Facultad de Ciencias

ANÁLISIS DE IMAGEN PARA EL CONTEO AUTOMÁTICO DE OBJETOS EN IMÁGENES BIOLÓGICAS

(Image analysis for automatic object counting in biological images)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN CIENCIA DE DATOS / DATA SCIENCE

Autor: Eduardo Ruiz Ruiz

Director: Steven Van Vaerenbergh

Codirector: Marcos Cruz Rodríguez

RESUMEN

Estimar el número de objetos o partículas en una imagen o en un video, que al final es una secuencia de imágenes, es algo en lo que siempre ha habido intereses, desde el gobierno para controlar a sus habitantes, hasta organizaciones ecológicas para llevar control de poblaciones de ciertas especies de fauna y flora. El primer método para contar partículas en una imagen que a cualquiera se le ocurre es hacerlo a mano, pero normalmente se espera tener que contar un gran número de partículas, resultando el método manual demasiado cansado. El método CountEm utiliza un muestreo geométrico y sistemático para realizar una estimación del tamaño de la población pseudo-manual, superpone una rejilla de ventanas de muestreo definida por unos parámetros sobre la imagen y se contarán las partículas de dichas ventanas de muestreo para dar una estimación. Esto convierte una tarea tediosa y poco práctica de contar a mano, en algo más simple, más fácil de verificar y que no consume tanto tiempo, esto no quiere decir que no se pueda mejorar.

En este proyecto trataremos de buscar un método que pueda dar una estimación inicial del tamaño de la población para que CountEm pueda elegir unos parámetros iniciales para la rejilla más adecuados y de manera automática. Para ello se estudiarán y analizarán dos métodos de procesado digital de imágenes, uno de ellos más simple y otro que estará basado en machine learning.

Palabras clave: Imágenes, conteo de objetos, aprendizaje automático, análisis de imagen, estimación del tamaño de la población, coeficiente de error

ABSTRACT

Estimating the number of objects or particles in an image or video, which is basically a sequence of images, is something that has always been of interests, from the government to control its inhabitants, to environmental organizations to keep track of populations of certain species of fauna and flora. The first method of counting particles in an image that anyone can think of is do it by hand, but normally one expects to have to count a large number of particles, making the manual method too tiring. The CountEm method uses a geometric and systematic sampling to make a pseudo-manual population size estimate, superimposes a square grid of quadrats defined by some parameters on the image, and the particles in those quadrats will be counted to give an estimate. This makes the tedious and impractical task of counting by hand simpler, easier to verify and less time consuming, but this does not mean that it cannot be improved.

In this project we will try to find a method which can give an initial population size estimate so that CountEm can choose a more suitable initial square grid parameters automatically. In order to do so, two methods of digital image processing will be studied and analyzed, one of them will be simpler and the other based on machine learning.

Keywords: Images, object counting, machine learning, image analysis, estimation of the size of the population, coefficient of error

ÍNDICE

Capítulo 1: Introducción

Desde que se dispone de la capacidad de registrar eventos de manera gráfica con fotografías o videos existe una necesidad también de contar elementos de interés que aparecen en ellos. Por ejemplo, un gobierno tendría la necesidad de contar cuantas personas hay en una manifestación o, algo de bastante relevancia actualmente, contar cuanta gente está reunida en una sola zona por motivos de seguridad y salud. Aparte del control de personas, el conteo de objetos en imágenes resulta muy útil para el control de poblaciones de animales ya sea para estudios de investigación de la fauna animal o para llevar un control de la población de un parque natural o tener controlado el número de animales de una especie amenazada.

La primera idea que se le viene a cualquiera a la cabeza es contarlos a mano, pero acaba siendo inviable por lo lento y tedioso que resulta (sobre todo con muchos elementos para contar). Para sopesar el problema existen diversas técnicas, aplicaciones y algoritmos para estimar el conteo de elementos en imágenes.

En este trabajo analizaremos varios de estos métodos los cuales tendrán distintos enfoques. Nuestro punto de partida será el método CountEm [1] para el conteo de elementos dado a que es un software rápido y eficiente que utiliza el muestreo sistemático con una rejilla de ventanas de muestreo. CountEm sirve para cualquier tipo de objeto, pero necesita ajustar manualmente los parámetros que definen la rejilla y además el conteo de objetos de la muestra se realiza de forma manual.

Debido a que CountEm necesita de ciertas tareas manuales para su apropiada ejecución, nuestro objetivo es analizar si dichas tareas pueden realizarse con ayuda al menos parcial de un software de conteo automático.

Para este análisis disponemos de un dataset de imágenes con miles de aves de Canadá proporcionadas por el Canadian Wildlife Service, con anotaciones manuales de las posiciones y el tipo de las aves.

Primero observamos el rendimiento de un método básico como el algoritmo Watershed y comprobamos que no es lo suficientemente preciso para las imágenes que utilizamos.

Tras una búsqueda bibliográfica de un método más avanzado y reciente acabamos eligiendo Class-Agnostic Counting (CAC) porque dispone de código y puede ser entrenado para cualquier tipo de objeto, de ahí lo de “Class-Agnostic”, que es lo que necesitamos para preservar las ventajas de CountEm.

En el trabajo adaptamos y entrenamos CAC para las imágenes de aves que nos interesan obteniendo resultados satisfactorios, aunque requieren de pruebas adicionales en trabajos futuros.

Capítulo 2: Estudio previo de los métodos empleados

En este capítulo explicaremos el concepto general detrás de los métodos que queremos analizar, además de pruebas guiadas de su aplicación práctica para estimar el tamaño de las poblaciones.

Primero hablaremos del método base de este proyecto CountEm y después procederemos a los métodos de procesamiento digital de imágenes, Watershed y Class-Agnostic Counting.

Aunque CountEm si usa procesamiento de imágenes básico, no es lo principal detrás de su funcionamiento, sino el muestreo geométrico, como se va a ver a continuación.

2.1. CountEm

CountEm es un programa que ofrece una estimación insesgada del número de partículas en una imagen [1].

Partiendo de que una población es un conjunto finito de N elementos o partículas individuales, surge la necesidad de estimar el tamaño de dicha población ya sea mediante enfoques puramente visuales en la cual la estimación se hace teniendo en cuenta la densidad, o más manuales, basándose en la habilidad humana para contar las partículas, siendo una tarea lenta y cansada. Como alternativa a esto CountEm realiza un muestreo sistemático mediante una rejilla de ventanas de muestreo, siendo lo ideal muestrear y contar entre 50 y 200 partículas para estimar poblaciones de cualquier tamaño y distribución. La única limitación a este método es que las partículas de la población deben ser identificables mediante el conteo manual.

Este método está basado en el conteo manual de una cantidad razonablemente pequeña de partículas, contenidas en ventanas de muestreo, dentro de la imagen de la que se quieran estimar el número total de partículas. Realizando medidas simuladas mediante el método de Monte Carlo, los desarrolladores han demostrado que contando alrededor de 100 partículas en unas 30 ventanas de muestreo no vacías produce unos coeficientes de error entre el 5-10%.

Mostrándolo con un ejemplo, en la siguiente imagen de espectadores, la idea es contar el número de espectadores considerando como partículas sus caras. Para ello se crea una rejilla de ventanas de muestreo, cuyo tamaño, separación y ángulo además de otros parámetros se podrán especificar, hablaremos de ellos más adelante.

Figura 1: Imagen de ejemplo de espectadores con la rejilla de ventanas de muestreo superpuesta [2].

Las partículas de las ventanas de muestreo se cuentan a mano, puede ser 0 el número de una ventana de muestreo, aunque para dar una estimación razonable la cuenta total de partículas debería estar entre 50 y 200. Para evitar el sesgo debido a los bordes, es decir, que partículas se deben contar o no contar, si están ligeramente dentro del cuadrante, si están a la mitad, si tocan ligeramente el borde, pues para eso se aplica la regla de la línea prohibida. Para ilustrarlo cogeremos la ventana de muestreo señalada en la imagen anterior y haremos el recuento de sus partículas teniendo en cuenta la regla.

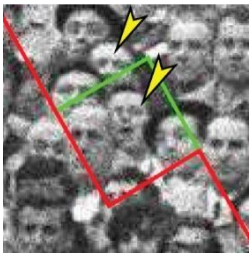

Figura 2: Ventana de muestreo con las partículas que se han contado señaladas [2].

El conteo aplicando la regla de la línea prohibida funciona de manera que contamos las partículas si están enteramente dentro de la ventana de muestreo o si están parcialmente dentro tocando la línea permitida del borde (de color verde) y si en ningún caso toca la línea extendida prohibida (de color rojo). En la imagen podemos ver que cuenta dos espectadores: uno está completamente dentro de la ventana de muestreo y otro tiene una parte dentro tocando solo el borde permitido. Da la impresión de que hay más caras dentro, pero todas ellas tocan la línea prohibida, por lo que no se cuentan.

Una vez explicado la idea detrás del funcionamiento del programa, procederemos a explicar el procedimiento para estimar la cantidad de partículas de una imagen, primero comenzaremos explicando los parámetros de la cuadrícula que serán necesarios para establecer la rejilla de ventanas de muestreo en la imagen como se ve en la **Figura 1**.

Hay dos parámetros para definir la cuadrícula f y n_0:

- f: "Sampling fraction" o fracción de muestreo y equivale a la longitud de un lado de un cuadrante (t) al cuadrado dividido entre la separación de los centros de los cuadrantes (T) al cuadrado.
- n_0: "Initial number of quadrats" o Número inicial de cuadrantes y se puede entender como la anchura por la altura de la imagen divididas entre T^2.

El procedimiento a seguir para la estimación con CountEm es el siguiente:

Primero un paso opcional, pero recomendable, es recortar las zonas visiblemente vacías de partículas.

Cargar la imagen de la que se desean estimar sus partículas y elegir buenos valores para f y n_0, los valores por defecto de $f = 0.04$ y $n_0 = 100$ son una buena opción.

Figura 3: Ventana de selección de parámetros de la aplicación CountEm con la imagen de espectadores cargada.

Como vemos en la imagen en la parte inferior central nos permite elegir los parámetros de la rejilla, cuya selección queremos automatizar en este proyecto con alguno de los métodos que veremos más adelante. Además, también nos permite elegir otros parámetros de menor importancia como el ángulo de rotación que, para reducir la varianza, se recomienda usar un valor que evite el alineamiento entre la rejilla y los patrones de la población. Una vez establecidos los parámetros pulsamos "Process" para generar una rejilla que se superpondrá en la imagen, hay que tener en cuenta que cada vez que se ejecuta la rejilla se coloca en una posición al azar. Una vez generada la rejilla pasaremos a la siguiente fase.

Contaremos manualmente el número total de partículas muestreadas por cada ventana de muestreo, siguiendo la regla de la línea prohibida como se ha explicado antes, para asegurar que la estimación no esté sesgada.

Figura 4: Ventana de CountEm con la rejilla superpuesta sobre la imagen.

Pulsaremos en la opción "Image view" para entrar en el modo de visualización de cuadrantes, lo que nos permitirá tener una mejor vista de las ventanas de muestreo que tendremos que contar manualmente.

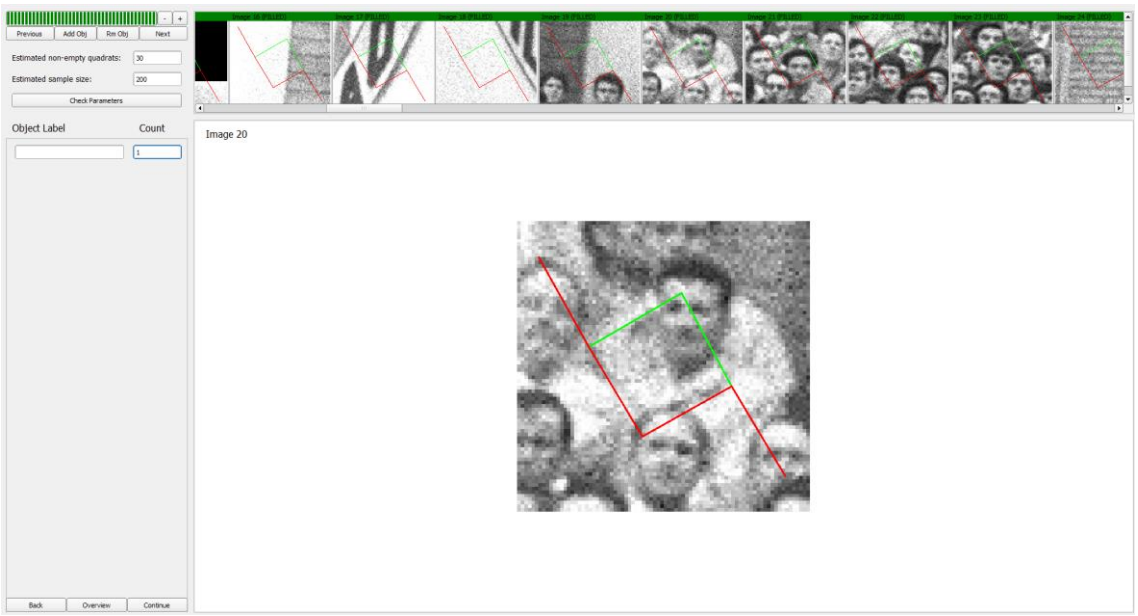

Figura 5: Modo de CountEm de visualización detallada de las ventanas de muestreo de la rejilla.

Iremos desplazándonos entre cuadrantes anotando el conteo para cada uno en el cuadro de texto donde pone "Count", por ejemplo, en la ventana de muestreo de la **Figura 5** el conteo es de 1, dado que la cara del espectador está parcialmente dentro del cuadrante tocando solo la línea permitida (verde). La barra de progreso de la esquina superior izquierda se ira llenando y una vez contemos todas las ventanas de muestreo se volverá verde, como se ve en la misma imagen, indicándonos que podemos proceder al siguiente paso pulsando "Continue".

Object Label	Estimated number of visible particles (estN):	Number of particles counted in the quadrats:	Number of non empty quadrats:	Predicted standard error of estN:	Predicted coefficient of error of estN (%):
Spectators	1075.0	43.0	37.0	92.55	8.61

Figura 6: Ventana de CountEm con la estimación del tamaño de la población de la imagen de los espectadores.

Por último, CountEm nos mostrará en pantalla el número estimado de partículas totales $\widehat{N}$ que es aproximadamente la fracción de muestreo multiplicada por el número de partículas totales contadas en todos los cuadrantes Q.

$$\widehat{N} = f.Q = \frac{t^2}{T^2}.Q$$

También podremos ver el error estándar relativo predicho mediante una formula descrita en [2]. Vemos que el error predicho se encuentra en el rango de error aceptable del programa del 5-10%.

Comprobamos el número real de personas usando un archivo de la página de CountEm donde se indican las posiciones de cada espectador en la imagen, es decir, una fila por cada espectador, dando un total de 1120 espectadores en la imagen real. El coeficiente de error entre la estimación de CountEm (1075) y la cantidad total es de un 4%.

Hay que tener en cuenta que como la rejilla se posiciona al azar sobre la imagen cada vez que ejecutamos CountEm la estimación del tamaño de la población nos dará distinta, si promediáramos miles de estimaciones con CountEm acabaríamos obteniendo el número exacto de partículas, dado que es un método insesgado. Pero como el error tiene dos componentes: varianza y sesgo, aunque el sesgo sea cero todavía hay un error asociado a la varianza de muestreo. Cuantificación de la calidad de la estimación mediante el error cuadrático medio [3].

$$\mathbb{E}(\widehat{N} - N) = Var(\widehat{N}) + \left(\mathbb{E}(\widehat{N} - N)\right)^2$$

$$Error\ cuadrático\ medio = Varianza + Sesgo^2$$

Al simular 1000 veces acabaríamos hallando el sesgo empírico que será prácticamente 0, por eso es insesgado, entonces el coeficiente de error solo es la varianza de muestreo.

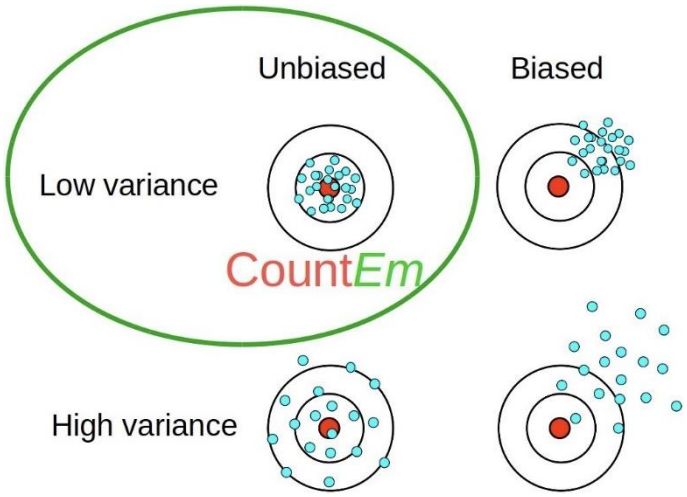

Figura 7: Diagrama varianza/sesgo, donde el círculo rojo sería la medida real y los azules las distintas estimaciones. Marcos Cruz CC-BY.

Que el coeficiente de error solo sea la varianza es una ventaja porque se puede predecir, a diferencia del sesgo, con un solo resultado, siendo el coeficiente que predice CountEm (8.61% para nuestra estimación en la **Figura 6**).

En el caso de este ejemplo de espectadores dicha simulación ya se hizo [2] para ciertos parámetros en los que en las ventanas de muestreo no vacías de la rejilla se contaban 200 partículas a mano y se obtuvo un coeficiente de error de 5%, siendo algo mayor que el 4% de la estimación de la población respecto a la real, pero menor a la predicha.

En general el método CountEm proporciona unos resultados razonables en pocos minutos para estimar número total de partículas en una imagen, con el inconveniente de que hay que estimar a mano ciertas ventanas de muestreo. Se podría decir que es un programa que proporciona la rejilla de ventanas de muestreo con la que se realiza la estimación, por lo que necesita de un componente visual humano. En este trabajo pretendemos testar algún método automático que puede combinarse con CountEm para mejorar ese aspecto.

2.2. Métodos de procesamiento digital de imágenes

El procesamiento digital de imágenes consiste en algoritmos de computación aplicados sobre una imagen con el fin de modificarla, normalmente mejorarla, o de obtener información de ella, como ya hemos comentado para este proyecto nos interesa lo último. En general es un tipo de procesamiento de señal en el cual la entrada (input) es una imagen y la salida (output) es la imagen modificada o unas características asociadas a la imagen de entrada.

El impacto que han tenido en los últimos años las técnicas computacionales en los campos de las ciencias de la información ha sido amplio y profundo, no es distinto para el campo concreto del procesamiento digital de imágenes, siendo una de las tecnologías que más rápidamente ha avanzado en la actualidad, la cual se ha empleado para la manipulación de imágenes en un amplio rango de aplicaciones científicas, convirtiéndose en un área principal de investigación dentro de los campos de ingeniería y ciencia de computación.

El procesamiento de imágenes puede desglosarse en los siguientes pasos:

1. Cargar la imagen en el espacio de trabajo
2. Analizar y manipular la imagen, incluyendo procesados previos que se le quieran realizar.
3. Obtención de la salida, que puede ser una imagen modificada o unas características basadas en un análisis a la imagen.

Primero, para comprender el procesamiento digital de imágenes, tenemos que entender como es una imagen en la computación. Una imagen es definida como una función de dos dimensiones $F(x, y)$, donde x e y son coordenadas espaciales, y la amplitud de F para cualquier conjunto de coordenadas (x, y) se llama *intensidad* de la imagen en ese punto. Llamamos imagen digital a una imagen cuando sus pares de coordenadas x e y, y su valor de amplitud de F son finitos y discretos.

Es decir, una imagen se puede expresar con un vector de dos dimensiones, o matriz. Y en concreto una imagen digital se compone de un número finito de elementos, dado que los valores de sus coordenadas x e y son finitas y discretas, y en donde cada uno de esos elementos tiene un valor específico, finito y discreto. Conocemos esos elementos como *pixeles* de la imagen.

A continuación, podemos ver como se representaría una imagen como una matriz, siendo W y H la anchura y altura de la imagen respectivamente.

$$F(x,y) = \begin{bmatrix} F(0,0) & F(0,1) & \cdots & F(0,W-1) \\ F(1,0) & F(1,1) & \cdots & F(1,W-1) \\ \vdots & \vdots & \ddots & \vdots \\ F(H-1,0) & \cdots & \cdots & F(H-1,W-1) \end{bmatrix}$$

Dependiendo del rango y tipo de valores finitos y discretos que pueden tomar los píxeles de una imagen digital, podemos clasificar la imagen en diferentes tipos. Estos son los que nos resultarán más interesantes para las técnicas de procesado de imagen que probaremos:

- **Escala de grises (formato 8 bits)**: El valor de cada pixel $F(x,y)$ solo puede tomar valores en un rango $[0,255] \in \mathbb{Z}$, siendo el 0 el color negro, y 255 el color blanco. Este tipo de imágenes tienen 256 tonalidades de grises (incluyendo el blanco y negro).
- **Imagen Binaria**: Los píxeles de este tipo de imagen solo pueden tener 2 valores posibles, 0 (negro) o 1 (blanco). Es común obtener este tipo de imágenes después de aplicar un umbral a una imagen en escala de grises para obtener el primer plano y el fondo de la imagen, esto será una de las bases del método Watershed que estudiaremos más adelante.
- **Imagen a color (formato 16 bits)**: Es el tipo de imagen en el que se encontraran la mayoría de imágenes que queremos procesar, es un formato a color que también se conoce como formato High Color y es soportado por la mayoría de sistemas modernos. El formato de 16 bit se divide en otros 3 formatos que son rojo (Red), verde (Green) y azul (Blue) conocido comúnmente como formato RGB, donde normalmente 5 bits son asignados para cada uno de los componentes rojo y azul haciendo $2^5 = 32$ niveles cada uno y 6 bits al componente verde formando $2^6 = 64$ permutaciones, esto es debido a que el ojo humano es más sensible a este color. Este formato puede formar una paleta de colores de $32 \times 64 \times 32 = 65536$ distintos tipos de colores.

Ahora que ya hemos expuesto los conceptos básicos comenzaremos con las diversas técnicas de procesamiento digital de imágenes para obtener información, Watershed basado en un algoritmo simple, usaremos un enfoque más avanzado del mismo, pero simple, al fin y al cabo, y Class-Agnostic Counting un método basado en aprendizaje automático con una arquitectura compleja de redes neuronales.

El objetivo concreto para estas técnicas de procesamiento de imágenes en este proyecto es la estimación del tamaño de una población de partículas del mismo tipo y ver si pueden ser útiles al método CountEm para estimar aproximadamente el tamaño de la población N y así seleccionar sus parámetros de forma automática.

Primero veremos el método simple de procesamiento digital de imágenes Watershed.

2.2.1. Watershed

El método Watershed es una transformación sobre una imagen en escala de grises cuyo objetivo final es la segmentación de dicha imagen, su nombre hace referencia a un término topográfico conocido en español como línea divisoria de drenaje que establece el límite entre dos cuencas hidrográficas contiguas.

Las metáforas topográficas están bastante bien usadas en lo que respecta a esta técnica, dado que sirven para explicar de una manera más entendible su funcionamiento.

Partimos de que cualquier imagen en escala de grises puede ser entendida como una superficie topográfica donde una alta intensidad de valores para los píxeles serían las cimas y colinas, y una baja intensidad se vería como un valle. La idea es comenzar a llenar cada valle aislado, que en referencia a una matriz 2D de una imagen serían mínimos locales, con agua de distintos colores, que serían las etiquetas de la segmentación, en algún punto mientras se sigan llenando los valles de agua de distintos colores, esas aguas comenzarán a juntarse. Eso se evita poniendo muros donde las aguas se juntan. Este proceso se continua hasta que todos los picos están bajo el agua. Los muros que se han creado nos proporcionan la segmentación final, siendo cada uno de los segmentos de un color distinto.

Aunque en la práctica esta transformación produce una sobre-segmentación debido al ruido o las irregularidades en los gradientes de la imagen (cambio direccional de la intensidad de un color en la imagen, paso de un valle a una cima en el ejemplo topográfico).

Para solucionar esto, la librería OpenCV, que será la que utilizaremos para procesar imágenes con Watershed, ofrece un enfoque del algoritmo Watershed basado en marcadores en el que se le ha de especificar cuáles son los valles que se han de llenar de agua y cuáles no. Básicamente es un enfoque que requiere de un procesado previo de la imagen a segmentar. La idea de este enfoque es dar una etiqueta a la región de la imagen que estamos seguros que son las partículas a contar o el primer plano (foreground), dar otra etiqueta distinta a lo que consideremos seguro que es el fondo (background) y por último etiquetar la región restante que será lo desconocido. La región desconocida valdrá 0, el fondo 1 y cada uno de los elementos del primer plano que estén separados recibirán una etiqueta distinta denotando que son distintas partículas o segmentos. Estas etiquetas representan a los valles que se han especificado para llenarse de agua de distintos colores y juntarse cuando se aplique Watershed en la metáfora topográfica, es la clave de este enfoque. Al final eso será nuestro marcador al que aplicaremos Watershed y nos devolverá un marcador actualizado con las etiquetas que pusimos y las fronteras entre los segmentos etiquetadas con el valor -1.

La idea general detrás de Watershed y el enfoque basado en marcadores está explicado gráficamente, siguiendo la metáfora topográfica, y con algunas animaciones del proceso de llenado de agua de los valles en [4].

Ahora veremos un ejemplo de cómo aplicar este enfoque de Watershed mediante la implementación de OpenCV a una imagen bastante simple de monedas.

Figura 8: Imagen de monedas para el ejemplo básico de Watershed

Se puede ver que esta imagen tiene un buen contraste entre el fondo y el primer plano, además no hay elementos superpuestos, como única dificultad es que los objetos están tocándose mutuamente.

Como se ha comentado para tener las bases de Watershed se necesita la imagen en escala de grises, por lo que la convertiremos. Además, en el enfoque basado en marcadores se necesita definir el fondo y el primer plano, por lo que tendremos que convertir la imagen en escala de grises a una imagen binaria, con solo dos valores de pixeles 0 y 1 (negro y blanco) aplicando un método de umbralización. Existen varios métodos de umbralización, pero uno conocido que usaremos en este trabajo para poder aplicar Watershed es la binarización de Otsu, apartado "Image Thresholding" de [5]. Sin entrar en detalles, lo que hace Otsu es minimizar la varianza entre los pixeles en blanco y negro con un umbral, la ventaja de Otsu es que elige dicho umbral automáticamente.

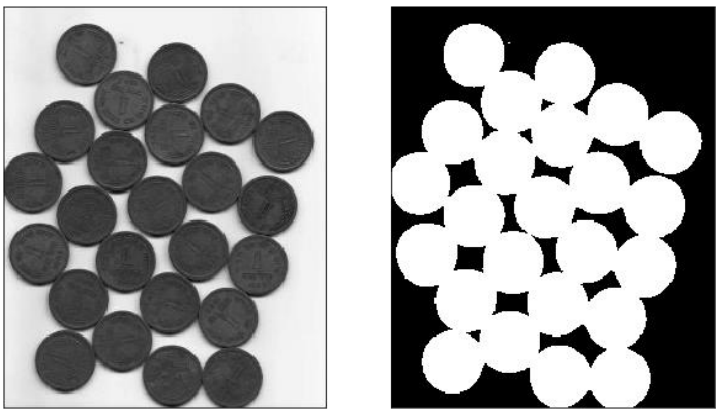

Figura 9: Imagen de ejemplo de monedas en los formatos: escala de grises (izquierda) y binaria (derecha)

A continuación, se eliminan los ruidos blancos de la imagen binaria, en este caso no hay demasiado ruido, pero se puede apreciar un poco a la derecha de la moneda más superior, este tipo de ruido nos puede producir algún falso positivo a la hora de segmentar la imagen. Para quitar el ruido se pueden aplicar operaciones morfológicas, las cuales usarán una matriz de kernel para operar con los pixeles de la imagen, en el ejemplo más avanzado explicaremos estos conceptos con más detalle. En este caso es recomendable usar la operación morfológica llamada *opening* que lo que hace es

erosionar (contraer) las partes de más intensidad (blancas) de la imagen y después dilatarlas (expandirlas), lo que en el proceso hace desaparecer el pequeño ruido blanco y deja las partes blancas más grandes como estaban.

Una vez tengamos la imagen binaria con el mínimo ruido posible tendremos que definir lo que consideramos seguro que es el primer plano y el fondo.

Para el fondo es normal usar operaciones morfológicas de *dilatación*, lo que expandirá la zona del primer plano más allá de sus fronteras, por lo tanto, estaremos seguros de que lo que queda del fondo es el fondo real.

Por otra parte, para definir de manera segura el primer plano se podría usar un método de *erosión* que contraerá la zona del primer plano más adentro de sus límites, dejando un primer plano que se podría decir que es seguro que es el primer plano, pero tiene el problema de que las partículas se tocan, por lo que funcionará incorrectamente, se muestra en la siguiente imagen.

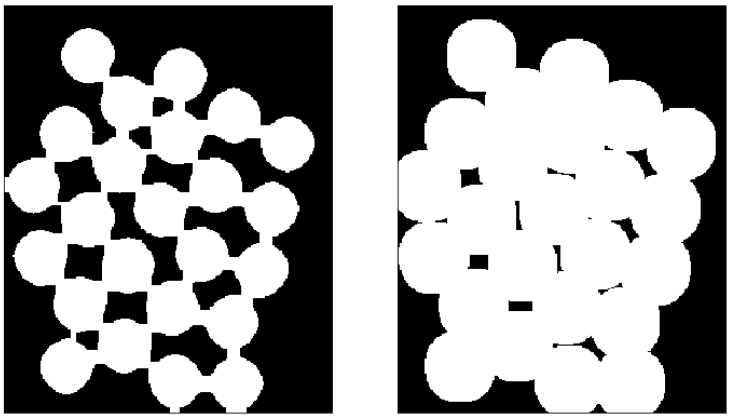

Figura 10: Lo que es seguro que es el primer plano mediante *erosión* (izquierda) y el fondo mediante *dilatación* (derecha)

Se observa que lo que es seguro que es el primer plano ha mantenido unidos sus distintos elementos, después de la *erosión*, debido a su contigüidad. Para solucionar esto aplicamos, en vez de la *erosión*, la *transformada de la distancia euclídea*, esta operación obtiene la distancia euclídea de cada uno de los pixeles del primer plano al pixel más próximo del fondo.

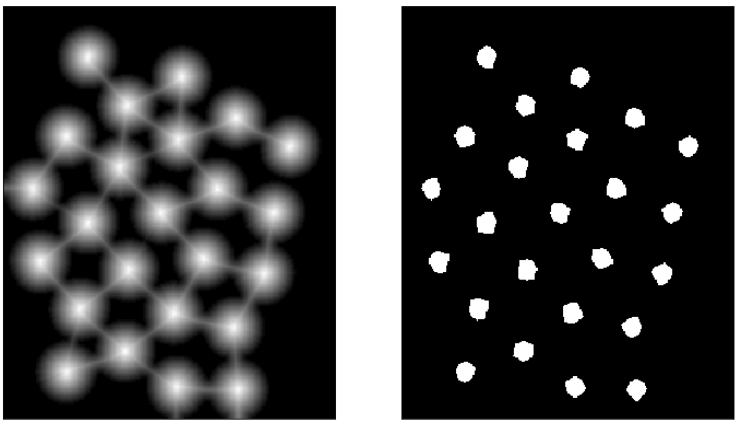

Figura 11: La transformada de la distancia euclídea de la imagen binaria (izquierda) y lo que es seguro que es el primer plano después de umbralizarla (derecha).

Podemos ver cómo después de aplicar un umbral para volver a convertir a binaria a la transformada de la distancia euclídea obtenemos lo que podemos asegurar que es el primer plano con todos sus elementos separados.

Ahora creamos el marcador del que se ha hablado al principio y que es la base de este enfoque de Watershed. El marcador es una matriz de las mismas dimensiones que la imagen original, y tendrá etiquetados los distintos elementos del primer plano y el fondo con valores numéricos distintos, aparte de que también etiquetaremos en el marcador la zona desconocida, que se obtendrá como la diferencia entre lo que es seguro que es el primer plano y lo que es seguro que es el fondo.

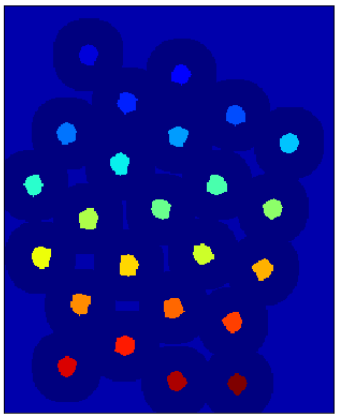

Figura 12: Marcador inicial de la imagen de ejemplo de monedas.

En la imagen se ve gráficamente como sería el marcador, siendo las zonas de azul oscuro entre el fondo y los elementos la zona desconocida.

Aplicamos finalmente Watershed pasándole como parámetros la imagen original y el marcador que hemos creado y nos devuelve el marcador actualizado. Terminando con la metáfora topográfica, los valles indicados por la zona que aseguramos que era el primer plano hasta sus cimas (límite de la zona desconocida con el fondo) se llenaron de agua de distintos colores hasta juntarse y se construyeron los muros separando dichas aguas en la cima. Esos muros forman los límites de la segmentación indicados con el valor -1 en el marcador.

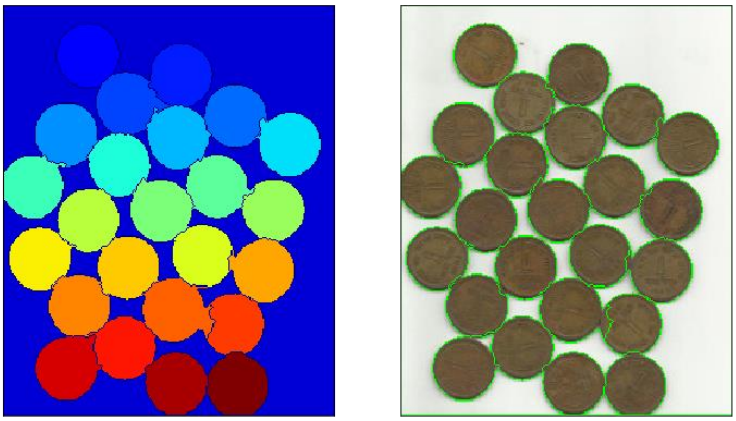

Figura 13: Marcador actualizado por Watershed (izquierda) y la imagen original con las fronteras del marcador actualizado marcadas (derecha)

Quizá la frontera de la segmentación resultante tiene sus imperfecciones, pero no es lo que nos importa en este proyecto, para contar las partículas obtenidas por Watershed simplemente contamos el número de etiquetas distintas del marcador actualizado, dado que cada etiqueta del marcador es una partícula diferente y le restamos dos, una por la etiqueta del fondo y otra por la etiqueta que define la frontera entre el fondo y las demás partículas.

El resultado son 24 monedas contadas, siendo 24 monedas en la imagen real, por lo que tiene un coeficiente de error del 0% para esta imagen tan simple, pero es una imagen que se puede contar prácticamente sin dificultad a mano, para ver su utilidad para estimar la población N veremos un ejemplo más avanzado.

Este ejemplo con monedas se ha conseguido del tutorial del apartado “Image Segmentation with Watershed Algorithm” de [5].

Para finalizar este método de conteo, lo probaremos con un ejemplo más complicado, una bandada de aves, que no tienen relación con el conjunto de datos de aves que se utilizará para obtener los resultados finales y hacer la comparación.

Figura 14: Imagen de una bandada de aves para el ejemplo complejo de Watershed.

Claramente esta imagen, aunque tenga un fondo plano y homogéneo como la imagen anterior, sus partículas del primer plano tienen una complejidad mayor, no solo por sus distintas posiciones o sus tonalidades de colores más variadas, sino también por su superposición entre ellas, antes la dificultad estaba en que las partículas a detectar solo se tocaban, pero ahora también están solapadas.

Probamos a seguir los pasos anteriores: conversión a escala de grises y umbralización de Otsu.

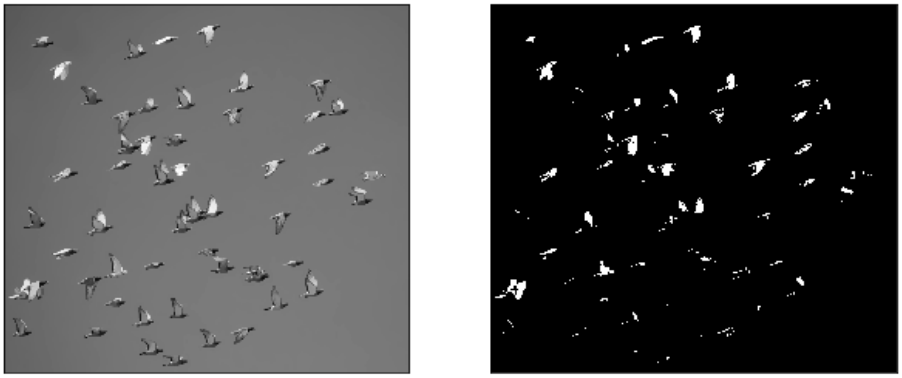

Figura 15: Escala de grises (izquierda) y formato binario (derecha).

Se puede apreciar que la binarización de la imagen no ha sido bastante buena para detectar las distintas partículas. Proseguimos definiendo lo que es seguro que es el primer plano y el fondo, obtenemos los marcadores y aplicamos Watershed obteniendo los nuevos marcadores con la frontera.

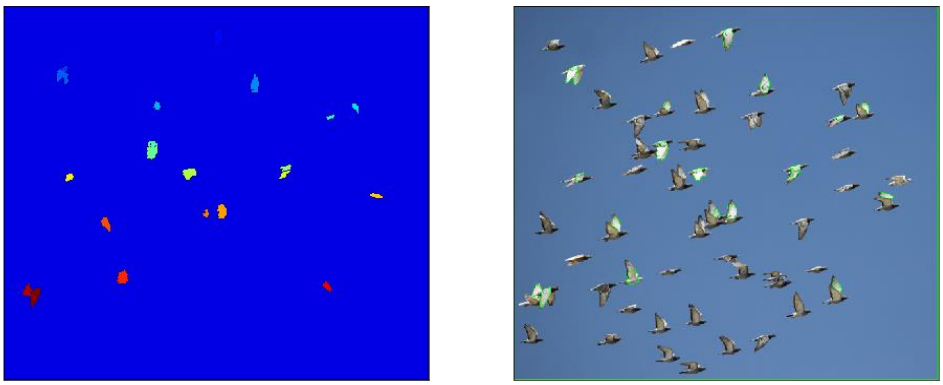

Figura 16: Marcador actualizado por Watershed (izquierda) y la imagen original con la frontera marcada (derecha)

La segmentación ha detectado bastante mal los elementos, expresándolo empíricamente ha detectado 21 elementos de los 56 que son realmente, obteniendo un error del 62.5%. El principal fallo está en las conversiones iniciales de la imagen dado que al aplicar el umbral de Otsu, después de pasarla a escala de grises, hace que el tono azul oscuro del fondo y las partes con tonos oscuros de algunos pájaros sean detectados ambos como fondo. Con un resultado como este quedaría descartado para servir de apoyo a CountEm, pero este método todavía tiene bastante margen de mejora si se profundiza más en el procesamiento previo de imágenes.

Ahora intentaremos mejorar ese resultando aplicando transformaciones morfológicas que hemos mencionado antes.

Las transformaciones morfológicas son operaciones simples basadas en la forma de la imagen que necesitan dos parámetros de entrada, la imagen a procesar y el kernel, elemento estructurante o matriz convolucional que definirá la naturaleza de la operación. Ya hemos visto antes las dos operaciones morfológicas básicas: dilatación y erosión.

Los kernels, al igual que la imagen sobre la que operan, son arrays de dos dimensiones (los más básicos de 3×3 generalmente) que funcionan operando en los valores asociados a los pixeles con operaciones aritméticas directas. En el siguiente ejemplo vemos un kernel para difuminado de imágenes (blur o Gaussian blur) como el multiplicando de la izquierda, después se toma un pixel de la imagen (de *intensidad* 85 en el ejemplo) y se hace coincidir su dimensión con la del kernel, es decir tomando los valores de los píxeles a su alrededor formando una matriz, siendo el multiplicando de la derecha. Los valores de la matriz resultante de la multiplicación se suman entre ellos y el nuevo valor es el valor del pixel transformado.

$$\begin{pmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{pmatrix} \times \begin{pmatrix} 103 & 81 & 94 \\ 83 & 85 & 88 \\ 126 & 108 & 110 \end{pmatrix} = \begin{pmatrix} 103 & 81 & 94 \\ \times 0.0625 & \times 0.125 & \times 0.0625 \\ 83 & 85 & 88 \\ \times 0.125 & \times 0.25 & \times 0.125 \\ 126 & 108 & 110 \\ \times 0.0625 & \times 0.125 & \times 0.0625 \end{pmatrix}$$

$$= 6.43 + 10.12 + 5.87 + 10.37 + 21.25 + 11 + 7.87 + 13.5 + 6.87 = 93$$

Este proceso se hace para cada uno de los pixeles de la imagen que se quiere procesar. En [6] se puede apreciar esta operación de manera gráfica e interactiva.

Para operar y transformar pixeles se puede usar cualquier matriz como kernel, pero si se quiere conseguir un efecto en concreto en la imagen se deben elegir los valores cuidadosamente. Existen kernels ya definidos para distintos objetivos: Difuminación de la imagen, detección de bordes, acentuación de los detalles, por nombrar los más conocidos.

- Difuminado gausiano (Gaussian blur): $\frac{1}{16}\begin{pmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{pmatrix}$ kernel del ejemplo anterior.
- Acentuación (sharpen): $\begin{pmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{pmatrix}$
- Detección de bordes (edge detection): $\begin{pmatrix}1 & 0 & -1\\ 0 & 0 & 0\\ -1 & 0 & 1\end{pmatrix}$
- Difuminado gausiano 5x5: $\frac{1}{256}\begin{pmatrix}0 & 1 & 2 & 1 & 0\\ 1 & 4 & 8 & 4 & 1\\ 2 & 8 & 16 & 8 & 2\\ 1 & 4 & 8 & 4 & 1\\ 0 & 1 & 2 & 1 & 0\end{pmatrix}$

En la anterior prueba con la bandada de pájaros se usaban operaciones morfológicas básicas con un kernel simple de 3x3 de todo unos, que no dio buenos resultados.

Para mejorar esos resultados se procesará previamente la imagen antes de realizar la umbralización de Otsu. Aplicaremos una transformación morfológica vista antes, *opening* con un kernel gaussiano 5x5, lo que suavizará los pájaros de la imagen dejando el fondo prácticamente igual.

Figura 17: Imagen de una bandada de aves suavizada mediante una operación morfológica.

Como el fondo es plano se ha mantenido igual, pero los pájaros ya están suavizados. Obteniendo la diferencia entre la original y esta imagen conseguiremos un fondo negro, dado que las diferencias entre los fondos de las dos imágenes serán valores muy bajos,

mientras que los pájaros sí que habrán sufrido una transformación significativa, por lo que al hacer la diferencia resaltarán.

Figura 18: Diferencia entre la imagen de la bandada de aves original y la suavizada

Esto ayuda a la binarización ya que en vez de un fondo de tono azul oscuro ahora es negro lo que hará más fácil umbralizar la imagen con mayor precisión.

Ahora observamos la nueva imagen binaria.

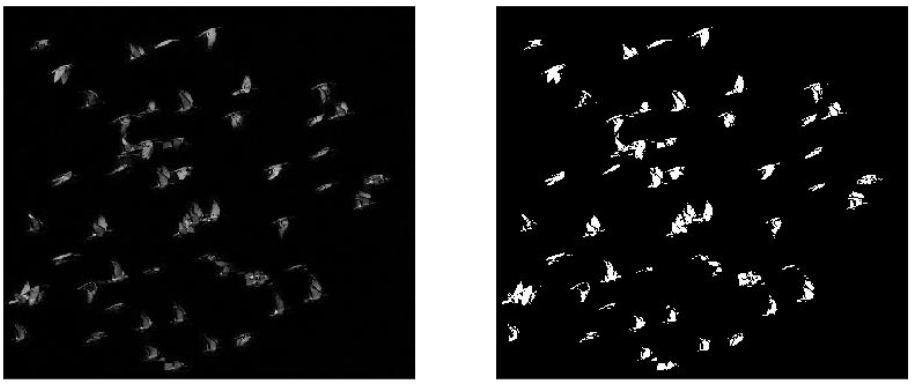

Figura 19: Imagen en escala de grises (izquierda) y en formato binario (derecha).

Se puede observar una clara mejoría en el umbral a la hora de diferenciar los pájaros del fondo habiendo procesado antes la imagen.

Aun así, se observa mucho ruido en las detecciones, para definir lo que es seguro que es fondo y primer plano deberemos realizar otro procesado algo más complejo que solo aplicar *dilatación* y *erosión*, y el *opening* con kernel simple para quitar ruido blanco.

Ya se ha hablado de lo que son las operaciones morfológicas y de algunas operaciones en concreto como *dilatación* y *erosión*. La operación morfológica que utilizaremos para el procesado de obtener el fondo es *closing* que consiste en aplicar *dilatación* primero y después *erosión*. Esto consigue eliminar el ruido e imperfecciones dentro del primer plano (lo blanco), es decir se expande hasta cerrar todos los agujeros de las figuras detectadas en la umbralización y luego vuelve al tamaño normal con los agujeros cerrados, es justo lo contrario a *opening*. Una vez cerradas las imperfecciones aplicamos una *dilatación* para obtener lo que sabremos seguro que es el fondo. Por otro lado, para obtener el primer plano primero utilizaremos la operación *gradient*, la cual se trata de la diferencia entre *erosión* y *dilatación*, lo que nos da los bordes de los elementos del primer plano y luego aplicamos *closing* para cerrar dichos bordes, obteniendo el primer plano sin ruido exterior ni interior. Por último, hacemos el proceso de la transformada

de la distancia euclídea para definir lo que es seguro que es primer plano. Si se está interesado en conocer más sobre las operaciones morfológicas se puede consultar el apartado “Morphological Transformations” de [5].

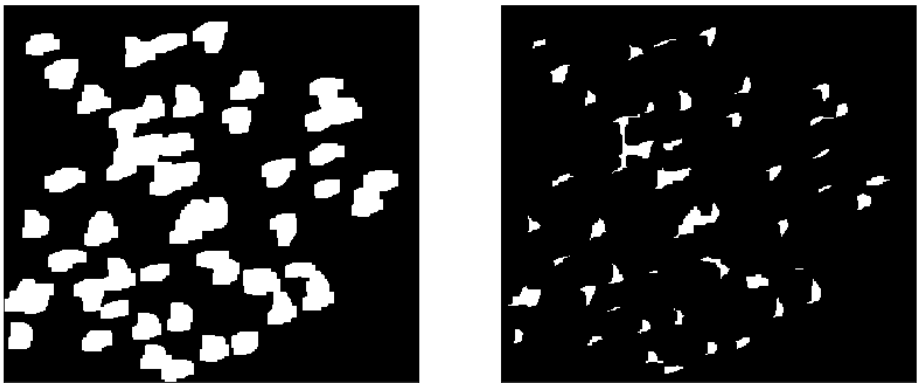

Figura 20: Lo que es seguro que es el fondo (izquierda) y el primer plano (derecha) habiendo aplicado transformaciones morfológicas más complejas.

Para finalizar obtenemos la región desconocida junto con los marcadores a los que aplicaremos el algoritmo de Watershed que los actualizará con la segmentación final.

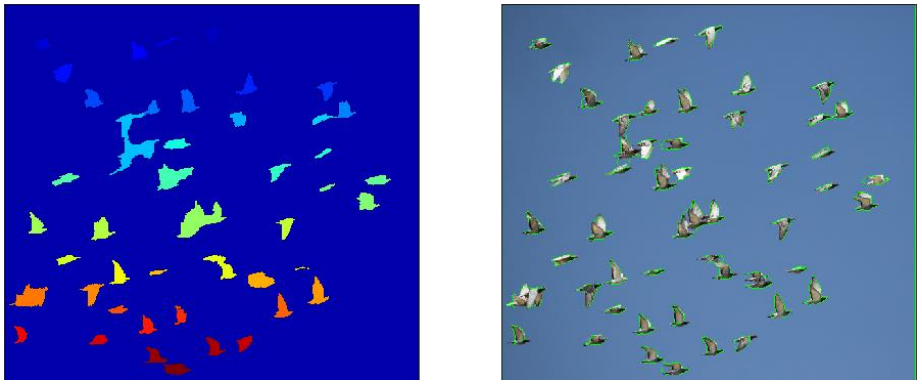

Figura 21: Marcador actualizado con las partículas detectadas por Watershed (izquierda) y la imagen original con la nueva frontera marcada (derecha).

Podemos apreciar como ahora Watershed ha obtenido una frontera considerablemente mejor definida que antes. En esta ocasión Watershed ha contado 49 elementos o aves distintas de 56 que son en realidad dando un error del 12.5% una reducción impresionante respecto al 62.5% obtenido con la imagen sin aplicar ningún procesado previo. Esto demuestra la gran importancia de tratar la imagen correctamente antes de intentar aplicar el algoritmo Watershed sobre ella.

En lo que respecta al objetivo de este proyecto, este método es poco viable para ser de utilidad como apoyo para la automatización de CountEm. Lo que queremos es un método que estime la población de partículas en una imagen para así poder automatizar la elección de los parámetros de la rejilla de CountEm. Si para automatizar los parámetros de la rejilla necesitamos una estimación de la población N y dicha estimación se obtiene mediante Watershed que, para ofrecer unos buenos resultados, y tampoco es que sean excelentes, necesita de un procesado previo elaborado e individual, entonces lo estaría complicando más, no haciéndolo más automático.

Todo el código de estas pruebas, y otras adicionales de monedas, se puede encontrar en el repositorio [7] y en los archivos adjuntos en formato de Jupyter Notebook.

Lo siguiente será adentrarnos en el método basado en aprendizaje automático y ver si cumple nuestras expectativas como método estimador de N para apoyar a CountEm.

2.2.2. Class-Agnostic Counting

La mayoría de métodos basados en machine learning o aprendizaje automático para procesamiento de imágenes, suelen ser algoritmos para detectar ciertas clases de objetos concretos como coches, personas, células, etc, pero están limitados diferenciar una clase concreta de instancias. Por eso este método llamado Class-Agnostic Counting (CAC) [8], elegido para observar si se puede conseguir un buen resultado estimando el conteo de partículas de un mismo tipo en imágenes, es "class-agnostic". Esto quiere decir que no está creado para ninguna clase de objetos en específico, tiene una capacidad de generalización para detectar objetos que se le pasen como muestra y se le podrá especializar en la detección de ciertas clases de objetos más concretas.

A diferencia de Watershed que es un método conocido incluido en librerías de procesamiento de imágenes, CAC es un proyecto cuyo código se encuentra disponible en Github [9] que usaremos en este proyecto, cabe mencionar que recientemente ha aparecido una implementación en PyTorch. Está formado por varios archivos escritos en Python, entre los cuales hay, por nombrar algunos, un archivo principal o main, otro para cargar los datos y un archivo que genera la arquitectura de la red neuronal, lo que hace que en el proyecto las librerías de Keras y Tensorflow jueguen un papel importante.

La red CAC es capaz de contar instancias de objetos con bastante flexibilidad en una imagen proporcionando un parche de ejemplo con la partícula (objeto) de interés, es decir una imagen recortada centrándose en la partícula que se quiere detectar. Esta red se sustenta en el principio de autosimilitud de imágenes, siendo esta la capacidad de que el parche de ejemplo se repita en un cierto grado en la imagen.

Para esto se desarrolla una Generic Matching Network (GMN) que contará instancias detectando los parches autosimilares existentes en la misma imagen respecto al parche de ejemplo. El entrenamiento de esta GMN se aprovecha de la disponibilidad de una gran cantidad de datos de video etiquetado que contiene repeticiones de las mismas instancias con diferentes planos, ángulos y posiciones, perfecto para entrenar una red flexible y genérica. La guía para obtener el dataset genérico para entrenar la GMN se puede encontrar en [10], pero si se quiere saltar este paso el repositorio del proyecto Class-Agnostic Counting [9] ya provee el archivo con los pesos de la GMN entrenada.

CAC ofrece un modo para especializar la detección mediante autosimilitud, en vez de depender de una red genérica, que muchas veces no satisface nuestras necesidades. Esto nos permitirá adaptar el modelo a cosas más concretas que difícilmente han sido tenidas en cuenta para entrenar la red genérica, como conjuntos de células, que será el ejemplo que usaremos para ilustrar el método, o alguna característica concreta dentro de una clase de objetos como las imágenes de las que disponemos para el proyecto, aves de color blanco.

La GMN consiste en tres módulos: "embedding", "matching" y "adapting". En el siguiente esquema tomado del documento de Class-Agnostic Counting [8] se puede ver la arquitectura básica de la red y sus tres módulos.

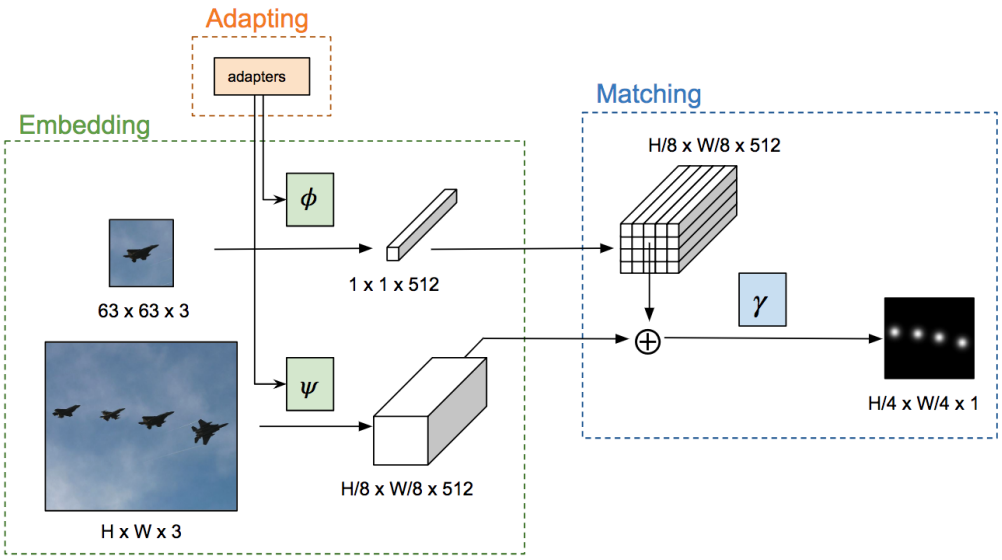

Figura 22: Esquema de la GMN [8], en el que se pueden apreciar los tres módulos en los que está dividida.

En el módulo de "embedding" se crea una red de dos flujos para transformar imágenes RGB en codificaciones de características de alto nivel. Un flujo es para transformar la imagen de ejemplo o parche, de tamaño $63 \times 63 \times 3$ por ejemplo, siendo las dos primeras dimensiones la altura y la anchura, y la tercera la profundidad o los canales (imagen RGB, por lo que tiene 3 canales de colores), en un vector de características de $1 \times 1 \times 512$. El otro flujo se encarga de mapear la imagen completa o de búsqueda $H \times W \times 3$ en un mapa de características $H/8 \times W/8 \times 512$. Estas transformaciones se representan como funciones: ϕ para el flujo del parche y Ψ para el de la imagen de búsqueda, que consisten en varias capas neuronales de convolución, pooling, activación y normalización de batch. Después al resultado de estas funciones se les aplica una normalización L2.

A continuación, en el módulo de "matching" el vector de características es convertido al mismo tamaño de los mapas de características para a continuación ser concatenados, ahora que sus dimensiones coinciden, en un solo flujo. Las operaciones realizadas sobre este flujo se representan como γ que consiste también en varias capas de normalización de batch, convolución, convolución traspuesta y activación. Devolviendo al final como resultado un mapa de calor de dimensiones $H/4 \times W/4 \times 1$.

Esto sería de manera simplificada la arquitectura detrás de la parte genérica de la red. El entrenamiento de la red, tanto en la manera genérica, como la manera especializada se realiza poniendo gaussianas en las posiciones de cada instancia, definidas en un archivo de etiquetas asociado a cada imagen de entrenamiento. Las etiquetas podrían definirse de varias formas, pero la que usa el código de CAC es una imagen de fondo negro que tiene pixeles rojos en las posiciones de cada instancia. Con una función de perdidas MSE (Mean Squared Error) se evalúan las predicciones del modelo.

Una parte importante del entrenamiento es que tanto el parche y la imagen de búsqueda se reescalan, el parche generalmente a dimensiones de 64×64 quedando el objeto de interés en el centro de la imagen reescalada, mientras que las imágenes de búsqueda se reescalan al tamaño definido por la *bounding box* (w, h), generalmente cuanto más grande sea, dará un resultado más claro y preciso, recordemos que las

dimensiones del resultado final son $h/4 \times w/4 \times 1$, pero también más recursos computacionales consumirá.

Por último, comentaremos el módulo de "adapting" que será la que utilicemos en este proyecto para especializar la red. Básicamente al módulo de "embedding" de la GMN se le añaden unos módulos de adaptación, formados por varias redes convolucionales. Partiendo de que la GMN ya tiene sus pesos genéricos entrenados, este módulo congela esos parámetros y solo entrena las capas añadidas por esos módulos de adaptación y las capas de normalización de batch de toda la red.

Si se quiere observar toda la arquitectura de la red utilizada a nivel de código (Keras) se puede observar el archivo *model_factory.py* del repositorio de CAC [9].

Ahora observaremos el funcionamiento de un modelo CAC con un ejemplo. Se ilustrará con un dataset de células VGG disponible en [11] popular para realizar benchmarks de técnicas de machine learning. Consistirá en 200 imágenes de células con sus 200 imágenes de etiquetas correspondientes, dichas etiquetas como ya se ha mencionado consisten en una imagen en negro con pixeles en rojo señalando la posición de las partículas a detectar (células).

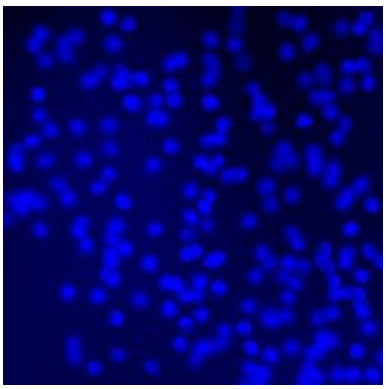

Figura 23: Imagen usada de ejemplo de células VGG número 112 del dataset [11].

El modelo en concreto será entrenado con 100 de estas imágenes y etiquetas asociadas, siendo 80 de estas para entrenamiento y 20 para validación. Las imágenes y etiquetas, y cuáles de estas son de entrenamiento y cuáles de validación (puedes especificar que no haya validación) se definen y pasan al código mediante un archivo .npz, que sirve para guardar listas de vectores de la librería Numpy de Python.

Los parches se reescalarán a tamaño 64×64, mientras que las imágenes de busqueda de tamaño 256×256 se reescalarán a una *bounding box* de $(1024{,}1024)$ resultando en una imagen de salida de $1024/4 \times 1024/4 \times 1 = 256 \times 256 \times 1$, siendo un mapa de calor equivalente en tamaño a su imagen original.

Tendremos que indicarle ciertos parámetros como la tasa de aprendizaje, el número de épocas de entrenamiento, el dataset que utilizaremos (células VGG) y su directorio, y algo importante a especificar es el modo, dado que lo que estamos haciendo es "adapt" y, por tanto, además habrá que especificar el archivo que contiene los pesos ya entrenados del modelo genérico. Un parámetro a tener en cuenta es permitir a Tensorflow, librería encargada de entrenar y ejecutar la red neuronal, usar la GPU de la máquina, es decir, el coprocesador gráfico. Esto es clave, porque usando solo el procesador para tratar toda la carga de procesamiento de imágenes puede llevar mucho

tiempo. Dependiendo del número de épocas y el tamaño de la *bounding box*, podemos estar hablando de más de un mes de entrenamiento, mientras que usando la GPU de una máquina potente que nos fue habilitada por la Universidad de Cantabria para este trabajo, de la que hablaremos en el capítulo 3, no tarda mucho más de un día.

Cuando la red es creada se puede observar la arquitectura de la red con más detalle, como la muestra Keras, en la siguiente imagen. Se aprecian los dos flujos de la red, uno para el parche (azul) y otra para la imagen de búsqueda (rojo) y como se acaban concatenando ambos canales cuando ya tienen las mismas dimensiones en un solo flujo (amarillo).

Layer (type)	Output Shape	Param #	Connected to
image_patch (InputLayer)	(None, 64, 64, 3)	0	
resnet50_patchnet (Model)	(None, 8, 8, 512)	1538624	image_patch[0][0]
global_average_pooling2d_2 (Glo	(None, 512)	0	resnet50_patchnet[1][0]
exemplar_l2 (L2_Normalization_L	(None, 512)	1	global_average_pooling2d_2[0][0]
image (InputLayer)	(None, 1024, 1024, 3	0	
repeat_vector_2 (RepeatVector)	(None, 16384, 512)	0	exemplar_l2[0][0]
resnet_base (Model)	(None, 128, 128, 512	1538624	image[0][0]
reshape_2 (Reshape)	(None, 128, 128, 512	0	repeat_vector_2[0][0]
image_f_l2 (L2_Normalization_La	(None, 128, 128, 512	1	resnet_base[1][0]
concatenate_2 (Concatenate)	(None, 128, 128, 102	0	reshape_2[0][0] image_f_l2[0][0]
upsample_conv_1 (Conv2D)	(None, 128, 128, 256	2359552	concatenate_2[0][0]
upsample_bn_1 (BatchNormalizati	(None, 128, 128, 256	1024	upsample_conv_1[0][0]
activation_91 (Activation)	(None, 128, 128, 256	0	upsample_bn_1[0][0]
upsample_convt_1 (Conv2DTranspo	(None, 256, 256, 256	590080	activation_91[0][0]
upsample_bn_2 (BatchNormalizati	(None, 256, 256, 256	1024	upsample_convt_1[0][0]
activation_92 (Activation)	(None, 256, 256, 256	0	upsample_bn_2[0][0]
output (Conv2D)	(None, 256, 256, 1)	2305	activation_92[0][0]

Figura 24: Modelo de la red neural CAC visualizada por Keras, señalando los distintos flujos en distintos colores.

La arquitectura de la red se creará cada vez que la queramos entrenar o probar, donde se cargarán unos pesos ya entrenados. Procederemos a generar el resultado usando como parche de ejemplo la siguiente imagen.

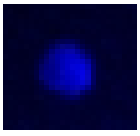

Figura 25: Imagen de ejemplo o parche de una célula VGG.

Preparamos la red adaptada a las células VGG y utilizando como entradas la imagen del grupo de células mostrada antes y el parche que se acaba de mostrar generaremos el mapa de calor correspondiente.

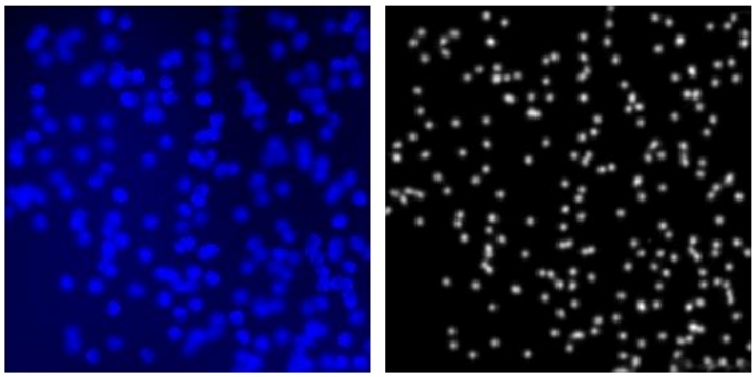

Figura 26: Imagen de células VGG (izquierda) y el mapa de calor resultante después de predecir con la red (derecha).

La imagen que hemos obtenido es una imagen en escala de grises donde se han generado gaussianas en las posiciones en las que ha detectado células en la imagen original. Si ha detectado claramente la célula la gaussiana será más intensa, y más tenue si no se aprecia tan bien.

Teniendo esto en cuenta, para detectar y contar todos los elementos del mapa de calor resultante lo que haremos será aplicar una detección de picos o de máximos locales. Tratándose de gaussianas en 2D se contará como un máximo la zona donde los valores alcancen su valor más intenso estando rodeados de valores que van disminuyendo. Siendo esta detección a nivel local, cada vez que haya una oscilación de valores disminuyendo y aumentando se intentará detectar el pico. En la siguiente imagen, obtenida de [12], se ve una representación en 3D de un mapa de calor de gaussianas que ayuda a entender mejor el concepto que imaginarlo en 2D.

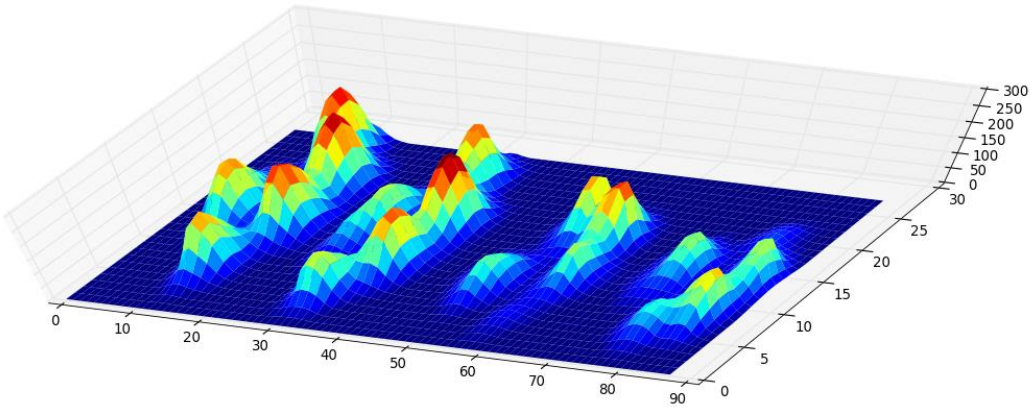

Figura 27: Mapa de calor de gaussianas en 3 dimensiones [12].

Lo que queremos es detectar todos esos picos, habrá algunos menos prominentes (menos intensos) y habrá otros muy cercanos entre ellos que parecerá que casi son el mismo, esto se deberá a que las instancias se solapan en la imagen real, pero si hacemos los parámetros de la detección lo bastante sensibles deberíamos ser capaces de detectar la mayoría.

Para finalizar, se muestra la detección que ha hecho de los elementos o máximos del mapa de calor obtenido anteriormente de la imagen de células VGG.

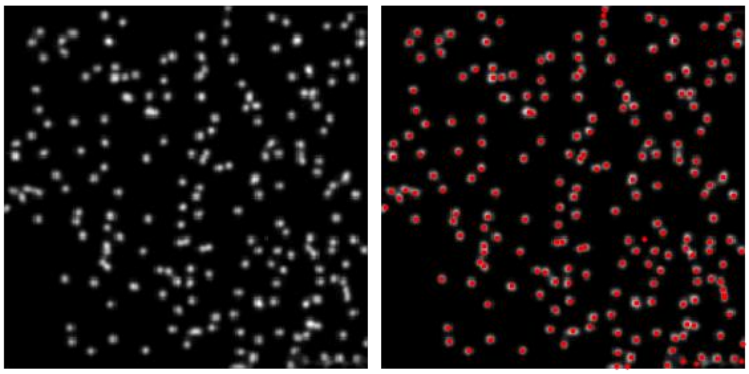

Figura 28: Mapa de calor de una imagen de células VGG (izquierda) y sus máximos detectados (derecha).

Se han detectado 206 picos en el mapa de calor, siendo realmente 221 (cifra obtenida de contar los píxeles distintos de cero en la imagen de etiquetas asociada). El error de la estimación de CAC es del 6.78%. Obtenemos un buen resultado para la complejidad de las imágenes, el problema del solapamiento parece que se maneja bastante mejor que con Watershed, que directamente era incapaz de diferenciar instancias solapadas. Se pueden tunear los dos parámetros de detección de máximos para que se ajuste la sensibilidad de la detección:

- *min_distance* (distancia mínima): mínimo número de píxeles que deberá haber entre todos los máximos para ser detectados.
- *threshold_rel* (umbral): intensidad mínima (valor del pixel) que debe tener un máximo para que se detecte.

En este caso podemos ajustarlos para hacerlos más sensibles y que detecten mejor, aun así, eso no libra al método de detectar algún falso positivo y falso negativo.

A diferencia de Watershed este método si parece que pueda servir como apoyo a la automatización de los parámetros de CountEm. Su estimación de la población N es razonablemente buena y una vez obtenido un modelo adecuado solo lleva unos minutos obtener el resultado, segundos si se dispone de una GPU potente. Mientras que a Watershed el hecho de que para tener buenos resultados hubiese que dedicarle un procesado individual a cada imagen lo hacía inviable para ser un método de apoyo para la automatización de CountEm, en Class-Agnostic Counting solo hay que tunear los dos parámetros mencionados para detectar los máximos del mapa de calor. Además, ambos parámetros son numéricos, por lo que dedicándole tiempo se podría obtener una pareja de valores que obtuviese buenos resultados con la mayoría de las imágenes haciendo una automatización total del método.

Los pesos de la red utilizados para el dataset de células VGG y el código de las pruebas realizadas están disponibles en el repositorio del proyecto [7] y en los archivos adjuntos.

Capítulo 3: Desarrollo

En esta sección comentaremos sin extendernos demasiado las distintas herramientas, aplicaciones, lenguajes, librerías, etc empleados durante la realización de este proyecto.

Como el punto de partida de este trabajo para estimar el tamaño de poblaciones de partículas en imágenes empleamos la aplicación **CountEm** [1] que se trata de un software disponible gratuitamente bajo la licencia de Creative Commons 4.0. Ha sido desarrollado por Javier González-Villa, Marcos Cruz, Domingo Gómez y Luis Manuel Cruz-Orive en el departamento de matemáticas, estadística y ciencia computacional de la Universidad de Cantabria.

Los dos métodos que se analizarán para ver si pueden servir como apoyo a la automatización de CountEm utilizarán **Python** [13] como lenguaje de programación, dado que es muy versátil en el campo de la ciencia de datos por todas las librerías y frameworks que existen en para este ámbito. Muchas de estas librerías servirán para procesar imágenes digitales que usaremos para aplicar el algoritmo de Watershed. Además, el método de Class-Agnostic Counting tiene su código fuente escrito en Python. Cabe mencionar la utilización de **Jupyter Notebooks** [14] para la visualización del código y comentarios de una manera más didáctica. Básicamente es un sistema de computación interactiva que soporta entornos de ejecución en varios lenguajes de programación, entre ellos Python.

En lo que respecta a Watershed, la librería más relevante que utilizamos es **OpenCV** [15], utilizaremos otras librerías, pero la que atañe al procesado de imágenes es esta. Se trata una librería popular desarrollada en C++, pero con conectores a otros lenguajes como Python, que provee herramientas para el procesado de imágenes digitales, todos los métodos de procesado de imágenes usados para las pruebas de Watershed en este proyecto pertenecen a esta librería.

Para el método de aprendizaje automático se ha usado **Class-Agnostic Counting** [8] que es un proyecto desarrollado por Erika Lu, Weidi Xie y Andrew Zisserman cuyo código fuente está disponible en Github [9]. Se trata de una red neuronal avanzada para el procesamiento de imágenes que permite entrenar una red genérica GMN (Generic Matching Network) y también adaptar la red a conjuntos de imágenes específicos. En el propio repositorio hay algunos comandos de ayuda sobre como entrenar y adaptar la red, así como los pesos de la GMN ya entrenados y listos para usarse.

Para que el código fuente del Class-Agnostic counting funcione hay dos librerías que son esenciales: **Keras** [16] y **Tensorflow** [17]. La primera es una API de deep learning que sirve como interfaz para abstraer y hacer más intuitiva la implementación de capas (layers) y modelos de redes neuronales, corre sobre la plataforma de machine learning Tensorflow. Tensorflow es una librería de código abierto que se encarga de construir y entrenar las redes neuronales (creadas a un nivel superior mediante la interfaz Keras). Actualmente Tensorflow da soporte a Keras, por lo que esta se encuentra dentro de la librería de Tensorflow y su propia librería acabará quedándose obsoleta.

El rendimiento de los métodos expuestos hasta el momento se comparará para ver si la estimación de la población que puedan dar mejora a la propia estimación de CountEm y sirve para ayudar en su automatización de parámetros. La comparación se realizará

estimando las partículas (aves) de un dataset de imágenes de bandadas de aves blancas el cual los directores de este trabajo han proporcionado su uso. Dicho conjunto de imágenes se llama **The Eastern Canada (ECA) Flocks** y pertenece al Canadian Wildlife Service que trabajan en conjunto con los directores de este proyecto. El dataset consiste en dos tipos de aves: El eider común para el cual los machos son blancos y las hembras marrones, y el ganso blanco, como indica su nombre de color blanco, esta última será la especie de ave de la cual estimaremos el conteo de su población. Cada imagen del dataset tiene su archivo CSV adjunto en el que se indica para cada ave la posición horizontal (X) y vertical (Y) en la imagen real y su tipo (blanco o marrón). Como estas imágenes se están usando en otra investigación cuentan con un embargo, siendo esta la razón por la que no se han podido subir los notebooks de las pruebas completos al repositorio del trabajo (la parte de los resultados de las imágenes de las aves, como aparecían estas imágenes, ha sido cortada), pero aun así en los archivos adjuntos de este proyecto se pueden encontrar los notebooks completos con todos los resultados.

Por último, comentaremos la máquina en la nube sobre la que se ejecuta Jupyter con la que hemos entrenado y ejecutado los modelos de CAC. Esta máquina cuenta con permisos de superusuario útiles para instalar las librerías necesarias como las mencionadas anteriormente. Aunque, lo principal y más importante es que cuenta con una **GPU NVIDIA V100 TENSOR CORE** [18] que emplea una arquitectura Nvidia Volta ofreciendo un alto rendimiento para aplicaciones de IA y deep learning siendo perfecta para entrenar redes neuronales complejas como la que utiliza CAC. Según los benchmarks realizados este coprocesador gráfico ofrece el rendimiento de hasta 32 CPUs convirtiéndose en una pieza clave para realizar este proyecto. Como ya se ha comentado entrenar el modelo con la CPU de un computador personal tenía un tiempo estimado de alrededor de un mes, convirtiéndolo en un proyecto inviable, utilizando esta máquina nos llevaba alrededor de un día entrenar un modelo CAC.

Capítulo 4: Resultados

En esta sección se expondrán las estimaciones de población obtenidas por CountEm, el método de base, y los métodos Watershed y Class-Agnostic Counting que se pretenden evaluar para comprobar si ofrecen un rendimiento superior en la estimación, que pueda usarse para automatizar la selección de parámetros de CountEm.

Las imágenes con las que usaremos métodos para obtener una estimación de su población de aves serán:

- **GSGO_059_C**: cuenta con pocas aves, sin sombras y un fondo en general homogéneo excepto alguna zona.
- **GSGO_004_C**: cuenta pocas aves, con sombras y un fondo homogéneo.
- **GSGO_019_C**: cuenta con un fondo no homogéneo y pocas aves, pero unas en vuelo y otras en tierra.
- **GSGO_018_C**: cuenta con un fondo homogéneo, pero con muchas aves y algunas se solapan.

Como se ha explicado en el capítulo 2, Watershed necesita de un procesado previo de las imágenes, que dependiendo de la imagen puede variar, por lo que se explicará lo que se ha hecho cuando se comenten los resultados de cada imagen, pero para CAC se genera un modelo con el que se procesan todas las imágenes. Por lo que antes de comenzar a exponer los resultados comentaremos como hemos creado el modelo.

Primero para crear el modelo necesitamos etiquetas asociadas, del mismo tamaño que la imagen original que indiquen la posición de las partículas en la imagen. Pero podemos crear las imágenes de etiquetas fácilmente con un simple programa de Python gracias a los archivos CSV adjuntos a cada imagen que contienen la posición de las aves que aparecen en ellas.

Ahora que ya disponemos de las etiquetas necesitamos un archivo .npz que indique al código que imágenes y etiquetas serán para entrenamiento y cuales para validación, si se quiere usar. En este caso será algo más complicado que con el ejemplo de células VGG, porque las imágenes de bandadas tienen diferentes tamaños entre ellas y una dimensión bastante más grande que 256×256. Con dimensiones demasiado grandes hacer el reescalado de la *bounding box* de 1024×1024 no será muy eficaz dado que se estaría reduciendo su tamaño, tampoco se puede usar una *bounding box* mucho mayor debido a que el computador tiene una memoria de cómputo limitada. La solución a este problema es tratar con imágenes de 256×256, así al aplicar el reescalado inicial a 1024×1024 devolverá un mapa de calor también de 256×256 (las dimensiones de entrada reescaladas entre 4). Por tanto, las imágenes que usemos para entrenar la red, junto con sus respectivas etiquetas se dividirán en bloques de 256×256,mediante otro programa de Python, sin incluir los bloques sin partículas. El programa también generara el archivo .npz que contendrá los bloques de 256×256 para entrenar y validar de las imágenes de bandadas **GSGO_001_C**, **GSGO_008_C**, **GSGO_014_C**, **GSGO_18_C**, **GSGO_019_C** y **GSGO_059_C** y de sus respectivas etiquetas. Las imágenes para entrenar se han escogido por tener una gran variedad de casos, desde imágenes con pocas aves, sombras y fondos homogéneos, hasta imágenes muchas aves y fondos muy complejos. Serán un total de 479 imágenes de 256×256 las que usaremos para entrenar.

Una vez disponemos del dataset de entrenamiento etiquetado, el archivo .npz y las dimensiones de los reescalados definidas, podemos proceder a adaptar la GMN al dataset de aves blancas como ya se hizo para las células VGG.

A la hora de ejecutar el modelo para procesar una imagen nos veremos en un problema similar al del entrenamiento las imágenes son muy grandes, después del reescalado la salida es de 256×256, si una imagen no muy grande del dataset es 1233×834 el mapa de calor resultante tendrá una gran perdida de información. La solución a esto es similar al enfoque del entrenamiento, en el método de ejecución dividiremos la imagen de búsqueda en bloques de 256×256 que uno a uno serán reescalados a la *bounding box* de 1024×1024 y procesados por el modelo de la red CAC, obtendremos como predicción un mapa de calor de las mismas dimensiones al bloque antes del reescalado. Por último, los pedazos del mapa de calor se irán montando hasta formar el mapa de calor completo con dimensiones equivalentes a las de la imagen original. Esta solución presenta un problema derivado, qué ocurre si en la frontera entre dos bloques una partícula queda dividida. Lo que puede ocurrir en este caso es que al detectar media partícula el modelo ponga una gaussiana tenue en el mapa de calor y al unir dichos bloques queden dos gaussianas tenues contiguas produciendo un falso positivo en la detección de máximos.

La solución a este otro problema es tener en consideración una zona de solapamiento en la división de los bloques y en la reconstrucción de los mismos en el mapa de calor.

El enfoque del método de ejecución con solapamiento funciona de manera que cuando se van cogiendo bloques y se pasa al siguiente, en vez de avanzar los índices de los píxeles en 256 (tamaño del bloque), se avanza 256 menos lo que se haya considerado de solapamiento, en nuestro caso será 32 píxeles. El solapamiento debe ser par, para poder hacer la división de la frontera entre los dos bloques correctamente.

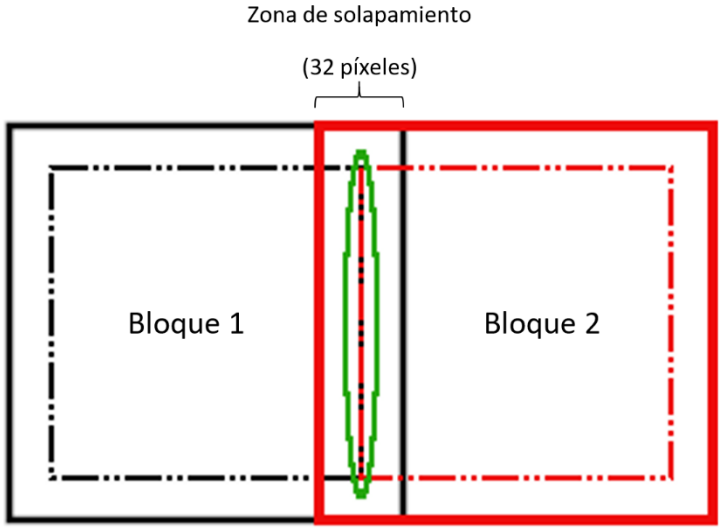

Figura 29: Esquema de la zona de solapamiento para el montaje de los bloques del mapa de calor.

Al montar los pedazos, el nuevo pedazo se montará con un desplazamiento de la mitad del solapamiento (16 píxeles) produciendo que no se tengan en cuenta los 16 píxeles de los bordes conflictivos de ambos bloques y así evitar que surjan dobles gaussianas erróneas en el mapa de calor completo. El algoritmo implementado en el método de ejecución no es tan simple y tiene en cuenta algunas cosas más, como que si es el primer

bloque de la matriz de la imagen (esquina superior izquierda) no se hará ningún desplazamiento, solo lo harán los bloques nuevos que encuentren otro bloque ya montado a su izquierda o arriba suyo o ambos casos a la vez.

Ya tenemos un método de ejecución funcional, una vez solucionados todos los problemas, para Class-Agnostic Counting sobre el dataset de aves blancas. Procedemos a analizar las imágenes y obtener una estimación del tamaño de su población de partículas con los métodos de procesamiento de imágenes digitales. Los resultados serán comparados al final con unos resultados ya obtenidos de CountEm.

La primera imagen a analizar será **GSGO_059_C**, el número de aves que aparecen en esta imagen es de 164, cifra obtenida del CSV asociado a la imagen del dataset (cada fila es un ave).

Figura 30: Imagen **GSGO_059_C**

Primero procesaremos esta imagen para aplicar Watershed, tiene un fondo homogéneo, pero con ciertos detalles que pueden producir mucho ruido en la binarización, por lo antes de pasarlo a escala de grises practicamos un aplanado sobre la imagen.

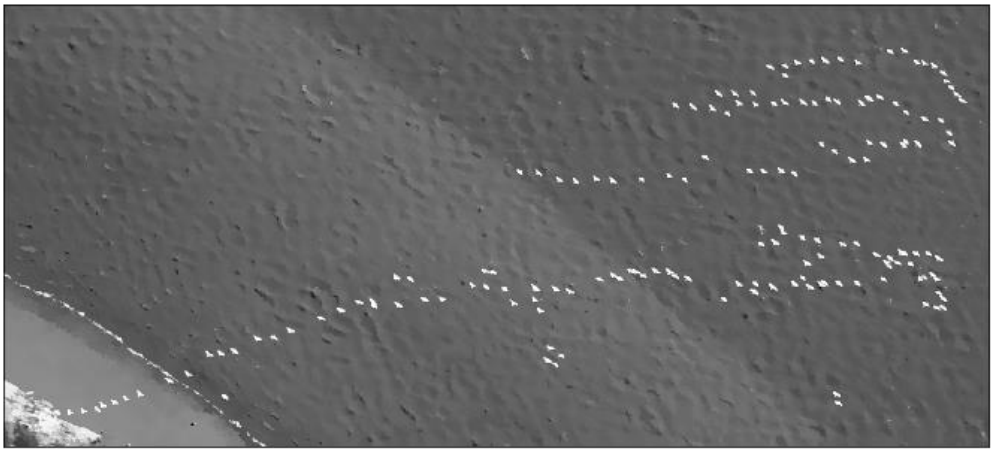

Figura 31: Imagen **GSGO_059_C** en escala de grises después de aplicarla un alisado.

Con esta imagen más lisa y sin tantas imperfecciones podemos umbralizarla para obtener la imagen binaria.

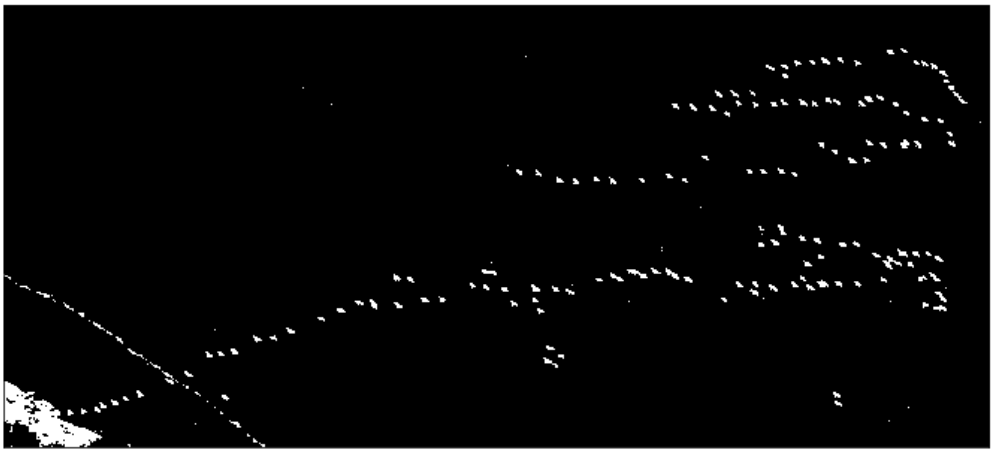

Figura 32: Umbralización de la imagen que se está analizando.

Se puede ver que salvo por la roca de tonos blancos de la esquina inferior izquierda, en general ha diferenciado el fondo y el primer plano razonablemente bien.

Seguidamente quitamos todo el ruido blanco posible de la imagen y obtenemos lo que es seguro que es el fondo y el primer plano y a partir de eso generamos los marcadores. Con los marcadores ya podemos aplicar Watershed para obtener los segmentos (partículas) que ha detectado.

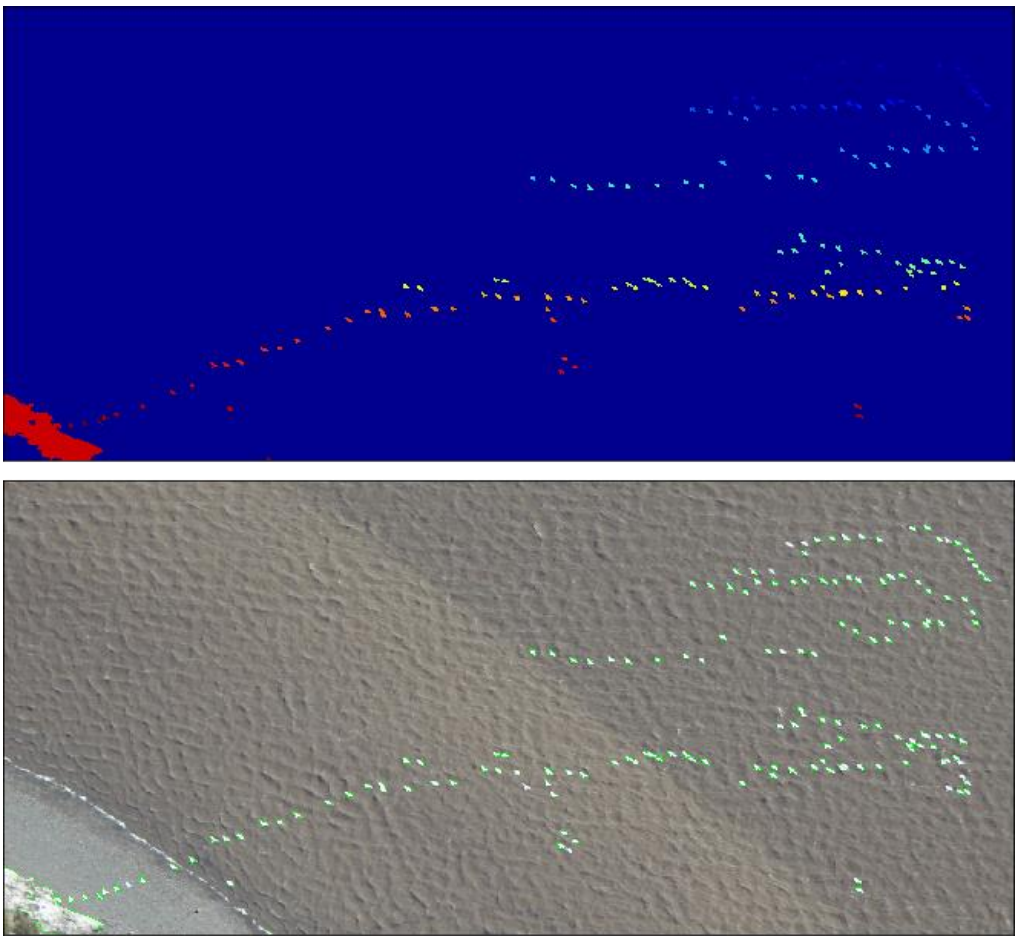

Figura 33: Marcadores de la imagen que se está analizando (arriba) y sus fronteras marcadas en la imagen original (abajo).

Mediante Watershed se han detectado 145 partículas, teniendo en cuenta que el número real es 164, el coeficiente de error de la estimación del tamaño de la población es de 11.58%. A parte de la roca de la esquina inferior izquierda que la pilla como una partícula (falso positivo), se ve unas cuantas aves que no detecta, falsos negativos.

Continuaremos exponiendo los resultados obtenidos por Class-Agnostic Counting para esta imagen.

El parche, la imagen de ejemplo, usado para procesar todas las imágenes de bandadas que se analizaran con la red.

Figura 34: Imagen de ejemplo o parche de un ave blanca.

Generamos el mapa de calor con la red entrenada y la imagen y el parche anterior como entrada.

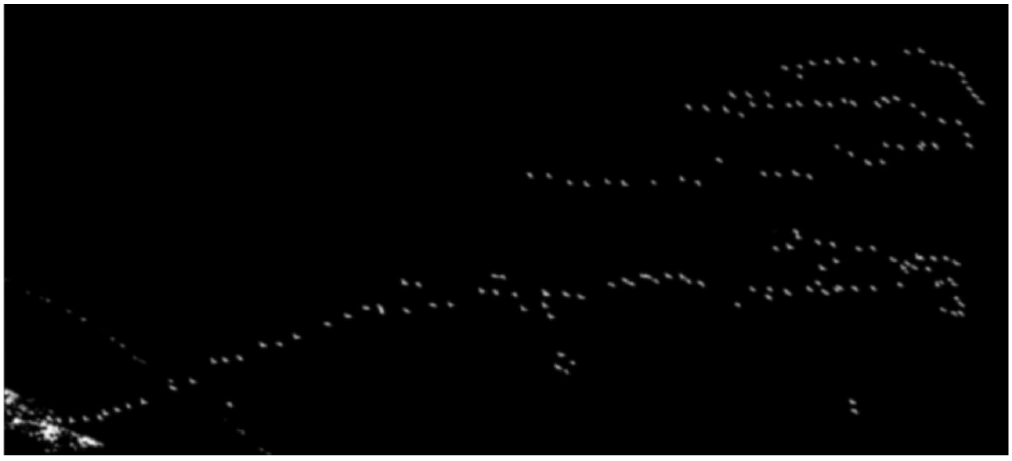

Figura 35: Mapa de calor de la imagen que se está analizando.

Tratamos de detectar los máximos de esas gaussianas del mapa de calor.

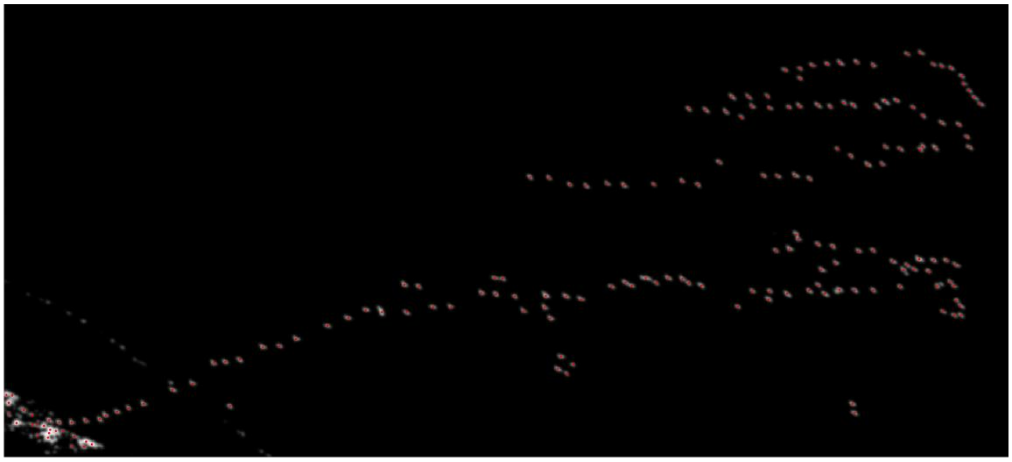

Figura 36: Máximos detectados sobre el mapa de calor de la imagen que se está analizando.

Observamos que la mancha de la parte inferior izquierda causada por la roca de tonalidades blancas ha repercutido en el modelo y nos ha generado falsos positivos. Aunque para reducirlos y dar una mejor estimación se han tuneado los dos parámetros del método de detección de máximos locales, la distancia mínima de 2 píxeles y el umbral de 0.4.

En este caso el número de máximos detectados es de 178 haciendo un coeficiente de error del 8.53%, detectando mejor las aves visibles que Watershed. Aunque ha fallado más que este en la mancha de la esquina inferior izquierda.

La segunda imagen será **GSGO_004_C**, el número de aves que aparecen en realidad en esta imagen es de 601.

Figura 37: Imagen **GSGO_004_C**.

Es una imagen con un fondo homogéneo, muy liso, con el inconveniente de que hay sombras y una gran cantidad de aves. No hará falta prácticamente ningún procesado previo, dado que el contraste entre las partículas, muy blancas, y el resto que son las sombras y el fondo, bastante más oscuros, resultarán en que la umbralización de Otsu sea bastante precisa.

Con la imagen binaria procedemos a asegurar lo que es fondo y primer plano, como ya hemos visto y a crear los marcadores que serán actualizados mediante el algoritmo Watershed para segmentar la imagen con las partículas que ha detectado.

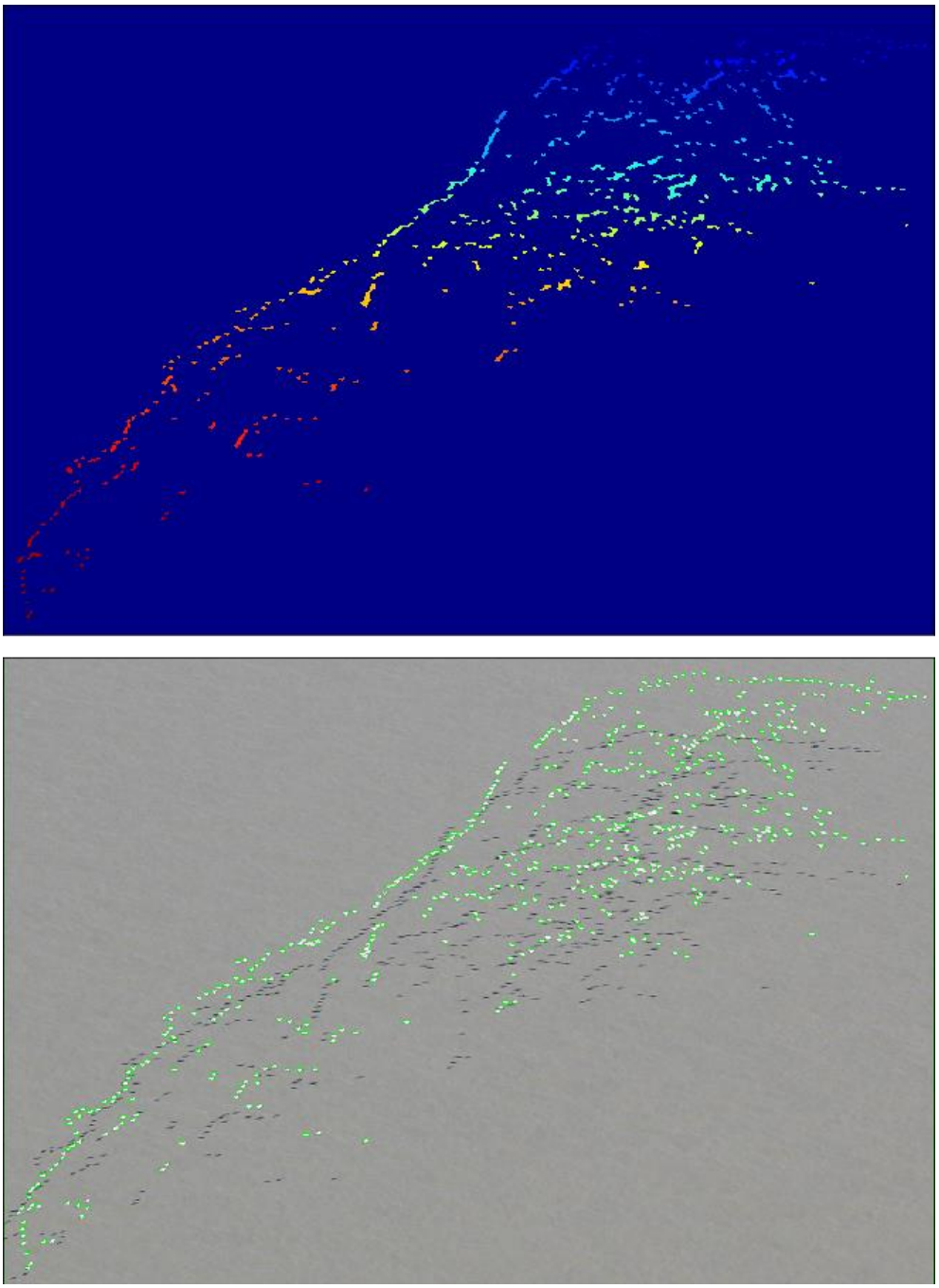

Figura 38: Marcadores de la imagen que se está analizando (arriba) y sus fronteras marcadas en la imagen original (abajo).

Hemos detectado con Watershed 407 partículas, siendo en realidad 601, lo que da un coeficiente de error de 32.27%, un error bastante alto. Al error se debe, aunque no se pueda apreciar con segmentos tan pequeños en la imagen, a que muchas aves están dentro de un solo segmento. En la imagen de los marcadores esto se ve más claramente, porque va por colores, hay una gran cantidad de segmentos que abarcan varias imágenes debido a la contigüidad y solapamiento de las partículas. Esto produce una cantidad muy elevada de falsos negativos.

Ahora el procederemos con el análisis de la misma imagen para CAC, generando el mapa de calor.

Figura 39: Mapa de calor de la imagen que se está analizando.

Se puede apreciar que las gaussianas parecen adaptarse bastante bien a las aves de la imagen original sin ninguna perturbación producida por él fondo. Por lo que para detectar los máximos usaremos una distancia mínima de 1 y un umbral de 0.08 para que sea bastante sensible a detectar todos lo picos.

Figura 40: Máximos detectados sobre el mapa de calor de la imagen que se está analizando.

Estimamos un tamaño de población de 574 partículas de 601 reales quedando un coeficiente de error de 4.49%. Un error bastante bueno de por sí, bastante más si lo comparamos con el obtenido por Watershed, claramente se ve que en este caso si ha

podido diferenciar con más facilidad las distintas gaussianas de cada partícula, a diferencia de los segmentos de Watershed.

La Tercera imagen con la que trataremos es **GSGO_019_C**, el número de aves que aparecen en realidad en esta imagen es de 669.

Figura 41: Imagen **GSGO_019_C**.

Esta imagen presenta un fondo bastante complicado, un simple suavizado no va a poder conseguir que la umbralización de Otsu de buenos resultados. Haremos un procesado previo bastante elaborado, como hicimos en el segundo ejemplo del capítulo 2. Aplicando varias transformaciones morfológicas de *opening* conseguimos que las partículas o aves de la imagen cambien, se oscurezcan debido a que el kernel de la operación morfológica tiene en cuenta en los bordes de las partículas los píxeles oscuros del fondo. En contraposición a esto, el fondo al ser más grande y extenso que las partículas, aunque tenga zonas complejas, no se modifica visiblemente demasiado con estas transformaciones.

Figura 42: Imagen a analizar después de aplicar una transformación morfológica de *opening*.

Si a esta imagen modificada le restamos la original el resultado debería ser un fondo mucho más oscuro y homogéneo y unas partículas mucho más resaltadas.

Figura 43: Imagen resultado de la diferencia entre la imagen transformada y la original.

Después de que ya tenemos una imagen procesada y con un fondo y primer plano mejor distinguidos pasamos a convertir la imagen a binaria después de convertirla a escala de grises.

Figura 44: Umbralización de la imagen que se está analizando después del procesado previo.

A primera vista, la imagen binaria muestra una buena detección del primer plano y fondo. Lo siguiente que haremos será eliminar ruido residual que pueda haber y obtener lo que sabemos seguro que es el fondo y el primer plano como ya hemos visto en otros ejemplos. Una vez lo tengamos, generamos el marcador y le aplicamos el método de Watershed para que lo actualice con las partículas que ha detectado.

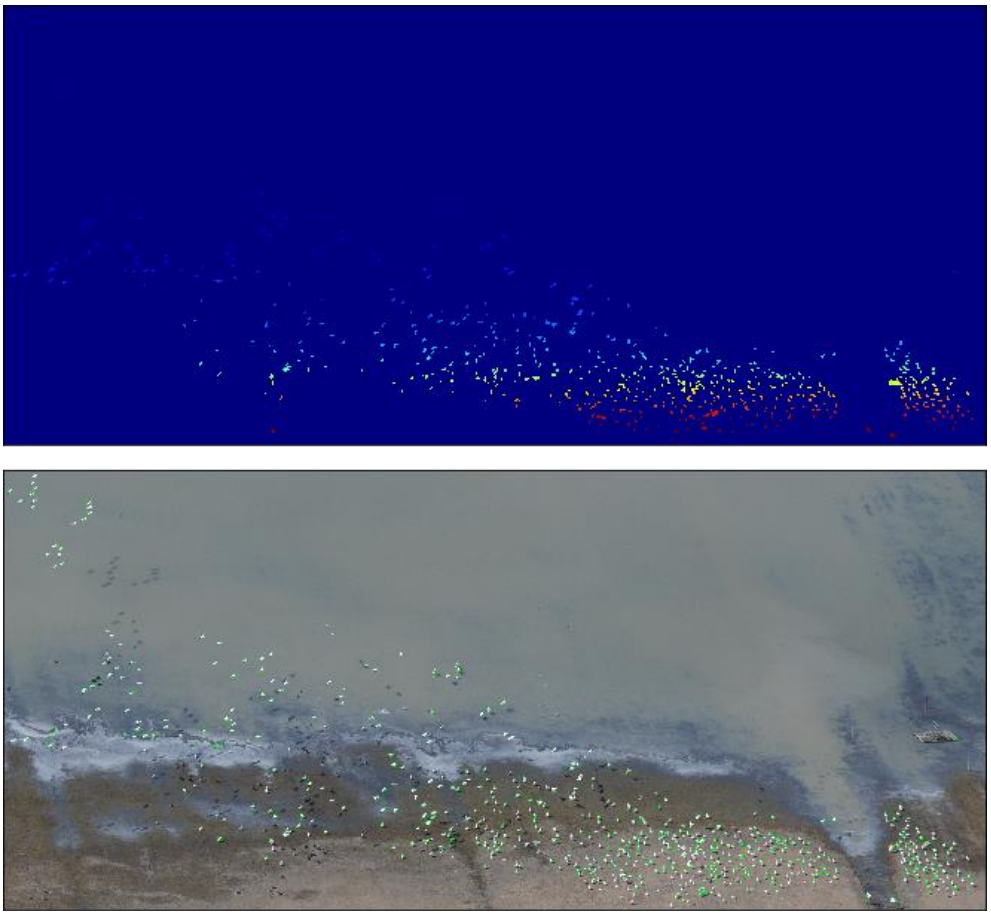

Figura 45: Marcadores de la imagen que se está analizando (arriba) y sus fronteras marcadas en la imagen original (abajo).

Analizando esta imagen con Watershed hemos estimado unas 590 partículas siendo realmente 669, lo que hace un coeficiente de error del 11.80%. No es un mal porcentaje para tratarse de Watershed, la mayoría de errores vienen de falsos negativos por que se detectan varias partículas como una debido mayormente al solapamiento.

A continuación, procesaremos y analizaremos la misma imagen con la red de Class-Agnostic Counting. En la imagen siguiente se muestra su mapa de calor generado.

Figura 46: Mapa de calor de la imagen que se está analizando.

Detectamos los máximos del mapa de calor, se ve que hay gaussianas más intensas que otras, pero no parece que haya ninguna perturbación en el mapa de calor por lo que podemos usar unos valores normales para los dos parámetros de la detección, una distancia mínima de 1 y un umbral de 0.15.

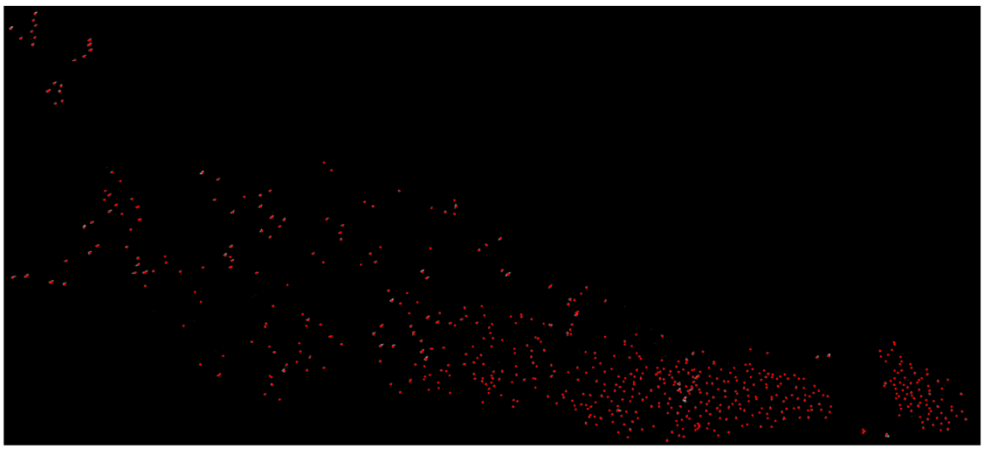

Figura 47: Máximos detectados sobre el mapa de calor de la imagen que se está analizando.

Estimamos con CAC que la población de la imagen tiene un tamaño de 651 siendo 669 el tamaño real, lo que nos da un error del 2.69%. Es un error muy bueno teniendo en cuenta la complejidad del fondo y que tiene aves en vuelo y posadas y CAC ha sabido detectarlas con precisión.

La cuarta y última imagen que usaremos para el análisis se trata de **GSGO_018_C**, cuyo número de aves total es de 8109.

Figura 48: Imagen **GSGO_018_C**.

La principal dificultad de esta imagen es la gran cantidad de aves que tiene, algunas probablemente con solapamientos, aunque por otra parte están distribuidas de una manera bastante uniforme alrededor de toda la imagen.

Como su fondo es homogéneo, pero presenta ligeras irregularidades, que son las olas del mar de fondo, la más adecuado para afrontar la umbralización de Otsu para aplicar el método Watershed es suavizar la imagen como se hizo con la primera imagen de este análisis.

Figura 49: Imagen **GSGO_059_C** en escala de grises después de aplicarla un alisado.

Una vez aplicado el suavizado y la imagen convertida a escala de grises vemos que las imperfecciones del fondo ya no destacan tanto. Lo siguiente es umbralizar la imagen.

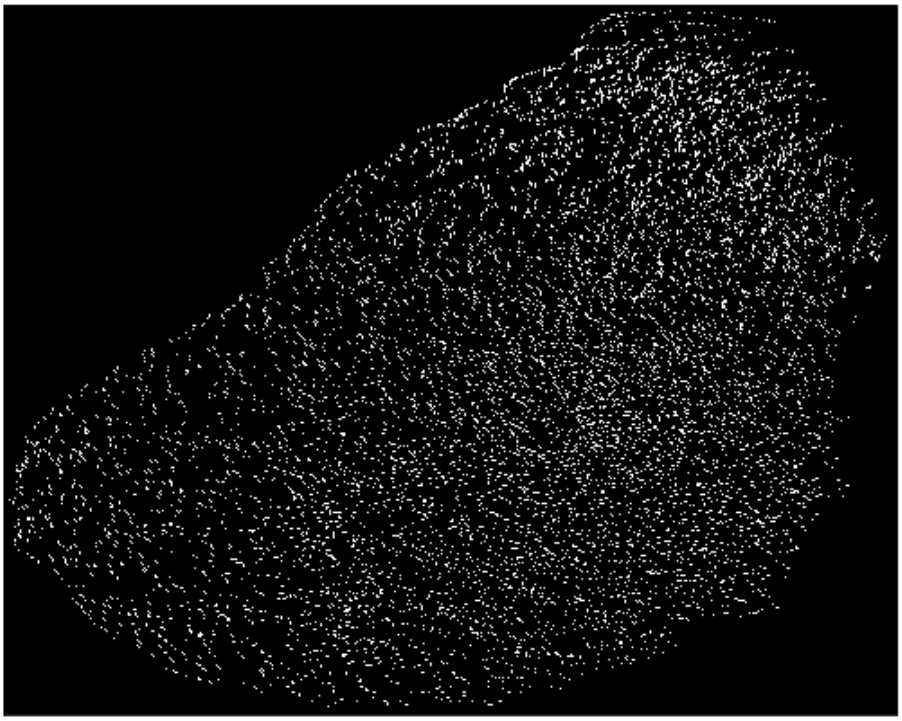

Figura 50: Umbralización de la imagen que se está analizando.

Observamos que en la imagen binaria la distribución del primer plano parece concordar razonablemente bien con la de las aves en la imagen original. Continuamos como en los ejemplos anteriores, obteniendo lo que sabemos seguro que será el fondo y el primer plano y crearemos los marcadores para finalmente aplicar Watershed y obtener la segmentación de las partículas de la imagen.

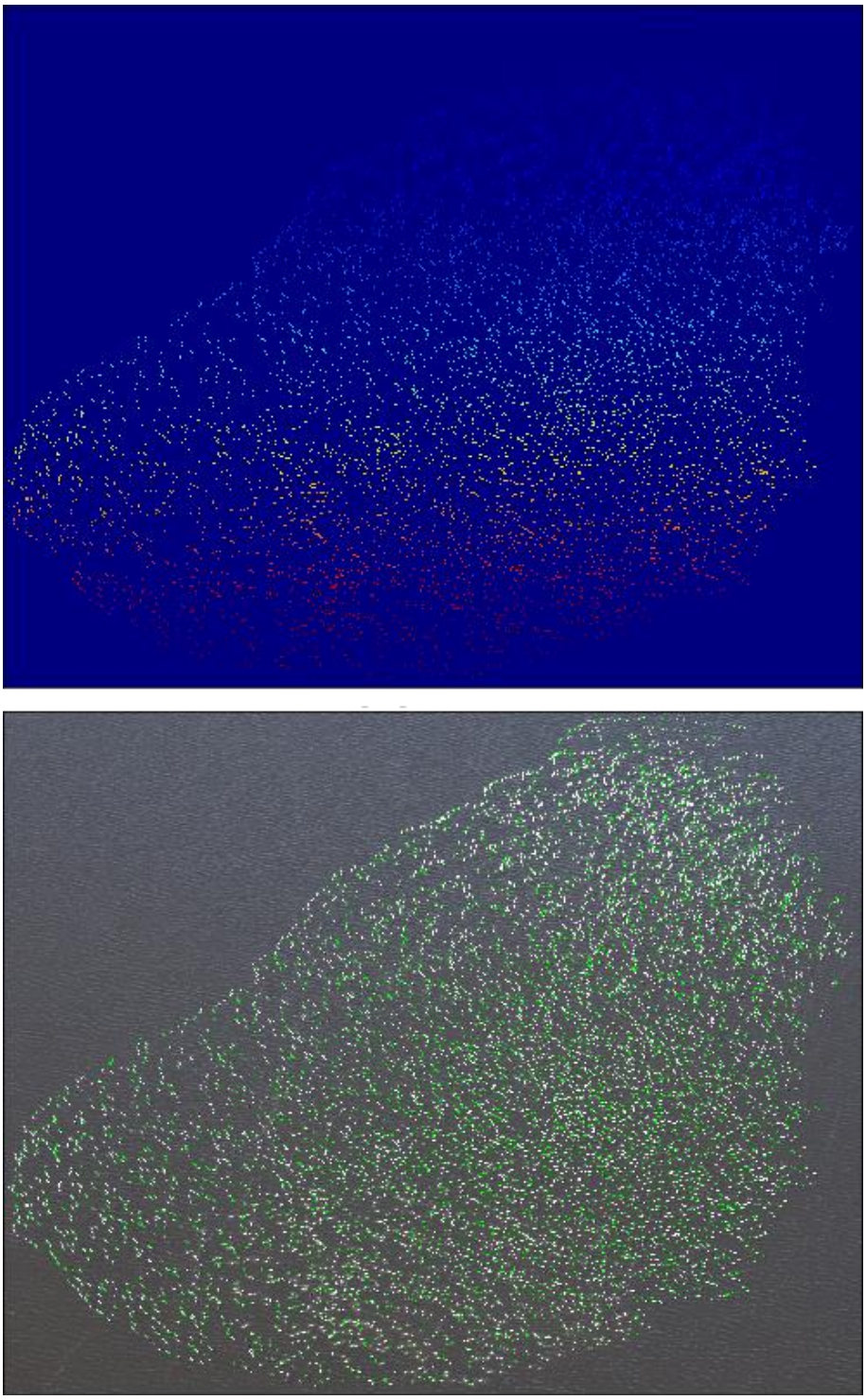

Figura 51: Marcadores de la imagen que se está analizando (arriba) y sus fronteras marcadas en la imagen original (abajo).

Se han detectado 8270 aves de 8109 que son en realidad y por lo tanto tiene un coeficiente de error del 1.98%. Es un coeficiente asombrosamente bajo para Watershed, en la imagen es difícil apreciar si los está captando bien o no, pero parece que los segmentos de los marcadores se adaptan bastante bien a la distribución de partículas de la imagen original.

Hacemos el procesado y análisis de CAC para esta imagen obteniendo el siguiente mapa de calor.

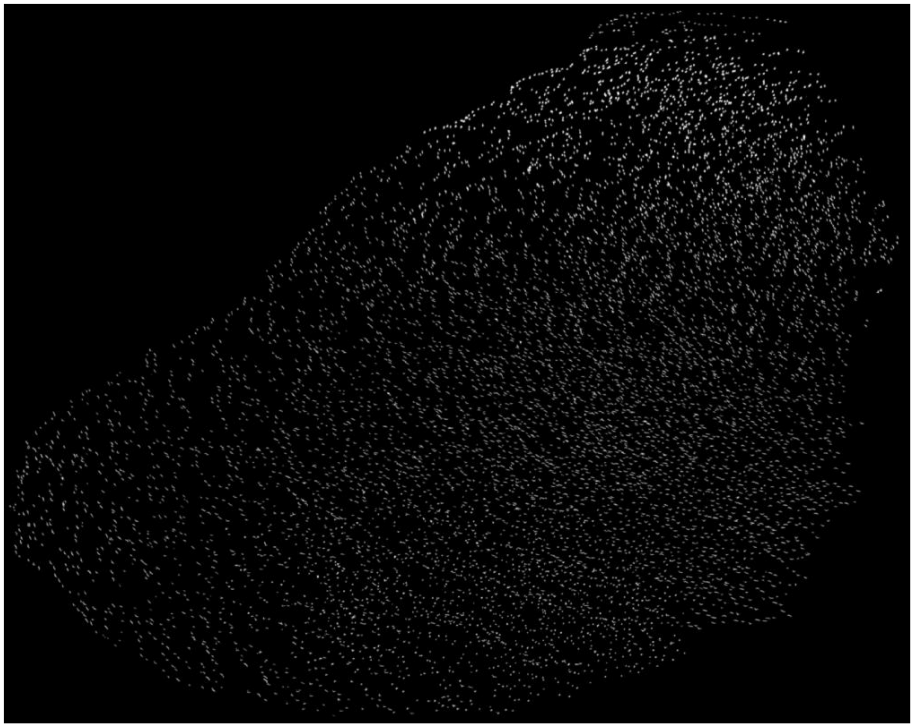

Figura 52: Mapa de calor de la imagen que se está analizando.

Parece un mapa de calor que se corresponde bastante bien con la distribución de partículas de la imagen original. Esta imagen en concreto es de unas dimensiones bastante grandes, por lo que, para no detectar muchos falsos positivos, debido a que detecte más máximos de los que debería en las gaussianas más extensas por el solapamiento, ponemos una distancia mínima de 2 y un umbral normal de 0.1.

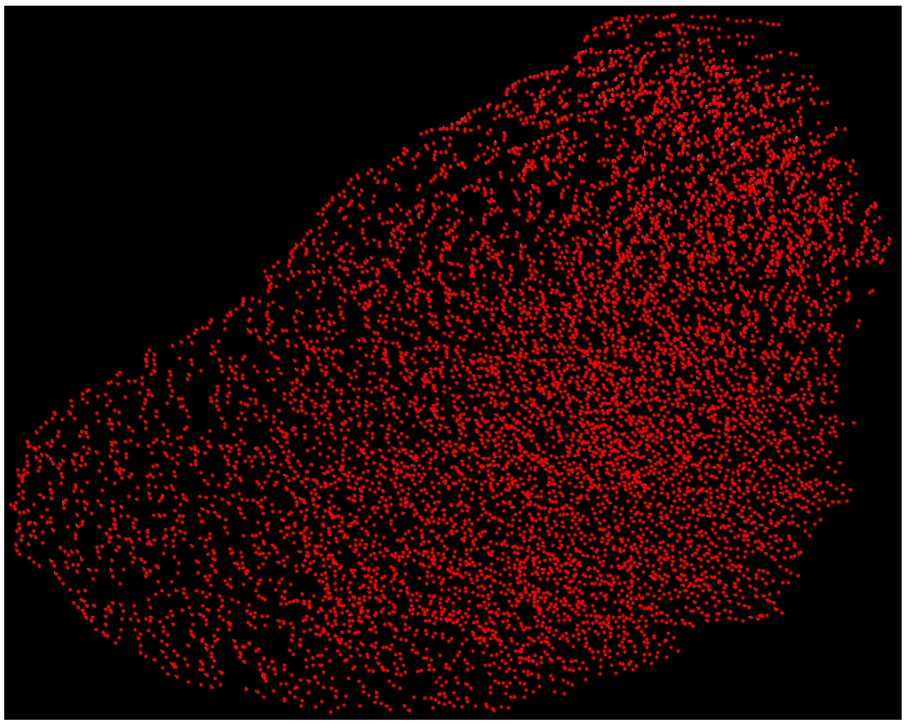

Figura 53: Máximos detectados sobre el mapa de calor de la imagen que se está analizando.

Con CAC obtenemos una estimación del tamaño de la población de 8040 partículas siendo 8109 las que hay realmente, tiene un coeficiente de error del 0.85%. Un coeficiente extremadamente bajo para estas estimaciones, llegando a superar el sorprendentemente bajo coeficiente del método Watershed para esta misma imagen.

Obtenidos todos los resultados de los dos métodos de procesamiento de imágenes digitales, los compararemos con los coeficientes de error de CountEm para esas imágenes. Los notebooks con todo el código necesario para producir estos resultados y el modelo del CAC entrenado con aves blancas se encuentran en los archivos adjuntos del proyecto, no se han podido subir al repositorio por el embargo de las imágenes.

	Coeficiente de Error		
	Watershed	**Class-Agnostic Counting**	**CountEm**
GSGO_059_C	11.58%	**8.53%**	-
GSGO_004_C	32.27%	**4.49%**	7.6%
GSGO_019_C	11.80%	**2.69%**	5.9%
GSGO_018_C	1.98%	**0.85%**	7.4%

Tabla 1: Tabla de comparación de los coeficientes de error medidos por los 3 métodos para las 4 imágenes analizadas.

La imagen **GSGO_059_C** no tiene su coeficiente de error disponible dado que al tener solo 169 aves el tamaño de la población N es comparable al tamaño muestral normalmente necesario para dar una estimación razonable, que era entre 50-200 partículas que necesitaban muestrearse en la rejilla. Todo esto hace que no sea útil usar CountEm en esta imagen.

También es necesario comentar que estos coeficientes de error promedio de CountEm se han obtenido promediando unas 1000 simulaciones de Monte Carlo para cada imagen. Teniendo las coordenadas de las partículas de la imagen, sustituimos las partículas de la imagen por puntos como si fueran las etiquetas usadas para entrenar CAC y con un programa se calcula cuantos de esos puntos caen en la rejilla dada por CountEm [2]. El proceso se repite 1000 veces y se promedia.

Los resultados de los coeficientes de error usando CountEm se pueden encontrar, cuando esté publicado, en el artículo [19].

En el siguiente capítulo expondremos las conclusiones de estos resultados y de su utilidad para apoyar como método automatizado o semiautomatizado a CountEm.

Capítulo 5: Conclusiones

El objetivo de este proyecto era el testeo de un par de métodos de detección de partículas para valorar si alguno puede usarse como método de apoyo para automatizar la selección de parámetros de la aplicación CountEm. Los métodos son un algoritmo simple llamado Watershed y uno basado en machine learning llamado Class-Agnostic Counting.

Sobre el algoritmo Watershed cabe mencionar que obtuvo un resultado bastante decente para la imagen **GSGO_018_C**, aunque peores que los de CAC para esa misma imagen. Por lo demás, tiene unos resultados demasiado "inestables" pasando de 2% de coeficiente de error a 30% para unas imágenes similares. Además, es importante tener en cuenta el contexto de estos resultados, Watershed funciona relativamente bien para aves blancas dado que se suele crear un contraste de color pronunciado entre las partículas (blancas y lisas) y el fondo, pero CountEm debe funcionar para cualquier partícula mientras sea visiblemente contable. Si la partícula tiene detalles más complicados entonces es probable que Watershed empeore sus resultados. Todas estas razones nos sirven para afirmar que un método simple no nos vale y necesitamos uno que tenga en cuenta más cosas que los gradientes y zonas marcadas de una imagen en escala de grises, este método nos servirá como punto de partida para ver cuánto puede mejorar Class-Agnostic Counting sus resultados.

La flexibilidad del contexto de la imagen no es un problema para CAC, porque solo hay que adaptar la red a un dataset de entrenamiento específico de la partícula que se quiera detectar, que dependiendo del computador tomará más o menos tiempo. En los resultados obtenidos por CAC se ve reflejada una clara superioridad frente a los otros métodos. Los coeficientes de error de CountEm, que básicamente son los errores asociados a la varianza muestral, porque es insesgado como se ha explicado, son en comparación más elevados que los coeficientes de error de la estimación del método CAC. A diferencia de CountEm, los métodos de aprendizaje automático no tienen varianza, tienen sesgo.

Por lo que CAC si nos sirve como método para automatizar la selección de parámetros de la rejilla de CountEm al proporcionar una estimación inicial aceptable del tamaño de la población, si la estimación es muy buena como en el caso de **GSGO_018_C** se puede especificar que use esa directamente. Aunque en estos resultados no hemos automatizado completamente la estimación con este método, ya que como ya se ha visto, a la detección realizada se le tiene que ajustar dos parámetros. Aunque son solo dos parámetros numéricos que dedicándole algo más de tiempo se podría encontrar una solución para su automatización, como comentaremos en la sección de trabajo futuro.

En lo personal, gracias a este proyecto, he reforzado el conocimiento que había adquirido en el master en asignaturas como "Estadística para Data Science", en lo referente a CountEm como método de estimación del tamaño de la población mediante muestreo geométrico. También, probar y trabajar con el código de Class-Agnostic Counting me ha llevado a utilizar y reforzar el conocimiento aprendido en la asignatura "Machine Learning I". Además, el proyecto en sí me ha hecho adentrarme en el campo

del procesamiento digital de imágenes que no es algo que se vea en el master, pero que merece la pena haberlo visto y aprendido.

El proyecto ha sido asequible, pero en concreto el método de aprendizaje automático Class-Agnostic Counting me resulto bastante laborioso de entender el funcionamiento de su código. Al ser un proyecto de Github no está prácticamente documentado, al contrario de Watershed que tiene documentación en las librerías que lo implementan y una gran cantidad de hilos en internet con información y guías. Tardé bastante tiempo en hacerlo funcionar, aunque gracias a la máquina en la nube habilitada por el IFCA el entrenamiento de modelos no me tomo un tiempo excesivo. Sin embargo, al ver los buenos resultados finales puedo decir que ha merecido la pena el esfuerzo y tiempo empleados en las pruebas de este proyecto.

Capítulo 6: Futuras Mejoras

A lo largo del desarrollo del trabajo se nos ocurrían ciertas mejoras o maneras de llevar a cabo ciertas partes. A continuación, se listarán esas mejoras.

- Los resultados de los métodos se obtuvieron de 4 imágenes que, excepto la imagen **GSGO_004_C**, se usaron para entrenar el modelo, por lo que habría sido interesante probar que tal lo hace con más imágenes ajenas al modelo, aunque con la que no se utilizó obtuvo buenos resultados.
- El modelo solo se ha entrenado con imágenes de aves blancas, por lo que es razonable pensar que las manchas indeseadas en el mapa de calor por tonalidades blancas en el fondo de la imagen original, como ocurría en la imagen **GSGO_059_C**, son debido a un sobreajuste al color blanco. Por lo que sería interesante ver como se desempeña un modelo entrenado con un dataset que combine bloques de las imágenes de aves blancas y las aves marrones.
- Como los resultados del mapa de calor producidos por CAC son obtenidos con detección de máximos, hubiera sido de ayuda poder visualizar de alguna forma los falsos positivos y falsos negativos en la imagen de las detecciones. La idea general detrás de esto estaría en coger la posición de cada ave en la imagen de etiquetas (pixeles rojos) y ver si hay algún máximo detectado cerca sino es un falso negativo, y lo mismo con los máximos detectados que no se hayan relacionado con ninguna etiqueta, que serán falsos positivos.
- Ha la hora de detectar los máximos en el mapa de calor producido por CAC si queremos obtener los mejores resultados posibles necesitaremos tunear dos parámetros, la distancia mínima (en píxeles) entre los máximos y el umbral mínimo de intensidad de los máximos. El hecho de que sea necesario ajustar parámetros hace que como método de apoyo a CountEm no sea del todo automático. Una manera fácil de solucionar esto es realizar un “GridSearch” simple, que consiste en un método de búsqueda al que le pasas listas con valores a probar de cada parámetro que haya que tunear, en este caso dos y ambos numéricos. Lo que tendría que hacer el método es probar cada pareja de valores de parámetros para que detecte los máximos de una serie de mapas de calor. La pareja que obtenga menor coeficiente de error en su estimación en la mayor cantidad de imágenes será la pareja de parámetros que se usará para estimar un tamaño de la población inicial de manera automática en CountEm.

Agradecimientos

Nos gustaría reconocer el apoyo proporcionado por el grupo de Computación Avanzada y e-Ciencia en el Instituto de Física de Cantabria (IFCA-CSIC-UC) por habilitarnos una máquina en la nube con una GPU NVIDIA V100 TENSOR CORE [18].

Bibliografía

[1] *CountEm Efficient Unbiased Estimation*. (s.f.). Recuperado el 30 de Agosto de 2020, de CountEm: https://countem.unican.es/en

[2] Cruz, M., Gómez, D. y Cruz-Orive, L. M. (2015). Efficient and Unbiased Estimation of Population Size. *PLoS ONE* 10(11): e0141868. https://doi.org/10.1371/journal.pone.0141868.

[3] Cruz, M. (2018). Apuntes de la asignatura Estadística y Optimización.

[4] *Image Segmentation and Mathematical Morphology*. (2010). Recuperado el 30 de Agosto de 2020, de Centre for Mathematical Morphology: http://www.cmm.mines-paristech.fr/~beucher/wtshed.html

[5] Mordvintsev, A. y Rahman, A. (2013). OpenCV-Python Tutorials. *OpenCV*.

[6] *Image Kernels*. (s.f.). Recuperado el 30 de Agosto de 2020, de Setosa.IO: https://setosa.io/ev/image-kernels

[7] *TFM: Análisis de Imagen para el Conteo de Objetos en Imágenes Biológicas*. (2020). Recuperado el 30 de Agosto de 2020, de Github: https://github.com/ERR82-96/TFM_Analisis_De_Imagen_para_el_Conteo_de_Objetos_en_Imagenes_Biologicas

[8] Lu, E., Xie, W. y Zisserman, A. (2018). Class-Agnostic Counting. En *Asian conference on computer vision* (pp. 669-684). Springer, Cham.

[9] *Class-Agnostic Counting*. (2018). Recuperado el 30 de Agosto de 2020, de Github: https://github.com/erikalu/class-agnostic-counting

[10] *Step-by-step instructions to create your own curated ILSVRC15 dataset*. (2016). Recuperado el 30 de Agosto de 2020, de Github: https://github.com/bertinetto/siamese-fc/tree/master/ILSVRC15-curation

[11] *Learning to Count Objects in Images*. (2010). Recuperado el 31 de Agosto de 2020, de University of Oxford: http://www.robots.ox.ac.uk/~vgg/research/counting/index.html

[12] *Topological peak detection in two-dimensional data*. (2018). Recuperado el 1 de Septiembre de 2020, de Stefan Huber: https://www.sthu.org/code/codesnippets/imagepers.html

[13] *Python 3.8.5 documentation*. (s.f.). Recuperado el 1 de Septiembre de 2020, de Python: https://docs.python.org/3

[14] *The Jupyter Notebook*. (2015). Recuperado el 1 de Septiembre de 2020, de Jupyter Notebook: https://jupyter-notebook.readthedocs.io/en/stable

[15] *OpenCV modules*. (s.f.). Recuperado el 1 de Septiembre de 2020, de OpenCV: https://docs.opencv.org/master/index.html

[16] *About Keras*. (s.f.). Recuperado el 1 de Septiembre de 2020, de Keras: https://keras.io/about

[17] *API Documentation: Tensorflow Core v2.3.0*. (s.f.). Recuperado el 1 de Septiembre, de Tensorflow: https://www.tensorflow.org/api_docs

[18] *GPU Nvidia V100 Tensor Core*. (s.f.). Recuperado el 1 Septiembre, de Nvidia: https://www.nvidia.com/es-es/data-center/v100

[19] Cruz et al. (2020). Multi-image flock size estimation with CountEm: A case study with over half a million Common Eiders and Greater Snow Geese. Enviado a *Ecological Applications*.