Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología

Trabajo de Fin de Grado

Grado en Ingeniería Informática

"Sistema automatizado de inventario de cartas de MTG"

"Authomatized inventary system for MTG cards"

Sergio Morente Rodríguez

La Laguna, 1 de julio de 2018

D. Jesús Miguel Torres Jorge con **DNI** 43.826.207-Y, profesor Contratado de la Universidad de La Laguna, como tutor

C E R T I F I C A :

Que la presente memoria titulada:

"Sistema automatizado de inventario de cartas de MTG"

ha sido realizada bajo su dirección por **D. Sergio Morente Rodríguez**,

con N.I.F. 43.838.911-Z

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 15 de mayo de 2018

Agradecimientos

Al tutor D. Jesús Miguel Torres Jorge por el apoyo proporcionado.

A la Universidad de La Laguna por la oportunidad de poder trabajar en este proyecto.

Licencia

Resumen

El objetivo de este proyecto ha sido realizar una aplicación de reconocimiento óptico de caracteres (OCR) a través de diferentes tecnologías, utilizando como lenguaje de la implementación Python.

La totalidad del proyecto se divide en tres fases de implementación, teniendo una fase previa de preparación de imágenes de prueba y dos fases de desarrollo de aplicaciones pudiendo separar estas dos últimas dependiendo de la tecnología aplicada a la resolución del problema.

Para la fase previa se utilizaron principalmente las librería PIL (Python Images Library) y OpenCV para la creación y modificación de las imágenes de prueba. A estas imágenes se les aplicaría ruido y pequeñas rotaciones para simular condiciones más reales a las que se encuentran las imágenes de prueba sin modificar y así poder afirmar que el sistema es capaz de reconocer casos lo más reales posibles independientemente de pequeños inconvenientes.

En cuanto a la primera fase del desarrollo, esta ha sido realizada con un enfoque más tradicional que la segunda, siguiendo una metodología y uso de herramientas más comunes. Esta solución se implementó a través del uso de OpenCV y librerías de Tesseract para Python realizando el OCR únicamente con estas. Los resultados son satisfactorios y el reconocimiento se realiza correctamente tanto en imágenes de prueba como en cartas reales (aplicando en las cartas reales un recorte a la imagen entera y trabajando únicamente sobre la región de interés) sin embargo se ha querido ampliar el proceso de reconocimiento aplicando tecnologías más novedosas y mejorando la eficacia del sistema, esta aplicación es la que se ha llevado a cabo en la segunda fase.

Las herramientas utilizadas en la implementación de la segunda versión del proyecto han sido redes neuronales y mecanismos de Deep Learning, aplicando diferentes operaciones y múltiples capas de procesamiento y aprendizaje para conseguir identificar y clasificar los elementos que se recogen visualmente en forma del texto que componen las imágenes.

Palabras clave: Imagen, OpenCV, OCR, Python, Tesseract, Redes Neuronales, Deep Learning

Abstract

The main objective of this project has been to develop an application for optical character recognition (OCR) with the use of different tools and technologies, the language in which is based this development is Python.

We can divide the project into three different development phases, starting with a previous preprocessing and preparatory phase relating the input images and continuing with two differentiated phases, each one characterized by the approach we give to the solution.

For the preprocessing phase we made use of PIL (Python Images Library) and OpenCV for the creation and modification of the input images. After creating them we add some kind of noise and rotations with the aim of simulating more real conditions in the event of recognition than the synthetic generated images could have by themselves and so, be able to confirm that the application is able to recognize all sort of characters in the input images, independently of external factors.

In the case of the first development phase, the solution has been approached in a more traditional way than the way it has been done in the second one, selecting for that task more frequently used tools and methodologies. This solution has been mostly implemented using OpenCV and Tesseract for Python and performing the OCR only by those. The results are very satisfactory and the recognizing works as designed both in generated images and real ones (in the case of this ones, we cut and select just the region of interest where we know the information that we need is, and work with that part only). Nonetheless, we aim to upgrade this solution, making the application more efficient and innovative. The approach we have followed to achieve this is captured in the second development phase.

The tools used in this second development phase have been neural networks and Deep Learning mechanisms, applying different operations and layers of processing and learning with the goal to be able to recognize and classify the elements that are collected with the optical device, which make up the text that forms part of the images.

Keywords: Image, OpenCV, OCR, Python, Tesseract, Neural Networks, Deep Learnin

Índice general

Índice de figuras

Introducción

Justificación

Existen muchos casos en los que es necesario procesar gran cantidad de elementos para su categorización y organización en un inventario. Es el caso de los productos que se venden o alquilan en cualquiera tienda, taller o biblioteca. Como ya se sabe, hoy en día, para trazar la incorporación, venta, alquiler o manipulación de cualquier producto, lo más común es utilizar códigos de barra. ¿Pero qué ocurre cuando cuando dicho código no está disponible? Tal vez porque se carece del embalaje original. En ese caso podría ser interesante disponer de un sistema para trazar cierto tipo de productos usado sus propiedades más características, como su color, su forma, marcas o textos. Es decir, usando técnicas de visión artificial para clasificar los objetos y gestionar con esa información una base de datos de inventario.

Ese es el objetivo del presente proyecto, sólo que aplicado a un tipo concreto de producto: cartas del juego “Magic: The Gathering (MTG)”. Se pretende capturar imágenes de cartas para identificarlas respecto a una base de datos de cartas conocidas y gestionar su manipulación en una base de datos de inventario.

También se pretende que el sistema sirva para procesar grandes cantidades de cartas de forma automática. Por eso el prototipo se desarrollará preferentemente para un sistema empotrado que se pueda conectar a un sistema industrial que se encargue de la manipulación de las cartas.

Antecedentes

El proyecto se sustenta sobre la idea que el desarrollo de un TFG anterior consiguió materializar, el cual a través del uso de la API de Google Cloud para visión por computador conseguía realizar la identificación de cartas del juego “Magic: The Gathering” [Figura 1: “Carta MTG”] de manera automatizada. El coste del uso de la API depende del número de cartas a identificar, y aunque este coste no es muy elevado parece interesante el desarrollo de una solución propia.

La función del TFG actual consiste en, partiendo de dicha base, realizar el desarrollo de una API propia, que a través de librerías de OpenCV o diferentes tecnologías permita separarse del uso de Google a la hora de realizar el reconocimiento de texto y así poder ejecutar la tareas de registro y ordenación de dichas cartas con una tecnología propia.

Esta ordenación y registro se puede realizar a través de unos datos identificadores que tienen todas las cartas de ediciones relativamente modernas en la esquina izquierda inferior [Figura 2: “Región de interés”], por lo únicamente debemos obtener el texto que se indica en esta región de la carta y siendo capaces de procesarlo podríamos realizar el inventario de estas de forma automatizada.

Actualmente estas tareas se realizan de forma manual sin que intervenga ningún agente automático. Aplicando este método desarrollado se puede conseguir una mejora de cantidad en el número de cartas que se organizan y ordenan en un tiempo determinado además de conseguir mejorar la calidad del trabajo de aquellas personas que tengan esta obligación.

Figura 1: "Carta MTG"

Figura 2: "Región de Interés"

Problemática y estado del arte

Problemática y planteamiento principal:

A la hora de desarrollar un proyecto de estas características uno de los primeros pasos es realizar un planteamiento principal acerca de cuál va a ser el foco central del desarrollo.

En este caso el objetivo mayoritario ha sido implementar, siguiendo un desarrollo iterativo, diferentes formas de resolver el problema planteado en los antecedentes, el reconocimiento y procesamiento del texto que identifica a las diferentes cartas de forma inequívoca.

Una de las primeras decisiones del proceso de desarrollo fue la elección de Python como lenguaje en el que se basaría la implementación de la solución. Esto nos obliga a responder a la siguiente pregunta.

¿Por qué Python?

Python es un lenguaje interpretado bastante intuitivo y fácil de aprender a usar, pero que a la vez tiene a su disposición gran cantidad de módulos, librerías y paquetes instalables para realizar infinidad de operaciones de fines totalmente diferentes, e incluso con prácticamente cero experiencia (en mi caso personal) en el desarrollo de aplicaciones utilizando este lenguaje, el proceso de implementación ha podido ser realizado sin demasiados inconvenientes ya que existe una enorme cantidad de documentación y recursos en la red acerca de cómo utilizar las herramientas que este lenguaje nos proporciona y diferentes guías y tutoriales acerca de su funcionamiento. Otro de los motivos por el que se ha elegido este lenguaje es el hecho de que las diferentes herramientas para diseñar y trabajar con redes neuronales que hemos utilizado se encuentran instauradas en Python y su manejo es extremadamente más sencillo en comparación con las implementaciones en otros lenguajes como C++.

Una vez que se ha decidido el lenguaje que nos va a servir para resolver el problema planteado tenemos que identificar los diferentes pasos o fases a través de las cuales segmentar el desarrollo para así poder obtener resultados a corto plazo y poder ampliar la funcionalidad del programa a través de ellos.

Obviando una fase previa a todo el desarrollo, enfocada principalmente a la preparación de las herramientas (entorno de desarrollo, instalación de paquetes y programas necesarios) y a la búsqueda de documentación y aprendizaje acerca de toda la temática que engloba el proyecto, podemos hablar de una primera fase en la que nos centraremos a generar imágenes de prueba **[Figuras 3 y 4]** que posteriormente nos servirían como comprobación para validar nuestro OCR.

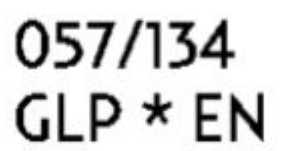

Figura 3: "Identificación artificial."

Figura 4: "Carácter independiente con ruido y rotación."

Habiendo conseguido generar las pruebas suficientes, realizamos un acercamiento a la solución a través de métodos y herramientas relativamente tradicionales, esto es a través del uso de las librerías para Python de OpenCV y Tesseract. Aunque los detalles de esta implementación se especifiquen en el apartado correspondiente al desarrollo del proyecto debemos hacer un pequeño hincapié en lo que supuso esta decisión a nivel de las subfases del desarrollo que tuvieron lugar a partir de esta.

Tesseract nos facilita enormemente la tarea de realizar el OCR sobre cualquier tipo de imagen que tenga texto y las pruebas que se realizaron una vez habiendo implementado este método, utilizando como entradas las imágenes generadas sintéticamente, proporcionaron resultados correctos y satisfactorios, sin embargo a la hora de realizar pruebas con imágenes de cartas reales la complejidad de la situación aumenta. Nos encontramos ante varios problemas que debemos resolver para que Tesseract realice su función adecuadamente incluso en cartas reales, estos problemas son los siguientes:

En primer lugar las cartas reales suelen estar ilustradas haciendo uso de diferentes colores, lo cual puede provocar inconsistencia a la hora de separar los bordes del texto del fondo, generando en ocasiones resultados incorrectos. Para hacer frente a este inconveniente se implementó un método que a través de la imagen de entrada en formato RGB generase una imagen binarizada, únicamente compuesta de blanco y negro.

El segundo de los problemas detectados a la hora de realizar las pruebas con cartas reales fue el hecho de que las cartas tienen una gran cantidad de texto escrito en diferentes lugares. Existen a parte del texto identificador que necesitamos recoger tanto el título de la carta, como la descripción de su utilidad dentro del juego y las normas que debe seguir, además también de en algunos casos una pequeña historia o explicación del contexto de la carta dentro de la historia del juego. Para solucionarlo se ha diseñado un método que a partir de la imágen de entrada (siendo esta la imagen de la carta completa) se recorta una región de interés utilizando medidas relativas, ya que esta región siempre se encuentra en la misma posición en todas las cartas independientemente de la edición, quedándonos al final únicamente con la región que contiene el texto identificador. Una vez extraída esta región se le aplicaría el proceso explicado en el método de binarización y se procedería a realizar el OCR a través de Tesseract.

Es en este punto donde finaliza la implementación de la primera versión del proyecto y comienza la segunda.

En esta segunda versión el planteamiento que se ha seguido a la hora de intentar resolver el problema del reconocimiento es completamente distinto al del uso de Tesseract para realizar el OCR.

El objetivo principal que subyace en esta fase del desarrollo es implementar una aplicación de la tecnología de Deep Learning y redes neuronales con el fin de brindar un enfoque más novedoso al método por el que se realiza el OCR.

Al igual que en el caso de la generación de imágenes para realizar pruebas con Tesseract, en esta versión fue necesaria la creación de una cantidad bastante elevada de imágenes, no solo para poder validar el funcionamiento de la aplicación sino también para poder entrenar a la red neuronal ocupada de entender y clasificar los elementos que conforman el texto. Esta clasificación y entendimiento se hace a través de diferentes capas, principalmente aplicando convoluciones sobre la imagen de entrada, y tomando decisiones en base a los resultados de estas operaciones.

La estructura de directorios que se siguió para la organización de estas imágenes y sus respectivos valores se puede observar en la **[Figura 5: "Árbol de directorios"]**

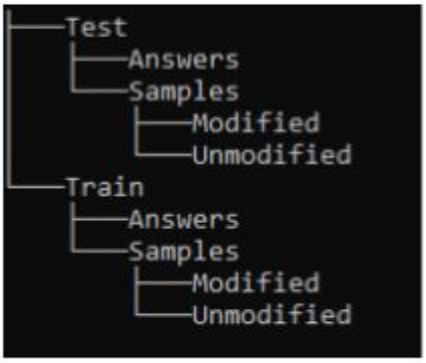

Figura 5: "Árbol de directorios."

Principalmente las muestras generadas que realmente nos interesan, y las encargadas de entrenar a nuestra red neuronal, se encuentran en las subcarpetas "Modified" dentro "Samples" dentro de "Test" y "Train". Estas imágenes son clasificadas como modificadas ya que se les ha aplicado un ruido aleatorio y una ligera rotación también aleatoria con el fin de simular las condiciones de procesamiento más reales posibles.

Sin embargo para poder realizar el entrenamiento de la red necesitamos mantener los valores de los elementos que representan las imágenes, esta información se recoge dentro de ficheros de texto que se encuentran en las subcarpetas "Answers" y cuyo identificador numérico en el nombre coincide con el del nombre de la imagen a la que representa.

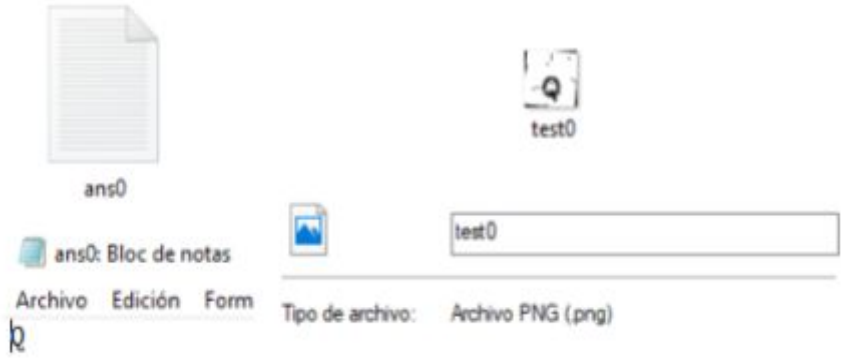

Figura 6: "Prueba de coincidencia."

Es importante explicar que nuestra red neuronal no va a ser capaz de reconocer el texto del identificador entero, sino que recibirá los segmentos que contienen cada elemento (letra o número) como imágenes por separado y las irá procesando de manera independiente, produciendo la salida de cada elemento, para generar al final la cadena completa al concatenarlos todos.

Una vez habiendo planteado el enfoque de la solución y el método a utilizar nos falta por especificar un último factor de gran importancia en este campo.

¿Que tipo de red neuronal utilizaremos y cómo está compuesta su arquitectura interna?

La respuesta q esta pregunta se encuentra en el apartado siguiente, donde hablaremos a fondo acerca de las herramientas y las tecnologías utilizadas.

Herramientas y tecnologías:

El desarrollo del proyecto ha conllevado la utilización de diferentes tecnologías, ampliamente usadas hoy en día en el campo de la informática. Aunque en principio la implementación de la solución pueda parecer que solo se ve afectada por el uso de herramientas relacionadas con el tratamiento de imágenes (como son las librerías de OpenCV o la realización de OCR a través de Tesseract) en este caso, como ya se ha introducido en el apartado anterior, se han aplicado herramientas y tecnologías cuya funcionalidad abarca un campo mayor al que se han limitado en esta implementación.

Hablamos en este caso del uso de redes neuronales y estrategias de Deep Learning aplicadas al reconocimiento de los caracteres que conforman el identificador de las cartas.

Si bien esta tecnología ha sido utilizada en este caso para este fin, su funcionalidad es ampliamente mayor que eso y puede ser utilizada en todo tipo de aplicaciones que requieran procesamiento y tratamiento de información de la que busquemos obtener una conclusión (identificar características, patrones, generar resultados en base a dichos patrones detectados), ya que una de sus características más llamativas es la versatilidad que ofrecen este tipo de estructuras de procesamiento. Dependiendo su utilidad únicamente de la arquitectura interna (tipos de capas, números de neuronas, forma de preprocesar la información de entrada, forma de representar la información de la salida) que utilicemos para formar la red de neuronas virtuales.

Una vez habiendo dejado claro que las herramientas utilizadas van más allá del uso de simples librerías de tratamiento de imágenes se realizará una explicación detallada de cada una de las herramientas más importante aplicadas durante el desarrollo del proyecto.

Empezaremos hablando acerca de OpenCV y las funcionalidades que proporciona. OpenCV es una librería de software libre cuyas principales funciones abarcan los campos de la visión por computador y machine learning. Una de las principales características que nos ofrece es la independencia de la plataforma en la que queramos utilizarla (puede ser usada tanto en Python, como C++, Java, etc.) y nos permite desarrollar aplicaciones que pueden ser ejecutadas en la mayoría de los sistemas operativos actuales.

Hoy en día, hablar de visión por computador y pensar en OpenCV como herramienta principal para esa tarea es una acción extremadamente normal, hecho que refuerza la comunidad de más de 50.000 usuarios enfocada al desarrollo y utilización de esta herramienta.

El proceso necesario a seguir para poder utilizar OpenCV en nuestro proyecto fue bastante sencillo. Únicamente hay que instalar el módulo correspondiente en nuestro entorno de desarrollo a través del comando **“pip install cv2”** y a continuación importarlo a nuestro código.

Figura 7: "OpenCV."

Otra herramienta utilizada para realizar la solución ha sido el uso de Tesseract como motor del OCR en la versión más tradicional de la aplicación. Sus funcionalidades nos permiten realizar el reconocimiento de forma sencilla y rápida, con un nivel de precisión bastante elevado. Para poder hacer uso de él únicamente debemos instalar en el entorno de desarrollo el módulo ocupado de empaquetar sus funcionalidades en Python. Esto lo haremos a través del comando **“pip install py-tesseract”**.

Además de las funcionalidades de OpenCV y Tesseract también se han utilizado módulos cómo **“matplotlib”, “numpy”, “scipy”** entre otros, para poder realizar las diferentes operaciones matemáticas necesarias además de trabajar con vectores y estructuras de datos compuestas.

Hasta este punto se recogerían las diferentes herramientas utilizadas en el desarrollo de la primera fase de la aplicación. A continuación se indicarán las que han sido aplicadas en la versión resuelta a través del uso de redes neuronales.

Obviamente, el núcleo del desarrollo en esta versión se compone de la utilización de las redes neuronales como medio para solucionar el problema. Así que debemos explicar su funcionamiento teórico y la arquitectura implementada, desarrollando el motivo de esta decisión y valorando su funcionalidad y potencial.

La aplicación de redes neuronales a problemas modernos es un enfoque que el campo de la informática lleva experimentando cada vez más en aumento por parte de los investigadores durante los últimos años. La funcionalidad de estas estructuras de datos complejas nos permiten simular funcionamientos parecidos a los que tendría el cerebro humano, generando a partir de unas entradas unas suposiciones de un número de posibles resultado marcado por el tipo de red que se utilice y sus capas de salida.

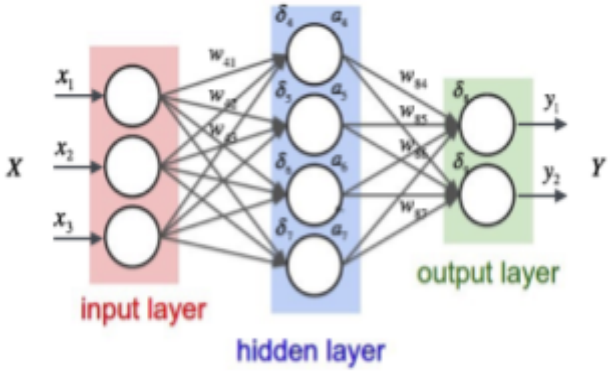

Figura 8: “Red Neuronal”

Existen multitud de diferentes aplicaciones de esta tecnología a través de diferentes tipos de redes neuronales. Estas diferencias son principalmente generadas por la forma en la que los datos de entrada se procesan y se generan las “conclusiones”. En algunos casos la solución que queremos generar se adapta mejor a un tipo de red neuronal específica pudiendo llegar a ser prácticamente imposible resolverlas en otro tipo de red distinta. Algunos ejemplos de los diferentes tipos de redes neuronales que existen son las Redes Convolucionales, las Redes Neuronales Recursivas o las Redes Neuronales Reiterativas.

El primero de los tipos nombrados, las Convolucionales, fue el elegido para dar solución a nuestra problemática. Ya que su funcionamiento es perfecto para la tarea que queremos realizar.

Antes de entrar a explicar como funciona internamente una red neuronal Convolucional y desarrollar sus características, comentaremos por qué esta se adapta tan bien a nuestra aplicación.

Al hablar de reconocimiento de texto en imagen debemos tomar entonces estas imágenes como datos de entrada de la red neuronal. Las operaciones que se deben realizar sobre ellas y el comportamiento resultante de estas será lo que determine las variaciones de pesos en las neuronas, provocando al final la naturaleza de las predicciones. Dicho esto entonces, queda claro que las operaciones a realizar sobre los datos de entrada deben ser capaces de abstraer información a nivel espacial, ya que en nuestra imagen de entrada aunque se trate de una matriz de píxeles que varían dependiendo del color, el orden de los mismos es intrínseco a la imagen. Cada fila y cada columna posee unos valores específicos y su posición dentro de la matriz general, en conjunto con el resto de columnas y filas, es lo que compone la imagen. Si moviesemos alguna de estas filas a otra posición de la matriz la imagen sería otra completamente distinta y, a lo mejor en una imagen de resolución elevada en la que podemos tener millones de píxeles esta variación no provoca ningún cambio real visible para nuestro ojo, pero en imágenes con dimensiones pequeñas, como son las que estamos usando para trabajar, puede convertir una letra en un símbolo completamente ilegible.

Es por este motivo por el que las redes Convolucionales se adaptan tan bien a nuestra solución, ya que su funcionamiento es capaz de abstraer diferentes características en secciones espaciales diversas de la imagen.

Ahora que entendemos el motivo por el que hemos elegido este tipo de red lo ideal es desarrollar una explicación acerca del funcionamiento interno de esta y explicar cómo hemos adaptado nuestra solución a través de la arquitectura personalizada que se ha implementado. A continuación, en la **[Figura 9]** podemos observar una representación gráfica de nuestro modelo, el cual procederemos a explicar a fondo en breve.

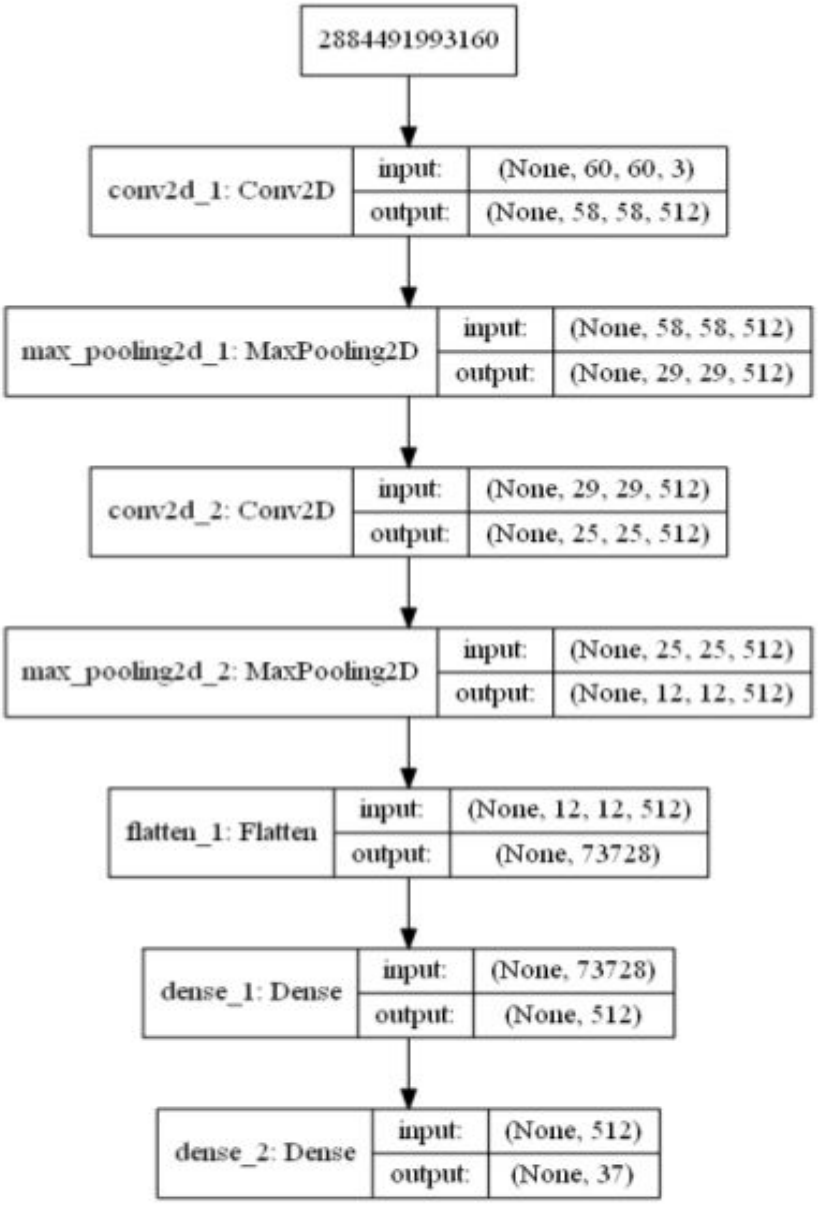

Figura 9: "Nuestro modelo"

En primer lugar lo más importante acerca de nuestro modelo es el tipo de entrada que recibe. Es capaz de procesar imágenes de 60x60x3, como podemos observar en la primera capa de la imagen.

Esta primera capa realiza la operación por excelencia de nuestra red y a la que debemos el funcionamiento adecuado, es la que identifica a nuestra red como Convolucional y efectivamente la operación que realiza es una convolución sobre la imagen.

Ahora bien, **¿Qué es una convolución?** Una convolución podría ser definida como una abstracción de la información que contienen diferentes regiones de la imagen, aplicando operaciones matemáticas, principalmente multiplicaciones, entre los elementos de la región y los que se encuentran en una posición equivalente en diferentes filtros que se generan a la hora de realizar este proceso. Este proceso nos genera imágenes de menor tamaño, aunque no demasiado, ya que los bordes al verse afectados por estas operaciones matemáticas no tienen el número de elementos necesarios para poder operar, por lo que se acaban obviando pequeñas secciones.

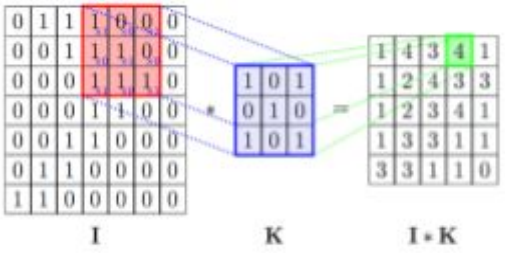

Figura 10: "Operación de convolución"

Es en el resultado de la convolución donde se abstrae por primera vez los diferentes patrones que puede presentar la imagen y algunas de las característica de los elementos, como pueden ser líneas o curvas. Las características de la capa a nivel de implementación podemos verlas en la **[Figura 9]**. De la entrada de imágenes de 60x60x3 obtenemos un total de 512 neuronas formadas por capas de matrices de 58x58. Este número de neuronas es lo que se conoce como hiper parámetro y es un valor variable a decisión del diseñador de la red neuronal, su valor debe variar en función de la eficiencia que muestre la red y ser modificado para mejorarla.

La siguiente capa en nuestro modelo se ocupa de simplificar la salida de la capa anterior, si bien en la salida de la convolución tenemos una matriz con características, lo ideal es reducir sus dimensiones para que sea más fácil trabajar con ella y poder aplicar nuevas capas sobre este resultado Para ello aplicamos la función de Max Pooling. Este proceso nos genera una matriz bastante más pequeña pero que mantiene independientemente del tamaño la representación de las características generadas por la convolución. En la **[Figura 11]** podemos observar una representación gráfica de este proceso.

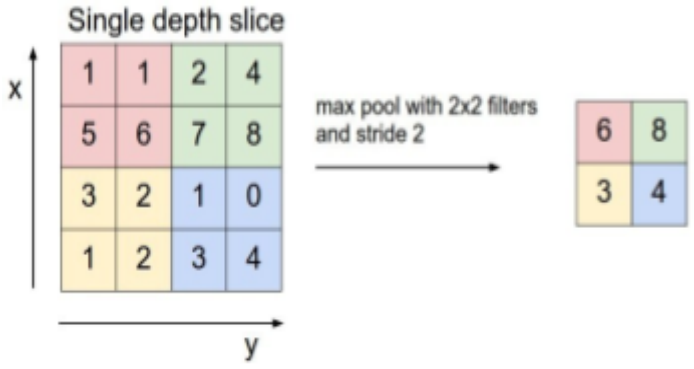

Figura 11: "Max Pool"

La idea que subyace en este proceso es "agrupar" la información que ha sido generada anteriormente. Para ello se elige el tamaño de las regiones que se van a agrupar de la capa anterior y se suman los elementos que componen estas regiones, separadas por un factor también decidido en la implementación. En el caso de nuestra red las regiones son de 2x2 la separación entre cada una de estas regiones también viene definida en la implementación y es de 2 píxeles entre cada.

El resultado de aplicar nuestra función de Max Pooling nos genera el mismo número de neuronas, sin embargo, estas son bastante más pequeñas, como podemos observar en la **[Figura 9]** con unas dimensiones de 29x29.

El proceso continúa aplicando nuevamente una capa convolucional. Esto es necesario ya que la primera capa únicamente nos genera características muy simples, por lo que necesitamos demasiadas combinaciones para generar un resultado y el proceso de entrenamiento sería extremadamente más lento. Así que aplicamos otra vez la convolución a la salida de la primera capa Max Pooling, generando así unas nuevas 512 neuronas que dependen de las primeras 512 y de las cuales podemos diferenciar en cada una una característica diferente recogida por la red neuronal.

Por consiguiente, después de generar estas abstracciones por segunda vez, aplicamos de nuevo un simplificación de la salida generada reduciendo el tamaño de los resultados a matrices de 12x12, todo esto a través de una segunda capa Max Pool.

A continuación nos encontramos con lo que denominamos como la última capa de las capas escondidas. Cómo podemos ver en la **[Figura 8]** las capas escondidas o **"hidden layers"** son aquellas que conforman el procesamiento interno de nuestra red neuronal. Realizan operaciones que a nivel del usuario que utiliza la red no tienen ningún sentido real, ya que las conclusiones que se generan en base a las variaciones de estos resultados no tiene ningún significado util hasta que se materializan en la capa de salida u **"output layer".**

Esta penúltima capa, la cual se trata de una capa densa se ocupa de recoger la salida de la segunda función Max Pool. En este tipo de capas, también denominadas **"Fully connected"** simplemente se generan resultados dependiendo de los valores que se obtienen en la entrada. La función que se ocupa de conectar con las salidas genera estas conexiones en base a las entradas de forma directa y lineal, el funcionamiento sería el mismo que el de un circuito lógico, al activarse con pesos mayoritarios ciertas entradas

las salidas se activan automáticamente.

Por último la capa final es la ocupada de transformar la información que la red neuronal puede generar en un dato que nosotros podamos recoger y formatear para trabajar. Es una capa densa con 37 salidas y cada una de estas determina uno de los diferentes elementos que queremos reconocer, teniendo por un lado a los números del 0 al 9 y por otro a las letras mayúsculas de la A a la Z. El resultado que se produce a partir de una predicción es un vector codificado como **"one-hot" [Figura 12]** lo cual significa que únicamente se ve activada como valor 1 la posición del elemento que representa, mientras que el resto se mantiene a 0.

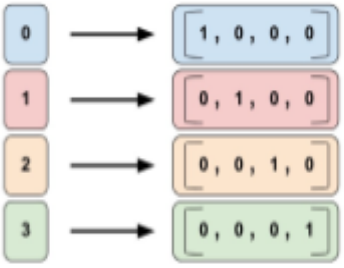

Figura 12: "One-Hot coding"

Habiendo desarrollado y explicado la arquitectura de nuestra red neuronal a nivel teórico debemos especificar también cuales han sido las tecnologías y herramientas que nos han permitido realizar este diseño.

Principalmente toda la implementación se ha realizado en base a Keras.

Pero, ¿Qué es Keras?

Keras es una API de alto nivel para Python diseñada explícitamente para trabajar con redes neuronales. Su nivel de abstracción abstracción permite realizar diseños de forma bastante sencilla y ágil. La mayoría de los métodos son intuitivos y permiten una rápida comprensión de sus funcionalidades, tipos de parámetros y variaciones en base a preferencias del diseñador. Además de esto contamos también con una amplia documentación acerca del uso de esta API en internet, no solo por parte de los creadores de Keras en su página oficial, sino en infinidad de foros y repositorios online. Aunque Keras nos sirva como interfaz para el desarrollo a alto nivel la implementación real se sustenta sobre tres posibles Backend en la actualidad. Podemos elegir entre usar TensorFlow, Theano o CNTK.

En nuestro caso el Backend utilizado ha sido TensorFlow, el cual procederemos a explicar a continuación.

TensorFlow es un framework de software libre diseñado por Google con el fin de permitirnos trabajar con tensores y manipularlos.

¿Qué es un Tensor?

Un Tensor es un elemento de dimensión variable, que mantiene información y datos acerca de las operaciones que se realizan en nuestra red. Existen diferentes tipos de tensores dependiendo de sus dimensiones:

- Un tensor de rango 0 es un escalar.
- Un tensor de rango 1 es un vector.
- Un tensor de rango 2 es una matriz.
- A los tensores de rango 3 se les denomina Tensor y son elementos 3D con coordenadas X, Y, Z.

Esta versatilidad a la hora de elegir su rango nos proporciona múltiples opciones en cuanto a cómo queremos representar la información con la que estamos trabajando y que tipo de operaciones vamos a realizar sobre ella. Lo más importante es tener en cuenta que la información de salida no es la que se mantiene guardada directamente en el tensor, ya que no se trata de una estructura de datos convencional, sino que lo único que designa el tensor es el tipo de relación entre la entrada y la salida. Algunos de los tipos de relaciones que se producen son el producto escalar o el producto vectorial. En el caso de nuestra red, podemos observar cómo estas operaciones se utilizan durante el procesado de la información, especialmente el producto escalar a la hora de realizar la convolución. Podemos ver una representación gráfica en la **[Figura 13]**.

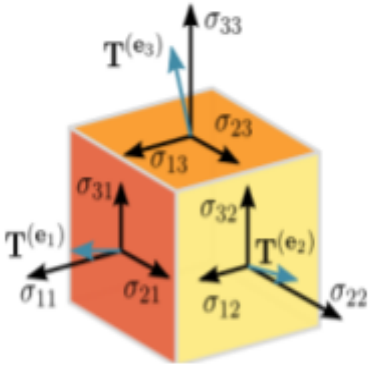

Figura 13: “Tensor”

Para utilizar Keras en nuestro proyecto simplemente debemos instalar el módulo correspondiente a través de **“pip install keras”** e importarlo en nuestro código.

Objetivos

En este apartado desarrollaremos los diferentes objetivos que se han planteado dentro del proceso de implementación del proyecto. Si bien los objetivos están bastante relacionados con las diferentes fases y versiones que tienen el proyecto, alguna de estas fases ha englobado más de un objetivo específico.

Objetivos previos. Visión en el Anteproyecto

En la siguiente tabla se pueden observar los objetivos que se marcaron en el planteamiento previo del proyecto:

Tarea
Tratamiento de imágenes con OpenCV
Generación de imágenes de muestra
OCR en OpenCV
Tesseract OCR
Diseño e implementación de una red neuronal aplicada
Validación del desarrollo a través de pruebas

La mayoría de los objetivos han sido completados y validados a través de las pruebas que se recogen en las diferentes figuras que aparecen en la memoria. Sin embargo alguno de estos objetivos ha sido ligeramente modificado y adaptado para facilitar la implementación de la aplicación.

En primer lugar, la realización del OCR con OpenCV ha sido relevada a una mejora que se realizará posteriormente, hablaremos de ella en el apartado de líneas futuras.

Por otro lado, el diseño de la red neuronal supuso una puerta abierta hacia un gran número de posibles aplicaciones diferentes y maneras de enfocar el desarrollo. Entre estas se encontraba la aplicación de redes neuronales reiterativas a través de LSTM con el fin de solucionar ciertos inconvenientes que encontramos usando las convoluciones. Sin embargo este matiz ha sido también prorrogado a una mejora futura ya que la naturaleza del problema no se adapta tan bién a este tipo de red neuronal y la mejora, aunque existente, sería mínima.

Dejando de lado estas dos tareas, el resto han sido completadas con éxito.

Fases y desarrollo del proyecto

Como ya se ha explicado en los anteriores apartados de esta memoria el proyecto está dividido en tres fases diferentes. Empezando por un periodo de preparación de las imágenes de prueba y seguido por cada una de las dos aplicaciones desarrolladas para resolver nuestra problemática. A continuación desarrollaremos extensamente cada una de estas fases a nivel de implementación y hablaremos acerca del código y las funcionalidades diseñadas.

Fase previa al desarrollo. Visual Studio

Toda la implementación del código que compone nuestro proyecto ha sido realizada a través de Visual Studio 2017. Se ha elegido este entorno de desarrollo debido a la potencia y utilidades que posee, siendo una de las opciones principales a la hora de desarrollar en Windows.

Una de las funcionalidades que nos permite realizar Visual Studio es elegir qué entorno de Python queremos utilizar para la implementación de nuestra aplicación **[Figura 14]**. Esto es fácilmente configurable a través de la interfaz gráfica y el único requisito existente es tener instalada la versión de Python que queramos usar en nuestro ordenador.

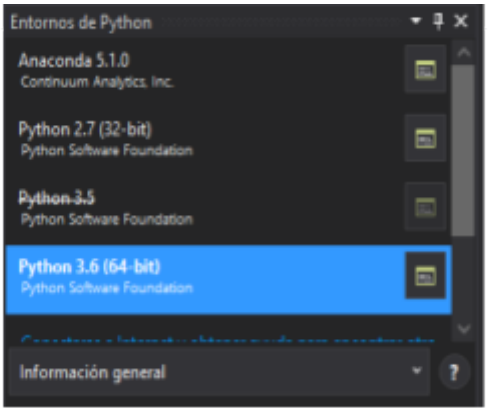

Figura 14: "Entornos de Python"

En nuestro caso la versión de Python utilizada ha sido la versión **“Python 3.6 (64-bit)”** ya que muchas de las operaciones que queríamos desarrollar contaban con mayor soporte en la versión 3 del lenguaje, frente a la 2 que es la otra opción a elegir a la hora de desarrollar en Python. Eligiendo la versión 3 como base para nuestro proyecto, con la 3.6 como última actualización, únicamente necesitamos crear nuestro proyecto y empezar a trabajar.

Sin embargo el desarrollo de la solución no está aglomerada en un único proyecto, ya que las diferentes funcionalidades y los diferentes objetivos nos han obligado a modularizar y por consiguiente el diseño se divide en los siguientes proyectos:

- PillowImageGenerator: El código comprendido en este proyecto se ocupa de generar los diferentes tipos de imágenes de prueba.
- KerasNN: Este proyecto sirve como fundación y generación de nuestra red neuronal, pudiendo modificar el código que contiene para adaptar la red a nuestras preferencias.
- OCRTry: Éste es el proyecto “global” y el que se ocupa de realizar el reconocimiento de caracteres. El código que lo compone tiene diferentes tipos de métodos dependiendo de si queremos realizar el OCR a través de la tecnología de Tesseract o queremos crear una red neuronal y trabajar con las predicciones que realice.

Figura 15: “Carpeta del proyecto OCRTry”

Como podemos comprobar en la **[Figura 15]** en la carpeta del proyecto OCRTry, además de guardar el código tenemos diferentes subcarpetas que recogen imágenes para pruebas, como en el caso de la carpeta Reales, que posee imágenes de algunas cartas reales para poder comprobar que el OCR efectivamente se realiza correctamente sobre estas. Entraremos más a fondo en el funcionamiento específico de este proyecto al hablar de la fase que le corresponde. Pero por ahora, comenzaremos con el desarrollo del primero de los proyectos, PillowImageGenerator, ya que es el que se ocupa de realizar la primera fase del desarrollo, la creación de imágenes de prueba.

Fase de creación de imágenes de prueba

La funcionalidad principal de este proyecto, como ya explicamos en apartados anteriores, ha sido la de servirnos como generador de las imágenes de prueba que utilizamos para validar nuestra aplicación. Además, a parte de esto, también lo hemos utilizado para generar los diferentes tipos de imágenes de entrenamiento y test para las fases de preparación y aprendizaje de la red neuronal. Es importante tener en cuenta que para esta fase también tuvimos que recoger los valores que representan las imágenes, ya que a la hora de realizar el entrenamiento y validar si las predicciones son correctas son estos valores los que lo determinan.

Los principales módulos utilizados en este proyecto han sido OpenCV y Python Image Library, además de OS y Numpy, entre otros.

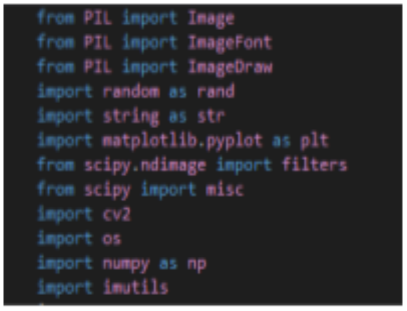

Figura 16: "Módulos utilizados en PillowImageGenerator"

El código se compone de diferentes métodos, diseñador para realizar funciones diferentes aplicadas a cada una de las versiones de la aplicación que se han implementado. La generación de las imágenes se divide en 4 métodos diferentes dependiendo del tipo de imagen que queramos generar. En primer lugar tenemos el método "**create_imgText(iter)"** el cual a través de un entero iterador que le pasamos por parámetro nos genera dos tipos de imágenes con las siguientes estructuras:

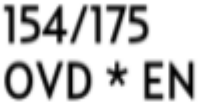

Figura 17: "Imagen generada base"

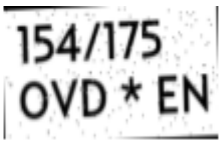

Figura 18: "Imagen generada modificada"

Como podemos observar, una de las versiones ha experimentado una modificación, añadiendo ruido y provocando una ligera rotación. Esto se hace para intentar simular condiciones lo más reales posibles a la hora de aplicar el reconocimiento, poniendo pequeños inconvenientes, demostrando que aún así nuestra aplicación es capaz de resolver el problema planteado.

Además de estas dos imágenes también se genera un fichero de texto con la respuesta correcta, es decir, con el texto que aparece en la imagen, para que nuestra red neuronal a la hora de aplicar el entrenamiento pueda comprobar si efectivamente la predicción ha sido acertada. Obviamente esta respuesta no se le tiene que introducir al sistema una vez que el entrenamiento ya haya sido realizado y funcione correctamente, ya que en ese momento la aplicación es capaz de reconocer correctamente los caracteres y no necesita validación.

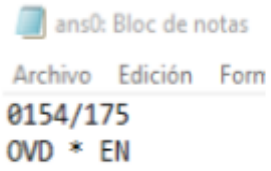

Figura 19: “Respuesta”

La imagen que se utiliza como lienzo para generar estas imágenes es un archivo png completamente en blanco, de tamaño 200x100. El proceso que se sigue es el siguiente:

Se obtienen dos números completamente aleatorios entre 0 y 200, pero siempre evitando que el primero sobrepase al segundo. Estos representarán los números que aparecen en la primera línea de la imagen. Ya que se trata de un identificador de un número dentro del total de cartas de una edición el primero siempre debe ser menor o igual que el segundo.

Habiendo elegido estos números continuaremos seleccionando tres letras aleatorias, en mayúsculas, que representan el nombre de la edición a la que pertenece la carta. Este formato es el mismo que se utiliza en las cartas reales así que la simulación de los datos es bastante cercana a la información real. Por último se comprueban los números elegidos y se formatea la cadena de texto en función a cuáles fueron estos, ya que en caso de haber obtenido números de menos de 3 cifras tenemos que rellenar con ceros y así mantener el formato estándar en todas las imágenes.

Con toda la información preparada generamos la imagen y escribimos en ella, para separar el nombre de cada una de las pruebas utilizamos el iterador que recibimos como parámetro, haciendo que cada fichero tenga un nombre diferente en función del orden en que fueron creados. También escribimos la respuesta, siendo esta la misma cadena escrita en la imagen, pero en los ficheros de texto también diferenciados por el iterador.

Las dos siguientes funciones que comentaremos son prácticamente una misma versión diferenciada únicamente por el tipo de elemento que escriben en las imágenes. Aunque entraremos a hablar más a fondo en el desarrollo y las fases que experimentó nuestra implementación de la red neuronal en los siguientes apartados necesitamos entender que el funcionamiento y la manera en la que se quería introducir la información a la red neuronal no fue siempre la misma que ahora.

En un principio, la aplicación se diseñó para realizar identificaciones de números y letras por separado, ya que una de las primeras ideas era separar cada línea del identificador y obtener dos resultados de dos redes neuronales diferentes, una de números del 0 al 300 y otra de letras del abecedario. Esta idea se acabó descartando y se implementó una red conjunta que reconoce ambos tipos de elementos y, aunque en realidad las funciones ocupadas de generar estas pruebas e imágenes de entrenamiento no se usan, supusieron una fase importante en el desarrollo de este segmento del proyecto por lo que es interesante nombrarlas y explicar su función.

Estos métodos son **"create_dataletras"** y **"create_datanumeros"**. Cada uno de ellos con una versión diferente para generar imágenes de entrenamiento o imágenes de prueba.

El funcionamiento que esconden estos métodos es muy similar al que presenta **"create_imgText"** diferenciándose únicamente uno del otro por el tipo de elemento que escriben en la imagen, siendo uno números y el otro letras pero manteniendo la generación de las versiones base sin modificar, las modificadas y las respuestas en modo texto.

Figura 20: "create_dataletras"

Figura 21: "create_datanumeros"

Por último, el método que nos falta por comentar dentro del proyecto de creación de imágenes es el método **"sp_noise(image, prob)"**.

A través de este método, enviando por parámetro una imagen y una probabilidad conseguiremos una imagen resultante con un ruido **"Salt and Pepper"** aplicado sobre ella. Todas las imágenes modificadas sufren una transformación a través de este método aunque se reserve una copia de la imagen sin modificar. El proceso que se realiza en este método es bastante sencilla y únicamente se trata de una conversión de píxeles a blanco o negro dependiendo el número de píxeles de la probabilidad que le indiquemos por parámetro, a mayor probabilidad mayor porción de la imagen será afectada.

Fase de OCR con librerías de visión por computador

Habiendo explicado cuál ha sido el desarrollo llevado a cabo en la fase preparación de las imágenes se procederá a continuación a explicar la implementación de la versión del proyecto que realiza el OCR a través de las librerías de OpenCV y Tesseract.

El proyecto que contiene esta solución se llama **“OCRTry”** y además de este enfoque a la solución también podemos encontrar el que aplica el uso de redes neuronales dentro del código. Cada uno de estos enfoques está representado como un método dentro del proyecto de Visual Studio y es ejecutado en base a las preferencias del modo que se quiera aplicar.

En cuanto a la parte del proyecto que nos incumbe al hablar de la fase aplicada únicamente a través de OpenCV y Tesseract podemos distinguir cinco métodos que en conjunto realizan todo el proceso de reconocimiento de texto, tanto en imágenes de prueba como en imágenes de las cartas reales.

En primer lugar, el método que se ocupa de ir realizando cada parte del procesado, llamando uno a uno al resto de los métodos es **“OCR(imgCarta, name)”** el cual recibe por parámetro una imagen y su nombre, y nos devuelve el texto que contiene.

En el caso de las cartas reales es importante realizar un filtrado de la información innecesaria, lo cual significa básicamente quedarnos con una región de interés, la cual posee en su interior el identificador que queremos traducir a texto. En caso de trabajar con imágenes de cartas reales entraríamos a hacer uso de la primera sub llamada al método **“RoInterestCartaReales(input)”** que, recibiendo una imagen como parámetro de entrada es capaz de, a través de operaciones utilizando medidas relativas al tamaño de la imagen a recortar, devolvernos una sección de la carta que contiene únicamente el identificador que nos interesa. Esto es posible ya que este identificador siempre se encuentra en la misma región de todas las cartas, independientemente de su edición (en la esquina inferior izquierda) y al utilizar medidas relativas, siempre y cuando la imagen de la carta se limite a la carta en sí y no tenga ningún otro fondo (caso bastante factible a través de un preprocesado en caso de no cumplir estas características, haciendo un recorte de la silueta de la carta y quedándonos con esa región), el corte siempre devuelve la misma sección.

Figura 22: “Región de Interés”

Una vez que tenemos la región de interés sobre la que queremos realizar el reconocimiento aplicaremos un binarizado a través del método **"Binarization(imgCarta)"** el cual nos devuelve una imagen completamente binaria pero igual a la que se le pasó como parámetro y cuyo funcionamiento es el siguiente. Dentro de la imagen que se va a transformar se comprobarán los píxeles que la conforman y a través de un umbral que define únicamente como valores válidos el blanco y el negro se irán convirtiendo todos los que no cumplan esta condición a bien blanco o negro dependiendo de cuál esté más cerca en la escala cromática. En el caso de la **[Figura 21]** parece que este proceso no es necesario, y en parte es cierto, ya que el borde de la carta es completamente negro y las letras blancas, pero existen modelos de cartas que tienen degradados de colores, simulación de texturas y letras de diferentes colores al blanco en los bordes, por lo que a la hora de realizar la identificación podemos vernos afectados negativamente.

A continuación se genera un kernel para poder aplicar operaciones morfológicas sobre la imagen, con la intención de eliminar el ruido y las imperfecciones que puedan aparecer. El proceso realiza una erosión con la intención de que los píxeles de ruido desaparezcan y así poder aplicar el reconocimiento con mayor exactitud.

Una vez que la imagen se encuentra binarizada y completamente libre de ruido se aplica una detección de contornos con la intención de separar los diferentes segmentos del texto, es decir, cada letra y número. Esto se genera a través de la llamada a la función **"cv2.findContours(input, mode, method)"**.

Esta función es un método interno de OpenCV y nosotros únicamente tenemos que llamarlo, el diseño e implementación ya está realizado.

Al realizar la detección de contornos recibimos una lista que contiene los datos para generar las **"box"** que definen cada área de contorno detectada por el método. Esto se realiza a través de la llamada a la función, también perteneciente a la librea de OpenCV, **"cv2.minAreaRect".** La cual nos genera un cuadrado alrededor del elemento que hemos detectado y nos indica su centro, dimensiones y angulación. Este cuadrado tiene la característica de poseer el área mínima necesaria para contener al elemento en cuestión.

El proceso aplicado a continuación es un bucle a lo largo de todo este vector de cajas, pudiendo dibujar el contorno del elemento si se desea a través de la función **"cv2.drawContours"**. El objetivo de iterar sobre estos elementos es poder recortar cada uno y separarlo en una imagen independiente, mucho más sencilla de reconocer. Para ello utilizamos la función **"skewAndCrop(input, box)"**, la cual entraremos a explicar más adelante.

Ahora que ya hemos conseguido separar esta imagen simplemente aplicaremos la función **"Image_To_String(input, lang)"** para obtener el resultado como cadena de texto.

Esta función pertenece a la librería de Tesseract y no ha tenido que ser implementada, sino simplemente puede ser utilizada importando el módulo de Tesseract al proyecto, como ya explicamos anteriormente. La función recoge una imagen y una cadena de texto que identifique al idioma que queremos tomar como estándar para que a la hora de realizar el reconocimiento de texto la gramática y elementos usados tengan sentido.

Todo parece bastante sencillo pero, ¿Cómo se realiza internamente el reconocimiento? Existen multitud de diferentes maneras de enfocar la solución a este problema, por ejemplo, la detección de patrones como pueden ser líneas que componen

los caracteres y cómo en conjunto estos patrones determinan el carácter que estamos reconociendo. Otro acercamiento a cómo realizar el reconocimiento de los caracteres se basa en utilizar un reconocimiento de las áreas que activan ciertos caracteres al aplicar un filtro de detección de regiones sobre ellos, creando estos filtros a través de las fuentes y tipografía típica.

Dejando de lado la explicación del funcionamiento interno del OCR es necesario añadir un pequeño apunte al proceso realizado en la función **"OCRTry"** en nuestro proyecto. En caso de que la imagen no detecte ningún contorno se llamará a la función de Tesseract igualmente para intentar aplicar el OCR sobre toda la imagen, ya que al ser imágenes binarizadas y libres de ruido, tampoco suponen un problema demasiado grande a la hora de procesarlas de una pieza. El resultado de la llamada a la función del OCR podemos observarlo en las siguiente figuras:

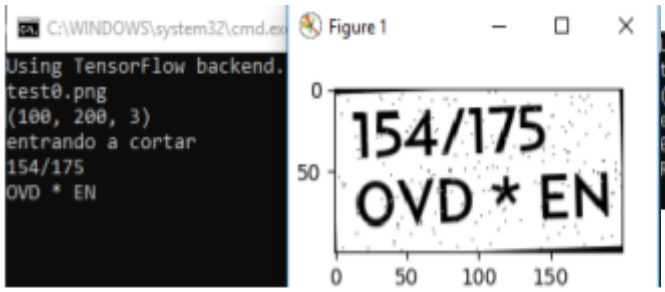

Figura 23: "Ejecución sobre imagen generada"

Figura 24: "Ejecución sobre imagen real"

Los elementos que restan por explicar en el proceso que se sigue en esta fase del proyecto son **"mostrarImg(input)"** y **"skewAndCrop(input, box)".**

El primero de estos métodos simplemente es una encapsulación de las llamadas a las funciones de la librería Python Images **"imshow"** y **"show"**, añadiendo una espera por parte del programa al final para confirmar que la ventana de la imagen se ha cerrado antes de continuar con la ejecución.

En cuanto a la función **"skewAndCrop"**, aunque ya hemos comentado que se ocupa de generar las diferentes imágenes de cada uno de los cuadrados que recogen los contornos de la imagen, expliquemos más a fondo cual es el proceso realizado para ello.

El método recibe dos parámetros de entrada, por un lado la imagen sobre la que queremos recortar y por otro la **"box"** que define el área del corte. Estas cajas son tuplas generadas por el método **"cv2.minAreaRect"** y agrupadas en la lista **"areas"** en la función **"OCRTry"** que contienen los tres datos necesarios para representarlas gráficamente dentro de la imagen. Estos datos son, el centro del área como posición absoluta en la imagen original, sus dimensiones en cada eje y el ángulo de inclinación que presentan. Una vez recogidos se asignan a variables auxiliares para poder operar con ellos ya que en Python las tuplas son estructuras de datos constantes y no podemos modificar sus valores.

La primera comprobación a realizar es el ángulo de rotación que presenta la caja, si se detecta que la caja está rotada por encima de un umbral se le suman 90º para rectificar este inconveniente y se aplica una transformación a través de la función **“cv2.warpAffine”** la cual nos permite realizar una transformación afín a la imagen utilizando como matriz de rotación el resultado de la función **“cv2.getRotationMatrix2D”**.

Por último, se obtiene la sección de la imagen que queremos recoger, denotandola como imagen propia a través de la función **“cv2.getRectSubPix”.**

Una vez realizado el recorte, este es retornado al método del OCR y se continúa con el proceso que ya fue descrito anteriormente.

Es en este punto donde termina la explicación acerca de los diferentes métodos diseñador para el OCR a partir de las librerías de OpenCV y Tesseract. A continuación se desarrollará la explicación referente a la solución planteada a través del uso de redes neuronales.

Fase de OCR con aplicación de Redes Neuronales

Si bien hemos comentado que la implementación de la solución está recogida dentro del mismo proyecto tanto para la que utiliza la aplicación de librerías de visión por computador como para la que se especializa en el uso de las redes neuronales, necesitamos separar en dos diferentes sub proyectos todo lo relacionado con el diseño e implementación de la red neuronal utilizada.

Por un lado tenemos el proyecto que engloba ambas soluciones, **“OCRTry”**, pero por otro tenemos el proyecto que nos ha servido como mesa de trabajo para experimentar y poder diseñar el modelo de la red neuronal ocupada del reconocimiento de caracteres. Este proyecto se llama **“KerasNN”** y como su nombre indica contiene todo el código necesario para realizar la implementación de diferentes modelos a través de la API de Keras.

Comenzaremos explicando y desarrollando cual es el contenido de este proyecto, y en qué funcionalidades se sustenta el código que lo compone.

En primer lugar, el código se puede dividir en tres grandes fases de ejecución. Comenzando por la lectura y preparación de los datos de entrada, en este caso leyendo las imágenes de los ficheros correspondientes y separándolas en los diferentes vectores de entrenamiento o test, dependiendo de su función. Estas son las imágenes que fueron generadas por el proyecto **“PillowImageGenerator”** y cada uno de los directorios que conforman la salida de esta generación de imágenes tiene una funcionalidad definida cómo se puede observar en la **[Figura 5]**.

Una vez que se tienen agrupadas las imágenes y sus valores correspondientes en los diferentes vectores del procesamiento (tomamos el convenio de denotar a los vectores con entradas a estudiar por **X_** y a los vectores que recogen las respuestas por **Y_** separándolos a su vez entre **train** y **test** en función de su objetivo) se aplica un proceso

de estandarización de información para que la red neuronal pueda trabajar correctamente.

El proceso consiste en convertir todas las imágenes a flotante y reducir el rango que puedan tener cada uno de los elementos de la matriz de la conforman a entre 0 y 1. Como las imágenes con las que trabajaremos van a ser de carácter cromático binario esto no supone ningún problema, ya que únicamente simplifica la información y reduce el coste computacional de trabajar con los datos al reducir su tamaño.

Otro procesado previo a trabajar con las imágenes en la red neuronal es convertir el vector que recoge las soluciones a un formato que nuestro sistema de predicciones pueda comprobar y contrastar.

Se supone que queremos reconocer cualquier número comprendido entre el 0 y el 9 además de todas las letras que conforman el abecedario. Para ello en el vector que recoge las soluciones únicamente guardamos enteros que representen estos elementos, en caso de los números es un proceso sencillo, ya que son ellos mismos. Pero en el caso de las letras la solución que aplicamos es empezar a contar a partir del 10 para la A y asignarle cada uno de estos valores a las letras siguientes obteniendo así un modelo de representación de la información que únicamente mantiene enteros. Esto es importante ya que nuestras predicciones se entienden como activaciones de uno de los 37 elementos de la capa final de nuestra red neuronal, lo cual se representa muy fácilmente con, como ya hemos explicado anteriormente, un vector **“one-hot”**. Teniendo en cuenta que las letras no pueden ser transformadas directamente este tipo de codificación, primero las convertimos a enteros y posteriormente aplicamos la función **“np_utils.to_categorical(V, N)”**. Esta función recibe el vector que queramos convertir a esta codificación y el número de elementos que queremos diferencias y nos retorna un nuevo vector que representa dicha información pero en este modelo de codificación.

Una vez que tenemos adaptadas las entradas de nuestra red neuronal para que el funcionamiento sea el correcto el siguiente paso es, sin duda, realizar el diseño y la implementación de cada una de las capas. Para poder definir así cuales son las operaciones que aplicamos sobre la información de entrada y que tipo de transformaciones experimentarán los valores de los pesos de las neuronas y por ende que tipo de conclusiones podremos generar finalmente.

Como el proceso que lleva a cabo cada una de las capas ya ha sido recogido en apartados anteriores a continuación únicamente realizaremos una explicación a nivel de código indicando las principales características que tiene nuestra implementación.

A la hora de diseñar un modelo de una red neuronal debemos especificar el tipo de modelo que se trata. En este caso nuestro modelo es un modelo secuencial, es decir, una sucesión de capas una detrás de la otra y a las que se conectan como entradas las salidas de las anteriores, todo esto hasta llegar a la capa final en la que se materializa la predicción. Para indicarlo, en el código únicamente debemos crear la variable de nuestro modelo y asignarle el valor de retorno de la llamada a la función **“Sequential()”** de la librería de Keras.

A partir de ahí la sintaxis utilizada es bastante sencilla y únicamente se limita a añadir, con la función **“model.add()”** las diferentes capas de nuestra red (entre los paréntesis, ya que se le pasan como argumento).

Podemos ver como funciona este proceso en la siguiente **[Figura 24]:**

```
model = Sequential()
model.add(Conv2D(512, (3,3), activation="relu", input_shape=(60,60,3)))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Conv2D(512, (5,5), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
```

Figura 25: "Primera parte de nuestro modelo"

Podemos observar cómo se aplica en cada capa de convolución una rectificación linear a través del parámetro **"activation= "relu""** . Esto simplemente indica el tipo de función que utilizaremos para decidir si una neurona debe ser activada o no. Existen diferentes variantes a este parámetro, como puede ser el uso de una función sigmoide **[Figura 25]** (a través de un factor de entrada se obtiene un resultado de forma no lineal entre -1 y 1, cuanto más nos acercamos al infinito, mayor es la relación con el 1 mientras que al recoger números muy negativos nos acercamos al -1 en la salida) o la función **"softmax"**, que es utilizada en la última capa para realizar la decisión del elemento final.

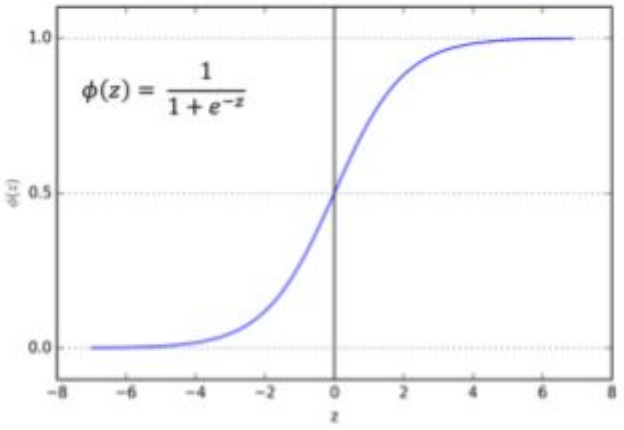

Figura 26: "Función Sigmoide"

En este caso la operación realizada por la rectificación lineal es la siguiente, se obvian los valores que no consideramos válidos y únicamente tenemos en cuenta aquellos que superan el umbral de decisión convirtiendo a la función que determina la salida en una función lineal. Podemos observar un ejemplo gráfico en la siguiente **[Figura 26]:**

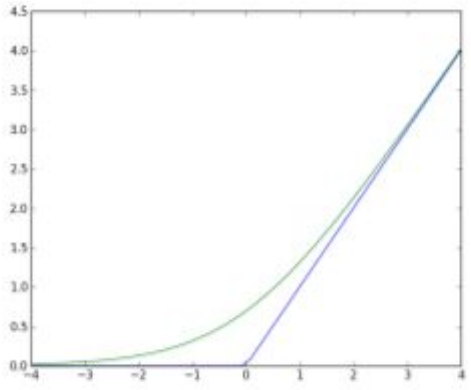

Figura 27: "Rectificación lineal"

El único detalle digno de mención en lo que resta de la implementación del diseño de nuestro modelo es la la adición de la capa **"Flatten"**. La funcionalidad de esta capa, se puede razonar al observar en la **[Figura 9]** la transformación que experimentan los datos que transitan por la red. Su utilidad es la de convertir el conjunto de información generado por las capas anteriores, hasta ahora tridimensional, a un solo vector del tamaño equivalente al número de columnas por filas por neuronas existentes.

Esto es necesario ya que la capa siguiente, la primera capa densa de nuestro modelo, únicamente puede recibir elementos con estas dimensiones, recibiendo uno por cada imagen que estemos procesando. Después de que el resultado de "aplastar" la información sea reconocido por la capa densa y se le aplique una activación a las neuronas que la conforman las conexiones con la última capa se activan, provocando la generación de nuestra predicción. La implementación de estas últimas capas podemos verla en la siguiente **[Figura 27]:**

```
model.add(Flatten())
model.add(Dense(512,activation='relu'))
model.add(Dense(37, activation='softmax'))
```

Figura 28: "Segunda parte de nuestro modelo"

A continuación, una vez habiendo creado nuestro modelo lo único que nos falta es compilarlo y entrenarlo. En la función **“model.compile”** indicamos los parámetros deseados para especificar que tipo de función de pérdida, optimizador y métricas queremos utilizar.

La función de pérdida es un factor bastante importante en nuestra red, ya que es la ocupada de indicar en cada iteración del entrenamiento el avance que realizamos valorandolo con un nivel de acierto indicándonos así, si el porcentaje de predicciones acertadas aumenta o decrece y por ende la eficiencia de nuestro sistema.

Una vez que hemos compilado el modelo, la operación de entrenamiento únicamente requiere la llamda al método **“model.fit”** donde debemos indicar como datos de entrada los vectores de imágenes y respuestas, el número de epochs (veces que se repetirá el entrenamiento) y el tamaño del batch a utilizar (número de elementos que se procesarán en cada iteración), además de, en caso de querer validar el entrenamiento, los vectores de imágenes y respuestas para pruebas. Para realizar la validación invocamos al método **“model.evaluate”**.

Ahora bien, habiendo entrenado a nuestra red, lo más importante es guardar el progreso realizado. Ya que si no, el entrenamiento se perdería y no serviría para nada, lo cual tratándose de un proceso tan lento y acaparador de recursos de la computadora que lo realiza no parece algo agradable.

Para poder mantener una copia de los valores de los pesos de las neuronas que conforman nuestra red utilizaremos la función **“model.save_weights”** indicando como parámetro de entrada el nombre del fichero donde vamos a guardar dichos pesos.

El modelo que hemos generado previamente también puede ser guardado. En nuestro caso se ha decidido recoger esta información dentro de un fichero **json** y exportar la arquitectura de la red a través del método **“model.to_json”**. Esto es bastante importante, ya que si en el proyecto que estamos explicando ahora mismo no se realiza ninguna operación referente al OCR si que es donde se entrenó y exportó la red que sería cargada en el proyecto ocupado de realizar el reconocimiento. Como se puede suponer, al igual que exportar y guardar todos estos datos en ficheros locales también podemos leer y cargarlos, para poder utilizar nuestra red en cualquier otro sitio.

Una vez que ya hemos explicado el funcionamiento interno de nuestra red y el método que utilizamos para cargar y guardar los valores generados por el entrenamiento procederemos a desarrollar cuál ha sido la aplicación de nuestra red dentro del sistema de OCR en el proyecto **“OCRTry”.**

En este segmento del proyecto podemos diferenciar dos grandes métodos que se ocupan de realizar las diferentes operaciones necesarias para la resolución del problema.

El primero de estos métodos es el llamado **“PreparарEntradas(input,name)”** y su funcionamiento es el siguiente:

Debido a las rotaciones provocadas sobre las imágenes de prueba para simular

condiciones reales nos encontramos que en muchos casos en estas imágenes aparecían secciones de fondo negro sobre el blanco para rellenar la sección que por culpa de la rotación ya no está.

Este tipo de artefactos en la imagen nos provocan bastante inconsistencia a la hora de poder encontrar los contornos de los elementos, ya que en muchas ocasiones se confunden como elementos del texto y se recogen involuntariamente. Para evitar este inconveniente recortamos la imagen ligeramente por los bordes eliminando esas secciones negras, usando nuevamente mediciones relativas al tamaño del fichero y así asegurarnos que la táctica es factible para cualquier tipo de imagen.

Una vez que tenemos la imágen preparada le aplicamos una binarización, al igual que como realizamos con la otra versión del OCR.

A continuación se introduce esta imagen en la función de reconocimiento de contornos, la cual nos devuelve los contornos de cada elemento que componen el identificador. Estos contornos se van recogiendo como cajas que serán las que contengan a los elementos del identificador, siguiendo un procedimiento prácticamente similar al que se sigue en la otra versión del OCR.

Lo siguiente es recortar cada uno de estos elementos y separarlos en imágenes distintas. Si bien en la versión del OCR anterior este procedimiento no era del todo necesario, ya que en la mayoría de los casos la imagen completa era reconocida correctamente por Keras en este caso sí que es estrictamente necesario. Esto es debido a que nuestra red neuronal es capaz de identificar letras y números a partir de una imagen del elemento en cuestión, pero no de identificar secuencias compuestas por varios elementos dentro de una sola imagen. Siendo esto así, la necesidad de separar cada elemento en una imagen independiente aumenta. Para ello utilizamos la función **“skewAndCropWithCords”**. Esta función puede parecernos bastante familiar, ya que el nombre es prácticamente igual al de la ocupada de hacer este proceso en la otra versión del OCR. Y efectivamente, las similitudes entre ambas son bastantes. Aún así, la diferencia, aunque única, es bastante importante y determinante a nuestra aplicación.

Esta diferencia es que, a la hora de retornar la imagen recortada devolveremos en conjunto las coordenadas que la sitúan dentro de la imagen original. Esto es bastante importante ya que a la hora de detectar los contornos el orden en el que van apareciendo no siempre sigue un patrón estricto, por lo que a la hora de recoger los elementos para introducirlos en la red neuronal nuestra secuencia de entrada no tiene ningún sentido real para nosotros.

A continuación una vez habiendo obtenido la salida de la función **“skewAndCropWithCords”**, debido a que el tamaño de las imágenes de entrada de nuestra red neuronal es fija (60x60x3) debemos obtener como imágenes de salida de **“PreparарEntradas”** imágenes con ese tamaño. Para ello aplicamos unos bordes de color blanco alrededor del segmento recortado con dimensiones que se calculan relativamente dependiendo de la cantidad de píxeles que le falten al recorte original para llegar a 60.

También es importante indicar que a la hora de retornar las imágenes de los elementos que vamos a reconocer, en vez de retornar todos los elementos que se encuentran se realizan dos operaciones previas con la intención de simplificar la tarea de la identificación.

Como anteriormente obtuvimos las coordenadas que sitúan estos segmentos dentro de la imagen original podemos separar entre elementos de la primera fila y la segunda si comparamos la coordenada Y de cada uno con la mitad del tamaño en altura de la imagen original. Así también, dentro de cada fila de elementos podemos ordenarlos a partir de los valores de las coordenadas X originales y así simular el orden que mantenían en la imagen sin recortar.

Teniendo en cuenta que para el proceso de identificación únicamente necesitamos los tres primeros elementos de cada una de estas filas (los tres primeros números de la primera fila son suficientes ya que las tres letras que aparecen en la segunda identifican la edición y esto nos genera toda la información necesaria para realizar el inventario) a la hora de retornar el resultado de la función **"Preparar Entradas"** únicamente se devuelven dichos primeros tres valores de cada vector, es decir las imágenes de los primeros tres elementos de cada fila.

El siguiente método que tiene protagonismo en nuestra implementación es **"OCRDeepLearning(result)"**. Como entrada recibe el resultado generado de la función de preparación de las imágenes y de él obtenemos el resultado de convertir el texto de estas imágenes a una cadena de texto compuesta por todos los caracteres identificados, cuya utilidad es justamente la de identificar inequívocamente a la carta en cuestión.

El procedimiento que se realiza dentro del método es el siguiente:

Las imágenes a identificar, que ya de por si vienen separadas en dos posiciones del vector de imágenes de entrada, se añaden, dependiendo del elemento que representen (si se trata un elemento de la primera o segunda fila), a dos vectores diferentes. Esto se realiza simplemente con la intención de segmentar la información a procesar.

A continuación se carga el modelo de la red neuronal y los pesos que se atribuyen a cada neurona. Únicamente necesitamos compilar el modelo y podremos utilizarlo completamente para la identificación.

Después de completar la compilación del modelo, realizamos dos predicciones de clase a través del método **"model.predict_classes(X)"**, las cuales nos devuelven los elementos identificadores de cada carácter reconocido por la red neuronal. Estas respuestas son guardadas y procesadas (en caso se ser identificadores de letras se vuelven a convertir de entero a carácter) para poderlas visualizar como una cadena conjunta. El resultado podemos observarlo en la siguiente **[Figura 28]:**

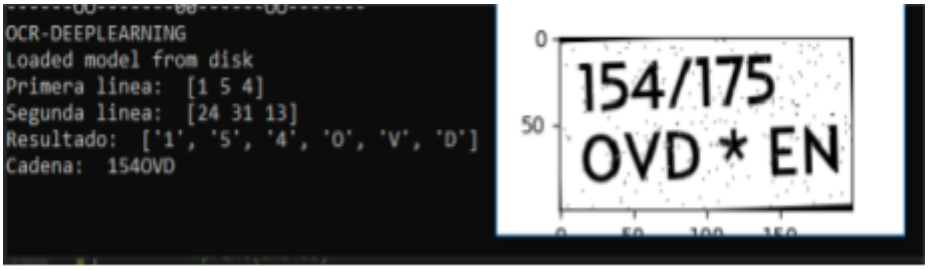

Figura 29: "Resultado OCR-CNN"

Conclusiones y resultados

En este apartado desarrollaremos cuáles han sido los resultados objetivos del trabajo y valoraremos la solución en base a los requisitos previstos.

Como bien se planteaba en los apartados de Antecedentes y Objetivos, la idea principal detrás del desarrollo de este proyecto era generar un sistema capaz de realizar el reconocimiento de cartas a través de herramientas y tecnologías de software libre.

Las soluciones implementadas, han presentado resultados satisfactorios, como se puede observar en las figuras **[Figura 23]**, **[Figura 24]** y **[Figura 28]**.

Si bien la versión ocupada de realizar el OCR a través de las librerías de visión por computador es capaz de obtener resultados prácticamente perfectos (confundiendo en un pequeño número de ocasiones algunos elementos con otros, principalmente debido a la fuente de texto utilizada y a la similitud entre ellos) tanto en imágenes de pruebas generadas sintéticamente como en imágenes de cartas reales, en el caso de la solución aplicando redes neuronales las pruebas únicamente han generado valores correctos con las imágenes sintéticas. Esto se debe a las proporciones de la imagen y al preprocesado necesario para adaptar la entrada a la red neuronal, el cual se ha prorrogado para este tipo de imágenes como una ampliación futura.

Aún así hemos demostrado la capacidad y posibilidad de aplicar esta tecnología para resolver la problemática, y las pruebas, aunque solo sean capaces de generar resultados adecuados con imágenes generadas, nos validan el correcto funcionamiento interno y la toma de decisiones en base a las imágenes de entrada.

Summary and Results

In this part of the report we will talk about the milestones of the developing process and we will value the solution with the previous requirements.

Just like it was proposed in the parts about the Background and Objectives, the main idea of this project was to generate a system capable of performing the cards recognition fundamented in the use of open source tools and technologies.

The solutions that had been implemented return us satisfactory results, as it can be seen in the **[Picture 23] , [Picture 24] and [Picture 28]**.

The version of this application based on the use of computer vision for the task of performing the OCR is able to generate practically perfect results (mixing in some cases some of the elements, mostly because the resemblance between them caused by the text font used) by using both synthetically generated images and images of real cards. Nonetheless, the one that's approached by the use of neural networks is only able to work correctly with the synthetic ones. This is mostly caused by the proportions of the image and by the preprocess needed to adapt the input to the neural network, which has been prorogued for future updates.

Even though, with this application we have been able to show and determine the capacity and possibility about approaching this kind of problems with this technology. And even with the fact that we are unable to validate the neural network with real cards images it just serves us as a way to validate the correct internal functioning and the take of decisions based on the input images.

Valoración personal y líneas futuras

Tanto el campo de la visión por computador como la aplicación y diseño de redes neuronales han sido tecnologías completamente nuevas para mi. Esto ha supuesto una fase de estudio y comprensión previa que me ha ayudado a entender la importancia que ostentan estos campos de la informática en el mundo actual.

No solo por el constante uso de herramientas que permiten obtener información de imágenes a través de librerías como OpenCV, por ejemplo el reconocimiento facial o la detección de contornos en imágenes para detectar objetos, o la utilidad que proporcionan los motores de OCR como Tesseract para poder recoger información de forma automática en procesos productivos cuya velocidad no permite el reconocimiento por parte de personas, ya que supondría un lastre. Sino también por encontrarme por primera vez con las aplicaciones y utilidades que nos brinda el uso de redes neuronales aplicadas.

Una tecnología que, en principio y para aquellos que no han estudiado su funcionamiento interno, puede parecer mágica. Que permite diferenciar si la fotografía que se le está introduciendo es de un gato o no, por ejemplo. Abre un mundo de posibles aplicaciones y soluciones a problemas que, cada día en mayor medida, están siendo el centro de atención de muchas empresas.

Si bien para el desarrollo de este proyecto he tenido que entender el funcionamiento básico que se esconde detrás de estas tecnologías, considero que aún me queda bastante por “excavar” y documentarme. Por ejemplo, una de las características propuestas como mejora posterior a la finalización del proyecto es la aplicación de otro tipo de red neuronal diferente (una red neuronal recursiva LSTM) para el proceso del OCR.

Esta aplicación de un modelo de red más complejo que las redes convolucionales tiene un beneficio, y es que permite la identificación de patrones en el tiempo, provocando una simulación de una máquina de estados en la que la salida del estado anterior afecta a la decisión de cuál va a ser la salida del estado actual. Es una tecnología bastante interesante y que merece ser estudiada a fondo, ya que nos permitiría identificar el texto en las imágenes por grupos de caracteres (palabras por ejemplo), sin la necesidad de separar en pequeñas imágenes con cada elemento individualmente.

Otro de los apartados que se intentará añadir al desarrollo como mejora futura es la adaptación del sistema para que pueda reconocer el identificador de las cartas reales a través del proceso que utiliza redes neuronales.

Dicho esto, y obviando los inconvenientes encontrados durante el desarrollo por culpa principalmente del choque conceptual que supuso adentrarme a trabajar con estas tecnologías sin una base previa, considero que el proceso de desarrollo de este proyecto ha sido especialmente fructífero para mi carrera profesional, ayudándome a conocer y a empezar a entender ciertas tecnologías que sin duda volveré a utilizar próximamente, posiblemente más a fondo.

Presupuesto

En este apartado se recoge el presupuesto estimado de la implementación de este proyecto.

Tipos	Cantidad	Valor por unidad	Total
Horas de trabajo:	170 Horas	20€	3.400€
Licencia Visual Studio:	1 u.	641€	641€

Tabla 2: Resumen de tipos

El resultado del presupuesto asciende a un total de **4.041€** como valor final de la implementación de nuestro proyecto.

Bibliografía

“OpenCV by Examples” de Prateek Joshi, David Millán Escrivá
Documentación oficial Keras
Hackernoon - OCR a través de Deep Learning
Ejemplos de Python - Programcreek
Stack Overflow
Towardsdatascience - Clase online sobre sistemas de reconocimiento automatizados
Youtube
Wikipedia
Documentación OpenCV
Documentación Tensorflow
Documentación Tesseract
Usando Pytesseract para aplicar OCR

Fuentes de figuras:

[Figura 1] : Carta MTG
[Figura 7]: OpenCV
[Figura 8]: Red Neuronal
[Figura 10]: Operación de convolución
[Figura 11]: Max Pool
[Figura 12]: One-Hot coding
[Figura 13]: Tensor
[Figura 26]: Función Sigmoide
[Figura 27]: Rectificación lineal