

# YOLO CON MÓDULOS DE ATENCIÓN SE – CBAM

Universidad Autónoma de Occidente

Especialización en Inteligencia Artificial

Deep Learning Avanzado

Docente: PhD. Juan Carlos Perafan

Alba Ramirez, 2216260 – Carlos Arbey Mejia, 2210549 - Andres Felipe Guerra, 2211058 – Milton Guarin, 2210702

**Abstract**— The detection of objects in images implies, not only identifying what type of object it is, also locating it within the image (obtaining the coordinates of the "box" that contains it). In other words, detection = classification + location.

YOLO (You Only Look Once) is the most popular object detection algorithm, it uses deep learning and CNN to detect objects, and it stands out from its competitors (RCNN, Faster-RCNN) because, as its name indicates, it requires “seeing” the image only once, allowing it to be the fastest of all (although it does sacrifice a bit of accuracy). This speed allows you to easily detect objects in real time in video (up to 30 FPS). [3]

In the present work, the YOLO algorithm will be used with two attention mechanisms, SE (Squeeze-and-Excitation) and CBAM (Convolutional Block Attention Module), validating performance in a customized image dataset with a single class. The results will be presented and evaluated to make a comparison of the model with the best performance in the object detection task.

**Keywords:** Deep Learning, CNN (Convolutional Neural Networks), YOLO (You only look once, SE (Squeeze-Excitation), CBAM (Convolutional Block Attention Module))

## I. Introducción

Cuando se trata de tareas de detección de objetos en visión computacional con deep learning hay tres algoritmos de detección que se utilizan principalmente:

- R-CNN y sus variantes, Fast R- CNN, and Faster R-CNN

- Single Shot Detector (SSDs)

- YOLO

Si bien las R-CNN tienden a ser muy precisas, el mayor problema con la familia de redes R-CNN es su velocidad: eran increíblemente lentas, obteniendo solo 5 FPS en una GPU.

Para ayudar a aumentar la velocidad de los detectores de objetos basados en el aprendizaje profundo, tanto los Single Shot Detector(SSD) como YOLO utilizan una estrategia de detector de una etapa.

Estos algoritmos tratan la detección de objetos como un problema de regresión, tomando una imagen de entrada dada y aprendiendo simultáneamente las coordenadas del cuadro delimitador y las probabilidades de etiqueta de clase correspondientes (ver fig.1)

En general, los detectores de una etapa tienden a ser menos precisos que los detectores de dos etapas, pero son significativamente más rápidos.

YOLO, fue presentado por primera vez en 2015 por Redmon et al., Su artículo, You Only Look Once: Unified, Real-Time Object Detection, detalla un detector de objetos capaz de detectar objetos en tiempo super real, obteniendo 45 FPS en una GPU.

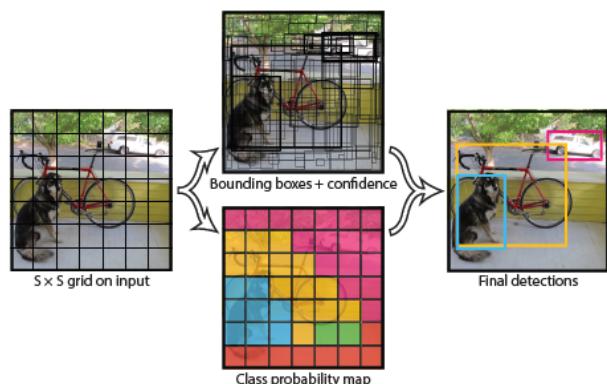


Fig.1 El modelo de YOLO

Lo primero que hace el algoritmo es procesar la imagen, pasándola a escala de grises y detectando los bordes de los objetos con un filtro.

Luego reduce la imagen a un problema de clasificación de clases y enmarca el objeto identificado en una caja. Para esto, el algoritmo trata de encerrar el objeto en varias cajas, que luego reduce al mejor candidato que ubique mejor el objeto.

## II. Implementación

### 2.1 Creación del dataset

Para la generación del dataset de imágenes personalizado se definió la clase “helmet” (cascos de motocicleta/bicicleta).

Se descargaron 245 imágenes de internet de personas utilizando cascos de motocicleta o bicicleta en diferentes escenarios.



Fig.2 Imágenes del dataset

Una vez que recopilamos las imágenes, el siguiente paso fue etiquetarlas. En el contexto de la detección de objetos, etiquetar significa dibujar cuadros delimitadores alrededor de los objetos que estamos interesados en detectar en las imágenes y asociarlos con las clases / categorías de objetos correspondientes para que podamos mostrarlo claramente a la máquina.

Para esto instalamos la herramienta LabelImg tomada del repositorio Git LabelImg [4]. Se utilizó la aplicación Anaconda para realizar una instalación de entorno Python, a este entorno se le instalo lo siguiente para la correcta utilización del programa LabelImg:

```
conda install pyqt=5
conda install -c anaconda lxml
pyrcc5 -o libs/resources.py resources.qrc
```

Fig.3 Comandos para instalación de Conda

Para etiquetar los objetos en cada imagen, simplemente cargas todas las imágenes, dibujar el recuadro sobre el

objeto de clase que quieras detectar, seleccionas la clase y se guarda la imagen cuando termines todos los recuadros de los objetos (ver fig.4). El proceso se repite para cada una de las imágenes.

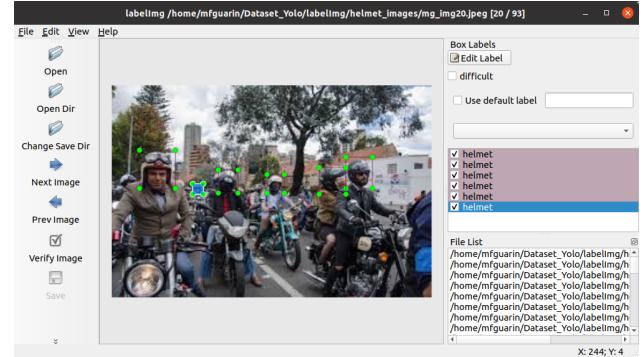


Fig.4 Interfaz gráfica de LabelImg

Cuando guarda las etiquetas después de cada imagen, LabelImg crea un archivo de texto para cada imagen con el mismo nombre que la imagen directamente en formato YOLO.

Cada archivo txt, tendrá los siguientes datos:

object-id center\_x center\_y width height

- object-id representa el número que corresponde a la clase. Como solo tenemos una clase, ese número siempre será 0.

- center\_x y center\_y representan el punto central del cuadro delimitador. Pero se normalizan para que oscilan entre 0 y 1 dividiendo por el ancho y el alto de la imagen.

- width height representa el ancho y alto del cuadro delimitador. Nuevamente normalizado al rango de 0 a 1 dividido por el ancho y alto original de la imagen.

El archivo de texto de anotación generado contendrá cada línea como la anterior para cada límite en la imagen y un archivo de texto para cada imagen [5].

```
0 0.431310 0.211286 0.073482 0.149606
0 0.563898 0.202100 0.054313 0.131234
0 0.720447 0.246719 0.047923 0.078740
0 0.649361 0.317585 0.059105 0.078740
0 0.816294 0.282152 0.038339 0.081365
0 0.037540 0.162730 0.068690 0.241470
```

Fig.5 Estructura de archivo txt formato YOLO

Una vez creado el dataset, procedemos con la implementación de YOLO V3 y sus respectivas variantes con los módulos de atención SE y CBAM.

## 2.1 Implementación modelos YOLOV3

Para llevar la implementación y el entrenamiento de YOLOv3 con nuestra clase helmet se creó un cuaderno en Google Colab **YOLO\_tiny\_se.ipynb**, el cual presenta los llamados necesarios a la librería de YOLOv3 tomado del repositorio GitHub PyTorch-YOLOv3.

Nos basaremos en la arquitectura Tiny YOLOV3, que es una versión simplificada de YOLOv3. Es un algoritmo de detección en tiempo real desarrollado para dispositivos integrados con capacidades de procesamiento de datos deficientes. La estructura del modelo es simple y es actualmente el algoritmo de detección de objetos más rápido, pero la precisión de detección es baja, especialmente en detección de objetos pequeños.

Tiny YOLOV3 redujo la red de detección de funciones YOLOv3 darknet-53 a 7 capas de convolución tradicional y una capa de agrupación máxima de 6 capas, utilizando una predicción de dos escalas  $13 \times 13$ ,  $26 \times 26$  red para predecir el objetivo. La estructura de la red se muestra en la Figura 5.

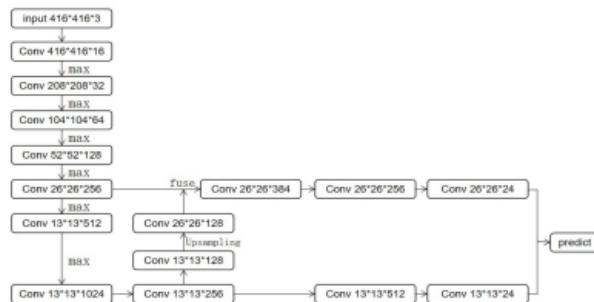


Fig.6 Arquitectura Tiny YOLOV3

El dataset con 245 imágenes se dividió 80% para entrenamiento (196 imágenes) y 20% para pruebas (49 imágenes)

```
1 #@title **Se genera el dataset de entrenamiento y de test**
2 Directorio = '/content/drive/MyDrive/YOLO_CUSTOM_DATABASE'
3 train = 0.8
4 test = 0.2
```

Clonamos el repositorio:

<https://github.com/promach/PyTorch-YOLOv3.git>

Instalamos los requerimientos necesarios en Google Colab:

```
!pip3 install -r requirements.txt
```

```
numpy
torch>=1.0
torchvision
matplotlib
tensorflow
tensorboard
terminaltables
pillow
tqdm
```

Para realizar el entrenamiento se ejecuta el script *train.py*, con los siguientes parámetros:

```
--model_def: yolov3-tiny_custom.cfg
--data: custom.data
--pretrained_weights: yolov3-tiny.weights
--epochs: 100
```

En el archivo *yolov3-tiny\_custom.cfg* se deben configurar los siguiente parámetros:

```
--Capa #15 [convolutional] se cambia
filters=18
```

```
--Capa #16 [yolo] se cambia classes=1
```

```
--Capa #22 [convolutional] se cambia
filters=18
```

```
--Capa #23 [yolo] se cambia classes=1
```

Y utilizamos los pesos del modelo preentrenado *yolov3-tiny.weights*

## 2.2 Implementación Yolo v3 con SE

Para implementar el módulo de atención SE (Squeeze-and-Excitation), creamos un nuevo archivo de configuración de YOLOv3, insertando el mecanismo.

El entrenamiento se ejecuta igualmente con el script *train.py*, con los siguientes parámetros:

```
--model_def: yolov3-tiny_custom_se.cfg
--data: custom.data
--pretrained_weights: yolov3-tiny.weights
--epochs: 100
```

En el archivo *yolov3-tiny\_custom\_se.cfg* se deben configurar los siguiente parámetros: entre la capa convolucional #12 y #13 se inserta el módulo SE (Squeeze-and-Excitation):

```
--[se]
--reduction=16
```

## 2.3 Implementación Yolo v3 con CBAM

Para implementar el módulo de atención CBAM (Convolutional Block Attention Module), creamos un nuevo archivo de configuración de YOLOV3 insertando el mecanismo.

El entrenamiento se ejecuta igualmente con el script `train.py`, con los siguientes parámetros:

```
--model_def: yolov3-tiny_custom_cbam.cfg
--data: custom.data
--pretrained_weights: yolov3-tiny.weights
--epochs: 100
```

En el archivo `yolov3-tiny_custom_cbam.cfg` se deben configurar los siguiente parámetros: entre la capa convolucional #12 y #13 se inserta el módulo CBAM

```
--[cbam]
--kernelsize=7
```

## III. Resultados

Implementamos YOLOv3 Tiny con dos mecanismos de atención, Squeeze-Excitation (SE) y Módulo de Atención de Bloques Convolucionales (CBAM). Para el análisis presentamos los resultados de desempeño de los modelos propuestos aplicados a la base de datos customizada. Usamos como métrica de rendimiento la Precisión media (mAP) para calcular así la precisión de todo el modelo ya que esto nos da el porcentaje de predicciones correctas. Como se observó en la tabla anterior.

**Cambias tabla**

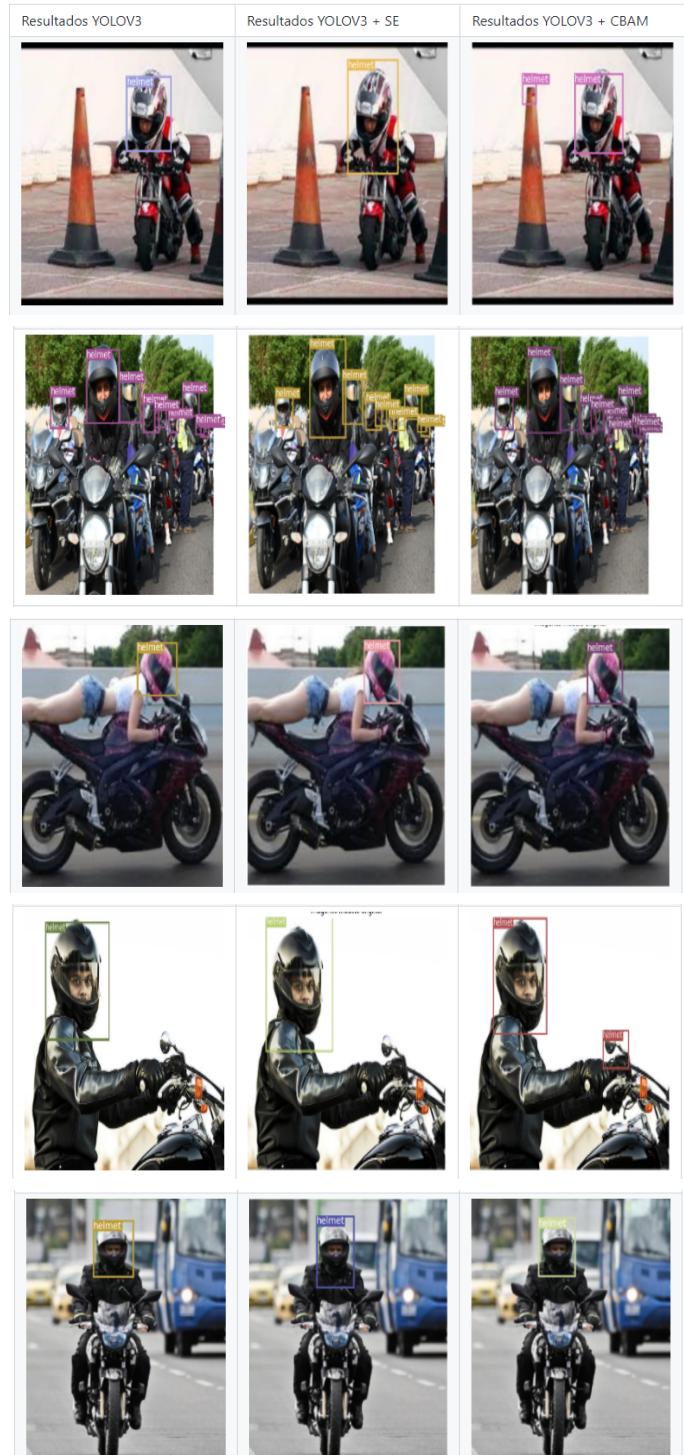
Modelo	Épocas	mAP
YOLOV3	100	0.62992
YOLOV3+SE	100	0.53420
YOLOV3+CBAM	100	

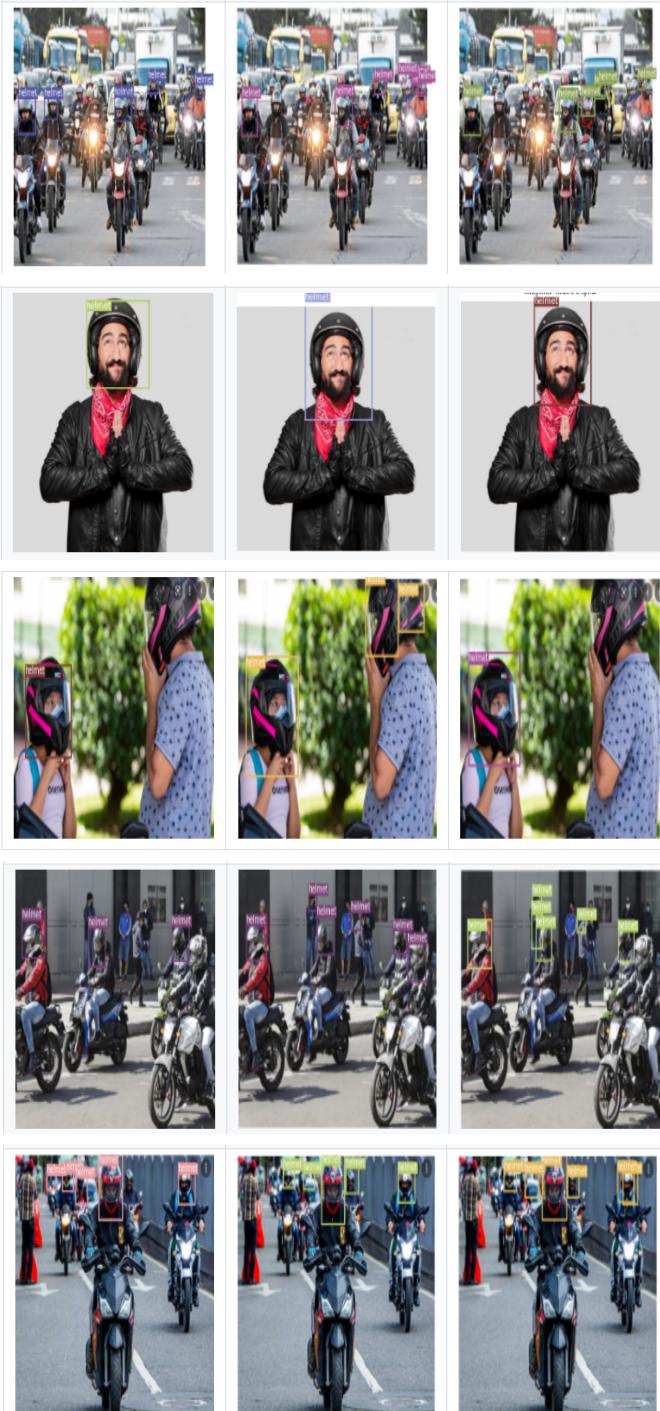
Tabla 1. Comparativo de desempeño de los modelos

Se puede observar que los modelos con módulo de atención Yolov3-tiny\_CBAM, superan a los demás modelos básico con un mAP de validación de **xxxx**, lo que en teoría comprueba mejora la detección de las imágenes.

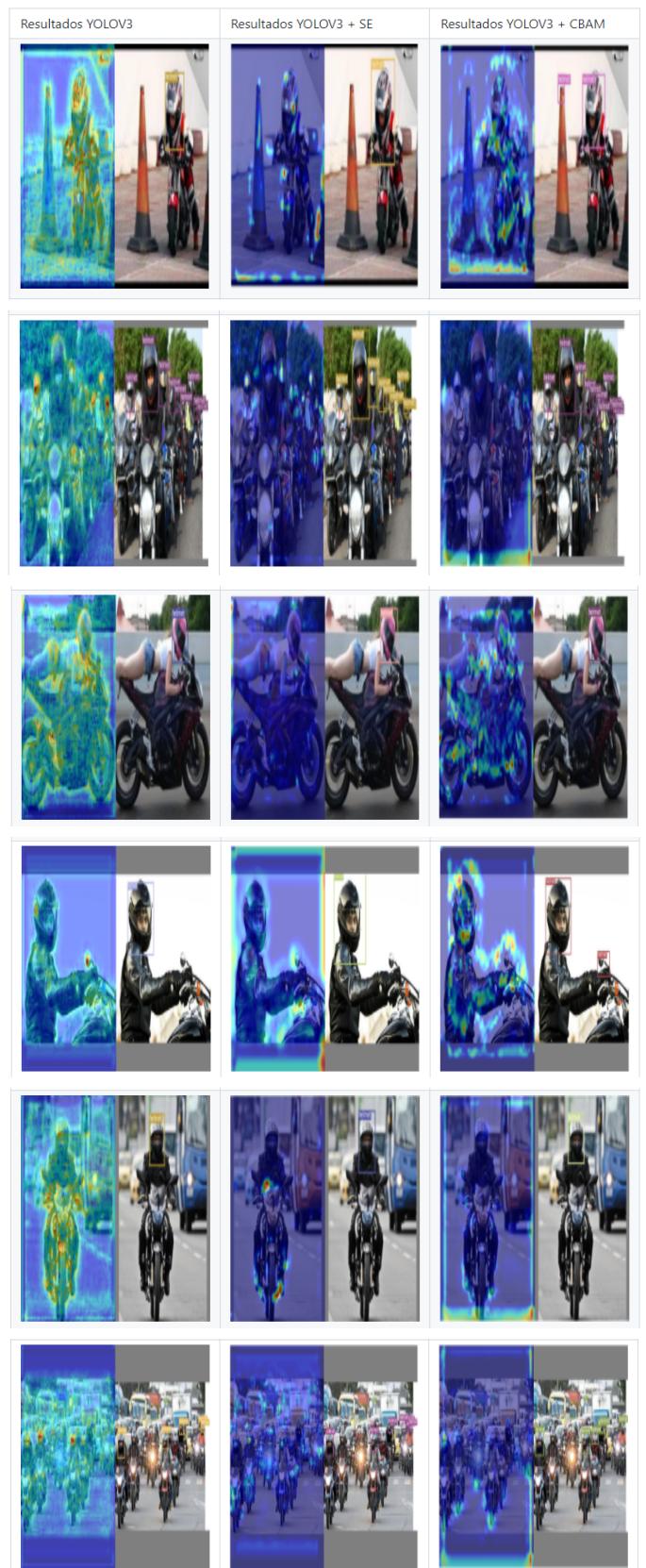
A continuación se muestra la validación de las arquitecturas en 10 diferentes imágenes de validación visualizando la atención en la clase 'helmet'

### 1. Detector de objetos en imágenes - clase "helmet"

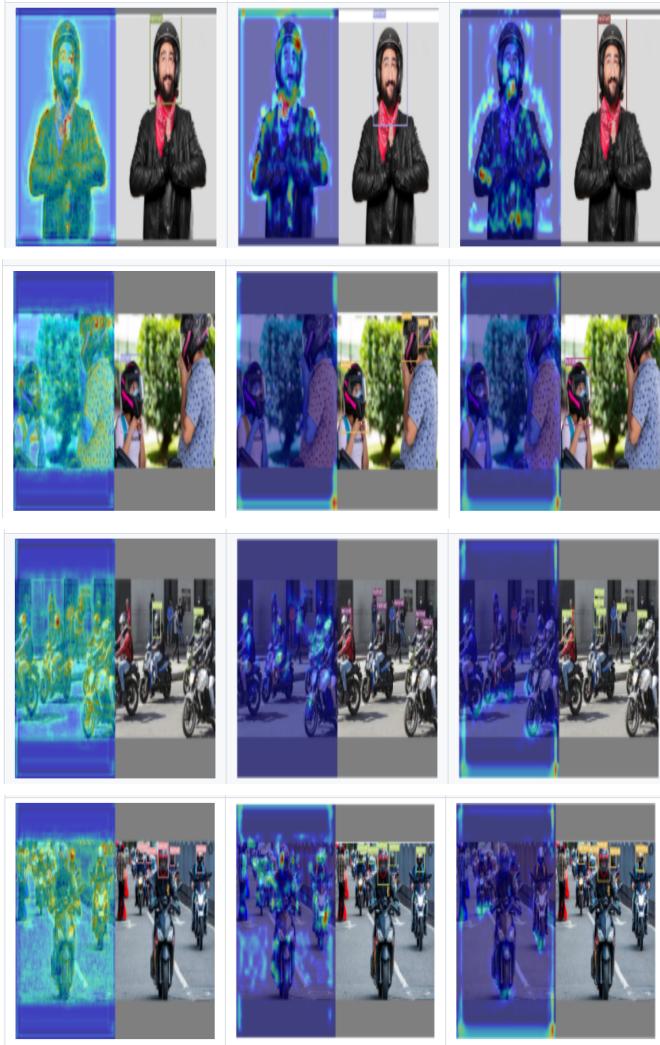




2. Imagenes con Heatmap - clase "helmet"



Por último realizamos el Grad-CAM para ver qué partes particulares de la imagen influyeron en la decisión del modelo para una etiqueta. A continuación se muestra la validación de las arquitecturas en 10 diferentes imágenes de validación visualizando la atención en la clase 'helmet' con GradCam:



Como se observa la arquitectura con módulo de atención agregados logra centrarse en rasgos deseados de la imagen que parecen significativos a la hora de predecir la detección; el módulo de atención SE fue quien detectó mejor las imágenes de prueba y logró clasificar más objetos en una sola imagen que las demás arquitecturas.

El modelo de cbam tuvo mejor resultado de mAP 0.7078 con respecto a los otros modelos, aunque fue más sensible en rasgos de las imágenes en casos como la img1 donde detecta al cono de transito como si fuera helmet.

## Conclusiones

A través de la implementación realizada y descrita anteriormente, se pudo validar que para la aplicación de la detección y clasificación de casco de motociclistas en un ambiente de tráfico, donde la confusa información de fondo causa un gran impacto en la detección del objetivo. Mediante la arquitectura de red YOLOv3-tiny se observó que hubo dificultades en la identificación de objetos superpuestos, y no fue fácil para el modelo distinguirlas.

En este sentido, se pudo validar que los mecanismos o modelos de atención definitivamente mejoran el poder de enfoque, cuando integramos los módulos de atención SE y CBAM a la red YOLO se observó mejoras en la precisión de detección a los objetivos pequeños.

Se consideró que los modelos pueden tener mejoras con muchas más épocas de entrenamiento y con un dataset más grande para así poder evidenciar mejores resultados en el modelo CBAM como se indica en los artículos estudiados en clases. Con nuestros datos actuales el mejor comportamiento tuvo el modelo de atención SE dando que fue más preciso en detectar los objetos que corresponden a la clase.

Partiendo de los resultados de este artículo y el artículo presentado anteriormente *RESNET50 CON MÓDULOS DE ATENCIÓN SE – CBAM* podemos concluir que con pocas épocas y imágenes de entrenamiento el mejor modelo es el SE, y teóricamente el modelo CBAM daría mejores resultados con más épocas de entrenamiento y un dataset más robusto según la información del estado del arte.

## Referencias

- [1] Redmon, Joseph. Divvala, Santosh. Girshick, Ross. Farhadi, Ali. “You Only Look Once: Unified, Real-Time Object Detection”, <https://arxiv.org/pdf/1506.02640.pdf>, Mayo 2016.
- [2] Chablani, Manish. “YOLO — You only look once, real time object detection explained”, <https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006>, Agosto 2017
- [3] A, Enrique. “Detección de objetos con YOLO: implementaciones y como usarlas”, <https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246>, Mayo 2018

- [4] Tzutalin. LabelImg. Git code  
<https://github.com/tzutalin/labelImg> , 2015
- [5] Rosebrok, Adrian. “YOLO object detection with OpenCV”,  
<https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/>, Noviembre 2018
- [6] Ulyanin Stepan. “Implementing Grad-CAM in PyTorch”  
<https://medium.com/@stepanulyanin/implementing-grad-cam-in-pytorch-ea0937c31e82> ,2019
- [7] Ramirez.A,Guarín.M,Mejía.C,Guerra.A. “Resnet 50 con Módulos de atencion SE-CBAM”  
[https://github.com/cmejia99/Resnet50\\_SE\\_CBAM/blob/main/Comparativo%20ResNet50\\_SE\\_CBAM.docx.pdf](https://github.com/cmejia99/Resnet50_SE_CBAM/blob/main/Comparativo%20ResNet50_SE_CBAM.docx.pdf),2021

