



ugr

Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Detección de trayectorias anómalas usando *tracking* de personas basado en *deep learning*

Autor

Laura Hernández Muñoz

Directores

Francisco Herrera Triguero
Siham Tabik



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

—

Granada, julio de 2020

Detección de trayectorias anómalas usando *tracking* de personas basado en *deep learning*

Laura Hernández Muñoz

Palabras clave: aprendizaje automático, aprendizaje profundo, clustering, tracking, anomalías

Resumen

El aprendizaje automático es un área de la inteligencia artificial que ha experimentado un auge importante durante los últimos años, en especial, la rama del aprendizaje profundo o *deep learning*. Este desarrollo se debe al aumento de la capacidad de cómputo de los ordenadores, así como a la gran cantidad de datos disponibles gracias a Internet, que permite que se utilicen modelos complejos que hace años eran prácticamente intratables. Estos modelos se usan en muchísimas tareas como el reconocimiento de objetos en imágenes o el procesamiento del lenguaje natural, demostrando en algunas de ellas mayor destreza que los humanos.

Uno de los campos de estudio más tratados en los últimos dos o tres años es la detección de anomalías, sea cual sea el área de aplicación. En concreto, podemos encontrar numerosas propuestas en la detección de anomalías en el comportamiento humano, es decir, individuos que presentan una conducta extraña con respecto a lo que se considera como “normal” tanto en lo relativo a su desplazamiento como a acciones específicas.

En este trabajo nos acercamos a esta rama del descubrimiento de anomalías para intentar detectar individuos cuya trayectoria se desvía del comportamiento normal en términos de velocidad y/o dirección, empleando para ello métodos de aprendizaje automático y profundo. Un sistema de detección de este tipo podría ser de gran utilidad en zonas con videovigilancia como, por ejemplo, un aeropuerto, ya que se podría ver si alguien se desvía de las rutas habituales y entra en una zona restringida. También podríamos ver, por ejemplo, si en una zona peatonal alguien va en algún vehículo, como una bicicleta, ya que su velocidad será superior a la del resto de individuos.

Durante la fase de investigación vimos que hay sistemas de este tipo basados tanto en aprendizaje supervisado como en no supervisado. En el caso de las técnicas de aprendizaje supervisado, el principal inconveniente es que es necesario contar con datos etiquetados para que el modelo aprenda. Por eso, si no se tienen datos suficientes, el algoritmo en cuestión puede fallar, clasificando un comportamiento normal como anómalo, porque nunca antes lo ha visto, o clasificando uno anómalo como normal, porque no ha visto suficientes ejemplos extraños. El aprendizaje no supervisado, por otro lado, no presenta esta limitación, ya que no necesita entrenar para aprender a distinguir unos comportamientos de otros, sino que extrae las características que los diferencian de forma automática.

Uno de los pocos trabajos que encontramos en la literatura con este enfoque no supervisado es el descrito en [24], donde los autores proponen la detección de trayectorias anormales mediante *clustering* a partir de un conjunto de trayectorias dadas y, posteriormente, realizar análisis a nivel de píxel para refinar las detecciones y reconocer comportamientos extraños más allá de la dirección y la velocidad. Sin embargo, en este modelo se usan métodos de *tracking* clásicos basados en extracción de características [25, 26], lo cual implica (en mayor o menor medida) una intervención manual. Por ello, tomando este modelo como base, proponemos una implementación propia en la que el conjunto de trayectorias se extrae usando un método mucho más actual y preciso, donde tanto la detección de personas como el propio *tracking* se basan en *deep learning*. La implementación de este método es pública y está disponible en GitHub.

En este trabajo pretendemos poner de manifiesto la importancia de un buen *tracking* de cara a la detección de las anomalías, ya que si los datos que obtenemos en este paso previo no son buenos, la detección tampoco lo es. Con esta idea en mente, analizamos los resultados obtenidos con el método basado en *deep learning* que hemos comentado y los comparamos con los de un algoritmo mucho menos potente, viendo que, efectivamente, el rendimiento del algoritmo de detección de anomalías se ve fuertemente afectado por la calidad del *tracking*.

Abnormal trajectory detection using deep learning-based people tracking

Laura Hernández Muñoz

Keywords: machine learning, deep learning, clustering, tracking, anomalies

Abstract

Machine learning is an area of AI that has grown significantly during last years, especially a branch called deep learning. This development is due to the increase in the computing capacity of computers, as well as the large amount of data available via Internet, which makes possible to use complex models that could not be run years ago as they were practically untreatable. These models are used in many tasks, such as object recognition in images or natural language processing and, in some of them, they show better skill than humans.

One of the fields with more research in the last two or three years is anomaly detection, whatever the area of application. Specifically, we can find many proposals in abnormal human behavior detection, i.e. individuals who show strange behavior with respect to what is considered as “normal”, both in terms of their motion and specific actions.

In this work, we make our approach to this field inside anomaly detection and try to detect those people whose trajectory goes away from normal behavior in terms of speed and/or direction, using machine and deep learning methods for that purpose. Such a detection system could be very useful in places with video surveillance, such as an airport, since it could detect if someone deviates from the usual path and goes to a restricted area. We could also detect, for instance, if someone uses a vehicle, such as a bicycle, in a pedestrian zone, since his speed will, probably, be higher than other people's.

During research phase, we saw that there are systems like this based on both supervised and unsupervised learning. Focusing on supervised learning techniques, the main drawback is that it is necessary to have labeled data so the model can learn from it. So, if we do not have enough data, the algorithm may fail, classifying normal behavior as abnormal because it has never seen that behavior before, or classifying abnormal behavior as normal because it has not processed enough strange instances. Unsupervised learning, on the other hand, does not present this limitation, since it is not necessary a training step where it learns to distinguish some behaviors from others, but rather extracts that differentiate them automatically.

One of the few works we can found in the literature with this unsupervised approach is the one described in [24], where the authors propose to detect abnormal trajectories by applying clustering to a set of given trajectories and, then, use pixel-based analysis to refine detections and recognize unusual behaviors beyond directions and speed. However, in this model, they use classic tracking methods based on feature extraction [25, 26], which implies (to a greater or lesser extent) manual intervention. Therefore, taking this model as a basis,

we propose an own implementation in which the set of trajectories is extracted using a much more modern and precise method, where both the detection of people and the tracking itself are based on deep learning. The implementation of this method is publicly available on GitHub.

With this work we try to highlight how important is a good tracking in order to detect human anomalies because, if data obtained in this previous step is not good enough, detection is not good either. With this in mind, we analyze the results obtained with the deep learning-based method that we have discussed and compare them with those of a much less powerful algorithm, seeing that, indeed, the performance of the anomaly detection algorithm is strongly affected by tracking quality.

Yo, **Laura Hernández Muñoz**, alumna del Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones de la Universidad de Granada**, con DNI 26502248S, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.



Fdo: Laura Hernández Muñoz

Granada, 7 de julio de 2020

D. **Francisco Herrera Triguero** y D.^a **Siham Tabik**, profesores del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado **Detección de trayectorias anómalas usando tracking de personas basado en deep learning**, ha sido realizado bajo su supervisión por **Laura Hernández Muñoz**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 7 de julio de 2020.

Los directores:



Francisco Herrera Triguero



Siham Tabik

Agradecimientos

En primer lugar, dar las gracias a mis padres, que me han apoyado en todo momento, sobre todo en los momentos malos, y a mis abuelos, con los que he convivido durante estos cuatro años de universidad y han sufrido mis llantos y compartido mis alegrías. También dar las gracias a Alberto, a quien conocí en primero y del que no me he separado desde entonces; gracias por obligarme a descansar y por aguantarme y ayudarme cuando ya no podía más.

Quiero agradecer también a mi compañera y amiga María Jesús, MJ, que conocí el primer día que entré en la facultad y con quien he trabajado, reído y llorado durante estos cuatro años, y a Dani, con el que he compartido largas charlas quejándonos de las asignaturas y los profesores, que también acaban la carrera este año.

Mencionar también a Jesús, Carlos y Fran, con los que he forjado una gran amistad y considero como parte de la familia y con los que me dejado las manos jugando al futbolín los fines de semana.

Agradezco también a los profesores que me han dado clase durante estos cuatro años, ya que, aunque algunos me han gustado más y otros menos, todos me han enseñado algo y me han ayudado a crecer, tanto académica como personalmente, y a llegar a donde estoy ahora. En especial, dar las gracias a mis tutores, Francisco y Siham, y a Paco, que siempre ha estado disponible para resolver cualquier duda que se me presentara y ayudar en lo que hiciera falta.

Introducción	12
Fundamentación teórica	14
2.1. Aprendizaje automático o machine learning	14
2.1.1. Algoritmos de aprendizaje	14
Las tareas	15
La medida de efectividad	16
La experiencia	16
2.1.2. Generalización, overfitting y underfitting	17
2.2. Clustering	18
2.2.1. Algoritmo de propagación de afinidad	19
2.3. Redes neuronales	22
2.3.1. Introducción	22
2.3.2. Estructura de las redes neuronales	23
La neurona	24
Arquitectura y capas de una red neuronal	26
2.4. Redes neuronales convolucionales (CNN)	28
2.4.1. Operación de convolución	28
Características de la convolución	31
2.4.2. Capas en una red neuronal convolucional	32
Capa convolucional	32
Capa de activación	33
Capa de pooling	33
Capa completamente conectada	34
2.4.3. Estructura de una CNN	34
2.5. Redes neuronales residuales (ResNet)	35
2.6. Redes neuronales convolucionales basadas en regiones (R-CNN)	37
2.6.1. Búsqueda selectiva	38
2.6.2. R-CNN rápida (Fast R-CNN)	39
RoI pooling	40
Función softmax	40
2.6.3. R-CNN más rápida (Faster R-CNN)	41
Region Proposal Network (RPN)	42
2.7. Feature Pyramid Networks (FPN)	44
2.8. Redes neuronales siamesas (SNN)	47
3. Metodología	49
3.1. Conjunto de datos: UCSD Anomaly Detection Dataset	49
3.2. Algoritmo de tracking: Tracking without bells and whistles	51
3.2.1. Funcionamiento del algoritmo	52
3.3. Método base de detección de trayectorias anómalas: Towards Abnormal Trajectory and Event Detection in Video Surveillance	55
3.3.1. Funcionamiento del algoritmo	55

Tracking individual y grupal	56
Análisis de trayectorias	57
Análisis a nivel de píxel	60
3.4. Algoritmo desarrollado para la detección de anomalías basada en trayectorias	60
3.4.1. Explicación de las funciones implementadas	69
3.5. Análisis y elección de factores multiplicadores para el intervalo de confianza de velocidad	75
3.6. Resultados experimentales	79
3.6.1. Comparación de resultados	80
4. Conclusiones y trabajo futuro	82
4.1. Conclusiones	82
4.2. Trabajo futuro	82
Índice de figuras	84
Sección 2	84
Sección 3	84
Bibliografía	86

1. Introducción

El campo de la inteligencia artificial se ha desarrollado mucho desde hace algunos años, aunque no es algo nuevo, pues esta disciplina nació hace ya más de medio siglo para intentar duplicar las facultades humanas. Sin embargo, ha tenido etapas oscuras debido a las dificultades derivadas de la escasa capacidad de cómputo y memoria de los equipos más antiguos, así como de la escasez de datos. Precisamente, gracias al aumento de la memoria y la capacidad de cálculo que los ordenadores han experimentado desde hace algunas décadas y la gran cantidad de información con la que contamos en la actualidad, debida a la expansión de Internet, la inteligencia artificial ha recuperado la atención que perdió hace tiempo y ahora es una de las ramas de la informática sobre las que más se estudia.

Dentro de la inteligencia artificial, el área que más ha crecido estos años ha sido el aprendizaje automático o *machine learning* y, en especial, la rama del *deep learning*. El *machine learning* busca que las máquinas aprendan a partir de datos de manera autónoma y apliquen ese conocimiento para predecir comportamientos futuros o identificar patrones. El *deep learning* se centra en este objetivo usando modelos más complejos, para conseguir un aprendizaje más profundo y avanzado.

Una gran cantidad de estudiosos se ha acercado a la rama del *deep learning* últimamente y ha desarrollado potentes modelos y algoritmos capaces de realizar tareas de distinta índole, demostrando en muchas de ellas una mayor eficacia que los seres humanos. Entre dichas tareas encontramos, por ejemplo, la clasificación de objetos, la detección de anomalías o la predicción de datos; aunque la lista es interminable, ya que el *deep learning* se aplica hoy en día en muchísimas áreas.

Este trabajo se centra en la tarea de detección de anomalías en vídeo a través del uso del *deep learning*. En concreto, se trata de detectar personas cuya velocidad o dirección es anómala con respecto al comportamiento general. Dicho comportamiento general, que podemos denominar “normal”, puede aprenderse a partir de la observación, ya que, viendo cómo se mueve la gente por una zona, es posible extraer características comunes o patrones que se repitan en la mayoría.

Para resolver esta tarea se ha usado un algoritmo para detectar y hacer el seguimiento de las personas que aparecen en un vídeo, es decir, el *tracking*, que se vale de redes neuronales para ello y, posteriormente, con los resultados obtenidos por este algoritmo, se han extraído distintas zonas con comportamientos similares mediante *clustering*, para poder detectar así las trayectorias anómalas en comparación a lo normal en dichas zonas.

La estructura del trabajo es la siguiente. En el apartado 2 hacemos un estudio teórico del aprendizaje automático, haciendo hincapié en redes neuronales y agrupamiento. En la primera sección introducimos el problema del aprendizaje, así como algunos conceptos relacionados con este. A continuación, explicamos los fundamentos del *clustering* y entramos en detalle en el algoritmo de propagación de afinidad. La tercera sección sirve de introducción a las redes neuronales y abre el camino a las secciones siguientes, las cuales abordan modelos surgidos a partir de las redes clásicas. En concreto, primero comentamos las redes neuronales convolucionales, viendo en qué consiste la operación de convolución, que está presente en las capas principales de estas redes, y describiendo sus capas y estructura. En la quinta sección vemos los fundamentos básicos de las redes neuronales residuales y en la sexta estudiamos distintos tipos de redes neuronales basadas en regiones, concretamente, R-CNN, Fast R-CNN y Faster R-CNN. Finalmente, en las dos últimas secciones comentamos la extracción de características con *Feature Pyramid Networks* y las redes neuronales siamesas.

El capítulo 3 incluye la parte práctica del trabajo. En esta sección encontramos varios subapartados: en el primero de ellos describimos el conjunto de datos con el que hemos realizado la experimentación, que hemos generado a partir de uno ya existente. En la segunda subsección explicamos el algoritmo que utilizamos para realizar el seguimiento de individuos, cuyo código fuente hemos obtenido del repositorio de GitHub de sus creadores. En cuanto a la tercera, comentamos el artículo en el cual nos hemos basado para desarrollar nuestro algoritmo de detección de anomalías en trayectorias y en la cuarta, describimos la implementación propia. A continuación, mostramos el estudio realizado para seleccionar determinados factores multiplicadores en la implementación del algoritmo, así como los resultados obtenidos al aplicar nuestro algoritmo sobre conjuntos de trayectorias extraídos con dos métodos de *tracking* diferentes. Podemos comprobar que esta es una parte muy importante a la hora de detectar comportamientos raros en la trayectoria de una persona, ya que con el algoritmo más sencillo se consiguen peores resultados, debido a que los cuadros delimitadores no son demasiado buenos.

Por último, en la sección 4, comentamos las conclusiones obtenidas tras la realización del trabajo y proponemos una serie de líneas de investigación para ampliar y mejorar el modelo desarrollado.

2. Fundamentación teórica

En este apartado nos centraremos en los fundamentos teóricos del trabajo. En concreto, veremos cómo funcionan las redes neuronales, de manera general y, específicamente, las redes neuronales convolucionales, las basadas en regiones (tanto la implementación original como las versiones más rápidas que se han ido desarrollando), las residuales y las siamesas, ya que estas son las que se usan en el algoritmo de *tracking*. Asimismo, también se explicará cómo trabajan los algoritmos de *clustering* o agrupamiento, entrando en detalle en el algoritmo de propagación de afinidad, que es el que se ha usado en la experimentación.

En primer lugar, veremos qué es el *machine learning*, citado anteriormente, y algunos conceptos importantes relacionados con él. A continuación, veremos los fundamentos del *clustering* y entraremos en más detalle en el algoritmo que se ha mencionado antes. Por último, nos centraremos en las redes neuronales, dando en primer lugar una descripción más general y clásica y, a continuación, particularizando en las redes neuronales convolucionales y otros modelos derivados.

2.1. Aprendizaje automático o *machine learning*

Como comentábamos al principio, el *deep learning* es una de las ramas del *machine learning*, por lo que, en primer lugar, antes de entrar en detalle con las redes neuronales y otros modelos, vamos a ver qué es el aprendizaje automático y algunos conceptos que van asociados a este.

El aprendizaje automático da a los ordenadores la capacidad de reconocer patrones y obtener conocimiento a partir de una gran cantidad de datos para que sean capaces de hacer predicciones y realizar tareas sin ser programados específicamente para ellas. A continuación, veremos con más detalle lo que es un algoritmo de aprendizaje automático.

2.1.1. Algoritmos de aprendizaje

Podemos definir un algoritmo de aprendizaje como un algoritmo que, a partir de un conjunto de datos, tiene capacidad para aprender a realizar una tarea determinada. Este aprendizaje está determinado, según el libro *Machine Learning*, de Thomas Mitchell, [1] por la tarea, la experiencia con respecto a esa tarea y una medida de efectividad. Según su definición “se dice que un programa informático aprende de la experiencia E con respecto a una clase de tareas T , con una medida de efectividad P , si su efectividad en las tareas T , medida con P , mejora con la experiencia E ”. Partiendo de esta definición, en los apartados siguientes veremos distintas tareas, medidas de efectividad y experiencias que pueden usarse para plantear algoritmos de este tipo.

Las tareas

El aprendizaje automático intenta abordar tareas que resultan difíciles para el ser humano o que pueden ser resueltas de forma más eficiente por una máquina. El aprendizaje es el medio para adquirir la habilidad o capacidad necesaria para realizar una tarea, por tanto, el proceso de aprendizaje consiste en especificar cómo debe comportarse el programa para ser capaz de realizar dicha tarea. Normalmente, las tareas de aprendizaje automático se describen de acuerdo a cómo tiene que comportarse el sistema frente a un ejemplo determinado, entendiendo un ejemplo como una colección de características medidas de un objeto o evento, que queremos que el sistema procese. Por tanto, para representar un ejemplo se suele usar un vector $x \in \mathbb{R}^n$, donde cada elemento x_i es una característica.

Hay muchas tareas que se pueden resolver haciendo uso del *machine learning*, entre las cuales destacamos las siguientes:

- Clasificación. En esta tarea, el programa debe decidir a qué categoría, de entre k distintas conocidas por él, pertenece una entrada concreta. Para ello, el algoritmo de aprendizaje debe aprender una función que asocie una determinada entrada a una de esas categorías, $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$. De acuerdo con esta función, la salida $y = f(x)$ indica que el modelo asigna la categoría y a la entrada x . Hay otras variantes de esta tarea, donde la entrada o la salida es múltiple o la función f devuelve una distribución de probabilidad, en lugar de una clase concreta.
- Regresión. Para esta tarea, el programa debe predecir un valor numérico continuo dada una entrada, es decir, es similar a la clasificación, pero la función que aprende el programa da como salida un número en lugar de una clase. Por tanto, la función anterior se convierte en $f: \mathbb{R}^n \rightarrow \mathbb{R}$.
- *Clustering* o agrupamiento. En esta tarea, el modelo debe agrupar en distintas categorías los ejemplos de entrada, pero, en vez de conocer las categorías de antemano, como ocurría en la clasificación, el modelo debe inferir dichas clases de forma autónoma a partir del conjunto de datos.

En este trabajo hacemos uso del *clustering* para extraer distintas zonas donde la gente se comporta de una manera determinada, que etiquetamos como normal, y detectar, comparando los ejemplos nuevos con estas zonas, quién tiene un comportamiento inusual.

- Detección de anomalías. En esta tarea, el programa observa un conjunto de eventos u objetos y debe detectar aquellos que sean atípicos o inusuales con respecto al resto. Esta es la tarea en la que se centra este trabajo, localizando personas que tienen una trayectoria anómala con respecto al comportamiento normal.

La medida de efectividad

Para evaluar el comportamiento de un algoritmo de aprendizaje automático es necesario diseñar una medida de eficacia, la cual, normalmente, depende de la tarea que realice el sistema. Por ejemplo, en clasificación se suele medir la precisión (*accuracy*) del modelo, que es la proporción de ejemplos para los que el modelo predice la clase correctamente. En este trabajo usamos una serie de métricas que se especifican en el apartado de resultados experimentales.

Por norma general, lo que nos interesa es ver cómo se comporta el algoritmo que hemos diseñado en datos nuevos, que no ha visto durante la etapa de entrenamiento, para ver su capacidad de generalizar. Para ello, el conjunto de datos disponible se suele dividir en dos subconjuntos, uno con datos de entrenamiento y otro con datos de test, que es el que nos permite conocer la efectividad del modelo en datos no vistos anteriormente. En secciones posteriores explicaremos con más detalle cómo se han generado estos subconjuntos de datos para el trabajo.

La experiencia

Dependiendo del tipo de información que un algoritmo de aprendizaje automático tenga disponible durante el entrenamiento (experiencia), podemos distinguir entre aprendizaje supervisado y no supervisado.

- Aprendizaje supervisado. En este caso, la información o los ejemplos de entrada están formados por un conjunto de características y tienen asociada, además, una etiqueta con el resultado correcto de clasificación. De esta forma, el algoritmo de aprendizaje estudia las distintas características junto con las etiquetas y puede aprender qué características determinan una u otra categoría para ser capaz de predecir a qué clase pertenece un ejemplo nuevo. Dentro del aprendizaje supervisado encontramos las tareas de clasificación y regresión, comentadas anteriormente.
- Aprendizaje no supervisado. En estos modelos, el conjunto de ejemplos contiene sus características, pero no tienen asociada ninguna etiqueta o clase objetivo que haya que predecir. En vez de predecir esa clase, los algoritmos de aprendizaje no supervisado buscan extraer y aprender propiedades de la estructura del conjunto de datos. Dentro de esta categoría encontramos el *clustering* y la detección de anomalías; el caso del problema que tratamos en este trabajo.

De forma general, en el aprendizaje no supervisado se observa un conjunto de ejemplos para aprender propiedades o características de dichos ejemplos, mientras que en el aprendizaje supervisado, se observa ese conjunto de datos y su etiqueta correspondiente para aprender a predecir cada etiqueta a partir de las características.

2.1.2. Generalización, *overfitting* y *underfitting*

El desafío principal en el aprendizaje automático es que el algoritmo desarrollado debe funcionar correctamente en entradas nuevas, que no haya visto anteriormente, no sólamente en los ejemplos con los que fue entrenado. Esta capacidad de predecir correctamente las entradas del conjunto de test es lo que se denomina capacidad de generalización.

Normalmente, al entrenar un modelo tenemos un conjunto dedicado a ello, etiquetado, por lo que podemos medir el error que se comete sobre dicho conjunto y tratar de minimizarlo. Por tanto, nos enfrentamos a un problema de optimización donde la función a optimizar es dicho error y lo que buscamos es minimizarlo. Aparte de minimizar este error, como hemos comentado en el párrafo anterior, también buscamos que el error sobre el conjunto de test sea mínimo.

Para mejorar el comportamiento del modelo en datos no vistos antes debemos asumir que el conjunto de datos con el que estamos trabajando está formado por ejemplos independientes y que el conjunto de test y de entrenamiento están distribuidos de la misma forma. Es decir, que se trabaja con variables aleatorias independientes e idénticamente distribuidas. Teniendo esto en cuenta, se puede estudiar la relación entre el error del modelo en los ejemplos de entrenamiento y el error en los ejemplos de test, que, dadas estas asunciones, podemos esperar que sea el mismo.

Sin embargo, en una aplicación real, el conjunto de datos no se obtiene tras fijar una distribución de probabilidad concreta, sino que contamos con un conjunto de entrenamiento y buscamos un modelo que minimice el error cometido en dicho conjunto y, después, comprobamos si el error en los datos de test es el mismo. De esta forma, el error cometido en test será igual o mayor al cometido en entrenamiento, lo que da lugar a dos evaluaciones de la capacidad de predicción de un algoritmo. Por una parte, la capacidad de minimizar el error de entrenamiento y, por otra, la capacidad de reducir la diferencia entre el error en entrenamiento y en test.

Estos dos factores de la evaluación casan con dos problemas comunes durante el diseño de un algoritmo de aprendizaje:

- *Underfitting*. Este déficit de ajuste se produce cuando nuestro algoritmo tiene un error elevado en los ejemplos de entrenamiento y ocurre cuando el modelo es demasiado simple.
- *Overfitting*. Este sobreajuste ocurre cuando la diferencia entre el error de entrenamiento y test es muy grande y se produce cuando el modelo aprende “demasiado bien” los ejemplos de entrenamiento.

Tanto en un caso como en otro, la generalización del modelo no será buena. En el caso del underfitting, la frontera de decisión aprendida es demasiado simple, por lo que, si

el modelo no puede predecir correctamente los datos observados, probablemente tampoco actuará bien con los datos nuevos. Por otro lado, cuando el modelo está sobreajustado, esta frontera de decisión es muy compleja y, ante datos con características distintas a las aprendidas, no reacciona correctamente.

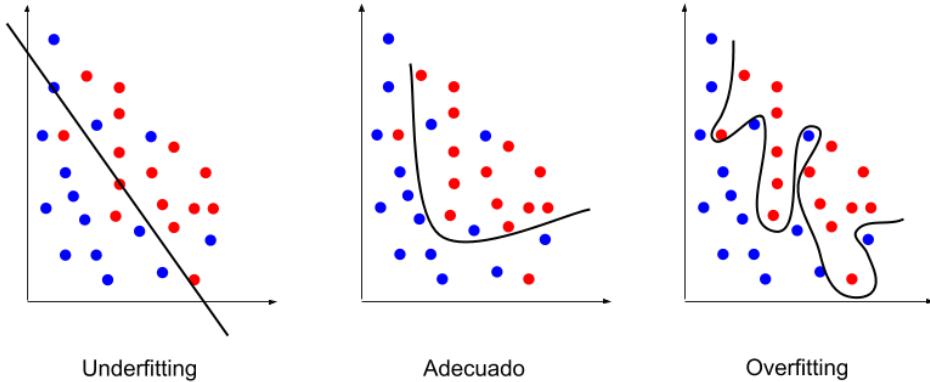


Figura 2.1. (Izda.) Underfitting. (Centro) Modelo adecuado. (Dcha.) Overfitting

2.2. *Clustering*

En algunos problemas, el objetivo es agrupar un conjunto de instancias para crear distintos grupos dentro de los cuales haya características similares. Cada uno de esos grupos diferentes se denomina *cluster* y los objetos que hay en cada uno son similares entre ellos, a la vez que son distintos a los objetos de otros clusters. El *clustering*, por tanto, consiste en segmentar una población o conjunto de ejemplos heterogéneos en subgrupos homogéneos o clusters.

Dentro de los algoritmos de agrupamiento podemos distinguir dos tipos principales:

- Algoritmos de particionamiento. En esta aproximación se construye una partición de la base de datos inicial en un conjunto de k clusters, siendo k un valor prefijado (como ocurre, por ejemplo, en *k-means*) o no (como ocurre en el algoritmo *DBSCAN* o en el algoritmo de propagación de afinidad, que usamos en este trabajo).
- Algoritmos jerárquicos. En este caso, se construye un dendrograma o árbol que representa las relaciones de similitud entre los distintos elementos del conjunto de datos y, a continuación, se explora ese árbol para generar clusters, pudiendo obtenerse un agrupamiento distinto dependiendo del nivel de corte que seleccionemos. Esta exploración del árbol da lugar a dos aproximaciones dentro del *clustering* jerárquico:
 - *Clustering* jerárquico aglomerativo. En este tipo de *clustering* se comienza con tantos grupos como individuos y, en cada paso, se van fusionando los clusters más similares.

- *Clustering* jerárquico divisivo. En esta aproximación se da la situación contraria a la anterior, ya que se empieza con un único cluster y, en cada paso, se selecciona el cluster entre cuyos elementos hay menos similitud y se subdivide.

2.2.1. Algoritmo de propagación de afinidad

Tras haber dado una introducción al *clustering*, nos centramos ahora en el algoritmo concreto que hemos usado en este trabajo, el algoritmo de propagación de afinidad (*Affinity Propagation*), que se publicó por primera vez en el año 2007 en un artículo de *Science* escrito por Brendan J. Frey y Delbert Dueck [2].

En el algoritmo de propagación de afinidad se trabaja con puntos representativos o *exemplars*, como los denominan en el artículo original, de forma similar al algoritmo *k-means* y los centroides. En el algoritmo *k-means* se empieza con un conjunto de puntos escogidos aleatoriamente como centroides de los clusters y se asocia el resto de puntos a cada uno de esos grupos e, iterativamente, se van cambiando los centroides y refinando los clusters hasta que no hay cambios. Este algoritmo es muy sensible a la inicialización, por lo que, en muchas ocasiones, hay que ejecutarlo varias veces con conjuntos iniciales diferentes para encontrar una buena solución y, además, es necesario fijar de antemano el número de clusters.

En propagación de afinidad, en vez de tomar un conjunto inicial, se consideran todos los puntos como posibles representantes. Cada uno envía mensajes a los demás puntos informándoles de lo atractivo que le parece cada punto y, entonces, cada uno de los receptores responde a todos los remitentes informando a cada uno de su disponibilidad para asociarse con él, según los mensajes que ha recibido del resto de emisores. Cuando reciben este mensaje, revisan la información y vuelven a mandar mensajes a todos, repitiéndose el proceso hasta que se llega a un consenso y todos los puntos relacionados con el mismo representante se colocan en el mismo cluster.

El algoritmo toma como entrada una colección de valores que representan la similitud entre cada punto, donde cada valor $s(i, j)$ indica cómo de bueno es el punto j para ser el representante del punto i . Este valor de similitud se define como la distancia euclídea entre dos puntos, pero con signo negativo.

$$s(i, j) = - \|x_i - x_j\|^2$$

De esta forma, cuanto mayor es la distancia entre dos instancias, más pequeña es la similitud entre ellas. Respecto a los valores de similitud $s(j, j)$, se denominan “preferencias” y determinan el número de clusters resultantes; cuanto más pequeño sea este valor, menor será el número de grupos obtenidos. Esto se debe a que, para un elemento i , buscaremos uno j que maximice la similitud, la cual será, en todo caso, menor o igual a 0. Por tanto, si el valor $s(j, j)$ es alto, por ejemplo, 0, obtendríamos que el elemento más parecido a un punto dado es él mismo y habría un número elevado de clusters, mientras que si este valor es

más bajo, cada punto i encontrará el valor de similitud más alto en otro punto $i \neq j$ y se formarán, así, menos clusters.

Partiendo de esta información, hay dos tipos de mensajes que se pueden enviar, el valor de responsabilidad ($r(i, j)$) y el de disponibilidad ($a(i, j)$). La responsabilidad $r(i, j)$ que el punto i envía al candidato a representante j refleja cómo de bueno es j para convertirse en el representante de i , teniendo en cuenta a los demás candidatos de i .

$$r(i, j) \leftarrow s(i, j) - \max_{j', j' \neq j} \{a(i, j') + s(i, j')\}$$

Es decir, el valor $r(i, j)$ cuantifica cómo de similar es i con respecto a j , comparado con un j' , teniendo en cuenta la disponibilidad de j' ; de manera que, la responsabilidad de j respecto a i disminuirá conforme la disponibilidad de j' respecto a i aumente. Al principio, todos los valores $a(i, j)$ son 0, por lo que, en la primera iteración, los valores $r(i, j)$ se establecen como la diferencia entre la similitud $s(i, j)$ y la mayor similitud entre i y cualquier otro candidato distinto de j .

La disponibilidad $a(i, j)$, enviada desde el candidato j al punto i , refleja cómo de apropiado sería para el punto i elegir a j como su representante, teniendo en cuenta el apoyo de otros puntos a j como representante. El valor de disponibilidad se actualiza de acuerdo a la siguiente regla:

$$a(i, j) \leftarrow \min \left\{ 0, r(j, j) + \sum_{i', i' \notin \{i, j\}} \max \{0, r(i', j)\} \right\}$$

Es decir, $a(i, j)$ es la responsabilidad propia de j más las responsabilidades positivas de j hacia elementos distintos de i . Únicamente se tienen en cuenta las responsabilidades positivas porque solo es necesario que un representante explique bien algunos de sus puntos (ser positivamente responsable), independientemente de lo mal que explique otros (negativamente responsable). En el caso de que la responsabilidad propia sea negativa (es decir, sería mejor que j se asociara a otro representante, en lugar de serlo él), la disponibilidad de j como representante puede aumentar si otros puntos tienen valores de responsabilidad positivos con j como su representante ($r(i', j) > 0$). Para evitar que los valores más altos de responsabilidad tengan demasiado peso, se pone un umbral para que la suma nunca supere el 0.

La disponibilidad propia $a(j, j)$ refleja la evidencia acumulada de que el punto j es un representante basándose en las responsabilidades enviadas a j por el resto de puntos. Se calcula con la siguiente fórmula:

$$a(j, j) \leftarrow \sum_{i', i' \neq j} \max \{0, r(i', j)\}$$

Los valores de responsabilidad y disponibilidad se van actualizando iterativamente hasta llegar a un número prefijado de iteraciones o hasta que los valores permanezcan

constantes o el cambio observado esté por debajo de un umbral. Un punto i será asignado a un representante j que tenga una responsabilidad alta, así como una disponibilidad alta, respecto a i . Para encontrar el representante j , se suman los valores $r(i,j)$ y $a(i,j)$ ($c(i,j) \leftarrow a(i,j) + r(i,j)$) y, para cada i , se toma el resultado más alto como representante. Los puntos que comparten el mismo representante, pertenecerán al mismo cluster. El pseudocódigo de este algoritmo es el siguiente:

Datos de entrada: Conjunto de instancias

$X = [x_1, x_2, \dots, x_n]$ con
 $x_n = [x_{n1}, x_{n2}, \dots, x_{nm}]$ el conjunto de valores de cada instancia

Resultado: Lista de clusters

$C = [c_1, c_2, \dots, c_n]$ con
 c_n = número de cluster al que pertenece la instancia n

```
#Inicializar matriz de similitudes, disponibilidad y
responsabilidad
s(i,j) ← - ||xi - xj||2
a(i,j) ← 0
r(i,j) ← s(i,j) - max{a(i,j') + s(i,j')} (j' ≠ j)
```

```
#Intercambio de mensajes (actualización de valores)
while not condicion_de_parada:
```

```
a(i,j) ← min{0, r(j,j) + ∑ max{0, r(i',j)}} (i' ∈
{i,j})
a(j,j) ← ∑ max{0, r(i',j)} (i' ≠ j)
r(i,j) ← s(i,j) - max{a(i,j') + s(i,j')} (j' ≠ j)
```

```
#Búsqueda de representantes
c(i,j) ← a(i,j) + r(i,j)
```

```
representantes = []
```

```
for i in numero_instancias:
    representantes ← max (c(i,*))
```

```
#Formación de clusters
```

```
C = []
```

```
#Selección de representantes únicos
repr_unicos ← set(representantes)
```

```
#Búsqueda del índice del cluster dentro de los representantes
únicos
```

```
for i in representantes:
```

```
C(i) ← find_index(representantes(i), repr_unicos)
```

2.3. Redes neuronales

2.3.1. Introducción

Los perceptrones multicapa (MLP), más conocidos como redes neuronales, son los modelos más usados en *deep learning* y, hoy en día, hay muchísimas variantes que han ido surgiendo desde las redes originales. El objetivo de una red neuronal es aproximar una función f^* . Por ejemplo, si vamos a usar una red neuronal para realizar una tarea de clasificación, esa función f^* nos lleva cada elemento x de entrada hasta uno y de salida, es decir, $y = f^*(x)$, con $y \in \{1, 2, \dots, k\}$. Una red neuronal define una función $y = f(x; \theta)$, donde x es el vector de entrada y θ , los parámetros, y aprende los valores de θ que aproximan mejor la función.

Estos modelos de redes se conocen como redes *feedforward*, alimentadas hacia delante, ya que la información únicamente se mueve en una dirección, es decir, primero entra el vector x por la capa de entrada, a continuación, las capas intermedias se encargan de los cálculos dados por la función f y, por último, la capa de salida devuelve un resultado y . Como vemos, no hay ninguna retroalimentación en la que las salidas del modelo vuelvan a entrar en este, lo cual sí se da en otro tipo de redes, denominadas redes neuronales recurrentes.

Las redes neuronales se denominan *redes* porque normalmente son representadas como una composición de varias funciones, donde cada una de ellas se correspondería con una capa de la red, siendo el número total de capas la profundidad del modelo. Precisamente, de este término *profundidad* es de donde surgió el concepto de *deep learning*. En una red neuronal, la última capa se conoce como capa de salida y la información de entrenamiento, que va etiquetada, especifica qué debe devolver esta capa final para cada ejemplo de entrada; es decir, si un ejemplo x_1 va etiquetado como y_1 , entonces, la capa de salida debe producir un valor lo más cercano posible a y_1 . Sin embargo, el comportamiento de las capas intermedias no está explícitamente en los datos de entrenamiento, sino que es el algoritmo de aprendizaje el que debe decidir cómo usar dichas capas para obtener la salida deseada, es decir, para implementar una buena aproximación de f^* . Debido a que los datos de entrenamiento no especifican cuál es la salida esperada para esas capas, estas se denominan capas ocultas.

Respecto a la segunda parte del nombre, *neuronales*, se debe a que surgieron para imitar el cerebro humano, de manera que cada capa oculta suele ser una función vectorial y cada componente de ese vector puede entenderse como una neurona. Cada una de esas componentes es una unidad que actúa en paralelo recibiendo información de entrada, procesándola y devolviendo un valor acorde a dicha entrada, de forma parecida a las neuronas del cerebro, que reciben estímulos, los procesan y envían una respuesta a ellos. Esta fue la motivación original, sin embargo, como comentábamos al principio de la sección,

se han diseñado numerosas variantes de estas redes originales, dando lugar a modelos mucho más complejos que ya no imitan la estructura del cerebro humano.

En cuanto al nacimiento de las redes neuronales, surgieron para lidiar con las limitaciones de los modelos lineales, como la regresión logística o la lineal, para el aprendizaje automático. Dichos modelos son muy sencillos y fueron los primeros que se usaron en el campo del *machine learning*, pero son muy limitados, ya que únicamente pueden aprender funciones lineales, es decir, funciones en las que la relación entre el valor de entrada x y el de salida y sea del tipo $f(x) = ax + b$. Por tanto, al no poder aprender funciones no lineales, tampoco puede explotar la posible relación que pueda haber entre algunas variables de entrada.

Una forma de ampliar estos modelos lineales para que también acepten funciones no lineales es aplicar el modelo lineal a una transformación no lineal de la entrada, es decir, en lugar de aplicarlo a x , se aplica a $\phi(x)$. Sin embargo, hay varias formas de elegir esa transformación ϕ .

1. La primera opción es elegir una función muy general, ya que si la transformación $\phi(x)$ tiene una dimensionalidad suficiente, tendremos capacidad suficiente para ajustar el conjunto de entrenamiento, aunque la generalización del modelo será bastante pobre y los resultados en el conjunto de test no serán demasiado buenos (se daría el fenómeno de sobreajuste u *overfitting*).
2. Otra opción es elegir de forma manual la función ϕ , lo cual era lo más usual antes del desarrollo del *deep learning*. Esta técnica, conocida como selección de características, consiste en elegir manualmente el mapa de características (funciones ϕ_i de la muestra) y aprender una función lineal de dichas características. Sin embargo, se requiere de un gran conocimiento experto y los avances en un área no suelen ser aplicables a otras.
3. Por último, nos encontramos la opción del *deep learning*, aprender la función ϕ . Desde esta aproximación, tenemos un modelo $y = f(x; \theta, w) = \phi(x; \theta)^T w$, donde θ son los parámetros que se usan para aprender ϕ de una amplia clase de funciones y w , los parámetros o pesos que sirven para ponderar los resultados de las funciones y producir la salida deseada. En este enfoque podemos ver englobados los dos anteriores, ya que la función puede ser muy general (si escogemos una clase de funciones suficientemente amplia), permitiéndonos aprender cualquier función, a la vez que el conocimiento experto puede ayudar a la generalización si un humano diseña una clase de funciones $\phi(x; \theta)$ que considera que puede dar buenos resultados (en lugar de encontrar una función exacta, solo debe dar con la clase de funciones exacta).

2.3.2. Estructura de las redes neuronales

Como se ha comentado anteriormente, las redes neuronales se componen de varias capas, cada una de las cuales representa una función y transforma los valores de entrada

en un valor de salida, de acuerdo a dicha función, y este resultado es usado como entrada para las capas siguientes. Cada capa está compuesta por neuronas que operan con funciones $f: \mathbb{R}^n \rightarrow \mathbb{R}$.

La neurona

Las neuronas son las unidades básicas de procesamiento de una red neuronal. Cada una de ellas tiene varias conexiones de entrada a través de las que recibe los valores de entrada con los que debe trabajar como un vector de $X \in \mathbb{R}^n$. A dichos valores se les aplica una función o transformación lineal y se genera un valor de salida. Esta transformación lineal consiste en una suma ponderada de los valores de entrada, donde la ponderación de cada entrada se corresponde con el peso w_i que se asigna a cada una. A dicha suma ponderada se le añade, además, un término independiente denominado sesgo o *bias* (b). Dichos pesos w_1, \dots, w_n , así como el sesgo, son los parámetros que la red neuronal debe aprender. Una vez que tenemos esa combinación lineal, se le aplica una función de activación f para introducir algunas deformaciones no lineales y generar la salida definitiva. Por tanto, el esquema básico de una neurona es el siguiente:

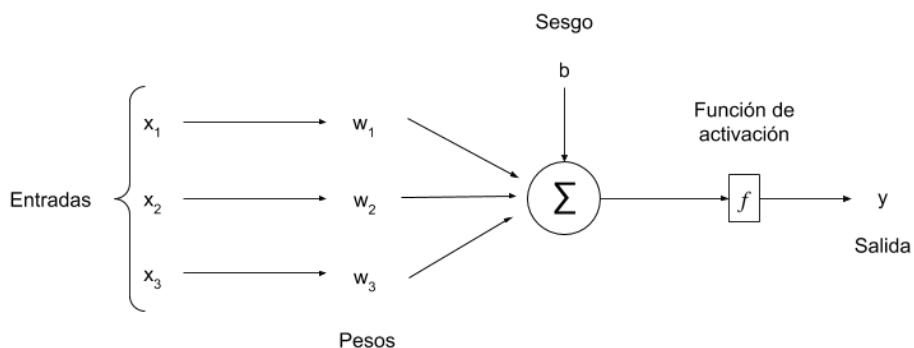


Figura 2.2. Estructura de una neurona

Cada neurona, a la vista de este esquema, genera una salida $y = f(\sum_{i=1}^n w_i x_i + b)$.

De forma simplificada, podemos entender que antes de aplicar f , cada neurona realiza un problema de regresión lineal, lo cual presenta una gran limitación, y es que, aunque encadenáramos el resultado de varias neuronas, el resultado final seguiría siendo lineal, como si solo tuviéramos una neurona en la red. Para solucionar esto se introduce la función de activación, que “deforma” dicha salida lineal.

La función de activación más simple es una función escalonada que da como resultado 1 si el valor de la suma $\sum_{i=1}^n w_i x_i + b$ es mayor que 0, y 0 en caso contrario.

$$y = 1 \text{ si } \sum_{i=1}^n w_i x_i + b > 0$$

$$y = 0 \text{ si } \sum_{i=1}^n w_i x_i + b \leq 0$$

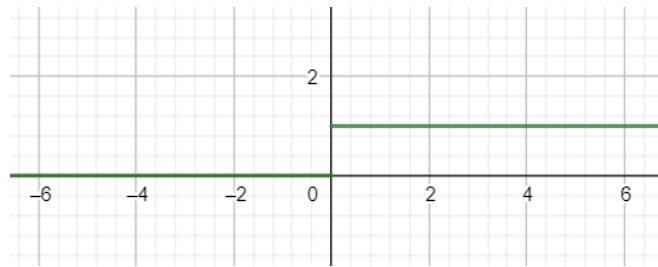


Figura 2.3. Función escalonada

Sin embargo, en la actualidad se usan funciones de activación mucho más complejas, ya que esta función presenta algunas características, como su no continuidad, que dificultan su optimización. Algunas de las funciones que más se usan son las siguientes (para evitar escribir $\sum_{i=1}^n w_i x_i + b$, lo sustituiremos por z):

- **Sigmoide.** Esta función es similar a la anterior, ya que, cuando z es mayor que 0, los valores se acercan a 1, mientras que, cuando z es menor que 0, dichos valores tienden a 0. Sin embargo, suaviza la forma de la anterior, haciéndola continua y derivable, lo cual es más adecuado para la optimización. Su expresión y representación gráfica son las siguientes:

$$y = \frac{1}{1 + e^{-z}} = \sigma(z)$$

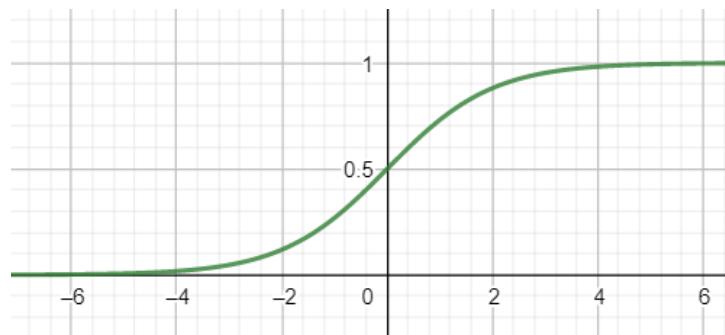


Figura 2.4. Función sigmoide

- **Tangente hiperbólica.** El comportamiento de esta función es muy parecido al de la sigmoide, excepto que ahora, para valores negativos de z , la función toma valores cercanos a -1, en lugar de a 0. La función tangente hiperbólica se expresa de la siguiente forma:

$$y = \frac{(e^z - e^{-z})}{(e^z + e^{-z})} = \frac{2}{1 + e^{-2z}} - 1 = \tanh(z)$$

Su gráfica es:

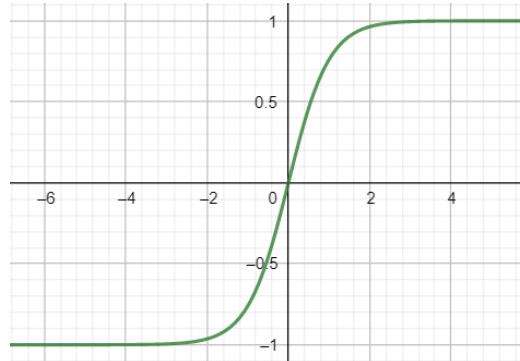


Figura 2.5. Función tangente hiperbólica

- Función lineal rectificada (ReLU). Esta función se comporta como una lineal cuando el valor de entrada es positivo y se mantiene constante en 0 mientras los valores de entrada son iguales o menores a 0. No es derivable en todo su dominio, pero sí a trozos, y su derivada no es nula para las entradas positivas, lo cual es útil para la optimización. Esta función se expresa de la siguiente manera:

$$y = \max\{0, z\}$$

La representación gráfica de la función ReLU es la siguiente:

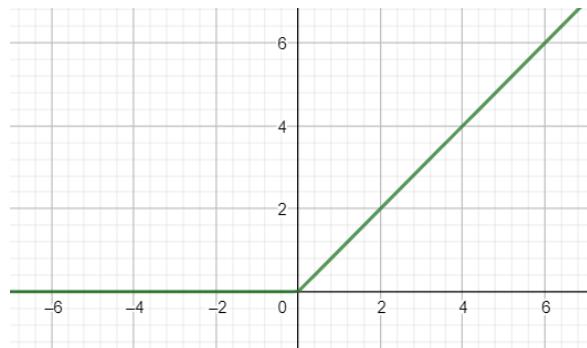


Figura 2.6. Función ReLU

Estas son las funciones de activación más utilizadas, aunque hay muchas otras y el uso de una u otra, o incluso una combinación de varias, suele depender de las características concretas de cada problema tratado.

Arquitectura y capas de una red neuronal

Después de haber visto en detalle las neuronas, podemos pasar a ver cómo se organizan estas dentro de las redes neuronales, formando estructuras más complejas, las capas. Cada capa está compuesta por un grupo de neuronas que reciben las mismas entradas de la capa anterior y envían sus resultados a la siguiente. De esta forma, la red puede aprender conocimiento jerarquizado, aprendiendo cosas más básicas en las primeras capas y usándolas para ir adquiriendo y desarrollando un conocimiento más elaborado y

complejo a medida que se va avanzando por las capas de la red. Por tanto, cuantas más capas se añaden, más complejo es el aprendizaje.

En las redes neuronales nos encontramos con los siguientes tipos de capas:

- Capa de entrada. Es la primera capa de la red neuronal y se encarga de recibir los datos de entrada y pasárselos a la siguiente capa. En esta capa no se aplica ninguna función a los datos ni hay pesos o sesgo, por lo que podemos verla como si fuera el vector de características de entrada, donde la neurona i -ésima genera como salida el valor x_i .
- Capas ocultas. Estas capas se encuentran entre la de entrada y la de salida y están formadas por neuronas que aplican distintas transformaciones a los datos de entrada. Cada una recibe como entrada la salida de la capa anterior y envía su salida a la capa siguiente y cada neurona está conectada con todas las de la capa anterior y siguiente.
- Capa de salida. Es la última capa de la red neuronal y suele tener menos neuronas que las demás. De hecho, en muchos problemas, como, por ejemplo, en clasificación, tiene una única neurona que da como resultado la clase o categoría predicha por el modelo.

De forma simplificada, una red neuronal tiene la siguiente estructura:

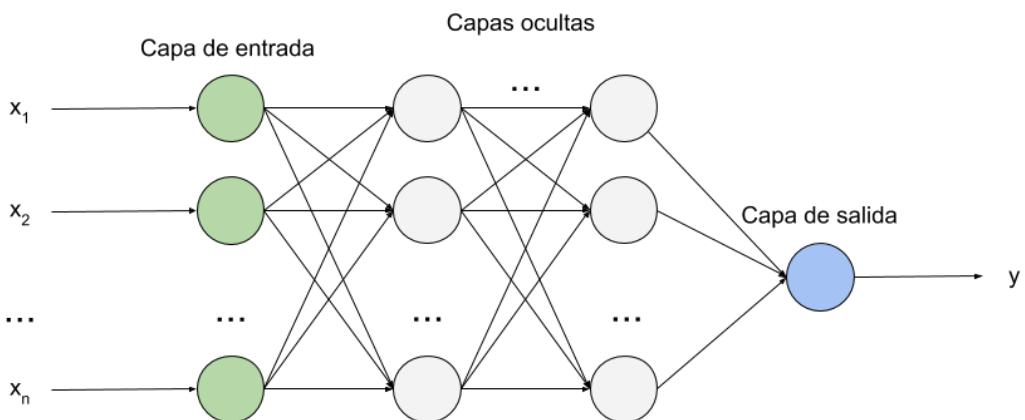


Figura 2.7. Estructura de una red neuronal

En la capa de entrada tendremos tantas neuronas como datos haya en el vector de entrada, es decir, n neuronas para un vector de \mathbb{R}^n . En las capas ocultas podemos tener tantas neuronas como queramos, pudiendo haber una capa con 5 neuronas y otra con 3, por ejemplo. Por último, la capa de salida, que en este esquema se presenta como una única neurona, aunque, como hemos comentado, puede estar compuesta por más.

Escoger el número de capas ocultas, así como el número de neuronas de cada una de ellas es uno de los problemas que presenta el diseño de una red neuronal, ya que, con

un número más elevado de ellas se puede obtener un conocimiento más profundo, pero si ponemos demasiadas el modelo se vuelve muy complejo y, aparte de necesitar más recursos para su ejecución, puede aprender conocimiento no útil y generar resultados y predicciones peores que con menos capas o neuronas por capa. En cuanto al número de neuronas de cada capa oculta, por ejemplo, se puede recurrir a algunos “comodines”, como poner tantas neuronas como resulte de sumar las neuronas de la capa de entrada y la de salida o la mitad de dicho valor.

2.4. Redes neuronales convolucionales (CNN)

Como comentábamos anteriormente, el modelo de red neuronal que hemos visto en el apartado anterior es el modelo más clásico, donde todas las capas están totalmente conectadas; de ahí que también se conozcan como redes neuronales completamente conectadas. A partir de ellas han surgido numerosas variantes, entre las que se encuentran las redes neuronales convolucionales (abreviadas como CNN, del término en inglés, *Convolutional Neural Network*), las cuales están diseñadas para trabajar con datos con una topología similar a una cuadrícula, donde los elementos más cercanos tendrán, probablemente, una relación que no existirá entre elementos lejanos. Por ejemplo, se usan para trabajar con series temporales, que podemos ver como una cuadrícula de una dimensión donde cada valor se toma en un instante de tiempo y está muy relacionado con los inmediatamente anterior y posterior; o para procesar imágenes, que son cuadrículas de píxeles en dos dimensiones.

El concepto de red neuronal convolucional viene de la operación matemática que se aplica en las capas centrales de este tipo de redes, la operación de convolución, la cual explicaremos a continuación con más detalle.

2.4.1. Operación de convolución

En su forma más general, la operación de convolución es una operación matemática entre dos funciones de variable real que da lugar a otra función y se denota con un asterisco. Por ejemplo, si tenemos las funciones $f: \mathbb{R} \rightarrow \mathbb{R}$ y $g: \mathbb{R} \rightarrow \mathbb{R}$, la convolución de f y g es la siguiente:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(t - x)dx$$

Esta convolución es la integral del producto de ambas funciones al desplazar una de ellas, en este caso, g , una distancia t . En el caso de tener funciones discretas, la convolución se define de la siguiente forma:

$$(f * g)(t) = \sum_{n \in \mathbb{Z}} f(n)g(t - n)$$

Normalmente, el primer argumento, la función f , se conoce como entrada y se denota como x , mientras que el segundo argumento, la función g , se denomina *kernel* o núcleo y se denota como w . En el caso de tener una entrada bidimensional, como ocurre con las imágenes, la operación de convolución quedaría definida así:

$$(f * g)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} f(n, m)g(s - n, t - m) = (x * w)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} x(n, m)w(s - n, t - m)$$

Además, esta operación es conmutativa, por lo que la expresión anterior es equivalente a:

$$(w * x)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} x(s - n, t - m)w(n, m)$$

Por lo general, esta fórmula es más sencilla de implementar y más práctica. La propiedad conmutativa de la convolución surge al invertir el núcleo, ya que cuando los índices n y m aumentan, también lo hace el índice de entrada, a la vez que el del *kernel* disminuye. El único motivo por el que se realiza esta inversión es para hacer que la operación sea conmutativa y se pueda hacer el cambio anterior. En muchas bibliotecas de *deep learning* se implementa una función similar a la convolución, pero sin invertir el núcleo (es decir, no es conmutativa como la convolución), la operación de correlación cruzada:

$$(x * w)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} x(s + n, t + m)w(n, m)$$

Realmente, que la operación sea o no conmutativa no es muy importante, ya que los algoritmos de aprendizaje aprenden los valores del *kernel* que le sirven para extraer las características relevantes de los datos de entrada y el único cambio que se produce al usar la operación de convolución (conmutativa) o la de correlación cruzada (no conmutativa) es que dichos pesos estarán invertidos, pero los resultados son equivalentes.

De acuerdo a lo que hemos visto, la operación de convolución puede entenderse como una multiplicación de matrices: el núcleo es una pequeña matriz de pesos que va “deslizándose” sobre los datos de entrada, que componen una matriz más grande, y se va realizando una multiplicación elemento a elemento con la parte de la entrada que corresponda para, después, resumir el resultado en un único píxel de salida. Este proceso se repite para toda la matriz de entrada y se obtiene como resultado una matriz más reducida.

Podemos ver el proceso con la siguiente imagen:

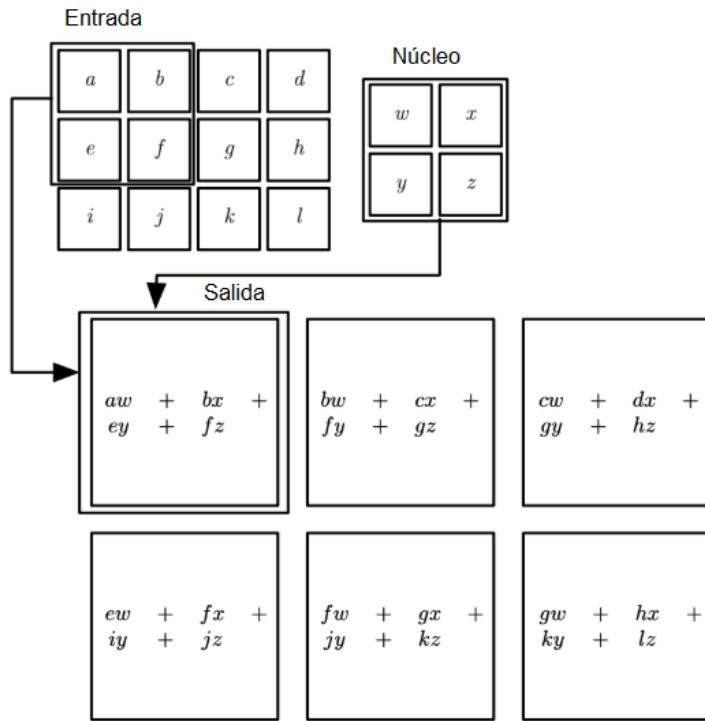


Figura 2.8. Operación de convolución. Imagen extraída y adaptada de [3, p. 325]

Así, dependiendo de los valores que especifiquemos en el núcleo, el resultado será distinto. Es decir, dar más peso a unos u otros valores nos permitirá quedarnos con determinada información de la entrada, a la vez que ignoraremos otra parte. A continuación, vemos el resultado de aplicar una convolución a una imagen:



Figura 2.9. (Izquierda) Imagen original. (Derecha) Imagen tras convolución.
Imagen extraída de [4]

Como podemos ver, al aplicar el filtro a la imagen original, obtenemos una nueva imagen donde únicamente aparecen resaltados los bordes (tanto verticales como horizontales y diagonales). Para obtener este resultado es necesario combinar en un solo núcleo los filtros para detectar las líneas horizontales, verticales y diagonales. Por tanto, el *kernel* que obtenemos es el siguiente:

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 2.10. Núcleo de convolución para obtener los bordes en una imagen

Con este filtro de 3x3 conseguimos que, si en una porción de este tamaño de la imagen original, los 9 píxeles tienen el mismo color, el resultado de multiplicar los valores del núcleo por los de dichos píxeles y sumarlos es igual a 0 (por ejemplo, si los 9 píxeles tienen valor 1, el resultado final sería $8*1 + 8*(-1) = 0$). Por otra parte, las zonas donde haya algún cambio de color (frontera entre dos zonas, entre las cuales habrá un borde), el resultado de la multiplicación será distinto de 0. Así, las zonas con intensidad 0 aparecerán de color negro en la imagen final, mientras que las zonas con intensidad distinta de 0, los bordes, se mostrarán de color más claro.

Hay muchos otros núcleos con valores prefijados para obtener resultados concretos (obtener solo las líneas horizontales o verticales o difuminar la imagen, por ejemplo), sin embargo, en el aprendizaje profundo, lo que se busca es que la red neuronal optimice los valores de la matriz como cualquier otro parámetro para que aprenda a extraer la información relevante para el problema que se esté tratando.

Características de la convolución

La convolución aprovecha tres ideas importantes que son útiles para mejorar los sistemas de aprendizaje automático, además de proporcionar un medio para trabajar con entradas de tamaño variable. Dichas ideas son las siguientes:

- **Interacciones dispersas.** En las redes neuronales clásicas, el vector de entrada se multiplica por un vector de pesos del mismo tamaño y la salida de esta multiplicación se envía a todas las neuronas de la capa siguiente. Es decir, cada unidad de salida interactúa con cada unidad de entrada. Sin embargo, en las redes convolucionales se trabaja localmente con porciones de la imagen original, por lo que, si tenemos un núcleo de tamaño 3x3, solo 9 unidades de salida se ven afectadas por cada unidad de entrada (y viceversa). Obviamente, esto se consigue cuando el tamaño del *kernel* es menor que el de la matriz de entrada, ya que si ambas matrices tienen las mismas dimensiones, el resultado equivale al de una red neuronal clásica. Esto es lo que se conoce como interacciones o pesos dispersos y permite que se extraigan características relevantes de pequeño tamaño y se vayan combinando desde las más simples hasta obtener otras mucho más complejas. Además, también mejora la eficiencia del algoritmo, al reducirse el número de operaciones.
- **Parámetros compartidos.** En cuanto a esta característica, hace referencia al uso del mismo parámetro por varias funciones del modelo. Como hemos comentado, en las redes neuronales convencionales se multiplica cada elemento de entrada por un

valor del vector de pesos, es decir, cada peso se utiliza una única vez, aunque todos los valores sean iguales. Esto implica que el vector de pesos tiene que tener el mismo tamaño que el de entrada, lo cual puede llevar a un consumo excesivo de memoria y nos obliga a fijar un tamaño para los vectores de entrada. Con los parámetros compartidos de las redes convolucionales podemos hacer frente a estos problemas, ya que, una vez que definimos el tamaño del *kernel* de convolución, cada uno de los valores de este se aplica a cada posición del vector de entrada (excepto, quizás, en algunos píxeles de los bordes, según se especifique cómo debe “moverse” el núcleo sobre la matriz de entrada), lo cual hace que no haya que aprender un conjunto de pesos de tamaño distinto para cada entrada y nos permite trabajar con entradas de tamaño variable. Además, dado que el *kernel* suele ser de tamaño pequeño, se reduce de manera considerable la demanda de memoria del modelo.

- Representación equivariante. El concepto de equivarianza o representación equivariante se debe al uso de parámetros compartidos y hace referencia al cambio que se produce en la salida cuando hay alguna modificación en la entrada. Una función se dice equivariante si, ante un cambio en la entrada, la salida cambia de la misma forma. Concretamente, una función $f(x)$ es equivariante a otra función $g(x)$ si $f(g(x)) = g(f(x))$. La convolución es equivariante a traslaciones, de manera que si tenemos dos vectores de entrada iguales, excepto que uno presenta un desplazamiento con respecto al otro, la salida de las capas de la red neuronal será la misma, pero con ese desplazamiento. También es equivariante a cambios de brillo, pero no lo es a otras transformaciones como pueden ser los cambios de escala o la rotación.

2.4.2. Capas en una red neuronal convolucional

A continuación, veremos qué capas solemos encontrarnos en las redes neuronales convolucionales. Normalmente, hay tres etapas: en la primera se aplican varias convoluciones en paralelo para obtener varios mapas de características básicas, a continuación, se aplica una función de activación para introducir no linealidad y, después, se usa una función de *pooling* para modificar la información y resumirla.

Capa convolucional

Este es el tipo de capa principal en una CNN, ya que es la que se encarga de aplicar la operación de convolución sobre la entrada. Esta capa recibe de entrada un tensor (de forma general, este término se usa para referirse a vectores, de dimensión 1, matrices, de dimensión 2, y otras estructuras equivalentes de dimensiones mayores) y aplica tantas convoluciones como se especifique sobre dicho tensor (los núcleos que se quieran aplicar a las entradas se almacenan en la memoria de esta capa). De esta forma, obtendremos tantos tensores de salida o mapas de características como filtros se hayan definido en la capa.

Al principio, los valores de cada *kernel* se establecen a un valor aleatorio y, a medida que el modelo va entrenando y adquiriendo conocimiento, dichos valores se van cambiando para ser capaces de extraer las características que más nos interesen de los tensores de entrada.

Lo normal es que se apilen varias capas de convolución, de manera que las primeras contienen núcleos más pequeños que extraen características muy básicas de las imágenes de entrada, como pueden ser los bordes o los colores, y las más profundas se encargan de combinar estas características para encontrar formas de mayor complejidad. Las primeras capas, por tanto, extraen características de bajo nivel, mientras que las siguientes extraen las de alto nivel.

Capa de activación

Esta capa, que siempre va a continuación de una capa de convolución, tiene el mismo objetivo que las funciones de activación en las neuronas; introducir una distorsión para que la salida sea no lineal. En este caso, la más utilizada es la función ReLU, que, como vimos, transforma los valores negativos en 0. De esta forma, se evita que los valores aprendidos se estanquen muy cerca del 0. En la imagen siguiente podemos ver el efecto de esta capa:

9	3	5	-8
-6	2	-3	1
1	3	4	1
3	-4	5	1

→

9	3	5	0
0	2	0	1
1	3	4	1
3	0	5	1

Figura 2.11. Efecto de la función ReLU

Capa de *pooling*

La capa de *pooling* se encarga de sustituir la salida de la red en un determinado punto por un resumen estadístico de un grupo de píxeles, es decir, reduce el tamaño o dimensión de la entrada que recibe. De esta forma, se disminuye la potencia computacional requerida para procesar los datos y, además, es útil para extraer características dominantes invariantes rotacional y posicionalmente. Los dos tipos de *pooling* más usados son el *max pooling* y el *average pooling*. En el primero se devuelve como salida el valor máximo de la porción de la imagen que cubra el *kernel* y en el segundo, se devuelve la media de dichos valores.

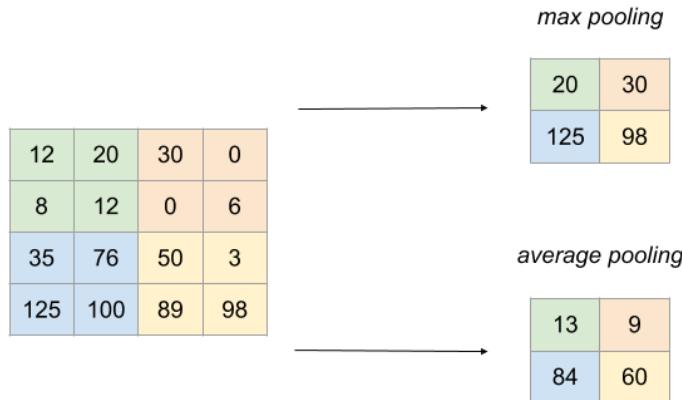


Figura 2.12. Resultado de aplicar *max pooling* y *average pooling*

Estas capas llevan asociadas varias ventajas. En primer lugar, hacen que la red sea más robusta frente a cambios pequeños en la entrada, ya que si estos valores varían ligeramente, los resultados del *pooling*, probablemente, no cambiarán; lo que disminuye el ruido. Además, como hemos visto, reduce el tamaño de las entradas que recibe, por lo que un grupo de varios píxeles queda resumido en uno solo. Esto, aparte de reducir los recursos computacionales y de memoria que necesita el modelo (al trabajar con imágenes más pequeñas, hay que hacer menos cálculos), también hace que este generalice mejor.

Capa completamente conectada

Las últimas capas de una red neuronal convolucional suelen ser de este tipo, al menos la última capa. Como pasaba en las redes neuronales clásicas, este tipo de capa tiene un tamaño de entrada fijo, por lo que se suele poner justo después de una capa de *pooling*, que nos permite transformar una entrada en una salida del tamaño que queramos.

La capa de convolución recibe una imagen determinada como entrada y devuelve otra con, prácticamente, el mismo tamaño (dependiendo de cómo se traten los píxeles de los bordes), que pasa por la capa de activación sin cambiar sus dimensiones. De esta forma, la salida de las distintas capas de convolución y activación tendrá un tamaño distinto, que no tiene por qué coincidir con el tamaño de la capa completamente conectada. Por eso se coloca una capa de *pooling* justo antes, para que, independientemente de las dimensiones de la entrada, las dimensiones de salida siempre sean las mismas. En la imagen del apartado anterior, por ejemplo, el *pooling* aplicado trabaja con una matriz de 2x2, por lo que la salida de esta capa, siempre que el número de píxeles original sea par, tendrá la mitad de píxeles, tanto de ancho como de largo.

2.4.3. Estructura de una CNN

Una vez explicadas las capas más importantes de las redes neuronales convolucionales, vamos a ver cómo se conectan. Normalmente, las capas de este tipo de redes se presentan en grupos de tres, concretamente, las tres primeras que hemos explicado en el apartado anterior, en ese mismo orden:

- Capa de convolución para extraer las características de la entrada como una combinación lineal.
- Capa de activación para transformar la salida de la capa anterior en una función no lineal del mismo tamaño.
- Capa de *pooling* para reducir el tamaño de la entrada o, si está justo antes de una capa completamente conectada, para transformar la entrada al tamaño que se necesite para dicha capa.

En las primeras capas, como hemos visto antes, se extraen características simples, sobre todo líneas rectas, y en las capas más profundas se buscan formas con más detalle y complejidad. Por tanto, las primeras capas suelen tener unos pocos núcleos de pequeño tamaño y, conforme vamos profundizando en la red, las capas tienen más núcleos y de mayor tamaño.

Una vez que se realizan todas las operaciones de convolución, activación y *pooling* necesarias, nos encontramos con una o varias capas completamente conectadas que interpretan la información obtenida de las capas anteriores y generan una salida para la red. Antes de estas, después de la última operación de *pooling*, suele haber una capa que se encarga de colapsar el tensor de entrada en un vector de una única dimensión para pasarlo a las capas completamente conectadas (se suele denotar como *flatten*) . El esquema de una red neuronal convolucional, por tanto, es el que podemos ver a continuación:

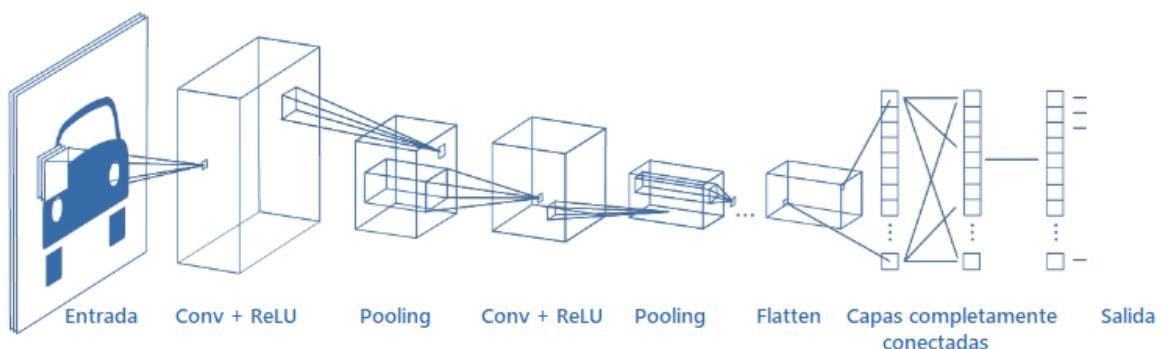


Figura 2.13. Esquema de una CNN. Imagen extraída y adaptada de [5]

2.5. Redes neuronales residuales (ResNet)

Un problema que nos encontramos cuando las redes neuronales se hacen demasiado profundas es que se degrada su capacidad de aprendizaje, problema que se conoce como desvanecimiento del gradiente. Este problema consiste en que el gradiente de la función de pérdida (función usada para medir el nivel de error), que se usa para saber cómo influyen los parámetros de las primeras capas en el resultado final, toma valores muy cercanos a 0. Esto impide que los pesos de la red se actualicen adecuadamente y, por tanto, la etapa de entrenamiento y aprendizaje se vuelven inútiles. En [6] estudiaron este

problema y ofrecieron como solución un nuevo tipo de red neuronal, las redes neuronales residuales o *ResNet* y su marco de aprendizaje residual profundo.

En lugar de aprender una transformación directa de x a y con una función $H(x)$, que se corresponde con una serie de capas no lineales apiladas, se define una función residual usando $F(x) = H(x) - x$, que equivale a $H(x) = F(x) + x$, donde $F(x)$ representa las capas no lineales apiladas y x , la función identidad donde la salida es igual a la entrada. La hipótesis de los autores es que resulta más sencillo optimizar la transformación residual, $F(x)$, que optimizar la original, $H(x)$. Usando una serie de capas de neuronas apiladas, es más fácil conseguir el resultado $y = x$ haciendo que $F(x)$ sea igual a 0 y sumándole x ($y = F(x) + x = 0 + x$), que haciendo que $F(x)$ sea directamente igual a x ($y = F(x) = x$). Esto es lo que se consigue haciendo uso de los bloques residuales, como vemos en la imagen siguiente:

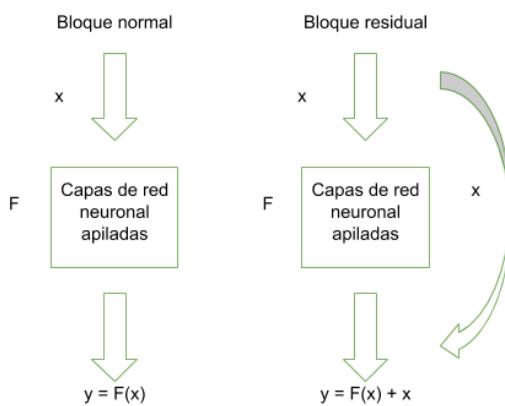


Figura 2.14. (Izda.) Bloque normal, $y = F(x) = x$. (Dcha.) Bloque residual, $y = F(x) + x = x$

De esta manera, la formulación de $F(x) + x$ puede realizarse con redes neuronales clásicas en las que haya algunos “atajos” o conexiones directas como la que vemos en el bloque residual. Dichas conexiones directas son aquellas que se saltan una o varias capas y, en este caso, simplemente actúan como función identidad y llevan el valor de entrada a la salida, para sumarlo con el resultado obtenido por las capas intermedias. Estas conexiones no introducen parámetros extra ni aumentan la complejidad computacional del modelo.

La función residual es flexible, es decir, puede cubrir dos o tres capas o incluso más. Sin embargo, si solo cubre una capa, el resultado es prácticamente el mismo que en una red usual, por lo que no se observan ventajas. Además, las capas que cubre no tienen por qué ser capas completamente conectadas, puede tratarse de capas convolucionales como las comentadas en la sección anterior.

En [6] evaluaron esta nueva arquitectura sobre el dataset *ImageNet* (clasificación de imágenes con 1000 categorías distintas). Hicieron experimentos con redes estándar, sin saltos, de 18 y 34 capas y con redes residuales con el mismo número de capas. En cuanto a los resultados obtenidos, las redes sin saltos obtuvieron un porcentaje de error bastante

más elevado que las residuales, siendo peor el valor de la de 34 capas que el de la red de 18. Las redes residuales consiguieron unos resultados considerablemente mejores, sobre todo la más profunda. Podemos ver las curvas de error que se obtuvieron en las siguientes gráficas:

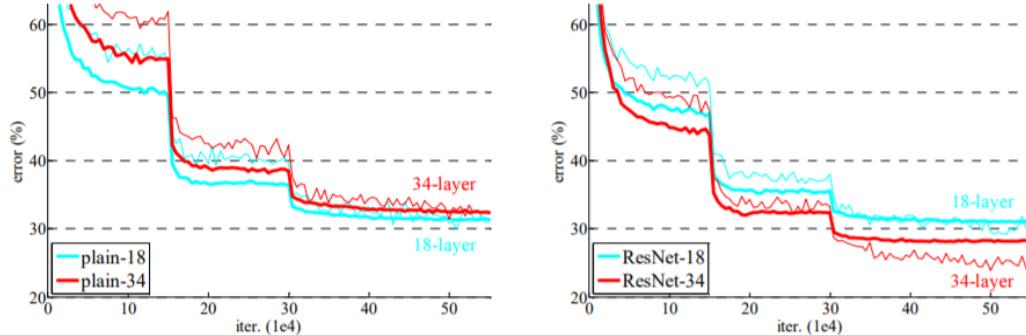


Figura 2.15. Curvas de error de las redes estándar frente a las ResNet.

Imagen extraída de [6, p. 5]

Aparte de estas arquitecturas, hay varias con mayor profundidad, como la ResNet-50 o la ResNet-101 (con 50 y 101 capas, respectivamente), en las que los bloques de 2 capas se cambian por bloques de 3 y se obtienen resultados aún mejores que en las que acabamos de ver. Estos modelos pre-entrenados en datasets como *ImageNet* o COCO se usan mucho en problemas de reconocimiento de objetos, ya que, al haber sido entrenados previamente, contienen los pesos y sesgos que representan determinadas características relevantes y pueden ser aplicables a otros conjuntos de datos (esto es lo que se conoce como transferencia de aprendizaje o *transfer learning*). En concreto, el algoritmo de *tracking* usado en este trabajo hace uso de ambas: la ResNet-50 para la reidentificación de las personas y la ResNet-101 para la detección en una red de tipo Faster R-CNN, que veremos en los apartados siguientes.

2.6. Redes neuronales convolucionales basadas en regiones (R-CNN)

Como comentábamos antes, las CNN han adquirido mucha importancia en los últimos años en temas como la clasificación de imágenes. Sin embargo, hay otras tareas, como la detección de objetos, que puede usarse, por ejemplo, en vigilancia, en los que el objetivo final no es decir a qué categoría pertenece dicho objeto, sino que lo que buscamos es delimitar dicho objeto, dibujando un cuadro a su alrededor. Además, no necesariamente tiene que ser un único objeto, ya que en la misma imagen podría haber varios que quisiéramos detectar, sin saber de antemano cuántos son. Por tanto, esta tarea es difícil de resolver con una red convolucional normal, ya que, según hemos visto en el apartado anterior, la capa final de esta debe ser completamente conectada, es decir, tiene una longitud de entrada y salida fija, y estamos considerando la posibilidad de que haya un número desconocido de objetos a detectar.

Una posible solución a esto sería seleccionar varias regiones de interés de la imagen en cuestión y usar una CNN para clasificar un objeto dentro de cada región. Sin embargo, si hubiera muchos objetos, habría que seleccionar muchas regiones, lo que podría provocar que el modelo “explotara” computacionalmente. Para encontrar dichas áreas de forma rápida y hacer frente a este problema, en [7] propusieron las redes neuronales convolucionales basadas en regiones o *R-CNN*.

El método propuesto selecciona 2000 regiones de la imagen original, de manera que, en lugar de intentar clasificar una cantidad ingente de regiones, se trabaja únicamente con esas 2000. Dichas regiones se seleccionan con el algoritmo de búsqueda selectiva propuesto en [8], que explicaremos a continuación.

2.6.1. Búsqueda selectiva

Los objetivos para un algoritmo de búsqueda son, a grandes rasgos, los siguientes:

- Detectar objetos de cualquier tamaño.
- Considerar criterios múltiples para agrupar (color, textura...).
- Ser rápido.

Ante esto, los autores de [8] propusieron el siguiente algoritmo:

1. Generar subsegmentación inicial, es decir, generar múltiples regiones, cada una de las cuales cubre, como máximo, a un objeto.
2. Combinar recursivamente regiones similares en otras de mayor tamaño. Para esta tarea se usa un algoritmo de tipo *greedy*, que va eligiendo las dos regiones más similares y combinándolas en otra más grande y repite este paso hasta que solo quede una región. Esto hace que haya una jerarquía de regiones cada vez más grandes.
3. Usar las regiones generadas para producir ubicaciones de objetos candidatas.

Visualmente, el comportamiento del algoritmo es el siguiente:

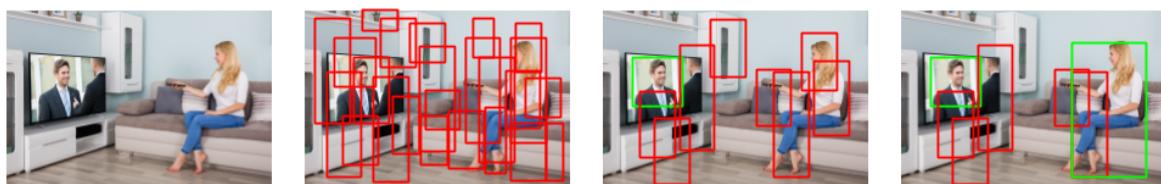


Figura 2.16. (Izquierda) Imagen original. (Derecha) Búsqueda selectiva

La primera imagen sería la entrada, y en la parte de la derecha vemos el primer paso, donde se seleccionan numerosas regiones, que, posteriormente, se van agrupando en el paso 2. Por último, se devuelven unas pocas zonas en las que hay objetos “interesantes”.

Una vez que tenemos las regiones definitivas, estas se pasan como entrada a una red neuronal convolucional que genera como salida un vector de características de dimensión 4096. La CNN, por tanto, actúa como un sistema extractor de características y la capa completamente conectada final (o capa densa de salida) tiene las características obtenidas de la imagen, que se introducen en un SVM (*Support Vector Machine*) para clasificar la presencia del objeto en esa región propuesta. Además de predecir la presencia de los objetos en esas regiones, el algoritmo también predice cuatro valores de desplazamiento para aumentar la precisión del cuadro delimitador y ajustarlo más al objeto en cuestión.

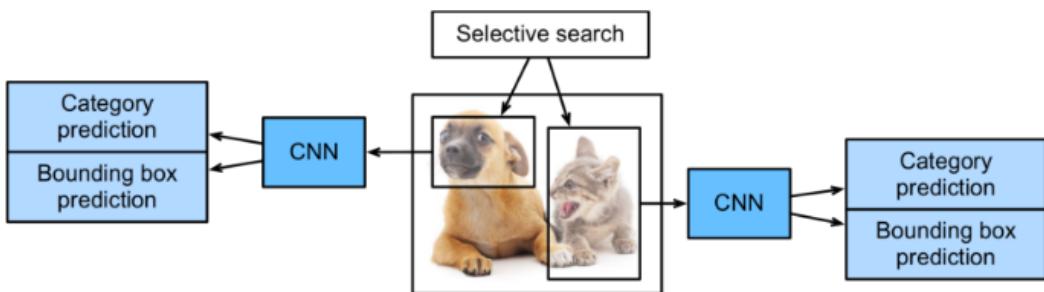


Figura 2.17. Arquitectura de una R-CNN. Imagen extraída de [9]

En esta imagen podemos ver, a grandes rasgos, el funcionamiento de una R-CNN. En este caso, se obtienen 2 regiones como resultado de la búsqueda selectiva, las cuales se introducen en la red neuronal convolucional, que se encarga de extraer las características de ellas. Una vez que se obtienen dichos vectores de características, se realiza la clasificación mediante SVM y se generan los valores de desplazamiento para ajustar el cuadro delimitador.

Aunque esta aproximación mejora la eficiencia de una red convolucional estándar, sigue necesitando una gran cantidad de tiempo, ya que hay que clasificar miles de regiones en cada imagen. Ante este inconveniente, el mismo autor de [8] propuso una mejora sobre este modelo, la cual mejora la eficiencia de forma considerable, en [10]. A continuación, entramos en más detalle en esta nueva implementación.

2.6.2. R-CNN rápida (Fast R-CNN)

Este algoritmo es muy similar al anterior, pero, en vez de introducir las regiones candidatas en la red convolucional, se le pasa directamente la imagen de entrada para generar un mapa de características convolucional. Una vez que tenemos dicho mapa, identificamos las regiones y las hacemos pasar por una capa de *RoI pooling* para darles un tamaño fijo e introducirlas en una capa completamente conectada. A partir del vector

obtenido tras la capa de *RoI pooling* se usa una función *softmax* para predecir la clase de la región concreta, así como los valores para ajustar la precisión de la caja delimitadora.

RoI pooling

Como explicamos en el apartado 2.4.2., la capa de pooling se usa mucho para convertir una entrada de tamaño variable en una salida de tamaño fijo, sobre todo, cuando la siguiente capa a esta es una totalmente conectada. Vimos, como ejemplo, el *max pooling* y el *average pooling*, y es, precisamente, el primero de ellos el que se suele utilizar en el *RoI pooling* de una Fast R-CNN.

El *RoI (Region of Interest) pooling* se usa para concatenar todas las características extraídas de cada región propuesta y devolver esto como una salida de tamaño fijo, haciendo uso, por ejemplo, del *max pooling*. Esta capa recibe dos entradas: un mapa de características obtenido por la CNN y una serie de regiones de interés. El *RoI pooling* toma cada región de la entrada y una sección del mapa de características que corresponde a dicha región y convierte esa parte del mapa en uno de dimensión fija. Una vez que tenemos los mapas de tamaño fijo de todas las regiones de interés, se introducen en la capa densa y, para cada una de ellas, se devuelve la clasificación, así como los valores de desplazamiento para mejorar la precisión del marco delimitador.

Función *softmax*

La función *softmax* convierte un vector de valores reales en otro vector de valores reales equivalentes cuya suma es igual a 1. La entrada puede contener valores negativos y positivos o 0, pero la salida siempre tendrá valores entre 0 y 1 que pueden interpretarse como probabilidades. De esta forma, un valor de entrada muy pequeño o negativo resultará en una probabilidad muy cercana a 0, mientras que los valores más grandes se acercarán más a 1. La fórmula de esta función es la siguiente:

$$\sigma(\bar{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Donde los valores z_i son los elementos de entrada, que pueden tomar cualquier valor. El término del denominador es el que se encarga de normalizar los valores para que la suma de todos ellos sea 1.

Esta función es muy utilizada en problemas de clasificación, ya que la última capa devuelve una serie de valores que no están normalizados, por lo que no pueden interpretarse como probabilidades. Con el uso de esta función, dichos valores se normalizan en el rango [0, 1], lo que nos indica la probabilidad de que una determinada entrada pertenezca a cada clase.

El comportamiento de una red Fast R-CNN, por tanto, puede verse en la siguiente imagen:

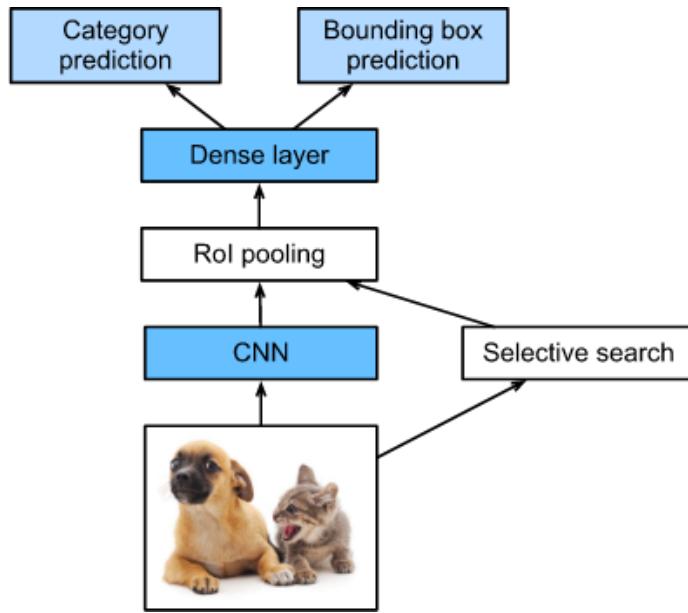


Figura 2.18. Arquitectura de una Fast R-CNN. Imagen extraída de [9]

Al aplicar únicamente una operación de convolución sobre la imagen de entrada, en lugar de pasarle 2000 regiones distintas a la red convolucional, esta implementación consigue una mejora muy importante en tiempo con respecto a la original. Sin embargo, también se demostró que, incluir las regiones candidatas hace más lento al algoritmo, convirtiéndose esto en el principal cuello de botella. Ante esto, de nuevo, Ross Girshick, junto con varios investigadores, autores de [6], propusieron una versión aún más rápida en [11], que veremos a continuación.

2.6.3 R-CNN más rápida (Faster R-CNN)

Tanto en R-CNN como en Fast R-CNN se usa la búsqueda selectiva para encontrar regiones candidatas, pero este proceso es lento y consume mucho tiempo, afectando al rendimiento total del modelo. Para hacer frente a este problema, en [11] propusieron un algoritmo que elimina esta búsqueda selectiva, reemplazándola por una red neuronal. Esto reduce el número de regiones propuestas, a la vez que mejora la precisión a la hora de detectar los objetos.

De la misma forma que en el algoritmo anterior, la imagen original se introduce en la red neuronal convolucional para extraer el mapa de características y, a continuación, se obtienen las regiones de interés, las cuales pasan por una capa de *RoI pooling* y, finalmente, llegan a la capa completamente conectada. Es decir, el comportamiento de una Faster R-CNN es igual al de una Fast R-CNN, excepto por el método para generar las regiones candidatas. El nuevo método consiste en usar una *Region Proposal Network* (RPN), que veremos con más detalle a continuación.

Region Proposal Network (RPN)

Este tipo de red neuronal introducido en [11] toma como entrada una imagen de cualquier tamaño y genera como salida una serie de propuestas rectangulares, cada una de ellas con una puntuación que mide la probabilidad de que esa sección contenga un objeto o sea simplemente fondo.

La imagen de entrada se introduce en una red neuronal convolucional, que extrae el mapa de características correspondiente. Para cada punto de ese mapa, la red tiene que aprender si hay un objeto presente en la imagen de entrada en su ubicación correspondiente y estimar su tamaño. Esto se hace colocando una serie de “anclas” en la imagen de entrada, una por cada ubicación del mapa de características obtenido por la red. Estos indicadores marcan posibles objetos de varios tamaños y ratios de aspecto en esa ubicación. En la siguiente imagen podemos ver 9 de estos indicadores de distintos tamaños y orientaciones.

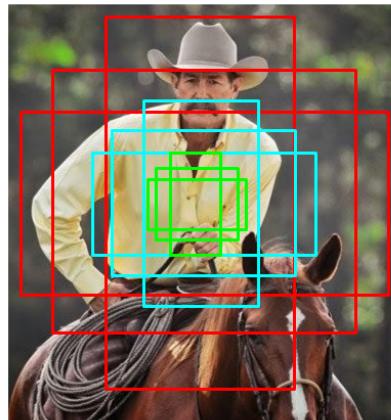


Figura 2.19. “Anclas” en una imagen

Conforme la red se va moviendo por cada punto del mapa de características, tiene que verificar si las “anclas” realmente contienen objetos y refinar las coordenadas de las cajas delimitadoras para devolverlas como regiones de interés. A continuación, se aplica una convolución de 3x3 para obtener un mapa de características de cada ubicación y después, se usa una capa de convolución 1x1 para clasificación y otra para regresión. En la de clasificación se devuelven las probabilidades de que cada punto del mapa de características contenga un objeto dentro de los indicadores o “anclas” de ese punto; en la de regresión se devuelven los cuatro valores de desplazamiento usados para mejorar la posición de los cuadros delimitadores que contienen objetos.

Como paso final, cuando tenemos las distintas regiones propuestas, se usa el algoritmo NMS (*Non-maximum Suppression*) para eliminar las cajas delimitadoras similares y superpuestas. Este algoritmo recibe como entrada la lista de regiones propuestas, con su puntuación de confianza correspondiente, y un umbral de solapamiento. El proceso que sigue es el siguiente:

1. Seleccionar la región con mayor puntuación, quitarla de la lista de entrada y añadirla a la lista definitiva.

2. Comparar esta región propuesta con el resto, calculando el valor IoU (*Intersection over Union*) con cada una. Si dicho valor es mayor que el umbral de solapamiento, se elimina la región que se esté comprobando de la lista de candidatas.
3. Este proceso se repite hasta que no quedan regiones candidatas.

En la siguiente imagen podemos ver el resultado de aplicar este algoritmo:

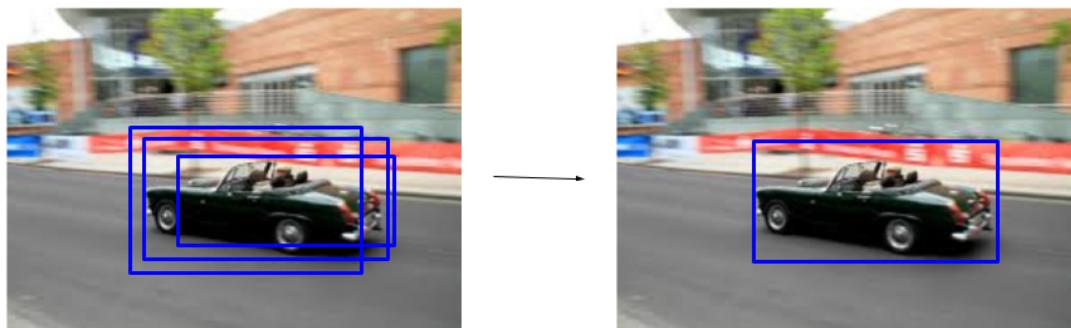


Figura 2.20. Resultado de aplicar NMS

Una vez visto el funcionamiento de una RPN, pasamos a explicar con más detalle el funcionamiento de una red Faster R-CNN.

En primer lugar, la imagen de entrada se introduce en una red neuronal convolucional para obtener el mapa de características y en otra de tipo RPN para detectar las regiones de interés. Para mejorar aún más la eficiencia, ambas redes comparten los mismos pesos. El paso siguiente, una vez que tenemos los resultados de ambas, es usar las regiones propuestas por la RPN para agrupar las características del mapa devuelto por la CNN. Este agrupamiento se realiza mediante *RoI pooling*, como explicamos en apartados anteriores. Una vez tenemos los datos con la dimensión fija que necesitamos, se pasan a las capas densas que se encargan de realizar la clasificación y la regresión de los valores de la caja delimitadora (estas capas densas son distintas a las que encontramos en la RPN). Visualmente, esta arquitectura es la siguiente:

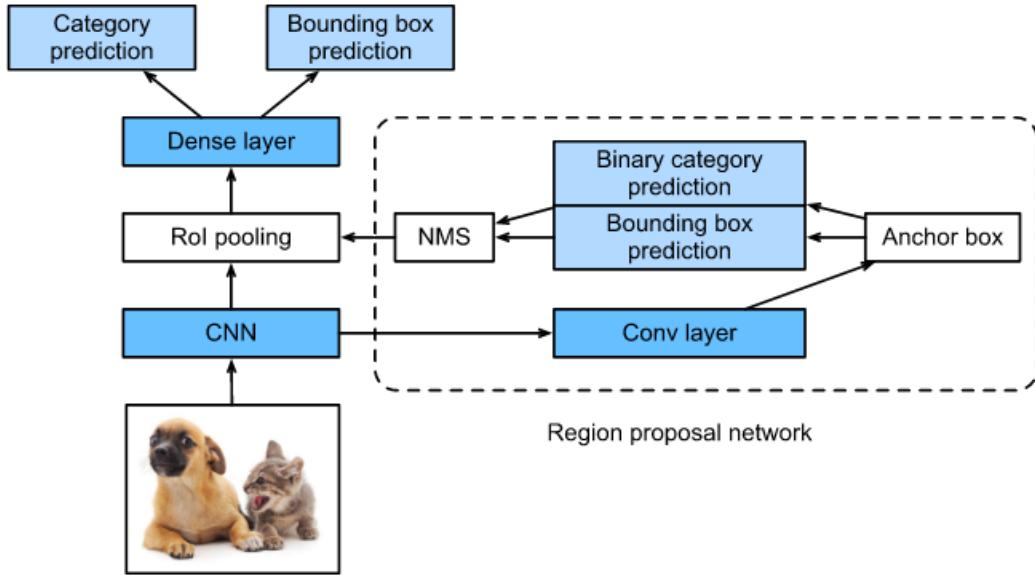


Figura 2.21. Arquitectura de una Faster R-CNN. Imagen extraída de [9]

Como ya comentamos al final del apartado 2.5., el algoritmo de tracking que hemos usado en este trabajo usa una red Faster R-CNN para detectar personas en un vídeo. Dicha red usa el modelo ResNet-101 junto con FPN (*Feature Pyramid Network*) como extractor de características. Una vez comentadas las redes Faster R-CNN y las residuales, nos faltan por estudiar las redes de pirámide de características (FPN), que vamos a explicar a continuación.

2.7. Feature Pyramid Networks (FPN)

La detección de objetos en escalas distintas es especialmente complicado en el caso de objetos pequeños. Para hacer frente a este problema, una posible solución es usar una pirámide sobre una imagen, donde cada nivel se corresponde con la imagen a una escala diferente. Sin embargo, tener que procesar en una red neuronal estas imágenes de distintas escalas requiere de mucho tiempo y memoria. Ante esto, en [12] propusieron usar una pirámide de características, en lugar de imágenes, aunque los mapas más cercanos a la imagen original están formados por estructuras de bajo nivel que no son muy efectivas para detectar objetos con precisión. Un FPN es un extractor de características de este tipo, diseñado para ser preciso y rápido y se usa, por ejemplo, en Faster R-CNN, generando mapas de características a distintas escalas, con mejor información que la que se obtiene con otros modelos.

La construcción de la pirámide incluye un camino de abajo a arriba (*bottom-up*), otro de arriba a abajo (*top-down*) y conexiones laterales. El primer flujo usa una red convolucional para extracción de características y, a medida que vamos subiendo, la resolución disminuye, a la vez que el valor semántico de cada capa aumenta. Con un *Single Shot Detector* (SSD) se realizan detecciones de los distintos mapas de características, excepto de los más cercanos a la imagen real, ya que están en alta resolución, pero no tienen un valor semántico lo suficientemente alto como para que compense la disminución

de velocidad del modelo. Por tanto, al usar únicamente las capas con menos resolución, el SSD no consigue buenos resultados en la detección de objetos pequeños. Para ello se introduce el segundo camino (*top-down*), que construye capas con una resolución más alta a partir de las capas con mayor valor semántico. Sin embargo, en estas capas reconstruidas, las ubicaciones de los objetos no son del todo precisas, para lo que se añaden conexiones laterales entre estas capas y los mapas de características correspondientes, que ayudan al detector a predecir las ubicaciones mejor. Además, estas conexiones se comportan como “atajos” que hacen el entrenamiento más sencillo, al igual que sucede en las ResNet. Esta es, por tanto, la estructura de una FPN:

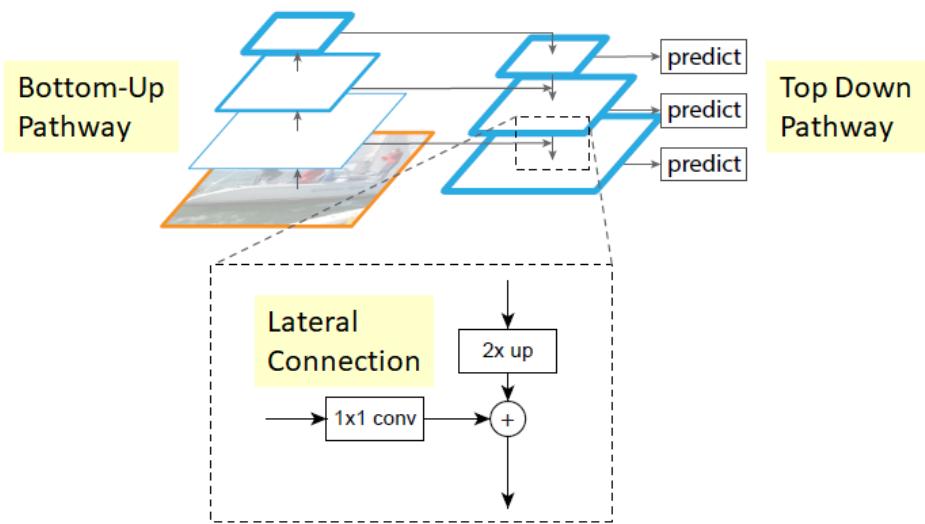


Figura 2.22. Esquema del modelo FPN. Imagen extraída de [13]

En el camino ascendente se usa una red residual y está compuesto por varios módulos de convolución, cada uno de los cuales engloba varias capas convolucionales. A medida que vamos avanzando hacia arriba, la dimensión espacial se reduce a la mitad y la salida de cada módulo de convolución se usa, posteriormente, en la ruta descendente.

En la ruta *top-down* se aplica una convolución de 1x1 para reducir la dimensión de la salida del último módulo de convolución del flujo ascendente y crear el primer mapa de características que se usará para la detección de objetos. Conforme vamos bajando, vamos aumentando la resolución espacial del mapa de características de la capa anterior por un factor de 2 y el mapa resultante se integra con el mapa correspondiente de la ruta ascendente, sumando sus elementos. A dicho mapa de la otra ruta se le aplica una convolución de 1x1 y a cada mapa resultante de la “mezcla”, se le aplica una de 3x3 para reducir el solapamiento que se produce al combinar las capas.

Este proceso se repite hasta llegar a la penúltima capa, ya que, como indicamos antes, el mapa más cercano a la imagen original requiere demasiado cómputo y ralentizaría el proceso. Podemos ver un esquema de su funcionamiento en esta imagen:

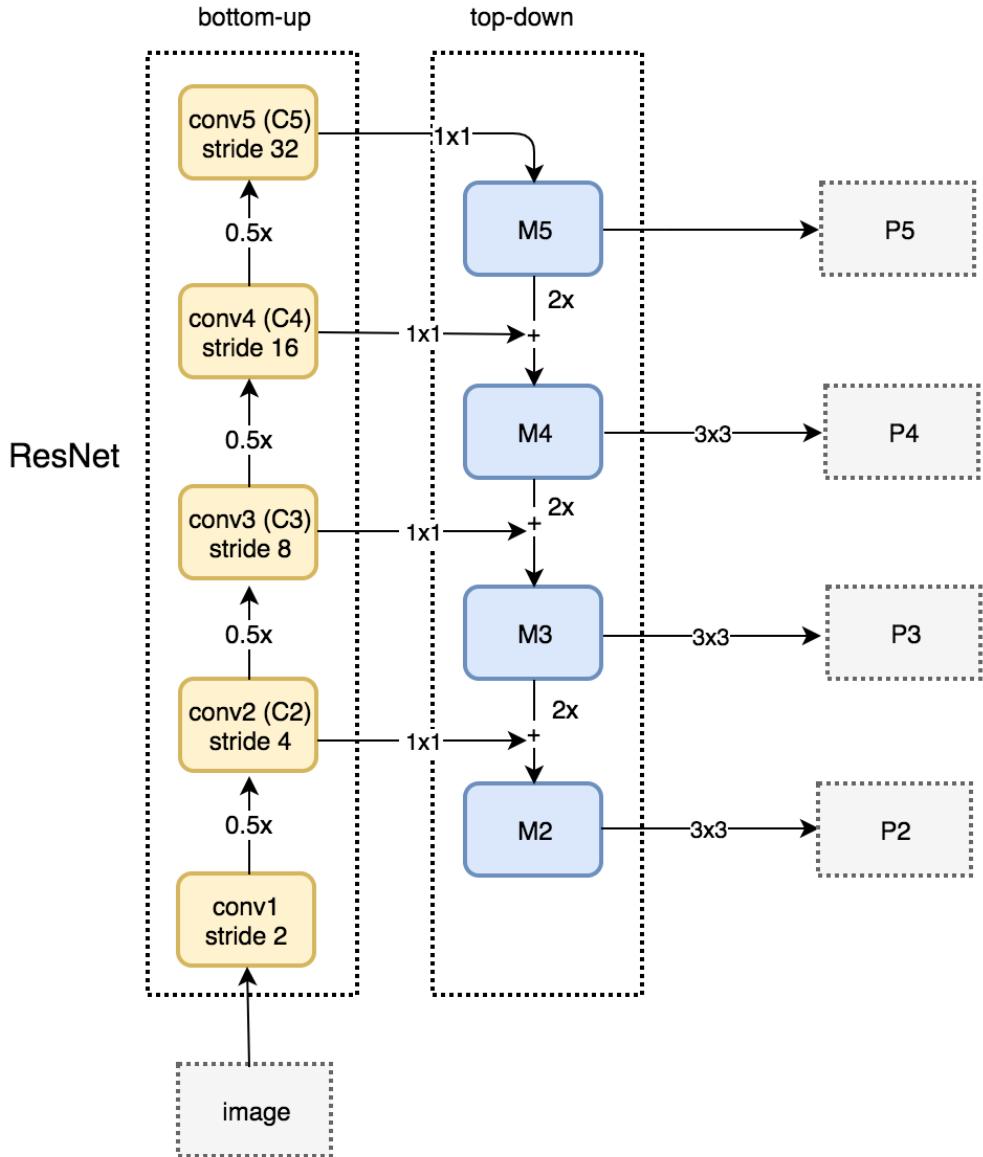


Figura 2.23. Funcionamiento de un modelo FPN. Imagen extraída de [14]

Como se ha indicado al principio de este apartado, FPN es un extractor de características, no un detector de objetos; por tanto, es necesario encadenar su salida con otro modelo para llevar a cabo esta tarea. Podemos, por ejemplo, introducir la salida de la FPN en un detector RPN, de manera que, en lugar de evaluar un solo mapa de características de escala única y realizar una clasificación y regresión para cada punto de este, siendo necesario tener “anclas” de distintas escalas, se cambia este mapa por la FPN. De esta forma, ya no se necesitan esas “anclas” de distintos tamaños, sino que a cada escala se le asigna un tamaño fijo, aunque en cada nivel sí se usan distintos ratios de aspecto (1:2, 1:1, 2:1). Así, la cabeza de la RPN (el conjunto de la última convolución de 3x3, seguida de las de 1x1 para clasificación y regresión) se aplica a los mapas de distinta escala.

La RPN era el extractor de regiones de interés que se usaba en las redes Faster R-CNN, por lo que, al cambiar su implementación haciendo uso del FPN, también podemos ver cambios en las FR-CNN.

Como hemos explicado antes, en la implementación original de la Faster R-CNN, cuando tenemos las regiones de interés y el mapa de características, se obtienen “parches” de características que pasan por una capa de *RoI pooling* para cambiar su tamaño. Al cambiar la RPN e introducir el FPN, para cada región de interés propuesta, podemos seleccionar el mapa que tenga la escala más adecuada para extraer esos “parches” que le pasamos a la capa de *pooling*. Así, una vez que hemos seleccionado el mapa más apropiado para una región, aplicamos el *RoI pooling* y pasamos la salida a las capas densas de clasificación y regresión. A continuación, podemos ver dos imágenes comparativas: la primera de ellas se corresponde con la implementación original de una Faster R-CNN y la segunda, con la implementación que usa FPN.

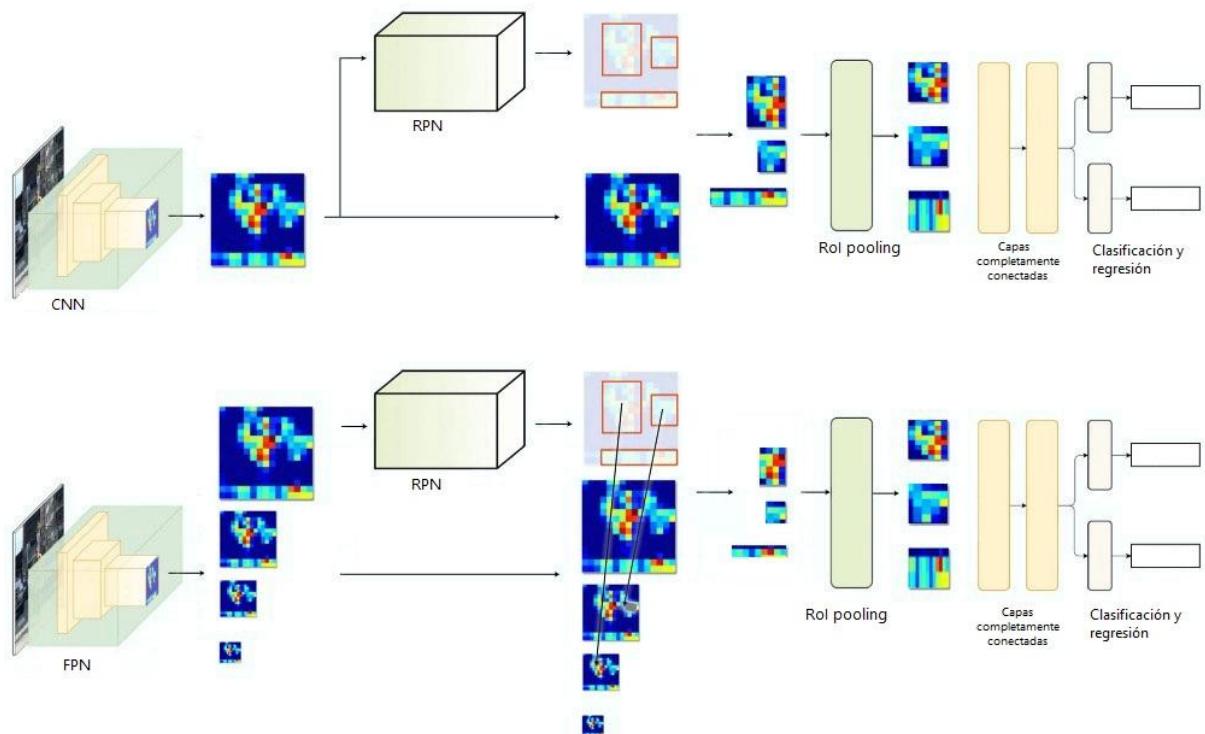


Figura 2.24. (Arriba) Faster R-CNN estándar. (Abajo) Faster R-CNN con FPN.
Imágenes extraídas de [14]

2.8. Redes neuronales siamesas (SNN)

Las redes siamesas fueron introducidas por Bromley y LeCun en [15] para verificar firmas mediante coincidencia de imágenes. Este tipo de redes está diseñado como dos o más redes neuronales iguales (gemelas) conectadas en una capa final que calcula la distancia entre las salidas generadas por estas redes y predice si los datos proporcionados como entrada (desde datos numéricos a imágenes) son iguales o no. Las redes que

componen este modelo comparten todos los pesos y sesgos, de ahí el nombre de gemelas. Estos parámetros compartidos hacen que dichas redes sean simétricas, lo cual hace que la red siamesa sea invariante a la conmutación de imágenes y, además, reducen considerablemente el tiempo de entrenamiento.

La última capa, que compara la salida de las redes gemelas, lleva la distancia entre las entradas a 0 o 1 según su similitud, para lo cual se usa una función llamada “contrastive loss”, que fue introducida por LeCun en [16].

Un ejemplo de estas redes es TriNet, que fue presentada en [17], en la cual se usa la arquitectura ResNet-50 preentrenada con los pesos propuestos en [6]. TriNet se usa para la reidentificación de personas y está compuesta por tres redes: a dos de ellas se les pasa como entrada una imagen del mismo individuo y a la tercera, una imagen de otra persona. Se usa como función de pérdida la conocida como *Triplet loss*, donde se fuerza a que la distancia entre las imágenes de la misma persona sea pequeña, lo más cercana a 0 posible, a la vez que la distancia de estas con respecto a la tercera imagen sea próxima a 1. Es precisamente esta arquitectura la que se utiliza para la reidentificación de personas en el algoritmo de *tracking* que usamos para la experimentación del trabajo.

3. Metodología

En esta sección nos centraremos en la detección de personas que tienen una trayectoria anómala en cuanto a velocidad o dirección respecto al comportamiento observado como normal. En cuanto al contenido del apartado, la distribución es la siguiente. En primer lugar, veremos el conjunto de datos que hemos usado para la experimentación. Después, estudiaremos en detalle el algoritmo de *tracking* basado en *deep learning* que hemos usado para realizar el seguimiento de personas y extraer sus trayectorias y el algoritmo del que hemos partido para clasificar dichas trayectorias como normales o anómalas. Por último, explicaremos el algoritmo desarrollado y veremos los resultados obtenidos en la experimentación.

3.1. Conjunto de datos: *UCSD Anomaly Detection Dataset*

Este conjunto de datos está disponible en [18]. Dicho conjunto de datos fue diseñado para experimentar y comprobar el rendimiento de un modelo de detección de anomalías presentado en [19] por un grupo de investigadores de la Universidad de San Diego. Los vídeos que conforman el dataset fueron grabados con una cámara en posición estática y se dividieron en dos subconjuntos, cada uno correspondiente a una toma distinta. En ambos subconjuntos se separaron los vídeos en varios de menor duración, alrededor de 200 fotogramas cada uno.

El primer subconjunto, *Peds1*, contiene vídeos de gente caminando que se acerca o se aleja de la cámara. En concreto, hay 34 vídeos de entrenamiento y 36 de test. Por otro lado, en el segundo, *Peds2*, los peatones se mueven de forma paralela a la cámara y nos encontramos con 16 vídeos para entrenamiento y 12 para test. Algunos ejemplos de anomalías presentes en estos vídeos son carritos de golf u otros vehículos motorizados, personas en monopatín o ciclistas.

Este conjunto de datos que hemos comentado es el que podemos descargar de [18], sin embargo, no es con este con el que hemos realizado los experimentos, ya que los fotogramas proporcionados tienen una resolución demasiado baja. Localizamos el correo electrónico de dos de los autores de [19] y les escribimos solicitando los vídeos originales, en lugar de los fotogramas del dataset, y, en un breve período de tiempo después de contactarles, nos dieron acceso a ellos. Estos vídeos originales tienen una resolución de 720x480 (30 fps) frente a la resolución de 238x158 (10 fps) del dataset, y encontramos 6 vídeos del subconjunto *Peds1* y 6 de *Peds2*, cada uno con una duración aproximada de 9 minutos y medio.

A partir de los vídeos originales del subconjunto *Peds1*, construimos nuestro propio conjunto de datos de la siguiente manera. En primer lugar, dividimos cada uno de los vídeos en sus fotogramas y nos quedamos con un tercio de ellos para transformarlos de 30 fps a

10 fps. Para ello, teniendo la secuencia completa numerada, seleccionamos los fotogramas que ocupaban una posición múltiplo de 3. A continuación, subdividimos el conjunto de frames resultante de cada vídeo en conjuntos más pequeños de 200 fotogramas, excepto el último, que tiene una extensión menor, y generamos los vídeos correspondientes a 60 fps. El hecho de montar los vídeos a una velocidad 6 veces mayor de lo normal fue, simplemente, para poder verlos en un período menor de tiempo, ya que resultaron alrededor de 330 vídeos, que a 10 fps habrían supuesto casi 2 horas, mientras que a 60 fps no llegaban a los 30 minutos.

Después, vimos cada uno de los vídeos para detectar las posibles anomalías en cada uno y clasificarlos como datos de entrenamiento o test. En nuestro caso, decidimos quedarnos con 50 vídeos en cada parte, por tanto, entre esos 330 que teníamos, elegimos 50 en los que no había ningún comportamiento anómalo para el conjunto de entrenamiento y otros 50 donde había anomalías, como personas en monopatín o bicicleta, gente quieta o corriendo y con direcciones distintas de la predominante, para el conjunto de test.

Además, para poder comprobar la bondad del algoritmo de detección de anomalías, en los vídeos de test se etiquetaron los comportamientos anómalos con la herramienta *ViTBAT*, publicada en [20].

Por tanto, el conjunto de datos final con el que hemos realizado la experimentación se compone de 50 vídeos de entrenamiento sin anomalías y 50 vídeos de test, en la mayoría de los cuales hay algún comportamiento anómalo, todos ellos con una resolución de 720x480 y una duración de 20 segundos, a 10 fotogramas por segundo, y obtenidos a partir de los vídeos originales correspondientes al conjunto *Peds1*. A continuación, dos fotogramas del conjunto, el primero, del conjunto de entrenamiento (sin anomalías), y el segundo, del conjunto de test (podemos ver un ciclista y una persona en monopatín, así como dos personas quietas):



Figura 3.1. (Izquierda) Fotograma del conjunto de entrenamiento.
(Derecha) Fotograma del conjunto de test

3.2. Algoritmo de *tracking*: *Tracking without bells and whistles*

La detección y seguimiento de personas en vídeo es una de las áreas más investigadas en el ámbito de la visión por computador y, en concreto, el paradigma más usado para afrontar esta tarea es el de seguimiento por detección (*tracking-by-detection*). Este enfoque divide el problema en dos subtareas: detectar ubicaciones de objetos independientemente en cada fotograma y formar trayectorias enlazando las detecciones correspondientes a lo largo del tiempo. Esta última etapa de asociación es muy difícil en sí misma, ya que hay que hacer frente a detecciones falsas, occlusiones e interacciones en escenas con muchas personas. Para abordar estos problemas se han ido desarrollando modelos cada vez más complejos, que se han presentado en la competición MOT (*The Multiple Object Tracking Benchmark*) año tras año. Frente a esto, en este algoritmo, presentado el año pasado en [21], usan únicamente un método de detección basado en *deep learning* para realizar todo el seguimiento, demostrando que se pueden obtener resultados similares, o incluso mejores, a los de los modelos más complejos con una red neuronal entrenada solo para la detección.

El seguimiento de múltiples objetos implica extraer las posiciones temporales y espaciales (trayectoria) de cada uno de los n objetos dada una secuencia de fotogramas de un vídeo. Dicha trayectoria se define como una lista ordenada de cuadros delimitadores: $T_n = \{b_{t_1}^n, b_{t_2}^n, \dots\}$, donde cada uno de esos cuadros delimitadores viene definido por 4 valores $b_t^n = (x, y, w, h)$, siendo x e y las coordenadas de la esquina superior izquierda y w y h el ancho y alto, respectivamente, y t un frame dentro del vídeo. El conjunto de cuadros delimitadores de un fotograma t se define como $B_t = \{b_t^{n_1}, b_t^{n_2}, \dots\}$.

En algunos algoritmos de detección basados en redes neuronales convolucionales, como en [11], se usa la regresión para mejorar la precisión de las cajas delimitadoras y en este algoritmo, los autores proponen explotar dicha regresión para realizar el seguimiento. Es decir, en vez de tener una parte de detección y otra de seguimiento, fusionan ambas en una sola, a la que denominan *Tracktor* (*tracker + detector*). Esto presenta dos ventajas frente a otros modelos: no es necesario un entrenamiento específico para esta tarea y se puede usar online, ya que no se realiza ninguna optimización compleja durante el test.

Como comentamos en el capítulo anterior, el elemento central del flujo de *tracking* del algoritmo es una red Faster R-CNN con ResNet-101 y FPN. En este caso concreto, la cabeza de clasificación de la red devuelve la probabilidad de que en la región propuesta haya una persona y el método aprovecha esta clasificación y la regresión para realizar el seguimiento de varias personas. En cada fotograma se suceden dos pasos: regresión de los cuadros delimitadores existentes e inicialización de cuadros nuevos. A continuación, explicamos con detalle el funcionamiento de este algoritmo.

3.2.1. Funcionamiento del algoritmo

El esquema de funcionamiento de este algoritmo es el siguiente:

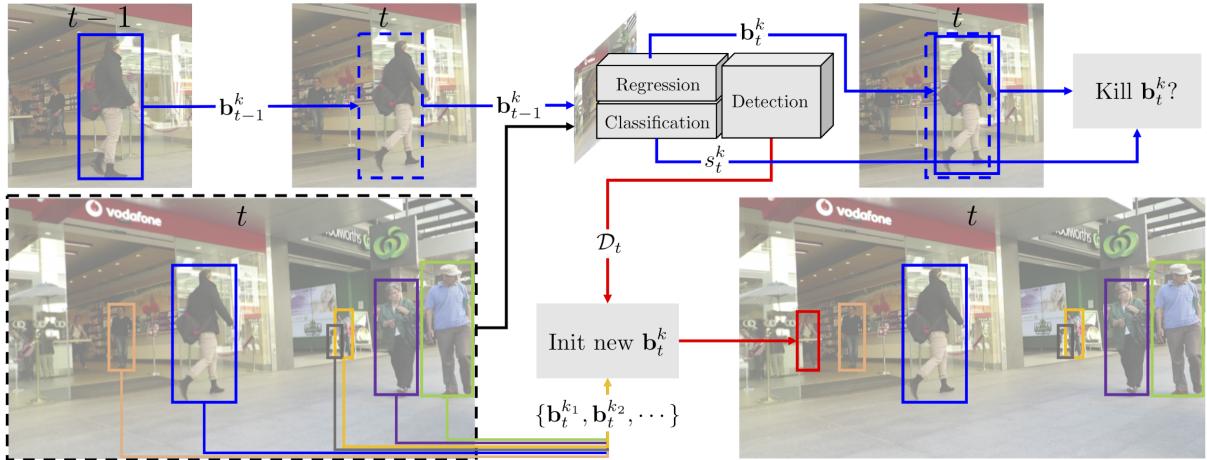


Figura 3.2. Esquema del algoritmo *Tracking without bells and whistles*.

Imagen extraída de [21, p. 2]

Como comentábamos, el algoritmo realiza dos pasos. El primero de ellos, la regresión de los cuadros delimitadores es el que aparece marcado en la imagen con el flujo de líneas azules. En esta etapa, la cabeza de regresión alinea los cuadros b_{t-1}^n del frame $t - 1$ de las trayectorias existentes a la nueva posición del objeto en el fotograma t . En el caso de la Faster R-CNN, esto se corresponde con aplicar *RoI pooling* sobre el mapa de características del frame actual (t) usando las coordenadas de los cuadros delimitadores del anterior ($t - 1$). Esto se hace bajo la suposición de que un determinado objetivo se habrá desplazado mínimamente entre frames, lo que suele estar garantizado en vídeos donde el número de fotogramas por segundo es alto. De esta forma, la identidad de la persona se transfiere automáticamente de la caja delimitadora anterior a la obtenida por regresión, creándose así una trayectoria al repetirse este paso para todos los frames.

En esta implementación, sin embargo, la etapa de *RoI pooling* se ha sustituido por otro tipo de *pooling*, llamado *crop and resize*, como propusieron Huang y col. en [22]. Esta operación extrae fragmentos de la imagen de entrada y los redimensiona usando interpolación bilineal o de vecino más cercano. De esta manera, junto con el uso de FPN, se consigue una mejora en la detección de objetos pequeños.

Una vez hecha la regresión, se estudian las distintas trayectorias para ver si hay que “desactivar” alguna, teniendo en cuenta la puntuación que genera la cabeza de clasificación: un objetivo que abandona la escena o es tapado por otro objeto se deja de seguir si su nueva puntuación de clasificación s_t^n está por debajo de un valor σ_{active} . Las occlusiones o solapamiento entre objetos se manejan aplicando el algoritmo NMS (que vimos en el

apartado de las redes Faster R-CNN) al conjunto de cuadros delimitadores del frame actual (B_t), siendo el umbral para el valor de IoU, λ_{active} .

Cuando se comprueban las trayectorias y se eliminan las que correspondan, pasamos al segundo paso de procesamiento, la detección de objetivos e inicialización de trayectorias nuevas, que aparece en el diagrama marcado con líneas rojas. En esta etapa, el detector genera un conjunto de detecciones D_t para el frame actual, t , y cualquiera de estas detecciones inicia una nueva trayectoria únicamente si el valor IoU de dicha detección con cualquiera de las trayectorias ya existentes, b_t^n , es más pequeño que el umbral λ_{new} . Es decir, se crea una trayectoria nueva si se detecta un objeto potencialmente nuevo, que no está cubierto por ninguna de las trayectorias que tenemos.

Respecto a los umbrales que hemos mencionado, σ_{active} , λ_{active} , λ_{new} , los valores que se usan son 0.5, 0.6 y 0.3, respectivamente. El hecho de que λ_{active} sea mayor que λ_{new} aumenta la estabilidad en el *tracking*, ya que hay menos trayectorias activas que se desechan debido al algoritmo NMS y se inician menos trayectorias nuevas.

Este es el comportamiento general del algoritmo, sin embargo, el método cuenta con dos extensiones que tienen como objetivo mejorar la preservación de la identidad a través de los fotogramas. Dichas extensiones son un modelo de movimiento y un algoritmo de reidentificación.

- **Modelo de movimiento.** Como hemos mencionado antes, durante la regresión de los cuadros delimitadores se supone que la posición de un objeto varía mínimamente entre dos fotogramas consecutivos, sin embargo, esto no es cierto en dos escenarios: vídeos con pocos frames por segundo o en los que hay movimientos grandes de la cámara. En estos casos (extremos), una caja delimitadora de un frame $t - 1$ podría no contener al objetivo en t . Para ello se introducen dos modelos de movimiento que mejoran la posición del cuadro delimitador en fotogramas futuros. Para secuencias con cámara en movimiento, se aplica una compensación de movimiento de cámara (CMC) sencilla, que consiste en alinear frames usando la maximización del coeficiente de correlación mejorada (ECC), como se propuso en [23]. Dicha maximización consiste en estimar la transformación geométrica entre dos imágenes y devolver la imagen de entrada transformada que se asimile más a la otra. En el caso de secuencias con un número bajo de frames por segundo, se aplica un supuesto de velocidad constante (CVA) para todos los objetos.
- **Reidentificación.** Para poder usar el algoritmo online, se propone una reidentificación a corto plazo basada en vectores de apariencia generados por una red neuronal siamesa. Con este fin, se almacenan las trayectorias desactivadas en su versión previa a la regresión (b_{t-1}^n) para un número prefijado de fotogramas, F_{reID} , y, posteriormente, se compara la distancia entre los cuadros delimitadores de las trayectorias desactivadas y las nuevas y se reidentifica una trayectoria de acuerdo a

un umbral. Para evitar muchas reidentificaciones incorrectas, únicamente se consideran pares de trayectorias nuevas y desactivadas con un valor IoU suficientemente grande. Como comentamos en apartados anteriores, para esta tarea se usa la arquitectura TriNet basada en ResNet-50.

Al detector completo, es decir, incluyendo estas dos extensiones, lo denominan *Tracktor++*, y es este el que se utiliza en la práctica.

Este algoritmo aborda la detección y seguimiento de personas, una parte crucial a la hora de detectar anomalías en las trayectorias de la gente, de forma más sencilla y ligera computacionalmente que otros modelos más complejos y, además, consigue buenos resultados, incluso mejores que algunos estos modelos complejos. Asimismo, su implementación es pública y el código (escrito en Python) puede encontrarse en GitHub, en la dirección https://github.com/phil-bergmann/tracking_wo_bnw, por lo que decidimos usar este método para realizar el *tracking* y obtener las trayectorias.

Para usar este algoritmo con nuestro dataset, seguimos las instrucciones que proponen para añadir conjuntos de datos:

1. Definir el nuevo conjunto de datos en el archivo *factory.py*, que encontramos en la ruta */src/tracktor/datasets/*.
2. Añadir una clase *x_sequence* (donde *x* es el nombre del dataset) en el mismo directorio que hemos mencionado, que se encarga de convertir los fotogramas que forman el conjunto de entrenamiento y/o test en objetos binarios que el algoritmo pueda manejar, así como de escribir los resultados con el formato MOT16/MOT17 (<frame>, <id>, <bb_left>, <bb_top>, <bb_width>, <bb_height>, <conf>, <x>, <y>, <z>).
3. Añadir una clase *x_wrapper* (donde *x* es el nombre del dataset) en la ruta anterior, donde se especifican las secuencias que pertenecen a entrenamiento y a test y se usa la clase anterior para adaptarlas.

Una vez hecho esto, en el archivo de configuración */experiments/cfgs/tracktor.yaml*, es necesario cambiar los valores de los atributos *module_name* (nombre del directorio de salida donde se almacenan los resultados) y *dataset* (nombre del conjunto de datos que hemos especificado en el archivo */src/tracktor/datasets/factory.py*).

Como modificación adicional, en el archivo */experiments/scripts/test_tracktor.py*, el archivo de ejecución, hemos añadido un fragmento de código (líneas 122-137) que se encarga de unir todos los fotogramas con los cuadros delimitadores que devuelve el algoritmo y generar un vídeo a partir de ellos.

3.3. Método base de detección de trayectorias anómalas: *Towards Abnormal Trajectory and Event Detection in Video Surveillance*

En este apartado comentaremos el método de detección de comportamientos anormales original en el que nos hemos basado para desarrollar nuestro algoritmo.

El campo del análisis y detección de comportamientos extraños se ha estudiado mucho en los últimos años, debido a los avances que conllevaría, por ejemplo, en la videovigilancia. Existen en la literatura numerosos artículos referidos a este tema en los que se emplean métodos de aprendizaje tanto supervisado como no supervisado, así como sistemas diseñados manualmente para localizar ciertas anomalías específicas de un problema. Dentro de la categoría de métodos no supervisados, hay dos enfoques distintos: la detección de anomalías a nivel de trayectoria y la detección de anomalías a nivel de píxel. El primer enfoque se centra en localizar comportamientos anormales de acuerdo a su velocidad o dirección, mientras que el segundo hace un análisis más detallado de los movimientos del cuerpo, en lugar de fijarse en el desplazamiento de la persona. De esta forma, con el primer método podemos reconocer personas corriendo o que estén quietas, mientras que con el segundo se puede detectar a alguien que, aún llevando una velocidad y/o dirección normal, tiene algún comportamiento extraño, como puede ser un salto.

Los autores de [24] proponen un método híbrido en el que se usen técnicas basadas en trayectoria, así como a nivel de píxel, para poder detectar comportamientos anormales relacionados con velocidad y dirección y otros más complejos que tengan que ver con el movimiento de cada persona. A continuación, vemos el funcionamiento del algoritmo propuesto.

3.3.1. Funcionamiento del algoritmo

Al igual que para el algoritmo de *tracking*, introducimos un esquema que representa el funcionamiento de este algoritmo para usarlo como base para una explicación más detallada:

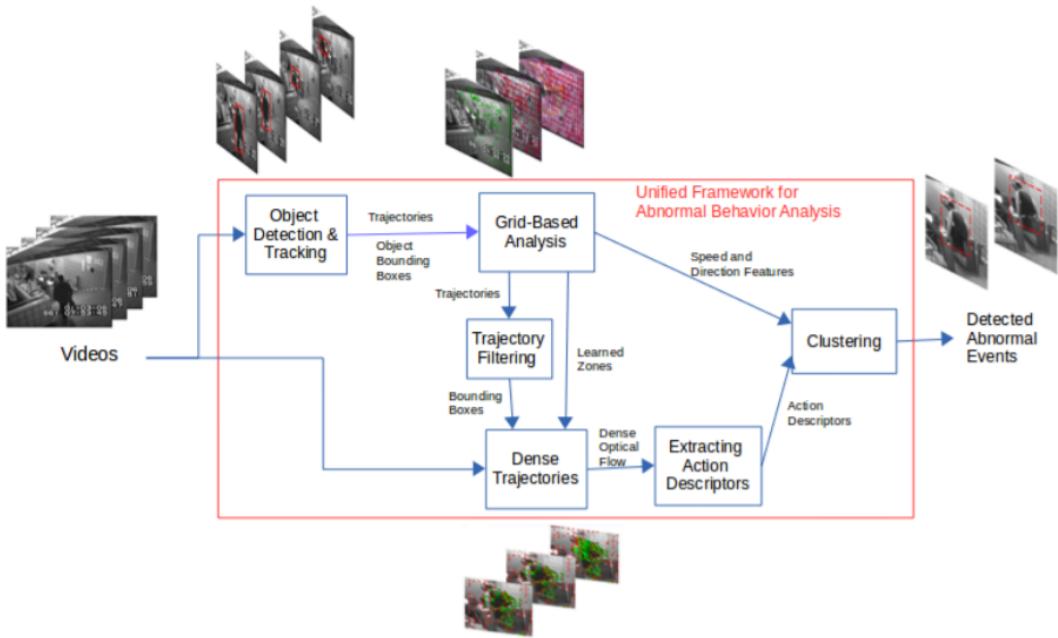


Figura 3.3. Esquema del algoritmo de detección de anomalías. Imagen extraída de [24, p. 4]

Según vemos en este diagrama, la entrada es un vídeo, del cual se extraen las trayectorias individuales y de grupo. Una vez que tenemos las trayectorias de cada individuo y/o grupo de personas (formadas por una serie de cuadros delimitadores), el método realiza un análisis para extraer, de forma automática, distintas zonas dentro de la escena, las cuales pueden tener tamaños y formas diversos, y calcula el patrón de velocidad y dirección de cada una de esas zonas. Con la información de dichas zonas, se realiza un filtrado de trayectorias, excluyendo aquellas que presentan una velocidad o dirección anómalas con respecto a la zona correspondiente. El siguiente paso, una vez que se han eliminado estas trayectorias, es el análisis a nivel de píxel. En esta etapa, como vemos en el esquema, se toman como entrada los cuadros delimitadores de los distintos individuos y la información de las zonas obtenida anteriormente y lo que se hace es extraer descriptores de acción dentro de la caja delimitadora de cada persona. De esta forma, se obtiene información sobre el movimiento corporal de cada objeto dentro de una zona. Finalmente, se aplica *clustering* a todas las características obtenidas de los distintos objetos, es decir, su velocidad, dirección y movimientos, para distinguir entre comportamientos normales y anormales dentro de la escena. Ahora explicaremos cada uno de estos pasos de manera más detallada.

Tracking individual y grupal

Como hemos mencionado, el algoritmo recibe como entrada un vídeo y lo primero que hace es extraer las trayectorias presentes en él, tanto individuales como grupales, para lo que usan los algoritmos propuestos en [25] y [26], respectivamente. En primer lugar, los objetos que están en primer plano se extraen y se calcula la similitud entre pares de objetos detectados en una serie de fotogramas (un número predefinido de ellos) usando descriptores como la distancia o el color. De esta forma, entre dos objetos se establece un enlace temporal si su valor de similitud está por encima de un umbral determinado. Para

disminuir la complejidad del algoritmo, para cada objeto solo se mide la similitud con los objetos vecinos que haya en un radio fijado de antemano. Así, teniendo una serie de estos enlaces temporales, se forman rutas o trayectorias para cada objeto y cada una de ellas va acompañada de una puntuación que depende de los enlaces que contiene. Por tanto, la trayectoria definitiva de un objeto concreto es aquella que tiene una puntuación más alta. Una vez que se obtienen las trayectorias individuales, se definen grupos con la idea de que un grupo son dos o más personas próximas espacial y temporalmente que se mueven con una velocidad y una dirección similares, y, a continuación, se pasa al análisis de las trayectorias. Como vemos, los métodos utilizados para extraer las trayectorias se basan en modelos clásicos en los que es necesaria cierta intervención manual.

Análisis de trayectorias

Esta etapa, que en [24] denominan *análisis basado en cuadrícula*, toma como entrada las trayectorias de cada objeto y lleva a cabo un análisis de ellas apoyándose en una cuadrícula. Dichas trayectorias se definen como una tupla $T_i = (tid_i, \{p_1, B_{p1}, \dots, p_n, B_{pn}\})$, donde tid_i es el identificador de la trayectoria y B_{pn} es el cuadro delimitador de un objeto en el punto $p_n = (x_n, y_n, t_n)$, siendo x e y las coordenadas de posición en la imagen y t , el instante de tiempo en el que se guarda esa posición. Hay tres pasos dentro de esta etapa:

1. *Snapping* de trayectorias (ajuste de trayectorias). El proceso de ajuste o *snapping* es una de las novedades que introduce este artículo y su objetivo es representar las trayectorias de forma más abstracta y resumida, lo cual permite encontrar zonas en la escena relacionadas con un patrón de movimiento específico, además de reducir la carga computacional del sistema.

En primer lugar se construye sobre la escena una cuadrícula en la que el tamaño de las celdas se corresponde con el del cuadro delimitador más pequeño de todas las trayectorias y, a continuación, cada punto de cada trayectoria se asigna al centroide de la celda en la que se encuentre. Podemos ver un ejemplo de este proceso en la siguiente imagen:



Figura 3.4. *Snapping* de una trayectoria

Como podemos ver, la trayectoria pasa por 3 celdas distintas y los puntos que están en cada celda, se colapsan en su centroide, lo que reduce considerablemente el número de puntos de las trayectorias y las simplifica, pero sin perderse el rumbo

original. Estas nuevas trayectorias son las denominadas trayectorias *snapped* (S_i) y de cada una se almacena la secuencia de puntos *snapped* (s_m) que la forman:

$$S_i = \{s_m\}_{m=1:M}$$

$$s_m = (cid_m, x_m, y_m, mint_m, maxt_m)$$

Respecto a cada punto *snapped*, s_m , se trata de una tupla formada por el identificador de la celda a la que pertenece el punto y las coordenadas x e y de su centroide y $mint_m$, $maxt_m$ son la marca de tiempo del primer y último punto consecutivos de la trayectoria T_i en la celda cid , respectivamente.

Además de esto, para cada celda se almacena una tupla C_j que incluye las coordenadas de su centroide (x e y), el número de puntos (normales) que se han asignado a él ($qtde$) y la velocidad media de todos los puntos de la celda ($avgS$). De esta forma, al procesarse la información de cada celda de forma individual, se mantienen las diferencias de movimiento entre las más cercanas a la cámara y las más alejadas. Después de hacer este ajuste sobre las trayectorias pasamos al segundo paso.

2. Descubrimiento automático de zonas. Esta parte es otra de las contribuciones de este artículo al área de la detección de anomalías y su objetivo es localizar regiones en la escena en las que haya un movimiento distintivo. Para ello se usa el algoritmo de *clustering* de propagación de afinidad sobre la información almacenada de las celdas de la cuadrícula. La elección de este método se debe a que en otros algoritmos en los que hay que fijar parámetros de antemano (como ocurre en *k-means* con el número de clusters), es necesario optimizar dichos parámetros. Tal como explicamos en el apartado 2.2.1., este algoritmo se basa en una matriz de similitudes que se calculan como la distancia euclídea entre dos puntos, pero con signo negativo. En este caso, cada uno de esos puntos es una tupla formada por la velocidad media de la celda y el número de puntos asignados a su centroide, que habíamos almacenado anteriormente ($C_j = (avgS_j, qtde_j)$). Usando estas características de densidad y velocidad pueden descubrir zonas en las que hay más o menos gente y se mueven más o menos rápido.

Cada una de las zonas descubiertas se puede escribir como una tupla $z_n = (R_n, \{S_n\}, Sspeed_n, Sdirection_n)$, donde R representa el polígono que engloba la zona, S es el conjunto de trayectorias *snapped* de la zona y $Sspeed$ y $Sdirection$ son las velocidad y dirección de cada una de las trayectorias de S . En cuanto a la velocidad, se obtiene a partir de los puntos *snapped* de una trayectoria dentro de la zona de la siguiente forma:

$$Sspeed_n = \frac{\sum_{i=1}^{N-1} \sqrt{(x_{i+1}-x_i)^2 + (y_{i+1}-y_i)^2}}{maxt_N - mint_1}$$

Es decir, para una determinada trayectoria *snapped* se calcula la distancia entre cada par de puntos y el valor total se divide entre la diferencia de tiempo entre el primero y el último. Respecto a la dirección, se toma la dirección global entre los puntos en los que la trayectoria entra y sale de la zona, pudiendo ser uno de los cuatro puntos cardinales (*Norte, Sur, Este, Oeste*). De esta manera, puede obtenerse el histograma de frecuencias de la dirección en cada zona. Una vez hecho esto, el paso final es detectar las anomalías.

3. Detección de anomalías basada en trayectorias. Una vez que se han detectado las distintas zonas, representando cada una de ellas un patrón de movimiento determinado, las trayectorias *snapped* de cada una se analizan para encontrar velocidades o direcciones anormales. Analizando las velocidades de cada una de las zonas ($Sspeed_n$) se observa que las velocidad de las trayectorias normales sigue una distribución normal, por lo que para localizar las anomalías se compara la velocidad de cada trayectoria con dos intervalos de confianza para ver si entra dentro de dicha distribución. Siendo μ_n la velocidad media de la zona, σ_n , la desviación típica y α , el nivel de confianza, una velocidad se considera normal si está dentro del intervalo siguiente:

$$(Sspeed_n > \mu_n + \alpha * \sigma_n) \vee (Sspeed_n < \mu_n - \alpha * \sigma_n)$$

En cuanto a las direcciones anómalas, primero se analiza si la dirección sigue una distribución uniforme dentro de las zonas. Para ello se usa la prueba chi cuadrado de Pearson:

$$\chi^2 = \sum_{b=1}^B \frac{(O_b - E_b)^2}{E_b}$$

Siendo O_b la frecuencia observada de una dirección y E_b , la frecuencia esperada en una distribución uniforme. Si el resultado es distinto de 0, la distribución no es uniforme y, por tanto, la dirección más frecuente será la predominante en la zona, aunque esto no siempre se cumple. Para las dos direcciones más frecuentes pueden darse tres casos: que no sean opuestas entre ellas; que sean opuestas, pero una de ellas tenga una probabilidad mayor; o que sean opuestas con la misma probabilidad. En el primer caso, dado que no son opuestas y ambas son importantes, se define la dirección dominante como la intermedia entre ellas; en el segundo caso se toma la dirección más frecuente, teniendo en cuenta que la otra dirección puede provocar falsas alarmas. En cuanto al tercer caso, al no haber una dirección más probable que otra, no se realiza análisis basado en dirección. Una trayectoria, por tanto, es anormal si se desvía 90° de la dirección principal de la zona en la que se encuentra. Es decir, las direcciones normales son aquellas dentro del intervalo siguiente, donde D_{z_n} es la velocidad principal de la zona z_n :

$$\left(Sdirection_n > D_{z_n} + 90^\circ \right) \vee \left(Sdirection_n < D_{z_n} - 90^\circ \right)$$

Una vez que se han detectado las anomalías relacionadas con velocidad y dirección, se eliminan las trayectorias correspondientes que ya hayan sido marcadas para eliminar ruido y se pasa al análisis a nivel de píxel.

Análisis a nivel de píxel

Como comentábamos al principio, en [24] introducen un método novedoso donde se combinan dos de las técnicas usadas en la detección de anomalías en el comportamiento. Como se ha indicado en apartados anteriores, nuestro algoritmo únicamente se centra en anomalías relativas a la dirección o velocidad de los individuos, por lo que no entraremos en detalle en la parte del método en la que se usan descriptores de acción para detectar comportamientos anormales a nivel de píxel.

3.4. Algoritmo desarrollado para la detección de anomalías basada en trayectorias

En este apartado comentaremos el algoritmo que hemos desarrollado, que, como hemos indicado anteriormente, está basado en la primera parte del método que acabamos de ver. El código implementado puede encontrarse en la dirección de GitHub <https://github.com/laurahernandezm/TFG>.

El esquema de funcionamiento de nuestro algoritmo es el siguiente:

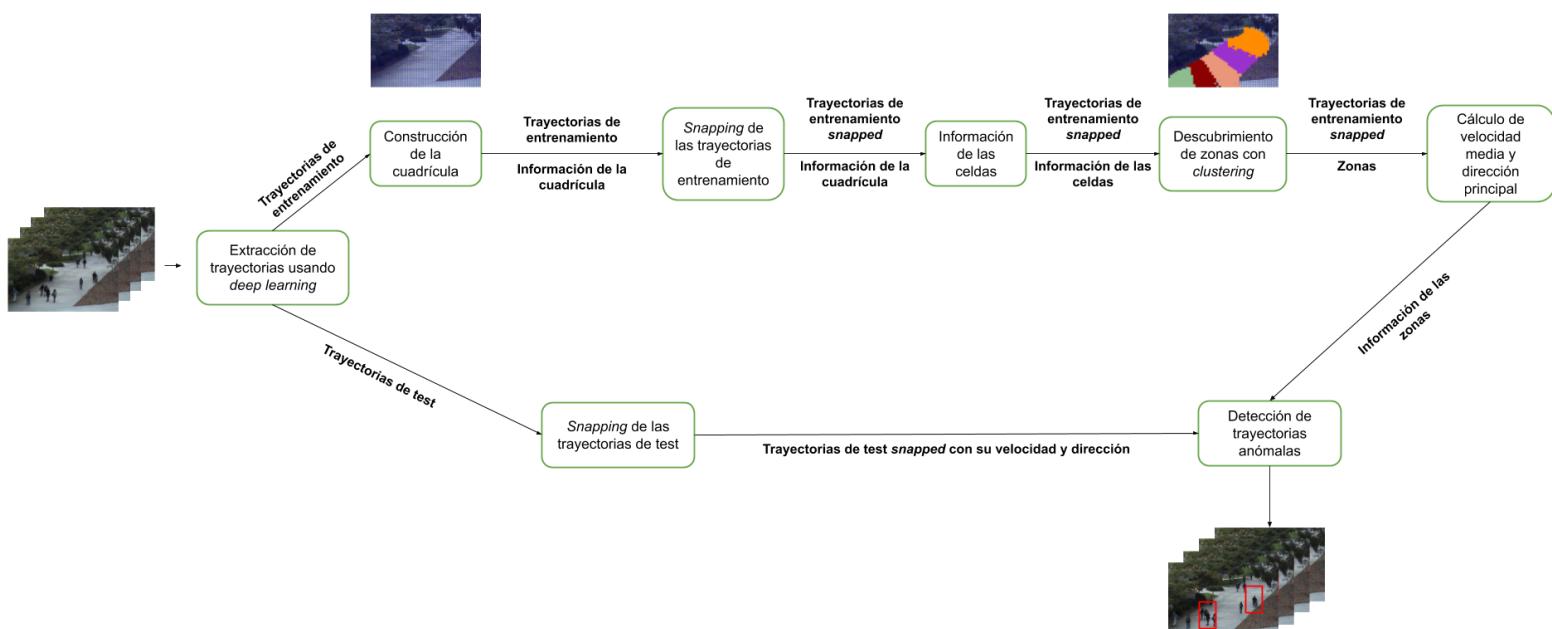


Figura 3.5. Diagrama del funcionamiento del algoritmo desarrollado

En primer lugar, en vez de recibir como entrada un vídeo y detectar trayectorias individuales y grupales, aplicamos el algoritmo de *tracking* basado en *deep learning* que hemos explicado en el apartado 3.2. para extraer las trayectorias individuales (no trabajamos a nivel de grupo) y es la salida de dicho algoritmo la que pasamos como entrada a este. Las trayectorias están en formato MOT, donde las coordenadas que tenemos del cuadro delimitador son las de la esquina superior izquierda. Las trayectorias que se leen deben estar ordenadas, primero, por número de trayectoria y, después, por número de frame. En nuestro caso, para ver la ruta real de cada persona, nos interesa un punto más cercano a los pies, por lo que, lo primero que hacemos es transformar el formato original cambiando las coordenadas de la esquina superior izquierda por las del punto medio del segmento inferior del cuadro delimitador. Visualmente:

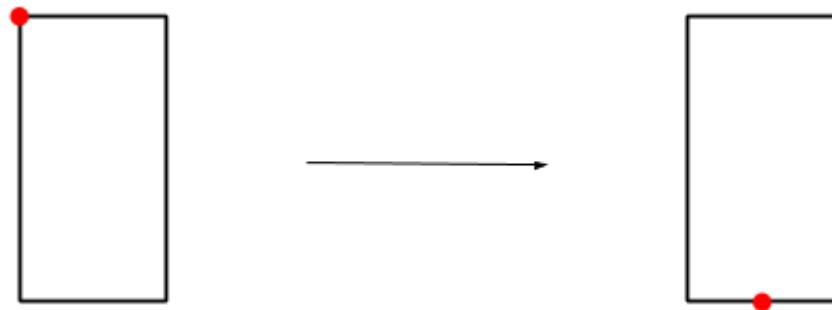


Figura 3.6. Transformación en los cuadros delimitadores

Adicionalmente, para mitigar el efecto negativo de las trayectorias rotas, hacemos un filtrado y nos quedamos solo con aquellas trayectorias que tengan una longitud mayor o igual a 20 fotogramas (dado que los vídeos con los que trabajamos están a 10 fps, esto equivale a permanecer en escena, como mínimo, 2 segundos). Una vez que tenemos las trayectorias representativas en el formato deseado, pasamos a construir la cuadrícula.

En [24] construyen la cuadrícula tomando como tamaño de celda el del cuadro delimitador más pequeño de todos los obtenidos. Sin embargo, en nuestro dataset la cámara está relativamente cerca de los peatones, por lo que las cajas delimitadoras son demasiado grandes. Por eso, decidimos usar un tamaño de celda prefijado de 15x10 píxeles, obteniendo así una cuadrícula de 48x48, ya que los vídeos con los que trabajamos tienen una resolución de 720x480:

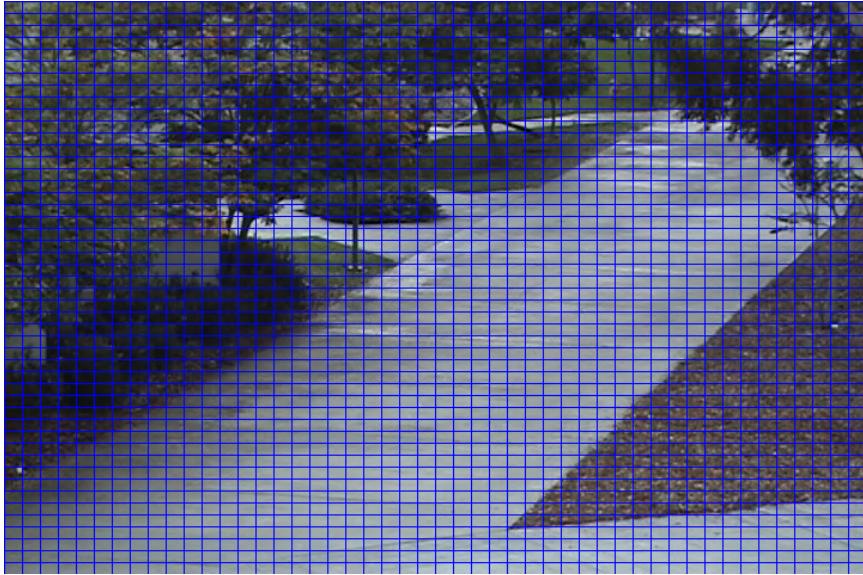


Figura 3.7. Cuadrícula de 48x48 sobre la que trabajamos

Una vez que tenemos la cuadrícula, pasamos al *snapping* de las trayectorias de entrenamiento de la siguiente forma. Para cada trayectoria comprobamos a qué celda pertenece cada uno de sus puntos y almacenamos una lista $[cid, x, y, mint, maxt, (x_{last}, y_{last}), dist, speed]$. Los cuatro primeros valores se corresponden con los que proponen en [24], mientras que (x_{last}, y_{last}) son las coordenadas del último punto consecutivo de la trayectoria asignado al centroide de la celda cid , $dist$ es la distancia euclídea acumulada entre los puntos consecutivos en la celda y $speed$ es la velocidad de la trayectoria en esa celda (tomando como distancia el valor de $dist$ y como tiempo, la diferencia entre $maxt$ y $mint$). A medida que se procesan puntos en una celda se van actualizando los valores de $maxt$, (x_{last}, y_{last}) , $dist$ y $speed$, así como el número de puntos asignados a ella, $qtdc$, en el resumen de información de las celdas.

Cuando tenemos todas las trayectorias en esta nueva forma calculamos y almacenamos la información de las celdas: $x, y, qtdc, avgS$. Para calcular $avgS$ vamos acumulando la velocidad que hemos calculado anteriormente para cada tramo de trayectoria en la celda correspondiente. Si dicho valor ($speed$) es distinto de 0, sumamos $1/speed$ y, en caso contrario, sumamos 0. Para calcular la velocidad media en una celda dividimos el número de puntos asignados a ella entre el valor acumulado de velocidad que hemos obtenido (si el acumulado es 0, la velocidad media también lo es). Se usa una media armónica, más adecuada a la hora de calcular medias de velocidades. Una vez hecho esto, eliminamos los valores (x_{last}, y_{last}) , $dist$ y $speed$ de los puntos *snapped* de todas las trayectorias. En la siguiente imagen podemos ver la distribución de las trayectorias de entrenamiento y las celdas a las que se asignan al realizar el *snapping*:

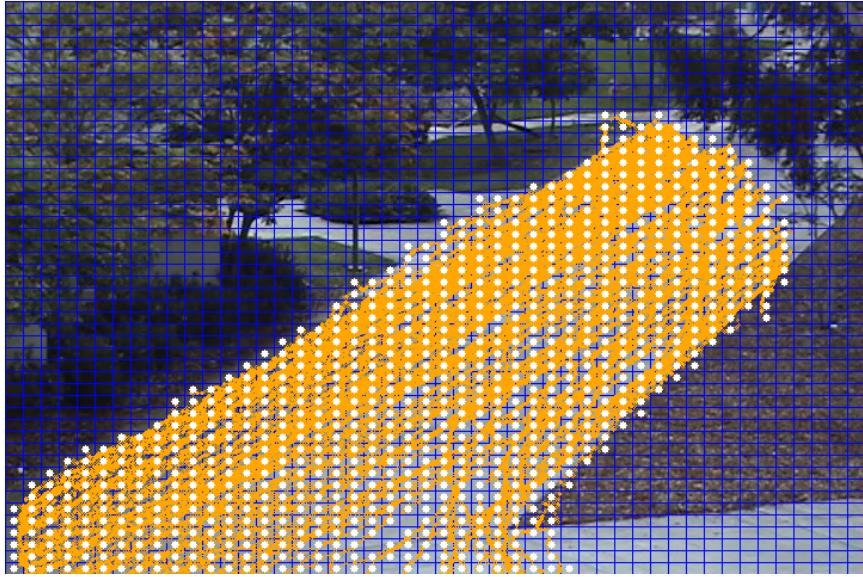


Figura 3.8. Trayectorias originales y celdas a las que se asignan durante el *snapping*

Teniendo las trayectorias simplificadas y la información de las celdas, pasamos al siguiente paso, el descubrimiento de zonas. En el método original proponen aplicar el algoritmo de propagación de afinidad sobre las características de densidad y velocidad de cada celda (*qtde* y *avgS*, respectivamente). Sin embargo, en nuestra implementación utilizamos también la información espacial, es decir, las coordenadas de los centroides, para forzar que las zonas sean lo más conexas posible. Además, asignamos un peso a cada valor, dándole más importancia a las coordenadas del centroide y a la velocidad media. Aplicamos el algoritmo de *clustering* de propagación de afinidad implementado en la biblioteca *scikit-learn* ([27]) cambiando algunos de los valores por defecto de los parámetros. En concreto, hacemos las siguientes modificaciones:

- **damping.** Este factor de amortiguación, que toma valores entre 0.5 y 1, representa la medida en la que un valor de la matriz de responsabilidad o disponibilidad se mantiene a la hora de actualizarlo. Tomando λ como este factor:

$$r_{t+1}(i,j) = \lambda * r_t(i,j) + (1 - \lambda) * r_{t+1}(i,j)$$

$$a_{t+1}(i,j) = \lambda * a_t(i,j) + (1 - \lambda) * a_{t+1}(i,j)$$

Es decir, en cada actualización se toma λ del valor anterior y se suma a $(1 - \lambda)$ del nuevo, de manera que hay una transición suave y no se producen oscilaciones grandes en la actualización de los valores (mensajes). El valor por defecto de este parámetro es 0.5 y nosotros lo hemos subido hasta 0.9 para ponderar más la información anterior y así obtener un número más reducido de clusters.

- **preference.** Tal como explicamos en el apartado 2.2.1, los valores de la diagonal de la matriz de similitud se denominan “preferencias” y dicho valor influye enormemente en el número de clusters resultantes: cuanto menor sea, menor será el número de clusters. El valor por defecto es la mediana de los valores de similitud.

En nuestro caso, lo hemos reducido considerablemente, hasta -300 (tras realizar varios experimentos con distintos valores).

- `max_iter`. Este parámetro hace referencia al número máximo de iteraciones del algoritmo. Hemos usado un valor 5 veces mayor al que viene por defecto, 1000 frente a las 200 originales. De esta forma, el algoritmo tiene más posibilidades de converger y formar clusters más compactos.

Como resultado de realizar este agrupamiento obtenemos qué celdas pertenecen a cada zona descubierta y las comprobamos para intentar fusionar aquellas zonas en las que haya “subregiones”. Es decir, para un mismo cluster puede haber una subzona en las celdas 3 y 4 y otra en las celdas 11 y 12, por tanto, si alguna de las casillas del primer subgrupo es vecina de alguna de las del segundo, ambos se fusionan. Tomando la siguiente imagen como ejemplo, las zonas marcadas con 1 pertenecen al mismo cluster, pero no están conectadas entre sí porque la unión entre sus celdas no es del tipo $celda_i \rightarrow celda_{i+1}$. Lo mismo sucede con las zonas marcadas con 2, sin embargo, entre las subregiones de la zona 1 podemos ver una conexión diagonal que entre las de la zona 2 no existe. Por ello, los dos subgrupos 1 se fusionarían en uno solo (tal como marca la línea roja), mientras que los subgrupos 2 permanecerían como están.

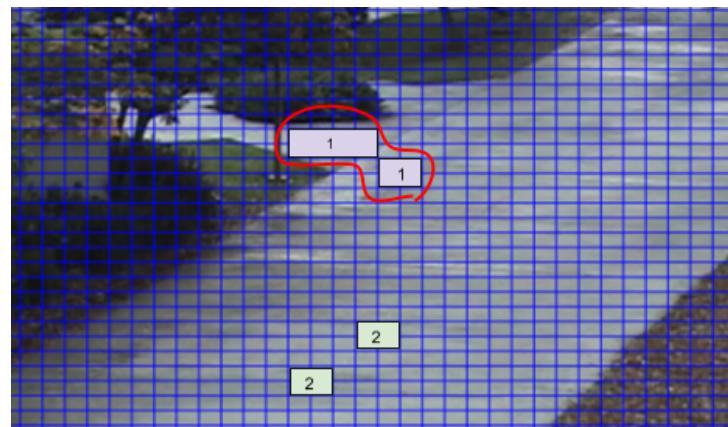


Figura 3.9. Fusión de zonas vecinas

Para realizar esta unión de grupos consideramos que una celda es “vecina” de otra si es una de las 9 casillas que se encuentra a su alrededor. Por ejemplo, el vecindario de la celda marcada con **X** son las casillas de color amarillo:

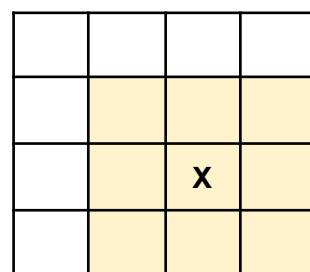


Figura 3.10. Vecindario de una celda

Al aplicar *clustering* sobre la información de las celdas ocupadas por las trayectorias de entrenamiento mostradas anteriormente obtenemos las siguientes zonas:

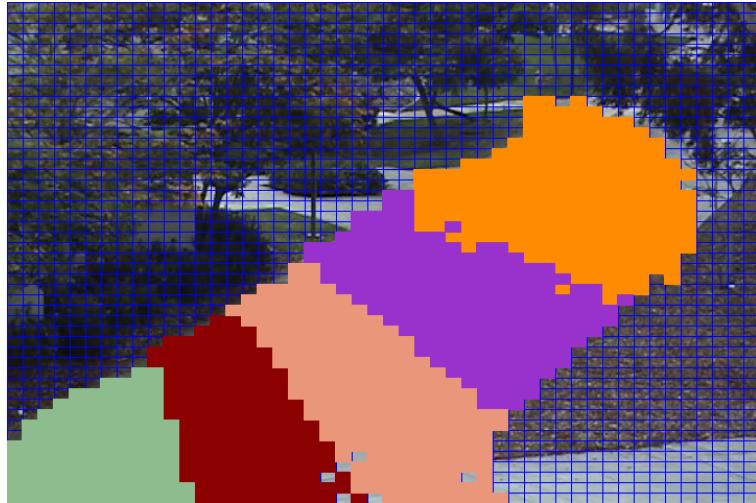


Figura 3.11. Zonas descubiertas

Como podemos ver, hemos obtenido 5 zonas bastante diferenciadas y conexas. La división por franjas que observamos se debe a la ponderación mayor en las coordenadas de los centroides y a la velocidad, ya que, según la distancia a la cámara, la velocidad observada es mayor o menor. Así, la zona inferior más a la izquierda, en tono verde, que está más cerca de la cámara, tiene una velocidad mayor que la zona superior derecha, en color naranja.

Una vez que hemos descubierto las distintas zonas de nuestra escena, pasamos a calcular la velocidad y dirección de las trayectorias dentro de cada una para poder calcular la velocidad media y dirección predominante. En primer lugar, a las zonas (representadas en este momento por las celdas que las componen, de forma equivalente al polígono R que proponían en el artículo) se les añade la lista de trayectorias *snapped* que contienen (el conjunto $\{S\}$) . Para ello vamos, zona por zona, comprobando la lista de trayectorias y anexamos los puntos *snapped* que pertenezcan a las celdas de cada zona. En el caso de las trayectorias de entrenamiento, los puntos *snapped* que se guardan son una lista $[cid, x, y, mint, maxt]$ como la tupla descrita en [24], mientras que si la trayectoria es de test, la lista se amplía con el identificador de la trayectoria ($[trid, cid, x, y, mint, maxt]$). Además, para intentar reducir los problemas generados por trayectorias rotas o cambios de identificador entre cajas delimitadoras, cuando estamos tratando trayectorias de test, únicamente asignamos una trayectoria a una zona si la duración de esta en dicha zona es igual o superior a 2 segundos (20 fotogramas).

A continuación, calculamos la velocidad de cada trayectoria *snapped*. Para ello, vamos, zona por zona, acumulando la distancia euclídea entre cada par de puntos *snapped* consecutivos de una trayectoria y, después, dividimos esta distancia total entre la diferencia de frames del primer y último puntos *snapped* (diferencia entre el instante en el que la trayectoria entra en una celda y sale de ella). En el caso de que esta diferencia sea 0

(porque una trayectoria toca un punto únicamente durante un frame), la velocidad se iguala a la distancia total acumulada. Al calcular la distancia euclídea introducimos pesos para darle más importancia al término y sobre el x e intentar compensar que la distancia recorrida horizontalmente es mayor que la recorrida en vertical. Esta información de velocidad se añade a la que ya teníamos de las zonas ($[R, \{S\}]$, respectivamente); en el caso de las trayectorias de test se añade una lista que contiene el identificador de trayectoria y su velocidad, mientras que para entrenamiento únicamente se almacena la velocidad. El hecho de almacenar los números de trayectoria durante el test es, simplemente, para facilitar su identificación si se detecta como anómala.

El siguiente paso es averiguar la dirección de cada trayectoria y añadir dicha información a la que tenemos hasta ahora sobre las zonas ($[R, \{S\}, Sspeed]$). Tal como explican en el artículo, para calcular la dirección hay que comparar los puntos en los que una trayectoria entra y sale de una zona. Para esto es necesario encontrar los parámetros a y b de la función lineal $y = ax + b$ que incluye dichos puntos, es decir, la pendiente y el término independiente. La pendiente podemos calcularla restando la coordenada y del último punto *snapped* menos la del primero y el término independiente, de la misma forma, pero con la coordenada x . Los autores del método definen cuatro direcciones posibles, que se corresponden con los cuatro puntos cardinales. En nuestro caso, hemos decidido afinar un poco más y definir 9 direcciones distintas: *Norte, Noreste, Este, Sureste, Sur, Suroeste, Oeste, Noroeste, Nula*. Consideramos que una trayectoria tiene dirección *Nula* si tanto la pendiente como el término independiente son 0, es decir, en la trayectoria no ha habido desplazamiento alguno o, tras haberse movido, el individuo ha vuelto al punto inicial. Al igual que antes, para las trayectorias de test guardamos tanto el identificador como la dirección, mientras que para entrenamiento únicamente almacenamos esta última.

Teniendo las zonas completas $[R, \{S\}, Sspeed, Sdirection]$, pasamos a calcular la media y la desviación típica de las velocidades, ya que consideramos que se distribuyen de forma normal. Simplemente recorremos las zonas y calculamos estos valores con las funciones `mean` ([28]) y `std` ([29]) de la biblioteca `numpy`, y los añadimos a la información de las zonas para poder construir los intervalos de confianza necesarios para hacer la detección de anomalías.

A continuación, averiguamos la dirección dominante dentro de cada zona. Primero obtenemos la frecuencia de cada una de las direcciones posibles dentro de cada zona y sobre eso aplicamos la prueba chi cuadrado de Pearson. Si la prueba da como resultado 0 significa que la distribución de direcciones es uniforme, por tanto no se almacena ninguna dirección ni se realiza análisis en esa zona. Si no, comprobamos las dos direcciones más frecuentes según el criterio seguido en el artículo, excepto para las direcciones opuestas:

- Si hay una única dirección se toma esa como la principal.
- Si hay dos direcciones no opuestas, se toma como principal la composición de ambas (16 direcciones posibles).

- Si hay dos opuestas y la diferencia de frecuencias es menor que la mitad del valor de la mayor, se toman ambas direcciones como principales, mientras que, si esta diferencia es mayor, se toma la más frecuente como principal.
- En cuanto a la dirección *Nula*, introducida por nosotros, si es la primera dirección más frecuente se toma como dirección principal, mientras que, si es la segunda, se toma como dirección principal la primera, ya que no existe composición alguna entre cualquier otra dirección y *Nula*.

Tal como mencionan los autores, el hecho de seleccionar como principal la dirección más frecuente cuando la segunda más probable es opuesta puede llevar a falsas alarmas, por eso hemos decidido almacenar ambas como direcciones principales. A continuación, podemos ver los histogramas con la distribución de frecuencias de cada dirección en cada zona:

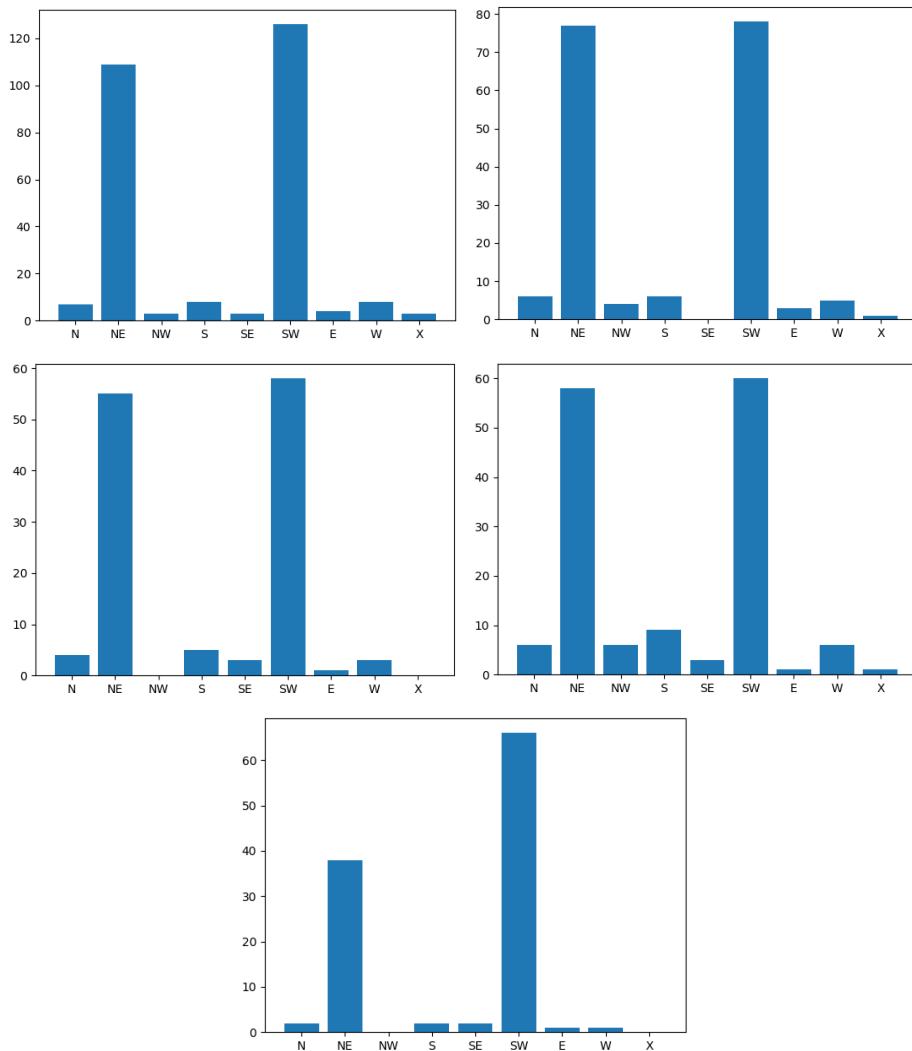


Figura 3.12. Histogramas de frecuencia de las direcciones en cada zona

Dado que, en nuestro conjunto de datos, el comportamiento “normal” es un desplazamiento diagonal con la misma dirección, pero distinto sentido, y las frecuencias son

muy similares, tenemos dos direcciones principales en cada zona. Como podemos ver, en todas las zonas las dos direcciones predominantes son *Noreste* y *Suroeste*. Respecto a este comportamiento “normal” en términos de dirección que hemos observado en nuestro conjunto de datos, podemos verlo representado en la siguiente imagen:



Figura 3.13. Dirección normal del conjunto de datos

Una vez que conocemos las direcciones predominantes, añadimos esta información a las zonas, que ya estarían completas para poder pasar a la detección de anomalías. Teniendo la velocidad y dirección de cada trayectoria de test, comparamos los valores con los intervalos de confianza correspondientes. En el caso de la velocidad, asumimos que una velocidad es normal si está dentro del intervalo siguiente (en nuestro caso, hemos considerado $\alpha = 0.9$):

$$(Sspeed_n > \mu_n + (\alpha * 1.4) * \sigma_n) \vee (Sspeed_n < \mu_n - (\alpha * 0.9) * \sigma_n)$$

Los factores que multiplican al término α tienen por objetivo refinar más los intervalos de búsqueda. Los valores 0.9 y 1.4 se han escogido tras una serie de experimentos en los que hemos probado todas las combinaciones posibles entre los rangos [0.5, 0.9] (para el intervalo inferior) y [1, 2] (para el intervalo superior), eligiendo aquella que daba mejores resultados en términos de verdaderos y falsos positivos.

Respecto a la dirección, los autores proponen que una trayectoria es anómala si se desvía más de 90° de la dirección principal. Sin embargo, creemos que es un abanico demasiado amplio, ya que, por ejemplo, un individuo caminando en dirección *Este* se diferencia considerablemente de otro que vaya en dirección *Norte*. Por tanto, hemos reducido este rango asumiendo que una trayectoria es anormal si se desvía más de 45° de la dirección dominante. Es decir, los intervalos de confianza serían los siguientes:

$$(Sdirection_n > D_{z_n} + 45^\circ) \vee (Sdirection_n < D_{z_n} - 45^\circ)$$

De forma adicional, tras el descubrimiento de zonas se crea una zona “por defecto” de forma automática a la que se asigna el identificador -1 y representa el conjunto de celdas de la cuadrícula por las que no pasa ningún punto y, por tanto, no están dentro de ningún cluster. Esta zona extra se añade antes de empezar a trabajar con las trayectorias de test y de ella se almacena el identificador, las celdas que la componen, la media y la desviación típica de la velocidad de la zona y su dirección. En cuanto a la media y desviación típica de la velocidad en esta zona sin trayectorias, la calculamos como la media de dichos valores en el resto de zonas. Respecto a la dirección, asignamos una nueva, *Def*, es decir, una dirección por defecto, que hemos decidido que sea opuesta a todas las demás. Desde nuestro punto de vista, alguien caminando por una celda por la que no ha pasado nadie en los vídeos de entrenamiento supone un comportamiento anómalo, sea cual sea su dirección, de ahí que consideremos que *Def* es opuesta a cualquiera de las demás direcciones. A la hora de detectar anomalías de velocidad, trabajamos con las mismas condiciones que en el resto de zonas. Para nuestro dataset concreto, dicha zona extra es la que engloba las celdas señaladas a continuación:

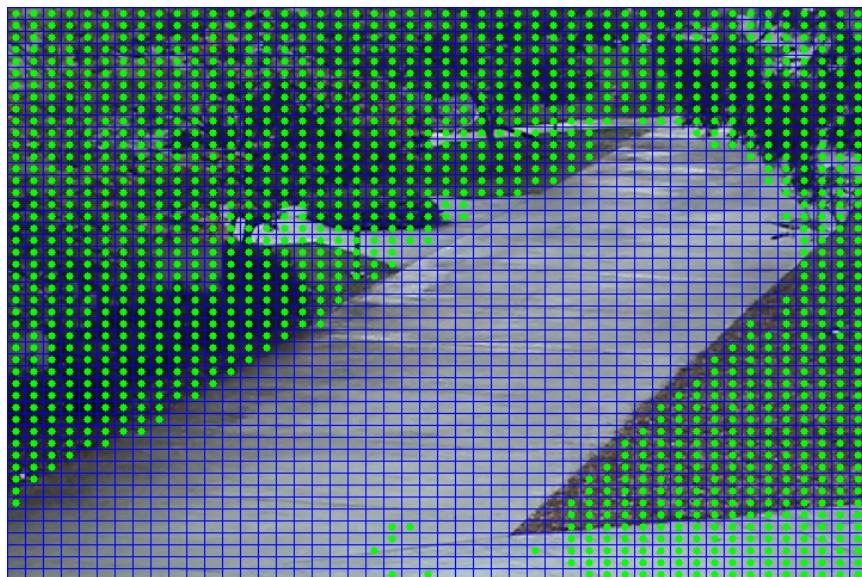


Figura 3.14. Celdas de la zona “por defecto”

3.4.1. Explicación de las funciones implementadas

Tras describir el funcionamiento de nuestro algoritmo de manera detallada, a continuación explicamos de qué se encarga cada una de las funciones implementadas. (Los ficheros que se guardan en la mayoría de funciones son meramente informativos, no son necesarios en la ejecución del algoritmo, ya que los valores con los que se trabaja se van almacenando en estructuras de datos. Únicamente los ficheros que se escriben en la función `filter_trajectories_both` son necesarios para evaluar el comportamiento del algoritmo).

La implementación, disponible en <https://github.com/laurahernandezm/TFG> está organizada en los siguientes archivos:

- `utils.py`. En este archivo encontramos algunas constantes y variables globales y funciones de utilidad general, como la creación de algunos directorios, la extracción de trayectorias a partir de los archivos de salida del algoritmo de tracking o la creación de los histogramas de frecuencia de las direcciones en las zonas. En concreto, las funciones de este archivo son las siguientes:

`create_folders`. Esta función simplemente comprueba si los directorios en los que se almacena la información existen y, en caso negativo, los crea.

`make_grid`. Esta función recibe como parámetro el fotograma sobre el que construiremos la cuadrícula y se encarga de calcular las dimensiones de cada celda y sus centroides, los cuales almacena junto con el identificador de la celda correspondiente. Devuelve la cuadrícula, representada como una lista de celdas, y una tupla con el ancho y alto de cada casilla.

`extract_trajectories`. Esta función se encarga de cambiar el formato de los cuadros delimitadores que recibe como entrada. A partir de un archivo dado, lee todas las líneas y de cada una extrae el número de frame, identificador de trayectoria, coordenadas x e y de la esquina superior izquierda, ancho y alto. Con estos datos se calcula el punto medio del segmento inferior y se crea una lista de trayectorias almacenando su identificador, coordenadas calculadas, frame, ancho y alto (de la caja delimitadora). Además, las trayectorias de menos de 20 frames se marcan y se eliminan.

`initialize_cells_summary`. En esta función se inicializa la información de las celdas creando un diccionario que contiene, de cada celda, su centroide, el número de puntos asignados a ella y su velocidad media (estos dos valores se inicializan a 0).

`grided_trajectories`. Esta función recibe como entrada una imagen (un fotograma) y la información de la cuadrícula, las trayectorias y las zonas y se encarga de dibujarla sobre la imagen. Se puede dibujar la cuadrícula, los centroides de todas las celdas, los centroides de aquellas que contienen algún punto *snapped*, las trayectorias originales, las zonas y las direcciones de las trayectorias. La imagen resultante se muestra en pantalla y se guarda.

`histograms`. Esta función realiza un recuento de la frecuencia de cada dirección en cada zona y lo representa en un gráfico de barras que se guarda como una imagen.

- `train.py`. En este archivo están las funciones que únicamente se usan durante el entrenamiento del modelo y no están relacionadas con la información de las zonas descubiertas. Son las siguientes:

`snap_train`. Este método recibe como entrada las trayectorias en el formato nuevo (el de la función anterior) y la información de las celdas y la cuadrícula y se encarga de comprobar, para cada punto de cada trayectoria, dentro de qué celda está y asignarlo a ella (*snapping*). Como comentamos en el apartado anterior, de cada punto *snapped* se guarda $[cid, x, y, mint, maxt, (x_{last}, y_{last}), dist, speed]$ y se van actualizando los cuatro últimos valores como se ha indicado antes, así como el número de puntos asignados a cada celda. Al final de la función se añaden las trayectorias *snapped* del vídeo que estemos procesando a una lista donde se almacenan todas las de entrenamiento.

`cell_average_speed`. En esta función se toman como entrada las trayectorias *snapped* de entrenamiento y la información de las celdas (en este punto, $[qtde, x, y, avgS = 0]$) y calcula la velocidad media de cada celda. Para ello se va acumulando la velocidad de cada punto *snapped* de una celda y, al final, se calcula la media con el número de puntos que haya asignados a dicha celda y se actualiza el valor de *avgS*. Además, se eliminan los tres últimos valores que habíamos guardado en las tuplas de los puntos *snapped* ($(x_{last}, y_{last}), dist, speed$, respectivamente). La información completa de las celdas se escribe en un fichero.

`save_train_snapped_trajectories`. Esta función simplemente escribe todas las trayectorias *snapped* de entrenamiento en un fichero.

`clustering`. Esta función recibe la información de las celdas calculada en la función anterior y aplica el algoritmo de *clustering* de propagación de afinidad sobre esta. Antes de aplicar el algoritmo, pasamos estos valores a una matriz y, como se ha explicado en el apartado anterior, ponderamos los valores, dándole un peso de 5 a las coordenadas *x* e *y* del centroide de las celdas, 3 a la velocidad media, *avgS*, y 1 al número de puntos resumidos en la celda, *qtde*. A continuación, se realiza el agrupamiento (con los parámetros *damping=0.9*, *preference=-300* y *max_iter=1000*) y, celda por celda, se comprueba a qué cluster pertenece. Al ir procesando las celdas, nos encontramos varios escenarios posibles:

- La celda actual pertenece al mismo cluster que la anterior. Añadimos el identificador de la celda a la misma zona que la anterior. Sin embargo, nos encontramos dos casos posibles:
 - Hay una única zona con esa etiqueta de *clustering*. Si solo hay una zona en ese cluster, no hacemos nada más.

- Hay varias subregiones dentro del mismo cluster. Si hay varias subzonas hay que comprobar si se pueden fusionar (como veímos en la figura 3.9). Si la celda actual es vecina de alguna de las celdas de alguna de esas subzonas, las fusionamos en una única zona y guardamos el índice donde está la información de la zona que hemos “eliminado” para borrarla por completo después.
- La celda actual no pertenece al mismo cluster que la anterior. Hay dos posibilidades:
 - Es la primera celda que se procesa de ese cluster. Si es la primera celda de un cluster, creamos una zona nueva, almacenando su identificador y guardamos el índice, dentro de la lista total, donde comienza esa zona.
 - No es la primera celda procesada en esa zona. Si ya se han visitado más celdas de ese cluster, comprobamos si la celda actual es vecina de alguna y, en caso afirmativo, la añadimos a la zona correspondiente. Si no, se crea una nueva subzona de esa.

Una vez que todas las celdas están asignadas a las zonas correctas, eliminamos la información de las subzonas que hemos ido fusionando y guardamos los datos definitivos (`[zoneid, cells]`) en un fichero.

- `zones.py`. Este archivo contiene las funciones que realizan algún cambio sobre la información de las zonas, tanto en tiempo de entrenamiento como durante test. Dichas funciones son:

`transform_zones`. Esta función se encarga de crear una lista con el identificador de cada zona y las coordenadas de los centroides de las celdas de cada una. Simplemente transforma los números de celda en sus centroides para que sea más sencillo dibujarlos en la función `grided_trajectories`.

`trajectories_in_zones`. Teniendo las celdas de cada zona y las trayectorias *snapped* (ya sean de entrenamiento o de test), se añade cada una a la zona correspondiente. Si estamos trabajando con datos de test, se almacena tanto el identificador de trayectoria como sus puntos *snapped*, mientras que si estamos entrenando, únicamente guardamos los puntos. Como comentamos en la sección anterior, para intentar mitigar el efecto negativo de trayectorias rotas, se almacenan únicamente aquellas con más de un punto *snapped* o, en si estamos trabajando en test, también las que tengan solo un punto, siempre que la diferencia entre *maxt* y *mint* de dicho

punto sea igual o superior a 20 (frames). Además, si en una zona no hay ninguna trayectoria, se elimina. La información actualizada de las zonas se escribe en un fichero.

`speed_in_zones`. Con la información disponible tras llamar a la función anterior se calcula la velocidad media de cada trayectoria. Para ello, recorremos todas las zonas y, para cada trayectoria *snapped*, vamos acumulando la distancia euclídea entre cada par de puntos (ponderando con un peso más alto la diferencia entre las coordenadas verticales) y, al final, dividimos ese valor entre la diferencia de fotogramas entre el punto de entrada y salida de la trayectoria en una zona. Al igual que en la función anterior, si trabajamos con trayectorias de test, almacenamos su identificador, si no, solo el valor de velocidad. Esta información se añade a la que ya teníamos y se escribe en un fichero.

`direction_in_zones`. Esta función opera de forma similar a la anterior, pero en lugar de calcular velocidades, calcula las direcciones. Para ello, como explicamos anteriormente, se calculan los parámetros de la función lineal que conecta el punto de entrada y salida de una trayectoria en una zona y, dependiendo de si son mayores, menores o iguales a 0, se asigna una dirección u otra. En trayectorias de entrenamiento no guardamos el identificador, mientras que en las de test, sí. La lista de direcciones de las trayectorias se añade a la información de las zonas y se almacena en un fichero.

`gaussian_dist`. Esta función se encarga de calcular la media y la desviación típica de la velocidad de cada zona. Para ello, se añaden a una lista todas las velocidades de las trayectorias de cada zona y sobre esa lista se aplican los métodos `mean` y `std` y los resultados se añaden a la información almacenada de cada zona. Al igual que en funciones anteriores, la nueva información se añade a la que ya teníamos (identificador de zona, números de las celdas que la forman, trayectorias *snapped* y velocidad y dirección de cada trayectoria) y se escribe en un fichero.

`main_direction`. En esta función se detecta la dirección dominante (o direcciones dominantes, en nuestro caso) de cada zona. En cada zona se hace un recuento de frecuencias, se almacenan las dos más repetidas y se hace la prueba chi cuadrado de Pearson. Si la prueba da 0 como resultado, la dirección que se almacena es *Noney* si es distinto de 0, se sigue el criterio especificado en la sección anterior. Por último, se guarda todo en un fichero.

`default_zone`. En esta función se toma como entrada una lista con el identificador de cada zona y los números de celda que las forman y se añade a ella la zona “por defecto”. Para ello, se comprueba qué casillas pertenecen a algún cluster y aquellas que estén libres se incluyen en la zona nueva

(identificada por -1). Esta nueva información se añade a los datos de entrada y se escriben en un fichero.

`default_zone_sp_dir`. Esta función recibe la información completa de las zonas descubiertas (identificador, celdas de la zona, trayectorias *snapped*, velocidades, direcciones, media y desviación típica de la velocidad y dirección principal) y lo que habíamos almacenado de la zona por defecto en la función anterior (identificador y celdas) y completa la información de esta. Las trayectorias *snapped*, velocidades y direcciones se quedan vacías, la media y la desviación típica de la velocidad se calculan como la media de dichos valores en el resto de zonas y como dirección principal se toma la que describimos en el apartado anterior (*Def*). Todo esto se escribe en un fichero.

- `test.py`. En este archivo encontramos las funciones que se ejecutan únicamente en test para realizar el snapping de las trayectorias y escribir la información de dichas trayectorias en las zonas correspondientes:

`snap_test`. Esta función es similar a `snap_train`, pero a la hora de guardar las trayectorias *snapped*, se almacena su identificador y se tienen en cuenta los puntos que están por debajo del límite inferior de la cuadrícula. Además, no se almacena información sobre el último punto, la distancia ni la velocidad. Por tanto, los datos que se guardan (y se escriben en un archivo) de cada trayectoria *snapped* son `[trid, [cid, x, y, mint, maxt]]`.

`write_complete_test`. Esta función simplemente escribe en un fichero la información de las trayectorias de test en las distintas zonas.

- `anomaly_detections.py`. Este archivo contiene las funciones en las que se realiza la detección de anomalías propiamente dicha y las que se encargan de generar los vídeos en los que aparecen dibujadas las cajas delimitadoras de los individuos que presentan trayectorias anormales. Son las siguientes:

`abnormal_speeds`. En esta función se realiza la detección de anomalías relacionadas con velocidad. Recibe como entrada la información de las zonas descubiertas en la etapa de entrenamiento y las trayectorias de test y comprueba, para cada zona, si las trayectorias de test que están dentro de ella están en el intervalo de confianza que hemos definido en la sección anterior. En caso negativo, se añade un par `[trid, trspeed]` a la lista de trayectorias anómalas en velocidad de cada zona. Todo ello se escribe en un fichero.

`abnormal_directions`. Esta función es equivalente a la anterior, pero se comprueban las direcciones de las trayectorias de test con el intervalo de confianza de dirección. En caso de que la dirección principal de una zona sea *None*, no se añade la lista de trayectorias anómalas en dirección `([[trid, trdir], ...])`, sino un mensaje informando de que no hay análisis de

dirección en esa zona. Una vez que tenemos esta información, lo escribimos en un fichero en el que cada zona se representa de la siguiente forma:

```
[id, [cells], [snappedtr], [snappedsp], [snappeddir],  
mean, std, maindir, [abnsp], [abndir]]
```

Es decir, identificador de la zona, lista de las celdas que la componen, lista de trayectorias *snapped* dentro de ella, lista de velocidades de las trayectorias, lista de direcciones de las trayectorias, media y desviación típica de la velocidad, dirección/es principal/es, lista de trayectorias anómalas según velocidad y lista de trayectorias anómalas según dirección.

`filter_trajectories_speed`, `filter_trajectories_direction`. En estas funciones se almacenan en un conjunto los identificadores de las trayectorias identificadas como anómalas en cuanto a velocidad y dirección, respectivamente, y se lee el archivo de trayectorias original para seleccionar las líneas correspondientes a esas trayectorias. Dichas líneas se escriben en otro fichero y, además, usando su información, se dibujan los cuadros delimitadores sobre los individuos anómalos en los fotogramas del vídeo que estemos procesando y se genera el vídeo con dichas cajas marcando las anomalías.

`filter_trajectories_both`. Esta función lleva a cabo la misma tarea que las dos anteriores, pero se marcan conjuntamente las trayectorias anómalas tanto por velocidad como por dirección (en lugar de por separado, como en las anteriores).

- `run.py`. Este archivo simplemente es el script de ejecución desde el cual se llama a las funciones de los archivos comentados.

3.5. Análisis y elección de factores multiplicadores para el intervalo de confianza de velocidad

Para elegir el valor de estos factores, realizamos una serie de ejecuciones con cada combinación posible dados los intervalos [0.5, 1] y [1, 2] y estudiamos la variación de distintas métricas para quedarnos con el par de valores más adecuado.

Respecto a las pruebas que realizamos, podemos ver en las siguientes gráficas cómo varía el número de verdaderos positivos, falsos positivos y falsos negativos con los distintos valores probados. El eje horizontal representa el intervalo de valores probados para el factor del intervalo inferior y el eje vertical, dependiendo de la gráfica, representa el número de verdaderos positivos, verdaderos y falsos positivos o verdaderos y falsos positivos y falsos negativos.

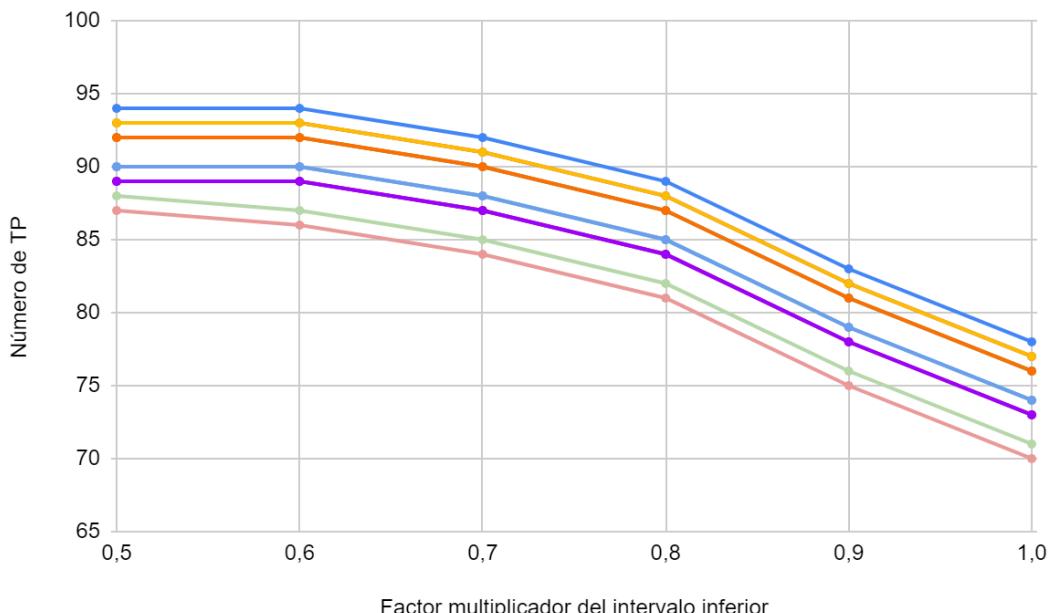


Figura 3.15. Gráfica que representa los cambios en el número de verdaderos positivos al variar el valor de los factores multiplicadores para los intervalos de confianza de velocidad

En este gráfico, cada línea representa un mismo valor del factor para el intervalo superior (entre 1 y 2) y, para dicho valor fijo, podemos ver cómo varía el número de verdaderos positivos al ir aumentando el factor multiplicador del intervalo inferior (entre 0.5 y 1). Realmente, hay 11 líneas, una por cada valor posible en el rango [1, 2], sin embargo, algunas de ellas presentan el mismo número de verdaderos positivos, por lo que aparecen solapadas.

Podemos ver cómo, independientemente del valor del factor del intervalo superior, al aumentar el factor del intervalo inferior, el número de verdaderos positivos va disminuyendo. En concreto, en el rango (0.8, 1] este número desciende de forma considerable y más rápidamente que en [0.5, 0.8]. Asimismo, si atendemos a los valores del factor del intervalo superior, vemos que a medida que aumenta, el número de verdaderos positivos disminuye, pudiéndose observar una diferencia mayor entre la línea del 1.4 (tercera línea visible empezando desde arriba) y las siguientes. A la vista de esta gráfica, podríamos pensar que la mejor combinación de valores es 0.5 para el intervalo inferior y 1 para el superior, sin embargo, también hay que tener en cuenta el número de falsas alarmas. Para ello, nos fijamos en la siguiente gráfica:

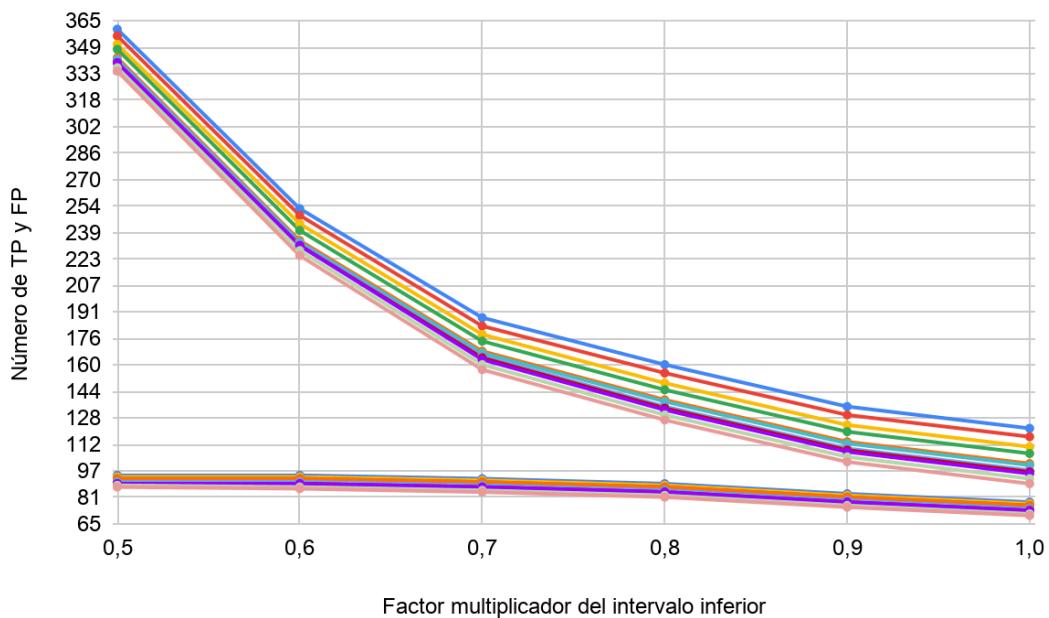


Figura 3.16. Gráfica que representa los cambios en el número de verdaderos y falsos positivos al variar el valor de los factores multiplicadores para los intervalos de confianza de velocidad

Si representamos en la misma gráfica el número de verdaderos y falsos positivos podemos ver que la idea inicial de escoger los valores 0.5 y 1 es favorable respecto al número de detecciones correctas, pero el número de falsas alarmas es demasiado elevado. De nuevo, vemos que, al igual que antes, el número de falsos positivos disminuye conforme vamos aumentando el valor de ambos factores multiplicadores. Tomando como referencia únicamente el número de falsos positivos, lo mejor sería coger los valores 1 y 2, respectivamente. Sin embargo, vimos en la gráfica anterior que estos valores también suponían el mínimo de verdaderos positivos. Por tanto, es necesario buscar un punto de equilibrio entre ambas métricas. Nos fijamos también en el número de falsos negativos para buscar dicho punto de equilibrio:

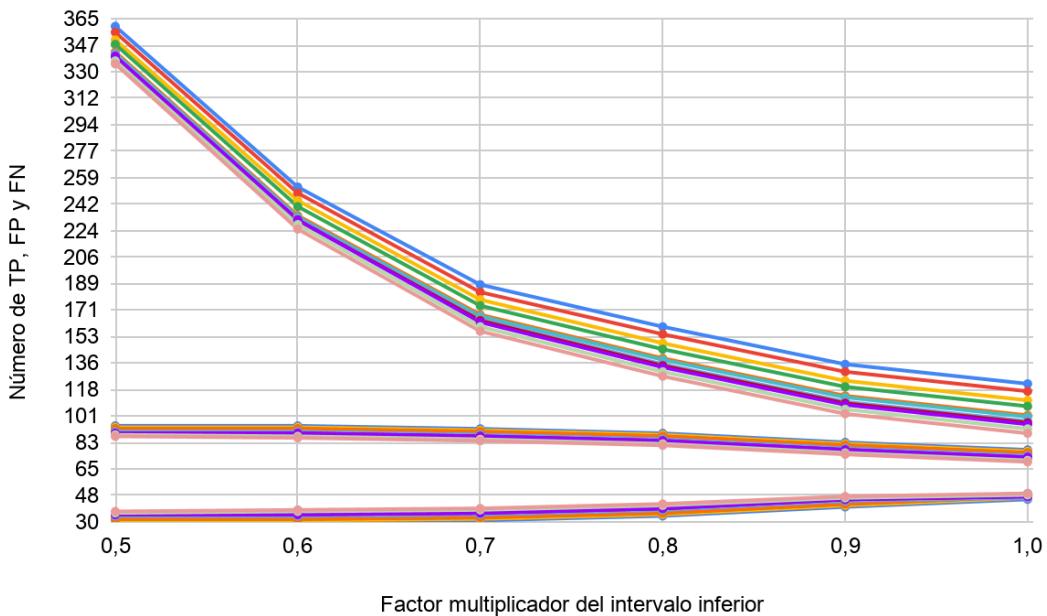


Figura 3.17. Gráfica que representa los cambios en el número de verdaderos y falsos positivos y falsos negativos al variar el valor de los factores multiplicadores para los intervalos de confianza de velocidad

Al contrario que en las gráficas anteriores, en este caso, el número de falsos negativos aumenta conforme aumentamos el valor de los factores multiplicadores, alcanzando el máximo en los valores 1 y 2, respectivamente. Intentando minimizar el número de falsos positivos, los valores escogidos deberían estar lo más próximos posible al máximo (1 y 2), sin embargo el número de verdaderos positivos se ve considerablemente afectado a partir de 0.8 y 1.4, como pudimos ver en la primera gráfica. Asimismo, el número de falsos negativos crece más deprisa a partir de 0.8 y aumenta al incrementar el valor del intervalo superior.

De acuerdo a los datos numéricos, estimamos que la mejor proporción TP-FP se obtenía para valores del factor del rango inferior entre 0.9 y 1 y 1.4 y 1.6 para el superior. La siguiente tabla recoge los valores de TP, FP, FN y cambios de identificador que resultan con los factores multiplicadores de la primera columna: [0.9, 1] para el intervalo inferior y [1.4, 1.6] para el superior.

Inf. - Sup.	TP	FP	FN
0.9 - 1.4	81	114	42
0.9 - 1.5	79	113	44
0.9 - 1.6	79	110	44
1 - 1.4	76	101	47
1 - 1.5	74	100	49
1 - 1.6	74	97	46

Figura 3.18. Resultados de las métricas TP, FP, FN y IDs para distintos factores multiplicadores de α

A la vista de los datos, tanto TP como FN presentan mejores valores cuando el factor inferior es 0.9. En cuanto al factor superior, entre 1.4 y 1.6 hemos decidido quedarnos con 1.4, ya que no se observa ninguno que destaque sobre los demás y este es el que presenta un mayor número de TP. Además, con esta combinación de valores se consigue el valor más alto de la medida F1.

3.6. Resultados experimentales

En este capítulo comentaremos los resultados experimentales obtenidos durante el desarrollo del trabajo. Veremos los resultados obtenidos al usar nuestro algoritmo con un sistema de *tracking* más sencillo y los compararemos con los obtenidos usando el método que hemos comentado en la sección anterior.

Para realizar la comparación de resultados, las métricas que utilizamos son las siguientes:

- Verdaderos positivos o TP. Trayectorias anómalas detectadas como anómalas.
- Falsos positivos o FP. Trayectorias normales detectadas como anómalas.
- Falsos negativos FN. Trayectorias anómalas detectadas como normales.
- Recall o ratio de verdaderos positivos. Predicciones correctas entre el total de positivos ($TPR = \frac{TP}{TP+FN}$).
- Precisión. Predicciones positivas correctas entre el número total de predicciones positivas ($PPV = \frac{TP}{TP+FP}$).
- Medida F1. Media armónica de recall y precisión ($F1 = 2 * \frac{PPV*TPR}{PPV+TPR}$).

3.6.1. Comparación de resultados

Para realizar una buena detección de anomalías basándonos en la trayectoria de una serie de personas es de vital importancia que la información relativa a dichas trayectorias sea lo más exacta posible. Para demostrar esto, decidimos probar el algoritmo implementado sobre un conjunto de trayectorias extraído con un método de *tracking* menos potente, usando *Single Shot Detector* (SSD) como detector y SORT como tracker, descritos en [30] y [31], respectivamente, y sobre otro extraído con el método anteriormente descrito.

El algoritmo SORT necesita como entrada una serie de detecciones, para lo cual hemos usado una implementación sencilla de SSD basada en MobileNet. Hemos realizado algunos cambios en el código original de SORT para que la salida fuera compatible con la entrada de nuestro algoritmo y para que los archivos se leyieran correctamente.

Los resultados obtenidos por nuestro algoritmo con ambos sistemas de *tracking* son los siguientes:

Método de tracking	TP	FP	FN	Recall	Precisión	F1
SSD + SORT	44	331	80	0.355	0.117	0.176
Tracking without bells and whistles	81	114	42	0.659	0.416	0.509

Figura 3.19. Resultados de la evaluación de nuestro algoritmo con dos métodos de *tracking*: SSD+SORT y Tracking without bells and whistles

Podemos ver que todas las métricas presentan unos valores mucho peores al usar SSD y SORT como método de extracción de trayectorias. La tasa de verdaderos positivos, TP, se reduce a la mitad y el número de falsas alarmas se dispara a casi el triple, por lo que la precisión del modelo desciende considerablemente. Además, la tasa de falsos negativos se duplica, lo que también perjudica el recall. La medida F1, al ser una media armónica, penaliza más los valores bajos, por lo que, en este caso, debido a la baja precisión, esta medida también obtiene un resultado bajísimo.

En cuanto al método de *tracking* basado en *deep learning* más actual, se mejoran considerablemente todas las métricas y podemos ver que, del total de trayectorias anómalas, más de un 65% se clasifican como tal (recall). La precisión es baja, dado que el número de falsos positivos sigue siendo alto, sin embargo, la mejora con respecto al método anterior también es importante, por lo que la medida F1 también consigue un valor más alto.

Tal como indicamos en la introducción del trabajo y a la vista de estos resultados, podemos afirmar que las trayectorias con las que trabajamos juegan un papel importantísimo a la hora de detectar cuáles son anómalas. Podemos ver que usando un detector más básico junto con un *tracker* de hace algunos años la detección es bastante peor que usando un sistema más actual. Viendo, además, algunos vídeos generados al realizar el *tracking*, hemos observado que se producen numerosos cambios de identificador

entre trayectorias, lo cual también hace crecer el número de falsos positivos. Por ejemplo, una persona andando hacia la derecha que intercambia el identificador de trayectoria con otra que va andando hacia abajo puede generar una alarma en cuanto a dirección. También observamos que el ajuste de los cuadros delimitadores a veces es problemático, ya que hay personas quietas cuyo cuadro delimitador “se mueve” debido a su ajuste continuo y esto provoca que su velocidad sea distinta de 0 y no se compute como velocidad anómala.

4. Conclusiones y trabajo futuro

4.1. Conclusiones

En este trabajo hemos mostrado cómo pueden usarse métodos de aprendizaje no supervisado para detectar trayectorias anómalas en vídeo, en lugar de utilizar modelos supervisados. Mientras que dichos modelos necesitan conjuntos de vídeos o datos etiquetados para aprender lo que se considera como un comportamiento normal o anómalo, nuestro enfoque únicamente requiere de una serie de vídeos sin anotación alguna, a partir de los cuales extrae las características que representan una trayectoria usual. Esto es una gran ventaja, porque es mucho más rápido y sencillo obtener datos sin etiquetar y se evitan problemas como el desbalanceo que podríamos encontrar en el caso de que los datos para entrenar no contuvieran suficientes ejemplos normales o anómalos.

Además, al usar *clustering*, es el propio modelo el que decide cómo se distribuyen los datos según sus características, por lo que podemos adaptarnos a las condiciones concretas de cada dataset simplemente procesando los vídeos que lo componen, sin necesidad de introducir conocimiento de forma manual o cambiar la implementación para cada uno.

Ha quedado demostrado también que, para que una detección de anomalías basada en trayectorias sea buena, la etapa previa de *tracking*, para obtener dichas trayectorias, es de vital importancia. Independientemente de la bondad del algoritmo de detección de anomalías diseñado, si el conjunto de datos que usamos como entrada no es bueno, dicho algoritmo no puede generar buenos resultados. Por tanto, la detección de trayectorias anormales está indudablemente ligada al *tracking* y se conseguirá un mejor rendimiento conforme los algoritmos de detección y seguimiento de personas avancen.

4.2. Trabajo futuro

En cuanto a posibles ampliaciones o mejoras sobre este trabajo, las vías de actuación serían las siguientes. En primer lugar, sería interesante integrar trayectorias grupales; esto podría ser útil para detectar un grupo de personas que se detiene o comienza a desplazarse a más velocidad o en una dirección extraña, como podría ser el caso de una banda de atracadores que huye después de asaltar a alguien.

Otra mejora interesante sería refinar el cálculo de las direcciones para que la dirección *Nula* se asociara únicamente a aquellas personas que no se han movido durante toda la secuencia de fotogramas, y no a las que han empezado y acabado su trayectoria en el mismo lugar, ya que durante todo ese recorrido han podido suceder muchas cosas. Además, se podrían introducir cambios en la zona por defecto, para refinar la detección relacionada con la dirección.

Por último, tal como proponen en [24], se podría construir un sistema mucho más completo detectando anomalías más allá de las trayectorias, teniendo en cuenta también el comportamiento que presenta cada persona. De esta forma podríamos localizar individuos que, a pesar de tener una dirección y velocidad usuales, están llevando a cabo alguna actividad fuera de lo normal.

Índice de figuras

Sección 2

- 2.1. (Izda.) Underfitting. (Centro) Modelo adecuado. (Dcha.) Overfitting
- 2.2. Estructura de una neurona
- 2.3. Función escalonada
- 2.4. Función sigmoide
- 2.5. Función tangente hiperbólica
- 2.6. Función ReLU
- 2.7. Estructura de una red neuronal
- 2.8. Operación de convolución
- 2.9. (Izquierda) Imagen original. (Derecha) Imagen tras convolución
- 2.10. Núcleo de convolución para obtener los bordes en una imagen
- 2.11. Efecto de la función ReLU
- 2.12. Resultado de aplicar *max pooling* y *average pooling*
- 2.13. Esquema de una CNN
- 2.14. (Izda.) Bloque normal, $y = F(x) = x$. (Dcha.) Bloque residual, $y = F(x) + x = x$
- 2.15. Curvas de error de las redes estándar frente a las ResNet
- 2.16. (Izquierda) Imagen original. (Derecha) Búsqueda selectiva
- 2.17. Arquitectura de una R-CNN
- 2.18. Arquitectura de una Fast R-CNN
- 2.19. “Anclas” en una imagen
- 2.20. Resultado de aplicar NMS
- 2.21. Arquitectura de una Faster R-CNN
- 2.22. Esquema del modelo FPN
- 2.23. Funcionamiento de un modelo FPN
- 2.24. (Arriba) Faster R-CNN estándar. (Abajo) Faster R-CNN con FPN

Sección 3

- 3.1. (Izquierda) Fotograma del conjunto de entrenamiento. (Derecha) Fotograma del conjunto de test
- 3.2. Esquema del algoritmo *Tracking without bells and whistles*
- 3.3. Esquema del algoritmo de detección de anomalías

- 3.4. *Snapping* de un trayectoria
- 3.5. Diagrama del funcionamiento del algoritmo desarrollado
- 3.6. Transformación en los cuadros delimitadores
- 3.7. Cuadrícula de 48x48 sobre la que trabajamos
- 3.8. Trayectorias originales y celdas a las que se asignan durante el *snapping*
- 3.9. Fusión de zonas vecinas
- 3.10. Vecindario de una celda
- 3.11. Zonas descubiertas
- 3.12. Histogramas de frecuencia de las direcciones en cada zona
- 3.13. Dirección normal del conjunto de datos
- 3.14. Celdas de la zona “por defecto”
- 3.15. Gráfica que representa los cambios en el número de verdaderos positivos al variar el valor de los factores multiplicadores para los intervalos de confianza de velocidad
- 3.16. Gráfica que representa los cambios en el número de verdaderos y falsos positivos al variar el valor de los factores multiplicadores para los intervalos de confianza de velocidad
- 3.17. Gráfica que representa los cambios en el número de verdaderos y falsos positivos y falsos negativos al variar el valor de los factores multiplicadores para los intervalos de confianza de velocidad
- 3.18. Resultados de las métricas TP, FP, FN y IDs para distintos factores multiplicadores de α
- 3.19. Resultados de la evaluación de nuestro algoritmo con dos métodos de *tracking*: SSD+SORT y Tracking without bells and whistles

Bibliografía

- [1] Thomas M. Mitchell. Machine Learning. 1.a ed. New York, NY, USA: McGraw-Hill, Inc., 1997, pág. 3. ISBN: 9780070428072.
- [2] Brendan J. Frey y Delbert Dueck. "Clustering by Passing Messages Between Data Points". En: Science 315 (5814) (16 de febrero de 2007), páginas 972-976. DOI: [10.1126/science.1136800](https://doi.org/10.1126/science.1136800). URL: <https://doi.org/10.1126/science.1136800>
- [3] Ian Goodfellow, Yoshua Bengio y Aaron Courville. Deep Learning. Cambridge, MA, USA: MIT Press, 2017. ISBN: 9780262035613
- [4] URL: <https://aishack.in/tutorials/image-convolution-examples/>
- [5] URL:
<https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148>
- [6] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun. "Deep Residual Learning for Image Recognition". En: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (junio de 2016). DOI: [10.1109/CVPR.2016.90](https://doi.org/10.1109/CVPR.2016.90). arXiv: [1512.03385](https://arxiv.org/abs/1512.03385). URL: <https://arxiv.org/abs/1512.03385>
- [7] Ross Girshick, Jeff Donahue, Trevor Darrell y Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation". En: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (junio de 2014). DOI: [10.1109/CVPR.2014.81](https://doi.org/10.1109/CVPR.2014.81). arXiv: [1311.2524](https://arxiv.org/abs/1311.2524). URL: <https://arxiv.org/abs/1311.2524>
- [8] Jasper R. R. Uijlings, K. van de Sande, T. Gevers y Arnold W. M. Smeulders. "Selective Search for Object Recognition". En IJCV 104 (2) (2013), páginas 154-171. DOI: [10.1007/s11263-013-0620-5](https://doi.org/10.1007/s11263-013-0620-5).
URL: <https://link.springer.com/article/10.1007%2Fs11263-013-0620-5>
- [9] URL: https://d2l.ai/chapter_computer-vision/rcnn.html

- [10] Ross Girshick. “Fast R-CNN”. En: IEEE International Conference on Computer Vision (ICCV) (diciembre de 2015). DOI: [10.1109/ICCV.2015.169](https://doi.org/10.1109/ICCV.2015.169). arXiv: [1504.08083](https://arxiv.org/abs/1504.08083). URL: <https://arxiv.org/abs/1504.08083>
- [11] Shaoqing Ren, Kaiming He, Ross Girshick y Jian Sun. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. (2015). arXiv: [1506.01497](https://arxiv.org/abs/1506.01497). URL: <https://arxiv.org/abs/1506.01497>
- [12] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan y Serge Belongie. “Feature Pyramid Networks for Object Detection”. En: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (julio de 2017). DOI: [10.1109/CVPR.2017.106](https://doi.org/10.1109/CVPR.2017.106). arXiv: [1612.03144](https://arxiv.org/abs/1612.03144). URL: <https://arxiv.org/abs/1612.03144>
- [13] URL:
<https://mc.ai/review-fpn>
- [14] URL:
https://medium.com/@jonathan_hui/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c
- [15] Jane Bromley, James W. Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Eduard Saeckinger, Cliff Moore y Roopak Shah. “Signature verification using a “siamese” time delay neural network”. En: International Journal of Pattern Recognition and Artificial Intelligence 7 (4) (agosto de 1993). URL: <https://doi.org/10.1142/S0218001493000339>
- [16] R. Hadsell, S. Chopra y Yann LeCun. “Dimensionality Reduction by Learning an Invariant Mapping”. En: IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’06), Nueva York, EEUU (junio de 2006). DOI: [10.1109/CVPR.2006.100](https://doi.org/10.1109/CVPR.2006.100). URL: <https://ieeexplore.ieee.org/document/1640964>
- [17] Alexander Hermans, Lucas Beyer y Bastian Leibe. “In Defense of the Triplet Loss for Person Re-Identification”. (2017). arXiv: [1703.07737](https://arxiv.org/abs/1703.07737). URL: <https://arxiv.org/abs/1703.07737>

- [18] URL: <http://www.svcl.ucsd.edu/projects/anomaly/>
- [19] Vijay Mahadevan, Weixin Li, Viral Bhalodia y Nuno Vasconcelos. “Anomaly Detection in Crowded Scenes”. En: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Francisco, CA, USA (junio de 2010). DOI: [10.1109/CVPR.2010.5539872](https://doi.org/10.1109/CVPR.2010.5539872). URL: <https://ieeexplore.ieee.org/document/5539872>
- [20] Tewodros A. Biresaw, Tahir Nawaz, James Ferryman y Anthony I. Dell. “ViTBAT: Video Tracking and Behavior Annotation Tool”. En: IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), Colorado, Springs, USA (agosto de 2016). DOI: [10.1109/AVSS.2016.7738055](https://doi.org/10.1109/AVSS.2016.7738055). URL: <https://ieeexplore.ieee.org/document/7738055>
- [21] Philipp Bergmann, Tim Meinhardt y Laura Leal-Taixe. “Tracking without bells and whistles”. En: IEEE International Conference on Computer Vision (ICCV) (2019). arXiv: [1903.05625](https://arxiv.org/abs/1903.05625). URL: <https://arxiv.org/abs/1903.05625>
- [22] Jonathan Huang y col. “Speed/accuracy trade-offs for modern convolutional object detectors”. En: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA (julio de 2017). DOI: [10.1109/CVPR.2017.351](https://doi.org/10.1109/CVPR.2017.351). arXiv: [1611.10012](https://arxiv.org/abs/1611.10012). URL: <https://arxiv.org/abs/1611.10012>
- [23] Georgios D. Evangelidis y Emmanouil Z. Psarakis. “Parametric Image Alignment Using Enhanced Correlation Coefficient Maximization”. En: IEEE Transactions on Pattern Analysis and Machine Intelligence 30 (10) (octubre de 2008). DOI: [10.1109/TPAMI.2008.113](https://doi.org/10.1109/TPAMI.2008.113). URL: <https://ieeexplore.ieee.org/document/4515873>
- [24] Serhan Coşar y col. “Towards Abnormal Trajectory and Event Detection in Video Surveillance”. En: IEEE Transactions on Circuits and Systems for Video Technology 27 (3), páginas 683-695 (marzo de 2017). DOI: [10.1109/TCSVT.2016.2589859](https://doi.org/10.1109/TCSVT.2016.2589859). URL: <https://ieeexplore.ieee.org/document/7509595>
- [25] Duc Phu Chau, François Bremond, Monique Thonnat y Etienne Corvee. “Robust Mobile Object Tracking Based on Multiple Feature Similarity and Trajectory Filtering” (marzo de 2011). arXiv: [1106.2695](https://arxiv.org/abs/1106.2695). URL: <https://arxiv.org/abs/1106.2695>

- [26] Duc Phu Chau y col. “Group interaction and group tracking for video-surveillance in underground railway stations” (2011). URL: <https://hal.inria.fr/inria-00624356/>
- [27]
<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AffinityPropagation.html>
- [28] <https://numpy.org/doc/1.18/reference/generated/numpy.mean.html>
- [29] <https://numpy.org/doc/1.18/reference/generated/numpy.std.html>
- [30] Wei Liu y col. “SSD: Single Shot MultiBox Detector”. En: European Conference on Computer Vision (ECCV) (2016). DOI: [10.1007/978-3-319-46448-0_2](https://doi.org/10.1007/978-3-319-46448-0_2). URL: <https://arxiv.org/abs/1512.02325>
- [31] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos y Ben Upcroft. “Simple Online and Realtime Tracking”. En: IEEE International Conference on Image Processing (ICIP) (2016). DOI: [10.1109/ICIP.2016.7533003](https://doi.org/10.1109/ICIP.2016.7533003). URL: <https://arxiv.org/abs/1602.00763>