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

DEEP LOCAL DIRECTIONAL EMBEDDINGS PARA DETECCIÓN DE OBJETOS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FABIÁN EDUARDO SOUTO HERRERA

PROFESOR GUÍA:
JOSÉ MANUEL SAAVEDRA RONDO

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER
AIDAN HOGAN
DOMINGO MERY QUIROZ

SANTIAGO DE CHILE
2020

Resumen

En el contexto de aprendizaje de máquinas, el presente trabajo aborda cómo una nueva arquitectura de redes neuronales profundas, llamada *DLDENet* propuesta por el autor, permite obtener mejores resultados en detección de objetos en imágenes, específicamente en detección de logos, en comparación con otros métodos del estado del arte.

El diseño de esta nueva arquitectura está inspirada en *RetinaNet* [37], una arquitectura propuesta por los investigadores de *Facebook AI research*, que es una *one-stage deep convolutional network*. Sobre esta arquitectura se proponen diversas mejoras, tanto en la arquitectura misma como en la función de pérdida. Estas mejoras incluyen la normalización de los vectores en los mapas de características, ocupar similitud coseno para generar los *logits* del módulo de clasificación, incrementar la pérdida de los *anchors* seleccionados como objetos y forzar que los vectores clasificadores sean perpendiculares entre sí. Esto permite mejorar la precisión, *recall*, *accuracy* y mAP del modelo. A esta arquitectura mejorada se le llamó *DLDENet* (con pronunciación *dieldínet*), a partir de su nombre extendido: *Deep Local Directional Embeddings Network*.

En la etapa de evaluación se utiliza el dataset FlickrLogos-32 [48], que contiene 32 logos distintos de diversas marcas y decenas de ejemplos por cada uno. Además de contar con un conjunto de evaluación y un kit para evaluar los resultados.

Esta nueva arquitectura está acompañada de un *framework* completo para su entrenamiento, testeo, extensión, transferencia de conocimiento y más. Éste lleva por nombre *Torchsight* [50], está desarrollado sobre *Pytorch* [45] y está completamente disponible en GitHub [50].

Con la experimentación se muestra que las mejoras propuestas en DLDENet permiten resolver el problema de la divergencia inicial de RetinaNet sin utilizar un *prior* en el *bias* del módulo de clasificación, lo que permite además mejorar la precisión, el *recall* y pasar de un mAP de 0.65 con RetinaNet a un mAP de 0.77 con DLDENet en la detección de logos usando el conjunto de test de FlickrLogos-32 [48].

A mis padres y amigos que siempre me han apoyado.
Especial mención a mis amigos del spot, gracias a ustedes la universidad se hizo un centro de pensamiento crítico y entretención.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

El problema

1.1. Introducción

La cantidad de contenido multimedia generado a nivel mundial está creciendo exponencialmente, de acuerdo a Bernard Marr de Forbes, en Instagram, por sí solo, se suben más de 95 millones de fotos y videos al día y más de 100 millones de personas ocupan las *"stories"* de esta aplicación [40]. Si a esto le sumamos la información generada por Facebook, Twitter y toda la web los números cuesta imaginárselos. Se genera mucho contenido multimedia, eso está claro.

Lo que no está tan claro es cómo se puede buscar dentro de todo ese contenido usando otras fuentes de datos. Google hace un gran trabajo en términos de búsqueda de acuerdo a lo información escrita que se encuentra junto con las fotografías en la web, además de su motor de búsqueda inversa de imágenes [8]. ¿Pero qué sucede cuando se desea buscar un objeto particular en una serie de fotografías? No se desea buscar la fotografía en sí misma, si no que el objeto que contiene, el cual podría estar contenido, potencialmente también en distinto tamaño, en otras fotografías. ¿Y si la base de datos con las fotografías son privadas? Se puede dar en un contexto personal o laboral, en cuyo caso Google no tiene acceso a esas fotografías y por ende no tendría sentido utilizar Google para hacer una búsqueda de ese tipo.

Este problema, en el contexto de aprendizaje de máquinas, se conoce como *detección de objetos*, cualquier objeto, incluso personas, animales, escenas; cualquier cosa que se pueda fotografiar.

Si se quiere saber cuántas veces una persona ocupó una misma camiseta, en cuántas fotos aparece el mismo cuadro, cuántas fotografías en Twitter al día contienen a la Torre Eiffel o en cuántos comerciales de un canal de televisión aparece el logo de Apple, lo que se quiere hacer es detectar el objeto relevante. Teniendo un método para poder encontrar estos objetos automáticamente podríamos responder todas estas preguntas y muchas más.

Similarmente se pueden detectar *clases* de objetos, no necesariamente una instancia en particular, sino más bien distintos grupos de objetos: aprender a detectar caballos, personas,

Figura 1.1: Ejemplo de detección de objetos. Detección del logo de UPS.

pelotas de tenis, sillas, cualquier clase de objetos.

En el presente trabajo, se propone un nueva arquitectura de red neuronal profunda, llamada DLDENet, la cual permite resolver estos problemas. El principal objetivo de esta nueva red, es ser capaz de generar pequeños vectores que representan a los distintos objetos en una fotografía, de forma que si se desea buscar una instancia o una clase de objeto en particular se pueda hacer simplemente por la similitud de estos vectores de acuerdo a su dirección, es decir, vectores que tienen dirección y sentido similar representan objetos similares.

Un correcto entrenamiento de esta red neuronal, puede permitir tener mayor robustez para detectar instancias de objetos con mucha varianza entre distintas fotografías. Por ejemplo, el logo de una bebida en una lata, en una botella o en un cartel.

DLDENet está basada en RetinaNet [37], una red creada por investigadores de Facebook AI Research para la detección de objetos, pero con varias modificaciones para lograr mejorar la convergencia y obtener mejores resultados, además de proveer un framework para tener una manera rápida y eficaz de entrenar, evaluar y transferir conocimiento entre distintas instancias de este modelo.

Para evaluar a DLDENet en detección de objetos se comparará contra la misma RetinaNet [37] y YOLO v3 [46], otra arquitectura famosa para resolver este tipo de problemas, sobre el dataset FlickrLogos-32 [48] el cual contempla imágenes en escenas cotidianas (en la calle, restaurantes, lugares abierto, cerrados, entre otros) con o sin logos de distintas marcas. Específicamente se evaluará la precisión, el *recall*, el *accuracy* y el *mean average precision* de estos modelos sobre el *test set* de FlickrLogos-32 que contempla 960 imágenes con logos y 3000 imágenes sin logos.

1.1.1. Preguntas de investigación

Las preguntas que se buscan resolver con este trabajo, y que se abordarán en la sección de resultados son:

- ¿Cómo afecta la normalización de los embeddings en los resultados?
- ¿Cómo afectan cada uno de los términos agregados a la función de pérdida?

- ¿Cómo afecta el *bias* en la clasificación?
- ¿Qué configuración permite obtener los mejores resultados?

Estas preguntas cobran más sentido al entender las modificaciones propuestas a RetinaNet que forman la nueva arquitectura DLDENet, la cual será explicada y comentada en detalle en el capítulo 3.

1.1.2. Hipótesis

La hipótesis principal de este proyecto es:

> *Normalizar los vectores en los mapas de características, ocupar similitud coseno para generar los logits del módulo de clasificación, incrementar la pérdida de los anchors seleccionados como objetos y forzar que los vectores clasificadores sean perpendiculares entre sí, permite obtener mejores resultados en detección de objetos.*

Para implementar estas mejoras y evaluarlas se diseña DLDENet. En las siguientes secciones y capítulos se verá cómo se define esta red, qué partes la componen, cuál es su función de pérdida, cómo se entrena, cómo se compara con estas otras arquitecturas, sus resultados y más temas.

1.1.3. Tareas específicas

Para responder las preguntas de investigación y validar la hipótesis planteada y cumplir el objetivo principal se llevan a cabo una serie de tareas, entre las cuales se encuentran:

- Desarrollo de un framework en PyTorch para definir la arquitectura, nuevos módulos e implementar el ciclo de entrenamiento y evaluación del modelo.
- Evaluar RetinaNet [37] y YOLO v3 [46] sobre FlickrLogos-32 [48] para obtener un *baseline* con el cual comparar el rendimiento de DLDENet.
- Entrenar DLDENet completa con todos los nuevos módulos y términos para obtener un nuevo *baseline*.
- Entrenar distintas configuraciones de DLDENet quitando o modificando sus distintos hiperparámetros para ver como afectan cada uno de las modificaciones agregadas a RetinaNet y responder las preguntas de investigación.

1.2. Visión artificial

El trabajo busca explicar cómo la nueva red DLDENet mejora la detección de objetos en comparación a modelos como RetinaNet [37] y YOLO v3 [46]. Pero para clarificar mejor

el problema se hará un breve repaso sobre el campo de la visión artificial, algunos de los problemas que existen y cómo se relacionan.

Este campo de estudio contempla los mecanismos y métodos para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de producir información que pueda ser manejada por un computador [16]. Uno de los objetivos de la visión artificial es poder darle mayor semántica a las imágenes, tal como lo hacen los humanos que no solamente ven una matriz de pixeles de distintos colores, sino que ven complejas escenas, con objetos que se relacionan en un espacio y pueden asignar semántica a esos pixeles que no están ordenados de forma al azar, representan cosas del mundo real.

Dentro de los distintos problemas en el campo de visión artificial, podemos encontrar: *clasificación de imágenes* y *detección de objetos*. A continuación se presenta de qué se trata cada uno, en qué se parecen y en qué se diferencian.

1.2.1. Clasificación

La clasificación de imágenes tiene como objetivo poner una o varias etiquetas a una fotografía, de forma de poder diferenciarlas o agruparlas. En términos más formales, dada una fotografía, queremos aplicarle un método de visión artificial de forma que este nos indique qué etiquetas tiene la fotografía, o en otros términos, de uso más común en el área, a qué *clases* pertenece la fotografía.

Un ejemplo concreto lo podemos ver en la figura 2.1, donde se desea identificar la probabilidad para cada fotografía de que sea un perro, gato, pato o ave. De esta forma podemos agrupar las fotografías de acuerdo a su clase, es decir, qué animal está presente en cada una.

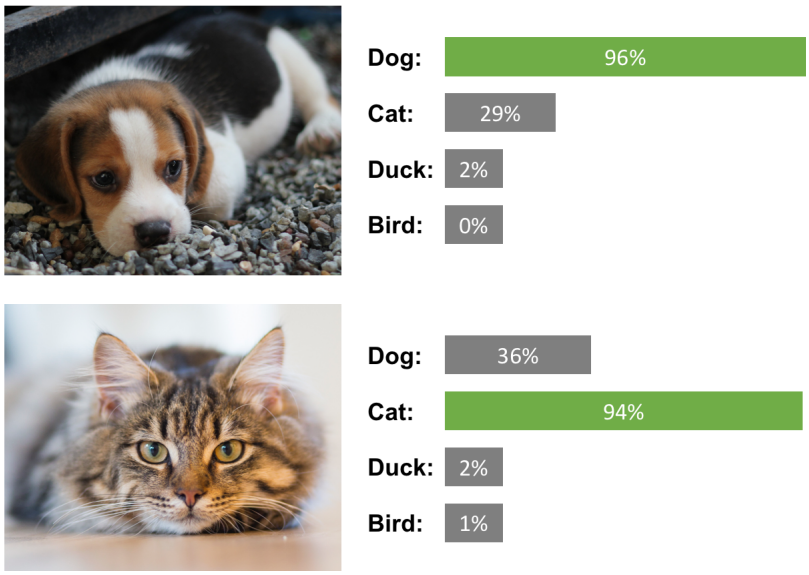

Figura 1.2: Un ejemplo de clasificación. Extraída de [5].

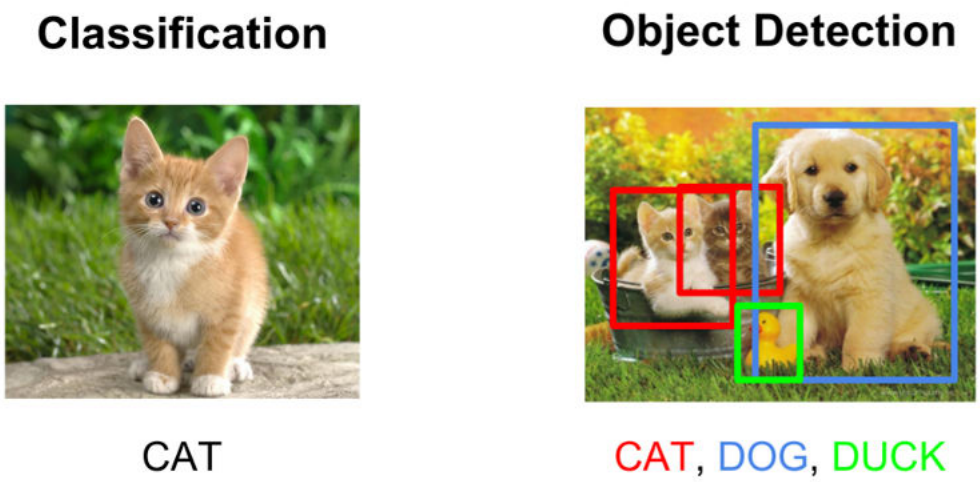

Figura 1.3: La diferencia entre clasificación y detección de objetos. Extraída de [1].

1.2.2. Detección de objetos

La detección de objetos es una tarea más compleja, pues no busca identificar la clase de la fotografía, o más bien, de la fotografía como un todo, sino que busca identificar objetos en esta por lo cual es un problema de clasificación y además localización, el método tiene que ser capaz de indicar *dónde* está el objeto que identificó y cuál es su clase.

En la mayoría de los casos estos se hace a través de una regresión que identifica la esquinas de una *bounding box* o *cuadro delimitador* y le pone una etiqueta a ese cuadro.

En la figura 1.3 se puede notar la diferencia. En el ejemplo de la derecha hay múltiples objetos donde cada uno está delimitado con su *bounding box* y el color indica la clase de cada uno.

1.2.3. Notaciones

1. Un conjunto de fotografías se escribe I, y una fotografía en particular como i.
2. Un conjunto de clases se escribe C y una clase en particular c.
3. Una fotografía i puede tener varias clases c. El proceso de asignar c a i se llama **clasificación**.
4. Una fotografía puede contener varios objetos que se identifican por un bounding box b compuesto por las coordenadas de su esquina superior izquierda (x_1, y_1) y su esquina inferior derecha (x_2, y_2). El objeto identificado puede tener una clase c. El proceso de asignar uno o más pares de (b, c) a una fotografía i se llama **detección de objetos**.

1.3. Objetivo

El objetivo general de este trabajo es mejorar el desempeño de la detección de objetos a través de modificaciones en la arquitectura y la función de pérdida tomando como base la arquitectura RetinaNet.

En específico:

1. Evaluar si la normalización de los vectores en el mapa de características mejora el rendimiento de la red.
2. Evaluar si forzar, a través de la función de pérdida, que los vectores clasificadores sean perpendiculares entre sí mejora el rendimiento de la red.
3. Evaluar si forzar la similitud entre vectores clasificadores y vectores del mapa de características mejora el rendimiento de la red.
4. Evaluar si incrementando la pérdida de los *anchors* seleccionados como *foreground* mejora el rendimiento de la red.
5. Evaluar el desempeño de la propuesta a través de las métricas: precisión, *recall*, *accuracy* y *mean average precision* (mAP).

1.3.1. Evaluación

El entrenamiento de todos los modelos se hace con el conjunto de entrenamiento de FlickrLogos-32 [48] y la evaluación sobre el conjunto de test del mismo dataset. Para computar la precisión, el *recall* y el *accuracy* el dataset provee un kit de evaluación y para computar el *mean average precision* (mAP) se ocupará un proyecto open source desarrollado por Rafael Padilla [44] que implementa el Pascal VOC mAP [25]. La definición formal de estas métricas de evaluación se pueden encontrar en la sección 2.4.

Capítulo 2

Marco Teórico

2.1. Trabajo relacionado

Para resolver estos problemas se han propuesto una serie de métodos de aprendizaje de máquinas. Actualmente, los más exitosos son los basados en redes convolucionales profundas, que dieron su salto a la fama gracias a Alex Krizhevsky. Él y su equipo en 2012 publicaron la arquitectura llamada Alexnet [35] con la cual ganaron la competición de la ILSVRC-2012 en clasificación de imágenes.

En la presente sección se abordarán los conceptos que permiten entender cómo funcionan las redes neuronales profundas y en específico los detectores de objetos.

2.1.1. Redes neuronales

En el contexto de aprendizaje de máquinas, las redes neuronales son sistemas computacionales inspirados en (pero no idénticos a) redes neuronales biológicas. Se toma el concepto de neurona y conexiones sinápticas para diseñar estructuras que imiten la comunicación que existe entre estas. Se crea un sistema donde se tiene *neuronas de entrada* que reciben un *input*, *neuronas de salida* que emiten un valor final y neuronas *escondidas* que conectan las de entrada con las de salida y le dan complejidad a la red. Con este simple pero profundo concepto, dado cualquier problema se desea ajustar una función $f_\theta(x) = y$ donde x es el valor de entrada, y es el valor de salida esperado y θ son los parámetros de la red neuronal.

La idea de intentar imitar el comportamiento de las redes neuronales biológicas en computación partió con Warren McCulloch y Walter Pitts en el año 1943 [41]. Durante todos estos años las redes neuronales han tenido momentos épicos, como en 1975 cuando Paul Werbos describió por primera vez cómo se podía entrenar una red neuronal haciendo *backpropagation* [55] del error; y momentos de muy poco avance durante años, conocidos coloquialmente como *inviernos de inteligencia artificial*.

A principios de los años 90 se hicieron populares las *Support Vector Machines* (SVM) [20] que prácticamente reemplazaron a las redes neuronales y las delegó a un segundo plano. Pero varios científicos siguieron trabajando en estas ideas, entre ellos los ganadores del Premio Turing [54] 2018 Yoshua Bengio, Geoffrey Hinton y Yann LeCun, a los cuales se les otorgó el premio por sus avances conceptuales y de ingeniería que hicieron posible que hoy se hable de redes neuronales profundas (*Deep Neural Networks*) [7].

Pero antes de entender qué son las redes neuronales profundas se necesitan algunos conceptos más básicos que se explican a continuación.

Perceptrón

La red neuronal más básica es la cual tiene una sola neurona, la cual es llamada perceptrón [26]. Esta estructura es bien sencilla y es básicamente un clasificador lineal binario.

Definición 2.1 *Un perceptrón se define como la función*

$$f(x) = \begin{cases} 1 \ si \ w \cdot x + b > 0 \\ 0 \ de \ otra \ manera \end{cases} \tag{2.1}$$

donde $x, w \in \mathbf{R^n}$, $\cdot$ *es el producto punto y* $b \in \mathbf{R}$ *es el bias.*

Definición 2.2 *Una capa fully connected es un conjunto de perceptrones* $f(\cdot)$ *que reciben el mismo input* $x \in \mathbf{R^n}$ *y computan en conjunto un vector de salida* $y \in \mathbf{R^m}$ *tal que:*

$$f(x) = y = \begin{bmatrix} f_1(x) \\ f_2(x) \\ ... \\ f_m(x) \end{bmatrix} \tag{2.2}$$

Notar que n puede ser distinto a m.

Perceptrón Multicapa (MLP)

Con la definición de capa *fully connected* se puede construir un Perceptrón Multicapa (MLP de su nombre en inglés *Multilayer Perceptron*) [28] que se compone de al menos 3 de estas: la que recibe el input x, una escondida (*hidden*) y la que compute el output y. El MLP también aplica una *función de activación* no lineal a la salida de cada capa, lo que permite que pueda separar datos que no son linealmente separables [22], algo que el perceptrón normal no puede hacer.

Usualmente se usan como funciones de activación la tangente hiperbólica, por su salida entre -1 y 1, la sigmoide por su salida entre 0 y 1 y la RELU (*rectified linear unit*) [42] que evita el problema de desvanecimiento del gradiente (*vanishing gradient problem*) que fue formalmente identificado por Sepp Hochreitter [32].

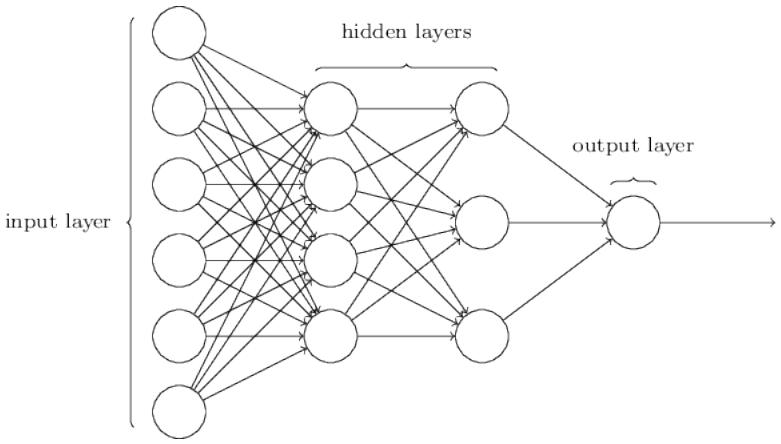

Figura 2.1: Un diagrama que ilustra un MLP. Extraída de [11].

En la figura 2.1 se puede apreciar una ilustración de un MLP. Cada círculo representa una neurona, salvo por los iniciales que son el input de la red. Cada línea representa una conexión o peso sináptico. Neuronas alineadas verticalmente son una capa. En la capa de entrada se ponen los valores del vector de entrada x y cada neurona j computa su valor de salida multiplicando los pesos sinápticos w_j que la conectan con las neuronas de la capa anterior y obtiene su valor de salida sumando esos resultados y aplicando su función de activación.

Formalmente, si L_{i} es la capa i con $\mathrm{i} \in 1, ..., n$ y n el número de capas, la salida de la neurona j de la capa L_{i} es

$$f_{(\mathrm{i},j)}(f_{(\mathrm{i}-1,1)}(\cdot), f_{(\mathrm{i}-1,2)}(\cdot), ...) = act(\sum_{k \in L_{\mathrm{i}-1}} w_{(k,j)} f_{(\mathrm{i}-1,k)}(\cdot) + b_{(k,j)}) \tag{2.3}$$

Donde $f_{(\mathrm{i},j)}(\cdot)$ es la salida de la neurona j de la capa L_{i}, $act(\cdot)$ es la función de activación, $w_{(k,j)}$ es el peso sináptico que conecta la salida de la neurona k de la capa $L_{\mathrm{i}-1}$ con la neurona j y $b_{(k,j)}$ es el *bias* de esa conexión.

Deep learning

El término aprendizaje profundo, o ampliamente conocido por su nombre en inglés *deep learning*, hace referencia al conjunto de métodos de aprendizaje de máquinas (*machine learning*) basados en redes neuronales. En específico, un MLP se podría entender como parte del conjunto más amplio que es *deep learning*. La verdad es que existen muchos tipos distintos de redes neuronales que pueden adecuarse mejor o peor según la tarea o problema a resolver, en este caso se hablará de las redes convolucionales y cómo pueden ser usadas para detectar objetos en fotografías.

2.1.2. Redes convolucionales

Las redes convolucionales, a pesar de que se suele pensar que son redes más complejas que un MLP, en efecto, son más simples, tienen menos conexiones y por lo tanto la red tiene menos pesos sinápticos que aprender. Es una versión más estructurada del MLP, por lo cual el diseño de este tipo de redes lleva mayor complejidad para el ingeniero o científico que la crea pero no para el algoritmo de aprendizaje.

Este tipo de redes están pensadas para *inputs* en formato de *mallas* o matrices, y por lo tanto suelen usarse en problemas de visión artificial y procesamiento de imágenes [28].

Sus principales ideas surgen de la biología también, pues están inspiradas en cómo funciona la corteza visual [27].

Contrario a otros métodos, las redes convolucionales no necesitan un mecanismo previo para la extracción de características ni diseño de éstas, es la misma red la que aprende a extraer características de las fotografías. Esto gracias a que aprende filtros, también llamados *convoluciones*, que se aplican en cadena para obtener un *mapa de características*.

A continuación se definen estos conceptos.

Convolución

La convolución es un concepto utilizado en más áreas que solo en *deep learning* [28]. Pero en este caso se verá cómo se define en visión artificial, qué cosas la componen y cómo se puede calcular.

Una convolución tiene una matriz llamada *kernel* $K(m, n)$ donde (m, n) es el tamaño del kernel en dos dimensiones y se puede aplicar en una sección de la fotografía haciendo calzar el primer valor del kernel (K(0, 0)) en el pixel $I(\mathrm{i}, j)$ y computando:

$$s(\mathrm{i}, j) = \sum_m \sum_n I(m, n) K(\mathrm{i} - m, j - n) \tag{2.4}$$

En la práctica en la mayoría de los programas de *deep learning* se usa una función relacionada a la convolución llamada correlación:

$$s(\mathrm{i}, j) = \sum_m \sum_n I(\mathrm{i} + m, j + n) K(m, n) \tag{2.5}$$

Usualmente se usan kernels cuadrados con tamaño impar, lo que permite tener un solo valor en el centro del kernel y es a ese pixel al cual se le asigna el valor computado. Si se aplica el kernel sobre toda la fotografía se puede obtener una nueva matriz la cual tiene *características* computadas a partir de los pixeles reales. Un ejemplo de esto se puede ver en la figura 2.2.

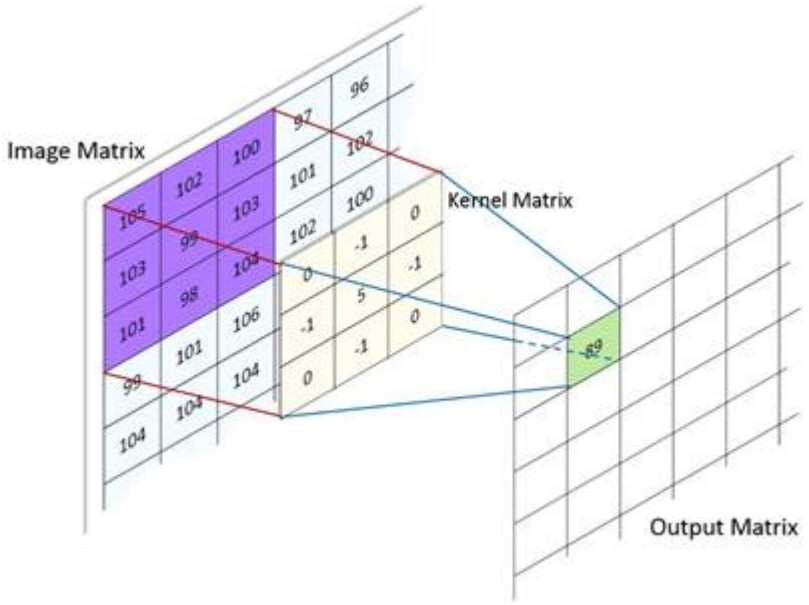

Figura 2.2: Ilustración del resultado de aplicar una convolución. Extraída de [3].

Como se puede apreciar en la figura 2.2, si aplicamos la convolución sobre todos los pixeles de la fotografía la matriz resultante no puede tener valores en los bordes, puesto que no se puede centrar el kernel en uno de los pixeles de los bordes ya que quedarían valores del kernel sin aplicarse, no tendrían un pixel asociado. Para resolver este problema, se le puede asociar un *padding* a la convolución, donde los bordes de la fotografía son acompañados con otro valor para poder aplicar el kernel sobre ellos. Esto se puede apreciar en la figura 2.3.

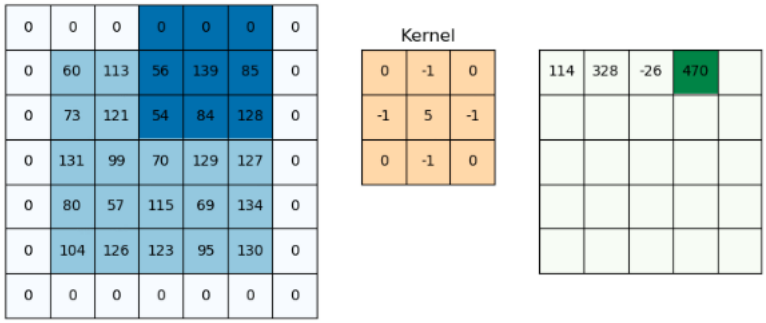

Figura 2.3: Input con padding para aplicar convolución en los bordes de la fotografía. Extraída de [10].

Y finalmente, a la convolución se le puede aplicar un *stride*, que indica cuántos pixeles se tiene que desplazar el kernel para computar su siguiente valor. Específicamente, un stride s quiere decir que si el kernel se está aplicando sobre $I(\mathrm{i}, j)$ la siguiente aplicación debe ser sobre el pixel $I(\mathrm{i}+s, j)$ y si este no existe (se acabaron los pixeles en esa fila de la fotografía) se debe aplicar sobre $I(\mathrm{i}, j+s)$.

Combinando *padding* y *stride* podemos obtener distintos tamaños de matrices como salida. Por ejemplo, poniendo un padding como el de la figura 2.3 y aplicando un stride s sobre una matriz de (m, n) se obtiene una matriz de características de $(\frac{m}{s}, \frac{n}{s})$.

Capas convolucionales y mapas de características

En la práctica no se aplica un solo kernel sobre el mismo input sino que varios kernels en paralelo y así se generan varias matrices de características a la vez. Este conjunto de convoluciones aplicadas sobre la misma matriz se llama **capa convolucional** y usualmente está definida por su padding, stride y tamaño del kernel -como una convolución-, pero además por los *canales de entrada* y de *salida* que emite.

Para entender bien porqué aparecen estos canales de entrada y de salida, primero debemos definir lo que es un *tensor*.

Definición 2.3 *Tensor es lo que generaliza los conceptos de vector y matriz. Está compuesto por* d *dimensiones y cada dimensión puede ser de tamaño arbitrario* $\mathrm{d_i} \in \mathbf{N}$ *y es zero-indexed (el* 0 *es el primer elemento).*

Una imagen puede ser representada con un tensor de 3 dimensiones: una dimensión para la altura, otra para el ancho y otra para los 3 canales RGB.

En la figura 2.3 se puede apreciar un tensor con forma $(5, 5)$ y padding 0. En la figura 2.4 se pueden ver ilustraciones de tensores con distintas dimensiones.

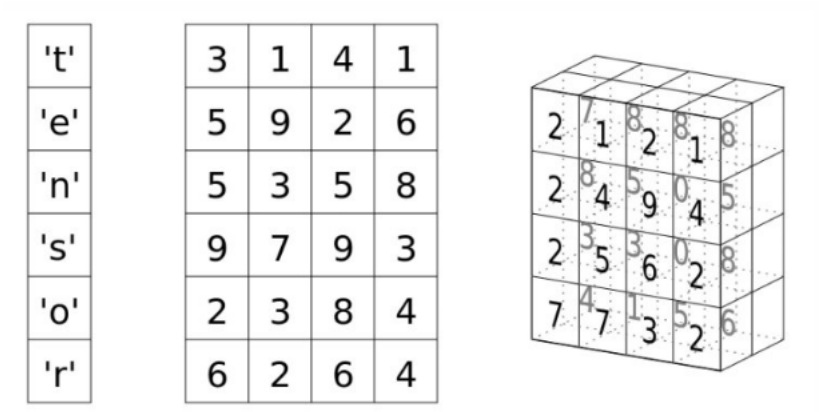

Figura 2.4: Ilustración con un tensor de forma $(6, 1)$, uno de $(6, 4)$ y uno de $(2, 4, 4)$. Extraída de [13].

Entonces una capa convolucional recibe como input un tensor de 3 dimensiones $T(c, h, w)$ donde c es el número de canales, h es la altura y w es el ancho del tensor; y emite una salida $T'(c', h', w')$ donde c' son los canales de salida, y h' y w' dependen del padding y del stride usados en la capa convolucional.

Al ponerse en conjunto varias capas convolucionales se puede crear finalmente una **red convolucional**. Ésta toma un tensor de entrada, como puede ser una fotografía y produce como salida un **mapa de características** o *feature map*.

Hay que notar que si se coloca un padding de forma de no perder los bordes de la fotografía de acuerdo al kernel, los strides de las capas convolucionales se pueden multiplicar y se puede obtener el stride de la red convolucional completa. Se puede ver que si se tienen 2 capas

seguidas con stride s_1 y s_2, y el input de entrada es $I(m, n)$, la salida de la primera capa tendrá forma $(\frac{m}{s_1}, \frac{n}{s_1})$, lo cual será la entrada de la segunda capa y por lo tanto la salida de la segunda capa será $(\frac{\frac{m}{s_1}}{s_2}, \frac{\frac{n}{s_1}}{s_2}) = (\frac{m}{s_1 s_2}, \frac{n}{s_1 s_2})$. Y así se puede extrapolar para cualquier número de capas convolucionales.

El *feature map* puede ser el input de una capa *fully connected* que puede hacer predicciones para clasificar la fotografía o incluso si el mapa de características tiene forma (c, s, s) se puede aplicar una convolución final con stride s y tantos canales de salida como clases existan, digamos k y obtener un tensor con forma $(k, 1, 1)$ con el puntaje para cada clase.

En la figura 2.5 se puede apreciar un diagrama a grandes rasgos de una red convolucional.

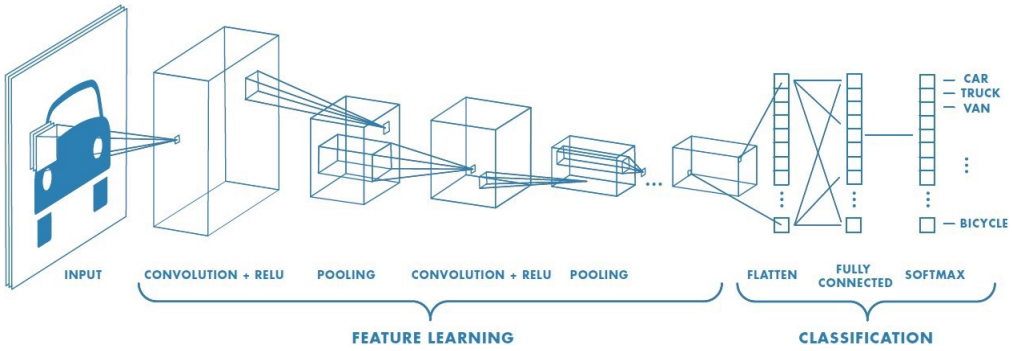

Figura 2.5: Ilustración de una red convolucional para clasificación. Extraída de [2].

La mayoría de las redes convolucionales busca tomar un tensor $T(c, h, w)$ y transformarlo en otro $T'(c', h', w')$ con $c' \gg c$, $h' \ll h$ y $w' \ll w$, como se ve en la figura 2.5 donde se va generando un *feature map* cada vez más pequeño pero mucho más profundo. Esto porque a medida que el *feature map* se va adentrando hacia las capas más profundas, se busca aplicar filtros que cubran mayor parte de la fotografía y obtengan características con mayor semántica.

Las primeras capas de una red convolucional suelen aprenderse filtros simples y bien concretos, como detectar bordes y líneas, luego las siguientes capas toman esas características y tratan de descubrir patrones más complejos como formas y figuras y las últimas capas que reciben características con mayor riqueza generan las características finales que son las que poseen mayor semántica para la toma de decisiones en clasificación o el problema que se desee resolver.

Asociado al *feature map* existe un concepto relevante de mencionar que es el *campo receptivo*, éste es la sección del *feature map* asociado a la fotografía inicial.

Por ejemplo, si una red convolucional tiene un stride de 32, al tomar una fotografía $I(3, 256, 256)$ genera un *feature map* de $T(c, 8, 8)$ con $c \gg 3$. Este *feature map* tiene $8 \times 8 = 64$ campos receptivos y cada uno de estos puede ser asociado a la fotografía inicial de forma que $T(c, \mathrm{i}, j)$ con $\mathrm{i}, j \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ cubre desde $I(3, 32\mathrm{i}, 32j)$ hasta $I(3, 32(\mathrm{i}+1)-1, 32(j+1)-1)$.

Extractor	Año publicación	Top 5 *accuracy* en ImageNet	Cantidad de parámetros
AlexNet	2012	84.70 %	62 millones
VGGNet	2014	92.30 %	138 millones
Inception	2014	93.30 %	6.4 millones
ResNet-152	2015	95.51 %	60.3 millones

Tabla 2.1: Tabla comparativa entre distintos extractores de características famosos.

2.1.3. Extractores de características

Las redes convolucionales son excepcionales para extraer características. Actualmente todos los algoritmos del estado del arte en visión artificial ocupan redes convolucionales [15]. Pero las redes convolucionales pueden tener distintas funciones y objetivos, un tipo de ellos es simplemente extraer características, a este grupo de redes convolucionales se les llama **extractores de características** o *backbones*.

Estos tipos de redes se miden principalmente de acuerdo a qué tan bien clasifican una serie de fotografías, la mayoría de las veces se comparan sobre ImageNet [23], pero no destacan por que las capas que hacen la clasificación (usualmente un pequeño MLP) sean muy elaboradas, sino por sus grandes cantidad de capas convolucionales y complejas arquitecturas.

Algunos extractores de características famosos son Inception [51], VGG [49], AlexNet [35], ResNet [30], ResNext [56] y EfficentNet [52]. En el presente trabajo se comenta en detalle ResNet, que es una de las más populares en la actualidad gracias a su buen comportamiento y multitud de implementaciones en distintos frameworks para su uso.

Una tabla comparativa entre los extractores de características más utilizados se puede apreciar en la tabla 2.1.

ResNet

ResNet es una arquitectura de redes convolucionales del año 2015 que impresionó a la comunidad científica por sus notables ideas y resultados.

La red tiene un componente principal llamado *residual block*, el cual está compuesto por una convolución de 1×1 la que reduce a un cuarto los canales de entrada, luego aplica una convolución de 3×3 y finalmente una convolución de 1×1 para recuperar los canales que habían entrado al bloque. Esta idea les permitió poder ahorrar poder computacional puesto que al reducir los canales para las convolución de 3×3, son menos kernels que aplicar y por lo tanto menos parámetros que aprender.

Pero una de las ideas que más se destacan de ResNet, es como enfrentó el problema del *desvanecimiento del gradiente* [32].

En cualquier red neuronal el error computado se propaga haciendo *backpropagation* [55] desde los últimos pesos hacia los primeros de esta. Para computar el error de los pesos más

lejanos de la salida se usa la regla de la cadena, por lo cual se multiplica muchas veces por valores menores a 1 lo que lleva a errores muy pequeños en los primeros pesos de la red. Este problema no permite tener redes demasiado profundas puesto que el gradiente es demasiado pequeño y cuesta mucho entrenar la red neuronal, incluso puede llevar a que la red no aprenda nada.

Para solucionar esto, los creadores de ResNet agregaron una *conexión residual* al bloque, lo que hace que el resultado final de éste sea la salida de sus capas convolucionales más la misma entrada. Formalmente:

$$H(x) = F(x) + x \tag{2.6}$$

Donde $H(x)$ es la salida del bloque y $F(x)$ es la salida de la última capa convolucional. Esto permite dos cosas notables.

La primera es que esto ayuda a disminuir el problema del desvanecimiento del gradiente puesto que la derivada de la función de costo C sobre x es:

$$\begin{aligned}\frac{\partial C}{\partial x} &= \frac{\partial C}{\partial H}\frac{\partial H}{\partial x} \\ &= \frac{\partial C}{\partial H}(\frac{\partial F}{\partial x} + 1) \\ &= \frac{\partial C}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial C}{\partial H}\end{aligned}$$

Lo cual tiene menos posibilidades de irse a cero y poder propagar bien el gradiente hacia los pesos de más atrás.

El segundo aspecto a destacar es que según los autores [30], si alguno de los bloques de la red resulta *inútil* es más fácil apagarlo puesto que resulta más sencillo para la red llevar los pesos de una capa convolucional a cero y así simplemente dejar pasar el mismo input que recibió.

Un diagrama de estos bloques residuales se puede observar en la figura 2.6.

Finalmente, con esta idea de estos bloques crearon redes de distinta profundidad con 18, 34, 50, 101 e incluso 152 capas convolucionales, lo más profundo que se había creado hasta el año 2015.

2.1.4. Detectores de objetos

Para clasificar imágenes -como se comentó en la sección anterior- se podría agregar un MLP al final de una red convolucional tal como lo hacen con ResNet o Inception. Entonces, una manera sencilla e ingenua para detectar objetos puede ser dividir la imagen de entrada

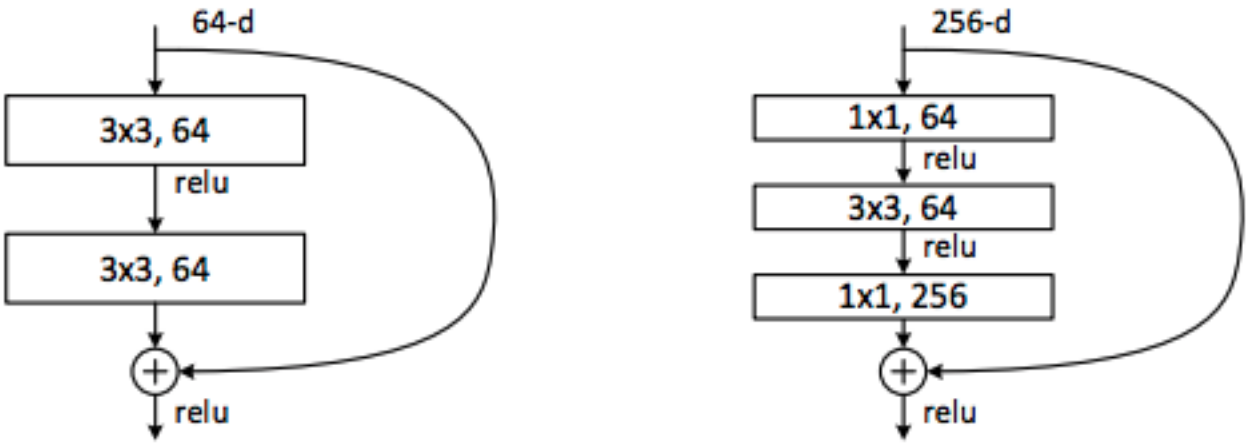

Figura 2.6: Un bloque residual sin disminución de canales y un bloque residual con disminución de canales previo a la capa convolucional de 3×3. Extraída de [30].

en k regiones de igual tamaño y aplicar la red convolucional para clasificación sobre cada una de esas secciones. Así entonces este nuevo mecanismo sería capaz de detectar hasta k objetos en la fotografía.

¿Qué sucede cuando hay más de k objetos en la escena? Se tendrían que tomar más regiones del mismo tamaño, afectando el área que esa región cubre y por lo tanto se corre el riesgo de estar detectando solo *parte de objetos* y no objetos completos. Peor aún, esto no resulta con objetos de distinto tamaño, pues como todas las regiones son iguales se está imponiendo que todos los objetos son del mismo porte. Para solucionar este problema se puede dividir la imagen en regiones de distintos tamaños y tener unas regiones contenidas en otras, lo que aumenta mucho el poder computacional que se necesita para procesar todos las regiones de una sola imagen y aún así sigue teniendo un problema: los objetos pueden tener relaciones de aspecto muy distintas, como un lápiz comparado contra una pelota.

Esto se puede hacer mejor.

Para resolver este problema se crearon nuevas arquitecturas llamadas *clasificadores de objetos* que están especializadas en detectar y asignar un *bounding box* al objeto además de clasificarlo. Estas arquitecturas se pueden dividir en *detectores de dos etapas* y de *una etapa*.

Detectores de objetos de dos etapas

Los detectores de objetos de dos etapas tienen dos pasos que se pueden resumir como:

1. Generar regiones de interés que pueden contener un objeto.
2. Clasificar cada región de interés como una de las clases posibles o ninguna si es que la región no contiene ningún objeto.

La arquitectura más famosa de dos fases es Faster R-CNN [47] la cual cuenta con un *backbone* que genera el *feature map* de la fotografía. Estas características alimentan la *Region Proposal Network* (RPN) que se encarga de generar los *bounding boxes* para posibles objetos y entregar una cantidad limitada de propuestas para clasificar y finalmente cada una de estas regiones alimentan una red que se encarga de la clasificación de objetos y refinamiento del *bounding box*.

Estas arquitecturas al tener dos etapas separadas pierden información y además resultan ser más lentas, no aprovechando todo el poder del paralelismo de la GPU. Esto motiva la creación de detectores de objetos de una sola etapa.

Detectores de objetos de una etapa

Los detectores de objetos de una sola etapa se destacan porque hacen un entrenamiento *end-to-end*, teniendo una sola gran red convolucional que se encarga de extraer las características, generar *bounding boxes* y clasificar cada uno de estas.

Hoy en día hay muchos exponentes de este tipo, entre los más populares se pueden mencionar YOLO v3 [46] y RetinaNet [37]. Esta última arquitectura presentó importantes avances para mejorar la precisión de los detectores y mejorar la detección a distintos y variados tamaños de objetos.

El presente trabajo aborda mejoras hechas a RetinaNet por lo que se explica esta arquitectura en detalle en la siguiente sección.

2.2. RetinaNet

Esta arquitectura está descrita en la publicación *Focal Loss for dense object detection* [37] de Tsung-Yi Lin et al. Pero un aspecto muy importante es la *Feature Pyramid Network* presentada en una publicación anterior del mismo autor [36].

En esta sección se abordarán los problemas que se presentan al querer detectar objetos de diferentes tamaños, el desequilibrio entre la cantidad de secciones con fondo de una fotografía (*background*) y los objetos que están en primer plano (*foreground*) y cómo RetinaNet propone una forma de resolver estos problemas.

2.2.1. Detección a diferentes escalas

La detección de objetos de distintos tamaños es compleja, se desea detectar objetos que cubren prácticamente toda la fotografía y también a esos objetos que se ven pequeños prácticamente en el fondo de la escena, no se debe dejar nada afuera.

Para atacar este problema se han propuesto varias formas que se pueden apreciar en la figura 2.7.

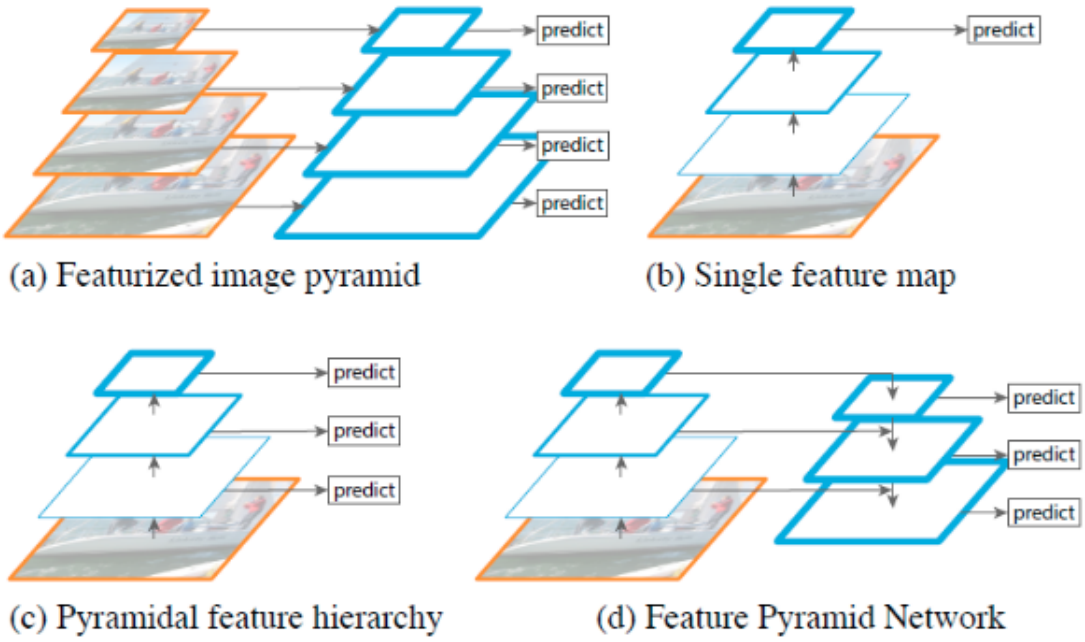

Figura 2.7: Distintas maneras para hacer detección a distintas escalas. Extraída de [36].

Featurized image pyramid

Consiste en disminuir el tamaño de la imagen varias veces y hacer predicciones por cada uno de esos tamaños. Es la opción más costosa puesto que se debe computar el *feature map* por cada nivel de la pirámide.

Single feature map

Es la más sencilla pero la que tiene mayores problemas para detectar objetos a distintas escalas. Este método simplemente computa un solo *feature map* usando algún *backbone* como ResNet e intenta hacer predicciones sobre éste.

Pyramidal feature hierarchy

Esta forma es más compleja que las anteriormente mencionadas y requiere un mayor entendimiento del *backbone* que se está utilizando.

En general, y tal como se comentó en la sección 2.1.2, las redes convolucionales suelen tener un *stride* asociado a la red completa, por lo que en sí misma ya está haciendo una disminución

del tamaño de la fotografía y generando un *feature map* con una gran profundidad pero bien pequeño en cuanto a su ancho y alto.

Por ejemplo, ResNet transforma una fotografía de tan solo 3 canales pero muy grande (como una de 512×512 pixeles) en un tensor con muchas características ($c \geq 512$ dependiendo de la profundidad de la red) pero 32 veces más pequeña (16×16), puesto que ResNet tiene un stride de 32.

Pero la disminución no es drástica en una sola capa con stride 32, el stride de la red se va acumulando dependiendo de las capas que van aplicando un stride de 2, en efecto, y como se puede concluir, ResNet tiene 5 capas convolucionales que aplican un stride de 2, por ende $2^5 = 32$ es el stride final de la arquitectura completa.

La misma red convolucional a medida que procesa la fotografía va generando *feature maps* de distintos tamaños para distintas escalas. En la figura 2.7 (c) se puede ver este ejemplo ilustrado. Y en el caso concreto de ResNet, el mismo *backbone* genera *feature maps* a 5 escalas distintas.

Así entonces se pueden hacer predicciones a distintas escalas con una sola pasada por el *backbone*.

Feature Pyramid Network (FPN)

Pero como se mencionó en la sección 2.1.2 y como lo menciona el autor en [36] también, las primeras capas de la red aprenden características locales bien concretas sin mayor riqueza semántica y las últimas capas generan características con mayor semántica pero pierden el detalle de la fotografía. Por lo que los *feature maps* más profundos tienen mayor robustez para clasificar objetos grandes, pero pésima precisión para detectar objetos pequeños y los primeros *feature maps* no tienen la riqueza semántica para poder clasificar bien estos objetos pequeños.

La *Feature Pyramid Network* intenta solucionar este problema agregando *lateral connections* entre los *feature maps* de distintas escalas.

Se divide en dos secciones principales: *bottom-up* y *top-down*. En la primera etapa simplemente se computan los *feature maps* para cada nivel y en la segunda etapa estos son unidos desde arriba hacia abajo para aportar la riqueza semántica a los primeros *feature maps* sin perder resolución.

Como los *feature maps* son de distinto tamaño, el más pequeño debe ser ampliado al doble de su tamaño para calzar con el de abajo, esto se hace simplemente interpolando los valores de los nuevos campos receptivos por el vecino más cercano. Luego, para poder tener la misma cantidad de canales se aplica una capa convolucional de 1×1 a cada uno de los *feature maps* y finalmente ambos se suman y se aplica una capa convolucional de 3×3 para obtener el *feature map* final de ese nivel y hacer la predicción.

Una ilustración de este proceso se puede apreciar en la figura 2.7 (d).

FPN en RetinaNet

En la práctica RetinaNet genera una *Feature Pyramid Network* de 5 niveles que los llama $P3$, $P4$, $P5$, $P6$ y $P7$.

Los tres primeros niveles son extraídos de la tercera, cuarta y final aplicación de stride en ResNet, de ahí sus números, por lo que tienen strides de 8, 16 y 32 sobre la imagen respectivamente.

Los dos últimos, $P6$ y $P7$ se obtienen de aplicar una capa convolucional de 3×3 y stride 2 a $P5$ y luego aplicar RELU [42] como activación para obtener $P6$ y una capa similar sobre éste para obtener $P7$. Por lo tanto tienen strides de 64 y 128 respectivamente.

2.2.2. Clasificación y regresión

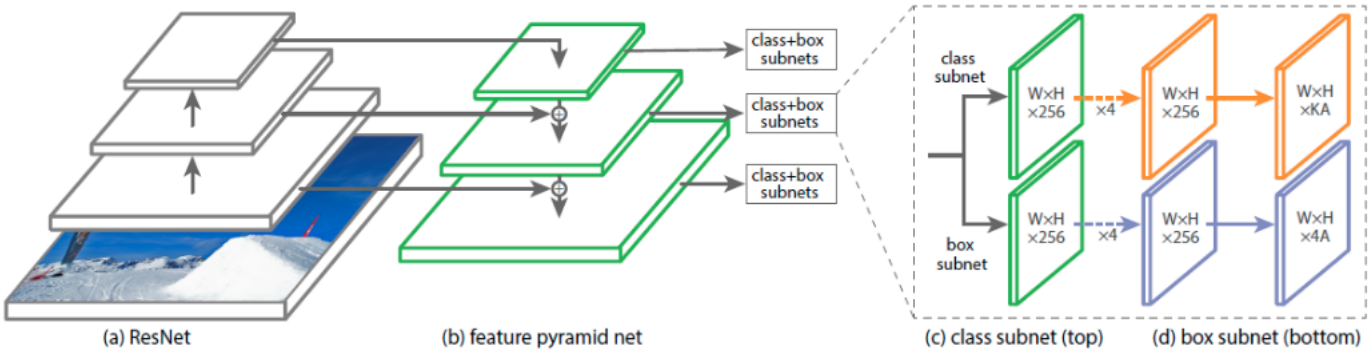

Figura 2.8: Ilustración de la arquitectura RetinaNet. Extraída de [37]

Al tener los distintos *feature maps* para distintas escalas éstos pasan a través de dos submódulos: uno para hacer la clasificación y otro para hacer la regresión de los valores para el *bounding box* de cada objeto.

Pero antes de poder comprender cómo funcionan estos submódulos se tiene que comprender qué son los *anchors* y por qué son claves para la detección.

Anchors

En la sección 2.1.2 se menciona que los *feature maps* tienen campos receptivos que están asociados a ciertas secciones de las fotografías. En detección de objetos a estos campos receptivos se le asocian *anchors*, que son *bounding boxes* base sobre las cuales se aplican la regresión y la clasificación.

En la práctica cada campo receptivo no lleva asociado solo un *anchor*, si no que dado un *anchor* cuadrado se ocupan distintas escalas y relación de aspecto para modificarlo.

En la figura 2.9 se puede observar una ilustración de cómo se pueden visualizar estos *anchors*.

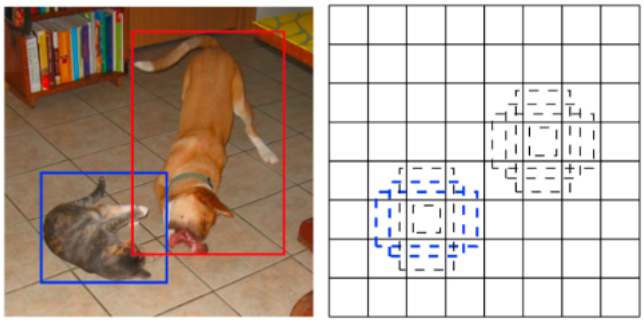

Figura 2.9: Ilustración de cómo se relaciona un *anchor* y distintas relaciones de aspecto con un campo receptivo. Extraída de [12].

En RetinaNet cada campo receptivo lleva centrado un *anchor* base con un tamaño de acuerdo a su *feature map*. Cada uno de estos *anchors* se escala a tres diferentes tamaños (2^0, $2^{\frac{1}{3}}$, $2^{\frac{2}{3}}$) y se ocupan tres diferentes relaciones de aspecto ($1:2$, $1:1$, $2:1$). Por lo tanto hay **9 anchors** (3 tamaños $\times$ 3 relaciones de aspecto) por campo receptivo.

Clasificación

Para poder clasificar cada *anchor* se hace pasar el *feature map* por el módulo de clasificación que consta de 5 capas convolucionales de 3×3 sin stride para mantener en el tamaño. Las 4 primeras tienen de salida 256 canales y la última tiene CA canales donde C es el número de clases y A es el número de *anchors* (9). Así entonces podemos obtener la puntuación de cada clase para cada *anchor*. Para poder obtener la probabilidad de cada clase, esta puntuación por clase se hace pasar por una función sigmoide y así obtener valores $\in [0, 1]$.

Regresión

La regresión funciona de forma similar, tiene 5 capas convolucionales de 3×3 también, donde las dos primeras tienen una salida de 256 canales y la última de $4A$ para obtener los cuatro valores con los cuales hacer la regresión del *anchor* base al *bounding box* preciso del objeto a detectar.

2.2.3. Entrenamiento

Asignación de etiquetas

Para poder entrenar un detector de objetos tenemos que tener las etiquetas de los objetos a detectar. Cada etiqueta está definida por 5 valores (x_1, y_1, x_2, y_2, c) donde (x_1, y_1) es la esquina

superior izquierda del *bounding box*, (x_2, y_2) es la esquina inferior derecha del *bounding box* y c es la etiqueta de la clase.

Se debe recordar que una fotografía es un tensor $I(3, h, w)$ donde h y w son el alto y el ancho de la fotografía y que los tensores son *zero-indexed*. El primer pixel de una fotografía $I(3, 0, 0)$ es el pixel que se encuentra en la esquina superior izquierda.

Si bien todos los campos receptivos tienen asociados 9 *anchors*, no tiene sentido que todos los *anchors* tengan asociada una etiqueta. Como se puede apreciar en la figura 2.10 hay varios *anchors* que no están conteniendo a ningún objeto, por ende no deberían tener ninguna etiqueta asociada.

Figura 2.10: Varios (no todos) *anchors* visualizados sobre una fotografía. Extraída de [14].

Para definir qué *anchors* deben tener etiquetas y cuáles no, se debe introducir el término *Intersection over Union* (IoU).

Definición 2.4 Intersection over Union (IoU) *es el resultado de computar la intersección de dos bounding boxes y dividirla por la unión de ambas.*

En RetinaNet se computa el IoU de todos los *anchors* contra todas las etiquetas y a cada *anchor* se le asigna la etiqueta con mayor IoU. Evidentemente muchos *anchors* van a quedar sin intersección con todas las etiquetas, por ende $IoU = 0$ y otros con un IoU tan bajo que no

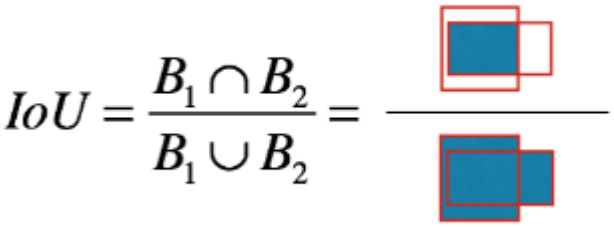

Figura 2.11: Diagrama para explicar cómo se computa el IoU de dos *bounding boxes*. B_{i} es el área (en pixeles) del *bounding box* i. El IoU es una medida que permite medir la similitud entre dos *bounding boxes*. Extraída de [6].

vale la pena asignarlas e éstos. Es por esto mismo que se define un límite de $IoU \geq 0{,}5$ para asignar un *anchor* como que *contiene un objeto* (*foreground*) y un $IoU \leq 0{,}4$ para asignar un *anchor* como *fondo* (*background*). Todos los *anchors* que quedan entre esos dos límites son simplemente ignorados.

Regresión

Al separar *background* de *foreground*, gracias al IoU de los *anchors* con las etiquetas, se pueden computar los valores de las *bounding boxes* finales para afinar la localización de los objetos.

Evidentemente sólo los *anchors* con IoU $\geq 0{,}5$ son los cuales serán utilizados para las regresiones, pues los otros *anchors* no tienen etiquetas asignadas.

La regresión es relativa al tamaño del *anchor* y se puede entender como una forma de *ajustar el tamaño del anchor al bounding box real*. Para esto, en el módulo de regresión se intentan predecir cuatro valores:

$$\begin{array}{ll} t_x = \frac{x - x_a}{w_a} & t_y = \frac{y - y_a}{h_a} \\ t_w = \log(\frac{w}{w_a}) & t_h = \log(\frac{h}{h_a}) \end{array} \tag{2.7}$$

donde x, y es el centro y w, h es el ancho y el alto de la *bounding box*. Lo mismo aplica para x_a, y_a, w_a, h_a pero al *anchor* base con el cual se esté comparando. Se puede notar a partir de estas ecuaciones que los valores a predecir son ajustes relativos al *anchor* que se esté ocupando como base, pues los valores finales de la *bounding box* predicha por el modelo son:

$$\begin{array}{ll} x = t_x w_a + x_a & y = t_y h_a + y_a \\ w = w_a \exp(t_w) & h = h_a \exp(t_h) \end{array} \tag{2.8}$$

Estos valores finales se pueden ajustar fácilmente a los que se han mencionado anterior-

mente (x_1, y_1, x_2, y_2) haciendo:

$$\begin{array}{ll} x_1 = x - \frac{w}{2} & y_1 = y - \frac{h}{2} \\ x_2 = x + \frac{w}{2} & y_2 = y + \frac{h}{2} \end{array} \tag{2.9}$$

Finalmente, el autor aplica una función de pérdida llamada *Smooth L1* sobre la diferencia entre el valor predicho y el target t_*:

$$\text{Smooth L1} = \begin{cases} 0{,}5x^2 & \text{si } |x| < 1 \\ |x| - 0{,}5 & \text{en otro caso} \end{cases} \tag{2.10}$$

Por lo que la función de pérdida para la regresión es:

$$L_{loc} = \sum_{a \in A_{for}} \sum_{\mathrm{i} \in x,y,w,h} \text{Smooth L1}(t^*_{\mathrm{i}} - t_{\mathrm{i}}) \tag{2.11}$$

donde t^* es el valor predicho y t es el esperado, generado a partir de la etiqueta asignada al *anchor*; y A_{for} son todos los *anchors* seleccionados como *foreground*, es decir, con IoU $\geq 0{,}5$ con alguna etiqueta.

Clasificación y Focal Loss

Para la clasificación también se podría pensar en utilizar solo los *anchors* seleccionados como *foreground* con IoU $\geq$ 0,5, pero la verdad es que también se necesita aprender de los *anchors* seleccionados como *background*, pues precisamente de ambos grupos se puede aprender a diferenciar entre objetos y fondo.

El enfoque más utilizado en clasificación es utilizar una función conocida como *Cross Entropy* (CE):

$$CE(p, y) = \begin{cases} -\log(p) & \text{si } y = 1 \\ -\log(1 - p) & \text{en otro caso} \end{cases} \tag{2.12}$$

donde p es la probabilidad predicha por el modelo de que el *anchor* contenga al objeto con clase $c = y$. El autor de RetinaNet [37] propone para simplificar la notación:

$$p_t = \begin{cases} p & \text{si } y = 1 \\ 1 - p & \text{en otro caso} \end{cases} \tag{2.13}$$

Una de los propiedades que identifica el autor es que la *Cross Entropy* asigna un valor de error no menor a objetos que están bien clasificados ($p_t \gg 0{,}5$), donde en un escenario

con muchos ejemplos bien clasificados y unos pocos mal clasificados la función de pérdida se ve orientada a perfeccionar los que ya están bien clasificados y no corregir a los que está clasificando mal.

Esto puede pasar en casos donde se tiene que en el 99 % de los ejemplos que se le muestran al modelo no contiene la clase c y sólo el 1 % restante sí, hay un gran desequilibrio entre $y = 1$ e $y = 0$.

Una técnica conocida para remediar este problema es ocupar un factor α-balanceador (con $\alpha \in [0, 1]$) donde el error obtenido gracias a la *Cross Entropy* se amplifica por α cuando $y = 1$ y $1 - \alpha$ cuando $y = 0$. Así se puede definir α_t similarmente a p_t:

$$\alpha_t = \begin{cases} \alpha & \text{si } y = 1 \\ 1 - \alpha & \text{en otro caso} \end{cases} \tag{2.14}$$

Con lo que se puede redefinir la CE como:

$$CE(p_t) = -\alpha_t \log(p_t) \tag{2.15}$$

Como se mostró en la sección de *anchors* 2.2.2, cada campo receptivo del *feature map* puede tener 9 *anchors*. Sabiendo que la FPN descrita en la sección 2.2.1 genera 7 *feature maps* de stride $8, 16, 32, 64$ y 128 con lo que sobre una imagen relativamente pequeña de 512×512 se pueden obtener feature maps de 64×64, 32×32, 16×16, 8×8 y 4×4. Es decir, tenemos la suma total de $49,104$ *anchors*.

El autor muestra que con estos grandes volúmenes de *anchors* a clasificar la CE se puede ver dominada por los ejemplos fácilmente clasificados perdiendo el foco sobre los que en verdad clasificó mal. Al poner el factor α-balanceador se puede manejar el desequilibrio entre ejemplos positivos/negativos, pero no entre fáciles/difíciles. Es por esto que propone un nuevo parámetro a agregar a la función de pérdida y la rebautiza como *Focal Loss* (FL), por el módulo de *atención* $((1 - p_t)^\gamma)$ que agrega:

$$FL(p_t) = -\alpha_t (1 - p_t)^\gamma \log(p_t) \tag{2.16}$$

El gráfico que muestra la diferencia entre *Focal Loss* y *Cross Entropy*, extraído de la misma publicación, se puede observar en la figura 2.12.

Con esto la función de clasificación queda definida como:

$$L_{cla} = \sum_{a \in A_{for\&back}} \sum_{c \in C} FL(p_{tac}, y_c) \tag{2.17}$$

donde $A_{for\&back}$ son los *anchors* seleccionados como *foreground* y *background*, notar que no son todos pues los que no fueron seleccionados como *background* ni como *foreground* no

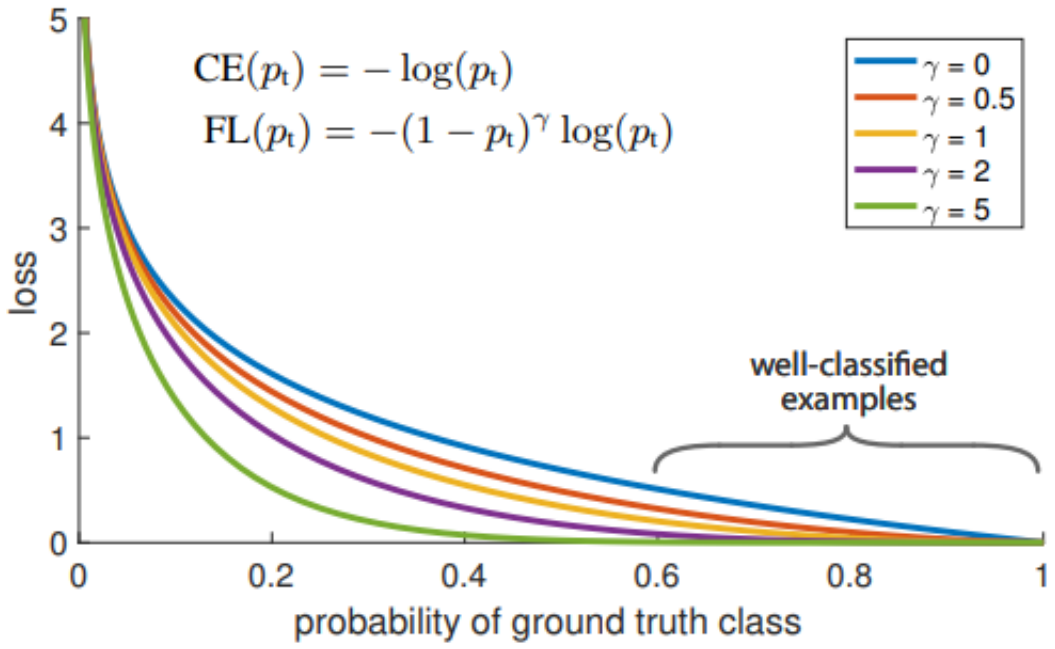

Figura 2.12: Gráfica comparativa entre *Cross Entropy* y *Focal Loss* con diferentes γ. Se puede apreciar como la FL disminuye el error sobre los ejemplos que están bien clasificados. Extraída de [37].

están presentes en este conjunto; p_{tac} es el p_t del *anchor* a para la clase c e y_c es la etiqueta del *bounding box* asignado a ese *anchor* para la clase c, donde si la etiqueta asignada al *anchor* contiene a la clase c entonces $y = 1$ si no entonces $y = 0$.

2.2.4. Función de pérdida final

Finalmente la función de pérdida del modelo queda definida como simplemente la suma entre la función de pérdida de clasificación y la función de pérdida de localización o regresión:

$$L = L_{cla} + L_{loc} \tag{2.18}$$

2.3. Optimizadores

Dada la función de pérdida de un modelo se puede computar un "*puntaje*"de pérdida el cuál se desea minimizar. Para esto se ocupan *optimizadores* que tienen distintas propiedades pero todos comparten el hecho de que son algoritmos iterativos basados en los gradientes de cada uno de los pesos de la red respecto a la pérdida. Es decir, se computa la pérdida para un conjunto de datos, también llamado *batch*, se hace *backpropagation* del error y se obtienen los gradientes para cada uno de los pesos y se avanza en la dirección contraria a este gradiente

para intentar minimizar el error. Se debe recordar que el gradiente apunta en la dirección de máximo ascenso en un punto dado, en este caso el punto es la pérdida computada para el *batch* de datos de entrenamiento y como se desea minimizar la pérdida se debe avanzar en la dirección contraria. El tamaño del avance usando la dirección del gradiente se conoce como la *taza de aprendizaje* o *learning rate* que indica cuánto se debe avanzar.

2.3.1. *Stochastic Gradient Descent* (SGD)

El optimizador más básico es el del descenso del gradiente estocástico, que cumple con lo mencionado al comienzo de este capítulo. En efecto, la formulación formal [21] para actualizar los pesos ω de una red utilizando una taza de aprendizaje α es:

$$\omega := \omega - \alpha \nabla_\omega L(\omega, X, Y) \tag{2.19}$$

Donde L es la función de pérdida que se computa en base a los pesos de la red ω, el *batch* de datos X y sus etiquetas respectivas Y.

SGD además se puede usar con *momentum* para reducir el ruido del descenso tratando de mantener una dirección con giros menos bruscos. Esta nueva versión se puede formalizar de la siguiente forma de acuerdo a [21]:

$$V_t = \beta V_{t-1} + (1 - \beta) \nabla_\omega L(\omega, X, Y) \tag{2.20}$$

$$\omega := \omega - \alpha V_t \tag{2.21}$$

Donde V_t es la dirección del descenso en el instante t y $b \in [0, 1]$ nos permite equilibrar entre las direcciones pasadas y la actual computada gracias al batch.

2.3.2. Adam

Otro optimizador popular a nivel mundial es Adam [34] gracias a su rapidez para lograr buenos resultados, el cual tiene momentum para el estimador del primer momento del gradiente (esperanza del gradiente) y además para el segundo momento del gradiente (varianza sin centrar). Estos quedan definidos como sigue:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\omega L(\omega, X, Y) \tag{2.22}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla^2_\omega L(\omega, X, Y) \tag{2.23}$$

Como estos estimadores se inicializan en cero y además en la práctica se ocupan β cercanos a uno tienden a irse a cero, por esto hacen una *corrección*:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.24}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.25}$$

Y finalmente se actualizan los pesos de la red:

$$\omega := \omega - \frac{\alpha}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t \tag{2.26}$$

Los autores recomiendan $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ y $\varepsilon = 10^{-8}$.

2.3.3. Adabound

Los autores de Adabound [39] muestran que optimizadores veloces como Adam [34] no logran obtener tan buenos resultados como SGD, que es más lento y toma más épocas para obtener buenos resultados. Es por esto que crean Adabound, un optimizar que en palabras de los autores es "tan rápido como Adam y tan bueno como SGD". En la práctica Adabound es simplemente una combinación de Adam y SGD, o mejor dicho una *transformación* de Adam en las primeras etapas del entrenamiento, para ocupar su velocidad, y luego pasar a ser SGD para ocupar su poder de generalización y mejores resultados. Es por esto que ocupa dos *learning rates*: una para Adam y otra para SGD (*final learning rate*). El proceso de transformación es muy suave y se da durante las primeras 1000 iteraciones del entrenamiento.

2.4. Técnicas de evaluación

Para evaluar los modelos de detección de objetos se ocupan distintas métricas, aparte de la función de pérdida que pueda ocupar cada modelo. En esta sección se menciona en detalle algunas de estas métricas.

2.4.1. Verdaderos, falsos, positivos y negativos

El resultado de una predicción de detección de un modelo puede ser clasificada como una de estas cuatro categorías:

- **Verdadero Positivo** (VP): El modelo predijo que en cierto lugar había un objeto de una clase c y efectivamente en ese lugar había un objeto de esa clase. Por eso es un *verdadero* (está en lo correcto) *positivo* (detectó algo).
- **Verdadero negativo** (VN): El modelo dijo que en cierta sección no había ningún objeto y efectivamente no había ningún objeto a detectar. Por eso es un *verdadero* (está en lo correcto) *negativo* (no detectó nada).
- **Falso positivo** (FP): El modelo dijo que en cierta sección había un objeto donde en verdad no había ninguno. Por eso es un *falso* (es incorrecto) *positivo* (detectó algo).
- **Falso negativo** (FN): El modelo no detectó un objeto que sí estaba en la escena. Por eso es un *falso* (es incorrecto) *negativo* (no detectó nada).

2.4.2. Precisión

La precisión es la relación entre verdaderos positivos y el total de predicciones que hizo el modelo, que se puede obtener como la suma de de falsos positivos y verdaderos positivos. Por lo tanto queda definida como:

$$\text{Precisión} = \frac{VP}{VP + FP} \tag{2.27}$$

Y se puede entender como qué tan preciso es el modelo cuando detecta algo, intenta responder la pregunta acerca de qué tanto podemos creerle al modelo cuando detecta un objeto.

2.4.3. *Recall*

El *recall* es la relación entre verdaderos positivos y el total de objetos que debía detectar el modelo, que se puede obtener como la suma de verdaderos positivos y falsos negativos. Por lo tanto queda definido como:

$$\textit{Recall} = \frac{VP}{VP + FN} \tag{2.28}$$

Y se puede entender como qué tan bueno es el modelo detectando todo lo que debiera detectar, cuántos objetos del total que tenía que detectar finalmente detectó.

2.4.4. AP y mAP

Ahora si bien la precisión y el *recall* nos aportan información, por si solos pueden ser engañosos.

Si se imagina un escenario donde dadas 100 fotografías, 50 tienen una clase de objeto y las otras 50 no y el modelo detecta un solo objeto en una de las 50 fotografías que sí tienen el objeto tendría precisión perfecta, puesto que quedaría VP = 1 y FP = 0 con lo que la precisión sería $\frac{1}{1} = 1$. Pero obviamente el *recall* sería muy malo, puesto que los falsos negativos serían FN = 49 y nos queda que *recall* $= \frac{1}{50} = 0{,}02$.

Así también se puede hacer el juego inverso y que el modelo en cada fotografía detecte la clase de objeto y se podría obtener un *recall* = 1 fácilmente, pero la precisión bajaría a 0,5.

El *Average Precision* (AP) intenta solucionar y balancear estas dos métricas. Éste se puede entender como el promedio de la precisión a medida que aumenta el *recall* si es que vamos cambiando el umbral de detección. Dado que el modelo predice con cierta probabilidad de que haya un objeto, si se cambia el umbral sobre el cual se considera que se está haciendo una predicción cambian los valores de precisión y *recall*.

En términos formales se define como:

$$AP = \int_0^1 p(r)\mathrm{d}r \tag{2.29}$$

Pero en la práctica se usa una suma finita sobre ventanas de *recall* y se interpola la precisión tomando la máxima precisión sobre todos los *recall* mayores al actual.

$$AP = \frac{1}{11} \sum_{r \in \{0,0,1,0,2,\ldots,1,0\}} p_{interp}(r) \tag{2.30}$$

donde $p_{interp}(r) = \max_{\hat{r};\hat{r}\geq r} p(\hat{r})$.

Si se computa el AP por cada clase de objeto, se puede tomar el promedio de todos los AP y obtener el *Mean Average Precision* (mAP):

$$\mathrm{mAP} = \frac{1}{|C|} \sum_{c \in C} AP_c \tag{2.31}$$

donde AP_c es el AP calculado para la clase c.

2.4.5. mAP en detección de objetos

En detección de objetos también se puede ocupar el IoU para identificar si una detección está correcta o no. Por ejemplo se puede tener que el modelo detectó que había un objeto de la clase c en cierta parte de la fotografía pero lo localizó mal obteniendo un IoU $\leq 0{,}3$ con el *bounding box* verdadero del objeto y se podría tomar esa detección como mala.

La métrica más famosa es la del dataset COCO [38] que computa el mAP tomando como correcta la predicción a diferentes escalas de IoU. En específico la métrica oficial es el promedio del mAP usando umbrales de IoU $\in \{0{,}5, 0{,}55, 0{,}6, 0{,}65, ..., 0{,}95\}$.

Algunos dataset como Flickr32 [48] no ocupan el IoU como validador y simplemente computan la precisión y el *recall* del modelo sobre el dataset de prueba.

2.5. *Data augmentation*

Para mejorar la robustez de cualquier modelo en aprendizaje de máquinas se ocupa un conjunto de técnicas llamado *Data Augmentation* para aumentar la cantidad de ejemplos que se tiene de cada clase. En la práctica no es más que aplicar distorsiones a las imágenes pero que preserven cierta estructura para que sigan perteneciendo a la misma clase.

Un ejemplo de *data augmentation* se puede observar en la figura 2.13.

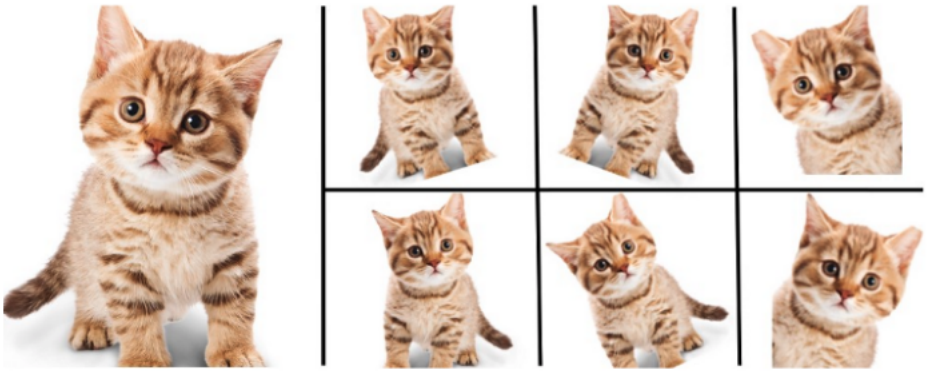

Figura 2.13: Demostración de cómo se pueden generar nuevos ejemplos para un modelo a partir de una misma fotografía. Extraída de [4].

En el presente proyecto se utiliza la librería Albumentations [17] con un pipeline que modifica la fotografía en runtime -justo antes de pasársela al modelo- que cuenta con:

- Ruido Gaussiano con una probabilidad de 0,5 de que se aplique sobre la fotografía.
- *Blur* Gaussiano para hacer la fotografía un poco más borrosa con probabilidad de 0,5 de que se aplique.
- Modificación de contraste y brillo aleatorio con una probabilidad de 0,5 de que se aplique.
- Rotación de la imagen entre 0 y 45 grados con una probabilidad 0,5 de que se aplique.
- Recorte aleatorio de la fotografía sin pérdida de *bounding boxes* con probabilidad de 0,5 de que se aplique.

Como se puede apreciar, todas las modificaciones son al azar, es decir, pueden modificar o no la fotografía. Resultados después de aplicar cada uno de estos filtros se puede observar en la figura 2.14.

Figura 2.14: Imagen de Google de izquierda a derecha: Sin modificaciones, con ruido Gaussiano, con *blur* Gaussiano, con modificación aleatoria de brillo y contraste, rotado y con un corte aleatorio.

Capítulo 3

DLDENet

RetinaNet [37] logró empujar el estado del arte en detección de objetos gracias a las diversas mejoras propuestas tanto de arquitectura como de función de pérdida. Pero que la red obtenga buenos resultados no quiere decir que no sea mejorable. En el presente capítulo se propone un conjunto de modificaciones sobre RetinaNet y su función de pérdida lo que da origen a la nueva arquitectura bautizada como DLDENet.

3.1. Deep Local Directional Embeddings

DLDENet recibe su nombre gracias a que el principal objetivo de la red es generar *embeddings* representativos de los objetos, que faciliten su clasificación y comparación. Un *embedding* es simplemente un vector en $\mathbf{R}^n$ que se ocupa como representación de algo. En este caso, un *embedding* $\in \mathbf{R}^n$ representa una sección de la fotografía.

Estos *embeddings* tienen por nombre *Deep Local Directional Embedding* pues como se verá en la presente sección, salen de la capa más profunda (*deep*), están localizados en un lugar en específico del *feature map* (*local*) y tienen norma unitaria por lo que el aspecto que aporta más información no es la magnitud del vector sino su dirección (*directional*).

3.1.1. Módulo de clasificación

El módulo de clasificación consta de dos partes importantes: el **encoder** y los vectores clasificadores.

Encoder

El módulo *encoder* es el encargado de generar los *embeddings* para el módulo de clasificación.

Como se muestra en la sección de la FPN de RetinaNet 2.2.1, el modelo tiene 5 *feature maps* que deben pasar por un módulo de clasificación el cual cuenta con 5 capas convolucionales que generan la salida esperada de CA canales para cada campo receptivo (C clases y A *anchors* por campo receptivo). En este caso, se hace la primera modificación a la arquitectura donde la salida de CA canales se cambia por una salida con tantos canales como debe tener el *embedding*, en específico se usan 256 canales para esta capa pero se maneja como un hiperparámetro del modelo y por ende modificable.

Formalmente, dado un *feature map* como tensor $FM(c, h, w)$ donde c es la cantidad de canales, h es el alto del feature map y w es el *anchor*, la salida del **encoder** será un tensor $E(\mathrm{d}, h, w)$ donde d es el tamaño del *embedding*, el cual está normalizado en su primera dimensión, es decir, todos los embeddings tienen norma unitaria.

Vectores clasificadores

El módulo de clasificación mantiene un *vector clasificador* w_c para cada clase $c \in C$. Estos vectores están almacenados en una matriz de $c \times \mathrm{d}$ donde c es el número de clases y d el largo de los *embeddings*.

Cada vector w_c mantiene la dirección en la que deben apuntar los embeddings que pertenezcan a la misma clase, por lo que para computar la probabilidad de que un embedding pertenezca a una clase en específico se aplica el producto punto para calcular la similitud del embedding a su vector clasificador, entre más similar más probabilidades hay de que ese embedding sea de un objeto de esa misma clase.

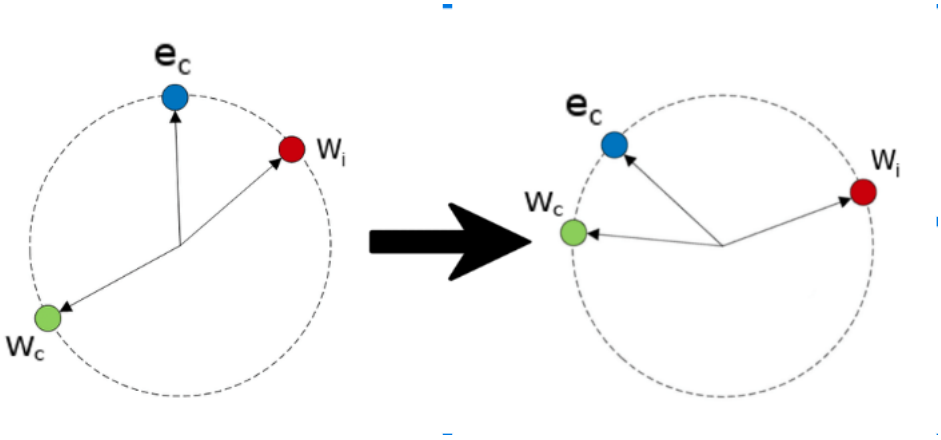

Figura 3.1: Visualización en 2D de un *embedding* e_c y dos vectores clasificadores w_c (correcto) y w_{i} (incorrecto). Lo que se busca a través del aprendizaje es que e_c apunte en una dirección más similar a w_c que w_{i}.

Se debe recordar que el producto punto tiene la igualdad:

$$a \cdot b = |a||b| \cos(\theta) \tag{3.1}$$

Por lo que si la aplicamos a un embedding e con un vector clasificador w_c queda:

$$\begin{aligned} \mathrm{e} \cdot w_c &= |\mathrm{e}||w_c| \cos(\theta_{\mathrm{ec}}) \\ &= |w_c| \cos(\theta_{\mathrm{ec}}) \end{aligned}$$

donde θ_{ec} es el ángulo entre el embedding e y el vector clasificador w_c.

Hay que notar algo importante, si se normaliza el vector clasificador w_c tendríamos que la similitud estaría $\in [-1, 1]$ lo que lleva a que las probabilidades obtenidas a través de aplicar la función sigmoide estarían $\in [0{,}26, 0{,}73]$ por lo que nuestro modelo sería incapaz de detectar un objeto con probabilidad mayor a 0,73.

Además de esto, que puede levantar alertas inmediatamente, tenemos que el factor que escala la similitud de la dirección es la magnitud del vector clasificador, por lo tanto permitiendo que el modelo tenga la libertad de modificar esta magnitud también permitimos que el modelo pueda aprender la varianza dentro de la misma clase, puesto que si la magnitud es ≥ 1 una similitud coseno positiva se va a ver incrementada y permite que sin tener la misma dirección exacta igual tenga altas probabilidades de ser de la misma clase.

Ahora que puede cambiar la magnitud del vector clasificador puede aprender la varianza dentro de la misma clase pero sigue apareciendo un problema, la función sigmoide está centrada en cero, es decir $sigmoide(0) = 0{,}5$ por lo que si dos clases son perpendiculares, es decir $w_{c_1} \perp w_{c_2}$, las probabilidades de los embeddings de una clase aunque sean muy altas, las de su clase perpendicular serán a lo menos 0,5. Para solucionar esto y permitir que el modelo pueda desplazar la curva de la sigmoide se agrega un *bias* también.

Estos casos mencionados se pueden observar en la figura 3.2.

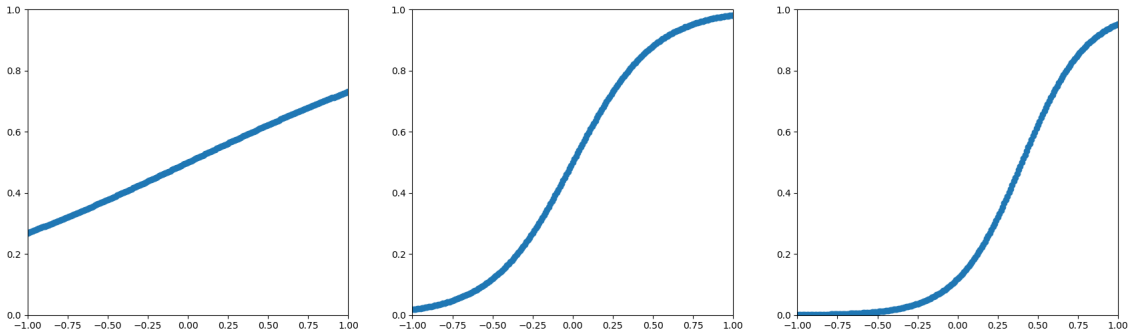

Figura 3.2: Sigmoides para $\cos(\theta)$, $4\cos(\theta)$ y $4\cos(\theta) - 2$.

De esta forma la probabilidad de que un embedding e pertenezca a una clase c identificada por su vector clasificador w_c es:

$$\begin{aligned} p(\mathrm{e}, c) &= \mathrm{e} \cdot w_c + b_c \\ &= |w_c| cos(\theta_{\mathrm{ec}}) + b_c \end{aligned}$$

Para aprovechar el paralelismo de la GPU esto se puede implementar haciendo una multiplicación de matrices:

$$P(\mathrm{e}, c) = E(\mathrm{e}, \mathrm{d})W_c(\mathrm{d}, c) + B_c(\mathrm{e}, c) \tag{3.2}$$

donde e es la cantidad de embeddings, c la cantidad de clases, d el tamaño de los embeddings, P es la matriz que tiene la probabilidad $P(\mathrm{i}, j)$ que el embedding $\mathrm{i} \in \{0, ..., \mathrm{e}\}$ sea de la clase $j \in \{0, ..., c\}$, W_c es la matriz que tiene todos los vectores clasificadores y B_c es una matriz que tiene el *bias* de cada clase para cada embedding. Notar que el *bias* de una clase no cambia entre embeddings.

3.1.2. Diagramas de RetinaNet y DLDENet

En la figura 3.3 se puede apreciar el diagrama de la arquitectura de RetinaNet extraída de [37]. En la figura 3.4 se puede apreciar los cambios hechos a la arquitectura, específicamente en el módulo de clasificación, donde se agrega una capa convolucional que genera el embedding de tamaño E para cada *anchor* de los A que hay en cada sección del *feature map* de altura H y ancho W. Finalmente se aplica la similitud coseno para computar los *logits* de cada clase de las K que hay.

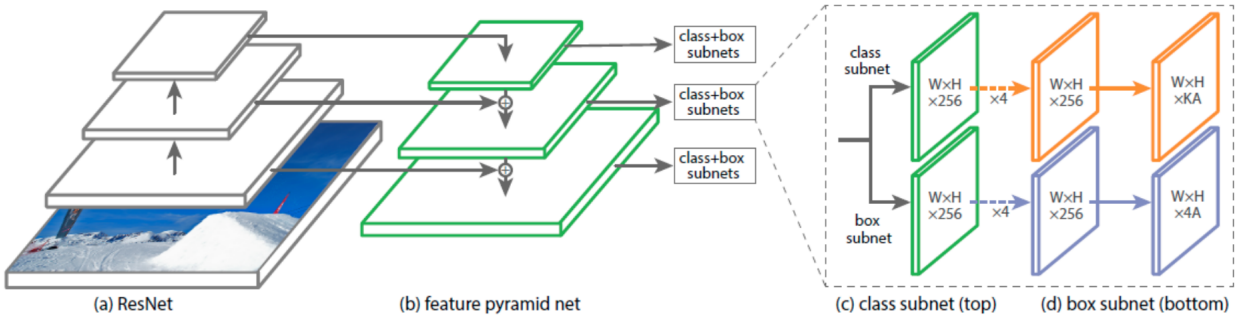

Figura 3.3: Diagrama de la arquitectura de RetinaNet.

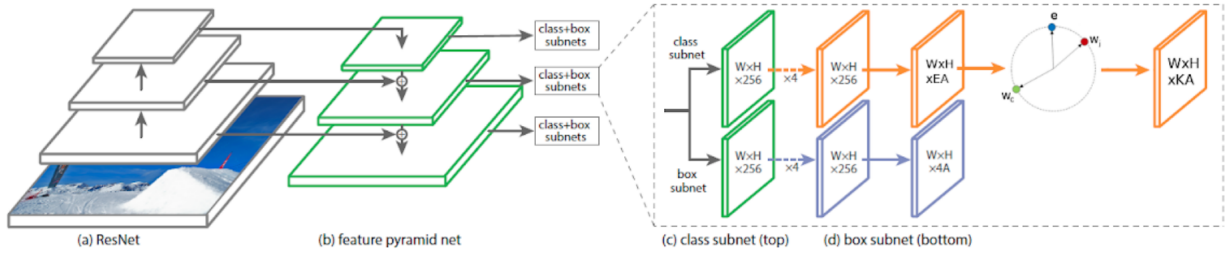

Figura 3.4: Diagrama de la arquitectura de DLDENet y los cambios hechos en el módulo de clasificación.

3.2. Función de pérdida

Con los cambios propuestos sobre la arquitectura de la red en la sección anterior, en esta sección se discutirá acerca de los cambios propuestos a la función de pérdida *Focal Loss* y cómo se compone la nueva función de pérdida propuesta, la $DLDELoss$.

3.2.1. Al comienzo del entrenamiento

Como se comentó en la sección 2.2.3, la *Focal Loss* tiene un módulo de *concentración* o *focus* con su hiperparámetro γ para penalizar más los ejemplos más difíciles y disminuir la pérdida con los ejemplos fácilmente clasificados. Así también ocupan un α para regular entre los ejemplos positivos y negativos. En la práctica, según [37] los mejores resultados los lograron con $\gamma = 2$ y $\alpha = 0{,}25$.

RetinaNet al comienzo del entrenamiento clasifica fácilmente la mayoría de los *anchors* como *background* sin mayor problema gracias a un *prior* en el *bias* de clasificación. El *prior* es una medida mencionada por los autores en [37] que permite disminuir el ruido inicial de la red y permite la convergencia.

Bajo la suposición de que una inicialización aleatoria de los pesos de la red hace que se obtengan salidas cercanas a cero en el módulo que genera las probabilidades, se puede aplicar una inicialización distinta a cero sobre el *bias*, para lograr obtener las probabilidades deseadas al comienzo.

Si la probabilidad para cada clase es computada usando una función sigmoide y el *bias* y la salida de la red es cero, la probabilidad será 0.5. Pues:

$$\begin{aligned} S(\omega x + b) &= \frac{1}{1 + \mathrm{e}^{-(\omega x + b)}} \\ S(0) &= \frac{1}{1 + \mathrm{e}^{0}} = \frac{1}{2} = 0{,}5 \end{aligned} \tag{3.3}$$

Por lo tanto si se sabe que $\omega x = 0$ y se desea que en primera instancia las probabilidades sean cercanas a p se puede inicializar el *bias* de forma que:

$$\begin{aligned} S(0 + b) &= p \\ \frac{1}{1 + \mathrm{e}^{-b}} &= p \\ 1 &= p + p\mathrm{e}^{-b} \\ \frac{1 - p}{p} &= \mathrm{e}^{-b} \\ -\log(\frac{1 - p}{p}) &= b \end{aligned} \tag{3.4}$$

Por lo tanto, si se desea que en el comienzo del entrenamiento RetinaNet obtenga la mayoría bien clasificada como *background*, se puede hacer p =0.1 y eso permite que la red pueda aprender sin tener el ruido inicial que la afecta cuando todas las salidas del módulo de clasificación están cercanas a 0.5.

Pero en la nueva arquitectura DLDENet, se ve en la sección de resultados que inicializando el *bias* en 0 permite obtener mejores resultados pero trae un problema: al comienzo del entrenamiento la función de pérdida está claramente dominada por los mal clasificados *embeddings* del *background*.

Se puede notar empíricamente que si se toman dos embeddings de largo 256 con valores al azar hay una gran probabilidad de que tengan similitud coseno 0, es decir son vectores perpendiculares. Puede parecer contra intuitivo al comienzo pero está dado por el gran tamaño del embedding. Esto hace que al partir con pesos al azar hay una alta probabilidad de que los embeddings generados comparados con los vectores clasificadores resulten en una probabilidad cercana a 0,5 para todas las clases. Si a esto se le suma que la gran mayoría de los *anchors* van a estar asignados a *background* la función de pérdida parte dominada por estos *anchors* mal clasificados.

Este problema hace que el modelo en las primeras fases aprenda a discriminar al *background* pero no toma en cuenta los *anchors* seleccionados como *foreground* puesto que partieron relativamente bien clasificados.

En la práctica solo el 0,1 % de los *anchors* son seleccionados como *foreground*, lo que lleva a que el modelo aprenda a generar embeddings que apuntan estrictamente en la dirección contraria a la de los vectores clasificadores, puesto que la función de pérdida está totalmente gobernada por las malas predicciones de *background* como *foreground* que cubren el 99,9 % del total de *anchors*. El modelo, gracias a este desequilibrio entre *background*/*foreground* tiene todo el incentivo a hacer que todos los embeddings tengan probabilidad cercana a cero para todas las clases.

Cuando el modelo ya logró bajar las probabilidades de esos 99,9 % de *anchors*, está en una posición muy incómoda para aprender a clasificar bien el restante 0,1 % de *anchors* pues debe transformar esos embeddings que apuntan en la dirección completamente contraria y darlos vuelta para que apunten en la dirección de sus clases.

En palabras sencillas, al comienzo del entrenamiento debido al desequilibrio entre *background*/*foreground* que están en relación $\{1000 : 1\}$ el modelo aprende a apuntar los embeddings en la dirección completamente contraria a la de los vectores clasificadores y con esto, cuando en la función de pérdida comienzan a tener más peso los *anchors* del *foreground*, el modelo está en una situación muy incómoda para aprender a distinguir entre clases y no puede obtener resultados decentes.

3.2.2. Modificando el α

Un entendimiento claro de la *Cross Entropy* puede llevar al *data scientist* a modificar el α de la *Focal Loss* para manejar el problema del desequilibrio de clases. Se podría colocar un α para cada clase usando el inverso de la frecuencia de la clase o como una relación entre la cantidad de ejemplos negativos (que no son de esa clase) y positivos (que sí son de esa clase) para balancear la pérdida de cada una de las clases. Es decir, el parámetro se pone de forma que:

$$\alpha = \frac{\text{Ejemplos negativos}}{\text{Ejemplos positivos}} \tag{3.5}$$

donde si $\alpha > 1$ aumentamos el *recall* del modelo y si $\alpha < 1$ aumentamos la precisión del modelo. Ocupando esta técnica sobre los *anchors* del modelo podemos terminar fácilmente con un $\alpha = 1000$ lo que evidentemente va a incrementar el *recall* del modelo, pues ahora le asignamos a la clase correcta del embedding 1000 veces más importancia por lo que el modelo puede detectar bien los objetos del *foreground* pero evidentemente la precisión y clasificación entre clases es pésima porque le estamos diciendo al modelo que es 1000 veces mejor que haga la predicción con alta probabilidad a que no haga ninguna predicción.

Cambiar el α no ayuda a manejar el desequilibrio entre *background*/*foreground*.

3.2.3. La *Focal Loss* en acción

Se puede ver el problema en concreto con datos reales. Se tienen E embeddings para una imagen, donde queremos detectar objetos de C clases distintas. Se puede tener un tensor *target* $T(E, C)$ donde $T(\mathrm{e}, c)$ indica si el embedding e es de la clase c o no.

Formalmente, el tensor *target* se define como:

$$T(\mathrm{e}, c) = \begin{cases} 1 & \text{si el embedding e es de un objeto de la clase } c \\ 0 & \text{si no} \end{cases} \tag{3.6}$$

Si se tiene una *Focal Loss* con $\alpha = 0{,}25$ y $\gamma = 2$, que son los que obtuvieron mejores resultados en [37], $E = 49{,}100$ embeddings y $C = 32$ clases donde solo 50 embeddings, gracias a sus *anchors*, fueron elegidos como *foreground* tenemos un tensor *target* T con $1,517,150$ elementos como 0 y 50 elementos como 1.

Se puede hacer la suposición de que el modelo hace un buen trabajo con los embeddings seleccionados como *background* y con las clases incorrectas de los embeddings seleccionados como *foreground* y les asigna a todas esas probabilidades un valor de 0,1, pero hace un trabajo mediocre con las clases que en verdad eran y les asigna un valor de 0,2. Formalmente:

$$p(\mathrm{e}, c) = \begin{cases} 0{,}1 & \text{si } T(\mathrm{e}, c) = 0 \\ 0{,}2 & \text{si } T(\mathrm{e}, c) = 1 \end{cases} \tag{3.7}$$

donde $p(\mathrm{e}, c)$ es la probabilidad de que el embedding e sea de la clase c según el modelo.

La *Focal Loss* asignará como error:

$$FL(0{,}2) = 0{,}2575101$$

$$FL(1 - 0{,}1) = 0{,}0002634$$

Que refleja correctamente nuestra opinión acerca de cómo se está comportando el modelo, está clasificando bien cuando el embedding no pertenece a la clase pero lo hace mal cuando sí pertenece a ésta. Pero se debe recordar que tenemos $1,517,150$ elementos bien clasificados y solo 50 elementos mal clasificados, por lo que:

$$FL(p|T) = \begin{cases} 1,517,150 \times 0{,}0002634 \approx 400 & \text{para } T(\mathrm{e}, c) = 0 \\ 50 \times 0{,}2575101 \approx 13 & \text{para } T(\mathrm{e}, c) = 1 \end{cases} \tag{3.8}$$

Por lo que el error final no refleja el comportamiento del modelo, pues está dominada por lejos por los embeddings bien clasificados.

3.2.4. Incrementar la función de pérdida de los embeddings de *foreground*

Con este problema en mente, se agrega un nuevo hiperparámetro a la *Focal Loss* para incrementar todos los errores de los embeddings de *foreground*, no tan solo el error de la clase correcta del embedding. Este nuevo hiperparámetro se identifica como w_f.

Como el modelo al comienzo del entrenamiento aprende fácilmente a clasificar bien los embeddings que son *background* se incrementa el error del pequeño conjunto de embeddings que son *foreground* para que también los tome en cuenta, como lo hace el α pero no solo con las clases correctas de los embeddings sino que también incrementa el error de las clases incorrectas del embedding, de esta forma el modelo mejora su ***recall* sin perder precisión** puesto que si el modelo incrementa la probabilidad para una clase incorrecta este error también se verá incrementado por w_f y el *focus* γ hace el resto del trabajo.

Se puede describir la diferencia y el poder de cada uno de los hiperparámetros como:

> Si el parámetro α regula el desequilibrio entre ejemplos positivos/negativos, el parámetro γ regula el desequilibrio entre ejemplos fáciles/difíciles, **el parámetro w_f regula el desequilibrio entre *foreground*/*background***.

Se debe notar que los embeddings que son seleccionados como *background* son para todas las clases ejemplos negativos y cada uno de los seleccionados como *foreground* es un ejemplo positivo para una clase y negativo para el resto.

Definición formal

Esta nueva función de pérdida está bautizada como *Increased Foreground Focal Loss* (IFFL) y se define como:

$$IFFL(p(\mathrm{e},c)) = \begin{cases} w_f FL(p(\mathrm{e},c)) & \text{si } T(\mathrm{e},c)=1 \\ w_f FL(1-p(\mathrm{e},c)) & \text{si } T(\mathrm{e},c)=0 \text{ pero } \exists c_\mathrm{i}; T(\mathrm{e},c_\mathrm{i})=1 \\ FL(1-p(\mathrm{e},c)) & \text{si } \forall c_\mathrm{i}\ T(\mathrm{e},c_\mathrm{i})=0 \end{cases} \tag{3.9}$$

3.2.5. *Cosine Similarity Loss*

Se ha descrito en las secciones anteriores que para que un embedding tenga una alta probabilidad de pertenecer a una clase este debe apuntar en la misma dirección que su vector clasificador w_c. Se puede entonces agregar esta indicación a la función de pérdida, indicándole al modelo que no solo debe clasificar bien, sino que debe hacer que los embeddings apunten en la dirección de su vector de clasificación.

La formulación de esta función de pérdida, bautizada como *Cosine Similarity Loss* (CSL) fue extraía de [29] y también se encuentra una formulación similar en [24]. Ésta se puede adaptar a este problema como:

$$\mathrm{CSL(e,\ c)} = \begin{cases} \frac{\frac{-\mathrm{e}\cdot w_c}{|w_c|}+1}{2} & \text{si } T(\mathrm{e},c)=1 \\ 0 & \text{en otro caso} \end{cases} \tag{3.10}$$

Donde se suma uno y se divide en dos para que la función entregue valores entre 0 y 1.

La función de pérdida final se calcula como el primero de esta función sobre todos los embeddings. Esto ayuda a incrementar la similitud de los embeddings con sus vectores de clasificación. Hay que notar que se normaliza por la magnitud del vector clasificador para penalizar solamente la dirección.

3.2.6. *Dispersion Loss*

Pensando de una forma similar, para poder diferenciar entre las clases se puede aplicar una función que penalice la similitud entre los vectores de clasificación de forma de alejar unos de otros y que no se confundan los embeddings entre las clases.

Concretamente, el caso perfecto es que $w_{\mathrm{i}} \perp w_j$ $\forall \mathrm{i} \neq j$. ¿Por qué no que apunten en la misma dirección pero en sentido contrario (formando un ángulo de 180°)? Pues se puede responder con otra pregunta: ¿Cuántos vectores pueden tener la misma dirección pero sentido contrario? Solamente dos, por lo que si se quiere tener más de dos clases esto no tendría sentido. En cambio, al exigir que los vectores sean perpendiculares se pueden tener $2n$ vectores perpendiculares en $\mathbf{R}^n$.

Con esto en mente se puede formular esta función, bautizada como *Dispersion Loss* (Disp), como:

$$\mathrm{Disp} = \frac{2}{|C|^2 - |C|} \sum_{\mathrm{i} \in C} \sum_{j \in C; \mathrm{i} \neq j} \mathrm{máx}(0, \frac{w_c \cdot w_j}{|w_c||w_j|}) \tag{3.11}$$

donde $\frac{2}{|C|^2-|C|}$ es el factor que permite calcula el promedio puesto que ese es el total de comparaciones entre vectores pues todos contra todos son $|C|^2$, pero no queremos comparar los vectores consigo mismos por lo tanto debemos restar $|C|$ y además dividir por 2 puesto que al hacer $|C|^2$ estamos contando dos veces la comparación entre dos vectores; y el máx$(\cdot, \cdot)$ es para no penalizar si es que los dos vectores están más separados que perpendiculares.

3.2.7. *DLDE Loss*

Finalmente, la función de pérdida de esta nueva red neuronal, queda definida como:

$$\mathrm{DLDE\ Loss} = \lambda_{\mathrm{i}ffl}\mathrm{IFFL} + \lambda_{reg}\mathrm{Reg} + \lambda_{csl}\mathrm{CSL} + \lambda_{\mathrm{dis}p}\mathrm{Disp} \tag{3.12}$$

donde los factores λ son para equilibrar y darle al *data scientist* la posibilidad de penalizar más o menos cada parte de la función de pérdida. Por ejemplo aumentando el λ_{reg} se puede penalizar más la regresión del modelo y poner mayor foco en ajustar bien las *bounding boxes* que genera para mejorar el IoU entre las predicciones y las verdaderas etiquetas de los objetos.

Función de pérdida	RetinaNet	DLDENet
Focal Loss	Sí	Sí
Increased Foreground	No	Sí
Cosine Similarity Loss	No	Sí
Dispersion Loss	No	Sí

Tabla 3.1: Comparación entre qué funciones de pérdida vienen de RetinaNet y cuáles son nuevas en DLDENet.

3.3. Implementación

La implementación del modelo, experimentos e interfaces para interactuar con los *datasets* fueron desarrollados en el lenguaje de programación Python usando, entre otros, el paquete para trabajo con tensores y redes neuronales en GPU, PyTorch [45]. Sobre esto se desarrolló un framework, bautizado como Torchsight [50] para reutilizar código, desarrollo ágil y utilizar buenas prácticas de arquitectura de software para facilitar el desarrollo del proyecto. Torchsight es completamente *open source* y está disponible en GitHub [50].

3.3.1. *Torchsight*

Instalación

La instalación del framework es bastante sencilla, simplemente se debe clonar el repo, crear un ambiente de anaconda e instalar las dependencias:

```
git clone https://github.com/SetaSouto/torchsight.git
cd torchsight
conda env create -f environment.yml
```

Y luego se puede instalar Torchsight en modo desarrollo y así poder hacer modificaciones al código fuente y que se vean directamente reflejadas en el paquete instalado haciendo:

```
python setup.py develop
```

Clases relevantes

El framework cuenta con múltiples módulos que permiten de una u otra forma reutilizar funcionalidades entre distintos experimentos, desde compartir la manera de acceder a la data a través de Datasets y DataLoaders, inicialización de funciones de pérdida y optimizadores, hasta el mismo algoritmo de entrenamiento que corre durante un par de épocas y guarda los pesos de la red neuronal para después utilizarlos en un ambiente productivo o para evaluar el modelo y obtener las métricas de evaluación correspondientes como el mAP por ejemplo.

Trainer

La clase Trainer permite tener objetos que contienen toda la lógica de un entrenamiento. Recibe una configuración en formato JSON que se le puede proveer como un diccionario de python en donde se definen todos los hiperparámetros del entrenamiento como la tasa de aprendizaje, dónde guardar el o los *checkpoints* del entrenamiento con todos los pesos y configuraciones y obviamente todos los hiperparámetros del modelo, optimizador, la función de pérdida, datasets y todo lo necesario para el entrenamiento de la red.

La clase tiene métodos abstractos que se deben implementar para poder tener una clase que permita tener objetos que puedan entrenar modelos. Es el caso de los métodos get_datasets, get_dataloaders, get_criterion, get_optimizer y get_model. Estos métodos deben inicializar sus módulos respectivos y retornarlos de modo de que el Trainer los pueda ocupar en su algoritmo de entrenamiento.

Así es como se implementa el entrenador para RetinaNet, llamado RetinaNetTrainer y de éste hereda el entrenador de DLDENet lógicamente, pues DLDENet es una extensión de RetinaNet. En la figura 3.5 se puede apreciar el diagrama de clases de estos Trainers.

Para reflejar lo práctico de tener este tipo de clases es que si alguien desease tener un entrenador de DLDENet dentro de su aplicación y tiene Torchsight instalado en su ambiente de python puede hacer simplemente:

```
from torchsight.trainers import DLDENetTrainer

DLDENetTrainer(hyperparameters=hyperparameters).train(epochs)
```

Donde hyperparameters es un diccionario con los hiperparámetros del entrenador y epochs es el número de épocas que debe ejecutarse el entrenamiento.

Evaluator

La clase Evaluator permite tener objetos que manejan toda la lógica de evaluación de un experimento. Dado el *checkpoint* generado por un *trainer* puede cargar los pesos del modelo correspondiente y evaluarlo sobre un dataset, computando las métricas necesarias. Básicamente sigue la misma filosofía de un *trainer* en cuanto a métodos abstractos que proveen lo necesario para la evaluación como Dataset, DataLoader y modelo.

El diagrama de clases de los *evaluators* presentes en Torchsight se puede apreciar en la figura 3.6.

Si alguien desease hacer un script en python para evaluar un checkpoint de DLDENet entrenado sobre el dataset FlickrLogos-32 puede escribir:

```
from torchsight.evaluators import DLDENetFlickr32Evaluator

DLDENetFlickr32Evaluator(checkpoint, params).evaluate()
```

Donde *checkpoint* es el string con el *path* al checkpoint creado por el *trainer* y *params* es un diccionario en python con los parámetros para el evaluador.

El resto de las clases del framework son extensiones a los clásicos *Dataset*, *DataLoader*, *Module* y otros de PyTorch.

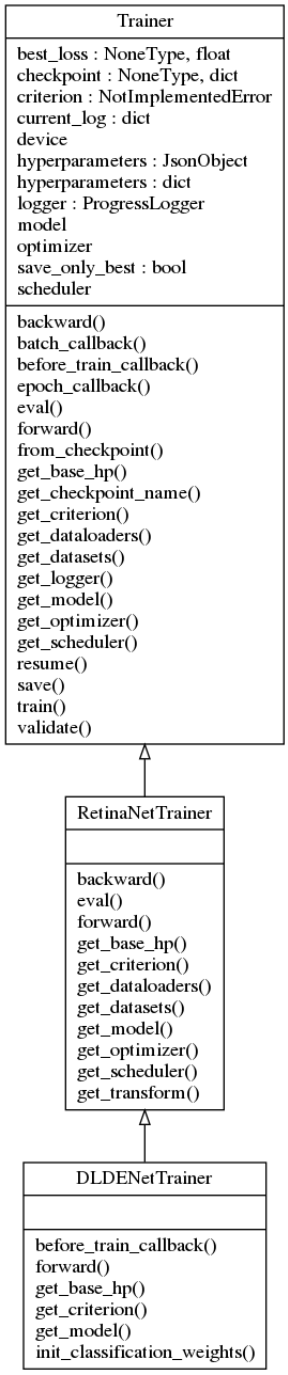

Figura 3.5: Diagrama de clases de los *trainers* presentes en Torchsight.

Command Line Interface (CLI)

Para facilitar la interacción con el proyecto se desarrolló una *Command Line Interface* (CLI) única de forma que contiene todos los comandos para iniciar un entrenamiento, evaluar, visualizar resultados, gráficos, etc. Basta con simplemente poner el comando *torchsight* en la consola y mostrará todos los comandos y grupos de comandos que se pueden ejecutar con su correcta documentación.

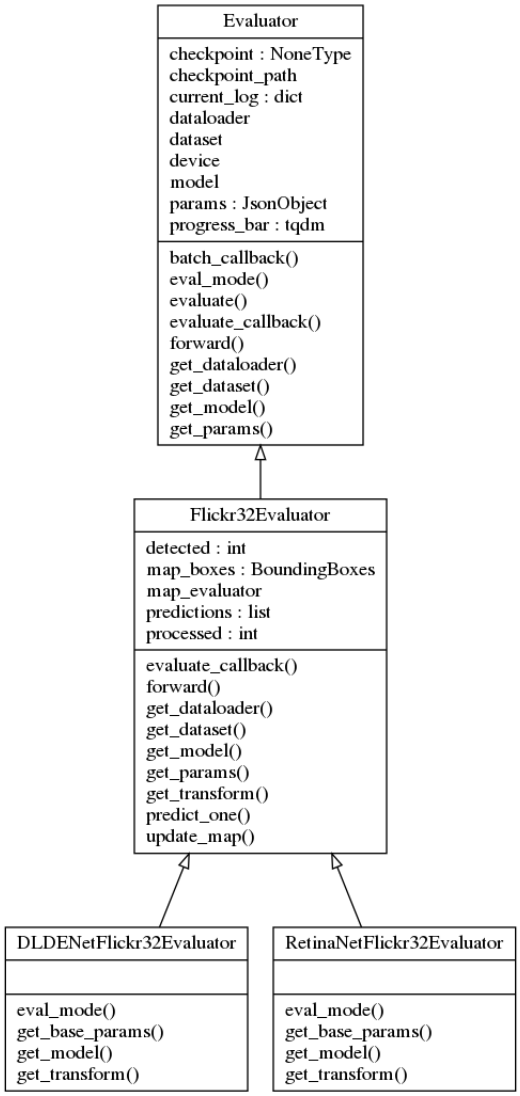

Figura 3.6: Diagrama de clases de los *evaluators* presentes en Torchsight.

Esto es muy útil, puesto que permite interactuar con el proyecto sin siquiera programar una línea. Por ejemplo para ejecutar un entrenamiento de DLDENet sobre el dataset FlickrLogos-32 basta con tener el archivo JSON con las configuraciones y (dado que Torchsight está instalado en el ambiente de python) ejecutar:

```
torchsight train dldenet --config ./hyperparams.json
```

Y luego, con el *checkpoint* generado al finalizar el entrenamiento se pueden evaluar los resultados haciendo:

```
torchsight evaluate dldenet -c ./checkpoint.pth.tar -d flickr32
    -dr /datasets/flickr32
```

donde */datasets/flickr32* es la ruta donde están todos los archivos del dataset.

Capítulo 4

Experimentación

En el presente capítulo se abordarán los diferentes experimentos hechos sobre el dataset FlickrLogos-32, los resultados obtenidos y el análisis respectivo para medir el impacto de cada una de las modificaciones propuestas en DLDENet.

4.1. FlickrLogos-32

El dataset FlickrLogos-32 [48] consiste en imágenes del mundo real extraídas de Flickr [9] que contienen en cualquier sección de la fotografía y de cualquier tamaño, logos de 32 compañías diferentes. Está diseñado para detección de objetos y recuperación de instancias y cuenta con 40 fotografías por logo además de 3000 imágenes sin logos para el entrenamiento. Para la validación de los resultados se tiene otro conjunto, disjunto al dataset de entrenamiento, con 30 imágenes por logo y otras 3000 imágenes sin logos.

Algunos ejemplos de fotografías de este dataset se pueden apreciar en las figuras 4.3 y 4.2.

4.2. Consideraciones generales

Los experimentos comparten configuraciones generales (salvo se indique lo contrario) entre las cuales podemos mencionar:

- **Tamaño imágenes**: En todos los experimentos, salvo se indique lo contrario, se ocupan imágenes de 512×512.
- **Data augmentation**: Se ocupa el pipeline descrito en 2.5.
- **Learning rate scheduler**: Que se encarga de multiplicar la *learning rate* por algún factor luego de que después de E épocas no se mejore la pérdida sobre el conjunto de validación. A la cantidad de épocas E se le llama paciencia. Como las diferencias entre épocas puedes ser ínfimas, se ocupa un *threshold* que mide una mejora porcentual, si

Figura 4.1: Imagen con las anotaciones para destacar el logo de DHL.

Figura 4.2: Imagen con la anotación para destacar el logo de Corona.

no mejora el modelo en cierto porcentaje respecto a su mejor resultado no se considera una mejora.

- Solamente se utilizan las 1280 imágenes con logos para el entrenamiento, dejando fuera las 3000 imágenes sin logos del *training set* para entrenar solo con imágenes que contienen objetos relevantes. Esto con el objetivo de no desbalancear aún más la diferencia entre *foreground* y *background*, puesto que estaríamos agregando 3000 fotografías más que solamente tendrían *anchors* asignados como *background* lo que puede disminuir el *recall* del modelo.

Además, en los resultados se ocupan algunas siglas que se explican a continuación:

- I = Todas las imágenes del conjunto de evaluación, con logos y sin logos.

- I_l = Todas las imágenes con logos.
- I_b = Todas las imágenes sin logos, solo *background*.
- I_{lc} = Imágenes con logos y detecciones correctas.
- I_{li} = Imágenes con logos y sin detecciones correctas.
- I_{bc} = Imágenes sin logos y sin detecciones, por ende correctas.
- I_{bi} = Imágenes sin logos y con detecciones, por ende incorrectas.
- Precisión = $\frac{I_{lc}}{I_{lc}+I_{li}}$
- *Recall* = $\frac{I_{lc}}{I_l}$
- *Accuracy* = $\frac{I_{lc}+I_{bc}}{I}$

Estas métricas son computadas por el mismo kit de evaluación del dataset [48] y el AP es computado a partir del trabajo de Rafael Padilla [44] para cada clase y finalmente el mAP es el promedio de los AP.

El kit de evaluación contiene ciertas debilidades, como por ejemplo que dada una imagen con múltiples logos si un modelo detecta uno solo y otro modelo los detecta todos, ambos modelos tendrán el mismo I_{lc}. Es por esta razón que se agrega el mAP, que es una métrica que permite reflejar de mejor manera estas situaciones.

4.3. Baselines

El presente trabajo aborda mejoras sobre RetinaNet, por lo cual en primera instancia se debe obtener el *baseline*, es decir, el punto de comparación básico para saber si la nueva arquitectura permite mejoras y por qué. Además de RetinaNet, se probará también con YOLO v3 [46], otra arquitectura con excelentes resultados y muy famosa a nivel mundial.

4.3.1. RetinaNet

Para generar el *baseline* de RetinaNet se ocupan los mismos parámetros definidos por su autor como los más exitosos en su experimentación. La configuración de la red se puede encontrar en la tabla 4.1. Se ocupó tanto SGD como Adabound para ver cómo variaban los resultados.

Resultados con SGD

Utilizando SGD como optimizador se entrenó durante 299 épocas logrando sus mejores resultados en la época 262 y se redujo la *learning rate* solamente en la época 251 y posteriormente en la 289 pero sin lograr mejores resultados.

Los resultados se pueden apreciar en la tabla 4.2.

Backbone	ResNet-50
Anchors	
Tamaños	20, 40, 80, 160, 320
Escalas	1, $2^{1/3}$, $2^{2/3}$
Relaciones de aspecto	1/2, 1, 2
Pre entrenamiento	Sobre ImageNet [23]
Función de pérdida	Focal Loss
α	0.25
γ	2.0
Peso pérdida regresión	1
Peso pérdida clasificación	10000
Tamaño imagen	512×512
Tamaño del batch	15
Optimizador	SGD
Learning rate	0.01
Momentum	0.9
Weight decay	0.0001
Optimizador	AdaBound
Learning rate	0.001
Final learning rate	0.1
Learning rate Scheduler	
Factor	0.1
Paciencia	25
Threshold	0.01

Tabla 4.1: Configuración baseline con RetinaNet

Resultados con AdaBound

Al entrenar con AdaBound se aprecia una mayor rapidez y mejores resultados que entrenando con SGD. Se entrena durante 152 épocas obteniendo los mejores resultados en la época 96 y se redujo la *learning rate* en las épocas 94, 123 y 149.

Los resultados se pueden apreciar en la tabla 4.2.

4.3.2. YOLO v3

Como el presente trabajo no usa YOLO v3 ni se especializa en sus configuraciones se utilizó las por defecto y la implementación de [53] y se entrenó durante 272 épocas.

Los resultados se pueden apreciar en la tabla 4.2.

4.3.3. Análisis

De acuerdo a los autores de RetinaNet y de YOLO v3, es de esperarse que RetinaNet tenga mejor comportamiento, puesto que YOLO v3 se destaca más en velocidad de detección que un mejor mAP. De todas formas YOLO v3 supera a RetinaNet con SGD, es solo la versión con Adabound la que da un salto interesante y alcanza un mAP de 0.65740, el que es utilizado como el *baseline* a superar por DLDENet.

Modelo	RetinaNet SGD	RetinaNet AdaBound	RetinaNet AdaBound	YOLO v3
Threshold	0.5	0.5	0.25	0.5
I_{lc}	675	840	**912**	715
I_{bc}	**2994**	2951	2815	2686
I_{bi}	**6**	49	185	314
I_{li}	285	120	**48**	245
Precisión	**0.964**	0.948	0.914	0.909
Recall	0.678	0.829	**0.869**	0.677
Accuracy	0.920	**0.946**	0.921	0.842
Clase	AP	AP	AP	AP
HP	0.20604	0.40024	**0.46229**	0.24299
adidas	0.13706	0.39146	**0.43495**	0.10719
aldi	0.42707	0.66038	**0.71064**	0.49096
apple	0.52449	0.54386	**0.59234**	0.44318
becks	0.43480	0.60939	0.60939	**0.61111**
bmw	0.48882	0.54850	**0.57250**	0.43095
carlsberg	0.25996	0.37498	**0.42682**	0.33308
chimay	0.50981	0.56604	**0.61891**	0.48752
cocacola	0.10494	0.42493	**0.54677**	0.07654
corona	0.70364	**0.89835**	0.89758	0.83512
dhl	0.03925	0.59877	**0.65936**	0.40184
erdinger	0.51796	0.59091	**0.60950**	0.45730
esso	0.70475	0.83392	**0.85201**	0.76305
fedex	0.27500	0.47692	**0.55194**	0.33090
ferrari	0.66116	0.73077	**0.76042**	0.46507
ford	0.43195	0.77778	**0.80147**	0.52639
fosters	0.66275	0.75676	**0.78051**	0.68542
google	0.51822	0.89736	**0.94510**	0.73237
guiness	0.67931	0.89205	**0.90604**	0.84703
heineken	0.40135	0.50052	**0.52442**	0.64203
milka	0.23622	0.38760	**0.42815**	0.22612
nvidia	0.24624	0.42070	**0.47107**	0.25651
paulaner	0.75762	**0.92244**	**0.92244**	0.71687
pepsi	0.09639	0.30820	0.34582	**0.35934**
rittersport	0.52956	0.57611	**0.61055**	0.33859
shell	0.31765	0.35714	**0.39048**	0.23128
singha	0.66371	0.89652	**0.92335**	0.59820
starbucks	0.82523	**0.93176**	**0.93176**	0.72165
stellaartois	0.53849	0.74919	**0.76907**	0.52080
texaco	0.60537	0.66880	**0.68381**	0.64578
tsingtao	0.55837	0.66222	**0.73177**	0.66666
ups	0.29346	0.51211	**0.56571**	0.31386
mAP	0.44865	0.62083	**0.65740**	0.48455

Tabla 4.2: Resultados de RetinaNet usando SGD y Adabound y YOLO v3 con distintos *thresholds*.

4.4. DLDENet

En este sección se entra en detalle a explicar las diversas configuraciones que se utilizaron para los experimentos, los distintos experimentos que se llevaron a cabo, sus resultados y comentarios para analizar los resultados obtenidos.

En primera instancia se genera un nuevo *baseline* con la configuración completa de DLDENet y luego se realizan varios experimentos para ver la sensibilidad a los distintos hiperparámetros y si mejora o empeoran los resultados y poder entender la contribución de cada una de las modificaciones.

4.4.1. Configuraciones generales

Backbones

Para generar la *Feature Pyramid Network* se ocupa como *backbone* Resnet-50 pre entrenado sobre Imagenet [23]. El pre entrenamiento es parte del framework PyTorch [45], puesto que cuenta con los pesos de las distintas versiones de ResNet de forma online y son éstos pesos los ocupados para inicializar el extractor de características que forman la FPN.

Optimizador

El optimizador utilizado es Adabound [39], el cual cuenta con mayor rapidez de convergencia que el clásico *descenso del gradiente estocástico* (SGD) [33] sin pérdida de calidad en los resultados. Además se ocupa un *learning rate scheduler* para ir ajustando la taza de aprendizaje si el modelo deja de mejorar durante cierto número de épocas.

En todos los experimentos, salvo se indique lo contrario, en el optimizador se ocupa una *learning rate* inicial de 0,001 para luego converger a una *learning rate* de 1 y en el *learning rate scheduler* se ocupa una paciencia de 10 épocas, un factor de ajuste de 0.1 y un *threshold* de mejora porcentual de 0.01. Es decir, se multiplica la taza de aprendizaje por 0.1 una vez que hayan pasado 10 épocas sin mejoras mayores a un 1 % relativo a la mejor evaluación obtenida anteriormente.

Inicialización

La inicialización del *backbone* está dado por los pesos pre entrenados de ResNet provistos por PyTorch. El resto de los módulos para computar los valores de la regresión y clasificación son inicializados de la forma por defecto que provee PyTorch también.

La única inicialización a destacar es la del *bias* en el módulo de clasificación, pues como se discute en la sección 3.2.1, éste puede ser inicializado con un *prior* lo que permite partir pre-

Backbone	ResNet-50
Tamaño imágenes	512×512
Data augmenation	El pipeline indicado en la sección 2.5
Tamaño del *batch*	10
Tamaño del *embedding*	256
Embeddings normalizados	Sí
Pre entrenamiento	Sobre Imagenet gracias a PyTorch
Bias	Inicializado en 0
Anchors	
Tamaños	20, 40, 80, 160, 320
Escalas	2^0, $2^{1/3}$, $2^{2/3}$
Relaciones de aspecto	0.5, 1, 2
Función de pérdida	DLDE Loss
α	0.25
γ	2
Peso *foreground* w_f	100
Peso Focal Loss $\lambda_{\mathrm{i}ffl}$	100
Peso regresión λ_{reg}	10
Peso similitud λ_{csl}	1
Peso dispersión $\lambda_{\mathrm{di}sp}$	10
Optimizador	AdaBound
Learning rate	0.001
Final learning rate	1
Learning rate scheduler	
Paciencia	10
Threshold	0.01
Factor	0.1

Tabla 4.3: Configuración base de DLDENet para los experimentos.

diciendo una misma probabilidad para cada clase al inicio del entrenamiento. En la mayoría de los experimentos el *bias* simplemente se inicializa en cero, salvo indicado lo contrario.

Hiperparámetros

En la tabla 4.3 se pueden apreciar los hiperparámetros base que se ocupan en todos los experimentos. Cada experimento busca editar alguno de estos parámetros de forma de poder medir la sensibilidad y el aporte que hace cada uno de éstos.

4.4.2. Resultados

Con todas la mejoras teóricas de DLDENet se corre un experimento con la configuración base. El entrenamiento se ejecuta durante 100 épocas logrando sus mejores resultados en la época 75. El *scheduler* disminuyó la *learning rate* en las épocas 87 y 98, es decir, se consiguió

Exp.	1	2	3	4	5	6	7	8	9	10	11	12
Norm.	Sí	Sí	Sí	Sí	**No**	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Bias	Sí	**No**	Sí	Sí	Sí	Sí	**No**	-1	-2	-1	Sí	-1
Prior	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	**0.25**	0.5
λ_{csl}	1	1	1	1	1	**0**	**0**	**0.1**	1	1	1	0
$\lambda_{\mathrm{dis}p}$	10	10	**0**	10	10	10	10	10	10	10	10	10
w_f	100	100	100	**0**	100	100	100	100	100	100	100	100
$\lambda_{\mathrm{i}ffl}$	100	100	100	100	100	100	100	100	100	100	100	100
I_{lc}	951	941	925	934	834	941	904	950	**958**	945	926	944
I_{bc}	2505	2737	2762	2885	**2950**	2652	2876	2275	1441	2606	2726	2535
$I_{b\mathrm{i}}$	495	263	238	115	**50**	348	124	725	1559	394	274	465
$I_{l\mathrm{i}}$	9	19	35	26	126	19	56	10	**2**	15	34	16
Pre.	0.936	0.955	0.944	0.933	0.95	0.950	**0.975**	0.936	0.895	0.959	0.946	0.935
Rec.	0.927	0.936	0.909	0.907	0.825	0.931	0.918	0.926	0.893	**0.944**	0.912	0.92
Acc.	0.857	0.918	0.918	0.948	0.945	0.895	**0.949**	0.799	0.580	0.887	0.910	0.863
mAP	0.761	0.773	0.740	0.745	0.558	**0.775**	0.756	0.758	0.739	0.774	0.745	0.750

Tabla 4.4: Resultados de los distintos experimentos de DLDENet.

el mejor resultado sin necesidad de cambiar la *learning rate*.

Los resultados se puede apreciar en la tabla 4.4 en el experimento 1 y muestran grandes avances sobre todos los otros resultados anteriores, incrementando el mAP de RetinaNet desde 0.65 a 0.76, 11 puntos más. Con estos resultados se sabe que las mejoras funcionan, con los siguientes experimentos se ve por qué y cuáles son los cambios que impactan de mayor forma sobre el rendimiento de la red.

Para medir qué modificaciones son más relevantes para el rendimiento del modelo se ejecutan una serie de experimentos donde en cada uno se modifica uno de los hiperparámetros y se obtienen los resultados. En la tabla 4.4 se pueden apreciar los resultados de los distintos experimentos donde es clave mencionar que:

- *Exp.* indica el número del experimento de forma de poder indicar de qué experimento se está hablando de forma más fácil.
- *Norm.* indica si el experimento ocupó o no la normalización de los embeddings.
- *Bias* indica si se ocupó un *bias* en el módulo de clasificación para desplazar la sigmoide como se indica en la sección 3.1.1 o si se ocupó un *bias fijo*, es decir un valor que no se aprende si no que simplemente desplaza la curva por ese valor.
- *Prior* indica el valor que se desea obtener como probabilidad para cada clase al comienzo del entrenamiento. Cuando el valor es 0.5 simplemente se inicializa el *bias* en cero.
- Los valores $\lambda_{\cdot}$ y w_f indican el peso utilizado para equilibrar todas las funciones de pérdida en la función de pérdida final.
- Los valores $I_{\cdot}$ son los mencionados en la sección 4.2.

Exp.	1	2	3	4	5	6	7	8	9	10	11	12
HP.	0.605	0.605	0.537	0.637	0.275	0.606	**0.694**	0.625	0.633	0.626	0.623	0.544
adida.	0.597	0.601	0.473	0.591	0.227	0.577	0.633	0.508	0.511	**0.651**	0.499	0.561
aldi	0.749	0.799	0.803	0.755	0.558	**0.822**	0.736	0.740	0.732	0.772	0.749	0.728
apple	0.797	0.799	**0.867**	0.810	0.661	0.860	0.866	0.700	0.482	0.750	0.650	0.723
becks	**0.739**	0.698	0.638	0.648	0.574	0.680	0.690	0.686	0.680	0.676	0.730	0.647
bmw	0.762	0.635	0.795	0.747	0.659	0.882	0.655	0.756	0.762	**0.900**	0.795	0.729
carls.	0.610	0.582	0.588	0.590	0.335	0.582	0.573	0.654	0.649	0.616	0.634	**0.693**
chima.	0.654	**0.702**	0.598	0.655	0.477	0.617	0.668	0.634	0.522	**0.702**	0.538	0.690
cocac.	0.769	**0.816**	0.603	0.659	0.218	0.806	0.791	0.717	0.767	0.769	0.794	0.682
coron.	0.970	**1.000**	0.991	0.998	0.842	0.963	**1.000**	0.994	0.974	0.939	0.953	0.969
dhl	0.738	0.743	0.629	0.685	0.516	0.738	**0.760**	0.735	0.726	0.715	0.679	0.716
erdin.	0.799	0.793	0.710	0.828	0.649	0.817	**0.855**	0.799	0.714	0.776	0.834	0.737
esso	0.880	0.872	0.865	0.901	0.753	0.859	0.868	0.806	0.888	**0.902**	0.895	**0.902**
fedex	0.753	0.811	0.801	0.688	0.533	**0.817**	0.746	0.796	0.748	0.772	0.808	0.729
ferra	0.744	0.844	0.779	0.844	0.634	0.812	0.781	**0.860**	0.793	0.844	0.829	0.780
ford	0.795	0.779	0.742	0.820	0.538	0.807	0.719	0.726	**0.844**	0.737	0.692	0.760
foste.	0.832	0.842	0.818	0.733	0.746	**0.863**	0.778	0.838	0.829	0.862	0.786	0.722
googl.	0.918	0.909	0.868	0.932	0.835	0.972	0.909	0.951	0.817	0.925	**0.983**	0.940
guine.	0.920	0.938	0.916	0.948	0.726	0.929	0.948	0.930	0.915	**0.964**	0.941	0.922
heine.	0.750	0.746	0.668	0.714	0.442	0.745	0.728	**0.782**	0.740	0.761	0.661	0.751
milka	0.542	**0.560**	0.504	0.456	0.224	0.545	0.474	0.484	0.498	0.487	0.490	0.505
nvidi.	0.590	**0.649**	0.642	0.613	0.480	0.625	0.595	0.614	0.568	0.622	0.573	0.624
paula.	0.943	0.972	**0.998**	0.971	0.902	0.961	0.983	0.916	0.992	0.918	0.968	0.982
pepsi	0.530	0.504	0.592	0.386	0.174	0.504	0.576	0.559	0.558	**0.650**	0.530	0.529
ritte.	0.662	0.717	0.693	0.608	0.307	0.743	0.683	0.749	**0.753**	0.675	0.709	0.706
shell	0.516	0.546	0.514	0.428	0.360	0.527	0.475	0.526	0.520	**0.555**	0.466	0.493
singh.	0.971	0.879	0.969	0.959	0.856	0.906	0.936	**0.975**	0.899	0.941	0.932	0.948
starb.	0.982	0.971	0.984	0.954	0.878	**0.994**	0.967	0.956	0.987	0.990	0.954	0.990
stell.	0.798	0.800	0.685	0.766	0.576	0.810	0.725	0.785	0.754	**0.844**	0.790	0.809
texac.	0.841	**0.896**	0.846	0.814	0.619	0.832	0.762	0.748	0.749	0.749	0.735	0.787
tsing.	0.766	**0.882**	0.792	0.867	0.687	0.752	0.788	0.855	0.836	0.869	0.807	0.862
ups	0.833	0.864	0.798	0.840	0.608	0.866	0.836	**0.869**	0.813	0.830	0.827	0.859

Tabla 4.5: Resultados de AP por clase en los distintos experimentos de DLDENet.

4.4.3. Discusión

Normalización

Claramente la modificación más importante, pues es la que afecta con mayor importancia al mAP obtenido, es la normalización de los *embeddings*. El experimento 5, donde solamente se quita la normalización de la configuración base, obtiene los peores resultados de todos los experimentos. A pesar de que obtiene buenos resultados con los falsos positivos, pues es el experimento con menor cantidad de imágenes sin logos con detecciones, es precisamente porque el modelo es el que tiene peor *recall* de todos los experimentos (0.825), aunque sigue siendo mejor que YOLO v3 (0.677 en la tabla 4.2) pero no mejor que RetinaNet con Adabound (0.869 en la tabla 4.2).

Se puede interpretar este resultado como que el modelo aumenta su capacidad de generalización solo enfocándose en la dirección del *embedding* en vez de enfocarse en su dirección y además en su norma, delegando todo el trabajo de ajustar la sigmoide para obtener valores en $[0, 1]$ a la norma del vector clasificador de la clase, no gasta esfuerzo en ajustar la norma del *embedding* para aumentar o disminuir la probabilidad de éste, solamente su dirección. Este mismo proceso permite que el vector clasificador sea el único responsable de ajustar la varianza de la clase y separarse de los otros vectores clasificadores, no es responsabilidad del *embedding*.

Aquí es clave hacer la mención justa de que es la normalización el aporte relevante, no el que se esté hablando de *embeddings*. Los *embeddings* por sí mismos no son un aporte nuevo, todas las redes neuronales generan embeddings en su última capa, es un producto natural de cualquier red, en este trabajo se ahonda en éstos y se imponen condiciones que puedan permitir obtener mejores resultados, entre las cuales la normalización de éstos es lo más relevante.

Bias

Uno de los resultados más interesantes es todo lo que tiene que ver con el *bias* en el módulo de clasificación. Como se menciona en las secciones anteriores y como destacan los autores de RetinaNet [37], su arquitectura tiene un entrenamiento demasiado inestable al comienzo debido a que la función de pérdida es dominada por los *anchors* del background mal clasificados o con probabilidades de pertenecer a alguna clase relativamente altas ($\geq$ 0,3). En este trabajo esto se pudo comprobar empíricamente puesto que sin el *prior* en el *bias* de RetinaNet no se puede obtener convergencia del modelo, simplemente diverge y no aprende nunca.

DLDENet no sufre de este problema, incluso, sorprendentemente, el modelo obtiene mejores resultados simplemente eliminando el *bias* del módulo de clasificación, como se puede observar en el experimento número 2. Esto significa que sin mover el centro de la sigmoide (que en 0 obtiene 0.5) obtiene mejores resultados que moviéndola para hacer más estricta la clasificación. Una posible opción para interpretar este resultado puede ser que el modelo

sin *bias* solo puede ocupar la extracción de características y la dirección del *embedding* para clasificarlo de manera correcta y mejorar la precisión y *recall* al detectar y disminuir la pérdida producto de los *embeddings* del *background* y esto fuerza al modelo a extraer mejores *embeddings*, pues elimina la posibilidad de ocupar el *bias* como herramienta para disminuir la pérdida producto de los *embeddings* del *background* mal clasificados.

Durante los experimentos se observa que el *bias* siempre se ajusta a valores negativos cuando el modelo lo aprende, esto permite que la detección sea más estricta (ya que para que un *embedding* tenga mayores probabilidades necesita apuntar aún más en la dirección del vector clasificador) y permite que la pérdida baje bastante gracias a los *embeddings* del *background* que bajan sus probabilidades (bajan los falsos positivos). Si se quita el *bias* el modelo no puede hacer más estricta la clasificación y solo tiene la opción de generar mejores *embeddings* y extraer mejores características para discriminar correctamente entre *foreground*, *background* y las diferentes clases.

Con esto, el vector clasificador contiene toda la información de la clase, su norma regula la confianza que tiene el modelo cuando un *embedding* apunta en cierto grado en la misma dirección y así disminuir la cantidad de falsos positivos y aumentar los verdaderos positivos. Se debe notar que, como se menciona en la sección 3.1.1, la norma del vector clasificador permite al modelo tener una sigmoid más *vertical*. Tener una sigmoid más vertical, sin ocupar el *bias*, indica un aspecto claro: cuando el *embedding* y el vector clasificador tienen una distancia angular menor a 90° (de forma que la similitud coseno es mayor a 0) el modelo le entrega más probabilidad. Y si no es así, si el caso es que el *embedding* tiene una distancia angular mayor a 90° el modelo entrega una baja probabilidad. Esto se puede entender como que el modelo está siendo más *confiado*, está *más seguro* de sus predicciones y los *embeddings* que genera y la dirección correcta de la clase.

Foreground embeddings y la dispersión

Se puede apreciar de los experimentos 3 y 4 que si se quitan los pesos que aumentan la pérdida de los elementos del *foreground* y se quita la función de pérdida que permite que los vectores clasificadores apunten en direcciones perpendiculares el rendimiento del modelo es peor.

En el experimento 3, sin la función de pérdida que fuerza que los vectores clasificadores se dispersen disminuye el *recall* y esto afecta al mAP que baja de 0.761 a 0.740. A pesar de que la precisión aumenta, lo que parece contraintuitivo, esto puede ser debido a que como el modelo se confunde más entre clases predice menos para no equivocarse tanto y afectar menos a la pérdida. Por eso el leve aumento de 0.008 en precisión pero una baja de 0.018 en el *recall*, más del doble que lo que mejora en precisión, por ende la baja del mAP.

Sin aumentar la pérdida de los *foreground embeddings* el modelo baja desde su resultado base con un mAP de 0.761 a 0.745 (experimento 4), principalmente por una baja en el *recall* y una leve baja en la precisión, lo que tiene sentido con la teoría puesto que aumentar la pérdida de estos *embeddings* es para aumentar el *recall* sin perder precisión. En efecto, la precisión incluso sube 0.003 puntos y el *recall* sube 0.020, lo que permite el aumento en el

mAP.

Similitud

Otra sorpresa de los resultados es el experimento 6, que elimina la función que hace que los *embeddings* estén más forzados a apuntar en la misma dirección que los vectores clasificadores, de hecho, logra el mejor mAP de todos los experimentos y suben todas las métricas respecto a la configuración base, es decir el modelo detecta más y mejor.

Esto indica que si se aumenta la similitud entre los *embeddings* y los vectores clasificadores se puede perder la varianza interna de la clase, pues se fuerza que todos los *embeddings* de una misma clase apunten exactamente en la misma dirección, lo que hace perder información puntual de cada instancia y por ende se pierde generalización en el modelo. En palabras sencillas, si todos los *embeddings* de una clase en el conjunto de entrenamiento apuntan exactamente en la misma dirección, luego el modelo detectando objetos que nunca ha visto, si uno de los *embeddings* de las instancias de esa clase no apunta *exactamente* en esa dirección se pierde, no se detecta o se clasifica mal. Esto se ve reflejado en el aumento en la precisión y el *recall* desde la configuración base de 0.936 a 0.950 y de 0.927 a 0.931 respectivamente.

Con este resultado se configura el experimento 7, sin *bias* y sin la función de pérdida que fuerza la similitud, el cual logra la mejor precisión y el mejor *accuracy* de todos los experimentos pero baja el *recall* en 0.012 lo que afecta al mAP.

Bias fijo

Con los resultados obtenidos en el experimento 7 se intuye que el *bias* sirve cuando no se está forzando la similitud pero por la naturaleza de la red y cómo se inicia el entrenamiento no logra aprender desde una posición muy cómoda. Esto se explica puesto que al comienzo del entrenamiento la pérdida está dominada por *background embeddings* que tienen probabilidades cercanas a 0.5 en vez de 0 y la forma más fácil de hacer que esas probabilidades disminuyan es haciendo que el *bias* se haga cada vez más negativo de forma de hacer la clasificación más estricta.

Entonces, en vez de ocupar un *bias* que la red puede aprender se puede fijar en cierto valor y ver qué resultados se obtienen. Así se configuran los experimentos 8, 9, 10, 11 y 12, donde en el 11 en vez de un *bias fijo* se ocupa un *prior* para partir con menores probabilidades y evitar ese aprendizaje de un *bias* tan negativo en el comienzo del entrenamiento.

Esta pequeña configuración, que quita un parámetro a aprender y lo transforma en hiperparámetro, logra mejorar la precisión y el *recall* del modelo sin *bias* y logra superar en 0.001 al mAP, una ventaja leve explicable gracias a que hay más falsos positivos en las fotografías sin logos. Con una configuración con un *bias* fijo de -1 se logra el mejor *recall* de todos los experimentos.

Tabla 4.6: Resultados comparados con otros métodos.

Método	Precisión	Recall	Accuracy	mAP
Bianco et al. 2015 [18]	0.909	0.845	0.884	-
Bianco et al. 2015 + Q.Exp [18]	0.971	0.629	0.904	-
Oliveira et al [43]	0.955	0.908	-	-
Bianco et al. 2017 [19]	**0.976**	0.676	0.910	-
YOLO v3	0.909	0.677	0.842	0.484
RetinaNet: threshold 0.5	0.948	0.829	0.946	0.620
RetinaNet: threshold 0.25	0.914	0.869	0.921	0.657
DLDENet: sin bias, sin similitud	0.975	0.918	**0.949**	0.756
DLDENet: sin similitud	0.950	0.931	0.895	**0.775**
DLDENet: Bias fijo en -1	0.959	**0.944**	0.887	0.774

4.4.4. Comparaciones con otros métodos

En la tabla 4.6 se muestra como DLDENet se comporta comparado con otros métodos sobre el mismo dataset. Como se aprecia, las diferentes versiones de DLDENet logran los mejores resultados, solo en la precisión no se logra el mejor resultado pero es engañoso, puesto que esa mínima diferencia de 0.001 puede ser superada simplemente aumentando el *threshold* de detección pero afectaría al resto de las métricas. Incluso, como se puede apreciar, Bianco et al. en sus resultados del 2017 obtiene un *recall* muy bajo, similar al de YOLO v3. El mAP no fue computado en los experimentos de Bianco y Oliveira, por eso no están presentes en la tabla.

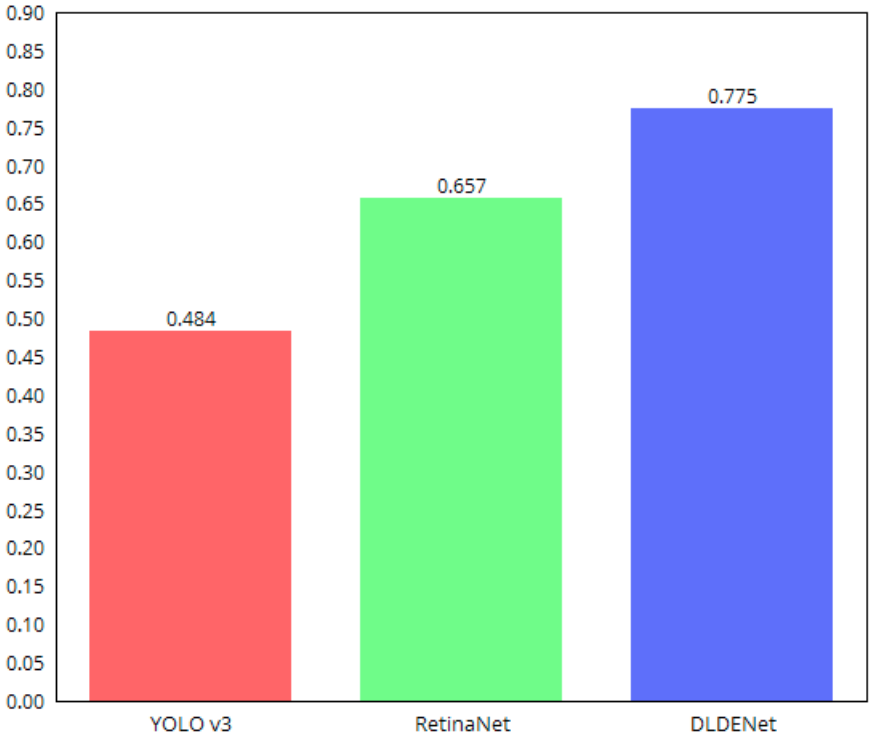

Figura 4.3: Comparación de los tres distintos modelos en cuanto a su mAP logrado sobre el conjunto de test del dataset FlickrLogos-32.

Conclusión

Detectar objetos en imágenes, en el contexto de visión artificial, es el problema donde dada una fotografía se busca que de manera automática, sin intervención humana, un algoritmo sea capaz de detectar algunas clases de objetos dentro de la escena y para cada uno de estos indicar dónde está el objeto y a qué clase pertenece.

En este trabajo se presenta el problema con un dataset llamado FlickrLogos-32 [48], que tiene 32 logos de marcas distintas en escenas comunes como calles, bares, oficinas y otros espacios y se crea un *baseline* con RetinaNet [37] y YOLO v3 [46], dos de los modelos del estado del arte en este problema. RetinaNet se destaca por su precisión y YOLO v3 por su velocidad.

Se toma RetinaNet como base y se proponen mejoras a la función de pérdida y a la arquitectura en sí misma para lograr mejorar los resultados del detector de objetos. Esta nueva arquitectura se bautiza como *DLDENet*, que proviene del nombre más largo *Deep Local Directional Embeddings Network*.

El nombre indica uno de los cambios claves que se proponen y que logran mejorar notablemente los resultados de RetinaNet: esta nueva arquitectura genera *embeddings*, vectores normalizados que representan distintas secciones de la fotografía los cuales son utilizados para clasificador los objetos presentes en esta. Los vectores son clasificados utilizando su similitud coseno con los *vectores clasificadores*, de los cuales se tiene uno por cada clase a detectar, y esa similitud se hace pasar por una función sigmoide para obtener valores entre 0 y 1. *Deep* viene de que la arquitectura es de *Deep Learning* y los embeddings se generan al pasar por decenas de capas. *Local* viene de que se generan *embeddings* en múltiples lugares de los fotografía localizados específicamente para poder identificar *dónde* están los objetos, en efecto, en una fotografía de 512×512 se generan más de $49,000$ *embeddings*. Y *Directional* puesto que se ocupa la dirección de estos *embeddings* para su clasificación, lo que aporta información más precisa ya que están normalizados, la norma de los *embeddings* no afecta a la decisión del modelo.

Aparte de las mejoras a la arquitectura se proponen cambios a la función de pérdida de RetinaNet. Se aumenta la pérdida de los *foreground embeddings*, de los que sí representan a un objeto para que la pérdida no esté dominada solamente por los *background embeddings* y se pueda aumentar la precisión y *recall* del modelo enfocándose en lo que de verdad importa, no en el *background*. También se agrega un componente que obliga a los vectores clasificadores a alejarse unos de otros y disminuir la confusión del modelo entre clases.

Estas mejoras, como se puede apreciar en la sección de resultados y discusión, permiten a DLDENet superar con creces a RetinaNet, pasando de un mAP de 0.657 con RetinaNet a un mAP de 0.775 con DLDENet, demostrando que los cambios funcionan, son útiles y permiten que la red extraiga de mejor forma la información y clasifique más y mejor.

El trabajo fue publicado en la *38th International Conference of the Chilean Computer Science Society (SCCC 2019)* [31].

Como trabajo futuro se desea poder evaluar las diferencias entre la similitud coseno y simplemente ocupar distancia euclideana así como también poder evaluar esta nueva arquitectura sobre COCO [38] utilizando imágenes en HD y la configuración óptima encontrada gracias a la experimentación, de forma de medir la robustez y calidad de las predicciones de esta nueva red pero en otro conjunto de datos.

Bibliografía

[1] A beginner's guide to object detection. https://www.datacamp.com/community/tutorials/object-detection-guide. Visitado: 2019-08-08.

[2] A comprehensive guide to convolutional neural networks — the eli5 way. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. Visitado: 2019-08-29.

[3] Convolution 2d. https://embarc.org/embarc_mli/doc/build/html/MLI_kernels/convolution_2d.html. Visitado: 2019-08-29.

[4] Data augmentation | how to use deep learning when you have limited data. https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/. Visitado: 2019-09-06.

[5] Deep learning for computer vision by rajalingappaa shanmugamani. https://www.oreilly.com/library/view/deep-learning-for/9781788295628/d9eef68b-4586-472c-bd5c-a244471a277f.xhtml. Visitado: 2019-08-08.

[6] Evaluating detection (intersection over union). https://www.oreilly.com/library/view/hands-on-convolutional-neural/9781789130331/a0267a8a-bd4a-452a-9e5a-8b276d7787a0.xhtml. Visitado: 2019-08-30.

[7] Fathers of the deep learning revolution receive acm a.m. turing award. https://www.acm.org/media-center/2019/march/turing-award-2018. Visitado: 2019-08-22.

[8] Find related images with reverse image search. https://support.google.com/websearch/answer/1325808?co=GENIE.Platform%3DDesktop&hl=en. Visitado: 2019-08-08.

[9] Flickr. https://www.flickr.com/. Visitado: 2019-09-12.

[10] Keras conv2d and convolutional layers. https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/. Visitado: 2019-08-29.

[11] Multilayer perceptron example. https://raw.githubusercontent.com/ledell/sldm4-h2o/master/mlp_network.png. Visitado: 2019-08-22.

[12] Object detection part 4: Fast detection models. https://lilianweng.github.io/lil-log/2018/12/27/object-detection-part-4.html. Visitado: 2019-08-30.

[13] Pytorch tensor basics. https://www.kdnuggets.com/2018/05/pytorch-tensor-basics.html. Visitado: 2019-08-29.

[14] Solving some image processing and computer vision problems with python libraries. https://sandipanweb.wordpress.com/category/computer-vision/. Visitado: 2019-08-30.

[15] The state-of-the-art in machine learning.

[16] Visión artificial. https://es.wikipedia.org/wiki/Visi%C3%B3n_artificial. Visitado: 2019-08-08.

[17] E. Khvedchenya V. I. Iglovikov A. Buslaev, A. Parinov and A. A. Kalinin. Albumentations: fast and flexible image augmentations. *ArXiv e-prints*, 2018.

[18] Simone Bianco, Marco Buzzelli, Davide Mazzini, and Raimondo Schettini. Logo recognition using cnn features. In Vittorio Murino and Enrico Puppo, editors, *Image Analysis and Processing — ICIAP 2015*, pages 438–448, Cham, 2015. Springer International Publishing.

[19] Simone Bianco, Marco Buzzelli, Davide Mazzini, and Raimondo Schettini. Deep learning for logo recognition. *CoRR*, abs/1701.02620, 2017.

[20] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

[21] Vitaly Bushaev. Stochastic gradient descent with momentum.

[22] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[24] Jiankang Deng, Jia Guo, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. *CoRR*, abs/1801.07698, 2018.

[25] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[26] Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, Dec 1999.

[27] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a me-

chanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[29] Yandong Guo and Lei Zhang. One-shot face recognition by promoting underrepresented classes. *CoRR*, abs/1707.05574, 2017.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[31] F. S. Herrera and J. M. Saavedra. Dldenet: Deep local directional embeddings with increased foreground focal loss for object detection. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–8, Nov 2019.

[32] S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[33] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[36] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016.

[37] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.

[38] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.

[39] Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. Adaptive gradient methods with dynamic bound of learning rate. *CoRR*, abs/1902.09843, 2019.

[40] Bernard Marr. How much data do we create every day? the mind-blowing stats everyone should read. `https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read`.

Visitado: 2019-08-08.

[41] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[42] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[43] Gonçalo Oliveira, Xavier Frazão, André Pimentel, and Bernardete Ribeiro. Automatic graphic logo detection via fast region-based convolutional networks. *CoRR*, abs/1604.06083, 2016.

[44] Rafael Padilla. Metrics for object detection.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[46] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[47] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[48] Stefan Romberg, Lluis Garcia Pueyo, Rainer Lienhart, and Roelof van Zwol. Scalable logo recognition in real-world images. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval*, ICMR '11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.

[49] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[50] Fabián Souto. Torchsight. `https://github.com/SetaSouto/torchsight`, 2019.

[51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[52] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[53] Ultralytics. Yolov3 in pytorch.

[54] Eric A. Weiss. Turing award winners. In *Encyclopedia of Computer Science*, pages 1795–1797. John Wiley and Sons Ltd., Chichester, UK.

[55] Paul John Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley-Interscience, New York, NY, USA, 1994.

[56] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.