Escuela
Politécnica
Superior

Detección automática de regiones de interés en partituras musicales

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
José Luis Gómez Antón
Tutor/es:
Jorge Calvo Zaragoza

Julio 2020

Universitat d'Alacant
Universidad de Alicante

Detección automática de regiones de interés en partituras musicales

Reconocimiento óptico de música

Autor
José Luis Gómez Antón

Tutor/es
Jorge Calvo Zaragoza
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2020

Preámbulo

En este proyecto se realiza un trabajo de investigación sobre detección de regiones de interés en imágenes de partituras musicales, llegando a obtener una implementación propia básica. Para su desarrollo contamos con una serie de manuscritos proporcionados por el departamento GRFIA de la Universidad de Alicante, los cuales utilizaremos para obtener nuestros propios conjuntos de datos utilizando la herramienta MURET para el etiquetado de los mismos.

Agradecimientos

En primer lugar, me gustaría darle las gracias a mi tutor, Jorge Calvo Zaragoza, por su gran ayuda a lo largo del desarrollo del proyecto y por brindarme la oportunidad de trabajar con él. Por otro lado, me gustaría agradecerles a mis amigos cercanos y a mi familia por todo el apoyo incondicional durante todos estos años.

Los dos días más importantes en tu vida
son el día que naciste
y el día que averiguas el porqué

Mark Twain.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1 Introducción

La notación musical hace referencia a un grupo de sistemas de escritura con los que se puede visualizar una amplia gama de música codificada. Además, es una herramienta esencial para preservar una composición musical, facilitando la permanencia del fenómeno efímero de la música. Funciona de la misma forma que el texto escrito puede ser precursor del discurso hablado.

Por más de 50 años, los investigadores han estado intentando enseñar a los ordenadores a leer notación musical, lo que se refiere como *Optical Music Recognition (OMR)* (Calvo-Zaragoza y cols., 2020), también conocido como Reconocimiento Óptico de Música. Sin embargo, este campo continúa teniendo un difícil acceso para los investigadores, especialmente los que no tienen un trasfondo musical importante, aunque algunos materiales introductorios son accesibles.

Optical Character recognition (OCR) también conocido como Reconocimiento Óptico de Caracteres, es una tecnología la cual ha permitido la automatización del proceso de escritura de textos. En analogía a OCR, el campo de OMR se encarga del proceso de automatización de lectura musical. Sin embargo, mientras los músicos son capaces de leer e interpretar partituras musicales muy complejas incluso en tiempo real, no existe aún un sistema informático que sea capaz de realizar dicho proceso de forma satisfactoria.

1.1 Motivación del proyecto

La música es una parte fundamental de la historia, que ha estado en constante evolución junto a nosotros desde hace siglos. Es por eso que sigue siendo tan importante en la actualidad. Desde mi punto de vista, ha tenido esta capacidad evolutiva, ya que siempre ha tenido cavidad de expresión y capacidad de almacenamiento. Por siglos, los compositores compartían sus obras redactando manuscritos a lápiz y papel. Es por eso que a día de hoy tenemos constancia de estas obras y mantenemos la historia multicultural que la música nos ha proporcionado.

Existen una gran cantidad de manuscritos multiculturales preservados en archivos históricos, los cuales normalmente se transcriben a formato digital para facilitar su acceso y uso. Sin embargo, esta transcripción no se estaba realizando de forma automática, si no que se empleaba de forma manual por profesionales de una forma lenta y tediosa.

Hoy en día, gracias a las técnicas de OMR, las cuales tienen el objetivo de decodificar estos manuscritos y conseguir transcribirlos de forma automática utilizando algoritmos de Machine

Learning (Calvo-Zaragoza y cols., 2020), acelerarían mucho este proceso de transcripción y permitirían un mejor estudio de los profesionales en el campo musical.

El propósito de este proyecto es el de realizar un pequeño workflow con objeto de detectar de forma automática pentagramas en imágenes de manuscritos musicales, que ayuden al estudio en el campo del OMR. Utilizando diferentes algoritmos e intentando entender al máximo posible este campo que engloba el reconocimiento de objetos mediante visión por computador.

1.2 Objetivos

Este proyecto de investigación abarca dos enfoques distintos. Por una parte el estudio e implementación desde cero de un algoritmo de reconocimiento de objetos (YOLO algorithm), basándonos en las explicaciones de la publicación oficial (Redmon y cols., 2015). Por otra parte, el uso de otra tecnología denominada UNet, la cual es una arquitectura neuronal que puede ser utilizada para detección de objetos (Ronneberger y cols., 2015).

Ambos enfoques están destinados para el mismo fin, que es detectar regiones de interés en imágenes. En concreto, se busca la detección de pentagramas en imágenes de partituras musicales.

Para completar este proyecto de investigación, se han intentado desarrollar los siguientes objetivos académicos:

- **Etiquetado del corpus de los manuscritos musicales.** Como primer objetivo, debemos contribuir en el etiquetado de las propias imágenes con la finalidad de obtener un corpus más detallado y amplio con el que poder trabajar y obtener resultados.

- **Investigación, preparación y preprocesado del corpus.** Una de las tareas más importantes en proyectos de Machine Learning para un obtener un buen desempeño final, es el de obtener un corpus limpio y equilibrado con el que poder trabajar, ya que sin él no es posible realizar el proyecto.

- **Diseño de un algoritmo de Machine Learning con capacidad para obtener características de manuscritos.** Se va a implementar un algoritmo con capacidad de reconocimiento de objetos.

- **Experimentación sobre el desarrollo de las arquitecturas utilizadas.** Los experimentos realizados se llevarán a cabo usando un corpus limpio y equilibrado con el objetivo de maximizar la tasa de aciertos y efectividad en nuestra arquitectura.

- **Investigación didáctica sobre el funcionamiento de algoritmos de reconocimiento de objetos.** Se realizará una investigación sobre el funcionamiento de algoritmos de reconocimiento de objetos con la finalidad de conocer su funcionamiento de forma correcta y poder llegara a entender parte o completamente su implementación.

1.3 Estructura del Proyecto

Para una sencilla visualización del proyecto se ha dividido en secciones. A continuación se van a detallar las secciones que cumplimentan el proyecto y qué podemos encontrar en cada uno de ellas:

- Sección 1, Introducción: En este apartado se comentan los fundamentos de nuestro proyecto y qué impacto tienen hoy en día. También describiremos los objetivos que intentaremos lograr del mismo.
- Sección 2, Estado del Arte: Esta sección está dedicada a proporcionar una idea sobre las bases teóricas generales aplicadas a este campo del problema y de qué forma podemos nosotros solventarlas con este proyecto.
- Sección 3, Metodología: Explicamos con detalle cuál ha sido el enfoque aplicado para resolver el problema, así como las técnicas necesarias para llevar a cabo su resolución.
- Sección 4, Tecnologías: Esta sección abarca, todas las tecnologías utilizadas para el desarrollo de nuestro proyecto, así como lenguaje de programación usado y entorno de programación.
- Sección 5, Implementación: Explicaremos con máximo detalles que algoritmos y estructuras hemos implementado en el proyecto, con objeto maximizar los resultados obtenidos.
- Sección 6, Experimentación: En este apartado, mostraremos y explicaremos la configuración y los resultados obtenidos de los experimentos en nuestro análisis global.
- Sección 7, Conclusión: Por último, comentaremos cuales son las conclusiones del proyecto realizado, obtenidas del propio desarrollo del mismo y de algunas ideas futuras para su continuación.

2 Estado del Arte

Para entender mejor el proyecto de investigación, es necesario hacer hincapié en algunos conceptos básicos relacionados directamente con el proyecto para saber de dónde obtenemos dichos resultados y el porqué se implementan de esta forma.

2.1 Introducción

Para empezar vamos a repasar la terminología musical básica requerida para poder entender una partitura musical, aunque como nuestro objetivo es el reconocimiento de pentagramas, los conocimientos son mínimos.

Toda partitura está compuesta por una serie de pentagramas, los cuales están delimitados por cinco líneas horizontales en las que se colocan las notas. Dependiendo de la posición vertical que ocupe esta nota en el pentagrama, decidirá qué tipo de nota es. Como se indica en la siguiente Figura 2.1:

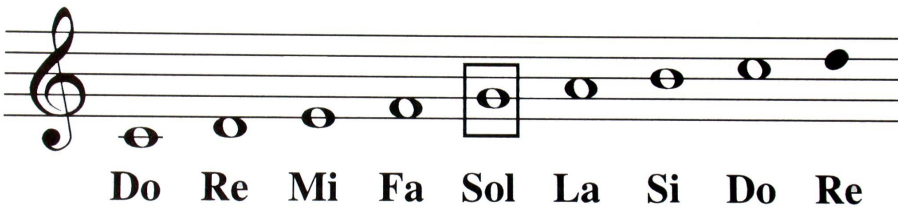


Figura 2.1: Pentagrama de ejemplo

El primer símbolo que podemos observar en el pentagrama nos va a indicar qué tipo de clave musical utiliza el manuscrito, a continuación de esta, encontraremos las notas. Estas pueden ser de diferentes tipos dependiendo del tipo de clave y de la región que ocupe en el pentagrama.

Ahora que ya sabemos que delimita un pentagrama y sabemos qué forma tiene, ya podemos identificarlo. A continuación mostraremos un ejemplo de manuscrito de uno de los corpus [1] utilizados para tener una idea más clara de a qué nos enfrentamos Figura 2.2.

[1] Corpus: conjunto de datos o imágenes empleados en el proyecto.

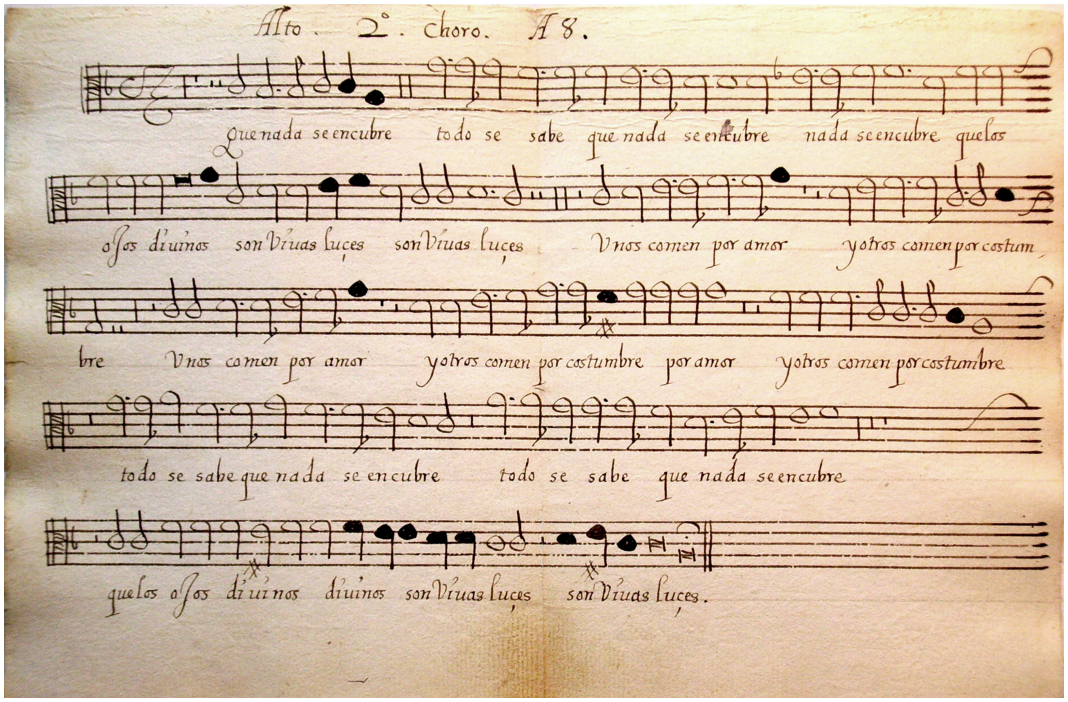


Figura 2.2: Manuscrito del corpus de Capitan

En este manuscrito podemos comprobar que además de los pentagramas, contiene la letra de la canción y en la parte superior aparece el título.

2.2 Inteligencia Artificial

Por extraño que parezca, el término *Inteligencia Artificial* o *IA* (Artificial Intelligence or AI), es una expresión que crece a ritmos exponenciales en boca de las multitudes. Esto no se debe a que la IA se haya desarrollado en el último lustro; es más, el primer intento lo hizo el matemático Alan Turing, que es considerado el padre de la computación en la década de los 50 .

Este científico inglés es más conocido por su máquina de Turing, la cual es una máquina conceptual que utilizó para formalizar los conceptos del modelo computacional que seguimos utilizando hoy en día. Entonces, ¿por qué hoy en día se está hablando más sobre IA? y ¿por qué está cogiendo tanta fuerza?

Esto se debe, a que en la actualidad tenemos la capacidad computacional necesaria para desarrollar aquellos proyectos que en la década de los 50 computacionalmente eran imposibles de realizar.

Como muestra la Ley de Moore, la cual expresa que cada dos años se duplica el número de transistores en un microprocesador, que junto a la salida del procesador multinúcleo explican este avance tecnológico, obviamente sin dejar de lado la existencia de GPUs, TPUs las cuales terminan haciendo hoy en día todos los cálculos vectoriales que optimizan mucho el tiempo de cómputo.

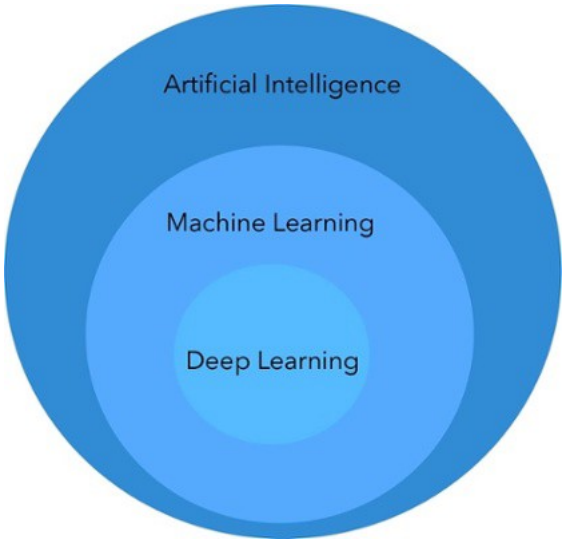


Figura 2.3: Taxonomía Inteligencia Artificial

Como hemos mencionado anteriormente, el término IA o Inteligencia Artificial está en auge, pero ¿qué significa exactamente este término? En términos muy generales es la inteligencia llevada a cabo por máquinas, aunque esta inteligencia está clasificada a su vez por subcategorías dentro de la inteligencia artificial como son *Machine Learning* y *Deep Learning*

que a continuación comentaremos.

Machine Learning (ML) y Deep Learning (DL) también son conocidos como Aprendizaje Automático y Aprendizaje Profundo. Ambas son subcategorías dentro de la Inteligencia Artificial, como se puede apreciar en la **Figura 2.3** (Hrabia, 2020).

Por un lado, Machine Learning engloba la creación de algoritmos, los cuales pueden modificarse a si mismos sin intervención humana para producir un resultado fruto de un entrenamiento con unos datos estructurados. En otras palabras, representa eventos u objetos reales con modelos matemáticos basados en datos. Estos modelos son construidos con algoritmos especiales que adaptan la estructura general del modelo, lo que ajusta los datos de entrenamiento. Dependiendo del tipo de problema a resolver, nosotros definimos algoritmos de Aprendizaje Supervisado o Aprendizaje Sin Supervisión, más conocidos como Supervised Learning y Unsupervised Learning algorithms.

La primera modalidad de aprendizaje que tiene el Machine Learning es la de Supervised Learning. Usándola, se entrena al algoritmo otorgándole un dataset y unas etiquetas. Estos datasets son conjuntos de datos utilizados tanto para entrenar la red como para posteriormente realizar las predicciones. Mientras que las etiquetas, son los valores resultantes del cómputo del algoritmo con el dataset asignado. Estas etiquetas le proporcionan al modelo la capacidad de saber si los resultados son los esperados.

Existen dos tipos de aprendizaje supervisado:

- Regresión: Tiene como resultado un número específico, donde las etiquetas suelen ser un valor numérico. Mediante las variables de las características podemos obtener valores continuos como dato resultante (Figura 2.4).

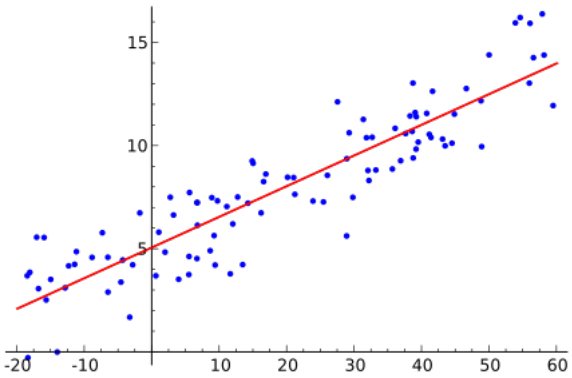


Figura 2.4: Regresión

- Clasificación: En este tipo, el algoritmo encuentra diferentes patrones y tiene por objetivo clasificar los elementos en diferentes grupos (Figura 2.5).

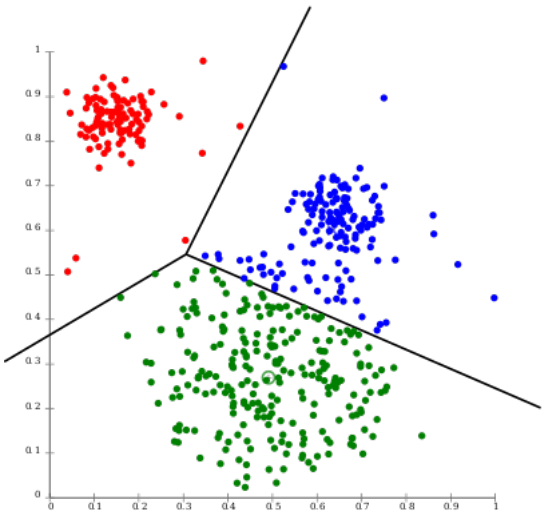


Figura 2.5: Clasificación

A diferencia del Supervised Learning, en el Unsupervised Learning sólo se le otorgan las características, sin proporcionarle al algoritmo ninguna etiqueta. Su función es la agrupación, por lo que el algoritmo debería catalogar por similitud y poder crear grupos, sin tener la capacidad de definir cómo es cada individualidad de cada uno de los integrantes del grupo.

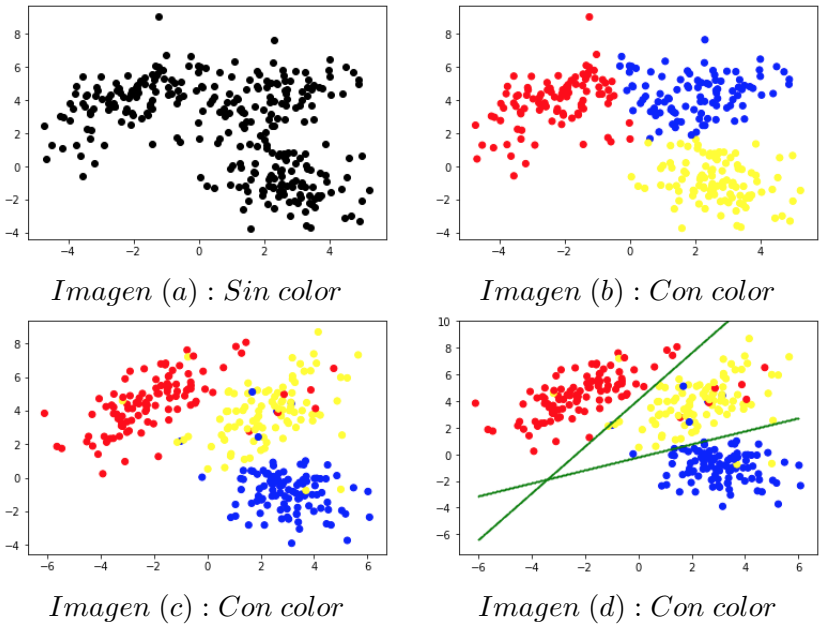


Imagen (a) : Sin color *Imagen (b) : Con color*

Imagen (c) : Con color *Imagen (d) : Con color*

Tabla 2.1: Muestras Unsupervised Learning

Por una parte, la imagen (a) (Tabla 2.1) presenta algunos datos descritos por las coordenadas x e y en el cual a priori no podemos percibir ningún patrón, mientras que la imagen (b) muestra los mismos datos pero coloreados. Este coloreado se debe al uso del el algoritmo K-means para agrupar estas muestras en 3 grupos. El algoritmo K-means es un algoritmo de unsupervised learning.

Por otra parte, la imagen (c) presenta un etiquetado diferente, se puede apreciar como la imagen contiene algunas muestras clasificadas de forma errónea. Mientras que en la imagen (d) utilizando el algoritmo SVM hemos conseguido trazar dos hiperplanos con los que conseguimos clasificar las muestras en 3 grupos. Como hemos comentado anteriormente, esta clasificación no es perfecta, pero es la mejor clasificación que podemos conseguir del conjunto de muestras.

Por último, el Deep Learning, a su vez es otra subcategoría dentro de Machine Learning, donde al igual que en Machine Learning está enfocado a la creación de algoritmos y funciones sólo que está vez hay numerosas capas sobre estos algoritmos. Cada capa proporciona una interpretación diferente de los datos proporcionadas por Artificial Neural Networks (ANN).

Uno de los ejemplos más básicos para su explicación es el perceptrón multicapa, el cual

podemos ver representado en la Figura 2.6. Este perceptrón lo podemos definir como un grafo que hace referencia a la red, la cual está compuesta por nodos (también llamados neuronas) que a su vez están conectadas mediante aristas. Estas aristas están compuestas por valores, más conocidos como pesos. Las neuronas están organizadas por capas, y cada capa de neuronas está conectada con las demás, como vemos en la **Figura 2.6**.

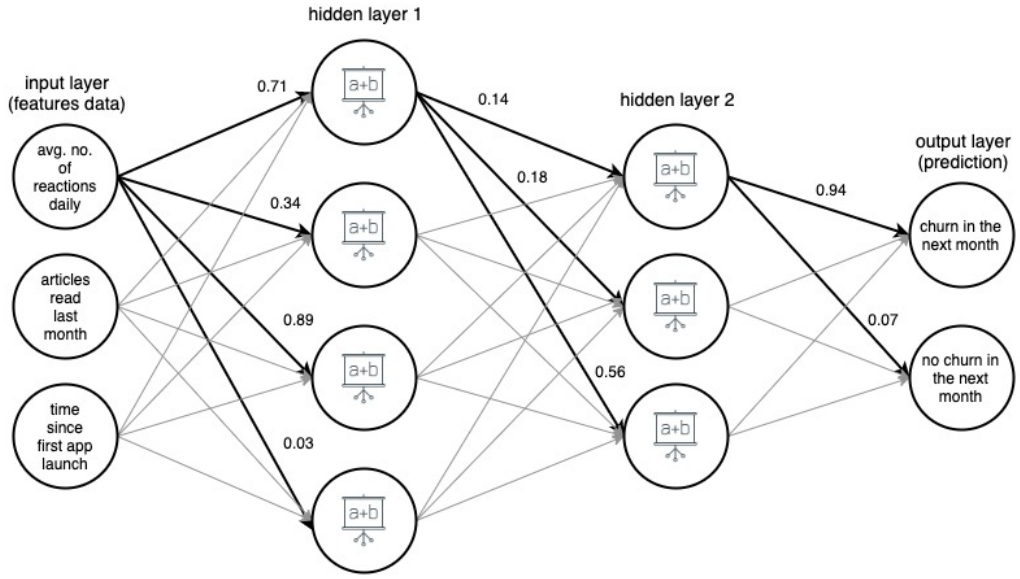


Figura 2.6: Perceptrón Multicapa

En cuanto a los datos, fluyen a través de la red desde la capa de entrada (Input Layer) hasta la capa de salida (Output Layer), siendo estos datos transformados en las neuronas y en las aristas entre ellas.

Cada vez que una muestra del conjunto de entrenamiento pasa a través de la red, lo que se conoce como predicción (salida real de la red), la comparamos con su salida esperada (muestra proporcionada por nosotros como referencia de la salida ideal que debe dar). Una vez comparado este valor adaptamos los parámetros (pesos) del modelo para hacer esa predicción mejor. Esto se consigue con un algoritmo llamado **Backpropagation**. Después de algunas iteraciones, si la estructura del modelo está bien diseñada y tenemos unos buenos datos de entrada y equilibrados (con un mismo número de muestras para cada caso), obtendremos un modelo con buenas predicciones. Esto supone una mayor cantidad de aciertos en los datasets que se le proporcionen posteriormente.

2.3 Redes Neuronales Convolucionales

2.3.1 Introducción a las CNN

En esta sección encontraremos una introducción a redes neuronales convolucionales, también conocidas como CNN (Convolutional Neuronal Network) (Na8, 2018). Las CNN son una de las técnicas más utilizadas para reconocimiento y clasificación de imágenes, detección de objetos, reconocimiento facial y visión por computador, entre otros.

Las CNN son una clase de red neuronal de aprendizaje profundo. Estas CNN tratan de clasificar imágenes de una forma rápida y eficiente. Para realizar dicho proceso de clasificación debemos proporcionarle un conjunto de imágenes o datos y sus respectivos resultados esperados como entrada de la red. A su vez, la salida de la red es una **Clase** (por ejemplo clase "Perro") o una **Probabilidad** de cuán segura está la red de que el conjunto o imagen proporcionada pertenece a dicha clase.

Estas redes procesan sus capas (aplicando convoluciones, pooling , etc. que se explicará más adelante) imitando al córtex visual del ojo humano para identificar distintas características en las entradas que en definitiva hacen que pueda identificar objetos y "ver". Para ello, la CNN contiene varias capas ocultas especializadas y con una jerarquía: esto quiere decir que las primeras capas pueden detectar líneas, curvas y se van especializando hasta llegar a capas más profundas que reconocen formas complejas como un rostro o la silueta de un animal.

Un ejemplo sencillo de CNN, podría diferenciar entre lo que es un Perro y lo que no es un perro, como en el ejemplo siguiente Figura 2.2:

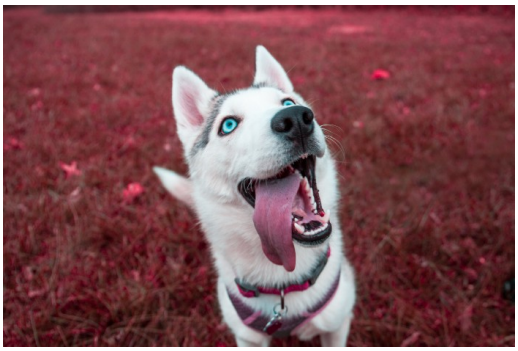


Imagen (a) : *Perro*

Imagen (b) : *No es un Perro*

Tabla 2.2: Reconocimiento de un Perro por CNN

Para entender cómo funcionan estas CNN, primero debemos entender como se representa una imagen por computador.

2.3.2 Visión por Computador y Detección de Objetos

El reconocimiento de objetos es un factor clave en nuestras vidas. Los seres humanos somos capaces de identificar casi cualquier tipo de objeto, incluso siendo este obstruido por otro. Este proceso que nosotros a priori vemos tan sencillo y efectuamos de forma tan intuitiva, en la visión por computador es algo más complejo como veremos a lo largo del proyecto.

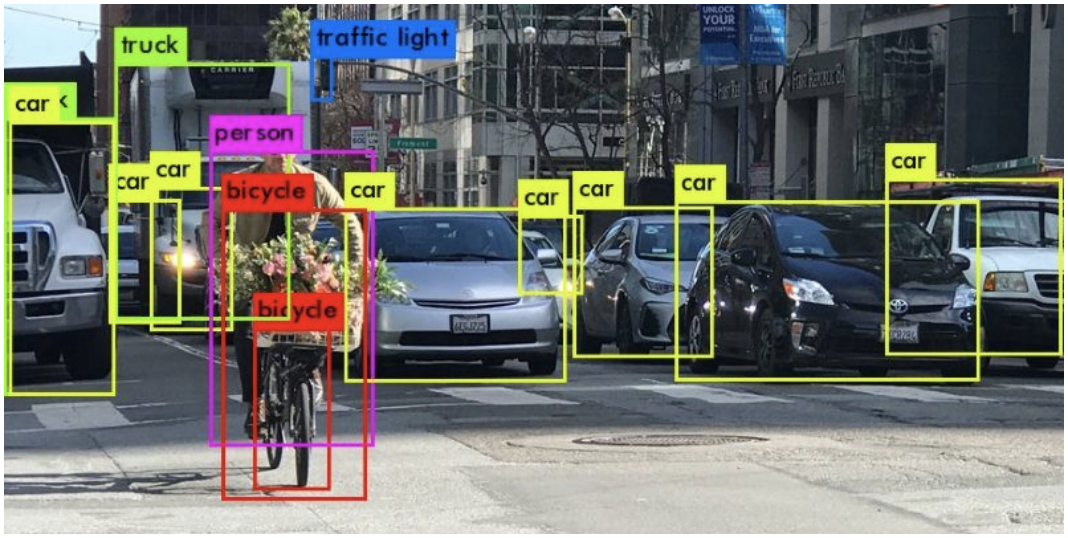


Figura 2.7: Reconocimiento de objetos en una carretera

La visión por computador es la ciencia que permite a las máquinas entender imágenes. Para entender cómo se lleva este proceso a cabo debemos entender primero como se representa una imagen en un ordenador.

Las imágenes digitales están compuestas por una cuadrícula de píxeles, donde el valor de cada píxel representa la intensidad del brillo de la imagen en ese punto. En imágenes en escala de grises este valor es un único valor representado entre [0 - 255], siendo 0 el color blanco y 255 el negro. Este valor también conocido como "umbral" se normaliza para la red neuronal en valores entre [0 - 1]. Como podemos apreciar en la siguiente imagen **Tabla 2.3**).

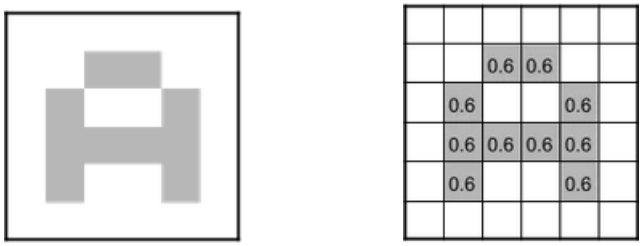


Imagen (a) : Sin Normalizar *Imagen (b) : Normalizada*

Tabla 2.3: Representación de una imagen en escala de grises

Si tenemos una imagen de tamaño 28x28x1 píxels de alto y ancho, siendo el uno final representante del canal de color de la imagen. Es uno puesto que la imagen está representada en escala de grises. Esto equivale a un total de 784 píxeles que ocuparía nuestra imagen. En el caso de que nosotros quisiésemos introducir esta imagen en nuestra red, necesitaríamos 784 neuronas para representar cada uno de estos píxeles.

Por otro lado, Si tuviéramos una imagen a color el valor del umbral [0 - 255] pasaría a ser un vector, donde cada componente del vector a su vez sería un valor entre [0 - 255], el cual hace referencia a la intensidad de cada color en el píxel.

Por ejemplo, en un sistema RGB (RED,GREEN,BLUE) si obtenemos un píxel (255,0,0) el píxel será completamente rojo. Mientras que si vamos disminuyendo ese valor vamos disminuyendo la intensidad total de ese color rojo. Si tuviésemos el mismo caso anterior, pero en vez de tener la imagen representada en escala de grises la tuviésemos representada en formato RGB, tendríamos 28x28x3, siendo el 3 representante del RGB, y necesitaríamos una cantidad de 2352 neuronas para representar dicha imagen.

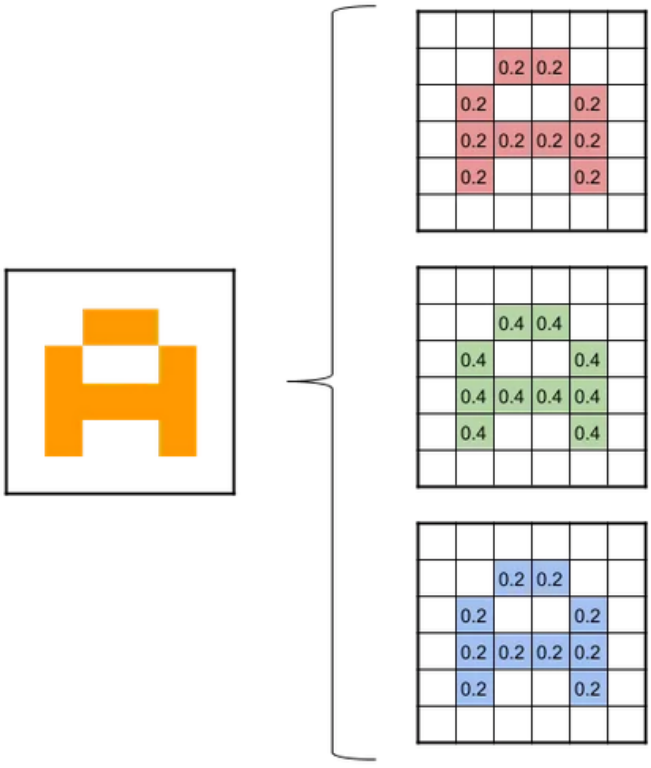


Figura 2.8: Representación imagen a color normalizada

En la imagen superior **Figura 2.8** podemos observar como para obtener la imagen de la izquierda, que tiene un color entre amarillo y naranja, se necesita en una representación RGB normalizada (0.2, 0.4, 0.2).

2.3.3 Funcionamiento de las CNN

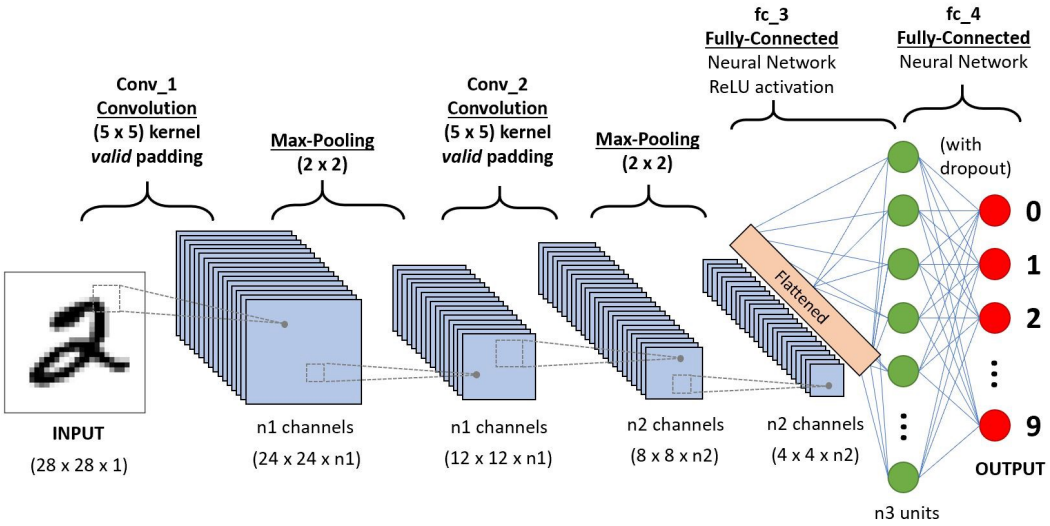


Figura 2.9: Estructura básica de una CNN

Las CNN suelen tener una estructura similar a la que podemos ver en la **Figura 2.9**, donde declaramos la **Arquitectura** que va a presentar el nuevo **Modelo**. Se denomina Arquitectura de la red al conjunto de capas y funciones que posee el modelo desde su **Input Layer (Capa de Entrada)** hasta su **Output Layer (Capa de Salida)**. El modelo es el resultado de aplicar esta arquitectura. Las CNN suelen tener una estructura donde se utilizan varias **Capas Convolucionales**, que suelen ir seguidas de un **Pooling** y para Finalizar una **Capa Densa** también conocida como **Fully Connected**. A continuación procederemos a explicar todas estas capas y cual es el resultado de aplicarlas sobre la imagen de entrada.

Empezando por las capas convolucionales, estas consisten en tomar "grupos de píxeles cercanos" de la imagen de entrada e ir aplicándole el producto escalar contra una matriz pequeña llamada **Kernel (Figura 2.10)** . Supongamos que el Kernel es de tamaño 3x3 píxeles, recorría todas las neuronas de entrada de izquierda a derecha y de arriba a abajo **Tabla 2.4**. Generaría una nueva matriz de salida, la cual sería nuestra nueva capa de neuronas oculta.

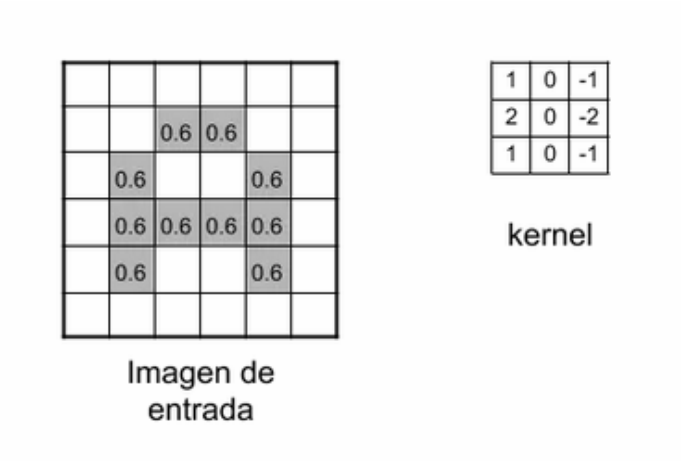


Figura 2.10: Imagen de Entrada a la CNN y Kernel 3x3

El **Kernel** tomará valores aleatorios y se irán ajustando mediante **Backpropagation**. Un **Filtro** es un conjunto de kernels. En realidad, a la imagen de entrada no sólo se le aplica un kernel, si no que se le aplica un filtro, por lo que estamos aplicándole muchos kernels. Por ejemplo, si en la primera convolución tenemos 32 filtros, realmente estamos obteniendo 32 matrices de salida. A este conjunto de matrices de salida se le denomina **Feature Mapping**. Si tenemos la imagen anterior de 28x28 y escala de grises (28x28x1), y le aplicamos estos 32 filtros obtenemos 25.088 neuronas que pasaran a nuestra primera capa oculta de neuronas.

Tabla 2.4: Producto matricial Imagen de entrada y Kernel 3x3

A medida que vamos desplazando el kernel, vamos obteniendo una nueva imagen filtrada por el Kernel. En esta primera convolución y siguiendo el ejemplo anterior es como si obtuviéramos 32 nuevas imágenes filtradas por el Kernel. Estas imágenes nuevas están destacando características de la imagen original. Esto en un futuro ayudará a la distinción de objetos, como es el caso del "Perro" y "No es un perro" anterior.

Existen dos propiedades importantes a la hora de emplear la convolución, que son **stride** y **padding**.

- **Stride**: Hace referencia al número de píxeles que debemos desplazar el filtro sobre la matriz de entrada después de cada operación. Cuando la cantidad es uno entonces trasladamos los filtros realizando pasos de un píxel. En el caso de ser dos trasladaríamos los filtros de dos píxeles en dos píxeles.
- **Padding**: Consiste en rellenar con filas y columnas con valor cero los bordes de la imagen. Esto suele aplicarse bien cuando la matriz de filtros no cuadra exactamente con la matriz de entrada, o cuando no queremos reducir tanto la matriz resultante. Como podemos observar en la **Figura 2.11**

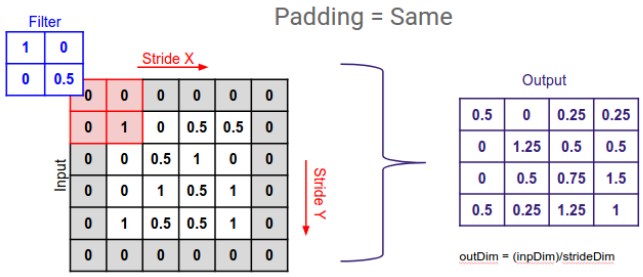


Figura 2.11: Representación de padding en matriz de entrada

En las redes neuronales, las capas de la red activan unos nodos u otros dependiendo de la función de activación que emplee. Explicaremos y nombraremos algunas funciones a continuación:

2.3.4 Funciones de activación

En redes computacionales, la Función de Activación de un nodo define la salida de un nodo dada una entrada o un conjunto de entradas. Se podría decir que un circuito estándar de computador se comporta como una red digital de funciones de activación al activarse como "ON" (1) u "OFF" (0), dependiendo de la entrada. Esto es similar al funcionamiento del Perceptrón Multicapa **Figura 2.6** explicado anteriormente.

La función de activación más utilizada en CNN es la llamada ReLU (Rectifier Linear Unit) (SHARMA, 2017) y consiste en F(x)=max(0,x). A continuación poder ver una representación gráfica de la función:

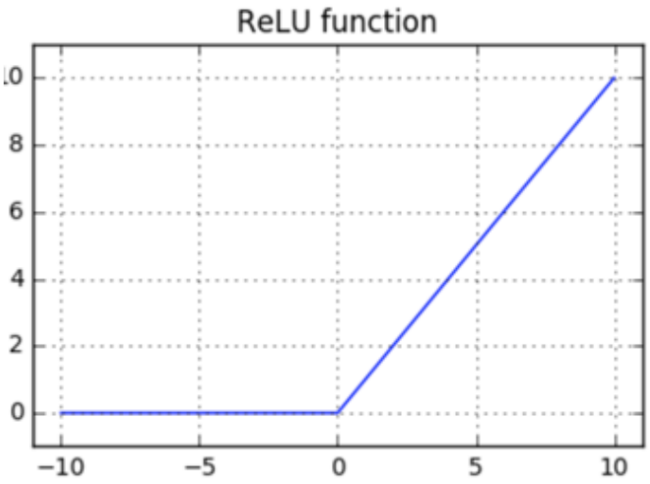


Figura 2.12: Representación ReLU

Existe otra variante de esta ReLU también conocida como LeakyReLU, la cual hemos utilizado en una parte de nuestro proyecto. Su representación gráfica es la siguiente:

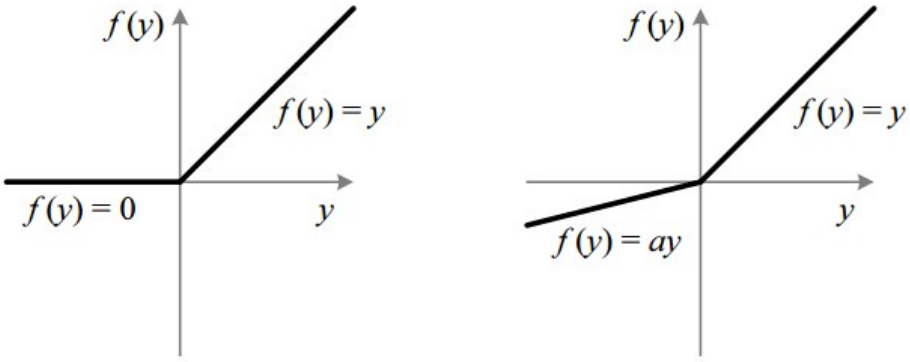


Figura 2.13: Representación ReLU vs leakyReLU

La diferencia entre ambas es que mientras la ReLU tiene un Rango de [0 - Infinito], la leakyReLU abarca desde [-Infinito - Infinito].

Existen muchas funciones de activación, de las cuales cada una de ellas está destinada a un fin en concreto. Algunas de ellas se ilustran en la **Figura 2.14**

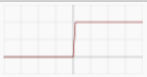
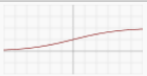
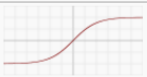
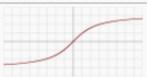
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Figura 2.14: Funciones de activación

2.3.5 Fully Connected Layer

Esta capa es una capa totalmente conectada como su nombre indica. El proceso realizado es similar que realizábamos en la **Figura 2.6**. Antes de aplicar esta capa, aplicamos otra capa denominada **Flatten** para transformar los datos de nuestra matriz en un vector, el cual será la entrada de la capa Fully Connected (Figura 2.15).

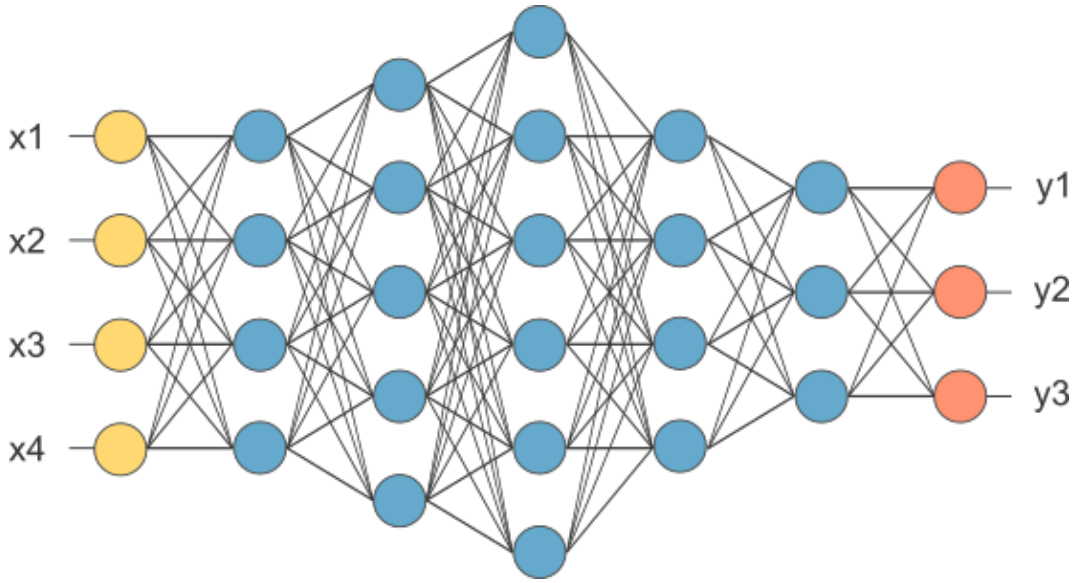


Figura 2.15: Fully Connected Layer representation

En el diagrama superior, encontramos los valores X1,X2,X3,X4 como vector de características resultante de la capa Flatten. En esta capa Fully Connected combinamos estas características para crear el modelo, que finalmente aplicándole una función de activación como softmax o sigmoid clasifica la salida con la que obtenemos una predicción como ”Es un gato”, ”Es un perro”, etc.

2.3.6 Subsampling con Max-Pooling

En esta capa se reducen la cantidad de neuronas antes de hacer una nueva convolución, como vemos en la **Figura 2.6** esta capa suele ir entre convolucionales. Como vimos en el apartado de convoluciones, después de aplicar una convolución a la imagen de entrada obtenemos una cantidad desorbitada de neuronas. Es por eso mismo, que después de una convolucional utilizamos un Subsamplig con la finalidad de reducir este número de neuronas para la siguiente capa, pero aunque se reduzca el tamaño de nuestras imágenes filtradas deberán prevalecer las características más importantes que detectó cada filtro. Si no lo hiciéramos obtendríamos un número de neuronas inmenso, lo que implicaría mayor tiempo de procesamiento.

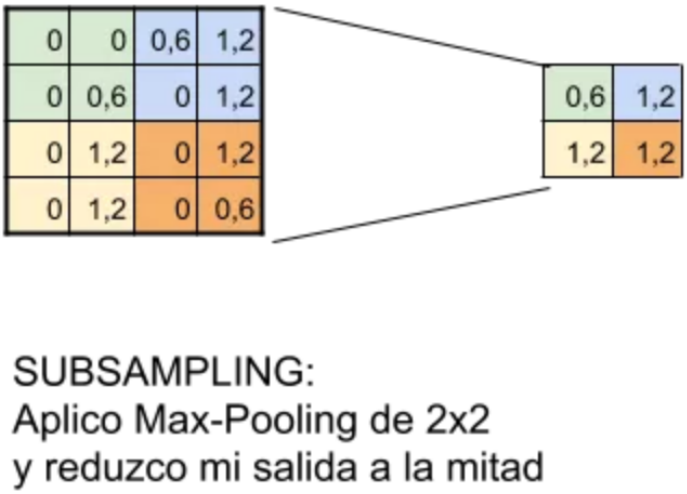


Figura 2.16: Subsampling Max-Pooling

Existen varios tipos de subsamplings, entre ellos el más utilizado para CNN es el Max-Pooling. Si, por ejemplo, utilizamos un Max-Pooling de 2x2 siguiendo el ejemplo anterior con una imagen 28x28 píxeles, quiere decir que vamos a recorrer cada una de nuestras imágenes de características obtenidas anteriormente de izquierda a derecha y de arriba a abajo, pero en vez de tomar un píxel, tomaremos 2x2px (2px de alto, 2px de ancho = 4px) e iremos preservando el valor más alto como podemos apreciar en la **Figura 2.16**. En este caso, aplicar un Max-Pooling de 2x2 implica reducir la cantidad de píxeles a la mitad, obteniendo como resultado 14x14 píxeles.

2.3.7 Funciones de pérdida

Las redes neuronales están entrenadas mediante el uso de un descenso de gradiente estocástico, el cual requiere de una función de pérdida para el diseño y configuración del modelo, con el objetivo de conseguir que el mismo modelo pueda ”entender” los valores de entrada. Existen bastantes funciones de pérdida para elegir y puede ser todo un reto saber cual es

mejor en cada caso.

En contexto, la función de pérdida es un algoritmo de optimización, el cual usa una función para evaluar una posible solución, esta función es denominada **Función objetivo**. Por lo general, nosotros solemos minimizar el error, aunque dependiendo de tu solución final es posible que necesites emplear una maximización.

A continuación comentaremos algunos de los problemas más típicos y qué tipo de función de pérdida se suele emplear:

- Problema de regresión: En este problema intentamos predecir un valor real. Se intenta minimizar el error de la función por lo que usamos **(MSE) Mean Squared Error**.
- Problema de clasificación binaria: Es un problema en el cual, intentamos clasificar un valor entre dos clases. Se suele utilizar una función de pérdida **Binary Cross Entropy**, ya que los valores de entrada se suelen categorizar.
- Problema de clasificación múltiple: Al igual que el anterior se suele usar **Binary Cross Entropy**.

Por último, comentar que cuando el problema al que nos enfrontamos es demasiado concreto, y se necesita de más detalle. Se puede crear una función pérdida propia, no es necesario el uso de una ya predefinida.

2.3.8 Regularización y Entrenamiento

La regularización es una de las partes más importantes a la hora de entrenar nuestro modelo, ya que sin esta, es muy probable que nuestro modelo no obtenga esa capacidad de aprendizaje.

La habilidad de que un modelo sea capaz de predecir correctamente una muestra nunca vista se denomina **Generalización**. Esta generalización es el propósito al que aspiramos cuando empezamos a entrenar dicho modelo, pero nuestros modelos pueden sufrir problemas por las siguientes razones:

- Underfitting: Este problema suele darse cuando nuestro modelo es demasiado simple o las características aprendidas no son suficientemente útiles.
- Overfitting: Este problema es fácil de captar, solemos tener este problema cuando nuestro modelo predice muy bien muestras ya vistas en el entrenamiento, pero no es capaz de predecir de forma correcta (generalizar) nuevas muestras jamás vistas. En este caso, podemos decir que la red está **"Memorizando muestras"** en vez de **"Entendiendo"** características de las mismas.

Para solucionar estos problemas debemos mantener un dataset (muestras de entradas) equivalentes para cada caso. Por ejemplo, en el caso de que queramos diferenciar entre un perro y un gato, debemos tener X muestras de perros y tener X muestras de gatos, intentado

igualar siempre la cantidad de muestras entre conjuntos. Si vemos que esto no es suficiente, y que las muestras dadas son suficientemente buenas como para que funcione, podemos añadir capas **Dropout** estas capas ayudan a la red a no ”Memorizar” las muestras dadas, desactivando algunas neuronas. En el caso de que nuestro dataset no sea lo suficientemente bueno o necesitemos más muestras de las que a priori no tenemos, podemos realizar un proceso de **Aumentado de Datos**. En este proceso, realizamos traslaciones, cambios de brillo, color, contraste, etc. con la finalidad de obtener más muestras diferentes y poder obtener mejores características.

3 Tecnologías

En este apartado, hablaremos sobre las tecnologías usadas para el desarrollo de nuestro proyecto como qué tipo de lenguaje de programación hemos usado, qué tipo de librerías hemos utilizado, entre otros.

3.1 Python

Python es un lenguaje de programación interpretado de alto nivel y multiparadigma, ya que soporta orientación a objetos, programación imperativa y programación funcional. Además se caracteriza por ser un lenguaje interpretado, dinámico y multiplataforma, siguiendo su filosofía característica, la cual es su fácil legibilidad de código. Esta fácil legibilidad del código hace que Python sea muy simple y que su aprendizaje en cuanto a la sintaxis del mismo sea muy sencilla. Por último, Python contiene un montón de librerías que facilitan mucho el desarrollo de proyectos basados en Inteligencia Artificial como Keras, Tensorflow, Scikit-Learn, entre otros. Esto proporciona a Python ser muy competente si lo comparamos con otros lenguajes de programación a la hora de desarrollar un proyecto de Machine Learning, Deep Learning o Inteligencia Artificial en general.

3.1.1 TensorFlow

TensorFlow es una librería de código abierto desarrollada por Google para el desarrollo de proyectos en aprendizaje automático. Esta librería nos brinda la capacidad de construir y entrenar redes neuronales a gran escala y con varias capas. Además, Tensorflow nos permite tanto el uso de CPUs, GPUs con extensión opcional de CUDA (Compute Unified Device Architecture diseñada por NVIDIA) y TPU (Unidad de Procesamiento del Tensor) las cuales son aceleradores de IA programable.

Originalmente, TensorFlow estaba destinada para el uso interno en Google, pero a principios de 2015 se publicó bajo licencia de código abierto Apache 2.0.

3.1.2 Keras

Keras es una API de redes neuronales de código abierto, el cual está escrito en Python por lo que es de alto nivel. Dicha librería es capaz de ejecutarse sobre TensorFlow, Microsoft Cognitive Toolkit o Theano. Sé diseñó para posibilitar la experimentación en poco tiempo con redes de Aprendizaje Profundo o Deep Learning (DL), siendo su parte fuerte la amabilidad para el usuario, modularidad y extensibilidad.

Keras contiene implementaciones de bloques constructivos de las redes neuronales como son capas, funciones objetivos, funciones de activación y optimizadores matemáticos. Por un

lado, nos permite generar modelos de forma secuencial, el cual se caracteriza por la generación de modelos capa tras capa. Por otro lado, podemos generar los modelos de forma funcional, el cual nos permite tener más flexibilidad a la hora de definir modelos con capas compartidas y modelos con múltiples inputs o outputs. Además, tiene soporte para redes neuronales convolucionales las cuales explicaremos más adelante y redes neuronales recurrentes.

3.1.3 NumPY

NumPY es una librería multiplataforma de funciones matemáticas de alto nivel y de código abierto escrita en Python, la cual nos permite operar con vectores y matrices por lo que es fundamental a la hora de desarrollar proyectos basados en computer science en Python. Numpy nos proporciona trabajar con arrays multidimensionales de objetos, aparte de operaciones con vectores y matrices también nos proporciona variedad de rutinas para realizar estas operaciones más rápido en vectores, incluyendo matemáticas, lógica, manipulación de forma, ordenación, selección, E/S, transformaciones de Fourier discretas, álgebra linear básico, operaciones estadísticas básicas, simulación de random y otras cosas.

El núcleo de la librería es el objeto ndarray. Este objeto encapsula las n-dimensiones del array de tipo homogéneo, con algunas operaciones realizadas en código compilado para mejor rendimiento.

3.1.4 Matplotlib

Matplotlib es una librería para generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPY.

3.1.5 Jupyter Notebook

Jupyter Notebook es una aplicación web de código abierto que te permite crear y compartir documentos que contienen código, ecuaciones, visualizaciones y texto. Además, funciona sobre más de 40 lenguajes de programación entre los cuales se encuentran Python y Scala. Es una herramienta potente a la hora de realizar tareas de data cleaning, transformaciones, simulaciones numéricas, modelado estadístico, visualización de datos, Machine Learning y demás.

3.2 Google Colaboratory

Google Colaboratory es un proyecto de Google creado para ayudar tanto a estudiantes como a investigadores en el campo del Machine Learning. Colab nos proporciona un entorno de máquinas virtuales basado en Jupyter Notebooks, en los cuales podemos elegir con qué vamos a ejecutar nuestros programas, pudiendo elegir CPU, GPU e incluso TPU de forma gratuita. Además, el ambiente de trabajo ya viene con muchas librerías instaladas listas para ser utilizadas, como por ejemplo TensorFlow entre ellas, aunque Colab tiene algunas restricciones, como por ejemplo que las sesiones duran hasta 12 horas, una vez pasado este tiempo nuestro entorno se reinicia de forma que perdemos toda la información que habíamos obtenido, tanto

archivos como variables que tuviésemos almacenados en Colab.

Sin duda, es una de las herramientas más utilizadas por estudiantes a la hora de hacer proyectos de Machine Learning y Deep Learning, ya que te brinda esa potencia computacional que necesitas a la hora de hacer proyectos de esta envergadura. Para su uso sólo se requiere tener acceso a una cuenta de Google Drive dónde vamos a poder crear, actualizar, almacenar y compartir los proyectos notebooks que necesitemos.

3.3 OpenCV

OpenCV (Open Source Computer Vision Library) es una librería software de Machine Learning y Visión por computador de código abierto. OpenCV se construyó para proporcionar una infraestructura común para aplicaciones de visión por computador y para acelerar el uso de la percepción por computador en productos comerciales.

La librería contiene más de 2500 algoritmos de optimización, los cuales incluyen un set comprensivo tanto de algoritmos clásicos como de algoritmos de Machine Learning. Estos algoritmos pueden ser usados para detectar y reconocer objetos, clasificar acciones humanas, entre otros.

Por último, OpenCV tiene una versión en C++, Python, Java y Matlab interfaces, además también tiene soporte multiplataforma. OpenCV se inclina más por aplicaciones de visión artificial en tiempo real y aplica ventajas de instrucciones MMX y SSE cuando es posible.

3.4 MuRET

MuRET es una herramienta utilizada para reconocimiento de música, encoding y transcripción. Cubre todas las fases de transcripción desde la fuente del manuscrito hasta el contenido digital del mismo. MuRET fue diseñado para permitir diferentes procesos de aproximación, y producir tanto como los documentos transcritos como los encodings standard y datos para el estudio del proceso de transcripción del mismo.

Usaremos esta herramienta para etiquetar nuestro dataset y descargar el corpus de los manuscritos.

3.5 Github

Github es una plataforma de desarrollo corporativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para crear proyectos abiertos de herramientas y aplicaciones, y se caracteriza sobre todo por sus funciones colaborativas que ayudan a mejorar el código. El código del proyectos que sean abiertos pueden ser descargados y revisados por cualquier usuario, lo que ayuda a mejorar el producto y crear ramificaciones del mismo, aunque también puedes mantener tu privacidad creando un proyecto privado

3.6 VSCode

VSCode (Visual Studio Code) es un editor de código fuente multiplataforma desarrollado por Microsoft. La característica principal de esta aplicación son sus extensiones, existen muchas variedades de extensiones entre las cuales incluyen extensiones para soporte de depuración de varios lenguajes de programación, control integrado de Git, resaltado de sintaxis, finalización inteligente de código en varios lenguajes, fragmentos y refactorización de código.

Por último, mencionar que VSCode es una aplicación de código abierto, y está basada en Electron.

4 Metodología

4.1 Introducción

En esta sección, procederemos a explicar los dos enfoques aplicados por lo cuales se ha resuelto el problema, así como cuál es el problema original. El problema a solventar es la obtención de regiones de interés, más en concreto **la detección de pentagramas**. En el primer enfoque hemos intentado implementar un algoritmo de detección de objetos desde cero basándonos en las bases de YOLO algorithm. Por otra parte, en el segundo repetimos el proceso con el uso de una red denominada UNet para detectar estos pentagramas. Este es el diagrama de flujo del proyecto Figura 4.1

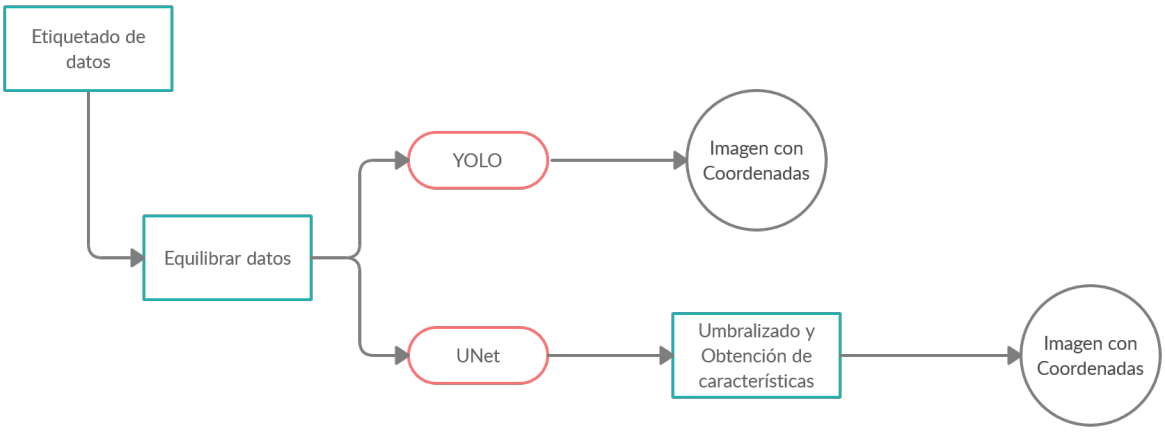


Figura 4.1: Diagrama de Flujo del proyecto

- **Etiquetado de datos.** Dado imagen original del manuscrito obtener las etiquetas necesarias para entrenar la red.
- **Equilibrar datos.** Elaborar una cantidad de datos equilibrada donde haya un equilibrio entre casos de muestras.
- **YOLO.** Enfoque de detección de objetos con YOLO algorithm.
- **UNet.** Enfoque de detección de objetos con UNet, obtenemos como salida de la red un mapa de calor.

- **Umbralizado.** Umbralización de las imágenes obtenidas por la UNet y aplicamos algoritmo de detección de componentes conexas para obtener las coordenadas de los bounding box.

- **Imagen con Coordenadas.** Obtención de pentagramas sobre la imagen original.

4.2 YOLO

YOLO (You Only Look Once)(Menegaz, 2018), es una red neuronal para detección de objetos. El proceso de detección de objetos consiste en determinar las coordenadas en la imagen donde cierto objeto está presente, además de saber de qué clase de objeto se trata Figura 4.2.

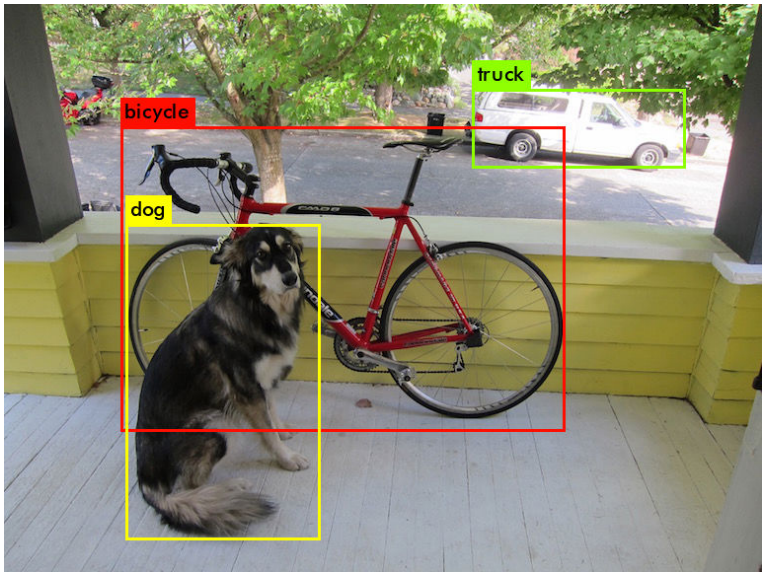


Figura 4.2: Ejemplo de predicción de YOLO

Los métodos previos a este, como las CNN recursivas y sus variaciones usan un pipeline para realizar la tarea en múltiples pasos. Esto puede ser lento de ejecutar y además difícil de optimizar, porque cada componente se entrena de forma individual. Yolo por otra parte consigue hacerlo todo con una red neuronal. ¿Qué consigue esta red? Le proporcionas una imagen a la entrada de la red, de la cual obtienes un **vector con las coordenadas** de la caja que delimitan el objeto y **la clase** a la que pertenece el objeto.

4.2.1 Vector de predicciones

El primer paso para entender YOLO es averiguar cómo codifica su salida, que explicaremos a continuación. La imagen de entrada es dividida por un grid[1] **S x S** donde por cada celda

[1] grid: cuadrícula

del grid se predicen **B** bounding boxes [2] (B predicciones por celda), la confianza para cada bounding box y **C** probabilidades de tipo de clase. Estas predicciones son codificadas por el siguiente tensor:

$$\mathbf{S \times S \times (B * 5 + C)}$$

Por ejemplo si aplicamos S = 7, B = 2, C = 20, nuestro tensor final será 7 x 7 x 30. El proceso empleado es como el de la siguiente Figura 4.3.

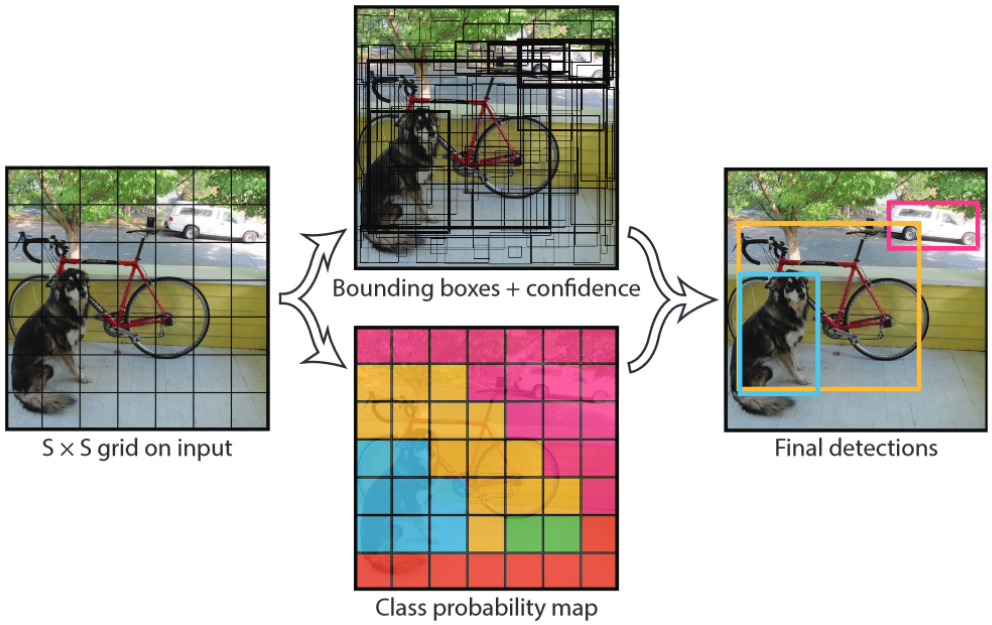


Figura 4.3: Proceso de YOLO

[2]bounding boxes: delimitadores de cajas

La predicción del bounding box tiene 5 componentes (x,y,w,h,confianza). Las coordenadas (x,y) representan el centro de la caja **(centroide)**, relativo a la celda del grid, cabe destacar que si un centroide no cae dentro de la su celda en el grid, esta celda no se hace responsable del centroide. Estas coordenadas están normalizadas entre [0 - 1]. Las coordenadas (w,h), hacen referencia al ancho y alto de las dimensiones de la caja, que también están normalizadas entre [0 - 1] relativo al tamaño de la imagen, como en la siguiente Figura 4.4.

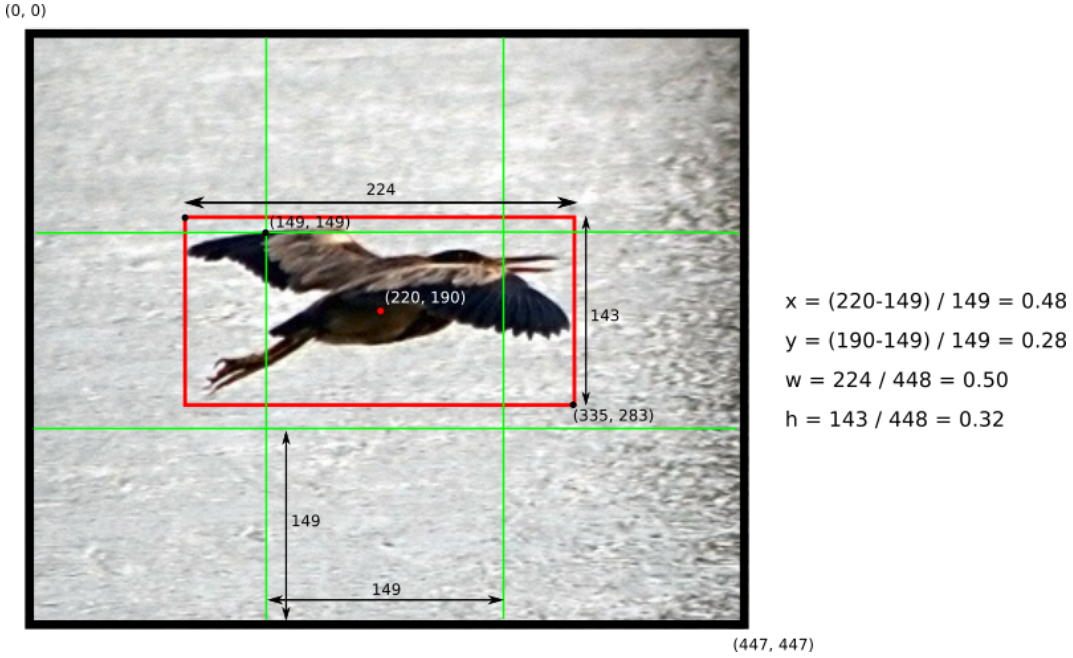


Figura 4.4: Coordenadas X,Y,W,H relativas

Formalmente, definimos la confianza como Probabilidad(Objeto)*IOU(real, esperada) [3] [34]. Por otra parte, nosotros buscamos que el valor de la confianza sea igual al IOU entre la predicción y el valor real (también conocido como ground truth).

[3]IOU: Intersection Over Union [34]

Por último, es necesario entender cómo se predice la probabilidad de la clase Pr(Class(i) | Objeto). Esta probabilidad está condicionada por la celda del grid que contiene ese objeto. Esto quiere decir que si en la celda no se predice ningún objeto, entonces la función de pérdida no penaliza como si fuera una predicción errónea. La red sólo predice un conjunto de probabilidad de clase por celda, independientemente del número de predicciones que hagamos por casilla (parámetro B). Esto quiere decir, que si B = 2, la red intentará encontrar dos predicciones por celda y por cada clase una predicción de tipo de objeto. **Figura 4.5**

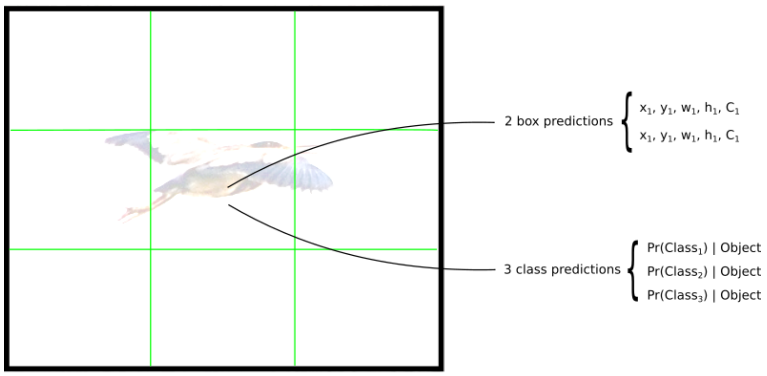


Figura 4.5: Ejemplo de predición de caja y clase con valores (S = 3, B = 2, C = 3)

.

4.2.2 IOU (Intersection Over Union)

IOU es una métrica excelente usada para evaluar detectores de objetos customizados, ya que es muy poco probable que cuando realicemos una predicción el valor del bounding box predecido y el real sean iguales. Para computar la intersección sobre la unión, se determina dividiendo el área sobrepuesta entre el área de la unión Figura 4.6

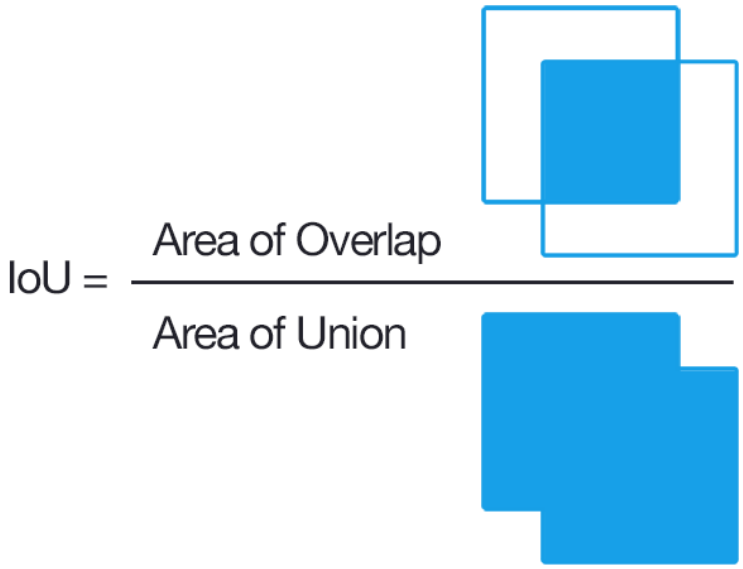


Figura 4.6: Ecuación IOU

Al estar comparando coordenadas es muy poco probable que sean idénticas, por lo que mediante IOU podemos evaluar cómo de buena es la predicción. Cuanto más solape tenga la predicción real con la esperada, mejor valor de IOU tendrá normalizado normalmente este valor entre [0 - 1].

Figura 4.7: Ejemplos de evaluación IOU

4.2.3 Red neuronal

Una vez que entendemos como las predicciones son codificadas, podemos pasar a la estructura de la red. Esta red es similar a una red CNN cotidiana, con sus capas convoluciones, max-poolings, fully connected, etc. Figura 4.8

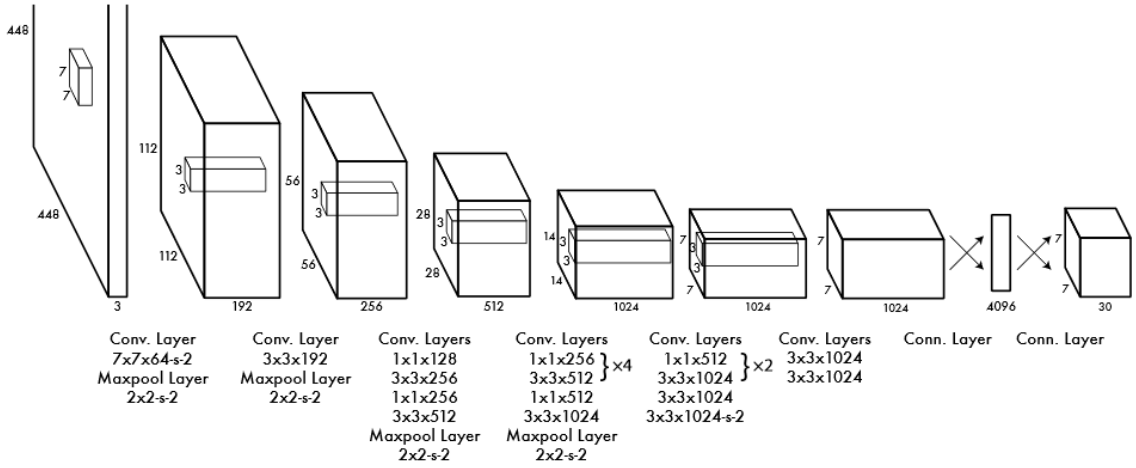


Figura 4.8: Arquitectura de Red YOLO (Redmon y cols., 2015)

Comentar que la imagen recibida como entrada es una imagen de tamaño (448,448), a la cual se le aplican sus convolucionales, etc. También la arquitectura usada para esta imagen contiene los siguientes parámetros S = 7, B = 2. C = 20. Esto explica porque la salida de la red es 7x7x30, ya que (SxSx(B*5+C)) = (7x7x(2*5+20)). Otro dato a destacar es que la red contiene muchas capas, por lo que para entrenarla se van a necesitar una gran cantidad de muestras.

4.2.4 Función de pérdida

La función de pérdida usada no es una función de pérdida común si no que está implementada de forma customizada basándose en 'MSE' (Minimum Square Error). Figura 4.9

$$
\begin{aligned}
&\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}
$$

Figura 4.9: Función de pérdida YOLO (Redmon y cols., 2015)

Para referirnos a la función de pérdida, vamos a dividirla en 5 partes, siendo cada una de ellas una nueva línea.

La primera y segunda parte, computan un 'MSE' sobre (x,y,w,h) predecidos y esperados, la cual sólo se aplica cuando existe un objeto en la celda, en el caso de no haberlo devolvería cero. El valor de **coord** es siempre un valor fijo, el cual es 5, que utiliza para dar más importancia al hecho de encontrar un objeto, frente al hecho de no encontrarlo. Si no se reflejara este proceso, nuestra red podría tener una tasa de acierto muy elevada, pero siempre dando el mismo resultado. Por ejemplo, decir que nunca hay objeto, ya que en la mayoría de casos acertaría 2.3.8.

La tercera y cuarta parte, computan la pérdida asociada al valor de la confianza para cada bounding box predecido. C para el valor de la confianza de la predicción y $\hat{C}$ para confianza obtenida de IOU. El valor de **noobj** es un valor fijo 0.5 como el anterior.

La última parte, parece similar a un error de suma cuadrática normal, excepto por término 1 Obj, el cual no penaliza la clasificación cuando no hay un objeto presente en la celda.

4.3 UNet

4.3.1 Red neuronal UNet

UNet es una arquitectura de red convolucional para segmentación rápida y precisa de imágenes. Este método mejora con creces una red neuronal convolucional (Ronneberger y cols., 2015).

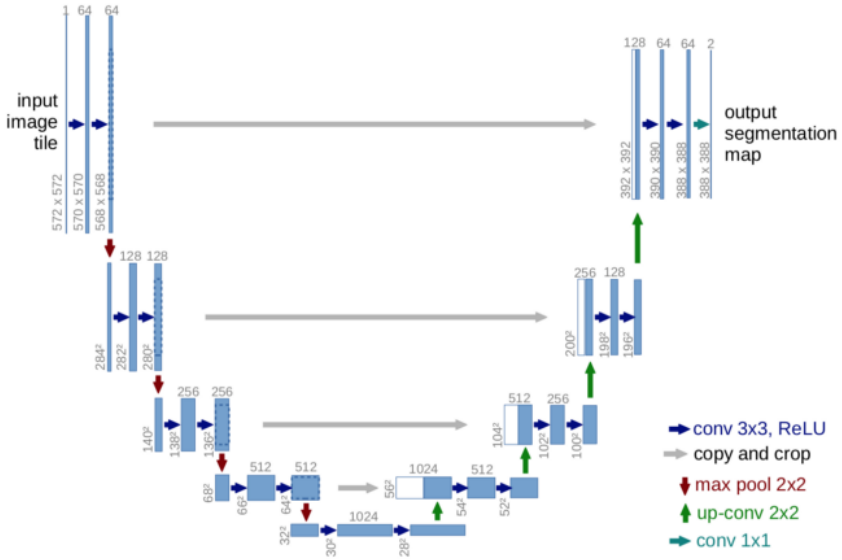


Figura 4.10: Estructura UNet

Esta red (Figura 4.10), a diferencia de las convolucionales que se concentran en la tarea de clasificar, donde la salida suele ser una etiqueta, está diseñada para localizar y distinguir bordes haciendo una clasificación píxel a píxel, en el que la imagen de entrada y la de salida de la red tienen el mismo tamaño. Por ejemplo, si tuviésemos una imagen de entrada 2x2 obtendríamos una imagen de salida 2x2.

A primera vista lo primero que nos impacta al ver la red es que forma una 'U' de la cual deriva su nombre. Su arquitectura es simétrica y consiste en dos grandes procesos. La parte de la izquierda está constituida por procesos de convolución, mientras que la parte de la derecha está constituida por capas de convolución transpuestas contrarias a las de la izquierda (Técnicas parecidas al proceso de upsampling).

La principal idea a la hora de utilizar esta metodología, es la de introducir conjuntos de imágenes con la finalidad de que obtengamos como resultado de la red el mismo tamaño de imagen, pero generando un mapa de calor. La región exenta de pentagramas quedará en negro y la región donde existen pentagramas en blanco. Una vez ya tenemos detectados estos pentagramas, debemos "umbralizar" (thresholding) la imagen para poder aplicar un algoritmo de detección de componentes conexas mediante el cual obtendremos las coordenadas de las cajas, que posteriormente ya podremos dibujar en la imagen original.

Proceso fraccionado en etapas Tabla 4.1

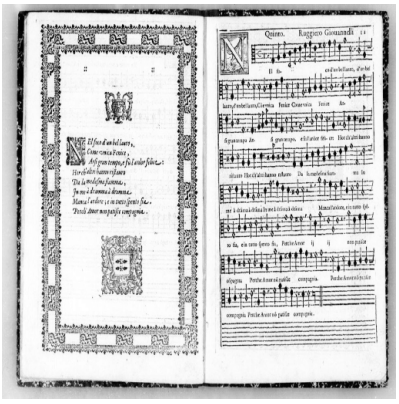


Imagen (a) : Original

Imagen (b) : Predicha por UNet

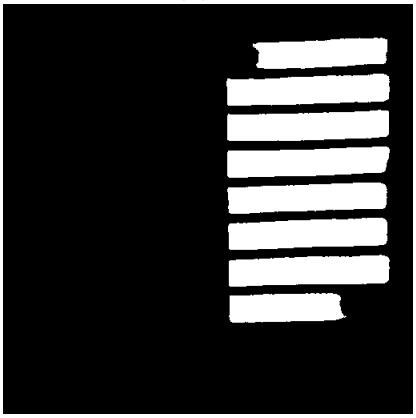


Imagen (c) : Umbralizada

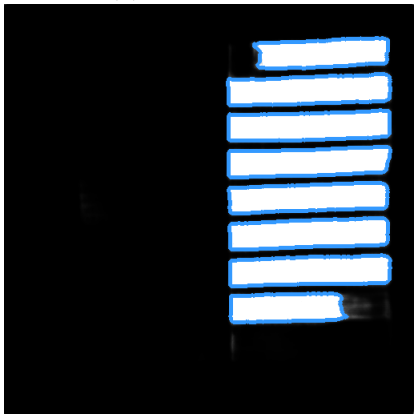


Imagen (d) : Umbralizada con bounding box

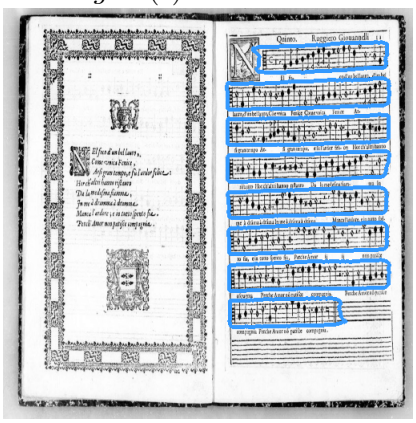


Imagen (e) : Resultado Final

Tabla 4.1: Proceso de detección UNet

5 Implementación

5.1 Introducción

En esta sección, procederemos a explicar con detalle la implementación del proyecto basándonos en la metodología previamente explicada. Para empezar debemos conocer cómo se generan los datos de entrada, los cuales son los mismos para el enfoque de YOLO, como para el enfoque con UNet.

5.2 Manuscritos usados

Los manuscritos usados en el proyecto son cargados mediante un JSON por cada imagen, en el que tenemos las coordenadas de los bounding box de pentagramas, títulos, letra, etc. que previamente hemos obtenido para poder entrenar la red. Los conjuntos de datos que vamos a usar para el proyecto son dos, por un lado tenemos el corpus de Il Lauro Secco y por otro lado el corpus Capitan. A continuación podemos ver un ejemplo de cada uno de ellos.

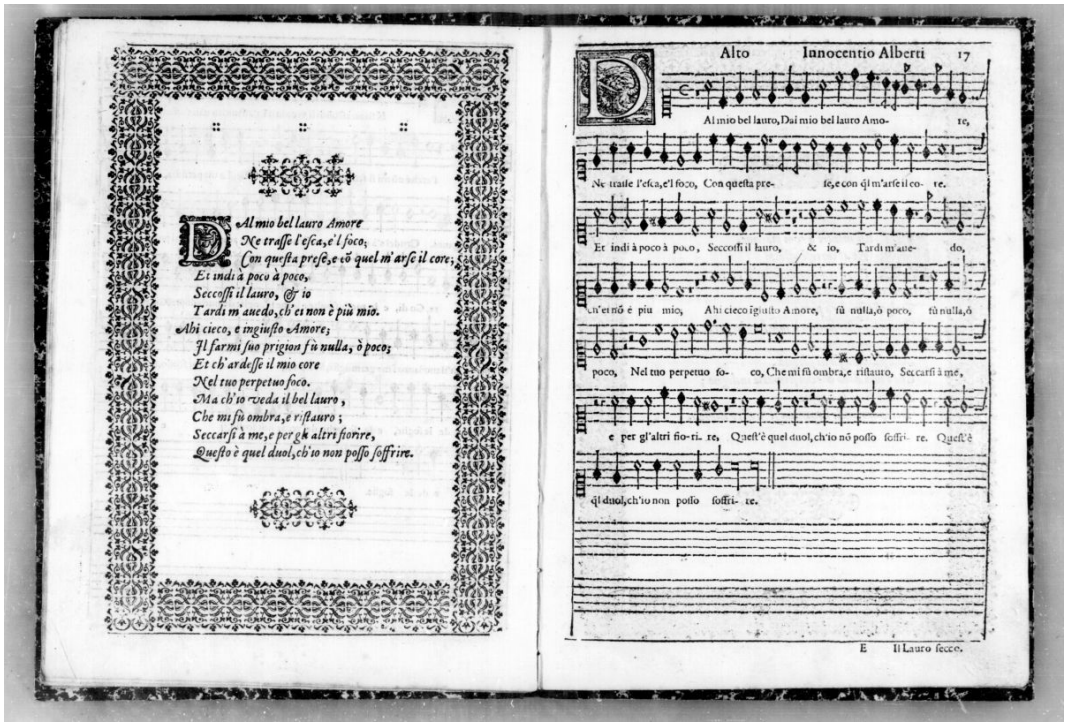


Figura 5.1: Manuscrito del corpus de Il Lauro Seco

Por un lado, en la **Figura 5.1** tenemos un ejemplo del formato de manuscrito que vamos a

obtener del corpus de Il Lauro Seco. Basándonos en nuestros intereses, que para este proyecto es el reconocimiento de pentagramas, podemos apreciar como la parte izquierda de la imagen no nos proporciona ninguna característica interesante para destacar. Mientras que en la parte de la derecha, podemos destacar 9 pentagramas de los cuales algunos se hallan compuestos por notas mientras otros los encontramos vacíos.

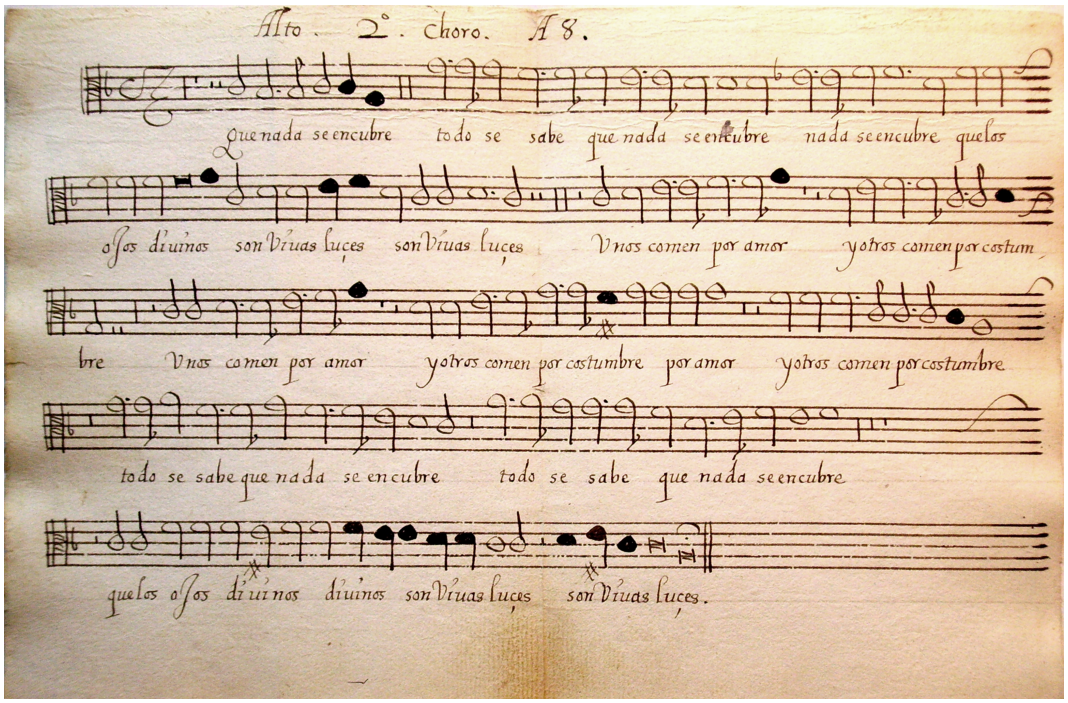


Figura 5.2: Manuscrito del corpus de Capitan

Por otro lado, la **Figura 5.2** nos muestra un ejemplo de manuscrito como los que obtendremos en el corpus de Capitan. Como podemos apreciar estos manuscritos no tienen varias páginas como en el anterior, si no que están compuestos por una única página en la que se hayan los pentagramas con sus respectivas notas, sin dejar cavidad a dibujos, bordados, etc. Únicamente muestran los pentagramas y la letra de la canción, por lo que a priori parece un corpus más sencillo para la obtención de pentagramas, ya que contiene menos elementos.

Una vez mostrados los ejemplos de manuscritos, podemos proceder a explicar la implementación de cada uno de los enfoques, por partes:

5.3 Implementación YOLO

Para el desarrollo de este proyecto se ha invertido gran parte del tiempo, ya que en el ”paper” oficial muestran y explican muy poca información. La mayoría de pasos tomados son meras especulaciones nuestras de cómo se debería implementar. Por lo que, existe muy poco contenido respecto a la creación de este algoritmo.

Para empezar, una vez conocidas todas las bases explicadas en la metodología, nos damos cuenta de que no somos capaces de entrenar la red, ya que no sabemos cómo obtener la salida de la misma sobre unos datos propios. Empezaremos explicando cómo preparar estos datos con los que entrenar la red.

5.3.1 Generación de los datos para entrenamiento

Como entrada de la red vamos a utilizar las muestras de los manuscritos redimensionadas al valor de entrada de la red. Como salida de la misma necesitamos una matriz con [Clase-Objeto , Centroide-X , Centroide-Y , Centroide-W , Centroide-H]. Para ello utilizamos un archivo de configuración que debemos pasarle al proyecto en el cual mostramos:

- gridRowSize. Hace referencia a la cantidad de columnas, por las que se divide la imagen.
- gridColSize. Hace referencia a la cantidad de filas, por las que se divide la imagen.
- predictionPerCell. Cantidad de predicciones que se deben hacer por celda (B)
- labels. Vector donde encontramos todos las clases de objetos que podemos encontrar
- img-size. Tamaño de la imagen
- test-split. Valor para separar el dataset en Training y Test.

Para simplificar un poco el proceso, vamos a mostrar solo la parte en la que se generan estos datos. Una vez ya generados se guardan en un fichero, el cual es pasado a la red. El método **processFile(dirpath, filePath,outFilePath)** es el encargado de generar el resultado de la red.

Código 5.1: Obtención de bounding boxes YOLO

```
def processFile(dirpath, filePath, outFilePath):
  outFile = open(outFilePath, 'w')

  with open(filePath) as JSON_file:
    data = JSON.load(JSON_file)

    img_name = data['filename']
    img = cv2.imread(dirpath + img_name, 0)
    img_h, img_w = img.shape[:2]

    for page in data['pages']:
      for region in page['regions']:
        if region['type'] == 'staff':
```

```
        w = region["bounding_box"]["toX"] - region["bounding_box"]["fromX"]
        h = region["bounding_box"]["toY"] - region["bounding_box"]["fromY"]
        cx = region["bounding_box"]["fromX"] + (w / 2)
        cy = region["bounding_box"]["fromY"] + (h / 2)

        cx, cy, w, h = normalizeData(cx, cy, w, h, img_w, img_h)

        outFile.write('staff ')
        outFile.write(str(cx) + ' ')
        outFile.write(str(cy) + ' ')
        outFile.write(str(w) + ' ')
        outFile.write(str(h) + '\n')

outFile.close()
```

Para empezar, creamos un fichero con el nombre que tenga el parámetro "outFilePath" y abrimos el JSON donde están las etiquetas de manuscritos. Con los bucles vamos accediendo a las posiciones del JSON, donde vamos obteniendo los valores(x,y,w,h) para posteriormente generar los centroides y guardarlos. Cabe destacar que en el tipo de clase está colocado de forma estática, ya que en nuestro problema sólo vamos a encontrar pentagramas. Una vez ejecutado este método obtenemos un fichero de salida parecido a este (Fichero mostrado con sólo son dos iteraciones):

staff 0.5004703668861712 0.07459677419354839 1.0 0.06854838709677419
staff 0.5004703668861712 0.1664986559139785 1.0 0.07560483870967742

Con este fichero y los manuscritos originales, ya podemos cargar los datos de entrenamiento con el siguiente método:

Código 5.2: Carga de imágenes YOLO

```
def loadData(directory, dr):
  X = []
  Y = []

  for filename in os.listdir(directory):
    print(filename)

    img_path = filename.split(sep='.data')[0]
    print(img_path)
    img = cv2.imread("data/" + img_path)
    print(img.shape)
    img = cv2.resize(img, (448,448))
    img = img[:,:,::-1] # Change image from BGR to RGB

    X.append(img)

    Y.append(dr.processFile(directory + filename))

```

```
    plt.imshow(img)
    plt.show()

  return X, Y

X, Y = loadData("processedData/", dr)
```

En este método generamos dos vectores, con los cuales vamos a alimentar posteriormente la red. Además cambiamos el formato de RGB a BGR porque openCV trabaja con un formato diferente, y aplicaremos un redimensionado a las imágenes para que encaje con la entrada de la red. El método dr.processFile(directory + filename) es el encargado de generar la matriz con los valores resultantes obtenidos del fichero anterior, ya que la salida de la red YOLO es una matriz compuesta de (x,y,w,h,C), por lo que para comparar estas soluciones debemos tener una matriz del mismo tamaño. Este es el método que va generando la matriz:

Código 5.3: Generación de matriz resultante de la red

```
def processFile(self, filePath):
    print("Procesing file " + filePath + "...")

    lines = open(filePath, 'r').readlines()

    gridRowSize = self.confParams["gridRowSize"]
    gridColSize = self.confParams["gridColSize"]
    predictionPerCell = self.confParams["predictionPerCell"]

    grid_data = [[[] for i in range(gridColSize)] for j in range(gridRowSize)]

    for line in lines:
      file_data = line.split()

      obj_c = file_data[0]
      x_n = float(file_data[1])
      y_n = float(file_data[2])
      w = float(file_data[3])
      h = float(file_data[4])

      try:
        obj_c = self.confParams["labels"][obj_c]

        grid_row = int(y_n / (1 / gridRowSize))
        grid_col = int(x_n / (1 / gridColSize))

        y = (y_n - ((1 / gridRowSize) * grid_row)) / (1 / gridRowSize)
        x = (x_n - ((1 / gridColSize) * grid_col)) / (1 / gridColSize)
        w = math.sqrt(w)
        h = math.sqrt(h)

        grid_data[grid_row][grid_col].append([x, y, w, h, obj_c])

      except KeyError:
```

```
35          print("Wrong class")
36          return None
37
38      grid_data = np.array(grid_data)
39      true_grid_data = np.zeros((gridRowSize, gridColSize, (predictionPerCell * ↩
            ↪ 5)))
40
41      for n_row, row in enumerate(grid_data):
42        for n_col, col in enumerate(row):
43          n_boxes = len(col)
44          n_trueBoxes = 0
45
46          if(n_boxes != 0):
47            np.random.shuffle(col)
48            if(n_boxes > predictionPerCell):
49              col = col[:predictionPerCell]
50
51            for n_box, box in enumerate(col):
52              for n_element, element in enumerate(box):
53                true_grid_data[n_row][n_col][(n_trueBoxes * 5) + n_element] = np.↩
                      ↪ float(element)
54
55              n_trueBoxes += 1
56
57
58      return true_grid_data
```

Con este mismo método cargamos en el proyecto la configuración del JSON, y creamos la matriz con los valores de los centroides.

5.3.2 Red neuronal

Este es el código con el que generamos la red, nos basamos en la imagen de la Figura [4.2.3]. Para la generación del modelo usamos **Keras Funcional API**, el cual es una forma de crear modelos con una tipología no lineal, modelos con capas compartidas y modelos con múltiples entradas o salidas. A continuación la implementación de la red:

Código 5.4: Implementación red YOLO Keras

```
def make_yolo_model(input_shape):
  model_input = Input(shape= input_shape)
  #================================================================
  # FIRST BLOCK
  #================================================================
  #Conv. Layer7x7x64-s-2
  model = Conv2D(64,(7,7),strides=(2,2),padding='valid') (model_input)
  model = LeakyReLU(alpha=0.1) (model)
  #Maxpool Layer2x2-s-2
  model = MaxPooling2D(pool_size=(2,2),strides=(2,2)) (model)
  #================================================================
  # Second BLOCK
  #================================================================
```

```
#Conv. Layer3x3x192
model = Conv2D(192,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Maxpool Layer2x2-s-2
model = MaxPooling2D(pool_size=(2,2),strides=(2,2)) (model)
#================================================================
# THIRD BLOCK
#================================================================
#Conv. Layers 1x1x128
model = Conv2D(128,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x256
model = Conv2D(256,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 1x1x256
model = Conv2D(256,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x512
model = Conv2D(512,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Maxpool Layer2x2-s-2
model = MaxPooling2D(pool_size=(2,2),strides=(2,2)) (model)
#================================================================
# FOURTH BLOCK
# Conv. Layers 1x1x256 + Conv. Layers 3x3x512 x 4
#================================================================
#Conv. Layers 1x1x256 1
model = Conv2D(256,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x512 1
model = Conv2D(512,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 1x1x256 2
model = Conv2D(256,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x512 2
model = Conv2D(512,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 1x1x256 3
model = Conv2D(256,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x512 3
model = Conv2D(512,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 1x1x256 4
model = Conv2D(256,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x512 4
model = Conv2D(512,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#================================================================
```

```
#Conv. Layers 1x1x512
model = Conv2D(512,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x1024
model = Conv2D(1024,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Maxpool Layer2x2-s-2
model = MaxPooling2D(pool_size=(2,2),strides=(2,2)) (model)
#======================================================================
# Fifth BLOCK
# Conv. Layers 1x1x512 + Conv. Layers 3x3x1024 x 2
#======================================================================
#Conv. Layers 1x1x512 1
model = Conv2D(512,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x1024 1
model = Conv2D(1024,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 1x1x512 2
model = Conv2D(512,(1,1),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x1024 2
model = Conv2D(1024,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#======================================================================
#Conv. Layers 3x3x1024
model = Conv2D(1024,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x1024-s-2
model = Conv2D(1024,(3,3),strides=(2,2),padding='valid') (model)
model = LeakyReLU(alpha=0.1) (model)
#======================================================================
# Sixth BLOCK
# #Conv. Layers 3x3x1024 x2
#======================================================================
#Conv. Layers 3x3x1024 1
model = Conv2D(1024,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#Conv. Layers 3x3x1024 2
model = Conv2D(1024,(3,3),padding='same') (model)
model = LeakyReLU(alpha=0.1) (model)
#======================================================================
# Seventh BLOCK
#======================================================================
#Conn. Layer Dense 4096 (Fully Connected)
model = Flatten() (model)
model = Dense(4096,activation="linear") (model)
#======================================================================
# Eighth BLOCK
#======================================================================
model = Dense(7 * 1 * 10,activation="linear") (model)
```

```
116  model_output = Reshape((7,1,10)) (model)
117
118  model = Model(inputs=model_input, outputs=model_output)
119
120  model.name = "yolo"
121
122  return model
```

La generación de la red está separada por comentarios, los cuales forman unos bloques que coinciden con la red publicada en el ”paper” oficial para facilitar su entendimiento y corrección. Todos los componentes de la red han sido explicados anteriormente, aunque cabe destacar que la salida de la red es diferente ya que el tensor que necesitamos sólo requiere de (S1xS2x(B*5)+C), siendo S1=7, S2=1, B=2, y C=0. (7x1x(2*5)). Estos cambios se deben, puesto que lo que buscamos es un pentagrama ancho, hemos cambiado la forma en la que se genera el grid, siendo dividido en 7 filas , pero con una única columna, ya que no queremos que el pentagrama aparezca en diferentes celdas, si no intentar cuadrarlo en una para que sea más sencilla su detección. Por otro lado, el número de predicciones se mantiene a dos (B), pero el valor de clases de objetos identificados (C) es cambiado a 0, puesto que nos es irrelevante conocer el tipo de objeto. En el caso de que detectara algún objeto, este siempre sería un pentagrama.

5.3.3 Generación de función de pérdida

Antes de poder entrenar los datos, necesitamos implementar la función de pérdida para obtener la capacidad de aprendizaje. Esta función de pérdida explicada en la metodología ha sido levemente modificada para nuestro caso en concreto. Generación de función de pérdida:

Código 5.5: Función de pérdida Yolo

```
1 import math
2 import keras.backend as K
3
4 # LOSS FUNCTION
5 def customLossFunction(expected, predicted):
6   #Empieza Función de pérdida
7   #Obtenemos matriz de 5 5 5 5 1 para cuando hay objeto o 0 0 0 0 0.5 para ↩
        ↪ cuando no
8   # X Y W H C X Y W H C
9   res = tf.ones_like(expected[:,:,:,0:5])
10  print("resInit: " + str(res.shape))
11  first = True
12  for npredictions in range (dr.confParams["predictionPerCell"]):
13    #Calculo de la C
14    c_expected = expected[:,:,:, ((npredictions * 5) + 4) : ((npredictions * 5)↩
          ↪  + 5)] #Obtenemos en la priemra iteracion la C de p1 y en la segunda↩
          ↪  la C de p2, en el caso de que sólos se hagan 2 predicciones
15    alpha = tf.scalar_mul(5,c_expected)
16    for _ in range (2):
17      print("primerConcat")
18      alpha = tf.concat([alpha,alpha],-1) ##Unimos las últimas capas para ↩
```

```
        ↪ obtener el 5 5 5 5 -> X Y W H
      print("Alpha_SHAPE: " + str(alpha.shape))
    #Matriz de unos para conseguir el valor de la C sea 1 o 0.5 haciendo un cá↩
        ↪ lculo
    ones = tf.ones_like(c_expected)
    ones = ones + c_expected
    ones = tf.scalar_mul(0.5,ones)
    alpha = tf.concat([alpha,ones],-1)

    if first:
      res = res * alpha
      first = False
      print("first")
    else:
      print("else")
      print("res_shape: " + str(res.shape))
      print("alpha_shape: " + str(alpha.shape))

      res = tf.concat([res,alpha],-1)

  print("resFinal")
  eval = K.square(res * (expected - predicted))
  eval = K.sum(eval)

  return eval
```

Para la implementación de esta función de pérdida hemos tenido que aprender cómo funcionan las operaciones con tensores en Keras, usando Keras.backend (Zakkay, 2019; Liu, 2019).

La finalidad es generar una matriz donde, en el caso de haber detectado un pentagrama, añadiremos [5,5,5,5] a la matriz para posteriormente multiplicarse con los valores de (x,y,w,h), como explica la fórmula de función de pérdida (coord = 5). Como la confianza de clase es irrelevante, siempre es un pentagrama, el siguiente valor insertado es 1. En el caso de que no encontrásemos un objeto, añadiríamos [0,0,0,0] que serán multiplicados por (x,y,w,h) para anular estos valores y por último tendríamos un 0.5 que hace referencia al valor de lambdaNoObj.

Por último, cuando completamos la matriz, que debe contener la siguiente forma:

$$[[5,5,5,5,1],[5,5,5,5,1],[0,0,0,0,0.5],[5,5,5,5,1]...]$$

Esta matriz es multiplicada por el error mínimo cuadrado de los datos de entrada y salida obtenida. Para posteriormente sumar los valores del tensor y devolver dicho valor.

5.3.4 Entrenamiento de la red

Una vez ya tenemos la arquitectura del modelo, los datos de entrenamiento y la función de pérdida implementada, podemos proceder a realizar el entrenamiento de la red. El siguiente

fragmento de código realiza dicho entrenamiento:

Código 5.6: Particionado de muestras Yolo

```
import sklearn
from sklearn.model_selection import train_test_split
from sklearn import metrics

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_split
    ↪ , random_state=42)

X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)

model = make_yolo_model((448,448,3))
print(model.summary())

model.compile(loss=customLossFunction,optimizer='adam', metrics=['accuracy'])

model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs, ↩
    ↪ validation_split=0.2, verbose=2, callbacks=[csv_logger]) #Entrenamiento ↩
    ↪ sin aumentado de datos con LOG
loss, acc = model.evaluate(X_test, y_test, batch_size=batch_size)
```

Cabe destacar que importamos la librería sklearn para utilizar el método train-test-split con el que vamos a separar conjunto de entrenamiento y conjunto de test. Después, una vez generado el modelo con el método ”make-yolo-model(input-shape)”, el cual se encarga de generar la red como ya hemos explicado antes, vamos a usar el método ”compile” para añadir la función de pérdida customizada hecha por nosotros, el optimizador ”adam”. Una vez tenemos el modelo compilado podemos proceder con el entrenamiento, método ”fit” al cual le pasamos el conjunto de entrenamiento y los resultados de los mismos, junto al número de épocas que queremos ejecutar, el tamaño del batch-size[1] y el tamaño que queramos tener para el conjunto de validación.

5.3.5 Generación de cajas

Por último, como ya hemos entrenado la red, ya tenemos las coordenadas de la imagen donde debemos dibujar cada caja. El siguiente método genera dichas cajas:

Código 5.7: Método de impresión de bounding box

```
def printBoxes(images, predictions):
  img_w = dr.confParams["img_size"]["col"]
  img_h = dr.confParams["img_size"]["row"]

  cell_w = img_w / dr.confParams["predictionPerCell"]
  cell_h = img_h / dr.confParams["predictionPerCell"]
```

[1] batch-size: Es el tamaño del conjunto de muestras de entrenamiento que escogemos para entrenar cada época.

```
7
8  for n_img in range(len(images)):
9    img = images[n_img]
10   pred = predictions[n_img]
11
12   for n_row, row in enumerate(pred):
13     for n_col, col in enumerate(row):
14       for npredictions in range (dr.confParams["predictionPerCell"]-1):
15         cx = col[0 + (npredictions * 5)]
16         cy = col[1 + (npredictions * 5)]
17         w = col[2 + (npredictions * 5)]
18         h = col[3 + (npredictions * 5)]
19
20         w = w**2 * img_w
21         h = h**2 * img_h
22
23         xmin = int((cx * cell_w) + (cell_w * n_col) - (img_w / 2))
24         ymin = int((cy * cell_h) + (cell_h * n_row) - (img_h / 2))
25         xmax = int((cx * cell_w) + (cell_w * n_col) + (img_w / 2))
26         ymax = int((cy * cell_h) + (cell_h * n_row) + (img_h / 2))
27
28         cv2.rectangle(img, (xmin, ymin), (xmax, ymax), (0, 0, 255), 2)
29
30
31     plt.imshow(img)
32     plt.show()
```

Para realizar las cajas obtenemos los centroides de cada bounding box y con la función de open-cv2.rectange pintamos las cajas en las coordenadas predichas.

5.4 Implementación UNet

Para su desarrollo, gracias al trabajo de investigación realizado en el anterior enfoque, se han podido conseguir resultados de forma más rápida y precisa al conocer mejor las bases.

5.4.1 Generación de salida de la red

Para comenzar a explicar la implementación de UNet, debemos saber que como en el enfoque anterior las etiquetas de los manuscritos están guardadas en archivos JSON. Estos JSON van a ser utilizados para generar la salida de la red con el objetivo de poder entrenar la misma. Como en los JSON tenemos las coordenadas de los bounding box, vamos a utilizar estas coordenadas para generar una imagen totalmente negra para las regiones donde no hayan pentagramas, y regiones blancas donde sí hay pentagramas, como en la siguiente Figura 5.3.

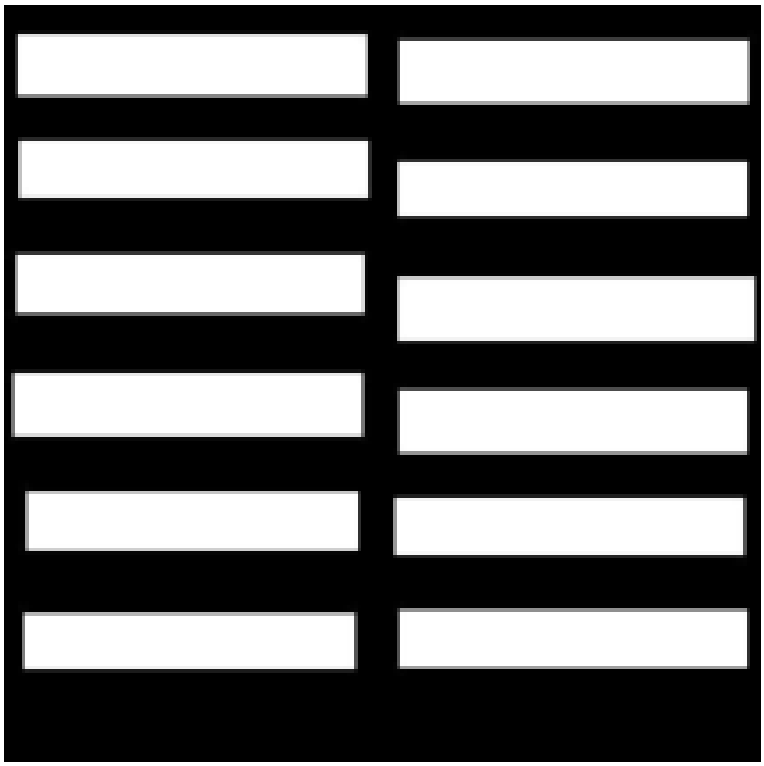


Figura 5.3: Ejemplo real de salida UNet

Para generar esta salida de la red utilizamos el siguiente método ”yGenerator”:

Código 5.8: Método para generación salida de la red UNet

```
1 def yGenerator(listaImagenes, resize):
2
3   lines = open(listaImagenes, 'r').read().splitlines()
4   lstfo = open("carga.lst", "w")
5   for line in lines:
```

```
    img_path, JSON_path = line.split('\t')
    img = cv2.imread(img_path)

    if img is not None:
        with open(JSON_path) as img_JSON:
            data = JSON.load(img_JSON)

            (rows, cols) = img.shape[:2]
            img_y = Image.new('1', (cols, rows))
            draw = ImageDraw.Draw(img_y)

            for page in data['pages']:
                if "regions" in page:
                    for region in page['regions']:
                        if region['type'] == 'staff' and "symbols" in region:
                            porcentaje = 0.2 #Porcentaje que queremos quitar
                            porcentaje_altura = 0.05
                            staff_fragment = int(((region["bounding_box"]["toY"] - region["bounding_box"]["fromY"] ) * porcentaje) / 2) #Calcular porcentaje que hay que quitar de la caja
                            staff_fragment_altura = int(((region["bounding_box"]["toX"] - region["bounding_box"]["fromX"] ) * porcentaje_altura) / 2)
                            staff_top, staff_left, staff_bottom, staff_right = region["bounding_box"]["fromY"] + staff_fragment, region["bounding_box"]["fromX"] ] + staff_fragment_altura, region["bounding_box"]["toY"] - staff_fragment, region["bounding_box"]["toX"] - staff_fragment_altura
                            draw.rectangle([(staff_left, staff_top), (staff_right , staff_bottom)], fill=1)
            path, image_name = img_path.rsplit('/', 1)

            try:
                os.mkdir(path + "_y")
            except OSError:
                pass

            img_y = img_y.resize((resize,resize))

            img_y.save(path + "_y/" + image_name)
            lstfo.write(img_path + "\t" + path + "_y/" + image_name + "\n")
lstfo.close()
```

Para el funcionamiento de este método generamos un fichero ".lst", donde listamos imagen original y JSON perteneciente a la imagen separados por un tabulador (equivalente a cuatro espacios normalmente), de forma que a la hora de cargarlos sabemos cual es su respectivo. Un ejemplo con solo dos muestras sería el siguiente:

b-59-850/12605.jpg b-59-850/12605.jpg.JSON
b-59-850/12606.jpg b-59-850/12606.jpg.JSON

Como hemos dicho anteriormente, este método generamos una imagen negra con la silueta de los pentagramas en blanco. Sin embargo, en la mayoría de nuestros datos las coordenadas de los pentagramas no se ajustan a lo que es el pentagrama real, si no que sobra parte de la región. Para estos casos reducimos el alto de la caja una cantidad X de forma equitativa entre el principio del bounding box y el final, reduciendo así el ancho de la imagen y ajustando más el bounding box al tamaño real del pentagrama. Además este método aparte de generarnos los datos para el entrenamiento, también nos genera un fichero ".lst" que usaremos posteriormente para cargar los datos.

5.4.2 Carga de datos de entrenamiento

Para cargar los datos en nuestro modelo utilizamos el siguiente método:

Código 5.9: Carga de muestras UNet

```
def load_and_normalize_data(path):
    x = []
    y = []
    x_names = []
    lines = open(path, 'r').read().splitlines()

    for line in lines:
        img_path, img_path_y = line.split('\t')


        img = cv2.imread(img_path, 0)
        img_y = cv2.imread(img_path_y, 0)

        img = cv2.resize(img, (512,512))
        img = np.expand_dims(img, axis=-1)
        img_y = np.expand_dims(img_y, axis=-1)

        img = np.float32(img / 255.)
        img_y = np.float32(img_y / 255.)

        x_names.append(img_path)
        x.append(img)
        y.append(img_y)

    return np.array(x), np.array(y), np.array(x_names)
```

Para realizar la carga de muestras de entrenamiento, recorremos el fichero ".lst" creado anteriormente, y vamos redimensionando todas las imágenes al tamaño de entrada de la red, ya que alguna de ella puede tener diferente resolución. Además de cargar los datos en vectores que posteriormente usaremos la alimentar la red neuronal.

5.4.3 Generación de la red neuronal UNet

El siguiente método es el encargado de generar, un modelo con la siguiente arquitectura (zhixuhao, s.f.) también realizada con Keras API:

Código 5.10: Implementación red UNet (Keras)

```
def unet(input_size = (512,512,1)):
    inputs = Input(input_size)
    conv1 = Conv2D(16, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(inputs)
    conv1 = Conv2D(16, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(32, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(pool1)
    conv2 = Conv2D(32, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(64, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(pool2)
    conv3 = Conv2D(64, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    conv4 = Conv2D(128, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(pool3)
    conv4 = Conv2D(128, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(256, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(pool4)
    conv5 = Conv2D(256, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv5)
    drop5 = Dropout(0.5)(conv5)

    up6 = Conv2D(128, 2, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(drop5))
    merge6 = concatenate([drop4,up6], axis = 3)
    conv6 = Conv2D(128, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(merge6)
    conv6 = Conv2D(128, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv6)

    up7 = Conv2D(64, 2, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv6))
    merge7 = concatenate([conv3,up7], axis = 3)
    conv7 = Conv2D(64, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(merge7)
    conv7 = Conv2D(64, 3, activation = 'relu', padding = 'same', ↩
        ↪ kernel_initializer = 'he_normal')(conv7)
```

```
30
31    up8 = Conv2D(32, 2, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv7))
32    merge8 = concatenate([conv2,up8], axis = 3)
33    conv8 = Conv2D(32, 3, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(merge8)
34    conv8 = Conv2D(32, 3, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(conv8)
35
36    up9 = Conv2D(16, 2, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv8))
37    merge9 = concatenate([conv1,up9], axis = 3)
38    conv9 = Conv2D(16, 3, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(merge9)
39    conv9 = Conv2D(16, 3, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(conv9)
40    conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same', ↩
          ↪ kernel_initializer = 'he_normal')(conv9)
41    conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)
42
43    model = Model(input = inputs, output = conv10)
44
45    model.compile(optimizer = Adam(lr = 1e-4), loss = 'binary_crossentropy', ↩
          ↪ metrics = ['accuracy'])
46
47    return model
```

Como podemos apreciar a diferencia de la red de yolo, esta red es bastante más pequeña y las capas convolucionales son menores, por lo que con menos datos deberíamos conseguir resultados. Además esta red no utiliza una función de pérdida custom, si no que utiliza ”binaryCrossentropy” que es de las más usadas para clasificación de imágenes.

5.4.4 Entrenamiento de la red

Para el entrenamiento de la red ya sólo necesitamos entrenar el modelo con los datos de entrenamiento:

Código 5.11: Proceso de entrenamineto y carga de muestras en UNet

```
1 #=============================================================================#
2 # Carga del modelo #
3 #=============================================================================#
4
5 model = unet()
6 if path.exists("unet.h5"):
7   unet = open("unet.h5", 'r').read()
8   model = load_model("unet.h5")
9 print(model.summary())
10 #=============================================================================#
11 # Entrenamiento #
12 #=============================================================================#
13
```

```
#model.compile(loss='categorical_crossentropy',optimizer='adam', metrics=['accuracy'])
print(x_train.shape)
print(y_train.shape)
#model.fit(x_train, y_train, epochs = _epochs, batch_size = _batch_size, validation_split=_validation_split)
model.fit(x_train_train, y_train_train, epochs = _epochs, batch_size = _batch_size, validation_data=validation_train)
```

En esta parte del código buscamos si tenemos unos pesos ya entrenados para la red, y en el caso de no tenerlos procedemos a entrenar la misma, pasándole las muestras de entrenamiento, épocas que deseamos realizar, tamaño del batch size, y cantidad de imágenes para validación.

5.4.5 Predicciones

Una vez ya entrenada la red, si hacemos predicciones sobre los datos de test, que son datos que la red no ha contemplado en la etapa de entrenamiento obtenemos unos resultados como los siguientes:

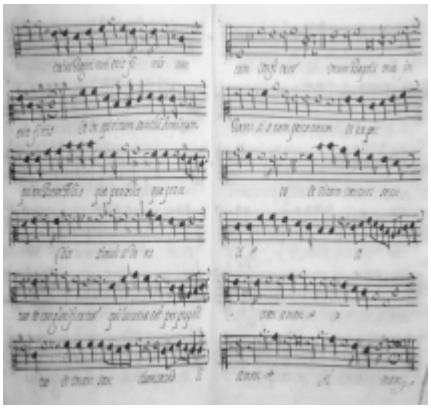
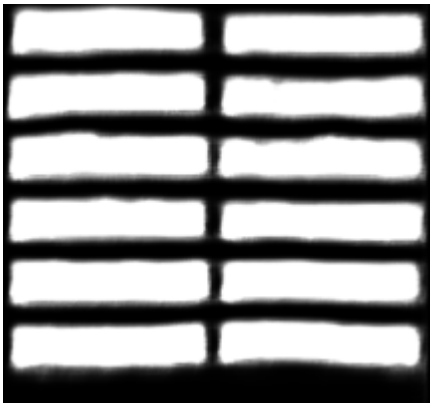


Imagen (a) : *Entrada original*



Salida (b) : *real*

Tabla 5.1: Proceso de mapa de calor UNet

Una vez ya hemos realizado este proceso, debemos umbralizar el resultado y aplicar el método ”findContours” de openCV para obtener las coordenadas del pentagrama. A continuación utilizaremos las coordenadas obtenidas para con el método ”drawContours” de openCV dibujar los bounding box en la imagen original, como muestra este método:

Código 5.12: Método de umbralizado y detección de componentes conexas

```
import numpy as np
import cv2 as cv

ficheroResultados = open("valoresBoundingBox.txt",'w')
ficheroResultados.write("PathImagen + x,y,w,h\n")

for i in range(45):
    img = cv.imread('predicts/' + str(i) + '_predicted.png')
```

```
    img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    ret, thresh = cv.threshold(img_gray,127,255,0)
    contours, hierarchy = cv.findContours(thresh, cv.RETR_TREE , cv.↩
        ↪ CHAIN_APPROX_SIMPLE)

    img_original = cv.imread('predicts/' + str(i) + '_original.png')
    res = cv.drawContours(img_original,contours,-1,(255,153,51),3)

    path = "contorno/ejemplo_" + str(i) + "thickness_3.png\n"
    ficheroResultados.write(path)
    for cnt in contours:
        x, y, w, h = cv.boundingRect(cnt)

        ficheroResultados.write(str(x) + "\t" + str(y) + "\t" + str(w) + "\t" + ↩
            ↪ str(h) + "\n")
    cv.imwrite("contorno/ejemplo_" + str(i) + "thickness_3.png", res)
ficheroResultados.close()
```

Por último, para realizar una prueba con MURET necesitamos guardar en un fichero las coordenadas del bounding box generado de cada imagen, para ello utilizamos el método boundingRect() de OpenCV. Este es un ejemplo del resultado obtenido:

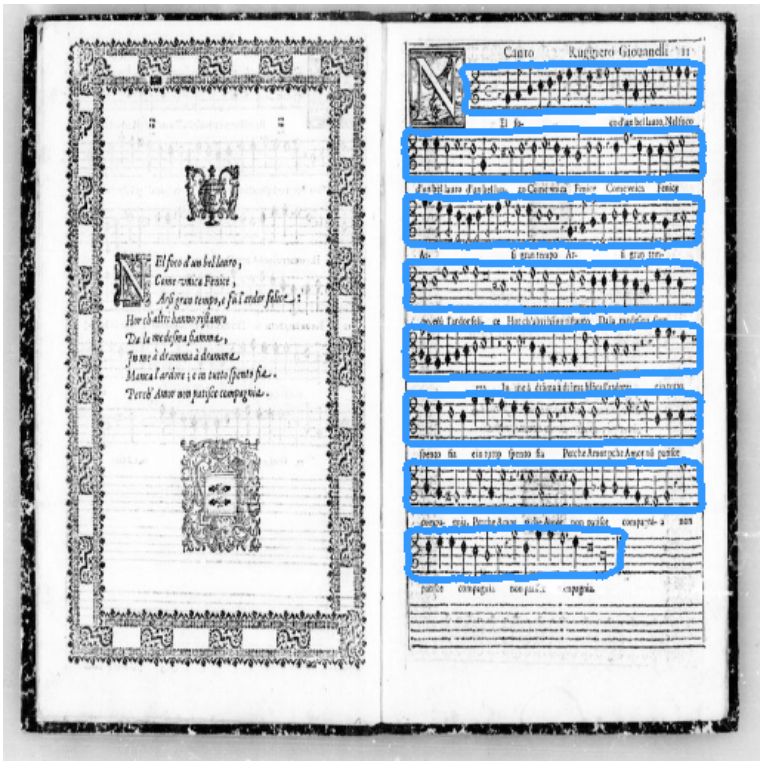


Figura 5.4: Ejemplo con bounding box UNet

6 Experimentación

En este capítulo, explicaremos los experimentos realizados en ambos enfoques. Así como los cambios realizados para obtener mejores resultados.

6.1 Experimentos YOLO

Como explicamos anteriormente, la gran mayoría de esta implementación está llevada a cabo por especulaciones. Ya que a la hora de entrenar el modelo no detallan cómo entrenarlo. La gran mayoría de usos están llevados a cabo desde su propia implementación.

En el primer experimento realizado en este proyecto, cargamos las muestras originales de los manuscritos e intentamos hacer una predicción con la función de pérdida **Minimum Square Error** y optimizador **adam**, aunque los resultados obtenidos, como era de esperar no fueron buenos. El error de la función de pérdida iba creciendo, como podemos ver en el siguiente fragmento:

Epoch 1/30 - 5s - loss: 0.8415 - val_loss: 3.0878e-04
Epoch 2/30 - 1s - loss: 67.6935 - val_loss: 3835.6030
Epoch 3/30 - 1s - loss: 6285.8260 - val_loss: 34948716.0000
Epoch 4/30 - 1s - loss: 62508146.4375 - val_loss: 2178870.2500
Epoch 5/30 - 1s - loss: 8240329.7500 - val_loss: 446765.0625
Epoch 6/30 - 1s - loss: 105256838.5312 - val_loss: 378840.0312

En este caso no habíamos separado las muestras en conjuntos de training y test, ya que solo intentábamos que entrenase, aunque sí se separó entre training y validación, siendo 80% y 20% respectivamente.

Viendo los resultados obtenidos, debíamos implementar la función de pérdida que explicamos en la implementación e intentar aplicarla al mismo caso. Los resultados obtenidos con la función de pérdida customizada no fueron mucho mejor, también modificamos la división de la imagen para nuestro proyecto en concreto, por lo que decidimos simplificar la división de columna, siendo esta S=1, en la cual íbamos a identificar el pentagrama. Por lo que si S1 = 7 y S2 = 1, tendríamos 7 filas con una columna cada una.

Con 100 épocas de entrenamiento que no son muchas, nos dimos cuenta que el error de la función de pérdida continuaba creciendo en vez de decrecer, llegando a obtener estos resultados:

Epoch 100/100 - 0s - loss: 19805077749366784.0000 - accuracy: 0.0857 - val_loss: 7555865917784064.0000 - val_accuracy: 0.0714

Al tener un valor de pérdida tan alto, cuando mostramos las coordenadas de la predicción, obtenemos valores que se salen del tamaño de la imagen, siendo algunos negativos en vez de positivos.

```
[[[ 2858335.5 -605351.5 1724322.2 667674.4 -925366.8 339167.3 -381466.97 -192443.78
  -50968.312 3910339.5 ]]
 [[-1091620.2 1298392.1 2803340.8 2762423.8 -3949318.8 1016750.25 229968.56 -4736821.5
   1634038. 725016.8 ]]
 [[-1711160.1 868426.75 -2624876. -813731.7 -4748900. -234000.2 770964.44 108021.06
   147843.75 -3774465. ]]
 [[-1025731.44 -2717349.5 1064099.2 13776.8125 -1366194.6 -161629. 1227343.9 2582580.8
  -1343169. 2459195. ]]
 [[-2516531.8 3908239.5 211014.7 2189832. 2752463.8 1640422.1 2715274.8 2733134.2
   2442790.5 -3800867. ]]
 [[ 395644.7 -887026.4 -1349437.9 -4111293.5 3967027. 568783.2 705810.25 -833275.7
   1035264.5 1005211.25 ]]
 [[ 26013.818 -1371662.2 1065495.1 -1868630.5 -5252364.5 -521769.6 189530.64 388336.44
   389762.1 -4153725.8 ]]]
```

Para representar estos valores, obtenemos 7 valores matriciales, los cuales hacen referencia a las 7 filas que hemos generado, y 10 valores en cada uno de ellos, puesto que hacemos 2 predicciones por celda. Obtenemos (x,y,w,h,C) de la primera predicción y a continuación (x2,y2,w2,h2,C) de la segunda predicción. Al haber valores negativos y valores que salen fuera del tamaño de la imagen, cuando dibujamos los bounding box en la imagen original, no los vemos reflejados Figura 6.1.

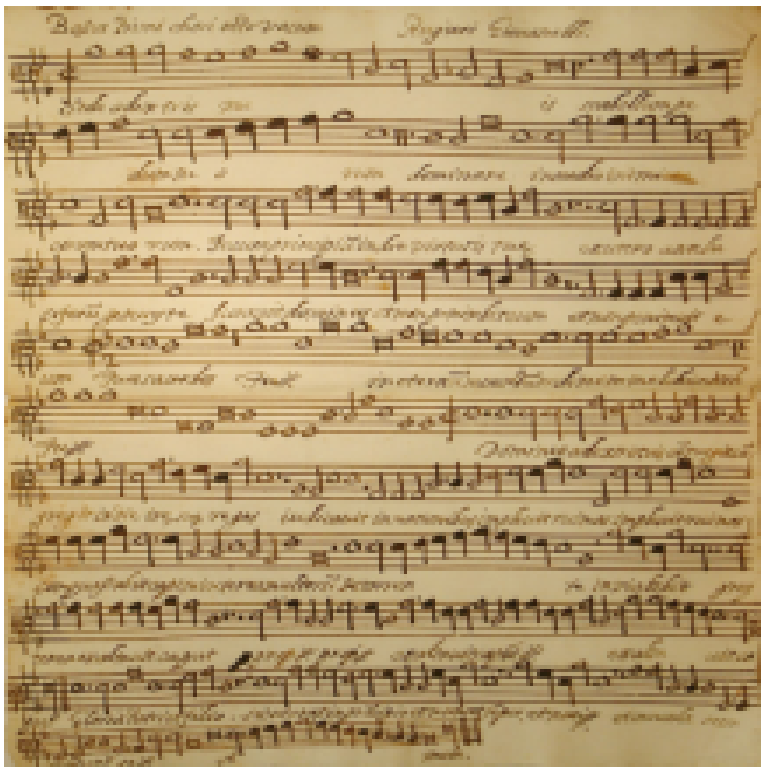


Figura 6.1: Ejemplo salida bounding box YOLO

Al comprobar que el error de la función de pérdida iba aumentando en vez de disminuyendo, implementamos una serie de datasets sintéticos, ya que el principal problema podría ser la escasez de imágenes. El siguiente método fue el utilizado para generar datasets sintéticos de forma aleatoria:

Código 6.1: Generación dataset sintético aleatorio

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
from random import randint

!mkdir processedDataSintetic
!mkdir dataSintetic

backColor = (208, 179, 119)
color = (55, 29, 4)
thickness = -1
staffHeight = 19
imgHeight = 448
imgWidth = 448

for i in range(2000):
  img = np.array([[backColor for x in range(imgHeight)] for y in range(imgWidth)], dtype=np.uint8)
```

```
outFile = open("processedDataSintetic/"+ str(i) + ".data", 'w')

for _ in range(1):

  yRand = randint(0, imgHeight - staffHeight)
  startPoint = (0, yRand)
  endPoint = (imgWidth, yRand + staffHeight)
  #Generamos la Y de la Red
  w = (endPoint[0] - startPoint[0]) / imgWidth
  h = (endPoint[1] - startPoint[1]) / imgHeight
  cx = (startPoint[0] + ((endPoint[0] - startPoint[0]) / 2)) / imgWidth
  cy = (startPoint[1] + ((endPoint[1] - startPoint[1]) / 2)) / imgHeight

  outFile.write('staff ')
  outFile.write(str(cx) + ' ')
  outFile.write(str(cy) + ' ')
  outFile.write(str(w) + ' ')
  outFile.write(str(h) + '\n')

  img = cv2.rectangle(img, startPoint, endPoint, color, thickness)

img = img[:,:,::-1]
cv2.imwrite("dataSintetic/" + str(i) + ".jpg", img)

img = img[:,:,::-1]
plt.imshow(img)
plt.show()
outFile.close()
```

El rango del primer bucle hace referencia a la cantidad de imágenes sintéticas que vamos a generar y el segundo bucle, la cantidad de pentagramas en cada imagen generada de forma aleatoria. Al principio las generamos con varios pentagramas, pero visto que los resultados no mejoraban, decidimos simplificarlo a un único pentagrama por muestra, como podemos apreciar en la siguiente Figura 6.2.

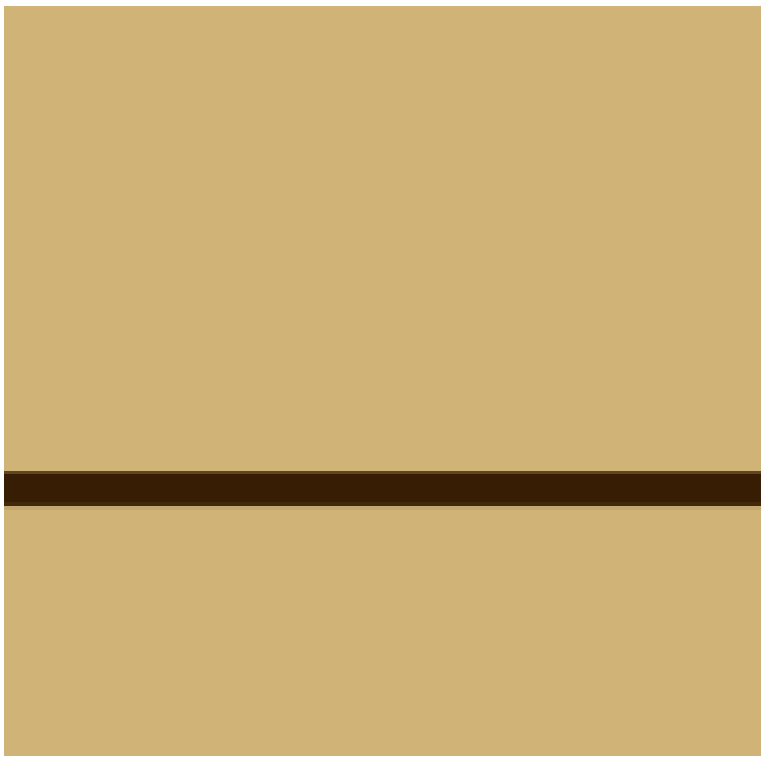


Figura 6.2: Ejemplo muestra sintética con un pentagrama

Para los siguientes resultados se ha utilizado la siguiente configuración del proyecto:

- Uso de 2000 imágenes con un pentagrama para entrenamiento (80% Training y 20% Validación)
- Función de pérdida customizada
- Entrenamiento de 100 épocas
- Batch size de 32 imágenes

Con estos datos conseguimos una tasa de error alta en el entrenamiento, de la cual vamos a mostrar las 3 últimas iteraciones:

Epoch 98/100 - 17s - loss: 2295243783264429824.0000 - accuracy: 0.0525 - val_loss: 656998597776533120.0000 - val_accuracy: 0.1277
Epoch 99/100 - 17s - loss: 274255334254575616.0000 - accuracy: 0.0612 - val_loss: 174716125685219328.0000 - val_accuracy: 0.0152
Epoch 100/100 - 17s - loss: 176615609331613696.0000 - accuracy: 0.0121 - val_loss: 71568721532944384.0000 - val_accuracy: 0.0000e+00

Al ser el error tan alto, cuando realizamos una predicción, obtenemos valores exponenciales positivos y negativos que no poder representar como bounding box:

```
[[[-5.21133600e+06 -2.33247560e+07 -2.65038880e+07 -2.07876080e+07 -1.67276800e+06
   4.22398304e+08 1.07255024e+08 7.68553152e+08 5.81609024e+08 -1.32313600e+06]]
 [[ 6.47500440e+07 1.42910720e+07 -6.73076800e+07 8.35580240e+07 -1.70768000e+05
  8.30110848e+08 1.76599072e+08 -1.36828032e+09 -1.00232208e+08 -4.69732000e+05]]
 [[ 3.16258640e+07 -1.27443200e+06 4.26029880e+07 -6.22464000e+06 -3.29326400e+06
  3.27993632e+08 3.06197248e+08 1.44738608e+08 -1.99886144e+08 -3.95727600e+06]]
 [[ 4.71990400e+06 9.32030400e+06 9.92233600e+06 -1.42836720e+07 -1.84952800e+06
  -1.70341216e+08 -1.38481997e+09 1.06477146e+09 3.30193216e+08 9.70904000e+05]]
 [[ 6.34396800e+06 6.23932000e+05 -8.60508000e+06 1.60783200e+06 1.13736000e+06
  2.48404672e+08 1.09585664e+09 4.58829376e+08 -1.15501280e+07 -2.09593450e+06]]
 [[ 3.64863200e+06 -3.63793600e+06 1.93821200e+06 1.82981600e+06 9.41522000e+05
  -4.00246720e+08 7.90730240e+08 4.16238464e+08 3.01853184e+08 7.14776000e+05]]
 [[-2.41105600e+06 6.49710400e+06 -6.05849200e+06 -3.96451200e+06 1.41376000e+05
 -1.13260640e+08 -4.41600000e+07 -2.30439194e+09 -2.97197056e+08 1.00688400e+06]]]
```

Llegados a este punto, damos por hecho que o bien necesitamos más datos, pero el hecho de generar millones de datos en google collaboratory llevaría mucho tiempo y al cabo de 12 horas perdemos la sesión y su contenido. O bien hemos cometido algún error a la hora de crear la función de pérdida, que a priori desconocemos. Por lo que vamos a proceder al siguiente enfoque.

6.2 Experimentos UNet

La primera idea que tuvimos a la hora de utilizar esta red, era la de particionar las imágenes originales de los manuscritos y aplicar aumentado de datos a dichas imágenes con el objetivo de que pudiese llegar a entender trozos de partituras, varias partituras en una misma imagen, trozos de pentagramas, entre otros.

A la hora de entrenar este modelo no obtuvimos un mejor resultado de antemano, por lo que decidimos directamente entrenar la red sin aumentado de datos y con las imágenes originales redimensionadas al tamaño de entrada de la red.

Utilizando el corpus de Il Lauro Seco, el cual hemos dividido en 30% de imágenes para el conjunto de test, 70% para el conjunto de training del cual volveremos a particionar en 20% para validación y 80% para training, lo que supone 46 imágenes totales para test, 20 para validación y 80 para training.

Obteniamos un 91% de acierto, del cual sacábamos las siguientes muestras Tabla 6.1

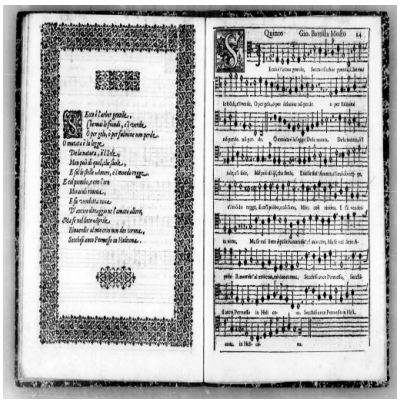


Imagen (a) : *original*

Imagen (b) : *predicha*

Tabla 6.1: Primer resultado obtenido de UNet

Sin embargo, como podemos ver en la imagen los bounding box de los pentagramas no encajan muy bien. Esto produce que en nuestra salida casi se produzcan solapamientos, que una vez aplicado el umbralizado y habiendo obtenido las coordenadas, en algunas imágenes obtenemos fallos. (Figura 6.3):

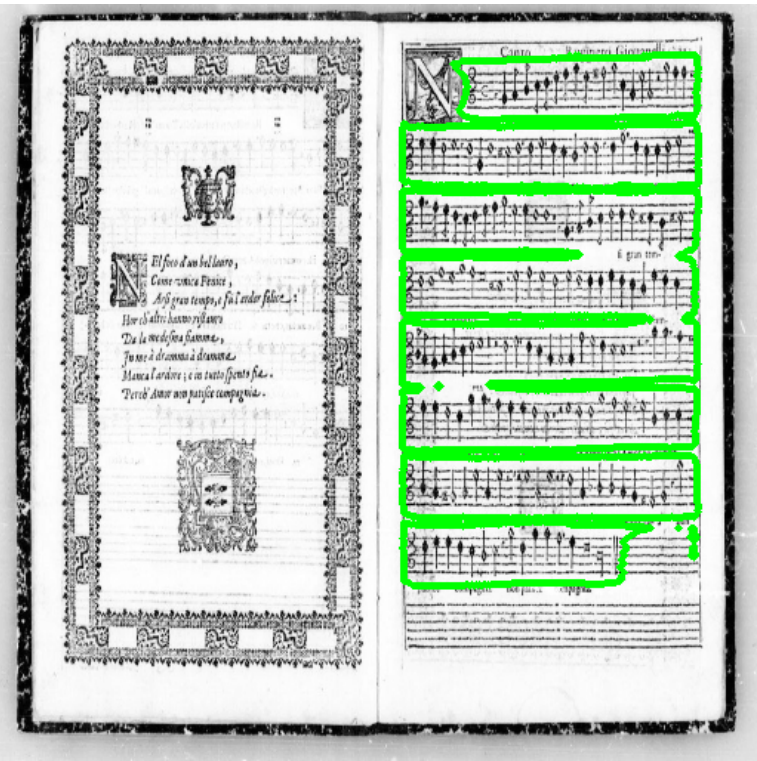


Figura 6.3: Errores obtenidos de los datos del primer entrenamiento

Como podemos ver en la imagen (Figura 6.3) algunos bounding box están solapados y no estamos captando los pentagramas vacíos. No podemos captar los pentagramas vacíos, ya que en los datasets no se han etiquetado, en algunos casos se han colocado, pero en otros no. Por lo que si intentamos entrenar la red para que detecte dichos pentagramas vacíos, termina no entendiendo nada, y obtenemos resultados similares a estos Figura 6.4.

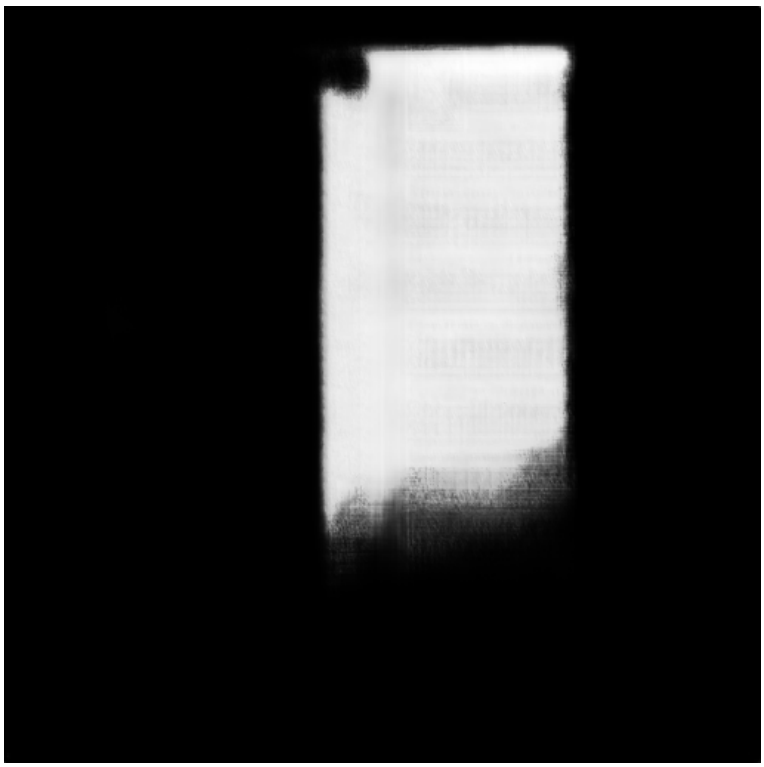


Figura 6.4: Error al entrenar con pentagramas vacíos

Para solucionar estos problemas, decidimos no añadir los pentagramas vacíos, ya que no era necesario para después poder acceder a las notas. Por otro lado, para solucionar el solapamiento de los bounding box, se realizan procesos de reducción de bounding box de 5%, 10% y 20% con el objetivo de ceñir estos pentagramas, llegando a obtener los mejores resultados con una reducción del 20% de la región del bounding box.

Gracias a este proceso obtenemos una tasa de acierto de 95%, y conseguimos evitar los solapamientos de estas coordenadas, como podemos ver en el ejemplo de la siguiente Figura 6.5.

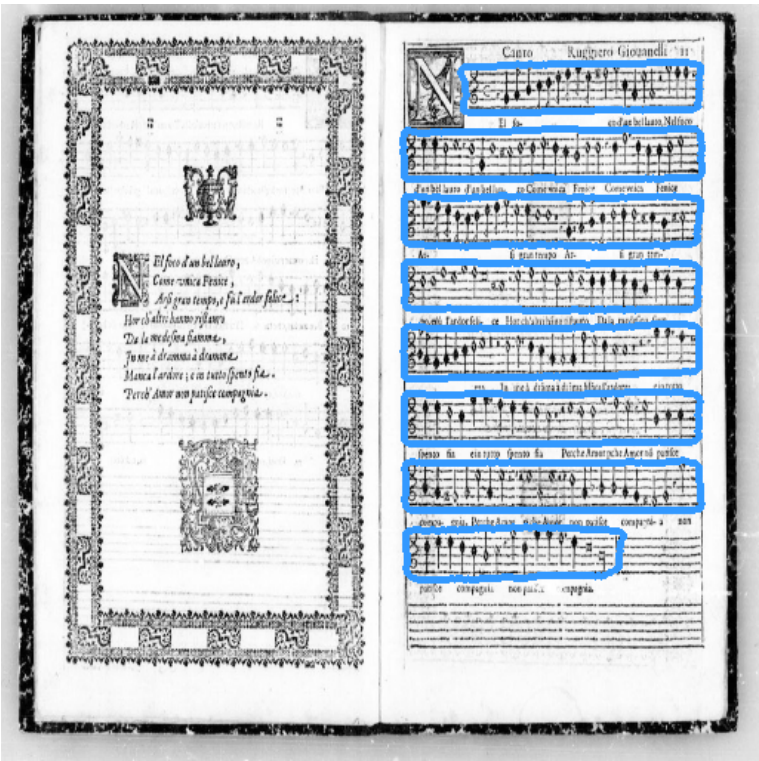


Figura 6.5: Ejemplo Il Lauro Seco final

Por otro lado, realizamos el mismo proceso para el corpus de Capitan, el cual consta de 73 imágenes. También realizamos las mismas particiones que en anterior corpus, aunque existe una diferencia significativa entre ambos. Para este corpus los resultados aplicando una reducción del 20% del bounding box nos daban peores resultados, por lo que se experimentó con un 10% y un 5%. Como resultado final aplicamos la reducción del 5% llegando a obtener resultados como el de la Tabla 6.2.

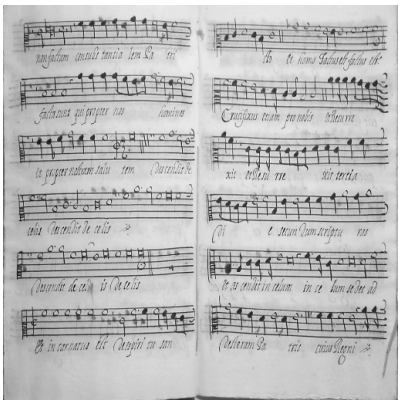


Imagen (a) : *original*

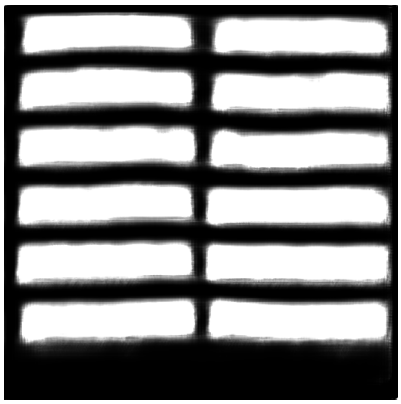


Imagen (b) : *predicha*

Tabla 6.2: Imagen original y predicción corpus de Capitan

A esta predicción le aplicamos el proceso de umbralizado por el cual obtenemos las coordenadas. Como podemos apreciar en la siguiente Figura 6.6

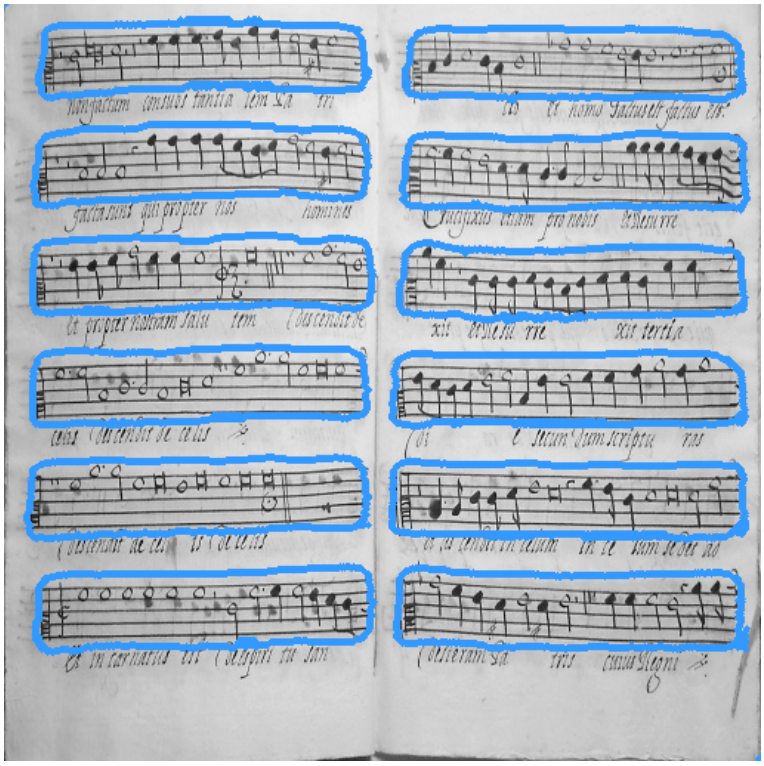


Figura 6.6: Resultado final UNet de Capitan

Para finalizar la experimentación, se ha realizado una prueba más, en la que demostramos que el proceso de automatización con el algoritmo es prácticamente idéntico a los resultados obtenidos si realizásemos el proceso de etiquetado a mano. Para ello, guardamos las coordenadas de los pentagramas de cada imagen en un fichero el cual vamos a utilizar en el proyecto de Muret.

Muret se encargará de detectar las notas de los pentagramas utilizando las coordenadas obtenidas con nuestra red y las obtenidas si el proceso se hiciese a mano. Devolviendo el ratio de notas incorrectas con ambos procesos. **GT** hace referencia a cuando se entrena o se usa el detector de símbolos con cajas etiquetadas manualmente, mientras que **PRED** hace referencia a cuando se entrena o se usa el detector de símbolos con cajas predichas con la red UNet.

A continuación se muestras las siguientes tablas de resultado 6.3 y 6.4, donde mostramos los resultados obtenidos para el corpus de Il Lauro Secco y Capitan respectivamente.

Entrenando con cajas de	prediciendo sobre cajas de	Ratio de error
GT	GT	2.9
GT	PRED	4.7
PRED	PRED	3.1

Tabla 6.3: Resultados de símbolos incorrectos de Il Lauro Secco

Entrenando con cajas de	prediciendo sobre cajas de	Ratio de error
GT	GT	11.6
GT	PRED	13.6
PRED	PRED	11.8

Tabla 6.4: Resultados de símbolos incorrectos de Capitan

En conclusión, podemos ver como los resultados obtenidos de ambos procesos son prácticamente idénticos, tanto si se usa para entrenar como si se usa para predecir. Por lo tanto, quiere decir que el proceso de automatización apenas pierde precisión en el reconocimiento final de la música. De forma que ahorramos mucho tiempo a la hora de detectar pentagramas automáticamente perdiendo poca precisión.

7 Conclusión

7.1 Resumen

En este trabajo, nos propusimos el objetivo de detectar regiones de interés en partituras musicales de forma automática. La motivación de este proyecto viene dada por la transcripción de archivos históricos para su preservación y explotación. Además, de tener la oportunidad de poder desarrollar un proyecto de ayuda sobre un patrimonio importante como es el musical. Estos trabajos de transcripción siguen realizándose de forma manual por profesionales, es por eso que el desarrollo de sistemas "OMR" (Reconocimiento óptico de música) aceleran mucho este proceso. En nuestro trabajo, proponemos un pequeño workflow OMR que es la detección automática de pentagramas, este proyecto es importante, ya que hasta la fecha no se han realizado nuevos procesos con técnicas modernas de IA. Este trabajo está constituido por dos enfoques.

En primer lugar, intentamos replicar un algoritmo de detección de objetos denominado YOLO, a partir de las referencias encontradas en el paper oficial, implementado desde cero la red neuronal, función de pérdida, etc. Esta red neuronal que diseñamos a partir del paper contiene muchas capas, lo que produce una cantidad inmensa de parámetros en nuestra red, también nos obliga a producir un dataset enorme para entrenarla. Por otro lado, la función de pérdida implementada no es cotidiana, sino que se trata de una función derivada de 'MSE' (Minimum square error) concreto para el problema. Para finalizar, aunque después de la experimentación no obtenemos unos resultados concretos, ya que las coordenadas predichas aparecen fuera de la imagen, sí obtenemos unos conocimientos puros sobre la materia.

En segundo lugar, toda esta experimentación nos lleva a una buena resolución del problema mediante el segundo enfoque, con una red basada en capas convolucionales denominada UNet, esta red es bastante peculiar, ya que las capas van formando una 'U', realizando procesos de convolución por el lado izquierdo de la red y procesos de convolución transpuestos en el lado derecho. Para el proceso de resolución, llevamos a cabo la obtención de un mapa de calor, el cual es proporcionado por la salida de la red. Este resultado es posteriormente umbralizado con el fin de aplicar un algoritmo de detección de componentes conexas. Una vez aplicado este algoritmo, obtenemos los bounding box de los pentagramas, los cuales dibujaremos sobre la imagen original.

Por último, aunque el diseño de nuestro algoritmo YOLO contenga algunos fallos y no hayamos podido obtener unos resultados óptimos, con nuestro segundo enfoque utilizando la red UNet sí obtenemos unos muy buenos resultados llegando a automatizar el proceso de forma rápida y precisa.

7.2 Conclusión

Como conclusión, podemos afirmar que ambos enfoques son útiles para un desarrollo conjunto. Este desarrollo nos ha permitido adquirir nuevas habilidades tanto en ámbito académico, como en personal.

En primer lugar, el enfoque basado en YOLO, nos ha permitido entender el trabajo que conlleva un proyecto de investigación. Para el desarrollo del mismo se han investigado más a fondo varios conceptos explicados en la carrera y otros no contemplados, los cuales hemos tenido que entender e implementar para replicar YOLO basándonos en la publicación oficial. Como conclusión, aunque no hayan resultados tangibles del mismo, hemos adquirido diferentes conocimientos y habilidades; como el hecho de entender el funcionamiento de YOLO (aplicando en algunos casos "reversing" sobre el código para entender cómo funciona), la realización de diferentes experimentos, aplicación de mejoras sobre el mismo proyecto, implementación de CNN, uso de tensores y operaciones con keras backend, etc.

Por otro lado, el segundo enfoque con la red UNet nos premia con una nueva arquitectura, la cual no habíamos visto durante la carrera y con la que además conseguimos unos buenos resultados. Para la realización del proyecto, entendemos diferentes utilidades sobre redes neuronales convolucionales, ya que en este caso, el resultado de la misma no es una etiqueta ni el resultado final, sino un paso intermedio para la obtención del resultado. Además, para la obtención de este resultado final se ha tenido que profundizar un poco en técnicas de visión por computador, como es el proceso de umbralizado y detección de componentes conexas.

Para finalizar, desde mi punto de vista el proyecto engloba diferentes ámbitos dentro de Machine Learning y muestra diferentes formas de solventar el problema. Además de haber conseguido automatizar el proceso de forma rápida y precisa.

7.3 Trabajo futuro

Como trabajo futuro, en primer lugar, se deberá revisar si la función de pérdida implementada está funcionando de forma correcta en YOLO.

Además, otros aspectos que podrían tratarse en el enfoque YOLO serían los siguientes:

- Investigar diferentes configuraciones para el proyecto como puede ser aplicar diferentes funciones de pérdida customizadas.
- Realizar cambios significativos a nivel de red como reducción de capas convolucionales para intentar obtener un resultado con un dataset más pequeño.
- Aplicar aumentado de datos con el fin de entrenar la red con mayor cantidad de datos.
- Ajustar parámetros distintos a los usados para comprobar cómo se comportan diferentes modelos desconocidos.
- Configurar un workstation dónde poder generar una cantidad enorme de muestras sintéticas con las que posteriormente entrenar la red.

Por otro lado, en el enfoque de UNet podríamos:

- Etiquetar todos los pentagramas vacíos para que posteriormente la red pueda diferenciar entre pentagramas vacíos y con notas.
- Aplicar dos redes UNet realizando el mismo proceso, para con una captar pentagramas y con la siguiente percibir notas, ya que los resultados para pentagramas son buenos.
- Aplicar aumentado de datos para poder llegar a entender trozos de pentagramas, pentagramas girados, etc.

Bibliografía

API, K. (s.f.). Descargado 26-07-2020, de https://keras.io/api/metrics/

Bonner, A. (2019). *The complete beginner's guide to deep learning: Convolutional neural networks and image classification.* Descargado 26-07-2020, de https://towardsdatascience.com/wtf-is-image-classification-8e78a8235acb

Calvo-Zaragoza, J., Jr., J. H., y Pacha, A. (2020, julio). Understanding optical music recognition. *ACM Comput. Surv.*, *53*(4). Descargado de https://doi.org/10.1145/3397499 doi: 10.1145/3397499

Hrabia, M. (2020). *Deep learning vs. machine learning.* Descargado 26-07-2020, de https://towardsdatascience.com/deep-learning-vs-machine-learning-e0a9cb2f288

Kathuria, A. (2018). *What's new in yolo v3?* Descargado 26-07-2020, de https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

LeCun, Y., Bengio, Y., y Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

Liu, Y. (2019). *On writing custom loss functions in keras.* Descargado 26-07-2020, de https://medium.com/@yanfengliux/on-writing-custom-loss-functions-in-keras-e04290dd7a96

Menegaz, M. (2018). *Understanding yolo.* Descargado 26-07-2020, de https://hackernoon.com/understanding-yolo-f5a74bbc7967

Na8. (2018). *¿cómo funcionan las convolutional neural networks? visión por ordenador.* Descargado 26-07-2020, de https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

Prabhu. (2018). *Understanding of convolutional neural network (cnn) — deep learning.* Descargado 26-07-2020, de https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

Redmon, J., Divvala, S. K., Girshick, R. B., y Farhadi, A. (2015). You only look once: Unified, real-time object detection. *CoRR*, *abs/1506.02640*. Descargado de http://arxiv.org/abs/1506.02640

Redmon, J., y Farhadi, A. (2016). Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*.

Redmon, J., y Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv*.

Rizo, D., Calvo-Zaragoza, J., y Iñesta, J. M. (2018). Muret: a music recognition, encoding, and transcription tool. En *Proceedings of the 5th international conference on digital libraries for musicology, dlfm 2018, paris, france, september 28, 2018* (pp. 52–56). Descargado de https://doi.org/10.1145/3273024.3273029 doi: 10.1145/3273024.3273029

Ronneberger, O., Fischer, P., y Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *CoRR, abs/1505.04597*. Descargado de http://arxiv.org/abs/1505.04597

Rosebrock, A. (2016). *Intersection over union (iou) for object detection.* Descargado 26-07-2020, de https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

SHARMA, S. (2017). *Activation functions in neural networks.* Descargado 26-07-2020, de https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Tensorflow. (s.f.). Descargado 26-07-2020, de https://www.tensorflow.org/api_docs/python/tf/ones_like

U-net: Convolutional networks for biomedical image segmentation. (2019). Descargado 26-07-2020, de https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Zakkay, E. (2019). *Advanced keras — constructing complex custom losses and metrics.* Descargado 26-07-2020, de https://towardsdatascience.com/advanced-keras-constructing-complex-custom-losses-and-metrics-c07ca130a618

zhixuhao. (s.f.). Descargado 26-07-2020, de https://github.com/zhixuhao/unet