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

EVALUACIÓN DE AUTOATENCIÓN NO LOCAL EN MÉTODOS *ANCHOR-FREE* PARA LA DETECCIÓN DE OBJETOS EN IMÁGENES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

LUKAS DARÍO PAVEZ BAHAMONDES

PROFESOR GUÍA:
JOSÉ MANUEL SAAVEDRA RONDO

MIEMBROS DE LA COMISIÓN:
CLAUDIO GUTIÉRREZ GALLARDO
JOSE BENGURIA DONOSO

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: LUKAS DARÍO PAVEZ BAHAMONDES
FECHA: 2021
PROF. GUÍA: JOSÉ MANUEL SAAVEDRA RONDO

EVALUACIÓN DE AUTOATENCIÓN NO LOCAL EN MÉTODOS *ANCHOR-FREE* PARA LA DETECCIÓN DE OBJETOS EN IMÁGENES

La detección de objetos es un área de gran interés dentro de las tareas de visión por computadora, donde hoy en día se van proponiendo distintas arquitecturas de redes neuronales que intentan resolver el problema de distintas maneras. El estado del arte esta asociado con modelos basados en redes neuronales convolucionales profundas, que utilizan distintos filtros para extraer características de una imagen.

Los métodos para detección actuales se pueden clasificar de diversas maneras. Una de estas clasificaciones se basa en facilitar la regresión respecto a los *bboxes*, aquí se diferencian dos tipos de métodos: los *anchor-free* y *anchor-based*. Un *anchor* corresponde a un rectángulo base con el que se infiere la ocurrencia de un objeto, por lo que las redes *anchor-based* se centran en revisar esas regiones para realizar la detección. Si bien hoy en día las redes *anchor-based* tienen los mejores resultados, cuando se piensa en como se realiza la búsqueda utilizando *anchors* resulta contra-intuitivo, debido a que para poder identificar un objeto, una persona no se centra en regiones de su visión, sino que ocupa la mayor parte de la información que tiene disponible. Aquí entran las redes *anchor-free*, donde al no utilizar *anchors* se tienen que buscar otras maneras de encontrar las regiones de interés para la detección. Para este fin, utilizar técnicas de autoatención resulta relevante. Por lo tanto, en este trabajo se presenta una evaluación de dos métodos del estado del arte donde se agregan módulos de autoatención no local.

Para realizar la evaluación del impacto de agregar el módulo que utiliza *NonLocal features* se realizan experimentos con dos redes *anchor-free*: CornerNet y FCOS, además de proponer una arquitectura que es una mezcla entre ambas redes. Los experimentos propuestos consisten en evaluar el desempeño de la red antes y después de agregar el módulo de *NonLocal features*.

En este trabajo se puede ver que el utilizar *NonLocal features* resultó efectivo para el caso de dataset de ropa, donde incluso se logró superar métodos basados en *anchors*. Sin embargo en el caso de otros datasets se tuvieron resultados variables, donde se concluye que falta realizar más investigación.

Agradecimientos

En primer lugar quiero agradecer a mis amigos de bachillerato, con quienes entré a la universidad hace ya unos 8 años, y aunque cuando entramos a ingeniaría casi todos nos fuimos a especialidades diferentes, igual nos las arreglamos para seguir almorzando juntos casi todos los días.

También quisiera agradecer a mis amigos del laboratorio de robótica, a Cris, Gio, Jose, Nico, Peluka, Ulises, Camilo y el resto del equipo, con quienes trabajamos viviendo en el laboratorio para sacar adelante a Bender y a Maqui. Trabajando con los robots aprendí muchas cosas que nunca hubiera conocido si no hubiera entrado al laboratorio. Nunca se van a olvidar las pizzas, los palitos de ajo, las papas, los queques de cumpleaños, las salidas al esperanto y las partidas de Xonotic.

A mis compañeros de DualVision, Gio, Hans, Jp, Kevin y el resto del equipo, donde aprendí que aunque ya este terminando la u igual se siguen aprendiendo cosas nuevas siempre, sobre todo con Kevin.

Finalmente quiero agradecer a mi profesor guía José Saavedra por ayudarme durante todo el proceso del trabajo de memoria.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Motivación

En los últimos años, hemos visto significativos avances en IA, especialmente de la mano de *deep-learning*. Así, la tarea de visión por computadora, que corresponde al desarrollo de modelos computacionales a través de imágenes con el objetivo de interpretar el entorno, ha sido una de las áreas significativamente impactada a través de estos modelos de IA.

Existen múltiples tareas a resolver con visión por computadoras. Entre las más populares se encuentran las siguientes tareas: Clasificación de imágenes, que consiste en asignar una etiqueta (o clase) a una imagen; detección de objetos, que busca encontrar y localizar uno o más objetos en una imagen; segmentación semántica, tiene como objetivo asignar una clase a cada píxel de la imagen; segmentación de instancias busca identificar cada objeto detectado como una instancia diferente, aunque tengan la misma clase.

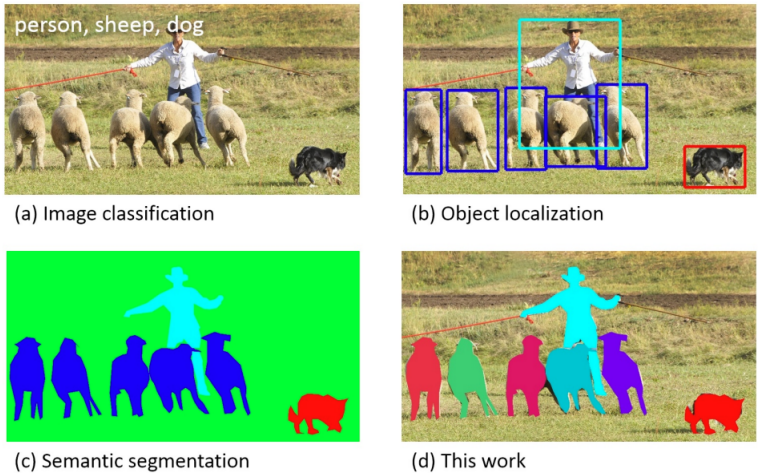

Figura 1.1: Diferentes tareas de visión por computadora. (a) Clasificación de imágenes. (b) Detección de objetos. (c) Segmentación semántica. (d) Segmentación de instancias. Fuente: [1].

Las metodologías que se usan actualmente se basan en redes neuronales profundas, donde se tiene un dataset de imágenes de distintos objetos y se pasan por la red para que la red aprenda a detectar en qué parte de la imagen hay un objeto. En este ámbito también hay dis-

tintas formas para que la red aprenda, la más utilizada es usar lo que se llama *anchor-boxes*, que corresponde a generar una gran cantidad de rectángulos que se sitúan en la imagen, que se transforman para contener a los objetos de interés. El objetivo del aprendizaje es encontrar esa transformación. Un ejemplo de modelo que usa *anchors* es RetinaNet [2], que construye una pirámide con niveles de distinto tamaño, donde asigna *anchors* de un tamaño específico a cada nivel.

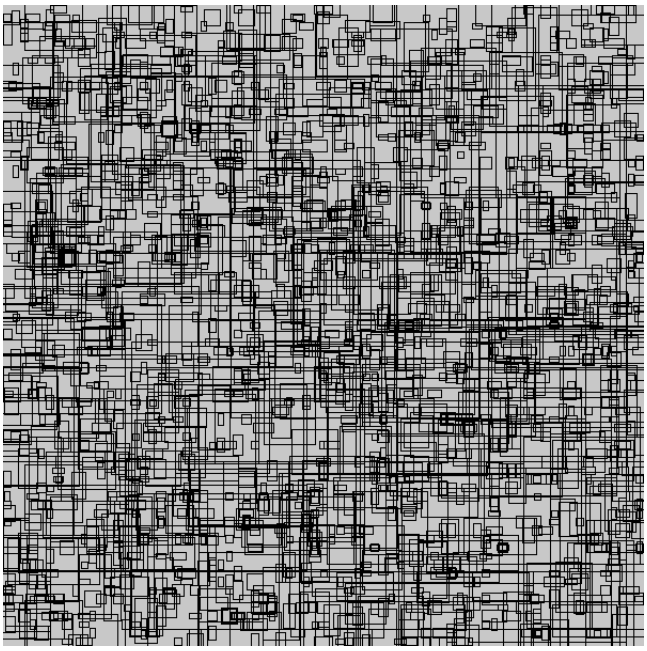

Figura 1.2: Muestra de anchors usados en RetinaNet. Fuente: [3]

Los métodos que no usan *anchors*, por lo general, se basan en generar *heatmaps* en la imagen, indicando dónde se encuentran los objetos de interés, y para esto se utiliza información que puede venir de distintas partes de la imagen. Como ejemplo FCOS [4] es un modelo *anchor-free*, donde también construye una pirámide, pero ahora cada punto de cada nivel realiza una regresión para encontrar las distancias desde el punto hasta el *bounding box*.

Si bien, la metodología *anchor-based*, actualmente, tiene los mejores resultados, para una persona es contra-intuitivo el detectar objetos revisando rectángulos, debido a que nosotros utilizamos gran parte de la información de la imagen que estamos viendo para detectar un objeto, y no solo un área acotada. Naturalmente, realizamos una correlación de distintas partes de la imagen para poder identificar un objeto. Por esto, se quiere estudiar métodos que no utilizan *anchors*, sino que se estudia el efecto de agregar información del contexto para realizar la detección, con el objetivo de determinar el efecto en la precisión y efectividad de aplicar estos métodos en la detección de objetos.

Por lo anterior, se propone aplicar métodos de autoatención no local. La atención surgió con el objetivo de acumular información de contenido de tamaño variable, donde su habilidad de aprender a enfocarse en información importante dado un contexto hizo que sea una componente crítica en modelos basados en texto. La autoatención se define como atención aplicada a un único contexto [5], y el objetivo de aplicar autoatención no local es la capacidad

de aplicar información de toda la imagen para realizar la detección.

1.2. Relevancia del problema

En éste trabajo de memoria se estudiará el impacto de aplicar *Nonlocal Features* en redes que no utilizan *anchors*, analizando cómo cambia su precisión y tiempos de inferencia al detectar objetos en una imagen. Un módulo de *NonLocal features* aplica información de toda la imagen para analizar cada punto.

Al no trabajar con *anchors* se evita el problema de definirlos para el entrenamiento, ya que una de las desventajas que tienen es que la precisión de la red se puede ver altamente afectada por como se definen, debido a que los *anchors* deben de ser capaces de capturar efectivamente un objeto, por lo que si sus tamaños cambian mucho, entonces no se detecta nada.

1.3. Hipótesis y Objetivos

1.3.1. Hipótesis

Considerando que *NonNocal Features* es una generalización de los métodos de *pooling* utilizados por los modelos de detección que son *anchor-free*, se propone que la aplicación de este módulo permitirá mejorar el desempeño de los modelos de detección que no utilizan *anchor-boxes*.

1.3.2. Objetivo General

Evaluar el impacto de agregar el módulo de *NonLocal Features* en redes de detección de objetos que no utilizan *anchors*.

1.3.3. Objetivos Específicos

1. Implementar módulo *NonLocal Features* en red CornerNet
2. Entrenar y evaluar CornerNet con el dataset COCO [1]
3. Entrenar y evaluar CornerNet con el dataset ModaNet [6]
4. Implementar módulo *NonLocal Features* en red FCOS
5. Entrenar y evaluar FCOS con el dataset COCO [1]
6. Entrenar y evaluar FCOS con el dataset ModaNet [6]
7. Comparar rendimiento obtenido entre las redes con el módulo implementado y las redes originales.

1.4. Estructura de la memoria

A partir de este punto, el documento se estructura de la siguiente manera:

- Capítulo 2: Marco teórico y estado del arte
  En este capítulo se explican los conceptos necesarios para entender todo el trabajo, además de presentar modelos del estado del arte investigados.

- Capítulo 3: Desarrollo
  En este capítulo se explica la metodología utilizada para resolver el problema, preparación de datasets, implementación y detalle de entrenamiento.

- Capítulo 4: Resultados experimentales y discusión
  En este capítulo se muestran los resultados obtenidos del entrenamiento y su respectivo análisis.

- Capitulo 5: Conclusiones
  Conclusiones obtenidas a partir de los resultados.

Capítulo 2

Marco teórico y estado del arte

A lo largo de este capítulo se explicarán los conceptos necesarios para entender el problema planteado, empezando desde las bases de *machine learning*, hasta explicar distintas arquitecturas de redes neuronales del estado del arte en el tema de detección de objetos.

2.1. *Machine learning*

Machine learning es una rama de inteligencia artificial y se define como la capacidad de las máquinas de aprender a partir de datos. Básicamente se tiene un algoritmo al que se le introducen datos y el algoritmo debe adaptarse a un comportamiento deseado.

Un algoritmo de *machine learning* tiene la forma de la Ecuación 2.1, donde una función H (el modelo) que tiene parámetros W se aplica a x (datos), y eso da un resultado y. Por lo que al tener datos, una tarea a cumplir y una forma de medir el rendimiento, se tiene el objetivo de encontrar la función H, y hay aprendizaje cuando al ir suministrando los datos a la función H, la métrica de rendimiento aumenta.

$$y = H_W(x) \tag{2.1}$$

Existen distintas formas de aprendizaje, y para cada una hay distintos modelos de implementación, la división clásica es aprendizaje supervisado, aprendizaje no supervisado y aprendizaje reforzado.

El aprendizaje supervisado se basa en que para cada dato que se introduce al modelo, se tiene una etiqueta, que corresponde a la respuesta deseada, por lo que al saber cual es la respuesta a la que se debe llegar entonces el modelo se puede adaptar para responder de forma correcta.

El aprendizaje no supervisado se basa en que los datos no están etiquetados, por lo que el modelo debe analizar los datos en busca de una relación y ser capaz de dividirlos de forma coherente, por lo que la salida del modelo no es conocida. También es llamado autosupervisado, debido a que el mismo modelo se va ajustando según la relación entre los datos que va recibiendo.

El aprendizaje reforzado consiste en un agente que debe realizar una tarea, para esto tiene recompensas positivas o negativas según las acciones que realiza. Esta relacionado a tomar la mejor decisión en cada paso.

2.2. Redes Neuronales

Las redes neuronales nacen del estudio de simular las neuronas biológicas y la comunicación entre ellas, donde se tiene un impulso de entrada, pasa por neuronas intermedias y se tiene un impulso de salida.

A partir de esto, se define una neurona como una unidad que tiene una o más entradas y una o más salidas, donde cada entrada se pondera por un valor llamado peso y se suma un valor llamado *bias*, para luego pasar la suma de todos los resultados por una función llamada función de activación, en la Figura 2.1 se puede ver un ejemplo de una neurona con 3 entradas y una salida, donde la salida y sigue la Ecuación 2.2, que corresponde a una función F aplicada a la suma de las entradas x_i multiplicadas por sus respectivos pesos w_i.

$$y = F(\sum_i x_i * w_i) \tag{2.2}$$

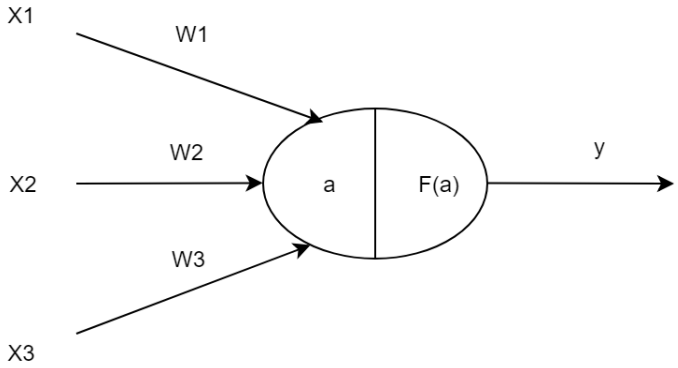

Figura 2.1: Ejemplo neurona con 3 entradas, Fuente: [7].

Una red neuronal consiste de un conjunto de neuronas, donde las neuronas se dividen en capas, las conexiones entre una capa y otra dependen de la arquitectura del modelo. En la Figura 2.2 hay un ejemplo de una red neuronal simple donde la entrada se pasa por una capa de neuronas y eso se conecta a una neurona de salida. En esta red, cada neurona esta conectada a todas las neuronas de la capa siguiente, cuando esto ocurre se llama una capa *fully connected*.

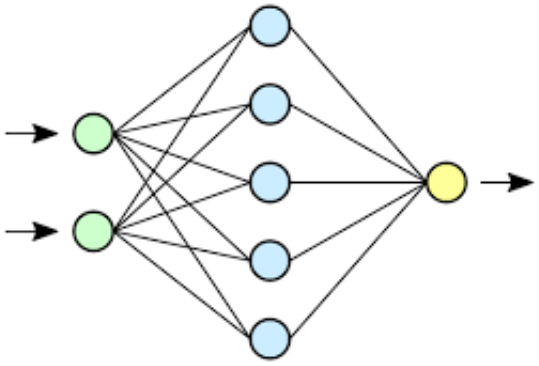

Figura 2.2: Ejemplo red neuronal con 2 entradas y una salida.

La forma en que una red aprende es actualizando el valor de los pesos y los *bias*. Como es un modelo de aprendizaje supervisado, se conoce el valor de la salida que corresponde a un dato de entrada, por lo que si da un valor incorrecto, se debe modificar cada peso de la red, esta modificación se aplica según como afecta cada neurona al resultado final, esto se hace con una operación llamada *backpropagation*, donde el error se propaga desde la última neurona hacia el resto para modificar sus pesos.

Para calcular cuánto es el error, se utiliza una función de costo, o *loss*, que indica el error del modelo. La función de *loss* depende del problema a resolver y del modelo utilizado. El objetivo del entrenamiento es minimizar el valor dado por el *loss*.

2.2.1. Funciones de *loss* populares

2.2.1.1. Mean Square Error (MSE)

MSE *loss*, o error cuadrático medio, también llamado *loss* L2, se calcula como el promedio de las diferencias al cuadrado entre el valor actual y con el valor predicho por la red $\hat{y}$.

$$L_{MSE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.3}$$

Dado su estructura, el *loss* MSE se utiliza cuando los datos siguen una distribución normal alrededor de un valor promedio. Al ser una diferencia al cuadrado, el error aumenta bastante cuando el valor predicho es diferente al esperado, por lo que es sensible a *outliers*.

2.2.1.2. Mean Absolute Error

MAE *loss*, o error absoluto medio, también llamado *loss* L1, se calcula como el promedio de las diferencias absolutas entre el valor actual y con el valor predicho por la red $\hat{y}$.

$$L_{MAE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{2.4}$$

Este *loss*, a diferencia de MSE, no es sensible a *outliers*, por lo que se usa cuando los

outliers no son una parte importante del problema a resolver.

2.2.1.3. Binary Cross Entropy (BCE)

Binary Cross Entropy *loss*, también llamado *log loss* se utiliza cuando se calculan puntajes entre 0 y 1. Y se calcula en partes, en el caso binario, cuando el valor actual y de la clase puede ser 1 o 0, se tiene la Ecuación 2.5, en el caso donde la clase actual es 1, se toma el valor negativo del logaritmo del valor predicho, esto hace que el objetivo sea aumentar el valor predicho. En el caso contrario, se toma la diferencia entre 1 y el valor predicho, con el objetivo de dirigir la predicción a 0, para que el valor dentro del logaritmo se acerque a 1, lo que hace que el error sea menor.

$$L_{BCE}(y, \hat{y}) = \begin{cases} -log(\hat{y}) & y = 1 \\ -log(1 - \hat{y}) & y = 0 \end{cases} \tag{2.5}$$

2.2.1.4. Intersection Over Union (IOU)

El IOU *loss* se utiliza cuando se hace una regresión de *bounding boxes* (*bboxes*), donde se analiza cuán sobrepuestas está una sobre la otra. Para esto ocupa la Ecuación 2.6, donde se tiene el *bounding box* del *ground truth* b, y el calculado por la red $\hat{b}$. "intersection" y "union" son funciones que calculan el área de la intersección y unión entre b y $\hat{b}$ respectivamente. El *loss* se calcula con la Ecuación 2.7, que busca maximizar el valor de la métrica IOU.

$$IOU(b, \hat{b}) = \frac{intersection(b, \hat{b})}{union(b, \hat{b})} \tag{2.6}$$

$$L_{IOU}(b, \hat{b}) = 1 - IOU(b, \hat{b}) \tag{2.7}$$

2.2.2. Redes neuronales convolucionales

Las redes neuronales convolucionales son redes neuronales que se caracterizan por tener capas que encargan de realizar una operación de convolución. En el ámbito de las imágenes, una convolución se aplica para filtrar la imagen, donde se tiene un kernel (o filtro), que es una matriz de tamaño pequeño, por ejemplo 3x3, que se posiciona en la imagen de entrada, y se realiza un producto punto entre el sector seleccionado de la imagen con el kernel, y el resultado se suma para obtener un valor de la imagen de salida, como se puede ver en la Figura 2.3. En una red convolucional, la imagen correspondería a la salida de una capa (o a la entrada), y el kernel corresponde a los pesos asociados a las entradas de las neuronas de la capa siguiente.

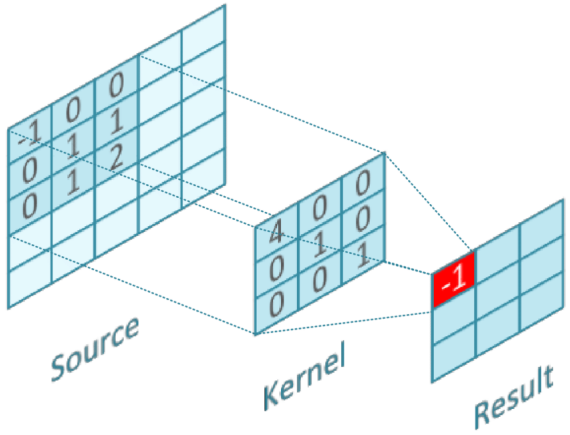

Figura 2.3: Operación de convolución con kernel 3x3.

Al tener capas convolucionales, se pueden usar los filtros para extraer características de distintas partes de la imagen, y cuando se comenzaron a utilizar estas capas se aumentó bastante la eficiencia en la detección. Una de las primeras redes conocidas en hacerlo es AlexNet [8]. La estructura propuesta por AlexNet es una red convolucional de 8 capas, entre las que se encuentran capas convolucionales, capas de *pooling* y capas *fully connected* (FC). Una operación de *pooling* corresponde a una reducción de tamaño al tomar un grupo de valores y pasarlo a un solo valor, como se puede ver en la Figura 2.4, donde se muestran dos ejemplos de *pooling*, uno donde se toma el máximo valor y otro donde se toma el promedio. Una capa *fully connected* corresponde a una capa donde cada neurona está conectada a todas las neuronas de la capa siguiente (Figura 2.2).

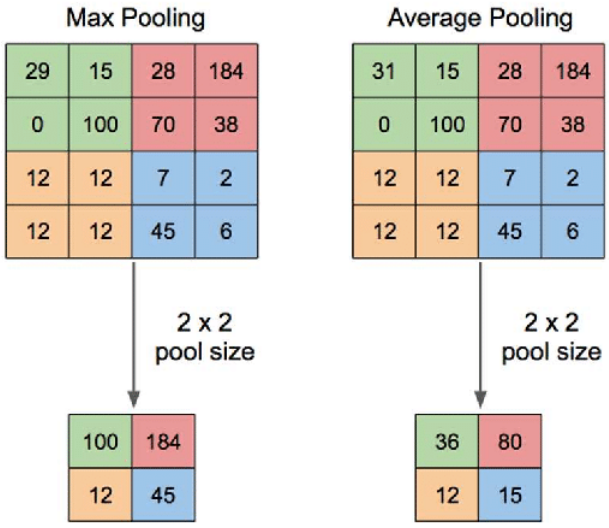

Figura 2.4: Operación de *max pooling* y *average pooling*.

2.3. *Backbones*

En la actualidad, las arquitecturas de las redes tienen una etapa inicial de extracción de características, y para esto utilizan redes conocidas, que se conocen como *backbone* de la

arquitectura. A continuación se muestran algunos *backbones* conocidos:

2.3.1. ResNet

La arquitectura de la red ResNet [9] introdujo la idea de bloques residuales, donde se toma la entrada, se pasa por capas convolucionales, y el resultado se suma con la entrada, de ahí el nombre "residual", se puede ver de manera más clara en la Figura 2.5.

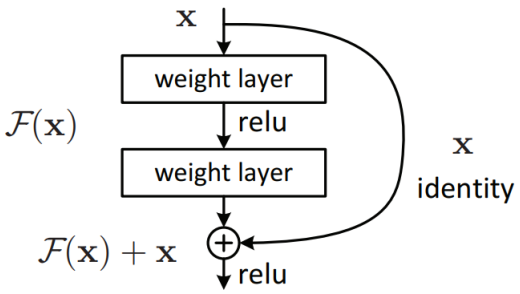

Figura 2.5: Bloque residual, Fuente: [9].

ResNet propone arquitecturas con distintas cantidades de capas, en la Figura 2.6 se muestran arquitecturas con 18, 34, 50, 101 y 152 capas, todas las arquitecturas se dividen en 5 etapas convolucionales (convN_x), donde en cada etapa se muestra la estructura de un bloque convolucional y cuantas veces se repite. Las redes que lo usan como *backbone* toman la salida de alguna de las etapas convolucionales y trabajan con eso.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				

Figura 2.6: Arquitecturas propuestas por ResNet, Fuente: [9].

2.3.2. Hourglass

Otra red utilizada como *backbone* es Hourglass [10]. Esta arquitectura propone un procesamiento donde se disminuye el tamaño de la entrada y luego lo aumenta, y esto se hace

repetidas veces, como se puede ver en la Figura 2.7. Esta arquitectura fue diseñada en un principio para estimación de pose de personas, pero al usarlo de *backbone* se utiliza para extracción de características para la red principal.

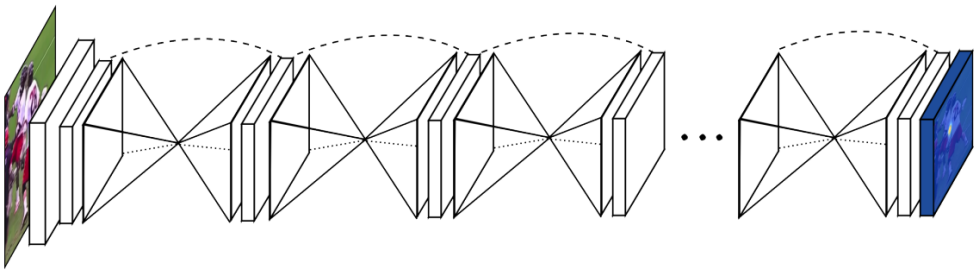

Figura 2.7: Arquitectura Hourglass, Fuente: [10]

2.3.3. Feature Pyramid Network (FPN)

En una red convolucional, como por ejemplo una ResNet, se parte con una imagen y se comienza a pasar por distintas etapas convolucionales, donde el tamaño de la imagen se va reduciendo cada vez más. La diferencia de la imagen entre cada etapa es que al inicio se tiene mayor resolución, pero a costa de que cada punto de la imagen tiene poca información semántica que sirve para realizar la detección. En las últimas etapas ocurre lo contrario, hay harta información pero poca resolución. Una forma de enfrentar esto es con una arquitectura de FPN [11], donde al llegar al último nivel que tiene mucha información semántica pero poca resolución, se comienza a generar niveles con mayor resolución aumentando el tamaño del nivel e incluyendo información de la etapa convolucional anterior (Figura 2.8), lo que hace esto es incluir un nivel con harta información semántica con un nivel con mayor resolución para ir recuperando información que se pudo haber perdido. Finalmente se utiliza cada nivel generado para realizar la predicción final.

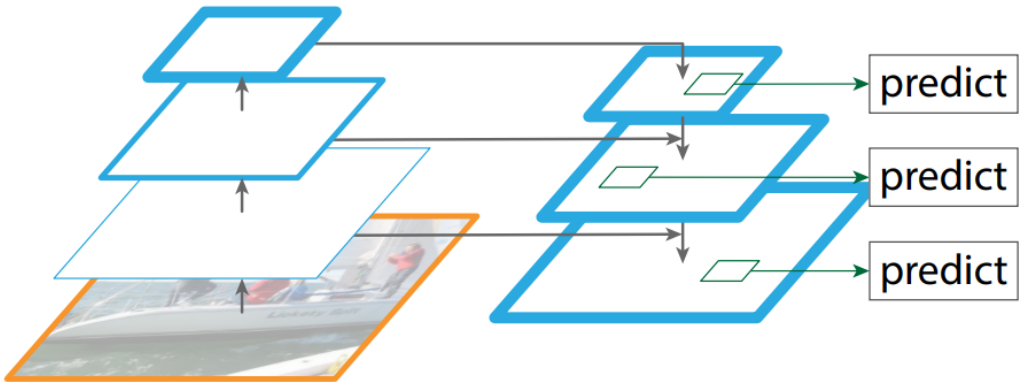

Figura 2.8: Arquitectura FPN, Fuente: [11]

2.3.4. PANet

PANet [12] tiene como objetivo el potenciar el flujo de la información que pasa a través de una FPN. La estructura se encuentra en la Figura 2.9, donde parte con un *backbone* compuesto de etapas convolucionales y se agrega una FPN (Figura 2.9.(a)). En la FPN el flujo de la información va en dirección *top-down*, donde se utilizan los niveles más pequeños para ir generando los siguientes. PANet propone agregar una estructura donde el flujo va en dirección *bottom-up*, tomando el último nivel generado por la FPN y va generando nuevos niveles más pequeños (Figura 2.9.(b)). El resto de la estructura (Figura 2.9.(c), .(d) y .(e)) corresponde a el procesamiento propuesto para realizar la detección, donde une los niveles de salida de la pirámide y se divide en una rama para obtener la clase y *bounding box* y otra para obtener una máscara con las predicciones.

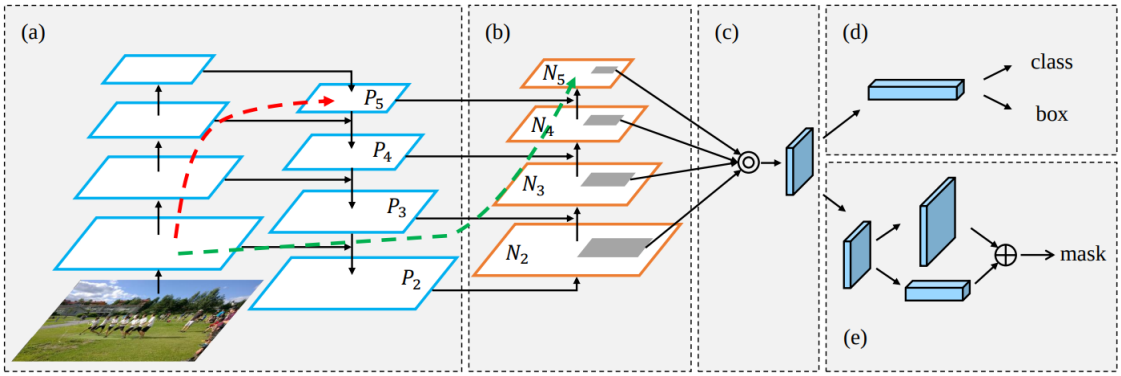

Figura 2.9: Arquitectura PANet, Fuente: [12]

2.4. Detección de objetos

En las tareas de visión por computadora, la detección de objetos va un paso más allá de la clasificación de imágenes, que busca asignar una clase a una imagen. En la detección se busca el localizar y etiquetar a cada instancia de objeto presente en la imagen, como en la Figura 2.10.

Para realizar detección de objetos se utilizan redes convolucionales. Con este tipo de redes se va perdiendo resolución de la imagen a medida que aumenta la profundidad de la red, donde se van generando mapas de características (*feature maps*) a través de las convoluciones. Si bien se va perdiendo resolución, lo que si aumenta es la información semántica de cada punto de los *feature maps* generados, donde un punto pasa a representar un sector de la imagen, llamado el campo receptivo, donde cada campo receptivo tiene información de distintas partes de la imagen.

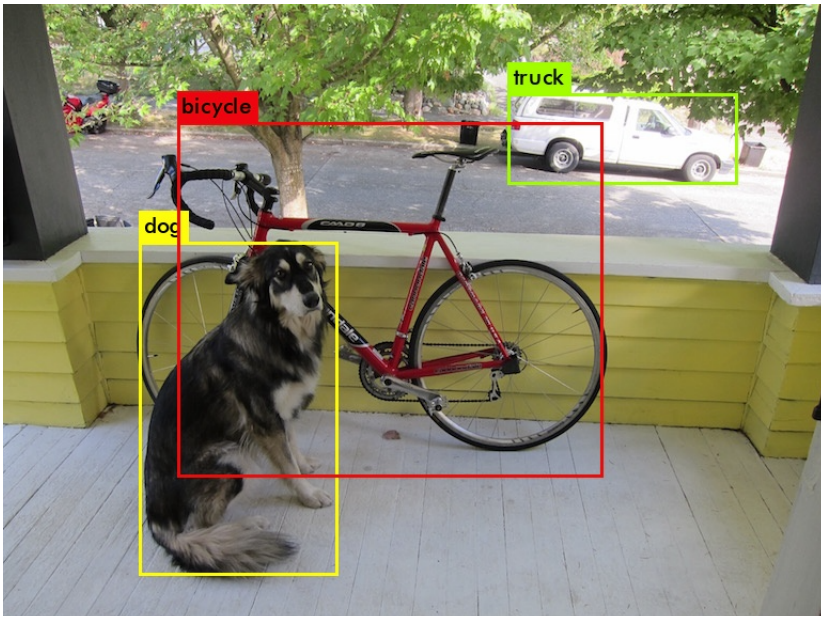

Figura 2.10: Ejemplo de detección de objetos, Fuente: [13]

2.5. Tipos de redes de detección de objetos

2.5.1. Redes de dos etapas

Las redes de dos etapas se dividen en una etapa de buscar en donde pueden haber objetos, independiente de la clase, es una etapa que extrae regiones de interés (ROI: Regions of Interest). La segunda etapa se encarga de tomar las ROI y realizar la clasificación. Este tipo de redes tienen buen rendimiento en localizar objetos y su clasificación, a costa de que el procesamiento tome una mayor cantidad de tiempo. Ejemplos de redes populares de dos etapas: todas las redes de la famila R-CNN, como R-CNN [14], Fast R-CNN [15] y Faster R-CNN [16].

2.5.2. Redes de una etapa

Este tipo de redes se centra en realizar la búsqueda de *bounding boxes* y su respectiva clase. Estas redes tratan el problema de detección de objetos como un problema de regresión, donde se toma una imagen de entrada y se aprenden las probabilidades de clase y las coordenadas del *bbox*. Por lo general ésta metodología tiene menor rendimiento que su contra-parte de 2 etapas, pero los tiempos de procesamiento son más bajos, por lo que son una alternativa a considerar cuando se prioriza la velocidad de detección. Ejemplos de redes populares de una etapa: todas las versiones de YOLO [17], FCOS [4], CornerNet [18] y SSD [19].

2.5.3. Redes *anchor-based*

Independiente de la cantidad de etapas de la red, una arquitectura puede ser *anchor-based*. Hasta ahora las redes que usan *anchors* tienen los mejores resultados. El utilizar *anchors*

significa que se reparten una gran cantidad de rectángulos en la imagen donde cada uno corresponde a una región de interés, donde se debe aprender la transformación de cada uno para ajustarse a los objetos de la imagen. En el caso de una red de 2 etapas como Faster R-CNN, los anchors son los ROI de la primera etapa, y la etapa debe realizar dos cosas: clasificar cada ROI como objeto o no objeto, y entregar un vector de transformación para el *anchor*. En el caso de una red de una etapa, se puede tomar como ejemplo SSD, donde reparte los *anchors* en niveles de distintas escalas en una red convolucional para directamente pasar a predecir la clase y transformación de cada uno.

Dentro de las desventajas que viene el trabajar con *anchors* es que son sensibles a la escala de los objetos que se desea detectar, por ejemplo si se usan *anchors* muy pequeños en una imagen con objetos grandes va a ser muy difícil detectar algo, por lo que la definición del tamaño de los *anchors* afecta directamente al rendimiento de la red. Otro problema es que aumenta la cantidad de hiper-parámetros de la red, que serían la cantidad y tamaño de *anchors*.

2.5.4. Redes *anchor-free*

Como ahora ya no se tienen ROI definidas, la red debe ser capaz de encontrarlas. Una forma es con un método basado en *keypoints*, donde se pueden definir distintos puntos de importancia en la imagen como hiper-parámetro, o incluso la red puede aprender a generar estos puntos. A partir de los *keypoints* se generan los *bboxes*. Un ejemplo de red que hace esto es CornerNet [18], donde cada *bbox* se define como dos *keypoints*. Otra forma de encontrar objetos es con un método basado en puntos o partes centrales del objeto, donde se toma un punto y se calculan las distancias a los bordes del *bbox*. Un ejemplo es la primera versión de YOLO [20], donde divide la imagen en una grilla de SxS, y la celda que contiene el centro del objeto es responsable de detectarlo.

Este tipo de redes elimina por completo todo hiper-parámetro relacionado a los *anchors*, por lo que se evita el problema de las escalas dado por definir esos hiper-parámetros.

Si bien los resultados del estado del arte en detección lo tienen redes *anchor-based*, existen redes *anchor-free* que se acercan bastante. En ATSS [21], que es un trabajo que compara ambos tipos de redes, llegan a la conclusión de que la diferencia de resultados se da por el cómo se definen las muestras positivas y negativas de entrenamiento. En la Figura 2.11 se realiza una comparación de RetinaNet [2] (*anchor-based*) con FCOS [4] (*anchor-free*), con respecto a como calculan las muestras positivas (marcadas con 1) y las muestras negativas (marcadas con 0). En el caso de RetinaNet se calcula el puntaje IOU entre los anchors y el *ground truth*, y dado un umbral se asigna como muestra positiva o negativa. En el caso de FCOS se toma cada punto del *feature map* y ve cuales se encuentran dentro del *ground truth*, para luego marcar como muestra positiva si el *bbox* tiene la escala que busca el nivel de la pirámide.

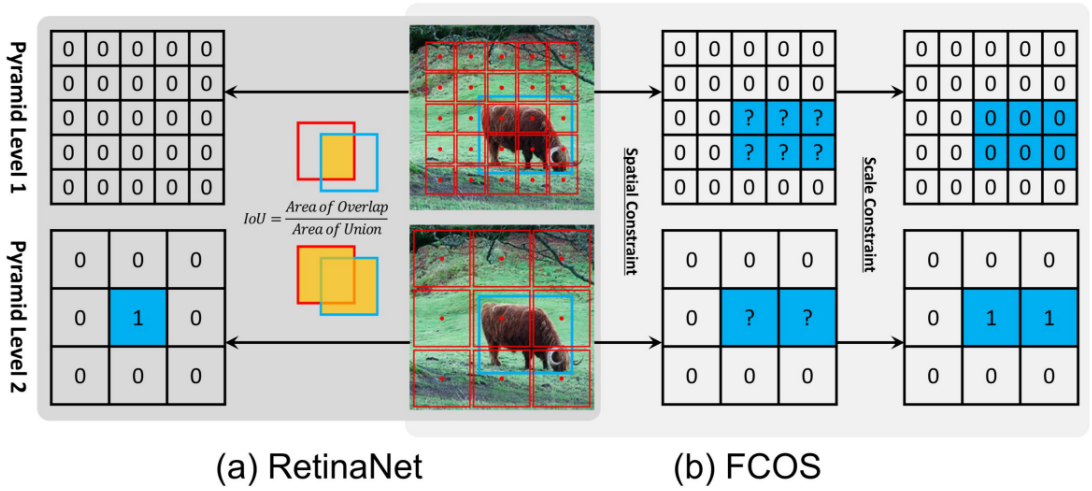

Figura 2.11: Calculo de muestras positivas y negativas en RetinaNet y FCOS, Fuente: [21]

2.6. Ejemplos de Redes *anchor-free*

A continuación se presentarán distintas redes para detección de objetos que no utilizan *anchors*:

2.6.1. CornerNet

CornerNet [18] propone detectar objetos como la búsqueda de pares de esquinas, en específico la esquina superior/izquierda y la esquina inferior/derecha. Como se puede ver en la Figura 2.12, se parte por pasar la imagen por una red convolucional, donde esta red tiene como salida dos módulos, uno con esquinas superior/izquierda y otro con esquinas inferior/-derecha, para luego caracterizarlos y unir ambos canales en las *bounding boxes* que rodean a los objetos detectados.

La arquitectura completa se puede ver en la Figura 2.13, donde la red usa como *backbone* una red Hourglass para extraer características, y después pasa a dos módulos de predicción, uno para cada esquina. Los módulos parten por realizar la operación de *corner pooling*.

Descrito en la Figura 2.14, *corner pooling* separa el input en dos mapas de características y las analiza por separado, en el caso de esquinas superior/izquierda, parte por dejar los máximos valores desde izquierda a derecha y desde abajo hacia arriba en los respectivos mapas de características, y luego suma el resultado.

Luego de realizar la operación de *pooling*, CornerNet termina por calcular *heatmaps* que describen a que clase pertenece cada esquina, *embeddings* o vectores que describen a la esquina y que sirven para agrupar con las otras esquinas, y *offsets* producidos por reducir el tamaño de la imagen.

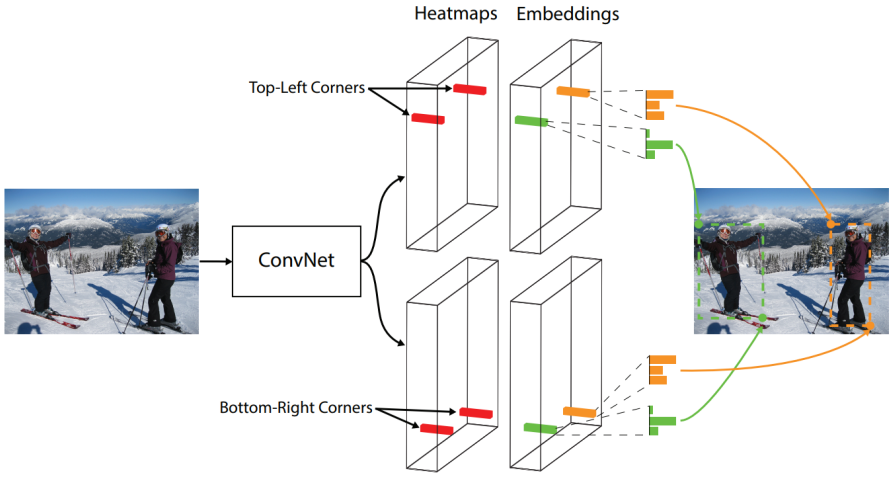

Figura 2.12: Arquitectura CornerNet, Fuente: [18]

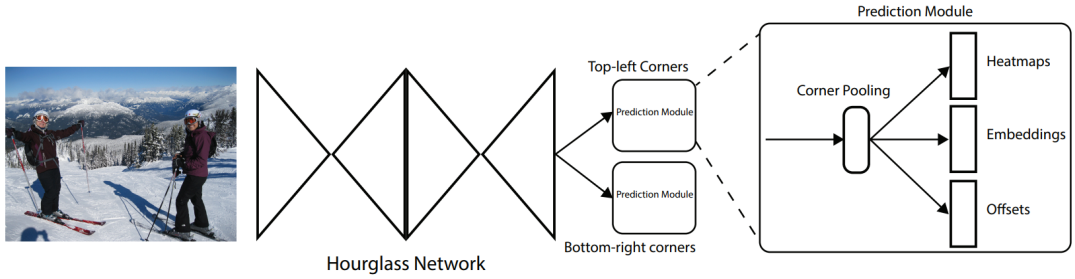

Figura 2.13: Arquitectura extendida CornerNet, Fuente: [18]

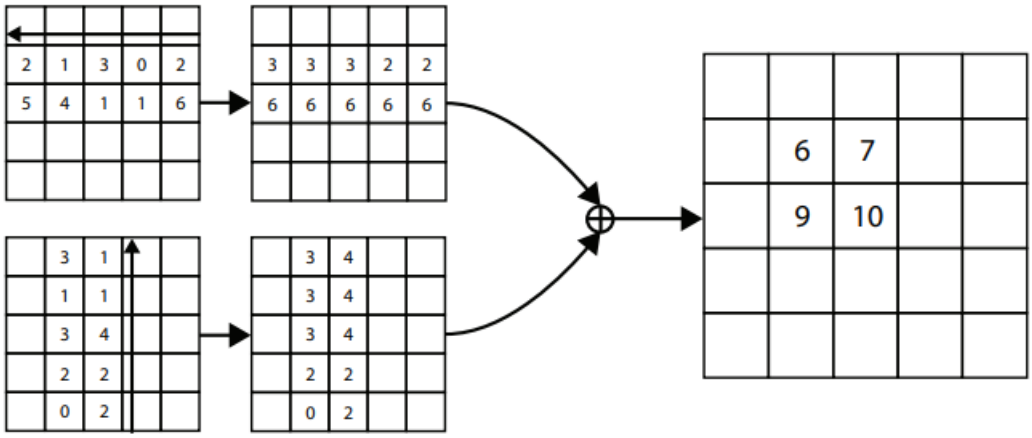

Figura 2.14: *Corner pooling*, esquina superior/izquierda, Fuente: [18]

2.6.2. Centernet

Centernet [22] se basa en CornerNet, donde busca un par de esquinas, y además agrega la búsqueda del punto central, por lo que ve a cada objeto como una tripleta de puntos. La arquitectura de la red se puede ver en la Figura 2.15, donde, como en CornerNet, se tiene

como *backbone* una red Hourglass para extraer características. Luego se toman los mapas de características obtenidos del *backbone* y se pasan a dos módulos: uno para buscar esquinas y otro para buscar centros. Para predecir esquinas utiliza *Cascade Corner Pooling*, se puede ver una descripción en la Figura 2.16.c. Este método se basa en el *Corner Pooling* de CornerNet (Figura 2.16.b) que busca máximos en un rango vertical y horizontal por separado, pero acá se parte por encontrar un límite para buscar estos máximos dentro de los rangos encontrados. Para predecir el punto central se utiliza *Center Pooling*. Como se muestra en la Figura 2.16.a, se toman los valores máximos en dirección vertical y horizontal para encontrar el centro.

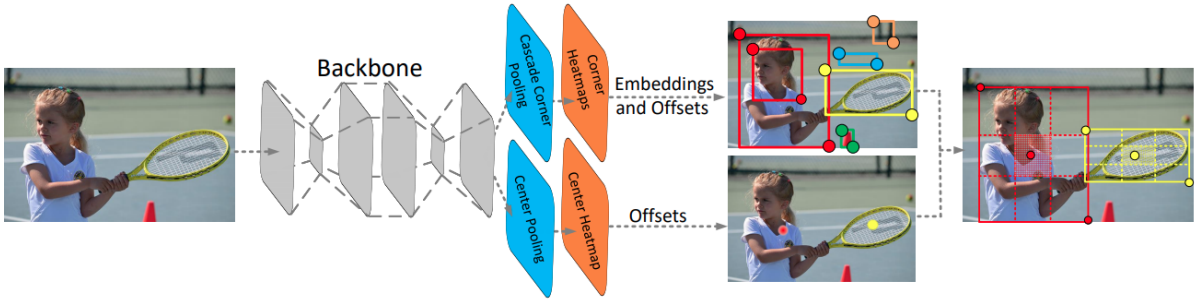

Figura 2.15: Arquitectura Centernet, Fuente: [22]

Figura 2.16: (a) Center Pooling. (b) Corner Pooling. (c) Cascade Corner Pooling., Fuente: [22]

2.6.3. FCOS

FCOS [4] es una red que hace una detección por cada píxel de la imagen, esto significa que para cada punto calcula si es que pertenece a un objeto, y si pertenece calcula las distancias desde el punto hasta los bordes de la imagen (Figura 2.17).

La arquitectura de FCOS se puede ver en la Figura 2.18. Usa como *backbone* una red ResNet, donde obtiene la información obtenida de las últimas 3 etapas convolucionales. A partir de esa información construye una FPN. Esto ayuda a la predicción de objetos de múltiples tamaños, ya que cada nivel de la pirámide se centra en buscar un tamaño específico. De cada nivel de la FPN se aplica una serie de convoluciones, donde los pesos se comparten para todos los niveles, y se obtienen 3 outputs: un *heatmap* con las clases en cada punto, un valor que indica la distancia desde el punto al centro del objeto, y la regresión del punto, que son las distancias hacia los bordes del objeto.

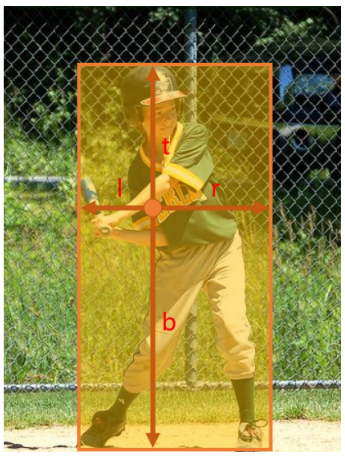

Figura 2.17: Detección con FCOS, Fuente: [4]

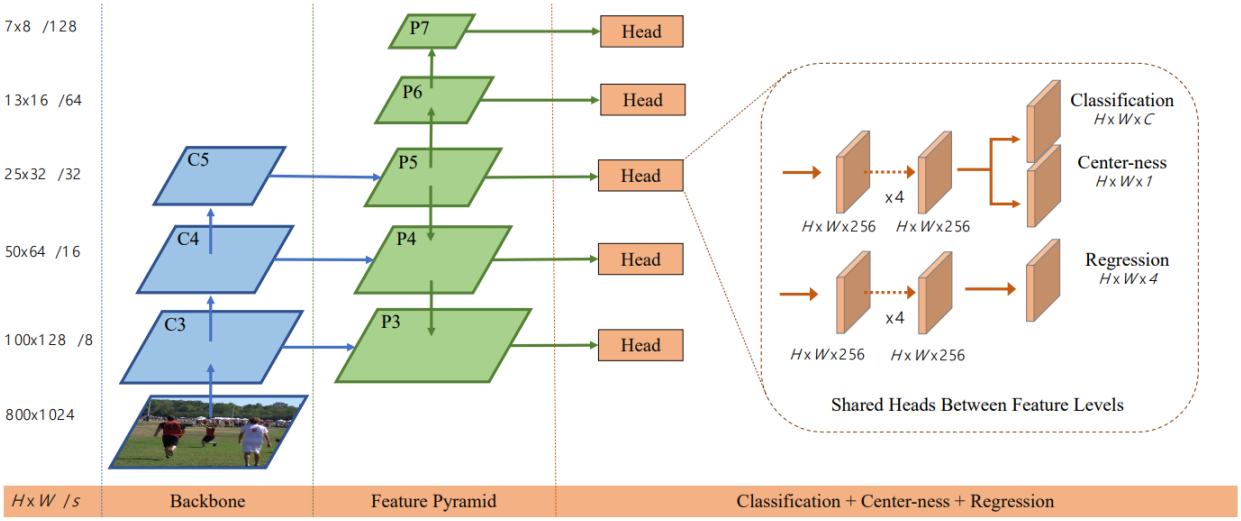

Figura 2.18: Arquitectura FCOS, Fuente: [4]

2.7. NonLocal Networks

El trabajo propuesto en NonLocal Networks [23] surge de las operaciones de *NonLocal means*, que es un algoritmo que realiza un filtro de la imagen tomando el promedio de todos los puntos de la imagen. En este trabajo se propone un bloque que toma cada punto como una suma ponderada del resto de los puntos. La estructura del bloque es como en la Figura 2.19, en el ejemplo que se muestra se tiene una dimensión extra que representa una dimensión temporal T, para efectos de este trabajo se omite esa dimensión. En el bloque se parte por realizar convoluciones con kernel de tamaño 1x1 para tomar una representación de la entrada, y luego realiza multiplicaciones matriciales, en estas multiplicaciones es cuando en cada punto añade la información del resto. Al final toma el resultado y lo suma con la entrada para obtener la salida.

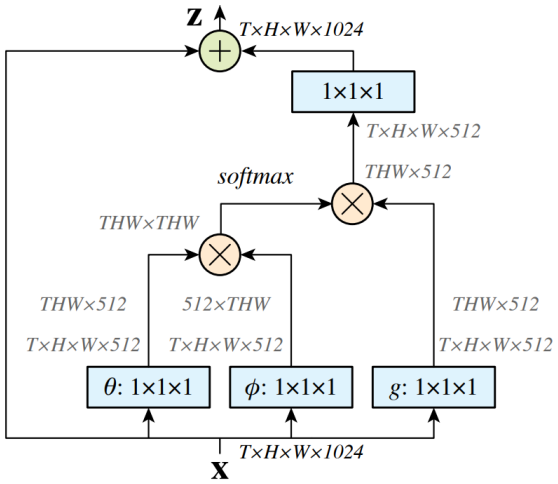

Figura 2.19: Bloque de NonLocal Features, Fuente: [23]

El Bloque de la Figura 2.19 sigue la Ecuación 2.8, que es la fórmula de autoatención propuesta en [24]. Lo que hace es aplicar pesos θ y ϕ a la entrada y multiplicarlos de forma matricial. A esto se le aplica softmax y luego el resultado se multiplica por una función g aplicada a la entrada.

$$z = softmax(x^T W_\theta^T W_\phi x) g(x) \tag{2.8}$$

2.8. Métricas

Sobre el como medir el rendimiento de cada red, se tienen distintas métricas. Primero se deben definir las métricas básicas, donde si se clasifica una muestra para ver si corresponde o no a una clase, puede caer en una de estas 4 posibilidades:

- Verdadero Positivo (TP): la muestra fue clasificada correctamente como la clase esperada.
- Verdadero Negativo (TN): la muestra no fue clasificada como la clase esperada cuando no lo era.
- Falso Positivo (FP): la muestra fue clasificada como la clase esperada cuando no lo era.
- Falso Negativo (FN): la muestra no fue clasificada como la clase esperada cuando si lo era.

De lo anterior, se calculan nuevas métricas, 2 de estas son *precision* y *recall*. *Precision* se define como la Ecuación 2.9, que mide la proporción del total de muestras clasificadas correctamente sobre el total de muestras clasificadas como la clase.

El *recall* se define como la Ecuación 2.10, donde mide la proporción de las muestras clasificadas correctamente sobre el total de muestras de la clase esperada.

Por otro lado, se tiene un valor utilizado como umbral para indicar si una detección se descarta o no durante el entrenamiento, este valor se llama "IoU" (*Intersection over Union*), donde toma el *bounding box* detectado y lo compara con el *bounding box* real de la imagen. Si la división entre el área de la intersección de esas dos *bbox* es mayor al área de su unión, entonces la muestra corresponde a un TP, y en el caso contrario, pasaría a ser un FP.

$$Precision = \frac{TP}{TP + FP} \tag{2.9}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.10}$$

Ahora se puede pasar a definir las métricas utilizadas para comparar en el paper, que son *Average Precision (AP)* y *Average Recall (AR)*. Para calcular el valor de AP, se utiliza la Ecuación 2.11. Donde se tiene una curva de *precision* con respecto al *recall* $P(r)$, esa curva se divide en R valores de *recall*, y se promedia la suma de precisiones en esos puntos. Esta métrica es comúnmente usada para medir la exactitud de la arquitectura propuesta.

$$AP = \frac{\Sigma_r P(r)}{R} \tag{2.11}$$

Dentro de los resultados se pueden ver que el valor de AP se calcula de distintas formas, donde cada una calcula de forma diferente los valores de *precision* y *recall* (más exactamente, los valores de TP y FP). En la evaluación realizada por COCO, el valor de AP final es el promedio de los valores de AP para cada categoría, lo que se comúnmente se llama mAP (*mean Average Precision*), pero en COCO no se hace diferencia entre AP y mAP. Se tienen 6 variaciones de AP, las primeras 3 son con respecto al valor del umbral "IoU", y son las siguientes:

- AP: IoU toma 10 valores desde 0.5 hasta 0.95 (sumando 0.05 entre cada valor). Se calcula el valor de AP para cada uno de esos valores y el resultado final es el promedio de esos valores
- AP^{50}: AP con umbral IoU de 0.5
- AP^{75}: AP con umbral IoU de 0.75

Las siguientes 3 son con respecto al área en píxeles del objeto detectado, y son las siguientes:

- AP^s: área menor a 32^2
- AP^m: área entre 32^2 y 96^2
- AP^l: área mayor a 96^2

Con respecto al *Average Recall*, se calcula como el área bajo la curva de *recall* con respecto al umbral IoU, donde solo se toman en cuenta los valores de IoU desde 0.5 a 1. Igual que con AP, se calcula el valor de AR para cada clase y se promedian esos valores para el valor final

de AR.

También se tienen 6 categorías de AR, las 3 primeras son equivalentes a la división por área que se hace con AP, mientras que las otras 3, el cálculo de AR se basa en el máximo AR dado un número fijo de detecciones por imagen, donde ese número puede ser 1, 10 y 100.

Capítulo 3

Desarrollo

En este capítulo de explica el desarrollo realizado para tratar el tema planteado, donde se parte por detallar los datasets utilizados, el formato de las anotaciones para entrenamiento y el diseño de las arquitecturas.

3.1. Datasets utilizados

3.1.1. COCO

COCO [1] es un dataset preparado para tareas de detección de objetos, segmentación, y detección de keypoints. El dataset consiste en alrededor de 120.000 imágenes, con un total de 80 categorías que consisten en personas, tipos de animales, comida y objetos generales. El dataset es ampliamente utilizado por diversos trabajos del área de detección y/o segmentación.

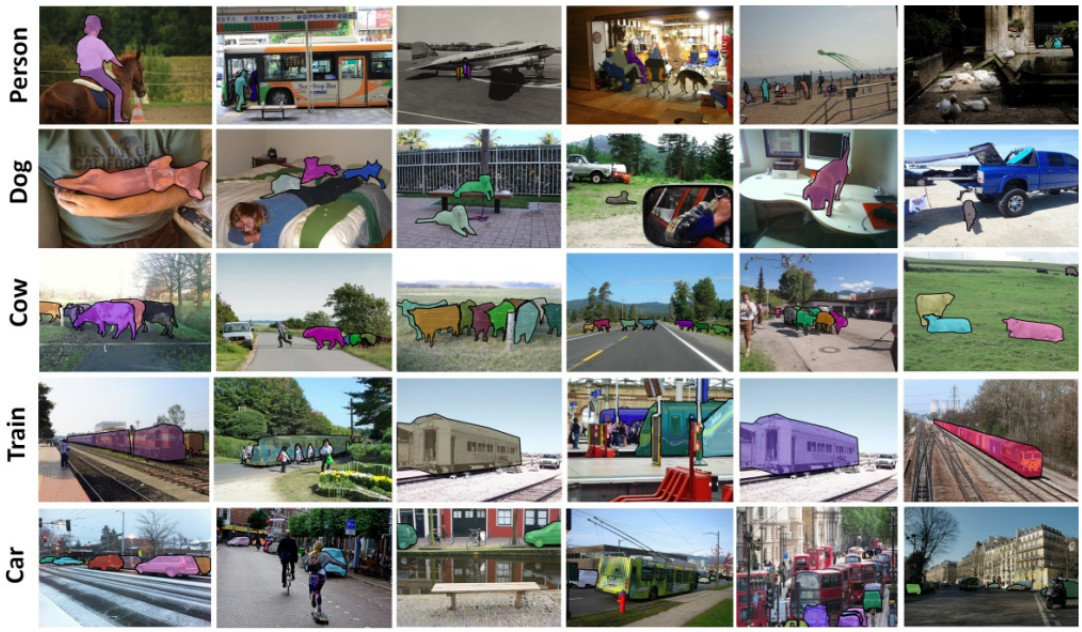

Figura 3.1: Muestra de imágenes del dataset COCO, Fuente: [1]

En la Figura 3.2 se muestra la cantidad de instancias por categoría en el dataset COCO comparado con el dataset Pascal. Si bien la mayoría de las clases tienen alrededor de 10.000 instancias, hay algunas que están poco representadas, como *hair drier*, *hair brush* y *toaster*, que tienen 1.000 o menos instancias. También existen casos de clases sobre-representadas, con alrededor de 100.000 instancias como *shoe*, *window* y *hat*.

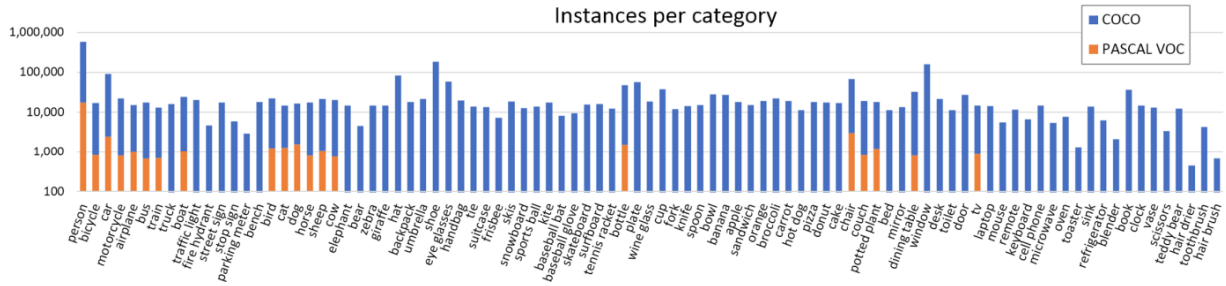

Figura 3.2: Número de instancias anotadas por clase en dataset COCO y dataset PASCAL VOC [25], Fuente: [1]

3.1.2. ModaNet

ModaNet [6] es un dataset preparado para detección de objetos, segmentación semántica y segmentación de instancias. El dataset consiste en alrededor de 50.000 imágenes, con un total de 13 categorías relacionadas con ropa.

Figura 3.3: Muestra de imágenes del dataset ModaNet, Fuente: [6]

En la Figura 3.4 se muestra la cantidad de instancias por clase en el dataset. Las clases *footwear* y *top* son las con mayor presencia en el dataset. Con el resto empieza a disminuir bastante la cantidad de instancias por cada clase, donde las menos representadas son *scarf&tie*, *headwear* y *shorts*.

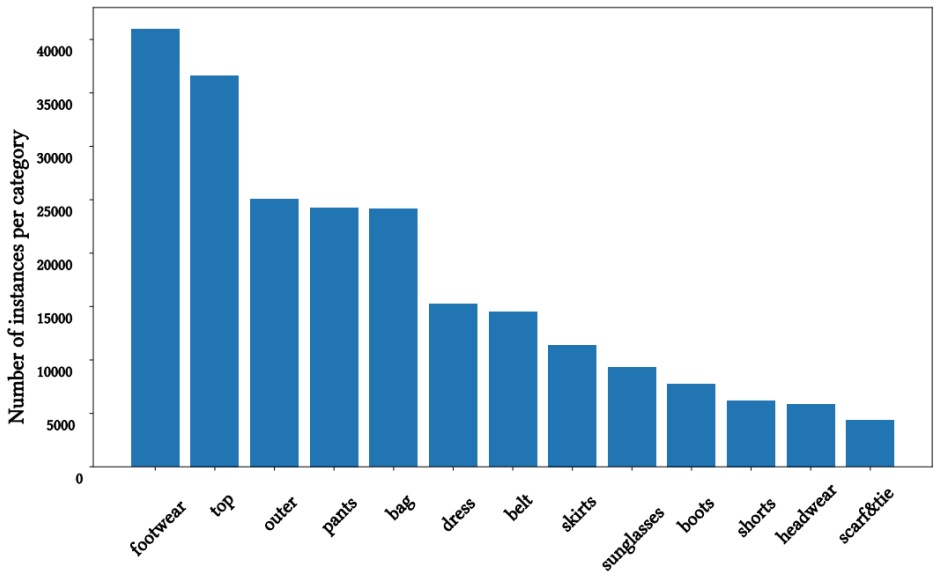

Figura 3.4: Número de instancias anotadas por clase en dataset ModaNet, Fuente: [6]

3.1.3. Montos

Adicionalmente, el profesor guía provee un dataset compuesto por montos escritos (Figura 3.5). El dataset consiste en 6370 anotaciones junto con sus imágenes. Cada anotación es un archivo de texto con la siguiente estructura:

<clase>: <x>, <y>, <w>, <h>

Donde la clase es un número entre 0 y 9, x e y son la esquina superior izquierda del *bounding box*, y w con h son el ancho y alto del *bounding box*. Para múltiples dígitos se escribe cada uno en una línea diferente, como en ejemplo mostrado en el Código 3.1

Código 3.1: Ejemplo de anotación de montos.

```
4: 83, 8, 43, 53
6: 143, 11, 39, 50
0: 204, 11, 43, 43
6: 264, 15, 43, 43
6: 329, 11, 36, 51
```

En la Figura 3.6 se muestra el balance de instancias por clase, donde en general todas las clases tienen una cantidad similar de instancias, con un conteo de entre 2.000 y 4.000, con excepción de la clase 0, que se encuentra altamente representada, con un total que bordea las 10.000 instancias.

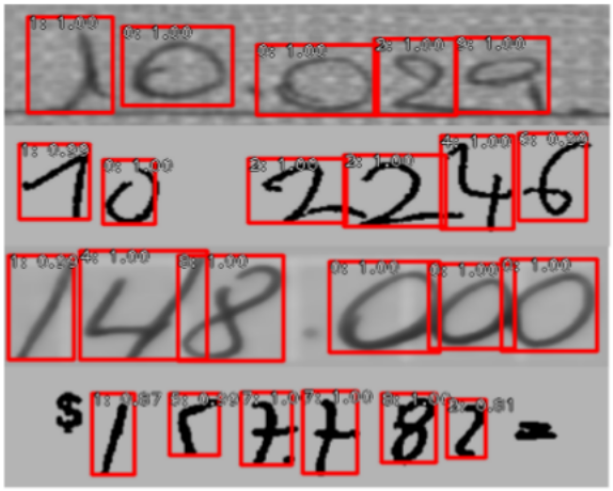

Figura 3.5: Muestra de dataset de montos.

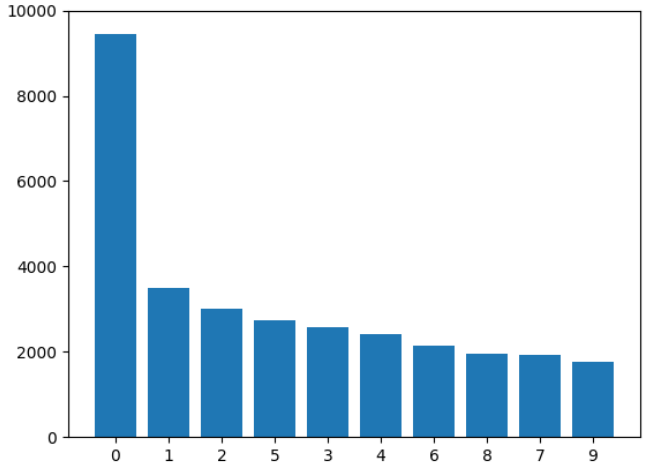

Figura 3.6: Número de instancias anotadas por clase en dataset de montos.

3.2. Baseline

Para medir los efectos de aplicar el módulo de *NonLocal features*, se parte por entrenar redes sin los módulos para tener un punto de comparación. Para esto se entrenan las redes FCOS y CornerNet con los datasets COCO y ModaNet cada uno. Con respecto al código, se utilizaron las implementaciones utilizadas por los papers de cada red, ambas disponibles en Github[1] [2].

[1] FCOS: https://github.com/tianzhi0549/FCOS
[2] CornerNet: https://github.com/princeton-vl/CornerNet

3.2.1. Preparación de datasets

Ambas redes están programadas para entrenar y mostrar resultados recibiendo datos con el formato de COCO. El formato que se espera consiste en una carpeta con todas las imágenes y un archivo tipo json con las anotaciones, La estructura del archivo para detección de objetos es la siguiente:

Código 3.2: Anotación en formato COCO para detección de objetos.

```
{
    "info": info,
    "images": [image],
    "annotations": [annotation],
    "licenses": [license],
    "categories": [category]
}
```

Código 3.3: Partes de una anotación en COCO.

```
info = {
    "year": int,
    "version": str,
    "description": str,
    "contributor": str,
    "url": str,
    "date_created": datetime,
}

image = {
    "id": int,
    "width": int,
    "height": int,
    "file_name": str,
    "license": int,
    "flickr_url": str,
    "coco_url": str,
    "date_captured": datetime,
}

license = {
    "id": int,
    "name": str,
    "url": str,
}

annotation = {
    "id": int,
    "image_id": int,
    "category_id": int,
    "segmentation": RLE or [polygon],
    "area": float,
```

```
    "bbox": [x,y,width,height],
    "iscrowd": 0 or 1,
}

category = {
    "id": int,
    "name": str,
    "supercategory": str,
}
```

En el caso de COCO, el dataset ya viene con este formato, pero en el caso de ModaNet se tiene una carpeta con imágenes y otra carpeta con un archivo de anotación por imagen que consisten en las categorías y *bounding box* de cada imagen, por lo que se tuvieron que generar las anotaciones en el formato COCO para poder continuar, obviando partes poco importantes para el entrenamiento como "info" y "licenses", y el polígono perteneciente a la segmentación se dejó como el *bounding box* que rodea al objeto.

3.3. Diseño de arquitecturas

3.3.1. Módulo NonLocal

Debido a que se trabajó sobre las implementaciones originales de CornerNet y FCOS y ambas están programadas con PyTorch, se implementó un módulo NonLocal para utilizarlo en ambas redes. El módulo es el siguiente:

Código 3.4: Código de módulo NonLocal.

```
class NonLocalBlock(torch.nn.Module):
    def __init__(self, cfg, in_channels):
        super(NonLocalBlock, self).__init__()

        self.bottleneck_channels = in_channels // 2
        self.theta = nn.Conv2d(in_channels, self.bottleneck_channels, kernel_size=1, stride
    ↪ =1)
        self.phi = nn.Conv2d(in_channels, self.bottleneck_channels, kernel_size=1, stride=1)
        self.g = nn.Conv2d(in_channels, self.bottleneck_channels, kernel_size=1, stride=1)

        self.W = nn.Conv2d(self.bottleneck_channels, in_channels, kernel_size=1, stride=1)


    def forward(self, x):
        batch = x.shape[0]
        height = x.shape[2]
        width = x.shape[3]

        theta = self.theta(x)
        theta = torch.reshape(theta, (-1, self.bottleneck_channels))

```

```
        phi = self.phi(x)
        phi = torch.reshape(phi, (self.bottleneck_channels, -1))

        g = self.g(x)
        g = torch.reshape(g, (-1, self.bottleneck_channels))

        theta_phi = torch.matmul(theta, phi)
        theta_phi = F.softmax(theta_phi, dim=1)

        theta_phi_g = torch.matmul(theta_phi, g)
        theta_phi_g = torch.reshape(theta_phi_g, (batch, -1, height, width))

        w = self.W(theta_phi_g)

        z = w + x

        return z
```

En el código se crea la clase "NonLocalBlock" que representa al módulo Nonlocal. Para si inicialización recibe el número de canales del tensor de entrada, esto lo usa para crear las capas convolucionales necesarias para el bloque, que son las convoluciones de tamaño 1x1 para generar las representaciones de la entrada θ, ϕ y g, que también reducen la cantidad de canales a la mitad. También se crea la capa que realiza la convolución final W, donde se aumenta la cantidad de canales a la cantidad inicial (Figura 2.19).

Cuando pasa un tensor por el bloque, se pasa por separado a θ, ϕ y g (Ecuación 2.8), para luego convertir los tensores resultantes en matrices de dos dimensiones, una de tamaño $BxHxW$ y otra de tamaño igual a la mitad de los canales de entrada. Luego multiplica θ con ϕ, aplica softmax y vuelve a multiplicar el resultado anterior con g. Para terminar transforma el tensor de resultado en un tensor de 4 dimensiones (B, C, H, W), aplica la convolución W y le suma la entrada.

3.3.2. CornerNet + NonLocal

Luego de implementar el módulo de *NonLocal features* (Figura 2.19), se decidió que el mejor lugar de la arquitectura de CornerNet para insertarlo es reemplazar las operaciones de *corner pooling*, debido a que esta la operación de *pooling* toma información de la imagen en forma vertical y horizontal para identificar esquinas, y el módulo de *NonLocal features* utiliza información de toda la imagen. La arquitectura final se encuentra en la Figura 3.7, donde la imagen pasa por el *backbone* Hourglass, y desde ahí se aplican los bloques NonLocal para encontrar las esquinas superior/izquierda e inferior/derecha. A partir de estos bloques se obtienen los *heatmaps*, *embeddings* y *offsets*.

Con respecto a la función de *loss*, se divide en 4 partes. Para los *heatmaps*, que su tarea es predecir la clase de cada objeto en la imagen, se utiliza FocalLoss [2], que es una variación del Cross Entropy Loss. FocalLoss se diseñó pensando en el problema donde hay una gran diferencia entre la cantidad de clases de objeto y *background* (relación 1:1000). La fórmula de *loss* es la Ecuación 3.1, sin embargo, en CornerNet se realiza una modificación para no

penalizar demasiado a los puntos cercanos a una esquina, esto lo hace agregando valores siguiendo una distribución normal en un radio de la esquina, en el *loss* de la Ecuación 3.2 se ve reflejado en el parámetro $(1 - y_{cij})^{\beta}$ utilizado cuando $y_{cij} \neq 1$, de esta forma, si se está lejos de una esquina el valor se vuelve 1, pero si se esta cerca entonces el valor disminuye. Las sumatorias extras se agregan debido a que se crea un *heatmap* por clase, por esto se recorre cada *heatmap* de cada clase para calcular su *loss*.

$$FocalLoss(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \begin{cases} -(1 - \hat{y_i})^{\gamma} log(\hat{y_i}) & y_i = 1 \\ -\hat{y_i}^{\gamma} log(1 - \hat{y_i}) & y_i \neq 1 \end{cases} \tag{3.1}$$

$$FocalLoss(y, \hat{y}) = \frac{-1}{N} \sum_{c=1}^{C} \sum_{i=1}^{H} \sum_{j=1}^{W} \begin{cases} (1 - \hat{y_{cij}})^{\gamma} log(\hat{y_{cij}}) & y_{cij} = 1 \\ (1 - y_{cij})^{\beta} (\hat{y}_{cij}^{\gamma}) log(1 - \hat{y_{cij}}) & y_{cij} \neq 1 \end{cases} \tag{3.2}$$

Para los *embeddings*, que son una caracterización de las esquinas, se aplican 2 funciones de perdida: una para agrupar esquinas y otra para separarlas. Los valores de los *embeddings* no son lo importante, sino que las distancias entre ellos, de esta forma las funciones de *loss* puede agrupar y separar esquinas. Ambos *loss* se encuentran en las Ecuaciones 3.3 y 3.4, donde e_{t_k} es el *embedding* de la esquina superior/izquierda k, e_{b_k} es el *embedding* de la esquina inferior/derecha de la esquina k, y e_k es el promedio entre e_{t_k} y e_{b_k}. El *loss* que agrupa esquinas (L_{pull}) intenta minimizar la distancia entre cada esquina con su promedio. El *loss* que separa esquinas (L_{push}) tiene como objetivo el maximizar la distancia entre pares de esquinas, para esto compara cada par con todos los otros pares.

$$L_{pull} = \frac{1}{N} \sum_{k=1}^{N} [(e_{t_k} - e_k)^2 + (e_{b_k} - e_k)^2] \tag{3.3}$$

$$L_{push} = \frac{1}{N(N-1)} \sum_{k=1}^{N} \sum_{\substack{j=1 \\ j \neq k}}^{N} max(0, 1 - |e_k - e_j|) \tag{3.4}$$

Para los *offsets*, se utiliza el *loss* SmoothL1 para reducir la diferencia entre los *offsets* del ground truth y los *offsets* calculados por la red. Dado un *offset* o_k calculado por la red y $\hat{o_k}$ dado por el *ground truth*, se calcula el *loss* utilizando la Ecuación 3.6. El *offset* se da por la reducción del tamaño de la imagen de entrada por un *stride* n, por lo que se produce una diferencia en la imagen de salida, dada por la Ecuación 3.5.

$$\hat{o_k} = \left(\frac{x_k}{n} - \left\lfloor \frac{x_k}{n} \right\rfloor, \frac{y_k}{n} - \left\lfloor \frac{y_k}{n} \right\rfloor \right) \tag{3.5}$$

$$SmoothL1 = \frac{1}{N} \sum_{k=1}^{N} \begin{cases} 0.5 * (o_k - \hat{o_k})^2 & |o_k - \hat{o_k}| < 1 \\ |o_k - \hat{o_k}| - 0.5 & |o_k - \hat{o_k}| \geq 1 \end{cases} \tag{3.6}$$

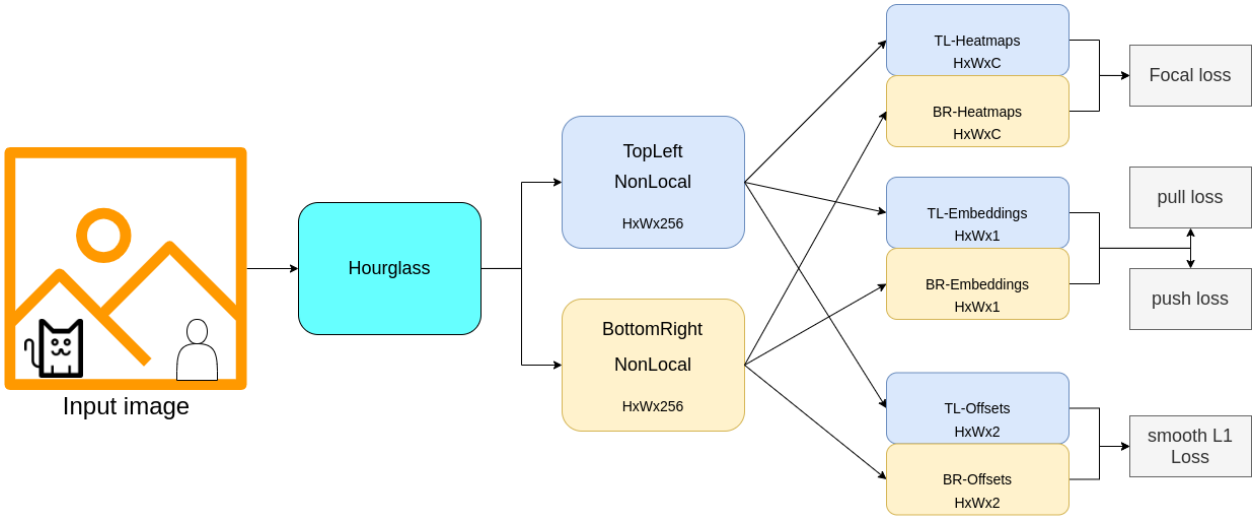

Figura 3.7: Arquitectura propuesta para CornerNet + NonLocal.

3.3.2.1. CornerNet + reducción de tamaño + NonLocal

Se realizó una variación de la arquitectura presentada anteriormente, donde se realiza una reducción del tamaño de la entrada de cada bloque NonLocal. En el Código 3.5 se muestra la modificación realizada al bloque, donde se agrega una capa convolucional con un *stride* de 2, y en la función *forward* se comienza por realizar la reducción, luego continúa utilizando el tensor de tamaño reducido para realizar la operación del bloque.

Código 3.5: Reducción a la entrada de módulo NonLocal.

```
class NonLocalBlock(torch.nn.Module):
    def __init__(self, cfg, in_channels):
        super(NonLocalBlock, self).__init__()
        self.cnv = nn.Conv2d(in_channels, in_channels, kernel_size=1, stride=2)
        self.bottleneck_channels = in_channels // 2
        ...

    def forward(self, x):
        down = self.cnv(x)

        batch = down.shape[0]
        height = down.shape[2]
        width = down.shape[3]

        theta = self.theta(down)
        ...
```

3.3.3. FCOS + NonLocal

Analizando la arquitectura de FCOS (Figura 2.18), el *backpone* + FPN se usa para extracción de características, mientras que los *headers* realizan la detección, por lo que se agregó el

módulo de *NonLocal features* dentro de los *headers*, esto hace que también se compartan los pesos en cada nivel. La arquitectura propuesta se muestra en la Figura 3.8, donde la imagen entra al *backbone* ResNet + FPN, y de cada nivel de la FPN se extraen los *headers*. La diferencia con los *headers* anteriores (Figura 3.9) es que ahora se agrega el bloque NonLocal al inicio del *header*.

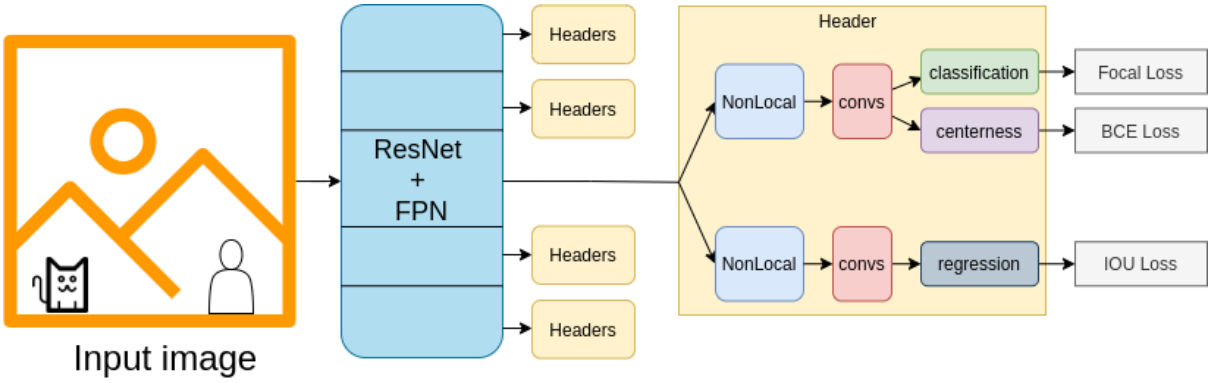

Figura 3.8: Estructura propuesta de FCOS + bloque NonLocal.

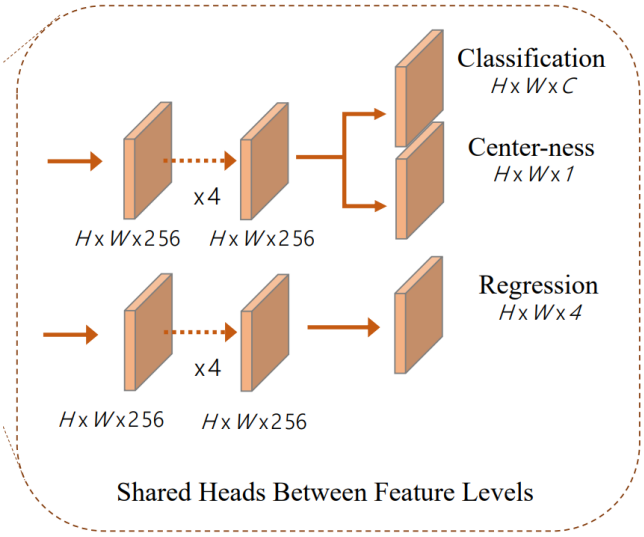

Figura 3.9: *Headers* FCOS, Fuente: [4].

Con respecto al calculo del *loss*, se divide en 3 partes:

Para la clasificación se utiliza FocalLoss debido a la gran cantidad de muestras de *background*. La clasificación retorna un vector p de tamaño C (cantidad de clases), dado el ground truth c^*, el *loss* de clasificación se calcula con la Ecuación 3.7, donde N_{pos} es la cantidad de muestras positivas.

$$L_{cls} = \frac{1}{N_{pos}} \sum_{x,y} FocalLoss(p_{x,y}, c^*_{x,y}) \tag{3.7}$$

Para *centerness*, que es un valor que va entre 0 y 1, se utiliza Binary Cross Entropy *loss*.

El valor *ground truth* de *centerness* se calcula utilizando los valores *ground truth* l, t, r, b, utilizando la Ecuación 3.8

$$centerness = \sqrt{\frac{min(l,r)}{max(l,r)} * \frac{min(t,b)}{max(t,b)}} \tag{3.8}$$

Para la regresión, que corresponde a un *bounding box*, se utiliza IOU *loss*. Más específicamente, se utiliza una versión llamada Generalized IOU [26]. La métrica GIOU propuesta se calcula con la Ecuación 3.9, donde b y $\hat{b}$ son los *bounding box ground truth* y predicho, y C es el *bounding box* que rodea a ambos *bboxes* (Figura 3.10). Finalmente, el *loss* GIOU se calcula con la Ecuación 3.10, con el objetivo de maximizar la métrica GIOU.

$$GIOU(b, \hat{b}) = IOU(b, \hat{b}) - \frac{area(C) - union(b, \hat{b})}{area(C)} \tag{3.9}$$

$$L_{GIOU}(b, \hat{b}) = 1 - GIOU(b, \hat{b}) \tag{3.10}$$

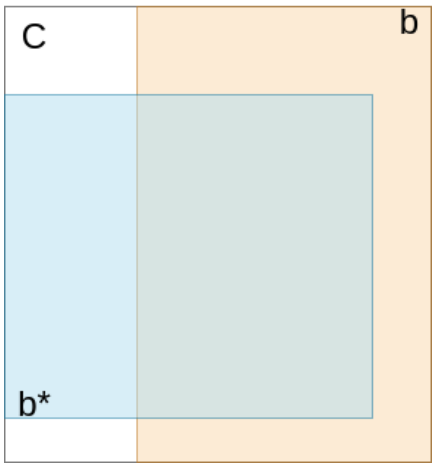

Figura 3.10: *bbox* C encerrando a el *bbox ground truth* y *bbox* predicho por la red.

Se plantean 3 experimentos según la ubicación del módulo NonLocal en los *headers*. La estructura de un *header* FCOS (Figura 3.9) toma la entrada de un nivel de la FPN y la divide en dos ramas: la rama de clasificación y la rama de regresión. Los experimentos planteados consisten en ubicar el módulo de *NonLocal features* al inicio de cada rama: un experimento con el módulo solo al inicio de la rama de clasificación (Figura 3.11), otro solo al inicio de la rama de regresión (Figura 3.12) y otro al inicio de ambas ramas (Figura 3.13).

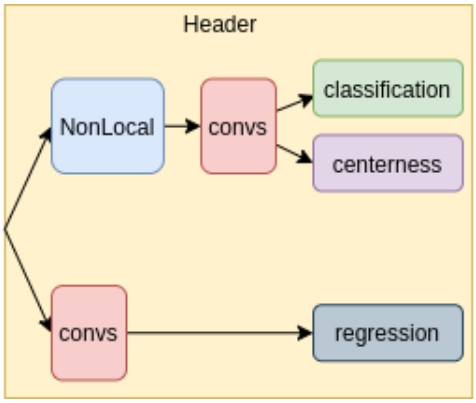

Figura 3.11: *Header* FCOS con bloque NonLocal al inicio de la rama de clasificación.

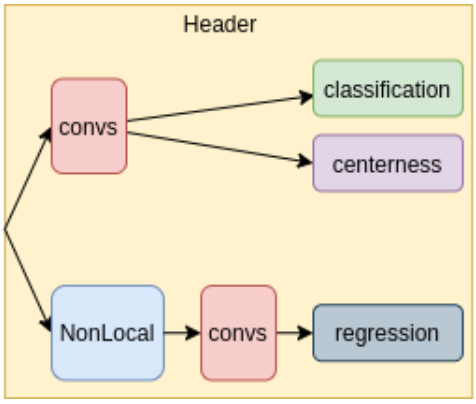

Figura 3.12: *Header* FCOS con bloque NonLocal al inicio de la rama de regresión.

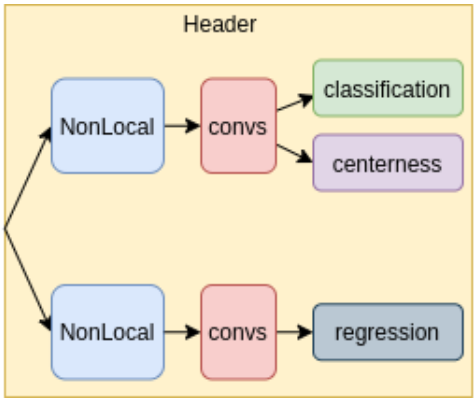

Figura 3.13: *Header* FCOS con bloque NonLocal al inicio de las ramas de clasificación y regresión.

3.3.3.1. FCOS + PANet + NonLocal

Otro experimento propuesto es el de agregar la estructura *bottom-up* de PANet a la arquitectura de FCOS, debido a que ambos consisten en una estructura similar de *backbone* de

convoluciones por etapas + FPN. La diferencia entre ambos es que FCOS agrega 2 niveles a la FPN a partir del nivel inicial P5 (Figura 2.18), mientras que PANet no lo hace en ninguna de las 2 etapas de pirámides (*top-down* y *bottom-up*). Por esto, se propone la arquitectura de la Figura 3.14, donde se generan los 2 niveles (N6 y N7) extra a partir de la última pirámide (*bottom-up*). La sección final es igual a FCOS: se toma el output de cada nivel y se aplican headers FCOS (Figura 3.9).

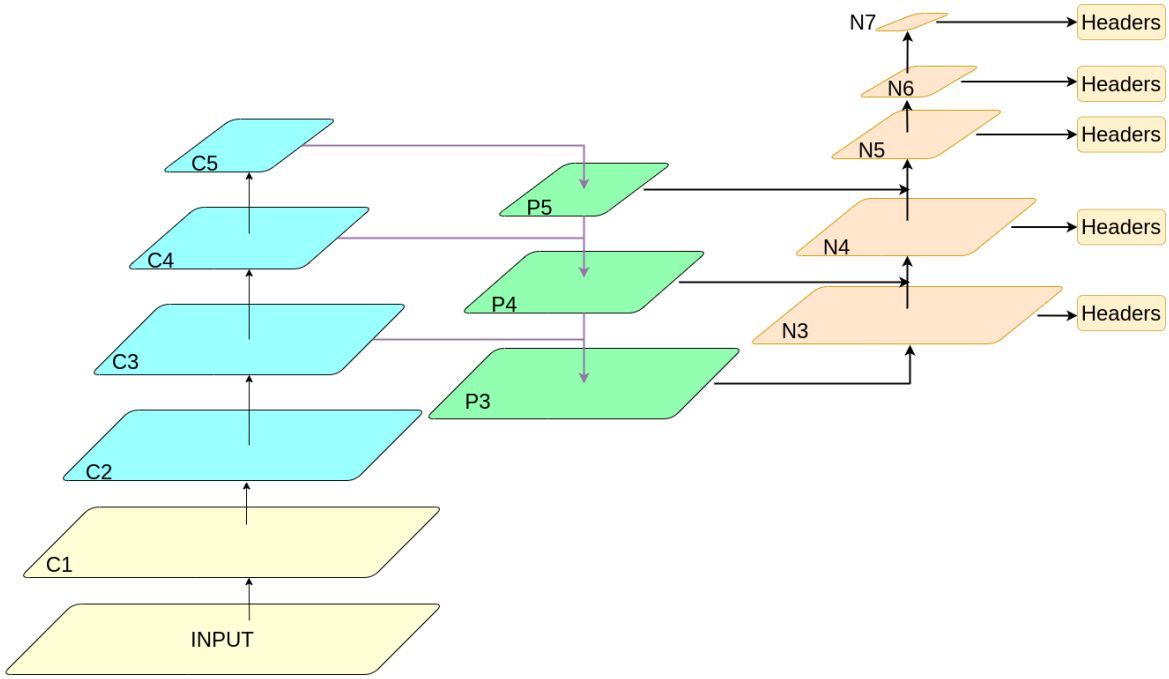

Figura 3.14: Arquitectura propuesta para usar PANet con FCOS.

3.3.4. FCOS + CornerNet

La última arquitectura propuesta se basa en FCOS y CornerNet, donde se propone tomar el *backbone* + FPN de FCOS y reemplazar los *headers* por el módulo de CornerNet que realiza la predicción (Figura 3.15), reemplazando el Corner Pooling por el bloque NonLocal. Al igual que ocurre con los *headers* FCOS, acá también se comparten los pesos por cada nivel de la FPN.

Con respecto a los experimentos, se propone el realizar entrenamientos usando todos los niveles de la FPN (Figura 3.16). El siguiente experimento es una arquitectura donde solo se utiliza el último nivel generado en la FPN (P3) para realizar la predicción (Figura 3.17). Con esta arquitectura no es necesario generar los niveles P6 y P7, ya que no se utilizarían. En otros experimentos propuestos, se limita el rango del tamaño de los objetos por cada nivel (como lo hace FCOS), y otro donde se intenta detectar todos los tamaños en todos los niveles. Por último, todos los experimentos se realizan en dos versiones: una donde se usan *NonLocal features* en los bloques TopLeft y BottomRight (Figura 3.15), y otra donde en vez de NonLocal se utiliza una convolución.

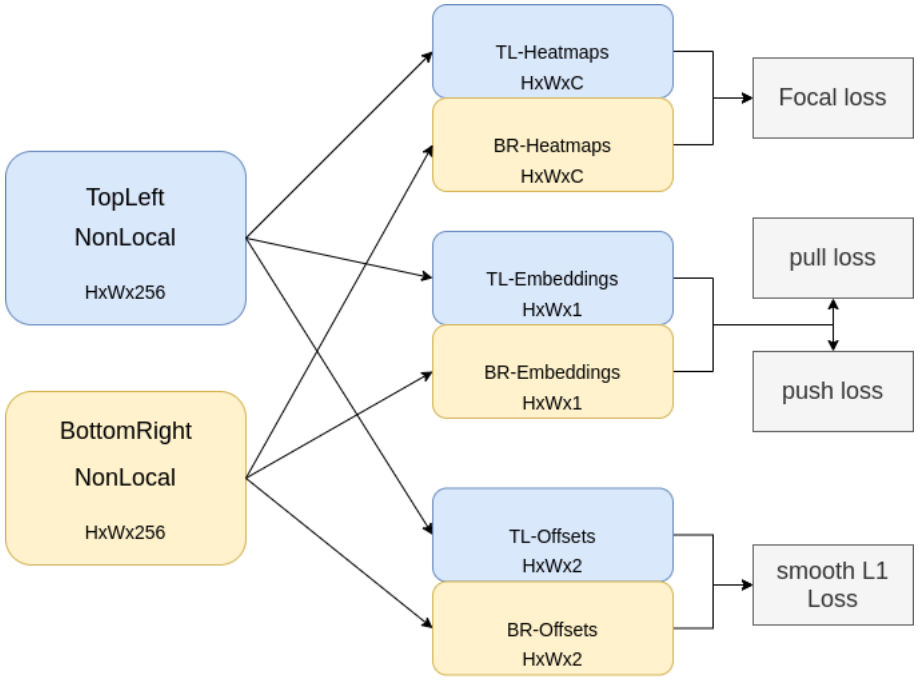

Figura 3.15: Módulo de predicción de CornerNet con módulo NonLocal.

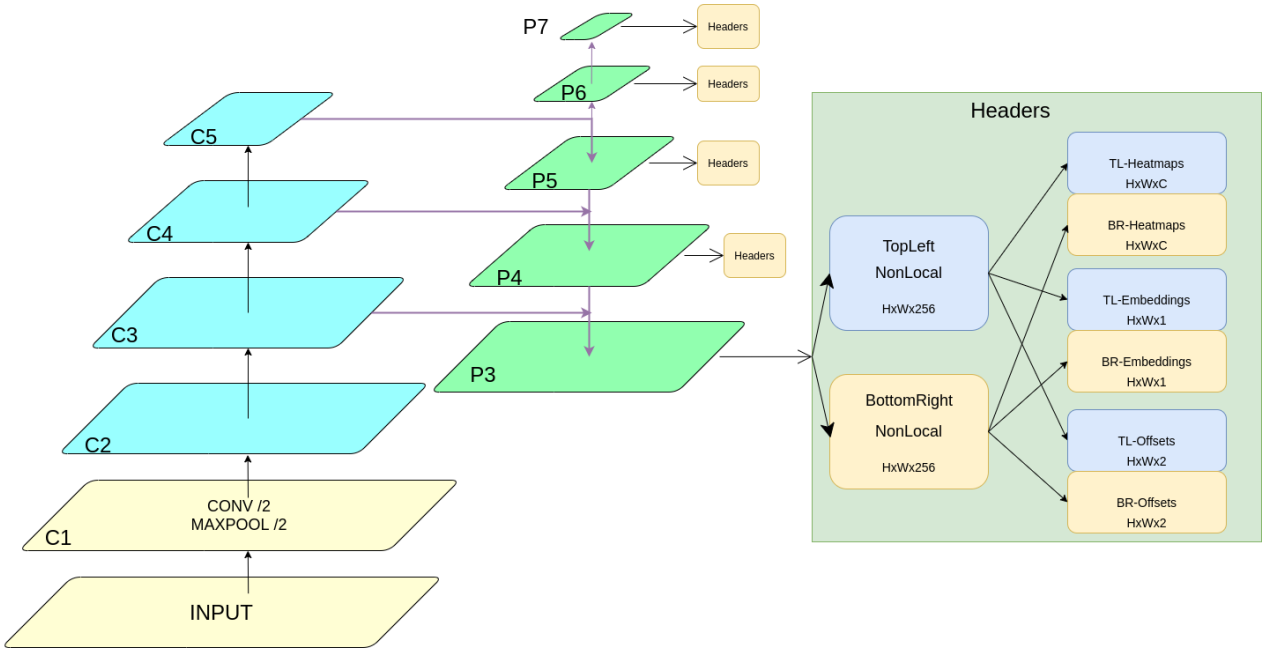

Figura 3.16: Arquitectura CornerNet + FCOS, se utilizan los *headers* en todos los niveles.

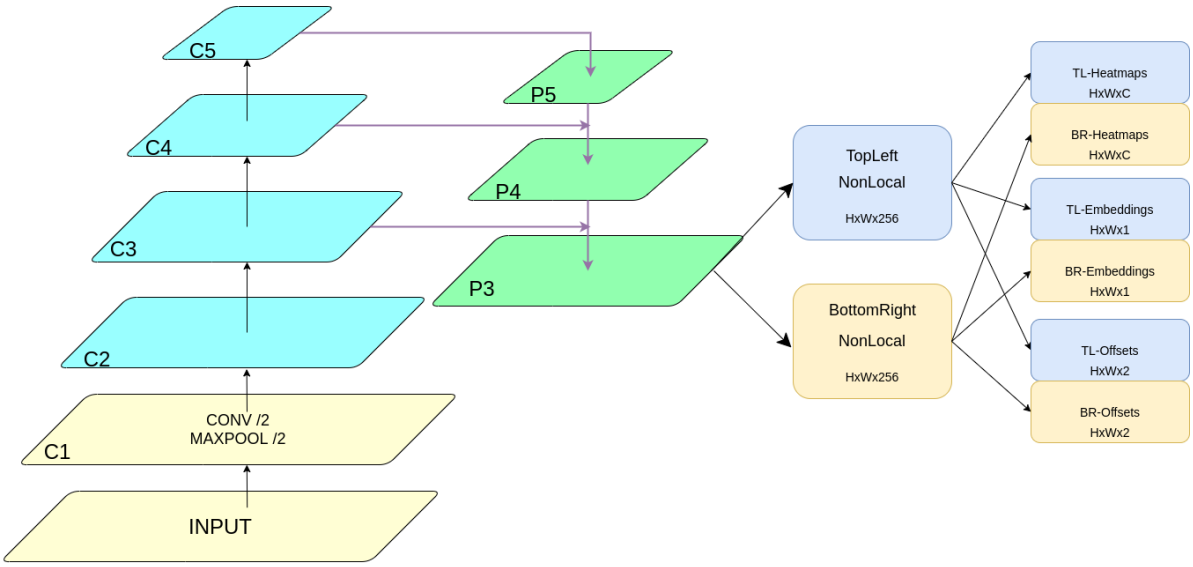

Figura 3.17: Arquitectura CornerNet + FCOS, donde se utiliza el nivel P3 para realizar la predicción.

Capítulo 4

Resultados experimentales y discusión

En este capítulo se muestran los resultados obtenidos en los experimentos utilizando las arquitecturas planteadas en el capítulo anterior, donde se parte por explicar el *hardware* y el *software* utilizados para el desarrollo, y luego se explican los experimentos junto con sus resultados.

4.1. Hardware y Software

Con respecto al servidor utilizado para el entrenamiento, en un inicio de contaba con 2 GPUs Nvidia TITAN X, con 12Gb de memoria cada una. Más adelante, se actualizaron ambas GPUs por Nvidia TITAN RTX, de 24Gb de memoria cada una.

En cuanto al software utilizado, se utilizó PyTorch, con respecto a la versión utilizada, se usaron las versiones que se instalan en el ambiente generado por CornerNet y FCOS, en el caso de CornerNet, se utiliza PyTorch 0.4, mientras que para FCOS se usa PyTorch 1.4.

4.2. FCOS + NonLocal

Tabla 4.1: Resultados al agregar modulo NonLocalFeatures, dataset COCO.

Red	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l	AR^1	AR^{10}	AR^{100}	AR^s	AR^m	AR^l
FCOS	38.5	57.1	41.3	22.5	42.1	49.3	32.1	53.3	56.9	37.3	61.9	72.1
NL-FCOS	29.1	46.4	30.8	20.5	34.8	32.3	26.1	46.6	50.3	35.2	56.8	58.9
NL-FCOS-101	33.4	50.7	35.8	17.1	37.6	46.1	29.6	48.7	51.8	28.0	58.4	71.2

En la Tabla 4.1 se tienen los resultados de entrenar FCOS usando como *backbone* una red ResNet-50. Para comparar se tienen los resultados de agregar el módulo *NonLocal features* a ambas ramas del módulo de predicción (Figura 3.9). También se agrega una comparación con FCOS con *backbone* ResNet-101. Los resultados no fueron satisfactorios, con una diferencia de al casi 10 % en los entrenamientos con ResNet-50. En este caso puede estar ocurriendo que el módulo NonLocal no es capaz de aprender a extraer correctamente la información de toda la imagen para agregarla a cada punto, y esto impacta en la detección final.

Con respecto al entrenamiento con ModaNet, los resultados son los siguientes:

Tabla 4.2: Resultados al agregar modulo NonLocalFeatures, dataset ModaNet.

Red	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l	AR^1	AR^{10}	AR^{100}	AR^s	AR^m	AR^l
FCOS	59.6	79.3	66.4	16.9	46.2	62.0	64.8	71.4	71.4	28.5	61.2	73.5
NL-FCOS-cls	62.1	82.2	68.9	17.7	50.0	62.3	66.5	74.5	74.5	30.0	68.2	74.8
NL-FCOS-reg	62.1	82.0	68.4	18.9	49.9	63.0	66.5	74.5	74.5	30.8	68.7	74.6
NL-FCOS	62.0	82.3	69.2	21.0	49.9	62.7	66.3	74.4	74.5	29.4	67.6	74.9
NL-FCOS-PANet	62.2	82.3	69.3	19.5	50.6	63.9	66.7	74.8	74.9	28.1	68.3	75.5

En la Tabla 4.2 se muestran 5 experimentos: el primero consiste en el entrenamiento de la arquitectura normal de FCOS, y el resto corresponde a diferentes pruebas con el módulo de *NonLocal features*, donde (-cls) indica que sólo se usó en la rama de clasificación y (-reg) indica que solo se usó en la rama de regresión. Los últimos experimentos usan el módulo en ambas ramas, y uno agrega la estructura de PANet a la red.

Con respecto a los resultados, se puede ver una pequeño mejora de alrededor 3 % al agregar el nuevo módulo, por lo que acá si se pudo aprender correctamente a incluir la información externa a cada punto. En cuanto a donde ubicar el módulo, en general los resultados son bastante parecidos, donde la mayor diferencia se ve en el AP de objetos pequeños, donde el usar el módulo en ambas ramas da mejores resultados.

Al agregar PANet también se ve una mejora pequeña, por lo que la arquitectura *bottom-up* puede estar ayudando a una mejor extracción de características que una FPN sola, pero en general la mejora es baja.

Los tiempos de inferencia obtenidos se muestran en la Tabla 4.3. En general los tiempos no cambian demasiado, hay una diferencia pequeña al agregar el módulo de *NonLocal features*, donde el tiempo aumenta en $2ms$. Entre los otros experimentos no hay diferencia en donde se ubique el módulo el tiempo es el mismo, esto se debe a que se parte por calcular las características con el módulo y después se escoge donde se usan. Al agregar PANet hay un incremento de $0.5ms$, que se debe al nuevo procesamiento que se debe realizar con la estructura agregada, de igual manera la diferencia es ínfima al usar o no PANet.

Tabla 4.3: Tiempos de inferencia FCOS.

Red	inference time/img
FCOS	33ms
NL-FCOS-cls	35ms
NL-FCOS-reg	35ms
NL-FCOS	35ms
NL-FCOS-PANet	35.5ms

4.3. CornerNet + NonLocal

Tabla 4.4: Resultados al entrenar CornerNet, dataset ModaNet.

Red	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l	AR^1	AR^{10}	AR^{100}	AR^s	AR^m	AR^l
CornerNet	66.0	81.3	71.5	21.4	50.6	66.8	71.2	80.3	80.8	43.7	74.0	81.9
NL-CornerNet	28.7	43.8	29.9	6.2	18.6	26.8	46.8	55.3	56.0	19.0	45.7	54.8
NL-CornerNet*	28.4	46.5	28.8	5.1	18.2	26.9	44.0	52.4	52.9	17.3	42.7	52.1

En la Tabla 4.4 se encuentran 3 experimentos: CornerNet sin modificación, entrenamiento de CornerNet con módulo NonLocal usando batch tamaño 1, y CornerNet con batch tamaño 4 (el mayor que se pudo usar con la memoria disponible, marcado con *). Como se puede ver, los resultados no fueron satisfactorios, debido a que al agregar el módulo de *NonLocal features* reemplazando las operaciones de *corner pooling* disminuye bastante el rendimiento. Se piensa que un factor importante es el tamaño del batch utilizado, ya que para entrenar la versión sin NonLocal se usó batch 16, mientras que para los otros experimentos se usó 1 y 4. Mientras que en el paper de CornerNet realizan el entrenamiento con batch de 49.

4.3.1. CornerNet + reducción de tamaño + NonLocal

Al reducir el tamaño aumentó de manera considerable la cantidad de memoria disponible para entrenar, si antes el tamaño de batch era 4, ahora permite entrenar con 16, y el mayor batch posible es 29. Los resultados son los siguientes:

Tabla 4.5: Resultados al entrenar CornerNet con reducción de tamaño, dataset ModaNet.

Red	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l	AR^1	AR^{10}	AR^{100}	AR^s	AR^m	AR^l
CornerNet-64	67.2	82.9	73.5	15.6	53.0	70.1	71.7	80.7	81.6	46.5	76.2	82.8
NL-CornerNet-64	61.8	79.4	68.0	12.6	48.0	64.2	67.6	76.9	77.7	39.7	72.2	78.8
NL-CornerNet-64*	59.3	76.0	65.6	15.8	44.5	61.0	66.8	74.2	75.0	40.2	68.4	76.2

Se llamó CornerNet-64 a la red con tamaño reducido debido a que el tamaño de la entrada del módulo de *pooling* es 128x128 en la red original, y en esta versión se redujo a 64x64. Con respecto a los resultados presentados en la Tabla 4.5, con esta versión de la red se obtuvieron mejores resultados incluso con la versión sin el módulo NonLocal. Si bien ahora las versiones con el módulo son mejores que antes, siguen siendo peor en rendimiento que la versión normal, y ahora que el tamaño de batch no es el problema, esto puede deberse a otros factores de la red o a que el módulo no es suficiente para generalizar la operación de *corner pooling*.

Con respecto a los tiempos de inferencia por imagen, la Tabla 4.6 muestra los tiempos de cada red, donde se puede ver que con y sin reducción de tamaño, el utilizar el módulo de *NonLocal features* causa que la detección sea más lenta.

Tabla 4.6: Tiempos de inferencia CornerNet.

Red	inference time/img
CornerNet	185.5ms
NL-CornerNet	195.0ms
CornerNet-64	100.0ms
NL-CornerNet-64	117.5ms

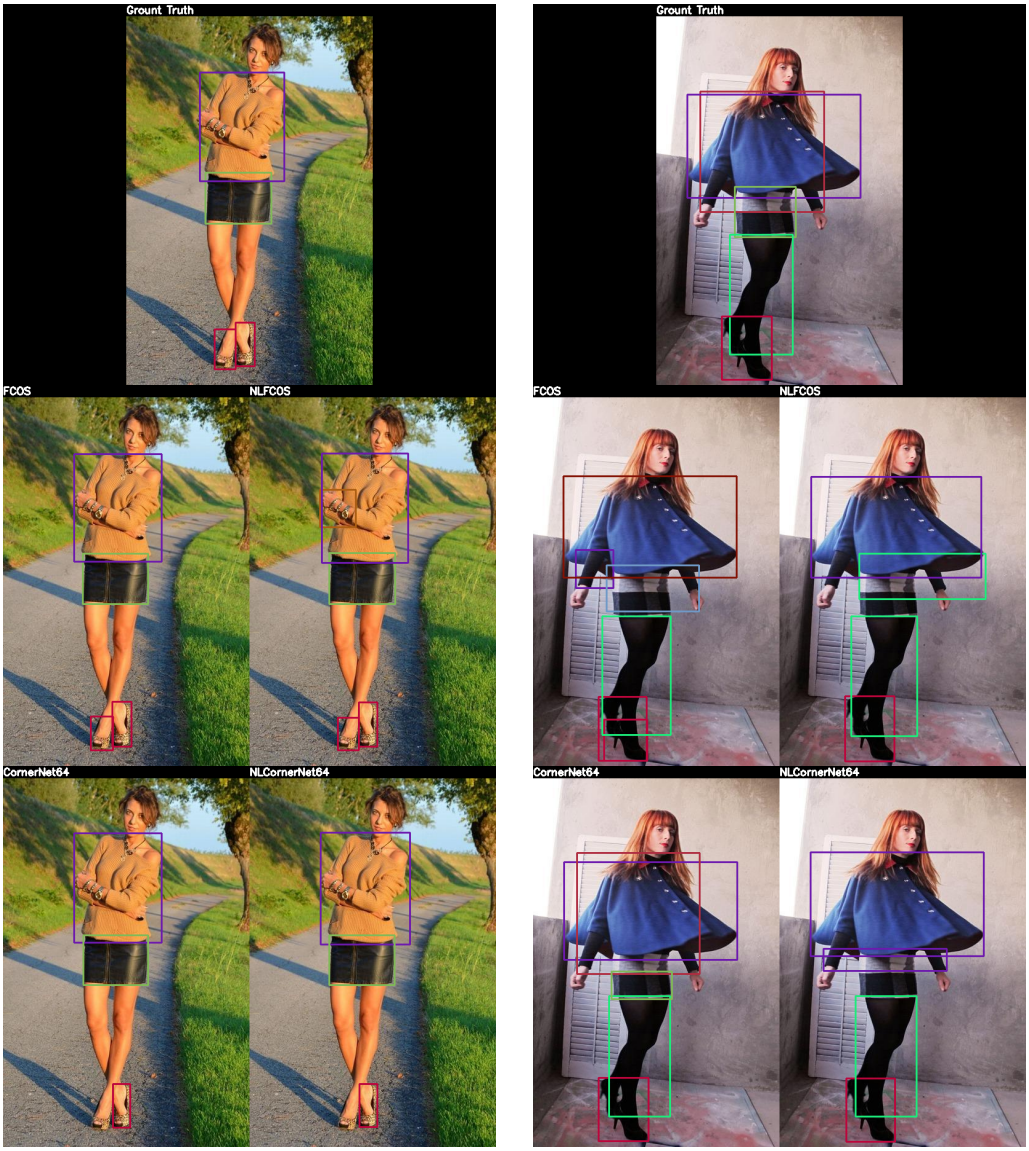

Figura 4.1: Ejemplos de comparación de detecciones por modelo, dataset ModaNet.

En la Figura 4.1 se muestran dos comparaciones de la detección realizada por los modelos FCOS y CornerNet64, ambos con y sin el módulo NonLocal. La primera imagen corresponde al *ground truth* con sus respectivas anotaciones. Más abajo se muestran las detecciones realizadas. En la imagen de la izquierda los 4 modelos tuvieron buenas detecciones, donde ambos CornerNet64 fallaron en detectar una de las botas, pero en general se detecto todo

bien. En la imagen de la derecha hubieron más problemas, donde solo CornerNet64 detecto todo bien. Los otros modelos tuvieron problemas para detectar el abrigo que consiste de dos prendas, donde FCOS y NLFCOS detectaron solo una de las 2, donde NLFCOS detectó la prenda correcta. Además FCOS fue el único modelo en separar las botas en dos detecciones cuando solo era una.

4.4. FCOS + CornerNet

Tabla 4.7: Resultados al entrenar CornerFCOS con y sin *NonLocal features*, dataset Modanet.

CornerFCOS	targets	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l	AR^1	AR^{10}	AR^{100}	AR^s	AR^m	AR^l
NL-single	-	52.2	71.2	58.2	10.7	39.4	51.3	59.7	69.5	69.8	31.0	62.1	69.1
single	-	47.7	66.6	53.0	9.5	35.6	45.0	55.7	65.3	65.6	22.3	56.5	62.7
NL-multi	normal	43.3	68.4	46.4	13.1	34.0	40.7	53.6	63.2	64.2	27.7	57.2	62.7
multi	normal	43.3	68.2	46.5	9.4	33.2	42.2	53.7	63.1	63.8	25.9	55.2	63.6
NL-multi	all	19.9	36.3	18.3	4.1	14.7	21.4	24.5	62.0	62.9	19.6	54.8	64.9
multi	all	19.4	36.3	17.5	3.4	14.1	21.1	23.6	61.3	62.1	17.9	52.1	64.3

En la Tabla 4.7 se muestran los distintos experimentos realizados con la combinación de FCOS + CornerNet (CornerFCOS para abreviar). Los experimentos se dividen en *single* y *multi*, donde *single* indica que la arquitectura solo utiliza las características de un nivel de la FPN (P3), y *multi* indica que se usaron los 5 niveles, con los pesos compartidos en los *headers*. Con respecto a los *targets*, se indica con *all* si no se limita el tamaño de objeto por nivel de la pirámide (en *single* no aplica), el caso normal es que cada nivel se preocupa por rango de tamaños de objetos. Con respecto a los resultados, los mejores resultados se tienen utilizando solo un nivel de la FPN, donde se tienen mejores resultados aplicando el módulo de *NonLocal features*. Una causa de esto puede ser que al utilizar múltiples niveles también aumenta la cantidad de muestras negativas, ya que al limitar el rango de tamaño por nivel quiere decir que si hay un objeto en un nivel, no hay nada en los otros, y la detección de esquinas puede ser sensible a esto. Por el otro lado, cuando se intentan detectar todos los objetos en todos los niveles, cada nivel puede estar intentando aprender distintos tamaños y se pierde el sentido de la pirámide, además de estar sobrecargando cada nivel, lo que puede afectar de forma negativa a la detección.

Tabla 4.8: Tiempos de inferencia CornerFCOS.

Red	inference time/img
NL-single	35.0ms
single	34.0ms
NL-multi	36.0ms
multi	35.0ms

Con respecto a los tiempos de inferencia indicados en la Tabla 4.8, el efecto de agregar el módulo NonLocal afecta en $1ms$ al tiempo, lo mismo ocurre al pasar de estructura *single* a

multi. En general casi no hay diferencia entre los tiempos de cada uno.

4.5. Montos

Finalmente, se realizaron experimentos con el dataset de montos. Para esto se utilizaron las redes FCOS con y sin módulo de NonLocal, y CornerNet con el módulo de NonLocal. Debido a que no alcanzó el tiempo no se pudo realizar el entrenamiento con la arquitectura CornerNet sin el módulo NonLocal.

Tabla 4.9: Resultados al entrenar diferentes arquitecturas con el dataset de montos.

Red	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l	AR^1	AR^{10}	AR^{100}	AR^s	AR^m	AR^l
NLFCOS	39.0	56.8	47.6	61.4	29.4	-1	33.6	48.7	50.3	66.8	43.9	-1
FCOS	38.7	56.2	47.2	61.3	29.1	-1	33.3	48.2	49.8	67.0	43.2	-1
NLCornerNet	52.3	77.1	64.8	47.9	56.3	-1	50.7	69.8	72.9	67.5	75.1	-1
CornerNet	69.2	97.9	87.1	66.4	71.0	-1	61.0	79.1	79.4	77.5	80.2	-1

Los resultados obtenidos se muestran en la Tabla 4.9, y se ven relativamente bajos en el contexto del problema presentado, donde se esperan resultados sobre 70 % u 80 % en AP. Esto puede deberse a un mal manejo en el traspaso del dataset a formato COCO, o también pueden haber ocurrido errores a la hora de configurar las redes. De igual manera, con los resultados obtenidos se ve una mejora en el paso de FCOS normal a FCOS con módulo NonLocal, mientras que NLCornerNet obtiene mucho mejores resultados que las otras 2 redes. Como los dígitos en las imágenes son de tamaño relativamente pequeño, no hay ninguno que cumpla la condición *large*, por lo que AP^l y AR^l tienen valor -1.

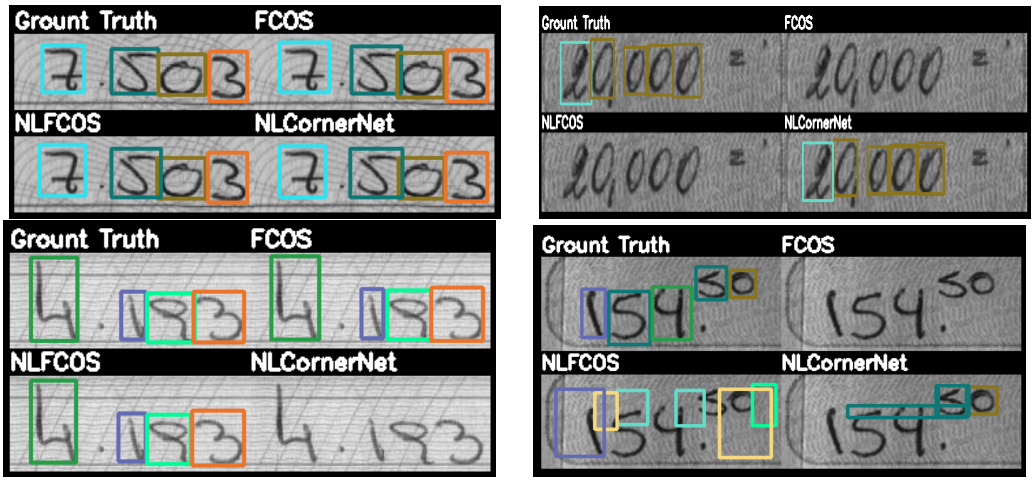

Figura 4.2: Ejemplos de comparación de detecciones por modelo, dataset de montos.

Se piensa que pudo haber un error durante la configuración debido a que hay imágenes donde no se detecto nada, cuando la detección de dígitos es una tarea simple. Como se puede ver en los ejemplos presentados en la Figura 4.2, donde hay casos que ambas todos los

métodos detectaron todos los dígitos correctamente, como también ocurre que hay casos en que hay modelos que no detectan nada.

4.6. Comparación con método *anchor-based*

En el trabajo de memoria "Detección de prendas de vestir utilizando modelos de detección de objetos basados en *deep learning*" [27] se realizaron experimentos con distintos modelos de detección de objetos. El con mejor rendimiento con el dataset ModaNet lo tuvo el modelo TridentNet [28], que es un modelo *anchor-based*. En la Tabla 4.10 se muestra la comparación de los métodos que utilizan el módulo NonLocal con los resultados obtenidos en el trabajo de memoria mencionado, donde los métodos *anchor-free* tienen un rendimiento mucho mayor que el modelo *anchor-based*.

Tabla 4.10: Comparación de modelos propuestos con modelo *anchor-based* TridentNet.

Red	AP	AP^{50}	AP^{75}
TridentNet	54.0	77.5	62.2
NLFCOS	62.0	82.3	69.2
NL-CornerNet-64	61.8	79.4	68.0

Tabla 4.11: Comparación de tiempos de inferencia entre modelos *anchor-free* y *anchor-based*.

Red	inference time/img
TridentNet	450ms
NLFCOS	35ms
NL-CornerNet-64	117.5ms

Finalmente, en el caso de ModaNet la red FCOS resultó ser más eficiente que CornerNet en cuanto a velocidad debido a que el tiempo de inferencia de NLFCOS es casi 6 veces menor que los tiempos obtenidos por CornerNet, además de tener rendimientos comparables, y ambos son buenas alternativas a métodos *anchor-based*, donde el modelo con el mejor rendimiento encontrado en [27] es mucho más lento que los modelos propuestos, siendo NLFCOS el más rápido de todos.

Capítulo 5

Conclusiones

5.1. Conclusiones

A partir de los experimentos realizados, se puede concluir que el aplicar el módulo de *NonLocal features* no siempre mejora los resultados, y que esto puede depender de distintos factores, como el dataset con el que se está entrenando. Basta con comparar los resultados de FCOS entrenado con COCO y con ModaNet: el agregar el bloque NonLocal redujo en alrededor de un 9 % el rendimiento en el caso de COCO, mientras con ModaNet se aumentaron los resultados en alrededor de un 3 %. Con CornerNet se redujo el rendimiento en el caso normal y el caso con reducción de tamaño entrenando con ModaNet.

Para el caso de la arquitectura propuesta que utiliza una parte de FCOS y una parte de CornerNet, si bien el incluir el módulo NonLocal aumenta el rendimiento, falta realizar una comparación donde se utilice CornerPooling en vez de una convolución normal. A parte de esto, el rendimiento obtenido en los experimentos en general fue bajo comparado con el rendimiento de las otras redes.

Con respecto a los tiempos de inferencia, el aplicar el módulo NonLocal aumenta el tiempo de inferencia por imagen por un pequeño margen, por lo que se concluye que el aplicar el módulo no afecta a la velocidad de detección.

Con esto se cumplen los objetivos planteados, que consiste en evaluar el impacto de aplicar el módulo de *NonLocal features* en redes *anchor-free*. En cuanto a si el impacto es positivo o negativo, es algo que no se puede concluir directamente con los resultados obtenidos debido a que hay casos en que mejora el rendimiento y otros en el que empeora. Por lo que el impacto del módulo va a depender de la arquitectura general de la red.

Dentro de los modelos propuestos, NLFCOS presenta una buena alternativa para uso en aplicaciones de ventas en internet (*e-commerce*), en el sector de venta de ropa, dado que el entrenamiento fue realizado con ModaNet, un dataset que permite diferenciar hasta 13 tipos de prendas de vestir. Se propone NLFCOS como mejor alternativa debido a su velocidad de detección, que es un factor importante en este tipo de aplicaciones, además de tener un buen rendimiento.

5.2. Trabajo futuro

Dado los resultados obtenidos en este trabajo de memoria se proponen los siguientes avances a futuro:

- **Realizar experimentos agregando el módulo NonLocal a otras arquitecturas *free-anchor***: si bien éste trabajo se centró en realizar pruebas con CornerNet y FCOS, se podrían agregar más experimentos con otras redes, como CenterNet.
- **Implementar CornerPooling en la estructura CornerNet+FCOS**: en los experimentos sin el módulo NonLocal se utilizó una convolución simple al inicio del *header* de predicción. Se propone utilizar CornerPooling en vez de una convolución. Luego se puede comparar con los resultados ya obtenidos con el módulo NonLocal.
- **Realizar experimentos con otros datasets**: como se pudo observar en los resultados, con COCO se tiene un impacto negativo al aplicar el módulo, pero con ModaNet el resultado es positivo con FCOS y negativo con CornerNet y con montos es positivo en FCOS. Por lo que se propone entrenar con más datasets para ver el impacto ocasionado.

Bibliografía

[1] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[2] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *CoRR*, vol. abs/1708.02002, 2017.

[3] A. Christiansen, "Anchor boxes — the key to quality object detection." https://towardsdatascience.com/anchor-boxes-the-key-to-quality-object-detection-ddf9d612d4f9. Accessed: 2021-09-24.

[4] Z. Tian, C. Shen, H. Chen, and T. He, "FCOS: fully convolutional one-stage object detection," *CoRR*, vol. abs/1904.01355, 2019.

[5] P. Ramachandran, N. Parmar, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens, "Stand-alone self-attention in vision models," *CoRR*, vol. abs/1906.05909, 2019.

[6] S. Zheng, F. Yang, M. H. Kiapour, and R. Piramuthu, "Modanet: A large-scale street fashion dataset with polygon annotations," *CoRR*, vol. abs/1807.01394, 2018.

[7] J. M. Alvarez, "El perceptrón como neurona artificial." http://blog.josemarianoalvarez.com/2018/06/10/el-perceptron-como-neurona-artificial/. Accessed: 2021-09-24.

[8] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 01 2012.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[10] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," *CoRR*, vol. abs/1603.06937, 2016.

[11] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016.

[12] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," *CoRR*, vol. abs/1803.01534, 2018.

[13] J. Redmon, "Yolo: Real-time object detection." https://pjreddie.com/darknet/yolo/. Accessed: 2021-09-25.

[14] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[15] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015.

[16] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[17] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[18] H. Law and J. Deng, "Cornernet: Detecting objects as paired keypoints," *CoRR*, vol. abs/1808.01244, 2018.

[19] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015.

[20] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[21] S. Zhang, C. Chi, Y. Yao, Z. Lei, and S. Z. Li, "Bridging the gap between anchor-based and anchor-free detection via adaptive training sample selection," *CoRR*, vol. abs/1912.02424, 2019.

[22] K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, and Q. Tian, "Centernet: Keypoint triplets for object detection," *CoRR*, vol. abs/1904.08189, 2019.

[23] X. Wang, R. B. Girshick, A. Gupta, and K. He, "Non-local neural networks," *CoRR*, vol. abs/1711.07971, 2017.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[25] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International Journal of Computer Vision*, vol. 111, pp. 98–136, Jan. 2015.

[26] S. H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. D. Reid, and S. Savarese, "Generalized intersection over union: A metric and A loss for bounding box regression," *CoRR*, vol. abs/1902.09630, 2019.

[27] S. S. Osses, "Detección de prendas de vestir utilizando modelos de detección de objetos basados en *deep learning*." http://repositorio.uchile.cl/handle/2250/176546.

[28] Y. Li, Y. Chen, N. Wang, and Z. Zhang, "Scale-aware trident networks for object detection," *CoRR*, vol. abs/1901.01892, 2019.