



Ojo de la Providencia

Alejandro Hernández Rodríguez

Julio 2021

Resumen (Abstract)

Celaya, Guanajuato es una de las regiones con mayor delicuencia a nivel nacional. El uso de herramientas modernas para mejorar la seguridad local es de conveniencia para toda la comunidad. Con esta problemática y esta hipótesis de resolución. He decidido utilizar el modelo de reconocimiento de objetos YOLO(You Only Look Once) v3 para reconocer un conjunto de objetos que se consideran peligrosos para la comunidad.

Introducción

Contexto del problema

El estado de Guanajuato ha habido una escala de violencia como nunca antes, además de ser el estado con mayor crecimiento del país en la última década, también ha sido uno de los más azotados por la violencia causada por los conflictos entre grupos delictivos. En el caso de Guanajuato es el conflicto entre el Cartel de Jalisco Nueva Generación y el Cartel de Santa Rosa de Lima, que pelean por el control de las 'plazas', ubicaciones privilegiadas geográficamente para la distribución, robo o producción de drogas y/o combustible ilícito.

Este conflicto considerado una guerra, no es convencional. Es definido por un conjunto de enfrentamientos armados esporádicos y aperiódicos en distintas partes del estado. Se caracterizan por ser repentinos y no tener una ubicación precisa, esta incertidumbre sobre donde y cuando será el siguiente enfrentamiento crea pánico y desesperación en la comunidad.

Propósito del estudio

Desarrollar un sistema de vigilancia con Inteligencia Artificial para la detección de armas, en un circuito cerrado de cámaras IP.

Descripción del Proyecto

Desarrollo de una arquitectura cliente servidor de un sistema de vigilancia de cámaras en tiempo real. El servidor recibe la transmisión de la cámara, procesa la imagen y en una plataforma presenta la transmisión con detección de objetos.

Hipótesis del trabajo

En un enfrentamiento armado, segundos pueden definir la vida y la muerte. Tener un sistema de vigilancia que pueda anticipar al menos por segundos la exposición de un arma de fuego puede ser vital para la seguridad de la comunidad, alertandola antes de que comience algún enfrentamiento armado.

Flujo de trabajo

Obtención del modelo

En la página oficial de Darknet <https://pjreddie.com/> de los creadores de YOLO se pueden descargar el archivo de configuración y los pesos de cualquier versión de YOLO entrenado con el conjunto de datos COCO.

Recolección y etiquetado de datos

Para la recolección de datos utilice el sitio de OpenImages
<https://storage.googleapis.com/openimages/web/visualizer/index.html>

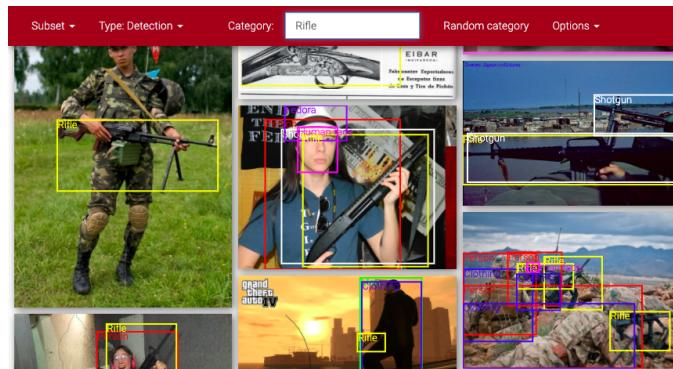


Figura 1: Sitio web de OpenImage

Para la descarga de imágenes utilicé un repositorio de github [referencia], que contenía un script de python que automatizaba la descarga, lo que hace es descargar las imágenes utilizando un comando de aws s3 y creando una carpeta para cada clase, que contiene dos tipos de archivos.

- Archivo jpg: Contiene la imagen

-
- Archivo txt: En cada fila contiene el nombre de la clase y 4 números flotantes que son las coordenadas del centro del bounding box de la clase, su anchura y altura.

Preprocesamiento de datos

Para el preprocesamiento de datos, lo que realice fue cambiar las anotaciones de cada archivo txt de cada imagen para hacerlos compatibles con los parámetros de darknet/YOLO. Además de juntarlos todos en una misma carpeta y crear un archivo que referencia la localización de las imágenes y sus archivos txt con la información de los bounding boxes. Para hacerlo también utilicé un script del repositorio que utilice para la descarga y etiquetado.

Entrenamiento con aprendizaje transferido

Darknet ya contiene un conjunto de comandos para realizar el aprendizaje transferido sin necesidad de crear el modelo en una paquetería convencional Tensorflow, Pytorch, etc., cargarle los pesos y tener que modificar la ultima capa para realizar el entrenamiento.

Para realizar el entrenamiento es necesario crear un archivo llamado model.data y modificar el archivo de configuración de la arquitectura de YOLO que vayamos a utilizar.

El archivo data contiene lo siguiente:

```
class = 10 # Numero de clases
train = data/train.txt # Ubicacion del conjunto de entrenamiento
valid = data/test.txt # Ubicacion del conjunto de validacion
names = data/obj.names # Ubicacion de los nombres de las clases
backup = backup # Carpeta donde se almacenan los pesos de la red neuronal
Y modifiqué los siguientes parámetros en el archivo yolov3-tiny.cfg
batch=32
subdivisions=2
```

Y cambie los filters a cada capa de convolución antes de la capa YOLO a 45, $(classes + 5) * 3$ en mi caso son 10 clases.

Antes de entrenar es necesario descargar los pesos para realizar el aprendizaje transferido, para descargarlas utilicé el comando:

```
./ darknet partial cfg/yolov3-tiny.cfg yolov3-tiny.weights yolov3-tiny-15.conv.1
```

Estos pesos son para el modelo tiny pre-entrenado de YOLOv3.

El comando para entrenar que utilicé fue:

```
./ darknet detector train cfg/model.data cfg/yolov3-tiny-custom.cfg yolov3-tiny
```

En mi caso utilice la arquitectura de YOLO-tiny, una versión ligera con menos capas de convolución y menor exactitud al momento de clasificar, por la limitada capacidad de procesamiento que contiene mi computadora.

Implementación de detección en tiempo real

Para la implementación en tiempo real, utilicé el modulo de OpenCV para python. Fue necesario la construcción manual del modulo para poder utilizar GPU de Nvidia con CUDA en el paquete DNN para poder realizar una rápida detección, ya que en caso contrario la detección era muy lenta. El submodule DNN ya tiene un método definido para la creación de una Red Neuronal Artificial a partir de los pesos y archivo de configuración definidos en el entrenamiento.

Conexión a cámaras de seguridad

Las camaras a las que el sistema se conecta son de la compañía dahua, para la transmisión en vivo de las camaras se puede utilizar el protocolo RTSP(Real Time Streaming Protocol) para obtener el stream de las cámaras de seguridad.

```
rtsp://user:pwd@ip_address/cam/realmonitor?channel=1&subtype=0
```

Para la conexión de las cámaras, realice un port forward en el puerto 554 para la transmisión en tiempo real con el protocolo RTSP de la red donde están conectadas las camaras de seguridad. Utilice opencv para utilizar RTSP y poder visualizar las camaras en cualquier computadora con su dirección ip publica.

Creación de plataforma

Para el deployment del modelo utilice el modulo de python llamado streamlit. Es una aplicación sencilla que su objetivo es notificar si detecta un arma en la cámara que se escoge, esta notificación será un sonido en la plataforma y un mensaje.



Figura 2: Captura de pantalla del sistema

En la plataforma puedes escoger que cámara observar y que versión de YOLO utilizar, esto último es para fines demostrativos, y tiene un botón **run** que es para activar o desactivar el video.

Mapeo del sistema

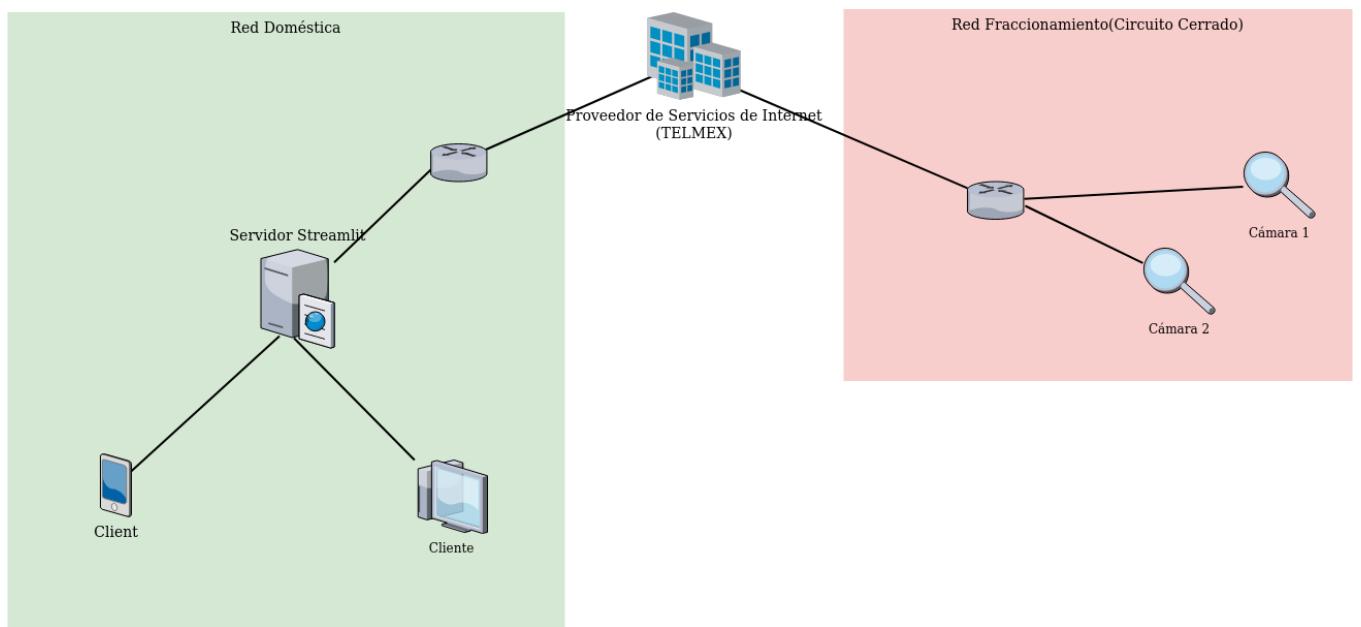


Figura 3: Arquitectura del Sistema

Definición de Métricas

En artículo de YOLO mencionan cómo utiliza como función de pérdida la suma ponderada de 3 distintas funciones de costo.

Cada una está definida de la siguiente forma:

- Iou loss = $\sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2$

El IOU(Intersection over union) define que tanto peso tiene la intersección de dos bounding boxes sobre su unión. Esto define si descomponemos en dos cajas de diferentes clases o solo se mantiene una de una clase.

- Class loss = $\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$

Función de error de cada clase.

- Average mse = $\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$

El mse de las bounding boxes.

Exploración de datos

En el apartado de flujo de trabajo ya explique como etiquetar los datos para poder entrenar YOLO, en la figura 4 presento un ejemplo.



```
Open ▾ *0a46f9b3
~/Documents
1 # Clase
0.5209375 0.626188541727672 # Coordenadas centro
0.8912500000000001 # Alto
0.6060039999999999 # Ancho|
```

Figura 4: Imagen y su descripción

Métodos y modelo

Como menciona previamente el modelo utilizado es un arquitectura de una Red Neuronal Artificial (ANN) por sus siglas en inglés, en particular una Red Neuronal Convolutacional. En esencia el modelo de YOLO es una red neuronal convolucional convencional, lo esencial de este modelo es la forma en la que se representan los datos de entrenamiento.

La neurona

Las neuronas son celulas que junto con las celulas Glia conforman todo el sistema nervioso del cuerpo humano. Todas las neuronas estan compuestas de un cuerpo celular, que mantiene una carga eléctrica interna, un conjunto de dendritas y un axón, Fig. 5. A la conexión entre dos neuronas se le llama sinapsis.

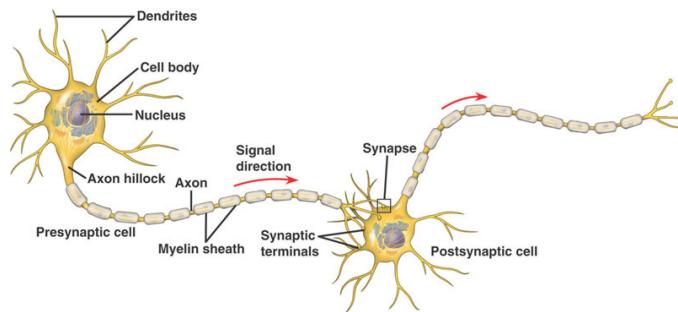


Figura 5: Imagen de una neurona

Las neuronas conectan con otras neuronas a través del axón, y son conectadas por otras neuronas a través de las dendritas. Se dice que una neurona es estimulada cuando a través de sus dendritas recibe carga eléctrica por parte de otras neuronas. Si la carga recibida es mayor a un potencial de membrana, que es la diferencia de potencial entre el exterior y el interior del cuerpo celular de la neurona, la neurona se deshace o 'dispara' a través del axón la diferencia de potencial extra que tiene a las dendritas de otras neuronas.

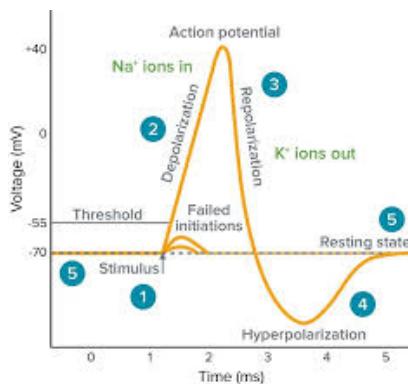


Figura 6: Gráfica de potencial de membrana de una neurona, si el voltaje interno supera -55 [mV] la neurona disparará.

Es así cómo, según la teoría conectivista, todos los procesos cognitivos del cuerpo humano son representados como el conjunto de entradas y salidas de cada una de las neuronas que son estimuladas por una cascada de señalización de hormonas por parte de otras células no nerviosas que son estimuladas bioquímicamente por el mundo externo.

Perceptrón

En el año 1943 McCulloch y Pins, inspirados por la fisiología básica de la neurona desarrollan un modelo de una neurona artificial, esta neurona solo tendría dos estados activada(dispara) y desactivada(no dispara).

A partir de ese trabajo Frank Rosenblatt define el Perceptrón en (1962), donde el voltaje que reciben las dendritas se modela como los datos de entrada y los pesos se interpretan como la cantidad de mielina de los axones de las neuronas que disparan a las dendritas de la neurona. El voltaje interno de la neurona es la combinación lineal de los pesos y las entradas, y el threshold para disparar es definido por la función heaviside (no lineal).

Además prueba el **Teorema de convergencia del perceptrón** *Dado un conjunto de datos separables linealmente el perceptron convergerá a una solución.*

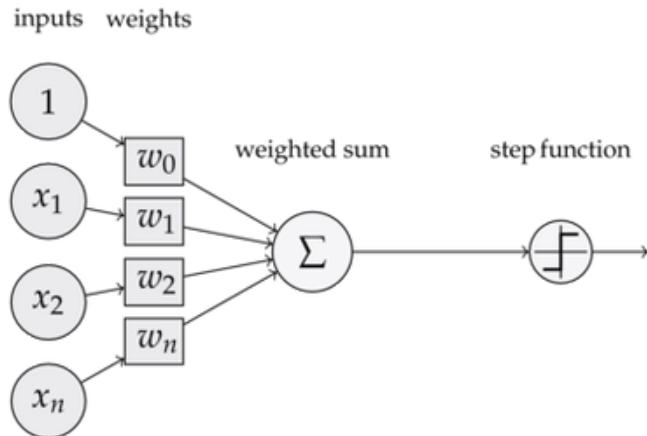


Figura 7: Perceptrón

Se define de la siguiente manera

$$h(x) = \begin{cases} 1 & \text{si } \sum_i w_i x_i \geq 0 \\ 0 & \text{si } \sum_i w_i x_i < 0. \end{cases}$$

El objetivo del perceptrón es aproximar una función $f : X \rightarrow Y$ utilizando un conjunto de pares de datos de la forma $\{(x_i, y_i) | i \in \mathbb{N}, i \leq m\}$ donde m es la cantidad de pares, a partir de una medida de rendimiento que en Machine Learning se le conoce como función de perdida.

Función de perdida:

$$L(x, y) = \sum_i (\hat{y}_i - h(\hat{x}_i))^2,$$

Dada las limitantes de poder clasificar datos únicamente separables linealmente, y el fracaso en poder clasificar la función lógica XOR. Nace la idea de crear un Perceptrón Multicapa.

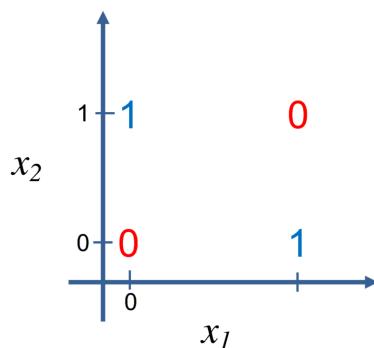


Figura 8: Función XOR

Redes Neuronales Artificiales

Un perceptrón multicapa(MLP) o Red Neuronal Artificial, como se le conoce coloquialmente, es el modelo matemático de la concatenación de capas de perceptrones. En la Fig. 9 hay una representación gráfica del MLP, cada uno de los nodos representa un perceptrón. A diferencia de la primera columna de nodos, eso representa los datos de entrada. A cada columna se le llama capa, a la primera capa se le llama capa de entrada, a la última se le llama capa de salida y las intermedias se les llama capas ocultas.

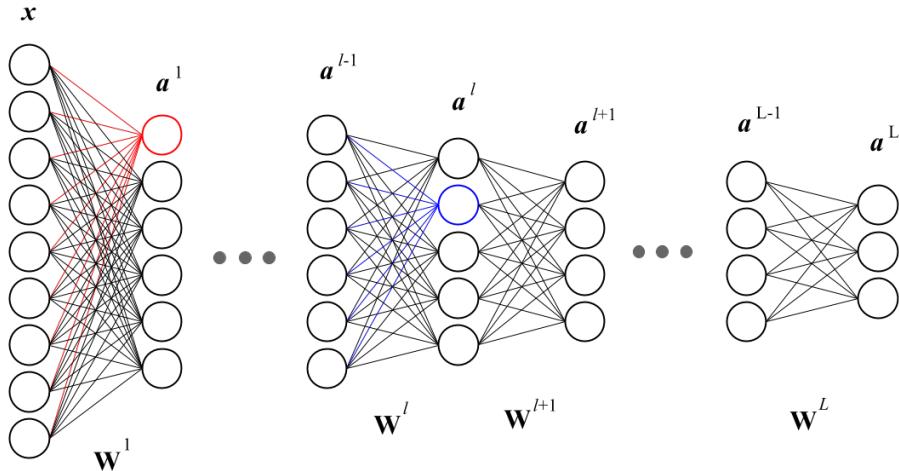


Figura 9: Representación gráfica de Perceptrón Multicapa.

Hay unas cuantas diferencias con el perceptrón original además de la cantidad de perceptrones, como lo es una diferente función de activación y las entradas de cada perceptrón. En la figura 9 se observa la notación que tiene el perceptrón multicapa, a^l representa la $l - \text{ésima}$ capa, W^l es la matriz de pesos de todos los perceptrones entre la capa a^l y a^{l-1} , y también b^l son los sesgos o pesos constantes de la capa a^l . a^L representa la capa final, y $a^1 = x$ es la capa de entrada.

Ahora las entradas del perceptrón que se encuentra en la capa l de tamaño K son las salidas de los perceptrones de la capa $l - 1$ de tamaño J :

$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^{l-1}, \quad (1)$$

$$a_k^l = \sigma(z_k^l), \quad (2)$$

o en notación matricial

$$a^l = \sigma(W^l a^{l-1} + b^{l-1}). \quad (3)$$

La función de activación se cambió de una función escalonada a una función sigmoide (ecuación 4) para que sea diferenciable y se pueda aplicar el algoritmo de propagación hacia atrás o **Backpropagation** en inglés.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (4)$$

Esta función es parecida a la función escalonada gráficamente.

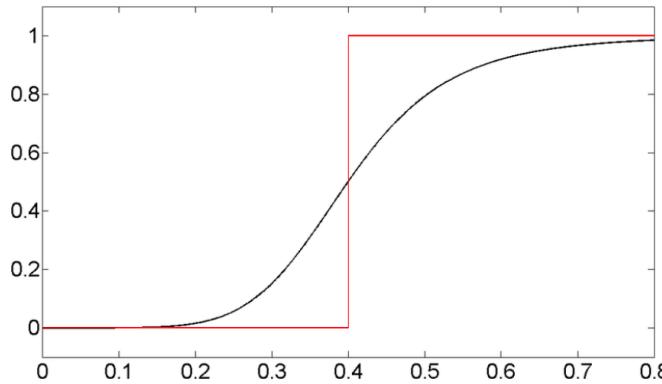


Figura 10: Gráfica función heaviside (rojo) y sigmoide (negro).

La función de costo es la misma que el perceptrón normal pero la diferencia sería únicamente con la capa a^L

$$L(x, y) = \sum_i^n (\hat{y}_i - a^L)^2,$$

donde

$$a^L = \sigma(W^{(L)}\sigma(\dots\sigma(W^{(2)}\sigma(W^{(1)}\hat{x}_i + b^{(1)}) + b^{(2)})\dots) + b^{(L)}).$$

Entonces para reducir la perdida igual que en el perceptrón original el objetivo es encontrar el tensor de pesos que reduzca la función de perdida lo mayor posible, es decir, encontrar el mínimo de la función L . Debido a que L es una función real y convexa podemos utilizar los criterios de la primera y segunda derivada para encontrar el mínimo. El problema es que es casi imposible obtener analíticamente las derivadas parciales de la función de perdida respecto a cada peso

$$\frac{\partial L}{\partial w_{ij}^l},$$

para eso es el algoritmo de backpropagation, que consiste en encontrar numéricamente las derivadas parciales respecto a cada peso.

Backpropagation

Suponiendo que tenemos las capas de la figura 9, sabemos por regla de la cadena que

$$\frac{\partial L}{\partial w_{kj}^l} = \frac{\partial L}{\partial a_k^l} \frac{\partial a_k^l}{\partial z_k^l} \frac{\partial z_k^l}{\partial w_{kj}^l},$$

aplicando nuevamente regla de la cadena

$$\frac{\partial L}{\partial w_{kj}^l} = \left(\sum_m \frac{\partial L}{\partial z_m^{l+1}} \frac{\partial z_m^{l+1}}{\partial a_k^l} \right) \frac{\partial a_k^l}{\partial z_k^l} \frac{\partial z_k^l}{\partial w_{kj}^l},$$

sustituyendo y simplificando

$$\frac{\partial L}{\partial w_{kj}^l} = \left(\sum_m \frac{\partial L}{\partial z_m^{l+1}} w_{mk}^{l+1} \right) \sigma'(z_k^l) a_j^{l-1}.$$

Por convención se define δ que es el *error*.

$$\delta_k^l \equiv \frac{\partial L}{\partial z_k^l}$$

Y sabemos que

$$\delta_k^l = \left(\sum_m \frac{\partial L}{\partial z_m^{l+1}} w_{mk}^{l+1} \right) \sigma'(z_k^l) = \left(\sum_m \delta_m^{l+1} w_{mk}^{l+1} \right) \sigma'(z_k^l)$$

Llegando a la ultima ecuación podemos observar que las δ de la capa l están en función de la capa $l + 1$, así es como se *propaga* el error hacia atrás. Así que lo primero que importa es obtener δ^L , para obtener las demás δ .

$$\delta_j^L = \frac{\partial L}{\partial a_j^L} \sigma'(z_j^L)$$

Recordando que una de las propiedades de la función sigmoide es

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x)),$$

podemos reescribir δ^L como:

$$\delta_j^L = \frac{\partial L}{\partial a_j^L} \sigma(z_j^L)(1 - \sigma(z_j^L)).$$

Con esta última expresión podemos observar que para realizar el backpropagation únicamente tenemos que calcular una derivada parcial $\frac{\partial L}{\partial a_j^L}$, y todas las demás las obtenemos propagando $\delta^{L-1} = \delta^L W^L \circ \sigma'(z^L)$, hasta δ^1 .

Teniendo las definiciones anteriores el algoritmo es el siguiente:

$$\delta^L = \nabla_{a^L} L \circ \sigma'(z^L),$$

$$\delta^l = (W^{(l+1)})^T \delta^{l+1} \circ \sigma'(z^l),$$

$$\frac{\partial L}{\partial b_k^l} = \delta_k^l,$$

$$\frac{\partial L}{\partial w_{kj}^l} = \delta_k^l a_j^{l-1}.$$

Una vez obtenido el gradiente, se utilizan algoritmos de optimización basados en gradiente, para encontrar el mínimo de la función L , por ejemplo descenso por gradiente:

$$w_{kj}^l = w_{kj}^l - \alpha \frac{\partial L}{\partial w_{kj}^l}.$$

Red Neuronal Convolucional

Por el teorema de aproximación universal, el perceptrón multicapa es capaz de aproximar a cualquier función del tipo $f : X^m \rightarrow [0, 1]^n$. Teóricamente es cierto pero prácticamente no se tienen los recursos computacionales para desarrollar un MLP (Multilayered Perceptron) con una cantidad infinita de perceptrones. Dada esta limitación el desempeño de los MLP no es muy buena al momento de clasificación de vectores de alta dimensionalidad, dado que se tiene que entrenar por mucho tiempo, necesitan cantidades exorbitantes de datos y es muy sensible a variaciones de la entrada.

Por ejemplo en clasificación de imágenes, los imágenes RGB de dimensión $512 \times 512 \times 3$ tendrían que ser transformadas a vectores fila de dimensión 786432, esta dimensionalidad causa los problemas descritos anteriormente.

Para la solución de clasificación de imágenes se desarrollaron las Redes Neuronales Convolucionales, llamadas así porque en lugar de tener perceptrones tiene unidades de convolución.

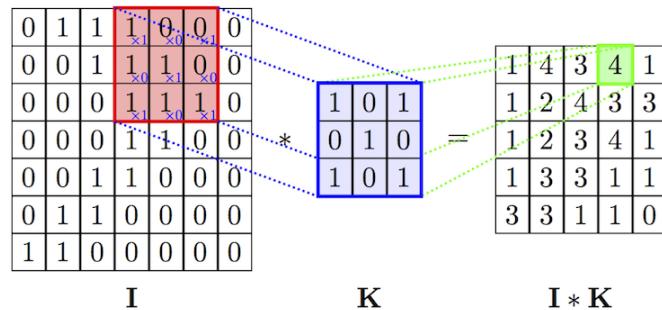


Figura 11: Operación de convolución.

Recordemos la operación de convolución (representación gráfica en la figura 11).

$$I * K(x, y) = \sum_{a=0}^{k-1} \sum_{b=0}^{k-1} I(x-a, y-b)K(a, b)$$

Una unidad de convolución en lugar de tener un peso con cada unidad de la capa anterior, como los perceptrones, convoluciona con la capa anterior, como se muestra en la figura 12. Esto permite que aprenda relaciones locales de los datos. En el caso de las imágenes existen relaciones fuertemente locales entre pixeles, ya que un pixel tiende a estar con pixeles similares, en caso de no ser una esquina o un borde.

Usando la notación del MLP podemos definir la una unidad de convolución como:

$$z_{(x,y)}^l = w_{(x,y)}^l * a_{(x,y)}^{l-1}) = \sum_{x'} \sum_{y'} w_{(x',y')}^l a_{(x-x',y-y')}^{l-1}$$

Donde x' y y' son las dimensiones de la unidad de convolución $a_{(x-x'+1,y-y'+1)}^l$

$$a_{(x,y)}^l = \sigma(z_{(x,y)}^l) + b_{x,y}^l.$$

Es importante notar que ahora cada capa de la red será bidimensional, en la figura 12 hay una ilustración gráfica.

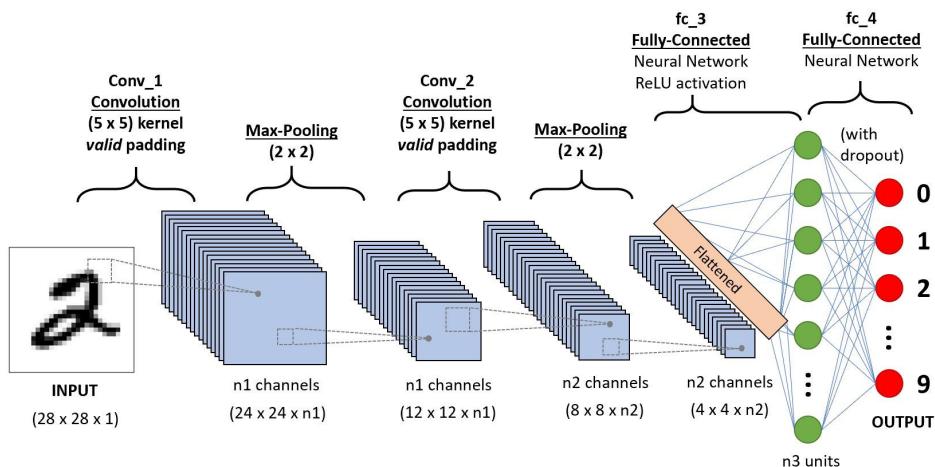


Figura 12: Red Neuronal Convolucional.

Similar al MLP, se utiliza la misma función de pérdida, también se optimiza utilizando Backpropagation y descenso con gradiente.

El algoritmo de BackPropagation tendrá una ligera variante para la unidad convolucional, pero se basa en el mismo principio, encontrar deltas de error y propagarlo a capas anteriores.

Como en el MLP tenemos que

$$\frac{\partial L}{\partial w_{x,y}^l} = \sum_{x'} \sum_{y'} \frac{\partial L}{\partial z_{x',y'}^l} \frac{\partial z_{x',y'}^l}{\partial w_{x,y}^l}. \quad (5)$$

Ahora las deltas de error, no serán vectores sino matrices.

$$\delta_{(x,y)}^l \equiv \frac{\partial L}{\partial z_{(x,y)}^l} \quad (6)$$

También nos será útil obtener lo siguiente

$$\frac{\partial z_{x',y'}^l}{\partial w_{x,y}^l} = \sigma(z_{(x-x',y-y')}^{l-1}) \quad (7)$$

$$\frac{\partial L}{\partial w_{x,y}^l} = \sum_{x'} \sum_{y'} \delta_{(x',y')}^l \sigma(z_{(x-x',y-y')}^{l-1}) \quad (8)$$

Una vez definido lo anterior podemos proceder a obtener las $\delta^{(l+1)}$

$$\frac{\partial L}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \frac{\partial L}{\partial z_{x',y'}^{l+1}} \frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \delta_{(x',y')}^{l+1} \frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l} \quad (9)$$

Ahora falta obtener $\frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l}$, desarrollando $z_{x',y'}^{l+1}$

$$z_{x',y'}^{l+1} = \sum_{x''} \sum_{y''} w_{(x'',y'')}^{l+1} \sigma(z_{(x'-x'',y'-y'')}^l) \quad (10)$$

Entonces sabiendo por los índices de la ecuación 10 que $x'' = x' - x$, por que cuando $x'' = x' - x \implies \sigma(z_{(x'-x'',y'-y'')}^l) = \sigma(z_{(x,y)}^l)$ y ese es el único coeficiente que no se vuelve cero después de aplicar la derivada.

$$\frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l} = w_{(x'-x,y'-y)}^{l+1} \sigma'(z_{(x,y)}^l) \quad (11)$$

Sustituyendo 11 en 9 obtenemos las siguientes definiciones

$$\frac{\partial L}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \delta_{(x',y')}^{l+1} w_{(x-x',y-y')}^{l+1} \sigma'(z_{(x,y)}^l) \quad (12)$$

$$\frac{\partial L}{\partial z_{x,y}^l} = \delta_{(x,y)}^{l+1} * w_{(x,y)}^{l+1} \sigma'(z_{(x,y)}^l) \quad (13)$$

Obtenidas las derivadas parciales, solo es aplicar los pasos del backpropagation.

You Only Look Once

En mayo del 2016 Joseph Redmon et al. publican el artículo You Only Look Once:Unified, Real-Time Object Detection. En el artículo definen un modelo inspirado en la arquitectura convolucional de GoogLeNet para mejorar la detección de multiples objetos en una imagen. En la ultima capa utilizan una función de activación leaky ReLu y como función de perdida se utilizara el error cuadrático medio con algunas restricciones extra.

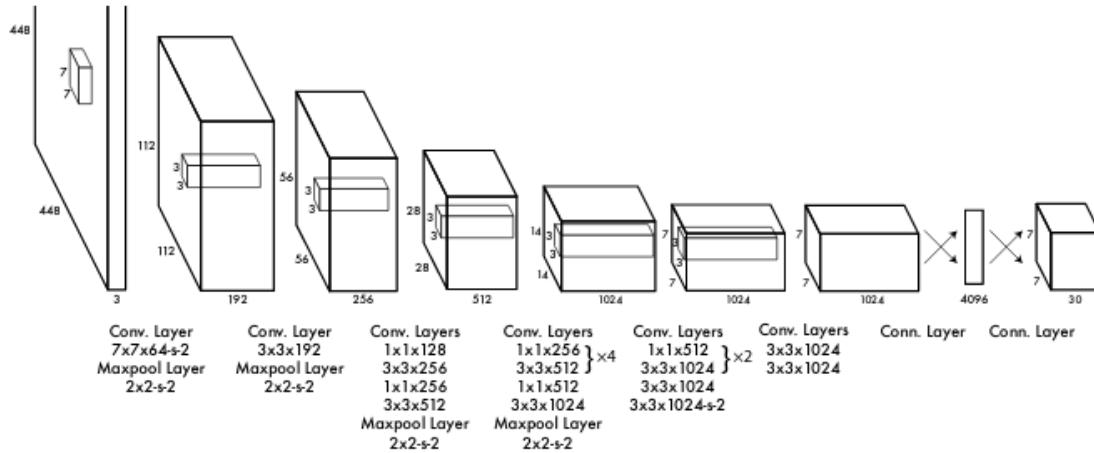


Figura 13: Arquitectura YOLO(Imagen del artículo general).

Lo revolucionario de YOLO fue la definición del target con el cual se entrena la red. Se define un vector de dimension $S \times S \times (B * 5 + C)$ donde S es la dimensión la cuadricula que se crea en la imagen, B es el número de Bounding Boxes(Cajas que rodean el objeto) se multiplica por que por cada Bounding Box se almacenan 5 datos las coordenadas del centro, la altura de la caja, el ancho de la caja y la probabilidad que haya un objeto ahí, por ultimo C es el número de clases.

$$y = [p_c^1 b_x^1 b_y^1 b_h^1 b_w^1 c_1^1 \dots c_n^1 \dots p_c^{S \times S} b_x^{S \times S} b_y^{S \times S} b_h^{S \times S} b_w^{S \times S} c_1^{S \times S} \dots c_n^{S \times S}]^T \quad (14)$$

Donde n es el número de clases. En la Figura 12 se observa gráficamente cómo es la búsqueda de los bounding boxes, y como define donde esta cada objeto a partir de las celdas por cada clase.

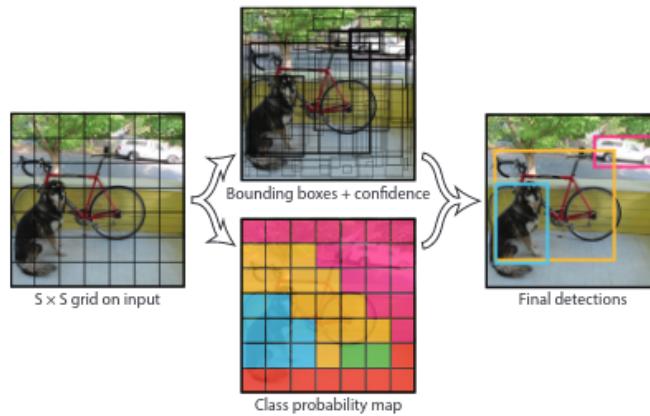


Figura 14: Visualización del vector target.

Una de las limitaciones de YOLO es que asume fuertes relaciones espaciales entre las predicciones de un solo Bounding Box esto quiere decir que si dos objetos parecen estar en la misma Bounding Box YOLO solo predecira una clase en lugar de las dos.

Aun con esa limitación YOLO es un sistema relativamente rápido a comparación de otros modelos. Es por eso que es uno de los modelos más usados para la detección de objetos.

Evaluación de Modelo

```
(next mAP calculation at 10170 iterations)
Last accuracy mAP@0.5 = 50.27 %, best = 50.27 %
10000: 1.969065, 1.685366 avg loss, 0.000010 rate, 1.224987 seconds, 320000 images, 0.023935 hours left
Resizing to initial size: 416 x 416 try to allocate additional workspace_size = 52.43 MB
CUDA allocate done!

calculation mAP (mean average precision)...
Detection layer: 16 - type = 28
Detection layer: 23 - type = 28
8140
detections_count = 178985, unique_truth_count = 19571
class_id = 0, name = Rifle, ap = 56.70%          (TP = 516, FP = 206)
class_id = 1, name = Handgun, ap = 77.71%         (TP = 368, FP = 56)
class_id = 2, name = Shotgun, ap = 42.22%         (TP = 79, FP = 30)
class_id = 3, name = Car, ap = 32.96%            (TP = 708, FP = 661)
class_id = 4, name = Truck, ap = 68.45%           (TP = 784, FP = 205)
class_id = 5, name = Person, ap = 29.83%          (TP = 1369, FP = 1232)
class_id = 6, name = Vehicleregistrationplate, ap = 0.00%      (TP = 0, FP = 0)
class_id = 7, name = Knife, ap = 67.92%            (TP = 308, FP = 79)
class_id = 8, name = Motorcycle, ap = 66.75%       (TP = 894, FP = 204)
class_id = 9, name = Humanface, ap = 51.32%        (TP = 1694, FP = 783)

for conf_thresh = 0.25, precision = 0.66, recall = 0.34, F1-score = 0.45
for conf_thresh = 0.25, TP = 6720, FP = 3456, FN = 12851, average IoU = 49.09 %

IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
mean average precision (mAP@0.50) = 0.493871, or 49.39 %
Total Detection Time: 68 Seconds
```

Figura 15: Métricas de entrenamiento.

En la figura 13 se observan las métricas del modelo al terminar de entrenarse, se observa que no tiene un muy buen desempeño con un valor de 0.45 de F1 esto es por que del dataset público que conseguí no pude encontrar la misma cantidad de imágenes por clase.

Respuesta a las hipótesis planteada

Inconclusa, debido a los escasos recursos no pude desarrollar un sistema de vigilancia con alta eficacia y precisión. En el análisis de resultados hago una comparativa sobre utilizar un modelo tiny y un modelo normal de YOLO.

Análisis de resultados

Utilice dos distintos modelos que entrene con diferente cantidad de clases. Además de la diferencia de modelos, no tuve los mejores resultados por la calidad de las cámaras de seguridad.

En la figura 14 se observa la diferencia de los dos modelos. Es importante resaltar que en el tiny no detecto el arma ni el coche. Y en el caso del normal sí.

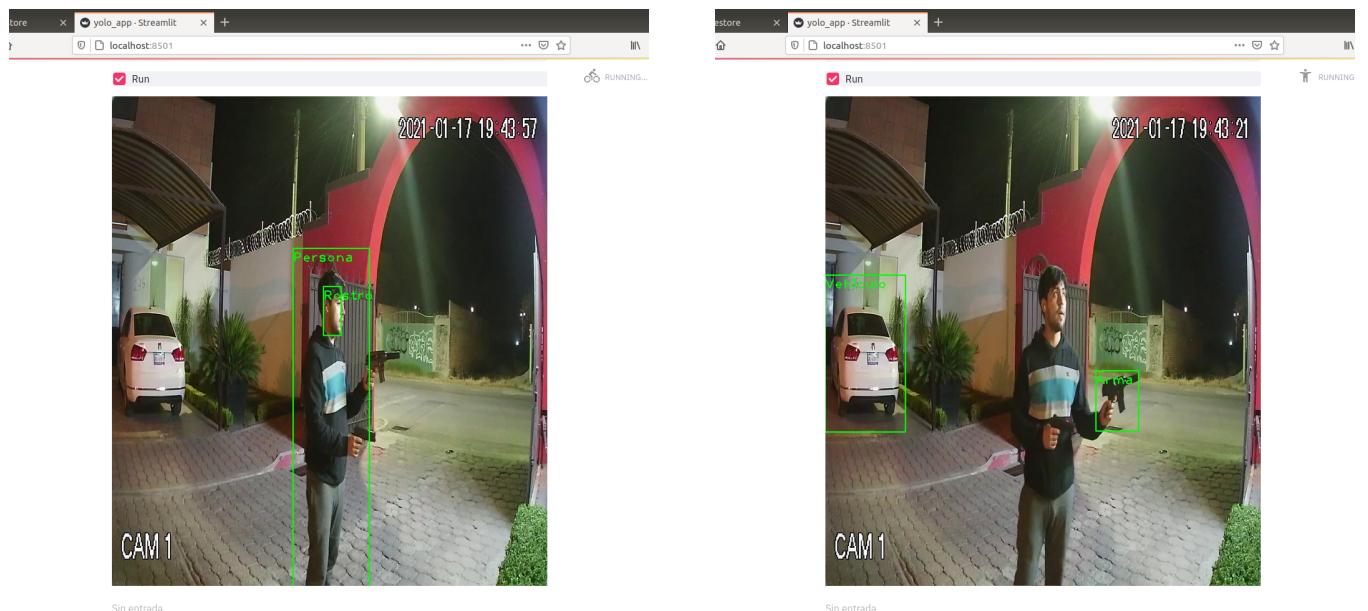


Figura 16: Modelo personalizado tiny a la izquierda y modelo personalizado normal a la derecha

Yolo V3 personalizado

Lo entreno con las clases Arma, Vehículo y Rifle. Lo hice con estas porque para entrenarlo me costo aproximadamente dos días. Este modelo cuenta con más capas de convolución de mayor área y con mayor cantidad de unidades de convolución.

Yolo V3 personalizado tiny

Lo entrene con las clases Rifle, Pistola, Escopeta, Carro, Camion, Persona, Placas, Navaja, Motocicleta y Rostro. Utilice más clases para entrenar este modelo ya que duraba mucho menos entrenarlo. Aproximadamente 3 horas. Pero su desempeño no es muy bueno cuando no hay una buena iluminación.

Conclusiones y recomendaciones

Conclusiones

- Al usar el modelo tiny de YOLO, la detección disminuye considerablemente. Se observa en la comparación realizada en el análisis de resultados.
- Falta de paralelización para visualizar todas las camaras al mismo tiempo, actualmente streamlit no cuenta con la opción de ejecutar procesos o hilos. Entonces no pude mostrar cámaras de manera simultanea.
- Entrene con GTX 1050 ti, y un servidor sin conexión directa al ISP. Entonces la velocidad de streaming es lenta y momento de procesar la imagen a través del modelo disminuye los fps.

Recomendaciones

- Utilizar una tarjeta gráfica de la gama 2000 de Nvidia.
- Data augmentation y etiquetado manual de los datos para aumentar la cantidad de datos que no pude bajar solo de internet.

Referencias

- [1] <https://pjreddie.com/darknet/yolo/>
- [2] International Narcotics Control Board 1999, United Nations, accessed 1 October 1999, <http://www.incb.org>
- [3] Erik Hallström, 2016, <https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d>
- [4] Jefkine, 2016, <https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/>
- [5] Alberto Nájar, 2020, <https://www.bbc.com/mundo/noticias-america-latina-51200888>
- [6] Trevor Hastie, 2001, Elements of statistical learning

- [7] YOLOv3: An Incremental Improvement, Joseph Redmon <https://arxiv.org/pdf/1506.02640.pdf>
- [8] You Only Look Once:Unified, Real-Time Object Detection, Joseph Redmon <https://arxiv.org/pdf/1804.02767.pdf>
- [9] Peter Norvig and Stuart J. Russell, 1994, Artificial Intelligence: A Modern Approach
- [10] MINIMUMWIDTH FORUNIVERSALAPPROXIMATION, <https://openreview.net/pdf?id=O-XJwyoIF-k>
- [11] <https://leimao.github.io/blog/Perceptron-Convergence-Theorem/>
- [12] https://github.com/theAIGuysCode/OIDv4_ToolKit
- [13] <https://github.com/AlexeyAB/darknet>