Reconocimiento de lenguaje Signo-Escritura mediante Deep Learning.

Trabajo de Fin de grado
Curso 2018–2019

Autor
John Byron Sánchez Jiménez
Samuel López Prieto
José Ángel Garrido Montoya

Director
Alberto Díaz
Antonio F. García Sevilla

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Reconocimiento de lenguaje Signo-Escritura mediante Deep Learning.

Trabajo de Fin de Grado en Ingeniería Informática
Departamento de Ingeniería del Software e Inteligencia Artificial

Autor
John Byron Sánchez Jiménez
Samuel López Prieto
José Ángel Garrido Montoya

Director
Alberto Díaz
Antonio F. García Sevilla

Convocatoria: *Junio 2019*
Calificación:

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

31 de mayo de 2019

Autorización de difusión

Los abajo firmantes, matriculados en el Grado en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "Lenguaje de signos", realizado durante el curso académico 2018/2019 bajo la dirección de Alberto Díaz y Antonio F. García Sevilla en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

John Byron Sánchez Jiménez
Samuel López Prieto
Jose Angel Garrido Montoya

31 de mayo de 2019

Agradecimientos

A nuestros profesores, familiares y amigos, por apoyarnos y hacer esto posible.

Resumen

Este proyecto consiste en la detección de la SignoEscritura mediante técnicas de deep learning. Esta tarea consta de dos partes: la clasificación de los símbolos y su localización dentro de una imagen.

Para su realización, en primer lugar se ha construido una red neuronal convolucional (CNN) para la clasificación de los símbolos. Esta CNN se ha desarrollado usando la arquitectura de LeNet5, la cual ha proporcionado resultados satisfactorios.

En segundo lugar, se ha llevado a cabo un detector de imágenes que ha combinado las tareas de clasificación y de localización mediante el uso del algoritmo YOLO. Los resultados obtenidos en este segundo punto también han sido prometedores.

El código asociado al desarrollo de este proyecto puede encontrarse público en la plataforma GitHub [1].

Palabras clave

Inteligencia Artificial, Clasificación de signos, Localización de objetos, Redes neuronales, YOLO, CNN, Redes Neuronales Convolucionales, Lenguaje de Signos, SignoEscritura

Abstract

This project is based on detecting SignWriting using deep learning. This task is splitted in: classifying the symbols and finding where they are in the image. Our work begins with the construction of a CNN to recognize SignWriting symbols. This CNN was created through the LeNet5 architecture, with good results.

Then, we built an object detector which unifies classification and localization of the symbols using the YOLO algorithm. The results of this classifier have also been promising.

The code associated with the development of this project can be found public on the GitHub platform [1].

Keywords

Artificial Intelligence, Sing Recognition, Neural Networks, Computer Vision, YOLO, CNN, Convolutional Networks, Sign Language, Sign Writing

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este proyecto consiste en reconocer imágenes de la SignoEscritura [2].

La SignoEscritura crea un estándar que permite plasmar en dos dimensiones los gestos creados por el lenguaje de signos, para que éstos puedan ser visualizados, por ejemplo, en papel. Se puede ver un ejemplo en la Figura 1.1

Figura 1.1: Ejemplo de la transcripción en SignoEscritura sobre un signo [2]

1.1. Introducción al lenguaje de signos

El lenguaje de signos se caracteriza por el uso de un canal visual, en vez de uno auditivo como la mayoría de lenguajes comunes. Esto permite que las Personas Sordas puedan comunicarse entre sí sin tener que recurrir a formas más lentas de comunicación, como puede ser la escrita.

Los lenguajes de signos son propios de cada país, y aunque dos países compartan la misma lengua auditiva, no comparten la misma lengua de signos. Por ejemplo, un español puede hablar en castellano con un mejicano, pero un español no puede utilizar el lenguaje de signos español con un mejicano, ya que usan una lengua de signos distinta.

Aunque también pueden compartir signos, de hecho, cuando dos personas

no hablan el mismo idioma, pueden utilizar gestos para comunicarse, como por ejemplo: señalar la muñeca para pedir la hora.

1.2. Clasificación de imágenes y detección de objetos

La clasificación de imágenes consiste en decidir qué tipo de objeto hay en ella, como por ejemplo, el gato de la Figura 1.2 (izquierda). Para decidir si esta imagen es un gato o no, se debe realizar una clasificación en función de las características propias de cada clase de objeto, como por ejemplo en este caso, la forma de las orejas y los bigotes de la cara. Estas características deben extraerse y evaluarse para poder catalogar dichas imágenes.

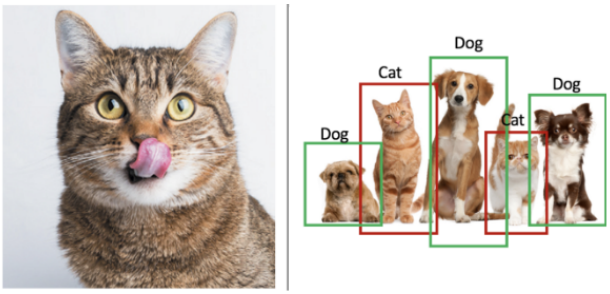


Figura 1.2: Clasificación de imágenes (izquierda) y detección de objetos (derecha) [3]

Por otro lado, la detección de objetos se basa en la búsqueda de la localización de cada objeto en una imagen, y su clasificación, como por ejemplo, los gatos y perros de la Figura 1.2 (derecha).

1.3. Objetivo

El objetivo del proyecto se basa en el uso de algoritmos de detección de objetos para su aplicación en la SignoEscritura, de modo que a partir de imágenes transcritas se pueda reconocer su significado. Para ello, se ha hecho uso de las técnicas de machine learning para el correcto reconocimiento de las imágenes, concretamente Redes Neuronales para el tratamiento de éstas.

Por ejemplo, en la imagen de la izquierda de la Figura 1.3 y la Figura 1.4 se observa la entrada que recibe el sistema, y a la derecha se muestra el resultado esperado con los distintos símbolos localizados y clasificados.

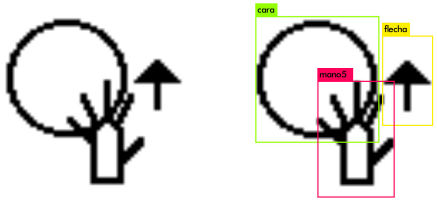


Figura 1.3: Reconocimiento del signo abochornar

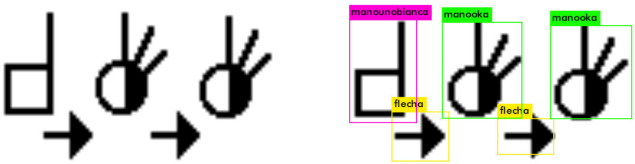


Figura 1.4: Reconocimiento del signo cien

La implementación de este sistema permitiría el desarrollo de aplicaciones que ayudarían en el uso y aprendizaje de la SignoEscritura, como por ejemplo, la creación de un diccionario.

Chapter 2

Introduction

This project is based on recognizing SignWriting images [2].

SignWriting is a standard that allows to represent gestures from Sign Languages in two dimensions. This is very useful as they can be represented in different ways, like in a piece of paper as the Figure 2.1 shows.

Figure 2.1: Example of a SignWritting transcription of a sign [2]

2.1. Sign language introduction

Sign languages differ from most languages in that they use a visual channel instead of an auditory one. This allows Deaf People to communicate between them without having to use slower methods like writing.

Sign languages are usually different in every country, even if they share the same spoken language. For example a spanish and a mexican can use spanish to speak between them, but they can's use sign language to communicate.

Many signs are the same in every language, some times when 2 people don't speak the same language, they will use signs to communicate, like pointing to your wrist to ask for the time.

2.2. Image classification and object detection

Image classification consists in deciding what object is it showing, see Figure 2.2 (left) for an example. In order to decide if an image is a cat or not it must be classified according to characteristics owned by that class. For a cat it would be whiskers and pointy ears. This characteristics need to be extracted and evaluated to be able to classify those images.

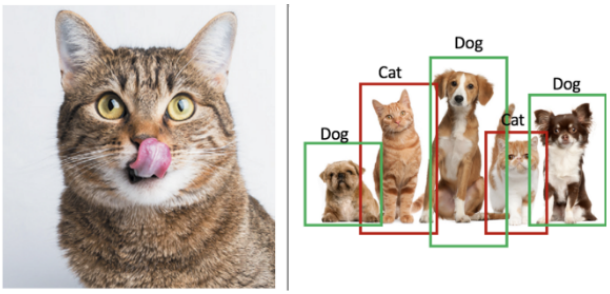


Figure 2.2: Image classification (left) and object detection (right) [3]

Object detection is based on identifying what and where are every object in that image. Figure 2.2 (right).

2.3. Goal

The goal of this project is to recognize SignWritting symbols, so that we can find the meaning of a transcription. To accomplish this we will be using machine learning techniques such as Neural Networks.

Here is an example of what we want to create: Figure 2.3 is the input and Figure 2.4 is the output.

The output is showing what and where are the symbols in the transcription.

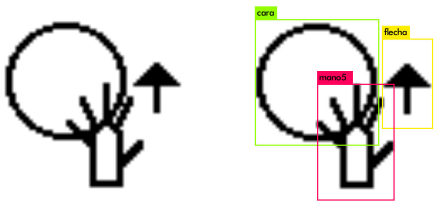


Figure 2.3: Recognizing the "embarrassed" transcription

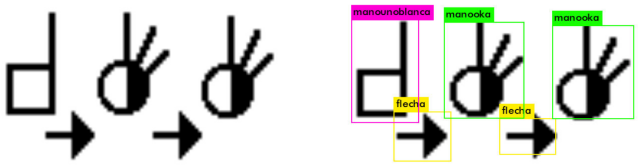


Figure 2.4: Recognizing the "100" transcription

The implementation of this system would allow to develop applications to use and learn about SignWriting, such as a dictionary or educational apps.

Capítulo 3

Estado del Arte

3.1. Nociones básicas de la SignoEscritura

Los símbolos de la SignoEscritura están representados desde la perspectiva propia, también conocida como perspectiva del signante, es decir, la persona que realiza el signo, tal y como se muestra en la Figura 3.1

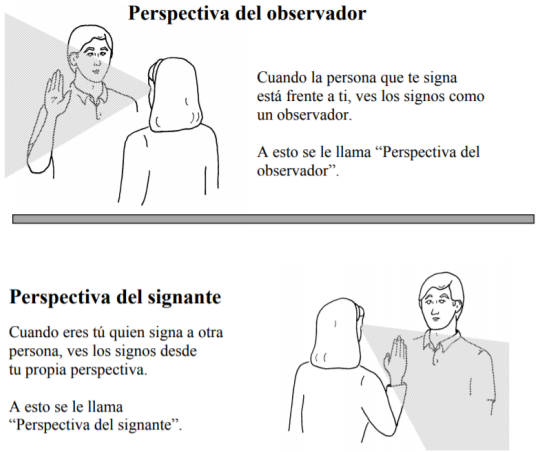


Figura 3.1: Ejemplo de la perspectiva en la SignoEscritura [4]

3.1.1. Orientación de la mano

La orientación de la mano debe ser representada tal y como muestra la Figura 3.2, es decir, con la palma en color blanco, y el dorso en color negro.

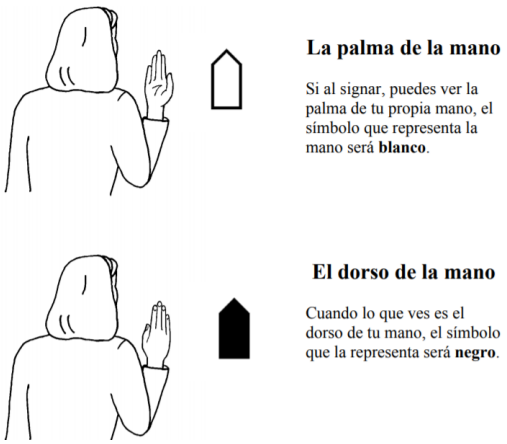


Figura 3.2: Orientación de la mano [4]

En caso de ver ambos lados de la mano, se utilizan los dos colores comentados anteriormente, y siguiendo la regla anterior (Figura 3.3).

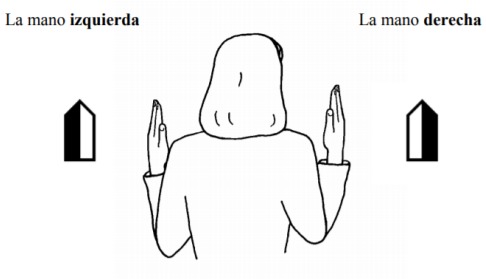


Figura 3.3: Lateral de las manos [4]

3.1.2. Rotación de la mano

La rotación de la mano se muestra de una forma muy intuitiva, simplemente rotando el símbolo (Figura 3.4).

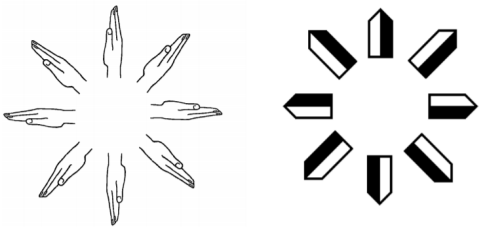


Figura 3.4: Rotaciones [4]

3.1.3. Contacto

Otra característica que se puede representar en la SignoEscritura, es el contacto entre partes del cuerpo. Para denotar un contacto se utiliza un asterisco, y en caso de que haya más de un contacto, se emplea ese mismo número de asteriscos (Figura 3.5).

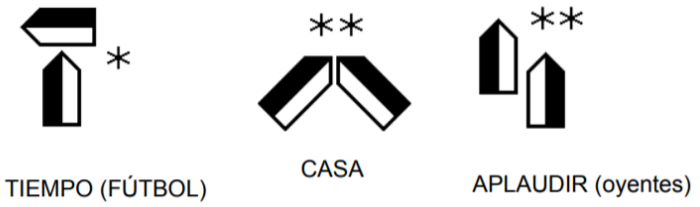


Figura 3.5: Ejemplo de contactos con las manos y su significado [4]

3.1.4. Puño y cabeza

Para representar una mano cerrada en forma de puño se utiliza un cuadrado, tal y como se muestra en la Figura 3.6.

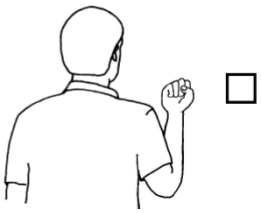


Figura 3.6: Representación de un puño [4]

Para representar la cabeza (vista desde atrás) se utiliza un círculo blanco. Por ejemplo, en la Figura 3.7 se observa un dedo de la mano derecha haciendo contacto con la cabeza por la derecha.

Figura 3.7: Representación de un dedo de la mano derecha haciendo contacto con la cabeza [4]

3.1.5. Ángulos de visión

Sin embargo, existe dificultad para representar una mano en algunos ángulos, por ejemplo, cuando está paralela al suelo desde la perspectiva del signante, como se aprecia en la Figura 3.8, donde es difícil ver su configuración.

Figura 3.8: Signante señalando con la mano derecha paralela al suelo [4]

Para solventar este problema, se representa la mano vista desde arriba añadiendo un espacio al símbolo en los nudillos (Figura 3.9)

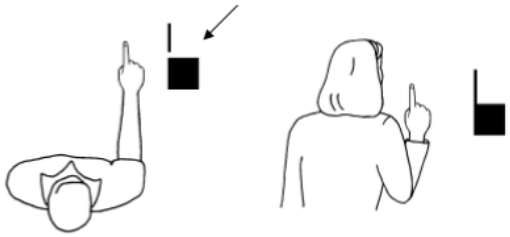


Figura 3.9: Signante con la mano paralela al suelo (izquierda), signante con la mano perpendicular al suelo (derecha) [4]

3.1.6. Movimiento

El movimiento se representa con flechas, tal y como se observa en la Figura 3.10

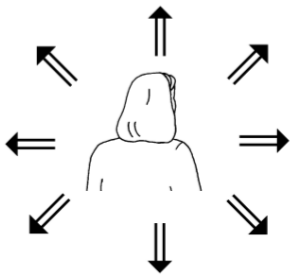


Figura 3.10: Ángulo desde el que se muestran las direcciones de los movimientos [4]

Para diferenciar el movimiento de la mano derecha e izquierda, se usa el color de la punta de la flecha. El color blanco indica que se mueve la mano izquierda, y el color negro que se mueve la derecha (Figura 3.11).

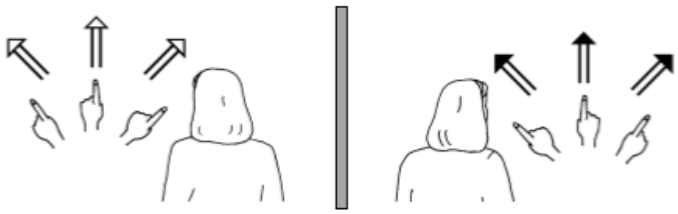


Figura 3.11: Representación de los movimientos de la mano izquierda y derecha [4]

3.2. Redes neuronales

Una red neuronal es un modelo simplificado que se inspira en el modo en que el cerebro humano procesa la información. Funciona simultaneando un número elevado de unidades de procesamiento interconectadas, que se organizan en capas y transmiten información procesada de unas a otras (véase la Figura 3.12). Una red neuronal consta principalmente de las siguientes partes:

- **Inputs:** capa que recibe los datos de entrada.
- **Capas ocultas:** capas intermedias e interconectadas que se encargan del procesamiento de la información. Puede haber una cantidad variable de ellas, y además con distinto número de neuronas.
- **Outputs:** capa que devuelve los resultados obtenidos de la red neuronal.
- **Weights:** Pesos de cada entrada, los cuales sirven para dar más importancia a una entrada que a otra.
- **Activation function:** Función limitadora o umbral. Modifica el valor resultado o impone un límite que se debe sobrepasar antes de propagarse a otra neurona.

Las unidades se conectan con fuerzas de conexión variables (o ponderaciones). Los datos de entrada se presentan en la primera capa, y los valores se propagan desde cada neurona, hasta la neurona de la capa siguiente. Finalmente, se envía el resultado desde la capa de salida.

Inicialmente, todas las ponderaciones pueden ser aleatorias o especificadas por un programador, y en función de los resultados obtenidos, se deben modificar los pesos para conseguir mejores resultados.

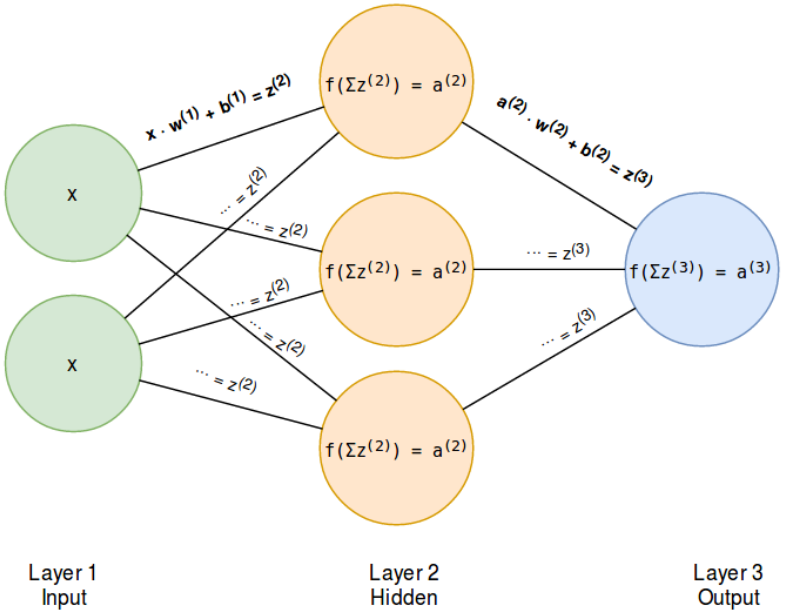


Figura 3.12: Esquema de una red neuronal [5]

3.3. Redes neuronales convolucionales (CNN)

Una parte esencial de este trabajo es la clasificación de imágenes. Para poder llevar a cabo esta tarea, se ha investigado entre los diferentes tipos de redes neuronales, seleccionando finalmente las redes neuronales convolucionales para su realización.

Una red convolucional es un tipo de red neuronal artificial, en donde las neuronas corresponden a campos receptivos de forma muy similar a las neuronas en la corteza visual primaria de los cerebros biológicos. Este tipo de red es una variación de un perceptrón multicapa para trabajar con el mínimo pre-procesamiento, en este caso de imágenes. Esto desemboca en que la red sea capaz de filtrar la información relevante por sí misma, no como en otros algoritmos en los que se ha de hacer a mano.

Para ello, se presentan continuamente a la red ejemplos para los que se conoce el resultado, y las respuestas que proporciona se comparan con los resultados conocidos. La información procedente de esta evaluación se pasa hacia atrás a través de las capas de la red, cambiando las ponderaciones gradualmente, y a medida que progresa el entrenamiento, la red aumenta su precisión a la hora de predecir. Una vez entrenada, la red se puede aplicar a casos en los que se desconoce el resultado y se quiere hallar su predicción.

3.3.1. Diseño

Una CNN consiste en una capa de entrada, una de salida y una cantidad variable de capas ocultas. Estas capas ocultas pueden ser de los tipos Convolutional, Pooling o Fully Connected.

- **Convolutional layers:** aplican una operación convolucional a la entrada y pasan los resultados a la siguiente capa. Solo se reciben subáreas de las salidas de la capa anterior. Estas capas emulan las neuronas visuales de un individuo.

- **Pooling layers** combinan las salidas de las neuronas anteriores y las pasan a una sola neurona en la siguiente capa.

- **Connected Layers:** conectan neuronas de una capa a neuronas de otra capa.

  Además, existen distintos tipos de neuronas con diferentes funciones:

- **Neuronas convolucionales:** están localizadas en las primeras capas de la CNN, y su función es la extracción de características, en donde se aplica una función a los datos de entrada. A la característica extraída se le añade un peso o influencia, y ese resultado se evalúa con una función de activación, donde se estudia si es prometedor.

- **Neuronas de reducción de muestreo:** estas neuronas se encargan de reducir el ruido y la complejidad en los datos de entrada, haciendo las imágenes más simples, de forma que el resultado de dos imágenes idénticas, pero tal vez con distinta iluminación, o un cambio mínimo en los objetos que hay en ella, produzca el mismo resultado.

- **Neuronas de clasificación:** La información que ya ha sido previamente filtrada y extraída, se clasifica en estas neuronas.

3.3.2. Hiperparámetros y definiciones relacionadas con las CNN

- **Epochs:** Es el número de veces que la CNN trata todas las imágenes dataset. En cada iteración se actualizan los pesos y se mejora el aprendizaje. El número de epoch depende del problema: pocas epochs podrían generar underfitting, y muchas epochs podrían dar lugar a overfitting, por lo que se debe buscar un valor intermedio.

  El underfitting, tal y como se muestra en la imagen derecha de la Figura 3.13, se produce cuando hay insuficientes epochs, e indica que el modelo no ha aprendido lo suficiente como para diferenciar y evaluar las distintas clases.

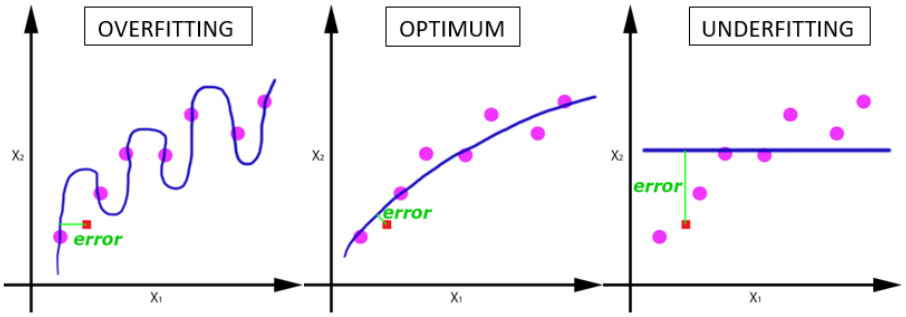


Figura 3.13: Epochs frente a predicción. Overfitting (izquierda), modelo óptimo (Centro), underfitting (derecha) [6]

Por otro lado, el overfitting se observa en la imagen izquierda de la Figura 3.13. Se origina cuando hay demasiadas epochs, e que implica que el modelo no sabe predecir nada que no sea idéntico a las imágenes con las que ha entrenado.

Por tanto, la clave se basa en poder generalizar la predicción (imagen central de la Figura 3.13), evitando el underfitting y el overfitting.

- **Batches:** Dada la gran cantidad de datos del dataset, la red neuronal no puede evaluarlo entero de una sola vez, por lo que se divide en bloques denominados batches.
- **Batch size:** Número total de ejemplos de entrenamiento presentes en un batch.
- **Iterations:** Es el numero de batches necesarios para completar una epoch.

3.4. Clasificación de imágenes con CNN

A continuación se describen distintos modelos de CNN que permiten abordar el problema de la clasificación. La selección de uno u otro depende de la complejidad del problema, por ejemplo, el número de características necesarias para diferenciar objetos. La extracción de más o menos características dependerá de la profundidad de la red.

3.4.1. Lenet5

La red más clásica y que primero se aprende, tanto por su diseño, como por su fácil entendimiento, es la arquitectura Lenet5, la cual se caracteriza por el pequeño tamaño de la imagen de entrada, lo que hace que disminuya la complejidad, y que en consecuencia, funcione muy bien sobre una CPU

sin necesidad de tener aceleradores para el procesamiento. Inicialmente, fue creada para el reconocimiento de caracteres escritos a mano por *Yan Lecun, Leon Bottou, Yosuha Bengio y Patrick Haffner* en los 90.

Tal y como se muestra en la Figura 3.14, consta de dos capas convolucionales, dos capas de pooling y otras dos capas connected.

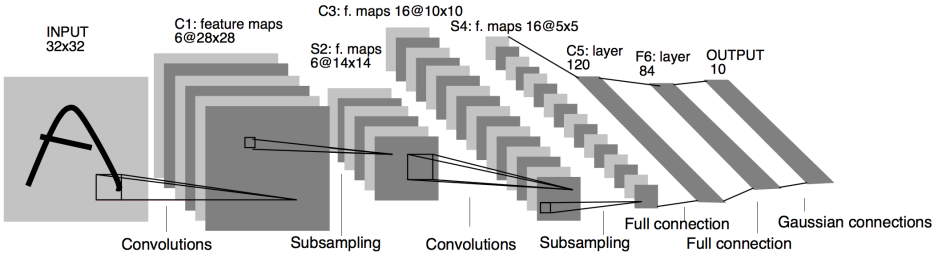


Figura 3.14: Arquitectura LeNet 5 [7]

3.4.2. AlexNet

La arquitectura AlexNet es muy similar a Lenet5 (Figura 3.15). Su diferencia radica en que tiene más filtros por capa, por lo que se obtienen más características, el tamaño de la imagen de entrada es mucho mayor (227x227), y además, puede usarse un acelerador, y por tanto, puede trabajar con imágenes más grandes que Lenet5. Fue creada por *Alex Krizhevsky* y publicada por *Ilya Sutskever* y *Geoffrey Hinton*.

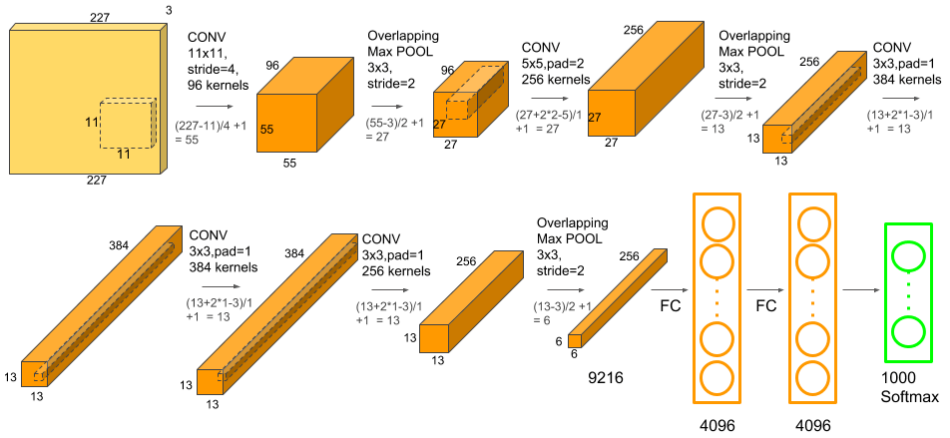


Figura 3.15: Arquitectura AlexNet [8]

3.4.3. VGG

Otro modelo de CNN es VGG (Figura 3.16), similar a AlexNet. Dispone de una arquitectura muy uniforme, y es la que más se usa para extraer características de las imágenes. Fue creada por *Simonyan* y *Zisserman*.

Figura 3.16: Arquitectura VGG [9]

3.5. Detección de objetos en imágenes

A continuación se describen algoritmos y consideraciones para la detección de imágenes.

3.5.1. Bounding boxes

Las bounding boxes son cajas imaginarias de diferentes dimensiones que generan los algoritmos sobre las imágenes, y predicen si dentro de sus límites hay un objeto. Si lo hay, guardan la información sobre las coordenadas del centro del objeto, las dimensiones de la caja, la clase a la que pertenece el objeto y una tasa de confianza para la predicción. La mayoría de las bounding boxes serán descartadas, ya sea porque el valor de confianza es demasiado bajo, o porque muchas de ellas predicen el mismo objeto de la imagen con una alta confianza, causando una redundancia que deberá ser eliminada. Las dimensiones de las bounding boxes pueden especificarse y de denominan anchors.

3.5.2. Redes neuronales convolucionales basadas en la región (R-CNN)

Las R-CNN son uno de los primeros detectores de objetos basados en el aprendizaje profundo. Este algoritmo de detección de objetos se divide principalmente en tres partes, tal y como se recoge en la Figura 3.17:

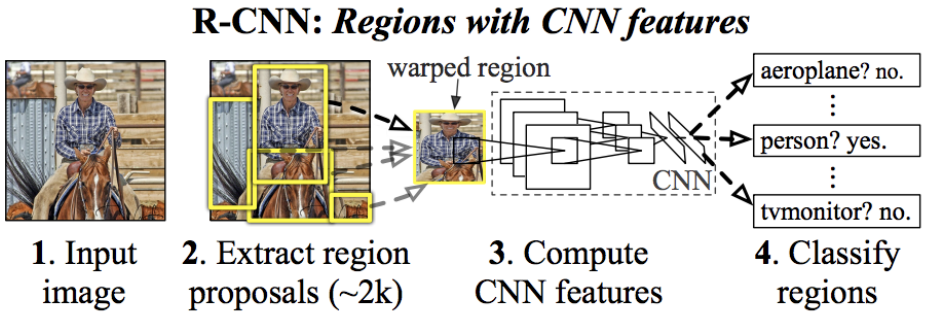


Figura 3.17: Funcionamiento de una R-CNN [10]

3.5.3. Búsqueda selectiva

Se generan aproximadamente 2000 bounding boxes mediante filtros de color, texturas, bordes, etc. (Figura 3.18) sobre la imagen.

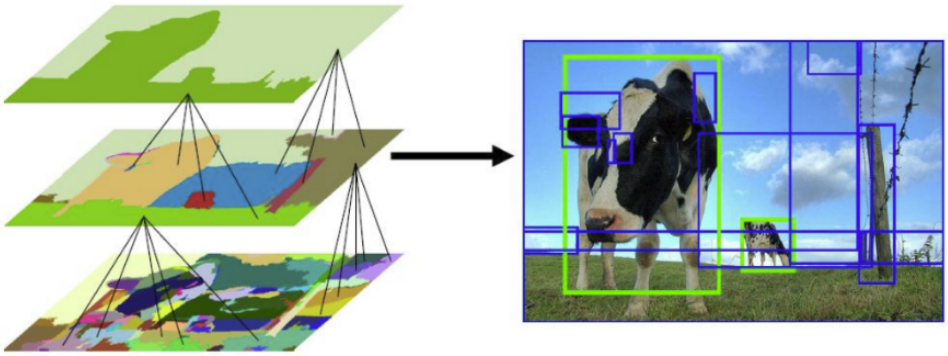


Figura 3.18: Ejemplo de una búsqueda selectiva [10]

3.5.4. Aplicación de la CNN

Se ha aplicado la CNN a cada bounding box generada en la etapa anterior, clasificando así cada una de las regiones.

3.5.5. Eliminación de ruido

La eliminación de ruido y redundancia que puede producirse si varias regiones hacen predicciones incorrectas o las mismas, se realiza con técnicas como non-max suppression y thresholding.

3.5.6. You Only Look Once (YOLO)

YOLO[12] es un algoritmo de detección de imágenes cuya principal característica es la realización en un sólo paso de la clasificación y la localización de objetos en imágenes.

3.5.7. Idea de alto nivel

Para detectar objetos en una imagen, YOLO ejecuta una serie de pasos, que son los que se recogen en la Figura 3.19.

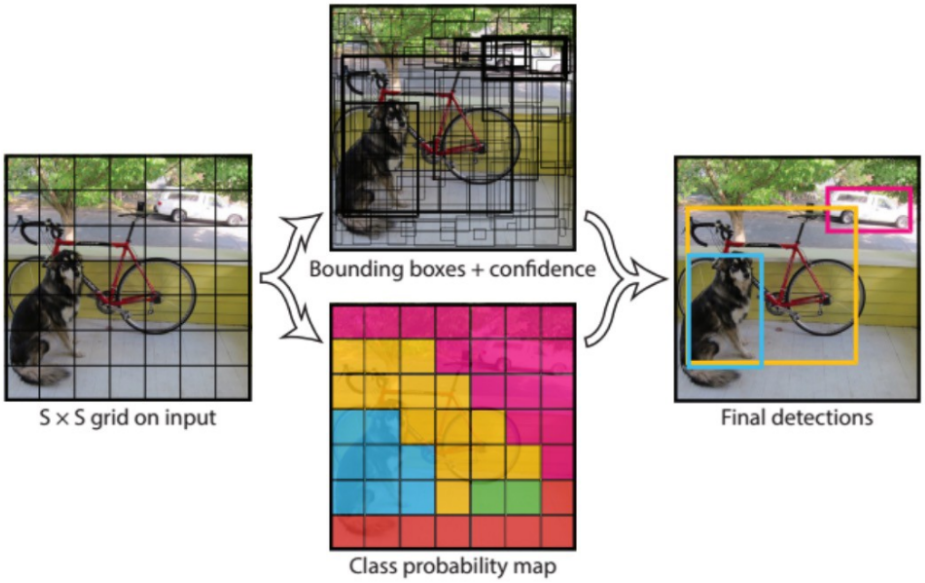


Figura 3.19: Funcionamiento del algoritmo YOLO [11]

Para ello, en primer lugar divide la imagen en MxM celdas (el número de celdas depende principalmente de las dimensiones de la imagen a tratar), y posteriormente, cada celda se encarga de predecir las bounding boxes deseadas. Cada una de ellas sólo devuelve una predicción y una probabilidad de que el objeto predicho sea de alguna clase.

Una vez realizadas las predicciones, se elimina el ruido y la redundancia.

3.5.8. YOLO CNN

La red neuronal consta de 24 capas convolucionales, seguida de 2 capas totalmente conectadas. Además, se añaden capas convolucionales de 1x1 para reducir las características de las capas anteriores (Figura 3.20).

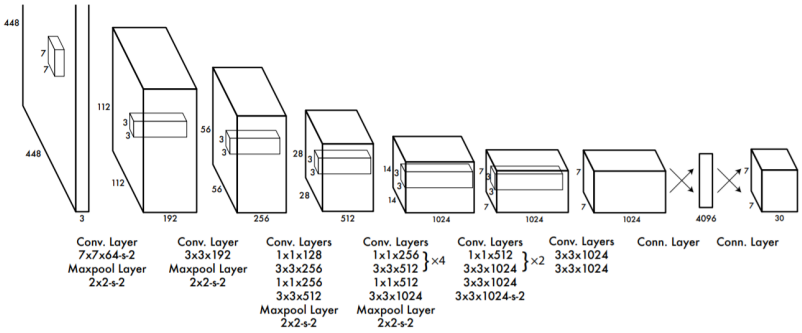


Figura 3.20: Arquitectura de la red neuronal [11]

Cabe resaltar que es posible mejorar la precisión pre-entrenando las capas convolucionales, y posteriormente haciendo la detección sobre el doble de la resolución.

La CNN para el YOLO debe de ser configurada en base al número de clases que se quieren identificar, las dimensiones de los objetos y la definición de las medidas de los anchors.

En función del número de clases se deben especificar las dimensiones de salida de cada capa convolucional, y en función de las dimensiones de los objetos a detectar se definen los tamaños prioritarios para las bounding boxes. Por último, se estudia el conjunto de datos y las dimensiones de los objetos a detectar con el objetivo de definir unos anchors adecuados.

3.5.9. Non max supression y Threshold

Con el fin de eliminar el ruido y la redundancia de la imagen se aplican las dos técnicas que se describen a continuación:.

La primera de ella, Non max supression [13], trata de eliminar resultados redundantes de la imagen para que sólo quede una caja por objeto, tal y como puede observarse en la Figura 3.21.

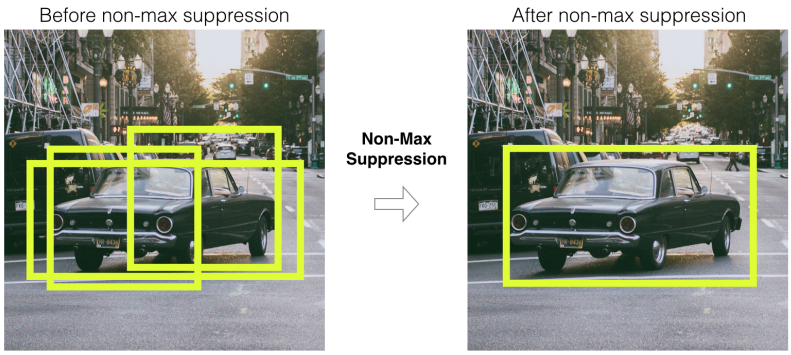


Figura 3.21: Aplicación de la técnica non max suppression

Esto se consigue eliminando las cajas que tengan menor confianza de las que se superpongan. También se usa para mejorar el área predicha juntando el área de las cajas superpuestas que tienen una buena confianza.

La segunda de ellas,Threshold, establece un límite entre las predicciones que se aceptan y las que no, lo que permite eliminar ruido de la imagen. En la Figura 3.22 se recoge un ejemplo del funcionamiento con un Threshold adecuado y uno demasiado bajo. Como puede observarse, un valor demasiado bajo acepta predicciones con una probabilidad muy baja, por lo que no se filtra el ruido ni la redundancia, y por tanto, da lugar a un mal funcionamiento.

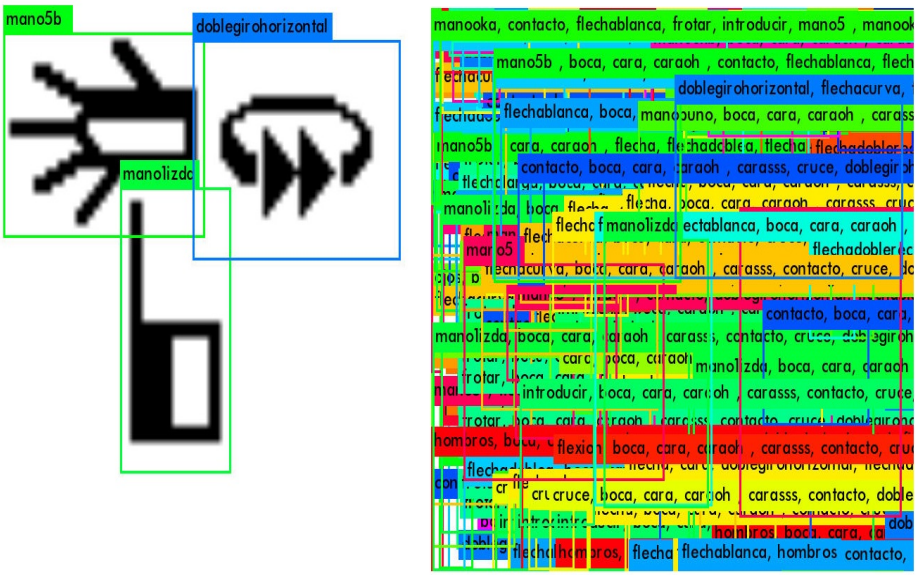


Figura 3.22: Imagen con un valor adecuado de threshold y con un threshold demasiado bajo

3.5.10. Darknet

Darknet es un framework de código libre disponible en GitHub desarrollado por Joseph Redmon [14] y que implementa el algoritmo YOLO. Se trata de la herramienta que se utilizará para reconocer los símbolos de la SignoEscritura del dataset.

Darknet dispone de pesos pre-entrenados para detectar objetos comunes como personas, coches, bicicletas. Por lo tanto, no hace falta entrenar la CNN si sólo se quieren detectar estos objetos. En la Figura 3.23 se observa el resultado de los pesos pre-entrenados.

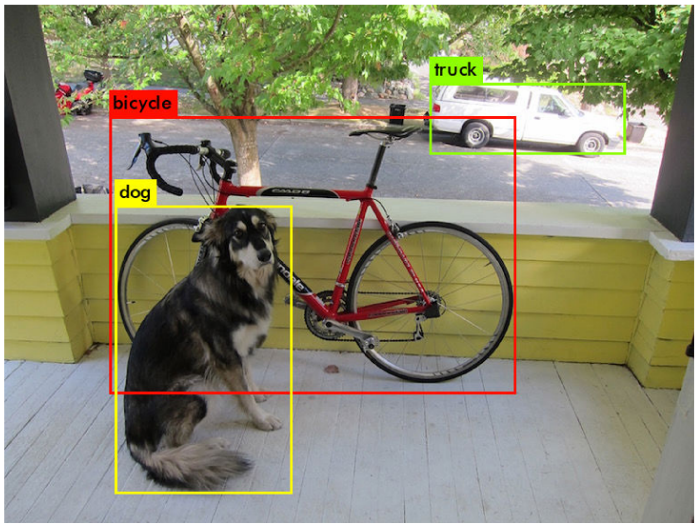


Figura 3.23: Detección de un perro, una camioneta y una bicicleta

En esta Figura se muestra una imagen sencilla, pero también funciona correctamente con imágenes más complejas, en donde existe mayor número de objetos, y además están superpuestos, como en la Figura 3.24.

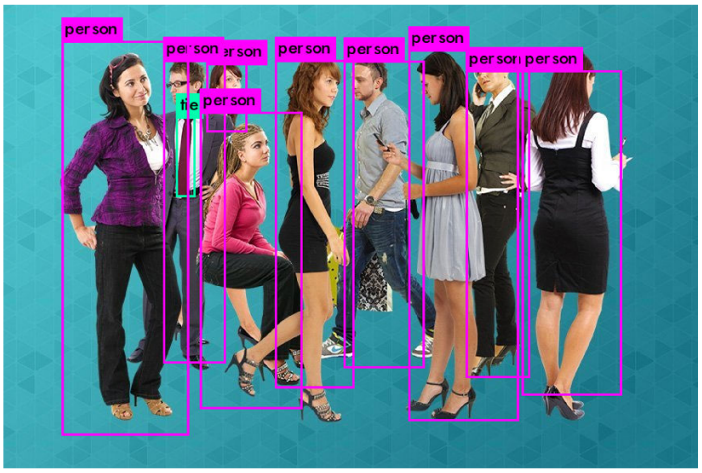


Figura 3.24: Detección de personas y una corbata

Para realizar un entrenamiento en Darknet se deben preparar los datos, y además se debe definir, tanto la CNN, como las heurísticas para buscar los objetos, como por ejemplo, definiendo unos anchors adecuados en función de las dimensiones de los objetos que se buscan.

3.5.11. Prepararación de los datos

Para el uso del algoritmo YOLO es necesario preparar el dataset, indicando en cada imagen dónde están los objetos que se quieren reconocer y asignándoles una clase. Este proceso se debe repetir en cada una de las imágenes. Para ello, se usa una herramienta llamada Yolo-Mark [15], a la cual se deben especificar las clases que se desean conocer y la localización de las imágenes a marcar.

Tal y como se muestra en la Figura 3.25, se deben marcar los objetos que hay en cada una de las imágenes, además de indicar la clase a la que pertenecen, como es en este caso el marcaje de los aviones. Este proceso se debe repetir para cada una de las imágenes del dataset.

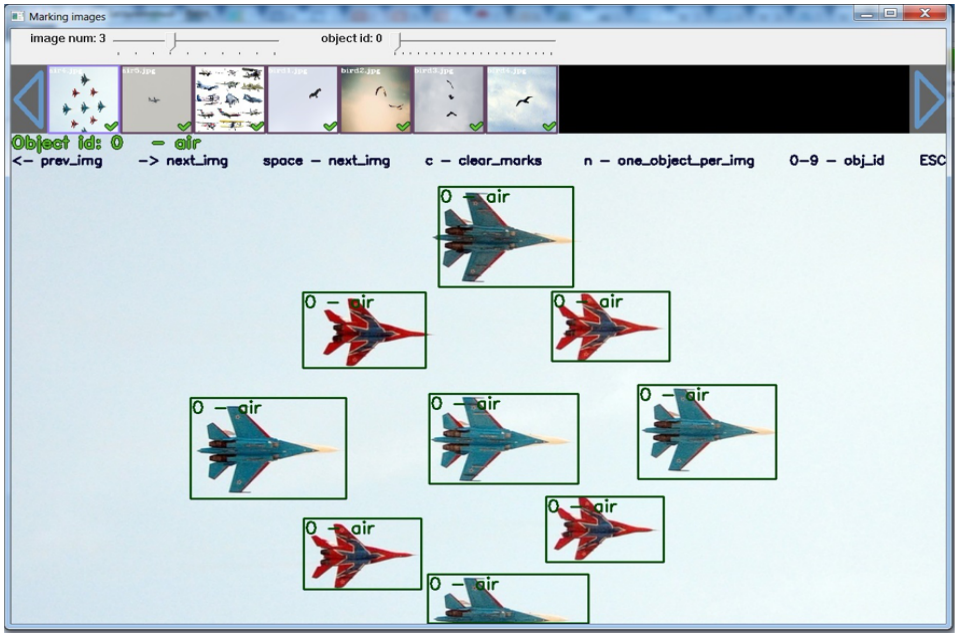


Figura 3.25: Ejemplo de uso de la herramienta Yolo Mark

Una vez finalizado el proceso, la herramienta devolverá las anotaciones correspondientes a cada imagen en formato txt y con el mismo nombre de la imagen con la localización de los objetos. El formato de las anotaciones será: número de la clase correspondiente a la caja, coordenadas x e y del centro de la caja y dimensiones (altura y anchura) de la misma.

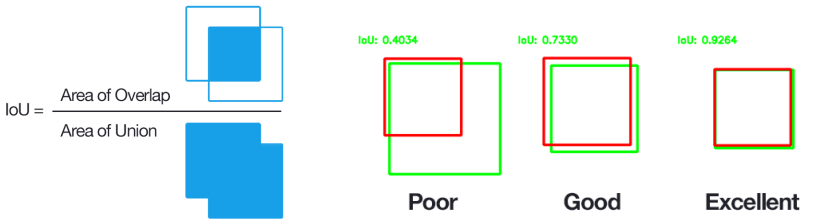
3.6. Métricas de precisión

Para medir la precisión de los algoritmos de detección de objetos se han usado diferentes métricas. Las más relevantes son: IoU (intersección sobre unión), mAP (mean average precision) y coste (Avg Loss).

mAP mide la media del porcentaje de predicciones correctas para cara clase.

Coste (Avg Loss): Esta métrica registra el error en las predicciones del modelo. A mayor coste, peores resultados se obtienen.

Intersección sobre unión Cociente de la intersección entre la unión de las bounding boxes solapadas con predicciones de clase iguales (Figura 3.26a). Cuanto mejor sea la predicción, más se acerca a un valor de 1, y cuanto peor es, más se acercará a 0 (Figura 3.26b).



(a) Intersección sobre unión. [16]

(b) Puntuación de Intersección sobre unión [16]

Figura 3.26: Explicación del IoU

3.7. Librerías

3.7.1. TensorFlow

Inicialmente, Google Brain, un equipo de científicos de Google con experiencia en computación, crearon DistBelief como un sistema propietario de aprendizaje automático basado en redes neuronales de aprendizaje profundo o deep learning. Más tarde, Google asignó múltiples científicos computacionales, incluyendo a Jeff Dean para simplificar y reconstruir el código base de DistBelief en una biblioteca de aplicación mas rápida y robusta, cuyo resultado es TensorFlow.

En 2009, el equipo dirigido por Geoffrey Hinton había implementado una propagación hacia atrás generalizada, así como otras mejoras que permitieron generar redes neuronales mejores, por ejemplo, una reducción del 25 % de errores en reconocimiento del habla.

En el artículo sobre TensorFlow para principiantes [17] se índica el propósito de usar la librería es generar un grafo computacional que se puede ejecutar de manera mucho más eficiente. TensorFlow primero genera el grafo de tensores y su flujo de datos para, una vez ejecutado, ser más rápido y eficiente. También puede calcular automáticamente los gradientes que se necesitan para optimizar las variables del grafo, a fin de que el modelo funcione mejor. Esto se debe a que el grafo es una combinación de expresiones matemáticas simples, por lo que el gradiente de todo el grafo se puede calcular

utilizando la regla de la cadena para el cálculo de las derivadas al optimizar la función de coste. Internamente, los cómputos están expresados como estructuras denominadas stateful dataflow graphs.

Estas estructuras constan de varias partes:

- Placeholder: variables utilizadas para transformar los datos de entrada a un grafo.
- El modelo: se trata de una función matemática que calcula los resultados a partir de las variables Placeholder.
- Coste: medida que se puede usar para guiar la optimización de las variables.
- Método de optimización: actualiza las variables del modelo.

Por ejemplo, suponiendo que se tiene el siguiente código:

```
a = tf.add(3, 5)
print(a)

# Tensor("Add_3:0", shape=(), dtype=int32)
```

TensorFlow crea un nodo suma (Add) a partir del código de mostrado anteriormente, y dado que no se definen los nombres en las entradas, TensorFlow los determina automáticamente. En este caso, x es 3, e y es 5. En el grafo, los nodos representan operaciones, variables y constantes, y las aristas, que son las que conectan los nodos, son los tensores, que representan datos y fluyen a través del grafo. Los datos llegan a nodos que realizan operaciones con ellos. Es por eso que se llama TensorFlow. El grafo está compuesto por operaciones, ops, las cuales reciben como entrada cero, o más tensores o pueden generar nuevos tensores [18].

```
    # Ejemplos de tensores de rango 0 o escalares
    nombre   = tf.Variable("Jose", tf.string)
    edad     = tf.Variable(45, tf.int16)
    estatura = tf.Variable(1.75, tf.float64)
    complex  = tf.Variable(12.3 - 4.85j, tf.complex64)

    # Ejemplos de tensores de rango 1 o Vector
    vector_string  = tf.Variable(["uno", "dos"], tf.string)
    vector_int     = tf.Variable([1, 2], tf.int32)
    vector_floats  = tf.Variable([1.43, 3.57], tf.float32)
    vector_complex = tf.Variable([12.3 - 4.55j, 3.8 - 5.78j], tf.complex64)

    # Ejemplos de tensores de rango 2 o superior (Matriz)
```

```
matrix_string = tf.Variable([["uno"], ["dos"]], tf.string)
matrix_int    = tf.Variable([[1, 2], [4]], tf.int16)
matrix_floats = tf.Variable([[]1.43, 2.4], 3.57], tf.float32)
matrix_bool   = tf.Variable([[True, False],[False, True]], tf.bool)
```

Actualmente, TensorFlow es la plataforma de Aprendizaje Profundo más importante del mundo. Se puede usar sobre CPU y GPU, y cuenta con una herramienta de visualización llamada TensorBoard, que facilita la depuración para la optimización de las redes neuronales [19].

3.7.2. Keras

Keras [20] es una biblioteca de Redes Neuronales de Código Abierto escrita en Python que es capaz de ejecutarse sobre TensolFlow, Microsoft Cognitive Toolkit o Theano. Está especialmente diseñada para posibilitar la experimentación en, relativamente poco tiempo, con redes de Aprendizaje Profundo o Deep Learning. Sus fuertes se centran en ser amigable para el usuario, además de modular y extensible. A continuación se muestra un ejemplo de cómo se definen capas en Keras:

```
# ejemplo capa convolucional
model.add(Conv2D(filters = 6,
    kernel_size = (5, 5),
    strides = 1,
    activation = 'relu',
    input_shape = input_image_shape
))
"""
con esta capa se le dice que se quieren 6 filtros de 5 x 5,
que vaya de 1 en 1 celdas, y que use como función de activa-
ción la función relu. Al ser la primera capa, se de debe dar
el tamaño de imagen que se le va a a meter
en input_shape.
"""

# ejemplo maxpooling
model.add(MaxPooling2D(filters = 16,
    kernel_size = (5, 5),
    strides = 1,
    activation = 'relu'
))
"""
Este maxpooling viene definido por 16 filtros de 5 x 5, y se
le indica que vaya de 1 en 1 celda, y que use como función
```

```
de activación la función relu.
"""

# ejemplo capa fully connected (red neuronal)
model.add(Dense(units = 30,
    activation = 'relu'
))
"""
Esta capa viene definida por 30 neuronas, cuya función de
activación será relu.
"""
```

Tal y como se acaba de demostrar, es muy sencillo crear una red neuronal convolucional en poco tiempo y sin mucha dificultad. Estos son sólo unos ejemplos de los múltiples tipos de capas que se se pueden introducir [21].

Inicialmente, fue desarrollada como parte de los esfuerzos de investigación del proyecto ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System). Su autor principal y mantenedor ha sido el ingeniero de Google François Chollet, el cual explicó que Keras fue creado para actuar como interfaz, en lugar de ser otro framework, de ahí que sea más sencillo de usar que TensorFlow. No sólo soporta Redes Neuronales Convolucionales, sino que también soporta Redes Neuronales Recurrentes, y al funcionar por encima de frameworks, como TensorFlow, también permite ser ejecutado con GPU para un entrenamiento mucho mas rápido.

3.7.3. Google Colaboratory

Uno de los problemas de desarrollar proyectos de machine learning es el entrenamiento del modelo, ya que suele entrenarse con miles de ejemplos, y puede tardar mucho tiempo si se utiliza un entrenamiento tradicional con CPU. Sin embargo, dada la índole de las operaciones, éstas son paralelizables, ya que no hay dependencia de datos entre ellas, por lo que se puede usar un acelerador como una GPU para entrenar y reducir así significativamente los tiempos de entrenamiento.

Google Colaboratory provee un notebook que puede ejecutarse de forma remota en una máquina, con el fin de entrenar modelos de Machine Learning, además de permitir configurar el entorno para utilizar una GPU (Tesla K80), y así agilizar los cálculos. Además, es una herramienta totalmente gratuita. Su único requisito es la imposición de no poder ejecutar un programa que tenga ejecución infinita, principalmente para evitar que se use para minar bitcoins.

El funcionamiento es exactamente igual a jupyter notebook, ya que sigue el mismo modelo de cajas de código que pueden ser ejecutadas independientemente.

Tal y como puede observarse en la Figura 3.27, este notebook puede utilizar python y comandos de la shell de Linux, lo que permite, si previamente se ha montado una carpeta que se tenga en nuestro drive, instalar librerías con *'pip install'* o descargarse cualquier repositorio de GitHub.

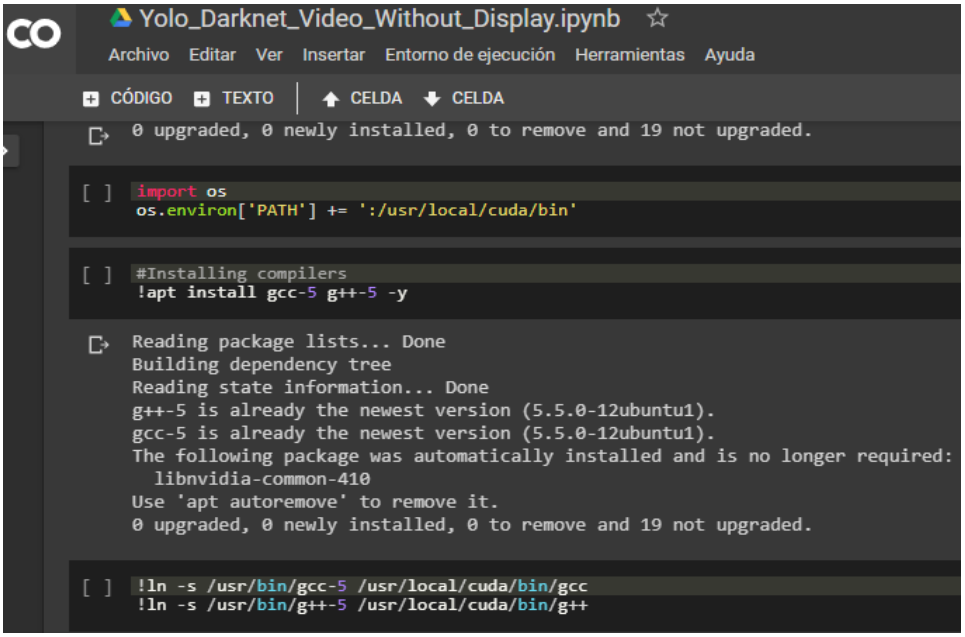


Figura 3.27: Interfaz de Google colaboratory

Un inconveniente de esta herramienta es que el entorno de ejecución se borra automáticamente cuando se queda inactivo, es decir, es necesario instalar todas las dependencias, librerías etc. cada vez que alguien se conecta. Sin embargo, esto no supone un gran problema, ya que almacena automáticamente todo el código, y simplemente hay que ejecutar todas las celdas.

Capítulo 4

Desarrollo

4.1. Dataset original

El dataset con el que se ha trabajado ha sido un subconjunto [22] de la SignoEscritura obtenido de SignWritting [2], y se encuentra dividido en 2 partes: símbolos y transcripciones. Concretamente, 31 símbolos y 52 transcripciones.

Los símbolos representan algo indivisible, por ejemplo, una mano, una cabeza, una dirección, etc., mientras que las transcripciones son conjuntos de símbolos que representan una palabra en la SignoEscritura.

Por ejemplo, en la Figura 4.1 se tienen tres símbolos: una cabeza, una mano y un contacto. Esta transcripción tiene el significado de callarse.

Figura 4.1: Ejemplo de transcripción del dataset

Es necesario tener en cuenta que no sólo importan los símbolos que están en la transcripción, sino también su posición, por lo que es muy importante detectar qué símbolos hay, y dónde se encuentran dentro de la imagen.

4.2. Clasificación de símbolos

En esta sección se muestra el desarrollo del clasificador mediante una CNN.

4.2.1. Preprocesamiento del dataset

El dataset presenta diferentes dificultades para el entrenamiento, entre las que se encuentra tener pocos ejemplos, y además una baja resolución de éstos.

Para aumentar el número de ejemplos, se ha creado una función que recibe una imagen y un número, y devuelve un array con la imagen rotada ese número de veces. Se ha llamado a la función con números altos para todos los símbolos, y aunque se generen símbolos prácticamente iguales, se consigue aumentar en gran medida el volumen de datos.

Otro de los problemas es que las imágenes no tienen la misma resolución, tal y como puede observarse en la Figura 4.2. Debido a que las redes neuronales sólo aceptan entradas del mismo tamaño, se han redimensionado todas las imágenes para que su tamaño sea el mismo. Además, todas las imágenes se han hecho cuadradas para que no se modifique su resolución al rotarlas (imagen de la derecha en la Figura 4.2).

Figura 4.2: Flecha en su estado original y tras el redimensionado

4.2.2. Experimentos con CNN para símbolos

Tras comentar las arquitecturas de CNN en el capítulo anterior, se van a mostrar los experimentos que han sido realizados y las pruebas que se han llevado a cabo. Para ello, se ha usado un dataset formado por 31 símbolos, en donde cada imagen es un símbolo independiente, y 52 transcripciones. A menos que se especifique lo contrario, todos los experimentos se han realizado con 30 epochs y un batch size de 32.

En primer lugar, se ha investigado cuales son las arquitecturas de CNN más usadas o conocidas, y tras este estudio, se han realizado experimentos para construir una CNN capaz de distinguir entre los símbolos proporcionados por el dataset. Como arquitectura para la CNN se ha empezado probando AlexNet para ver su funcionamiento. Esta arquitectura acepta imágenes de como mínimo de 227x227, por lo que se han tenido que redimensionar los símbolos en gran medida, ya que eran de 30x30, o 40x40 de media. Aunque era posible que esta redimensión afectara al rendimiento, aun así se ha probado para estar completamente seguros.

Sobre el papel, AlexNet debería funcionar mejor que LeNet5, ya que tiene más capas, y esto, en teoría, se traduce en más características de las imágenes. Sin embargo, el resultado con AlexNet ha sido no ha sido el esperado, ya que

se ha obtenido un coste de 15 y una precisión muy baja. Además, ni el coste ha disminuido, ni la precisión ha aumentado.

Este resultado puede deberse a que al redimensionar la imagen significativamente (de 30 píxeles aproximadamente a 227), se pierde mucha información y los símbolos quedan confusos. Esto ocurre porque en cada convolución pasa un filtro que se encarga de detectar los bordes de las figuras, y si se ha redimensionado mucho la imagen, ésta se ve emborronada; se difumina mucho, tanto la figura, como los límites del símbolo. Por esta misma razón, no se ha probado la arquitectura VGG, ya que esta configuración también acepta imágenes de dimensiones grandes, como AlexNet (unos 224x244 píxeles).

Por tanto, se ha probado directamente LeNet5, ya que esta arquitectura acepta imágenes de como mínimo 32x32, y no es necesario redimensionarlas tanto. Esta arquitectura ha devuelto mejores resultados, y su implementación en Keras ha sido la siguiente:

```
# se define el modelo secuencial.
model = Sequential()

# capa 1
model.add(Conv2D(filters = 6,
    kernel_size = (5, 5),
    strides = 1,
    activation = 'relu',
    kernel_regularizer = l2(0.01),
    input_shape = input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2),
    strides = 2))
model.add(Dropout(0.25))

# capa 2
model.add(Conv2D(filters = 16,
    kernel_size = (5, 5),
    stides = 1,
    kernel_regularizer = l2(0.01),
    activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2),
    strides = 2))
model.add(Dropout(0.5))

# capa 3
model.add(Flatten())
model.add(Dense(units = 120, activation = 'relu',
```

```
        kernel_regularizer = l2(0.01)))

    # capa 4
    model.add(Dense(units = 84, activation = 'relu',
        kernel_regularizer = l2(0.01)))

    # capa 5
    model.add(Dense(units = 31, activation = 'softmax'))
```

Como se puede observar, se tiene un modelo de 5 capas listo para ser ejecutado. Prácticamente cada una de las partes está explicada en el capítulo de Keras de la introducción, a excepción de dos cosas: Dropout y Kernel Regularizer.

Dropout es una técnica propuesta por Srivastava [23], y consiste en seleccionar aleatoriamente neuronas e ignorarlas durante el entrenamiento. Esto quiere decir que su contribución a la activación de las siguientes neuronas es eliminada temporalmente en la etapa de propagación de la información hacia delante. Además, cualquier actualización de los pesos no se aplica a estas neuronas eliminadas en la etapa de propagación de información hacia atrás. A medida que la neurona aprende, los pesos se establecen en función del contexto que ésta ocupa en la red. Los pesos de las neuronas están configurados para características específicas que proporcionan cierta especialización. Las neuronas vecinas se vuelven dependientes de esta especialización, por lo que si se lleva lejos, esta especialización puede dar a un modelo muy ajustado a los ejemplos de entrenamiento. Entonces, si aleatoriamente se eliminan las neuronas, se consigue que el resto tenga que hacerse cargo de aprender características, especializarse y hacer predicciones de las neuronas que faltan.

Con esto, se consigue que las neuronas sean menos sensibles a los pesos, y por lo tanto, se obtiene un modelo que generaliza mejor.

Por otra parte, el kernel regularizer se encarga de meter regularización a la matriz de pesos [24].

El dataset de símbolos está formado por 31 símbolos, pero de cada uno sólo se tiene una muestra. Para ampliar el conjunto de datos y poder entrenar la CNN de manera que fuera capaz de reconocer cada símbolo independientemente con una probabilidad aceptable, se ha estudiado cual iba a ser el input del programa final, es decir, las transcripciones. Tras este análisis, se ha concluido que la única variación posible es que los símbolos aparezcan rotados, así que se ha rotado cada símbolo varias veces.

El programa se ha lanzado con la siguiente configuración:

- Rotaciones: 8.
- Conjunto de validación: 5 ejemplos.
- Conjunto de testing: 5 ejemplos.

Con esto, se tienen 248 ejemplos de entrenamiento que, quitando 5 ejemplos validación, y otros 5 para testing, se quedan en 238. A través de esta modificación se obtiene una buena precisión en el entrenamiento, tal y como se muestra en la Figura 4.3. Sin embargo, el coste es algo elevado (Figura 4.4), lo que puede afectar en la predicción de los símbolos.

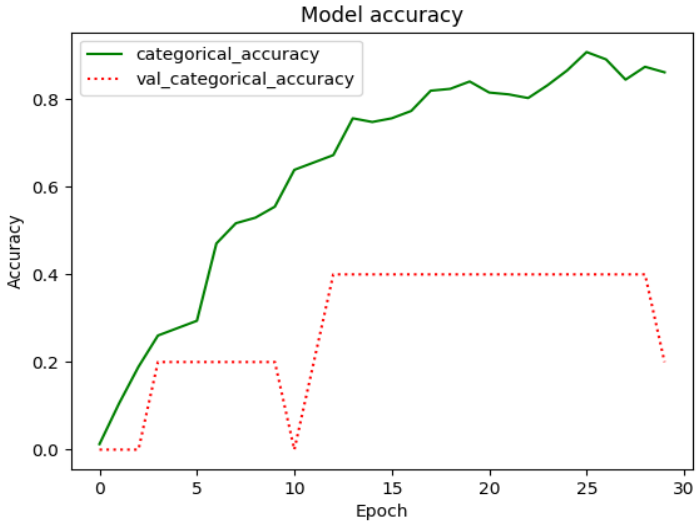


Figura 4.3: Precisión con 8 rotaciones

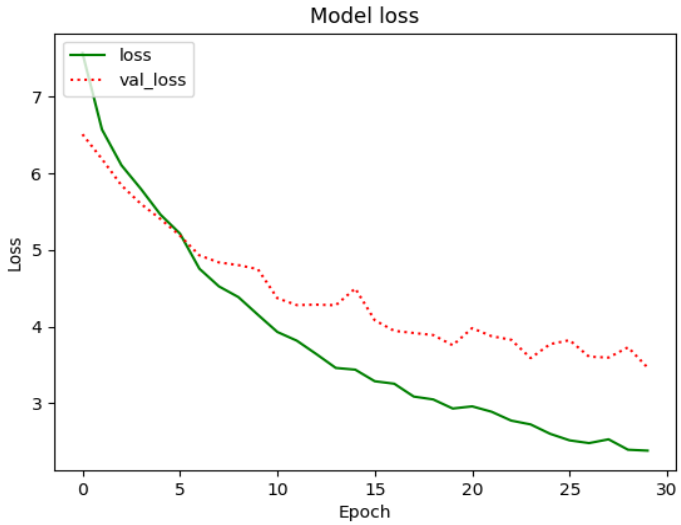


Figura 4.4: Coste con 8 rotaciones

Por otra parte, la precisión en el conjunto de validación es muy baja, tan sólo de un 30 %, y el coste un poco más elevado que en el entrenamiento, concretamente de 3,46, con un cierto matiz, y es que la precisión en el entrenamiento sube y el coste baja, descartando así el overfitting.

Aun así, han sido necesarios más ejemplos de entrenamiento, ya que aunque el modelo generaliza bien dentro de los pocos datos que tiene, la precisión en el conjunto de validación es baja. Para entenderlo mejor se adjunta el testing en la Tabla 4.1.

Símbolo	Predicción	Probabilidad	Resultado
flechadoblegirodoble	flechadoblegirodoble	40 %	Acierto
introducir	introducir	68 %	Acierto
manooka	manookb	44 %	Fallo
flexion	flechalarga	43 %	Fallo
flechadoblegiro	manounoblanca	46 %	Fallo

Tabla 4.1: Resultados del testing con 8 rotaciones

Aunque acierta con la identificación del símbolo de la flecha, la probabilidad es muy baja, con apenas un 40 %. Además, también acierta, incluso con más probabilidad, el símbolo de introducir con un 68 %. Esto se debe a que no se parece a ningún símbolo otro del dataset, y por tanto, es fácil de diferenciar del resto.

Sin embargo, con las manos se equivoca bastante al parecerse mucho a otros elementos, por lo que se ha aumentado el número de rotaciones. Las estadísticas de recogen a continuación en la Tabla 4.2.

	Training	Validation	Testing
Precisión	86 %	30 %	40 %
Coste/Fallo	2.38	3.46	60 %

Tabla 4.2: Estadísticas con 8 rotaciones

En el siguiente experimento se ha aumentado el numero de símbolos al incrementar el numero de rotaciones por símbolo. Su configuración es la siguiente:

- Rotaciones: 40.
- Conjunto de validación: 20 ejemplos.
- Conjunto de testing: 20 ejemplos.

Con esta configuración se tienen 1240 símbolos, aunque quitando 20 para la validación, y otros 20 para testing, se queda en 1200 ejemplos para entrenar. Una vez ejecutado se han obtenido los siguientes resultados:

En la Figura 4.5 se puede comprobar que la precisión fluctúa bastante, aunque el coste se mantiene decreciente (Figura 4.6).

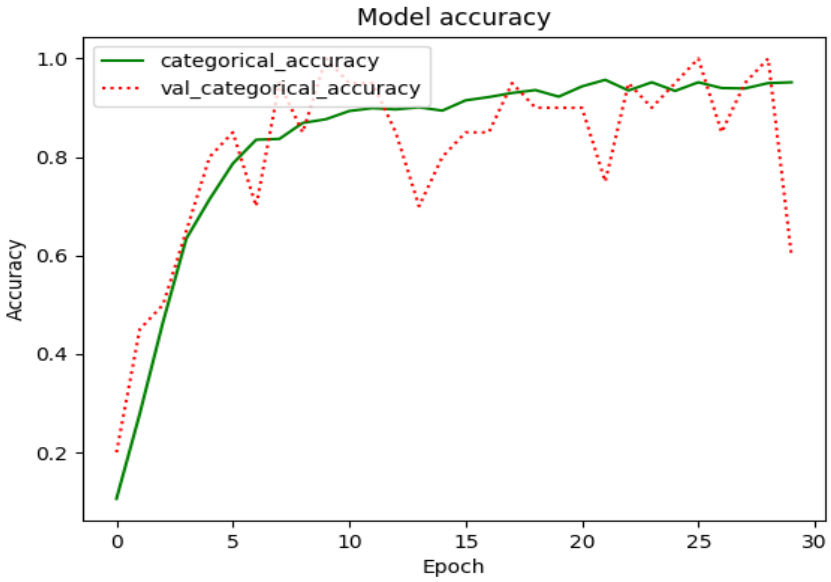


Figura 4.5: Coste con 40 rotaciones

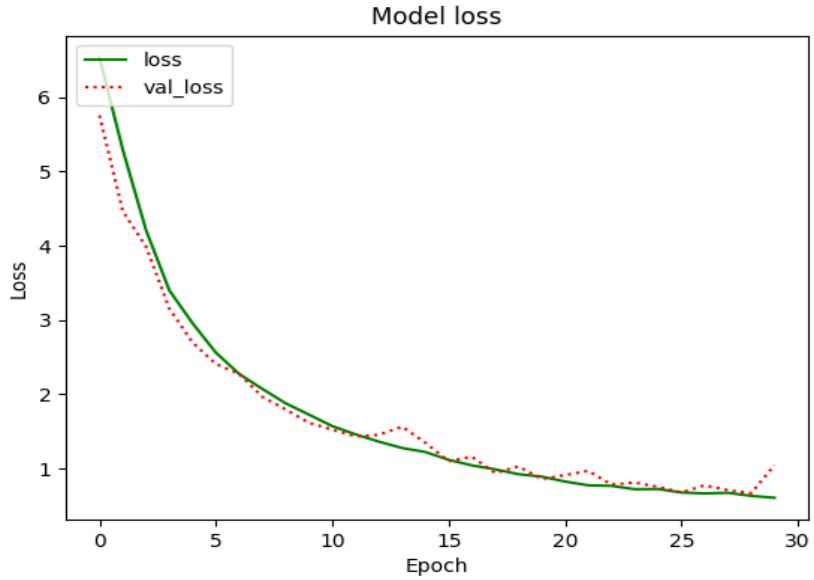


Figura 4.6: Coste con 40 rotaciones

Esto puede deberse a que no se ha aplicado la regularización, ya que quedó descartada desde el principio al aplicar una función de Keras llamada Dropout y kernel regularizer, tal y como se ha descrito anteriormente.

Otro posible motivo de la fluctuación, se debe a tener insuficientes ejemplos de entrenamiento.

Además, en la Tabla 4.3 se adjuntan los resultados del testing, de modo que se puedan extraer más conclusiones sobre su funcionamiento.

Símbolo	Predicción	Probabilidad	Resultado
flechadoblegiro	flechadoblegiro	93 %	Acierto
flecha	flecha	93 %	Acierto
doblegirohorizontal	doblegirohorizontal	83 %	Acierto
doblegirohorizontal	doblegirohorizontal	68 %	Acierto
flexion	flechalarga	43 %	Fallo
cruce	cruce	100 %	Acierto
flechadobleb	cruce	54 %	Fallo
mano1dcha	mano1dcha	62 %	Acierto
carasss	ojos	54 %	Fallo
flechadobleb	cruce	34 %	Fallo
ojos	carasss	52 %	Fallo
flechadoblegirodoble	flechadoblegirodoble	56 %	Acierto

Tabla 4.3: Testing con 40 rotaciones

También se puede comprobar que tiene buena precisión de manera general, aunque con las caras y las flechas siguen teniendo problemas en muchas ocasiones, pese a no arrojar malos resultados en las estadísticas finales (Tabla 4.4).

	Training	Validation	Testing
Precisión	95 %	60 %	90 %
Coste/Fallo	0.6	1.04	10 %

Tabla 4.4: Estadísticas con 40 rotaciones

Finalmente, se ha aumentado el número de rotaciones para descartar la causa de tener pocos ejemplos de entrenamiento.

En el siguiente experimento se ha ejecutado el programa con la siguiente configuración:

- Rotaciones: 80.
- Conjunto de validación: 100 ejemplos.

- Conjunto de testing: 100 ejemplos.

Con esta modificación se han obtenido 2480 ejemplos, de los que 100 forman parte del conjunto de validación, y otros 100 del conjunto de testing, quedando así 2280 ejemplos para entrenar.

La precisión en el conjunto en el entrenamiento y de validación se muestra en la Figura 4.7, la cual presenta un buen resultado, y además se consigue que la fluctuación sea menor.

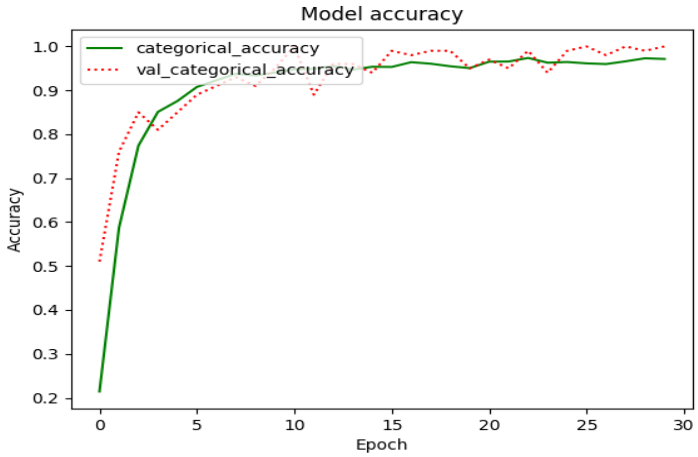


Figura 4.7: Precisión con 80 rotaciones

En la Figura 4.8 también se puede comprobar que el coste va prácticamente parejo al de validación, por lo que se puede afirmar que generaliza correctamente.

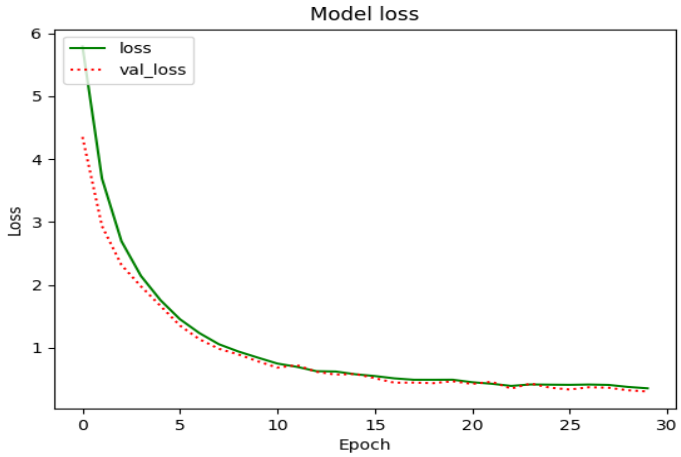


Figura 4.8: Coste con 80 rotaciones

Los resultados que devuelve el testing se recogen en la Tabla 4.5 que aparece a continuación:

Símbolo	Predicción	Probabilidad	Resultado
manookb	manookb	100 %	Acierto
flechablanca	flechablanca	98 %	Acierto
boca	boca	94 %	Acierto
doblegirohorizontal	doblegirohorizontal	68 %	Acierto
carasss	carasss	62 %	Acierto
mano5	flechadoblerectanegra	64 %	Fallo

Tabla 4.5: Testing con 80 rotaciones

Pese a tener buena precisión, generalizar correctamente y acertar con buena precisión *carasss*, se equivoca con *mano5*. Para su corrección se han planteado varias posibilidades: la primera consiste en volver a ejecutarlo, ya que los pesos se calculan de manera aleatoria y podrían arrojar una red neuronal mejor, o seguir aumentando el número de ejemplos del dataset (opción por la que finalmente se ha optado para comprobar si se mejora la precisión del testing).

Las estadísticas del modelo se adjuntan en la Tabla 4.6 siguiente:

	Training	Validation	Testing
Precisión	97 %	99 %	99 %
Coste/Fallo	0.35	0.30	1 %

Tabla 4.6: Estadísticas con 80 rotaciones

En el siguiente experimento se han subido las rotaciones a 120, y la ejecución se ha lanzado con la siguiente configuración:

- Rotaciones: 120.
- Conjunto de validación: 150 ejemplos.
- Conjunto de testing: 150 ejemplos.

Con esta modificación se han obtenido 3720 ejemplos, de los cuales se han usado 150 para validar, y otros 150 para el testing, permaneciendo entonces 3420 ejemplos para el entrenamiento. Con esta configuración se han obtenido los resultados que se muestran en la Figura 4.9 y en la Figura 4.10 que aparecen a continuación:

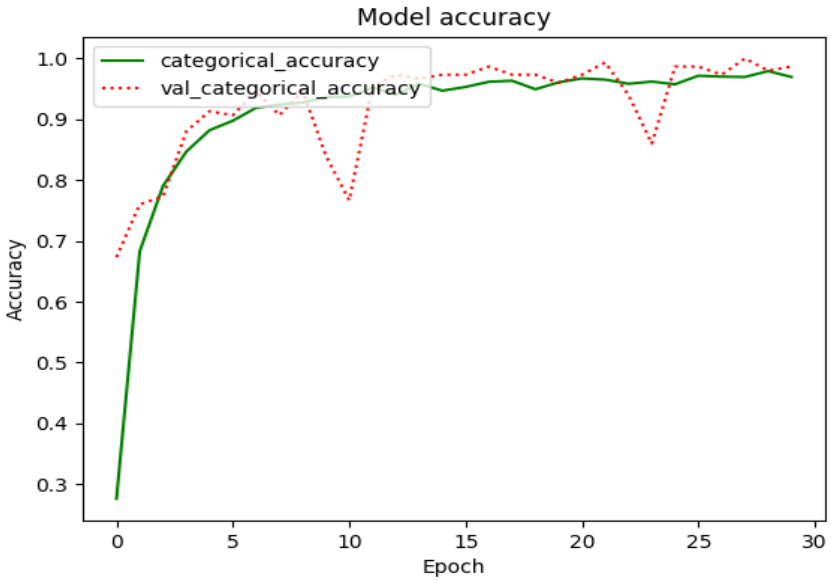


Figura 4.9: Precisión con 120 rotaciones

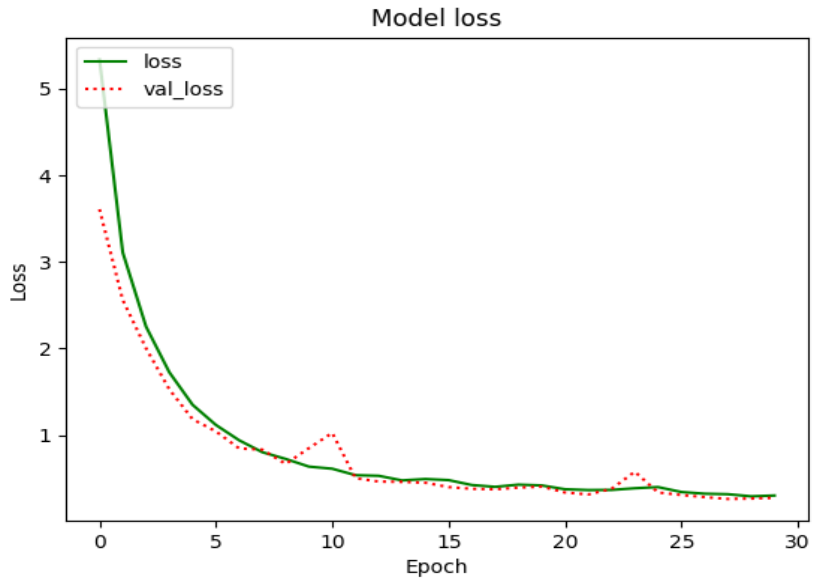


Figura 4.10: Coste con 120 rotaciones

En ambas figuras se puede observar su uniformidad, salvo en dos momentos que tienen dos picos bastante pronunciados. Esto puede deberse a que como el modelo se entrena con el conjunto de entrenamiento, ha habido

algunos ejemplos del conjunto de validación que no han sabido predecir bien. El resultado con el conjunto de testing es se recoge en la Tabla 4.7 siguiente:

Símbolo	Predicción	Probabilidad	Resultado
mano5b	mano5b	100 %	Acierto
carasss	carasss	61 %	Acierto
carasss	carasss	79 %	Acierto
manooka	manooka	95 %	Acierto
caraoh	caraoh	62 %	Acierto
carasss	carasss	61 %	Acierto
flehadoblegiro	flechadoblegiro	95 %	Acierto

Tabla 4.7: Testing con 120 rotaciones

Las estadísticas del modelo son las que se muestran a continuación en la Tabla 4.8:

	Training	Validation	Testing
Precisión	96 %	98 %	100 %
Coste/Fallo	0.30	0.27	0 %

Tabla 4.8: Estadísticas con 120 rotaciones

Tal y como puede comprobarse, estos resultados son bastante buenos, así que se ha escogido finalmente esta configuración, es decir, guardando los pesos y el modelo de la CNN actual.

En todos estos experimentos se han modificado parámetros que afectan al dataset, pero no se ha realizado ninguno variando los parámetros de la CNN, como por ejemplo, las epochs.

A continuación se muestra un experimento cuyo objetivo ha sido comprobar si las *epochs*, o épocas, pueden influir sobre el resultado final. Lo que se hace en cada epoch es pasar un dataset entero hacia atrás, y hacia delante, a través de la red neuronal, de modo que se recalculen los pesos. Para ello, se ha subido el número de epoch a 50 con la misma configuración del experimento anterior.

En la Figura 4.12 y en la Figura 4.11 se puede observar que el coste y la precisión, tanto en el entrenamiento, como en la validación, son prácticamente iguales a los de la configuración anterior de epochs.

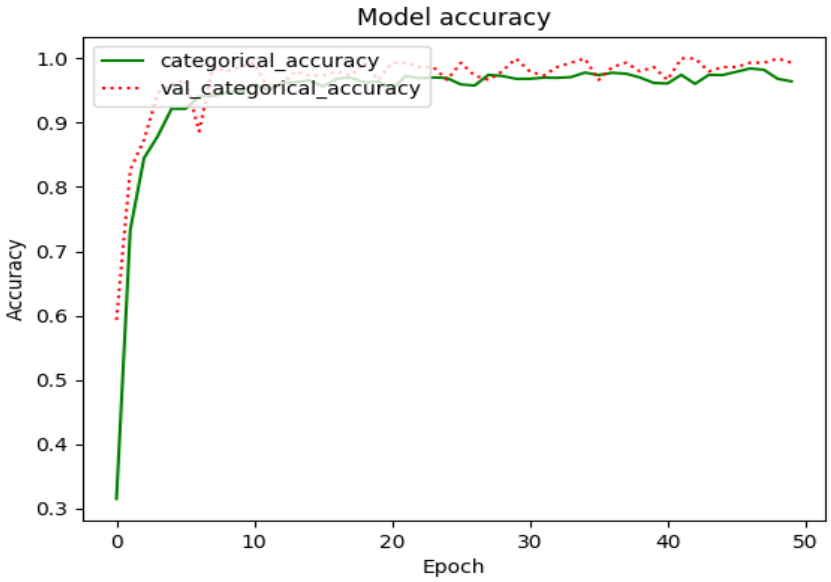


Figura 4.11: Precisión con 120 rotaciones, 50 epochs

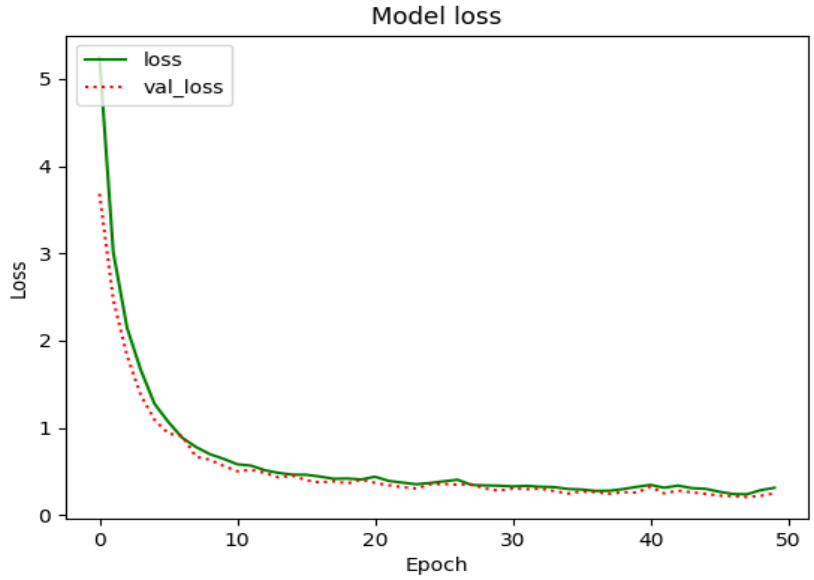


Figura 4.12: Coste con 120 rotaciones, 50 epochs

Los resultados del testing se recogen en la Tabla 4.9 siguiente:

Símbolo	Predicción	Probabilidad	Resultado
mano5b	mano5b	99 %	Acierto
carasss	carasss	57 %	Acierto
carasss	carasss	88 %	Acierto
caraoh	caraoh	98 %	Acierto
manookb	manookb	100 %	Acierto
mano5	mano5	98 %	Acierto
boca	boca	83 %	Acierto
mano5b	mano5b	99 %	Acierto
flehadoblegiro	flechadoblegiro	95 %	Acierto

Tabla 4.9: Testing con 120 rotaciones, 50 epochs

Las estadísticas de este modelo se adjuntan en la Tabla 4.10:

	Training	Validation	Testing
Precisión	96 %	99 %	100 %
Coste/Fallo	0.31	0.25	0 %

Tabla 4.10: Estadísticas con 120 rotaciones, 50 epochs

Además, tal y como se ha mostrado en las tablas anteriores los resultados son bastante buenos, pero no mejores que los de la configuración del experimento anterior. Por consiguiente, se han seleccionado los pesos y el modelo anterior con 120 rotaciones y 30 epoch.

4.3. Detección de símbolos en imágenes

Para la detección de los símbolos en imágenes se ha implementado el algoritmo YOLO anteriormente descrito, y se han llevado a acabo las siguientes tareas:

4.3.1. Preprocesamiento del dataset

Se ha hecho uso de la herramienta Yolo-Mark, explicada en el Apartado 2.5. Para ello, se ha preparado el data set para su uso, y se han definido los nombres de las clases con el nombre de los símbolos de los que se dispone en el data set, concretamente 31. Posteriormente, se han agrupado todas las imágenes, tanto símbolos, como transcripciones, en una misma carpeta, y se ha marcado en cada una de ellas la localización y la clase de cada uno de los objetos presentes.

4.3.2. Ajuste de las bounding boxes

Para hallar los anchors mencionados en el Apartado 2.4.1, se han determinado las dimensiones de los objetos a buscar en las imágenes del data set del que se dispone, así como en las dimensiones de los objetos grandes y pequeños, de modo que se pueda realizar una estimación de cuales deben ser las medidas para las bounding boxes que mejor encuentran estos objetos.

Para esta tarea se ha aplicado el algoritmo de clasificación KNN, que consiste en agrupar los datos según sus características, en este caso la dimensión de los objetos en las imágenes. Concretamente, para este problema se han agrupado en 9 grupos (o clusters), analizando los valores medios de cada cluster, o centroide, se obtienen las dimensiones representativas de los distintos tamaños de objeto que hay en nuestro data set. A partir de estos datos, se ha procedido a definir el tamaño de las bounding boxes. Debido a que previamente se han marcado los objetos del dataset, solo se deben recoger estas medidas y aplicar KNN.

En la CNN usada para el YOLO, hay tres capas de clasificación localizadas en distintas profundidades de la red. Cada una de estas capas de clasificación se encarga de reconocer distintos tamaños de objeto, ya que según se profundiza en la red, la imagen se va fragmentando en mayor medida. Concretamente, para la primera capa de clasificación se busca detectar objetos de una medida aproximada de 60x60, para la segunda capa de 30x30, y para la tercera capa de 15x15.

Para ello, se han usado como anchors las dimensiones de los objetos devueltos por los centroides calculados por el algoritmo knn, y se han asignado tres dimensiones distintas para las bounding boxes de cada capa, respetando las medidas anteriormente mencionadas. Se debe tener en cuenta que KNN es un mero clasificador, por lo que las medidas de los centroides son un sondeo y no representan todas las medidas de un cluster. Por tanto, se deben de ajustar los anchors en función de los resultados obtenidos para afinar los resultados.

En la Figura 4.13 se representa la evolución en la precisión del reconocimiento en función de los anchors. Concretamente, en la imagen de la izquierda, se observa que los anchors son demasiados grandes para los objetos que se quieren detectar, por lo que las cajas que dibuja para las figuras superiores ocupan toda la imagen, y como consecuencia, los objetos pequeños, como es el caso de la flecha y de la segunda mano inferior, no llegan a detectarse. Por otro lado, en la imagen central se puede observar un primer ajuste de los anchors disminuyendo la dimensión de las bounding boxes por capa, lo que implica resultados mejores pero no suficientemente buenos. Por último, en la imagen derecha se ve cómo la última configuración de anchors propuesta consigue resultados más adecuados, adaptándose mejor a las medidas de los objetos que se busca detectar, incluso en el caso de las flechas.

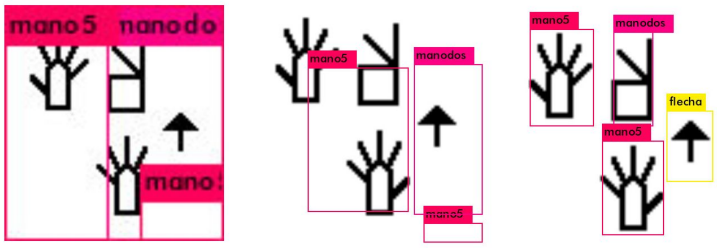


Figura 4.13: Evolución de los resultados según los anchors

4.3.3. Variación en distintos modelos de CNN para el YOLO - Evaluación mAP e IOU de los distintos modelos

Se han implementado los tres modelos de CNN descritos en el estado del arte para el YOLO, y se ha adaptado progresivamente con el objetivo de mejorar los resultados.

4.3.3.1. Modelo 1

El primer modelo se trata de la red neuronal original sin ningún cambio.

En el análisis de los resultados del modelo uno, tanto IoU como mAP se mantienen constantes en torno a 0, lo que se traduce como una solución mala para el problema.

En la Figura 4.14 se observa el resultado de la detección para este modelo. En ella se aprecia un error, tanto en localización, como en clasificación. Estos resultados se deben principalmente al movimiento de la ventana deslizante, al ajuste de los anchors y a la falta de datos.

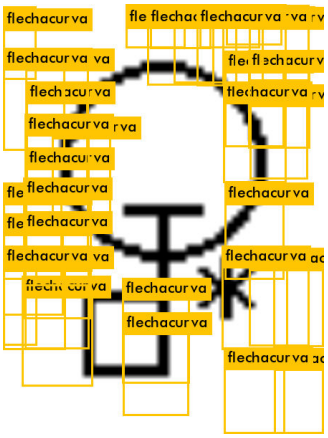


Figura 4.14: Resultados del modelos uno

4.3.4. Modelo 2

En el segundo modelo se ha adaptado la búsqueda de la red neuronal para objetos más pequeños, disminuyendo el movimiento de la ventana deslizante.

Al igual que en el modelo uno, tanto IoU como mAP, se mantinen constantes a 0, lo que sigue siendo una mala solución para el problema.

En la Figura 4.15 se aprecia una mejora respecto a la clasificación, pero no en cuanto a localización. Esto se debe principalmente a que el algoritmo no usa unos anchors adecuados para buscar los objetos dentro de la imagen.

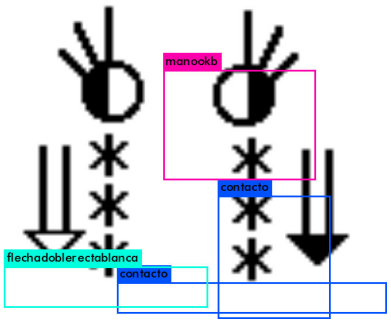


Figura 4.15: Resultados del modelo dos

4.3.4.1. Modelo final

En el tercer modelo se ha modificado el tamaño de las bounding boxes para mejorar la detección (Figura 4.16).

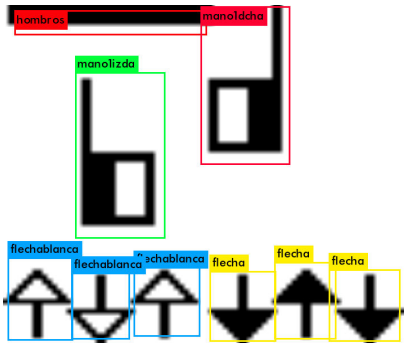


Figura 4.16: Ejemplo de ejecución del modelo final sobre la palabra Bilbao

Para el estudio de la evolución del modelo finalmente usado en la implementación, se han analizado las métricas descritas en el estado del arte y el coste (o average loss).

En la Figura 4.17 se muestra el coste del modelo con respecto a las iteraciones.

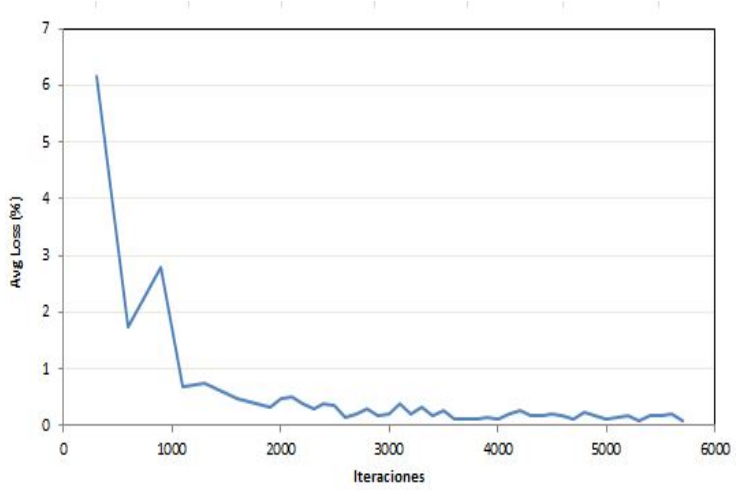


Figura 4.17: Coste respecto a iteraciones

Tal y como puede observarse, en las primeras iteraciones del modelo final el coste no es alto, siendo el valor máximo de aproximadamente el 6 %. Además, disminuye rápidamente en tan sólo 1000 iteraciones a valores menores del 1 % y se mantiene dentro de este rango, lo que supone una mejora significativa. También puede apreciarse que los valores se mantienen constantes en torno al 0,2 %, sin disminuir de este valor, lo que se debe a la falta de datos con los que entrenar y mejorar sus predicciones.

Además, en la Figura 4.18 se representa el mAP con respecto a las iteraciones.

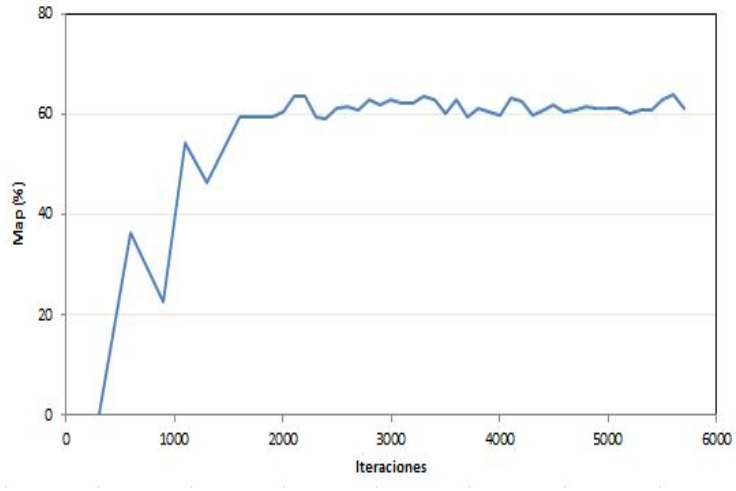


Figura 4.18: mAP respecto a iteraciones

En ella, puede observarse un crecimiento rápido en las primeras 1000

iteraciones, y un límite aproximadamente sobre la iteración 2000, a partir de donde se mantiene constante hasta el final, en torno al 60 %. Si se quisiera mejorar este resultado se requeriría de más datos para entrenar y un mayor número de iteraciones. Además, también puede apreciarse que la curva no es creciente en todo su dominio, ya que al principio aparecen decrementos sustanciales, lo que se debe a las características de la CNN en el proceso de realimentación y al ajuste de pesos, lo que puede dar lugar a mejores, o peores resultados. Por ello, parte del proceso se basa en guardar el estado de los pesos en distintos momentos, y seleccionar siempre el que proporcione la mejor solución.

En la Figura 4.19, la cual muestra el IoU con respecto a las iteraciones, puede observarse también una gran mejora en las primeras 2000 iteraciones, aunque es más lento y el crecimiento es irregular y no constante en comparación con la Figura 4.18, los valores se estabilizan en torno al 80 %, teniendo un comportamiento similar a la evolución del mAP.

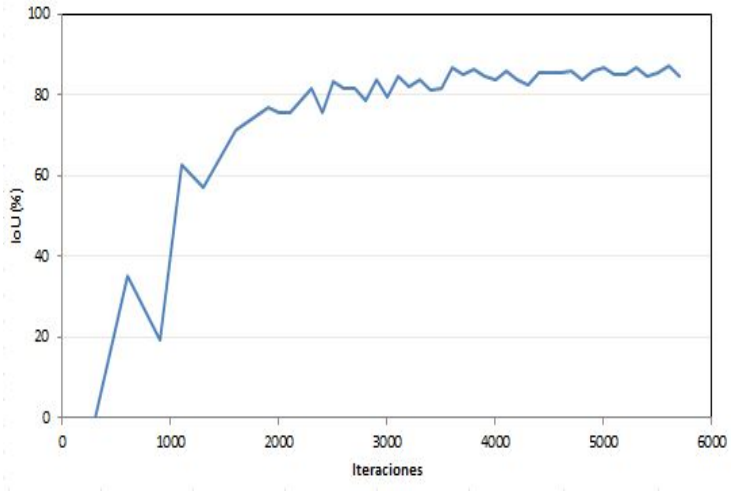


Figura 4.19: IoU respecto a iteraciones

4.3.5. Búsqueda de un threshold adecuado para los resultados

Para filtrar ruido y datos erróneos se aplica un límite a la probabilidad de las detecciones, de modo que se puedan aceptar solo las predicciones cuya probabilidad lo superen. Es importante encontrar un valor para el thresholding que nos permita eliminar todo el ruido, pero que a su vez nos devuelva el mayor número de predicciones correctas.

En la imagen izquierda de la Figura 4.20, se muestra la imagen original, en la central aparece el resultado del sistema para un threshold de 0,5 y a la derecha con threhold de 0,1.

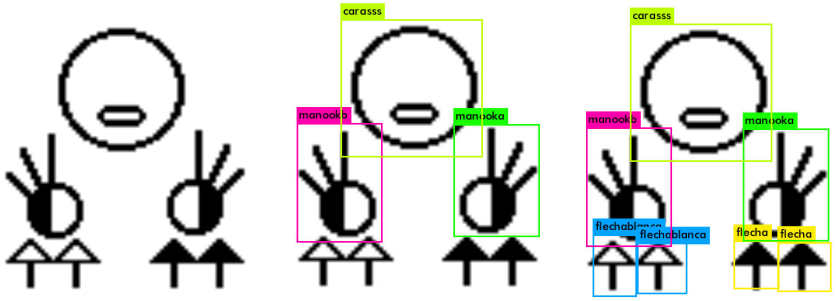


Figura 4.20: Imagen original (izquierda), Thresh 0,5 (centro) y Thresh 0,1 (derecha)

Con un threshold de 0,5 se detectan los objetos grandes, pero todas las flechas inferiores no, lo cual se cree que se debe a que hay muchas, y además están muy juntas.

Por último, con un threshold de 0,1 sí que se consiguen detectar todas las clases correctamente, y además no se generan falsos positivos.

Tras múltiples pruebas con las imágenes del dataset, se ha seleccionado 0,1 como el threshold del sistema, ya que valores menores provocan la aparición de falsos positivos, y los mayores descartan muchos buenos resultados.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

Los resultados del clasificador de símbolos son prometedores, aunque no reflejan resultados reales debido a las características del dataset, lo que se debe a que se han tratado sólo unos pocos símbolos de todos los que hay en la SignoEscritura completa, y a que las imágenes son todas digitales. Por ejemplo, no hay imágenes escritas a mano con distintas iluminaciones y/o distintos trazos. Todas estás variables nos obligaran a desarrollar una CNN mucho más compleja, la cual extraiga más características para la clasificación.

Se han conseguido unos buenos resultados para la detección de símbolos usando el algoritmo YOLO, ya que las métricas han devuelto valores del 60 % para mAP, y del 80 % para el IoU.

En la mayoría de las imágenes del dataset, los resultados de la detección son perfectos, aunque aún hay ejemplos en los que no se consigue detectar nada. La razón principal de que YOLO a veces se confunda, o no detecte nada, es debido a que los anchors definidos no son los correctos.

Por otro lado, una revisión de la construcción de la CNN también ayudaría en la mejora de los resultados. Concretamente, el elemento que más confunde nuestro YOLO son las flechas, ya se trata de objetos muy parecidos entre ellos.

Además, y como en todos los algoritmos de Machine Learning, YOLO necesita una gran cantidad de datos para funcionar correctamente. Sin embargo, dado el problema que se está tratando, y debido a las características de la SignoEscritura para generar más datos, se requiere de mucho más tiempo para la ampliación de los datos, lo que supone una gran limitación al avance del sistema.

Por último, y en base a los resultados obtenidos, aunque el camino a re-

correr para conseguir un sistema real es lento y largo, se puede afirmar que es posible conseguir un sistema capaz de reconocer un código de comunicación humano.

5.2. Trabajo Futuro

Durante la realización de este proyecto, y dadas sus características, se han detectado muchas aspectos que serían mejorables de cara a una ampliación de éste.

El principal se basa en poder proporcionar al sistema de una gran cantidad de datos para poder mejorarlo. A medida que se aumenten los datos, habrá mucha más variedad en el tipo de imágenes que se vayan a tratar, lo que dará lugar a nuevos problemas, ya sea en las dimensiones de las imágenes, o en los símbolos y transcripciones escritos a mano, etc. En consecuencia,, se deberán de hacer las oportunas adaptaciones al algoritmo YOLO para solventarlos.

Una buena implementación podría dar lugar a multitud de aplicaciones, como por ejemplo un diccionario de signo a texto, y viceversa, o herramientas didácticas para la SignoEscritura

Chapter 6

Conclusions and Future Work

6.1. Conclusions

The results of the symbol classifier are promising, but they don't show actual results because of the dataset's characteristics. This is because the dataset contains a very small amount of symbols, and they are all digital instead of handwritten. So they will always be exactly the same instead of similar. Adding all these variables would create the need of a more complex CNN.

We have achieved nice results for symbol detection using the YOLO algorithm, obtaining a 60% mAP and a 80% IoU.

In most of the dataset images symbol detection is flawless, but there are some images where it doesn't find any symbol. The reason for this is that the anchors are not defined properly.

Another way of improving the results would be reviewing the CNN. Our CNN has trouble to identify different types of arrows, since they are very similar.

Just like every other machine learning algorithm YOLO needs a lot of data to work well. But due to the nature of SignWriting, generate more data it's hard and needs a lot of time. This is a big limitation for our system.

Based on our results we can conclude that it is possible to create a system that is able to recognize SignWriting using YOLO, but it's incredibly hard and it would take a lot of time.

6.2. Future Work

During the development of this project we have detected a lot of areas that could be improved before expanding it. The priority would be generating a bigger and better dataset. The more data we can provide to training

the model, the better it would be able to predict new data. Adding more diversity with handwriting transcriptions would help generalizing the problem. As a consequence this will make a lot harder setting up the CNN and all the parameters of the YOLO algorithm.

A good implementation could lead to multiple applications, such as a dictionary between SignWritting and text and vice versa, or SignWriting teaching.

Capítulo 7

Aportaciones individuales

7.1. Aportaciones de John Byron Sánchez Jiménez

Empecé investigando sobre las características del lenguaje de signos y la SignoEscritura. Una vez definidos los objetivos del proyecto, investigué y redacté las tecnologías propuestas para resolverlo, la detección de imágenes mediante el uso del algoritmo YOLO y la clasificación de símbolos mediante el uso de CNNs.

Investigué y aprendí el uso de las librerías TensorFlow y Keras para la implementación de CNNs.

También investigué, redacté y configuré las herramientas para desarrollar el algoritmo YOLO, Yolo Mark y Darknet.

Además, realicé la investigación y redacción de las técnicas de eliminación de ruido y las métricas para los sistemas de detección de imágenes.

Implementé distintos modelos de CNN para la clasificación de símbolos, realicé la preparación del dataset para su uso en el algoritmo YOLO haciendo uso de la herramienta Yolo Mark y desarrollé los modelos dos y tres de YOLO, ejecutando su entrenamiento, recopilando, analizando y redactando los resultados obtenidos.

Además, llevé a cabo y redacté las pruebas para hallar los thresholds adecuados para el sistema y realicé la configuración de la plataforma Google Colab y de mi propio ordenador para ejecutar los entrenamientos y test de las implementaciones.

Por ultimo, edité y modifiqué toda la memoria.

7.2. Aportaciones de Samuel López Prieto

Comencé investigando y documentando el Lenguaje de Signos y de SignoEscritura, su origen, finalidad y como funcionan. A continuación, busqué información sobre Keras y Tensorflow para implementar la primera CNN, que era simplemente un clasificador de perros y gatos.

Modifiqué el dataset original para hacerlo más adecuado para un modelo de machine learning, añadiendo más ejemplos mediantes modificaciones (como rotaciones) de las imágenes utilizando librerías como numpy y openCV. Estas modificaciones al Dataset sirvieron únicamente para la CNN de símbolos de SignoEscritura, no para el detector de objetos con YOLO.

Investigación y documentación sobre técnicas de detección de objetos como R-CNN y YOLO. Incluyendo su funcionamiento, arquitectura, métricas y algoritmos para eliminar ruido y mejorar las predicciones. Documentación sobre Darknet, el framework de YOLO que utilizamos para la detección de objetos y pruebas con los pesos pre-entrenados sobre objetos comunes para compararlos con nuestro modelo. Realización de test mediante la plataforma Google Collaboratory de los pesos obtenidos por mi compañero John Byron para el modelo de detección de SignoEscritura.

Traducción al inglés de la introducción, resumen y conclusiones de esta memoria.

7.3. Aportaciones de José Ángel Garrido Montoya

Inicialmente, investigué sobre las distintas arquitecturas de CNNs que existen, viendo sus características y las capas que tiene cada una. Finalmente, desarrollé la CNN para la clasificación de imágenes usada en este Trabajo de Fin de Grado basada en la arquitectura de LeNet5. Acto seguido redacté las arquitecturas más conocidas como son AlexNet, VGG y LeNet5 para la memoria. Realicé los experimentos, entrenamiento, testing y tuning de la CNN para la clasificación de imágenes de este Trabajo de Fin de Grado y su correspondiente redacción para la memoria con gráficas y tablas sobre los experimentos. Para la experimentación, tuve que investigar sobre el uso de numpy para el manejo de imágenes y Keras como framework para desarrollar la CNN. Con esto, redacté el capítulo de Hiperparametros y definiciones relacionadas con las CNNs y de los frameworks Keras y Tensorflow. También, ayudé en la redacción de la sección sobre Google Colaboratory y configuración para poder ejecutar los notebooks con GPU. Sobre la detección de objetos, investigué sobre el algoritmo YOLO para entender cómo funciona para, posteriormente, entrenar y testear un modelo de YOLO (concretamente Yolo-tiny en el repositorio de darknet) con el dataset creado por mi compañero John Byron con la herramienta Yolo Mark.

Bibliografía

[1] Jimenez John Byron, Sánchez, Prieto Samuel, López, and Montoya José Ángel, Garrido. Reconocimiento del lse. https://github.com/NILGroup/TFG-1819-SignoEscritura.

[2] Valerie Sutton. Sutton, v. 2009.signwriting: Sign languages are written languages! the sign-writing press, la jolla, ca. http://www.signwriting.org/.

[3] Prince Grover. Evolution of object detection and localization algorithms. https://towardsdatascience.com/evolution-of-object-detection-and-localization-algorithms-e241021d8bad.

[4] Steve Parkhurst and Dianne Parkhurst. Parkhurst, stephen & dianne parkhurst. (2002). "sistema de escritura en las l. s." en apuntes de lingüística de la lengua de signos española. (pp. 273-312). madrid: Fundación cnse para la supresión de las barreras de comunicación. (trad. josé francisco fernández martínez).

[5] Ritchie Vink. Esquema red neuronal. https://www.ritchievink.com/blog/2017/07/10/programming-a-neural-network-from-scratch/.

[6] Anup Bhande. What is underfitting and overfitting in machine learning and how to deal with it. https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76, 2018.

[7] Max Pechyonkin. Key deep learning architectures: Lenet-5. https://medium.com/@pechyonkin/key-deep-learning-architectures-lenet-5-6fc3c59e6f4.

[8] Muneeb ul Hassan. Alexnet imagenet classification with deep convolutional neural networks. https://neurohive.io/en/popular-networks/alexnet-imagenet-classification-with-deep-convolutional-neural-networks/.

[9] Yugandhar Nanda. What is the VGG neural network? https://www.quora.com/What-is-the-VGG-neural-network.

[10] Sik-Ho Tsang. R-cnn (object detection). https://medium.com/coinmonks/review-r-cnn-object-detection-b476aba290d1.

[11] Manish Chablani. Yolo, you only look once, real time object detection explained. https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006.

[12] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[13] Michal Maj. What is object detection? introduction to yolo algorithm. https://appsilon.com/object-detection-yolo-algorithm/.

[14] AlexeyAB. Darknet. https://github.com/AlexeyAB/darknet/.

[15] AlexeyAB. Yolo mark. https://github.com/AlexeyAB/Yolo_mark, 2017.

[16] Adrian Rosebrock. Intersection over union (iou) for object detection. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/.

[17] Jcgonzalez. TensorFlow para principiantes (I). https://www.apsl.net/blog/2017/12/05/tensor-flow-para-principiantes-i/.

[18] Omar Sanseviero. Introduccion a tensorflow. https://medium.com/ai-learners/introducci%C3%B3n-a-tensorflow-parte-1-840c01881658.

[19] TensorFlow. Tensorflow core 1.13. https://www.tensorflow.org/api_docs/python/.

[20] Keras. Documentación keras. https://keras.io/.

[21] Convolutional Layers. Capas convolucionales. https://keras.io/layers/convolutional/.

[22] Antonio F. García Sevilla. Dataset para el aprendizaje automático de signoescritura para lse. https://github.com/agarsev/swlearn/.

[23] Jason Brownlee. Dropout regularization in deep learning models with keras. https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/.

[24] Shubham Jain. An overview of regularization techniques in deep learning (with python code). https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/, 2018.

¿Qué es la vida? Un frenesí.
¿Qué es la vida? Una ilusión,
una sombra, una ficción,
y el mayor bien es pequeño,
que toda la vida es sueño,
y los sueños, sueños son.

La vida es un sueño
Calderón de la Barca