
Proyecto

Reentrenamiento de red Tiny Yolo

Profesor:
Javier Ruiz del Solar

Auxiliar:
Patricio Loncomilla Z.

Guía:
Cristopher Gómez.

Ayudantes:
Francisco Leiva C.
Nicolas Cruz Brunet

Alumno:
Carlos Contreras M.

Fecha:
8 de Marzo, 2019

1. Introducción

Este proyecto tiene como objetivo la obtención de una base de datos para reentrenar Tiny Yolo para que pueda clasificar objeto de uso hogareño, como: cajas, botellas, latas, etc.

Para este trabajo, no se cuentan con datos para empezar a armar la base, por lo que es necesario capturar las objetos, segmentarlos y plasmarlos sobre un contexto el cual sería un plano.

Para lograr la implementación necesaria se programará en *Python 3* junto con la librería *OpenCV* y otros paquetes requeridos para hacer operaciones sobre imágenes.

Este trabajo consiste en las siguientes secciones:

- **Marco teórico:** En esta sección se describen los elementos importantes a definir bajo este proyecto, en los cuales se enfoca en convolución, transformaciones, *Data Augmentation* y redes neuronales convolucionales.
- **Desarrollo:** Aquí se mostrará el desarrollo necesario para la instalación de librerías requeridas, obtención de las máscaras de los objetos y la generación de escenas de objetos sobre el plano. Finalmente se expondrá sobre el entrenamiento de *Tiny Yolo* y los resultados obtenidos.
- **Conclusiones:** Finalmente se exponen los puntos que fueron difíciles de abordar, junto con los aprendizajes obtenidos, dando a conocer si los resultados obtenidos fueron los esperados o cual fue el motivo de que los resultados no fuesen favorables.

2. Marco Teórico

Una imagen es la combinación espacial (en este caso en 2 dimensiones) de puntos con diversos grados de iluminación por color, para esta tarea se focalizará solo en escala de grises, la cual va desde 0 (negro) a 255 (blanco), es decir, 256 valores enteros posibles de iluminación.

Es posible realizar diversas operaciones en las imágenes, de las cuales se detallarán a continuación las más importantes para la elaboración de esta tarea.

2.1. Convolución

El proceso de convolución corresponde a la aplicación de un filtro lineal g a una imagen f , en el cual un píxel resultado corresponde a la interacción del filtro en conjunto de los vecinos del píxel en cuestión.

Dado el caso de las imágenes, el proceso de convolución es discreto, representado por la segunda ecuación de la siguiente figura :

$$f(x, y) * g(x, y) = \int_{\tau_1=-\infty}^{\infty} \int_{\tau_2=-\infty}^{\infty} f(\tau_1, \tau_2) g(x - \tau_1, y - \tau_2) d\tau_1 d\tau_2 \quad (1)$$

$$f[x, y] * g[x, y] = \sum_{n=1}^N \sum_{m=1}^M f[n, m] g[x - n, y - m] \quad (2)$$

Figura 1: Convolución continua(1). Convolución discreta(2).

Para una imagen de tamaño $N \times M$ y un filtro de tamaño definido, la convolución corresponde sobreponer la máscara sobre la matriz de la imagen píxel a píxel, donde el resultado es la suma del producto entre la imagen y la máscara, como se muestra en la siguiente figura:

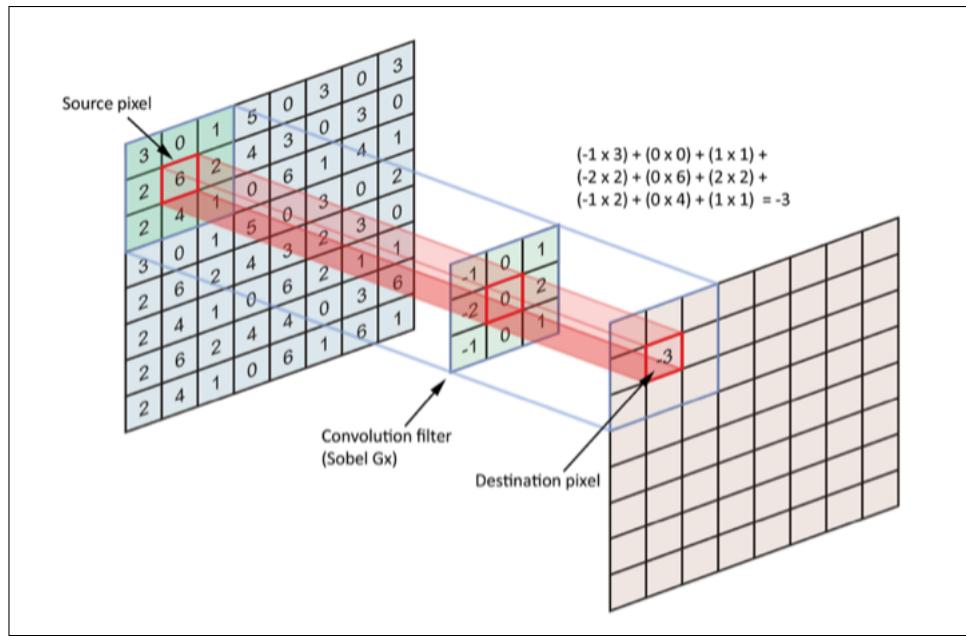


Figura 2: Proceso de convolución para un filtro específico en un punto $[x,y]$.

Dado a que la definición de la convolución, en el proceso se pierden píxeles en el borde dependiendo del tamaño del filtro. Hay diversas soluciones, como extraer los píxeles, dejarlos en negro, disminuir el tamaño, eliminando los píxeles faltantes o la solución empleada en esta tarea, repetir los píxeles originales en las orillas.

2.1.1. Detección de bordes

Para detectar bordes en una imagen, existen métodos que ocupan máscaras para ser convolucionada contra la imagen original.

Estas máscaras se basan en el cálculo de la gradiente existente en el cambio de iluminación de los píxeles en una escala (por ejemplo en grises).

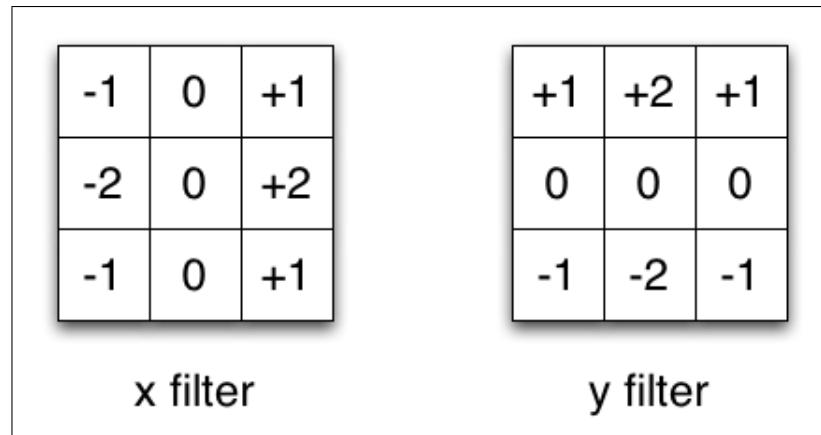


Figura 3: Filtros Sobel en x e y .

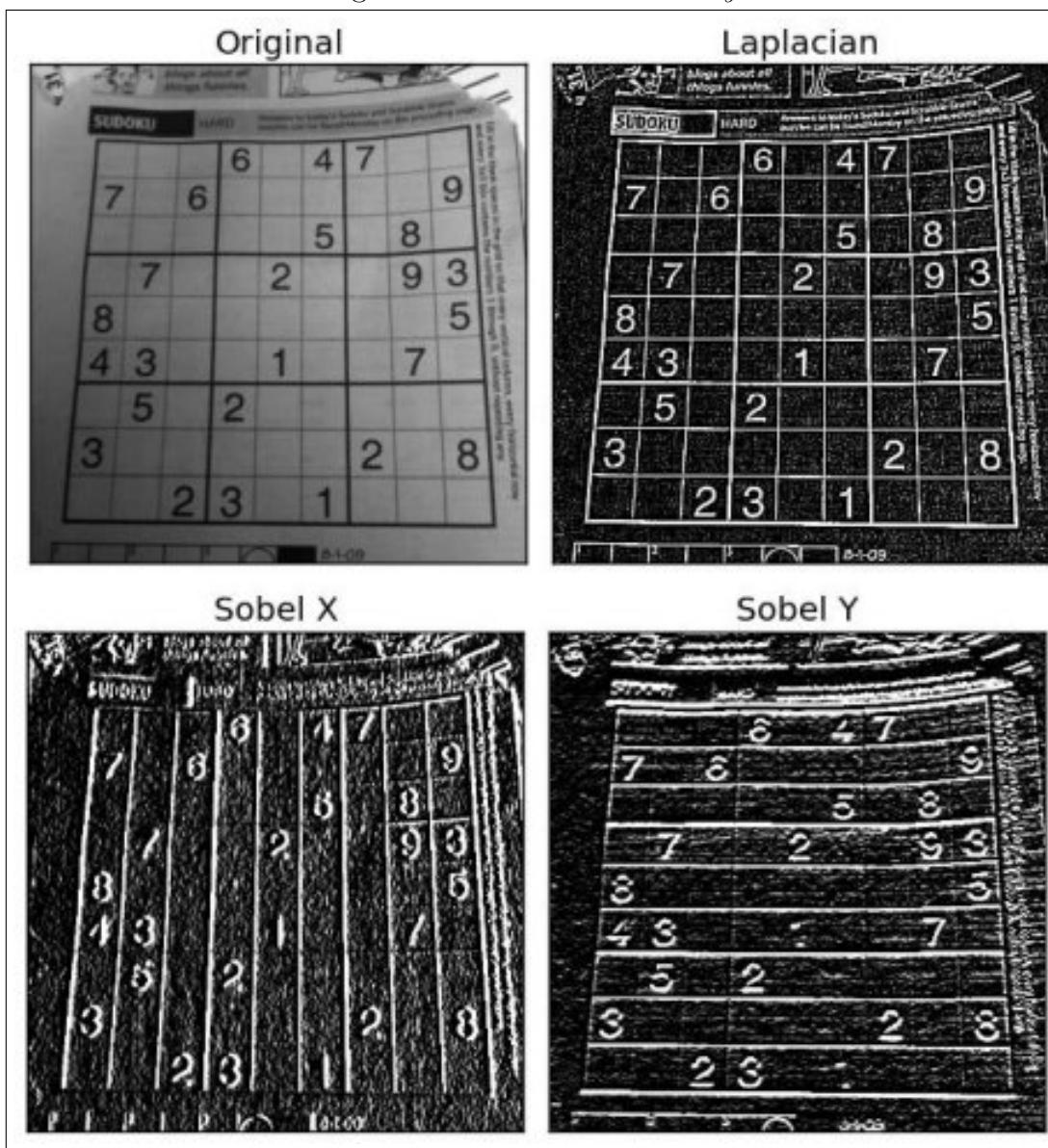


Figura 4: Ejemplo de detección de bordes para diferentes filtros.

2.2. Transformación de Homografía

Se denomina homografía a cualquier transformación proyectiva que establece una correspondencia entre dos formas geométricas, de modo que a un elemento, punto o recta, de una de ellas le corresponde otro elemento de la misma especie, punto o recta, de la otra, es decir, no hay cambio de naturaleza geométrica.

Las transformaciones homográficas son: la traslación, las simetrías, el giro, la homotecia, así como la homología. Esta operaciones podría ser en aspectos generales transformaciones de semejanza y afín.

2.2.1. Transformación de semejanza

Esta transformación se define por un cambio de escala, traslación y rotación en una imagen, la cual puede representarse de la siguiente manera:

$$\begin{pmatrix} u \\ v \end{pmatrix} = e \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Figura 5: Ecuación de transformación de semejanza para un punto (u,v) .

Es posible ver que tiene 4 parámetros libres. *Epsilon* corresponde al cambio de escala. *Theta* es el cambio en la rotación. Finalmente *t_x* y *t_y* corresponden en el cambio de posición en el plano.

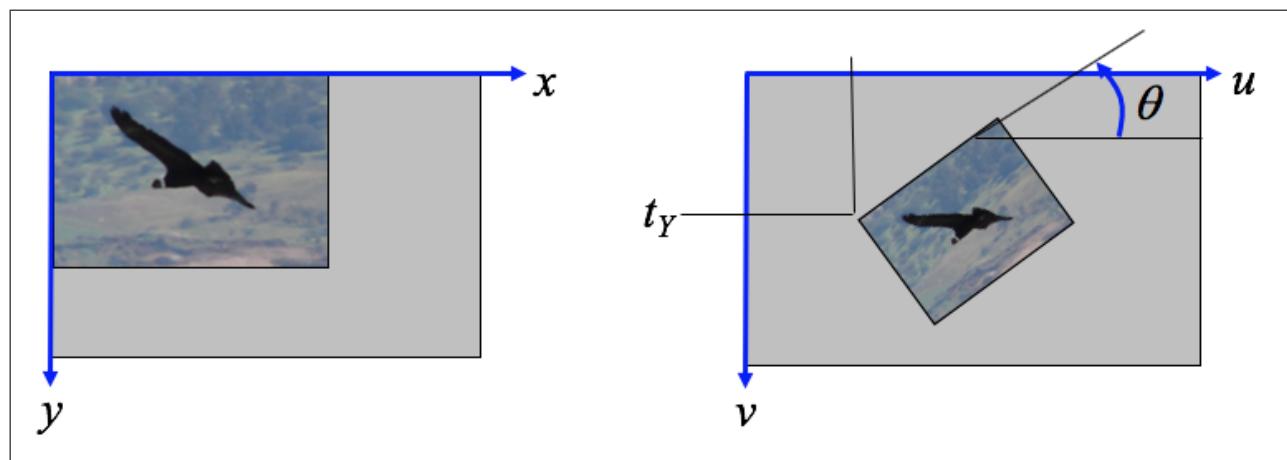


Figura 6: Ejemplo de transformación de semejanza.

2.2.2. Transformación afín

Esta transformación se define por un cambio de escala, traslación y rotación en una imagen. La diferencia con la transformación de semejanza es que esta soporta leves rotaciones de la imagen fuera del plano.

Esta transformación se puede expresar de la siguiente manera:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \end{pmatrix}$$

Figura 7: Ecuación de transformación afín para un punto (u,v) .

Es posible ver que tiene 6 parámetros libres, de los cuales t_x y t_y son el cambio de la posición (traslación).

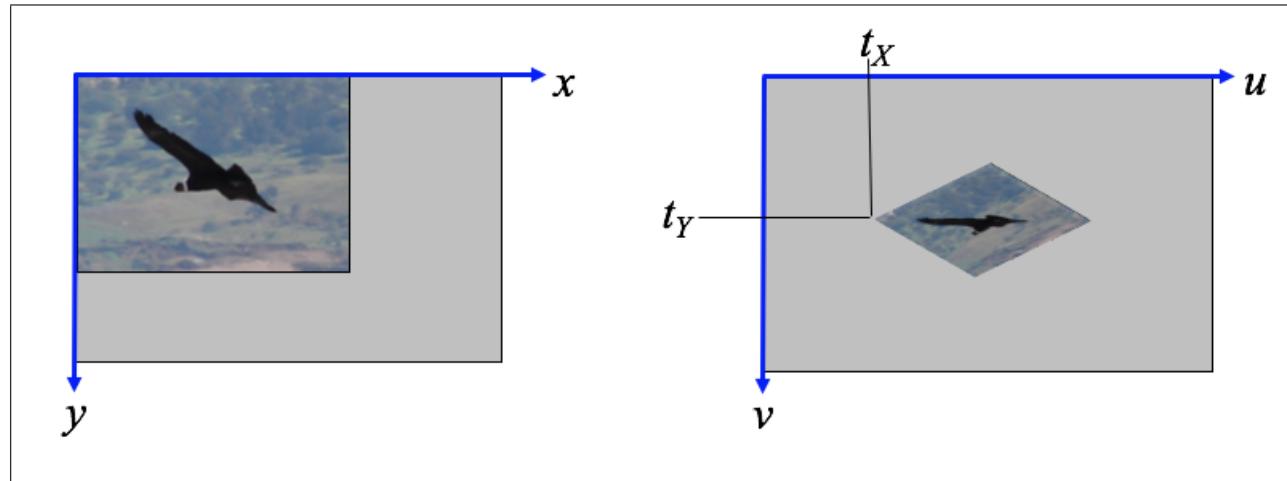


Figura 8: Ejemplo de transformación afín.

2.3. Red Neuronal

Una red neuronal corresponde a un sistema interconectado de perceptrones, las cuales están basadas en la biología.

Cada perceptrón es capaz de modelar una operación simple, basada en propios *pesos sinápticos* y función de activación.

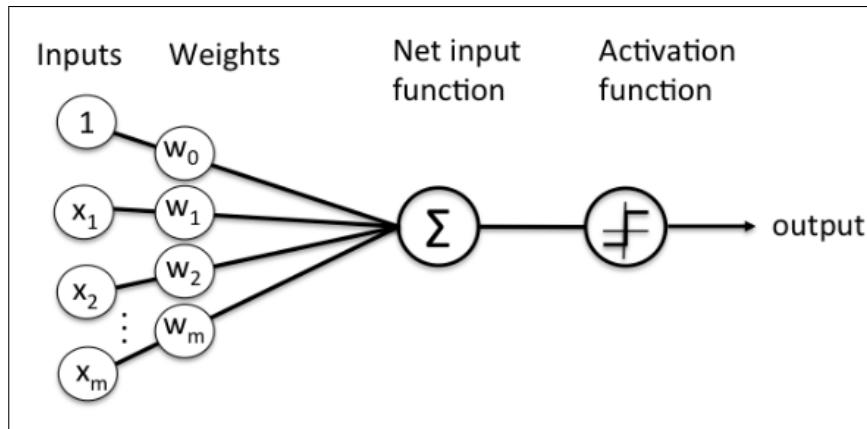


Figura 9: Modelo esquemático de una neurona o perceptrón

En la Figura 9, x_1 a x_m corresponden a la entrada de características por elemento en el vector de características. Desde w_1 a w_m corresponden a los pesos sinápticos, estos se multiplican con sus correspondientes elementos de entrada.

Existe el caso particular del *bias* que es una variable que no tiene una entrada del vector de características , por lo que se le puede considerar un pesos sináptico (w_0) con una entrada de 1, para que al ser multiplicada, mantenga su valor.

Cada peso multiplicado por su entrada correspondiente es sumado y esta suma pasa por una función de activación. Esta función debe ser elegida por el usuario, pues hay múltiples funciones posibles, tales como la función signo (simil a *SVM*), función sigmoideal o la función *ReLU* entre otras. La elección de esta función es importante para el rendimiento en una red neuronal.

Es necesario considerar que cuando se entrena una neurona, lo que se va modificando son los pesos sinápticos de ella.

Una red se compone de múltiples neuronas conectadas por capa.

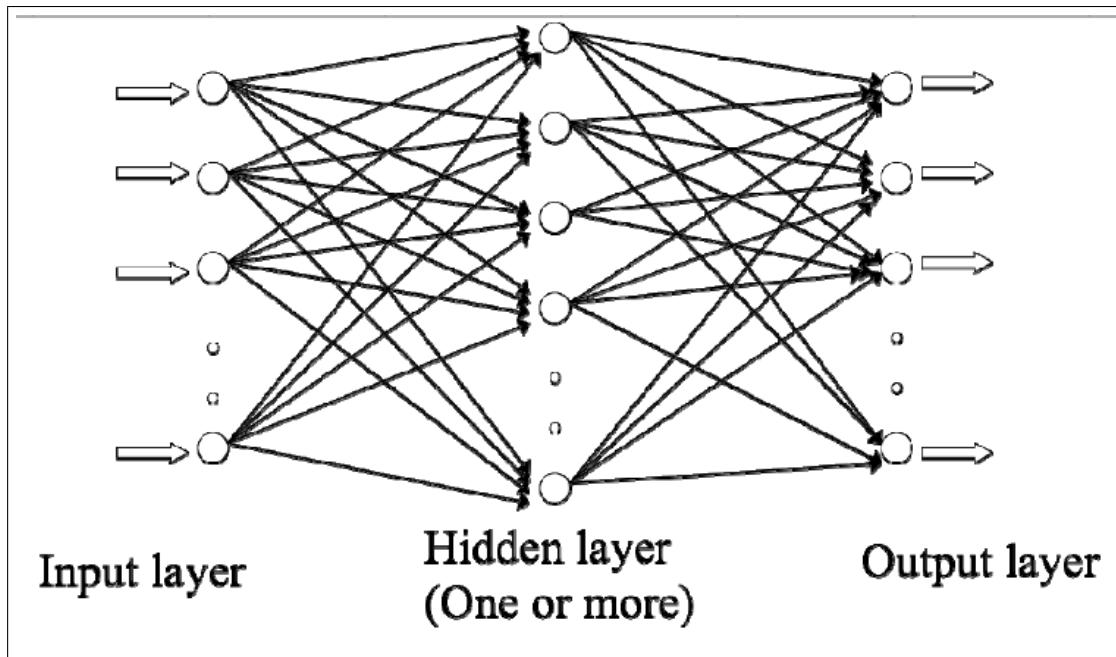


Figura 10: Multi-Layer Fully Connected Feedforward Network

La red de la Figura 10 se compone de 3 capas, una de entrada la cual toma los inputs, una capa oculta y otra de output la cual entrega la salida predicha por la red. Tanto la capa oculta como la de salida tienen neuronas, la capa de entrada solo son los elementos del vector de características. Es visualmente claro, los elementos de entrada van a todas las neuronas de la capa oculta y las salidas de estas van a la entrada de la siguiente capa (*fully-connected*).

Es posible crear redes con más capas ocultas, pero en general, solo es necesario una, dado que para más capas se necesitarían más ejemplos para entrenarla.

2.4. Redes Convolucionales

Una red convolucional es una red neuronal con muchas capas, en que en las primeras (mayor cantidad) realizan convoluciones (filtros) a la imagen de entrada y solo las últimas capas son fully-connected y son las encargadas de clasificar las características obtenidas en las capas convolucionales.

Las capas de una red convolucional se dividen en 3 tipos: capas convolucionales, capas de pooling (submuestreo) y capas fully-connected.

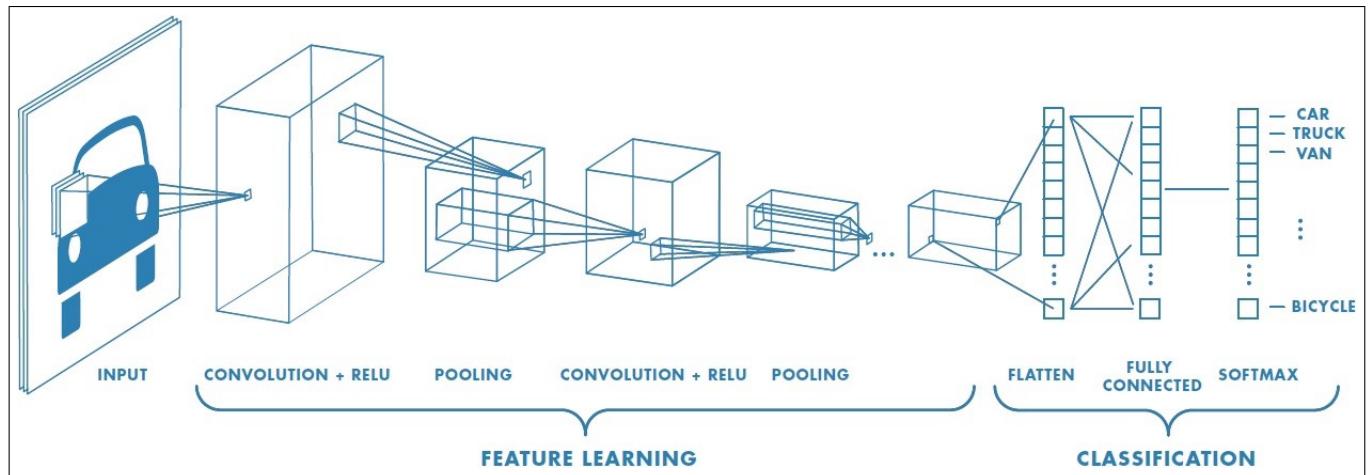


Figura 11: Esquema general de una red CNN.

En la Figura 11 es posible ver que cada bloque pertenece a operaciones de convolución y al momento de disminuir su tamaño se realiza una operación de pooling. Finalmente el vector de características obtenido entra a una red fully-conected que realiza la clasificación correspondiente.

Toda la red está formada por perceptrones, los cuales van ajustando sus pesos mediante técnicas de ajuste del error como el gradiente descendente estocástico, por lo que es posible entrenar y ajustar todos los pesos de la red en conjunto y no por separado.

Uno de los problemas de tener una red con muchas capas, es la cantidad de pesos por perceptrón, los cuales deben ajustarse, pero en una red CNN, las capas convolucionales tienen los pesos de un mismo filtro por capa, por lo que los pesos en las neuronas de una misma capa no son independientes, por lo que disminuye considerablemente el número de pesos a ajustar.

Solo en las últimas capas tipo fully-connected los pesos de los perceptrones son independientes.

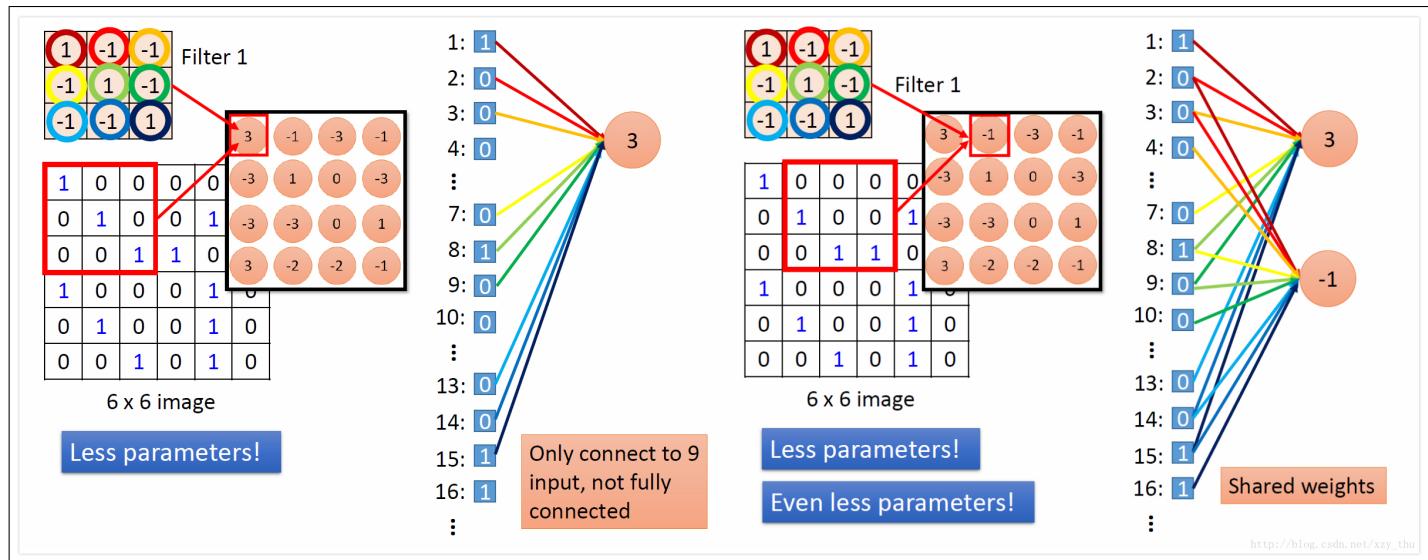


Figura 12: Ejemplo de capa convolucional.

En la Figura 12, se está representando la convolución dado un filtro en alguna capa convolucional. Cada peso es representado por un color dependiendo del valor en el filtro. Es posible ver que para diferentes perceptrones (ejemplo a la derecha), los pesos son los mismos y como entrada para cada perceprón, solo se tienen los valores de entrada en el espacio donde se está operando el filtro y no toda la imagen.

Otro problema para el entrenamiento de una red CNN, es el número de ejemplos necesarios para entrenar la red.

Hoy en día se tienen bases de datos de libre acceso particularmente grandes con respecto a clases específicas, lo que disminuye el problema de número de ejemplos para algunos problemas, no todos.

Para entrenar una red CNN se necesita mucho tiempo de cómputo dado la cantidad de ejemplos. Para este problema se tiene el avance en la evolución de CPU y arquitectura para GPU, además del tipo de entrenamiento con minibatch, lo que implica que los pesos no se actualizan por cada ejemplo, sino que junta el error por una cierta cantidad de ejemplos de entrenamiento y luego se actualizan los pesos. Además existen disponibles redes preentrenadas, por lo que solo basta obtenerlas y cargar la red, omitiendo el tiempo de entrenamiento.

Finalmente, existen múltiples arquitecturas de redes convolucionales, entre las más conocidas están: ResNet, AlexNet y VGG, cada una con una distribución distinta de capas y tamaño de imágenes de entrada.

2.4.1. Yolo (You Only Look Once)

Yolo es un tipo de red CNN, con la particularidad de además de clasificar un objeto determinar su posición en la imagen.

Al ingresar una imagen a la red esta se divide en una grilla. Por cada cuadro de la grilla se calcula la probabilidad de clase para ese cuadro y además de que sea un cuadro delimitador del objeto.

La salida de la red es una matriz donde se expresa las posiciones de los objetos y las clases predichas con su probabilidad. Esta salida se ocupa para recalcular los pesos y ajustar la red entera.

Lo que diferencia a Yolo es su capacidad de hacer todos los pasos una sola red CNN, en comparación a sus "competidores", las cuales pasan por un proceso para seleccionar posibles cuadros contenedores de algún objeto para luego pasar por una red CNN para su clasificación, lo que divide el proceso en 2 partes.

Cabe mencionar que el desempeño de Yolo corriendo en una GPU Titán X es de 45 FPS. Como no todos tienen acceso a esta GPU, existen variantes de Yolo (como tiny, micro o little Yolo) que pueden ser ejecutadas en hardware más accesible (GPU con 1 Gb), logrando ejecuciones más rápidas a cambio de precisión en la clasificación.

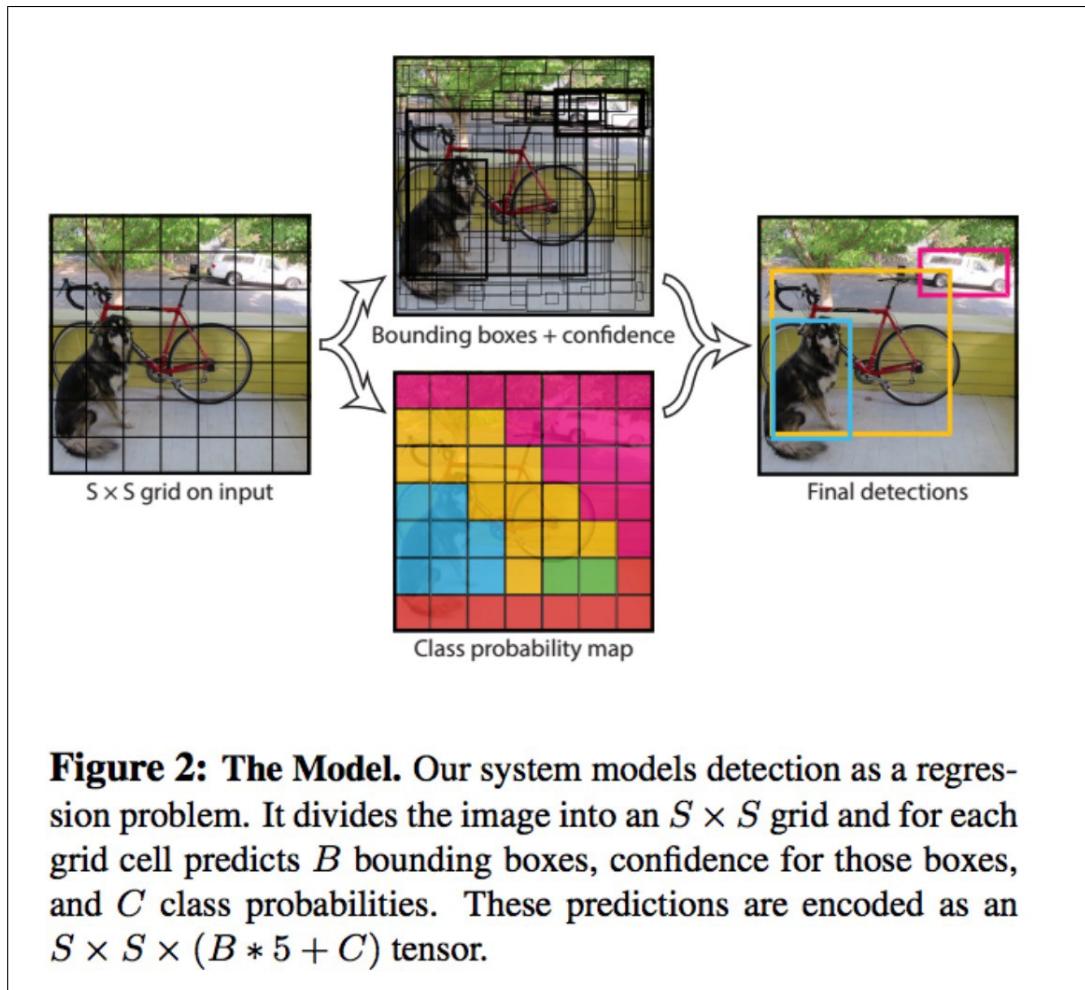


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

Figura 13: Representación del procesamiento de una imagen en Yolo.

En la Figura 13 se aprecia que la imagen de entrada pasa por una grilla de $S \times S$ a cada cuadro se le calcula la probabilidad de la clase a la que pertenece y además se calcula si este es borde de un objeto o no. Estos datos van juntos en un tensor de salida, donde se tiene la grilla $S \times S$, la cantidad de cuadros contenedores de objetos (B) las posiciones en la imagen original (5) y la clase que se predice (C).

3. Desarrollo

3.1. Requerimientos y paquetes necesarios

Para este trabajo, se realiza el desarrollo sobre un *notebook* con *Intel(R) Core™ i7-6700HQ CPU @ 2.60GHz* con *8Gb* de memoria RAM junto con una tarjeta gráfica *Nvidia(R) GeForce GTX 950M 4Gb*. El sistema operativo es *Ubuntu 18.04* de 64 bits.

La instalación de los paquetes se hacen para Python 3.6, por lo que se crea un ambiente virtual en Anaconda para operar los códigos necesarios.

```
1 $ conda create -n proyecto python=3.6.8 anaconda
```

La instalación de paquetes por consola para Anaconda se realiza de la siguiente manera:

```
1 $ conda install numpy
2 $ conda install cv2
3 $ pip3 install imgaug
```

Dada la instalación, se tienen las siguientes versiones instaladas:

- **Numpy:** 1.14.3
- **OpenCv:** 3.4.2
- **imageio:** 0.2.8

Cabe destacar que la implementación y documentación de *imageio* se encuentra en el siguiente *repositorio*.

Por otra parte también es necesario descargar la implementación de la red con la cual trabajar, en este caso *tiny-yolo*, la cual es la tercera versión de *Darknet*, clonada por git desde <https://github.com/pjreddie/darknet>.

También es necesario instalar *CUDA 10* para el aprovechamiento de la tarjeta gráfica en los procesos de entrenamiento y detección. Este software se puede descargar desde su desarrollador directo que es *Nvidia* desde el siguiente *enlace* la cual también entrega los pasos a seguir para su instalación.

Además es necesario tener instalado el último *driver* compatible con la tarjeta gráfica, el cual en este caso corresponde al *nvidia-418*. Este puede ser instalado en Ubuntu automáticamente con:

```
1 $ sudo ubuntu-drivers autoinstall
```

O puede instalarse de forma individual como:

```
1 $ sudo apt install nvidia-418
```

Para la compilación de la red, entrenamiento, validación y detección se usó la documentación del repositorio de *Darknet AlexeyAB*.

Finalmente, hay que recordar que antes de compilar, hay que activar el flag para CPU y CUDA en el *makefile* de *Darknet*:

```
1 GPU=1
2 CUDNN=0
3 OPENCV=0
4 OPENMP=1
5 DEBUG=0
```

3.2. Implementación

Para realizar el proyecto, se tiene como una base el paper *Few-shot Learning based on Context-aware Network Expansion with Artificial Training Data for Picking in Warehouse Automation* del laboratorio JSK de la Universidad de Tokio, en el cual se propone la creación de datos de entrenamiento para un robot pick-up.

En el paper se propone obtener los objetos, segmentarlos, agregarlos digitalmente a un contexto (en ese caso dentro de una caja) y aplicar *data augmentation* para aumentar la cantidad de ejemplos obtenidos, de esta manera crear una base de datos auto-generada para entrenar.

Tomando como ejemplo el paper, se proponen 4 etapas para la auto-generación de la base de datos que sirva para reentrenar *tiny-yolo*. Estos son: obtención de fotos de los objetos desde diferentes ángulos y su segmentación, ocupar data augmentation con transformaciones afines sobre los objetos segmentados, creación de escenarios con los objetos sobre un contexto (plano) y por ultimo aplicar data augmentation sobre los escenarios con diversos filtros.

3.2.1. Objetos y segmentación

Los objetos deben ser elementos de casa, en este caso: botellas, cajas de cereal, sobres de harina, latas, cajas de leche, tarros, vasos de yogur, bolsas de azúcar y cajas de té¹.

Las fotos fueron tomadas desde un iPhone SE, el cual tiene una cámara de 13 Mp.

Para obtener la segmentación de los objetos estos deben estar separados en un directorio como en la carpeta "*fotos*". Luego se debe correr el archivo *bordes.py* desde la línea de comandos como sigue:

```
1 $ python bordes.py "fotos" "directorio_salida"
```

En donde el *directorio_salida* corresponde a donde se guardarán las imágenes con los objetos segmentados. Luego de ejecutarse, además de la segmentación se obtendrá un archivo *obj.name* necesario para la etapa de reentrenamiento.

Dentro de *bordes.py* se encuentra la función *mascarasObjetos* definida como:

```
1 def mascarasObjetos(directorio, clase, c, out):
2     contador = 0
3     carpeta = os.listdir(directorio+"/"+clase)
4     for img in carpeta:
5         image = cv2.imread(directorio+"/"+clase+"/"+img)
6         print("\t" + img)
7         height, width, _ = image.shape
8         gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
9         blurred_x = cv2.GaussianBlur(gray, (3, 3), 0)
10        blurred3 = cv2.resize(gray,(int(width/2),int(height/2)))
11        blurred2 = cv2.resize(blurred3,(int(width/4),int(height/4)))
12        blurred = cv2.resize(blurred2,(int(width/6),int(height/6)))
13
14        # Detección de bordes con canny
15        auto = auto_canny(blurred)
16        edges = cv2.dilate(auto, None, iterations=3)
17        edges = cv2.resize(edges,(int(width/8),int(height/8)))
18
19        auto2 = auto_canny(blurred2)
20        edges2 = cv2.dilate(auto2, None, iterations=3)
21        edges2 = cv2.resize(edges2,(int(width/8),int(height/8)))
22
23        auto3 = auto_canny(blurred3)
24        edges3 = cv2.dilate(auto3, None, iterations=3)
25        edges3 = cv2.resize(edges3,(int(width/8),int(height/8)))
```

¹Es necesario aclarar que los 3 últimos objetos están en la base de datos, pero no participan en los la generación de los datos de entrenamiento, pues sus ejemplos son pocos (4) en promedio de los demás objetos (10)

```
26
27
28     # Consolidacion de bordes
29     edges = edges + edges2 + edges3
30
31     edges = cv2.resize(edges,(int(width/8),int(height/8)))
32     image = cv2.resize(image,(int(width/8),int(height/8)))
33
34     # Busqueda del contorno mas grande
35     max_contour = maxContorno(edges)
36
37     # Creacion de la mascara
38     mask = np.zeros(edges.shape)
39     cv2.fillConvexPoly(mask, max_contour[0], (255))
40     mask_stack = np.dstack([mask]*3)
41
42     # Obtencion de objeto segmentado
43     mask_stack = mask_stack.astype('float32')/255.0
44     image = image.astype('float32')/255.0
45     masked = (mask_stack * image) + ((1-mask_stack) * MASK_COLOR)
46     masked = (masked * 255).astype('uint8')
47
48     # Guardar elementos segmentado
49     cv2.imwrite(out+"/"+c+"_"+str(contador)+".jpg", masked)    # Save
50     contador += 1
```

En la función se puede ver como se itera sobre todas las imágenes para una clase de objeto, pasando a escala de grises y siendo filtrado por una pequeña gaussiana, luego se obtienen 3 imágenes reducidas en la mitad, el cuarto y el sexto de la imagen original. A cada una se les calculan los bordes con el método Canny, el resultado es dilatado y ajustado a un octavo de la imagen original.

Los tres cálculos de bordes son sumados y se encuentra el contorno convexo más amplio, el cual correspondería al objeto a segmentar.

Luego se crea la máscara y la imagen es filtrada en sus 3 canales, dejando lo exterior a la máscara con un fondo negro ([0,0,0]) el cual es la variable global *MASK_COLOR*.

Finalmente se guarda en el directorio de salida y como nombre tiene el código de su respectiva clase seguida por el número que la identifica dentro de su propia clase.

Otro código importante dentro de este archivo es *maxContorno*:

```
1 def maxContorno(lados):
2     contornos_info = []
3     contornos, hierarchy = cv2.findContours(lados, cv2.RETR_TREE, cv2.←
4         CHAIN_APPROX_SIMPLE)
5     for c in contornos:
6         contornos_info.append((
7             c,
8             cv2.isContourConvex(c),
9             cv2.contourArea(c),
10            ))
11    contornos_info = sorted(contornos_info, key=lambda c: c[2], reverse=True)
12    return contornos_info[0]
```

A muy grandes rasgos es una función que recibe los bordes y obtiene por medio de `findContours` todos los contornos encontrados. Luego estos son agregados a un arreglo para luego ser ordenados de mayor a menor área, entregando finalmente el contorno más grande encontrado.

3.2.2. Data Augmentation

El proceso de data augmentation corresponde a aumentar el número de ejemplos por medio de aplicación de transformaciones y convoluciones a los datos ya obtenidos. Para esta parte tenemos 2 tipos, data augmentation con transformaciones afines aplicadas a los objetos y filtros aplicados a las escenas (objetos sobre un contexto).

Se tiene el archivo *daugmentation* el cual se puede ejecutar desde la línea de comandos como sigue:

```
1 $ python daugmentation.py "directorio" 1 n_ejemplos
```

Este comando es para obtención de data augmentation con transformaciones a fines y el que sigue para aumentar los datos por medio de filtros:

```
1 $ python daugmentation.py "directorio" 0 n_ejemplos
```

En el caso de los comandos, "*directorio*" corresponde a la ubicación de la data y *n_ejemplos* corresponde al número de imágenes que se desea obtener por cada filtro.

Las transformaciones afines corresponden a *scale*, *rotate* y *shear*. Mientras los filtros corresponden a *add*, *dropout* (ruido pimienta) y *contrast normalition*. Todos estos son obtenidos desde *imgaug* y son agrupados en arreglos globales.

La función encargada de ejecutar el aumento de los datos es *aumentarData*:

```
1 def aumentarData(dir_in, transformaciones, contador, iterador, copia = False):
2     for dau in transformaciones:
3         carpeta = os.listdir(dir_in)
4         for img in carpeta:
5             if img[-1] == 'g':
6                 foto = img.split('.')[0]
7                 image = cv2.imread(dir_in+"/"+img)
8                 for count in range(iterador):
9                     image_au = dau.augment_image(image)
10                    cv2.imwrite(dir_in+"/"+foto+"_"+str(contador)+"_"+str(count)+".jpg", image_au)
11 # Copia de coordenadas para Yolo
12                    if copia:
13                        with open(dir_in+"/"+foto+".txt", 'rb') as forigen:
14                            with open(dir_in+"/"+foto+"_"+str(contador)+"_"+str(count)+".txt", 'wb') as fdestino:
15                                shutil.copyfileobj(forigen, fdestino)
16
    contador += 1
```

Se puede apreciar que itera primeramente sobre las transformaciones y luego por las imágenes, las imágenes conservan su nombre original, pero se adiciona el numero de contador más *count* el cual indica que número de la repetición sobre la transformación.

Tener en consideración que los datos aumentan en ley de potencia, es decir, si solo se tiene 1 imagen y se le aplican 3 transformaciones en donde por cada una se requieren 4 ejemplos, como resultados se tendrán, 5 imágenes en la primera transformación, 25 en la segunda y 125 en la tercera. Por lo que para este proyecto se aumentaron los datos en 3 imágenes por transformación afín (x 64) y en 2 para las transformaciones sobre las escenas (x 27).

En el caso que sea una transformación sobre los escenarios, se requiere duplicar el archivo *.txt* vinculado a la escena el cual contiene la posición de cada objeto y su tamaño relativo a la imagen, de ahí el código que sigue de la línea 12 del código.

Luego de obtener la segmentación de los objetos es necesario obtener el cuadro contenedor, esto se hace en la función *obtenerRecuadro*:

```
1 def obtenerRecuadro(dir_in):
2     carpeta = os.listdir(dir_in)
3     for img in carpeta:
4         if img[-1] == 'g':
5             foto = img.split('.')[0]
6             image = cv2.imread(dir_in+"/"+img)
7             gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8             auto = auto_canny(gray)
9             lados = cv2.dilate(auto, None, iterations=1)
10            max_contorno= maxContorno(lados)
11            x,y,w,h = cv2.boundingRect(max_contorno[0])
12            rec= image[y:y+h , x:x+w]
13            cv2.imwrite(dir_in+"/"+img, rec)
```

Como se ve, esta función opera sobre todas las imágenes de un directorio. El tratamiento es similar a segmentar un objeto. La imagen se transforma a escala de grises, se obtienen los bordes para luego conseguir el contorno más grande.

Con el contorno se puede obtener la posición y el tamaño de cuadro contenedor del contorno. Con los valores obtenidos se corta la imagen y se reescribe.

3.2.3. Contexto

Para poder completar los escenarios se necesitan imágenes que simulen el contexto, es este caso son imágenes de planos desde distintos ángulos. Por lo que se sacaron fotos a una cartulina amarilla como plano como se ve en la Figura 14 los cuales fueron segmentados con la ayuda de Power Point.

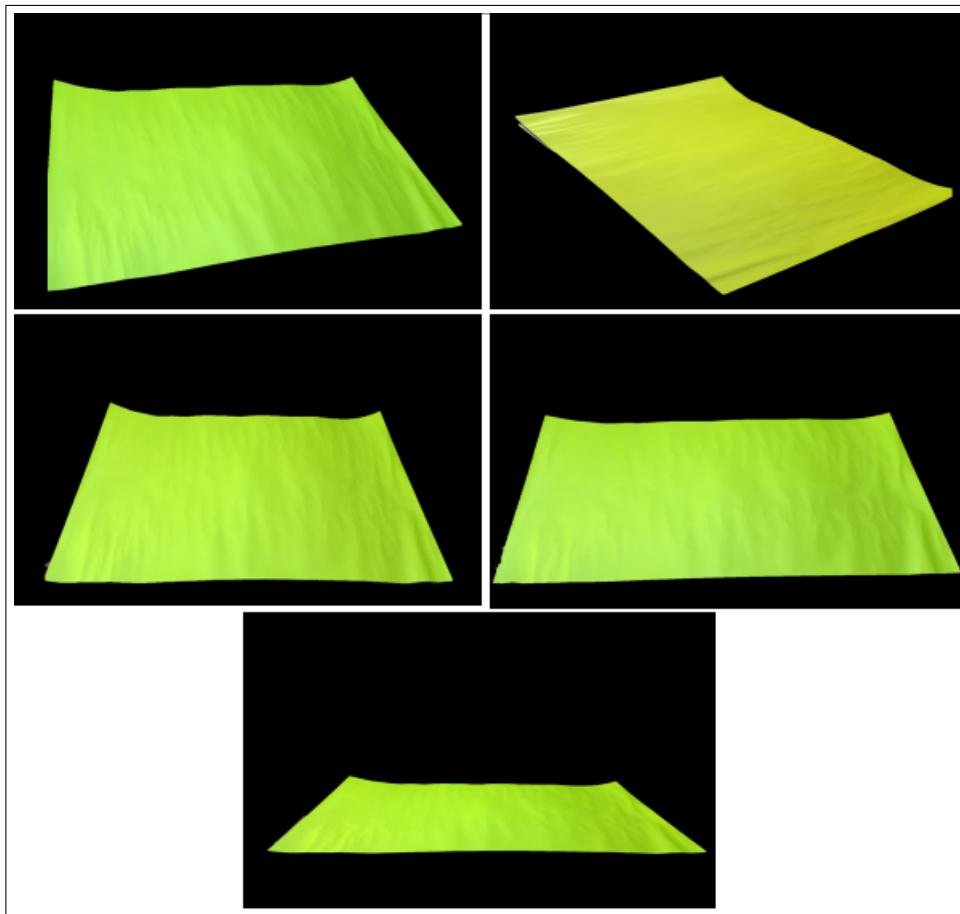


Figura 14: Planos obtenidos de forma manual.

Al avanzar en la implementación, se percató de que los planos no deberían obtenerse de manera manual y que además deberían tener texturas, por lo que se procedió a programar un pequeño archivo *contexto.py* como sigue:

```
1 if __name__ == '__main__':
2     textura = cv2.imread(sys.argv[1])
3     textura = cv2.resize(textura,(3000,1900))
4     mask = np.zeros([3000, 4000, 3])
5     blanco = np.ones([1900, 3000,3])
6     mask[600:2500,500:3500]= blanco*textura
7     cv2.imwrite("ejemplo.jpg",mask)
```

```
8      rows,cols, _ = mask.shape
9
10     pts1 = np.float32([[600,500],[600,3500],[2500,500]])
11     pts2 = np.float32([[1100,2000],[100,3000],[3000,2000]])
12     pts3 = np.float32([[50,2000],[1100,3000],[2200, 2000]])
13     pts4 = np.float32([[200,2000],[1500,3000],[2000,1500]])
14
15     M = cv2.getAffineTransform(pts1,pts2)
16     dst = cv2.warpAffine(mask,M,(cols,rows))
17     cv2.imwrite("contexto1.jpg",dst)
18
19
20
21     M2 = cv2.getAffineTransform(pts1,pts3)
22     dst = cv2.warpAffine(mask,M2,(cols,rows))
23     cv2.imwrite("contexto2.jpg",dst)
24
25     M3 = cv2.getAffineTransform(pts1,pts4)
26     dst = cv2.warpAffine(mask,M3,(cols,rows))
27     cv2.imwrite("contexto3.jpg",dst)
28
29     pts1 = np.float32([[600,500],[600,3500],[2500,500],[3000,3500]])
30     pts2 = np.float32([[1000,1500],[-8000,18000],[2000,1500],[8000,18000]])
31     pts3 = np.float32([[2000,1000],[-500,3000],[3400, 1500],[1200,3500]])
32
33     M = cv2.getPerspectiveTransform(pts1,pts2)
34     dst = cv2.warpPerspective(mask,M,(cols,rows))
35     cv2.imwrite("contexto4.jpg",dst)
36
37     M = cv2.getPerspectiveTransform(pts1,pts3)
38     dst = cv2.warpPerspective(mask,M,(cols,rows))
39     cv2.imwrite("contexto5.jpg",dst)
```

En el script se muestra que es necesario ingresarle una imagen con la textura deseada luego se genera un plano una matriz de OpenCV, la cual pasa por transformaciones afines y transformaciones de perspectiva según como se han movido los puntos elegidos, los cuales corresponden a los vértices del plano.

Una vez ejecutado el código se obtienen los planos expuestos en la Figura 15:

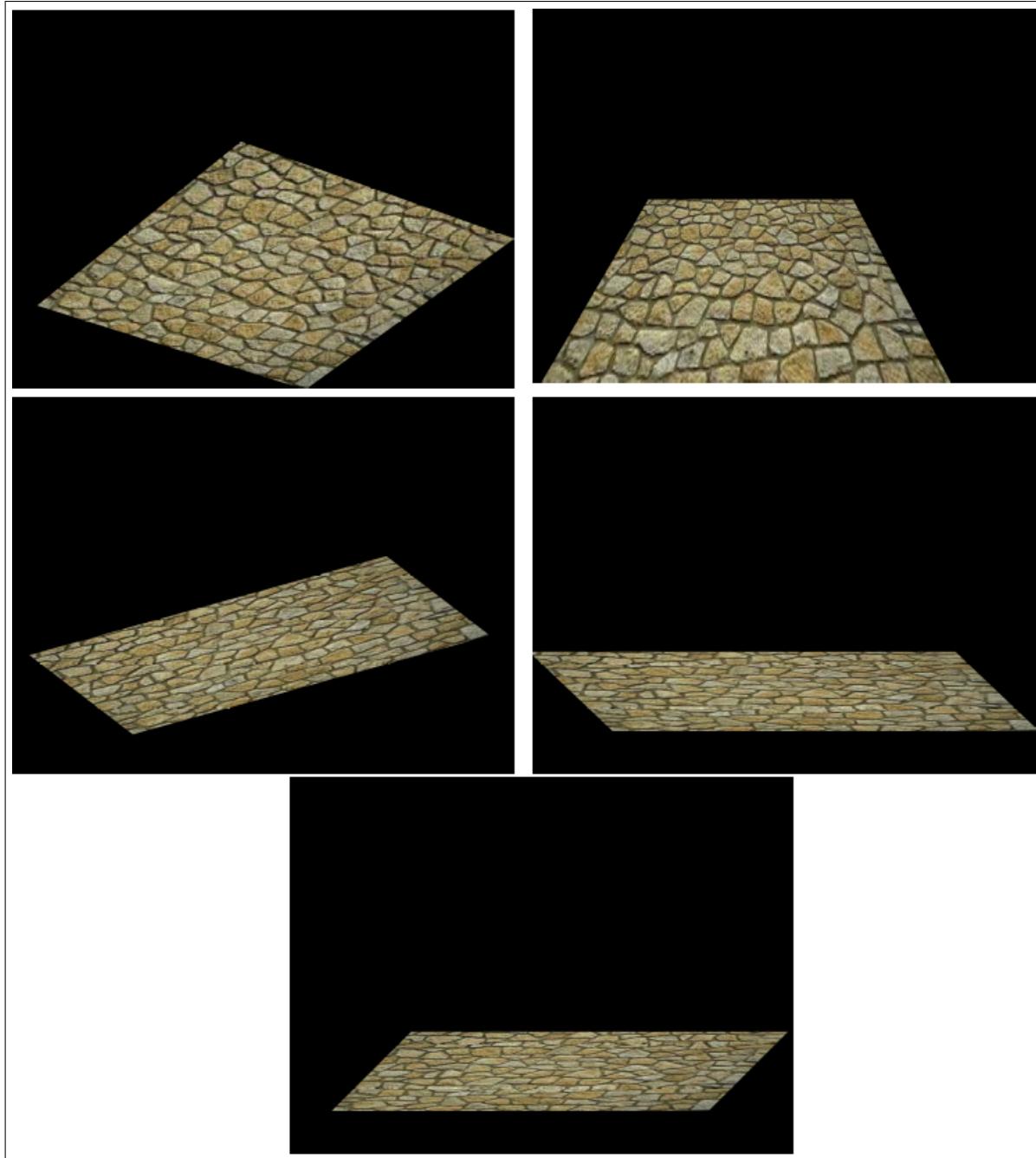


Figura 15: Planos obtenidos de forma manual.

Al momento de la implementación de este código, el trabajo había continuado con los contextos obtenidos de forma manual (pues fue la primera idea que se tuvo), siendo muy tarde para volver a probar con planos texturizados sobre las escenas, por lo que queda propuesto para un trabajo futuro la selección y prueba sobre diferentes texturas para las escenas.

3.2.4. Escenarios

Para poder (re)entrenar *tiny-yolo* es necesario que cada imagen este en formato *jpg* y conste con un archivo *.txt* con el mismo nombre, en el cual contenga por una línea por cada objeto en la imagen siguiendo el siguiente formato:

```
1 <object-class> <x> <y> <width> <height>
```

En donde *object-class* corresponde al valor numérico asignado por cada clase dependiendo de la posición en el archivo *obj.names*. Los demás atributos corresponden a la posición del punto central del objeto y su tamaño con respecto a la imagen, por lo que sus valores van desde 0 a 1.

Se tiene el archivo *escenas.py* el cual se puede ejecutar en la línea de comandos de la siguiente forma:

```
1 $ python escenas.py "contexto" "objetos" min_obj max_obj
```

Dentro del comando, *contexto* se refiere a la dirección del directorio con los contextos, al igual que *objetos* en el directorio en el que se encuentren.

Por otra parte *min_obj* y *max_obj* corresponden al número mínimo y máximo de objetos que se quieren obtener. Para el proyecto se obtuvieron escenas desde 0 objetos hasta 5.

La implementación para la generación de escenas contiene 2 funciones principales, la primera es *creador*:

```
1 def creador(dir_contextos, dir_objetos, min_elementos, max_elementos, salida=←
  " Data "):
2     os.mkdir(salida)
3     escenas = os.listdir(dir_contextos)
4     objetos = os.listdir(dir_objetos)
5     n_objetos = len(objetos)
6     contador = 0
7     #Iteracion por contexto
8     for contexto_x in contextos:
9         contexto_original = cv2.imread(dir_contextos + "/" + contexto_x)
10        #Iteracion por numero de objetos
11        for n_elementos in list(range(min_elementos, max_elementos+1)):
```

```
12     if n_elementos == 0:
13         n_combinaciones = 1
14     else :
15         n_combinaciones = 200
16     #Iteracion por combinaciones
17     for alpha in list(range(n_combinaciones)):
18         escena = contexto_original.copy()
19         archivo = open(salida + "/" +str(contador)+".txt",'a')
20         # Insercion de objetos
21         for i in list(range(n_elementos)):
22             c = random.randrange(numero_clases)
23             pos = random.randrange(n_objetos)
24             while int(objetos[pos].split("_")[0]) != c:
25                 pos = random.randrange(n_objetos)
26             objeto = cv2.imread(dir_objetos + "/" + objetos[pos])
27             clase = objetos[pos].split("_")[0]
28             insertar(escena, objeto, clase, archivo)
29             cv2.imwrite(salida + "/" +str(contador)+".jpg",escena)
30             archivo.close()
31             contador += 1
```

Esta función es básicamente es un iterador sobre los contextos. Por cada contexto se realizan 200 ejemplos a menos que el número de objetos por ejemplo sea 0, en ese caso el escenario es un contexto vacío por lo que no se itera (no se quieren repeticiones de escenas vacías).

Una vez dentro de la iteración se hace una copia del contexto original, la cual ya pasa a ser una escena en un principio vacía con su respectivo archivo *.txt*.

Luego se entra a la iteración por la inserción de elementos en la imagen, en donde primero se elige aleatoriamente una clase y luego se elige aleatoriamente un objeto, el cual se revisa si es la clase que se quiere, entrando en un loop hasta que el objeto seleccionado sea correspondiente a la clase elegida, de esta manera todas las clases tienen la misma probabilidad de aparecer en la escena no importando el número de ejemplos que tenga.

Para lograr comparar un objeto con su clase, hay que recordar que el primer carácter del nombre de la imagen de un objeto corresponde a su clase (subsección de segmentación y *data augmentation*).

Una vez seleccionado correctamente el objeto, este se inserta en la escena con la función *insertar*.

Al finaliza la iteración de inserción, se guarda la escena, se cierra su archivo *.txt* y se aumenta en 1 el contador para otorgarle un nombre a la siguiente escena.

Como ya fue mencionada, la segunda función importante es *insertar*:

```
1 def insertar( escena , objeto , clase , archivo):
2     alto_o , ancho_o , x = objeto.shape
3     alto_e , ancho_e , x = escena.shape
4     x = random.randrange(alto_e)
5     y = random.randrange(ancho_e)
6
7     # Elegir un punto dentro de la escena
8     while np.linalg.norm(np.array(escena[x][y])-fondo) < 9:
9         x = random.randrange(alto_e)
10        y = random.randrange(ancho_e)
11
12    # Inicializacion de posiciones
13    inicio_x = x - int(alto_o / 2)
14    inicio_y = y - int(ancho_o / 2)
15    inicio_x_escena = 0
16    inicio_y_escena = 0
17
18    # Valores para .txt Yolo
19    width = ancho_o / ancho_e
20    height = alto_o / alto_e
21    x_yolo = y/ ancho_e
22    y_yolo = x/ alto_e
23    archivo.write(clase+" "+str(x_yolo)+" "+str(y_yolo)+" "+str(width)+" "+str(
24                                height)+"\n")
25
26    #Seteo de valores en x
27    if inicio_x < 0:
28        inicio_x_escena = 0
29        inicio_x = inicio_x * -1
30    else:
31        inicio_x_escena = inicio_x
32        inicio_x = 0
33
34    #Seteo de valores en y
35    if inicio_y < 0:
36        inicio_y_escena = 0
37        inicio_y = inicio_y * -1
38    else:
39        inicio_y_escena = inicio_y
40        inicio_y = 0
41
42    # Iteracion sobre la escena y el objeto
43    count_x = 0
44    for h in list(range(inicio_x,alto_o)):
45        if count_x + inicio_x_escena >= alto_e :
46            break
47        count_y = 0
48        for w in list(range(inicio_y , ancho_o)):
```

```

49         if count_y + inicio_y_escena >= ancho_e :
50             break
51         if np.linalg.norm(np.array(objeto[h][w])-fondo) > 9:
52             escena[count_x + inicio_x_escena][count_y + inicio_y_escena] = ←
53             objeto[h][w]
54             count_y += 1
      count_x += 1

```

Esta función calcula las dimensiones de la escena y del objeto, para luego elegir un punto aleatorio con la condición que este no este en el fondo, si no, en el plano. Luego se inicializan las posiciones absolutas del objeto sobre el escenario (pudiendo dar posiciones fuera del escenario).

Siguiendo, se obtienen las posiciones relativas del objeto con respecto al escenario y se escribe en el archivo la línea correspondiente al objeto.

Luego se procede a inicializar los valores tanto de la escena como los del objeto sobre los cuales iterar, por ejemplo puede que el objeto no este contenido en su totalidad dentro de la escena, por lo que sus límites de iteración deben adecuarse.

Finalmente se itera sobre el objeto y la escena, para copiar la primera sobre la segunda, teniendo cuidado que la iteración no se salga de la escena y que se copie solo el objeto y no su fondo (negro).

Una vez obtenidos las escenas, se aplica *data augmentation* con los parámetros ya descritos en su subsección correspondiente.

3.2.5. Partición de los datos

Tiny-yolo necesita 2 archivos que indiquen cuales serán los datos de entrenamiento y cuales de validación. Para esto se tiene el archivo *particionData* con la función del mismo nombre definida como sigue:

```

1 def particionData(directorio, salida ,porcentaje_entrenamiento, ←
2     porcentaje_total):
3     carpeta = os.listdir(directorio)
4     carpeta = np.random.permutation(carpeta)
5     largo = len(carpeta)
6     maximo = int ((largo/2) * porcentaje_total)
7     maximo_entrenamiento = maximo *porcentaje_entrenamiento
8     train = open("train.txt", 'a')
9     valid = open("test.txt", 'a')
10    contador = 0
11    for image in carpeta:
12        if image[-1] == "g":
13            if contador < maximo_entrenamiento:
14                train.write(salida+"/"+image+"\n")

```

```
14     elif contador < maximo:  
15         valid.write(salida+"/"+image+"\n")  
16     else:  
17         break  
18     contador+=1
```

Aquí se muestra como se abre el directorio donde están las escenas y se realiza una permutación aleatoria de los archivos que se encuentran.

Luego se marca una máximo de elementos (si es que no se quiere ocupar todos), y luego el número de elementos a ocupar en entrenamiento.

Finalmente se itera sobre los elementos del directorio de entrada dejando los primeros para entrenamiento hasta su tope y el resto en validación.

La salida debe corresponder a la dirección donde se encontrarán dentro del repositorio de *Darknet* al momento del entrenamiento.

3.3. Resultados y análisis

3.3.1. Segmentación

Como ya se mencionó, para trabajar se consideraron 5 clases. Dentro de las clases habían en su mayoría objetos con geometría poligonal (cajas de cereal y leche) y otras más redondas redondas (botellas, latas y envases), como se muestra en la Figura 16, la segmentación para cajas funciona bien, dejando solo un fino borde del contorno. En cambio en la Figura 17 podemos ver que la sombra en ciertas zonas aumenta la segmentación del primer objeto que se muestra (tarro).



Figura 16: Segmentación de cajas. A la izquierda el original y la derecha segmentado.



Figura 17: Segmentación objetos con curvas o irregulares. A la izquierda el original y la derecha segmentado.

Cabe destacar que los fondos solo corresponden a cartulinas que contrastaran con el color del objeto y que estuvieran iluminadas uniformemente lo mejor que se pudiera.

Objetos con fondos con texturas, color similar al objeto o mucha sombra no pueden ser segmentadas correctamente bajo esta implementación.

3.3.2. Data Augmentation

En esta parte no hay nada que discutir frente a las imágenes obtenidas, puesto que se obtiene lo que se pide como se muestran en la Figura 18 y 19. Pero queda la discusión sobre cuantos ejemplos pedir tanto en las transformaciones a fines como en los filtros. Además de que filtros son convenientes o no, por ejemplo, ocupar filtro gaussiano o mediana, o ambos; ocupar más filtros para obtener más ejemplos o quedarme con pocos filtros pero representativos, son algunas de las disyuntivas encontradas.

Para este trabajo, no se pidieron muchos ejemplos por transformación, puesto que al intentar la primera vez con 10 ejemplos por cada iteración se encontró que la mayoría era muy parecido, rotaciones casi iguales, escalamientos absurdamente pequeños entre otros.

Al momento de elegir muchos filtros y ejecutar el código el computador se ponía excesivamente lento. Una vez que terminaba la ejecución, la memoria RAM estaba repleta, por lo que cualquier otro proceso a ejecutar se realizaba lentamente aún después de reiniciar el equipo.

Por lo anterior, se decidió dejar 3 filtros que tuvieran que ver con el color para aumentar los datos de las escenas, esos fueron adición aleatoria en algún canal, ruido pimienta y cambio de contraste, los cuales se piensan son un poco mas representativos de la realidad. Hay que recordar que estas transformaciones se superponen por lo que debe haber una escena con estos 3 filtros a la vez.



Figura 18: Objeto original a la izquierda y ejemplos de sus transformaciones afines a la derecha.

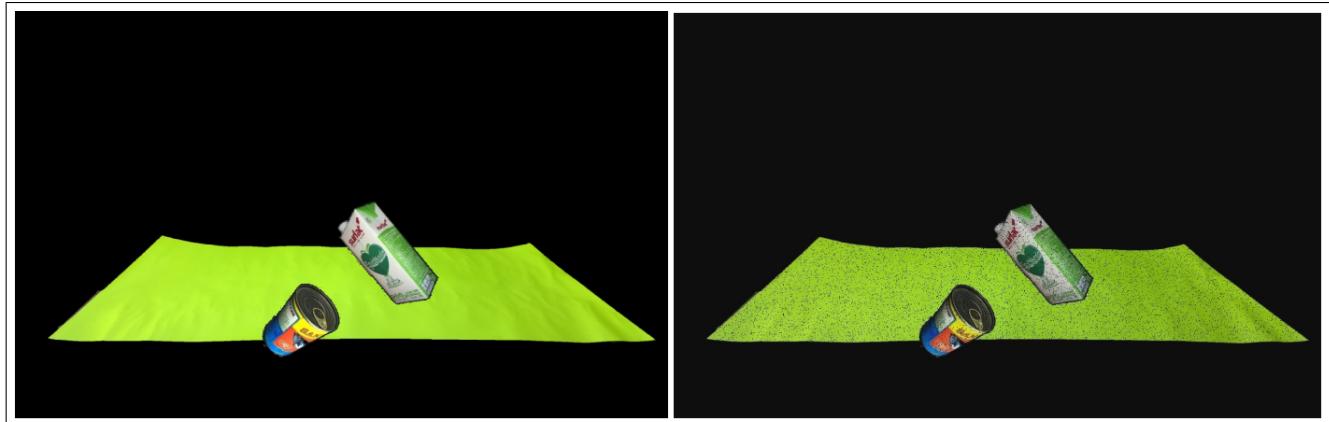


Figura 19: Escena con dos elementos original a la izquierda y con los tres filtros a la derecha.

3.3.3. Escenas y entrenamiento

En el caso de las escenas se puede observar en la Figura 20 ejemplos de escenas que aunque no parecen muy reales si reflejan de alguna manera objetos sobre un plano.

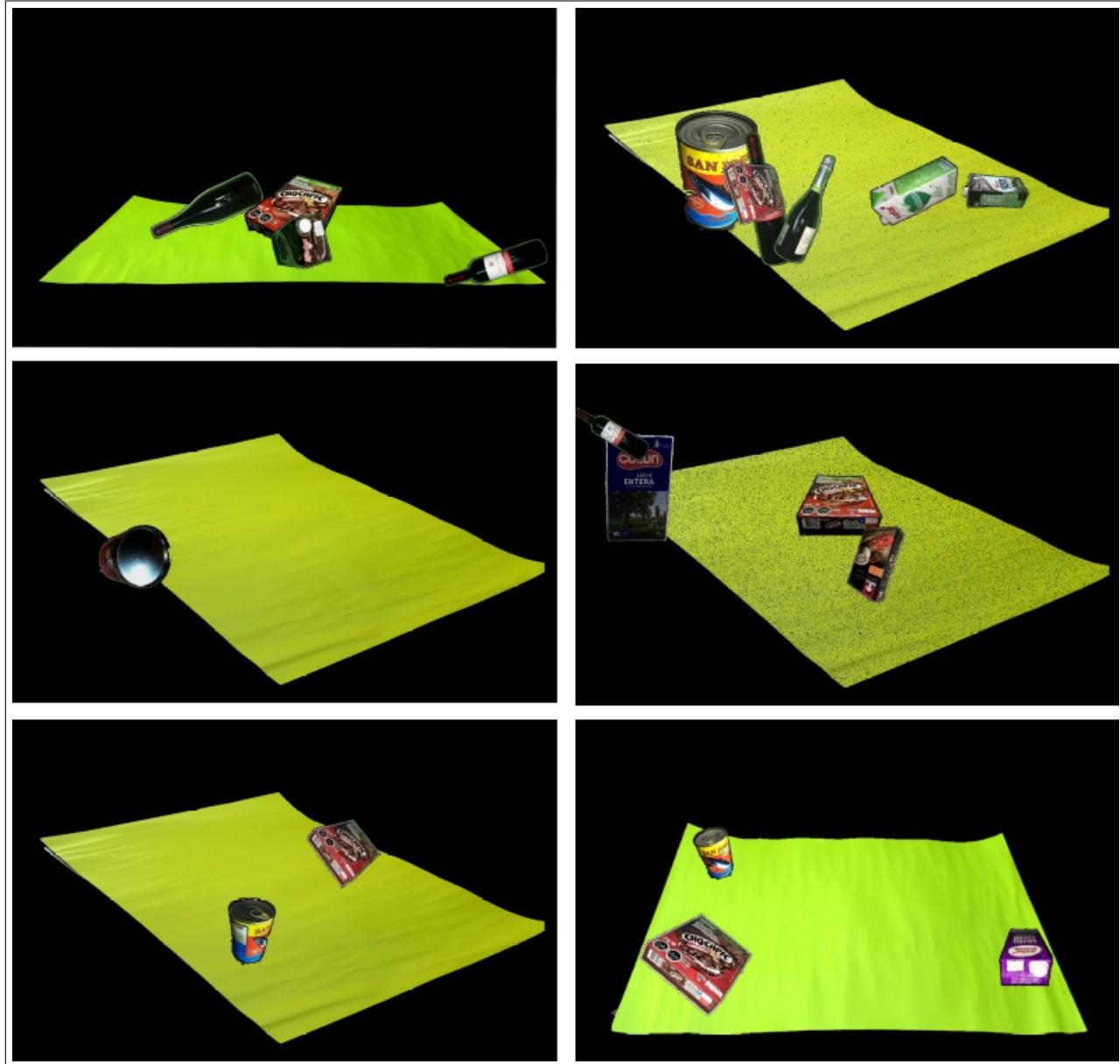


Figura 20: Ejemplo de escenas de entrenamiento.

El reparo que puede realizarse es que no se varió el contenido del contexto en textura, el cual puede afectar a la generalización al momento de entrenar.

Otro contra es que los objetos no reflejan el mismo ángulo que el plano. Esta implementación

agrega una dificultad extra al problema, puesto que habría que poner objetos dentro del plano en el mismo ángulo, para luego variar la escena completa. Habría que lograr obtener muchas fotos de los objetos en las mismas posiciones y diferente ángulo y para el mismo ángulo con diferentes posición, clasificar todos los ángulos para un tipo de escena y así ir insertándolas.

En el caso de del entrenamiento, se deja ejecutar hasta 12.000 iteraciones, pues la documentación recomienda dejar por lo menos 2.000 iteraciones por cada objeto. El entrenamiento duró aproximadamente 24 horas.

Lamentablemente al implementación con la que se está trabajando no va mostrando o guardando variables que representen la validación y al querer obtener estas métricas no se obtienen de una manera gráfica por lo que se procede a compilar la implementación de Darknet de *AlexeyAB*, la cual consta de comandos para obtener diversas métricas y algunas otras modificaciones al momento de entrenar. Se aconseja en un futuro trabajar con esta última implementación.

Para entrenamiento se ocupar el 49 % de las escenas generadas y para validación el 21 %, esto es para no ocupar todos los ejemplos y lograr generalizar al no cargar ejemplos similares.

Los resultados obtenidos fueron los siguientes:

```
class_id = 0, name = cereal, 0    ap = 90.83 %
class_id = 1, name = lata,        ap = 90.84 %
class_id = 2, name = leche,       ap = 90.66 %
class_id = 3, name = botella,     ap = 90.72 %
class_id = 4, name = tarro,       ap = 90.85 %
for thresh = 0.25, precision = 0.97, recall = 0.92, F1-score = 0.95
for thresh = 0.25, TP = 31478, FP = 884, FN = 2683, average IoU = 80.50 %
```

Figura 21: Escena con dos elementos original a la izquierda y con los tres filtros a la derecha.

Como es posible ver en la Figura 21, el promedio de precisión en la clasificación para todos los objetos es muy cercana a 91 % y el promedio de IoU es de 80.5 % lo que quiere decir que los cuadros encontrados son muy parecidos a los que deberían ser.

Los resultados obtenidos son aceptables para la etapa de validación, aún así se maneja un poco de desconfianza en el desempeño que la red tendría en la realidad puesto que no se ha hecho un test sobre escenas reales.

En las siguientes figuras se muestran resultados sobre una imagen del conjunto de validación, una imagen real y otra sacada de internet. Aunque no sean representativas sobre el desempeño real de la red, se puede ver que en estos ejemplos la red funciona de manera aceptable detectando algunos objetos presentes.

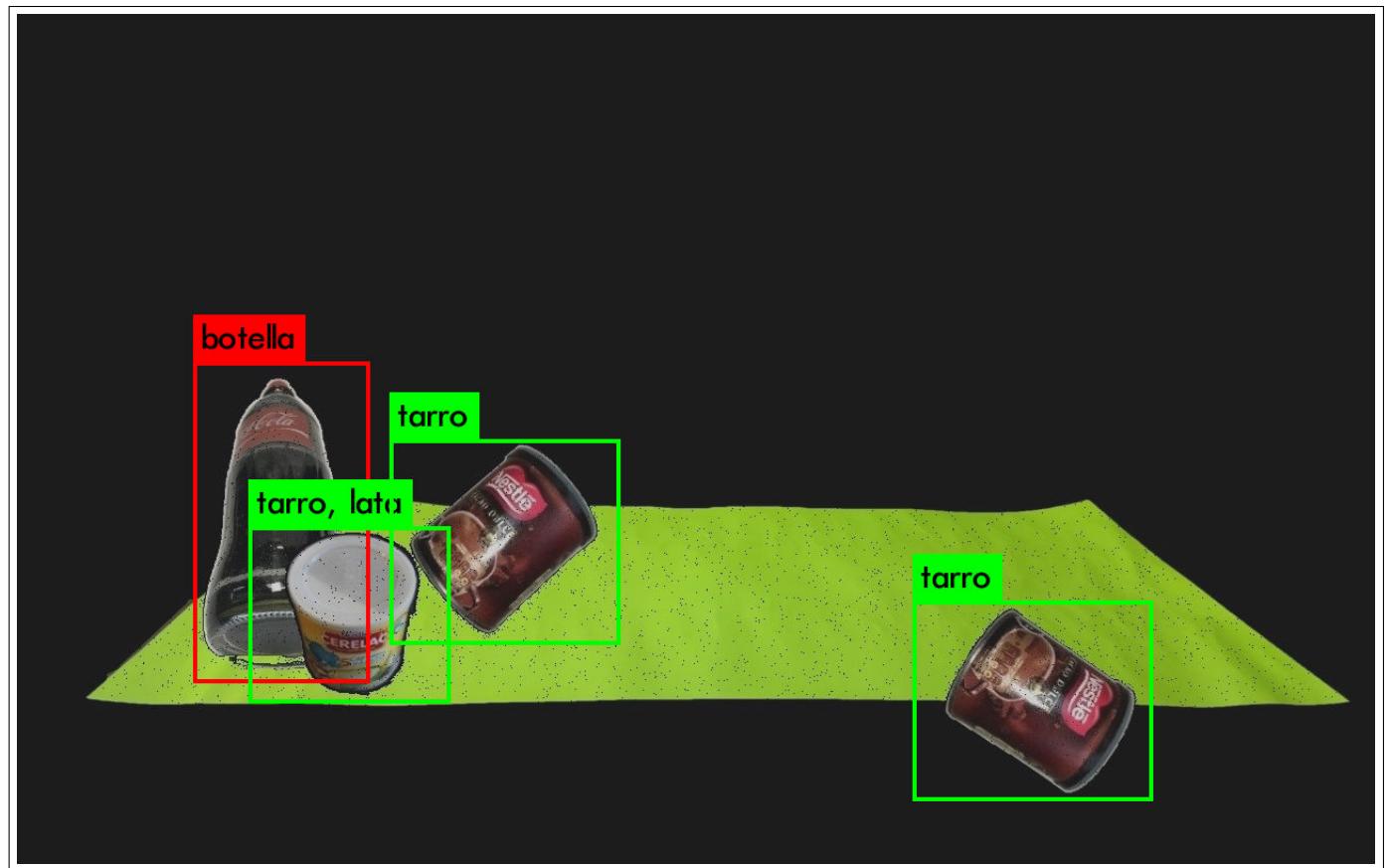


Figura 22: Detección sobre imagen con 4 objetos del conjunto de validación.

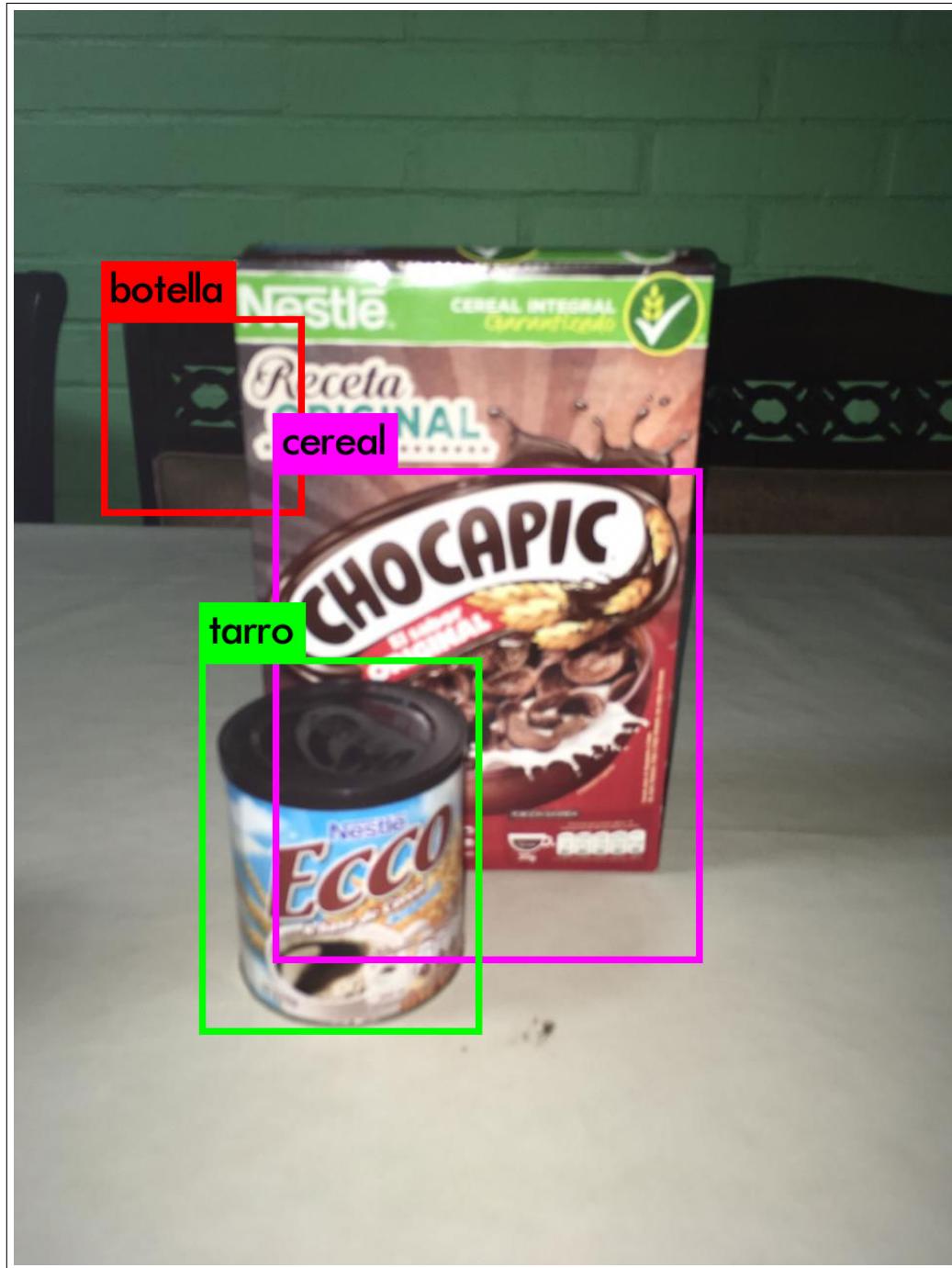


Figura 23: Detección sobre un escenario real con un tarro y cereal.



Figura 24: Detección sobre imagen con botellas extraído desde la web.

4. Conclusiones

La mayor dificultad para entrenar una red como *tiny-yolo* no se encuentra en la manera de hacerlo, la dificultad está en obtener un conjunto de ejemplos representativos, variados y grande de las escenas con las que se requiere trabajar.

Primero es necesario mencionar la etapa de segmentación. Esta etapa es importante para el resto del proceso, puesto que dependiendo la cantidad de objetos que se puedan segmentar correctamente, es la cantidad de objetos con los que voy a seguir operando.

Ya se vio que la cantidad de elementos a segmentar correctamente se ve influido por el fondo y la iluminación, cosa que para alguien sin experiencia en fotografía puede ser todo un desafío, pero para alguien dedicado en esta área, fotografiar objetos que sirvan para el proceso puede ser fácil y natural. Aún así se trató que la implementación fuese robusta, tomando en cuenta diferentes tamaños del objeto original, de esta manera al disminuir el tamaño se van perdiendo los detalles lo que facilita encontrar un contorno, pero el borde también se va enanchando un poco debido a que la diferencia entre color del borde con el fondo se va haciendo cada vez menos clara, por esto se necesita un fondo que contraste.

Para la etapa de data augmentation, se tienen algunos desafíos, como probar con otra cantidad de filtros y ejemplos, lo que actualmente es limitado por el hardware disponible (memoria RAM y en HDD).

El contexto es un punto que a pesar que se deja implementado para futuros trabajos, se mantuvo plano y fijo (el mismo color desde todos los ángulos).

Dado que armar las escenas toma un tiempo y espacio en disco considerable, no se probó con otros contextos, pero se considera que un solo contexto sin textura no alcanza a ser representativo de las escenas reales a detectar.

Por otra parte se tiene el entrenamiento. Dada la implementación ocupada en primera instancia no se tienen métricas de validación y la red va entrenando sin validar. Aunque 2000 iteraciones por objeto arrojó resultados aceptables, no se sabe si es el máximo rendimiento que pudo haber alcanzado la red.

Dada la segunda implementación ocupada, se puede ir calculado el MAP (mean average precision) cada cierta cantidad de iteraciones y de esta manera poder ver en donde se alcanza el máximo sin necesidad de tantas iteraciones o antes que la red se sobre ajuste al conjunto de entrenamiento.

Finalmente, aunque los resultados de rendimiento son aceptables, la red se pudo haber sobre entrenado con los mismos ejemplos de entrenamiento. Recordar que aunque se ocupó un 49% de las escenas creadas para entrenar, las de más escenas son similares, tanto en distribución de objetos y sus clases.

Dado lo anterior, la red debiese pasar por un testeo en donde se incluyan escenarios reales y se calcule su rendimiento.