UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

Ataques adversarios en una red neuronal de clasificación de señales de tráfico

Autor:
Jairo Fariña Mallón

Director:
Manuel Mucientes Molina

Grao en Enxeñaría Informática

Julio 2020

Trabajo de Fin de Grao presentado en la Escuela Técnica Superior de Ingeniería de la Universidad de Santiago de Compostela para la obtención del Grado en Ingeniería Informática

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

D. Manuel Mucientes Molina, Profesor del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela

INFORMA:

Que la presente memoria, titulada *Ataques adversarios en una red neuronal de clasificación de señales de tráfico*, presentada por **D. Jairo Fariña Mallón** para superar los créditos correspondientes al Trabajo de Fin de Grado de la titulación del Grado en Ingeniería Informática, fue realizada bajo mi dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

Y para que así conste a los efectos oportunos, expide el presente informe en Santiago de Compostela, a 30 de junio de 2020:

El director,	El alumno
MUCIENTES MOLINA MANUEL - 44810714K Firmado digitalmente por MUCIENTES MOLINA MANUEL - 44810714K Fecha: 2020.07.06 09:53:17 +02'00'	Firmado por FARIÑA MALLON, JAIRO (FIRMA) el día 06/07/2020
D.Manuel Mucientes Molina	Jairo Fariña Mallón

Agradecimentos

- Gracias a mi familia por ayudarme en todo lo posible para llegar hasta aquí.
- A mis amigos: Carlos, Muras, Nolas, Fernando, Brais y Dabo por hacerme esta etapa muchísimo más fácil en lo académico y, sobretodo, en lo no académico.
- A mi director Manuel Mucientes por ser tan rápido y eficaz en todas y cada una de mis preguntas.
- A Pablo Monteagudo, por echarme una mano de manera desinteresada en momentos de bloqueo.
- A Alba Martínez por toda su ayuda para la creación de los dataset de testing.

Resumen

En este trabajo se pretenden poner de manifiesto las vulnerabilidades que tienen hoy en día las redes neuronales profundas con la finalidad de que se deje de confiar ciegamente en ellas para determinadas actividades como reconocimiento facial en un aeropuerto o detección de señales de tráfico por cámara. A lo largo de la memoria se expondrán diferentes métodos para la generación de ejemplos adversarios en redes de detección y clasificación de imágenes especificando las similitudes y diferencias entre ellos para acabar extrayendo conclusiones y, aunque sea difícil, métodos de defensa existentes y por explorar contra estos algoritmos.

Índice general

1. Introducción **1**

1.1. Problemática y Motivación 1
1.2. Hipótesis a testar 2
1.3. Objetivos 3
1.4. Estructura de la memoria 3
1.5. Marco teórico 4
1.5.1. Diferentes problemas de la visión por computador 4
1.5.2. Aprendizaje supervisado 5
1.5.3. Clasificación y detección 6
1.5.4. Arquitectura de la red de clasificación implementada . . . 6
1.5.5. Arquitectura de la red de detección implementada 10
1.5.6. Introducción a los ataques adversarios 13
1.6. Diferencias entre los ataques adversarios en redes de clasificación y en redes de detección 14
1.7. Ataques de caja blanca y de caja negra 15

2. Estado del arte **16**

3. Materiales **18**

3.1. Entorno 18
3.1.1. Google Colaboratory 18
3.1.2. GPUs Utilizadas 18
3.2. Lenguaje de programación 19
3.3. Tensorflow y Keras 19
3.4. Bibliotecas de imágenes 21
3.4.1. Belgium Traffic Sign Dataset 21
3.4.2. German Traffic Sign Detection Dataset 22
3.5. Visualización 23
3.6. Gestión de la configuración 23
3.7. Cálculo del gradiente 24

4. Métodos **25**
4.1. Data Augmentation 25
4.2. Función de coste y métrica de validación de la red de clasificación 26
4.3. Función de coste y métrica de validación de la red de detección . . 26
4.4. Ataques adversarios no dirigidos en la red de clasificación 30
4.4.1. Análisis de los valores del hiperparámetro 30
4.5. Ataques adversarios dirigidos en la red de clasificación 31
4.6. Ataques adversarios dirigidos en la red de detección 32

5. Pruebas y discusión de resultados **34**
5.1. Validación y test en la red de clasificación 34
5.1.1. Conjunto de test en la red de clasificación 35
5.2. Validación y test en la red de detección 36
5.2.1. Conjunto de test en la red de detección 37
5.3. Ataques adversarios no dirigidos 39
5.3.1. Implementación en Python 39
5.3.2. Algunos resultados de los ataques no dirigidos 39
5.4. Ataques adversarios dirigidos 40
5.4.1. Implementación en Python 40
5.4.2. Algunos resultados de los ataques dirigidos 41
5.5. Ataques adversarios dirigidos en la red de detección 43
5.5.1. Implementación en Python 43
5.5.2. Algunos resultados de los ataques en la red de detección . 43
5.6. Ataques en videos en redes de detección 46

6. Conclusiones y posibles ampliaciones **48**

A. Código en Python de los algoritmos **50**
A.1. Ataques adversarios no dirigidos en la red de clasificación 50
A.2. Ataques adversarios dirigidos en la red de clasificación 51
A.3. Ataques adversarios en la red de detección 52

B. Apendice B **54**

C. Manual de usuario **59**
C.1. Instalación de la red de clasificación 59
C.2. Ejecución de la red de clasificación 59
C.2.1. Ataques adversarios dirigidos 60
C.2.2. Ataques adversarios no dirigidos 60
C.2.3. Pintado de imágenes 60
C.3. Instalación de la red de detección 60
C.4. Ejecución de la red de detección 61
C.4.1. Análisis de vídeos 62

C.4.2. Ataques adversarios en la red de detección 62
C.4.3. Visualización imágenes 63

Índice de figuras

1.1. Diferentes problemas de la visión por computador de relevancia en este TFG. Imagen obtenida de [32]. 5
1.2. Arquitectura de la red Inception V3 modificada. Imagen obtenida de [24]. 7
1.3. Funcionamiento de un filtro para cada uno de los canales RGB. Imagen obtenida de [40]. 8
1.4. Ejemplo de las capas de agrupación: Operación Máximo en la izquierda y Operación Media en la derecha. Imagen obtenida de [58]. 9
1.5. Todas las transformaciones que habría que clasificar. Imagen obtenida de Manuel Mucientes. 10
1.6. Arquitectura Fast RCNN. Imagen obtenida de [7] 11
1.7. Funcionamiento RPN. Imagen obtenida de [43]. 12
1.8. Arquitectura Faster R-CNN. Imagen obtenida de [43]. 12

2.1. Parches en reconocimiento facial y señales de tráfico. Imágenes obtenidas de [36] y [6] 17

3.1. Pipeline GPU, Keras y Tensorflow. Imagen obtenida de [48]. . . . 20
3.2. Ejemplo de las 62 clases que contiene el *Belgium Traffic Sign Dataset*. Imagen obtenida de [4]. 21
3.3. Ejemplo de las 5 clases de objetos que detecta la red. 23

4.1. Anchor, bounding box real y bounding box predicho. Imagen obtenida de [2]. 27
4.2. IoU. Imagen obtenida de [44]. 28
4.3. Gráfica de ejemplo para el cálculo del AP con el *Recall* en el eje de abscisas y la precisión en el eje de ordenadas. Imagen obtenida de [17]. 29
4.4. Rectángulos para el cálculo del área bajo la curva. Imagen obtenida de [17]. 29
4.5. Cómo afecta la modificación del hiperparámetro. Imagen obtenida de [53]. 31

5.1. Imagen de la clase 37 del conjunto de test de la red de clasificación. 36

5.2. Imagen del conjunto de test en la red de detección. 37
5.3. Antes y después de aplicar el NMS. Imagen obtenida de [1] 38
5.4. AP de la clase 1 38
5.5. AP de la clase 2 38
5.6. AP de la clase 3 38
5.7. AP de la clase 4 38
5.8. AP de la clase 5 38
5.9. Transformación de una curva izquierda (3) a una curva derecha (4). 39
5.10. Transformación de una señal de stop (21) a una señal de camino prioritario (61). 40
5.11. Transformación de una señal de stop (21) a una señal de límite de velocidad (32). 41
5.12. Transformación de una señal de dirección prohibida (22) a una señal de límite de velocidad (32). 41
5.13. Transformación de una curva izquierda (3) a una señal de peligro ganado (9). 42
5.14. Transformación de una curva izquierda (3) a una señal de límite de velocidad (32). 42
5.15. Imagen original. Clasificada como una señal de ceda el paso. . . . 44
5.16. Imagen atacada, no hay detección. 44
5.17. Ruido añadido a la imagen. 44
5.18. Imagen original. Clasificada como una señal de Stop. 45
5.19. Imagen atacada. Clasificada como una señal de prohibido el paso. 45
5.20. Modificación de la señal de Stop. 45
5.21. Imagen original. Clasificada como una señal de prohibido el paso. 45
5.22. Imagen atacada. Clasificada como una señal de Stop. 45
5.23. Ruido completo añadido a la señal de prohibido el paso. 46
5.24. Ejemplo de fotogramas en vídeo 47

B.1. Imagen Original 54
B.2. Rotacion 54
B.3. Desplazamientos Horizontales 55
B.4. Desplazamientos Verticales 55
B.5. Inclinación de la imagen 55
B.6. Zoom 55
B.7. Giro Horizontal 56
B.8. Imagen Original 56
B.9. Saturacion 56
B.10. Ejemplo de la clase 20 56
B.11. Ejemplo de la clase 19 57
B.12. Ejemplo de fotografía en las calles de santiago 57
B.13. Ejemplo de fotografía en las calles de santiago 58
B.14. Ejemplo de fotografía en las calles de santiago 58

C.1. Celda de ataques dirigidos. 60
C.2. Celda de ataques no dirigidos. 60
C.3. Celda de pintado de imágenes. 60
C.4. Celda dónde es necesario reiniciar. 61
C.5. Celda 18 de código. 62
C.6. Celda 22 de código. 63
C.7. Celda 20 de pintado. 63

Capítulo 1

Introducción

1.1. Problemática y Motivación

La inteligencia artificial y, en concreto, las redes neuronales, están en auge en la época actual y confiamos en ellas sin dudar en ningún momento de cómo están funcionando. Siendo realistas únicamente atendemos a la salida que esta nos otorga en función de las entradas que aportamos con una confianza ciega en que el entrenamiento fue adecuado y el resultado, por ende, es correcto.

Actualmente, multitud de dispositivos contienen cámaras para el reconocimiento de objetos, desde coches que detectan señales de tráfico en el cuadro de control hasta aplicaciones móviles que son capaces de identificar elementos con fotografías hechas con su propia cámara. En todos estos dispositivos se puede apreciar la problemática previamente descrita, aceptamos el resultado que mayormente es correcto sin preguntarnos cómo se llegó a esa conclusión y, en caso de que la clasificación falle, no provoca un gran problema a los usuarios.

Con la llegada de los vehículos autónomos esta problemática se vuelve un problema real puesto que un error en la detección de una señal de tráfico provocaría daños fatales para los usuarios. En estos casos, la precisión de la red es fundamental y necesitamos profundizar en la búsqueda de mejoras que nos permitan aumentar esta precisión. Este trabajo de investigación se ha realizado en los últimos años de manera bastante intensa haciendo redes muy profundas que obtienen una muy alta precisión en sus decisiones.

Sin embargo, ha surgido un nuevo problema: la manipulación de las entradas de la red para engañarla y obtener otro resultado. Si las entradas de la red se ven claramente manipuladas el usuario podría darse cuenta del engaño, pero si las entradas han sido manipuladas de manera invisible al ojo humano el problema es más grave.

Por todo esto, es especialmente importante entender y comprender cuáles son estas vulnerabilidades que tienen las redes neuronales ya que, únicamente conociéndolas, podremos realizar defensas que nos protejan de los atacantes en un futuro. En este trabajo nos centraremos en cómo realizar estos ataques partiendo de los fundamentos de bajo nivel y llevándolos a su implementación en un lenguaje de programación de alto nivel.

Es importante destacar que no es necesario tener control sobre la red para realizar estos ejemplos adversarios, conocidos como ataques de caja negra como se comentará posteriormente, pero si que es necesario que exista una red neuronal a la que atacar. Por ello, también es necesario que implementemos una red neuronal de clasificación y otra de detección de señales de tráfico sobre las que realizaremos los ataques.

1.2. Hipótesis a testar

En este trabajo veremos los distintos métodos existentes para realizar estos ejemplos adversarios. Debemos analizar la facilidad con la que se pueden lograr y la efectividad de los mismos en las distintas redes. Trabajos anteriores, que posteriormente comentaremos, han demostrado la eficacia de métodos basados en el gradiente para realizar estos ejemplos adversarios imperceptibles a simple vista.

Es fácil visualizar que en redes con pocas capas los ataques son realmente sencillos de ejecutar pues los pesos de cada una de las pocas neuronas de la red no son capaces de estabilizar el resultado al realizar pequeñas variaciones en la entrada. Este trabajo utilizará, por tanto, redes neuronales profundas (con muchas capas) a las que será mucho más complicado engañar con variaciones prácticamente imperceptibles para el ojo humano.

Debemos destacar también que las redes de clasificación resultan bastante más fáciles de atacar pues únicamente se obtiene un resultado (la clase predicha). Sin embargo, las redes de detección, que proporcionan el rectángulo que contiene al objeto y su clasificación, resultan bastante más complejas y los primeros avances en este campo llegaron más tarde.

1.3. Objetivos

Como se ha comentado anteriormente el objetivo general de este Trabajo de Fin de Grado es conocer, aprender y demostrar que es posible realizar ataques adversarios en redes neuronales que confundan las predicciones de la red a nuestro antojo. Se espera que, al final del trabajo, se obtenga un alto conocimiento de las características que permiten realizar los ejemplos adversarios posibilitando así el desarrollo de nuevos algoritmos de ataque en redes neuronales profundas.

Así pues, para la consecución de este objetivo global del trabajo se han elaborado 5 objetivos:

1. Implementar una red neuronal de clasificación de señales de tráfico en 62 categorías diferentes a partir del *Belgium Traffic Sign Database* transfiriendo el conocimiento desde el modelo *Inception V3* de Google.

2. Implementar un algoritmo que permita realizar ataques adversarios no dirigidos utilizando como base redes neuronales profundas de clasificación.

3. Implementar otro algoritmo que permita realizar ataques adversarios dirigidos utilizando como base redes neuronales de clasificación.

4. Construir una red neuronal de detección de señales de tráfico en 5 categorías a partir del *German Traffic Sign Dataset* partiendo del conocimiento de la *Faster R-CNN InceptionV2 COCO*

5. Implementar un algoritmo que permita engañar a la red neuronal de detección previamente descrita. Este algoritmo debe permitir ataques dirigidos a otras clases de la red y ataques de denegación de la detección.

1.4. Estructura de la memoria

El **capitulo 1**: comienza por una introducción a la problemática y a la visión por computador para terminar por las arquitecturas de las redes implementadas y los fundamentos en los que se apoyan los ataques adversarios.

En el **capitulo 2**: se define el estado actual del problema, el conocimiento existente y las diferencias de nuestro trabajo con ellos.

En el **capítulo 3**: se especifica todo lo relacionado con los materiales y entorno utilizados. También se incorporan los métodos matemáticos sobre los que se sustenta el trabajo.

En el **capítulo 4**: se explican los diferentes métodos utilizados para la realización del trabajo, su diseño y su funcionamiento.

En el **capítulo 5**: se plantean las pruebas realizadas y la discusión de resultados.

Finalmente en el **capítulo 6**: se extraen conclusiones y posibles ampliaciones del trabajo presentado

Esta memoria sigue la estructura de tipo A propuesta en el reglamento del Trabajo de Fin de Grado aprobada por la Junta de Escuela de la ETSE el 4 de Junio de 2020.

1.5. Marco teórico

1.5.1. Diferentes problemas de la visión por computador

La vista es uno de los sentidos más importantes de los seres humanos por la información que le aporta a nuestro cerebro a la hora de tomar decisiones. En múltiples aspectos de la vida cotidiana la vista juega un papel fundamental en nuestras elecciones desde escoger el camino más rápido según la gente que lo transite, escoger una cola del supermercado u otra y, entre otras muchas, el problema que vamos a tratar: seguir las indicaciones de las señales de tráfico mientras conducimos.

La relevancia del sentido la vista en nuestra vida diaria ha derivado en la necesidad de explorar métodos para analizar y comprender los elementos de una imagen a través de un ordenador. A esto lo denominamos **visión por computador** y comenzó a desarrollarse a finales de la década de 1960 en universidades pioneras en inteligencia artificial.[13]

Sin embargo, no en todos los casos queremos analizar y procesar una imagen de la misma manera pues los objetivos esperados pueden ser muy diversos. Principalmente podemos distinguir 4 grandes problemas en la visión por computador, en relación con la temática de este TFG:

1. **Clasificación**: Probablemente sea el problema más conocido y más básico. Consiste en clasificar una imagen dada como entrada en una de las muchas categorías posibles definidas en la red. Esta tarea se ha ido perfeccionando a lo largo de los años hasta el punto en el que ya existen redes neuronales de clasificación que superan el rendimiento humano. Esto se puede observar en trabajos como el presentado en el ICVV de 2015. [12]

2. **Localización**: Este problema consiste en encontrar la ubicación de un único objeto dentro de una imagen. Se puede combinar, además, con el problema de clasificación previamente expuesto para categorizar al objeto dentro de una clase.

3. **Detección**: Este problema está relacionado con la iteración de los dos anteriores. En este caso debemos detectar y clasificar un número de objetos **variable** dentro de una imagen. Esa condición de variable es lo que lo hace diferente a lo anterior y lo que lo complica.

4. **Segmentación**: En este problema no sólo debemos detectar los objetos de una imagen sino que, además, debemos encontrar una máscara de píxeles de cada uno de los objetos detectados.

En la figura 1.1 podemos apreciar todo esto de una manera mucho más visual.

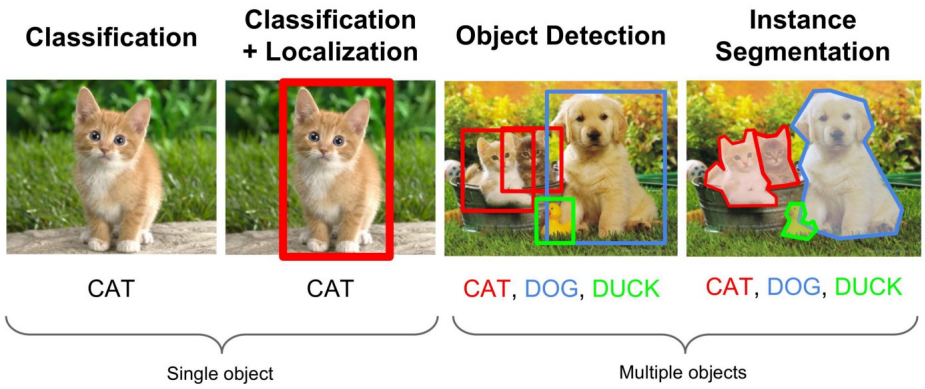

Figura 1.1: Diferentes problemas de la visión por computador de relevancia en este TFG. Imagen obtenida de [32].

1.5.2. Aprendizaje supervisado

La técnica utilizada para el entrenamiento de las redes de clasificación y detección implementadas es el aprendizaje supervisado. Esta técnica consiste en que los datos de entrenamiento son pares de objetos: el primer componente está formado por los datos de entrada y el segundo componente sería la salida deseada.

En la red de clasificación tendríamos como par de objetos de entrenamiento una imagen de una señal de tráfico y un valor correspondiente a la clase de la señal de tráfico.

Para la red de detección el objeto de entrenamiento estaría formado por la imagen de entrada y 5 valores: 4 correspondientes a las coordenadas del rectángulo de detección y 1 correspondiente a la clase por cada objeto detectado.

Es importante entender que, **para una sola imagen**, siempre tendremos un único valor de salida para la red de clasificación mientras que, en la red de detección, se puede detectar más de un objeto en la misma imagen.

1.5.3. Clasificación y detección

En este apartado nos centraremos especialmente en los dos problemas que tenemos que resolver en este trabajo. Se dará una explicación básica y a grandes rasgos que luego iremos detallando a lo largo de la memoria.

En el problema de clasificación que se nos presenta, debemos introducir una imagen de una señal de tráfico pero únicamente la señal, sin el entorno. Así, esperamos obtener como salida un único resultado en formato vector: la probabilidad de pertenencia de la imagen para cada una de las 62 clases. Entenderemos, por tanto, que la probabilidad más alta corresponde a la categoría que la red ha predicho para la señal.

Con respecto a la red de detección, en este problema introduciremos una imagen de un entorno que incluya (o no) señales de tráfico. La red deberá devolvernos como salida varios parámetros para cada objeto detectado: la probabilidad de que ese objeto pertenezca a una de las 5 clases (nos quedaremos con la mayor) y 4 valores más que conforman el rectángulo de detección del objeto.

Nótese entonces que la red de detección necesita de un clasificador que solucione el problema de clasificación para poder categorizar cada detección en una clase.

1.5.4. Arquitectura de la red de clasificación implementada

Como ya se introdujo previamente para que nuestros ataques y pruebas tengan validez es necesaria una red de clasificación profunda que sea realmente difícil de engañar. Es fácil observar que, una red con muy pocas neuronas es sencilla de manipular.

Por ello,vamos a partir del conocimiento previo de la red *InceptionV3* [50]. Es la tercera versión de una red neuronal convolucional de clasificación desarrollada por Google que comenzó como un módulo de su modelo *GoogleNet*. Esta red está

entrenada bajo la biblioteca de imágenes *ImageNet* [18] que contiene más de **1 millón de imágenes** pertenecientes a **más de 1.000 categorías**.

Sin embargo, a nosotros no nos interesan esas 1.000 categorías de clasificación final, necesitamos una clasificación en 62 clases dónde cada una representa una señal de tráfico diferente. A este proceso dónde partimos de una red previamente entrenada y la adaptamos a nuestro modelo se le conoce como **transferencia de conocimiento**.

Para la implementación de esta red vamos a apoyarnos en el artículo *Transfer Learning Based Traffic Sign Recognition Using Inception-v3 Model* [24] publicado en 2018. En este artículo transfieren el conocimiento del modelo *InceptionV3* entrenado en *ImageNet* a un modelo de reconocimiento de señales de tráfico usando, exactamente, la misma librería de imágenes que nosotros. La arquitectura de la red modificada se puede ver en la imagen 1.2:

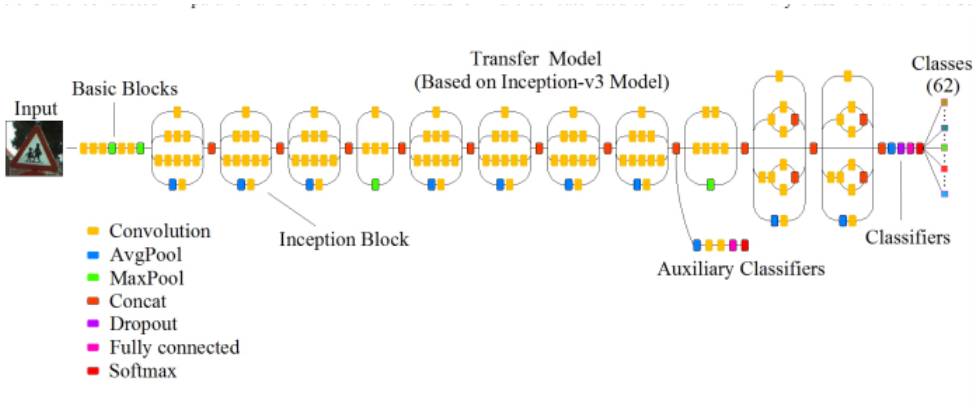

Figura 1.2: Arquitectura de la red Inception V3 modificada. Imagen obtenida de [24].

Como se puede apreciar por la mayoría de capas, se trata de una red convolucional. La idea detrás de una red neuronal convolucional (*CNN* en inglés) es que aplicamos un conjunto de filtros a las imágenes para extraer sus características. Este proceso se repite en las siguientes capas, utilizando como entrada el mapa de características generado por la capa previa.

Aplicar un **filtro** a la imagen no es más que un conjunto multiplicador dígito a dígito que transforma unos valores de entrada en otros de salida tal y como se puede ver en la figura 1.3. Estos filtros en una imagen *RGB* se aplican para cada uno de los canales de la imagen. Lo interesante de estos filtros es que, aunque se inicializan de manera aleatoria, van aprendiendo con el entrenamiento para acabar extrayendo las características importantes de las imágenes. Además, si

inicializamos dos filtros de la misma manera, es probable que acaben extrayendo las mismas características por lo que la inicialización aleatoria favorece a que estos aprendan características diferentes.

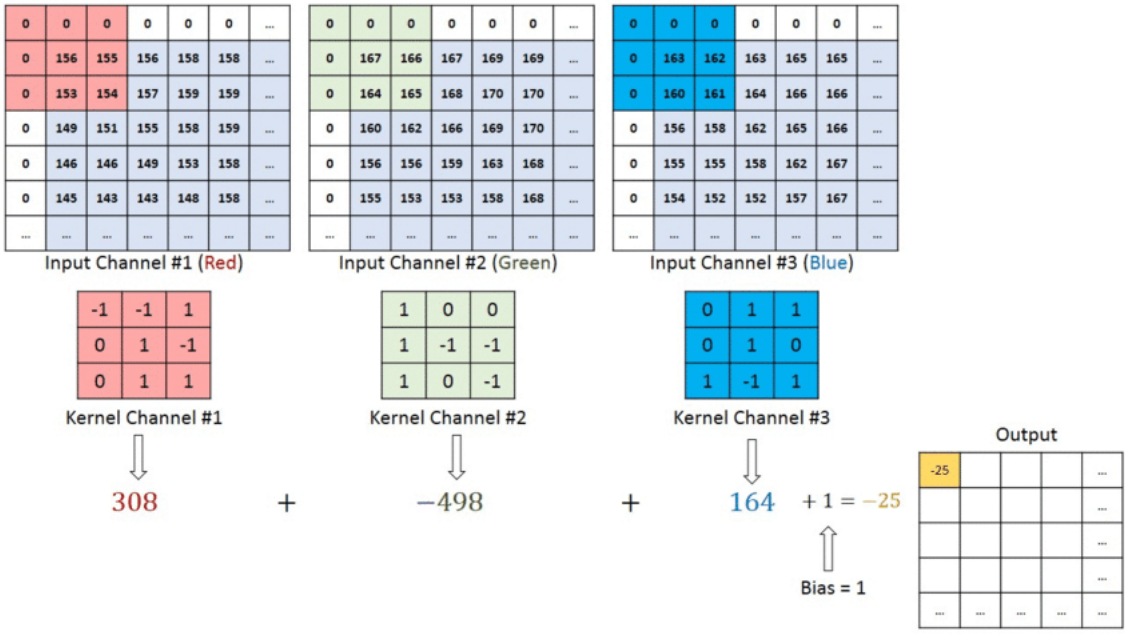

Figura 1.3: Funcionamiento de un filtro para cada uno de los canales RGB. Imagen obtenida de [40].

Tras realizar la operación previamente descrita es necesario utilizar una **función de activación.** Esta función de activación no-lineal evita que la red aprenda únicamente patrones lineales. Esta red utiliza la función de activación **Unidad Linear Rectificada (*ReLU* en inglés)** que viene definida así:

$$f(x) = max(0, x) = \begin{Bmatrix} x, x \geq 0 \\ 0, x \leq 0 \end{Bmatrix}$$

Por lo que podemos definir la operación de convolución como:

$$Y = f\left(\sum_{i=1}^{n} x_i * W + b_i\right)$$

Dónde Y representa la salida, W es el filtro aplicado, X es la entrada, b es el parámetro *bias* y f es la función de activación previamente descrita.

Estas capas van, normalmente, de la mano de **capas de agrupación (*Pooling* en inglés)**. Estas capas reducen el tamaño de un mapa de valores pero manteniendo sus características. Para esto, cogen un conjunto de píxeles vecinos y realizan alguna operación con ellos (media, máximo valor, sumatorio...). Esto se refleja en la figura 1.4.

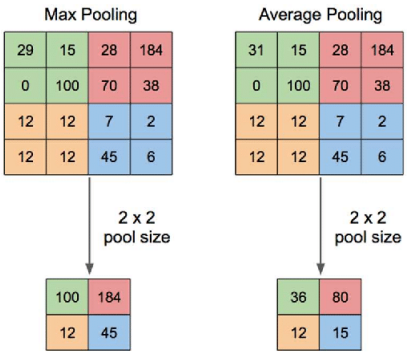

Figura 1.4: Ejemplo de las capas de agrupación: Operación Máximo en la izquierda y Operación Media en la derecha. Imagen obtenida de [58].

Finalmente las capas totalmente conectadas sirven para transformar todos los mapas de características en un vector de 1 dimensión. En la última capa totalmente conectada se aplica una función de activación que transforma los valores obtenidos en una distribución de probabilidad. Utilizaremos la función de activación *SoftMax*, utilizada cuando cada elemento pertenece únicamente a una clase, y que viene definida como:

$$SoftMax(y_j) = \frac{e^{y_j}}{\sum_{i=1}^{n} e^{y_i}}$$

Donde y_j representa la probabilidad de la clase j y n representa el número total de clases.

Se puede observar en la figura 1.2 que, en la red *InceptionV3*, contamos también con clasificadores auxiliares. Gracias a estos clasificadores obtenemos resultados de entrenamiento más estables y mejor convergencia en el gradiente.

También podemos observar capas de concatenación, que unen el resultado de varias capas en una dimensión especificada, y capas de *dropout* que sirven para reducir el sobre-entrenamiento(*overfitting* en inglés) desactivando neuronas durante el entrenamiento para favorecer la generalización [46]

Una vez conocidas las capas utilizadas debemos recordar que partimos de un modelo *InceptionV3* que clasifica en 1000 categorías por lo que debemos transformarlo para que clasifique en 62 categorías. Para ello, **eliminaremos la última capa de la red original** y concatenaremos a la red una capa de *Pooling Max* para adaptar el formato a nuestras clases seguida de una capa de activación *SoftMax* en 62 clases.

1.5.5. Arquitectura de la red de detección implementada

Una red de detección también necesita de un extractor de características para reconocer los objetos. Esto se logra con una red convolucional. Por ello, **en un primer momento se optó por utilizar la red convolucional previamente implementada** como extractor y seguir implementando el resto de fases de la arquitectura de detección. Sin embargo, decidimos descartar la opción y utilizar un método de transferencia de conocimiento similar al anterior debido a 3 circunstancias: el tiempo disponible, la dificultad de ejecución y la eficacia de la red (utilizar una red reconocida siempre va a funcionar mejor que una red implementada por nosotros).

Una vez realizamos la parte convolucional obtenemos fragmentos con las características que nos interesan de la imagen. A cada fragmento le aplicamos transformaciones (reescalado y aspect ratio) para abarcar un mayor número de objetos y cada transformación se clasifica como una categoría o como fondo (no detecta objeto). Esto computacionalmente es muy costo. Habría que clasificar exactamente todos los recuadros azules que se ven en la figura 1.5 que refleja las transformaciones de los fragmentos.

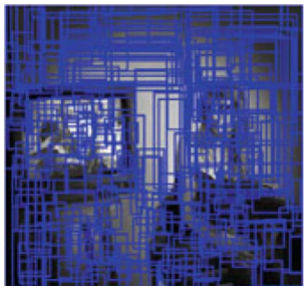

Figura 1.5: Todas las transformaciones que habría que clasificar. Imagen obtenida de Manuel Mucientes.

R-CNN, Fast R-CNN y Faster R-CNN

Para evitar realizar el cálculo de todas esas transformaciones surge la propuesta de Ross Girshick en 2014, la **R-CNN** [8]. Este artículo propone un método para extraer solo **2000 regiones** de cada imagen a las que llama **regiones de interés (RoI)**. Estas regiones se eligen a través de algoritmos de propuesta como la búsqueda selectiva [56] que se basa en la agrupación de regiones similares en función de características como el color, el tamaño o la forma. Pero aún tendríamos que clasificar 2000 fragmentos pasándolo por la red convolucional (CNN) y clasificando su salida a través de una función *SoftMax* para la clase y de 4 valores que especifican el *bounding box*[1].

[1]El bounding box es el recuadro delimitador de un objeto detectado.

Para solucionar esto surge la propuesta del mismo autor en 2015, la **Fast R-CNN** [7]. Esta nueva arquitectura en vez de utilizar la red convolucional 2000 veces, **la utiliza una única vez** y, a partir del resultado, obtiene las regiones de interés.

Sin embargo, al utilizar esta técnica es muy probable que las regiones de interés no sean del mismo tamaño. Para ello utilizamos una capa denominada ***RoI Pooling Layer*** cuya función es hacer que las RoI tengan todas el mismo tamaño con técnicas similares a las Pooling explicadas previamente.

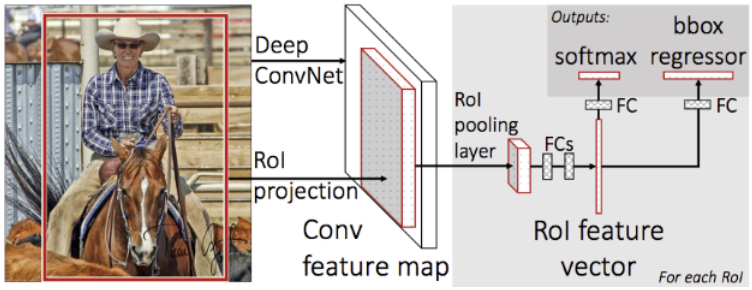

Figura 1.6: Arquitectura Fast RCNN. Imagen obtenida de [7]

En esta nueva red, aunque obtenemos mejoras muy notables, seguimos necesitando de un algoritmo de búsqueda de regiones de interés como la búsqueda selectiva, lo que ralentiza el proceso.

Para resolver este problema se publica la **Faster R-CNN** [43] que, como su propio nombre indica, mejora a la red anterior. En esta arquitectura las RoI no son generadas con algoritmos de búsqueda sino que son generadas a través de una red denominada ***Region Proposal Network (RPN)***. Esta RPN recibe los mapas de características de la red convolucional y utiliza una ventana deslizante sobre ellos. Para cada ventana, genera k *Anchor Boxes* variando el *aspect ratio* y el tamaño, por defecto 3 valores de escalado y 3 de aspect ratio (9 anchors) . Estos *Anchor Boxes* vienen definidos en el entrenamiento (**son fijos**). Para cada *Anchor*, la red predice 2 cosas: la probabilidad de que haya un objeto (no considera a qué clase pertenece) y ajusta el *Anchor* al *bounding box* del objeto. Esto se puede ver en la figura 1.7.

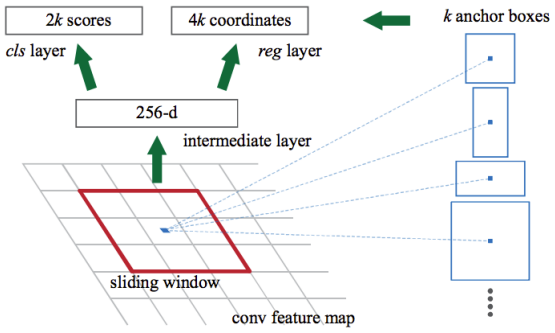

Figura 1.7: Funcionamiento RPN. Imagen obtenida de [43].

Ahora, volvemos a necesitar a la *RoI pooling* ya que nos volvemos a encontrar regiones de diferente tamaño. Finalmente las regiones que contengan objetos se unen a capas totalmente conectadas para obtener 5 valores de salida: una salida tras la función *SoftMax* con la probabilidad de que pertenezca a cada clase y 4 valores más que **modifican los *Anchor Boxes*** para ajustarlos al *bounding box* real. Todo esto se puede ver en la figura 1.8.

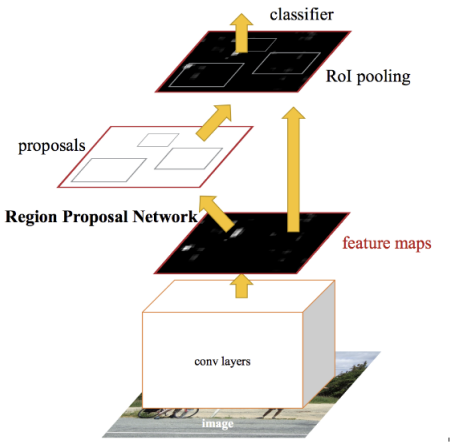

Figura 1.8: Arquitectura Faster R-CNN. Imagen obtenida de [43].

Esta será la **red de detección que implementaremos** con nuestra biblioteca de imágenes. Utilizaremos la *API de TensorFlow Object Detection* [55] y **transferiremos el conocimiento** desde la *Faster R-CNN InceptionV2 COCO*.

Se ha optado por usar la *Faster R-CNN* por delante de otras como la red YOLO [42] [41] debido a su integración con TensorFlow y nuestro conocimiento de este.

1.5.6. Introducción a los ataques adversarios

Es bien conocido por todos el algoritmo de entrenamiento de una red neuronal. En un primer momento introducimos un elemento de entrada en la red y realizamos los cálculos pertinentes para obtener su predicción de salida. A esta primera fase se le conoce como propagación hacia adelante (*forward propagation* en inglés).

Debemos definir un tamaño de *batch* N como un conjunto de N imágenes. Se calcula el error para cada uno de los elementos del *batch* y, con el error conjunto, se realiza la actualización de los pesos de la red. Para ello, calculamos **el gradiente** como la derivada multivariable de la función de coste con respecto a todos los parámetros de la red. Sin embargo, como tenemos multitud de parámetros el cálculo del gradiente se complica bastante. Para ello utilizamos el algoritmo de ***back-propagation*** o propagación hacia atrás dónde comenzamos calculando las derivadas parciales de la función de coste con respecto a la última capa y, este resultado, lo vamos propagando hacia atrás para acabar optimizando todos los parámetros de la red. En este caso **el problema de optimización es un problema de minimización** dónde buscamos minimizar el error de la red por lo que restaremos el vector gradiente obtenido multiplicado por una tasa de aprendizaje, el *learning rate*[2]. El vector gradiente se resta porque este valor nos da la máxima pendiente y nuestra tarea es todo lo contrario, buscar una disminución.

Ahora bien, si se quisiera confundir a la red la primera opción sería cambiar alguno de esos parámetros. Esta aproximación es errónea ya que, la modificación de alguno de esos parámetros, podría suponer un cambio general en la red dejándola inutilizada.

Descartada la opción de modificar la red, la segunda opción que surge es la manipulación de los datos de entrada para obtener una salida incorrecta. Recordemos que la red realiza las predicciones en función de los valores de los píxeles de la imagen de entrada por lo que, si modificamos estos píxeles de manera adecuada, podríamos lograr el error en la red sin realizar ningún cambio en esta.

Ahora el nuevo problema de optimización que surge es cómo **manipular los píxeles de entrada para maximizar el error** dentro de la red neuronal. Sin embargo, tenemos que añadir además otra condición a este problema. Si modificamos demasiado los píxeles de la imagen de entrada, es posible que, aunque

[2]El learning rate o tasa de aprendizaje es un parámetro utilizado en el entrenamiento que determina la velocidad de aprendizaje de la red neuronal (ratio de modificación de los pesos por iteración). Un valor muy pequeño ralentiza el proceso de aprendizaje (llegando incluso a bloquearse la red) mientras que un valor muy grande puede hacer que la red no converja nunca. La elección de su valor es realmente importante para un correcto funcionamiento.

la red falle, la imagen diste demasiado de la entrada original. Por ello, debemos añadir la **condición de que la diferencia entre la imagen original y la imagen modificada sea mínima**.

Obteniendo así el siguiente problema:

$$ImagenModificada \rightarrow MaximizarError + MinimizarDiferencias$$

Al ser un problema de optimización recurrimos otra vez al **cálculo del gradiente**. Aquí buscaremos **la perturbación que maximiza la función de coste (el error).**

En lenguaje matemático podríamos escribir el cálculo del gradiente como:

$$\bigtriangledown_x L(\theta, x, y)$$

Dónde x es la imagen de entrada, L la función de coste, θ los pesos de la red e y la salida que buscamos, que puede ser la salida real en ataques adversarios no dirigidos, o la salida que queremos obtener en ataques dirigidos

Todas estas técnicas se pueden ver en detalle en artículos como [9] [35] y [21] dónde incluso especifican que este tipo de técnicas se pueden llevar al mundo real puesto que **los ataques siguen siendo detectados por cámaras como la de un smartphone**. Es decir, **no son ilusiones creadas por computadoras** y podrían suponer fallos catastróficos en situaciones como la que estamos analizando.

1.6. Diferencias entre los ataques adversarios en redes de clasificación y en redes de detección

Los ataques en redes de clasificación y en redes de detección, aunque funcionan bajo la misma premisa, no son iguales. En las redes de clasificación únicamente obtenemos un resultado de salida por lo que, si se modifican los píxeles de manera correcta, siempre se logrará engañar a la red porque esta valora todos los píxeles como un conjunto.

Sin embargo, en redes de detección tal y como muestran trabajos como [23] y [57] al utilizar *bounding boxes* para la detección no es suficiente con modificar los píxeles de ese *bounding box* para realizar el ataque ya que, el entorno y los píxeles vecinos a los modificados, volverán a crear una caja de detección correcta en un lugar similar (aunque no igual) al *bounding box* atacado.

En este tipo de redes el ataque tiene que ir dirigido **a todos los píxeles incluidos en cajas de detección cuya probabilidad máxima sea la de la clase a atacar.** Por todo esto, aunque ambos ataques están relacionados y funcionan bajo la misma lógica, no podemos utilizar el mismo algoritmo de manipulación.

1.7. Ataques de caja blanca y de caja negra

Se definen como ataques de caja blanca a todos aquellos en los que tenemos acceso al modelo que queremos atacar y del que conocemos y podemos observar su funcionamiento interno. Todos los ataques y pruebas que vamos a realizar a lo largo de este trabajo serán ataques de caja blanca ya que nosotros mismos hemos creado las redes y, por lo tanto, tenemos control sobre ellas. Además, por las limitaciones temporales del TFG, no es posible profundizar en otros tipos.

Los ataques de caja negra son aquellos que se realizan sin tener acceso al modelo, únicamente podemos introducir unos datos de entrada (las imágenes) y obtener unos datos de salida (las probabilidades). Sin embargo, estudios como [34] especifican que los ataques adversarios también son posibles en "situaciones de caja negra".

En dicho artículo explican el procedimiento como:

1. **Creación de un modelo substituto**: Creamos una biblioteca de imágenes etiquetada a través de imágenes que son pasadas al modelo y anotando su salida como etiqueta. Con esta biblioteca entrenamos a la red.

2. **Atacamos el modelo substituto**: Generamos ataques adversarios como en un ataque de caja blanca con el modelo substituto.

Ya ha sido demostrado en [49] y [9] que los ataques se pueden transferir entre modelos. Es decir, aunque la arquitectura de la red varíe, **el ataque será efectivo en ambas si la finalidad de las redes es la misma.** Esta característica que hace posible también la realización de ataques adversarios en este tipo de escenarios los dota de un potencial enorme en cuanto a vulnerabilidades de seguridad.

Capítulo 2

Estado del arte

A día de hoy la generación de ejemplos adversarios está bastante avanzada. Desde la publicación del primer artículo [9] numerosos investigadores han ido explorando este ámbito extrapolando este algoritmo de generación de ataques en redes de clasificación a redes de detección o de segmentación.

Existen ya librerías como Cleverhans [33] que permiten la realización de ataques adversarios en redes de clasificación utilizando diferentes técnicas. Esta librería funciona bastante bien y, al ser una librería, aísla al usuario de la implementación. Esta librería incluye los ataques en redes de clasificación que vamos a desarrollar nosotros. Sin embargo, la finalidad de este trabajo es aprender y conocer el funcionamiento en bajo nivel de los ejemplos adversarios para poder desarrollar nuevos algoritmos propios en el futuro como el que desarrollaremos en la parte de detección.

Cleverhans también proporciona otro tipo de algoritmos para generar ataques adversarios diferentes a los aquí mencionados. Es decir, los algoritmos que vamos a implementar en este trabajo no son los únicos que existen para generar ejemplos adversarios pero, por falta de tiempo, debemos priorizar y no podemos abarcar todos.

En cuanto a los ataques en redes de detección, existen artículos como [23] y [57] que proponen algoritmos para la obtención de ejemplos adversarios en redes de detección. Sin embargo, estos algoritmos necesitan de la modificación de todos los píxeles de la imagen para lograr obtener un cambio en un determinado *bounding box* ya que, como especificamos previamente, el entorno influye en la detección de objetos.

Nuestra finalidad no es modificar todos los píxeles de una determinada imagen, **queremos modificar una señal de tráfico y que la detecte de manera incorrecta independientemente del entorno**. Por ello, hemos desarrollado

un algoritmo nuevo y diferente que soluciona esta problemática.

Por otra parte, también hay numerosos avances en artículos como [36] y [6] dónde generan parches para, al introducirlos en una imagen, que la red falle. Estos parches no son invisibles, esto implica que un usuario podría darse cuenta de que la imagen ha sido manipulada. Sin embargo, para el caso de las señales de tráfico podrían ser confundidos con *grafitis* y, en ejemplos como el primero de reconocimiento facial los parches podrían integrarse en complementos como gafas o collares y seguirían provocando un error en la red.

Figura 2.1: Parches en reconocimiento facial y señales de tráfico. Imágenes obtenidas de [36] y [6]

En dónde no existen tantos avances y artículos (aunque han ido proliferando en los últimos años) es en el ámbito de las **defensas contra estos ataques adversarios** ya que estos ejemplos están generados a partir del funcionamiento interno de las redes neuronales por lo que defenderse ante esto es realmente complicado.

Trabajos como [59] proponen la modificación de la función de activación de las neuronas para evitar la propagación del error a lo largo de la red y así evitar que pequeñas variaciones logren grandes cambios. Otros trabajos como [22] y [45] proponen la generación de un modelo generativo dónde tenemos un generador encargado de crear ataques adversarios y un clasificador entrenado con estos ejemplos adversarios pero con la etiqueta correcta.

Capítulo 3

Materiales

3.1. Entorno

El entrenamiento de redes neuronales profundas requiere gran potencia de cómputo para poder obtener los resultados en un tiempo de procesamiento razonable. Esta condición inicial implica directamente la necesidad de conectarse con un servidor de cómputo externo dotado de una potente GPU que realice las labores de entrenamiento previamente mencionadas.

3.1.1. Google Colaboratory

Por ello, hemos decidido utilizar el servicio de computación que nos ofrece Google a través de su plataforma Google Colaboratory [10].Este entorno de computación nos ofrece de manera gratuita GPUs para entre, otras muchas cosas, entrenar redes neuronales.

Sin embargo, las GPUs de la versión gratuita no son lo suficientemente potentes para nuestro propósito por lo que hemos decidido adquirir la versión PRO a través de una subscripción mensual de Google Colaboratory por 9.99$/mes [11].

En esta versión de pago las GPUs no son fijas y dependen de la carga útil de los servidores en ese momento. También podremos acceder a servidores con mucha más memoria RAM que en la versión gratuita pasando de 12GB a 25GB de memoria RAM útil.

3.1.2. GPUs Utilizadas

Al utilizar el entorno de Google Colaboratory PRO la GPU utilizada no es fija y depende de la carga actual de los servidores. Sin embargo siempre será uno de estos 3 modelos:

1. **Nvidia Tesla K80** [28]: Que permite hasta 2.91 teraflops de rendimiento en operaciones de precisión doble[1], hasta 8.73 teraflops de rendimiento en operaciones de precisión simple [2],25GB de memoria GDDR5 y 480GB/s de ancho de banda para el cálculo de las operaciones

2. **Nvidia Tesla T3** [30]: 8.1 teraflops en precisión doble, 16GB de memoria GDRR6 y 320GB/s para nuestro proyecto.

3. **Nvidia Tesla P100** [29]: 5.3 teraflops en precisión doble, 10.6 teraflops en precisión simple, 16GB de memoria GDRR6 y 732GB/s para nuestras operaciones.

Estas serán las GPUs utilizadas a lo largo de todo el proyecto tanto para el entrenamiento de la red, los ataques y las pruebas.

3.2. Lenguaje de programación

El lenguaje de programación utilizado para el desarrollo de la aplicación fue Python [38]. Nos acabamos decantando por este lenguaje debido a que es realmente fácil de aprender y desarrollar aplicaciones de deep learning con él por la cantidad de librerías de aprendizaje máquina y de tratamiento de imágenes que nos ofrece y que luego comentaremos. Además, este lenguaje es multiplataforma permitiéndonos el desarrollo y ejecución del proyecto que vamos a desarrollar en diferentes sistemas.

Otros lenguajes de programación que podríamos utilizar serían Matlab o R. Sin embargo, decidimos optar por Python debido a las bibliotecas Tensorflow y Keras que posteriormente comentaremos.

3.3. Tensorflow y Keras

Tensorflow [54] es una biblioteca de código abierto desarrollada por Google que permite crea grafos de flujos de datos. Estas estructuras describen cómo se mueven los datos a través de una serie de nodos de procesamiento donde cada nodo representa una operación matemática y cada conexión entre nodos está formada por una matriz de datos multidimensional (tensor).

Tensorflow nos proporciona todo esto a través del lenguaje Python dónde los nodos y tensores del grafo son objetos Python y las aplicaciones desarrolladas en

[1]Los FLOPS, del inglés *(floating point operations per second)* son una medida de rendimiento utilizada en GPUS y CPUS.

[2]Los números en formato precisión simple ocupan 32 bits en memoria mientras que los números en formato precisión doble ocupan 64 bits en memoria.

Tensorflow son, básicamente, programas Python.

Sin embargo, las operaciones matemáticas de bajo nivel no se realizan en Python. Tensorflow realiza transformaciones en archivos binarios de C++ de alto rendimiento, aumentando así la velocidad de cómputo y abstrayendo al desarrollador de todo esto.

Así, la mayor ventaja de Tensorflow es que proporciona para el desarrollo de aplicaciones de aprendizaje automático es la abstracción. Al utilizar Tensorflow sólo hay que definir la lógica general de la implementación sin centrarse en las capas más bajas (creación de una neurona por ejemplo).

En cuánto a Keras, es una API[3] de aprendizaje profundo escrita en Python y se ejecuta por encima de Tensorflow. En la figura 3.1 se puede observar el *pipeline* de ejecución de las aplicaciones.

Keras nos permite crear redes neuronales profundas en muy pocas líneas de código y con una sintaxis de alto nivel, por lo que resulta realmente cómoda de cara a implementar redes neuronales profundas.

Se escogieron estas dos tecnologías para el desarrollo de las redes neuronales por delante de Caffe2 o Pytorch debido a que ya se contaba con conocimiento previo de Tensorflow simplificando así el desarrollo del proyecto. Además, Tensorflow cuenta con una API para detección de objetos que posteriormente utilizaremos en el desarrollo de la red de detección.

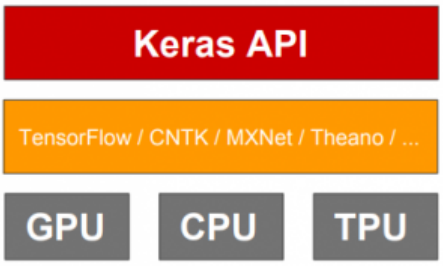

Figura 3.1: Pipeline GPU, Keras y Tensorflow. Imagen obtenida de [48].

[3]API siglas de *'Application Programming Interface'* hace referencia a los procesos y métodos que nos otorga una biblioteca de programación como abstracción para ser utilizada por otros programas.

3.4. Bibliotecas de imágenes

3.4.1. Belgium Traffic Sign Dataset

Para realizar el entrenamiento de nuestra red neuronal profunda de clasificación utilizaremos como biblioteca de imágenes el *Belgium Traffic Sign Dataset* utilizada en el Twentieth International Conference on Pattern Recognition [37] y a la que podemos acceder a través de su web pública [39] que cuenta con varias bibliotecas de imágenes con sus respectivas anotaciones tanto para clasificación como para redes de detección de objetos.

En nuestro caso nos centraremos en la biblioteca de imágenes para clasificación donde nos ofrece ya divididos dos conjuntos de datos, uno de entrenamiento y otro de validación clasificando las señales en las 62 clases que se pueden ver en la figura 3.2. Entre paréntesis tenemos el número de elementos del conjunto de validación por clase.

Figura 3.2: Ejemplo de las 62 clases que contiene el *Belgium Traffic Sign Dataset*. Imagen obtenida de [4].

Esta biblioteca de imágenes es de **uso público** y contiene una gran cantidad de imágenes. Contamos con 4638 imágenes en el conjunto de entrenamiento y 2583 en el conjunto de validación.

3.4.2. German Traffic Sign Detection Dataset

En la red de detección de señales de tráfico utilizaremos otra biblioteca de imágenes. En concreto utilizaremos el *German Traffic Sign Detection Dataset* que fue presentado en el International Joint Conference on Neural Networks de 2013 [15]. Podemos acceder a ella a través de su web de **uso público**[16]

Esta biblioteca de imágenes nos presenta 42 clases diferentes de señales de tráfico a detectar. Sin embargo, hemos decidido reducirlas a únicamente 5 clases por los siguientes motivos:

1. **Potencia de cómputo**: El entrenamiento de la red neuronal de detección requiere mucho más tiempo si aumentamos el número de clases. Al disponer de un tiempo bastante limitado, unido al acceso de pago a las GPUs, y poder analizar la problemática de igual forma, reducir el número de clases es la opción más óptima.

2. **Señales en el centro de Santiago**: Las pruebas que vamos a realizar serán sobre señales de tráfico vistas y fotografiadas en el centro de Santiago de Compostela por lo que, señales como: peligro curva, precaución nieve, camino de animales, etc. no se visualizan.

3. **Señales más importantes**: Además, las categorías de señales escogidas finalmente son las necesarias para conducir por el centro de Santiago, la modificación de cualquiera de ellas provocaría situaciones muy peligrosas.

Así pues, las 5 clases finales estarían conformadas por las siguientes etiquetas de la biblioteca de imágenes original:

1. **Límite de velocidad**: Las etiquetas de la biblioteca original (0,1,2,3,4,5,7,8) dónde se agrupan diferentes niveles de límite (30-50-100...) pasan a formar parte de una única clase, la clase 1. En esta clase contamos con 95 fotos en total dónde 76 serán usadas para el entrenamiento y 19 fotos serán usadas en la validación.

2. **Ceda el paso**: Las etiquetas originales con el número 13 serán ahora clase 2. En esta clase tenemos 82 fotos en total donde 66 se utilizan en el conjunto de entrenamiento y 16 en la validación.

3. **Sentido obligatorio**: Las etiquetas (33,34,35,36,37,38,39) dónde las distintas clases indican la dirección de la flecha pasan a ser ahora clase 3. En esta clase tenemos 128 fotos en total donde 102 se utilizan para el entrenamiento y 26 para validación.

4. **Prohibido el paso**: La etiqueta 17 pasa a ser ahora clase 4. Contamos con 28 fotos, 24 para el entrenamiento y 4 para la validación.

5. **Stop**: La etiqueta 14 ahora es clase 5. Tenemos 32 fotos dónde 26 son de entrenamiento y 6 son fotos del conjunto de validación.

Nótese que se mantiene una proporción para cada clase dónde el **80 %** de las imágenes de una misma clase pertenecen al conjunto de entrenamiento y, el **20 %** restante, pertenecen al conjunto de validación.

Las anotaciones para cada clase del conjunto descargado inicial son, por tanto, manipuladas para que sigan la lógica previamente descrita. El resto de clases proporcionadas quedan eliminadas del archivo de anotaciones.

Figura 3.3: Ejemplo de las 5 clases de objetos que detecta la red.

3.5. Visualización

Las herramientas de visualización de imágenes que vamos a utilizar tienen que estar basadas en Python. Usaremos principalmente la librería de uso libre OpenCV [31]. Esta librería nos otorga multitud de funciones a la hora de manipular y pintar imágenes en pantalla. Es posible que, en algún caso, se produzcan incompatibilidades con algunas matrices de píxeles y sus rangos por lo que también podremos recurrir a la librería Matplotlib [25] en caso de ser necesario.

Centrándonos en las herramientas de análisis de la red neuronal utilizaremos TensorBoard [52] debido a que está especialmente diseñada para trabajar con Tensorflow. Esta herramienta nos permite inspeccionar los grafos generados por Tensorflow y nos proporciona gráficas del estado de la red a lo largo del entrenamiento

3.6. Gestión de la configuración

Se decide utilizar una cuenta de Google Drive [5] dónde almacenar las bibliotecas de imágenes. Esta decisión se toma debido a su gran compatibilidad con en el entorno de desarrollo Google Colaboratory (se puede acceder al almacenamiento en el propio cuaderno) y a que únicamente almacenaremos allí las bibliotecas

dónde las imágenes serán estáticas, por lo que no necesitamos un control de versiones sobre ellas.

Para el control de versiones de los cuadernos utilizamos el propio sistema de Google Colaboratory que permite acceder a todas las versiones anteriores del cuaderno. Con este método no tenemos la necesidad de utilizar otras herramientas como Github.

3.7. Cálculo del gradiente

A lo largo de todo el trabajo haremos alusiones constantes al cálculo del gradiente. Matemáticamente podemos definir el gradiente como:

$$\bigtriangledown f(r) = \left(\frac{\delta f(r)}{\delta x_1} ... \frac{\delta f(r)}{\delta x_n}\right)$$

Es decir, la derivada parcial de la función con respecto a todas las variables de esa función. El gradiente es y será utilizado en problemas de optimización ya sea para maximizar o minimizar valores tales como el error o el ruido aplicado a una imagen.

Capítulo 4

Métodos

4.1. Data Augmentation

Data Augmentation o aumento de datos en castellano, es una técnica utilizada para hacer crecer nuestra librería de imágenes a partir de las imágenes ya existentes. Para ello realizaremos ciertas operaciones de transformación en las imágenes originales de la librería para obtener nuevas imágenes que también utilizaremos en el entrenamiento.

En concreto, para el entrenamiento de la **red de clasificación** utilizaremos las siguientes operaciones:

1. **Rotación 30º**: Rotaremos las imágenes 30 grados en sentido anti-horario.
2. **Desplazamientos aleatorios en vertical y en horizontal**: Las imágenes no estarán siempre centradas con respecto al eje vertical y horizontal.
3. **Inclinación de la imagen**: Estiramos la imagen fijando un cierto eje.
4. **Zoom**: Alejamos y acercamos el centro de la imagen.
5. **Giro horizontal**: La imagen se girará en modo espejo.

Para la **red de detección** también utilizaremos esta técnica. Sin embargo, utilizaremos otras transformaciones:

1. **Giro horizontal**
2. **Saturación aleatoria**: Se cambia el valor de saturación de la imagen.
3. **Cambio en los píxeles**: Multiplicación de los canales de los píxeles de una imagen por una constante, modificándolos.

Ejemplos de todas estas transformaciones se pueden ver en el Apéndice A [Apéndice B]

4.2. Función de coste y métrica de validación de la red de clasificación

Para el entrenamiento de la red neuronal de clasificación debemos definir una función de coste que vamos a optimizar (minimizar) durante en entrenamiento. Hemos escogido la *Categorical CrossEntropy* definida como:

$$CE = -log\left(\frac{e^{s_p}}{\sum_j^C e^{s_j}}\right)$$

Donde s_p es la probabilidad de la clase correcta, y s_j es la probabilidad de la clase j.

Por definición, esta función de coste es realmente efectiva cuando las imágenes pertenecen **únicamente a una clase del modelo**, en otro caso sería necesario emplear una variante multi-etiqueta.

Para medir el rendimiento de esta red utilizaremos el *accuracy* que se define como:

$$accuracy = \frac{aciertos}{(aciertos+errores)}$$

El número de aciertos de la red frente al número de aciertos y errores (imágenes totales). De manera trivial nótese que este valor estará entre 0 y 1 y representa el % de aciertos de la red.

4.3. Función de coste y métrica de validación de la red de detección

La función de coste en una red de detección es más compleja. Es necesario tener en cuenta el coste de la salida desde dos puntos de vista: las probabilidades de que la imagen esté correctamente clasificada y que el *bounding box* que representa esa probabilidad sea similar al *bounding box* verdadero definido previamente ya que, **para el entrenamiento de la red de detección, no otorgamos una clasificación global** sino que indicamos 4 coordenadas que especifican dónde se encuentra el objeto y, ese objeto, a qué clase pertenece.

Así, la función de coste quedaría definida como:

$$L(p_i, t_i) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i*) + \lambda \frac{1}{N_{reg}} \sum_i p_i * L_{reg}(t_i, t_i*)$$

Dónde p_i es la probabilidad predicha para el objeto i, p_i* la probabilidad de la real, N_{cls} sirve para normalizar el valor y es el tamaño de batch de entrenamiento. Este tamaño de batch viene definido por el número de regiones de interés especificadas, por lo que nuestro valor será de 256 que es el utilizado por defecto. N_{reg} normaliza el segundo parámetro de la suma y está definido por el número de *anchors*, 2400 por defecto. El valor de λ sirve para balancear ambos lados de la suma y será de 10.

$$L_{cls}(p, u) = -logp_u$$

Dónde p_u es la probabilidad de que pertenezca a la clase u siendo esta la clase correcta

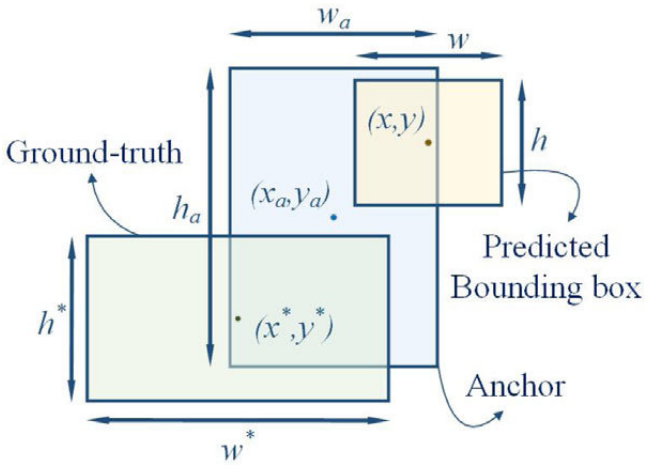

Figura 4.1: Anchor, bounding box real y bounding box predicho. Imagen obtenida de [2].

Ajustamos ahora los valores de cada *anchor* haciendo una **regresión** definida como:

$$\begin{aligned} t_x &= (x - x_a)/w_a \\ t_y &= (y - y_a)/h_a \\ t_w &= log(w/w_a) \\ t_h &= log(h/h_a) \end{aligned} \tag{4.1}$$

(x,y) es la coordenada de la esquina superior izquierda, **w** es el ancho y **h** es el alto del *bounding box* predicho. El **recuadro amarillo** en la figura 4.1.

Y, de igual forma, para **(x*,y*,w*,h*), recuadro en color verde**, las coordenadas del *bounding box* real. Aquí x_a e y_a **el ancho y alto del *anchor*, en color azul**. Esto provoca una normalización del error sin importar el tamaño del *anchor*.

$$L_{reg}(t^u, v) = \sum_{i \epsilon x,w,w,h} smooth(t_i^u - v_i)$$

Siendo t el valor previamente calculado y v el valor del *bounding box* correcto, en el entrenamiento está definido en el fichero utilizado previamente descrito. Nótese que se aplica la función sumatorio para cada una de los valores del *bounding box*.

$$smooth(x) = \left\{ \begin{array}{c} 0{,}5x^2, si\,|x| < 1 \\ |x| - 0{,}5, si\,|x| \geq 1 \end{array} \right\}$$

En cuánto a los métodos matemáticos utilizados **para la validación de la red de detección** utilizaremos la métrica del ***mAP(mean Average Precision)***. Esta métrica necesita del **IoU (Intersection over Union)**. Esta medida calcula cómo de preciso es el *bounding box* de detección con respecto al real, para ello divide el área de la intersección de ambos entre el área de la unión de ambos. Esto puede verse de forma visual en la figura 4.2

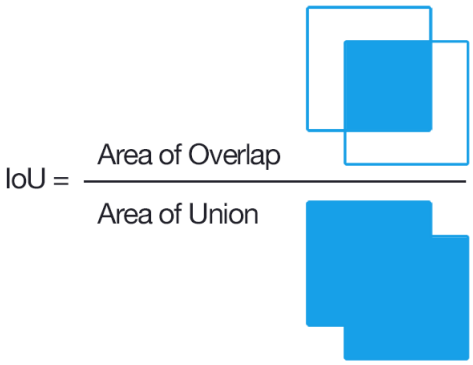

Figura 4.2: IoU. Imagen obtenida de [44].

Nótese que si ambos son iguales el valor será de 1 (o 100 %) y si no tienen nada en común será de 0 ya que no habrá intersección.

Además el *mAP* necesita definir 4 estados de detecciones posibles:

1. **Verdaderos positivos(TP)**: Detecciones de objetos en su categoría correcta y con un IoU $\geq$ un umbral y cuya clase predicha es correcta.

2. **Falsos positivos(FP)**: Detecciones incorrectas, bien porque el objeto no pertenece a la categoría seleccionada o bien porque aunque pertenezca no ha sido detectado con un IoU $\geq$ umbral.

3. **Verdaderos negativos(TN)**: Todos los objetos que no debemos detectar y no detectamos. Esta métrica es inútil para el mAP ya que no se utiliza en el cálculo.

4. **Falsos negativos(FN)**: Objetos que debemos detectar y no detectamos.

Aquí el umbral sería un valor establecido por nosotros para el cálculo de la métrica. A mayor valor más exhaustiva es la métrica ya que nuestra red debe ser mucho más precisa en las detecciones. Lo normal es utilizar los valores de las COCO Metrics, **0.5** y desde 0.5 hasta 0.95 con un paso de 0.05 **(0.5:0.95)** siendo la segunda mucho más exhaustiva que la primera.

Definido todo esto, ordenamos las detecciones de la red por categoría y, dentro de cada categoría, las ordenamos de mayor a peor probabilidad. Definimos entonces 2 nuevas medidas:

1. **Precision**: $\frac{TP}{(TP+FP)}$

2. **Recall**: $\frac{TP}{(TP+FN)}$

Nótese que la precisión mide exactamente eso, la precisión que tiene la red a la hora de clasificar, y el *Recall* mide el porcentaje de aciertos con respecto al número total de elementos que tiene que detectar (denominador constante).

Nótese también que, en casos dónde no se detecten todos los objetos, el *Recall* nunca llegará a 100 % y, que los casos de falsos positivos, disminuyen la Precisión pero no aumentan el *Recall*.

Con todo esto, calculamos los valores de estas medidas para cada valor de una clase empezando por los valores con probabilidad mayor obteniendo una gráfica con el *Recall* en el eje de abscisas y la precisión en el eje de ordenadas (realizada en el apartado de pruebas). Así, **el valor de la métrica AP** (*Average Precision* o precisión media en castellano) **sería el área bajo esa gráfica** que, como es discreta, se puede calcular a través del ancho y alto de los rectángulos que la forman. Esto se refleja en la figura 4.4.

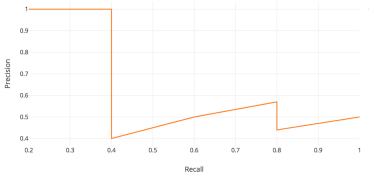

Figura 4.3: Gráfica de ejemplo para el cálculo del AP con el *Recall* en el eje de abscisas y la precisión en el eje de ordenadas. Imagen obtenida de [17].

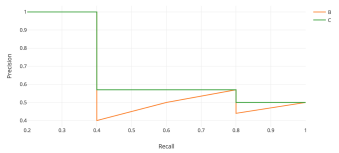

Figura 4.4: Rectángulos para el cálculo del área bajo la curva. Imagen obtenida de [17].

Una vez obtenido el valor del AP para cada clase realizamos la **media aritmética del AP de todas las clases**, este resultado será nuestro $\boldsymbol{mAP}$.

4.4. Ataques adversarios no dirigidos en la red de clasificación

Un ataque adversario no dirigido consiste en encontrar la perturbación que maximiza la función de coste y, por ende, maximiza el error. Para la implementación de los ataques adversarios no dirigidos recurriremos al artículo de Ian Goodfellow [9] e implementaremos su *Fast Gradient Sign Method*.

Para ello recurriremos al cálculo del gradiente de la imagen de entrada con respecto a la función de coste y limitaremos el resultado dentro de un rango de valores para que la imagen obtenida no diste demasiado de la imagen original. Como función de coste vamos a utilizar la *Categorical CrossEntropy* definida previamente en el trabajo. Operamos tal que así:

$$x_0^{adv} = x$$

$$x^{adv}{}_{N+1} = Clip\left\{x_N^{adv} + \varepsilon * signo\left\{\bigtriangledown_x L(\theta, x, y_{verdadera})\right\}\right\}$$

$$signo(x) = \begin{Bmatrix} 1, x \geq 0 \\ -1, x < 0 \end{Bmatrix}$$

Donde x es la imagen de entrada, $y_{verdadera}$ la etiqueta correcta predicha por la red, L la función de coste, θ los pesos de la red, ε un hiperparámetro que podremos modificar y que ahora analizaremos y $Clip$ una función que limita la diferencia entre los píxeles nuevos y los píxeles de la imagen original x

4.4.1. Análisis de los valores del hiperparámetro

En el artículo original publicado por Ian Goodfellow no se propone un método iterativo de cálculo de ejemplos adversarios y utiliza la variable ε como hiperparámetro que controla la variación de la imagen adversaria con respecto a la imagen original. De esta manera, cuanto más grande es ε **mayor es la probabilidad de conseguir un ataque adversario** en un único paso pero**, a cambio, mayor es la diferencia entre la imagen original y la conseguida**. Todo esto se puede ver reflejado en la figura 4.5.

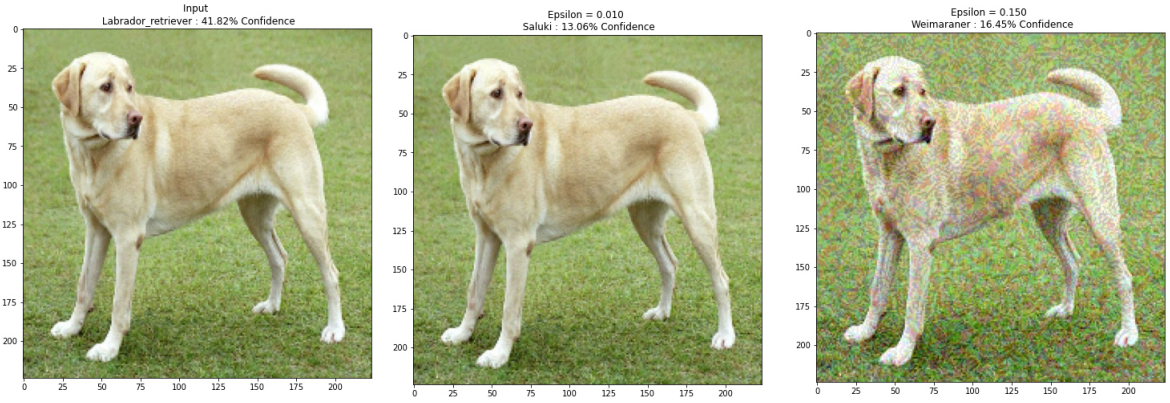

Figura 4.5: Cómo afecta la modificación del hiperparámetro. Imagen obtenida de [53].

Sin embargo, nuestro planteamiento será algo diferente. Fijaremos ε como 0,01 y utilizaremos una función *Clip* para limitar la diferencia entre la imagen original y la imagen adversaria en 0,2 para cada uno de los canales (RGB) de manera que un píxel adversario no puede diferenciarse en $\pm 0,2$ de un píxel original.

4.5. Ataques adversarios dirigidos en la red de clasificación

Para los ataques adversarios dirigidos el planteamiento es un poco diferente. En este caso buscamos maximizar la probabilidad de que la imagen sea clasificada dentro de la categoría escogida.

Para ello, no definiremos una función de coste como anteriormente, **nuestra función de coste será la probabilidad de que la imagen sea de la clase pedida** y buscaremos maximizar este resultado. Operamos tal que:

$$x_0^{adv} = x$$

$$x^{adv}{}_{N+1} = Clip\left\{x_N^{adv} + \bigtriangledown_x L(x, y_{ataque})\right\}$$

Donde x es la imagen de entrada, y_{ataque} la etiqueta que queremos obtener, L la probabilidad de que la imagen x sea de la clase y_{ataque} y $Clip$ una función que limita la diferencia entre los píxeles nuevos y los píxeles de la imagen original x.

En este caso, la función *Clip* permitirá, como máximo $\pm 0,01$ de diferencia entre la imagen original y la imagen atacada para hacerlo, aún más si cabe, invisible al ojo humano.

4.6. Ataques adversarios dirigidos en la red de detección

En este apartado hemos tenido que **desarrollar un algoritmo propio** que se ajustara a nuestras necesidades. Partimos de la base especificada en [57] y adaptamos la teoría a nuestro ejemplo.

En [57] proponen el *Dense Adversary Generation (DAG)*, un algoritmo que ataca a todos los *bounding boxes* que pertenezcan a la clase objetivo modificando los píxeles de la imagen en función del gradiente. Es necesario realizar un ataque a todos los *bounding box* ya que, como comentamos anteriormente, los píxeles vecinos ayudan a corregir la predicción en caso de ataque adversario en una única caja de detección.

Sin embargo, este algoritmo propone la modificación de todos los píxeles de la imagen para el ataque de un determinado objetivo. Nosotros no queremos modificar el entorno, queremos modificar únicamente la señal de tráfico, para que la red falle, independientemente del entorno. Para ello operamos tal que así:

Denotamos x_{ataque} como la parte de la imagen perteneciente al ***bounding box* con mayor puntuación inicial** y x como la imagen inicial Denotamos $T = \{t_1, t_2, ..., t_N\}$ al conjunto de *bounding boxes* que queremos atacar, los que son de la clase inicial, $L = \{l_0, l_1 ..., l_N\}$ como el conjunto de etiquetas que vamos a modificar y denotamos $L' = \{l'_0, l'_1 ..., l'_N\}$ como el conjunto de etiquetas objetivo. En nuestro algoritmo sólo nos centraremos en una clase de ataque por lo que los conjuntos L y L' sólo tendrán un elemento. $f(x, ln)$ es la probabilidad de que la imagen x sea de la clase n.

Así pues, computaremos el ruido añadido a la imagen como:

$$x_0 = x$$

$$r = \sum_{i=0}^{N} [\bigtriangledown x f(x_m, l'_n) - \bigtriangledown x f(x_m, l_n)]$$

Es decir, buscaremos todos los *bounding box* cuya probabilidad máxima sea la etiqueta inicial l (en la práctica también incluiremos aquellos que la probabilidad de que la clase l sea mayor de 0,1). Para cada uno, calcularemos su gradiente con respecto a la imagen en la etapa m. También calcularemos el gradiente entre la imagen en la etapa m y la etiqueta atacante l'. Así pues, **nótese, que podemos aumentar la probabilidad de que la *bounding box* sea de la clase l', aumentándola directamente o reduciendo que sea de la clase l**

Ahora normalizamos el valor del ruido puesto que, al utilizar el cálculo del gradiente, este irá creciendo a medida que el coste baje. γ es un hiperparámetro

que fijaremos en 0.8. Esta operación no sería necesaria en nuestra implementación ya que limitamos el valor de perturbación a través de una función clip. Sin embargo, la normalización favorece el tiempo que tarda el algoritmo en converger.

$$r' = \frac{\gamma}{\|rm\|} r$$

Finalmente:

$$x_{ataque} = Clip\{Focalizar\{x_{ataque}, x_m + r'\}\}$$

Dónde $Focalizar$ es una función que sólo permite sumar el gradiente en los píxeles que pertenecen a la imagen ataque, es decir, solo modificamos la señal y $Clip$ es la función que limita la perturbación de los píxeles de la imagen.

Capítulo 5

Pruebas y discusión de resultados

5.1. Validación y test en la red de clasificación

Para el entrenamiento de la red de clasificación, no podemos seguir al pie de la letra el artículo previamente mencionado [24] ya que no especifican el parámetro de *batch-size* utilizado y, por lo tanto, no podemos seguir sus valores de *learning rate* ya que estos van muy ligados al tamaño de *batch*. Dada la definición de *batch size* y de *learning rate*, si queremos mantener el mismo cambio en los pesos de la red neuronal al aumentar el tamaño de lote, debemos modificar también el valor de learning rate en su parte proporcional. Utilizaremos el conjunto de validación de la red para tomar decisiones de los valores de *learning rate* y *batch size*.

Además, trabajos como el [14] muestran mejores resultados de entrenamiento con tamaños de *batch* más grandes. Estudios como el [20] recogen que, al **multiplicar el *batch size* por** k, debemos multiplicar el l*earning rate* **por la raiz cuadrada de** k.

Por ello, vamos a partir de 3 *learning rates* diferentes: *0,1, 0,01, 0,001* y de un *batch size* de 32. Entrenaremos a la red durante 30 épocas[1] y nos quedaremos con la época que más precisión tenga con respecto al conjunto de validación ya que, a partir de cierto punto, la red empieza a reconocer mejor el conjunto de entrenamiento y peor el de validación (no generaliza). A este fenómeno se lo conoce como sobre-entrenamiento u *overfitting* en inglés y debemos evitarlo. Finalmente, aplicando la fórmula previamente descrita, escogeremos el *learning rate* con mejor resultado y alargaremos el tamaño de *batch* hasta 128.

Otro de los factores importantes que debemos comentar en el entrenamiento es que el tamaño de las imágenes de entrada debe ser de 299x299 píxeles tal

[1]Una época finaliza cuando ha pasado todas las imágenes de la biblioteca de imágenes de entrenamiento y se han realizado todas las actualizaciones de los pesos en función de estas.

y como se especifica en la documentación de *Inception V3*. Sin embargo, contamos con la función de keras: *tf.keras.applications.inception_v3.preprocess_input* que nos preprocesa imagenes de cualquier tamaño a los requisitos de la red.

Los resultados obtenidos se reflejan en el cuadro 5.1.

	Coste E	Precisión E	Coste V	Precisión V
Learning rate: 0,1	0,0141	0,9967	0,1310	0,9810
Learning rate: 0,01	0,0085	0,9974	0,0955	0,9861
Learning rate: 0,001	0,1355	0,9620	0,2192	0,9417

Cuadro 5.1: Resultados del conjunto de entrenamiento (E) y del conjunto de validación (V).

Por lo que, utilizaremos el LR 0.01. Pasaremos a un batch size de 64 ($32 * 2$), por lo que tendremos que multiplicar el LR por 1.414 ($\sqrt{2}$) obteniendo un **LR de 0.014**.

Finalmente los resultados con un **LR de 0.014** y un **batch size de 64** son los reflejados en el cuadro 5.2:

	Coste E	Precisión E	Coste V	Precisión V
Learning rate: 0,014	0,0037	0,9985	0,0892	0,9837

Cuadro 5.2: Resultados del conjunto de entrenamiento (E) y del conjunto de validación (V) para el learning rate final.

Realmente el resultado es muy similar al del LR de 0.01 pero, como se ha demostrado anteriormente en [14], el entrenamiento generaliza mejor a mayor tamaño de *batch* por lo que optaremos por este resultado.

5.1.1. Conjunto de test en la red de clasificación

En cuánto al conjunto de test, un conjunto totalmente separado del entrenamiento y que sirve para valorar la eficacia de nuestra red, hemos optado por crear una librería de imágenes propia. Seleccionaremos 1 o 2 imágenes de cada clase para tener una muestra de todas las clases y utilizaremos este conjunto para valorarla. Un ejemplo de estas imágenes se puede ver en la figura 5.1. Se pueden ver más en el apéndice B.

Figura 5.1: Imagen de la clase 37 del conjunto de test de la red de clasificación.

Como resultado la red acierta 88 imágenes de 108 imágenes totales **obteniendo así una precisión del 81.481 % para el conjunto de test**

5.2. Validación y test en la red de detección

Para el entrenamiento de la red de detección hemos escogido un tamaño de *batch size* de 1 porque en este tipo de redes, dada su arquitectura, el reconocimiento de cada imagen ocupa mucho más espacio en memoria y no disponemos de tantos recursos.

En cuánto al valor del *learning rate* es variable. Entrenaremos a la red durante 100.000 iteraciones, al inicio el *learning rate* será de 0,0002 y en la iteración 60.000 pasará a ser de 0,00002, este *learning rate* variable buscará generalizar mucho al principio y acabar de ajustar (con un *learning rate* más pequeño) en etapas posteriores.

Para el cálculo de la métrica del *mAP* contamos con una herramienta de la API de Tensorflow que nos la calcula directamente pero sin darnos las gráficas de precisión y Recall correspondientes. Nosotros implementaremos estas gráficas de para el conjunto de test. Así, en el conjunto de validación obtendríamos el *mAP* visible en el cuadro 5.3:

	mAP
IoU=0.50:0.95	0,580
IoU=0.50	0,926

Cuadro 5.3: Resultado del mAP en el conjunto de validación

Un mAP de 0.58 para el IoU desde 0.5 a 0.95 con paso 0.05 y un mAP de 0.926 para un IoU de 0.5.

5.2.1. Conjunto de test en la red de detección

Para la creación del conjunto de test en la red de detección hemos realizado fotografías en las calles de Santiago de Compostela y creado las anotaciones correspondientes de manera manual. Un ejemplo de imagen de este conjunto se puede ver en la figura 5.2. Más ejemplos del conjunto de test de detección se pueden ver en el apéndice B.

Figura 5.2: Imagen del conjunto de test en la red de detección.

Para el cálculo del valor del *mAP* en este conjunto hemos desarrollado un código que nos permita visualizar las gráficas de *AP* para mostrarlas en la memoria y calcular, a partir de estas, el valor del *mAP*

En primer lugar, una vez ordenadas las detecciones de cada clase por probabilidad, existen numerosas detecciones (con variación de muy pocos píxeles en la *bounding box*) que corresponden a la misma detección. Esto es provocado por la propia definición de los *anchors*. Para ello, comenzaremos aplicando una técnica conocida como *Non-maximum Suppression (NMS)* [19]: partimos de las detecciones ordenadas por probabilidad y, para cada imagen, si la clase de detección es la misma comprobamos la superposición de los *bounding boxes* detectados. Si **esta superposición es mayor de 0.6** entonces nos encontramos con que la detección es la misma y únicamente nos quedamos con la detección cuya probabilidad es mayor. Esto se puede observar en la figura 5.3

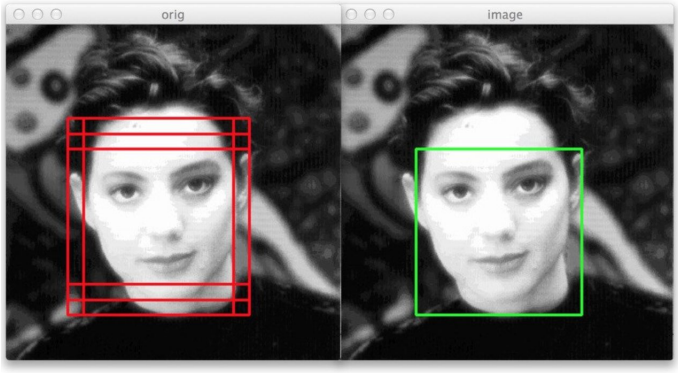

Figura 5.3: Antes y después de aplicar el NMS. Imagen obtenida de [1]

Esta operación ya está implementada en el cálculo automático de la API de Tensorflow obtenido en el apartado anterior pero, en este caso, debemos implementarlo a mano.

Una vez aplicado el algoritmo NMS, generaremos las gráficas de AP para cada clase:

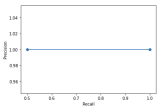

Figura 5.4: AP de la clase 1

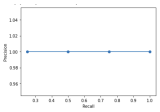

Figura 5.5: AP de la clase 2

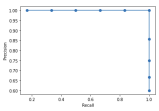

Figura 5.6: AP de la clase 3

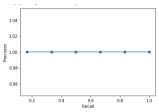

Figura 5.7: AP de la clase 4

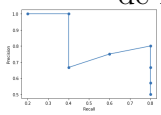

Figura 5.8: AP de la clase 5

Así el AP sería el área bajo la curva, para la clase 1 sería de 1, para la clase 2 sería de 1, para la clase 3 sería de 1, para la clase 4 sería de 1 y para la clase 5 sería de 0.771.

El valor del mAP sería, por tanto la media aritmética de todos ellos obteniendo **un mAP de 0.9542 en el conjunto de test**

5.3. Ataques adversarios no dirigidos

5.3.1. Implementación en Python

El procedimiento empieza por comprobar la etiqueta inicial de la imagen (la correcta). Luego generamos un tensor de la imagen que modificaremos y creamos los límites de las perturbaciones, por arriba y por abajo

Posteriormente obtenemos el tensor de predicciones (probabilidad de todas las clases) y creamos un tensor *one-hot* (un 1 en el lugar indicado de la etiqueta y 0s en el resto) y calculamos el coste entre las predicciones totales y la "perfección"que es el tensor *one-hot* con la *Categorical Cross Entropy*

Calculamos el gradiente y obtenemos su símbolo, lo evaluamos para transformarlo en un array (pasar de tensor a array) y aplicamos la formula previamente descrita.

Finalmente limitamos las perturbaciones con la función clip tanto en el rango de perturbación como en el rango máximo de valor de imagen (-1,1). Iteramos hasta que la etiqueta nuevamente generada sea diferente de la etiqueta original.

Este ataque se ha implementado para que las imágenes se envíen a través de una *url* web para facilitar y simplificar el trabajo de pruebas. Podría ampliarse sin demasiada dificultad para que lea, también, archivos de un directorio.

5.3.2. Algunos resultados de los ataques no dirigidos

A través de este método podemos obtener resultados como los siguientes:

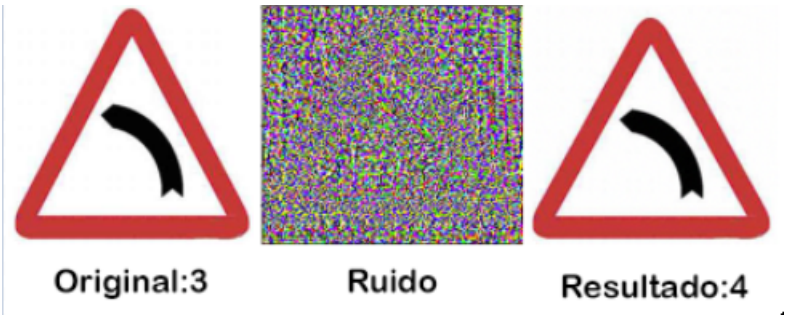

Figura 5.9: Transformación de una curva izquierda (3) a una curva derecha (4).

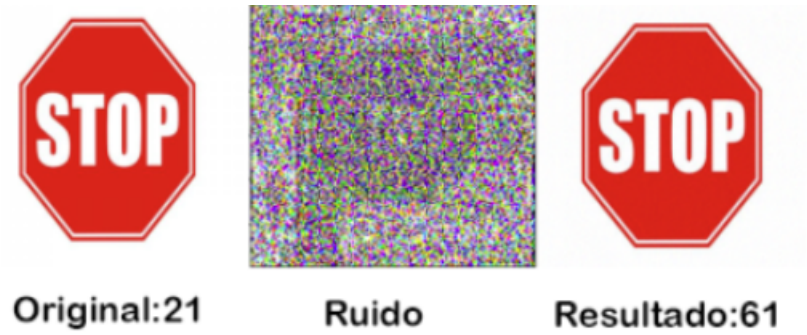

Figura 5.10: Transformación de una señal de stop (21) a una señal de camino prioritario (61).

Como se puede apreciar, en este tipo de ataques **no se logran dos clasificaciones erróneas iguales** ya que no se puede predecir cuál será la clase de salida y únicamente nos centramos en maximizar el error de la clase correcta.

5.4. Ataques adversarios dirigidos

5.4.1. Implementación en Python

En este ataque en concreto, el código parte de [3], que ha sido la inspiración de este trabajo. El código está modificado y adaptado en función de nuestro modelo de clasificación.

En este caso la implementación parte de la modificación del grafo [2] de Tensorflow. Tensorflow, de manera interna, crea un grafo de la red por la que van pasando y fluyendo los datos (los tensores). Aquí, vamos a modificar este orden natural para favorecer la velocidad de computación.

Comenzamos definiendo el tensor de entrada, la imagen, y el tensor de salida (las probabilidades). Como antes preparamos las imagenes y establecemos las perturbaciones límite.

La función de coste ahora sería la probabilidad de que la imagen pertenezca a la clase atacada. Posteriormente calculamos el gradiente entre la entrada y el coste y, aquí radica la diferencia, **creamos una función de Keras a la que llamaremos de manera cíclica para que nos calcule el gradiente** dándole como entrada la imagen en el ciclo actual y obteniendo como salida su gradiente

[2]Un grafo es un conjunto de objetos llamados nodos unidos por enlaces llamados arcos y que permiten representar las relaciones entre los elementos del conjunto

con respecto al coste.

Finalmente limitamos las perturbaciones. Iteramos hasta que las probabilidades de que la imagen pertenezca a la clase atacada sean >= 95 %.

5.4.2. Algunos resultados de los ataques dirigidos

A través de este método podemos obtener resultados como los siguientes:

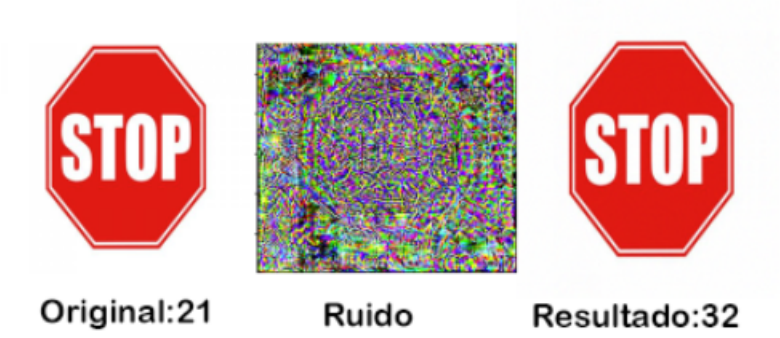

Figura 5.11: Transformación de una señal de stop (21) a una señal de límite de velocidad (32).

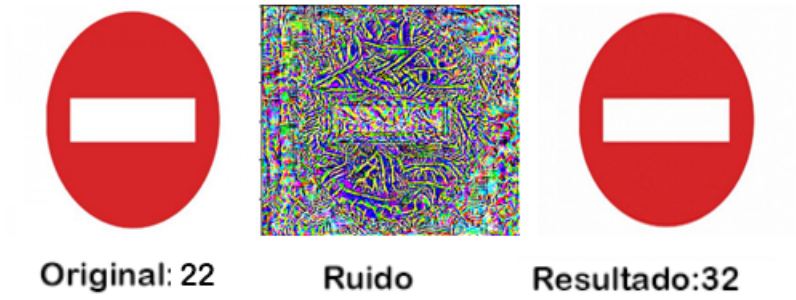

Figura 5.12: Transformación de una señal de dirección prohibida (22) a una señal de límite de velocidad (32).

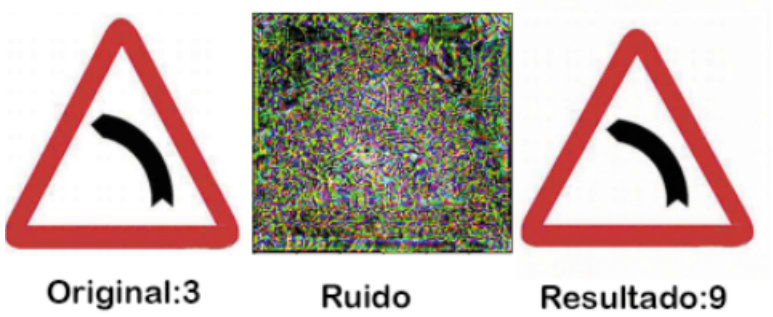

Figura 5.13: Transformación de una curva izquierda (3) a una señal de peligro ganado (9).

En este caso **se intentó realizar un ataque a la clase 32 en ambos ejemplos** sin embargo en el segundo no fue posible lograrlo y decidimos atacar la figura original 5.13 a la clase 9. Es **muy importante** entender que, **si limitamos la perturbación máxima, es muy probable que no se puedan lograr ataques que supongan grandes diferencias entre imágenes**. En este ejemplo, no se ha podido lograr el ataque con una perturbación máxima de 0,01 ya que aunque siguiéramos iterando el error no aumentaba. Esto es debido a que tendríamos que lograr que la red confunda una señal triangular con una redonda de un color diferente con muy poca variación, algo realmente complicado.

Si eliminamos la limitación de perturbación podríamos llegar a conseguirlo pero la imagen se verá claramente modificada tal que:

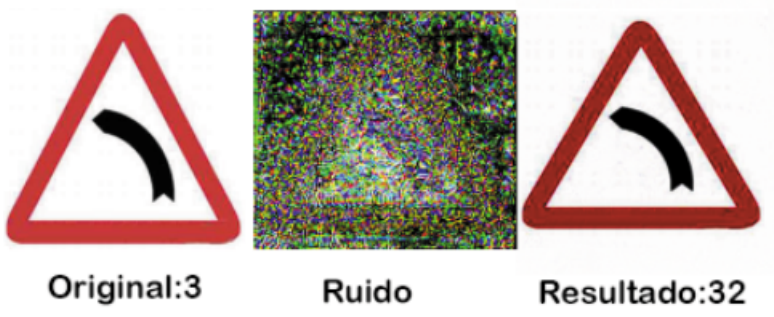

Figura 5.14: Transformación de una curva izquierda (3) a una señal de límite de velocidad (32).

5.5. Ataques adversarios dirigidos en la red de detección

5.5.1. Implementación en Python

En este algoritmo comenzamos obteniendo cuál es la parte de la imagen que contiene la señal. Luego, especificamos la clase actual y la clase objetivo de ataque. Comenzamos a iterar y, si el *bounding box* ya pertenece a la clase atacada, nos detenemos. De otro modo,reconocemos todos los *boxes* cuya clase principal es la clase inicial. En casos dónde estemos atacando desde la clase 0, sólo será necesario atacar al primer *bounding box* porque la mayoría serán de clase 0. Recorremos todos los píxeles de esos *bounding boxes* definiendo vectores *one-hot* con la etiqueta real y la etiqueta ataque. Obtenemos los valores de predicción de esas clases multiplicándolas por las predicciones totales (al sólo tener un 1 en lo deseado el resto se convierte en 0). Calculamos los gradientes tal y como se definieron anteriormente y realizamos su diferencia.

Finalmente calculamos la norma, obtenemos el valor de r' y sumamos el gradiente únicamente al la señal inicial. Limitamos los valores de la imagen con las perturbaciones máximas.

De esta manera somos capaces de realizar ataques adversarios que, únicamente, modifican la señal sin tener en cuenta el entorno. A través de este algoritmo **podemos lograr dos cosas: que la red detecte la señal como otra clase o que la red no detecte la señal** (clase 0 o fondo).

5.5.2. Algunos resultados de los ataques en la red de detección

A través de este método podemos obtener resultados como los siguientes:

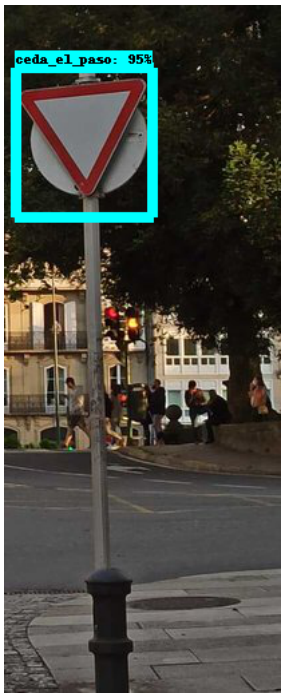

Figura 5.15: Imagen original. Clasificada como una señal de ceda el paso.

Figura 5.16: Imagen atacada, no hay detección.

Este tipo de ataques de eliminación no son posibles con los artículos comentados anteriormente en ataques en redes de detección ya que solo permiten el cambio entre clases.

Figura 5.17: Ruido añadido a la imagen.

Ahora **vamos a cambiar una señal de Stop a una de prohibido el paso y viceversa**. Para ello utilizaremos el método previamente descrito cambiando los parámetros de target class y clase actual.

Figura 5.18: Imagen original. Clasificada como una señal de Stop.

Figura 5.19: Imagen atacada. Clasificada como una señal de prohibido el paso.

Figura 5.20: Modificación de la señal de Stop.

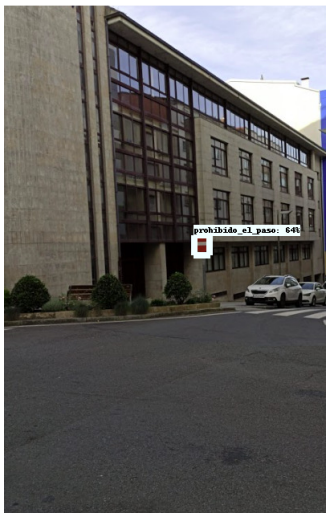

Figura 5.21: Imagen original. Clasificada como una señal de prohibido el paso.

Figura 5.22: Imagen atacada. Clasificada como una señal de Stop.

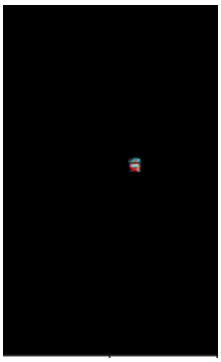

Figura 5.23: Ruido completo añadido a la señal de prohibido el paso.

En la red de detección **es mucho más complicado realizar ataques adversarios dirigidos en señales diferentes**. Aunque somos capaces de conseguir prácticamente la eliminación de todas las señales (algo que ya supondría un gran problema) no es tan sencillo lograr un cambio de categoría.

Uno de los métodos que podemos utilizar si queremos realizar un cambio de categoría es: en vez de realizar el cambio directamente, transformar de la etiqueta inicial a la clase 0 (no detección) y, partiendo de esta, intentar alcanzar la señal deseada. Empíricamente este método ha sido eficaz en varios ejemplos adversarios generados.

Otras opciones, como siempre, son aumentar la perturbación limitante con todo lo que ello supone. Artículos ya mencionados como [36], [6] proponen la utilización de *parches* en imágenes o rostros para generar ataques adversarios realmente efectivos. Con estas técnicas no ocultaríamos el ataque pero, quizás un usuario normal no se daría cuenta al confundirlo con "grafitis".

5.6. Ataques en videos en redes de detección

También se ha logrado en este trabajo el análisis de señales en vídeos grabados internamente en un coche. Para ello, se ha realizado un método iterativo de lectura de fotogramas por separado, se realizan las detecciones en cada imagen y se vuelve a montar el vídeo.

Aunque no se ha implementado por falta de tiempo y potencia de cómputo, esta técnica serviría también para la generación de ejemplos adversarios en vídeos a través del análisis de los fotogramas.

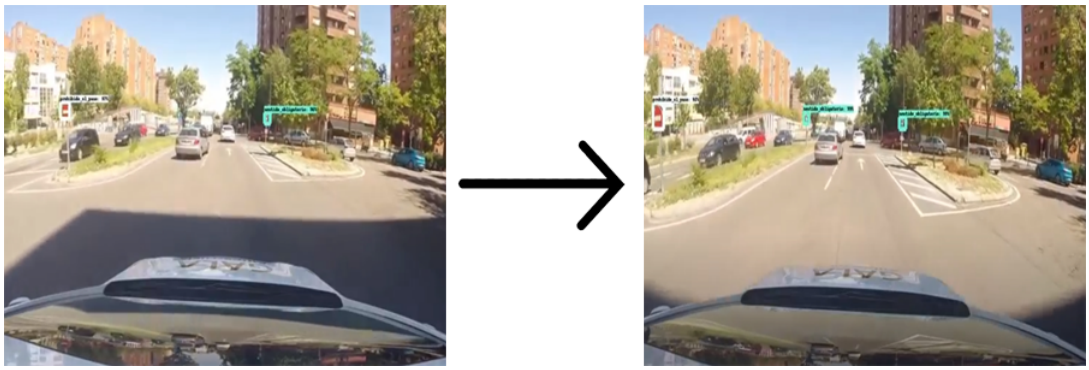

Figura 5.24: Ejemplo de fotogramas en vídeo

Capítulo 6

Conclusiones y posibles ampliaciones

Este trabajo ha puesto de manifiesto las facilidades con las que se puede lograr generar ejemplos adversarios que provocan el engaño en la red neuronal. Esto se ha logrado partiendo del estudio y análisis del funcionamiento de los ataques adversarios a bajo nivel para, posteriormente, implementar diversos algoritmos en lenguaje de alto nivel que permitan comprobar el funcionamiento teórico de estos. Los métodos desarrollados permiten el engaño de redes de clasificación y de redes de detección que han sido generadas como base para las pruebas del trabajo.

Se puede concluir por tanto, que, a través de los métodos proporcionados, se puede inducir al fallo a las redes neuronales profundas convolucionales. Además, ya que los ataques se basan en el propio funcionamiento de la red neuronal y que es posible transferir ataques entre diversas redes con igual finalidad, que hace que no sea necesario contar con la implementación de la red a atacar, nos lleva a una situación donde las defensas son realmente complicadas de implementar. Los ataques se han logrado en un **100 % de los casos probados** tanto en las redes de detección como en las redes de clasificación implementadas en este TFG.

Por ende, se pone de manifiesto que, **a día de hoy, las redes neuronales tienen fallos de seguridad que deben ser solucionados antes de poder poner toda nuestra confianza en ellas.** Esto complica bastante el auge de tecnologías como los coches autónomos o el reconocimiento facial automático ya que se podrían manipular a nuestro favor provocando grandes problemas.

Las ampliaciones que se podrían realizar en este trabajo son: la exploración de otros métodos de generación de ejemplos adversarios como Deepfool [27] u otros métodos de generación de ataques adversarios que no están basados en el cálculo del gradiente y cuyo funcionamiento es realmente interesante de explorar.

Además, de igual forma que hemos hecho una extrapolación del método *DAG* propuesto por [57] a nuestro caso concreto, se podría extrapolar nuestro trabajo centrado en señales de tráfico a otras redes de detección donde nuestro objetivo sea la manipulación de un único objeto sin importar el entorno.

Sería interesante la búsqueda de un algoritmo que permita **realizar ataques adversarios en vídeos de manera eficiente** (sin la necesidad de analizar fotograma a fotograma) o, en otro caso, extrapolar y optimizar nuestro método para este fin.

Por último, también se podría profundizar en el desarrollo de defensas basadas en modificaciones de la función de activación para evitar la propagación de un error mínimo a lo largo de la red como [59], en la búsqueda de métodos de detección de ejemplos adversarios como [26] o en el desarrollo de redes generativas como en [22] y [45].

Apéndice A

Código en Python de los algoritmos

A.1. Ataques adversarios no dirigidos en la red de clasificación

```
objetoPerdidas=tf.keras.losses.CategoricalCrossentropy()

def crearAtaqueNoDirigido(url,epsilon):
        imagen=preparar(url)
        etiqueta=predecir(preparar(url))
        etiqueta_nueva=predecir(preparar(url))
        adversarial=url_to_array(url)
        adversarial=comoTensor(adversarial)

        perturbacion=0.2 #Maxima perturbacion por pixel
        max_per=imagen+np.array([perturbacion])
        min_per=imagen-np.array([perturbacion])

        while etiqueta==etiqueta_nueva:
         predicciones = red(adversarial)
         [...]
         tensorEtiqueta = tf.one_hot(etiqueta, 62)
         [...]
         loss=objetoPerdidas(tensorEtiqueta,predicciones)
         gradiente = tf.gradients(loss,adversarial)
         signoGradiente = tf.sign(gradiente)
         array=tf.keras.backend.eval(signoGradiente)
         adversarialArray=adversarialArray+epsilon*array[0]
         adversarial=np.clip(adversarialArray,min_per,max_per)
```

```
        adversarialArray=np.clip(adversarial,-1,1)
        [...]
        etiqueta_nueva = predecir(adversarialArray)
    return adversarial
```

La simbología [...] indica que existe más código intermedio pero se omite por no ser relevante. Se puede consultar el código completo en el cuaderno de programación.

A.2. Ataques adversarios dirigidos en la red de clasificación

```
def crearAtaqueDirigido(url,claseEsperada):
entrada=red.layers[0].input
salida=red.layers[-1].output
target_class=claseEsperada
imagen=preparacionArray(url)
adversarial=np.copy(imagen)

perturbacion=0.01 #Maxima perturbacion por pixel
max_per=imagen+np.array([perturbacion])
min_per=imagen-np.array([perturbacion])

coste=salida[0,target_class] #Funcion a maximizar

gradiente=tf.keras.backend.gradients(coste,entrada)[0]
optimiza_gradiente=tf.keras.backend.function [.....]
cost=0.0
while cost < 0.95:
gr,cost=optimiza_gradiente([adversarial,0])
adversarial+=gr

adversarial=np.clip(adversarial,min_per,max_per)
adversarial=np.clip(adversarial,-1,1)

return adversarial
```

La simbología [...] indica que existe más código intermedio pero se omite por no ser relevante. Se puede consultar el código completo en el cuaderno de programación.

A.3. Ataques adversarios en la red de detección

```
adversarial=tf.keras.preprocessing.image.img_to_array([...])
boxes1,scores,classes,score_todas = reconocer_imagen([...])
target_class= 0
clase_actual= 4

while m<1000:

boxes,scores,classes,score_todas = reconocer_imagen([...])
lista_boxes=detectar_boxes(score_todas,clase_actual)
pred= score_todas[0][0][target_class]

if(pred >0.5 or np.argmax(score_todas[0][0])==target_class):
break

if(clase_actual==0):
lista_boxes=[0]

for x in lista_boxes:
verdad = labels_reales_one_hot([...])
ataque=labels_cambiar_one_hot([...])

[...]

adv_log = tf.math.multiply(predicciones_tensor,ataque_tensor)
real_log = tf.math.multiply(predicciones_tensor,verdad_tensor)

rm_sum = tf.reduce_sum(adv_log)
rm_sum2 = tf.reduce_sum(real_log)

arraylossAtaque = tf.keras.backend.eval(rm_sum)
arraylossActual = tf.keras.backend.eval(rm_sum2)

rm = np.gradient(adversarial,arraylossAtaque)
rm2 = np.gradient(adversarial,arraylossActual)

rm_total = (rm[0]-rm2[0])

[...]

norma_calc=tf.norm(rm_tensor,axis=(0,1))
```

```
rm_prima =  (gamma/norma)*rm_total
adversarial = sumar_gradiente(adversarial,boxes1,rm_prima)

adversarial=np.clip(adversarial,min_per,max_per)
adversarial =np.clip(adversarial,-1,1)

m=m+1
```

La simbología [...] indica que existe más código intermedio pero se omite por no ser relevante. Se puede consultar el código completo en el cuaderno de programación.

Apéndice B

Apendice B

Data Augmentation en la red de clasificación. Ejemplos obtenidos de [47]

Figura B.1: Imagen Original

Figura B.2: Rotacion

Figura B.3: Desplazamientos Horizontales

Figura B.4: Desplazamientos Verticales

Figura B.5: Inclinación de la imagen

Figura B.6: Zoom

Figura B.7: Giro Horizontal

Data Augmentation en la red de detección.Ejemplos obtenidos de [51]

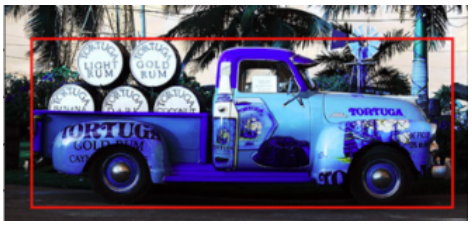

Figura B.8: Imagen Original

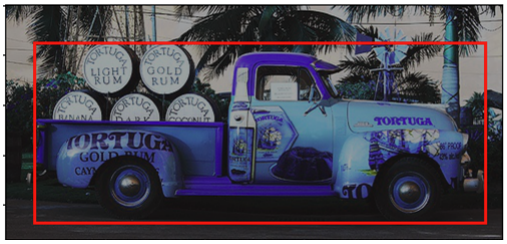

Figura B.9: Saturacion

Ejemplos del conjunto de test de la red de clasificación

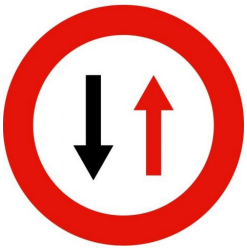

Figura B.10: Ejemplo de la clase 20

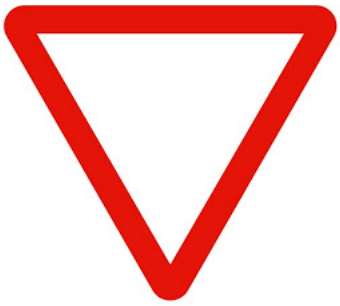

Figura B.11: Ejemplo de la clase 19

Ejemplos del conjunto de test de la red de deteccion

Figura B.12: Ejemplo de fotografía en las calles de santiago

Figura B.13: Ejemplo de fotografía en las calles de santiago

Figura B.14: Ejemplo de fotografía en las calles de santiago

Apéndice C

Manual de usuario

C.1. Instalación de la red de clasificación

- Crear una cuenta de Google a través de `https://accounts.google.com/signup`
- Abrir la cuenta del servicio Google Drive asociado en `https://drive.google.com/drive/my-drive`
- Crear una carpeta en la raíz (my-drive) que se llame Universidad, dentro de esta, una carpeta que se llame TFG y dentro de esta una carpeta que se llame BelgiumTrafficModel.

  /Universidad/TFG/BelgiumTrafficModel

- En el directorio anterior extraer el archivo DatasetClasificacion.zip adjunto.
- Entrar en este link: `https://colab.research.google.com/drive/1dP6n40vPObCcDp-J8cEyF usp=sharing`. Este link contiene el código completo de ejecución. También se adjunta como **Clasificacion.ipynb** en caso de que se quiera ejecutar en local pero habría que modificar el apartado de Drive por carpetas locales.

El dataset de entrenamiento y validación ha sido obtenido de [47], el resto de archivos adjuntos son de creación propia.

C.2. Ejecución de la red de clasificación

Una vez abierto el enlace de la red anterior se debe ir ejecutando casilla por casilla y, en caso de ser necesario, completar la información que nos den estas como respuesta. Las **casillas 3 a 8 (ambas incluídas)** son para el entrenamiento de la red y pueden omitirse en caso de utilizar la red entrenada otorgada. Cada casilla contiene un comentario a su inicio que define cuál es su finalidad. La red de clasificación se carga a través del archivo *redEntrenada.h5*

C.2.1. Ataques adversarios dirigidos

La casilla 12 contiene el código de ataques adversarios dirigidos. En la última línea tenemos la llamada a la función previa *crearAtaqueDirigido*. Esta función tomará como **parámetros**: **una URL** de imágen en formato *.jpg* para atacar y, como segundo parámetro, **la clase de ataque esperada**.

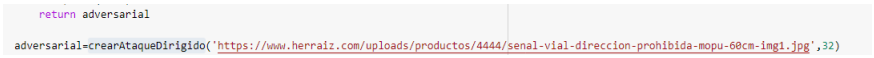

Figura C.1: Celda de ataques dirigidos.

C.2.2. Ataques adversarios no dirigidos

La casilla 13 contiene el código de ataques adversarios no dirigidos. En la última línea tenemos la llamada a la función previa *crearAtaqueNoDirigido*. Esta función tomará como **parámetros**: **una URL** de imágen en formato *.jpg* para atacar y, como segundo parámetro, **un valor de máxima perturbación por píxel**. Este valor ya se analizó anteriormente y está relacionado con la distorsión visual máxima con respecto a la imagen original.

Figura C.2: Celda de ataques no dirigidos.

C.2.3. Pintado de imágenes

Las casillas 14 y 15 contiene código para la visualización de las imágenes originales y atacadas. **La única variante sería la modificación de la URL de ejemplo por la URL de la imagen atacada**, el resto de código se mantiene inalterado.

Figura C.3: Celda de pintado de imágenes.

C.3. Instalación de la red de detección

- Crear una cuenta de Google a través de `https://accounts.google.com/signup`

- Abrir la cuenta del servicio Google Drive asociado en `https://drive.google.com/drive/my-drive`
- Crear una carpeta en la raíz (my-drive) que se llame Universidad, dentro de esta, una carpeta que se llame TFG y dentro de esta una carpeta que se llame GermanTrafficModel

  /Universidad/TFG/GermanTrafficModel

- En el directorio anterior extraer el archivo DatasetDeteccion.zip adjunto.
- Entrar en este link:`https://colab.research.google.com/drive/1t7HKgMbZi_dvEg3eMZ2OSnOMykkDVxyd?usp=sharing`. También se adjunta como **Deteccion.ipynb** en caso de que se quiera ejecutar en local pero habría que modificar el apartado de Drive por carpetas locales.

Tras ejecutar la casilla número 4 el entorno debe de reiniciarse ya que se han importado los archivos a la memoria del servidor pero no los ha instalado por lo que el código no funcionara. Para ello vamos a *Entorno de ejecución* → *Reiniciar entorno de ejecución* y volvemos a ejecutar las casillas en orden para que se produzca la instalación.

La carpeta de models contiene la API de Tensorflow obtenida de [55]. Los dataset de entrenamiento y validación han sido obtenidos de [51]. El resto de archivos adjuntos son de creación propia.

C.4. Ejecución de la red de detección

Una vez abierto el enlace de la red anterior se debe ir ejecutando casilla por casilla y, en caso de ser necesario, completar la información que nos den estas como respuesta. Las **casillas 5 a 11 (ambas incluidas)** son para el entrenamiento de la red y pueden omitirse en caso de utilizar la red entrenada otorgada. Cada casilla contiene un comentario a su inicio que define cuál es su finalidad. La red de detección entregada está cargada en la carpeta *Grafo* dónde se especifica todo el procesamiento que deben seguir los datos para obtener una salida (casilla 13).

```
#Librerias y requerimientos necesarios
%cd /content/drive/'My Drive'/Universidad/TFG/GermanTrafficModel
%cd ./models/research
!protoc object_detection/protos/*.proto --python_out=.
!python setup.py build
!python setup.py install

os.environ['PYTHONPATH'] += ':/content/drive/My Drive/Universidad/TFG/GermanTrafficModel/models/research/:/content/drive/My Drive/Universidad/TFG/GermanTrafficModel/models/research/slim/'

import sys
#Añado al sistema los pathings de los modelos
sys.path.append('./object_detection')

##Prueba de que los modelos funcionan y los path son correctos
!python object_detection/builders/model_builder_test.py
```

Figura C.4: Celda dónde es necesario reiniciar.

C.4.1. Análisis de vídeos

La casilla 18 contiene el código necesario para la ejecución de la red neuronal en un vídeo fotograma a fotograma. Habría que modificar *./video.mp4* por la ubicación del vídeo que queremos analizar. Obtendríamos *salida.avi* como salida.

```
##RED EN VIDEOS##

%cd /content/drive/My\ Drive/Universidad/TFG/GermanTrafficModel
import cv2
import numpy as np
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils
from google.colab.patches import cv2_imshow
with tf.device("/gpu:0"):
  cap = cv2.VideoCapture("./video.mp4")
  ret, image_np = cap.read()
  weight = image_np.shape[0]
  height = image_np.shape[1]
  size=(height,weight)
  out = cv2.VideoWriter('./salida.avi',cv2.VideoWriter_fourcc('m', 'p', '4', 'v'), 10 , size)
  cap = cv2.VideoCapture("./video.mp4")
  while cap.isOpened():
    ret, image_np = cap.read()
    if ret == True:
        color = cv2.cvtColor(image_np,cv2.COLOR_BGR2RGB)
        imagen,boxes,scores,classes=reconocer_imagen_array(color,0.9)
        imprimir = color = cv2.cvtColor(imagen,cv2.COLOR_BGR2RGB)
        out.write(imprimir)
    else:
      cap.release()
      out.release()
      break
```

Figura C.5: Celda 18 de código.

C.4.2. Ataques adversarios en la red de detección

La casilla 22 contiene el código para la **ejecución de ataques adversarios en redes neuronales de detección**. Podemos modificar *./testing24.jpg* por la imagen original a atacar. *perturbacion=0.3* por el valor máximo de perturbación en los píxeles de la imagen. *target_class* por la clase esperada de salida y *clase_actual* por la clase actual de la imagen.

Cómo se comentó anteriormente, en muchos casos puede ser necesario hacer un ataque de *clase_actual* $\rightarrow 0$ y luego $0 \rightarrow$ *target_class* para aumentar velocidad. Depende de cada imagen en particular, varias imágenes testadas funcionaron mejor con conversión directa mientras que otras funcionaron mejor con conversión en tres pasos.

Obtendríamos como salida la variable adversarial y un fichero *salida.npy* dónde se almacena la imagen adversaria.

```
with tf.device("/gpu:0"):
  adversarial=tf.keras.preprocessing.image.img_to_array(Image.open('./testing24.jpg'))
  boxes1,scores,classes,score_todas = reconocer_imagen_array(adversarial)
  adversarial = extraer_imagen(adversarial,boxes)
  #LA TRANSFORMO EN -1,1
  adversarial=adversarial/255
  adversarial=adversarial-0.5
  adversarial=adversarial*2
  perturbacion=0.3 #Maxima perturbacion por pixel
  max_per=adversarial+np.array([perturbacion])
  min_per=adversarial-np.array([perturbacion])
  #LA TRANSFORMO EN 0-255
  adversarial=adversarial/2
  adversarial=adversarial+0.5
  adversarial=adversarial*255
  gamma=1
  m=0
  arrayloss=5000
  mejorarrayloss=-5000
  r=np.zeros(shape=(adversarial.shape[0],adversarial.shape[1],adversarial.shape[2]))
  target_class= 5
  clase_actual= 4
```

Figura C.6: Celda 22 de código.

C.4.3. Visualización imágenes

Las casillas 20 y 23 permiten visualizar imágenes (originales o atacadas) **tras ser procesadas por la red.** Para ello modificaríamos *./testing24.jpg* por la imagen original o descomentamos *adversarial=np.load('salida.npy')* para visualizar la imagen atacada tras ser procesada por la red.

La casilla 21 permite visualizar la extracción de la señal con respecto al fondo modificando *./testing3.jpg* por la dirección de la imagen.

```
##VER IMAGENES##

from object_detection.utils import visualization_utils
from google.colab.patches import cv2_imshow
with tf.device("/gpu:0"):
  def pintar_imagen_array(image_np,min_score_thresh):

    #Se espera de la forma:[1, None, None, 3]
    image_np_expanded = np.expand_dims(image_np, axis=0)

    (boxes, scores, classes, num,score_todas) = sess.run(
          [detection_boxes, detection_scores, detection_classes, num_detections,detection_score_todas],
          feed_dict={image_tensor: image_np_expanded})

    #Lanzo visualizador

    visualization_utils.visualize_boxes_and_labels_on_image_array(
        image_np,
        np.squeeze(boxes),
        np.squeeze(classes).astype(np.int32),
        np.squeeze(scores),
        category_index,
        use_normalized_coordinates=True,
        line_thickness=8,
        min_score_thresh=min_score_thresh)

    return image_np,boxes,scores,classes,score_todas


  imagen1=tf.keras.preprocessing.image.img_to_array(Image.open('./testing24.jpg'))
  #adversarial=np.load('salida.npy')
  imagen,boxes2,scores2,classes2,salida_todas2=pintar_imagen_array(imagen1,0.6)

  img_float_32 = np.float32(imagen)
  imp = cv2.cvtColor(img_float_32,cv2.COLOR_BGR2RGB)

  cv2_imshow(imp)
```

Figura C.7: Celda 20 de pintado.

Bibliografía

[1] BannerBob. *Faster non maximum suppression in Python*. URL: https://www.codemade.io/faster-non-maximum-suppression-in-python-pyimagesearch (visitado 02-06-2020).

[2] Young-Jin Cha y col. "Autonomous structural visual inspection using region-based deep learning for detecting multiple damage types". En: *Computer-Aided Civil and Infrastructure Engineering* 33.9 (2018), págs. 731-747.

[3] Dot CSV. *Ataques adversarios, cómo romper una RED NEURONAL — Programando IA*. URL: https://www.youtube.com/watch?v=JoQx39CoXW8&t=1962s (visitado 02-06-2020).

[4] Sebastien Derhy. *Ejemplo de las 62 clases del Belgium Traffic Sign Dataset*. URL: https://github.com/sebderhy/TrafficSignsClassif (visitado 01-06-2020).

[5] Google Drive. *Web de acceso a la plataforma de almacenamiento Google Drive*. URL: https://www.google.com/intl/es_ALL/drive/ (visitado 02-06-2020).

[6] Ivan Evtimov y col. "Robust physical-world attacks on deep learning models". En: *arXiv preprint arXiv:1707.08945* (2017).

[7] Ross Girshick. "Fast r-cnn". En: *Proceedings of the IEEE international conference on computer vision*. 2015, págs. 1440-1448.

[8] Ross Girshick y col. "Rich feature hierarchies for accurate object detection and semantic segmentation". En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, págs. 580-587.

[9] Ian J Goodfellow, Jonathon Shlens y Christian Szegedy. "Explaining and harnessing adversarial examples". En: *arXiv preprint arXiv:1412.6572* (2014).

[10] Google. *Entorno de computación de Google Colaboratory*. URL: https://colab.research.google.com/ (visitado 01-06-2020).

[11] Google. *Entorno de computación de Google Colaboratory Profesional*. URL: https://colab.research.google.com/signup (visitado 01-06-2020).

[12] Kaiming He y col. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". En: *The IEEE International Conference on Computer Vision (ICCV)*. 2015.

[13] HiSoUR. *Historia de la visión por computador*. URL: `https://www.hisour.com/es/computer-vision-42799/` (visitado 02-06-2020).

[14] Elad Hoffer y col. "Augment your batch: better training with larger batches". En: *arXiv preprint arXiv:1901.09335* (2019).

[15] Sebastian Houben y col. "Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark". En: *International Joint Conference on Neural Networks*. 1288. 2013.

[16] Sebastian Houben y col. *Web del German Traffic Sign Detection Dataset*. URL: `http://benchmark.ini.rub.de/?section=gtsdb&subsection=news` (visitado 01-06-2020).

[17] Jonathan Hui. *mAP (mean Average Precision) for Object Detection*. URL: `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173` (visitado 02-06-2020).

[18] ImageNet. *Página web de documentación de ImageNet*. URL: `http://www.image-net.org/` (visitado 02-06-2020).

[19] Sambasivarao. K. *Non-maximum Suppression (NMS)*. URL: `https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c` (visitado 02-06-2020).

[20] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". En: *arXiv preprint arXiv:1404.5997* (2014).

[21] Alexey Kurakin, Ian Goodfellow y Samy Bengio. "Adversarial examples in the physical world". En: *arXiv preprint arXiv:1607.02533* (2016).

[22] Hyeungill Lee, Sungyeob Han y Jungwoo Lee. "Generative adversarial trainer: Defense to adversarial perturbations with gan". En: *arXiv preprint arXiv:1705.03387* (2017).

[23] Quanyu Liao y col. "Category-wise Attack: Transferable Adversarial Examples for Anchor Free Object Detection". En: *arXiv preprint arXiv:2003.04367* (2020).

[24] Chunmian Lin y col. "Transfer learning based traffic sign recognition using inception-v3 model". En: *Periodica Polytechnica Transportation Engineering* 47.3 (2019), págs. 242-250.

[25] Matplotlib. *Web de documentación de Matplotlib*. URL: `https://matplotlib.org/` (visitado 02-06-2020).

[26] Jan Hendrik Metzen y col. "On detecting adversarial perturbations". En: *arXiv preprint arXiv:1702.04267* (2017).

[27] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi y Pascal Frossard. "Deepfool: a simple and accurate method to fool deep neural networks". En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 2574-2582.

[28] Nvidia. *Tarjeta gráfica Nvidia Tesla K10*. URL: `https://www.nvidia.com/es-es/data-center/tesla-k80/` (visitado 01-06-2020).

[29] Nvidia. *Tarjeta gráfica Nvidia Tesla P100*. URL: `https://www.nvidia.com/en-us/data-center/tesla-p100/` (visitado 01-06-2020).

[30] Nvidia. *Tarjeta gráfica Nvidia Tesla T4*. URL: `https://www.nvidia.com/es-es/data-center/tesla-t4/` (visitado 01-06-2020).

[31] OpenCV. *Web de documentación de OpenCV*. URL: `https://opencv.org/` (visitado 02-06-2020).

[32] Arthur Ouaknine. *Review of Deep Learning Algorithms for Object Detection*. URL: `https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852` (visitado 02-06-2020).

[33] Nicolas Papernot y col. "cleverhans v2. 0.0: an adversarial machine learning library". En: *arXiv preprint arXiv:1610.00768* (2016).

[34] Nicolas Papernot y col. "Practical black-box attacks against machine learning". En: *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 2017, págs. 506-519.

[35] Nicolas Papernot y col. "The limitations of deep learning in adversarial settings". En: *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE. 2016, págs. 372-387.

[36] Mikhail Pautov y col. "On adversarial patches: real-world attack on ArcFace-100 face recognition system". En: *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*. IEEE. 2019, págs. 0391-0396.

[37] Victor Adrian Prisacariu y col. "Integrating object detection with 3D tracking towards a better driver assistance system". En: *Twentieth International Conference on Pattern Recognition*. Istanbul, Turkey, 2010, págs. 1-4.

[38] *Python*. URL: `https://www.python.org/` (visitado 01-06-2020).

[39] Luc van Gool Radu Timofte Karel Zimmermann. *Página web de descarga del Belgium Traffic Sign Dataset*. URL: `https://btsd.ethz.ch/shareddata/` (visitado 01-06-2020).

[40] Santhiya Rajan. *How will channels (RGB) effect convolutional neural network?* URL: `https://www.researchgate.net/post/How_will_channels_RGB_effect_convolutional_neural_network` (visitado 02-06-2020).

[41] Joseph Redmon y col. *YOLO Documentación*. URL: https://pjreddie.com/darknet/yolo/ (visitado 02-06-2020).

[42] Joseph Redmon y col. "You only look once: Unified, real-time object detection". En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 779-788.

[43] Shaoqing Ren y col. "Faster r-cnn: Towards real-time object detection with region proposal networks". En: *Advances in neural information processing systems*. 2015, págs. 91-99.

[44] Adrian Rosebrock. *Intersection over Union (IoU) for object detection*. URL: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (visitado 02-06-2020).

[45] Pouya Samangouei, Maya Kabkab y Rama Chellappa. "Defense-gan: Protecting classifiers against adversarial attacks using generative models". En: *arXiv preprint arXiv:1805.06605* (2018).

[46] Leonardo Araujo Santos. *Dropout Layer*. URL: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/dropout_layer.html (visitado 02-06-2020).

[47] Sumit Sarin. *Exploring Data Augmentation with Keras and TensorFlow*. URL: https://towardsdatascience.com/exploring-image-data-augmentation-with-keras-and-tensorflow-a8162d89b844 (visitado 02-06-2020).

[48] Team Data Thales Sophia-Antipolis. *Pipeline GPU-Tensorflow-Keras*. URL: https://punchplatform.com/2019/07/08/tensorflow-keras-pml-pipeline/ (visitado 01-06-2020).

[49] Christian Szegedy y col. "Intriguing properties of neural networks". En: *arXiv preprint arXiv:1312.6199* (2013).

[50] Christian Szegedy y col. "Rethinking the inception architecture for computer vision". En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 2818-2826.

[51] Telesens. *Data Augmentation in SSD (Single Shot Detector)*. URL: https://www.telesens.co/2018/06/28/data-augmentation-in-ssd/ (visitado 02-06-2020).

[52] Tensorboard. *Web de documentación de Tensorboard*. URL: https://www.tensorflow.org/tensorboard (visitado 02-06-2020).

[53] Tensorflow. *Adversarial example using FGSM*. URL: https://www.tensorflow.org/tutorials/generative/adversarial_fgsm (visitado 02-06-2020).

[54] *Tensorflow*. URL: https://www.tensorflow.org/ (visitado 01-06-2020).

[55] TensorFlow. *TensorFlow Object Detection API*. URL: https://github.com/tensorflow/models/tree/master/research/object_detection (visitado 02-06-2020).

[56] Jasper RR Uijlings y col. "Selective search for object recognition". En: *International journal of computer vision* 104.2 (2013), págs. 154-171.

[57] Cihang Xie y col. "Adversarial examples for semantic segmentation and object detection". En: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, págs. 1369-1378.

[58] Yani, Muhamad and Irawan, S, and S.T., M.T. "Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail". En: *Journal of Physics: Conference Series* 1201 (mayo de 2019), pág. 012052. DOI: 10.1088/1742-6596/1201/1/012052.

[59] Valentina Zantedeschi, Maria-Irina Nicolae y Ambrish Rawat. "Efficient defenses against adversarial attacks". En: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. 2017, págs. 39-49.