



FACULTAD DE MATEMÁTICA, ASTRONOMÍA Y FÍSICA

UNIVERSIDAD NACIONAL DE CÓRDOBA

TRABAJO FINAL:

Técnicas mixtas de seguimiento y aprendizaje para tracking en secuencias de video

Facundo Gaich

Director:

Jorge Adrián Sánchez

25 de septiembre de 2015



Técnicas mixtas de seguimiento y aprendizaje para tracking en secuencias de video por Facundo Gaich se distribuye bajo una Licencia Creative Commons Atribución-CompartirIgual 2.5 Argentina.

Resumen

Dentro del campo de visión por computadora, el problema de *tracking* consiste en seguir la posición de un objeto en una secuencia de video. Este ha sido abordado de variadas maneras, principalmente con métodos basados en flujo óptico y, más recientemente, empleando técnicas de detección y aprendizaje automático. Particularmente, el tracker TLD [10] ha demostrado buenos resultados combinando ambos esquemas. Sin embargo, el algoritmo de clasificación Nearest Neighbour utilizado en TLD tiene requerimientos altos en memoria y poder de cómputo. En este trabajo se exploran otros métodos de clasificación bajo el mismo esquema usado en TLD, de manera de mantener buena desempeño de tracking pero sin dichas limitaciones. Más específicamente, se utiliza un clasificador lineal junto con representaciones basadas en vectores de Fisher, cuyo uso de recursos es considerablemente menor a NN. Se compara experimentalmente el desempeño de tracking de este nuevo esquema con TLD original, validando el uso de este nuevo esquema para abordar el problema de tracking.

Abstract

In the area of computer vision, the task of *tracking* consists in following the position of an object throughout a video sequence. This challenge has been approached in different ways, such as optical flow methods and, more recently, using object detectors and machine learning. In particular, TLD [10] has achieved good results using a combination of both. However, the Nearest Neighbour classification algorithm used in TLD has high requirements in terms of memory and processing. In this work, we explore the use of other classification methods under the framework of TLD, with the objective of keeping similar tracking results while using less resources. More specifically, we use a linear classifier together with a Fisher vector representation, as such method uses considerably less resources than NN. We run experiments to compare the tracking precision of this new scheme against the original TLD, validating its use as an approach to solve the task of tracking.

Índice general

1. Introducción	2
2. Marco Teórico	5
2.1. Tracking Clásico	5
2.1.1. Flujo óptico	5
2.1.2. Algoritmo de Lucas-Kanade	7
2.1.3. Median Flow	13
2.2. Tracking por Detección	18
2.2.1. Detección de ventana deslizante	19
2.2.2. Clasificadores utilizados en detección	23
2.3. Modelos Mixtos	31
2.3.1. TLD	32
3. TLD-FV	39
3.1. Introducción	39
3.2. RANSAC Flock of Trackers	39
3.3. Clasificación lineal	40
3.3.1. Función de pérdida	41
3.3.2. Descenso de gradiente	43
3.3.3. Vectores de Fisher	45
3.3.4. Clasificación lineal con vectores de Fisher	46
3.4. TLD-FV	48
4. Experimentos	50
4.1. Secuencias de prueba	50
4.2. Parámetros	51
4.3. Resultados	53
5. Conclusiones y trabajos a futuro	58

Capítulo 1

Introducción

A lo largo del desarrollo de la computación y tecnologías relacionadas, ha aparecido como motivación el lograr que una computadora realice funciones de un humano. Con los años, esta brecha se ha ido achicando lentamente en diversos aspectos, uno de ellos la visión. El área de *visión por computadora* es la encargada de lograr que una máquina procese un video o secuencia de imágenes tomado de una escena natural y lo analice, extrayendo información relevante y dándole sentido.

Esta capacidad se puede utilizar para diversas aplicaciones, como por ejemplo el uso de visión en robots de líneas de ensamblaje, navegación autónoma o asistada en vehículos, control y monitoreo en áreas públicas a través de identificación de rostros, procesamiento de imágenes médicas o interacción entre humanos y máquinas. Para llevar a cabo estas tareas de manera satisfactoria, se deben desarrollar técnicas que le permitan a una computadora reconocer formas, objetos y caras, agrupar y seleccionar estos objetos según su importancia, adaptarse a fuentes de iluminación, interpretar la estructura tridimensional a través de la profundidad y realizar un seguimiento en espacio y tiempo de cómo se mueven estos objetos, entre otras cosas.

Más recientemente, la masificación del acceso a internet y el advenimiento de las redes sociales ha desembocado en la creación de inmensos repositorios de imágenes y videos naturales, como pueden ser Facebook¹ y YouTube² respectivamente. Esto ha dado lugar a nuevos requerimientos de escala y robustez para los algoritmos de visión por computadora como así también a nuevas aplicaciones basadas en el procesamiento y análisis masivo de datos. Como respuesta a estos nuevos factores se han ido incorporando cada vez más el uso de nuevas técnicas relacionadas a aprendizaje automático y reconocimiento de patrones, para poder analizar de manera automática estos grandes conjuntos de datos.

Este trabajo se centra alrededor del problema de seguimiento en video, conocido en inglés como *tracking*. De forma general, consiste en identificar el mismo objeto u objetos en cuadros de video consecutivos, de manera de saber la ubicación espacial de este o estos a lo largo de todo el video. Por ejemplo, seguir varios autos en una autopista, el movimiento de una persona en la calle o incluso el movimiento de las extremidades de una persona bailando. A su vez, es una herramienta que sirve de base en aplicaciones de más alto nivel como

¹<http://www.facebook.com>

²<http://www.youtube.com>

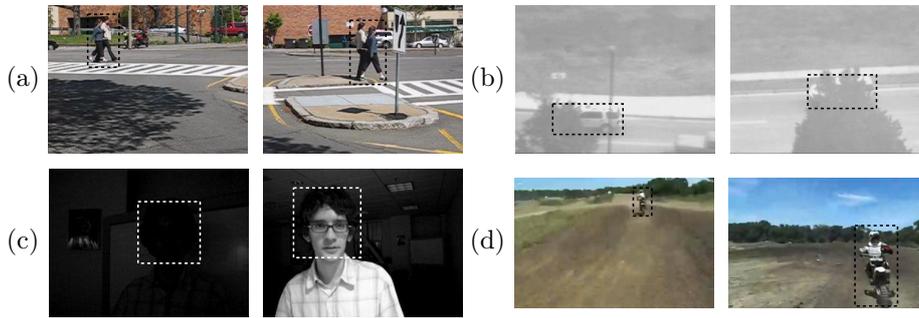


Figura 1.1: Problemas típicos a tener en cuenta a la hora de trackear un objeto: (a) Cambios de pose, (b) oclusiones parciales y totales, (c) cambios de iluminación y (d) cambios de escala.

interpretación de trayectorias, monitoreo de zonas, reconocimiento de objetivos y análisis del comportamiento.

Un algoritmo de tracking puede tomar como entrada un video y una región dentro del primer cuadro indicando cuál es el objeto a seguir. Luego, para cada cuadro siguiente debe indicar cuál es la región donde se encuentra el objeto en ese momento, o bien indicar que ha salido fuera de cámara. En el caso más simple, el objeto y su entorno sólo se trasladan de manera alineada al plano de la cámara y no cambian de apariencia ni se mueven a alta velocidad, por lo que un algoritmo que busca una región parecida (para cierta noción de semejanza) a la indicada en el cuadro anterior funciona de forma satisfactoria. Sin embargo, en un caso real el objeto puede cambiar de posición, escala o color, desaparecer de la escena y volver a entrar o quedar total o parcialmente ocluido, hasta el punto de cambiar completamente su apariencia con respecto al primer cuadro. También pueden intervenir otros factores como cambios de iluminación, movimientos de cámara más allá de traslado en el plano, cuantificación de los sensores digitales y artefactos de compresión. Algunos ejemplos se pueden ver en la Figura 1.1.

El problema de tracking se ha venido abordando en la literatura desde por lo menos 1981, año en que Lucas y Kanade publicaron el método de tracking que lleva sus nombres [13]. Hasta el día de hoy este sigue siendo importante y de hecho forma la base de uno de los algoritmos desarrollados en este trabajo. Desde entonces se han desarrollado infinidad de técnicas para hacer algoritmos más robustos antes los problemas más típicos, como pueden ser el uso de descriptores de la imagen más complejos.

Algunos de los algoritmos clásicos de tracking se basan en la noción de *flujo óptico*, es decir, en el desplazamiento de los píxeles que representan un objeto de un cuadro a otro. Dada la región de interés en un cuadro, se pueden tomar todos o algunos de los píxeles que esta contiene y luego buscar en el cuadro siguiente la nueva posición de estos. Con esta información, luego se estima el movimiento de todo el objeto. Muchos de estos algoritmos utilizan técnicas de optimización para encontrar la hipótesis de movimiento que minimice alguna función de puntaje. Este tipo de algoritmos suelen asumir un límite en el cambio de posición y apariencia del objeto de un cuadro al siguiente y son incapaces de manejar oclusiones totales.

Buscando mitigar las limitaciones y problemas de los algoritmos más clásicos de tracking, se han propuesto algoritmos utilizando esquemas de *detección* de objetos. Estos se basan en detectar el objeto en cada cuadro individualmente, ya sea simplemente basado en su apariencia inicial o alguna representación más compleja de esta. Esto lo hace robusto ante movimientos muy rápidos del objeto de un cuadro a otro e incluso ante la eventual salida del objeto de cámara. Generalmente, incorporan técnicas de aprendizaje automático de variadas maneras. Sin embargo, dichos esquemas de aprendizaje suelen requerir mucho poder de cómputo y memoria en comparación a esquemas de flujo óptico. Además, suelen tener varios parámetros sensibles que deben ser estimados u optimizados de alguna manera, generalmente empíricamente o por búsqueda exhaustiva.

Más recientemente, se ha buscado utilizar esquemas mixtos de tracking clásico y basado en detección, de manera de lograr sistemas de tracking que funcionan satisfactoriamente bajo variadas condiciones. Una forma de lograr esto es utilizar las técnicas más rápidas, es decir tracking por flujo óptico, cuando las condiciones son más favorables y utilizar las técnicas más robustas como detección cuando se vuelve más difícil seguir al objeto. Uno de los desafíos a la hora de crear estos sistemas es cómo combinar la salida de los dos tipos de algoritmos en una sola o, aún más, cómo hacer que los dos esquemas puedan interactuar directamente para mejorar su desempeño mutuamente.

En este trabajo se revisan algunos esquemas de tracking de distinto tipo y se propone un nuevo tracker mixto, llamado TLD-FV, el cual combina diversos conceptos para balancear las ventajas y desventajas de estos individualmente. En el Capítulo 2 se desarrollan los algoritmos de tracking relevantes. En la Sección 2.1 se presentan los algoritmos basados en flujo óptico. En la Sección 2.2, aquellos basados en detección. En la Sección 2.3 se revisa el sistema de tracking TLD [12], el cual forma la base de TLD-FV. En el Capítulo 3 se delinea la conformación del sistema de tracking TLD-FV. En el Capítulo 4 se muestran los resultados experimentales de este. Por último, en el Capítulo 5 se encuentran las conclusiones y posibles trabajos a futuro.

Capítulo 2

Marco Teórico

2.1. Tracking Clásico

2.1.1. Flujo óptico

Consideremos de nuevo un video natural tomado por una cámara captando una escena. El movimiento relativo entre nuestro observador, la cámara, y los objetos de la escena crean un patrón de movimiento de los píxeles del video. Este patrón de movimiento es conocido como el *flujo óptico* del video. Analizándolo podemos extraer cierta información del movimiento de la escena.

Más específicamente, el flujo óptico codifica el *movimiento aparente* de todos los objetos en la escena. Se le denomina “aparente” ya que puede diferir del movimiento real que realizaban los objetos a la hora de captar la escena. Un ejemplo clásico de este fenómeno es el del *barber pole*, que se puede ver en la Figura 2.1. En general, esta diferencia se debe al hecho de que la cámara esté captando la escena desde un punto de vista en particular, conjugado con otros factores como rotaciones y cambios de iluminación. Más allá de esta discrepancia, sigue resultando útil tener la información de movimiento aparente de un objeto.

Consideremos un video en el cual solo vemos el canal de luminancia, es decir el brillo de los píxeles y no su color. Denotemos con $I(x, y, t)$ el brillo del píxel en la posición (x, y) en el tiempo t .

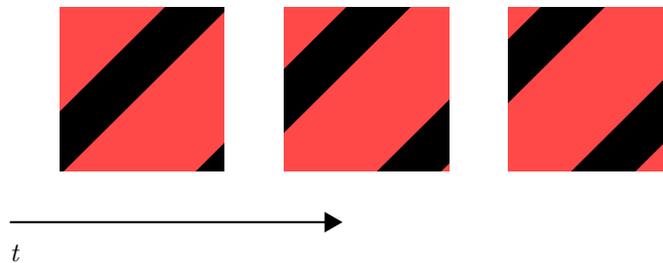


Figura 2.1: Efecto *barber pole*: ¿En qué sentido y dirección se mueven las líneas negras? Su movimiento aparente es hacia la izquierda y arriba, pero en realidad podrían estar moviéndose de infinitas otras maneras, por ejemplo hacia arriba solamente.

Supongamos que un pixel en la posición (x, y) se movió Δx y Δy en un espacio de tiempo Δt . Asumiendo un movimiento pequeño podemos realizar la siguiente aproximación por una serie de Taylor de primer orden:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) \approx I(x, y, t) + \frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t. \quad (2.1)$$

Supongamos además que la iluminación en la escena es uniforme y que el objeto en la escena es plano, de manera tal que al moverse su brillo aparente no cambia. Esta restricción es conocida como la *condición de brillo constante*. Se traduce en:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t). \quad (2.2)$$

Combinando las ecuaciones (2.1) y (2.2) llegamos a

$$\frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t = 0 \quad (2.3)$$

y por último dividimos por Δt , obteniendo

$$\frac{\partial I}{\partial x} \frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y} \frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t} = 0. \quad (2.4)$$

Las derivadas parciales $\frac{\partial I}{\partial x}$, $\frac{\partial I}{\partial y}$ y $\frac{\partial I}{\partial t}$ son los respectivos componentes de la *imagen gradiente* en el punto (x, y) en tiempo t y las podemos escribir como I_x , I_y e I_t . Notar que las dos primeras tienen información de los alrededores del punto (x, y) en el tiempo t mientras que la última tiene información de cómo cambia el brillo de los píxeles que pasan por la posición fija (x, y) a lo largo del tiempo. Los factores $\frac{\Delta x}{\Delta t}$ y $\frac{\Delta y}{\Delta t}$ son los componentes de la *velocidad* $\mathbf{v} = (u, v)^T$ del pixel, los cuales se convierten en nuestras incógnitas:

$$I_x u + I_y v = -I_t. \quad (2.5)$$

Llegamos así a una ecuación lineal con dos incógnitas, por lo que el sistema está *subespecificado*. Esto es conocido como el *problema de apertura* y es lo que obliga a poner más condiciones sobre cómo se modela el flujo óptico para poder llegar a una solución.

Para entender mejor el problema de apertura, reescribamos nuestra ecuación usando los vectores gradiente y velocidad:

$$\nabla I \cdot \mathbf{v} = \nabla I^T \mathbf{v} = -I_t. \quad (2.6)$$

Esta ecuación nos dice que el componente de la velocidad en la dirección de la gradiente es directamente proporcional al cambio de brillo en el tiempo.

Visto de otra manera, estamos aproximando la apariencia local de la imagen de manera lineal conforme al sentido, dirección y magnitud de la gradiente y al movimiento local como una traslación de esta aproximación de forma tal que el brillo en el punto cambia proporcionalmente a $-I_t$. Una representación visual de esto se puede ver en la Figura 2.2.

Sin embargo, para un cambio de brillo dado, existen infinitas traslaciones que lo causan, correspondientes a la línea dentro la aproximación a la imagen que une todos los puntos con el mismo valor de brillo deseado. Por lo tanto,

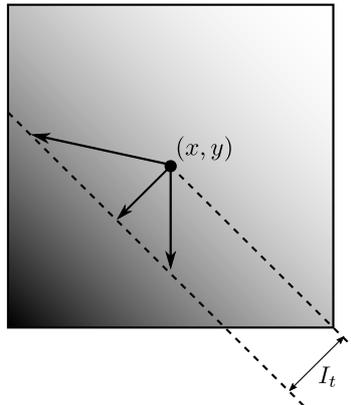


Figura 2.2: *Problema de apertura*: En esta aproximación, el cambio de brillo es lineal según dirección y magnitud de $\nabla I(x, y)$. Debido a esto, existen infinitos vectores \mathbf{v} que son solución de (2.6).

el grado de libertad que tenemos para resolver la ecuación puede interpretarse como la componente de la velocidad en la dirección de tal línea.

Este problema es análogo al ejemplo del *barber pole* de la Figura 2.1. El hueco o “apertura” a través del cual vemos el barber pole no nos deja ver todo el contexto necesario para dilucidar de manera precisa el movimiento del objeto. De la misma manera, la falta información en las ecuaciones de flujo óptico no permite desambiguar la forma exacta del vector \mathbf{v} : podría estar apuntando en la dirección de la gradiente, directamente hacia abajo, directamente hacia la izquierda o de muchas otras maneras.

El problema de apertura es evidencia de que no podemos trackear un solo punto en la imagen. Una forma de lidiar con esto es introduciendo más información a la especificación del problema.

2.1.2. Algoritmo de Lucas-Kanade

El algoritmo de Lucas-Kanade extiende las condiciones de flujo óptico discutidas en la sección anterior a todos los píxeles dentro de una ventana rectangular centrada en un punto. Este recorte de la imagen original es también conocido como un *parche* de la imagen. Se obtiene así un sistema de ecuaciones de flujo óptico como el siguiente, donde q_1, q_2, \dots, q_n son los puntos en el parche y donde extendemos I_x, I_y e I_t a todos ellos:

$$\begin{aligned}
 I_x(q_1)u + I_y(q_1)v &= -I_t(q_1) \\
 I_x(q_2)u + I_y(q_2)v &= -I_t(q_2) \\
 &\vdots \\
 I_x(q_n)u + I_y(q_n)v &= -I_t(q_n).
 \end{aligned}
 \tag{2.7}$$

La idea detrás de esta extensión es que los píxeles del parche, por estar en la cercanía del punto central, tienen un vector velocidad aproximadamente similar.

Es deseable que la ventana sea suficientemente grande como para que las gradientes de la imagen dentro del parche tengan información suficiente para poder resolver estas ecuaciones. Por el contrario, mientras más grande sea el parche contendrá píxeles más alejados del centro, por lo cual la aproximación lineal de la velocidad será peor e introducirá más error en los cálculos, disminuyendo la precisión de nuestra solución.

Generalmente, el sistema en (2.7) está sobreespecificado, ya que la ventana rectangular suele contener más de dos píxeles. El algoritmo LK utiliza mínimos cuadrados para llegar a una solución que ajuste lo mejor posible a nuestras condiciones.

Consideremos discretizar el tiempo de manera que $\Delta t = 1$ y este sea el tiempo entre dos cuadros consecutivos. Llamemos a estos cuadros I y J , donde el segundo viene después que el primero y sean $I(x, y)$ y $J(x, y)$ los valores de brillo en dichos puntos. Supongamos que nuestro punto a trackear tiene coordenadas (p_x, p_y) y sean w_x y w_y dos enteros que determinan el tamaño de la ventana. Queremos luego encontrar el vector velocidad que minimice

$$\epsilon(\mathbf{v}) = \epsilon(u, v) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (I(x, y) - J(x + u, y + v))^2. \quad (2.8)$$

Notar que el tamaño de la ventana es $(2w_x + 1) \times (2w_y + 1)$. También suele ser llamada *ventana de integración*. Valores típicos de w_x y w_y son de 2 a 7.

En términos más generales, la ecuación (2.8) plantea encontrar la traslación que haga más “similares” a I y J dentro de la ventana de integración, una técnica conocida como *alineado de imágenes*. En este sentido, para que el algoritmo sea más robusto ante grandes movimientos de píxeles, es preferible que la ventana sea más grande de manera que el alineado abarque una mayor extensión de la imagen. Más específicamente, es deseable que $u \leq w_x$ y $v \leq w_y$ para los valores óptimos de u y v .

Como podemos ver, hay un balance entre precisión y robustez a la hora de elegir el tamaño de la ventana. Al final de esta sección presentamos un método que permite abordar este problema.

Veamos cómo minimizar la función en (2.8). Derivando en \mathbf{v} obtenemos

$$\frac{\partial \epsilon(\mathbf{v})}{\partial \mathbf{v}} = -2 \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (I(x, y) - J(x + u, y + v)) \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix}. \quad (2.9)$$

Substituyamos $J(x + u, y + v)$ por su expansión de Taylor de primer orden alrededor de $\mathbf{v} = (0, 0)^T$:

$$\frac{\partial \epsilon(\mathbf{v})}{\partial \mathbf{v}} \approx -2 \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \left(I(x, y) - J(x, y) - \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix} \mathbf{v} \right) \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix}. \quad (2.10)$$

Notar que la diferencia $J(x, y) - I(x, y)$ se puede interpretar como la derivada temporal $I_t(x, y)$. También notar que $\begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix}$ es simplemente ∇J^T . Más aún, podemos aproximar ∇J^T por ∇I^T asumiendo que el movimiento es pequeño.

Luego, podemos reescribir (2.10) como

$$\frac{1}{2} \frac{\partial \epsilon(\mathbf{v})}{\partial \mathbf{v}} \approx \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (\nabla I^T \mathbf{v} + I_t) \nabla I^T, \quad (2.11)$$

donde omitimos los parámetros para simplificar la notación.

En (2.11) vuelven a surgir las ecuaciones planteadas en (2.7) dentro de una suma de productos, lo que nos da la pauta de que efectivamente plantear mínimos cuadrados como lo hicimos nos aproxima a una solución de ese sistema.

Por último, reescribamos (2.11) como

$$\frac{1}{2} \left[\frac{\partial \epsilon(\mathbf{v})}{\partial \mathbf{v}} \right]^T \approx \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \left(\begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \mathbf{v} - \begin{bmatrix} -I_t I_x \\ -I_t I_y \end{bmatrix} \right). \quad (2.12)$$

Denotemos

$$\mathbf{G} \doteq \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \quad (2.13)$$

$$\mathbf{b} \doteq \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} -I_t I_x \\ -I_t I_y \end{bmatrix}. \quad (2.14)$$

Luego, (2.12) se puede escribir como

$$\frac{1}{2} \frac{\partial \epsilon(\mathbf{v})}{\partial \mathbf{v}} \approx \mathbf{G} \mathbf{v} - \mathbf{b}, \quad (2.15)$$

y por lo tanto la solución es

$$\mathbf{v} = \mathbf{G}^{-1} \mathbf{b}. \quad (2.16)$$

La ecuación (2.16) solo es válida si la matriz \mathbf{G} tiene inversa. Esta matriz es conocida también como el *tensor de estructura* de la imagen alrededor del punto (p_x, p_y) .

Se puede mostrar que los autovalores del tensor de estructura dan una medida de cómo está distribuido ∇I dentro de la ventana [2]. En particular, serán ambos positivos, y por lo tanto la matriz invertible, si la ventana contiene suficiente información de gradiente en ambas direcciones x e y . Una manera de asegurar esta condición es aumentar el tamaño de la ventana, incurriendo en las desventajas ya mencionadas.

Todo el desarrollo anterior se puede extender a modelos de movimiento más complejos que simplemente una traslación (u, v) . Podemos, por ejemplo, agregar un par de parámetros s y θ que modelen el cambio de escala y la rotación respectivamente. Para esto, debemos reformular (2.8) como

$$\epsilon(s, \theta, u, v) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (I(x, y) - J(s\mathbf{R}\mathbf{x} + \mathbf{v}))^2 \quad (2.17)$$

donde la matriz de rotación \mathbf{R} tiene la forma

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (2.18)$$

y minimizar esta nueva función. Otro ejemplo aún más general es el de una *transformación afín*, la cual engloba cambios de escala de cada eje independientemente, rotaciones y sesgado de la imagen.

En la práctica, para obtener una mejor precisión en los resultados, la implementación de este algoritmo es iterativa. En la iteración k , utilizamos la mejor estimación hasta ese momento, llamémosla \mathbf{v}_{k-1} , para trasladar a la imagen J , de manera que definimos

$$J_k(\mathbf{x}) \doteq J(\mathbf{x} + \mathbf{v}_{k-1}). \quad (2.19)$$

Luego, utilizamos (2.16) para calcular la velocidad residual entre I y J_k ,

$$\mathbf{h}_k = \mathbf{G}^{-1} \mathbf{b}_k. \quad (2.20)$$

Notar que \mathbf{G} no depende de k ya que J no aparece en (2.13), por lo que podemos pre-calcularla. Esta es una ventaja de aproximar ∇J^T por ∇I^T , como hicimos en (2.11). La nueva estimación es

$$\mathbf{v}_k = \mathbf{v}_{k-1} + \mathbf{h}_k. \quad (2.21)$$

El algoritmo iterativo se inicializa con $\mathbf{v}_0 = (0, 0)^T$ e itera hasta que la velocidad residual sea lo suficientemente pequeña o se llegue a un máximo de iteraciones.

Se puede ver el pseudocódigo del algoritmo LK iterativo en el Algoritmo 1. Notar que aproximamos I_x e I_y utilizando *diferencias centradas*.

Algoritmo 1 Implementación iterativa de Lucas-Kanade

Entrada: I, J imágenes, (p_x, p_y) coordenadas del punto a trackear en I

$$I_x(x, y) \leftarrow \frac{I(x+1, y) - I(x-1, y)}{2}$$

$$I_y(x, y) \leftarrow \frac{I(x, y+1) - I(x, y-1)}{2}$$

$$\mathbf{G} \leftarrow \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2(x, y) & I_x(x, y)I_y(x, y) \\ I_x(x, y)I_y(x, y) & I_y^2(x, y) \end{bmatrix}$$

$$\mathbf{v}_0 \leftarrow (0, 0)^T$$

$$k \leftarrow 1$$

repeat

$$J_k(x, y) \leftarrow J(\mathbf{x} + \mathbf{v}_{k-1})$$

$$I_t(x, y) \leftarrow J_k(x, y) - I(x, y)$$

$$\mathbf{b}_k \leftarrow \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} -I_t(x, y)I_x(x, y) \\ -I_t(x, y)I_y(x, y) \end{bmatrix}$$

$$\mathbf{h}_k \leftarrow \mathbf{G}^{-1} \mathbf{b}_k$$

$$\mathbf{v}_k \leftarrow \mathbf{v}_{k-1} + \mathbf{h}_k$$

until $k \geq k_{max}$ o $\|\mathbf{h}_k\| \leq d_{min}$

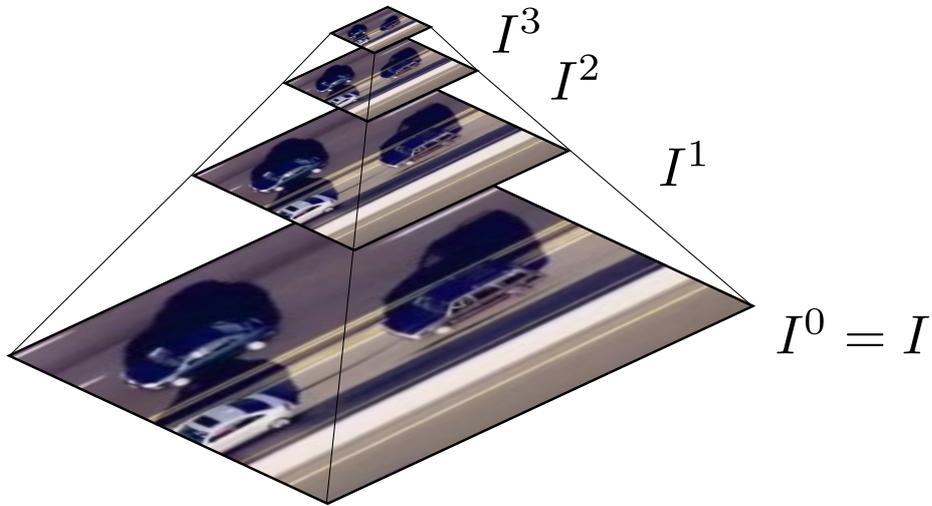


Figura 2.3: Pirámide de resolución: cada nivel tiene la mitad de resolución que el nivel inferior.

Lucas-Kanade Piramidal

Ya vimos que al elegir el tamaño de la ventana debemos optar entre mayor precisión o mayor robustez de trackeo. ¿Existe una manera de obtener lo mejor de los dos mundos? Una técnica que lo hace posible es el uso de pirámides de resolución. Este tipo de representación permite aumentar el rango de movimiento que el algoritmo detecta sin tener que aumentar el tamaño de la ventana.

La pirámide de resolución de una imagen I está formada por n imágenes I^0, I^1, \dots, I^n , donde $I^0 = I$ y la imagen I^k es una versión de menor resolución de la imagen I^{k-1} para $k \geq 1$ (ver Figura 2.3).

Generalmente, de un nivel a otro la resolución lineal disminuye en factor 2. Si la imagen en I^0 tiene tamaño $n_x^0 \times n_y^0$, entonces la imagen I^k tendrá tamaño

$$n_x^k = \left\lfloor \frac{n_x^0}{2^k} \right\rfloor, \quad n_y^k = \left\lfloor \frac{n_y^0}{2^k} \right\rfloor. \quad (2.22)$$

Debido al carácter exponencial de esta construcción, en la práctica n no superará 4 o 5. La generación de los niveles de la pirámide se hace de manera recursiva, con I^0 como caso base.

No es trivial realizar el subsamplio de un nivel a otro de la pirámide, ya que es importante que mantenga la suficiente información del nivel inferior para que el algoritmo funcione correctamente. Para lograr esto, primero se aplica un *filtro de suavizado* por cada imagen I^k , el cual consiste, conceptualmente, en reemplazar cada pixel por la suma ponderada de él mismo y sus vecinos.

Una vez filtrada I^k , podemos generar I^{k+1} con la fórmula

$$I^{k+1}(x, y) = I^k(2x, 2y), \quad (2.23)$$

la cual se conoce como el *diezmado* de I^k . Notar que si bien estamos tomando un cuarto de los pixeles de I^k , gracias al filtro estos contienen información de sus alrededores.

En la práctica, el paso de filtrado se realiza sólo sobre los pixeles que serán seleccionados por el paso de diezmado de la imagen.

¿Cómo se utiliza esta representación para mejorar los algoritmos de Lucas-Kanade? Primero, se generan las pirámides de resolución de las imágenes I y J . Con estas, se corre el algoritmo descrito en la sección anterior entre los pares de imágenes I^k y J^k de cada nivel, de manera secuencial desde el nivel superior y propagando resultados intermedios hacia abajo. Al final se obtiene una estimación de \mathbf{v} para el nivel inferior, el cual contiene a las imágenes originales I y J .

En más detalle, dado un punto (p_x, p_y) y n el nivel máximo de las pirámides, primero trasladamos este punto al nivel n con la fórmula

$$p_x^n = \frac{p_x}{2^n}, \quad p_y^n = \frac{p_y}{2^n}. \quad (2.24)$$

Luego, utilizamos LK para encontrar la mejor estimación del desplazamiento (p_x^n, p_y^n) entre I^n y J^n , llamémosla $\mathbf{v}^n = (u^n, v^n)^T$. Propagamos la estimación \mathbf{v}^n al nivel de abajo trasladando J^{n-1} por $2\mathbf{v}$ (el factor 2 corresponde a la relación de resolución entre un nivel y otro) y corremos LK entre I^{n-1} y esta nueva imagen trasladada.

En general, para cada nivel k la acumulación de propagar todas las estimaciones anteriores será

$$\mathbf{d}^k = (d_x^k, d_y^k) = \sum_{l=k+1}^n 2^l \mathbf{v}^l. \quad (2.25)$$

Es decir, en vez de minimizar (2.8) para I^k y J^k , minimizamos la siguiente ecuación

$$\epsilon(\mathbf{v}^k) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (I^k(x, y) - J^k(x + d_x^k + u^k, y + d_y^k + v^k))^2. \quad (2.26)$$

Esta modificación no cambia substancialmente ni el análisis ni la implementación de la minimización con respecto a la de la versión no piramidal, efectivamente es solo trasladar J^k en las cuentas. Al finalizar, la estimación global es

$$\mathbf{v} = \mathbf{d}^0 + \mathbf{v}^0. \quad (2.27)$$

El Algoritmo 2 muestra la implementación de LK piramidal, donde la función LK_{guess} es el Algoritmo 1 modificado para minimizar la ecuación (2.26).

Notar que al minimizar (2.26) el tamaño de la ventana es el mismo en cada nivel, ya que w_x y w_y no dependen de k . Esto implica que cada aplicación de LK tiene la misma precisión que una corrida en la versión no piramidal. Sin embargo, al ir acumulando los desplazamientos a lo largo de las $n + 1$ corridas el rango de movimiento efectivo aumenta. Si suponemos que una sola corrida puede detectar hasta \mathbf{v}_{max} , el rango total según (2.25) y (2.27) será

$$\sum_{l=0}^n 2^l \mathbf{v}_{max} = (2^{n+1} - 1) \mathbf{v}_{max}. \quad (2.28)$$

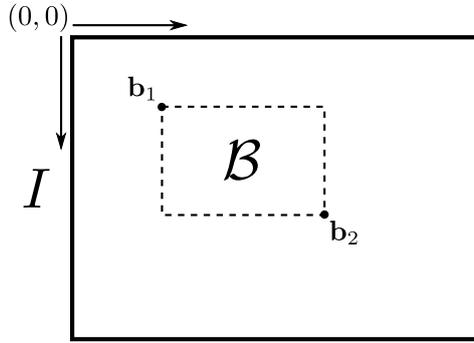


Figura 2.4: Bounding Box.

Algoritmo 2 Lucas-Kanade piramidal

Entrada: I, J imágenes, (p_x, p_y) coordenadas del punto a trackear en I

$I^0, \dots, I^n \leftarrow$ Pirámide de resolución de I

$J^0, \dots, J^n \leftarrow$ Pirámide de resolución de J

$\mathbf{d}^n \leftarrow (0, 0)^T$

for $l = n$ **to** 0 **do**

$p_x^l, p_y^l \leftarrow \frac{p_x}{2^l}, \frac{p_y}{2^l}$

$\mathbf{v}^l \leftarrow LK_{guess}(I^l, J^l, \mathbf{d}^l, \mathbf{p}^l)$

$\mathbf{d}^{l-1} \leftarrow 2(\mathbf{d}^l + \mathbf{v}^l)$

end for

$\mathbf{v} \leftarrow \mathbf{d}^0 + \mathbf{v}^0$

2.1.3. Median Flow

A los fines del tracking de objetos no es suficiente con trackear un solo punto. En un video natural, el objeto a trackear generalmente cambiará de apariencia, ya sea porque cambia de escala, color, forma o es iluminado de otra manera. Muchas veces, esto hará que la condición de brillo constante falle para varios de los pixeles que corresponden a la superficie del objeto. Si intentamos trackear al objeto por uno de estos puntos, lo perderemos fácilmente. Algo similar pasará si parte del objeto es ocluida por otro, en cuyo caso podríamos terminar trackeando un objeto diferente. Por lo tanto, queremos trackear el objeto a partir de varios puntos en el video.

Una forma muy básica pero práctica de especificar qué objeto queremos trackear es a través de un rectángulo delimitador, o *bounding box* en inglés. Por lo general un bounding box \mathcal{B} se define a partir de dos puntos, $\mathcal{B} = (\mathbf{b}_1, \mathbf{b}_2)$, que definen las esquinas superior izquierda e inferior derecha respectivamente. Usualmente, el punto $(0, 0)$ es la esquina superior izquierda del video y las coordenadas aumentan hacia la derecha y abajo (ver Figura 2.4).

Volviendo a nuestros dos cuadros de video consecutivos I y J , si nos dan un bounding box \mathcal{B}_I correspondiente al objeto a trackear en I , queremos devolver un bounding box \mathcal{B}_J que corresponda a la posición del objeto en J . Podemos correr el algoritmo LK sobre todos los pixeles adentro de la región delimitada por \mathcal{B}_I , pero dados los puntos en J resultantes, ¿cómo podemos hacer para inferir un \mathcal{B}_J que delimita la nueva posición del objeto? (Ver figura 2.5).

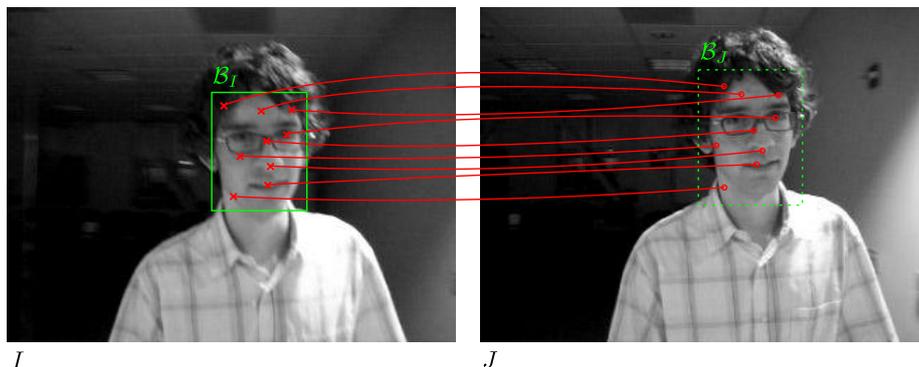


Figura 2.5: Median Flow: ¿Cómo inferir \mathcal{B}_J a partir de los puntos trackeados dentro de \mathcal{B}_I ?

Una solución *naïf* para inferir \mathcal{B}_J sería elegir la más chica que incluya a todos los puntos en J . Si nuestro algoritmo de tracking fuese infalible, esta heurística podría llegar a ser bastante buena, incluso bajo cambios de escala. Sin embargo, por los problemas ya mencionados, en realidad habrá muchos puntos con mucho error de trackeo. Si usamos estos puntos para inferir \mathcal{B}_J , sus errores se propagarán al bounding box.

Necesitamos una forma de estimar el error de trackeo de un punto en particular. Con esta información podemos filtrar los puntos y quedarnos solo con los más confiables. Veremos a continuación dos formas de estimación de error de trackeo.

Correlación cruzada normalizada (NCC)

La *correlación cruzada* es una medida de similitud utilizada en muchas áreas de procesamiento de señales. Dadas dos imágenes A y B y un desplazamiento (d_x, d_y) se define como

$$E_{CC}(d_x, d_y) = \sum_{x,y} A(x,y)B(x + d_x, y + d_y). \quad (2.29)$$

Notar que no hemos especificado el rango de x e y en la sumatoria de (2.29). En principio, podríamos limitarlo a aquellos puntos donde ambos factores dentro de la sumatoria se encuentran definidos. Equivalentemente, podemos extender A y B de manera que cualquier posición afuera de sus límites tenga valor 0 y luego hacer que (x, y) tome todos los valores dentro de los límites originales de A o B o ambos.

La correlación cruzada aumenta cuando el desplazamiento (d_x, d_y) hace que A y B estén mejor “alineadas”, en el sentido de que coincidan sus superficies. En un caso ideal, si A es igual a B salvo una traslación, entonces (2.29) será maximal cuando (d_x, d_y) coincida con dicha traslación. Si A es un parche extraído de B , entonces al maximizar (2.29) obtendremos la posición dentro de B de la cual fue extraído, salvando el caso que haya muchos parches idénticos a A dentro de B . En general, podemos maximizar la correlación cruzada para obtener la mejor forma de alinear dos imágenes.

Una desventaja de usar (2.29) es que utiliza valores absolutos, por lo que si hay un parche muy brillante en B, el máximo gravitará hacia esa zona sin importar qué tan bien se alineen las imágenes. Para prevenir esto, se define la *correlación cruzada normalizada*, abreviada NCC por sus siglas en inglés, como

$$E_{NCC}(d_x, d_y) = \frac{\sum_{x,y} [A(x, y) - \bar{A}] [B(x + d_x, y + d_y) - \bar{B}]}{\sigma_A \sigma_B} \quad (2.30)$$

donde

$$\bar{A} = \frac{1}{N} \sum_{x,y} A(x, y), \quad \bar{B} = \frac{1}{N} \sum_{x,y} B(x + d_x, y + d_y) \quad (2.31)$$

con N igual a la cantidad de puntos dentro del rango y

$$\sigma_A = \sqrt{\frac{1}{N} \sum_{x,y} [A(x, y) - \bar{A}]^2}, \quad (2.32a)$$

$$\sigma_B = \sqrt{\frac{1}{N} \sum_{x,y} [B(x + d_x, y + d_y) - \bar{B}]^2}. \quad (2.32b)$$

Notar que (2.31) y (2.32) definen simplemente el promedio y la desviación estándar, respectivamente, del brillo de las imágenes A y B trasladada. La definición de NCC no solo relativiza los valores utilizados en la sumatoria sino que también normaliza el valor de (2.30) al rango $[-1, 1]$.

Volviendo a nuestra motivación de tener una medida de error de trackeo, dado un desplazamiento estimado (u, v) podemos calcular $E_{NCC}(u, v)$ entre I y J , utilizando como rango a la ventana de integración.

Error Forward-Backward (FB)

El error Forward-Backward es una métrica basada en la robustez de una trayectoria trackeada. Sea $S_f = (I_0, I_1, \dots, I_n)$ una secuencia de imágenes. Dado un punto \mathbf{x}_0 en I_0 , supongamos que usamos un tracker arbitrario para obtener una trayectoria $T_f = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_n)$.

Para calcular el error FB, construimos primero una trayectoria de validación trackeando el último punto \mathbf{x}_n hacia atrás en el tiempo. Más precisamente, invertimos S_f para obtener $S_b = (I_n, I_{n-1}, \dots, I_0)$ y luego usamos el mismo tracker sobre esta última para obtener una trayectoria $T_b = (\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, \dots, \hat{\mathbf{x}}_n)$, con $\hat{\mathbf{x}}_0 = \mathbf{x}_n$.

Con estas dos trayectorias, podemos definir al error FB como la distancia entre ellas, para cierta noción de distancia. Una definición simple pero práctica de distancia es la distancia Euclídea entre el punto inicial de T_f y el final de T_b , por lo que

$$E_{FB}(T_f|S_f) = \text{distancia}(T_f, T_b) = \|\mathbf{x}_0 - \hat{\mathbf{x}}_n\|. \quad (2.33)$$

La idea del error FB se basa en la noción de reversibilidad temporal de una secuencia de imágenes, en el sentido que el movimiento aparente de un objeto cuando se ve la secuencia en orden invertido es precisamente el opuesto a cuando se ve en orden natural. Esto implica que si una trayectoria fue trackeada robustamente, entonces deberíamos poder trackear la trayectoria inversa con la misma robustez y deberían ser similares.

Por otro lado, cuando las condiciones para que funcione un algoritmo de tracking no se cumplen, el error introducido en la trayectoria es en cierta parte aleatorio. Lo mismo sucede cuando se trackea la trayectoria de validación. Bajo estos errores es poco probable que las trayectorias coincidan.

Algoritmo Median Flow

Con las medidas de error NCC y FB ya definidas estamos en condiciones de retomar la discusión del tracking robusto de puntos en un parche.

El algoritmo que describimos a continuación tiene el nombre de *Median Flow* [11]. Toma como entrada dos cuadros I y J y un bounding box \mathcal{B}_I . Tiene como salida otro bounding box \mathcal{B}_J , el cual representa la estimación de cómo se movió la región delimitada por \mathcal{B}_I de un cuadro al siguiente.

El primer paso es trackear puntos dentro de \mathcal{B}_I entre I y J . Si bien la posibilidad ya mencionada es trackear todos los puntos dentro de la región, podemos elegir un conjunto más pequeño de ellos, como ser una grilla de puntos o incluso generarlos usando una variable aleatoria. En este caso debemos conseguir un balance entre eficiencia y redundancia: es probable que dos puntos muy cercanos se comporten de manera tan similar que trackear a ambos no aporte más precisión que trackear a uno solo.

Luego, estimamos el error de trackeo de cada punto. Ya explicamos cómo se calcula E_{NCC} en base a la salida del algoritmo de LK. Para calcular E_{FB} utilizamos simplemente la secuencia $S_f = (I, J)$, es decir que la trayectoria T_f solamente contiene al punto trackeado y a su estimación en J .

Una vez que tenemos E_{NCC} y E_{FB} para cada punto, tomamos el 50% mejor rankeado según cada uno y de estos dos conjuntos nos quedamos con su intersección. Así, obtenemos un conjunto de puntos trackeados de manera confiable para utilizar en la estimación de \mathcal{B}_J . Identifiquemos a este conjunto con los pares $(\mathbf{x}_0, \mathbf{y}_0), \dots, (\mathbf{x}_n, \mathbf{y}_n)$, donde \mathbf{x}_i es un punto en I e \mathbf{y}_i la estimación correspondiente en J . Para estimar los parámetros de la transformación de \mathcal{B}_I a \mathcal{B}_J utilizamos la mediana, un estimador robusto ante *outliers* en los datos, de la siguiente manera:

Traslación Para estimar la traslación que sufrió el objeto, utilizamos el desplazamiento $\mathbf{d}_i \doteq \mathbf{y}_i - \mathbf{x}_i$ de cada punto. Para cada eje se toma la mediana de estos desplazamientos a lo largo de ese eje, de manera que

$$h_x = \text{mediana}(d_{0_x}, \dots, d_{n_x}), \quad h_y = \text{mediana}(d_{0_y}, \dots, d_{n_y}). \quad (2.34)$$

Escala Para estimar el cambio de escala, calculamos la distancia euclidiana entre todos los pares $(\mathbf{x}_j, \mathbf{x}_k)$ y entre todos los pares $(\mathbf{y}_j, \mathbf{y}_k)$. Sean $\mathbf{x}^{j,k} \doteq \mathbf{x}_j - \mathbf{x}_k$ e $\mathbf{y}^{j,k} \doteq \mathbf{y}_j - \mathbf{y}_k$, calculamos la proporción entre distancias correspondientes y utilizamos nuevamente la mediana de estas:

$$s = \text{mediana}\left(\left\{ \frac{\|\mathbf{x}^{j,k}\|}{\|\mathbf{y}^{j,k}\|} \mid j \neq k \right\}\right). \quad (2.35)$$

Rotación En [14] se extiende la misma idea anterior para estimar la rotación del objeto. Si bien esta estimación no se implementa en los algoritmos que veremos más adelante, se la presenta por completitud. Utiliza la diferencia

entre los ángulos que forman los pares $(\mathbf{x}_j, \mathbf{x}_k)$ e $(\mathbf{y}_j, \mathbf{y}_k)$. Se calcula el arcotangente de los vectores $\mathbf{x}^{j,k}$ e $\mathbf{y}^{j,k}$ a través de la función atan2 ¹, la cual toma en cuenta en qué cuadrante se encuentra para devolver un valor en $[-\pi, \pi]$. De estos valores se toma la mediana,

$$\theta = \text{mediana}(\{\text{atan2}(\mathbf{x}_x^{j,k}, \mathbf{x}_y^{j,k}) - \text{atan2}(\mathbf{y}_x^{j,k}, \mathbf{y}_y^{j,k}) \mid j \neq k\}). \quad (2.36)$$

Con todos estos datos, generamos \mathcal{B}_J aplicando las transformaciones correspondientes a \mathcal{B}_I . Sea $\mathcal{B}_I = (\mathbf{b}_1, \mathbf{b}_2)$, luego

$$\mathcal{B}_J = \left(s\mathbf{R}\mathbf{b}_1 + [h_x \ h_y]^T, s\mathbf{R}\mathbf{b}_2 + [h_x \ h_y]^T \right), \quad (2.37)$$

Algoritmo 3 Median Flow

Entrada: I_0, I_1, \dots, I_n cuadros de video, \mathcal{B} bounding box

for all I_i en I_0, I_1, \dots, I_{n-1} **do**

$\mathbf{x}_0, \dots, \mathbf{x}_m \leftarrow$ Sampleo de puntos dentro de \mathcal{B}

for all \mathbf{x}_j en $\mathbf{x}_0, \dots, \mathbf{x}_m$ **do**

$\mathbf{y}_j \leftarrow LK_{\text{Pyramidal}}(\mathbf{x}_j, I_i, I_{i+1})$

$\hat{\mathbf{x}}_i \leftarrow LK_{\text{Pyramidal}}(\mathbf{y}_i, I_{i+1}, I_i)$

$NCC_j \leftarrow ENCC(\mathbf{x}_j, \mathbf{y}_j, I_i, I_{i+1})$

$FB_j \leftarrow E_{FB}(\mathbf{x}_j, \hat{\mathbf{x}}_j)$

end for

$\bar{\mathbf{y}}_0, \dots, \bar{\mathbf{y}}_p \leftarrow \{\mathbf{y}_k \mid NCC_k \geq \text{mediana}(NCC)\} \cap \{\mathbf{y}_k \mid FB_k \leq \text{mediana}(FB)\}$

$\bar{\mathbf{x}}_0, \dots, \bar{\mathbf{x}}_p \leftarrow \{\mathbf{x}_k \mid NCC_k \geq \text{mediana}(NCC)\} \cap \{\mathbf{x}_k \mid FB_k \leq \text{mediana}(FB)\}$

$\mathbf{d}_0, \dots, \mathbf{d}_p \leftarrow \bar{\mathbf{y}}_0 - \bar{\mathbf{x}}_0, \dots, \bar{\mathbf{y}}_p - \bar{\mathbf{x}}_p$

$h_x, h_y \leftarrow \text{mediana}(d_{x_0}, \dots, d_{x_p}), \text{mediana}(d_{y_0}, \dots, d_{y_p})$

$\mathbf{x}^{0,1}, \dots, \mathbf{x}^{p,p-1} \leftarrow \{\bar{\mathbf{x}}_j - \bar{\mathbf{x}}_k \mid j \neq k\}$

$\mathbf{y}^{0,1}, \dots, \mathbf{y}^{p,p-1} \leftarrow \{\bar{\mathbf{y}}_j - \bar{\mathbf{y}}_k \mid j \neq k\}$

$s \leftarrow \text{mediana}\left(\left\{ \frac{\|\bar{\mathbf{y}}_j - \bar{\mathbf{y}}_k\|}{\|\bar{\mathbf{x}}_j - \bar{\mathbf{x}}_k\|} \mid j \neq k \right\}\right)$

$\theta \leftarrow \text{mediana}(\{\text{atan2}(\mathbf{x}_x^{j,k}, \mathbf{x}_y^{j,k}) - \text{atan2}(\mathbf{y}_x^{j,k}, \mathbf{y}_y^{j,k}) \mid j \neq k\})$

$\mathcal{B} \leftarrow s\mathbf{R}\mathcal{B} + (h_x, h_y)$

end for

Llegamos así a un algoritmo que trackea un objeto entre dos cuadros de video. Solo basta utilizar el segundo cuadro y bounding box correspondiente como entradas de una nueva corrida del algoritmo para trackear al objeto dentro de todo el video. Podemos ver el pseudocódigo de Median Flow en el Algoritmo 3.

Si bien en la práctica Median Flow presenta buenos resultados de precisión y robustez, tiene algunos problemas a la hora de realizar tracking a largo plazo, inherentes a algoritmos de tracking basados en flujo óptico.

Por un lado, tenemos el problema de *oclusión* del objeto a trackear, es decir que otros objetos de la escena se interpongan entre este y la cámara. En el caso de que esta oclusión sea parcial, en el mejor caso el algoritmo debería seguir

¹<http://en.wikipedia.org/wiki/Atan2>



Figura 2.6: Problemas inherentes a tracking de flujo óptico que causan trackeo erróneo: (a) Pérdida del objeto durante una oclusión parcial y (b) la acumulación de error a lo largo del tiempo (*drifting*).

trackeando la parte visible del objeto, discriminándola como confiable gracias a las medidas NCC y FB. El resultado es peor cuando el objeto es ocluido totalmente, ya que todas las hipótesis de flujo óptico dejan de valer, resultando efectivamente en la falla total de trackear el objeto. Esto también sucede en el caso que el objeto queda fuera de vista.

Por otro lado, tenemos el hecho más sutil de la acumulación de error al ir trackeando puntos a lo largo de varios cuadros. A nivel macro, el error acumulado en cada punto se presenta como una “desviación” entre la posición estimada del bounding box y la posición real del objeto en el video, o *drifting* en inglés. Obviamente, este problema se acentúa mientras más tiempo se corra el algoritmo. En el peor de los casos, algunos puntos acumularán tanto error de trackeo que ya no serán parte del objeto. Una posible solución es “resetear” el algoritmo, volviendo a tomar un bounding box de input, pero obviamente esto es de utilidad limitada si hacerlo requiere intervención humana.

Podemos ver ejemplos de ambos problemas en la Figura 2.6. En la Sección 2.2 analizamos un paradigma distinto de tracking cuyas características lo hacen más robusto ante estos casos.

2.2. Tracking por Detección

En esta sección abordamos el problema de tracking desde otra perspectiva: como un problema de *detección de objetos*. Este tipo de problemas se encuentran dentro del problema general de *reconocimiento*, es decir, recuperar automáticamente la posición e información de los objetos dentro de una escena a partir de una imagen. En el caso de detección de objetos, queremos encontrar la posición de un objeto específico cuya apariencia aproximada es indicada de antemano. Esto es en contraste a querer reconocer todos los objetos que pertenecen a cierta *clase* general de objetos, como pueden ser “perro” o “bicicleta”. En ese contexto,

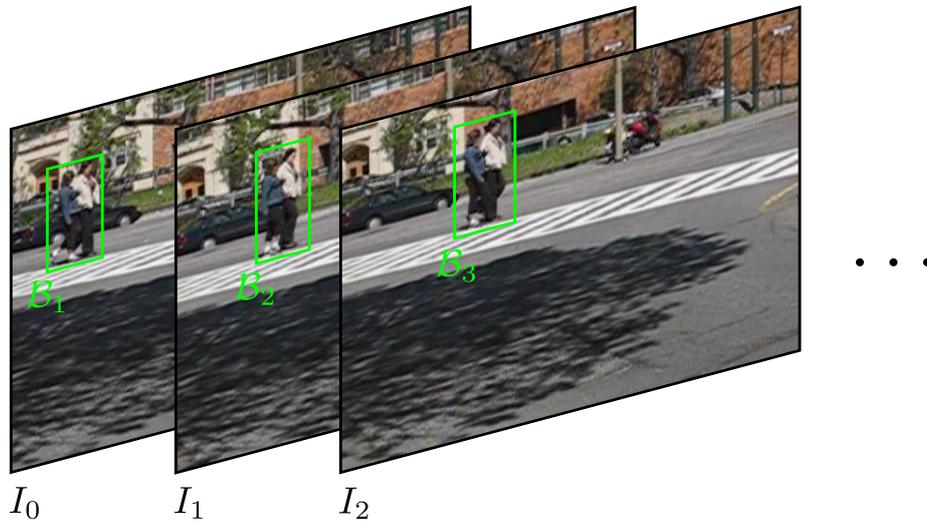


Figura 2.7: Tracking por detección: en base a una apariencia inicial, un detector identifica la posición del objeto en cada cuadro de manera independiente.

la detección de objetos se encarga de reconocer una *instancia* de la clase, es decir un perro o bicicleta en particular.

Una forma simple de trackear un objeto a partir de un algoritmo de detección es simplemente detectar el objeto en cada cuadro. La apariencia inicial del objeto estará dada por el bounding box indicado en el primer cuadro. En los cuadros subsiguientes detectamos la nueva posición del objeto, identificada por un bounding box, en base a esta apariencia. Luego, la trayectoria del objeto está dada por esta secuencia de bounding boxes (ver Figura 2.7). Notar que en este ejemplo la detección ocurre en cada cuadro independientemente y la relación temporal entre estos es solamente utilizada para ordenar estas detecciones en forma de trayectoria.

Si bien este ejemplo captura cómo se comportan en general los esquemas de tracking por detección, en la práctica hay más detalles que son tenidos en cuenta. Uno de los más importantes es el problema del cambio de apariencia del objeto a lo largo del video. Para que nuestro algoritmo de detección sea robusto se deben incorporar las nuevas apariencias del objeto a medida que lo detecta en cada cuadro. Otro problema a tratar es el caso en que el objeto es detectado en varias posiciones dentro de un mismo cuadro.

2.2.1. Detección de ventana deslizante

Los esquemas de *ventana deslizante* utilizan un detector para hacer búsqueda por fuerza bruta sobre toda la imagen. Sin entrar en detalles de su implementación, supongamos que tenemos un detector D el cual es inicializado con la apariencia del objeto en el primer cuadro. Este detector toma un cuadro I y un bounding box \mathcal{B} y devuelve 1 si detecta al objeto dentro del parche de I

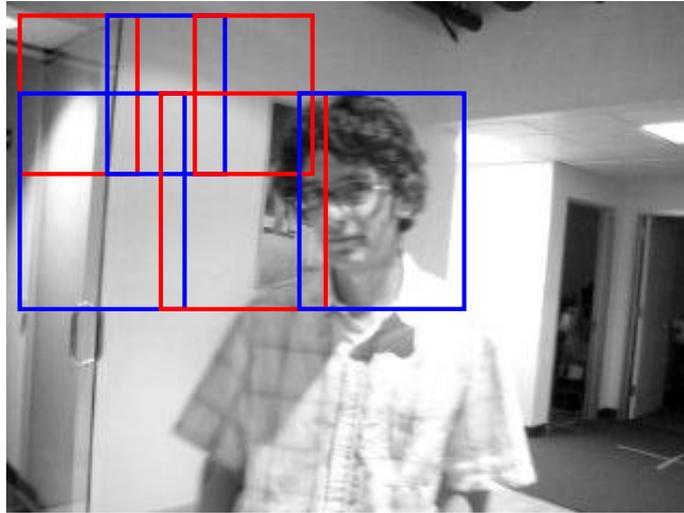


Figura 2.8: Ventana deslizante: El cuadro es cubierto por una grilla imaginaria de ventanas de distintos tamaños, en cada una se ejecutará el detector.

delimitado por \mathcal{B} o 0 en caso contrario,

$$D(I, \mathcal{B}) = \begin{cases} 1 & \text{si el objeto fue detectado,} \\ 0 & \text{caso contrario.} \end{cases} \quad (2.38)$$

Ahora, la cantidad total de bounding boxes en un cuadro es demasiado grande para calcular $D(I, \mathcal{B})$ en cada uno. Por otro lado, sería redundante calcularlo para dos bounding boxes que difieren en un pixel, ya sea horizontal o verticalmente. Por lo general, se utiliza sólo un subconjunto de todos los \mathcal{B} posibles.

Para generar este subconjunto, se toma un bounding box y se lo traslada horizontal y verticalmente en pasos fijos de x pixeles o bien relativos a las dimensiones de dicho bounding box, a lo largo de toda la imagen. Se realiza lo mismo para bounding boxes de distintos tamaños, para esto se toma como tamaño base el bounding box que identifica al objeto y se varía el factor de escala en pasos fijos de s unidades. Alternativamente y en la práctica, se escala la imagen dejando el tamaño del bounding box fijo. Valores típicos de x y s suelen ser 5 y 1,2 respectivamente. Se obtiene así una grilla multinivel que cubre toda la imagen (ver Figura 2.8), buscando garantizar que alguno de estos bounding boxes contengan exactamente al objeto, aun si este cambia substancialmente de escala.

Este tipo de esquemas tienen la ventaja de poder encontrar al objeto en cualquier parte del cuadro, sin depender para nada de su posición en los cuadros anteriores. Esto implica que son robustos ante oclusiones totales y a salidas de cuadro del objeto, en ambos casos este será detectado nuevamente en el cuadro que vuelva a ser visible. Esta cualidad les da una ventaja importante con respecto a los esquemas basados en flujo óptico, los cuales ya dijimos que fallan de manera rotunda ante tales casos.

Por otro lado, puede que el objeto sea detectado en varios bounding boxes dentro de un mismo cuadro, nuevamente en diferencia a los trackers de flujo

óptico. Esto puede suceder porque el detector marca como detecciones a varios bounding boxes muy cercanos entre sí, todos conteniendo al objeto parcial o totalmente. Otro caso es cuando el detector erróneamente detecta a un objeto similar al que se busca como tal. En el caso más extremo hay varias copias idénticas del objeto en el cuadro. Es normal que ambos casos se presenten simultáneamente, resultando en varios bounding boxes formando “grupos” alrededor de diferentes objetos (ver Figura 2.10 (a)).

Por lo general, para poder discernir entre estas múltiples detecciones, queremos aumentar nuestro detector para que nos devuelva un *puntaje* de detección s ,

$$D(I, \mathcal{B}) = \begin{cases} (1, s) & \text{si el objeto fue detectado,} \\ (0, s) & \text{caso contrario.} \end{cases} \quad (2.39)$$

Este puntaje es análogo al utilizado en la sección de flujo óptico y es un escalar que representa la confianza del detector en que el objeto se encuentra dentro de ese bounding box. Es usual que se encuentre en el rango $(0, 1)$, aunque dependiendo del detector esto puede variar. Notar que también devuelve un puntaje para las no detecciones, el cual puede ser interpretado dualmente como la confianza en que el objeto no se encuentra en ese bounding box. Veremos qué utilidad puede tener esto más adelante.

Lo primero que podemos hacer con este puntaje es aplicar un algoritmo de supresión de “no máximos” (NMS, por sus siglas en inglés). La idea es filtrar aquellos grupos de bounding boxes los cuales detectan a un mismo objeto dentro de la escena de manera que quede sólo un bounding box por grupo. Este debería ser el que más precisamente identifique al objeto y es ahí donde entra en juego el puntaje.

Un algoritmo simple de este estilo es el presentado en [8] y cuyo pseudocódigo se puede ver en el Algoritmo 4. Primero, la *intersección relativa* (overlap) de dos bounding boxes \mathcal{B}_0 y \mathcal{B}_1 se define como

$$\text{overlap}(\mathcal{B}_0, \mathcal{B}_1) = \frac{\text{area}(\mathcal{B}_0 \cap \mathcal{B}_1)}{\text{area}(\mathcal{B}_0) + \text{area}(\mathcal{B}_1) - \text{area}(\mathcal{B}_0 \cap \mathcal{B}_1)}, \quad (2.40)$$

donde el área de la intersección es según se puede ver en la Figura 2.9.

A partir de esto se ordena a todos los bounding boxes con detecciones por puntaje, de mayor a menor. Luego se procede de manera *greedy* a tomar cada bounding box de esta lista y eliminar a todos los siguientes que tengan intersección relativa de al menos c en relación a este. Queda así como “representante” de cada grupo el bounding box con mejor puntaje (ver Figura 2.10 (b)).

Una variante de este algoritmo es elegir como representante a un nuevo bounding box formado a partir del promedio de todo el grupo, cada uno ponderado por su puntaje. El puntaje de este nuevo bounding box será también el promedio de los puntajes.

Luego de realizar este filtrado puede quedar aún más de una detección, por lo que el siguiente paso es elegir una de ellas. En un caso simple en donde no tenemos ninguna otra información, una elección válida es la de mayor puntaje. Por otro lado, si aumentamos a nuestro detector con información obtenida por otro proceso, como puede ser uno basado en flujo óptico, podemos tomar decisiones más complejas con respecto a qué bounding box presentar como detección final.

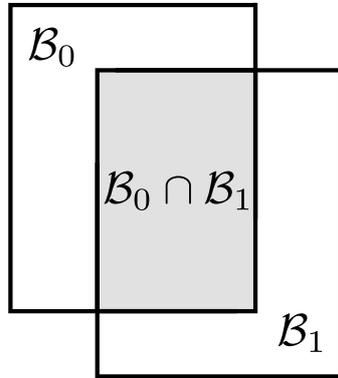


Figura 2.9: Áreas utilizadas para calcular overlap



Figura 2.10: *Supresión de no máximos (NMS)*: (a) Cada objeto es detectado en múltiples bounding boxes, (b) se elige a un representante de cada grupo.

Algoritmo 4 NMS

Entrada: $\mathcal{B}_0, \dots, \mathcal{B}_n$ bounding boxes con detecciones, s_0, \dots, s_n sus respectivos puntajes, c umbral de overlap.

Salida: Conjunto D de detecciones filtradas

$S \leftarrow \{(\mathcal{B}_0, s_0), \dots, (\mathcal{B}_n, s_n)\}$

$D \leftarrow \emptyset$

while $S \neq \emptyset$ **do**

$(\mathcal{B}, s) \leftarrow \text{máx}_{\text{puntaje}}(S)$

$D \leftarrow D \cup \{(\mathcal{B}, s)\}$

$N \leftarrow \{(\mathcal{B}', s') \in S \mid \text{overlap}(\mathcal{B}', \mathcal{B}) \geq c\}$

$S \leftarrow S \setminus N$

end while

Con todos estos ingredientes, podemos delinear un tracker basado en detección utilizando un detector como el de (2.39), cuyo pseudocódigo puede verse en el Algoritmo 5.

Algoritmo 5 Tracking basado en detección

Entrada: I_0, I_1, \dots, I_n cuadros de video, \mathcal{B} bounding box

```

init( $D, I_0, \mathcal{B}$ )
for all  $I_i$  en  $I_1, I_2, \dots, I_n$  do
     $\mathcal{B}_0, \dots, \mathcal{B}_m \leftarrow$  Bounding boxes dentro de  $I_i$ 
     $T \leftarrow \emptyset$ 
    for all  $\mathcal{B}_j$  en  $\mathcal{B}_0, \dots, \mathcal{B}_m$  do
         $d, s \leftarrow D(I_i, \mathcal{B}_j)$ 
        if  $d = 1$  then
             $T \leftarrow T \cup (\mathcal{B}_j, s)$ 
        end if
    end for
     $T \leftarrow NMS(T)$ 
     $\mathcal{B}, s \leftarrow \text{máx}_{\text{puntaje}}(T)$ 
end for

```

Un punto importante a tener en cuenta en este tipo de algoritmos es la cantidad de bounding boxes a revisar en cada cuadro, es decir el valor de n en el Algoritmo 5. Para ciertas aplicaciones es crítico que el algoritmo funcione en tiempo real, en cuyo caso se necesita poder analizar todos estos bounding boxes mientras el cuadro todavía se encuentra visible. Esto implica que la elección de $D(I, \mathcal{B})$ y su implementación deben cumplir dicha restricción.

Para analizar un ejemplo concreto, supongamos que un video tiene una resolución de 1920×1080 pixeles y se muestran 30 cuadros por segundo. Supongamos además que el bounding box indicado tiene un tamaño de 100×100 pixeles. Si tenemos 6 niveles de escala, 3 más pequeños y 2 más grandes que el dado en el cuadro inicial, con un paso de escala de 1,3, los tamaños de bounding box para cada nivel son: 46×46 , 59×59 , 77×77 , 100×100 , 130×130 y 169×169 . Si en cada nivel deslizamos los bounding boxes horizontal y verticalmente en un paso igual al 15% de su tamaño, llegamos a un total de 89794 bounding boxes a analizar en cada cuadro. Como cada cuadro se muestra sólo $\frac{1}{30}$ segundos, si queremos que nuestro algoritmo corra en tiempo real tenemos a lo sumo $\frac{1}{30 \times 89794}$ segundos, aproximadamente 371 nanosegundos. Para un procesador de 2 GHz cuya arquitectura es capaz de 2 instrucciones por ciclo, esto se traduce en 1484 instrucciones para ejecutar $D(I_i, \mathcal{B}_j)$ cada vez.

Si bien dicha cifra es relativa al poder de computación disponible, si se quiere utilizar un algoritmo de ventana deslizante, en la práctica se debe tener muy en cuenta el tipo de detector de objetos que habrá por debajo.

2.2.2. Clasificadores utilizados en detección

En esta sección nos adentramos en cómo implementar un detector de objetos. El enfoque que seguimos se basa en el *reconocimiento de patrones*, un problema mucho más general, omnipresente a lo largo de varias disciplinas. En el contexto de detección, se utiliza para reconocer los patrones dentro de la imagen que identifican la apariencia del objeto.

Al mismo tiempo, muchos algoritmos de reconocimiento de patrones incorporan esquemas de *aprendizaje automático* (conocido como *machine learning* en la literatura en inglés). Esto permite que los algoritmos tengan la capacidad de *generalizar*, es decir de reconocer correctamente patrones nunca antes vistos a partir de un conjunto inicial. En la práctica, esta capacidad resulta fundamental para obtener buenos resultados debido a la gran variabilidad que presentan los datos. En este trabajo estudiaremos exclusivamente este tipo de algoritmos de reconocimiento de patrones.

Supongamos que tomamos todos los píxeles dentro de una ventana de tamaño 100×100 y los representamos como un vector $\mathbf{x} \in \mathbb{R}^{10000}$. El objetivo es construir una función $y(\mathbf{x})$ que indique si el objeto se encuentra en la ventana,

$$y(\mathbf{x}) = \begin{cases} 1 & \text{si el objeto fue detectado,} \\ 0 & \text{caso contrario.} \end{cases} \quad (2.41)$$

Este no es un problema trivial, ya que como hemos visto la apariencia del objeto, y por lo tanto los valores de \mathbf{x} , puede variar de muchas formas posibles. Notar que esta definición es análoga a la de (2.38), solamente cambiamos a una representación que se adecúa más a los conceptos de aprendizaje automático.

Esta clase de problemas es conocida como de *clasificación*, ya que buscamos asignar a \mathbf{x} un valor discreto de un dominio finito (1 o 0 en el caso de (2.41)). Estos valores son conocidos como *categorías* o *clases*. Si hay más de dos clases C_1, \dots, C_m posibles el problema es conocido como *clasificación multiclase*, mientras que si son sólo dos se lo conoce como *clasificación binaria*. En nuestro caso, buscamos clasificar binariamente al parche de la imagen dentro de una y solo una de las clases correspondientes a “es objeto” o “no es objeto”.

La función $y(\mathbf{x})$ es conocida como *discriminante*, ya que es la que asigna a \mathbf{x} una de las clases. El discriminante junto con la maquinaria de aprendizaje automático forman en conjunto un *clasificador*.

Ahora, queremos evitar construir $y(\mathbf{x})$ a mano en base a reglas o heurísticas, por un lado porque estamos operando en un espacio de alta dimensión y por otro porque queremos codificar la noción de generalización ya mencionada. Es una mejor opción *aprender* la forma de $y(\mathbf{x})$ a partir de datos. En los esquemas de aprendizaje automático, se utiliza un conjunto $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ de posibles entradas para $y(\mathbf{x})$, conocido como el *conjunto de entrenamiento*. Para cada uno de estos ya sabemos a qué clase pertenece, es decir, tenemos un conjunto $\{t_1, \dots, t_n\}$ con $t_i \in \{0, 1\}$ tal que queremos $y(\mathbf{x}_i) = t_i$. Aquellos \mathbf{x}_i tal que $t_i = 1$ (resp. $t_i = 0$) se les llama ejemplos positivos (resp. negativos). En el caso de tracking los ejemplos positivos pueden ser, por ejemplo, parches que contienen las distintas apariencias del objeto, mientras que los negativos pueden ser parches que contienen otros objetos distintos o el fondo de la imagen.

Este conjunto de entrenamiento se utiliza para correr la *fase de entrenamiento* o *fase de aprendizaje*. A grandes rasgos, en esta fase los datos son utilizados para afinar los parámetros de un modelo estadístico. Este modelo codifica lo que el algoritmo “sabe” acerca de la apariencia del objeto y puede responder qué tan probable es que una nueva apariencia corresponda al objeto buscado. Es en relación a esto que el modelo debe poder generalizar, ya que la cantidad de ejemplos de entrenamiento suele ser sólo una fracción de todas las posibles entradas para $y(\mathbf{x})$. La fase de aprendizaje nos devuelve un discriminante $y(\mathbf{x})$ que tiene como

núcleo a ese modelo. La implementación del modelo y la fase de aprendizaje están fuertemente ligadas a la implementación elegida del clasificador, por lo que varían de igual manera.

Presentado de esta manera, el esquema de aprendizaje es uno *offline*, en el sentido que procesamos el conjunto de entrenamiento en su totalidad antes de devolver el discriminante. En la práctica, esto equivale a correr la fase de aprendizaje antes de ejecutar el algoritmo de detección propiamente dicho. Por un lado, esto implica que el algoritmo de aprendizaje no está restringido en tiempo de ejecución, por lo que hay más margen para implementar algoritmos más costosos. Por otro lado, no puede incorporar al modelo las nuevas apariencias del objeto que van surgiendo a medida que se analiza el video. Una manera de abordar este problema es correr nuevamente la fase de aprendizaje con el conjunto de aprendizaje aumentado con las nuevas apariencias, pero hacer esto regularmente en medio de la detección elimina las ventajas mencionadas anteriormente.

Alternativamente, el aprendizaje puede ser *online*, donde los ejemplos positivos y negativos son procesados uno por uno. El clasificador luego va siendo actualizado a medida que estos llegan y no hay que esperar que todos los ejemplos hayan sido procesados para utilizarlo. Esto último hace posible tener un algoritmo que aprenda nuevas apariencias en tiempo real, a medida que vamos detectando al objeto en cada cuadro.

Clasificadores en cascada

Recordemos que en un detector de ventana deslizante tenemos una gran cantidad de bounding boxes que procesar. Esto implica que queremos un clasificador robusto que detecte al objeto de manera confiable y al mismo tiempo que podamos ejecutar lo suficientemente rápido para poder procesar video en tiempo real.

Si queremos que el tracking sea a largo plazo y que el tracker pueda manejar cambios de apariencia del objeto, debemos agregar un esquema de aprendizaje online, del cual también debemos preocuparnos que sea veloz en ejecución y no crezca infinitamente en memoria.

Una manera de combinar varios clasificadores en uno solo para utilizar las ventajas de cada uno es usar un esquema en cascada. La idea es poner clasificadores en serie, con los menos robustos pero más veloces al principio. Luego, cada ventana pasa al clasificador siguiente solamente si el objeto es detectado en ella. La salida del último clasificador, el más robusto pero más intensivo en recursos, será la salida de todo el sistema.

La ventaja de este esquema es que muchos bounding boxes serán descartados por los primeros clasificadores, correspondiendo a aquellos donde es fácil darse cuenta que no contienen al objeto (por ejemplo, contienen una porción de cielo). Solo aquellos bounding boxes más ambiguos tendrán que ser procesados por los clasificadores que requieren más recursos.

En las subsecciones siguientes presentamos dos clasificadores en concreto, repasando sus ventajas y desventajas. Estos son, como veremos en la Sección 2.3.1, aquellos utilizados en un esquema en cascada por el sistema TLD. Si bien para los fines de tracking generalmente se necesita sólo clasificación binaria como se presentó al principio de esta sección, ambos clasificadores son multiclase por lo que son desarrollados como tales.

Nearest Neighbours

Supongamos que \mathbf{x} es un punto ubicado en un espacio D -dimensional. Modelemos las dos variables \mathbf{x} y C_k como generadas por una función de densidad de probabilidad conjunta $p(\mathbf{x}, C_k)$. A partir de este modelo, queremos ahora aproximar $p(C_k|\mathbf{x})$ para cada clase, es decir la probabilidad de que dado un punto \mathbf{x} este pertenezca a la clase C_k . Con esta información podemos tomar una decisión sobre a cuál clase asignarlo. Veamos primero cómo inferir $p(\mathbf{x})$ de manera no paramétrica.

Supongamos que el espacio es euclídeo y consideremos una pequeña región \mathcal{R} alrededor de un punto \mathbf{x} . La masa de probabilidad asociada a este región es

$$P = \int_{\mathcal{R}} p(\mathbf{x}) \, d\mathbf{x}, \quad (2.42)$$

es decir que una observación con distribución $p(\mathbf{x})$ tiene probabilidad P de estar dentro de \mathcal{R} . Si tenemos n observaciones, la variable aleatoria K que cuenta la cantidad de observaciones dentro de \mathcal{R} tiene una distribución binomial:

$$K \sim \text{Bin}(n, P). \quad (2.43)$$

Según (2.43), la esperanza y varianza para la fracción de los puntos dentro de la región son $\mathbb{E}[K/n] = P$ y $\text{var}[K/n] = P(1 - P)/n$, respectivamente. Esto implica que para valores grandes de n , esta varianza tiende a 0 y la distribución de K/n tiene un pico muy pronunciado alrededor de su esperanza. Luego,

$$K \simeq nP. \quad (2.44)$$

Por otro lado, si asumimos que la región es lo suficientemente pequeña para aproximar $p(\mathbf{x})$ como constante en ella, tenemos

$$P \simeq p(\mathbf{x})V \quad (2.45)$$

donde V es el volumen de \mathcal{R} . Combinando (2.44) y (2.45), obtenemos la estimación

$$p(\mathbf{x}) = \frac{K}{nV}. \quad (2.46)$$

Podemos utilizar (2.46) junto a n observaciones de una distribución para dado cualquier nuevo punto $\hat{\mathbf{x}}$ estimar $p(\hat{\mathbf{x}})$. Para esto, fijamos K y definimos a la región \mathcal{R} como una esfera centrada en $\hat{\mathbf{x}}$. Luego, aumentamos el radio de \mathcal{R} hasta que contenga exactamente K de las n observaciones. Esto determina el volumen V de \mathcal{R} a utilizar en (2.46) para aproximar $p(\hat{\mathbf{x}})$.

Notar que efectivamente estamos seleccionando las K observaciones más cercanas (en el sentido euclídeo) a $\hat{\mathbf{x}}$. Por esta razón, este esquema se conoce como el de los “ K vecinos más cercanos”, *K-nearest neighbours* en inglés y abreviado *K-NN*.

Veamos cómo utilizar este resultado para construir un clasificador. Interpretamos un conjunto de entrenamiento $\{(\mathbf{x}_1, C_{k_1}), \dots, (\mathbf{x}_n, C_{k_n})\}$ como n observaciones con distribución $p(\mathbf{x}, C_k)$, donde tenemos m clases distintas. Sean n_k la cantidad de observaciones que tienen clase C_k , de manera que $\sum_k n_k = n$. Para clasificar un nuevo punto $\hat{\mathbf{x}}$, tomamos nuevamente las K observaciones más

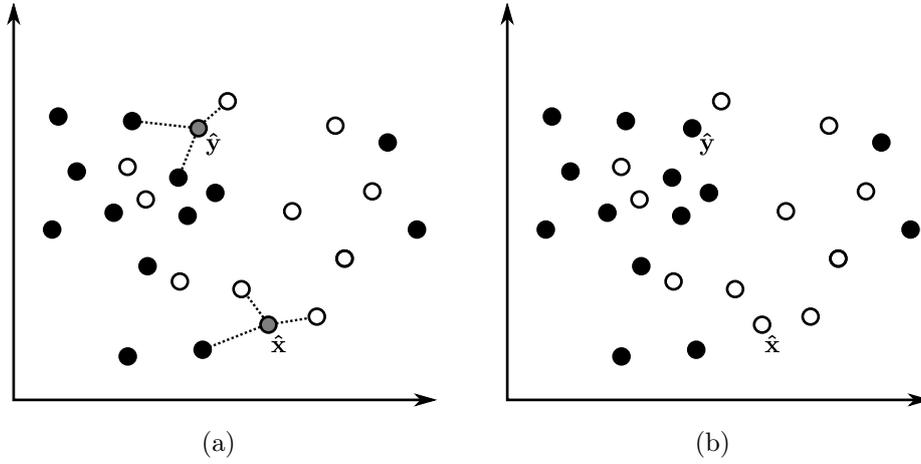


Figura 2.11: *3-Nearest Neighbours*: (a) Se consideran los tres vecinos más cercanos a los puntos a clasificar $\hat{\mathbf{x}}$ e $\hat{\mathbf{y}}$ y (b) se les asigna la clase mayoritaria entre estos.

cercanas y definimos K_k como la cantidad de estas que tienen clase C_k , por lo que también $\sum_k K_k = K$.

Para cada clase, podemos aproximar la probabilidad condicionada en esta con (2.46) de manera que

$$p(\hat{\mathbf{x}}|C_k) = \frac{K_k}{n_k V}. \quad (2.47)$$

Por otro lado, podemos estimar $p(C_k)$ con

$$p(C_k) = \frac{n_k}{n}. \quad (2.48)$$

Luego, en base a (2.46), (2.47) y (2.48) y utilizando el teorema de Bayes podemos escribir

$$p(C_k|\hat{\mathbf{x}}) = \frac{p(\hat{\mathbf{x}}|C_k)p(C_k)}{p(\hat{\mathbf{x}})} = \frac{K_k}{K}. \quad (2.49)$$

Por lo tanto, la clase C_k que maximiza (2.49) es la correspondiente al máximo de los K_k . Dicho de otra manera, debemos asignar a $\hat{\mathbf{x}}$ a la clase mayoritaria dentro de sus K vecinos más cercanos (ver Figura (2.11)). Los empates se pueden resolver aleatoriamente. El caso particular de $K = 1$ es conocido como el clasificador *nearest neighbour*, abreviado 1-NN o NN, y simplemente le asigna $\hat{\mathbf{x}}$ la clase de la observación más cercana. Podemos ver el pseudocódigo de K-NN en el Algoritmo 6.

Notar que una desventaja del algoritmo de K-NN es que requiere mantener todo el conjunto de aprendizaje en memoria todo el tiempo. Relacionado a esto, a la hora de clasificar un nuevo punto debe calcular las distancias a cada uno de los puntos del conjunto, lo que también es relativamente costoso computacionalmente. En resumen, la complejidad de K-NN en memoria y tiempo son lineales en la cardinalidad del conjunto de entrenamiento.

Para mitigar estos problemas, en la práctica se utilizan estructuras de datos y algoritmos para no tener que guardar el conjunto entero y para encontrar

Algoritmo 6 K-Nearest Neighbours

Entrada: $\{(\mathbf{x}_1, C_{k_1}), \dots, (\mathbf{x}_n, C_{k_n})\}$ conjunto de entrenamiento, $\hat{\mathbf{x}}$ punto a clasificar

Salida: C clase asignada a $\hat{\mathbf{x}}$

$(d_1, C_{k_1}), \dots, (d_n, C_{k_n}) \leftarrow (\|\mathbf{x}_1 - \hat{\mathbf{x}}\|, C_{k_1}), \dots, (\|\mathbf{x}_n - \hat{\mathbf{x}}\|, C_{k_n})$

$(\bar{d}_1, \bar{C}_{k_1}), \dots, (\bar{d}_n, \bar{C}_{k_n}) \leftarrow \text{sort}_{d_i}((d_1, C_{k_1}), \dots, (d_n, C_{k_n}))$

for $j = 1$ **to** m **do**

$c_j \leftarrow \#\{\bar{d}_i \mid i \leq K \wedge \bar{C}_{k_i} = C_j\}$

end for

$\hat{k} \leftarrow \arg \max_k c_k$

$C \leftarrow C_{\hat{k}}$

aproximaciones de los vecinos más cercanos en tiempo sublineal, respectivamente.

Por otro lado, en la Sección 3.3 veremos otra familia de clasificadores que no presentan estas desventajas.

Es fácil agregar un componente de aprendizaje online al algoritmo de K-NN: cada nueva clasificación $(\hat{\mathbf{x}}, C)$ es añadida al conjunto de aprendizaje. Esta modificación agrava más aún el problema de uso de memoria y tiempo de ejecución, ya que el tamaño del conjunto de entrenamiento aumentará indefinidamente. Para evitar esto, se puede elegir un límite para el tamaño del conjunto y luego periódicamente borrar de manera aleatoria elementos de manera de mantenerse dentro de ese límite. Una variante de esta solución es elegir los elementos a borrar de manera que queden los más “representativos” de cada clase.

Por último, también es posible incorporar la métrica utilizada al sistema de aprendizaje, de manera que en vez de ser constante sea la que mejor se adapta a los datos. En relación a esto, si bien hemos basado el análisis en la distancia euclídea, también se pueden utilizar otras métricas de distancia. Por ejemplo, la distancia de Hamming en variables discretas. Un caso particular en donde esta distancia es útil es cuando queremos clasificar imágenes o parches representados de manera binaria. En la sección siguiente vemos otro tipo de clasificador más apto a este tipo de representaciones.

Ensemble of Random Ferns

En [16] se presenta un clasificador basado en tests binarios elegidos aleatoriamente. Estructuralmente es un conjunto de clasificadores más simples llamados *ferns aleatorios*, por lo que se lo conoce justamente como “conjunto de ferns aleatorios” o *Ensemble of Random Ferns* en inglés. A diferencia de Nearest Neighbours, fue concebido originalmente para la clasificación de parches de una imagen. Veremos que su simplicidad le da ventajas tanto en implementación como en velocidad de cómputo.

Para clasificar un parche de la imagen se calculan f_1, \dots, f_n valores binarios (i.e. $f_i \in \{0, 1\}$) sobre el mismo, también llamados los *features* del parche. En [16] estos se implementan como comparaciones de la intensidad entre pares de píxeles del parche. Para que este tipo de comparación sea suficientemente robusta los parches son primero suavizados. Sea $I(\mathbf{x})$ un parche, dado un par

de coordenadas $\mathbf{x}_{j,1}$ y $\mathbf{x}_{j,2}$ podemos definir

$$f_j = \begin{cases} 1 & \text{si } I(\mathbf{x}_{j,1}) < I(\mathbf{x}_{j,2}), \\ 0 & \text{caso contrario.} \end{cases} \quad (2.50)$$

Si bien la discusión siguiente es válida para cualquier implementación de features binarios, la elección de este tipo en particular es clave tanto para la robustez de la clasificación como para la posibilidad de evaluarlos de manera rápida y fácil.

En base a este conjunto de features queremos clasificar al parche encontrando la clase C_k que maximice

$$P(C_k|f_1, \dots, f_n). \quad (2.51)$$

Si aplicamos el teorema de Bayes tenemos

$$P(C_k|f_1, \dots, f_n) = \frac{P(f_1, \dots, f_n|C_k)P(C_k)}{P(f_1, \dots, f_n)}. \quad (2.52)$$

Si asumimos que $P(C_k)$ es uniforme, como el denominador no depende de C_k y es simplemente un factor de escala el problema de maximizar (2.51) se reduce a maximizar

$$P(f_1, \dots, f_n|C_k) \quad (2.53)$$

en C_k .

Para estimar (2.53) de manera empírica se requeriría llevar una tabla con 2^n entradas por cada clase. Esto no es factible para valores de n que no sean pequeños. En particular, como las comparaciones de intensidad son una representación muy simple, en [16] utilizan $n \approx 300$.

Una forma de abordar este problema es asumir la independencia condicional de los features dada la clase,

$$P(f_1, \dots, f_n|C_k) = \prod_{j=1}^n P(f_j|C_k). \quad (2.54)$$

Sin embargo, este modelo puede resultar muy poco robusto ya que no tiene para nada en cuenta las correlaciones entre los features.

Un balance entre calcular (2.53) directamente y utilizar (2.54) consiste en particionar los features en h grupos de tamaño $s = \frac{n}{h}$. Sea $F_i = (f_{\sigma_i(1)}, \dots, f_{\sigma_i(s)})$, con $i = 1, \dots, h$ y σ_i una permutación aleatoria en el rango $1, \dots, n$. Estas estructuras son las llamadas ferns aleatorios y asumiendo independencia entre ellas tenemos

$$P(f_1, \dots, f_n|C_k) = \prod_{i=1}^h P(F_i|C_k). \quad (2.55)$$

Para la estimación empírica de (2.55) necesitamos solamente $h \times 2^s$ entradas por cada clase. De esta manera, regulando los valores de h y s podemos elegir un balance entre espacio necesario en memoria y robustez del clasificador. En [16] utilizan h entre 30 y 50 y s alrededor de 11, necesitando solo unas 80000 entradas en comparación a las 2^{300} mencionadas anteriormente.

Notar que existe una biyección entre los valores que puede tomar F_i y los números de 0 a $2^s - 1$, la cual consiste en interpretar $(f_{\sigma_i(1)}, \dots, f_{\sigma_i(s)})$ como

$$(a) F_1 \left(\text{img} \right) = (f_{\sigma_1(1)}, f_{\sigma_1(2)}, f_{\sigma_1(3)}, f_{\sigma_1(4)}) \rightarrow (0011)_2 = 3$$

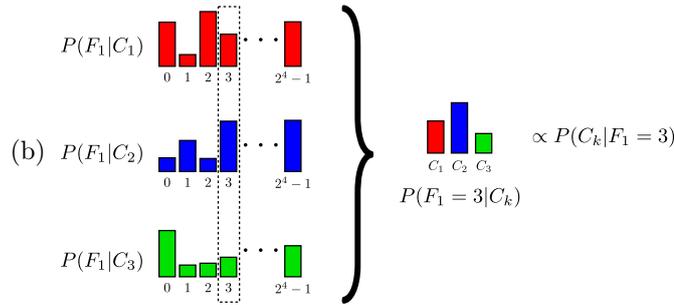


Figura 2.12: Ferns aleatorios: (a) Las comparaciones de píxeles determinan un número k entre 0 y $2^s - 1$. (b) A partir de este índice se obtienen fácilmente los valores de $P(F_i = k|C_k)$, los cuales en conjunto son proporcionales a $P(C_k|F_i = k)$, según (2.52).

los dígitos de un número en base 2. Esto nos permite usar la notación $F_i = m$ con m entre 0 y $2^s - 1$ para simplificar la notación siguiente.

El conjunto de entrenamiento utilizado para este clasificador es un conjunto de parches con sus respectivas clases. Sobre cada parche de este conjunto se calculan las comparaciones de píxeles correspondientes a cada fern de manera de obtener un nuevo conjunto de entrenamiento “procesado”. Con estos valores luego se estima

$$P(F_i = k|C_k = c) = \frac{N_{k,c} + 1}{N_c + 2^s}, \quad (2.56)$$

donde $N_{k,c}$ es la cantidad de parches en el conjunto de entrenamiento que evalúan a k con las comparaciones correspondientes al fern F_i y tienen clase c , y N_c es la cantidad de parches con clase c . Los términos 1 y 2^s en el numerador y denominador respectivamente actúan simplemente como regularizadores para evitar que la probabilidad sea 0 cuando $N_{k,c} = 0$ porque no hubo ningún observación de ese estilo.

Para clasificar un parche, nuevamente se calculan las comparaciones de píxeles de manera de obtener un valor k_i por cada fern F_i (ver Figura 2.12). Según (2.55), resta solamente calcular

$$\arg \max_c \prod_{i=1}^h P(F_i = k_i|C_k = c). \quad (2.57)$$

En el Algoritmo 7 vemos el pseudocódigo del entrenamiento de este clasificador [15]. Notar que a pesar de describir la implementación en bajo nivel, sigue siendo relativamente simple en comparación a los clasificadores vistos anteriormente. Por otro lado, requiere solo $h \times s$ comparaciones de píxeles y $h \times m$ multiplicaciones para calcular todas las probabilidades necesarias.

Estas cualidades hacen que este clasificador sea una buena elección que ofrece simplicidad y velocidad en aquellos casos que los features basados en comparaciones de pixeles ofrecen un buen grado de robustez.

Algoritmo 7 Ensemble of Random Ferns (entrenamiento)

Entrada: I parche a clasificar entre m clases, F_i ferns aleatorios con $i = 1, \dots, h$

Salida: $P_{I|C}[j] = P(f_1, \dots, f_n|C_j)$ para $j = 1, \dots, m$

```

for  $j = 1$  to  $m$  do
     $P_{I|C}[j] \leftarrow 0$ 
end for
for all fern  $F_i$  do
     $\text{index} \leftarrow 0$ 
    for  $k = 1$  to  $s$  do
         $\text{index} \leftarrow \text{index} \ll 1$ 
        if  $I(\mathbf{x}_{\sigma_i(k),1}) < I(\mathbf{x}_{\sigma_i(k),2})$  then
             $\text{index} \leftarrow \text{index} + 1$ 
        end if
    end for
    for  $j = 1$  to  $m$  do
         $P_{I|C}[j] \leftarrow P_{I|C}[j] \times P_{F_i}[\text{index}, j]$ 
    end for
end for

```

2.3. Modelos Mixtos

En las secciones anteriores hemos visto dos enfoques diametralmente distintos al problema de tracking.

El tracking por flujo óptico utiliza la información local y temporal de los pixeles que representan al objeto para seguirlo a lo largo de los cuadros. Para movimientos pequeños y poco cambio de apariencia son muy robustos mientras que al mismo tiempo su esquema de búsqueda local resulta en algoritmos rápidos de ejecutar. Sus mayores desventajas son la falla completa si el objeto es ocluido totalmente y el drifting a causa de la acumulación de error.

Por otro lado, el tracking por detección se basa en la búsqueda global del objeto en cada cuadro independientemente y se le acoplan esquemas de aprendizaje automático para aprender nuevas apariencias de manera online. Esto lo hace más propicio al tracking a largo plazo, ya que puede volver a trackear el objeto aun cuando entra y sale de la escena y cuando cambia de apariencia a lo largo del tiempo. Sin embargo, la conjugación de clasificación, aprendizaje y búsqueda en todo el cuadro hace que sean muy intensivos en recursos necesarios para el tracking en tiempo real.

En [12] los autores plantean que para el problema de tracking a largo plazo, ninguno de estos dos paradigmas es satisfactorio por sí mismo. Surge de esta proposición la idea de crear un sistema que use estos dos enfoques conjuntamente, para de esta manera poder abordar los diferentes problemas de cada uno de ellos. Sin embargo, no es trivial combinar estos dos esquemas que, como ya dijimos, tienen bases muy distintas.

2.3.1. TLD

El tracker *TLD* [12] aborda el problema de tracking a largo plazo planteando un sistema formado por tres componentes: un tracker basado en flujo óptico, uno basado en detección y un esquema de aprendizaje diseñado en base a ambos. En el marco de TLD estos son conocidos como los componentes de “tracking”, “detección” y “aprendizaje” respectivamente. Por lo tanto, en la discusión siguiente se llamará “tracker” al componente de tracking por flujo óptico y “detector” al basado en detección. La sigla TLD se refiere a los nombres de sus componentes en inglés: *Tracking, Learning, Detection*.

Los componentes de tracking y detección funcionan de manera separada y simultánea. El tracker sigue al objeto cuadro a cuadro y el detector busca al objeto en todo el cuadro en base a su apariencia. La salida de estos dos sistemas luego es procesada por una lógica de integración para poder identificar un único bounding box donde se encuentra el objeto.

Por otro lado, el componente de aprendizaje actualiza el modelo del detector, también en base a información de los otros dos componentes. El esquema de aprendizaje utilizado se conoce como *P-N Learning* [10] y fue creado por los mismos autores para un sistema como TLD. Su diseño fue planteado en vista a tres objetivos: ser robusto ante secuencias complejas donde el tracker falla frecuentemente, no degradar el detector si el video no contiene información relevante y correr en tiempo real.

En el sistema TLD, el objeto es representado alternativamente por un bounding box que indica su posición o un valor nulo que indica que no se encuentra visible en el cuadro. De esta manera, se pueden tener trayectorias discontinuas del objeto a lo largo del video, por ejemplo cuando el objeto sale del cuadro pero luego vuelve a entrar y es vuelto a trackear.

El módulo de tracking es simplemente una implementación del algoritmo Median Flow presentado en la Sección 2.1.3. A este se le añade una simple heurística para detectar cuándo falla en trackear el objeto. Sean \mathbf{d}_i los desplazamientos de los puntos y \mathbf{d}_m la mediana de estos. Si $\text{mediana}(|\mathbf{d}_i - \mathbf{d}_m|)$ es mayor a un umbral de 10 píxeles, se considera que el tracker ha perdido al objeto y se lo informa al sistema.

En lo que respecta al detector, para representar la apariencia del objeto se utilizan parches de la imagen, los cuales son normalizados a una resolución de 15×15 . Se define en base a dos parches la similitud entre ellos como

$$S(p_i, p_j) = 0,5(NCC(p_i, p_j) + 1). \quad (2.58)$$

El modelo de la apariencia del objeto se encuentra formado por un conjunto de parches $M = \{p_1^+, p_2^+, \dots, p_m^+, p_1^-, p_2^-, \dots, p_n^-\}$, donde los p_i^+ son apariencias mismas del objeto que funcionan como ejemplos positivos y los p_i^- son parches tomados de alrededor del objeto y funcionan como ejemplos negativos. Adicionalmente, los parches positivos se encuentran en el orden que fueron agregados al modelo, con p_m^+ el parche más reciente. Este modelo es el utilizado por el detector y es actualizado por el módulo de aprendizaje.

A partir de un parche p y un modelo M se definen las siguientes métricas de similitud:

$$S^+(p, M) = \max_{p_i^+ \in M} S(p, p_i^+), \quad (2.59)$$

$$S^-(p, M) = \max_{p_i^- \in M} S(p, p_i^-), \quad (2.60)$$

$$S_{50\%}^+(p, M) = \max_{p_i^+ \in M \wedge i < \frac{m}{2}} S(p, p_i^+), \quad (2.61)$$

$$S^r(p, M) = \frac{S^+(p, M)}{S^+(p, M) + S^-(p, M)}, \quad (2.62)$$

$$S^c(p, M) = \frac{S_{50\%}^+(p, M)}{S_{50\%}^+(p, M) + S^-(p, M)}. \quad (2.63)$$

Notar que $S_{50\%}^+$ calcula la máxima similitud entre el 50% de parches positivos más viejos. Por otro lado, S^r es la *similitud relativa* y tiene un valor entre 0 y 1. De la misma manera, S^c es la *similitud conservadora* y es análoga a la relativa pero utilizando $S_{50\%}^+$.

El módulo de detección de TLD se basa en un esquema de detección de ventana deslizante como el visto en la Sección 2.2.1. Como se explicó en dicha sección, estos esquemas deben procesar muchos bounding boxes en un tiempo muy pequeño. En TLD se utiliza un clasificador en cascada de tres etapas para abordar esta restricción.

Filtro de varianza La primer etapa utiliza un filtro de varianza para descartar bounding boxes. La varianza de la intensidad de un parche se calcula como $\mathbb{E}[\mathbf{x}^2] - \mathbb{E}^2[\mathbf{x}]$, donde \mathbf{x} son los valores de intensidad dentro del parche. Cada bounding box cuya varianza sea menor al 50% de la varianza del parche seleccionado inicialmente es descartado. Este primer filtro puede ser evaluado muy rápido utilizando “imágenes integrales”², las cuales una vez calculadas permiten saber la varianza de cualquier parche de la imagen en tiempo constante.

Ensemble of Random Ferns Para la segunda etapa utiliza un clasificador Ensemble of Random Ferns que distingue entre dos clases, parches positivos y negativos. Los features binarios son calculados sobre los parches 15×15 que forman el modelo. Estos son utilizados como los ejemplos positivos y negativos para entrenar al clasificador. Se obtienen así las probabilidades $P(F_i, C_k = 1)$, las cuales son calculadas para clasificar un nuevo parche. Por último, se clasifica al parche como positivo si el promedio de todas estas supera 0,5. Notar que esto es a diferencia de multiplicarlas como sugiere (2.57).

Nearest Neighbour La última etapa consiste en un clasificador Nearest Neighbour. Al igual que el clasificador Ensemble, este utiliza los ejemplos positivos y negativos del modelo para clasificar un parche en una de dos clases. La métrica de distancia utilizada en este paso es la similitud relativa definida en (2.62). Un parche p es clasificado como el objeto si $S^r(p, M) > \theta_{NN}$, con el umbral $\theta_{NN} = 0,6$.

²http://en.wikipedia.org/wiki/Integral_image

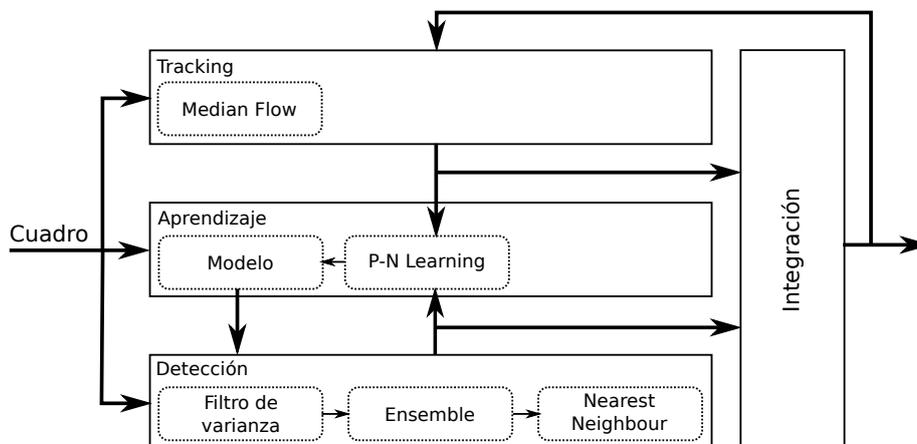


Figura 2.13: Esquema de TLD.

Podemos ver en la Figura 2.13 un esquema de estas tres etapas. De las aproximadamente 50000 ventanas que tiene el sistema como entrada solo 50 suelen llegar a la última etapa de clasificación.

La salida del tracker y el detector deben ser combinadas de manera de obtener un solo bounding box. Si ninguno de los dos pudo trackear correctamente al objeto, es decir si el tracker falló y el detector no tiene ninguna detección, simplemente se declara al objeto como no visible. De lo contrario, se toma a todos los bounding boxes del detector y al bounding box del tracker y se elige al que tenga la mayor similitud conservadora en el modelo. Si el bounding box elegido no es el del tracker o si este ha fallado en trackear el objeto, la lógica de integración lo reinicializa con el mismo.

Por último, tenemos el esquema de P-N Learning, el cual está conformado por dos “expertos”: el *P-expert* y el *N-expert*. El rol de estos expertos es estimar y corregir los errores del detector. Más específicamente, en cada cuadro el P-expert estima cuáles parches son falsos negativos, es decir parches positivos que fueron clasificados negativamente por el detector. Estos falsos negativos son luego introducidos como parches positivos al modelo. El N-expert hace lo análogo para los falsos positivos. Podemos ver este esquema en la Figura 2.13. La clave de estos expertos para estimar los errores es explotar la estructura espacial y temporal del video.

El P-expert utiliza la estructura temporal y asume que el objeto se mueve a lo largo de una trayectoria consistente. En base a esto, utiliza la posición del objeto en el cuadro anterior para estimar dónde se encuentra en el cuadro actual. Luego, los parches cerca de la posición esperada del objeto que fueron clasificados como negativos son asumidos como falsos negativos por el P-expert y marcados como positivos. En TLD, la salida del mismo módulo de tracking es utilizada como la estimación de la posición del objeto.

El N-expert en cambio utiliza la información espacial del video y asume que el objeto aparece sólo una vez en el cuadro. Dadas todas las apariencias detectadas del objeto en un cuadro, elige la más confiable como la posición verdadera de este y todas las demás son consideradas falsos positivos y marcados como negativos. En TLD, el N-expert utiliza las detecciones más confiables del detector y las

compara con el bounding box elegido por la lógica de integración. Aquellas que se encuentren a una distancia considerable de este último son tomadas como falsos positivos.

Para que ambos expertos hagan buenas estimaciones es necesario que sus hipótesis estén bien fundadas. Por esta razón, el modelo sólo aprende si se considera que el objeto está siendo trackeado confiablemente. TLD considera a una trayectoria de trackeo como confiable desde el momento en que para un parche p de esta se tiene $S^c(p, M) > \theta_{core}$, donde θ_{core} es una constante predefinida. La trayectoria sigue siendo considerada confiable hasta que el tracker falla o es reinicializado.

Aún así, la estimación de cada experto también contiene cierto error. Sin embargo, los autores demuestran que bajo ciertas condiciones estos errores se cancelan entre sí y el esquema de aprendizaje con P-N Learning resulta en una mejora neta de los resultados de detección. En la sección siguiente vemos en más detalle el formalismo utilizado para demostrar esto.

Es importante notar que el esquema de P-N Learning utiliza tanto la salida del tracker como el detector, combinándolas de manera de mejorar el modelo de este último. A su vez, el detector es quién reinicializa al tracker si este falla. Esta interdependencia entre los módulos es la manera en que TLD aprovecha ambos enfoques y los combina para obtener mejores resultados de tracking.

En relación a esto, los resultados experimentales publicados en [12] muestran que en 9 de 10 de los casos de prueba, TLD consigue los mejores resultados en comparación a otros trackers considerados como estado del arte al momento de la publicación. En cierta manera esto muestra el lugar de importancia que tienen este tipo de modelos a la hora de realizar tracking. Los resultados también muestran que en el caso específico de tracking a largo plazo, TLD es capaz de seguir correctamente un objeto por muchos más cuadros que la mayoría de los trackers rivales.

P-N Learning

Como ya dijimos, P-N Learning es uno de los puntos claves para combinar tracking y detección en TLD y en sí mismo resulta un aporte muy interesante de sus autores. Veamos más de cerca la base formal que lo fundamenta.

Un esquema de P-N Learning consiste en un clasificador binario $f : X \rightarrow \{0, 1\}$, un conjunto de entrenamiento inicial (X_l, Y_l) , un conjunto de datos sin categorizar X_u y un par de expertos. El rol de los expertos es aumentar el conjunto de entrenamiento con los datos sin categorizar para mejorar al clasificador.

El clasificador es inicializado entrenándolo con el conjunto (X_l, Y_l) . Luego, se usa esta primer versión para clasificar a todo el conjunto X_u , de manera de obtener un conjunto clasificado (X_u, Y_u) . Este último conjunto es luego analizado por los expertos, quienes estiman cuántas y cuáles de estas clasificaciones son erróneas. Este subconjunto de falsos positivos y falsos negativos es incorporado al conjunto de entrenamiento con sus clases cambiadas. El clasificador es ahora entrenado con el conjunto de entrenamiento aumentado y se itera el mismo proceso con los datos que todavía quedan en X_u . Podemos ver un esquema de este proceso en la Figura 2.14.

Presentado de este manera, tenemos un esquema de aprendizaje offline en donde el conjunto X_u está determinado de antemano. Este mismo proceso se puede realizar online, clasificando cada dato de x_u por separado, estimando si

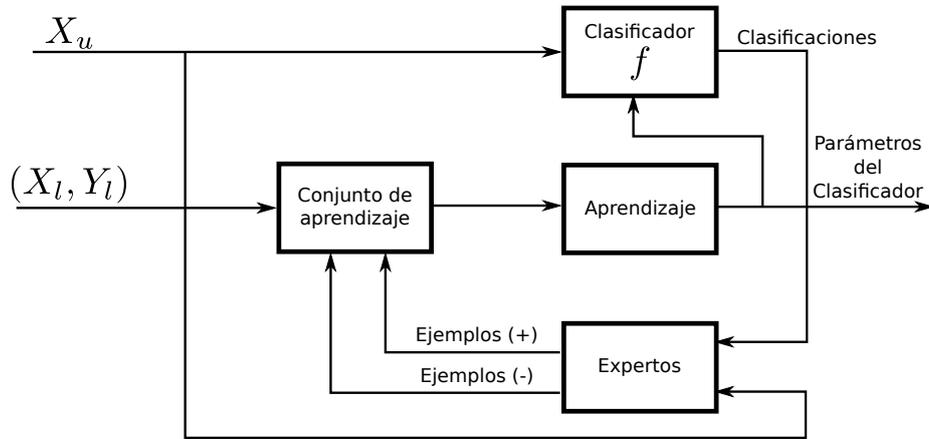


Figura 2.14: Esquema de P-N Learning.

es un falso positivo o falso negativo y luego añadiéndolo al conjunto de entrenamiento. Obviamente esto implica un clasificador entrenado también de manera online. Si bien en esta sección se demuestra la convergencia para el esquema offline, la prueba también es válida para la versión online.

Como explicamos anteriormente, los dos expertos se conocen como el P-expert y el N-expert, encargados de estimar por separado los falsos negativos y falsos positivos, respectivamente. Dada una iteración k , sea $n^+(k)$ la cantidad de falsos negativos que el P-expert añade al conjunto de entrenamiento como ejemplos positivos y sea $n^-(k)$ la cantidad de falsos positivos que el N-expert agrega como ejemplos negativos.

Para este análisis es clave tener en cuenta el error mismo que cometen los expertos en estimar ejemplos con la clase incorrecta. Caractericemos el error del clasificador como las cantidades *reales* de falsos negativos y falsos positivos en el conjunto de entrenamiento, sean estas $\alpha(k)$ y $\beta(k)$. Por otro lado, sea $n_c^+(k)$ la cantidad de ejemplos que fueron estimados correctamente como falsos negativos por el P-expert y sea $n_f^+(k) = n^+(k) - n_c^+(k)$ la cantidad de ejemplos estimados incorrectamente. Sean $n_c^-(k)$ y $n_f^-(k)$ lo análogo para el N-expert.

El error del clasificador en la iteración $k + 1$ luego puede ser escrito como

$$\alpha(k+1) = \alpha(k) - n_c^-(k) + n_f^+(k), \quad (2.64)$$

$$\beta(k+1) = \beta(k) - n_c^+(k) + n_f^-(k). \quad (2.65)$$

Según (2.64), el error disminuye si $n_c^-(k) > n_f^+(k)$, es decir si la cantidad de estimaciones correctas del N-expert supera la cantidad de estimaciones erróneas del P-expert. Sucede lo análogo con (2.65).

Definamos

$$P^+ = \frac{n_c^+}{n_c^+ + n_f^+}, \quad R^+ = \frac{n_c^+}{\beta}, \quad (2.66)$$

$$P^- = \frac{n_c^-}{n_c^- + n_f^-}, \quad R^- = \frac{n_c^-}{\alpha}, \quad (2.67)$$

donde P^+ es el P-precision, R^+ el P-recall, P^- el N-precision y R^- el N-recall y donde el parámetro k queda implícito para simplificar la notación. Luego,

$$n_c^+(k) = R^+ \beta(k), \quad n_f^+(k) = \frac{1 - P^+}{P^+} R^+ \beta(k), \quad (2.68)$$

$$n_c^-(k) = R^- \alpha(k), \quad n_f^-(k) = \frac{1 - P^-}{P^-} R^- \alpha(k). \quad (2.69)$$

En base a (2.68) y (2.69) podemos reformular a (2.64) y (2.65) como

$$\alpha(k+1) = (1 - R^-) \alpha(k) + \frac{1 - P^+}{P^+} R^+ \beta(k), \quad (2.70)$$

$$\beta(k+1) = \frac{1 - P^-}{P^-} R^- \alpha(k) + (1 - R^+) \beta(k). \quad (2.71)$$

Por último, definiendo un vector de estado $\vec{x}(k) = (\alpha(k), \beta(k))^T$ y una matriz

$$\mathbf{M} = \begin{bmatrix} 1 - R^- & \frac{1 - P^+}{P^+} R^+ \\ \frac{1 - P^-}{P^-} R^- & 1 - R^+ \end{bmatrix}, \quad (2.72)$$

podemos reescribir (2.70) y (2.71) como

$$\vec{x}(k+1) = \mathbf{M} \vec{x}(k). \quad (2.73)$$

Esta última ecuación puede ser interpretada como un sistema dinámico discreto.

A partir de esto último, se puede demostrar que el vector \vec{x} converge a cero si los autovalores λ_1, λ_2 de la matriz de transición \mathbf{M} son ambos menores a uno [10]. Esto se debe a que la matriz corresponde a una transformación lineal en el espacio de los errores de clasificación $\alpha(k), \beta(k)$. Como tal, los autovalores corresponden a factores de escala de la transformación que \mathbf{M} aplica a \vec{x} en cada iteración, en las direcciones de los correspondientes autovectores. Por lo tanto, si ambos son menores a uno, este vector y en consecuencia la cantidad de errores se verá reducida en cada iteración.

Esta es la única restricción que P-N Learning impone sobre la eficacia del clasificador y los expertos para garantizar mejoras. Dicho de otra manera, no importan los valores individuales de P^+, R^+, P^-, R^- mientras que λ_1, λ_2 sean menores a uno. Por lo tanto, es posible que los errores del sistema se vean reducidos aun cuando alguno de los componentes no realiza buenas predicciones.

En [10] se pone a prueba y valida este marco teórico utilizando un sistema parecido al de TLD, con experimentos tanto sintéticos como reales. En estos últimos, se observó que efectivamente aun bajo condiciones difíciles de trackeo, donde el clasificador tiene dificultades detectando el objeto, los autovalores son menores a uno y la clasificación mejora gracias a P-N Learning. Por otro lado, en los ejemplos peor condicionados, algunos autovalores son aproximadamente 0,99 y la mejoría es mínima.

Puntos débiles

Como vimos en la Sección 2.2.2, el uso de memoria de un clasificador NN aumenta linealmente con la cantidad de elementos en el conjunto de entrenamiento. Lo mismo sucede con la complejidad temporal para clasificar nuevos

elementos. Esta característica es indeseada en términos de tracking a largo plazo. En el caso de TLD, implicaría que la cantidad de parches en el modelo aumente indefinidamente, ya que siempre se debe agregar parches nuevos para mantener una apariencia reciente del objeto. Claramente, esto llevaría a un consumo ilimitado de memoria. Al mismo tiempo, clasificar nuevos parches se volvería cada vez más lento.

Para evitar esto, TLD admite una cantidad máxima de parches en el modelo. Para acomodar nuevos parches, se debe introducir una heurística para borrar (“olvidar”) parches viejos. La implementación de TLD elige aleatoriamente parches a borrar, lo cual funciona satisfactoriamente en la práctica. De todas maneras, es difícil discernir a nivel teórico cómo influye este comportamiento en la robustez del sistema, principalmente en relación al esquema de P-N Learning.

A su vez, estas limitaciones llevan a tener que implementar un modelo de clasificación en cascada. Es decir, minimizar la cantidad de parches que llegan hasta la etapa de NN, tanto para detección como para aprendizaje. Esto complejiza el sistema TLD tanto a nivel teórico como práctico. En relación a esto último, la implementación de TLD incluye dos clasificadores no triviales en términos de complejidad de código y de los recursos de memoria y procesamiento que necesitan. A esto se le suma tener que combinar este pipeline con el esquema de aprendizaje.

El parámetro θ_{core} también puede ser interpretado como una heurística que busca limitar los recursos que utiliza el sistema, evitando aprender todo el tiempo. Si bien esto está justificado ya que se busca evitar aprender apariencias erróneas del objeto, sería interesante tener una métrica más objetiva de cuándo se debe aprender. Por otro lado, este parámetro se compara con la similitud conservadora, la cual está definida para el 50% de parches más viejos y por lo tanto se ve afectada por el borrado aleatorio de parches.

En el Capítulo 3 combinamos nuevos esquemas de detección y aprendizaje bajo un marco basado en TLD, buscando mejorar algunos de estos puntos manteniendo la robustez que presenta el sistema TLD en relación al tracking a largo plazo.

Capítulo 3

TLD-FV

3.1. Introducción

En la Sección 2.3 se presentó al tracker TLD, el cual presenta resultados muy buenos en lo que refiere a tracking en general y en particular a largo plazo. También se discutieron algunos puntos débiles de los esquemas de detección que utiliza. Principalmente se habló de la desventaja de utilizar un clasificador Nearest Neighbour y el uso de memoria asociado.

En este trabajo se estudia cómo se pueden mejorar estos aspectos del sistema TLD, manteniendo las ideas claves que le permiten mostrar un gran desempeño. Principalmente, se cambian la representación y modelo de la apariencia del objeto por un esquema cuyo uso de memoria sea constante con respecto al tiempo. Este esquema a su vez permite utilizar un clasificador lineal en vez de NN, evitando así el costo computacional de este último a la hora de clasificar parches. Por último, se mantiene un esquema de P-N Learning pero adaptado a estos cambios, buscando preservar las propiedades de este que hacen al tracker robusto.

3.2. RANSAC Flock of Trackers

En lo que respecta al módulo de tracking, se buscó que Median Flow fuese más robusto tanto en la estimación del movimiento como en fallar más rápido si empieza a presentar drift. Con esto se buscó mejorar aún más cómo evoluciona el modelo de apariencia del objeto, evitando aprender apariencias erróneas a causa de errores del tracker. En [21], los autores muestran cómo mejorando el filtrado de puntos “mal trackeados” en Median Flow se pueden obtener mejores resultados. Para esto, proponen nuevos predictores de falla de trackeo locales.

Como se explicó en la Sección 2.1.3, Median Flow utiliza las medidas E_{NCC} y E_{FB} para el filtrado de puntos. En particular, calcular E_{FB} requiere correr el algoritmo Lucas-Kanade dos veces, hacia adelante y hacia atrás. Si interpretamos a Median Flow como un algoritmo del tipo *Flock of Trackers* como se presenta en [21], la estimación del algoritmo LK es un conjunto de observaciones de un modelo de movimiento. A su vez, los puntos que se consideran bien trackeados serán *inliers* mientras que los mal trackeados serán *outliers* de este conjunto.

En base a esto, se utilizó el algoritmo RANSAC [9]. Este es un algoritmo general para la estimación robusta de parámetros a partir de un conjunto de observaciones que se asume contienen outliers. A grandes rasgos, cada observación tiene una estimación de parámetros asociada, la cual funciona como su “voto” para la estimación global. En base a estos votos, el algoritmo divide a las observaciones en conjuntos de inliers y outliers, asumiendo que hay suficientes de los primeros como para discernir un modelo consistente y que los votos de los segundos no “conspiran” para votar consistentemente por otro modelo.

Utilizando como voto de cada punto su estimación de movimiento LK, se puede obtener una estimación de los mismos parámetros que en Median Flow original: traslación, rotación y cambio de escala. En [21] se demuestra que esta estimación resulta más robusta que la basada en la mediana de los puntos filtrados a partir de E_{ECC} y E_{FB} .

En el Algoritmo 8 se encuentra el pseudocódigo de este tracker.

Algoritmo 8 RANSAC Flock of Trackers

Entrada: I_0, I_1, \dots, I_n cuadros de video, \mathcal{B} bounding box
for all I_i en I_0, I_1, \dots, I_{n-1} **do**
 $\mathbf{x}_0, \dots, \mathbf{x}_m \leftarrow$ Muestreo de puntos dentro de \mathcal{B}
 for all \mathbf{x}_j en $\mathbf{x}_0, \dots, \mathbf{x}_m$ **do**
 $\mathbf{t}_j \leftarrow LK_{Piramidal}(\mathbf{x}_j, I_i, I_{i+1})$
 end for
 $s, \mathbf{R}, h_x, h_y \leftarrow RANSAC(\mathbf{t}_0, \dots, \mathbf{t}_m)$
 $\mathcal{B} \leftarrow s\mathbf{R}\mathcal{B} + (h_x, h_y)$
end for

3.3. Clasificación lineal

Como mejora al módulo de detección, se buscó utilizar otro tipo de clasificador que no requiera mantener el conjunto de entrenamiento en memoria. Los clasificadores *lineales* son capaces de cumplir tal condición, ya que solo deben guardar los parámetros del modelo subyacente. Veamos cómo se definen más concretamente.

Como vimos en la Sección 2.2.2, un clasificador se encarga de asignar los vectores de entrada a una de varias clases, dos en nuestro caso. Esto implica que el espacio de los vectores de entrada es dividido en regiones, también llamadas *regiones de decisión*. El límite entre estas regiones es el *límite de decisión* o *superficie de decisión*.

Los clasificadores lineales son aquellos cuyos límites de decisión son funciones lineales de \mathbf{x} . Para un espacio D -dimensional, el límite de decisión será un hiperplano de dimensión $D - 1$ (ver Figura 3.1). Notar que, dependiendo de los datos, no siempre será posible encontrar un límite de decisión lineal que los separe limpiamente. Aquellos conjuntos para los que esto es posible son llamados *linealmente separables*.

Analíticamente, un discriminante lineal se puede definir de manera general como

$$y(\mathbf{x}) = f(\mathbf{w}^T \mathbf{x} + w_0) \quad (3.1)$$

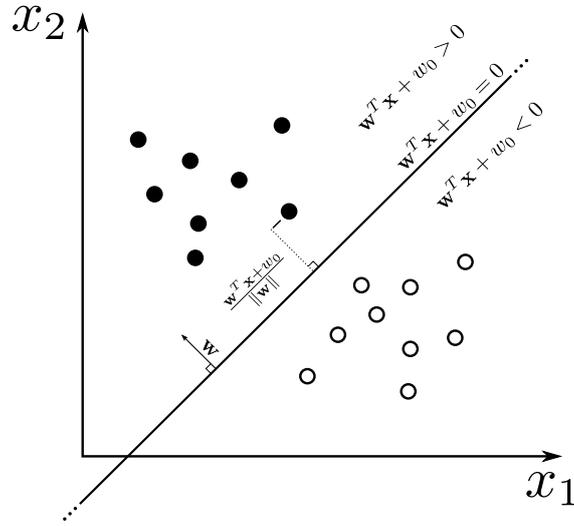


Figura 3.1: Clasificador lineal.

donde los valores del vector \mathbf{w} junto con w_0 son los parámetros del modelo lineal y la función f se conoce como función de activación, encargada de transformar el valor de $\mathbf{w}^T \mathbf{x} + w_0$ en 0 o 1. Para nuestro caso será simplemente

$$f(x) = \begin{cases} 1 & x \geq 0, \\ 0 & x < 0. \end{cases} \quad (3.2)$$

Analizando (3.1) y (3.2) podemos ver que aquellos valores tales que $\mathbf{w}^T \mathbf{x} + w_0 \geq 0$ serán asignados a una clase y para los que $\mathbf{w}^T \mathbf{x} + w_0 < 0$ a la otra. Efectivamente, el límite de decisión es el hiperplano definido por la ecuación $\mathbf{w}^T \mathbf{x} + w_0 = 0$.

Ahora, suponiendo que los datos son linealmente separables, existen infinitos hiperplanos distintos que los separan. Podemos exigir más condiciones para achicar el espacio de soluciones. En particular, una de estas condiciones puede ser que la superficie de decisión sea de *máximo margen*. Esto quiere decir que se maximiza la distancia de la superficie hacia los ejemplos más cercanos de cada clase (ver Figura 3.2). Este esquema es la base del funcionamiento de los clasificadores *Support Vector Machine* [3].

3.3.1. Función de pérdida

Existen varios algoritmos para aprender, a partir de un conjunto de entrenamiento, los parámetros \mathbf{w} y w_0 que mejor separan las clases. Una familia de estos se basa en optimizar los parámetros de manera indirecta a través de la minimización de una *función de pérdida*.

Para el caso de dos clases, dada una función g parametrizada según \mathbf{w} y w_0 y un conjunto de entrenamiento $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$, una función de pérdida tiene la forma

$$L(y_i, g(\mathbf{x}_i; \mathbf{w}, w_0)) \quad (3.3)$$

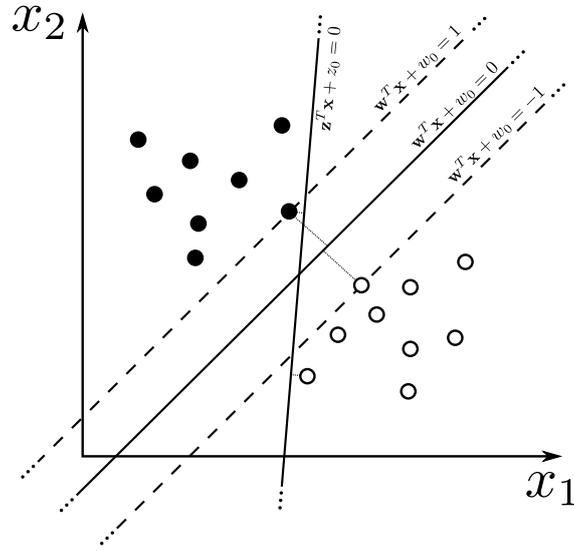


Figura 3.2: Clasificadores de máximo margen: \mathbf{w} maximiza la distancia de la superficie de decisión a los puntos más cercanos de cada clase y $\|\mathbf{w}\|$ es tal que ambos son clasificados con puntaje 1 y -1 . \mathbf{z} también separa correctamente todos los puntos, pero no es de máximo margen.

La función g representa la decisión que el clasificador toma sobre \mathbf{x}_i y se puede definir de varias maneras, por ejemplo $g(\mathbf{x}_i) = y(\mathbf{x}_i)$ o $g(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i + w_0$.

La idea de esta función de pérdida es tener una medida de qué tan indeseable es clasificar un ejemplo particular a la clase incorrecta. Por ejemplo, podemos definirla de manera que clasificar erróneamente un ejemplo a la clase 0 sea mucho más grave que hacer lo mismo a la clase 1. Esta medida luego guía la minimización, de manera de llegar a una solución que evita cometer los errores de clasificación menos deseados.

A partir de (3.3), definimos el *riesgo empírico* de todo el conjunto como

$$\frac{1}{n} \sum_{i=1}^n L(y_i, g(\mathbf{x}_i; \mathbf{w}, w_0)). \quad (3.4)$$

Buscamos luego los parámetros que minimicen (3.4),

$$\bar{\mathbf{w}}, \bar{w}_0 = \arg \min_{\mathbf{w}, w_0} \frac{1}{n} \sum_{i=1}^n L(y_i, g(\mathbf{x}_i; \mathbf{w}, w_0)). \quad (3.5)$$

Función de pérdida *hinge*

La función de pérdida *hinge* [2] se define como

$$H(y_i, g(\mathbf{x}_i; \mathbf{w}, w_0)) = \max(0, 1 - y_i g(\mathbf{x}_i)), \quad (3.6)$$

donde

$$g(\mathbf{x}_i; \mathbf{w}, w_0) = \mathbf{w}^T \mathbf{x}_i + w_0. \quad (3.7)$$

Esta función tiene la propiedad de que $H(y_i, g(\mathbf{x}_i)) = 0$ si $\text{sign}(y_i) = \text{sign}(g(\mathbf{x}_i))$ y $|g(\mathbf{x}_i)| > 1$, es decir que para los pares clasificados correctamente y a suficiente distancia de la superficie de decisión la pérdida es 0. Por otro lado, si los signos son diferentes, la pérdida aumenta linealmente.

Otra propiedad de esta función es que al minimizar el riesgo empírico asociado podemos obtener los parámetros de un clasificador lineal que separa los datos con una superficie de decisión de máximo margen. Aún más, lo hace de tal manera que para los ejemplos más cercanos de cada clase, sean estos \mathbf{x}_a y \mathbf{x}_b , se tiene que $g(\mathbf{x}_a) = 1$ y $g(\mathbf{x}_b) = -1$ (ver Figura 3.2).

3.3.2. Descenso de gradiente

Para aproximar el mínimo del riesgo empírico, podemos utilizar el método iterativo de *descenso de gradiente* [2]. Sea $F(x)$ la función a minimizar, asumiendo que esta es diferenciable y dada una estimación inicial $x^{(0)}$, podemos implementar la siguiente iteración:

$$x^{(k+1)} = x^{(k)} - \eta_k \nabla F(x^{(k)}). \quad (3.8)$$

Dado que la gradiente indica el sentido en que F crece más rápido desde $x^{(k)}$, la idea es “descender” en el sentido contrario, es decir, en el que F decrece más rápido.

El parámetro η_k es conocido como el *tamaño del paso de descenso* y regula qué tan “rápido” se mueve la estimación en cada paso. Para η_k lo suficientemente pequeño, se tiene que

$$F(x^{(k)}) \geq F(x^{(k+1)}). \quad (3.9)$$

Si todos los η_k cumplen dicha condición, tenemos

$$F(x^{(0)}) \geq F(x^{(1)}) \geq F(x^{(2)}) \geq \dots \quad (3.10)$$

y las iteraciones convergen a un mínimo local de F bajo condiciones favorables, como por ejemplo si F es convexa (ver Figura 3.3). Por lo tanto, es un parámetro crítico tanto para la convergencia del algoritmo como para la rapidez de esta. En el contexto de aprendizaje automático, a η_k se lo conoce como la *tasa de aprendizaje*.

Si bien podríamos aplicar esta técnica para minimizar el riesgo empírico directamente, existe una variante llamada *descenso de gradiente estocástico* [2] para el caso que la función a minimizar tenga la forma

$$F(x) = \sum_{i=1}^n F_i(x), \quad (3.11)$$

como es el caso de la ecuación (3.4), salvo una constante.

Este algoritmo consiste en actualizar la estimación utilizando sólo un dato $F_i(x)$ por vez, es decir:

$$x^{(k+1)} = x^{(k)} - \eta_n \nabla F_i(x^{(k)}). \quad (3.12)$$

El algoritmo realiza una pasada sobre todos los datos para llegar a la estimación final. Es posible que se necesiten varias de estas iteraciones para converger a

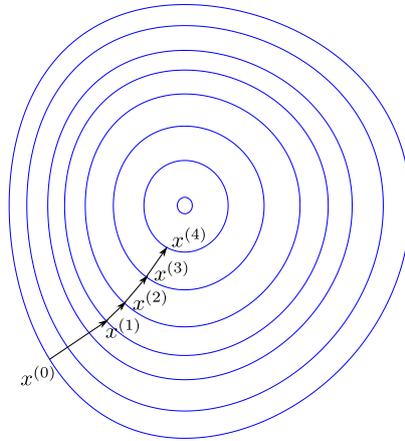


Figura 3.3: Descenso de gradiente: Cada paso desde $x^{(i)}$ es en la misma dirección que la gradiente en ese punto y por lo tanto perpendicular a la isolnea que pasa por él.

una solución aceptable. Podemos ver el pseudocódigo de descenso de gradiente estocástico en el Algoritmo 9.

Algoritmo 9 Descenso de gradiente estocástico

Entrada: $\mathbf{x}^{(0)}$ estimación inicial, η tasa de aprendizaje (constante), n número máximo de iteraciones.

Salida: \mathbf{x} mínimo aproximado

$\mathbf{x} \leftarrow \mathbf{x}^{(0)}$

repeat

for $i = 1$ **to** n **do**

$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla F_i(\mathbf{x})$

end for

until Se considera \mathbf{x} una aproximación adecuada

Esta versión tiene la ventaja de evitar tener que evaluar $\nabla F(x)$, optando por evaluar $\nabla F_i(x)$, la cual probablemente es más simple de calcular. Por otro lado, la propiedad de actualizar los parámetros de a un solo dato hace que sea intrínsecamente online.

Por último, vale la pena notar que la función *hinge* definida en la Sección 3.3.1 no es diferenciable para $y_i g(\mathbf{x}_i) = 1$, por lo que no podríamos usar descenso de gradiente para su minimización. Sin embargo, es una función convexa, lo que implica que podemos extender el algoritmo para usar un subgradiente de la función en sus puntos no diferenciables y obtener los mismos resultados de convergencia [4].

3.3.3. Vectores de Fisher

Hasta ahora no se ha discutido en detalle de qué manera se representa un parche como un vector \mathbf{x} . Sin embargo, en el caso de la clasificación lineal el tipo de representación es crítico para obtener buenos resultados.

Los Vectores de Fisher [17] ofrecen una manera robusta y eficiente de representar a parches de la imagen. Esta representación se basa en los features de bajo nivel del parche, los cuales se pueden calcular con bajo costo computacional. A su vez, ha sido utilizada satisfactoriamente junto a clasificadores lineales.

Sea $S = \{p_\lambda\}$ una familia de distribuciones parametrizada por un vector $\lambda = (\lambda_1, \dots, \lambda_M)^T$. Dado p_λ perteneciente a S , un conjunto de observaciones X puede ser caracterizado como

$$\nabla_\lambda \log p_\lambda(X). \quad (3.13)$$

Intuitivamente, esta gradiente describe la dirección hacia la que los parámetros deben ser modificados para que p_λ mejor explique las observaciones en X .

Dados dos conjuntos de observaciones X_a y X_b , el *Kernel de Fisher* se define como

$$K(X_a, X_b) = [\nabla_\lambda \log p_\lambda(X_a)]^T I_\lambda^{-1} [\nabla_\lambda \log p_\lambda(X_b)], \quad (3.14)$$

donde I_λ es la matriz de información de Fisher. Este funciona como una medida de similitud entre X_a y X_b basada en cómo afectarían al modelo si se modificaran sus parámetros en la dirección de (3.13).

La matriz I_λ se puede descomponer como $I_\lambda^{-1} = L_\lambda^T L_\lambda$. A partir de esto, podemos reescribir (3.14) como

$$K(X_a, X_b) = [L_\lambda \nabla_\lambda \log p_\lambda(X_a)]^T [L_\lambda \nabla_\lambda \log p_\lambda(X_b)]. \quad (3.15)$$

El *Vector de Fisher* de X se define a partir de (3.15) como

$$FV(X) = L_\lambda \nabla_\lambda \log p_\lambda(X). \quad (3.16)$$

Este puede considerarse como una representación del conjunto X en el espacio vectorial \mathbb{R}^M , donde la dimensión M depende solo de la distribución p_λ y no de la cardinalidad de X .

Por último, si asumimos que $X = \{x_1, \dots, x_n\}$ es un conjunto de observaciones independientes, podemos escribir (3.16) como

$$FV(X) = \frac{1}{N} \sum_{n=1}^N L_\lambda \nabla_\lambda \log p_\lambda(x_n). \quad (3.17)$$

Definición de p_λ para descriptores binarios

Supongamos que extraemos features de bajo nivel de una imagen en particular, los cuales están representados como un conjunto $X = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ de descriptores D -dimensionales. Si $p_\lambda : \mathbb{R}^D \rightarrow \mathbb{R}_+$ es una distribución que modela de manera general la generación de estos descriptores, luego $FV(X)$ es una manera robusta de representar a la imagen en el espacio \mathbb{R}^M . Es a partir de esta propiedad que los vectores de Fisher resultan favorables para la clasificación de imágenes.

En [17] se utilizan descriptores que se asumen de manera general en \mathbb{R}^M y por lo tanto se elige a p_λ como una mezcla de distribuciones Gaussianas. Sin embargo, en nuestro caso queremos utilizar descriptores rápidos de computar, como son los basados en features binarios. Por lo tanto, se siguió la extensión propuesta en [19], donde se desarrolla explícitamente al marco de vectores de Fisher con p_λ en la familia de distribuciones conformada por mezclas de C distribuciones de Bernoulli multivariantes.

Por otro lado, al utilizar este tipo de distribuciones es mejor normalizar a los vectores de Fisher resultantes para obtener mejores resultados experimentales. Se utilizó la función de normalización descrita en [18]:

$$f(z) = \text{sign}(z) \sqrt{\frac{|z|}{\|z\|_1}}. \quad (3.18)$$

3.3.4. Clasificación lineal con vectores de Fisher

En base a los conceptos y técnicas discutidos anteriormente en esta sección, se creó un nuevo módulo de detección para reemplazar al de TLD original. Este utiliza vectores de Fisher para representar a los parches de la imagen y un clasificador lineal para la detección propiamente dicha. Esta combinación resulta en un detector con menor utilización de recursos, en comparación al esquema de Nearest Neighbour de TLD.

La representación de los parches como vectores de Fisher está basada en descriptores binarios BRIEF [5]. Dado un pixel, los diferentes bits de su descriptor se calculan a partir de comparaciones de píxeles pertenecientes a una región cercana. El cálculo de este descriptor se realiza para todos los píxeles de una grilla que cubre a la imagen, de manera que la distancia entre ellos es de dos píxeles. Luego, el conjunto de descriptores de un parche es el conjunto de descriptores de los píxeles de esta grilla que caen dentro de él.

Si modelamos a estos descriptores como generados por una mezcla de distribuciones de Bernoulli multivariantes, podemos obtener una representación vectorial de un parche calculando el vector de Fisher de su conjunto de descriptores.

Sin embargo, ya hemos visto que en un esquema de ventana deslizante es importante poder procesar una gran cantidad de parches en poco tiempo. Esto se traduce a que debemos calcular la representación de cada parche de manera eficiente.

Para empezar, sea $X = \{\mathbf{d}_1, \dots, \mathbf{d}_n\}$ el conjunto de descriptores BRIEF de un parche. Asumiendo la independencia entre los \mathbf{d}_i , podemos utilizar (3.17) para calcular fácilmente $FV(X)$ como el promedio de los $FV(\mathbf{d}_i)$. Ahora, si precomputamos $FV(\mathbf{d}_i)$ para todos los descriptores \mathbf{d}_i posibles, el cálculo de la representación de un parche se convierte simplemente en buscar los $FV(\mathbf{d}_i)$ en una tabla y calcular el promedio de estos.

Para que dicho precómputo sea posible, la cantidad de bits b de los descriptores debe ser tal que sea factible calcular una tabla de tamaño 2^b . Sin embargo, generalmente los descriptores BRIEF tienen 256 o 512 bits, correspondientes a esa cantidad de comparaciones de píxeles. Por lo tanto, se tuvo que optar por una versión menos robusta que solo utiliza 16 bits.

El cálculo del promedio de los vectores también se puede hacer de manera más eficiente. Utilizando como base al concepto de *imágenes integrales* [20], dada una matriz A de tamaño $n \times m$, podemos definir una matriz integral $S(A)$ de tamaño $(n + 1) \times (m + 1)$ como

$$S(A)_{i,j} = \sum_{k=0}^{i-1} \sum_{l=0}^{j-1} A_{k,l}. \quad (3.19)$$

A partir de (3.19), podemos calcular la suma de los valores de cualquier submatriz de A con la siguiente fórmula:

$$\sum_{i=i_0}^{i_1} \sum_{j=j_0}^{j_1} A_{i,j} = S(A)_{i_1+1,j_1+1} - S(A)_{i_0,j_1+1} - S(A)_{i_1+1,j_0} + S(A)_{i_0,j_0}. \quad (3.20)$$

A partir de la grilla de descriptores BRIEF de la imagen, podemos obtener el vector de Fisher de cada uno de estos, resultando en una nueva grilla de vectores de Fisher. Si estos vectores se encuentran en \mathbb{R}^M , podemos pensar a cada coordenada $1 < m \leq M$ por separado como una matriz V^m de valores reales, donde $V_{i,j}^m$ corresponde al valor en la coordenada m del vector de Fisher en la posición (i, j) de la grilla. Si aplicamos (3.19) en cada V^m , luego podemos utilizar (3.20) coordenada a coordenada para calcular la suma de los vectores de Fisher que caen dentro de un parche. El último paso consiste en, a partir de esta suma, calcular (3.17) y aplicar la normalización (3.18).

Podemos así calcular la representación de un parche de manera lineal en M e independientemente del tamaño del parche. Se eligió modelar los descriptores con una mezcla de 32 distribuciones de Bernoulli, ya que los vectores de Fisher resultantes se encuentran en \mathbb{R}^{544} . Esto implica que se deben realizar 544 cálculos de matrices integrales por cuadro.

Podemos ver el pseudocódigo de este proceso en el Algoritmo 10. La función *INTEGRAL* calcula las matrices integrales como se explicó en los párrafos anteriores. Si bien el pseudocódigo procesa un solo cuadro, notar que el precómputo de los \mathbf{p}_d debe realizarse sólo una vez, independientemente de la cantidad de cuadros a procesar.

Una vez que tenemos a todos los parches de la imagen representados en el espacio \mathbb{R}^M , queremos aplicar clasificación binaria para detectar el objeto. Para esto, entrenamos un clasificador lineal con descenso de gradiente estocástico, optimizando la función de pérdida hinge. Los ejemplos positivos serán obviamente aquellos que contengan al objeto, mientras que los ejemplos negativos aquellos que contienen otras partes de la imagen.

Como conjunto de aprendizaje inicial, los ejemplos positivos son aquellos que comparten como mínimo 70 % del área con el bounding box inicial mientras que los negativos aquellos que comparten menos de 20 % del área con este. Esto es similar a TLD, con la diferencia de que podemos entrenar con un conjunto mucho más grande al no tener que guardarlo en memoria. Efectivamente, veremos que si se toman todos los parches negativos posibles en la imagen, se obtienen mejoras en la precisión del clasificador inicial.

Dependiendo el tamaño del bounding box inicial, puede que se tengan muchos más ejemplos negativos que positivos o viceversa. Para evitar que una clase domine a la otra durante el aprendizaje, se agregan pesos a cada una. Estos pe-

Algoritmo 10 Cálculo eficiente de vectores de Fisher

Entrada: I imagen con resolución $2m \times 2n$, $\mathcal{B}_0, \dots, \mathcal{B}_v$ bounding boxes**Salida:** $\mathbf{f}_0, \dots, \mathbf{f}_v$ vectores de Fisher normalizados de los bounding boxes**for** $d = 0$ **to** $2^{16} - 1$ **do** $\mathbf{p}_d \leftarrow FV(\{(d)_{base2}\})$ **end for**
$$\begin{bmatrix} \mathbf{d}_{1,1} & \dots & \mathbf{d}_{1,m} \\ \vdots & \ddots & \vdots \\ \mathbf{d}_{n,1} & \dots & \mathbf{d}_{n,m} \end{bmatrix} \leftarrow BRIEF(I)$$
$$\begin{bmatrix} \mathbf{s}_{1,1} & \dots & \mathbf{s}_{1,m} \\ \vdots & \ddots & \vdots \\ \mathbf{s}_{n,1} & \dots & \mathbf{s}_{n,m} \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{P}(\mathbf{d}_{1,1})_{base10} & \dots & \mathbf{P}(\mathbf{d}_{1,m})_{base10} \\ \vdots & \ddots & \vdots \\ \mathbf{P}(\mathbf{d}_{n,1})_{base10} & \dots & \mathbf{P}(\mathbf{d}_{n,m})_{base10} \end{bmatrix}$$
$$\begin{bmatrix} \bar{\mathbf{s}}_{1,1} & \dots & \bar{\mathbf{s}}_{1,m+1} \\ \vdots & \ddots & \vdots \\ \bar{\mathbf{s}}_{n+1,1} & \dots & \bar{\mathbf{s}}_{n+1,m+1} \end{bmatrix} \leftarrow INTEGRAL \left(\begin{bmatrix} \mathbf{s}_{1,1} & \dots & \mathbf{s}_{1,m} \\ \vdots & \ddots & \vdots \\ \mathbf{s}_{n,1} & \dots & \mathbf{s}_{n,m} \end{bmatrix} \right)$$
for $i = 0$ **to** v **do** $(x_0, y_0), (x_1, y_1) \leftarrow \lceil \frac{\mathcal{B}_i}{2} \rceil$ $area \leftarrow (x_1 - x_0 + 1) \times (y_1 - y_0 + 1)$ $\mathbf{f}_i \leftarrow (\bar{\mathbf{s}}_{y_1+1, x_1+1} - \bar{\mathbf{s}}_{y_0, x_1+1} - \bar{\mathbf{s}}_{y_1+1, x_0} + \bar{\mathbf{s}}_{y_0, x_0}) / area$ $\mathbf{f}_i \leftarrow sign(\mathbf{f}_i) \sqrt{\frac{|\mathbf{f}_i|}{\|\mathbf{f}_i\|_1}}$ **end for**

Los factores de escala afectan directamente a la función de pérdida, como un factor de escala de la pérdida total de cada ejemplo. Se calculan de manera que cada clase tenga igual “representación” en el riesgo empírico sin importar la cantidad de ejemplos en cada una.

Conforme nuevas apariencias del objeto o nuevos ejemplos negativos van apareciendo, estos pueden usarse para actualizar al clasificador de manera on-line, que como ya mencionamos se puede hacer fácilmente al usar descenso de gradiente estocástico.

A diferencia de Nearest Neighbour, este clasificador no debe guardar las representaciones de los parches. Como vimos en la ecuación (3.1), solo es necesario guardar en memoria el vector \mathbf{w} y w_0 . En nuestro caso, \mathbf{w} se encuentra en \mathbb{R}^{544} , por lo que solo hay que guardar 545 valores reales. Aun más importante, este valor se mantiene constante a lo largo de toda la ejecución.

A su vez, para clasificar un parche simplemente se debe evaluar (3.1), lo cual conlleva realizar 544 multiplicaciones y sumas, algo muy rápido de hacer en los procesadores actuales.

3.4. TLD-FV

El aporte de este trabajo es un nuevo tracker basado en TLD, llamado *TLD-FV*. Sigue la misma estructura que TLD original, pero reemplaza a los módulos de tracking y detección con el tracker de flujo óptico RANSAC Flock of Trackers

y el clasificador lineal basado en vectores de Fisher y descenso de gradiente, respectivamente.

En lo que respecta al módulo de detección, se mantiene el filtro de varianza pero las dos últimas etapas de la clasificación en cascada son reemplazadas íntegramente por el clasificador lineal. Esto es posible por sus mejores características de performance tanto en memoria como en cómputo, que le permiten procesar una mayor cantidad de bounding boxes que el clasificador Nearest Neighbour solo. De hecho, el filtro de varianza se mantiene no por motivos de performance, sino para filtrar de antemano posibles falsos positivos que degraden el modelo.

Uno de los parámetros importantes del clasificador lineal es el *umbral de la función de activación*. Los clasificadores aprendidos a partir de la función de pérdida hinge son tales que una región cercana a la superficie de decisión, aquella donde el puntaje de clasificación está entre -1 y 1, contiene los parches “difíciles de clasificar”. Por lo tanto, si se varía el umbral de 0 a 1, se puede manejar la cantidad de estos parches que serán considerados detecciones.

La integración de las hipótesis del tracker y el detector se mantiene similar a la usada en TLD. Se agrega solamente una heurística para darle prioridad al tracker con respecto a las detecciones. Esta se basa en ponderar el puntaje de las detecciones según su distancia al bounding box actual, de manera que a mayor distancia tenga menor puntaje. Esto es a raíz de que el modelo subyacente del clasificador es propenso a dar falsos positivos de alto puntaje, a diferencia del NN original, por lo que es preferible darle más peso a la hipótesis actual.

El esquema de P-N Learning se mantiene igual al original, aprovechando que su desarrollo teórico está planteado para clasificadores en general. En el caso de los falsos positivos elegidos por el N-expert, nuevamente podemos aprovechar el hecho de estar usando un clasificador lineal y elegir todos los parches en vez de sólo un subconjunto.

La actualización del modelo se realiza de manera natural a través del esquema de aprendizaje online inherente a descenso de gradiente estocástico. Esto lo hace más simple que TLD original, donde no solo hay que preocuparse por mantener actualizadas todas etapas del clasificador en cascada sino que en el caso de NN hay que elegir parches a olvidar. En nuestro caso, el clasificador lineal va “olvidando” las apariencias viejas de manera implícita al ir modificando la superficie de decisión en base a los ejemplos nuevos. En relación a esto, el algoritmo cuenta con dos parámetros importantes: tasa de aprendizaje y cantidad de iteraciones. Ambos regulan qué tan rápido se adapta el modelo a las nuevas apariencias del objeto.

Por último, a diferencia de TLD original, es factible mantener el algoritmo de aprendizaje corriendo en todos los cuadros, siempre y cuando haya una detección del objeto. En las pruebas esto resulta en un mejor desempeño, sobre todo en aquellos casos que el objeto se mueve muy rápido y varía en apariencia drásticamente en poco tiempo.

Resumiendo, TLD-FV busca superar algunas limitaciones de TLD en lo que respecta a uso de memoria y procesamiento, utilizando un esquema de detección totalmente distinto basado en clasificadores lineales. En el Capítulo 4 veremos cómo este sistema se comporta en comparación al original.

Capítulo 4

Experimentos

4.1. Secuencias de prueba

Para verificar la validez de las modificaciones de TLD-FV con respecto a TLD, se realizaron experimentos sobre secuencias de imágenes. Estas secuencias son un subconjunto de las utilizadas por los autores de TLD en [12]. De estas se omiten algunas de las secuencias más largas ya que por limitaciones de eficiencia de la implementación prototipo de TLD-FV llevan mucho tiempo en ser evaluadas. También se omiten aquellas secuencias que incluyen grandes cambios de escala, ya que en la implementación actual el detector no está preparado para tales situaciones.

En la Figura 4.1 podemos ver cuadros de ejemplo de estas secuencias. Los objetos a trackear en estas secuencias sufren varias de las condiciones críticas esperadas en un contexto de tracking a largo plazo. Como ejemplo, la secuencia David contiene amplias variaciones de iluminación, un fondo cambiante y cambios de pose del objeto. En el caso de Pedestrian 1, Pedestrian 2 y Pedestrian 3 tenemos al objeto ocluido parcial y totalmente, varias veces y de manera temporal y además hay rápidos movimientos de cámara. En Jumping el objeto se mueve muy rápidamente y tiene cambios abruptos de dirección, como también gran cantidad de cuadros borrosos.

Cada una de estas secuencias viene anotada manualmente con la posición del objeto en cada cuadro que este aparece. Esta secuencia de bounding boxes o valores nulos (para los cuadros que el objeto no está visible) es conocida como *ground truth*. Para evaluar el resultado de correr TLD-FV en una secuencia, se compara la salida de este contra el *ground truth* cuadro a cuadro.

Si tanto TLD-FV como el *ground truth* coinciden en que el objeto es visible y los bounding boxes tienen una intersección relativa, según (2.40), mayor a 0,25, se lo considera un *verdadero positivo*. Si estos bounding boxes no coinciden según esta medida, se lo considera un *falso positivo*. Si TLD-FV da un bounding box cuando el objeto no está visible, se lo considera también un *falso positivo*. Por último, si TLD-FV no da un bounding box cuando el objeto está visible, se lo considera un *falso negativo*.

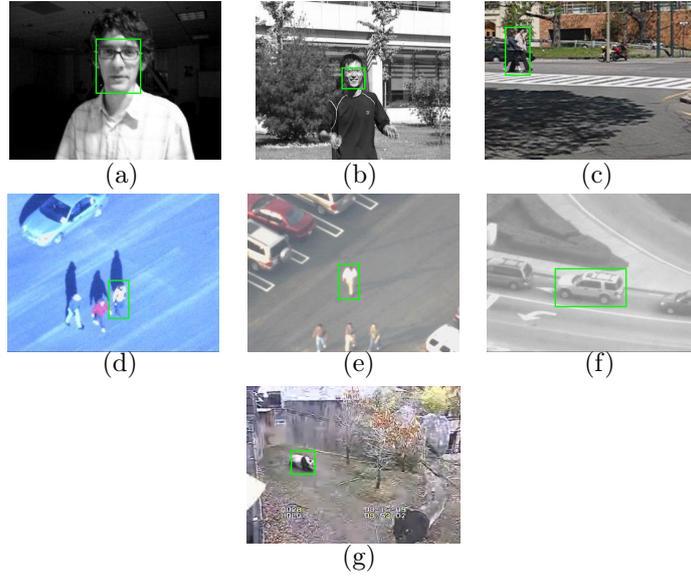


Figura 4.1: Secuencias de prueba: (a) David, (b) Jumping, (c) Pedestrian 1, (d) Pedestrian 2, (e) Pedestrian 3, (f) Car y (g) Panda.

En base al conteo de verdaderos positivos, falsos positivos y falsos negativos, se definen las siguientes métricas:

$$\text{Precision} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Positivos}}, \quad (4.1)$$

$$\text{Recall} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Negativos}}, \quad (4.2)$$

$$\text{F-Measure} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \quad (4.3)$$

Definimos además un puntaje de tracking basado en la proporción de bounding boxes trackeados con intersección relativa mayor a 0,25 con el ground truth en relación a todos los bounding boxes devueltos por TLD-FV:

$$\text{Score} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Positivos} + \text{Falsos Negativos}}. \quad (4.4)$$

4.2. Parámetros

Como se ha explicado a lo largo del Capítulo 3, TLD-FV cuenta con varios parámetros que pueden afectar su comportamiento. Resumiendo, estos son:

Umbral de aprendizaje Este umbral es equivalente al θ_{core} de TLD y maneja cuándo se realiza el aprendizaje del modelo. En TLD-FV, se aprenderá con los parches del cuadro actual si el puntaje de la hipótesis integrada es mayor al umbral de aprendizaje. Valores interesantes para este parámetro se encuentran en el intervalo $(0, 2)$. En particular, si es mayor a 1 solo aprenderemos cuando la hipótesis actual fue clasificada confiablemente. Inversamente, si es menor a 1 aprenderemos aun cuando la hipótesis

se considera “difícil” de clasificar. Esto último puede resultar una buena manera de mantener el modelo actualizado cuando el objeto comienza a cambiar de apariencia.

Cantidad de ejemplos negativos Esta es la cantidad de ejemplos negativos proporcionados por el N-expert que son pasados a la etapa de aprendizaje. Debido a que la cantidad de estos generalmente resulta mayor a la de ejemplos positivos del P-expert, es interesante limitarlos para mantener una proporción similar entre ambos. Por otro lado, si aumentamos este parámetro, el modelo debería aprender a discriminar el objeto entre los posibles falsos positivos de mejor manera.

Número de iteraciones del aprendizaje inicial Este parámetro corresponde directamente al algoritmo de descenso de gradiente. Entra en juego a la hora de regular qué tan discriminativo es el clasificador inicial. Para los casos en que el objeto cambia rápidamente de apariencia en los primeros cuadros, es mejor que sea menos discriminativo para poder seguir detectando y aprendiendo.

Número de iteraciones del aprendizaje online Análogo al anterior pero cuando se realiza el aprendizaje online. Este valor es menos crítico ya que depende además de qué tan seguido realizamos aprendizaje online.

Tasa de aprendizaje Esta es la tasa de aprendizaje del algoritmo de descenso de gradiente estocástico presentado en la Sección 2.2.2. En la implementación actual, se utiliza una tasa constante en todos los pasos del algoritmo.

Umbral de puntaje (SCT) Este es el puntaje de clasificación mínimo que debe tener un parche para ser considerado una detección. Nuevamente son interesantes valores relacionados con el umbral de clasificación, es decir valores entre 0 y 1. Cuanto menor sea, aumentará la cantidad de falsos positivos que devuelve el detector. El valor especial “Min” representa una heurística donde el SCT es asignado el menor puntaje de los parches positivos luego de cada ronda de aprendizaje.

Vale la pena notar que todos estos afectan directamente el aprendizaje. En el caso del umbral de aprendizaje, la cantidad de ejemplos negativos y el SCT, regulan la cantidad de ejemplos que son agregados al modelo. Por otro lado, la cantidad de iteraciones de inicialización y online modifican qué tan rápido cambia el modelo. Efectivamente, estos parámetros son importantes debido a que el modelo aprendido tiene una gran injerencia sobre la precisión de tracking en las secuencias.

Para simplificar el tratamiento de todos estos parámetros, dejamos fijos algunos de ellos. Estos son aquellos que en base a pruebas manuales se ha podido elegir un valor aparentemente óptimo o bien que no afectan en gran manera el resultado. El umbral de aprendizaje se fija en 2, efectivamente haciendo que se aprenda casi en todos los cuadros. La cantidad de ejemplos negativos se mantiene al máximo (es decir, se utilizan todos) ya que esto parece mejorar el desempeño la mayoría de las veces. Por último, los dos parámetros de la cantidad de iteraciones se mantienen iguales entre ellos.

Además, para entrenar el modelo de los descriptores binarios BRIEF, se utilizó un subconjunto del conjunto de entrenamiento de imágenes de PASCAL

VOC 2012 [7], el cual cuenta con 17125 imágenes naturales de diversa índole. Como se dijo anteriormente, este modelo utiliza descriptores BRIEF de 16 bits y una mezcla de 32 distribuciones Bernoulli.

4.3. Resultados

En el Cuadro 4.1 podemos ver los resultados de Score, Precision, Recall y F-Measure para TLD-FV con distintos parámetros. La tasa de aprendizaje se mantuvo en 0,05. La cantidad de iteraciones iniciales y online se configuró en 1, 5 y 20. Al mismo tiempo, el parámetro SCT se configuró como 0, 1 y “Min”.

En la sección izquierda de la tabla se encuentran las mismas métricas para TLD original, a modo de comparación. La implementación de TLD utilizada para obtener estos resultados no es la de los autores originales, sino que es una reimplementación en C++ que utiliza aceleración en GPU. Esta implementación está pensada para uso industrial.

Aquellos valores de F-Measure de TLD-FV que son mayores o se encuentran a una distancia de 0,1 del correspondiente resultado para TLD han sido remarcados en negrita.

TLD					TLD-FV											
					SCT = 0,0 ITERS. = 1				SCT = 0,0 ITERS. = 5				SCT = 0,0 ITERS. = 20			
Secuencia	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M
David	0,83	0,88	1,00	0,90	0,99	0,99	1,00	0,99	0,83	0,83	1,00	0,90	0,98	0,98	1,00	0,99
Jumping	0,88	0,99	1,00	0,99	0,94	0,95	0,99	0,97	0,95	0,96	0,99	0,97	0,92	0,93	0,99	0,96
Pedestrian 1	0,17	0,45	0,50	0,46	0,88	0,88	1,00	0,93	<u>0,14</u>	0,60	0,15	<u>0,25</u>	0,14	0,69	0,15	<u>0,25</u>
Pedestrian 2	0,36	0,97	1,00	0,98	<u>0,27</u>	0,27	1,00	0,42	0,75	0,75	1,00	0,86	<u>0,31</u>	0,31	1,00	0,47
Pedestrian 3	0,88	0,83	0,98	0,90	<u>0,78</u>	0,78	1,00	0,87	<u>0,82</u>	0,82	1,00	0,90	0,99	0,99	1,00	0,99
Car	0,95	0,97	1,00	0,98	<u>0,89</u>	0,89	1,00	0,94	<u>0,90</u>	0,90	0,99	0,95	<u>0,88</u>	0,88	1,00	0,94
Panda	0,19	0,69	0,91	0,78	0,42	0,42	1,00	0,59	0,36	0,36	0,99	0,53	0,84	0,84	0,99	0,91
					SCT = 1,0 ITERS. = 1				SCT = 1,0 ITERS. = 5				SCT = 1,0 ITERS. = 20			
Secuencia	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M
David	0,83	0,88	1,00	0,90	0,99	0,99	1,00	0,99	1,00	1,00	1,00	1,00	0,59	0,69	0,80	0,74
Jumping	0,88	0,99	1,00	0,99	1,00	1,00	1,00	1,00	0,98	0,99	0,97	0,99	0,90	0,95	0,94	0,94
Pedestrian 1	0,17	0,45	0,50	0,46	0,88	0,89	0,99	0,93	<u>0,14</u>	0,69	0,15	0,25	<u>0,14</u>	0,69	0,15	0,25
Pedestrian 2	0,36	0,97	1,00	0,98	0,46	0,46	0,99	0,63	0,78	0,78	1,00	0,88	<u>0,12</u>	0,94	0,12	0,21
Pedestrian 3	0,88	0,83	0,98	0,90	0,27	0,27	0,96	0,42	<u>0,83</u>	0,83	1,00	0,91	0,93	0,93	0,99	0,96
Car	0,95	0,97	1,00	0,98	0,15	0,56	0,17	0,26	<u>0,84</u>	0,85	0,99	0,91	<u>0,87</u>	0,90	0,97	0,93
Panda	0,19	0,69	0,91	0,78	0,49	0,49	0,99	0,66	0,42	0,42	0,99	0,59	0,44	0,44	0,98	0,61
					SCT = Min ITERS. = 1				SCT = Min ITERS. = 5				SCT = Min ITERS. = 20			
Secuencia	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M	Sc.	Pre.	Rec.	F-M
David	0,83	0,88	1,00	0,90	<u>0,80</u>	0,80	0,99	<u>0,89</u>	0,99	0,99	1,00	0,99	0,84	0,99	0,84	0,91
Jumping	0,88	0,99	1,00	0,99	1,00	1,00	1,00	1,00	0,96	0,96	0,99	0,98	0,20	0,95	0,20	0,34
Pedestrian 1	0,17	0,45	0,50	0,46	0,92	0,97	0,94	0,95	0,14	0,69	0,15	0,25	0,14	0,69	0,15	0,25
Pedestrian 2	0,36	0,97	1,00	0,98	0,57	0,57	1,00	0,72	0,79	0,79	0,99	0,88	0,12	0,94	0,12	0,21
Pedestrian 3	0,88	0,83	0,98	0,90	0,96	0,98	0,98	0,98	0,95	0,96	0,98	0,97	0,27	0,27	0,98	0,42
Car	0,95	0,97	1,00	0,98	0,46	0,61	0,66	0,63	0,68	0,71	0,94	0,81	0,60	0,73	0,76	0,75
Panda	0,19	0,69	0,91	0,78	0,42	0,42	0,99	0,60	0,27	0,43	0,43	0,43	0,40	0,40	0,97	0,57

Cuadro 4.1: Comparación entre TLD-FV y TLD original para distintos valores de SCT (arriba hacia abajo) y cantidad de iteraciones (izquierda a derecha). Valores en **negrita** indican mayor o igual al correspondiente valor en TLD, mientras que valores subrayados indican menor pero dentro de 0,1 del correspondiente valor en TLD. Los valores bajo TLD se repiten tres veces para facilitar su comparación.

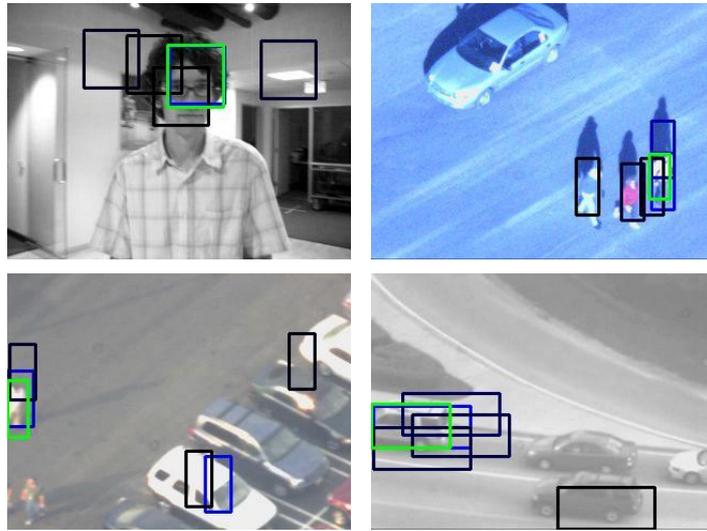


Figura 4.2: Falsos positivos: Ejemplo del funcionamiento de TLD-FV para cuatro secuencias. El recuadro verde es la hipótesis elegida, mientras que los azules son otras hipótesis del detector. Más saturación indica mayor puntaje de detección.

Podemos ver que en algunas secuencias TLD-FV tiene muy buenos resultados, en algunos casos hasta mejores que los de TLD original. Por otro lado, hay secuencias como Pedestrian 1 y 2 cuyo puntaje es en general muy bajo.

Es importante notar que para todas las secuencias hay una configuración de parámetros para la cual TLD-FV se comporta satisfactoriamente. Al mismo tiempo, no hay una configuración que se comporte bien para todas las secuencias. De hecho, en la mayoría de los casos sólo lo hace para alrededor de la mitad. Esto parece evidenciar que el desempeño en las secuencias es muy sensible a los parámetros y por extensión a qué tan rápido cambia el modelo subyacente.

Por otro lado, el parámetro SCT se comporta mejor cuando es 0. Para los casos en que es 1 y “Min”, empeora los resultados de varias secuencias mientras que sólo apenas mejora las que ya tienen buenos resultados. Por lo tanto, en los experimentos siguientes se lo mantiene fijo en 0.

Otra observación interesante sobre estos datos es que en la mayoría de los casos el puntaje de Recall es bueno mientras que el de Precision se comporta peor. Esto indica que el sistema es más propenso a dar falsos positivos que falsos negativos. Efectivamente, si revisamos más de cerca el comportamiento en algunas secuencias, podemos ver cómo a veces aparecen bounding boxes lejanos al objeto (ver Figura 4.2).

En base a estos resultados, se intentó realizar una búsqueda un poco más exhaustiva en el espacio de parámetros, específicamente la cantidad de iteraciones junto a la tasa de aprendizaje. Para esto se eligieron cuatro secuencias, aquellas más sensibles a los cambios en el aprendizaje: David, Pedestrian 1, Pedestrian 2 y Car. En la Figura 4.3 podemos ver graficado el F-Measure a medida que se varía la cantidad de iteraciones entre 1 y 50 y la tasa de aprendizaje entre 0,001 y 0,1.

Se puede ver claramente que diferentes secuencias se comportan mejor en regiones distintas de los parámetros. Notablemente, mientras David y Car se

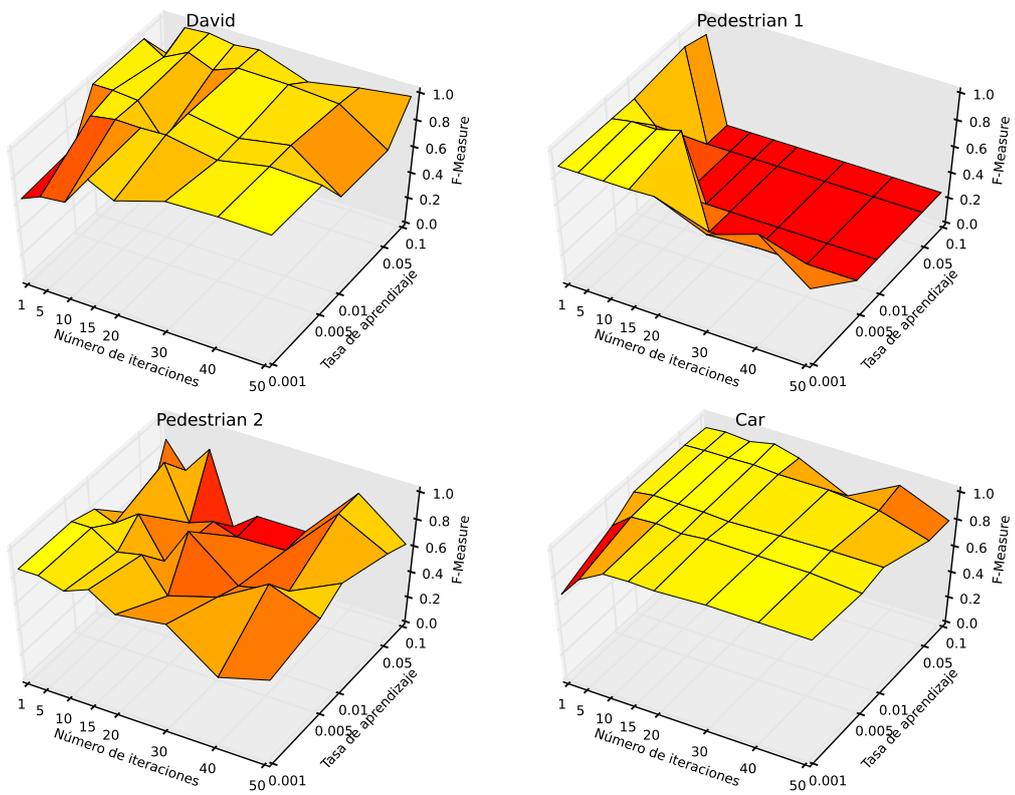


Figura 4.3: Resultados de F-Measure para TLD-FV variando número de iteraciones y tasa de aprendizaje.

Secuencia	TLD				TLD-FV					
	Sc.	Pre.	Rec.	F-M	Tasa = 0,005	Iters. = 5	Sc.	Pre.	Rec.	F-M
David	0,83	0,88	1,00	0,90	0,91	0,91	1,00	0,95		
Jumping	0,88	0,99	1,00	0,99	0,96	0,96	1,00	0,98		
Pedestrian 1	0,17	0,45	0,50	0,46	0,84	0,84	1,00	0,91		
Pedestrian 2	0,36	0,97	1,00	0,98	0,75	0,75	1,00	0,86		
Pedestrian 3	0,88	0,83	0,98	0,90	0,78	0,78	1,00	0,88		
Car	0,95	0,97	1,00	0,98	0,88	0,88	1,00	0,94		
Panda	0,19	0,69	0,91	0,78	0,52	0,52	1,00	0,68		
Promedio	0,60	0,82	0,91	0,85	0,80	0,80	1,00	0,88		

Cuadro 4.2: Resultados de TLD-FV con parámetros ajustados según Figura 4.3

comportan bien para un número de iteraciones y tasa de aprendizaje suficientemente grandes, lo contrario es verdad para Pedestrian 1. Visualmente, los gráficos de las primeras dos secuencias se comportan inversamente al de esta última. Pedestrian 2 muestra más variabilidad pero se asemeja a Pedestrian 1 en que su máximo se encuentra para valores chicos de los parámetros.

Observando estos gráficos además podemos ver que para la configuración de parámetros (Número de iteraciones = 5, Tasa de aprendizaje = 0,005) obtenemos buenos resultados en las cuatro secuencias. Corriendo TLD-FV con estos parámetros en todas las secuencias obtenemos los resultados del Cuadro 4.2. Se destaca que bajo estos parámetros se consigue un puntaje de Recall perfecto, si bien ya dijimos que esta métrica es favorecida por el algoritmo. Aunque esta configuración no llega a comportarse tan bien como TLD original, es un buen compromiso para obtener resultados admisibles en la mayoría de las secuencias.

Capítulo 5

Conclusiones y trabajos a futuro

En este trabajo se analizó un sistema de tracking del estado del arte conocido como TLD. A partir de este análisis, se identificaron puntos débiles en los distintos módulos que lo componen, principalmente los requerimientos de memoria y cómputo en el módulo de detección. Se propuso un nuevo tracker basado en este, llamado TLD-FV.

El tracker TLD-FV mantiene el esquema general de tracking, detección y aprendizaje de TLD original. El módulo de tracking Median Flow es reemplazado por una versión más robusta de este, llamada RANSAC Flock of Trackers. Los cambios más importantes son en el módulo de detección, donde se reemplaza los clasificadores Ensemble y Nearest Neighbour por un clasificador lineal basado en una representación de parches a través de vectores de Fisher.

Estos cambios buscan mantener la eficacia de tracking del sistema comparable a la de TLD pero pudiendo hacer menor uso de los recursos. A su vez, permitieron implementar técnicas de aprendizaje no posibles con las limitaciones de TLD original.

Los resultados experimentales mostraron que si bien TLD-FV puede alcanzar el desempeño de TLD en las secuencias de prueba, no logra hacerlo de manera consistente bajo una sola configuración de parámetros. Analizando estos resultados se determinó que las secuencias de prueba son muy sensibles a los parámetros del modelo de aprendizaje. Se pudo determinar una configuración de parámetros que da resultados aceptables en la mayoría de las secuencias.

Por otro lado, se determinó que en comparación a TLD, el módulo de detección es más susceptible a devolver falsos positivos. Esto se puede deber a las características inherentes del clasificador y representaciones subyacentes. Estos falsos positivos afectan de manera concreta a los resultados de las evaluaciones.

Salvando estas desventajas, el uso de representaciones basadas en vectores de Fisher parecer ser un paso correcto en construir sistemas de tracking que demanden menos recursos. Cabe destacar que la implementación prototipo de TLD-FV es relativamente simple y por lo tanto es fácil explorar nuevos esquemas.

Trabajos futuros

- Experimentar con el clasificador lineal y la forma de representación para que el detector devuelva una menor cantidad de falsos positivos. Existen diversos algoritmos de aprendizaje de clasificadores lineales. A su vez, se pueden probar otros tipos de descriptores y modelos subyacentes a los vectores de Fisher.
- Fallando lo anterior, encontrar nuevas maneras de tratar con estos falsos positivos. Los módulos de integración y P-N Learning de TLD, si bien generales en la teoría, fueron construidos pensando en un tipo de clasificador diferente a los lineales. Quizá haya mejores maneras de integrar y aprender las detecciones del clasificador lineal.
- Implementar una versión optimizada, posiblemente utilizando cómputos en GPU. La versión prototipo, si bien pensada para utilizar menos recursos desde el punto de vista teórico, prácticamente no se encuentra optimizada. Ya hay implementaciones de TLD sobre GPUs y sería interesante poder comparar la eficiencia en uso de recursos de ambas.
- Incorporar la capacidad de detectar en varias escalas y de detectar rotaciones. El detector TLD original es capaz de realizar lo primero, no así lo segundo. Estas capacidades no fueron incorporadas a TLD-FV por motivos tanto de tiempo como de prioridad. Agregarlas no debería representar mucha dificultad teórica y podría mejorar el desempeño del sistema. En lo breve que se experimentó con multi-escala, el clasificador parece preferir parches más chicos.

Bibliografía

- [1] *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007. ISBN: 1-4244-1179-3. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4269955>.
- [2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.
- [3] Bernhard E. Boser, Isabelle Guyon y Vladimir Vapnik. «A Training Algorithm for Optimal Margin Classifiers». En: *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*. Ed. por David Haussler. ACM, 1992, págs. 144-152. ISBN: 0-89791-497-X. DOI: 10.1145/130385.130401. URL: <http://doi.acm.org/10.1145/130385.130401>.
- [4] Léon Bottou. «Online Algorithms and Stochastic Approximations». En: *Online Learning and Neural Networks*. Ed. por David Saad. revised, oct 2012. Cambridge, UK: Cambridge University Press, 1998. URL: <http://leon.bottou.org/papers/bottou-98x>.
- [5] Michael Calonder y col. «BRIEF: Binary Robust Independent Elementary Features». En: *Computer Vision - ECCV 2010, 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV*. Ed. por Kostas Daniilidis, Petros Maragos y Nikos Paragios. Vol. 6314. Lecture Notes in Computer Science. Springer, 2010, págs. 778-792. ISBN: 978-3-642-15560-4. DOI: 10.1007/978-3-642-15561-1_56. URL: http://dx.doi.org/10.1007/978-3-642-15561-1_56.
- [6] Kostas Daniilidis, Petros Maragos y Nikos Paragios, eds. *Computer Vision - ECCV 2010, 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV*. Vol. 6314. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-15560-4. DOI: 10.1007/978-3-642-15561-1. URL: <http://dx.doi.org/10.1007/978-3-642-15561-1>.
- [7] M. Everingham y col. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. <http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html>.
- [8] Pedro F. Felzenszwalb y col. «Object Detection with Discriminatively Trained Part-Based Models». En: *IEEE Trans. Pattern Anal. Mach. Intell.* 32.9 (2010), págs. 1627-1645. DOI: 10.1109/TPAMI.2009.167. URL: <http://doi.ieeecomputersociety.org/10.1109/TPAMI.2009.167>.

- [9] Martin A. Fischler y Robert C. Bolles. «Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography». En: *Commun. ACM* 24.6 (1981), págs. 381-395. DOI: 10.1145/358669.358692. URL: <http://doi.acm.org/10.1145/358669.358692>.
- [10] Zdenek Kalal, Jiri Matas y Krystian Mikolajczyk. «P-N learning: Bootstrapping binary classifiers by structural constraints». En: *The Twenty-Third IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2010, San Francisco, CA, USA, 13-18 June 2010*. IEEE Computer Society, 2010, págs. 49-56. ISBN: 978-1-4244-6984-0. DOI: 10.1109/CVPR.2010.5540231. URL: <http://dx.doi.org/10.1109/CVPR.2010.5540231>.
- [11] Zdenek Kalal, Krystian Mikolajczyk y Jiri Matas. «Forward-Backward Error: Automatic Detection of Tracking Failures». En: *20th International Conference on Pattern Recognition, ICPR 2010, Istanbul, Turkey, 23-26 August 2010*. IEEE Computer Society, 2010, págs. 2756-2759. ISBN: 978-0-7695-4109-9. DOI: 10.1109/ICPR.2010.675. URL: <http://dx.doi.org/10.1109/ICPR.2010.675>.
- [12] Zdenek Kalal, Krystian Mikolajczyk y Jiri Matas. «Tracking-Learning-Detection». En: *IEEE Trans. Pattern Anal. Mach. Intell.* 34.7 (2012), págs. 1409-1422. DOI: 10.1109/TPAMI.2011.239. URL: <http://doi.ieeecomputersociety.org/10.1109/TPAMI.2011.239>.
- [13] Bruce D. Lucas y Takeo Kanade. «An Iterative Image Registration Technique with an Application to Stereo Vision». En: *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81), Vancouver, BC, Canada, August 1981*. Ed. por Patrick J. Hayes. William Kaufmann, 1981, págs. 674-679.
- [14] Georg Nebehay y Roman P.flugfelder. «Consensus-based matching and tracking of keypoints for object tracking». En: *IEEE Winter Conference on Applications of Computer Vision, Steamboat Springs, CO, USA, March 24-26, 2014*. IEEE Computer Society, 2014, págs. 862-869. ISBN: 978-1-4799-4984-7. DOI: 10.1109/WACV.2014.6836013. URL: <http://dx.doi.org/10.1109/WACV.2014.6836013>.
- [15] Mustafa Özuysal, Pascal Fua y Vincent Lepetit. «Fast Keypoint Recognition in Ten Lines of Code». En: *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007. ISBN: 1-4244-1179-3. DOI: 10.1109/CVPR.2007.383123. URL: <http://dx.doi.org/10.1109/CVPR.2007.383123>.
- [16] Mustafa Özuysal y col. «Fast Keypoint Recognition Using Random Ferns». En: *IEEE Trans. Pattern Anal. Mach. Intell.* 32.3 (2010), págs. 448-461. DOI: 10.1109/TPAMI.2009.23. URL: <http://doi.ieeecomputersociety.org/10.1109/TPAMI.2009.23>.
- [17] Florent Perronnin y Christopher R. Dance. «Fisher Kernels on Visual Vocabularies for Image Categorization». En: *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society, 2007. ISBN: 1-4244-1179-3. DOI: 10.1109/CVPR.2007.383266. URL: <http://dx.doi.org/10.1109/CVPR.2007.383266>.

- [18] Florent Perronnin, Jorge Sánchez y Thomas Mensink. «Improving the Fisher Kernel for Large-Scale Image Classification». En: *Computer Vision - ECCV 2010, 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV*. Ed. por Kostas Daniilidis, Petros Maragos y Nikos Paragios. Vol. 6314. Lecture Notes in Computer Science. Springer, 2010, págs. 143-156. ISBN: 978-3-642-15560-4. DOI: 10.1007/978-3-642-15561-1_11. URL: http://dx.doi.org/10.1007/978-3-642-15561-1_11.
- [19] Jorge Sánchez y Javier Redolfi. «Exponential family Fisher vector for image classification». En: *Pattern Recognition Letters* 59 (2015), págs. 26-32. DOI: 10.1016/j.patrec.2015.03.010. URL: <http://dx.doi.org/10.1016/j.patrec.2015.03.010>.
- [20] Ernesto Tapia. «A note on the computation of high-dimensional integral images». En: *Pattern Recognition Letters* 32.2 (2011), págs. 197-201. DOI: 10.1016/j.patrec.2010.10.007. URL: <http://dx.doi.org/10.1016/j.patrec.2010.10.007>.
- [21] Tomas Vojir y Jiri Matas. «The Enhanced Flock of Trackers». En: *Registration and Recognition in Images and Videos*. Ed. por Roberto Cipolla, Sebastiano Battiato y Giovanni Maria Farinella. Vol. 532. Studies in Computational Intelligence. Springer Berlin Heidelberg: Springer Berlin Heidelberg, ene. de 2014, págs. 113-136. ISBN: 978-3-642-44906-2. DOI: 10.1007/978-3-642-44907-9_6. URL: http://dx.doi.org/10.1007/978-3-642-44907-9_6.