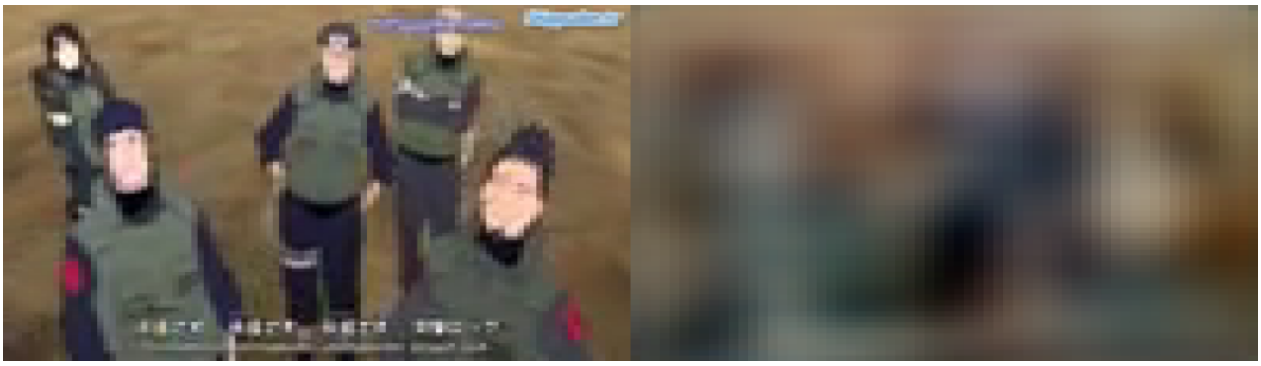


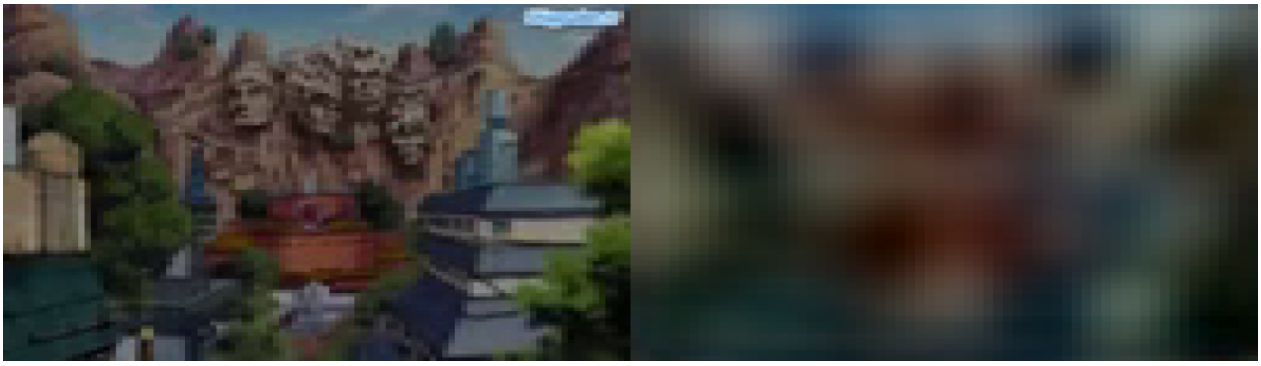


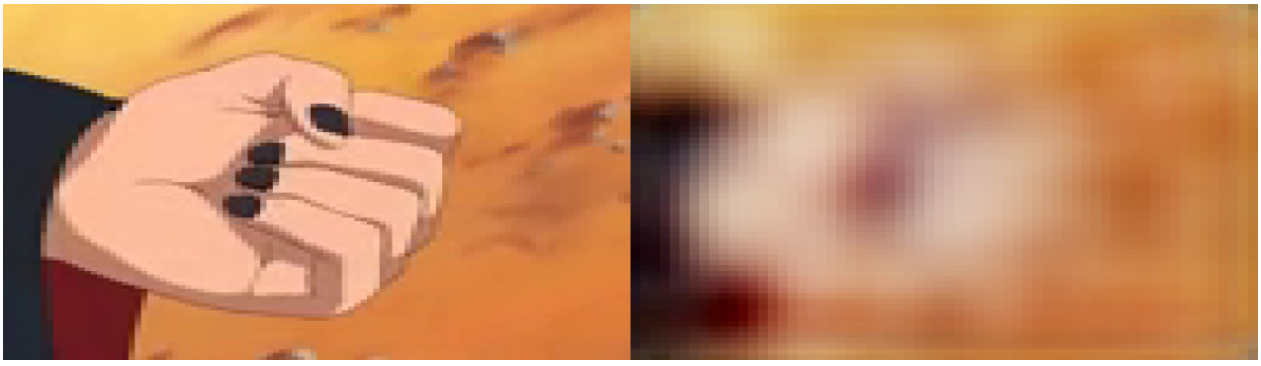












Apéndice B

Resultados Obtenidos

B.1. Resultados de efectividad

En la siguiente tabla se muestran los resultados obtenidos en los experimentos. Para cada configuración se tomó el promedio de recall, IoU y precisión.

configuración	**recall**	**IoU**	**precisión**
histDiff_AE_kdTree_5	0.975	0.970	0.540
histDiff_AE_LSH_10_3	0.603	0.640	0.940
histDiff_AE_SGH_10	0.673	0.473	0.845
histDiff_CL_kdTree_5	0.955	0.965	0.440
histDiff_CL_LSH_10_5	0.885	0.888	0.795
histDiff_CL_SGH_10	0.665	0.555	0.730
reduction_AE_kdTree_5	0.970	0.970	0.680
reduction_AE_LSH_10_3	0.638	0.553	0.955
reduction_AE_SGH_10	0.640	0.455	0.823
reduction_CL_kdTree_5	0.955	0.945	0.583
reduction_CL_LSH_10_5	0.880	0.858	0.823
reduction_CL_SGH_10	0.630	0.477	0.833

Tabla B.1: Resultados promedio por método.

B.2. Resultados Capítulo 143

1229.8	017	534.4	5.3	6.0	160.2	150	2.3	88.4	2.3
428.6	051-052	718.5	5.5	1.2	1265.3	150	1265.8	89.7	3.0
1229.8	088	820.3	6.3	5.0	160.2	151	358.0	88.4	3.2
159.7	129-130	524.6	88.9	7.0	1265.3	151	1265.3	89.7	5.4
159.7	131	447.5	89.5	6.7	160.2	152-153	749.7	88.4	2.4
160.2	132	383.4	88.4	4.5	1265.3	152-153	2710.2	88.6	13.8
160.2	133	1.0	88.4	1.4	1217.5	203	844.5	5.8	1.0
159.4	134	425.0	89.2	7.9	1048.0	204	49.1	6.5	12.0
160.2	135-136	281.7	88.4	3.5	1086.3	204	68.7	10.3	3.2
670.8	135-136	683.5	7.5	1.8	1097.4	204	82.8	5.2	9.0
160.2	137	344.1	88.4	3.1	1244.0	208	682.4	9.9	6.0
160.2	138	208.5	88.4	2.4	1119.4	299	1234.5	5.4	1.0
778.5	138	945.3	9.7	1.0	782.5	329	1022.9	6.1	7.0
774.9	138	641.8	10.3	2.3	365.7	331	721.4	5.7	10.0
780.0	138	697.2	7.6	1.0	384.9	331	733.5	5.6	1.5
159.4	139	234.4	89.2	6.7	419.6	331	747.7	8.3	1.4
160.2	140	415.7	88.9	1.7	494.3	331	763.6	10.5	3.2
159.4	141	352.4	89.2	1.8	790.5	331	772.7	7.6	6.5
159.4	142	121.3	89.2	10.1	842.5	331	788.1	5.8	11.0
1265.3	142	1205.7	89.7	4.2	903.8	331	796.3	8.6	1.1
160.2	144	329.7	88.4	1.8	921.1	331	805.6	10.8	4.5
1265.3	144	1205.2	89.7	12.7	941.9	331	817.1	11.1	1.7
159.7	145	132.6	88.9	13.8	1006.5	349	241.1	7.0	1.0
1265.3	145	1265.3	89.7	24.4	412.4	431	1059.3	6.5	2.2
160.2	146	361.9	88.4	4.2	441.0	431	1066.7	7.5	2.5
1265.3	146	1205.2	89.7	10.9	1229.8	449	422.3	5.3	2.5
160.2	147	384.2	88.4	2.3	1174.3	450	511.3	5.1	1.0
1265.3	147	1205.7	89.7	2.4	1202.4	452	243.6	6.8	7.0
159.7	148	47.6	88.9	5.3	1229.8	467	592.2	6.5	7.0
1265.3	148	1265.3	89.7	11.7	671.4	474	1142.3	7.0	1.8
159.7	149	0.6	88.9	7.8	671.4	489	1286.9	5.9	7.0
1265.3	149	1265.8	89.7	1.9	670.8	492	1301.8	7.5	1.8

Tabla B.2: predicciones obtenidas para el capítulo 143. En verde se muestran los segmentos duplicados detectados. En celeste y morado se muestran los segmentos duplicados que corresponden a la introducción y el cierre del capítulos respectivamente.