Reconocimiento de códigos grabados por láser mediante técnicas de Deep Learning

Autor: Marcos Minaya Montalvo
Director del trabajo: Rafael Pastor Vargas
Codirector del trabajo: Juan Mario Haut Hurtado
Universidad Nacional de Educacion a Distancia (UNED)
Departamento de Sistemas y Control
Máster Universitario en Ingeniería y Ciencia de Datos

7 de marzo de 2022

Dedicatoria

“ A mis padres, mi abuela y mi novia, los que realmente han sufrido mis noches de estudio y me han apoyado en todas y cada una de ellas, hacia quienes sólo puedo expresar mi más sincero agradecimiento por apoyarme durante la etapa académica que hoy culmina. ”

Agradecimientos

Agradecer a mis compañeros Jorge y Santi de la compañía 20 Overten en la que trabajé y desarrollé este proyecto, gracias chicos por ser la mejor fuente de inspiración.

Abstract

Traditional OCR algorithms have a long history, extensive development and wide applicability to a many of text digitization problems. However, in terms of generalizability, these algorithms are limited to their own ad-hoc parameterization for the problem in which they are applied and the necessary image pre-processing for them to work correctly. At the same time, the intrinsic operation of these algorithms, based on the contrast between the character to be recognized and the image background, means that in cases where this contrast is not very pronounced, the recognition task may be impossible.

The present work seeks to develop a new algorithm based on Deep Learning techniques, which are known for their high generalization capacity for numerous scenarios thanks to their neural network architecture, and of a sufficiently large and varied dataset. The results obtained by the developed D3POCR model far outperform traditional OCR algorithms, at least in this use case. This new composite model is based on 3 phases: the detection and localization of the code to be recognized, an adjustment and calibration of its coordinates, and finally the phase that most resembles a traditional OCR, which is in charge of the recognition of the code itself. For this purpose, 3 neural network models were trained and a parameterized algorithm called sliding window was designed. In turn, 3 sets of data were collected in image format for the training of the 3 models mentioned above.

Once the development and training phase of the D3POCR model was completed, it was deployed and the testing phase began to check the real accuracy of the model. The results obtained reflected the great potential of the model in the face of the enormous variability of the data being worked with and a poorly controlled working environment.

Finally, it should be noted that this algorithm was not designed as an advanced OCR model that can be implemented in other similar use cases, because it is a completely ad-hoc development for a real use case of a logistics company. However, seeing the results obtained by this new OCR approach, this work is intended to serve as an inspiration for new OCR models working in similar situations to those presented in this paper.

Resumen

Los algoritmos tradicionales de reconocimiento óptico de caracteres (OCR) tienen una larga historia, un amplio desarrollo y una gran aplicabilidad a muchos problemas de digitalización de textos. Sin embargo, en términos de generalización, estos algoritmos están limitados a su propia parametrización *ad-hoc* para el problema en el que se aplican y al preprocesamiento de la imagen necesario para que funcionen correctamente. Al mismo tiempo, el funcionamiento intrínseco de estos algoritmos, basado en el contraste entre el carácter a reconocer y el fondo de la imagen, hace que en los casos en los que dicho contraste no es muy pronunciado, la tarea de reconocimiento pueda ser imposible.

El presente trabajo pretende desarrollar un nuevo algoritmo basado en técnicas de *Deep Learning*, conocidas por su alta capacidad de generalización para numerosos escenarios gracias a su arquitectura en forma de red neuronal, y de un conjunto de datos suficientemente amplio y variado. Los resultados obtenidos por el modelo D3POCR desarrollado superan con creces a los algoritmos tradicionales de OCR, al menos en este caso de uso. Este nuevo modelo compuesto se basa en 3 fases: la detección y localización del código a reconocer, un ajuste y calibración de sus coordenadas, y finalmente la fase que más se asemeja a un OCR tradicional, que se encarga del reconocimiento del propio código. Para ello, se entrenaron 3 modelos de redes neuronales y se diseñó un algoritmo parametrizado denominado ventana deslizante. A su vez, se recogieron 3 conjuntos de datos en formato de imagen para el entrenamiento de los 3 modelos mencionados anteriormente.

Una vez finalizada la fase de desarrollo y entrenamiento del modelo D3POCR, se procedió a su despliegue y se inició la fase de pruebas para comprobar la precisión real del modelo. Los resultados obtenidos reflejaron el gran potencial del modelo frente a la enorme variabilidad de los datos con los que se trabajaba y a un entorno poco controlado.

Por último, cabe destacar que este algoritmo no fue diseñado como un modelo de OCR avanzado que pueda ser implementado en otros casos de uso similares, ya que es un desarrollo completamente *ad-hoc* para un caso de uso real de una empresa de logística. Sin embargo, viendo los resultados obtenidos por este nuevo enfoque de OCR, se pretende que este trabajo sirva de inspiración para nuevos modelos de OCR que trabajen en situaciones similares a las presentadas en este trabajo.

Índice general

Lista de figuras **VII**

Lista de tablas **IX**

1 Introducción **1**
1.1 *Optical Character Recognition* . . . 1
1.2 Modelos de *Deep Learning* . . . 2
1.3 Clasificación y detección de objetos . . . 8

2 Objetivo del proyecto **10**

3 Estado del Arte **12**
3.1 Análisis del proyecto . . . 12
3.2 Preprocesamiento de la imagen . . . 16
3.3 Primera aproximación con OCR . . . 17
3.4 Comparativa entre OCR y *Deep Learning* . . . 18

4 Deep Three-phase OCR **21**
4.1 Introducción al modelo D3POCR . . . 21
4.2 Fase 1: Detección de código . . . 23
4.3 *Dataset*, entrenamiento y resultados para la detección de código . . . 24
4.4 Fase 2: Ajuste de detección del código . . . 27
4.5 Algoritmo de ventana deslizante . . . 27
4.6 Visualización del funcionamiento de ventana deslizante . . . 32
4.7 Resultados obtenidos por la ventana deslizante . . . 37
4.8 Fase 3: Reconocimiento de caracteres . . . 38
4.9 *Dataset*, entrenamiento y resultados para el reconocimiento de código . . . 42
4.9.1 Generación del *Dataset* . . . 42
4.9.2 Entrenamiento . . . 46
4.9.3 Resultado del entrenamiento . . . 47

5 Resultados globales del modelo D3POCR **52**
5.1 Precisión teórica global del modelo D3POCR . . . 52
5.2 Velocidad de inferencia . . . 53

5.3 Precisión real global del modelo D3POCR 54

6 Trabajo futuro y posibles mejoras del modelo 56

Índice de figuras

Figura 1.1 Ejemplo matriz OCR carácter A. 1
Figura 1.2 Estructura del perceptrón simple. 4
Figura 1.3 Ejemplo de arquitectura del perceptrón multicapa. 5
Figura 1.4 Visualización del descenso de gradiente. 6
Figura 1.5 Diferentes características extraídas por parte de capas CNN. 7
Figura 1.6 Ejemplo de arquitectura de un modelo CNN para el procesamiento de imágenes. . . . 8
Figura 1.7 Ejemplo de detección por parte del modelo YOLOv3 9

Figura 3.1 Ejemplo código SW04X103 grabado. 13
Figura 3.2 Ejemplos de códigos 13
Figura 3.3 Configuración entorno de trabajo. 14
Figura 3.4 Ejemplo imagen lona y código OM40X020 grabado. 15
Figura 3.5 Preprocesamiento de código fallidos 18
Figura 3.6 Código JB20X124: Preprocesamiento código 19

Figura 4.1 Flujo del algoritmo D3POCR 22
Figura 4.2 Ejemplo de intersección sobre unión en *bounding boxes* inferidas 24
Figura 4.3 *Data augmentation* para YOLOv3: rotaciones aplicadas 25
Figura 4.4 Ejemplos detecciones YOLOv3 26
Figura 4.5 Confianzas obtenidas en la primera parte del algoritmo de ventana deslizante. 29
Figura 4.6 Ejemplos de recorte del carácter X con confianzas cercanas a 1.0 30
Figura 4.7 Confianzas procesadas. 31
Figura 4.8 Punto de partida de la ventana deslizante. 32
Figura 4.9 Recorte de partida de la ventana deslizante. 33
Figura 4.10 Ejemplos de recortes y respectivas confianzas. 34
Figura 4.11 Ejemplos de recortes de X y respectivas confianzas. 35
Figura 4.12 Filtrado de confianzas y obtención del recorte más centrado para X. 36
Figura 4.13 Recorte código(izquierda) vs recorte código ventana deslizante(derecha) 37
Figura 4.14 Recorte caracteres *raw* (arriba) vs recorte caracteres ventana deslizante (abajo) 37
Figura 4.15 Estructura modelo dígitos. 40
Figura 4.16 Estructura modelo letras. 41
Figura 4.17 Diagrama del desarrollo iterativo de modelos. 43
Figura 4.18 Ejemplos de imágenes de ambos *datasets* (letras y cifras) 45
Figura 4.19 Loss y accuracy del modelo de reconocimiento global 48

Figura 4.20 Loss y accuracy del modelo de reconocimiento de letras 48
Figura 4.21 Loss y accuracy del modelo de reconocimiento de dígitos 48

Índice de tablas

4.1 Instancias *dataset* YOLO 25
4.2 Instancias dataset reconocimiento caracteres 44
4.3 Configuración del entrenamiento de los modelos de reconocimiento 46
4.4 Precisión modelo dígitos 49
4.5 Precisión modelo letras 50
4.6 Precisión modelo global 51

5.1 Tabla nomenclaturas de precisiones de modelos 52
5.2 Tiempos de inferencia de cada fase 53
5.3 Resumen del comportamiento del modelo con proyectos reales 55
5.4 Acierto global del modelo con proyectos reales 55

Capítulo 1

Introducción

1.1. *Optical Character Recognition*

OCR, por sus siglas, *Optical Character Recognition* [1], es una rama de la visión por computador clásica, encargada del reconocimiento de caracteres a partir de una imagen, para transformar el texto que aparece en la misma a un formato de texto plano con el cual se pueda trabajar.

Un OCR basa su funcionamiento en la coincidencia de patrones entre los caracteres que se intentan extraer de una imagen, y los patrones que alberga el propio modelo de OCR. Básicamente, estos patrones propios del modelo son una serie de matrices de bits en los que por ejemplo, los 1's podrían ser partes del propio carácter, y los 0's los espacios en blanco. En la Figura 1.1 se muestra un posible ejemplo de este tipo de matrices en un modelo de OCR.

Aplicando estas máscaras sobre la imagen, y ciertas transformaciones para poder reconocer diferentes tipografías, los modelos de OCR son capaces de extraer y digitalizar textos que aparezcan en una imagen con una precisión excelente.

Sin embargo la resolución de esta tarea no es algo trivial. Debido a la gran cantidad de caracteres y

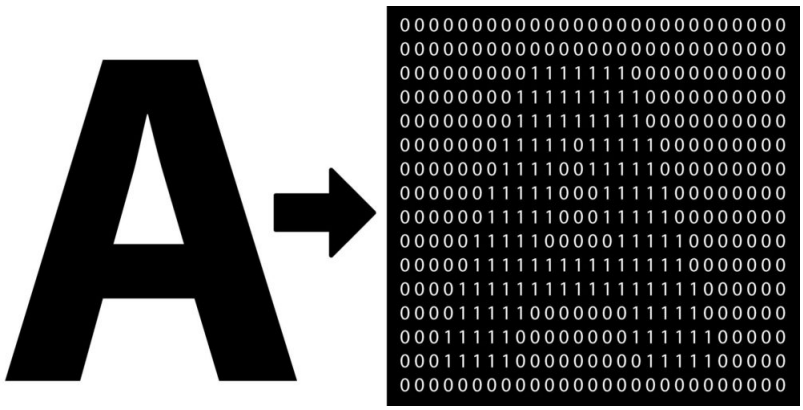

Figura 1.1: Ejemplo matriz OCR carácter A.

tipografías posibles, el abanico de patrones a analizar es muy amplio, y la calidad y nitidez de los textos que se quieren extraer deben ser las adecuadas. Otro problema que se plantea es que estos modelos necesitan que haya una diferencia clara de color entre el texto y el fondo en el que este se encuentra. Como se ha podido apreciar en el apartado 1.2, las condiciones lumínicas con las que se trabajan provocan que las imágenes que se toman de las lonas tengan fondos con tonalidades en la escala de grises muy variables, y de igual manera puede ocurrir con los propios códigos grabados. Para lidiar con esta cantidad de variabilidad en las imágenes será necesario realizar un preprocesamiento muy preciso de las mismas para facilitar la tarea del OCR.

1.2. Modelos de *Deep Learning*

Los modelos de *Deep Learning* [2] son una familia de modelos matemáticos inspirados en la propia estructura de las neuronas dentro del cerebro, generando de esta manera arquitecturas basadas en arquitecturas neuronales. De ahí otro de sus muchos nombres, redes neuronales. Estos modelos, al igual que los modelos de OCR, también requieren de una parametrización, pero no para el proceso de inferencia, si no para lo que se denomina proceso de entrenamiento, y de ahí la gran diferencia con modelos tradicionales de *Computer Vision*. Con la suficiente cantidad de datos, estos modelos son capaces de auto-ajustarse al problema en cuestión al que se quieran aplicar, sin necesidad de realizar un análisis previo ni un pre-procesamiento de los datos de entrada tan exhaustivos para extraer los parámetros correctos con el fin de ajustar el modelo.

Estos modelos de aprendizaje siempre requieren de un amplio conjunto de datos para poder alcanzar su objetivo, aunque este *dataset* tenga diferentes maneras de ser aplicado. Se pueden encontrar más subcategorías, sin embargo las cuatro principales en las que se pueden englobar estos modelos son las siguientes:

- **Aprendizaje supervisado**: En el aprendizaje supervisado [3] es aquel tipo de aprendizaje en el que además de contar con un conjunto de datos para el entrenamiento, estos datos se encuentran etiquetados. Básicamente, esto quiere decir que se cuenta de antemano con los resultados que se esperaría del modelo para dicho *dataset*, es decir, con las etiquetas de los datos de entrenamiento.
- **Aprendizaje no supervisado**: En el aprendizaje no supervisado [4], a diferencia del supervisado, no se cuenta con un conjunto de datos etiquetado, y por lo tanto no está definida cual debería ser la salida del modelo para dicho *dataset*. Sin embargo, se espera que el modelo infiera de dicho entrenamiento estas etiquetas.
- **Aprendizaje semi-supervisado**: El aprendizaje semi-supervisado [5] es una combinación entre el aprendizaje supervisado y el no supervisado. Normalmente se aplica a *datasets* en los que se cuenta con ciertos datos etiquetados, pero la gran mayoría están sin etiquetar. En esencia, se pretende entrenar el modelo con los datos etiquetados de los que se dispone, inferir las etiquetas sobre los datos no etiquetados, y volver a realizar un entrenamiento con el conjunto de datos global. A su vez este proceso se puede hacer de manera iterativa para mejorar las predicciones de etiquetas del modelo.

- **Aprendizaje por refuerzo**: En el aprendizaje por refuerzo [6] tampoco se cuenta con la salida esperada por el modelo para cada entrada, pero si que son conocidas las reglas que dicen como de buenas o malas son las decisiones que tome el modelo en cada situación, y por lo tanto se le puede premiar ó castigar, generando de esta manera un *feedback*, y utilizándolo para ajustar dicho modelo.

Estos modelos se pueden aplicar prácticamente a cualquier tipo de problema. En el caso específico de este proyecto se han aplicado a problemas de clasificación y regresión, los cuales se presentan a continuación:

- **Problemas de clasificación**: En los problemas de clasificación [7], la etiqueta de los datos es lo que se conoce como una clase. Es decir, los datos de entrada están clasificados entre un conjunto finito de posibles clases, y los modelos tienen que tratar de inferir dicha clasificación.
- **Problemas de regresión**: En los problemas de regresión [8], las posibles etiquetas de las entradas no son un conjunto de clases finitas, si no que la salida que se espera del modelo en cuestión es una salida continua.

En cuanto al formato de datos de entrada, también se puede encontrar un gran abanico. En el caso concreto de este proyecto el formato de los datos de entrada serán imágenes.

Hay una innumerable cantidad de posibles arquitecturas internas de estos algoritmos. En este apartado se van a presentar y explicar aquellas que se han utilizado para el desarrollo de esta solución.

En primer lugar es necesario como se organizan internamente estos modelos. Principalmente se cuenta con tres conceptos básicos en lo que a la arquitectura de las redes neuronales se refiere: neuronas, pesos y funciones de activación. Las neuronas, también denominadas perceptron [9], son la unidad básica de cálculo dentro de estos modelos. Estas tienen una serie de entradas y una serie de salidas, y su objetivos es el de procesar dichas entradas para generas las salidas oportunas. Por otro lado, los pesos se encargan, como su propio nombre indica, de ponderar las entradas y salidas de estas neuronas, con la finalidad de hacer que estas tengan una mayor o menor importancia dentro de la arquitectura del modelo. Finalmente, las funciones de activación son las encargadas de modelar la salida de los perceptrones. En la Figura 1.4 se puede apreciar la arquitectura de un perceptrón simple, creado por Frank Rosenblatt en 1958:

En dicha Figura 1.4 se pueden apreciar las 3 partes principales de un perceptrón simple. En cuanto al cálculo que se realiza en dicha estructura, básicamente los pesos se multiplican por el valor de las entradas a la neurona, para posteriormente calcular el sumatorio de estos valores. Finalmente se suma al resultado de este sumatorio un valor conocido como *bias*.

$$z = b + \sum_i x_i w_i \tag{1.1}$$

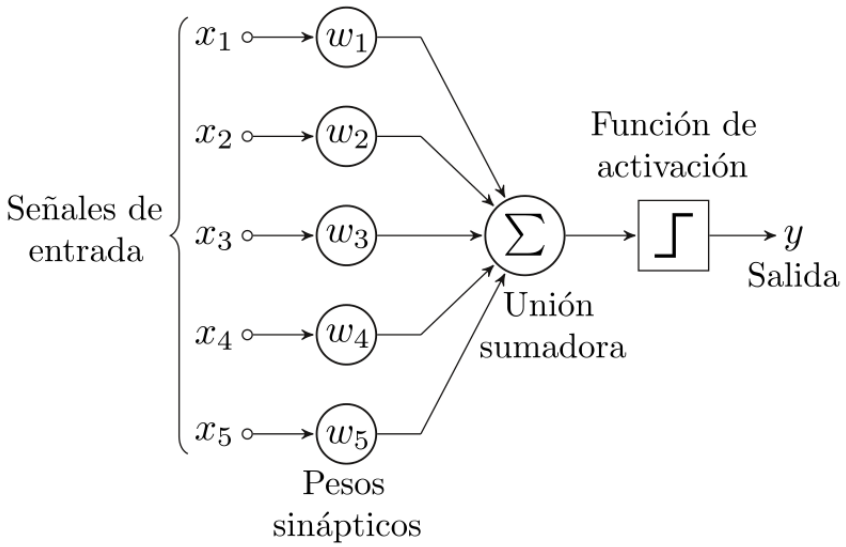

Figura 1.2: Estructura del perceptrón simple.

Contando con el valor z como la salida del perceptrón simple, se le aplica la función de activación para modelar la salida. A continuación se muestran las más utilizadas a día de hoy en el campo del *Deep Learning*:

- **Sigmoide**: La función de activación sigmoide [10] es una función ampliamente extendida en los modelos neuronales, la cual se utiliza tanto en capas intermedias del modelo como en capas de salida. Las salidas de esta función están comprendidas entre los valores 0 y 1.

$$y = \frac{1}{1 + e^{-z}} \tag{1.2}$$

- **Tangente hiperbólica**: Al igual que la función sigmoide, la función tangente hiperbólica [11] o *tanh* tiene una amplia aplicabilidad. En esta función, las salidas están comprendidas entre -1 y 1.

$$y = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{1.3}$$

- **ReLU**: Por sus siglas *rectified linear unit* [12], esta función cobró fuerza por encima de las dos anteriores, sobre todo en las capas intermedias del modelo. su particularidad es que los valores están comprendidos entre 0 e infinito, convirtiendo a 0 cualquier valor negativo que reciba.

$$y = max(0, z) \tag{1.4}$$

- **Leaky ReLU**: Esta función *Leaky ReLU* [13] es igual que la función ReLU, con la diferencia que no transforma a 0 todos los valores negativos que recibe, si no que simplemente los reduce sin llegar a anularlos, por medio de la multiplicación por una constante α.

$$y = max(\alpha z, z) \tag{1.5}$$

- **Softmax**: La función softmax [14] es una función que suele ser aplicada en las capas de salida del modelo. Esta función modela las entradas que recibe hacia una distribución de probabilidad, en la que la suma de sus valores es 1.

$$y_i = \frac{e^{z_i}}{\sum_{j=i}^{K} e^{j_i}} \, para \, i = 1, ..., K \; y \; z = (z_1, ..., z_K) \in \mathbb{R}^{\mathrm{K}} \tag{1.6}$$

Todos estos componentes conformar el anteriormente explicado perceptrón simple. Esta neuronal de manera individual solo es capaz de ajustarse a problemas que son linealmente separables. La verdadera clave de los modelos de *Deep Learning* es el uso de un gran número de estos perceptrones organizados en capas consecutivas, llamando a este tipo de arquitectura perceptrón multicapa [15]. Con la combinación de un número más o menos elevado de estas neuronas (Figura 1.3), se puede conformar lo que se llama un modelo neuronal profundo, o *deep neural network* [16], el cual es capaz de ajustarse a problemas de separación no lineal increiblemente complejos.

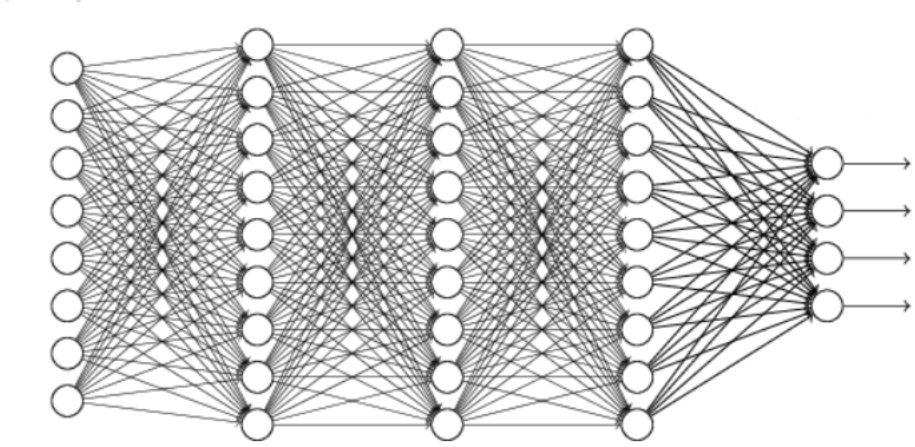

Figura 1.3: Ejemplo de arquitectura del perceptrón multicapa.

Una vez conocida la estructura básica de un perceptrón multicapa, se puede introducir el concepto de entrenamiento o aprendizaje. Como bien se comentaba en el apartado 1.3, los algoritmos de OCR no son algoritmos que se auto-ajusten, si no que necesitan de una intervención humana en el ajuste de los parámetros. La enorme ventaja de los algoritmos de *Deep Learning* es que no es necesario tener un conocimiento

del como resolver un problema, si no que solo se necesita el qué que quiere realizar, y el propio modelo aprenderá el como.

Al inicio del proceso de entrenamiento, los pesos w_{ij} del modelo suelen estar inicializados de manera aleatoria. Al modelo se introducen como entrada los datos de entrenamiento, se realizan todas las operaciones internas anteriormente mencionadas, y se obtiene la salida. Dicha salida que calcula el modelo se compara con la salida etiquetada real, y se calcula el error que el modelo ha cometido mediante la denominada función *loss* [17] o función de coste. Una vez se obtiene el error, comienza la fase de ajuste de pesos del modelo, denominada *backpropagation* [18]. Dado este error cometido, se calculan las derivadas de los pesos de las capas del modelo con el fin de calcular la pendiente de la función que minimiza dicho error, y modificar ligeramente dichos pesos hacia la dirección negativa de la pendiente para minimizar el error cometido. Este proceso se le asigna al optimizador del modelo. Hay numerosas variantes de optimizadores, pero el más conocido es el *gradient descent* [19] o descenso de gradiente.

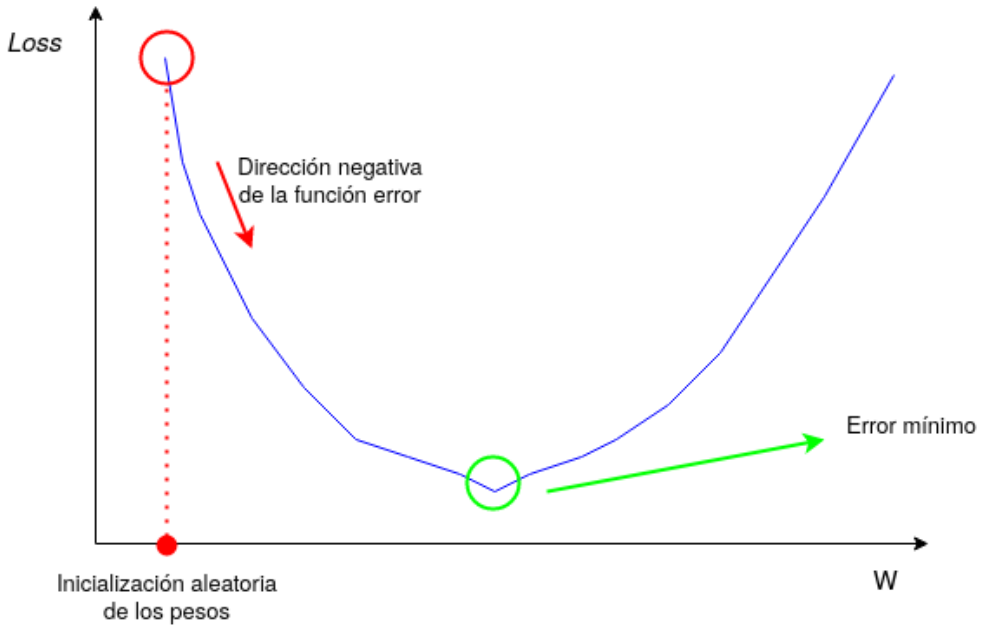

Figura 1.4: Visualización del descenso de gradiente.

Contando con un elevado número de imágenes con la suficiente variabilidad, y realizando un elevado número de iteraciones del proceso anteriormente explicado, se puede conseguir que los pesos del modelo se ajusten de tal manera que sean capaces de generalizar, y aplicarse a datos de entrada que no haya visto durante el proceso de entrenamiento, pero aun así siga siendo capaz de inferir las salidas esperadas. Gracias a la gran cantidad de datos con los que se cuenta a día de hoy, y la enorme capacidad de cómputo, se pueden obtener resultados verdaderamente asombrosos por parte de estos modelos. Como bien se comentaba, en los algoritmos tradicionales de OCR, el humano necesita ajustar los parámetros del modelo de manera manual, los cuales no serán muchos, tantos como el algoritmo tenga definidos. Sin embargo, el gran potencial de los modelos de *Deep Learning* es que pueden contar con cientos de miles o incluso millones de pesos, los

cuales se ajustan de manera completamente automática. Se puede entender por tanto la potencia de predicción diferencial que albergan estos modelos con respecto a modelos en los que los parámetros se ajustan de manera manual.

Se comentaba anteriormente que hay una enorme variedad de arquitecturas posibles en los modelos de *Deep Learning*. La que se presenta en la figura 1.3 es la denominada como *fully-connected* [20], debido a que todas las salidas de cada capa están conectadas con todas las neuronas de la capa siguiente, y las capas formadas por este tipo de conexiones se llaman capas densas. Esta es una implementación no demasiado eficiente y que no es capaz de ajustarse de manera correcta a todos los problemas. Por ello se desarrollaron numerosos tipos de capas internas de modelos para ajustarse a cada tipo de problema que se pueda presentar.

En el caso específico del procesamiento de imágenes, las capas *CNN* o capas convolucionales [21] están ampliamente extendidas. Cuando se analiza una imagen de manera visual, no se procesa toda la imagen globalmente, si no que se hace de manera local, lo cual es mucho más eficiente y aporta una información mucho más fiel y relevante. Las capas convolucionales aplican este mismo fundamento. En este caso no se conectan todas las salidas de una capa a toda las neuronas de la capa siguiente, si no que cada neurona está conectada a una región concreta de las salidas de la capa anterior. Con esta arquitectura, cada capa interna del modelo extrae un conjunto de características relevantes en forma de matriz de la imagen de entrada, se reduce su dimensionalidad (*pooling*) por motivos computacionales y de eficacia, y esas características sirven de entrada a la siguiente capa, la cual extraerá otro conjunto de nuevas características. Conforme se avanza en las diferentes capas del modelo, las características extraídas son más numerosas y más precisas, como se puede apreciar en la Figura 1.5 en la que se muestran características extraídas de imágenes de caras.

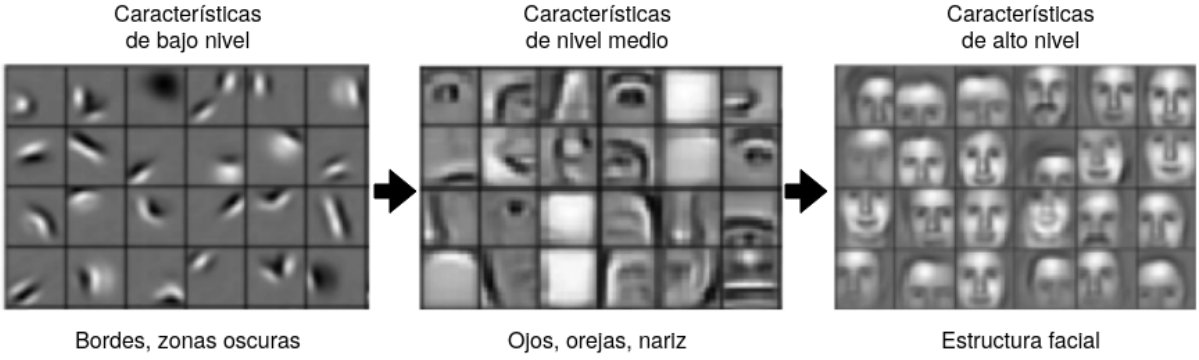

Figura 1.5: Diferentes características extraídas por parte de capas CNN.

Sin embargo, las capas densas en este tipo de problemas no se eliminan por completo de los modelos. Dichas capas se suelen utilizar como capas finales para extraer las últimas característica y clasificarlas antes de obtener el resultado final del modelo. Las características finales de las capas CNN que se obtienen se pueden reorganizar en una lista de valores que se introducen a las últimas capas densas del modelo. La arquitectura global de un modelo tradicional para el tratamiento de imágenes se presenta en la Figura 1.6 de ejemplo.

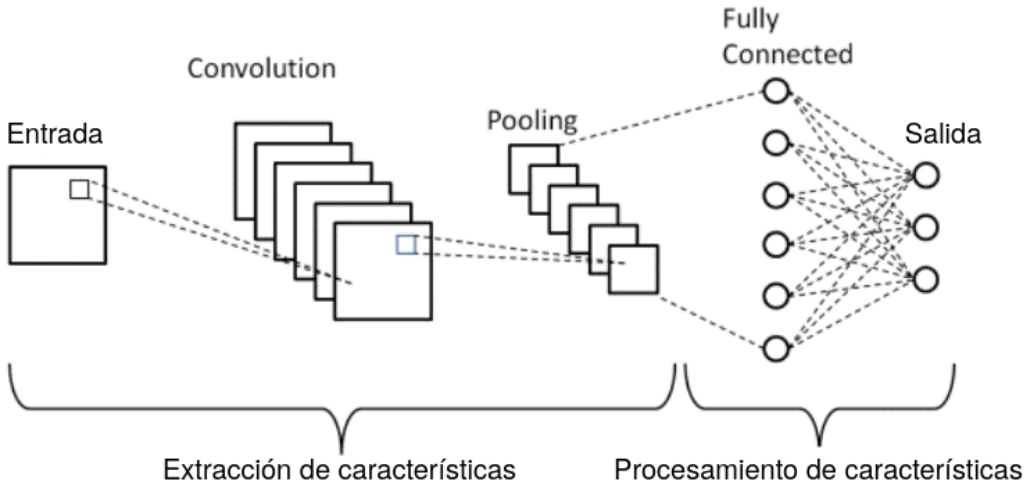

Figura 1.6: Ejemplo de arquitectura de un modelo CNN para el procesamiento de imágenes.

1.3. Clasificación y detección de objetos

Se hablaba en el apartado anterior de los problemas de aprendizaje supervisado, los cuales son la base de este proyecto, y también de los problemas de clasificación y regresión. Los modelos que conforman la tercera fase del algoritmo D3POCR, como se verán en el Capítulo 4, son algoritmos de reconocimiento, y se engloban en la familia de problemas de clasificación. Sin embargo, la primera fase de este modelo basa su comportamiento en un modelo de detección de objetos [22], el cual es un modelo compuesto que resuelve tanto tareas de clasificación como de regresión.

Estos modelos no solo clasifican los objetos que aparecen en una imagen, si no que a su vez los localizan en la misma. La localización de un objeto se puede hacer de multitud de maneras. En este caso en concreto, el algoritmo utilizado denominado YOLOv3 [23], para cada objeto devuelve la clase y la confianza del mismo, el centroide de un rectángulo y su altura y anchura. Este rectángulo, denominado comúnmente como *bounding box*, enmarcará al objeto que se desea detectar. Estos modelos tienen un gran potencial, ya que son capaces de detectar no uno, si no multitud de objetos que aparezcan en la imagen al mismo tiempo. En la siguiente Figura 1.7 se muestra un ejemplo clásico de detección por parte de este modelo.

La tarea de asignar una clase a cada uno de los objetos se considera una tarea de clasificación, pero por otro lado, la inferencia de las coordenadas de los *bounding box* predichos por el modelo es una tarea de regresión. Si bien el número posible de coordenadas es finito, y se limita a la resolución en píxeles de una imagen, los posibles valores que puede adquirir son demasiados como para intentar realizar una clasificación, lo cual no tendría sentido. Por ejemplo, en una imagen de una resolución de 512×512, los posibles valores de coordenadas son 262144, por lo tanto se modela como un problema de regresión.

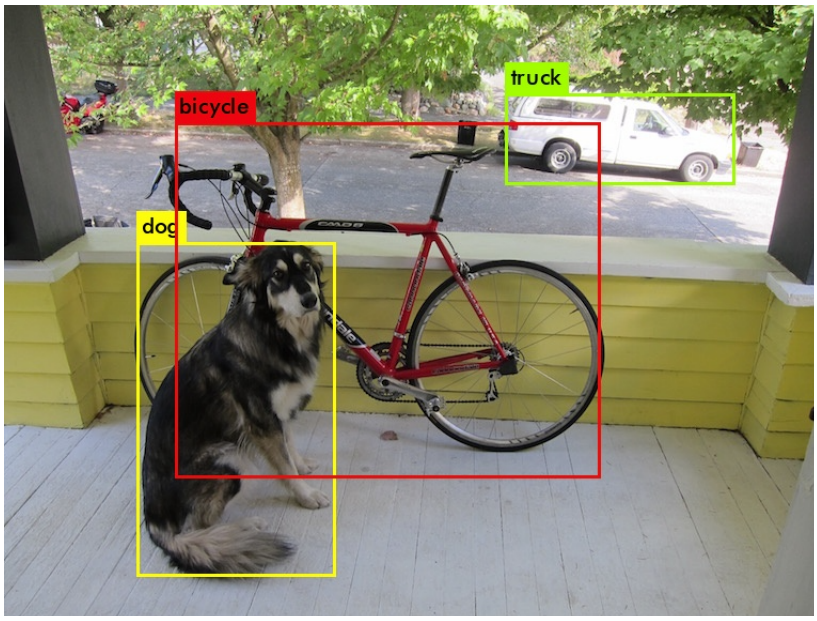

Figura 1.7: Ejemplo de detección por parte del modelo YOLOv3

Capítulo 2

Objetivo del proyecto

El objetivo de este Trabajo de Fin de Máster es la implementación de una solución para el reconocimiento de una serie de códigos grabados a láser en una chapas de aluminio con forma triangular para una empresa logística.

Para alcanzar este objetivo principal, es necesario desarrollar las siguientes tareas:

- **Gestión y análisis de datos**: Gestionar y procesar grandes cantidades de datos en formato de imagen,y su posterior análisis es crucial para el entendimiento de la problemática que se presenta. A su vez, el conocimiento de la zona de trabajo donde este proyecto se implementa es de gran importancia a la hora de elegir las técnicas adecuadas.
- **Estudio de hardware óptico**: Investigación acerca del tipo de hardware óptico más apropiado para este proyecto. Análisis del efecto de la luz al tipo de materiales con los que se trabaja. Estudio de distintos tipos de luz, filtros ópticos a instalar y cámaras industriales más adecuadas basadas en resolución de las imágenes, velocidad de disparo y resistencia a ambientes de trabajo.
- **Estudio de técnicas de OCR**: Estudio de diferentes tipos de técnicas de *Computer Vision* con aplicabilidad en este proyecto.Análisis de diferentes *softwares* orientados hacia el ámbito de *Computer Vision*. Aprendizaje y mejora en los conocimientos que ya se tenían en librerías como *OpenCV* ó *Pillow*, y a su vez en *frameworks* dedicados a OCR, como por ejemplo *Tesseract*.
- **Estudio de técnicas de *Deep Learning***: Utilización de modelos de *Deep Learning* como alternativa a los modelos de OCR tradicionales. Utilización de *Python* como principal lenguaje de programación, y *frameworks* y librerías tales como *Tensorflow*, *Keras* y *Pytorch*. Utilización de herramientas otorgadas por NVIDIA, en concreto *software* dedicado al despliegue y entrenamiento de modelos de *Deep Learning*, como CUDA y Cudnn, necesarios para la comunicación entre los modelos desarrollados y el *hardware*(GPU's) donde estos se implementan. Adquisición de conocimientos relacionados con los modelos de las propias tarjetas gráficas, y la compatibilidad de las mismas con las diferentes versiones de las librerías de *Python* anteriormente mencionadas.

- **Estudio y adquisición de conocimiento en computación *cloud***: Aprendizaje y adquisición de conocimientos en el uso de instancias *cloud* con GPU's dedicadas para el entrenamiento de los modelos, principalmente en la plataforma AWS. Debido a la necesidad de capacidad computacional de ciertos modelos utilizados en el desarrollo de este proyecto.
- **Optimización de modelos y procesos**: Optimización del *software* desarrollado, tanto referente a niveles de precisión por parte de los modelos tanto como por los tiempos de ejecución de los mismos. Análisis estadístico de los datos de comportamiento de los modelos para la mejora iterativa de los resultados. Implementación de técnicas de programación optimizadas hacia la mejora del rendimiento del *software* desarrollado.

Capítulo 3

Estado del Arte

En este apartado se va a mostrar un análisis y características del reconocimiento de códigos grabados a láser en chapas metálicas, así como la implementación y resultados obtenidos por algoritmos de OCR tradicionales.

La empresa logística para la que se realizó este proyecto realiza envíos de este tipo de materiales, y necesitaba de una solución para el control de dichos envíos, y asegurarse que, en cada pedido, las chapas que se envían son las correctas, y que no sobra ni falta ninguna de ellas, ya que si esto ocurriera, el coste de volver a enviar una chapa nueva por vía aérea consumiría gran parte de los beneficios del proyecto. Por ello, se decidió grabar en estas chapas códigos que las identificasen, y llevar un control de manera manual de estos envíos. Tras un tiempo se pensó en la posibilidad de automatizar dicho proceso, dando como resultado este proyecto.

3.1. Análisis del proyecto

El diseño de la solución ha de tener en cuenta el formato del código a reconocer y una serie de características del entorno de trabajo y de los materiales con los que están conformadas estas chapas, debido a que estos tienen una enorme influencia en la precisión del modelo a implementar.

En primer lugar es necesario saber que estos código tienen todos un mismo formato. Todos comienzan con dos letras indicando el país o la ciudad de destino a donde se enviarán estas chapas. Posteriormente irán seguidos de dos números lo cuales reflejan un identificador del proyecto de la empresa asociado a dicho envío. Después de esto aparecerá siempre el carácter X, el cual no aporta ninguna información relevante para la empresa, si no que es un carácter que sirve como carácter de calibración de código (apartado 4.4). Finalmente se podrán encontrar tres números, los cuales codifican la geometría del triangulo de la chapa en cuestión. A su vez, los caracteres tienen siempre las mismas dimensiones y están espaciados la misma distancia, por lo tanto el código siempre tendrá el mismo tamaño. En la Figura 3.1 se puede apreciar este formato de código.

En cuanto al entorno de trabajo, este no es un entorno aislado ni controlado, y sobre estas chapas de

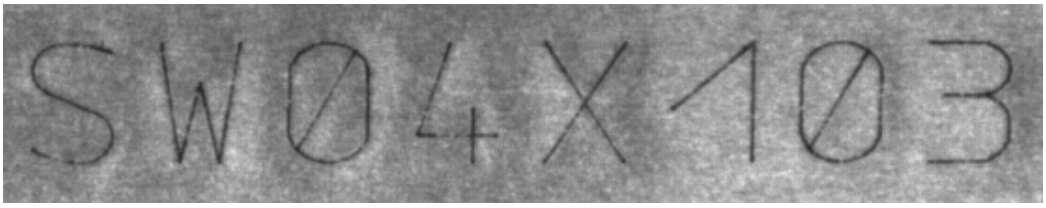

Figura 3.1: Ejemplo código SW04X103 grabado.

aluminio influye la luz ambiente y los cambios de esta, además de que el aluminio tiene una reflexión de luz con la que es muy complicado trabajar. Para intentar minimizar el impacto de la posible luz ambiente, se instaló en la cámara un filtro infrarrojo, el cual suavizaba las reflexiones de las luz sobre estas chapas, aunque estas siguen siendo muy notables. Sin embargo, no se puede eliminar la luz de forma completa debido a que es necesaria una luz mínima para que la cámara sea capaz de captar una imagen en la que el código sea visible. A su vez, en esta empresa trabajan en turnos nocturnos, y por lo tanto es necesario encender los focos internos de la nave industrial, los cuales también generan una serie de reflexiones en las lonas que dificulta también el reconocimiento de los códigos.

En la Figura 3.2 se muestran algunos ejemplos de la variabilidad de luces y contrastes que se pueden dar como fruto de un entorno de trabajo poco controlado en cuanto a luz se refiere. En esta Figura 3.2 se puede apreciar que a parte de un gran número de variaciones en la luz, contraste y nitidez de códigos grabaos, también se puede dar el caso de que las lonas de aluminio estén dañadas. En ocasiones pueden aparecer manchas sobre los códigos e incluso defectos, como por ejemplo arañazos en el propio material.

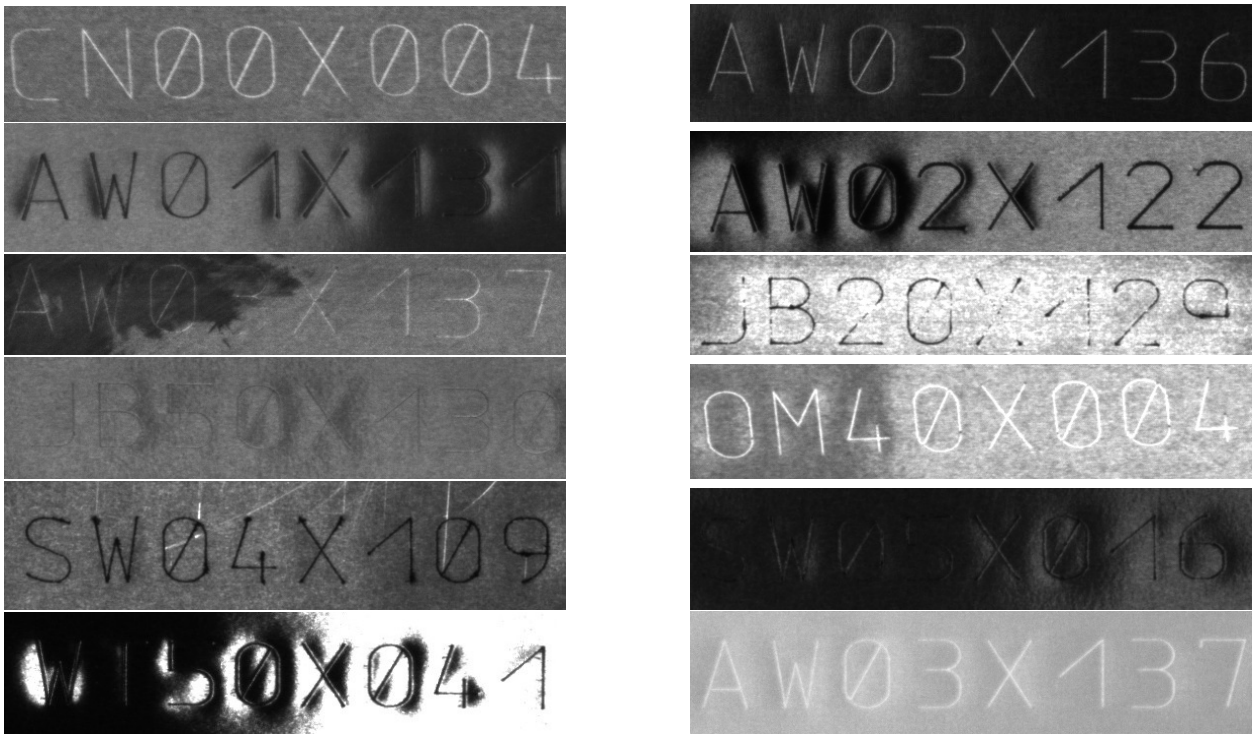

Figura 3.2: Ejemplos de códigos

Otra problemática del proyecto es la distancia a la que están ubicadas las cámaras encargadas de tomar las fotografías. Debido a que la zona de control de estas chapas es una zona de trabajo, era necesario ubicar las cámaras a una altura prudencial para que estas no sufrieran ningún tipo de daño. Por lo tanto las cámaras se colocaron a una altura aproximada de 5.5 metros sobre la zona de carga de chapas. A esto es necesario añadir que el tamaño del código grabado a láser es de 17.5mm de alto x 130mm de ancho. En la Figura 3.3 se muestra la posición de la cámara en el entorno de trabajo. Para tener la mejor calidad posible de imagen, las cámaras instaladas son de 20mpx, lo que genera imágenes de 5472x3648 píxeles, y el área total de la chapa que la cámara captura es un rectángulo de aproximadamente 120x90cm. En la Figura 3.4 se muestra una imagen real de lo que captaría esta cámara y con la cual habría que trabajar. El código se encuentra recuadrado. Se puede observar la complejidad de localizar el código incluso a simple vista. Cabe destacar que no todos los códigos son tan complejos de localizar, pero este caso se da en un gran número de lonas con las que se trabajar.

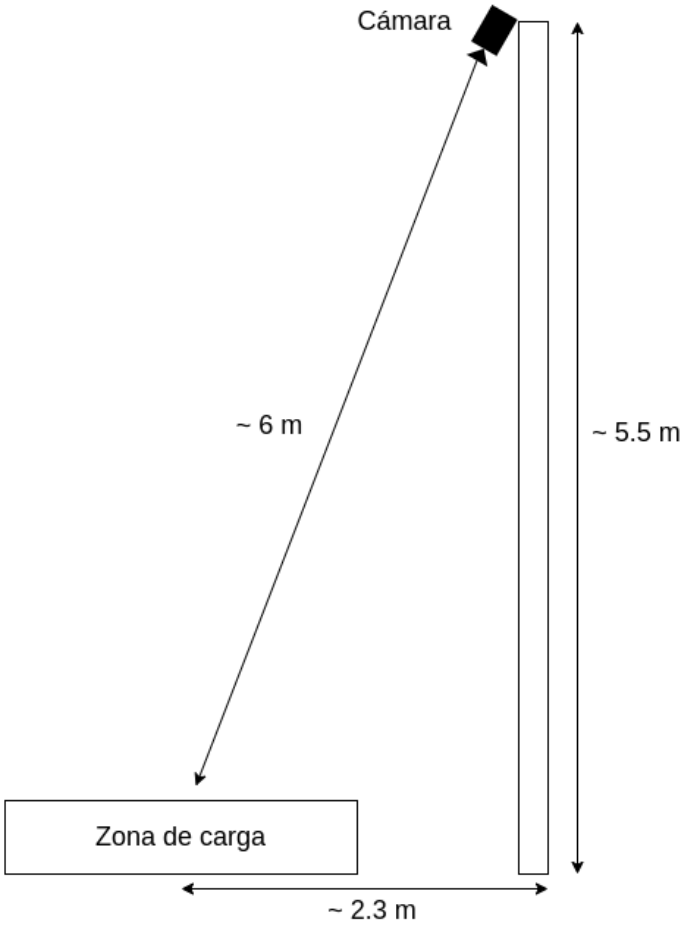

Figura 3.3: Configuración entorno de trabajo.

Por otro lado, la posición en la que puede estar grabado el código es aleatoria. El programa informático encargado de realizar el grabado a láser siempre graba el código en unas coordenadas en base a una posición absoluta con respecto a la base de las chapas. Sin embargo, las chapas tienen formas triangulares y el tamaño de estas varía de una chapas a otras, haciendo que por lo tanto, la posición del código con respecto a la chapa

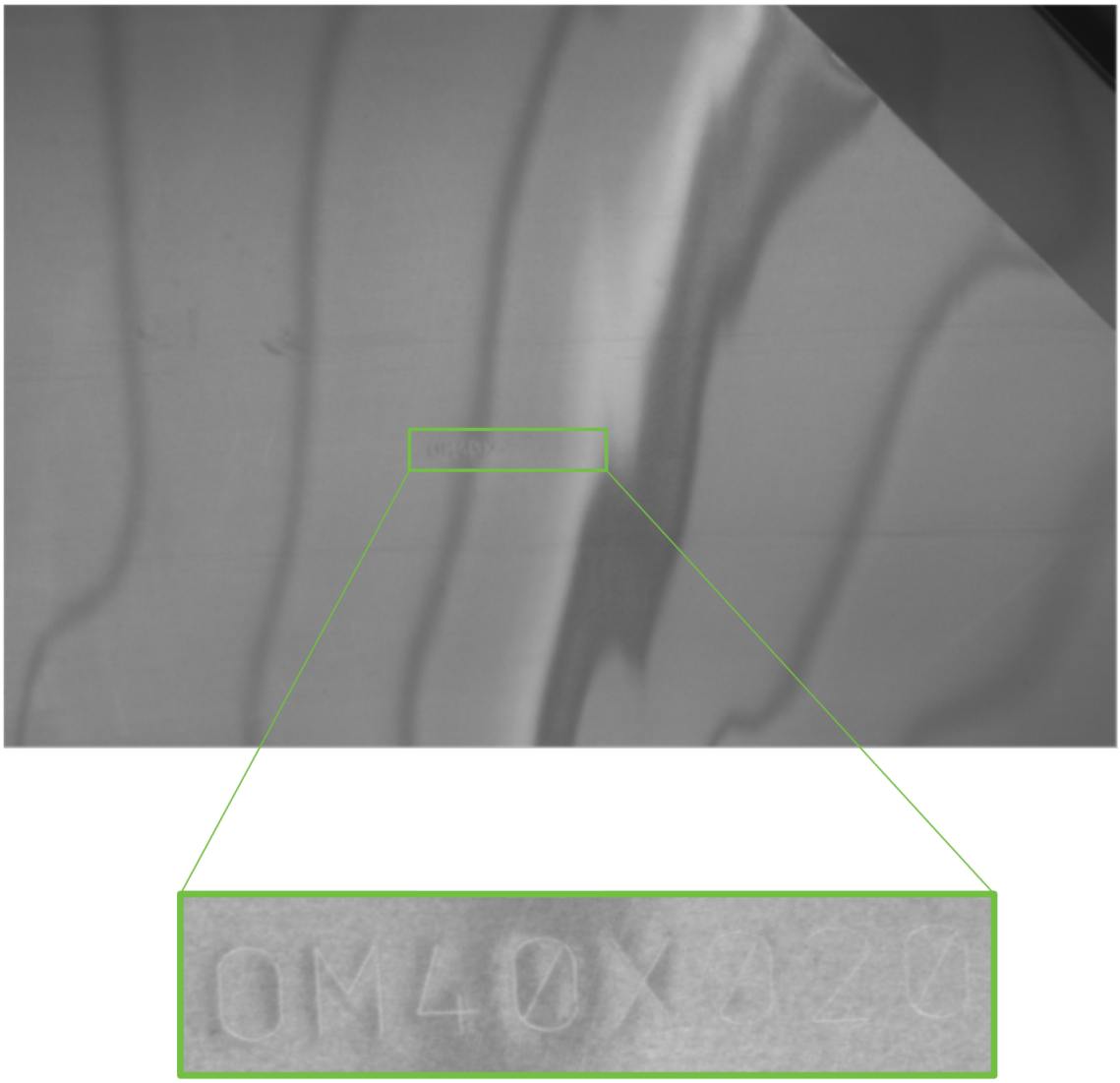

Figura 3.4: Ejemplo imagen lona y código OM40X020 grabado.

también varíe. De este modo, el código grabado a láser puede posicionarse en cualquier parte del rectángulo de 120×90 cm que la cámara captura en la imagen.

La calidad de los grabados no es siempre la misma debido al desgaste que sufre el cabezal del láser y la propia morfología del aluminio, y en numerosas ocasiones la nitidez y el contraste de este es muy baja. Esto se puede apreciar también en la Figura 3.4.

También es importante que la inferencia de los modelos sea lo suficientemente rápida para no entorpecer el trabajo de los operarios que cargan las lonas. En la zona de trabajo se encuentra un palé en el que los operarios colocan las lonas, y el tiempo que transcurren entre la carga de dos lonas consecutivas es aproximadamente 2 segundos. Este dato se debe tener muy en cuenta en el desarrollo del algoritmo, de tal modo que no entorpezca el proceso logístico de la empresa. Por lo tanto, la solución a implementar deberá tener un tiempo de inferencia inferior a este tiempo que transcurre entre la carga de lonas.

Por último es importante saber como se realizan las cargas de estos proyectos. Cuando se va a realizar un proyecto se cuenta con un fichero CSV el cual almacena la información relevante de dicho proyecto, siendo este fichero un *ground truth* sobre el que contrastar los resultados obtenidos en el reconocimiento. Estos ficheros almacenan los códigos de las lonas que se van a cargar así como el número de lonas de cada código que se van a cargar. Esto quiere decir que puede haber diferentes lonas con el mismo código. Como bien se comentaba anteriormente, los 3 últimos dígitos del código proporcionan la información de la geometría de la lona, y puede haber varias lonas con la misma geometría dentro de un pedido. Para cada carga que se realiza, los primero 4 caracteres del proyecto siempre coinciden, ya que estos indican el lugar de destino de las lonas y el código identificador del proyecto. Por lo tanto a la hora de diseñar la solución se podría utilizar estos primeros 4 caracteres a modo de información relevante para el reconocimiento del código. Esto se explicará más detalladamente en el apartado 4.9.3 .

3.2. Preprocesamiento de la imagen

Si bien es cierto que los modelos actuales de OCR son muy extendidos y tienen una amplia aplicabilidad, antes de tratar de reconocer un texto será necesario, en la mayoría de los casos, realizar un preprocesamiento de la imagen con el fin de obtener unos buenos resultados por parte del algoritmo.

Debido a que estos modelos están muy influenciados por los cambios de luz y color para el correcto reconocimiento de los caracteres, es necesario hacer que el texto sea lo más legible posible, lo que se traduce en una mejor distinción de los caracteres sobre el fondo en el que se encuentran. Para ello hay una gran variedad de técnicas que se pueden aplicar. Sin embargo, esta es una de las grandes desventajas de estos modelos. Dicho preprocesamiento deberá diseñarse en específico para cada problema que se quiera resolver.

A continuación se listan algunos de los preprocesamientos más típicos que se llevan a cabo antes de utilizar un modelo de OCR convencional y el objetivo que se busca con cada una de ellos:

- ***Histogram Equalization***: La ecualización de histograma [24] es una técnica utilizada para el ajuste/aumento del contraste de una imagen utilizando como información el histograma de colores de dicha imagen.
- ***Binarization***: Un proceso de binarización tiene como finalidad convertir los píxeles de una imagen a blancos y negros. Las imágenes se toman como matrices de valores, en las que cada uno de ellos determina el color de un píxel, en los que dichos valores se mueven en un rango que normalmente es de 0 a 255. Con el proceso de binarización la imagen se convertirá a una imagen en blanco y negro en la cual los valores blancos vendrán representados por 255's y los negros como 0's. El umbral a partir del cual un píxel pasa a ser o blanco o negro se denomina *threshold*, y es un parámetro esencial para el correcto funcionamiento de este procesamiento de imagen, y por lo tanto es necesario un ajuste preciso del mismo.

- ***Skew Correction***: En muchas ocasiones los textos que se tratan de extraer de las imágenes no están horizontales, y es necesario aplicar una corrección de la inclinación [25] de los mismos. Este preprocesamiento tiene como objetivo el dejar un texto horizontal.
- ***Noise Removal***: Los algoritmos de eliminación de ruido [26] buscan limpiar la imagen de pequeñas manchas, defectos o ruido en la misma con el fin de dejar el texto lo más legible posible. *Gaussian Filters* [27] y *Median Filters* [28] son dos de las técnicas más utilizadas.
- ***Erosion and Dilation***: Otra técnica para eliminar ruido de fondo y a su vez no perder legibilidad en el texto es usar técnicas combinadas de erosión y dilatación [29]. Básicamente se aplica un filtro a la imagen el cual trata de eliminar el ruido, y a continuación se aplica un nuevo filtro el cual resalta todo aquello que quede después de aplicar el primero, de este modo si algunos caracteres se han visto afectado por el primero, esto se pueden llegar a restablecer con el segundo.
- ***Edge Detection***: Técnica encargada de detectar bordes [30] de diferentes objetos o formas que aparezcan en una imagen. Para ello se suele utilizar diferencias en el gradiente en la imagen para detectar dichos bordes.

Estas son solo algunas de la gran cantidad de técnicas de preprocesado que se pueden aplicar. Como bien se puede apreciar, el gran número de preprocesamientos que pueden llegar a ser necesarios y a su vez el elevado número de parámetros que es necesario ajustar hacen que la primera parte que involucra a un proceso de OCR sea ardua y en algunos casos imposible de lograr mediante técnicas tradicionales.

3.3. Primera aproximación con OCR

En el desarrollo de este apartado se van a mostrar los resultados que se pudieron obtener aplicando técnicas tradicionales de *Computer Vision* para el procesamiento de la imagen, para posteriormente aplicar OCR's e intentar reconocer los códigos. En este apartado no se va a tratar el tema de la localización del código, si no que en primera instancia se realizó una pequeña prueba de concepto para ver si era factible el uso de técnicas tradicionales de OCR.

Como bien se ha comentado en el apartado anterior, hay un gran abanico de posibilidades de preprocesamiento de imágenes y de este modo prepararlas como datos de entrada a un modelo de OCR. En la siguiente Figura 3.5 se muestran una serie de resultados obtenidos a partir de este preprocesamiento el cual se basó en los siguientes pasos: primero una ecualización de histograma para tratar de aumentar el contraste de los caracteres sobre el fondo, a continuación una detección de bordes y por último una erosión y dilatación para tratar de eliminar ruido de fondo que pudiese interferir en el correcto reconocimiento.

Los códigos que se muestran en la Figura 3.5 son algunos de los casos en los que el OCR no era capaz de reconocer el código de manera correcta, o simplemente no era capaz de identificar ningún carácter en la imagen. Otra característica del proyecto que dificulta mucho la tarea de reconocimiento es el hecho de que puedan aparecer defectos en los códigos, como se puede apreciar en la última imagen de la Figura 3.5. Aun con un preprocesamiento de la imagen, la tarea de recuperar por completo el código y hacerlo legible se

Figura 3.5: Preprocesamiento de código fallidos

vuelve prácticamente imposible.

Una vez se encontró un flujo de preprocesamiento de imágenes que mejor se ajustaba a la variabilidad de posibles códigos, se pasó a realizar el reconocimiento y extraer resultados con OCR. Los resultados finales devolvían una tasa de acierto menor al 60 % en código completo. Por otro lado el tiempo de inferencia de estos modelos no era viable, debido a que en primer lugar era necesario hacer una gran cantidad de preprocesamiento, y el reconocimiento mediante el OCR. Todo ello sin contar el problema de intentar localizar el código a reconocer, ya que aplicar un OCR a una imagen de tamaño 5472×3648 para un código de aproximadamente 550×120 no es una solución demasiado eficiente.

3.4. Comparativa entre OCR y *Deep Learning*

Como se comentaba en el anterior apartado, inicialmente se realizaron pruebas con modelos de OCR. Sin embargo, esta solución no obtuvo los resultados deseados debido a que el OCR no era capaz de localizar ni reconocer de manera correcta el código en los casos en los que el grabado no estaba suficientemente marcado ni definido, o cuando las condiciones lumínicas no eran las adecuadas. Por lo tanto se investigaron otras vías para abordar el problema. Finalmente se decidió implementar una solución basada en *Deep Learning* debido a su alta capacidad de generalización y de detección de patrones difíciles de detectar a simple vista.

Conociendo las desventajas del OCR tradicional, es mas que razonable pensar en alguna solución alternativa para este problema en concreto, la cual sea lo suficientemente robusta frente a cambios en la iluminación y sobre todo que sea capaz de enfrentarse a la escasa legibilidad que presentan los códigos grabados a láser en las lonas de aluminio con las que se pretende trabajar.

Una de las grandes ventajas de los algoritmos de *Deep Learning* es su gran potencial a la hora de generalizar. Si se cuenta con un *dataset* lo suficientemente grande y variado para entrenar a estos modelos, estos

pueden ser capaces de detectar patrones sin la necesidad de realizar preprocesamientos, o al menos no es necesario preprocesar la imagen al mismo nivel que para un OCR. Como bien se comentaba en el apartado 3.2, hay una gran variedad de preprocesamientos disponibles. Si a esto se le suma la gran variedad de condiciones de luz y contrastes que se pueden dar en este caso de uso, y observando la Figura 3.2, el encontrar la combinación de correcta de preprocesamientos es algo muy complejo, incluso no tendría por que ser siempre la misma combinación de preprocesamientos la válida, si no que es muy probable que para cada imagen de entrada hubiese que realizar un preprocesamiendo en concreto. Para conocer la gran ventaja de las técnicas de *Deep Learning* frente a un OCR tradicional en este caso de uso en concreto, se puede observar la Figura 3.6. En ella se puede ver la imagen inicial en la parte superior, la imagen preprocesada ajustando el contraste en el medio y por ultimo la imagen con una ecualización de histograma, binarización y finalmente erosión y dilatación. Como bien se aprecia, es una tarea compleja el intentar hacer que el código sea más legible sin perder información de los propios caracteres en algunos de los pasos del preprocesamiento.

Aun con un preprocesamiento *ad-hoc* como este, el OCR tradicional no es capaz de reconocer el código que aparece en la imagen, mientras que el modelo de *Deep Learning* desarrollado en este trabajo si que es capaz de reconocer el código JB20X124 sin la necesidad de realizar ningún preprocesamiento, es decir, trabajando con la imagen original de la parte superior de la Figura 3.6.

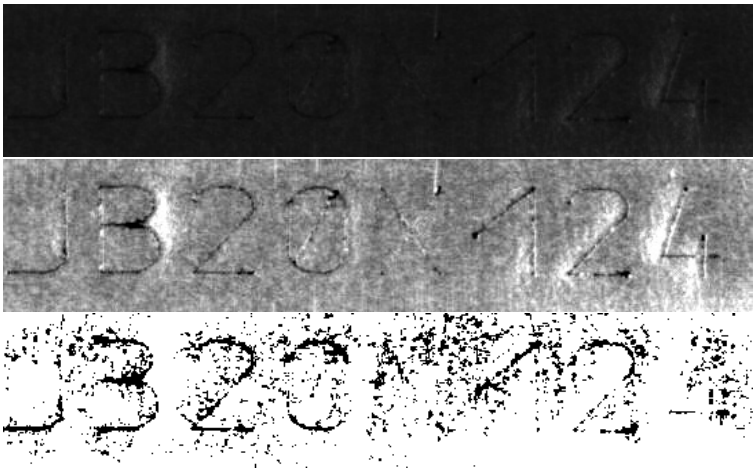

Figura 3.6: Código JB20X124: Preprocesamiento código

Sin embargo, los modelos de *Deep Learning* también cuentan con desventajas. Una de ellas es el gran trabajo que hay que realizar para recolectar los datos de entrenamiento. Si se habla de algoritmos supervisados, como es el caso de este trabajo, también será necesario etiquetar estos mismos, lo cual puede llegar a ser aun mas costoso que la propia recolección. Como se podrá apreciar en los apartados 4.3 y 4.9, la cantidad de imágenes es bastante elevada, y el etiquetado es una tarea muy costosa en términos de tiempo.

Una ventaja que pueden tener los modelos convencionales de OCR frente a los de *Deep Learning* es la velocidad de inferencia. En muchos casos, para conseguir una velocidad de inferencia elevada es necesario contar con algún hardware especializado en ejecutar este tipo de algoritmos, como bien pueden ser las tarjeta gráficas o *GPU's*, algo que no suele ser necesario en modelos de OCR. A su vez, este tipo de hardware es

de gran importancia para el entrenamiento de modelos de *Deep Learning*, mientras que para los OCR's no es necesario ningún entrenamiento de modelo. Sin embargo, para este caso de uso se cuenta con una GPU para el entrenamiento de los modelos, específicamente con un modelo RTX 2060, en la que la velocidad de inferencia de las técnicas de *Deep Learning* incrementa de manera notable. Teniendo en cuenta el preprocesamiento de la imagen que podría ser necesario para un OCR, estos algoritmos serían notablemente más lentos que un modelo de *Deep Learning* sobre *GPU* que, en un principio, no tendría por que necesitar ningún preprocesado. Conociendo las restricciones de tiempo de inferencia por parte del proyecto, esta es una ventaja de los modelos de *Deep Learning* muy a tener en cuenta.

Capítulo 4

Deep Three-phase OCR

4.1. Introducción al modelo D3POCR

El OCR en tres fases utilizando únicamente *Deep Learning* o D3POCR, es un algoritmo basado en modelos de detección de objetos y modelos de clasificación. En primer lugar, se cuenta con un modelo de detección de objetos para localizar el código en la chapa de aluminio. En segundo lugar se utiliza una técnica con la que se intenta ajustar de la mejor manera posible las coordenadas del código detectado. Finalmente, se cuenta con dos redes CNN, las cuales son las encargadas de realizar el reconocimiento de los caracteres de dicho código. Este algoritmo sigue siendo en cierto modo un OCR, sin embargo, se ha sustituido la parte en la que se utilizan matrices de bits para intentar reconocer los patrones de los caracteres por una red neuronal capaz de clasificarlos.

A lo largo de este apartado se explicarán cada una de las 3 fases que conforman el algoritmo D3POCR, así como el *dataset* que se ha utilizado para cada una de ellas y los procesos de entrenamiento que se han implementado. Al final de cada una de las fases se analizarán los resultados obtenidos. Finalmente se realizará un análisis de los resultados del modelo completo tanto de manera teórica como con datos reales obtenidos de la fase de pruebas real.

A continuación, en la Figura 4.1 se muestra como sería de manera esquematizada y resumida el flujo completo de funcionamiento del algoritmo D3POCR. Se muestra en la leyenda cada una de las fases del algoritmo representadas.

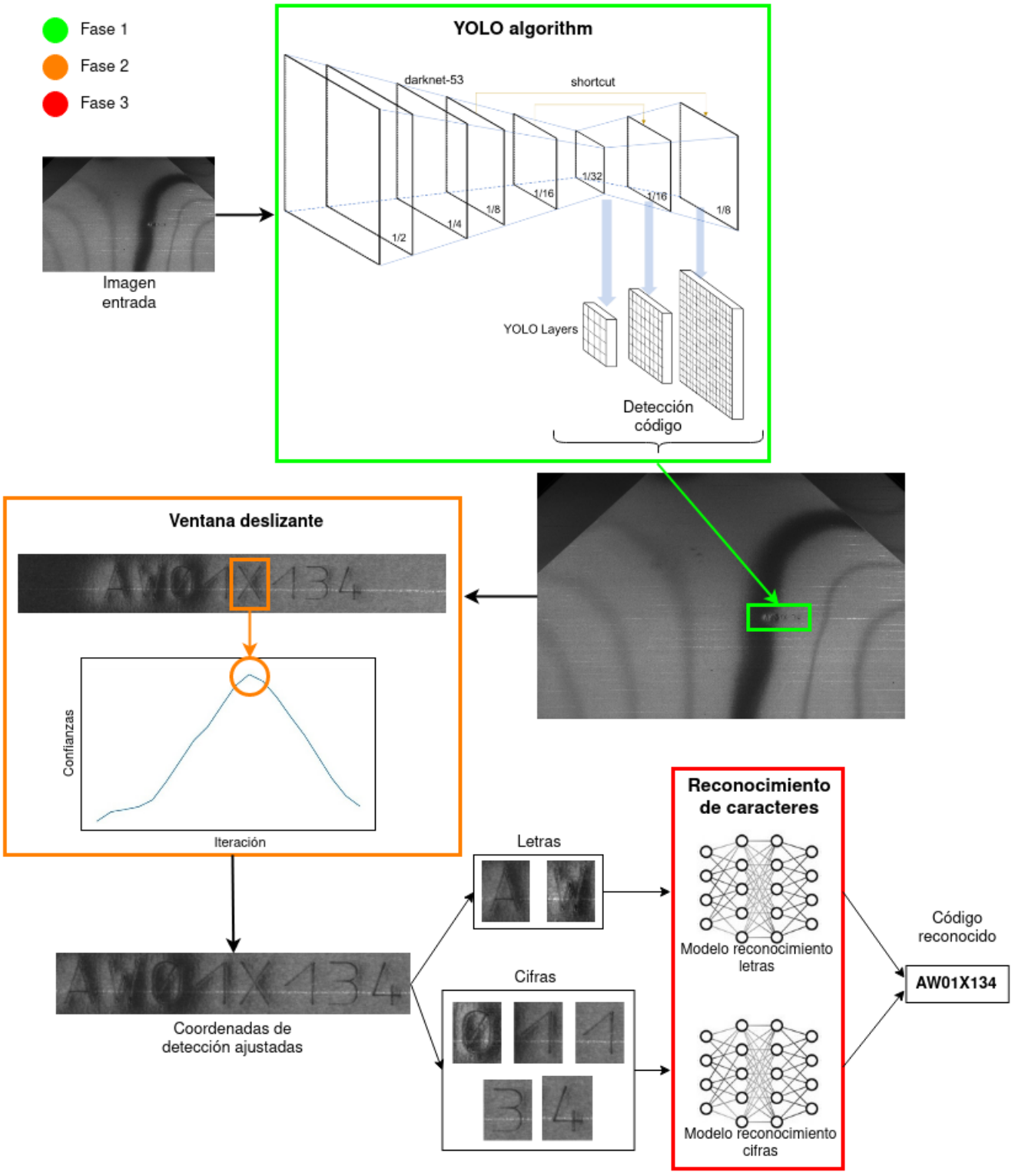

Figura 4.1: Flujo del algoritmo D3POCR

4.2. Fase 1: Detección de código

Para la localización del código, se ha decidido entrenar el modelo de *Deep Learning* de detección de objetos YOLOv3, debido a la mayor eficiencia que tiene con respecto a otros modelos de detección.

YOLO, por sus siglas, *You Only Look Once*, es un modelo de detección de objetos cuya principal diferencia frente a otros detectores es la velocidad de inferencia y la precisión. Las R-CNN [31] son también modelos de detección de objetos, sin embargo, estas calculan una serie de regiones sobre la imagen de entrada, en este caso 2000. Posteriormente cada una de estas regiones se pasa a un modelo CNN. Este modelo actúa como un extractor de características de la imagen, devolviendo un vector de longitud 4096. Finalmente, cada uno de estos vectores es introducido en una *support vector machine* [32] para que esta clasifique cada región.

Este modelo tiene una serie de inconvenientes. Por un lado, debido a la gran cantidad de pasos secuenciales intermedios, es un modelo prácticamente imposible de implementar para una solución en tiempo real, debido a que la velocidad de inferencia es extremadamente baja. Por otro lado, el algoritmo para la extracción de las regiones de la imagen de entrada es un algoritmo parametrizado de manera fija y no es posible realizar ningún ajuste mediante entrenamientos. Por lo tanto puede predecir regiones que realmente no interesen.

Posteriormente se desarrollaron los modelos Fast R-CNN [33] y Faster R-CNN [34], los cuales implementaban mejoras de rendimiento con respecto a su predecesor. Si bien es cierto que el modelo de Faster R-CNN si es implementable para problemas en tiempo real, sigue siendo un modelo más lento que YOLOv3 y con una menor precisión.

El modelo de YOLOv3 implementa una mejora muy significativa con respecto a los modelos anteriormente presentados, y es que este modelo analiza la imagen completa en una sola iteración, sin necesidad de calcular estas regiones de interés. Básicamente, este modelo genera un *grid* de dimensión $S \times S$ sobre la imagen. Cada celda de este *grid* es encargada de generar N *bounding boxes*, así como sus clases y sus confianzas. Para ello se utiliza una red CNN. Tras haber calculado las confianzas, a cada celda se le asigna la clase y el *bounding box* con la mayor confianza. Una vez realizado esto se obtienen una serie de detecciones. Sin embargo, estas detecciones no son las detecciones finales, debido a que hay muchas de ellas que no tienen la predicción correcta. Estas predicciones suelen ser un clúster de *bounding boxes* apuntando hacia un mismo objeto. Para resolver este problema se aplica un algoritmo denominado *Non-Max Suppression* [35]. Este algoritmo lo que intenta estimar es cual sería la mejor aproximación de las coordenadas del objeto en base a todas sus detecciones, y para ello, utiliza una técnica denominada IOU, por sus siglas en inglés, *Intersection Over Union*.

Tal y como se puede observar en la Figura 4.2, esta técnica calcula cual es el porcentaje de solapamiento entre dos *bounfing boxes*. Se escoge la detección con la confianza más alta y se compara con el resto, y si el porcentaje de IOU supera un cierto umbral, se da por hecho que esas dos detecciones apuntan hacia el

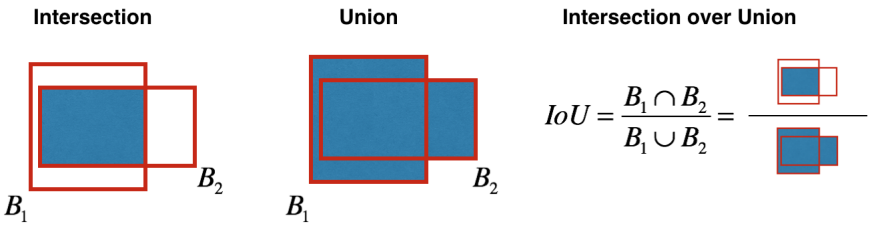

Figura 4.2: Ejemplo de intersección sobre unión en *bounding boxes* inferidas

mismo objeto, descartando por lo tanto aquella detección que tiene menor confianza. Este proceso se repite para todas las detecciones que se han obtenido del modelo hasta obtener las detecciones finales de los objetos que aparecen en la imagen.

En el caso de este proyecto, se extrapoló la detección de un objeto en un entorno a la detección de un código grabado a láser en una chapa de aluminio. La ventaja en este caso es que no es necesario manejar diferentes objetos detectados, debido a que estas chapas tan solo contienen un código, por lo que se sabe que el modelo solo debería predecir una sola *bounding box*.

4.3. *Dataset*, entrenamiento y resultados para la detección de código

Para formar el *dataset* del modelo de detección de código, se obtuvieron un total de 2057 imágenes de lonas, todas ellas de 20mpx, lo que arrojaba un total de 5472×3648 píxeles por cada imagen. Estas resoluciones son excesivamente grandes para un modelo como el YOLOv3, el cual está diseñado para trabajar con imágenes de un orden de magnitud menor. Por lo tanto a la hora de realizar el entrenamiento en el modelo se aplicaba un resize hasta un tamaño de 512x512 píxeles. De este modo la capa de entrada al modelo es de 512×512 píxeles.

El entrenamiento de este modelo no se realiza partiendo de 0, es decir, con pesos inicializados de manera aleatoria, si no que se cuenta con lo que se denomina un *backbone* el cual está pre-entrenado, aplicando de esta manera la técnica de *transfer-learning*. Este *backbone* pertenece a *darknet*, y su nombre es darknet53.conv.74. Gracias a esta aplicación de *transfer-learning* se puede reducir en gran medida el tiempo de entrenamiento, a parte de poder alcanzar unos buenos resultados con un número de imágenes relativamente pequeño para este tipo de problema.

Antes de comenzar con el entrenamiento del modelo, se decidió aplicar un *data augmentation*. Este se basó en realizar rotaciones en el eje horizontal y vertical a la imágenes originales (Figura 4.3). Debido a que este modelo no se encarga de realizar el reconocimiento de caracteres, sino únicamente de inferir su posición, parecía una buena opción la de rotar las imágenes tanto en el eje vertical como horizontal para de ese modo obtener 4 veces más número de coordenadas de objetos para entrenar al modelo. No fue necesario etiquetar todas las imágenes obtenidas del *data augmentation*. Simplemente se automatizó un proceso que calculaba cual sería la posición de los códigos para cada una de las rotaciones aplicadas. Por lo tanto, rea-

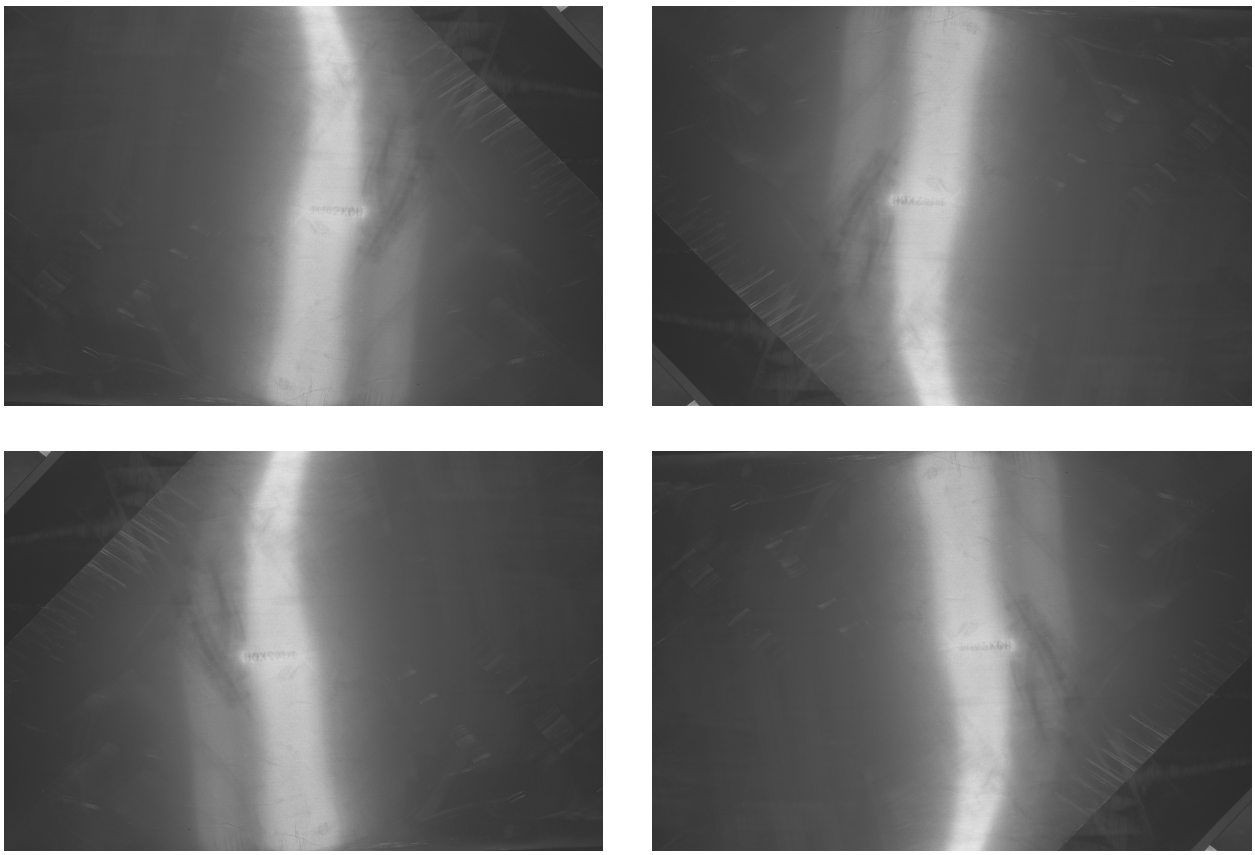

Figura 4.3: *Data augmentation* para YOLOv3: rotaciones aplicadas

lizando este *data augmentation* se obtuvo un total de 8228 imágenes. Para la separación entre conjunto de entrenamiento y test se realizó un *split* de 90 % / 10 %.

	Train	**Test**
YOLO	7405	823

Tabla 4.1: Instancias *dataset* YOLO

El utilizar rotaciones de imágenes de otro tipo como método de *data augmentation* no se contempló como opción debido a que los códigos nunca aparecen rotados, siempre se encuentran en una posición horizontal. La otra posible opción para realizar un aumento del *dataset* fue el realizar traslaciones para mover el código respecto a su posición original, sin embargo, el número de imágenes obtenidas mediante las rotaciones fue suficiente para alcanzar unos buenos resultados del modelo.

Una vez se completó el entrenamiento del modelo, se calcularon y analizaron sus diferentes métricas. Los modelos de detección de objetos tienen principalmente dos métricas de resultados de entrenamiento para medir su comportamiento. Estas son el mAP y el mIOU. El mAP calcula la precisión media del modelo sobre los objetos a clasificar. La aplicación de esta métrica en este caso de uso no tenía sentido, ya que

solo existe una única clase a predecir. Por otro lado, la métrica mIOU es la encargada de calcular cual es la precisión media de inferencia en cuanto al IOU se refiere, dando como resultado el acierto medio en el solapamiento de las predicciones de *bounding boxes*. Aunque esta métrica si que tiene sentido para calcular la precisión del modelo para este caso de uso, la segunda fase del modelo D3POCR es la encargada de mejorar dicho solapamiento mediante el ajuste de las coordenadas del *bounding box* inferidas por el modelo. Por lo tanto la métrica que se extrajo del entrenamiento fue simplemente la capacidad de que el modelo YOLOv3 fuese capaz de detectar o no un código en una lona, sabiendo que en todas ellas aparece dicho código. La precisión media del modelo a la hora de detectar un código medida tras el entrenamiento sobre el conjunto de test fue del 99.15 %.

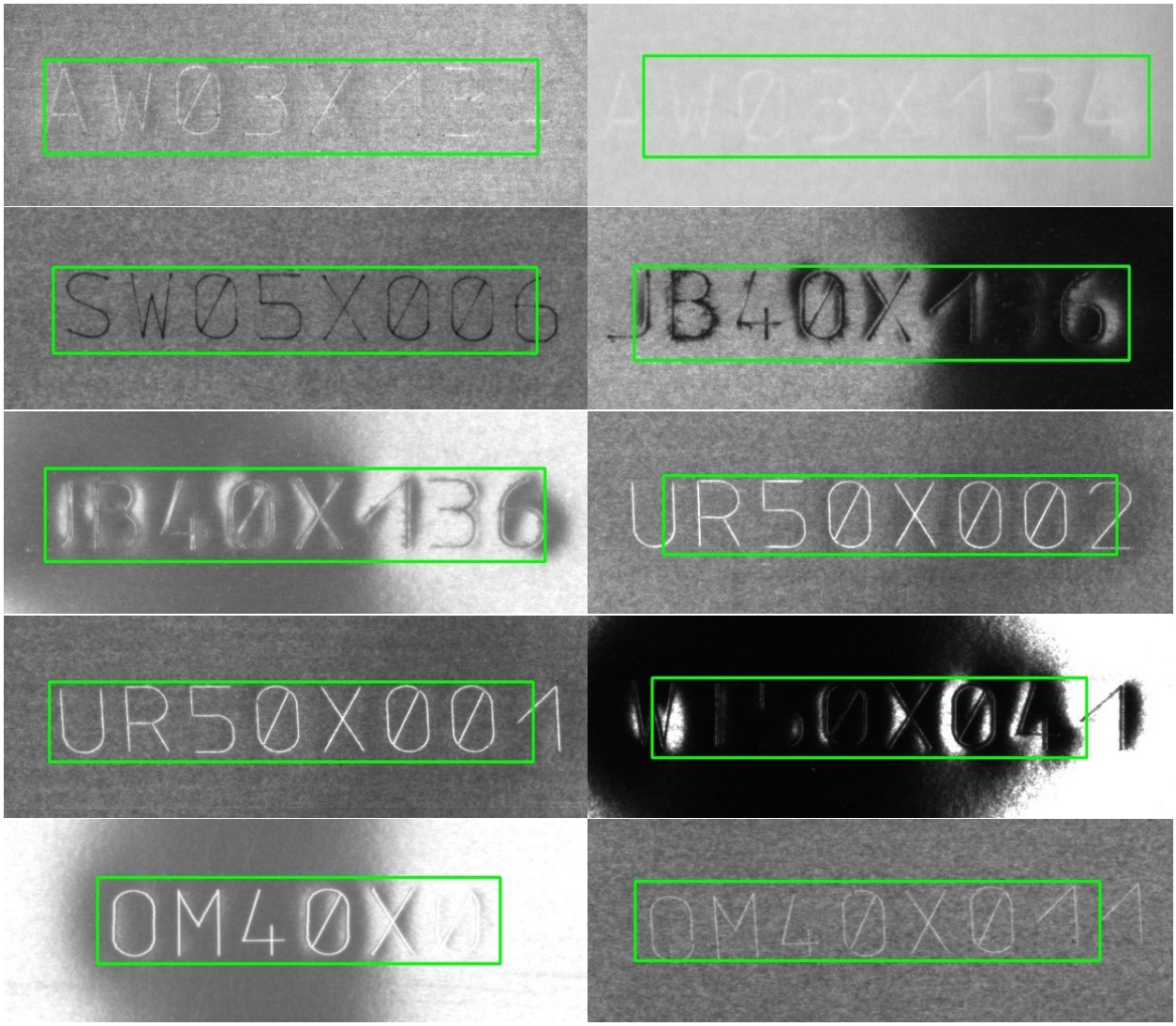

Figura 4.4: Ejemplos detecciones YOLOv3

Como bien se comentaba al inicio de este apartado, el modelo de detección de código no es suficiente para realizar un recorte preciso del código a reconocer, y por lo tanto, se necesita un paso intermedio de procesado de las coordenadas que retorna el modelo para un correcto reconocimiento. A continuación se

muestran una serie de detecciones de códigos por parte de YOLOv3 (Figura 4.4) para que se pueda comprender esta necesidad de ajuste de las coordenadas del código detectado. Para facilitar la visualización de estas imágenes, debido a que estas tienen un tamaño de 5472×3648, como bien se mencionaba antes, se ha realizado un recorte de dicha imagen alrededor de la detección, la cual se muestra como un rectángulo de color verde, la cual correspondería a las coordenadas que devuelve el modelo YOLOv3.

4.4. Fase 2: Ajuste de detección del código

La segunda fase del modelo es un ajuste del recorte del código. El modelo YOLOv3 es un modelo que puede alcanzar una gran precisión a la hora de inferir coordenadas de un objeto detectado. Sin embargo, esta precisión no es suficiente para el ajuste que se necesita a la hora de la correcta lectura del código, como bien se comentaba en el apartado anterior. Los modelos de reconocimiento de caracteres entrenados, reconocen el código carácter a carácter, siendo cada entrada al modelo una imagen recortada de cada carácter a reconocer del código completo. De este modo, estos recortes han de estar ajustados de una manera muy precisa, quedando el carácter en cuestión en una posición lo más céntrica posible dentro de la imagen. Si bien es cierto que si el modelo recibe una imagen de un carácter que no se ve en su totalidad, este es capaz de reconocerlo en la mayoría de situaciones, en otras hay caracteres que se pueden llegar a confundir con otros si estos aparecen recortados. Por ejemplo, el carácter O podría ser confundido con una C si este aparece recortado por el lado derecho. Así mismo podría confundirse, por ejemplo, con una Q si esta aparece recortada del mismo modo.

Con el fin de realizar un recorte preciso del código, se estudiaron diferentes vías. Una de ellas fue por ejemplo el uso de algoritmo de detección de bordes para detectar donde empezaba y acababa un código. Sin embargo este tipo de algoritmos no consiguió detectar de manera precisa dichos bordes, debido a la gran variedad de cambios de luz que podían sufrir las chapas. Por otro lado, hay códigos que en ocasiones están grabados con menos fuerza debido a la morfología de la chapa, dando lugar a caracteres muy poco marcados en los que era imposible aplicar esta técnica de manera correcta. En algunas ocasiones, estas chapas podían estar dañada arañazos o manchas, lo cual dificultaba aun más esta tarea. Finalmente se optó por una solución basada también en *Deep Learning* intentando desarrollar de esta manera una solución que fuese resistente a este amplio abanico de posibles defectos.

4.5. Algoritmo de ventana deslizante

Esta técnica de ventana deslizante basa su comportamiento en las confianzas del la clase X arrojadas por uno de los modelos de *deep learning* de reconocimiento de caracteres entrenado para esta solución, el cual recibe como entrada recortes del código completo. Para el correcto entendimiento de este apartado es necesario introducir brevemente como se han diseñado los dos modelos de *deep learning* de reconocimiento de caracteres, aunque en el apartado 4.7 se explicaran en profundidad. Para el reconocimiento de los caracteres se han desarrollado dos modelos diferentes, uno para el reconocimiento de letras, y otro para el reconocimiento de números, con el fin de reducir la dimensionalidad del problema a resolver, y de este modo tratar de aumentar la precisión a la hora del reconocimiento. Esto es debido a que la estructura del código

es conocida, y se sabe en que posición del código se encuentran las letras y en que posición los números.

La utilización en esta fase de la llamada confianza que devuelve un modelo para cada clase se puede explicar conociendo la fórmula explicada en la ecuación 1.6 en la que se explicaba la función de salida *softmax*.

Lo que se logra mediante la implementación de esta función de activación en las capas de salida de estos modelos de clasificación es que se transformen las salidas del modelo en una distribución de probabilidades de clases, siendo dicha suma de probabilidades 1. En cuanto al modelo se refiere, la interpretación de esta probabilidad es algo compleja, pero podría traducirse en como de seguro está el modelo de la clasificación que esta realizando sobre la entrada que ha recibido. Es decir, si sobre la clase A el modelo arrojase un valor de 0.8, se podría decir que el modelo está seguro en un 80 % que la imagen de entrada que ha recibido corresponde con la clase A. Este valor no corresponde con la probabilidad real de que ese sea el carácter A, pero si que es una buena aproximación para saber que entrada ha recibido un modelo. Por lo tanto, conociendo el funcionamiento de la salida de esta función *softmax*, se pretende obtener el valor para la clase X de la imagen que se recibe.

Sin embargo, hay una pequeña excepción en este diseño, y es que en el modelo de reconocimiento de cifras, se haya como posible clase a predecir la letra X. Como bien se puede apreciar en la estructura del código, la letra X se encuentra entre dos caracteres numéricos. En un principio, se utilizaba la clase X del modelo de reconocimiento de letras, sin embargo, cuando se aplicaba ese modelo a los diferentes recortes, este se encontraba con imágenes de números (que como bien se ha mencionado anteriormente están a izquierda y derecha de la X). Debido a que el modelo de reconocimiento de letras no había sido entrenado con números, las confianza que arrojaba eran aleatorias cuando obtenía como entrada este tipo de imágenes, y por lo tanto no era una base confiable de la que partir para realizar esta calibración en el recorte del código.

De este modo, se introdujo el carácter X en el modelo de reconocimiento de cifras, para que así fuese capaz de distinguir correctamente tanto la X como los propios números que la rodean, y por lo tanto las confianzas que devolviese fueran más precisas. La probabilidad de encontrar una letra diferente a la X en el proceso de la ventana deslizante es muy improbable, debido a que el algoritmo tiene en cuenta el error medio que puede cometer YOLOv3, y por lo tanto está ajustado para que esto no ocurra. Sin embargo, se puede dar el caso de que en este proceso se encuentren letras y el modelo de reconocimiento de cifras sufra comportamientos inesperados. Para tratar de evitar este problema se pensó en realizar un modelo mixto, el cual fuese un único modelo capaz de clasificar las imágenes tanto de letras como de cifras. Finalmente no se optó por esta opción, y la explicación para ello se puede encontrar en el apartado 4.9.3 .

La técnica de ventana deslizante cuenta con dos fases. La primera de ellas calcula un conjunto de confianzas de la clase X extraídas del modelo de reconocimiento de números y las almacenará en una lista, la cual se denominará C. El punto de partida son las coordenadas del centro del *bounding box*, denominadas x_c, y_c que devuelve YOLOv3 y el tamaño de dicho recorte, cuya altura y anchura se expresará como h y w respectivamente. Una vez se cuenta con estas coordenadas x_c, y_c, se realiza un desplazamiento D, previamente calculado, hacia la izquierda en la imagen partiendo del punto x_c, y_c, obteniendo un nuevo punto x_0,

y_c de partida. Se realiza un recorte del código de un tamaño previamente calculado, el cual correspondería a un solo carácter y cuya altura y anchura vendrán definidas por h y w', tomando como centro de dicho recorte el punto x_0, y_c. Una vez se obtiene este nuevo recorte, se introduce dicha imagen recortada al modelo de reconocimiento de *Deep Learning*, y se extrae la confianza para el carácter X de la salida del modelo, almacenando el valor de dicha confianza. A continuación se realiza un desplazamiento hacia la derecha de d píxeles, obteniendo el centro x_1, y_c de un nuevo recorte de igual tamaño al anterior, h y w'. Este nuevo recorte se volverá a introducir al modelo de reconocimiento para obtener la confianza para el carácter X de la salida del modelo y almacenarla. Este proceso se repetirá hasta que se hayan realizado n desplazamientos hacia la derecha sobre el código.

Una vez se ha completado esta primera fase, se obtendrá un conjunto de n confianzas, las cuales se puede ver representadas en la Figura 4.5, la cual es un ejemplo de una posible iteración de esta primera parte del algoritmo. Lo que se desea extraer mediante estas confianzas es, teóricamente, el recorte del carácter X más centrado, el cual debería corresponder con la confianza más alta para dicha lista y por lo tanto, la mejor aproximación al recorte más ajustado del código completo. Si es conocido el centro del recorte de uno de los caracteres, y que estos están espaciados la misma distancia por igual entre todos, se pueden realizar recortes iguales partiendo de dicho punto obteniendo los recortes del resto de caracteres.

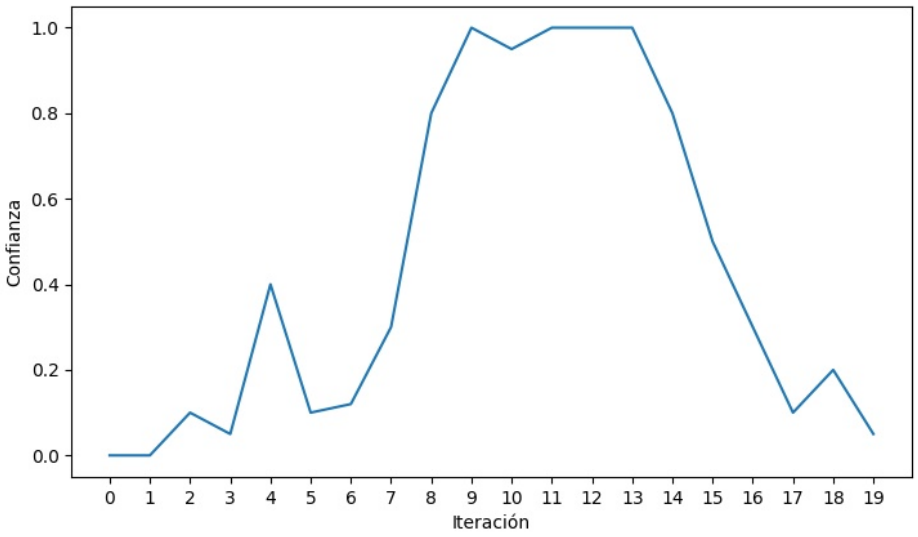

Figura 4.5: Confianzas obtenidas en la primera parte del algoritmo de ventana deslizante.

Sin embargo, si se observan las confianzas obtenidas, en la mayoría de los casos no hay una confianza mayor que el resto, si no que un conjunto de ellas pueden tener valor 1 o cercano a el, debido a que el modelo ha identificado con la máxima confianza diferentes recortes de la X, tal y como se puede apreciar en la Figura 4.5, mientras que el resto de confianzas más bajas corresponderán a recortes de otros caracteres, o a recortes del carácter X el cual no aparece en su totalidad.

Este comportamiento es lógico debido a que el modelo está diseñado para clasificar caracteres, y la confianza simplemente refleja como de seguro está el modelo de que el carácter que aparece en la imagen es el que dicho modelo esta prediciendo. En la Figura 4.6 se muestran tres imágenes del carácter X con las que el modelo arroja valores cercanos o iguales a 1 para dicha clase, aunque en ellas la posición de la X con respecto al centro de la imagen sea distinta. Solo se consideraría un recorte centrado la imagen central que aparece en esta Figura 4.6, mientras que los de la derecha y la izquierda, aunque con confianzas iguales, aparecen descentrados.

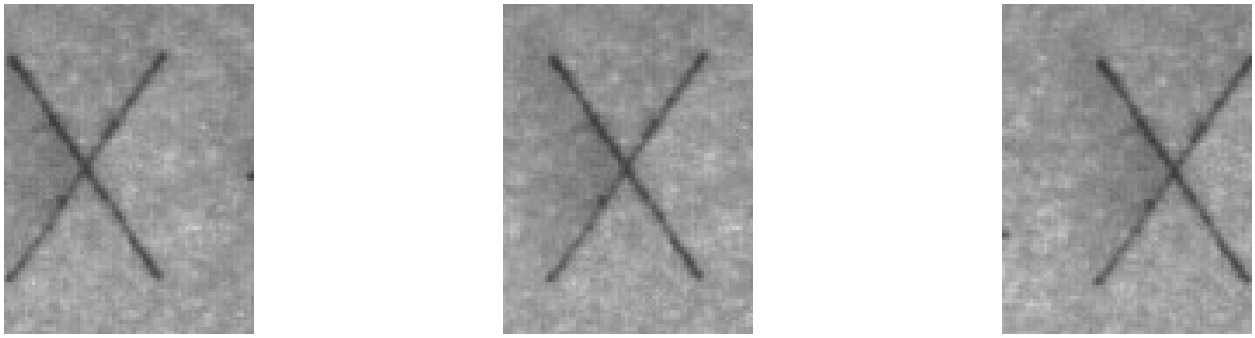

Figura 4.6: Ejemplos de recorte del carácter X con confianzas cercanas a 1.0

Para poder obtener con mayor precisión cual de todas las confianzas más altas corresponde al recorte del carácter X más centrado, se aplica esta segunda parte del algoritmo. Si se tiene en cuenta que las confianzas anteriormente calculadas se almacenan en forma de una lista ordenada según las iteraciones de la anterior parte del algoritmo, se puede aplicar un filtro F, de modo que las zonas dentro de la lista con confianzas más bajas se atenúen, y las zonas con confianzas mayores se acentúen.

En primer lugar se define un tamaño de filtro f, siendo este una máscara de 1 dimensión. A continuación se realiza un *padding* de tamaño $f/2$ a la lista de confianzas por ambos lados para asegurar que la lista que se obtenga tras aplicar el filtro es del mismo tamaño que la original, y de este modo no perder la trazabilidad de las posiciones originales sobre las que iteró la primera parte del algoritmo. A esta nueva lista con *padding* se le denominará C'. Este filtro de tamaño f básicamente es una función de sumatorio sobre los valores de las confianzas de C' sobre los que se encuentre posicionado. El filtro comenzará en la posición 0 de la lista C', calculará la suma de los f primeros valores, y la almacenará en una nueva lista denominada C''. El filtro se desplazará una posición sobre la lista C' y volverá a repetir la misma operación hasta llegar al final de C'. Gracias a este procesamiento se puede obtener una versión de la función que genera la lista de confianzas en la que sí que se obtiene un valor máximo, siendo este una mejor aproximación del recorte más centrado de X. En la Figura 4.7 se presenta esta nueva lista de confianzas procesadas. Dada esta nueva distribución de confianzas, se puede asumir que el recorte más centrado del carácter X corresponde a la posición con el valor más alto dentro de esta lista, el cual se denominará (x_f, y_c).

En el Algoritmo 1 se resume el funcionamiento, con $crop(x, y, w, h)$ la función que devuelve el recorte del tamaño de un carácter al código detectado, $modelx(img)$ como la función que llama al modelo y devuelve la confianza del carácter X y $pad(C)$ como la función que añade un *padding* a la lista C.

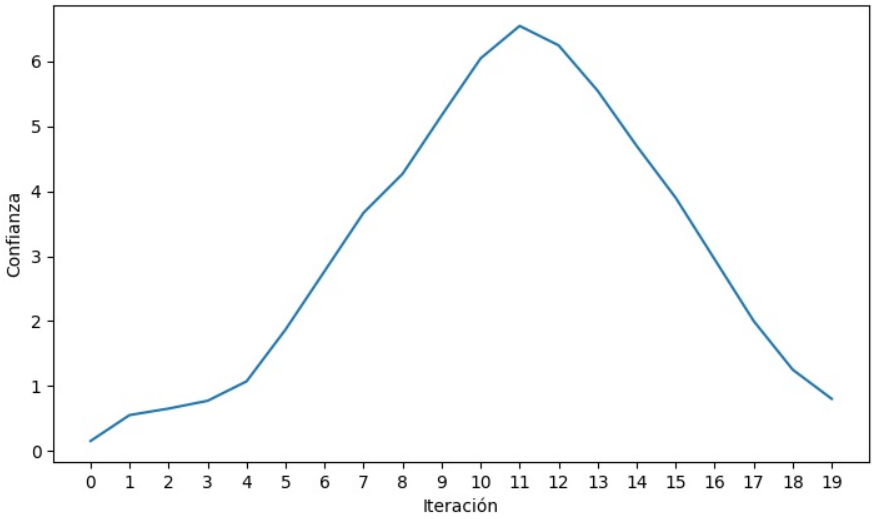

Figura 4.7: Confianzas procesadas.

Como bien se explicaba en el Apartado 3.1, se sabe la altura a la que se encuentra la cámara, y el tamaño del código a reconocer, por lo tanto se puede extrapolar el tamaño en píxeles que debería tener un código. Habiendo realizado los cálculos pertinentes, el código tiene un tamaño de 535 píxeles. Una vez se asume que ese valor máximo de la lista C'' corresponde al recorte centrado de X, y que el código tiene un tamaño de 535 píxeles, se pueden realizar 8 recortes de tamaño $ceil(535/8) = 67$ píxeles, obteniendo de este modo las imágenes de los caracteres que servirán como entrada para la siguiente fase del modelo.

Algorithm 1 Ventana deslizante

Require: x_c, y_c, w, w', h, D, f

$C[n], C'[m], C''[n]$ ▷ Se crean listas vacias de tamaño n y m

$x_0 \leftarrow (x_c - D)$

$i \leftarrow 0$

while $i \leq n$ **do**

$img \leftarrow crop(x_i, y_c, w', h)$

$C[i] \leftarrow modelx(img)$

$i \leftarrow (i + 1)$

end while

$C' \leftarrow pad(C)$

$i \leftarrow 0$

while $i \leq n$ **do**

$C''[i] \leftarrow \sum_{j=i}^{i+f} C'[j]$

$i \leftarrow (i + 1)$

end while

$(x_f, y_c) \leftarrow max(C'')$

4.6. Visualización del funcionamiento de ventana deslizante

El algoritmo de ventana deslizante es posiblemente la fase del algoritmo D3POCR más compleja y delicada. Por lo tanto a continuación se explicará de manera visual y esquematizada el funcionamiento del mismo. Se parte del centro de la detección que devuelve el modelo YOLOv3. Contando con esa posición, se aplicará un desplazamiento hacia la parte izquierda de la imagen a modo de margen de seguridad, debido a que los consecutivos desplazamientos que se realizará serán hacia la derecha, y se pretende asegurar que en alguno de estos desplazamientos aparezca el recorte del carácter X más centrado. En la Figura 4.8 se muestra un recorte de la imagen global de la lona, el punto que corresponde al centro de la detección de YOLOv3 x_c, y_c, y el punto de partida de este algoritmo, el cual se obtiene aplicando el desplazamiento D al punto x_c, y_c, obteniéndose la nueva coordenada x_0, y_c.

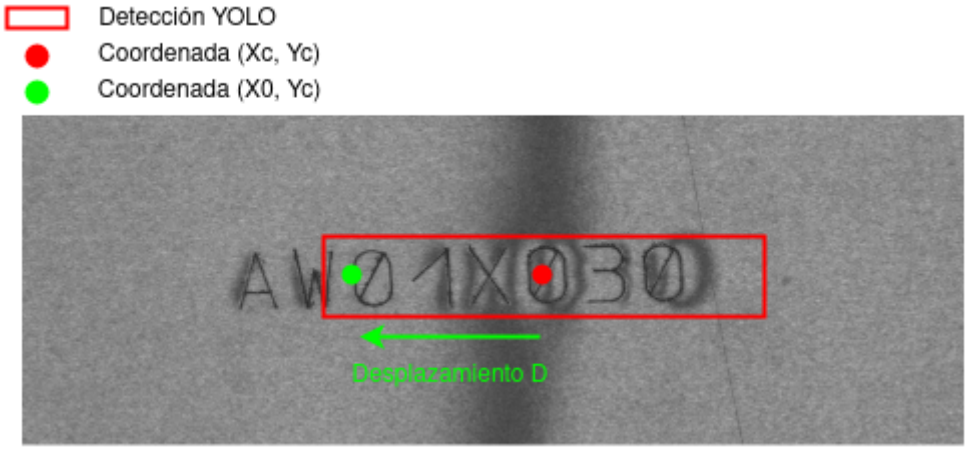

Figura 4.8: Punto de partida de la ventana deslizante.

Una vez se cuenta con el punto x_0, y_c, se puede comenzar a realizar movimientos iterativos hacia la derecha, para ir progresivamente obteniendo recortes del código de un tamaño igual al que se espera que tenga un carácter e ir introduciendo dichos recortes al modelo de reconocimiento de dígitos para obtener las confianzas del carácter X para cada uno de ellos. El posible recorte de partida sería el que aparece en la Figura 4.9.

El desplazamiento que se realiza hacia la derecha de esta ventana de recorte, el cual está definido en el anterior apartado con la variable d se ajustó a un total de 8 píxeles, y en total se realizan 20 desplazamientos. Este desplazamiento d y el número de iteraciones se ajustaron en base al error medio que comete el modelo YOLOv3 con respecto al centro real del código a detectar. Por lo tanto se recorre aproximadamente un 30 % del tamaño total del código con el algoritmo de ventana deslizante. En el desarrollo de este algoritmo se pensó en ampliar este margen de recorrido de la ventana deslizante, sin embargo no se quería incurrir en tiempos de ejecución demasiado elevados.

En la Figura 4.10 se muestran una serie de iteraciones del algoritmo. En cada una de estas iteraciones se puede apreciar como el algoritmo recorta una parte del código obteniendo el tamaño correspondiente a

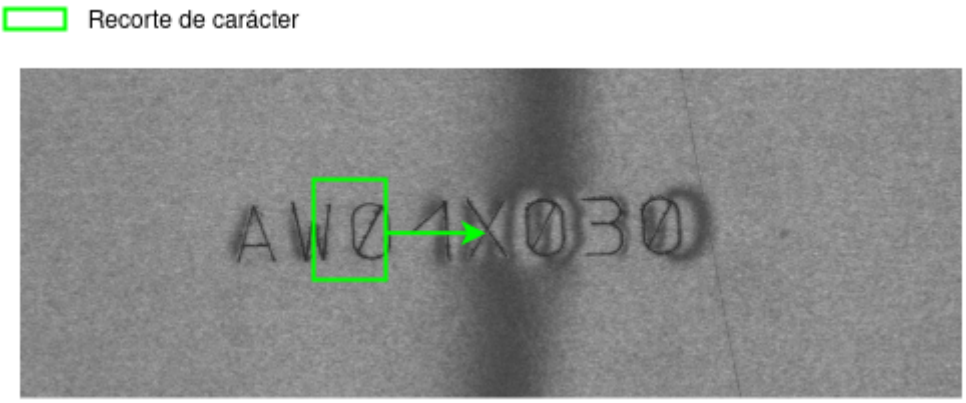

Figura 4.9: Recorte de partida de la ventana deslizante.

un carácter de dicho código. Una vez se obtiene dicho recorte se introduce en el modelo de reconocimiento de dígitos y se extrae la confianza de la clase correspondiente al carácter X. En esta misma figura se puede apreciar como para caracteres que no sean la X, la confianza arrojada es muy cercana a 1. Por otro lado, habrá numerosos recortes que se realicen que correspondan a espacios entre caracteres en los que además aparezcan partes de otros caracteres. Las confianzas arrojadas en este caso pueden ser muy aleatorias, pero es muy poco probable que en uno de estos casos dicha confianza se acerque a 1. Aun en el caso de que hubiera alguna confusión muy elevada por parte del modelo y arrojase una confianza elevada para un recorte, esto se solucionaría mediante la aplicación del filtro F, el cual añade robustez. Este filtro favorece a las zonas donde hay cierto número de confianzas muy elevadas, que deberían corresponder a sucesivos recortes X, y mitiga posibles errores cometidos por el modelo.

Según esta ventana deslizante se va aproximando a la X, se puede apreciar como poco a poco las confianzas van aumentando hasta llegar a valores máximos los cuales corresponden con este carácter (Figura 4.11). Sin embargo se genera una meseta en la gráfica, correspondiente a recortes de la X en diferentes posiciones, todos ellos con confianzas cercanas o iguales a 1; como bien se mencionaba en el apartado anterior, haciendo referencia a las Figuras 4.6 y 4.5.

De este modo, una vez obtenidas todas las confianzas C', estas se procesan mediante el filtro F, generando de esta manera un máximo en la nueva lista de confianzas C'', el cual debería corresponder teóricamente al recorte del carácter X más centrado de todos los procesados mediante la ventana deslizante. Esta segunda fase del algoritmo de ventana deslizante se corresponde con la Figura 4.12.

Finalmente, tomando como referencia este recorte del carácter X, se pueden realizar recortes de igual tamaño e igualmente espaciados a ambos lados obteniéndose los recortes del resto de caracteres, que servirán como entrada al modelo de reconocimiento.

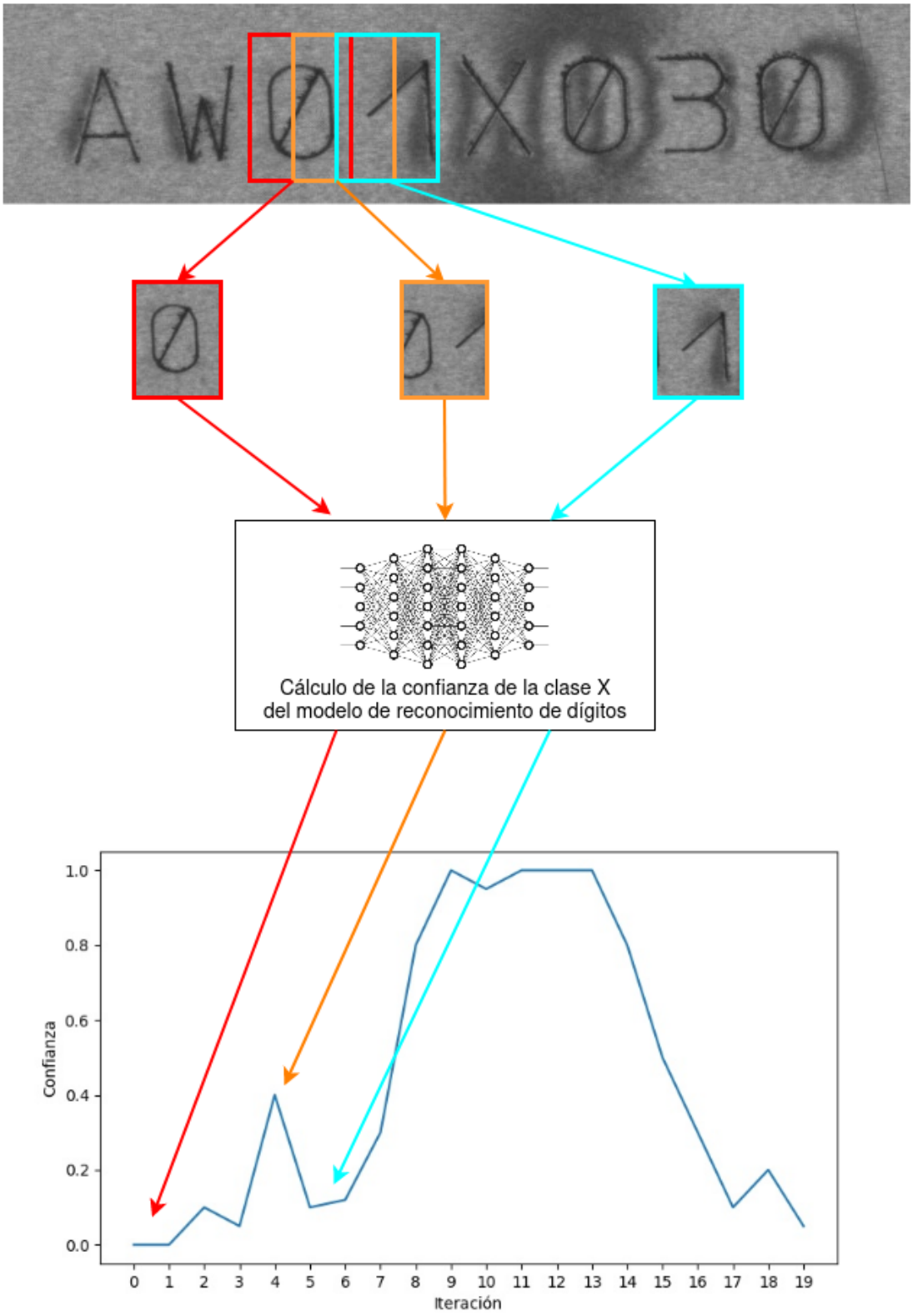

Figura 4.10: Ejemplos de recortes y respectivas confianzas.

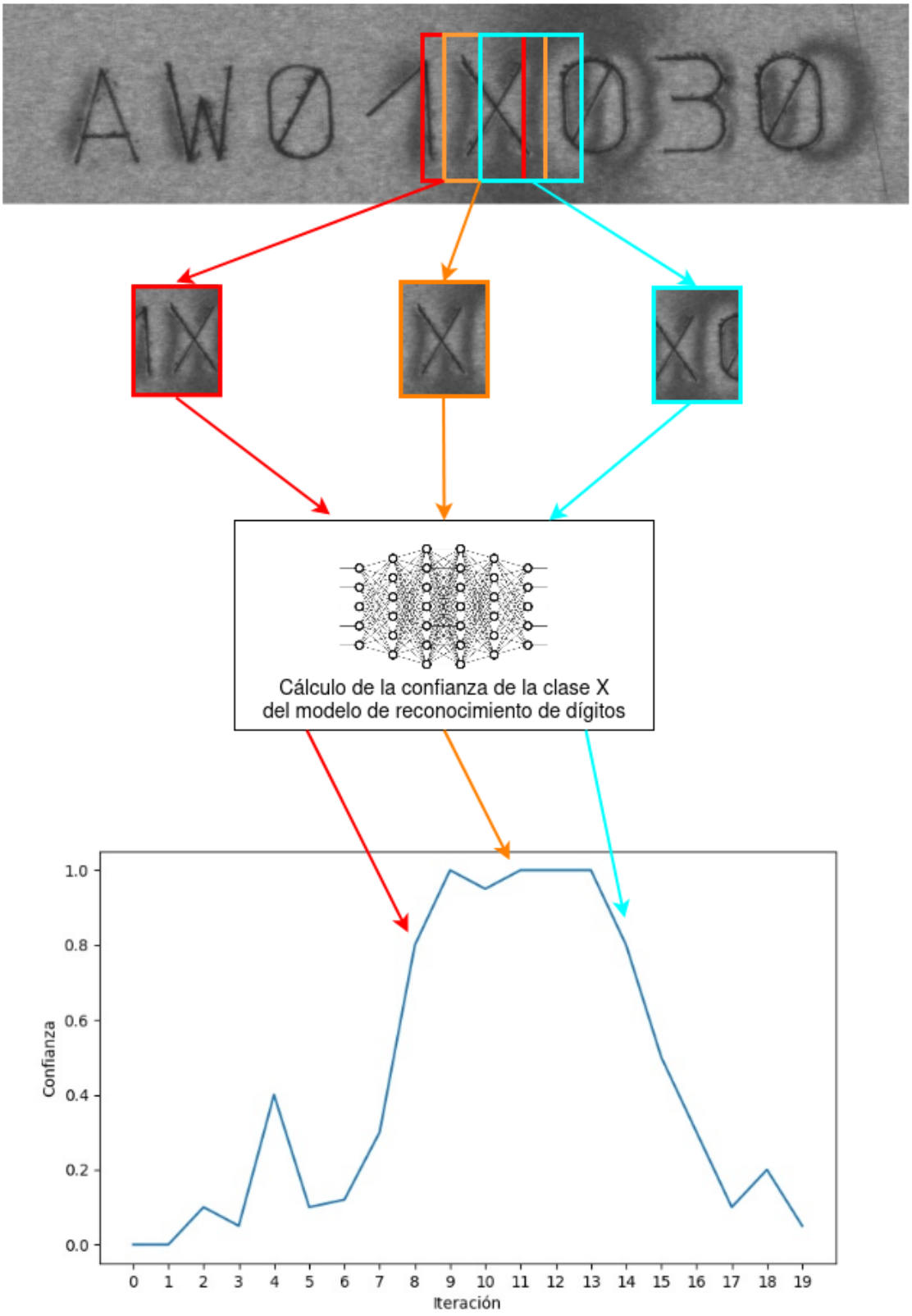

Figura 4.11: Ejemplos de recortes de X y respectivas confianzas.

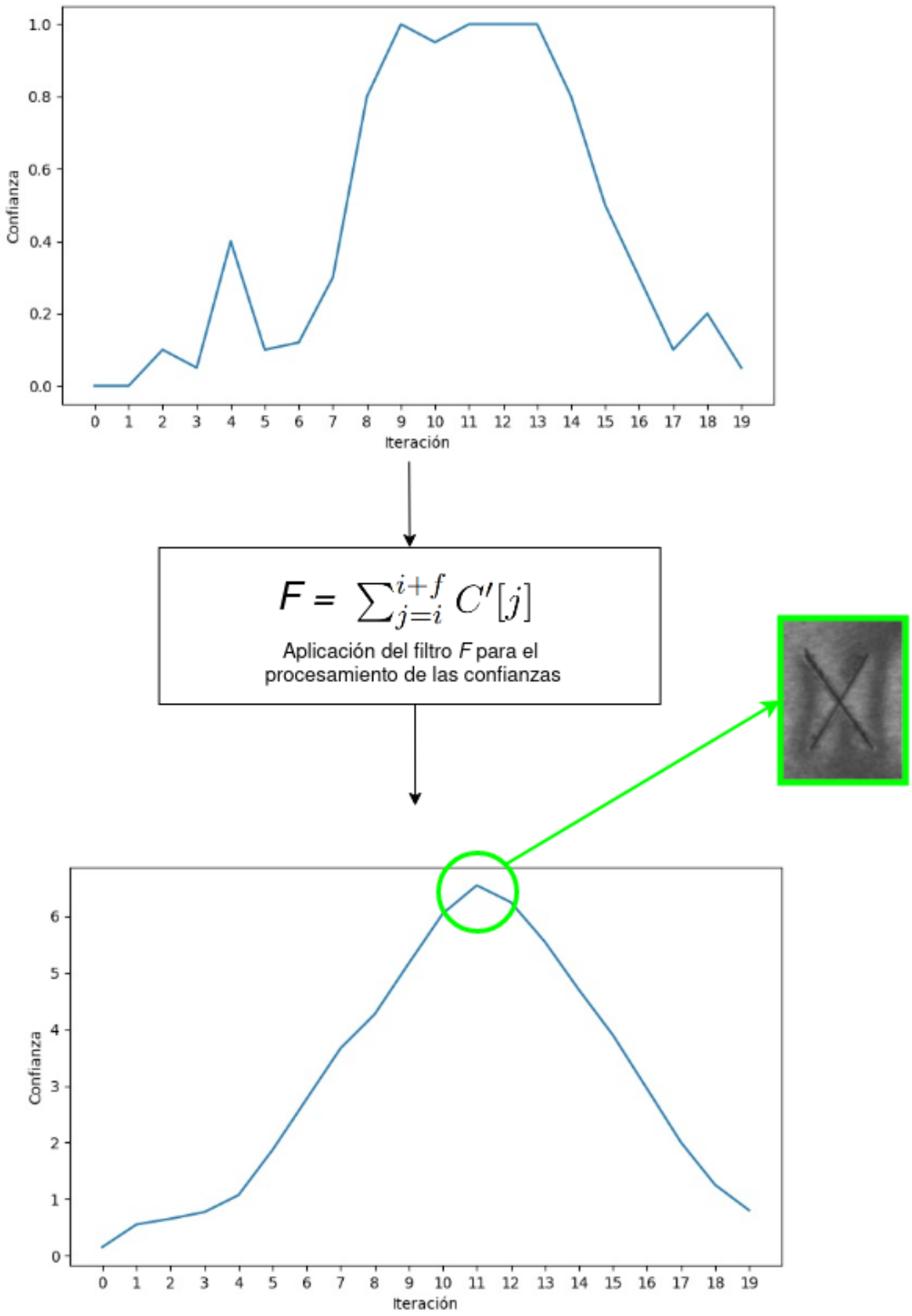

Figura 4.12: Filtrado de confianzas y obtención del recorte más centrado para X.

4.7. Resultados obtenidos por la ventana deslizante

En esta sección se pasan a mostrar algunos de los recortes de códigos obtenidos por el algoritmo YOLOv3 sin realizar ninguna calibración posterior, y los recortes ajustados mediante la ventana deslizante (Figura 4.13). A su vez se muestran los caracteres recortados que se obtendrían si no se realizase esta calibración del recorte del código y los recortes de los caracteres con el mismo código calibrado (Figura 4.18).

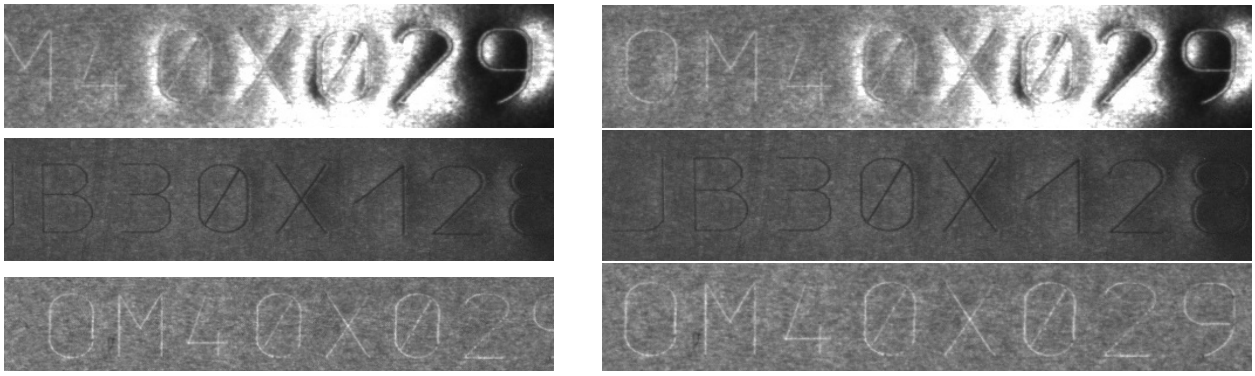

Figura 4.13: Recorte código(izquierda) vs recorte código ventana deslizante(derecha)

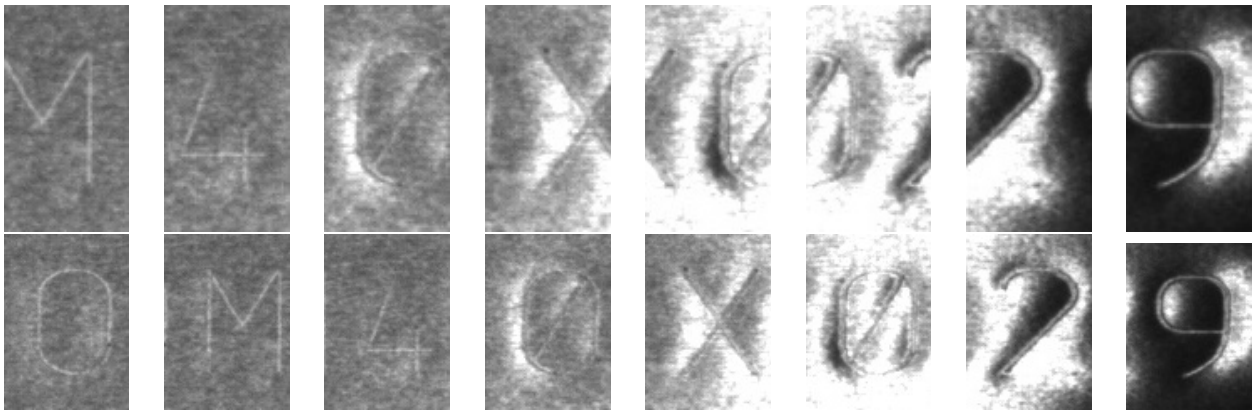

Figura 4.14: Recorte caracteres *raw* (arriba) vs recorte caracteres ventana deslizante (abajo)

Como bien se puede apreciar en la parte derecha de la Figura 4.13, los caracteres están organizados por columnas, pudiendo hacer recortes iguales a lo largo de todo el código y obteniendo los 8 caracteres recortados sin perder información de ninguno de ellos. Por otro lado, se puede ver como aunque el modelo de detección de código lo detecte de manera correcta, las coordenadas que devuelve de este no son precisas. Si sobre estas imágenes se realizasen 8 recortes iguales sobre el código, se obtendría algo parecido a los 8 caracteres de la parte superior de la Figura 4.18. Se observa como hay caracteres cortados, como bien podría ser la X, el 0 o el 2, y por otro lado, no coincide los caracteres con la posición en la que deberían estar. Sin embargo, la parte inferior de esta misma Figura 4.18 muestra como todos los caracteres aparecen completos en su correspondiente recorte, y cada recorte corresponde con la posición en la que debería aparecer.

El cálculo de la precisión de este modelo se ha realizado de forma diferente al resto de fases. Aunque el

funcionamiento de este se basa en el modelo de reconocimiento de dígitos y de este a su vez se conoce el *accuracy*, no se puede extrapolar el mismo debido a que la ventana deslizante busca el recorte más centrado del carácter X, y esa no es una métrica sobre la que se ajustase dicho modelo de reconocimiento. Dado que el algoritmo de ventana deslizante no es un modelo supervisado, tampoco se cuenta con un conjunto de instancias etiquetadas sobre el que comparar y calcular una métrica de *accuracy* en esta tarea.

Se ha decidido, por lo tanto, calcular la precisión de esta fase del modelo de manera manual. Se considera que el algoritmo de ventana deslizante ha conseguido realizar una correcta calibración del código, si en cada uno de los 8 recortes resultantes aparece un carácter completo, sin que este aparezca cortado. Se han analizado un total de 2000 resultados del algoritmo de ventana deslizante, arrojando una precisión de 98.53 % en la correcta calibración de los códigos.

4.8. Fase 3: Reconocimiento de caracteres

Esta es la última fase del modelo D3POCR. Una vez el algoritmo de ventana deslizante ha calibrado las coordenadas del código, se realizan 8 recortes iguales a lo ancho del código, obteniéndose de esta manera los 8 caracteres que conforman el código. Posteriormente se introduce como input cada imagen de cada carácter recortado al modelo CNN que sea oportuno, siendo los dos primeros caracteres para el modelo de reconocimiento de letras y los 5 caracteres restantes para el modelo de reconocimiento de cifras (ya que el caracter X no es necesario clasificarlo). De este modo se obtiene finalmente el reconocimiento del código grabado.

Para el reconocimiento de los caracteres se han implementados dos modelos basados en redes neuronales convolucionales o *CNN*. En ambos casos las redes tienen la misma arquitectura, salvo la capa de salida, que en el caso del modelo de reconocimiento de digitos cuenta con 11 neuronas de salida y en el caso del modelo de reconocimiento de letras cuenta con 26.

La implementación de dos modelos de reconocimiento en lugar de uno tiene principalmente dos motivos bastante relacionados. Como bien se comentaba en el apartado 3.1 hablando acerca de las características del proyecto, se sabe que el código siempre tiene el mismo formato, y por lo tanto se sabe en que posiciones aparecerán cifras y en cual de ellos letras, por lo tanto, surgió la idea de diseñar dos modelos por separado, y aplicar cada uno de ellos a la posición del código correspondiente, aunque esto incurría en multiplicar por dos el tiempo de entrenamiento.

Por otro lado, en un inicio se realizaron pruebas con un modelo global encargado de la clasificación tanto de cifras como de letras, obteniendo resultados peores que los dos modelos por separado. Se pensó que esto podía ser debido a que el *dataset* aun no era lo suficientemente completo como para generar un buen modelo. Sin embargo, una vez se obtuvo un *dataset* lo suficientemente grande, también se entreno el modelo global junto con los dos modelos por separado, obteniendo nuevamente resultados peores. El análisis de este modelo global también se expone en el apartado 4.9.3.

La estructura de ambos modelos se muestran en Las Figuras 4.15 y 4.16. Ambos están basados en redes CNN tradicionales como bien se puede comprobar, realizando la convolución en primer lugar y a continuación un *Max pooling* que reduce a la mitad la salida de dicha convolución. El tamaño de entrada al modelo es de 80x80 píxeles. Después de numerosas pruebas de entrenamiento con estos modelos, se vio la necesidad de al menos 5 capas de convolución en el modelo para obtener unos resultados de precisión aceptables. Como se puede apreciar en las arquitecturas de los modelos hay ciertas capas de convolución que no están seguidas de una capa de *Max pooling*. Esto es debido a que ya que son necesarias al menos 5 capas de convolución y que la entrada es de 80x80, no se puede aplicar un *Max pooling* de reducción a la mitad tras todas las convoluciones.

A su vez, en dichos modelos se han implementado mecanismos para evitar el *overfitting*, como las capas de *Dropout* y capas de *Batch Normalization*. Se podrá apreciar en el apartado 3.8 como la implementación de las capas estos mecanismo evita en gran medida que se produzca dicho *overfitting*.

En cuanto a las funciones de activación, en un inicio se utilizaron funciones de activación *Relu*, sin embargo se pudo comprobar como la utilización de funciones de activación *Leaky Relu* evitaba converger en la situación conocida como *dead Relu*. Esto ocurre cuando hay pesos que pasan a valer 0 cuando se derivan en el proceso de *backpropagation*, quedando inutilizados para el resto del entrenamiento, y por lo tanto, perdiendo el modelo en cuestión capacidad de aprendizaje. Debido a ello se implementaron activaciones *Leaky Relu* como función de activación.

El diseño de ambos modelos fue fruto de un gran número de entrenamientos y de pruebas que se realizaron a lo largo del desarrollo del proyecto. En concreto, tanto el modelo de reconocimiento de cifras como el modelo de reconocimiento de letras que se presentan en las Figuras 4.15 y 4.16 son la versión 17. Esta versión indica el número de *datasets* diferentes que se utilizaron para el desarrollo de todos los modelos de manera iterativa hasta llegar a esta versión 17. Las diferentes versiones de los *datasets* se fueron conformando de manera iterativa, añadiendo nuevas imágenes al *dataset*, modificando los data augmentation que se realizaban, incluso modificando el propio formato de las imágenes que conformaban el *dataset*.

A su vez para cada diferente versión del *dataset* se desarrollaron diferentes versiones de los modelos a entrenar, modificando principalmente su arquitectura hasta que se conseguían unos resultados los cuales parecían óptimos partiendo de la versión del *dataset* con el que se contaba. Realizando este proceso iterativo de modelos se pudo llegar a conseguir precisiones muy elevadas. Este desarrollo de *dataset* y modelos se detallará mejor en el apartado 4.9.1 .

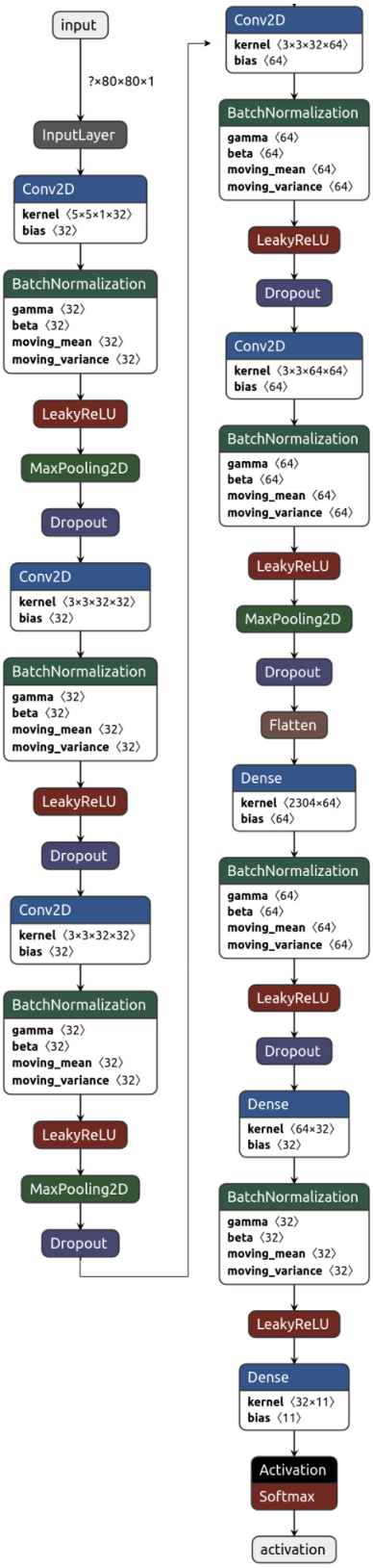

Figura 4.15: Estructura modelo dígitos.

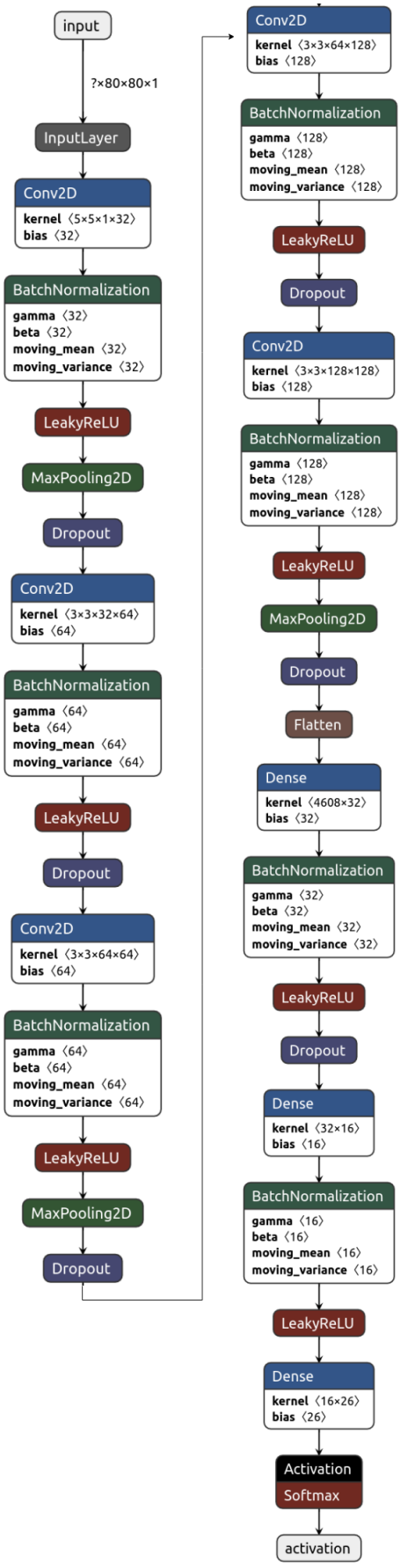

Figura 4.16: Estructura modelo letras.

4.9. *Dataset*, entrenamiento y resultados para el reconocimiento de código

4.9.1. Generación del *Dataset*

Para realizar el entrenamiento de los modelos se han conformado dos conjuntos de datos, uno para el modelo de reconocimiento de cifras y otro para el reconocimiento de letras. Cada uno de estos dos conjuntos a su vez se ha separado en un conjunto de entrenamiento y otro conjunto de test.

La recolección de este conjunto de datos de manera manual era una tarea que conllevaba mucho tiempo de trabajo humano debido a que era necesario en primer lugar realizar el recorte de los códigos, y por otro lado recortar el código en 8 caracteres para posteriormente etiquetarlos para el entrenamiento. Por lo tanto se decidió realizar una primera versión que fuese capaz de recortar los caracteres de manera semi-automática, reduciendo de esta manera la carga de trabajo humana. En primer lugar se obtuvo una versión estable del modelo de detección de código. Posteriormente se recolectó una pequeña parte del *dataset* y se aplicaron técnicas de *transfer learning* para entrenar una primera versión de los modelos de reconocimiento, pero con especial énfasis en el modelo de reconocimiento de cifras, el cual incluía la clase X, pieza clave del algoritmo de ventana deslizante.

Esta primera versión no tenía la suficiente precisión requerida por el proyecto (aproximadamente un 80 %), pero fue suficiente para que el algoritmo de ventana deslizante fuese capaz de calibrar una gran parte de los códigos de manera correcta y de este modo realizar recortes válidos de caracteres de manera automática, evitando la ardua tarea de realizar los recortes a mano. Finalmente estos recortes se analizaron de manera manual para descartar los recortes no válidos y etiquetar los válidos.

Una vez se realizaban pruebas con diferentes arquitecturas de los modelos para entrenarlos sobre estas nuevas versiones del *dataset*, se realizaban nuevas tomas de imágenes reales para aplicar estos nuevos modelos ajustados, volviendo a realizar este etiquetado semi-automático. Este proceso iterativo se realizó un total de 17 veces. Esto quiere decir que se tuvieron un total de 17 versiones diferentes del *dataset*, y para cada una de ellas, un conjunto de pruebas para ajustar las arquitecturas de los modelos.

Mediante este etiquetado semi-automático se redujo en gran medida el tiempo necesario para conformar un *dataset* más grande, y por lo tanto el tiempo de desarrollo de los modelos. En la Figura 4.17 se muestra de manera esquematizada este proceso iterativo de generación de *dataset* y modelos.

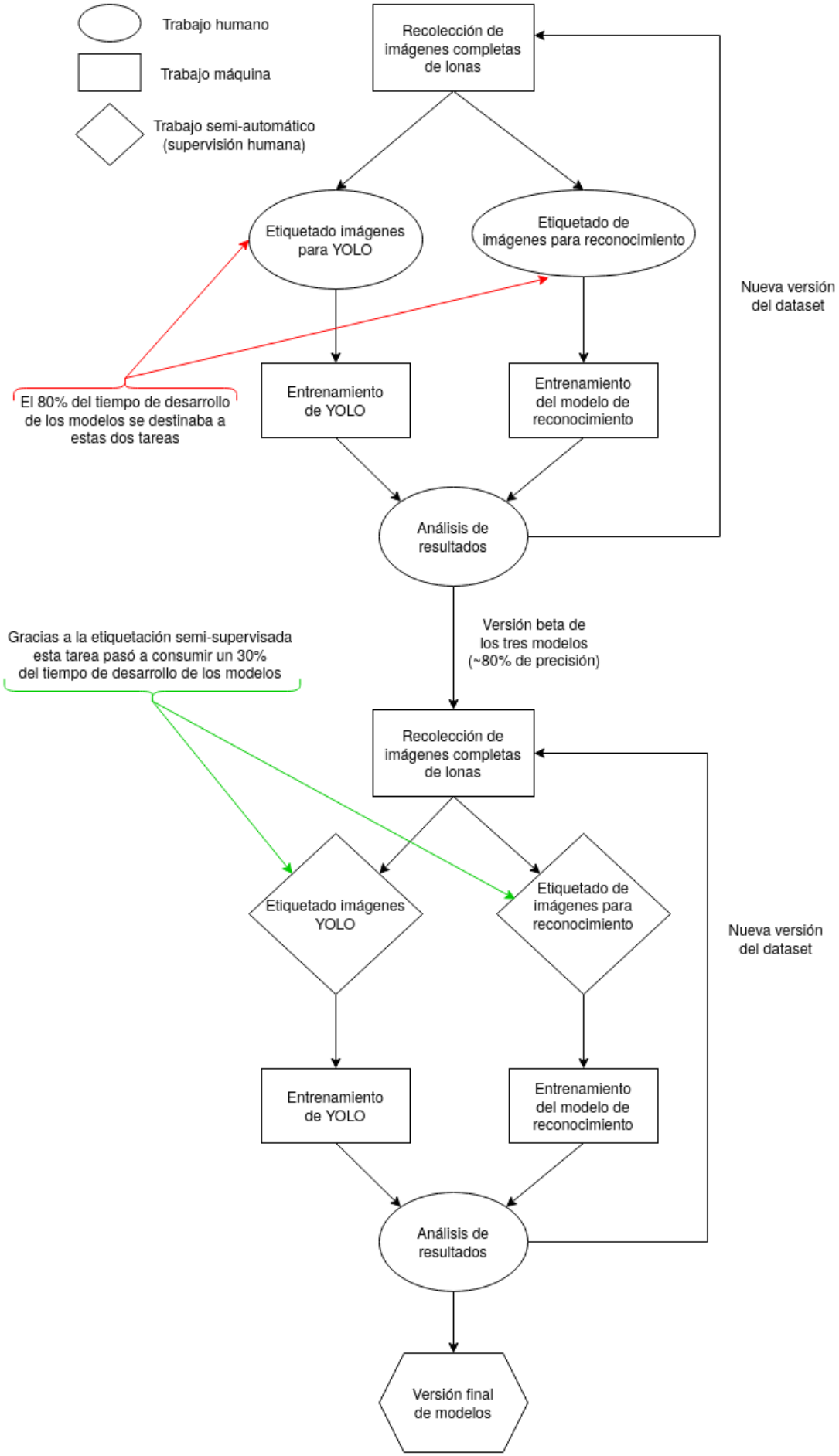

Figura 4.17: Diagrama del desarrollo iterativo de modelos.

A lo largo de este proceso iterativo se implementaron diferentes técnicas de *data-augmentation*. Como ya se ha explicado, la morfología de los caracteres a clasificar es siempre la misma, la única variación que hay en las imágenes es el brillo, contraste, nitidez y calidad del grabado, y en ocasiones, defectos en el aluminio. Sabiendo que la mayoría de las modificaciones en estas imágenes iban a venir por parte de las condiciones de luz, una de las técnicas de *data-augmentation* que se implementaron fue la variación de brillo y contraste en los recortes de los caracteres. A continuación se muestran las transformaciones aplicadas a las imágenes:

- ***Brightness and contrast variation***: Se implementó una función para realizar modificaciones aleatorias en el brillo y contraste de la imagen, modificando los parámetros α y β entre un rango de valores continuos predefinidos. La formula aplicada para las modificaciones de brillo es la siguiente:

$$pixel' = pixel * \alpha + \beta \tag{4.1}$$

- ***Rotation***: Se aplicaron rotaciones a las imágenes de manera aleatoria con un rango continuo de ángulos, desde -10 hasta 10 grados. Si bien es cierto que los códigos no vienen rotados en un principio, puede haber pequeños errores humanos a la hora de la colocación de las lonas de aluminio. Esto no supone una rotación que realmente tenga relevancia a la hora de recortar un código, pero si que puede influir en cierta medida a nivel de los caracteres.

En la siguiente tabla 4.2 se resume el número de instancias en cada uno de estos conjuntos de datos con el *data-augmentation* aplicado. Estos *datasets* están balanceado en relación a instancias/clases. Sin embargo se puede como comprobar como para cada clase del conjunto del *datasets* de letras se cuenta con menos instancias que en el de cifras. Esto es debido a que en cada código se cuenta con 5 números y con 2 letras, y por lo tanto su recolección era más complicada.

Una vez se contaba con los *datasets*, se pasó a realizar los entrenamientos de los modelos y el posterior análisis de resultados. Para un mejor análisis de estos, se mostrará la información de la precisión por clase de cada uno de los tres modelos entrenados.

	Train	**Test**	**Classes**
Cifras	88000	11000	11
Letras	130000	20000	26

Tabla 4.2: Instancias dataset reconocimiento caracteres

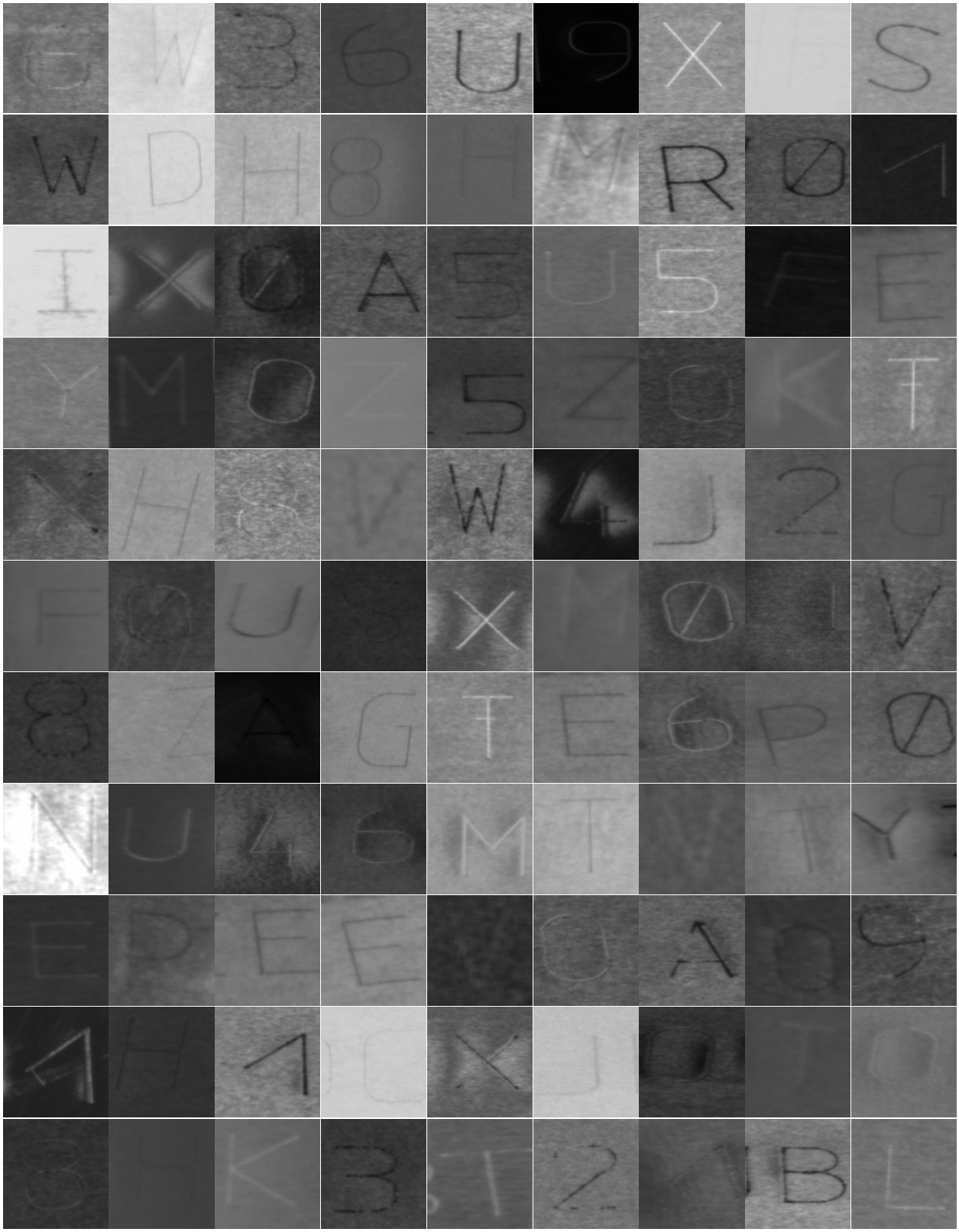

Figura 4.18: Ejemplos de imágenes de ambos *datasets* (letras y cifras)

4.9.2. Entrenamiento

Como se explicaba en el apartado anterior, se realizaron numerosos entrenamientos de modelos hasta llegar a una versión (la cual es la que se muestra en este proyecto) que se consideraba estable. En cuanto al proceso de entrenamiento cabe mencionar una serie de parámetros que pueden resultar interesantes. Ambos modelos se entrenador durante un total de 1000 *epochs*. En el caso del modelo de reconocimiento de cifras esto se traduce en un total de 88 millones de imágenes procesadas, mientras que para el modelo de reconocimiento de letras se procesaron un total de 120 millones. El tamaño de *batch* para el entrenamiento se definió en 128 imágenes, con el optimizador *Adam* y un *learning rate* $= 0{,}0005$.

Para asegurar un resultado óptimo del entrenamiento, se definieron una serie de instrucciones para que el modelo que se obtuviese de dicho entrenamiento no fuese el modelo final tras haber ejecutado 1000 *epochs*, si no que se guardase el modelo que había tenido un menor *loss* y una mayor precisión sobre el conjunto de validación en cualquier momento del entrenamiento. A su vez se implementó una función de parada del entrenamiento si no se mejoraba el resultado de los modelos en las siguientes 75 *epochs*.

Cuando se realizaron las primeras versiones de los *datasets* y de los modelos, se aplicaron técnicas de *transfer learning* ya que se contaba con un reducido número de imágenes. Sin embargo para los modelos finales estos entrenamientos de los modelos partían de 0, con los pesos inicializados de manera aleatoria.

Se cuenta tanto con conjunto de entrenamiento como de test, sin embargo a la hora de realizar el entrenamiento del modelo, y con el fin de analizar el comportamiento de este durante y después de dicho entrenamiento, se separaba un 10 % del conjunto de entrenamiento, el cual serviría como conjunto de validación. En la tabla 4.3 se muestra un resumen de la información del entrenamiento.

En el siguiente apartado se mostrarán y analizarán los resultados del entrenamiento de los dos modelos de reconocimiento y también del modelo global, con las clases tanto de cifras como de letras. De este modo se podrá comparar los resultados obtenidos para cada una de estas dos posibles soluciones.

Epochs	**Batch size**	**Optimizador**	**Learning rate**	**Imágenes entrenamiento**	**Imágenes validación**
1000	128	Adam	0.0005	79200	8800

Tabla 4.3: Configuración del entrenamiento de los modelos de reconocimiento

4.9.3. Resultado del entrenamiento

En primer lugar se van a presentar las gráficas de *accuracy* y *loss* de los tres modelos entrenados, los modelos de reconocimiento de letras y cifras por separado y a su vez del modelo global que engloba ambos. Antes de analizar estas gráficas cabe indicar que tipo de función de *loss* se ha aplicado para el entrenamiento. Debido a que se trata de un problema de multi-clasificación, la función de *loss* seleccionada es la llamada *categorical crossentrophy*. Esta función se encarga de calcular la diferencia entre dos distribuciones de probabilidad. Una de esta distribuciones es la etiqueta real de la imagen que se intenta clasificar, y la otra distribución de probabilidad es la proveniente de la función *softmax*, la cual se definía en el apartado 1.2. Conocidas las dos distribuciones, se puede calcular entonces la diferencia, y por lo tanto conocer el error que se está cometiendo en la clasificación. En la ecuación 4.2 se muestra el cálculo de la función *categorical crossentrophy*. Se define como y a la función de probabilidad real de la imagen a clasificar y como $\hat{y}$ al resultado de la función *softmax*.

$$Categorical\ crossentrophy = \sum_{i=1}^{num\ classes} y_i * log(\hat{y}_i) \tag{4.2}$$

A continuación se muestran las gráficas de *accuracy* y *loss* en las Figuras 4.19, 4.21 y 4.20 para los modelos de reconocimiento de cifras y de letras, y para el modelo que engloba a todas las clases. La primera apreciación que se puede observar es como el modelo de letras tarda aproximadamente el doble de iteraciones para converger que los otros dos modelos. Podemos ver como ninguno de los modelos ha sufrido *overfitting*. Incluso se puede observar como en el entrenamiento del modelo de reconocimiento de cifras y en algunas iteraciones de los otros dos modelos el *accuracy* sobre el conjunto de validación supera al del conjunto de entrenamiento. Esto parece algo contra intuitivo, sin embargo, se implementaron medidas como las capas de *dropout*, cuya finalidad es intentar evitar caer en el *overfitting*. Estas capas anulan una cantidad de pesos predefinida de manera aleatoria, haciendo que dichos pesos no se utilicen en el proceso de entrenamiento. Sin embargo, a la hora de realizar predicciones para medir el *accuracy* y el *loss* sobre el conjunto de validación se usan todos los pesos del modelo, teniendo este una mayor capacidad de clasificación que sobre el conjunto de entrenamiento, y por lo tanto este puede alcanzar unos mejores resultados.

También se puede apreciar como en el modelo global y el modelo de letras son menos estables en los resultados sobre el conjunto de validación que el modelo de reconocimiento de dígitos. Esto puede ser debido a que la muestra por clase en estos dos modelos es de menor tamaño que en el modelo de cifras, y por lo tanto son menos representativas. Por lo tanto, las variaciones que sufran estos modelos en el proceso del entrenamiento puede tener un mayor peso a la hora de calcular las métricas sobre el conjunto de validación.

Los resultados de precisión de los entrenamientos se pueden encontrar en las tablas 4.4, 4.5 y 4.6. Antes de pasar a realizar un análisis de los resultados cabe destacar cómo se utilizarán los modelos para realizar la inferencia.

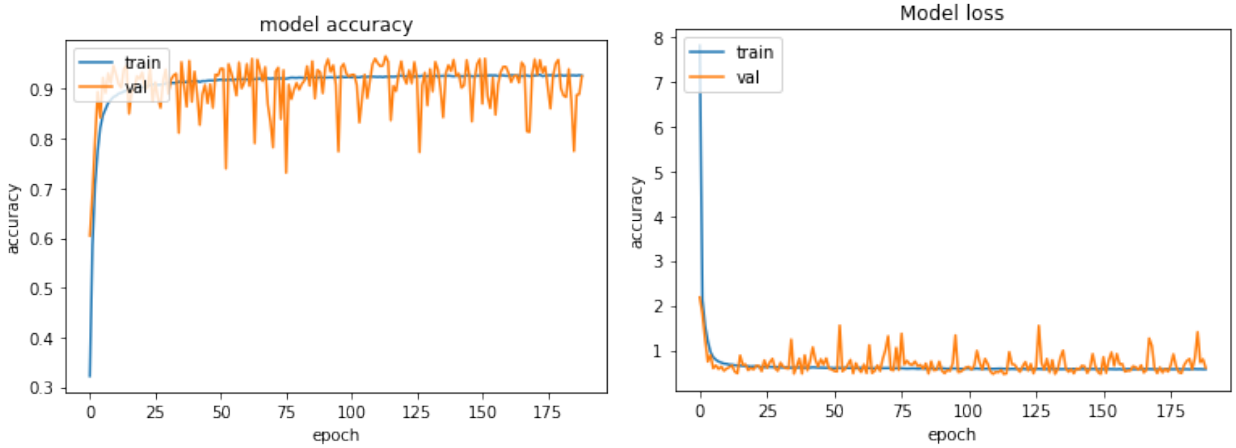

Figura 4.19: Loss y accuracy del modelo de reconocimiento global

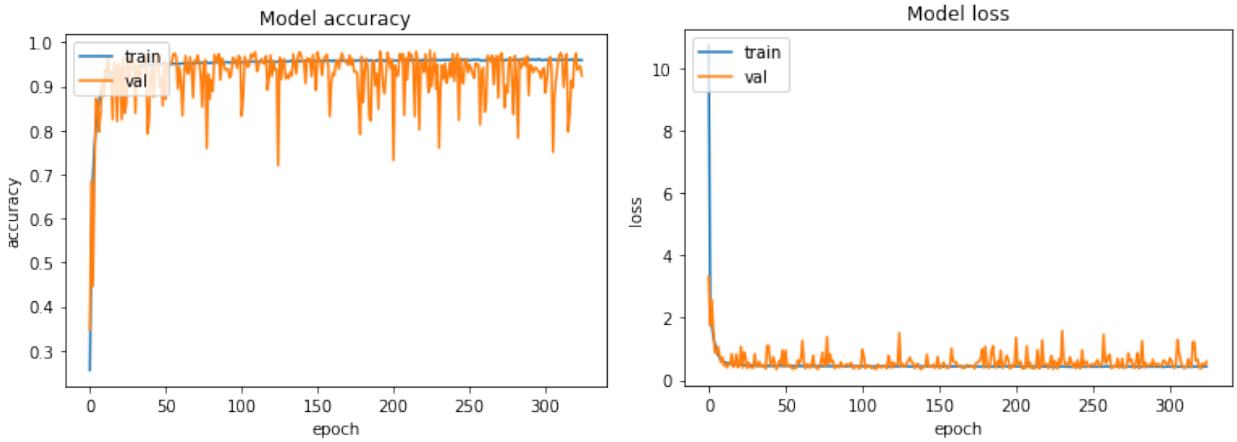

Figura 4.20: Loss y accuracy del modelo de reconocimiento de letras

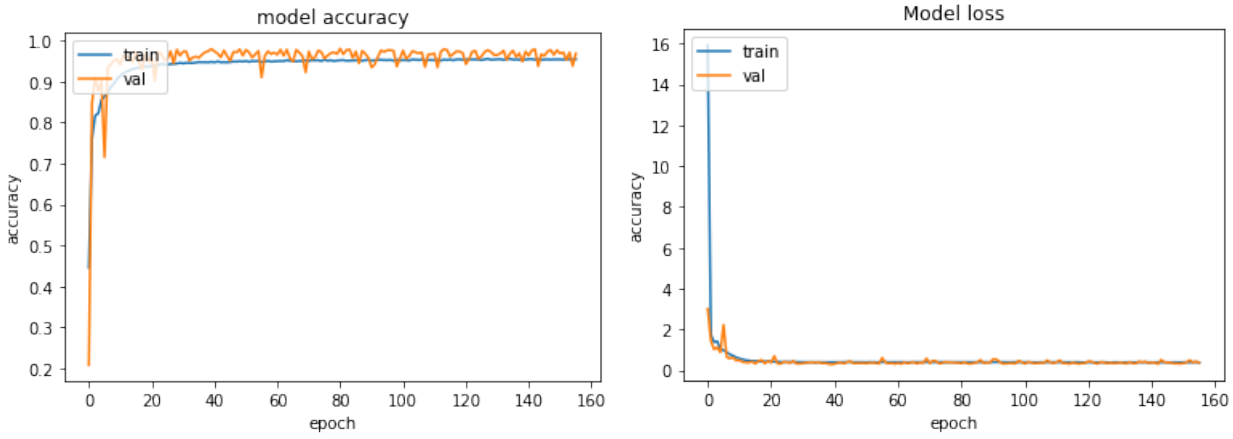

Figura 4.21: Loss y accuracy del modelo de reconocimiento de dígitos

Como bien se comentaba en el apartado 3.1, el código de proyecto siempre tiene el mismo formato, comenzando por dos letras y dos números. Estos indicaban el lugar de destino y el identificador del pro-

yecto. También se hablaba al final de este apartado de la existencia del fichero CSV con la información del proyecto que se iba a cargar. Una vez se obtuvieron los resultados de los modelos y se analizaron, se pudo comprobar como ambos modelos en ocasiones otorgaban confianzas muy cercanas para caracteres parecidos, por ejemplo para las I's las J's y las T's, o para los 1's y los 7's. Ya que se contaba con la información real de la carga que se esperaba realizar de las lonas en el fichero de proyecto CSV, a la hora de realizar los reconocimientos se decidió utilizar una estrategia de *Top-N accuracy* para los 4 primeros caracteres del código, los cuales eran iguales dentro de cada fichero CSV. *Top-N accuracy* hace referencia a que se contabiliza como un acierto que la clase esperada se encuentre entre una de las N confianzas más altas retornadas por el modelo.

Clase	**Top-1 accuracy(%)**	**Top-2 accuracy(%)**
0	99.01	99.23
1	99.62	99.8
2	98.81	99.4
3	97.39	98.83
4	98.6	99.2
5	99.2	99.2
6	99.23	99.97
7	98.06	98.97
8	96.7	98.6
9	97.59	99.0
X	99.43	99.4
Global	**98.5**	**99.18**

Tabla 4.4: Precisión modelo dígitos

De este modo se podían solventar ligeras diferencias en las confianzas del modelo. En el caso de las dos letras se aplicó un *Top-4 accuracy* y en el caso de los dos números un *Top-2 accuracy*. Para los 3 últimos dígitos no se aplicó ningún *Top-N accuracy* y se tomaba como como clase predicha la clase con mayor confianza. Aplicar un *Top-N accuracy* en estos 3 últimos dígitos podría conllevar un número significativo de falsos positivos.

Como bien se adelantaba en el diseño del modelo de reconocimiento de caracteres, el modelo global obtiene una menor precisión que los modelos de reconocimiento por separado en cuanto al *Top-1 accuracy* se refiere, obteniendo un 96.43 % de precisión, frente al 98.19 % que se obtiene de calcular la media de las precisiones obtenidas por los otros dos modelos. Por lo tanto se implementaron los dos modelos de reconocimietno por separado.

Clase	**Top-1 accuracy(%)**	**Top-4 accuracy(%)**
A	94.14	98.74
B	93.04	97.39
C	99.89	99.9
D	98.69	99.89
E	96.52	99.56
F	99.92	99.97
G	97.39	99.13
H	98.69	99.17
I	97.82	99.2
J	99.13	99.9
K	99.56	99.96
L	96.52	97.39
M	96.95	99.89
N	97.82	99.92
O	97.39	98.69
P	98.92	99.9
Q	98.26	99.98
R	96.52	99.89
S	98.26	99.56
T	97.82	98.69
U	98.2	99.79
V	98.13	99.97
W	98.26	99.13
X	99.03	99.9
Y	98.9	99.5
Z	99.13	99.86
Global	**97.88**	**99.41**

Tabla 4.5: Precisión modelo letras

Clase	**Top-1 accuracy(%)**
0	96.2
1	99.6
2	96.6
3	95.39
4	99.2
5	99.0
6	94.39
7	98.41
8	94.39
9	96.8
A	97.07
B	82.17
C	99.56
D	96.52
E	96.95
F	99.55
G	97.82
H	98.26
I	95.65
J	98.26
K	99.5
L	94.34
M	93.47
N	99.56
O	91.3
P	97.39
Q	91.31
R	93.45
S	99.15
T	96.08
U	99.95
V	96.95
W	92.61
X	99.2
Y	99.13
Z	97.39
Global	**96.43**

Tabla 4.6: Precisión modelo global

Capítulo 5

Resultados globales del modelo D3POCR

5.1. Precisión teórica global del modelo D3POCR

Una vez conocida la precisión teórica de cada una de las fases del algoritmo, se puede calcular el *accuracy* global teórico del mismo. En la tabla 5.1 se muestra la nomenclatura que se ha utilizado para definir cada una de las precisiones de las diferentes fases del modelo D3POCR así como el propio valor de las mismas.

Precisión de Fase	**Nomenclatura**	**Valor de precisión/100**
Detección de código	Ad	0.9915
Ventana deslizante	Av	0.9853
Top 4-acc reconocimiento letras	TAl	0.9941
Top 2-acc reconocimiento números	TAn	0.9918
Top 1-acc reconocimiento números	An	0.985

Tabla 5.1: Tabla nomenclaturas de precisiones de modelos

Conociendo los diferentes algoritmos que toman parte en el reconocimiento de código, el formato del mismo y realizando un cálculo sencillo de multiplicación de probabilidades de acierto, se puede extraer la siguiente expresión que da como resultado la precisión teórica del modelo D3POCR:

$$D3POCR\_acc = (Ad * Av * TAl^2 * TAn^2 * An^3) * 100 = \sim 90{,}75\,\% \tag{5.1}$$

5.2. Velocidad de inferencia

Una vez se contaba con el flujo completo de ejecución, se pasó a calcular el tiempo global de inferencia del modelo D3POCR. Para el calculo global del modelo es necesario saber el número de predicciones que realiza cada uno de ellos. En el caso de YOLOv3 se realiza solo una, la cual tiene un tiempo de inferencia aproximado de 40 milisegundos. Por otro lado los modelos de reconocimiento cuentan con un tiempo de inferencia de alrededor de 8 milisegundos, y en este caso tiene que hacer un total de 7 predicciones, obteniendo 56 milisegundos, debido a que es conocida la posición del carácter X y sobre este no es necesario realizar ninguna predicción. Finalmente el algoritmo de ventana deslizante utiliza el modelo de reconocimiento de cifras, que como bien se ha indicado cuenta con un tiempo de inferencia 8 milisegundos, y realiza un total de 20 iteraciones para encontrar el recorte del carácter X más centrado, por lo tanto su tiempo total de inferencia de es aproximadamente 170 milisegundos. En la Tabla 5.2 se muestran los tiempo de inferencia sobre la GPU RTX 2060 de cada una de las fases del modelo por separado medidos en milisegundos.

El tiempo de inferencia global no es siempre constante. Se pueden producir micro diferencias en los tiempos de inferencia debido a que ciertas instrucciones se ejecutan sobre la GPU y otras se ejecutan sobre la CPU, y hay procesos internos del PC que no pueden ser controlados y pueden incurrir en pequeños *delay's*. Estas pequeñas diferencias serían prácticamente imperceptibles si se analiza cada una de las fases del algoritmo por separado, pero cabe destacas que se realizan un total de 28 inferencias por parte de las diferentes fases del modelo D3POCR, y la suma de todos esos posibles *delay's*, aunque sigue siendo muy pequeña, puede incurrir en alguna centésima de segundo. Se estima que el error medio en las medidas de inferencia puede ser de aproximadamente ± 10 milisegundos. Cabe destacar que en estos tiempos de inferencia no se tiene en cuenta procesos ajenos al modelo D3POCR, como por ejemplo el guardado de imágenes de evidencia, ya que estos procesos se ejecutan mediante procesos paralelos y no afectan a la ejecución principal del modelo.

Se puede calcular el tiempo medio de inferencia completa del modelo, el cual es de aproximadamente 270 milisegundos (3.75 lonas/seg). Este tiempo es aproximadamente 8 veces inferior al tiempo que transcurre entre la carga de una lona y la consecutiva, por lo tanto se espera que la implementación de esta solución no incurra en retrasos en el proceso logístico.

Fase	**Latencia (ms)**
Detección de código	~ 40
Ventana deslizante	~ 170
Reconocimiento	~ 56
Total	~ 270

Tabla 5.2: Tiempos de inferencia de cada fase

5.3. Precisión real global del modelo D3POCR

Una vez se había desarrollado el algoritmo completo y se pudo comprobar que el comportamiento de este era estable, se desplegó el modelo completo en las instalaciones del cliente y se pasó a la fase de pruebas. Para esta fase de pruebas se desarrolló una aplicación para los operarios encargados de cargar las lonas. La función básica de la aplicación era ejecutar el algoritmo D3POCR y controlar si este había reconocido correctamente o no los códigos de las lonas. Esta aplicación contaba con un mecanismo que indicaba al operario si una lona se había reconocido mal, generando un mensaje de error con una alerta sonora. Si esto ocurría, el usuario debía insertar de manera manual el código de la lona que se acababa de cargar.

Para conocer el porcentaje de acierto real del algoritmo, se automatizó un proceso de recolección de una sería de métricas y calculaba con ellas el porcentaje de acierto de cada una de las fases del modelo. A su vez, el cliente pidió que se calculase cual era el porcentaje de acierto real del modelo sobre el código completo. En las siguientes Tablas 5.3 y 5.4 se muestra un resumen de los 8 proyectos que se llevaron a cabo en esta fase de pruebas del algoritmo.

Como se puede apreciar en los resultados de la tabla 5.4, estos difieren en 7 puntos percentuales de los resultados de precisión teórica extraídos de cada una de los algoritmos que forman parte del modelo global. Debido a que esta era una diferencia relevante, se decidió analizar ambas precisiones para comprobar si se había cometido algún error. Sin embargo, se pudo comprobar que estos resultados eran lógicos si se atendía a la frecuencia de caracteres que aparecían en proyectos reales. Hay ciertos caracteres que aparecen con mucha más frecuencia que otros en los códigos. Por ejemplo, es mucho más frecuente encontrarse 0's y 1's en los 3 últimos caracteres del código. Si se atiende a los resultados de la tabla 4.4 se puede apreciar que sobre las clases 0 y 1 el modelo tiene más precisión que sobre otras como por ejemplo el 8. Lo mismo ocurre con las letras. Se pudo observar que la letra W o T eran mucho más frecuentes que otras, teniendo estas un *Top-4 accuracy* más elevado.

También hay una relación directa entre la mayor frecuencia de estos caracteres y la precisión del modelo. Cuando se recolectaron los caracteres para conformar los *datasets*, estas frecuencias también se daban, y por lo tanto la variabilidad de datos en estos caracteres era mayor que en otros. Aunque se aplicaron técnicas de *data augmentation*, el obtener la variabilidad de datos reales siempre será más enriquecedor que si esta variabilidad se obtiene mediante data *data augmentation*.

Sin embargo, hay una gran cantidad de variables que pueden afectar a esta diferencia entre la precisión teórica del modelo y la real. La calidad de los grabados y las condiciones de luz afectan en gran medida a la precisión del modelo, aunque este tenga una gran capacidad de generalización.

Atendiendo a este porcentaje de acierto global del modelo se extrajo que, aproximadamente los operarios necesitaban corregir el código reconocido entorno a 3 o 4 veces de cada 100 lonas cargadas. Esto incurría aproximadamente en unos 20 minutos de retraso en una jornada de 8 horas de trabajo, lo cual era una carga temporal asumible de trabajo humano.

Proyecto	**Número Lonas**	**Bien reconocidos**	**Mal reconocidos**	**Códigos no detectados**
SW03	1852	1811	33	8
DV00	41	41	0	0
UR50	27	26	0	1
WT50	499	465	33	1
AW02	774	767	7	0
BH30	1218	1180	35	3
WT70	385	368	16	1
WT80	242	233	8	1
Sum	5038	4891	132	15

Tabla 5.3: Resumen del comportamiento del modelo con proyectos reales

	Acierto (%)
Reconocimiento completo	97.08 %

Tabla 5.4: Acierto global del modelo con proyectos reales

Capítulo 6

Trabajo futuro y posibles mejoras del modelo

Uno de los principales métodos en los que se pensó para mejorar la precisión de los modelos fue la investigación de nuevas técnicas de *data augmentation*. Los modelos GAN (*Generative Adversial Network*) son redes neuronales diseñadas para aprender la distribución que caracteriza al conjunto de entrenamiento, para poder generar nuevos datos. Esto modelos han demostrado un gran potencial a la hora de imitar y generar dichos datos. Una posible vía de mejora de las técnicas de *data augmentation* aplicadas en este proyecto es el entrenamiento de lo que se conoce como CGAN (*Conditional Generative Adversial Network*). Este tipo de modelos, a parte de ser capaces de aprender la distribución de los datos de entrada, también es posible controlar que clase de dicho *dataset* se quiere generar. Por lo tanto, esta sería una alternativa factible frente a las técnicas clásicas de *data augmentation* que se han aplicado en este proyecto.

Por otro lado, el modelo de detección de objetos utilizado es la versión 3 de YOLO. Actualmente hay dos nuevas versiones, v4 y v5, que superan a la versión 3 en cuanto a *accuracy* y velocidad de inferencia se refiere. Si bien es cierto que los resultados obtenidos por YOLOv3 son muy aceptables, se podría aumentar ligeramente la precisión de esta fase del modelo D3POCR actualizando la versión del detector de objetos.

Finalmente, la mayor desventaja del modelo D3POCR es los numerosos pasos intermedios que se ejecutan para poder reconocer el código, lo que conlleva una suma de errores consecutiva que quizás se podrían evitar. Esto ocurre sobre todo a la hora de realizar la clasificación de cada uno de los caracteres del código, ya que se realizan siete predicciones para el reconocimiento de un único código. En el inicio del proyecto se pensó en implementar un modelo de reconocimiento de código en la fase 3 que no recibiese los caracteres uno por uno, si no que directamente recibiese el recorte del código detectado, y lo reconociese con una sola predicción. Con esta posible solución también se evitaría realizar el ajuste de la detección del código mediante el algoritmo de ventana deslizante. Esta solución no se siguió desarrollando, pero sería una mejora muy interesante con respecto al modelo D3POCR.

Bibliografía

[1] Jamshed Memon, Maira Sami, Rizwan Ahmed Khan, and Mueen Uddin. Handwritten optical character recognition (ocr): A comprehensive systematic literature review (slr). *IEEE Access*, 8:142642–142668, 2020.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[3] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In *Machine learning techniques for multimedia*, pages 21–49. Springer, 2008.

[4] Paul W Cooper. Nonsupervised learning in statistical pattern recognition. In *Methodologies of Pattern Recognition*, pages 97–109. Elsevier, 1969.

[5] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.

[6] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[7] Brian D Ripley. Neural networks and related methods for classification. *Journal of the Royal Statistical Society: Series B (Methodological)*, 56(3):409–437, 1994.

[8] Tin-Yau Kwok and Dit-Yan Yeung. Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE transactions on neural networks*, 8(3):630–645, 1997.

[9] Frank Rosenblatt. Perceptron simulation experiments. *Proceedings of the IRE*, 48(3):301–309, 1960.

[10] Peter de B Harrington. Sigmoid transfer functions in backpropagation neural networks. *Analytical Chemistry*, 65(15):2167–2168, 1993.

[11] Barry L Kalman and Stan C Kwasny. Why tanh: choosing a sigmoidal function. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 4, pages 578–581. IEEE, 1992.

[12] Chaity Banerjee, Tathagata Mukherjee, and Eduardo Pasiliao Jr. An empirical study on generalizations of the relu activation function. In *Proceedings of the 2019 ACM Southeast Conference*, pages 164–167, 2019.

[13] Arun Kumar Dubey and Vanita Jain. Comparative study of convolution neural network's relu and leaky-relu activation functions. In *Applications of Computing, Automation and Wireless Systems in Electrical Engineering*, pages 873–880. Springer, 2019.

[14] Rob A Dunne and Norm A Campbell. On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function. In *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, volume 181, page 185. Citeseer, 1997.

[15] Leonardo Noriega. Multilayer perceptron tutorial. *School of Computing. Staffordshire University*, 2005.

[16] Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 479:480, 1969.

[17] MA Kashem, GB Jasmon, Azah Mohamed, and M Moghavvemi. Artificial neural network approach to network reconfiguration for loss minimization in distribution networks. *International Journal of Electrical Power & Energy Systems*, 20(4):247–258, 1998.

[18] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[19] Heng Guo and Saul B Gelfand. Analysis of gradient descent learning algorithms for multilayer feedforward neural networks. In *29th IEEE Conference on Decision and Control*, pages 1751–1756. IEEE, 1990.

[20] K-Y Hsu, H-Y Li, and Demetri Psaltis. Holographic implementation of a fully connected neural network. *Proceedings of the IEEE*, 78(10):1637–1645, 1990.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[22] Ajeet Ram Pathak, Manjusha Pandey, and Siddharth Rautaray. Application of deep learning for object detection. *Procedia computer science*, 132:1706–1717, 2018.

[23] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[24] Stephen M Pizer, E Philip Amburn, John D Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B Zimmerman, and Karel Zuiderveld. Adaptive histogram equalization and its variations. *Computer vision, graphics, and image processing*, 39(3):355–368, 1987.

[25] Hong Yan. Skew correction of document images using interline cross-correlation. *CVGIP: Graphical Models and Image Processing*, 55(6):538–543, 1993.

[26] Jae S Lim and Hamid Nawab. Techniques for speckle noise removal. In *Applications of speckle phenomena*, volume 243, pages 35–45. SPIE, 1980.

[27] Johannes PF D'Haeyer. Gaussian filtering of images: A regularization approach. *Signal Processing*, 18(2):169–181, 1989.

[28] George Jonghwa Yang. *Median filters and their applications to image processing*. PhD thesis, Purdue University, 1980.

[29] Pierre Soille. Erosion and dilation. In *Morphological Image Analysis*, pages 63–103. Springer, 2004.

[30] Djemel Ziou, Salvatore Tabbone, et al. Edge detection techniques-an overview. *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii*, 8:537–559, 1998.

[31] Chenyi Chen, Ming-Yu Liu, Oncel Tuzel, and Jianxiong Xiao. R-cnn for small object detection. In *Asian conference on computer vision*, pages 214–230. Springer, 2016.

[32] John Shawe-Taylor and Nello Cristianini. Machine learning and kernel methods. *American Mathematical Society*, 686:337–404, 1950.

[33] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[34] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[35] Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 3, pages 850–855. IEEE, 2006.