E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Creación de un sistema para la aplicación de redes neuronales convolucionales en un entorno de visión artificial

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Alumno: Adrián Errea López

Director: Mikel Galar Idoate

Pamplona, 7 de septiembre de 2020

30
1987-2017
upna

AGRADECIMIENTOS

A mis padres, por ayudarme a compatibilizar este trabajo final, así como el master y ser conscientes de lo que ello conlleva.

RESUMEN

El trabajo consiste en realizar una aplicación capaz de integrar el lenguaje de programación Python y todas sus funcionalidades en un entorno de C++ para la captura y el procesado de imágenes en tiempo real a partir de una cámara utilizada en entornos de producción de visión artificial. Concretamente, se realizará una aplicación que sea capaz de conectarse a una cámara industrial, configurarla y poder capturar fotos de manera que sea la base para aplicar Deep Learning sobre esas imágenes. Tras ello, y mediante la propia integración de Python en C++, se aplicarán redes neuronales convolucionales a la imagen obtenida por la aplicación, con el fin de obtener un resultado para cada imagen (clasificación).

PALABRAS CLAVE

CNN, Clasificación, Transfer Learning, Sistema Embebido, C++, VGG16, Python

Contenido

1. INTRODUCCIÓN 6
2. INTRODUCCIÓN AL MACHINE LEARNING Y DEEP LEARNING 8
2.1 MACHINE LEARNING 8
2.2 EVALUACIÓN DE MODELOS 12
2.2.1 MEDIDAS DE RENDIMIENTO 12
2.2.2 METODOLOGÍA DE EVALUACIÓN DE MODELOS 14
2.3 REDES NEURONALES 16
2.4 DEEP LEARNING 18
2.5 REDES NEURONALES CONVOLUCIONALES (CNN) 20
2.6 MODELOS DE CNN 24
2.7 TRANSFER LEARNING 30
3. DESCRIPCIÓN DE LA PROBLEMÁTICA 33
3.1 PROBLEMA A RESOLVER 33
3.2 COMPONENTES DEL SISTEMA EMBEBIDO 41
3.2.1 HARDWARE 42
3.2.2 SOFTWARE 45
3.3 SISTEMA EMBEBIDO 47
3.3.1 CONEXIÓN Y CONFIGURACIÓN DE LA CÁMARA 47
3.3.2 CREACIÓN DE UNA BASE DE IMÁGENES PARA ENTRENAMIENTO 48
3.3.3 EJECUCIÓN DEL MODELO CREADO SOBRE LA IMAGEN 49
4. APRENDIZAJE DE UN MODELO BASADO EN TRANSFER LEARNING PARA UNA APLICACIÓN INDUSTRIAL 51
4.1 DESCRIPCIÓN DEL PROBLEMA 51
4.2 METODOLOGÍA 51
4.2.1 EXTRACCIÓN DE LOS DATOS 52
4.2.2 PREPROCESAMIENTO DE LOS DATOS 52
4.3 EXPERIMENTOS Y PRUEBAS REALIZADAS 53
4.3.1 CNN BÁSICA 54
4.3.2 TRANSFER LEARNING (VGG16) 55
4.3.3 TRANSFER LEARNING (VGG16) CON FINE-TUNING 56
4.4 CONCLUSIONES Y DISCUSIÓN DE LOS RESULTADOS 58
5. DESPLIEGUE DEL SISTEMA 69

5.1 CONFIGURACIÓN NECESARIA ... 69
5.1.1 VISUAL STUDIO COMMUNITY ... 69
5.1.2 QTCREATOR ... 69
5.1.3 PYLON ... 69
5.1.4 PYTHON ... 70
5.1.5 CUDA Y CUDNN ... 70
5.1.6 VARIABLES DE ENTORNO ... 70
5.1.7 LIBRERÍAS PYTHON ... 70
5.2 FUNCIONAMIENTO DE LA APLICACIÓN ... 71
5.2.1 CONEXIÓN Y CONFIGURACIÓN DE LA CÁMARA ... 71
5.2.2 CREACIÓN DE UNA BASE DE IMÁGENES PARA ENTRENAMIENTO ... 76
5.2.3 EJECUCIÓN DEL MODELO CREADO SOBRE LA IMAGEN ... 77
6. CONCLUSIONES Y LÍNEAS FUTURAS ... 81
7. BIBLIOGRAFÍA ... 83

1. INTRODUCCIÓN

La visión artificial es uno de los campos de la Inteligencia Artificial que más impacto está teniendo en el mundo industrial a día de hoy. No es extraño encontrar sistemas de este tipo en grandes centros de producción ya sea con el fin de aumentar la calidad de los productos, verificar el correcto funcionamiento de elementos o realizar las tareas de robots corporativos. Además, estamos cada día un poco más cerca de poder observar vehículos autónomos tanto en la calle como en entornos industriales en donde es clave todos los sistemas de visión que se encargan de capturar y procesar la información en tiempo real.

La automatización de procesos es el gran objetivo de la mayoría de empresas de forma que los costes por proceso se reduzcan, la productividad aumente y los fallos causados por la mano humana desaparezcan. La visión artificial es clave en este aspecto ya que mejora la calidad de cualquier visión humana en un menor tiempo de procesado haciendo que los procesos sean cada vez mejores y, sobre todo, automatizados.

Por otra parte, dentro de la propia Inteligencia Artificial, el uso del Deep Learning y más concretamente de Redes Neuronales Convolucionales (CNN) está ofreciendo resultados nunca vistos hasta el momento y son un paradigma distinto a lo que se ha venido desarrollando en los últimos años. Ya se utilizan en medicina, redes sociales, automoción, satélites, banca, etc.

La gran mayoría de algoritmos y técnicas de inteligencia artificial utilizan Python como lenguaje de programación ya que dispone de gran documentación e implementación de multitud de algoritmos de Machine Learning y Deep Learning.

La motivación de este proyecto es poder combinar los resultados mostrados por las Redes Neuronales Convolucionales en entornos controlados (laboratorios, centros de datos, etc.) donde las condiciones son siempre parecidas, en un entorno industrial donde las condiciones pueden ser más cambiantes o, al menos, suponen un reto a la hora de replicar resultados vistos sobre el papel.

Adicionalmente, la aplicación de este proceso en un ambiente real también supone un reto ya que, a diferencia de un uso común donde se aplica este proceso sobre imágenes ya capturadas, en nuestro proyecto cada imagen se adquiere en tiempo real y se le aplica el modelo durante el proceso de producción.

Concretamente, el objetivo es detectar defectos en la producción de tortillas precocinadas, principalmente fallos en la impresión de etiquetas. Estos defectos pueden ser de diversos tipos y sobre diferentes tipos de etiquetas. Se procederá a su explicación detallada en la *Sección 3.1*.

La primera parte consiste en conectar la cámara a la aplicación, poder configurar sus parámetros para obtener una imagen de calidad y clasificar cada imagen del cliente manualmente para poder crear un modelo de Deep Learning a partir de la base de imágenes obtenidas.

Tras ello, tendremos que crear un modelo lo suficientemente fiable como para poder automatizar este proceso de manera que seamos capaces de discernir si una etiqueta está bien impresa o no y por tanto mejorar la calidad del proceso productivo del cliente.

Por último, la idea es utilizar e integrar ese modelo desarrollado en Python en la aplicación desarrollada en C++.

Con todo ello, y a partir del sistema específico descrito en el proyecto, el objetivo general es llevar a cabo un correcto estudio y configuración de cámaras industriales junto a la utilización de su API en una aplicación industrial, construir un modelo robusto de CNN capaz de clasificar imágenes a partir de las obtenidas por la propia aplicación e integrar el proceso de inferencia de Python dentro de la propia aplicación. Todo ello con el fin de crear un sistema de este tipo que sea capaz de adquirir imágenes mediante una de las cámaras que se utilizan en entornos de producción de visión artificial y poder aplicar un modelo de Deep Learning a esas imágenes en un entorno industrial cualquiera.

2. INTRODUCCIÓN AL MACHINE LEARNING Y DEEP LEARNING

En este apartado se realizará una introducción al Machine Learning (*Sección 2.1)* explicando qué tipos de problemas abarca, qué modelos de evaluación conocemos para aplicar sobre los modelos que creemos (*Sección 2.2),* qué son las redes neuronales y su funcionamiento *(Sección 2.3)* y qué es el Deep Learning y sus principales diferencias con el Machine Learning *(Sección 2.4)*. Tras ello, continuaremos explicando las Redes Neuronales Convolucionales (CNN) *(Sección 2.5)*, qué modelos de CNN existen en la actualidad y cuál es su arquitectura (*Sección 2.6)* y, por último, acabaremos presentando el Transfer Learning *(Sección 2.7).*

2.1 MACHINE LEARNING

El Machine Learning (o Aprendizaje Automático) es una rama de la Inteligencia Artificial cuyo objetivo es crear técnicas y algoritmos que hagan que las máquinas aprendan de manera automática. Por aprender entendemos que su desempeño o rendimiento en una tarea concreta mejora con la experiencia.

Dentro del Machine Learning podemos distinguir tres tipos de aprendizajes:

- Aprendizaje Supervisado: Para el problema en cuestión conocemos la salida deseada para los datos. Es decir, poseemos conocimiento previo del problema.
- Aprendizaje No Supervisado: En este caso, no poseemos conocimiento previo del problema por lo que no conocemos la salida de los datos.
- Aprendizaje Semi-Supervisado: Es un híbrido entre los dos anteriores. Conocemos la salida de algunos de los datos, pero no de la mayoría.

Dentro del Aprendizaje Supervisado distinguimos dos tipos de problemas principales: Regresión y Clasificación.

- Problemas de Regresión: En este tipo de problemas, la salida de cada clase es un valor numérico. Por ejemplo, la predicción de la temperatura o el crecimiento de la población para 2050. Dentro de este tipo de problemas podemos encontrar varias técnicas de regresión.

    - Regresión Lineal: Dadas dos variables X e Y, se trata de encontrar una función lineal que modele los datos de manera que minimice el error. En la *Figura 2.1* observamos una representación de puntos en el espacio (puntos azules) con la función lineal que los representa (línea roja). Para este caso se ha utilizado el MSE (Error Cuadrático Medio o Mean Squared Error, en inglés) que calcula el error cometido por la función. El MSE se calcula mediante la media de la suma de las diferencias al cuadrado entre el valor real y el valor estimado.

$$\mathrm{MSE} = \frac{1}{\mathrm{m}} \sum_{\mathrm{i}=1}^{\mathrm{m}} (\mathrm{real_i} - \mathrm{estimado_i})^2, \text{con m el número de ejemplos}$$

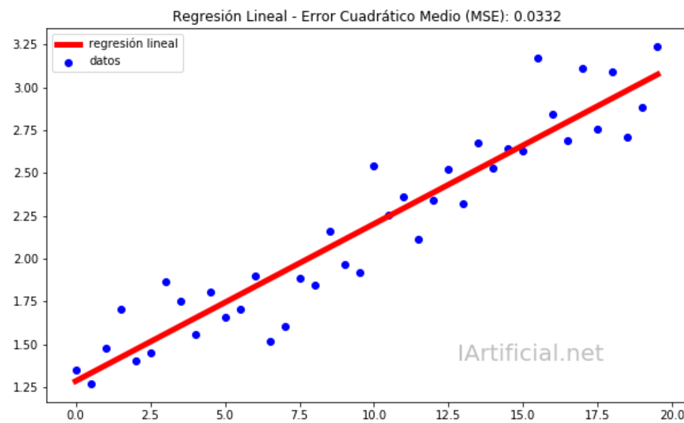

Figura 2.1: Ejemplo de Regresión Lineal. En IArtificial.net [1]

- Regresión No Lineal: En este caso, se utiliza una función no lineal debido a que no es posible modelar los datos con una función lineal. En la *Figura 2.2* se observa este comportamiento en donde los puntos azules representan los datos del problema y la línea roja la función de regresión no lineal.

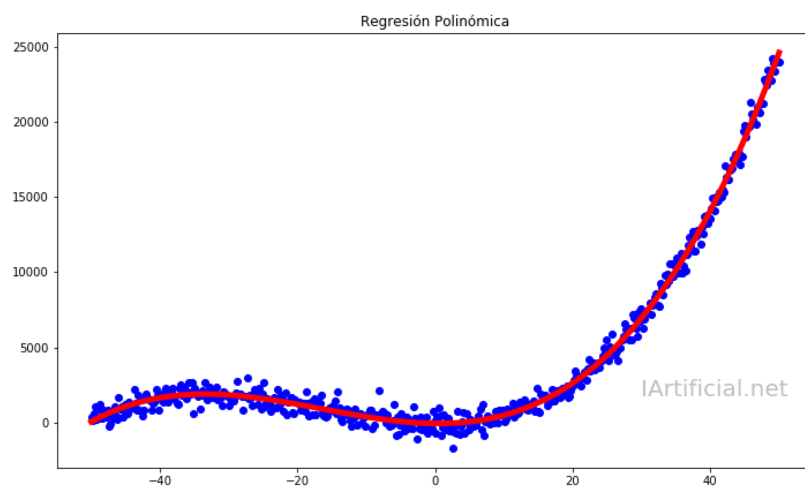

Figura 2.2: Ejemplo de Regresión No Lineal. En IArtificial.net [2]

- Problemas de Clasificación: En este caso, la salida de cada clase es una etiqueta. Por ejemplo, reconocimiento de dígitos o diagnóstico de imágenes. Para modelar este tipo de problemas se suelen utilizar diferentes tipos de algoritmos.

  - SVM: Las *Máquinas de Vector Soporte* (*Support Vector Machines,* en inglés) se basan en la construcción de un hiperplano óptimo de decisión de modo que el margen de separación entre clases sea máximo. En la *Figura 2.3* tenemos dos representaciones de un mismo problema. En la parte izquierda tenemos representados los datos de dos clases diferentes mediante cuadrados y círculos y representados en verde están algunos hiperplanos no óptimos construidos. En la parte derecha, observamos cómo sería el resultado tras aplicar el algoritmo SVM de forma que hemos encontrado el hiperplano óptimo ya que el margen entre ejemplos frontera (aquellos ejemplos más cercanos a la clase opuesta) es máximo.

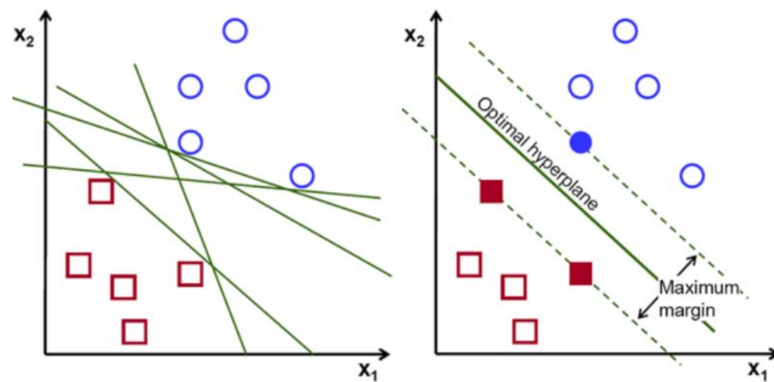

Figura 2.3: Ejemplo de SVM con el hiperplano óptimo. En medium.com [3]

  - KNN: El algoritmo de los *K vecinos más cercanos* (*K-nearest neighbors,* en inglés) trata de encontrar la frontera de decisión entre clases basándose en la agrupación de ejemplos similares a partir de las distancias entre sus vecinos. En la *Figura 2.4* podemos observar un ejemplo de su funcionamiento. En este caso queremos clasificar un ejemplo nuevo (representado con un interrogante). Para ello, calculamos su distancia con todos sus vecinos y nos quedamos con sus K vecinos más cercanos. Este cálculo de distancias se puede realizar mediante Distancia Euclídea, Distancia de Hamming, Distancia de Manhattan o Distancia de Minkowski. En el ejemplo K es igual a 3. Finalmente, para asignarle una clase se recurre a la votación que consiste en elegir una de las clases basándonos en la información de los K vecinos más cercanos. Los métodos más comunes para ello son votación por mayoría (mediante desempates entre clases) o votación con pesos en función de las distancias al ejemplo a clasificar.

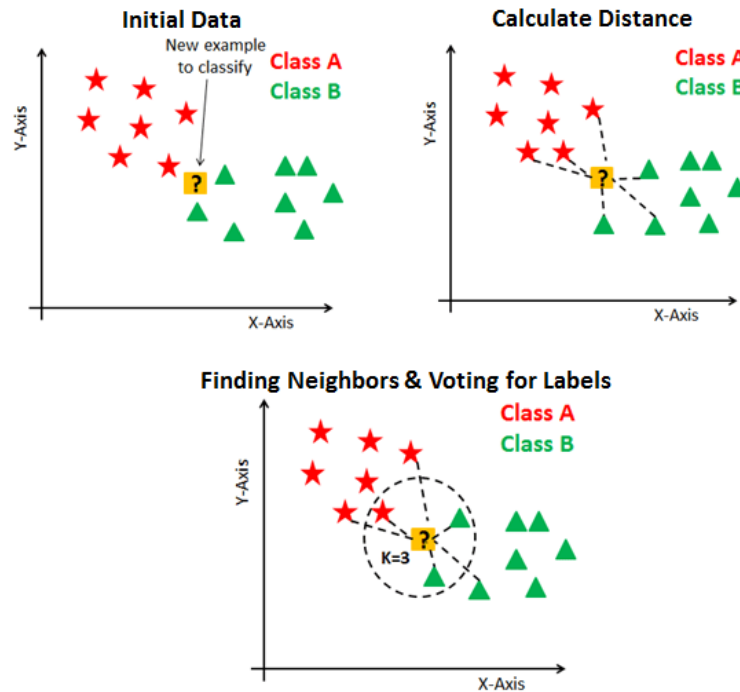

Figura 2.4: Figura con el proceso utilizado en KNN. En datacamp.com [4]

➢ <u>Árboles de Decisión:</u> Se construyen en base a las variables del problema para llegar a tener un conjunto de reglas que se aplicarán sobre el nuevo ejemplo a clasificar. En la *Figura 2.5* podemos observar un ejemplo de un árbol de decisión creado para clasificar pacientes con esquizofrenia. Para ello, empezamos del primer nodo (nodo raíz) que en este ejemplo corresponde a la imagen más arriba del árbol. En este caso, si la conectividad funcional de red (*FNC*) (representado en el nodo raíz) es menor de 0.34, se explora el nodo hijo derecho que representa la visión temporal [5]. Si este valor es menor que 0.25 entonces se etiqueta al paciente como Enfermo (*Patient*). De esta manera y para cada ejemplo a clasificar, iríamos recorriendo el árbol hasta llegar a una etiqueta (clase) que sería asignada al ejemplo.

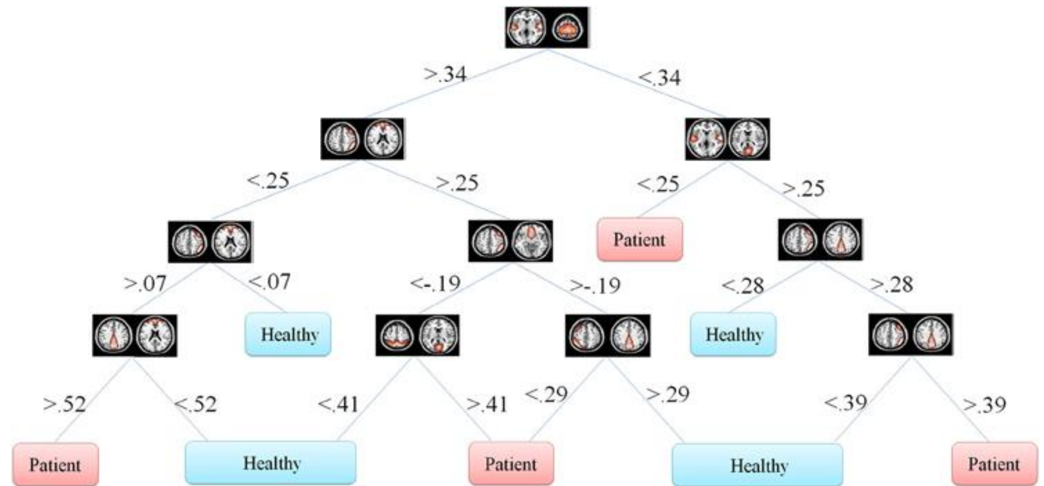

Figura 2.5: Ejemplo de un árbol de decisión para clasificación de pacientes con esquizofrenia. En cs.us.es [5]

- <u>Redes Neuronales:</u> Simula el comportamiento de las redes neuronales biológicas de forma que mediante neuronas conectadas entre sí y funciones de activación produce un valor de salida. Se explican con detalle en la *Sección 2.3.*

2.2 EVALUACIÓN DE MODELOS

Para llevar a cabo la evaluación de modelos es necesario definir unas medidas que indiquen la calidad de un modelo creado. Estas medidas de rendimiento se detallan en la *Sección 2.2.1.* De la misma forma, debemos encontrar una metodología de evaluación de modelos para validar cualquier modelo predictivo que creemos (*Sección 2.2.2).*

2.2.1 MEDIDAS DE RENDIMIENTO

Dado un problema de clasificación con *n* clases, una matriz de confusión es una matriz *n x n* en la que una entrada c_{ij} indica el número de ejemplos que se han asignado a la clase c_j, cuando la clase correcta es c_i. De esta forma, podemos visualizar el desempeño de un algoritmo de aprendizaje supervisado (sabemos la clase real de cada ejemplo). Una matriz de confusión estándar sería la que se encuentra en la *Tabla 2.1.* En ella se contabilizan cuatro posibles valores.

- VP: Verdadero Positivo. La clase real es positiva y la clase predicha también. Se cuenta como un acierto.
- FP: Falso Positivo. La clase real es negativa y la clase predicha es positiva. Es un fallo en la predicción ya que se predice un caso positivo que realmente no existe. Es el menos grave de los errores.
- VN: Verdadero Negativo. La clase real es negativa y la clase predicha también. Se cuenta como un acierto.
- FN: Falso Negativo. La clase real es positiva y la clase predicha es negativa. Es un fallo en la predicción ya que no se predice un caso positivo que realmente existe. Es el más grave de los errores.

		CLASIFICADO COMO	
		SI	NO
CLASE REAL	SI	VERDADERO POSITIVO (VP)	FALSO NEGATIVO (FN)
	NO	FALSO POSITIVO (FP)	VERDADERO NEGATIVO (VN)

Tabla 2.1: Matriz de Confusión

Basándonos en lo anterior podemos definir varias medidas de rendimiento:

- $Accuracy = \frac{VP+VN}{VP+VN+FP+FN}$ → Porcentaje de aciertos sobre todos los ejemplos clasificados. No es de utilidad con clases desbalanceadas.
- $Recall = TPR = \frac{VP}{VP+FN}$ → Ejemplos de la clase positiva (minoritaria, de interés) clasificados correctamente. De interés si queremos minimizar los falsos negativos.
- $Precision = \frac{VP}{VP+FP}$ → Ejemplos clasificados en la clase positiva que son realmente de la clase positiva. De interés si queremos minimizar los falsos positivos.
- $F1 - Score = 2\ x\ \frac{Precision\ x\ Recall}{Precision+Recall}$ → Es una medida que considera el balance entre Precision y Recall.
- $Especificidad = TNR = \frac{VN}{VN+FP}$ → Ejemplos de la clase negativa clasificados correctamente.
- $Accuracy\ Balanceado = \frac{TPR+TNR}{2}$
- $Media\ geométrica = \sqrt{TPR\ x\ TNR}$
- *Área bajo la curva (AUC):* La curva ROC muestra en el eje Y el TPR y en el eje X el FPR (1-TNR). Caracteriza el balance entre aciertos de la clase positiva y falsos positivos. Para la construcción de la curva ROC procederíamos de la siguiente manera:

  1. Clasificar todos los ejemplos y almacenar para cada uno de ellos la probabilidad de ser clasificado como clase positiva (la de interés).

2. Ordenar las probabilidades de mayor a menor.
3. Para cada valor de probabilidad (umbral), clasificamos los ejemplos con probabilidades mayores o iguales que ese umbral como positivos, y el resto como negativos. Calculamos el TPR y el FPR para ese umbral.
4. Una vez obtenido todos los TPR y FPR para todos los umbrales, se representa en una gráfica el TPR en el eje Y, y el FPR en el eje X para cada umbral.
5. Se unen los puntos representados en la gráfica y obtenemos la curva ROC. El área bajo la curva (AUC) corresponde al área que queda por debajo de la propia curva ROC.

En la *Figura 2.6* están representados cuatro clasificadores diferentes con sus correspondientes curvas ROC y el área que quedaría por debajo de cada uno de ellos. Con un AUC=1 conseguiríamos el clasificador perfecto y con un AUC=0,5 tendríamos un clasificador aleatorio representado en la gráfica por la línea discontinua.

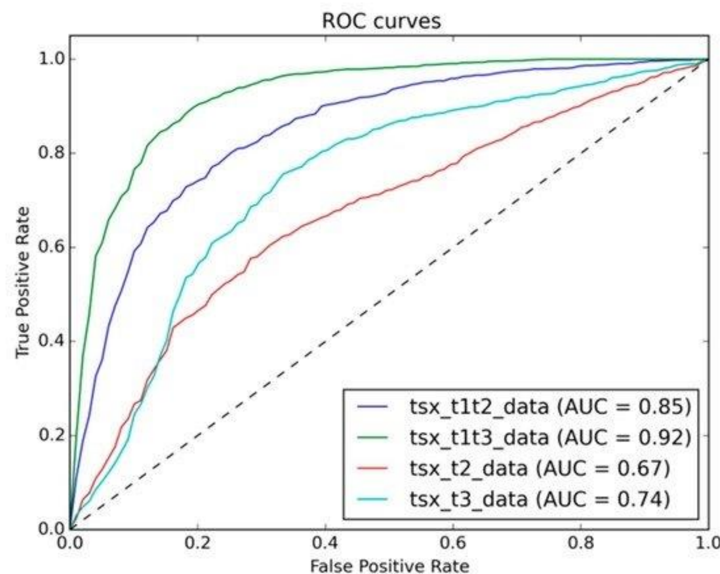

Figura 2.6: Diferentes curvas ROC. En mdpi.com [6]

2.2.2 METODOLOGÍA DE EVALUACIÓN DE MODELOS

Con el fin de validar los modelos de predicción creados se procede a crear diferentes conjuntos de ejemplos partiendo de los datos originales.

- Conjunto de entrenamiento: Se utiliza para realizar el aprendizaje del modelo. Cuanto más grande y diverso sea este conjunto, mejor será el modelo generado.
- Conjunto de validación: Se utiliza para evaluar el rendimiento del modelo y ajustar su configuración en base a los resultados obtenidos. De alguna manera afecta al modelo ya que lo ajustamos en base al rendimiento obtenido. Debido a ello, es necesario el conjunto siguiente.

- Conjunto de test: Se utiliza para evaluar el rendimiento del modelo ante nuevas situaciones. Cuanto más grande y diverso sea este conjunto, mejor será la estimación de la calidad del modelo. Se considera un conjunto aparte de los anteriores y solo se utiliza cuando el modelo ya está ajustado.

Si no disponemos de gran cantidad de datos para crear ambos conjuntos podemos recurrir a diferentes técnicas para obtener los conjuntos de train y validación:

- Hold-out: Consiste en dividir toda la base de datos inicial en dos conjuntos independientes, train y validación, de manera que el tamaño de train sea normalmente mayor que el de validación. Los ejemplos de train suelen obtenerse mediante muestreo sin reemplazamiento y aquellos que no caigan en train, irán a validación. Dado que esto puede llevar a que los ejemplos elegidos no sean representativos, se suele usar estratificación para asegurar que la distribución de ejemplos en ambos conjuntos es similar a la base de datos inicial.
- Validación cruzada: Se dividen todos los ejemplos iniciales en *k* subconjuntos. Éstos deben tener el mismo número de ejemplos de manera que los ejemplos se asignan una sola vez a cada subconjunto de manera aleatoria.
  El proceso consiste en elegir como conjunto de entrenamiento *k*-1 subconjuntos y el restante como conjunto de validación. Se realiza el aprendizaje con el conjunto de train y se obtiene el rendimiento con ambos conjuntos. Se realiza el proceso *k* veces de manera que se van rotando los subconjuntos de train y validación y el rendimiento obtenido es la media aritmética de las *k* iteraciones.
- Leave-one-out: Es un caso especial de validación cruzada donde *k* es igual al número de ejemplos. Es un proceso sin aleatoriedad y no es posible aplicar estratificación. Debido a su alto coste computacional solo es usa en base de datos muy pequeñas.
- Boostraping: Basado en muestreo con reemplazamiento a partir de una base de datos con *n* ejemplos se obtiene un conjunto de entrenamiento con *m* ejemplos. Para el conjunto de validación se escogen aquellos ejemplos que no se hayan elegido en train. La probabilidad de que un ejemplo no se haya elegido para validación es
  $\left(1-\frac{1}{N}\right)^N \approx e^{-1} = 0.368$, siendo N el número de extracciones.
  Por tanto, el conjunto de entrenamiento tendrá aproximadamente el 63.2 % de los ejemplos y el conjunto de validación el 36.8 %. Debido a que el modelo entrena con poca cantidad de ejemplos, el error en el conjunto de test suele ser bastante pesimista. Por ello, se combina con el error del conjunto de entrenamiento de la forma:

$$error_{validación} = 0.632 * error_{validación} + 0.368 * error_{train}$$

2.3 REDES NEURONALES

Las redes neuronales son un modelo computacional que simula el comportamiento de las redes neuronales biológicas. El elemento más básico de una red neuronal es la neurona.

Como se puede observar en la *Figura 2.7*, una neurona está compuesta de Dendritas que se encargan de captar los impulsos nerviosos de otras neuronas, el Soma que se encarga de procesar esos impulsos y el Axón que emite el nuevo impulso hacia otras neuronas.

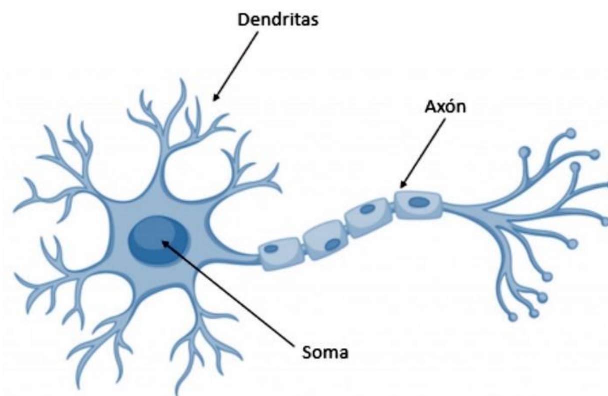

Figura 2.7: Esquema de una neurona biológica. En xeridia.com [7]

Su símil de manera artificial se puede representar como vemos en la *Figura 2.8.*

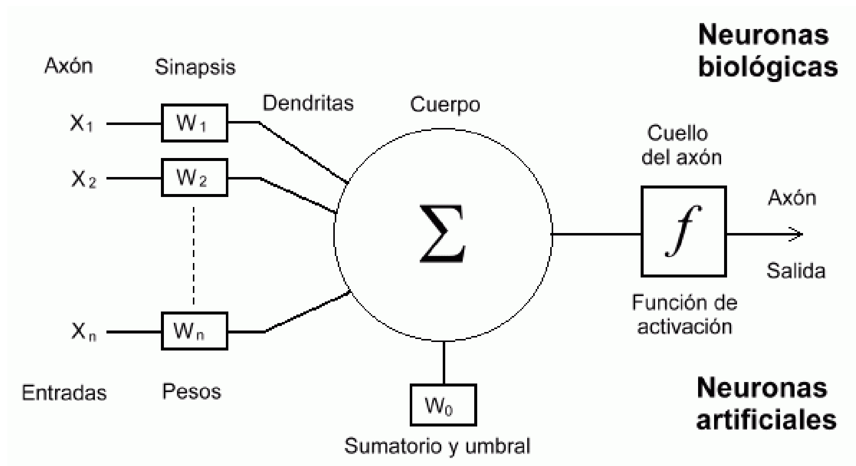

Figura 2.8: Esquema comparativo entre neurona biológica y neurona artificial. En cs.us.com [8]

En este caso, vemos como la suma de las entradas multiplicadas por sus pesos asociados determina el impulso nervioso que recibe la neurona (Dendritas). Este valor se procesa mediante una función de activación (Soma) que devolverá un valor de salida a la siguiente neurona (Axón).

Una red neuronal está compuesta de multitud de neuronas interconectadas entre sí y agrupadas en diferentes niveles que dan lugar a lo que se conoce como capas de neuronas. Las neuronas

de la primera capa (que reciben los datos de entrada) se conocen como capa de entrada. La capa que devuelve el resultado visible es la última capa y se le llama capa de salida. El resto de capas que se encuentran entre la capa de entrada y la capa de salida se llaman capas ocultas (que pueden ir desde 0 hasta *n* capas ocultas). Una representación sencilla con una capa de entrada, una capa oculta y una capa de salida se puede observar en la *Figura 2.9.*

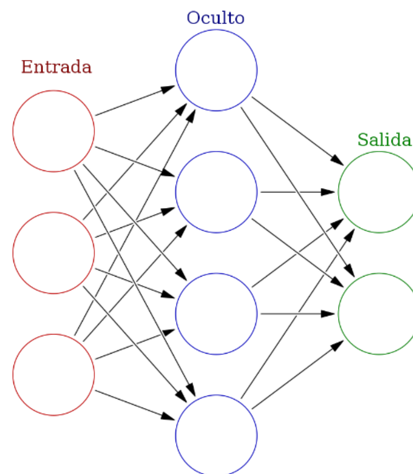

Figura 2.9: Red Neuronal con una capa de entrada, una capa oculta y una capa de salida. En wikipedia.com [9]

Cuando tenemos una red neuronal con más de una capa oculta entonces hablamos de un *Perceptrón Multicapa (MLP)*. Se caracteriza por ser capaz de resolver problemas que no son linealmente separables. Su aprendizaje normalmente se lleva a cabo mediante la *propagación hacia atrás* (*Backpropagation,* en inglés). Para ello, una vez fijada una arquitectura de una red neuronal concreta, lo único que se puede modificar para obtener diferentes resultados son los pesos entre neuronas de distintas capas. El objetivo consiste en encontrar los pesos que minimicen el error en la predicción final. Para ello se realiza el siguiente proceso:

1) Se inicializan aleatoriamente los pesos entre neuronas.
2) Se realiza la propagación hacia delante. Es decir, se multiplican las entradas por los pesos en cada capa y el resultado se pasa a la siguiente capa hasta obtener el resultado de la predicción.
3) Se calcula el error entre la predicción y el resultado esperado (el real).
4) Se aplica la propagación hacia atrás. Para ello se aplica el descenso por gradiente de la función de error respecto a los pesos actuales de la red. De esta forma se van actualizando los pesos entre neuronas de atrás hacia adelante.
5) Se vuelve a repetir el proceso completo hasta que el error se minimice lo suficiente o sea cero.

2.4 DEEP LEARNING

El Deep Learning (o Aprendizaje Profundo) es un área concreta del Machine Learning que consta de un conjunto de algoritmos basados en redes neuronales artificiales. Si lo englobamos dentro del marco de la Inteligencia Artificial y el Machine Learning podríamos representarlo de manera visual como en la *Figura 2.10.* En ella vemos como la Inteligencia Artificial engloba a programas y algoritmos con la habilidad de "aprender" como lo hacen los seres humanos. Una subcategoría sería el Machine Learning en donde los algoritmos se caracterizan por "aprender" sin ser explícitamente programados para ello; aprenden con la experiencia. Dentro del Machine Learning, incluiríamos al Deep Learning donde las redes neuronales artificiales se adaptan y aprenden de grandes cantidades de datos.

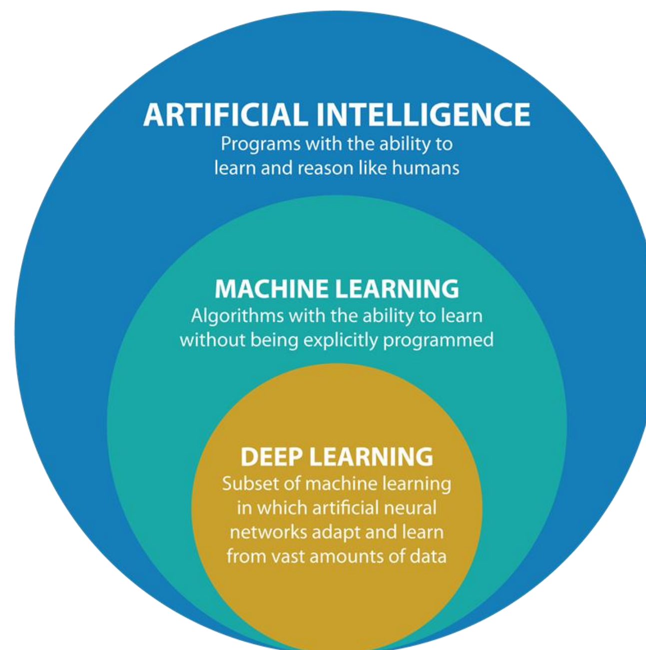

Figura 2.10: Comparación entre Inteligencia Artificial, Aprendizaje Automático y Aprendizaje Profundo. En datacatchup.com [10]

La idea detrás del Deep Learning es intentar que los propios algoritmos aprendan sin intervención humana previa, obteniendo ellos mismos las características que consideran relevantes para el problema en concreto. Esta es una diferencia significativa que queda reflejada en la *Figura 2.11.* En Machine Learning es necesario la intervención de alguien que tenga un conocimiento en el ámbito del problema a tratar y que, por tanto, identifique las características más importantes del problema que posteriormente utilizará el algoritmo para realizar su aprendizaje. Sin embargo, en Deep Learning el objetivo es que sea el propio algoritmo el que extraiga las propias características del problema que considere relevantes, para llegar a obtener un buen modelo sin necesidad de un agente externo experto en el problema.

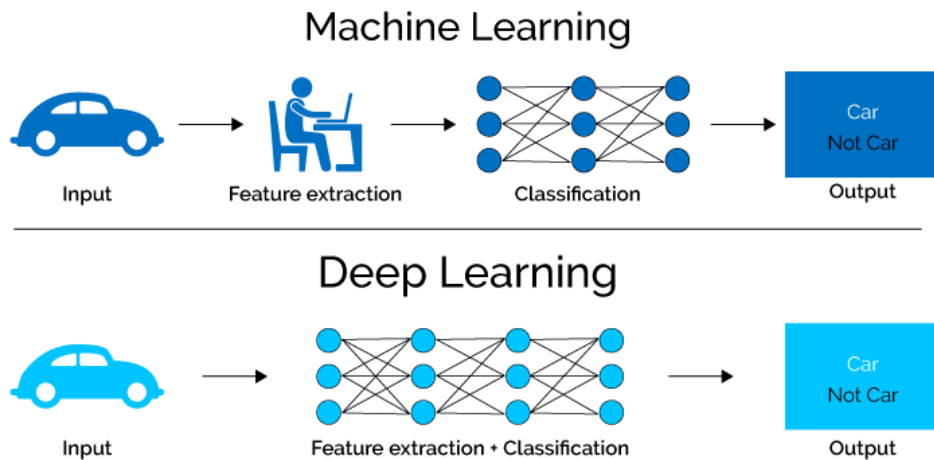

Figura 2.11: Diferencia principal entre Machine Learning y Deep Learning. En towardsdatascience.com [11]

Además de ello, el Deep Learning proporciona múltiples ventajas frente al Machine Learning tradicional:

- En Machine Learning, a más cantidad de datos no significa mejores resultados. En Deep Learning, el rendimiento suele ser creciente en la mayoría de casos.
- Cuando no hay un dominio claro del problema, el Deep Learning sobresale ya que permite tener un conocimiento menor sobre las propias características del problema.
- Ofrece mejores resultados en comparación con otros algoritmos de Machine Learning en ámbitos como Clasificación de Imagen o Procesamiento de Lenguaje Natural.

Sin embargo, también tiene algunas desventajas que pueden limitar su utilización:

- Dependiendo de la complejidad del problema, es necesario obtener gran cantidad de datos para un buen rendimiento.
- El entrenamiento lleva mucho más tiempo en comparación con algoritmos de Machine Learning.

Las redes neuronales profundas se caracterizan por tener multitud de capas ocultas y de neuronas (o unidades) en cada una de ellas. El funcionamiento general consiste en, dados unos valores de entrada a la red neuronal, la primera capa (capa de entrada) extraería la primera información y se la enviaría a la siguiente capa (capa oculta). En la siguiente capa, las neuronas identificarían ciertas características y se lo pasarían a la siguiente capa oculta. De esta forma, la última capa combinaría las características identificadas y obtendría un resultado para dicha entrada. Además, otra de sus características propias es que las funciones de activación utilizadas normalmente son no lineales lo que permite resolver problemas linealmente no separables. Ejemplos de funciones más utilizadas serían Softmax, Sigmoide, ReLU o tanH.

En la *Figura 2.12* podemos observar una red neuronal profunda con una capa de entrada, tres capas ocultas (con seis unidades en la primera capa oculta, seis unidades en la segunda capa oculta y ocho unidades en la tercera capa oculta) y una capa de salida con tantas unidades como valores de salida necesitemos.

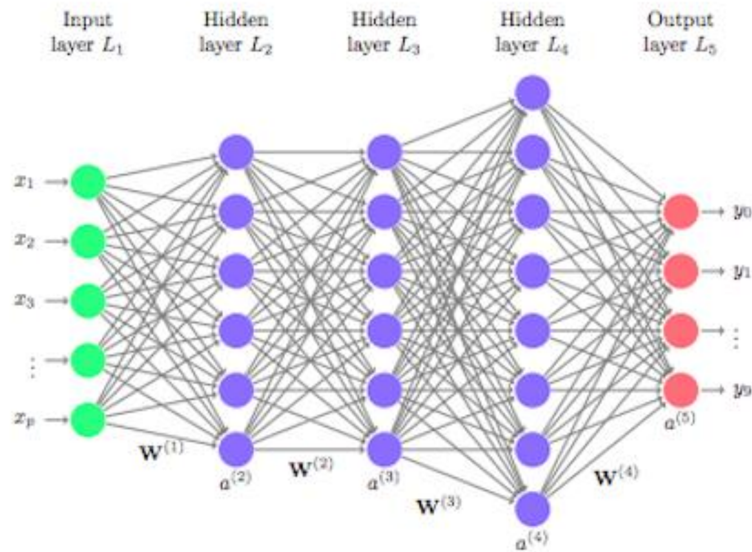

Figura 2.12: Representación de una red neuronal profunda. En towardsdatascience.com [12]

2.5 REDES NEURONALES CONVOLUCIONALES (CNN)

Comenzando con la evidencia relativa a las Redes Neuronales Convolucionales (Convolutional Neural Networks (CNN), en inglés), se ha visto como ofrecen buenos resultados en ámbitos médicos [13], reconocimiento facial [14] o incluso en conducción autónoma [15]. Por otra parte, la visión artificial (que está estrechamente relacionada con el uso de las CNN) también ha conseguido buenos resultados en aplicaciones como clasificación de objetos [16] y [17], utilización de robots [18] o sustituyendo a la vista en algunos casos [19]. Además, combinando ambas técnicas hay ciertos estudios que demuestran buenos resultados tanto en detección de objetos [20] y [21] como en segmentación de imágenes [22].

Sin embargo, trabajar con imágenes en redes neuronales no es un procedimiento trivial. Cuando trabajamos con una imagen, los parámetros de entrada de la red neuronal corresponden a los píxeles de cada una de las dimensiones de la misma. Esto hace que, por ejemplo, en una imagen relativamente pequeña de 1000x1000x3 el número de parámetros de entrada para la primera capa sea de 3 millones. Estos valores multiplicados por todos los valores de los pesos en cada una de las capas hacen que su computación sea inviable. Es aquí donde surgen las redes neuronales convolucionales.

Su nombre viene dado de la operación clave en su funcionamiento: la operación de convolución. Una convolución no es más que una multiplicación elemento a elemento de dos matrices. Cuando aplicamos una operación de convolución lo que hacemos es aplicar los valores de la segunda matriz (o filtro, también llamado kernel) a la primera matriz (en nuestro caso la imagen). Cuando los valores del filtro corresponden a mascaras para bordes verticales, horizontales, etc. es cuando obtenemos un mapa de características para ese filtro. De esta forma

obtenemos ciertas características de interés a partir de la imagen. Podemos observar un ejemplo en la Figura 2.13.

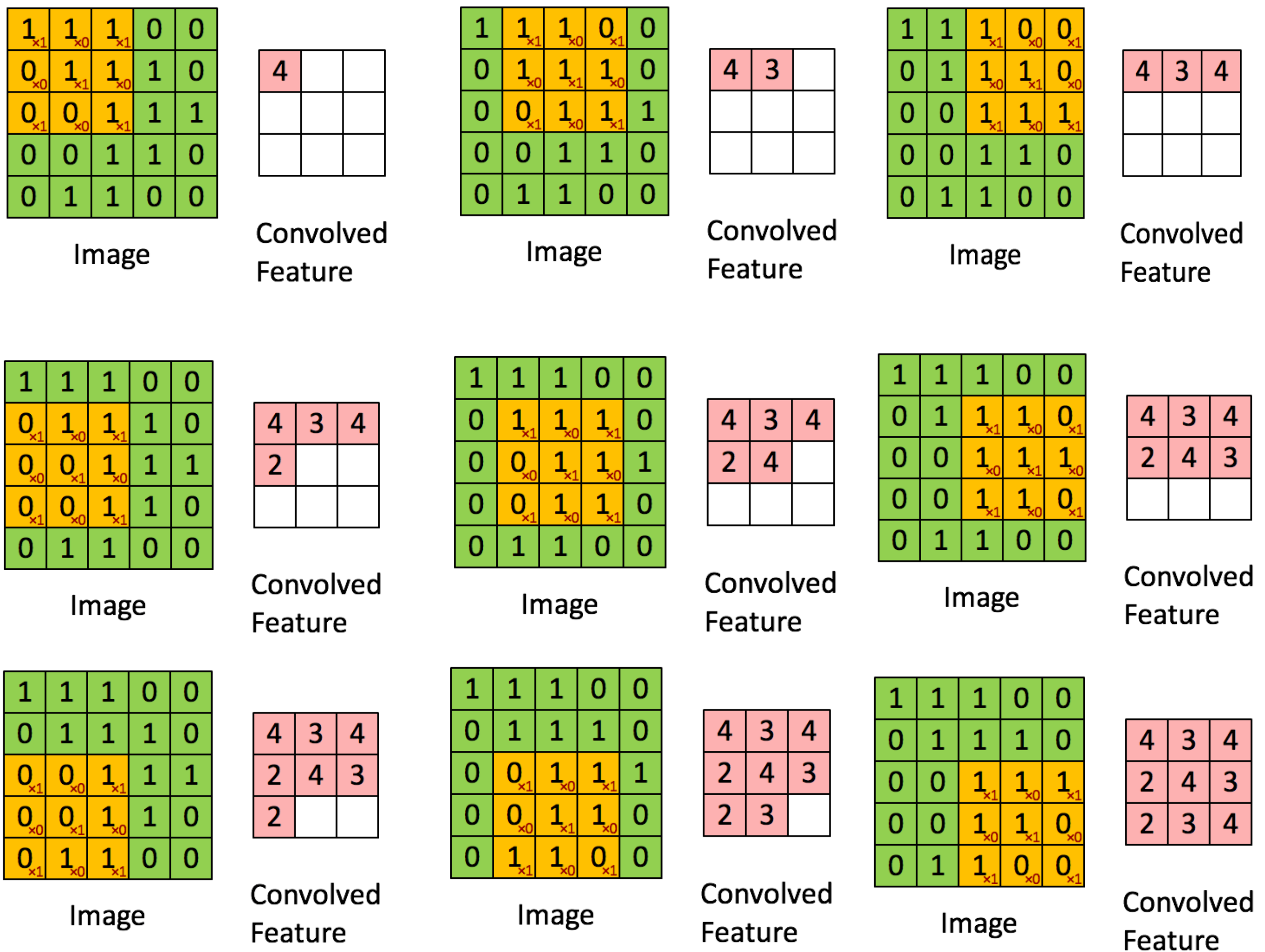

Figura 2.13: Ejemplo de una convolución de un filtro 3x3 sobre una imagen 5x5 dando como resultado una matriz 3x3. En medium.com [23]

En este caso, vemos como el filtro (matriz en amarillo) se va aplicando sobre la imagen de entrada dando como resultado el mapa de características para ese filtro. Este proceso se va haciendo por toda la matriz hasta completar la matriz de destino.

Hay que destacar que esta operación produce dos resultados adversos: cada vez que se aplica un filtro, el tamaño de la imagen se reduce. Por otro lado, los pixeles de las esquinas se utilizan menos que los píxeles centrales. Para resolver estos dos problemas se lleva a cabo la técnica de Padding. Consiste en añadir ciertas filas superiores e inferiores y ciertas columnas a la derecha y a la izquierda de la matriz origen (si el Padding es 1 se añade 1 fila superior e inferior y 1 columna a la derecha y a la izquierda). De esta manera, la imagen mantiene su tamaño original

y los pixeles de las esquinas de la imagen original aparecen un número similar de veces al aplicar la operación de convolución.

Por otro lado, otra técnica bastante utilizada es Striding. Consiste en aplicar la propia convolución, pero con cierto salto entre filas y columnas. Por ejemplo, si consideramos Stride 2, cada vez que aplicamos la convolución movemos el filtro dos columnas y dos filas en vez de una, que sería la forma por defecto.

Visto el epicentro de las CNN, podemos distinguir tres tipos de capas:

- Capas Convolucionales (CONV)
- Capas Pooling (POOL)
- Capas Totalmente Conectadas (FC)

Las Capas Convolucionales son aquellas donde se aplica la convolución previamente explicada y cuyo objetivo es obtener un conjunto de características que puedan servir para el aprendizaje.

Las Capas Pooling reducen la imagen de entrada con el fin de obtener los parámetros más representativos y de reducir el número de estos, así como la cantidad de computación necesaria. El proceso consiste en aplicar una función a una zona de la imagen y quedarnos con un valor que la represente. En la práctica, el método más utilizado es Max Pooling donde por cada zona que se aplica, nos quedamos con el máximo valor de los pixeles. Otras implementaciones como Average Pooling o Sum Pooling son menos utilizadas.

Por último, las Capas Totalmente Conectadas son las que conocemos de las redes neuronales profundas tradicionales donde todas las neuronas de una capa están conectadas con todas las neuronas de la capa siguiente. Se suelen utilizar al final de la CNN donde las características que llegan a ellas ya son en forma de elementos definidos y por tanto cobran más importancia a la hora de predecir la salida.

Un esquema típico de una CNN sería el observado en la *Figura 2.14.*

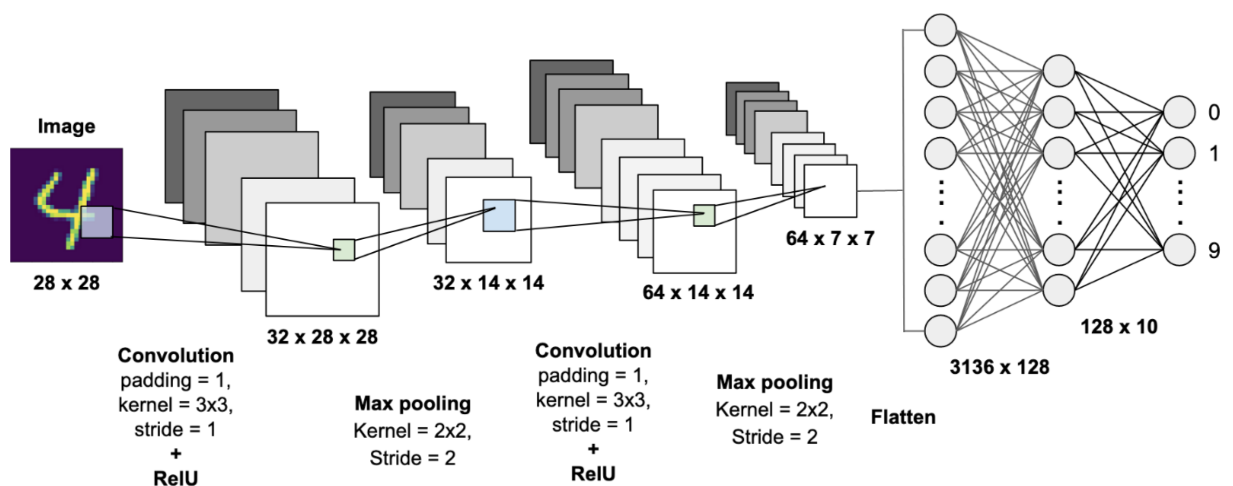

Figura 2.14: Ejemplo de una CNN. En mc.ai [24]

En este caso, observamos como la entrada es una imagen de 28x28 que representa un número del 0 al 9. La primera capa que se aplica es una Convolucional, con padding igual a 1 (no reducimos el tamaño), el filtro a aplicar tiene tamaño 3x3 y el stride es 1. Tras ello, se aplica una función de activación no lineal. En este caso la función ReLU. Cabe destacar que la dimensión que se obtiene en la capa de salida viene dada por la siguiente formula:

$d_s = \frac{n+2p-f}{s} + 1$, donde *n* es la dimensión de entrada, *p* es el padding, *f* es la dimensión del kernel (o filtro) y *s* es el stride aplicado.

En el ejemplo anterior, $d_s = \frac{28+2x1-3}{1} + 1 =$ 28. Por tanto, la dimensión de salida de la Capa Convolucional será *nº filtros (o nº canales) x dimension1 x dimension2*. En este caso, 32x28x28 ya que suponemos que se han aplicado 32 filtros 3x3 en esta Capa de Convolución.

El siguiente paso que se aplica es el correspondiente a una Capa Pooling. En concreto, Max pooling. Se aplica un stride de 2 y los filtros tienen tamaño 2x2 resultando en una capa de salida de tamaño 32x14x14. A continuación, vemos como se aplica el mismo proceso que antes. Tras ello, observamos dos Capas Totalmente Conectadas al final de la CNN. La primera tiene como entrada todos los valores de la última Capa Pooling (64x7x7) y los conecta con todas sus neuronas (128) dando lugar a 401408 parámetros en la primera Capa Totalmente Conectada. Esos parámetros vuelven a ser conectados con otra capa del mismo tipo, pero esta vez con 10 neuronas que darán lugar a una probabilidad de pertenecer a cada clase (número del 0 al 9).

Visto el funcionamiento general de las CNN podemos resumir sus ventajas de la siguiente forma:

- Reducción considerable de parámetros entre capas. En la *Figura 2.15* podemos observar como de una capa de 3x32x32 a una capa 6x28x28 aplicando una convolución con 6 filtros de tamaño 5x5 en una CNN deberíamos aprender 156 parámetros. Por otro lado, en una red neuronal profunda tradicional entre la capa de 3x32x32 (3072) y la capa 6x28x28 (4704) deberíamos aprender 14 millones de parámetros.

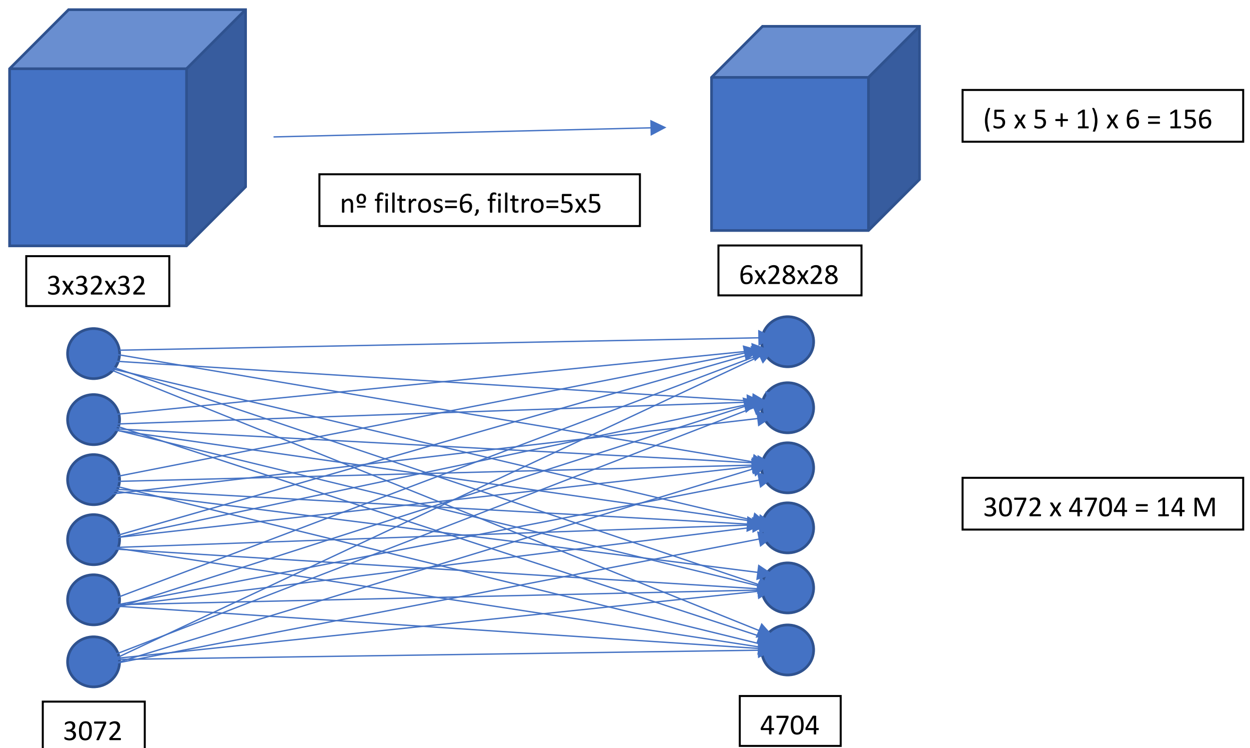

Figura 2.15: Representación de la diferencia de parámetros entre CNN y red neuronal profunda tradicional

- Reutilización de parámetros. Esta reducción tan considerable es debido a que ciertos parámetros utilizados en una parte de la imagen (bordes verticales, por ejemplo) pueden ser utilizados en otras partes de la imagen. Esta reutilización de parámetros hace que se tengan que aprenden muchos menos que de forma convencional.
- Menor coste computacional. Con todo ello, la reducción tanto de memoria y tiempo comparado con otras redes neuronales de arquitecturas similares junto con los rendimientos que ofrecen, las sitúan por delante de otro tipo de redes neuronales en el tratamiento con imagen .

2.6 MODELOS DE CNN

Dentro de las CNN existen diferentes arquitecturas o modelos que están basados en casos de estudio y que poseen características interesantes. En este apartado vamos a comentar los más conocidos y sus características particulares.

- **LeNet-5:** Una de las primeras redes neuronales convolucionales que se crearon. Sus creadores la lanzaron en 1990. Su arquitectura es bastante sencilla tal y como observamos en la *Figura 2.16.* Consiste en, dada una imagen en escala de grises, reconocer números escritos a mano.
    - La primera capa es una Convolucional con 6 filtros de tamaño 5x5 con stride 1.

- La segunda capa es una Average Pooling con un filtro 2x2 y stride de 2.
- La siguiente capa es una Convolucional con 16 filtros de tamaño 5x5 y stride 1.
- La cuarta capa es otra vez una Average Pooling con un filtro de 2x2 y stride de 2.
- La quinta capa es una Totalmente Conectada con 120 neuronas que conecta la capa anterior con todos los nodos de esta capa.
- La sexta capa es una Totalmente Conectada con 84 neuronas.
- La última capa es una Totalmente Conectada con una función softmax que predice un número del 0 al 9.

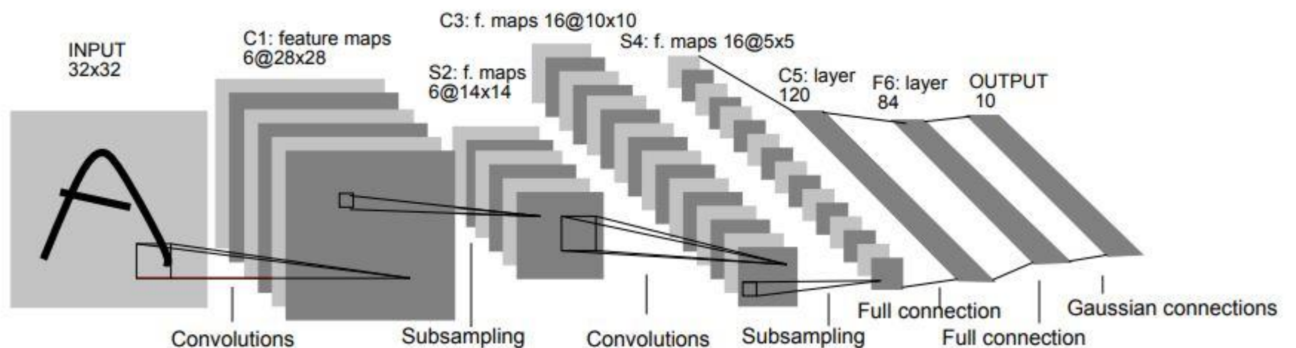

Figura 2.16: Arquitectura original LeNet-5. [25]

- **AlexNet:** Creada en 2012 fue entrenada con 1,2 millones de imágenes de gran resolución con 1000 clases, 60 millones de parámetros y 650000 neuronas. Su arquitectura se observa en la *Figura 2.17.* Dada una imagen en RGB,
  - La primera capa es una Convolucional con 96 filtros de tamaño 11x11 con stride 4.
  - La siguiente capa es una Max Pooling de tamaño 3x3 con stride 2.
  - La tercera capa consiste en una Convolucional con 256 filtros de tamaño 5x5 con stride 1.
  - La siguiente capa es igual que la Max Pooling anterior (3x3) y stride 2.
  - Las siguientes capas son tres capas Convolucionales. Tienen un tamaño de filtro 3x3 y stride 1. Las dos primeras usan 384 filtros y la tercera 256.
  - Le sigue una Max Pooling de tamaño 3x3, stride 2 y 256 filtros.
  - La novena capa es una Totalmente Conectada con 9216 neuronas.
  - La décima y undécima también son Totalmente Conectadas con 4096 neuronas cada una.
  - La última capa es la capa de salida softmax con 1000 posibles valores.

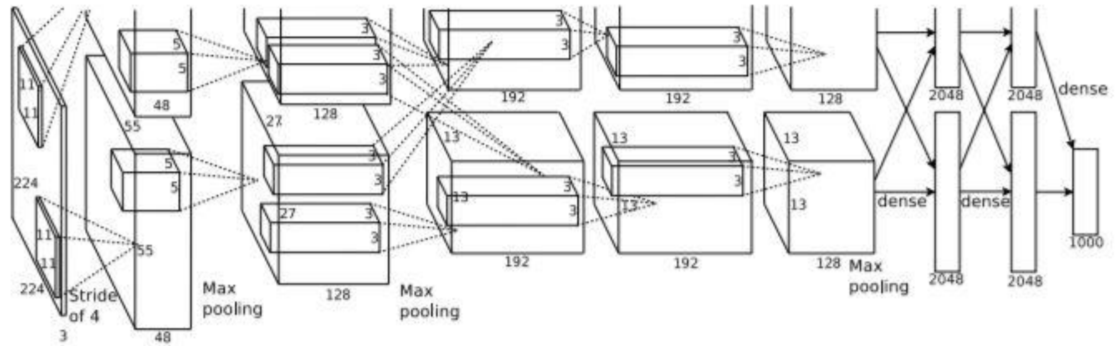

Figura 2.17: Arquitectura original de AlexNet. [26]

- **VGG-16:** El objetivo de los creadores era conseguir una red neuronal muy profunda para el reconocimiento de imágenes, pero reduciendo el número de hiperparámetros utilizados en las convoluciones y reducciones (Capas Convolucionales y Capas Pooling). Su nombre viene de *Visual Geometry Group* y el 16 es por el número de capas que poseen pesos para entrenar. Su arquitectura se puede observar en la *Figura 2.18.* Básicamente consiste en:
    - La primera y segunda capa son Capas Convolucionales con 64 filtros de 3x3 con stride 14 cada una de ellas.
    - Tras ellas, se aplica una Capa Max Pooling 2x2 con stride 2.
    - La cuarta y quinta capa son Capas Convolucionales con 128 filtros de tamaño 3x3 y stride 1.
    - La sexta es de nuevo una Max Pooling 2x2 con stride 2.
    - La séptima, octava y novena capa vuelven a ser Convolucionales con 256 filtros de 3x3 con stride 1.
    - A continuación, aparece una Max Pooling 2x2 con stride 2.
    - De la undécima a la decimotercera nos encontramos con tres Capas Convolucionales seguidas con 512 filtros de tamaño 3x3 y stride 1.
    - De nuevo, aparece una Max Pooling 2x2 con stride 2.
    - De la decimoquinta a la decimoséptima aparecen tres nuevas Convolucionales con 512 filtros de tamaño 3x3 y stride 1.
    - Tras ellas, una Max Pooling 2x2 con stride 2.
    - En la decimonovena y vigésima aparecen dos Capas Totalmente Conectadas con 4096 neuronas cada una.
    - La última capa es la capa de salida con la función softmax que devuelve una de las posibles 1000 clases.

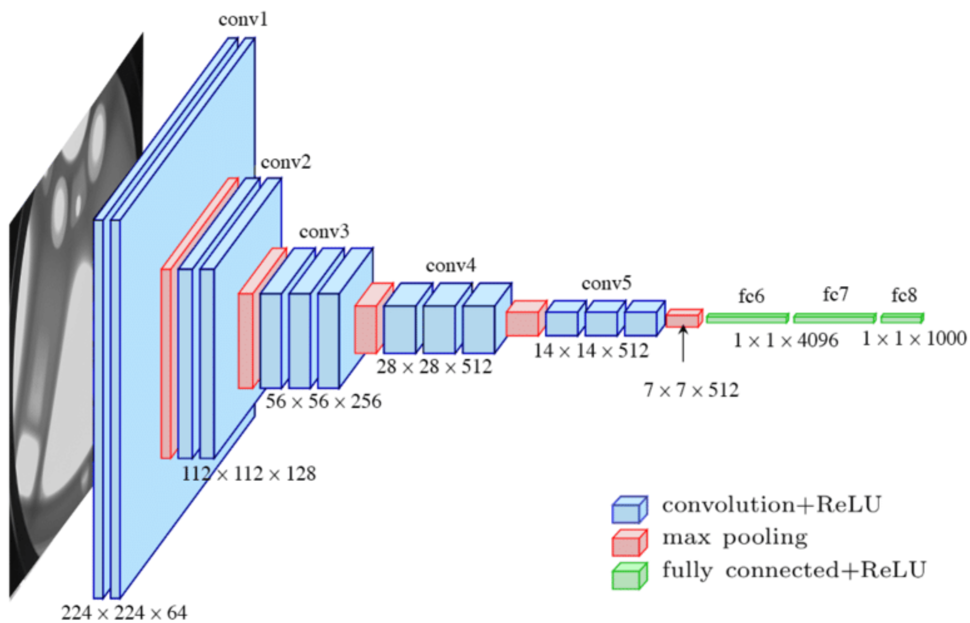

Figura 2.18: Arquitectura de VGG-16. En researchgate.net [27]

- **<u>ResNet:</u>** Las Redes Residuales (Residual Networks) son un tipo de redes neuronales convolucionales que vienen a resolver uno de los grandes problemas de las CNN. En redes neuronales muy profundas, los pesos se actualizan de forma proporcional al valor de la derivada de la función de error con respecto al peso actual. Si los gradientes son muy pequeños, los pesos prácticamente no cambian y por tanto el entrenamiento puede verse estancado. A este problema se le conoce como desvanecimiento del gradiente. Una de las posibles soluciones es trocear la red neuronal en bloques conectados y saltar el aprendizaje de ciertos pesos entre algunas capas. Esto permite entrenar un modelo con muchas capas ocultas, sin caer en el problema mencionado y conseguir mapear las características importantes de igual manera. Este es el principio de las ResNet. El termino residual viene de la conexión entre capas no consecutivas ya que a la capa destino le llega un residuo (valor de la función de activación) de la capa origen.

  Podemos observar un esquema de una ResNet en la *Figura 2.19.* En ella, podemos observar varios bloques residuales con diferentes capas en cada uno de ellos, así como las conexiones entre diferentes bloques.

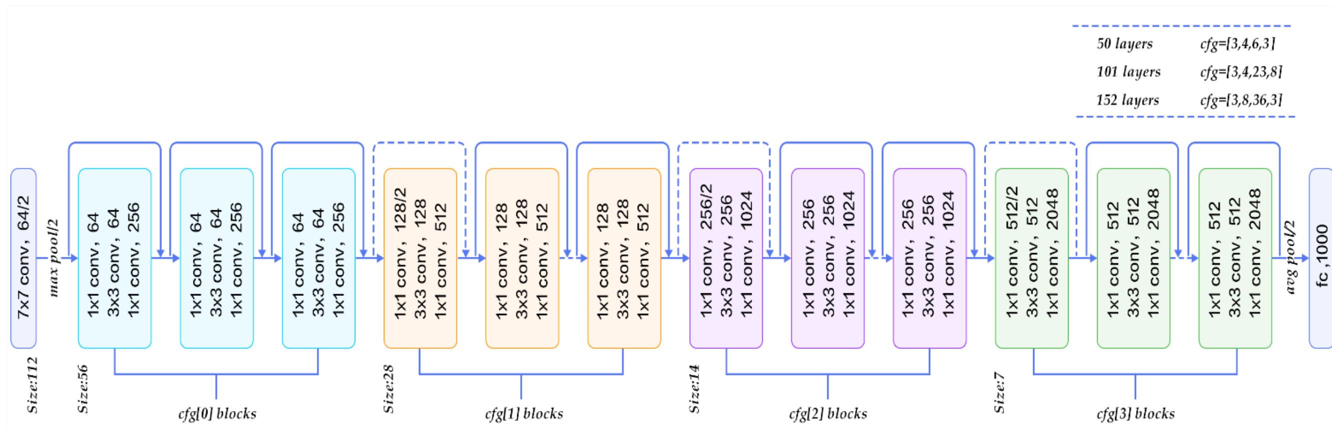

Figura 2.19: Ejemplo de una ResNet. En mpeker.com [28]

Para poder entender bien la arquitectura tenemos que definir sus dos bloques básicos. El bloque identidad es el bloque estándar usado en las ResNets y corresponde al caso donde la activación de entrada tiene la misma dimensión que la de salida. Un ejemplo se observa en la *Figura 2.20.*

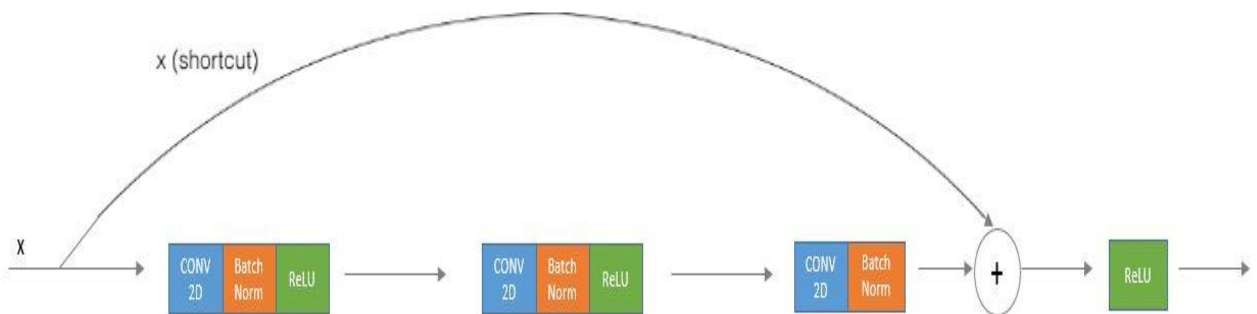

Figura 2.20: Bloque identidad. En engmrk.com [29]

Cuando estas dimensiones no son iguales, por ejemplo, tras aplicar una Capa Pooling, entonces añadimos una Capa Convolucional y formamos el bloque convolucional. Podemos observar un bloque convolucional en la *Figura 2.21.*

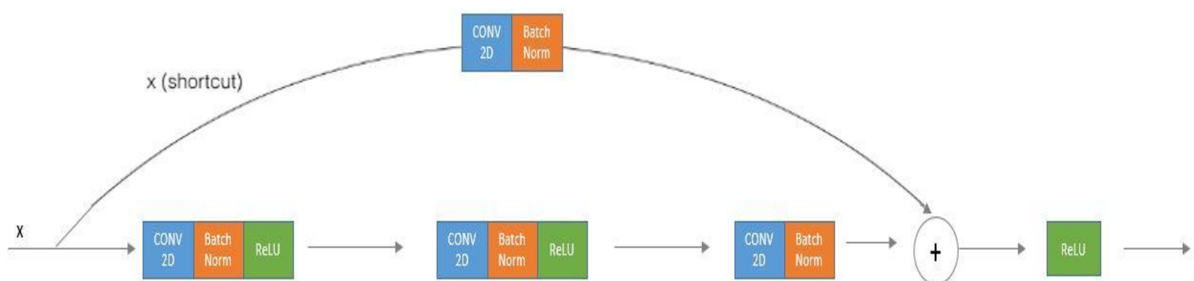

Figura 2.21: Bloque Convolucional. En engmrk.com [29]

- **Inception/GoogleNet:** Creada por Google en 2014. Su motivación principal es no tener que decidir entre un tipo de Convolución o un tipo de Pooling cuando podemos aplicar varios a la vez, concatenarlos y que la red ajuste sus valores. El primer problema a simple vista es desde el punto de vista computacional. El tener que computar varias operaciones de ese calibre a la vez, hace que el coste computacional sea enorme. La solución para ello es aplicar Convoluciones 1x1. Este tipo de convoluciones reduce el tamaño de la matriz de entrada previamente a realizar la convolución programada. Esta pre-reducción del tamaño permite reducir el número de parámetros a calcular.
  La propia red está formada por lo que se conoce como "Modulos Inception", como se observa en la *Figura 2.22.*

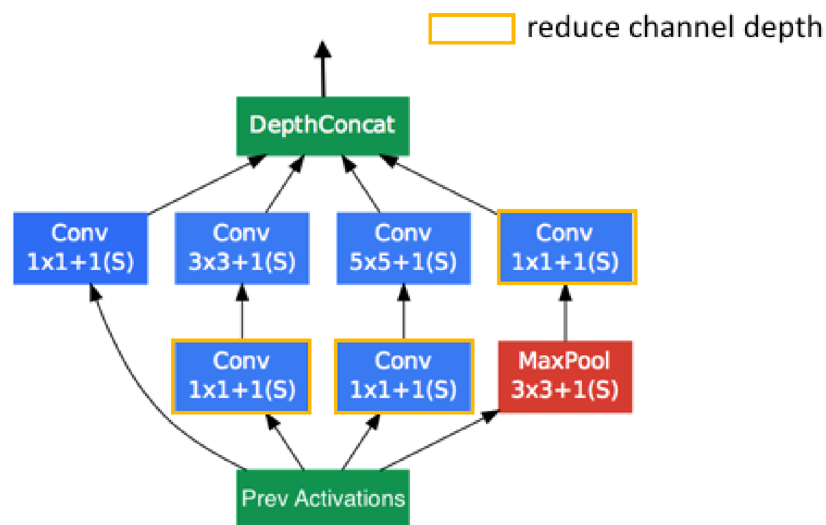

Figura 2.22: Ejemplo de un Módulo Inception. En jeremyjordan.me [30]

Cada módulo está compuesto de una función de activación previa, Capas Convolucionales 1 x 1, Capas Convolucionales *n x n* y Capas Pooling. El número y tipo de estas dependerá de su implementación concreta. Tras ello se concatenan todos los bloques para dar lugar al bloque final. La unión de varios módulos da lugar a la red Inception. Su arquitectura se puede observar en la *Figura 2.23*.

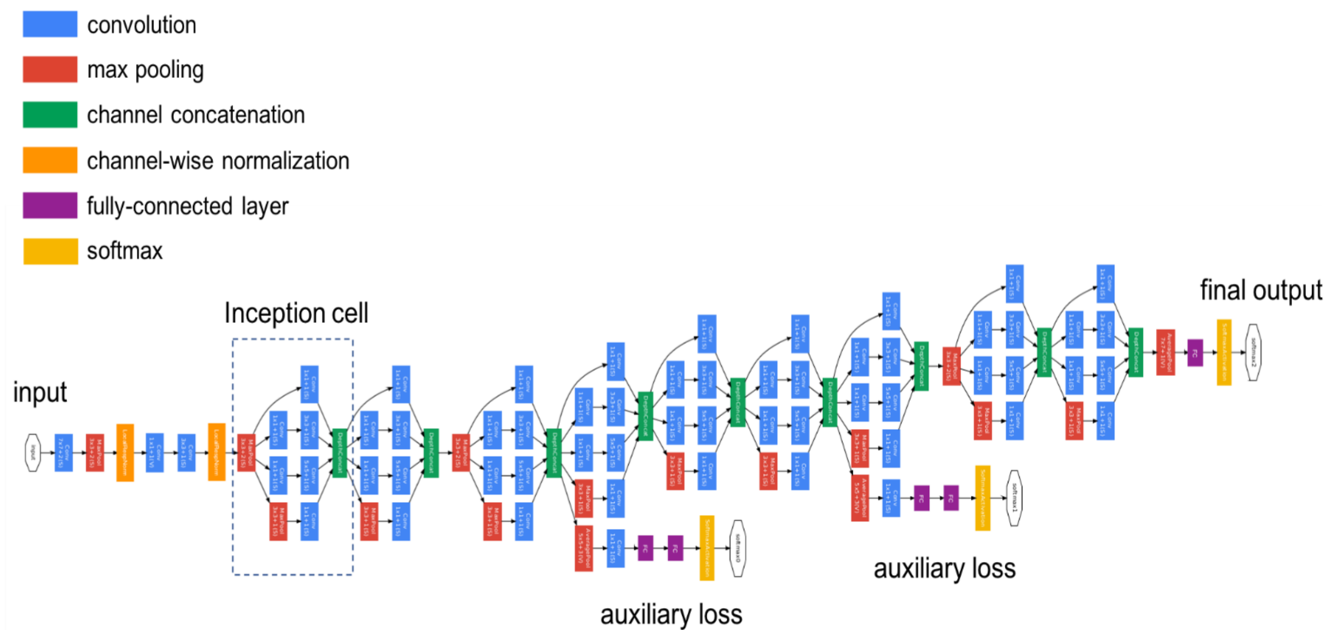

Figura 2.23: Arquitectura Red Inception. En jeremyjordan.me [30]

Pueden existir ciertas Capas Pooling para ajustar dimensiones y Capas Totalmente Conectadas al final de la red para devolver el resultado de la clase a predecir. Además, pueden existir ciertas ramas con Capas Totalmente Conectadas y capas de salida softmax en ciertas partes de la propia red. Esto se utiliza para verificar que la red, incluso en partes intermedias, es capaz de acertar varios ejemplos y también tiene un papel de regularización.

2.7 TRANSFER LEARNING

El Transfer Learning es una técnica utilizada en Deep Learning que consiste en aplicar un conocimiento previamente adquirido en una tarea en concreto y aplicarlo a un problema diferente, pero con cierta relación. Por ejemplo, si disponemos de cierto conocimiento para reconocer gatos podemos usar ese conocimiento para crear un modelo capaz de reconocer perros.

Cuando queremos crear un modelo para nuestra aplicación concreta, es posible que necesitemos de un número considerable de imágenes para entrenar y testear, así como, dependiendo de la complejidad del problema, gran capacidad computacional para llevar a cabo su resolución. En muchos casos, no seremos capaces de obtener más imágenes para mejorar el modelo o el coste computacional de generar nuestro modelo será inasumible.

Es aquí donde el Transfer Learning tiene mayor relevancia ya que podemos crear nuestro modelo a partir de redes neuronales profundas ya entrenadas. Esto se consigue importando la arquitectura y los pesos de la red neuronal a aplicar y tan solo tendríamos que entrenar los pesos de la última capa (la capa de salida) para aplicarlo a nuestro problema en concreto. El resto de pesos de la red neuronal no se entrenarían y quedarían congelados.

Otra posible variante dentro del Transfer Learning es la llamada Fine Tuning. Consiste en descongelar las ultimas capas de una CNN ya entrenada de manera que nuestro modelo debe ajustar los pesos de las capas no congeladas con el fin de adecuarse al problema específico. Es bastante utilizado en Transfer Learning ya que se utilizan las primeras capas de un modelo preentrenado como entrada del modelo a crear y se ajustan los pesos de las ultimas capas que extraen las características de nuestro problema concreto.

Hay que destacar que esta técnica funciona bien cuando transferimos el conocimiento de una tarea A a una tarea B que tienen la misma entrada (imágenes, audios, etc.) o cuando las características de las dos tareas son compartidas (bordes, texturas, etc.). También es posible su uso no solo cuando no poseemos suficientes datos de entrada como para no crear un modelo suficientemente robusto, sino que, aunque poseamos muchos datos, podemos utilizar los pesos de la red entrenada como inicialización, de forma que empecemos el aprendizaje de manera más encaminada.

Sin embargo, no hay que olvidar que puede haber casos donde esta técnica empeore el rendimiento. Si las tareas a utilizar no están muy relacionadas puede haber una caída del rendimiento al aplicar Transfer Learning que no ocurriría entrenando un modelo desde cero.

Cuando poseemos pocos datos de entrenamiento (lo cual es bastante común si consideramos utilizar Transfer Learning) y el modelo a encontrar es demasiado complejo o los datos de test no siguen la misma distribución que los datos de entrenamiento, es posible que el error en test sea mucho mayor que en train (también llamado sobreentrenamiento). Para evitar esto, existen un conjunto de técnicas diseñadas explícitamente para reducir dicho error. Entre ellas se encuentran:

- **Regularización L1, L2:** Se basan en limitar la capacidad de aprendizaje del modelo añadiendo un nuevo parámetro de penalización basado en la norma (L1 o L2) del conjunto de parámetros. De esta forma, los valores de los pesos se van reduciendo a medida que pasan las iteraciones que lleva a modelos más simples (también llamado *Weight Decay*).
- **Dropout:** Se basa en la idea de eliminar ciertas neuronas durante el entrenamiento de manera aleatoria y tener un ensemble de subredes que podemos obtener en base al número de veces que ha sido eliminada una neurona en test. De esta forma, en cada iteración hay una probabilidad de que se eliminen ciertas neuronas consiguiendo diversificar el conocimiento por toda la red.
- **Data Augmentation:** La mejor manera para que un algoritmo de Machine Learning generalice bien es entrenarlo con una mayor cantidad de datos. El problema es que estos suelen ser limitados. Para solventar este problema, esta técnica genera datos ficticios y los añade al conjunto de entrenamiento. Además, estos datos ficticios pueden ser modificados ligeramente de forma que añadamos más variedad al entrenamiento a la vez que incrementamos el número de ejemplos totales.
- **Early Stopping:** Cuando trabajamos con un modelo con suficiente capacidad como para sobreentrenar puede ocurrir que el error en train vaya disminuyendo en cada iteración, pero el error de test vaya aumentando. Esto significa que un modelo de iteraciones anteriores tiene mayor capacidad de generalización. De esta manera, si el error en test

disminuye, guardamos el modelo y evitamos quedarnos con un modelo con un mejor rendimiento en train pero considerablemente peor en test.

3. DESCRIPCIÓN DE LA PROBLEMÁTICA

En este apartado se comenzará explicando cuál es el problema concreto que queremos abordar (*Sección 3.1)*. A continuación, se realizará una explicación de los componentes del sistema embebido (*Sección 3.2)* incluyendo tanto el hardware (*Sección 3.2.1)* como el software (*Sección 3.2.2)* utilizado. Por último, se explicará cómo funciona el sistema final y cómo ha sido su integración (*Sección 3.3).*

3.1 PROBLEMA A RESOLVER

Las empresas dedicadas a la producción de productos precocinados tienen diversos problemas en sus procesos productivos. Éstos son muy variados y van desde la seguridad alimentaria y las medidas de higiene hasta la correcta distribución y abastecimiento a supermercados y restaurantes. Entre todos los problemas que pueden surgir, uno muy común y que puede representar un problema grave para el proveedor es la incorrecta impresión de la etiqueta del producto. En la etiqueta de cada producto están reflejados el tipo de producto, el nombre del cliente que lo pondrá a la venta, los ingredientes que lleva, advertencias para alérgicos, número de lote, etc. Todos estos elementos tienen más importancia de la que se puede creer en un primer momento, ya que la ausencia de alguno de ellos puede suponer que el consumidor final no conozca toda la información del producto. En el peor de los casos, esta situación puede llevar a un problema de salud alimentaria lo que acarrea multas de millones de euros e incluso el cierre de la propia empresa productora. Además de la etiqueta, pueden surgir ciertos errores de envasado que se detallarán más adelante que también es necesario detectar.

Para este proyecto, el proveedor de productos precocinados está especializado en el cocinado de tortillas de patata. Cuenta con numerosos centros de producción que suministran el producto a diferentes supermercados tanto a nivel nacional como internacional.

Para hacernos una idea de cómo es una línea de producción podemos recurrir a la *Figura 3.1*. En ella, observamos como las tortillas van sobre una cinta y, en primer lugar, llegan a la selladora que se encarga de poner el film mediante termosellado. A continuación, la etiquetadora es la encargada de imprimir la propia etiqueta que va sobre el film. Finalmente, la tortilla llega al sistema embebido que se describirá con detalle en la *Sección 3.3*. Dependiendo del resultado obtenido por el sistema embebido, se accionará el mecanismo de rechazo que básicamente baja una plataforma que desvía las tortillas a un almacén para su posterior revisión.

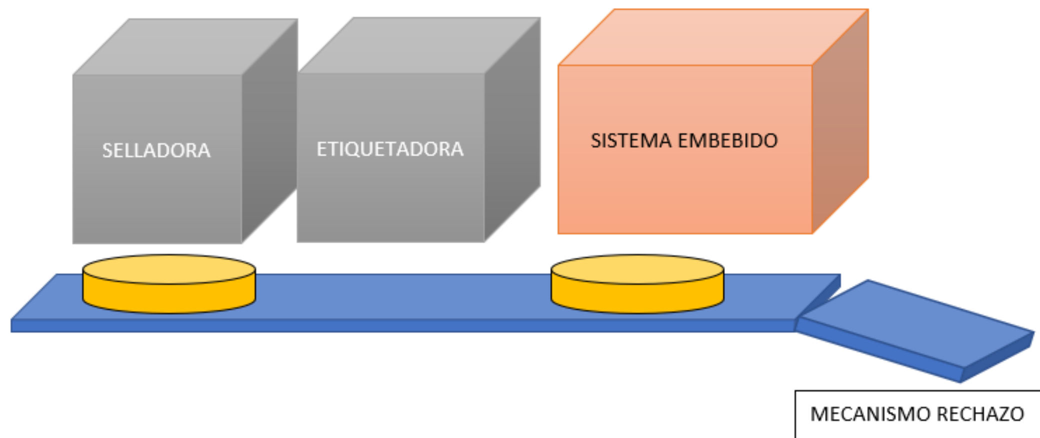

Figura 3.1: Representación de una línea de producción del cliente

Entre sus productos hay seis que copan la gran parte de su producción:

- Tortilla de Patata con Cebolla
- Tortilla de Papas con Cebolla
- Tortilla de Patata sin Cebolla
- Tortilla de Papas sin Cebolla
- Tortilla de Patata con Chorizo
- Tortilla de Papas con Chorizo

Realmente estas seis referencias podrían ser agrupadas en tres ya que la tortilla de patata y la tortilla de papas engloban al mismo producto, pero debido a que su etiqueta es diferente (y es el elemento que queremos analizar) consideraremos las seis referencias disponibles.

En las *Tabla 3.1* se muestran las referencias descritas anteriormente.

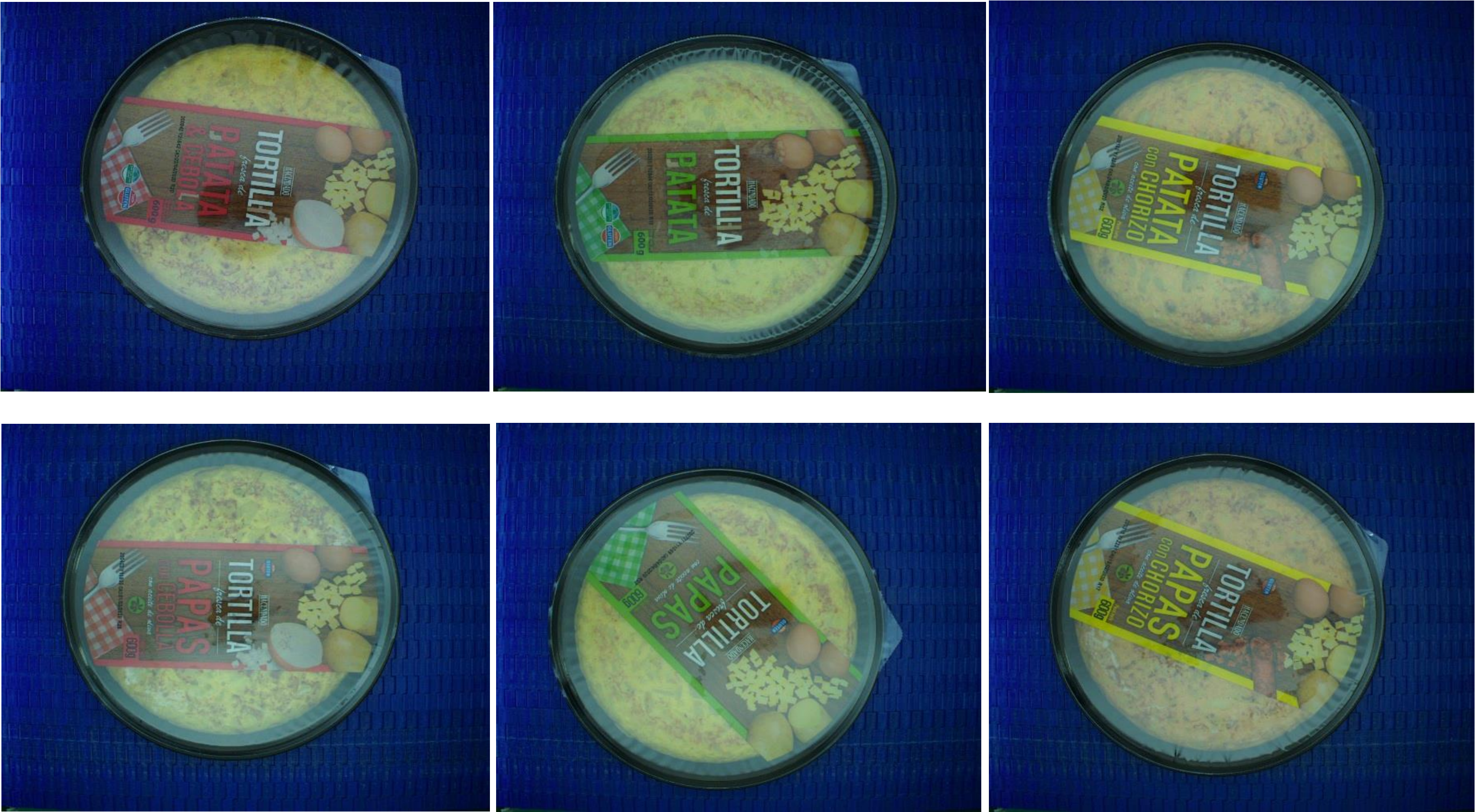

Tabla 3.1: Referencias del cliente. En la fila de arriba se muestran las referencias de "Patata" y en la fila de abajo las de "Papas"

A continuación, vamos a identificar cuáles son los problemas que el cliente manifiesta y que hay que conseguir detectar.

- <u>Ausencia de etiqueta:</u> Ocurre cuando la selladora falla y la tortilla no lleva film. Poco común. Podemos ver el defecto en la *Figura 3.2.*

Figura 3.2: Ausencia de Etiqueta

- Volteo de la tortilla: Ocurre cuando la tortilla entra en el sistema de visión con la parte de la etiqueta inferior hacia la cámara. Poco común. Se observa un ejemplo en la *Figura 3.3.*

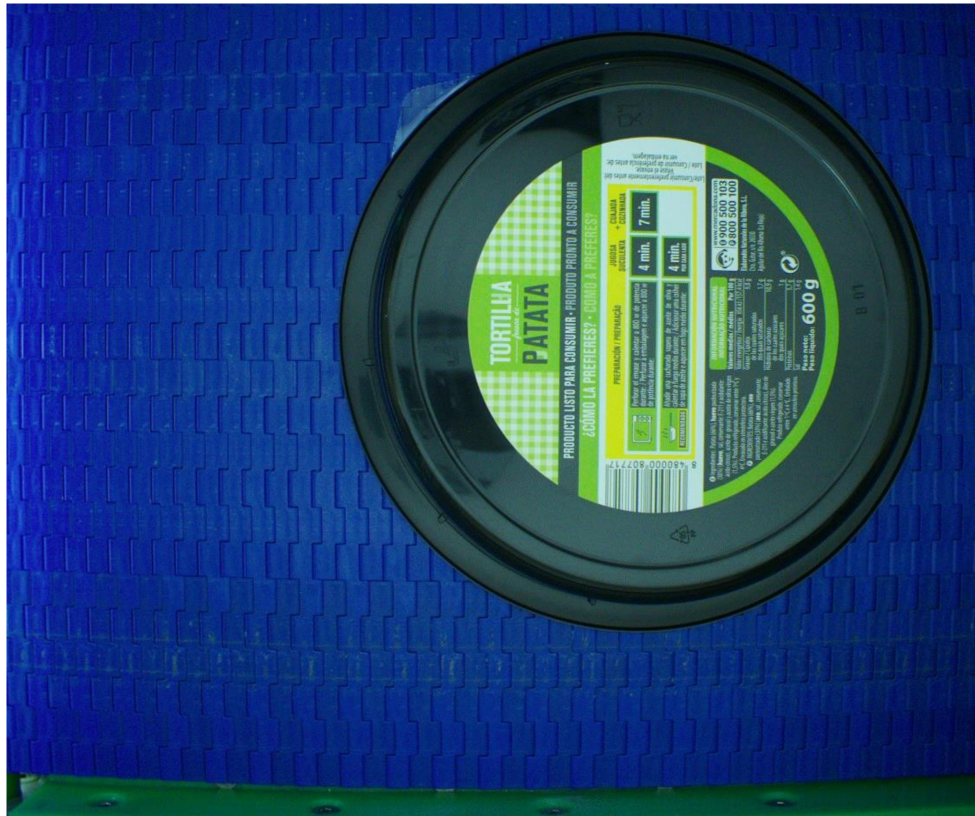

Figura 3.3: Tortilla volteada

- Ausencia de fecha de caducidad/lote: Ocurre cuando la etiquetadora no imprime la fecha de caducidad ni el lote sobre el film anterior. Es el fallo más común y uno de los más graves debido a que es un campo crítico de cara al consumidor final. Podemos observar el defecto en la *Figura 3.4* junto con un recuadro que indica dónde debiera aparecer.

Figura 3.4: Tortilla con ausencia de fecha de caducidad/lote

- Impresión de fecha de caducidad/lote fuera de lugar: Se da cuando la fecha de caducidad/lote está impresa, pero en otra parte de la etiqueta. Su posición correcta se encuentra en el hueco paralelo a los gramos del producto. Observamos el defecto en la *Figura 3.5* junto con un recuadro que indica la posición incorrecta de la fecha de caducidad/lote.

Figura 3.5: Tortilla con fecha de caducidad/lote fuera de lugar

- Presencia de más de una tortilla en la imagen: Cuando por alguna razón se amontonan los productos en la cinta, no podemos ser capaces de discernir si una tortilla es buena o mala ya que hay varias en la imagen y además en cualquier posición. Por tanto, esa imagen se clasificará como defectuosa. Podemos ver un ejemplo en la *Figura 3.6.*

Figura 3.6: Acumulación de tortillas en una misma imagen

- Falta de producto en el envase: En muy pocas ocasiones puede ser que aparezcan envases sin la propia tortilla debido a que la envasadora se ha activado sin producto o cualquier razón similar. Puede que la etiqueta esté bien impresa, pero al no haber producto hay que darla como defectuosa. Un ejemplo se observa en la *Figura 3.7.*

Figura 3.7: Ausencia del producto en el envase

3.2 COMPONENTES DEL SISTEMA EMBEBIDO

Vista la evidencia que existe sobre el uso de la visión artificial, el Deep Learning y las CNN podemos intuir como la integración de todo ello en un entorno industrial nos puede arrojar buenos resultados.

Para ello, es necesario la utilización de GPUs de manera que el uso de estas técnicas en Python pueda realizarse en tiempos razonables. Esto es debido a que, en sistemas que trabajan con imágenes, el número de cálculos a realizar es considerable y con el fin de reducir la velocidad de los mismos, las GPUs son capaces de realizar operaciones matriciales (una imagen se puede representar mediante matrices) en un corto periodo de tiempo. Esto ya se ha podido comprobar en numerosas ocasiones [31].

Para ello, llevaremos a cabo una aplicación embebida junto con una GPU de forma que consigamos una abstracción de todos los componentes con una entrada (producto a clasificar) y una salida (resultado del modelo).

Esta integración estará compuesta por un ordenador industrial, una cámara industrial Basler y una GPU RTX junto con la propia integración de Python y C++, lenguaje sobre el que corre la aplicación y que hace uso de todas las librerías necesarias para la agrupación de todos los componentes. Todo ello se detalla más adelante en la *Sección 3.3.*

Se espera obtener, por tanto, un sistema capaz de clasificar de forma acertada cada imagen que entra al sistema dando un resultado de forma que el proceso de producción de la empresa sea inteligente y automático.

El sistema creado consta tanto de parte hardware como de parte software. El objetivo es ser capaces de adquirir una foto de la tortilla, analizar sus defectos y devolver un resultado. Para ello, describiremos a continuación los componentes que lo forman. Más adelante, nos centraremos en cómo está formado el sistema, su integración y funcionamiento general.

3.2.1 HARDWARE

En este apartado se explicarán los componentes hardware que componen el sistema tales como una Cámara Basler (*Sección 3.2.1.1)*, un objetivo Fuji (*Sección 3.2.1.2)*, un ordenador Protech (*Sección 3.2.1.3)* y una Nvidia GForce RTX 2060 *(Sección 3.2.1.4).*

3.2.1.1 Cámara Basler

Basler AG es una empresa alemana especializada en hardware de visión tales como cámaras industriales, objetivos y demás accesorios utilizados en sistemas de visión artificial. En este proyecto se ha utilizado una cámara Basler acA2440-20gc para la obtención de imágenes en el entorno de producción. Además, para su funcionamiento es necesario alimentarla con 24V y disponer de un cable de red Ethernet de categoría 6 o superior. Este modelo de cámara permite obtener hasta 23 imágenes por segundo con una resolución de 2448 x 2048 a color [32]. Para este proyecto, se ha optado por este modelo debido a que es conveniente disponer de un mínimo de resolución, ya que es necesario identificar las presencias y ausencias de elementos vistos en la *Sección 3.1* y reduciendo la resolución podríamos no ser capaces de distinguirlos. Además, trabajaremos con imágenes en color ya que nos ofrece más opciones que hacerlo en monocromo. La cámara se conecta al ordenador mediante interfaz GigE [33].

Figura 3.8: Cámara utilizada en el proyecto. En baslerweb.com [32]

3.2.1.2 Objetivo Fuji

Fujifilm es una empresa japonesa especializada en hardware óptico y es uno de los líderes en el sector de la fotografía y la óptica. Para el proyecto se usado un objetivo Fuji HF9HA-1S de 9 mm de distancia focal. Esto nos permite el obtener una imagen de la cinta en su totalidad sin obtener demasiada distorsión que modifique la imagen real. Si usáramos un objetivo de 6 mm, por ejemplo, podríamos abarcar mucho más trozo de cinta pero los bordes de la imagen saldrían distorsionados [34]. Si, por el contrario, hiciéramos uso de un objetivo de 14mm obtendríamos menos campo de visión y puede que la tortilla nos saliera cortada.

Figura 3.9: Objetivo utilizado en el proyecto. En iberoptics.com [35]

3.2.1.3 Ordenador Protech Systems

Protech System es una empresa especializada en el diseño y manufacturación de equipos industriales. Proporcionan y ensamblan ordenadores por componentes a medida del cliente de forma que éste configura su ordenador en base a sus propias necesidades. En este proyecto se ha utilizado un ordenador Protech SP-6265RA-A1B que consta de un procesador Intel i7, 8 GB de RAM y dos discos duros SSD con 60 GB y 500 GB.

Figura 3.10: Vista lateral del ordenador utilizado. En protech.com.tw [36]

3.2.1.4 Gráfica Nvidia GeForce RTX 2060

Como hemos visto anteriormente, el uso del Deep Learning se ha ido haciendo más frecuente en el ámbito de la investigación y también en el ámbito industrial y es por ello que el uso de GPUs para el procesamiento de imágenes está claramente en auge. Nvidia es una empresa especializada en el desarrollo de unidades de procesamiento gráfico que se ha diferenciado del resto en este aspecto [37]. La creación de arquitecturas especializadas en la computación en paralelo (CUDA) así como librerías especializadas en redes neuronales (cuDNN) para su uso ha hecho que la demanda de tarjetas gráficas de esta empresa haya aumentado considerablemente. Para este proyecto ha sido necesaria la adquisición, instalación y configuración de una tarjeta gráfica Nvidia GeForce RTX 2060 para poder agilizar los cálculos en el uso de las CNN. Este modelo es capaz de obtener una frecuencia de reloj normal de 1365 MHz y dispone de 1920 Núcleos CUDA [38]. Va acoplada dentro del equipo Protech.

Figura 3.11: Tarjeta gráfica usada en el proyecto. En nvidia.com [38]

3.2.2 SOFTWARE

A continuación, se describen todos los programas, lenguajes y APIs que se utilizan en el proyecto. Hay que remarcar que todo se ve reflejado en la aplicación que correrá sobre el ordenador. Dicha aplicación se explicará con detalle en la *Sección 5.*

3.2.2.1 Windows 10

El sistema operativo que corre sobre el ordenador mencionado es Windows 10 Enterprise LTSC. Se escogió este sistema operativo para evitar problemas de incompatibilidades con la integración de programas, drivers y las APIs que se van a utilizar.

3.2.2.2 QTCreator

QTCreator es un IDE multiplataforma para el desarrollo multiplataforma con Interfaces Graficas de Usuario (Graphical User Interface (GUI), en inglés). Soporta los sistemas operativos Linux, Mac OS X y Windows. Permite la configuración de diversos kits de compilación, debugging, frameworks, etc. [39]. Es la herramienta utilizada para la creación de la aplicación que maneja todo el proyecto y en la que el usuario verá los resultados.

3.2.2.3 Python

Es un lenguaje interpretado, dinámico y multiplataforma que nació en 1991 y de código abierto. Su gran potencial es tanto su lenguaje legible como la cantidad de librerías de gran calidad que posee. La gran mayoría orientada a Data Science por lo que nos es de gran utilidad en este caso. Algunos ejemplos de estas librerías son TensorFlow, Scikit-Learn, Numpy, Keras, Pandas, etc. [40].

La creación de modelos, entrenamiento de las CNN, scripts y pruebas sobre las redes neuronales y que se describen en la *Sección 4* se han realizado sobre este lenguaje para su posterior integración en la aplicación principal.

3.2.2.4 Python/C API

Es la API que Python proporciona para su utilización en los lenguajes C y C++. Su uso es debido principalmente a dos razones. La primera es escribir extensiones de módulos para ámbitos específicos; es decir, módulos de C que se escapan del intérprete de Python. La segunda razón es usar Python como componente de otra aplicación; técnicamente referido como Python embebido en una aplicación [41]. Éste es nuestro objetivo en este proyecto de manera que nuestra aplicación vaya usando los módulos y características de Python en integración con el funcionamiento de otras librerías y lenguajes dentro del mismo sistema.

3.2.2.5 C++

Es un lenguaje creado en 1979 basado en C, pero con mecanismos que permiten la manipulación de objetos. Es un lenguaje de programación multiparadigma debido a que permite su uso tanto de programación orientada a objetos, programación estructurada y programación genérica [42]. La aplicación en la que está basado el proyecto está realizada en este lenguaje, ya que es un lenguaje utilizado en multitud de APIs y por tanto permite la integración de todos ellos en una misma aplicación. Concretamente la API de Basler y la API de Python se han utilizado en este lenguaje.

3.2.2.6 Pylon SDK

Es una suite de desarrollo de software de Basler que permite hacer uso de APIs en C, C++y .NET para la conexión, configuración y desarrollo de aplicaciones sobre cámaras Basler [43]. En el proyecto se ha usado concretamente la API de C++ para llevar a cabo toda la comunicación con la cámara descrita en la *Sección 3.2.1.1.*

3.3 SISTEMA EMBEBIDO

En este apartado, se explicará el funcionamiento y la integración de las tres secciones diferenciadas que consta la aplicación. Por un lado, tenemos la conexión y configuración de la cámara (*Sección 3.3.1),* a continuación, la creación de una base de datos de imágenes (*Sección 3.3.2)* y, por último, la aplicación de un modelo creado sobre una imagen (*Sección 3.3.3).*

3.3.1 CONEXIÓN Y CONFIGURACIÓN DE LA CÁMARA

Antes de empezar a utilizar alguno de los modos que se describirán en los siguientes apartados, es necesario haber realizado previamente una conexión y configuración correcta de la cámara en el entorno en el que se vaya a utilizar. De esta manera, estaremos seguros de que tenemos conexión con la cámara y, a partir de ahí, podemos configurar los parámetros de la misma con el fin de obtener una imagen de calidad. Para ello, es necesaria la integración de la API de Pylon con C++.

En primer lugar, es necesario la inicialización del sistema de Pylon. Para ello, ejecutamos PylonInitialize() de manera que sea la primera instrucción que se realice antes de llamar a cualquier otra función de la propia API.

Tras ello, debemos crear una instancia de la cámara para poder conectarnos y hacer uso de las funcionalidades propias de ese modelo de cámara. Para crear una instancia de la cámara Basler realizamos:

CBaslerUniversalInstantCamera camera (CTlFactory::GetInstance().*CreateFirstDevice*())

En donde el primer argumento del constructor CTlFactory::GetInstance().*CreateFirstDevice*() hace referencia al primer dispositivo conectado al equipo.

Tras ello, ya podemos activar la captura de imágenes con la siguiente llamada: camera.*StartGrabbing*(GrabStrategy_OneByOne)

En este caso se ha optado por una estrategia de captura de imágenes de una en una (GrabStrategy_OneByOne) ya que es lo apropiado para nuestra aplicación. Se podrían configurar como adquisición en continuo, con un número de imágenes fijo, etc.

A continuación, definimos un puntero a un buffer que será el encargado de almacenar la imagen: CGrabResultPtr ptrGrabResult

Ahora pasaremos a adquirir la imagen con la siguiente llamada:

camera.*RetrieveResult*(2000, *ptrGrabResult*,TimeoutHandling_ThrowException)

En este caso, la cámara nos devuelve el resultado de la imagen y la almacena en el buffer referenciado anteriormente. El primer argumento de la llamada corresponde al tiempo en milisegundos que esperamos a que la cámara nos devuelva el resultado (también conocido

como timeout). El último argumento es el manejador de excepciones, que lanzará una excepción con un código de error específico cuando dicho timeout se produzca.

Para utilizar la imagen recogida en el buffer lo primero es crear una instancia de un objeto de imagen de Pylon: Pylon::CPylonImage image

Tras ello, vinculamos los datos almacenados en el buffer con la imagen instanciada anteriormente de la forma: image.*AttachGrabResultBuffer*(ptrGrabResult)

En este momento, ya tenemos la imagen en la variable image y ya podemos mostrarla, guardarla o lo que creamos conveniente.

Para llevar a cabo la configuración de parámetros es necesario la utilización de alguna llamada más de la propia API. En la *Sección 5.2* se mostrarán tres parámetros de la cámara que tienen bastante relevancia a la hora de obtener una imagen de calidad. Uno de ellos es el Tiempo de Exposición (se explica en la *Sección 5.2.1*). Para poder configurarlo, debemos llamar a la función:

camera.ExposureTimeAbs.*SetValue*(tiempo_Exposicion)

En este caso, el parámetro de este modelo de cámara en concreto es ExposureTimeAbs y el valor que queramos asignarle será tiempo_Exposicion. De esta manera cuando capturemos una nueva imagen, ésta ya se habrá adquirido con este nuevo tiempo de exposición.

De esta forma tan sencilla, es posible cambiar tantos parámetros como consideremos con el fin de mejorar la calidad de la imagen.

Por último, no hay que olvidar la llamada a PylonTerminate() una vez hayamos acabado para liberar todos los recursos utilizados hasta el momento.

3.3.2 CREACIÓN DE UNA BASE DE IMÁGENES PARA ENTRENAMIENTO

La segunda funcionalidad de la aplicación es la posibilidad de etiquetar cada imagen obtenida por la cámara como OK/NOK. Para ello se hace uso de la API de Pylon en C++ tal y como se ha visto en el apartado anterior para capturar imágenes. Una vez capturada la imagen correspondiente, el usuario de la aplicación deberá etiquetar cada una de ellas dependiendo de si el producto lleva defecto o no. En la *Sección 3.1* se han visto los posibles defectos que pueden surgir a la hora de clasificar las tortillas. En este caso, todos los defectos se van a englobar como NOK de manera que podamos encontrar un modelo capaz de generalizar de forma correcta cualquier tipo de defecto de cualquier referencia que se produzca.

Cuando se etiqueta una imagen como OK, ésta se almacena en una carpeta llamada "OK" dentro del propio directorio donde corre la aplicación. De manera análoga ocurre con las "NOK". De esta forma, una vez acabada la clasificación de imágenes ya tenemos en ambas carpetas las imágenes clasificadas y podemos pasar a su posterior tratamiento. Dicho tratamiento se explicará con detalle en la *Sección 4* así como los resultados obtenidos con los modelos creados.

3.3.3 EJECUCIÓN DEL MODELO CREADO SOBRE LA IMAGEN

Una vez hayamos creado un modelo de Deep Learning con Python (*Sección 4*), debemos integrarlo con la aplicación para llevar a cabo la inferencia de cada imagen capturada en la línea de producción. Para ello, son necesarias dos integraciones diferentes. Por un lado, la integración de la API de Pylon tal y como hemos visto en la *Sección 3.3.1* para la captura de imágenes y, por otro lado, la integración de Python con la propia aplicación escrita en C++. Ésta última se describe a continuación.

Para esta integración se ha utilizado la API PYTHON/C que proporciona Python para realizar aplicaciones embebidas.

El primer paso es la creación del inicializador del intérprete de Python que se consigue mediante Py_Initialize(). Debe ser llamada antes de usar cualquier otra función de la propia API.

Tras ello, debemos importar el módulo de Python que hemos implementado con las funciones que vayamos a utilizar. Para ello, utilizamos la función PyImport_Import(module_Name), donde module_Name es el nombre del módulo escrito en Python. Ésta última variable debe de ser de tipo CPyObject por lo que antes de ello deberemos convertir el nombre del módulo a importar en una variable de ese tipo:

CPyObject module_Name = PyUnicode_FromString("MODULO_PYTHON")

Ahora ya podemos importar la función a utilizar dentro del módulo. Para ello, hacemos uso de la función:

CPyObject function_Name = PyObject_GetAttrString(Module, "applyInferenceName")

Donde Module es el objeto que contiene el módulo anteriormente importado y "applyInferenceName" es el nombre de la función de Python dentro del propio módulo.

Esta función recibe como argumento de entrada el nombre de la imagen a clasificar. Por tanto, debemos de cargar este argumento antes de hace la propia llamada. Para ello utilizaremos una tupla: CPyObject tuple = PyTuple_New(1)

De esta forma definimos una tupla con una posición (el nombre de la imagen). La rellenamos de la siguiente manera: PyTuple_SetItem(tuple, 0, PyUnicode_FromString(image_Name))

En este punto, solo nos queda hacer la llamada a la función pasándole el argumento definido.

CPyObject pValue = PyObject_CallObject(function_Name,tuple)

Debido a que la función de Python nos devuelve un valor (Imagen OK/Imagen NOK) recogeremos ese valor y realizaremos lo que consideremos con cada resultado.

Por último, no hay que olvidarse de destruir el intérprete y todos los recursos utilizados llamando a Py_Finalize().

Cabe destacar que se ha utilizado un script extra para manejar de forma automática todas las instancias creadas y control de punteros.

Para entender un poco más cómo es el flujo interno de la aplicación cuando estamos aplicando un modelo sobre una imagen, podemos recurrir a la *Figura 3.13.*

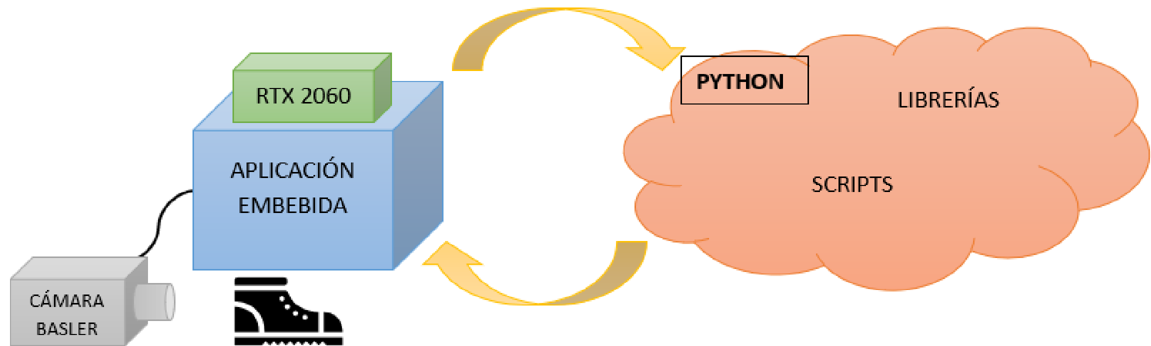

Figura 3.12: Flujo de la aplicación cuando se aplica Inferencia

Podemos ver como cuando obtenemos una foto a través de la cámara para evaluar el objeto en cuestión, ésta va directamente a la aplicación (haciendo uso de la API de Pylon). Una vez aquí, se realiza el proceso descrito anteriormente de forma que, mediante esta integración de Python en C++, se prepara a la aplicación para cargar y recibir las llamadas a los scripts y librerías de Python. Una vez que la imagen se encuentra en el contexto de Python, es aquí donde se hace uso de las librerías de Python y donde se aplica el modelo sobre la imagen utilizando para ello el potencial que nos ofrece la GPU. Cuando hemos obtenido el resultado, es entonces cuando la aplicación vuelve a tener el control, de manera que recibe el resultado de la imagen y lo muestra por pantalla.

Cabe destacar que este es el modo que se utiliza en producción, de manera que una vez que se ha creado un modelo suficientemente bueno para el caso en concreto, se vaya aplicando a cada imagen que pasa por el sistema (*Figura 3.1).*

4. APRENDIZAJE DE UN MODELO BASADO EN TRANSFER LEARNING PARA UNA APLICACIÓN INDUSTRIAL

En este apartado se mostrarán todos los procedimientos llevados a cabo a la hora de trabajar con las imágenes con el fin de obtener modelos basados en Transfer Learning que sean de utilidad para nuestro problema concreto. En concreto, se procederá a la descripción del problema (*Sección 4.1*), seguida de la metodología utilizada *(Sección 4.2*). Después se pasará a explicar de forma detallada todos los experimentos y pruebas realizadas con diferentes modelos (*Sección 4.3*) y posteriormente haremos una valoración y discusión de los resultados obtenidos (*Sección 4.4*).

4.1 DESCRIPCIÓN DEL PROBLEMA

Tras tener una perspectiva global del problema que se quiere resolver (*Sección 3*), el problema específico se centra en encontrar ciertos modelos que sean capaz de clasificar correctamente una tortilla OK/NOK procedente del centro de producción y que, por tanto, sea capaz de generalizar de forma correcta. Para ello, se quiere construir un modelo en Python capaz de obtener un buen rendimiento en cualquiera de las referencias que se han explicados en la *Sección 3.1*. Es por ello que las imágenes que se van a utilizar como entradas para los diferentes modelos van a consistir en un agrupamiento de todas las referencias. De esta forma, el modelo será capaz de discernir entre cualquier tipo de tortilla independientemente de la referencia que sea e incluso podría llegar a darse el caso de que generalizara bien incluso con referencias que no han sido tratadas anteriormente. Aunque hay que resaltar que no tiene por qué ser así, y el rendimiento probablemente sea mejor si el conjunto de datos de entrada sigue la misma distribución que los datos sobre los que se aplica el proceso de inferencia.

4.2 METODOLOGÍA

En este apartado se va a describir los pasos llevados a cabo para obtener las imágenes con las que entrenar los modelos, el preprocesamiento que se llevó a cabo para poder tener los datos en el formato requerido, así como la preparación de los mismos antes de pasar al proceso de entrenamiento y obtención de resultados.

4.2.1 EXTRACCIÓN DE LOS DATOS

En un primer lugar, ha sido necesaria la recopilación de las imágenes para poder llevar a cabo la construcción del modelo. Para ello, se ha llevado a cabo la utilización de la funcionalidad de "ENTRENAMIENTO" (*Sección 5.2.2*). Se han obtenido diferentes números de imágenes de cada referencia en base al porcentaje de producción que representa cada una de las referencias. De esta forma se han obtenido los siguientes números:

1. **REFERENCIA 1 (TORTILLA DE PATATA CON CEBOLLA):** 3255 imágenes
2. **REFERENCIA 2 (TORTILLA DE PATATA SIN CEBOLLA):** 2945 imágenes
3. **REFERENCIA 3 (TORTILLA DE PAPAS CON CEBOLLA):** 129 imágenes
4. **REFERENCIA 4 (TORTILLA DE PAPAS SIN CEBOLLA):** 102 imágenes
5. **REFERENCIA 5 (TORTILLA DE PATATA CON CHORIZO):** 1107 imágenes
6. **REFERENCIA 6 (TORTILLA DE PAPAS CON CHORIZO):** 21 imágenes

En total se han recopilado 7559 imágenes de las seis referencias. Esta distribución no uniforme y desbalanceada de las diferentes referencias es debido a la propia producción de la empresa ya que se demanda mucho más un tipo de referencia que otra y eso influye directamente en la cantidad de imágenes que se puede obtener de cada una de ellas. Una vez recopilada la información necesaria, procedemos al preprocesamiento de los mismos para su posterior utilización mediante Python.

4.2.2 PREPROCESAMIENTO DE LOS DATOS

Una vez obtenidas las imágenes, se ha procedido a clasificar de manera manual cada una de ellas de manera que por cada referencia teníamos dos conjuntos: OK y NOK. Es un proceso laborioso debido a la cantidad de imágenes y a la sutileza de algunos defectos.

Una imagen era clasificada como NOK si presentaba un defecto de los presentados en la *Sección 3.1*. Si no presentaba ninguno de ellos, entonces se etiquetaba como OK. De esta forma, la distribución de las imágenes de cada referencia quedaba de la siguiente forma:

1. **REFERENCIA 1 (TORTILLA DE PATATA CON CEBOLLA):** OK (2919 imágenes) y NOK (336 imágenes)
2. **REFERENCIA 2 (TORTILLA DE PATATA SIN CEBOLLA):** OK (2650 imágenes) y NOK (295 imágenes)
3. **REFERENCIA 3 (TORTILLA DE PAPAS CON CEBOLLA):** OK (49 imágenes) y NOK (80 imágenes)
4. **REFERENCIA 4 (TORTILLA DE PAPAS SIN CEBOLLA):** OK (90 imágenes) y NOK (12 imágenes)
5. **REFERENCIA 5 (TORTILLA DE PATATA CON CHORIZO):** OK (1030 imágenes) y NOK (77 imágenes)
6. **REFERENCIA 6 (TORTILLA DE PAPAS CON CHORIZO):** OK (3 imágenes) y NOK (18 imágenes)

Cada una de las imágenes están obtenidas en formato RAW por la propia cámara. En nuestro caso, este formato es .tiff y la dimensión de cada imagen es de 2448 x 2048 píxeles a color. Esto implica que cada imagen tenga un tamaño de 2448 x 2048 x 3 = 15040512B ~ 14 MB. De esta manera y con 7559 imágenes que disponíamos, el tamaño total de todas las imágenes era de unos 103 GB lo que hacía difícil tanto el poder cargar las propias imágenes, como el propio procesado de las mismas ya que el tiempo necesario para ello se demoraba demasiado a la vez que producía errores por exceso de recursos de los componentes.

Por estas razones se optó por convertir las imágenes a formato .jpg. Este tipo de formato mantiene un buen ratio *tamaño_imagen/calidad* y observamos como los defectos se seguían detectando a simple vista de forma bastante clara. Por tanto, se optó por convertir todas las imágenes a este formato manteniendo las dimensiones de cada una de ellas. Con esta conversión el tamaño de todas las imágenes pasó a 2,49 GB.

Debido a la gran dimensionalidad de las imágenes y con el fin de reducir los cálculos a realizar por las CNN, se ha optado por aplicar una segmentación a cada imagen, de forma que separemos el fondo de la propia tortilla y nos quedemos solo con esta para quitar pixeles innecesarios correspondientes a la cinta azul y, por tanto, poder hacer más hincapié en la impresión de la etiqueta. En la *Figura 4.1* se muestra una comparativa. A la izquierda, una imagen original de 2448 x 2048 en formato original (14,3 MB), la misma imagen de 2448 x 2048 en formato .jpg (345 KB) en el centro y, a la derecha, la imagen de 1780 x 1720 tras aplicar la segmentación en formato .jpg (164 KB).

Figura 4.1: Muestra de diferentes formatos de imagen durante el preprocesamiento

A partir de aquí, y debido a que la dimensionalidad de las imágenes aún era alta (1780x1720) se ha optado por reducirla hasta 990x990 que es lo que se ha considerado como una resolución mínima para observar los defectos expuestos en la *Sección 3.1.*

4.3 EXPERIMENTOS Y PRUEBAS REALIZADAS

En esta sección se explicará cómo se han implementado los modelos que han servido para realizar el proceso de inferencia visto en la *Sección 3.3.3*. Se ha llevado a cabo la creación y testeo

de diferentes arquitecturas con diferentes parámetros con el objetivo de encontrar el modelo que mejor funcione en nuestro problema concreto. Para poder llevar a cabo la implementación de los modelos se ha optado por utilizar las librerías de Python que se mencionan en la *Sección 5.1.4; Tensorflow y Keras.*

Concretamente se va a pasar a explicar tres modelos que han sido empleados para la obtención de los resultados. La idea es evaluar el rendimiento en una red neuronal convolucional sencilla e ir añadiendo diferentes técnicas hasta llegar a un modelo más completo y que, a priori, obtenga un mejor rendimiento.

Cabe destacar que debido al reducido número de imágenes que se poseen, así como a la posibilidad de que una tortilla pueda caer en cualquier posición se ha optado por utilizar *Data Augmentation* en todos los modelos que se van a explicar. Estos ejemplos ficticios se van a componer de rotaciones en 360 grados ya que es la única variación que puede haber en el entorno real de producción. Además, se ha añadido en todos los modelos la técnica de *Dropout* con el fin de intentar evitar el sobreentrenamiento en train, propio de este tipo de problemas con pocas imágenes. Como batch size se han utilizado 8 imágenes (debido a las dimensiones) y como optimizador Adam.

4.3.1 CNN BÁSICA

El primer modelo que se creó y el que consideraremos como base es una CNN con la estructura mostrada en la *Tabla 4.1*.

Tipo Capa	**Dimensión de salida**	**Función de Activación**	**Número de parámetros**
Capa Convolucional	988, 988, 16	ReLu	448
Max Pooling	494, 494, 16		0
Capa Convolucional	492, 492, 64	ReLu	9280
Max Pooling	246, 246, 64		0
Capa Convolucional	244, 244, 128	ReLu	73856
Max Pooling	122, 122, 128		0
Capa Convolucional	120, 120, 128	ReLu	147584
Max Pooling	60, 60, 128		0
Flatten	460800		0
Full Connected	512	ReLu	235930112

Droput	512		0
Full Connected	512	ReLu	262656
Dropout	512		0
Full Connected	2	Sigmoid	1026

Tabla 4.1: Arquitectura de la CNN básica

De esta manera obtenemos un modelo con 4 *Capas Convolucionales*, 4 *Capas Max Pooling*, 2 *capas Dropout*, 2 *capas ocultas Full Connected* y una capa de salida con dos neuronas ya que tenemos dos clases a clasificar y una función sigmoidea que nos dirá la probabilidad de pertenecer a cada clase. En total, 236424962 parámetros a entrenar.

4.3.2 TRANSFER LEARNING (VGG16)

El segundo modelo y que viene a mejorar al modelo anterior (4.3.1) consiste en la utilización de Transfer Learning para conseguir mejorar la extracción de características. Para ello, se ha utilizado el modelo VGG16 que fue explicado en la *Sección 2.6.* Concretamente, se ha importado toda la arquitectura del modelo, así como los pesos entre capas ocultas y se le ha eliminado la última capa de salida del modelo original sustituyéndola por la configuración de la *Tabla 4.2.* Los pesos importados corresponden a los propios del modelo VGG16 que ha sido pre-entrenado con el dataset ImageNet.

Tipo Capa	**Dimensión de salida**	**Función de Activación**	**Número de parámetros**
Modelo VGG16	460800		14714688
Full Connected	512	ReLu	235930112
Dropout	512		0
Full Connected	512	ReLu	262656
Dropout	512		0
Full Connected	2	Sigmoid	1026

Tabla 4.2: Arquitectura del modelo con Transfer Learning (VGG16)

De esta manera, conectamos todas las salidas del modelo preentrenado VGG16 con una capa *Full Connected* de 512 neuronas que a su vez se conecta con otra capa *Full Connected* de 512 neuronas. Por último, tenemos la capa de salida que será igual que en el modelo anterior. En este caso el número total de parámetros es de 250908482 de los cuales 236193794 (94%) se modificarán en el entrenamiento y que se corresponden con las capas *Full Connected* descritas en la *Tabla 4.2.*

4.3.3 TRANSFER LEARNING (VGG16) CON FINE-TUNING

Por último, se ha probado con una modificación sobre el modelo 4.3.2. En este caso, la diferencia consiste en descongelar las últimas capas convolucionales de la arquitectura VGG16 para pasar a entrenar los pesos de dichas capas con los datos de nuestro problema. En concreto, se han descongelado los bloques 4 y 5 de VGG16. Con esta modificación, se pretende obtener un mejor rendimiento ya que los pesos de las últimas capas del modelo son más específicos para nuestro problema. De igual forma, seguimos aprovechando el poder del Transfer Learning ya que no tenemos que entrenar toda la red de cero con nuestros datos lo que resultaría muy costoso en el tiempo. Podemos observar una comparativa en la *Figura 4.2.* Se observan tres arquitecturas diferentes para el modelo VGG. En la arquitectura de la izquierda observamos el modelo VGG original compuesto por 5 bloques (conjunto de Capas Convolucionales y Capas Max Pooling). En la arquitectura central, observamos el modelo descrito en la *Sección 4.3.2* donde se congelan todos los bloques y, por tanto, los pesos de esas capas no se modifican durante el aprendizaje del modelo. Esta parte se conoce como Extractor de Características que precede a la parte de Clasificación formado por las *Capas Full Connected* tal y como se ha descrito en la *Tabla 4.2.* Por último, aparece la arquitectura VGG aplicando la técnica *Fine-Tuning* en donde se han descongelado los bloques 4 y 5 de manera que los pesos de dichas capas se modificarán durante el entrenamiento haciendo que las últimas características sean más específicas para nuestro problema.

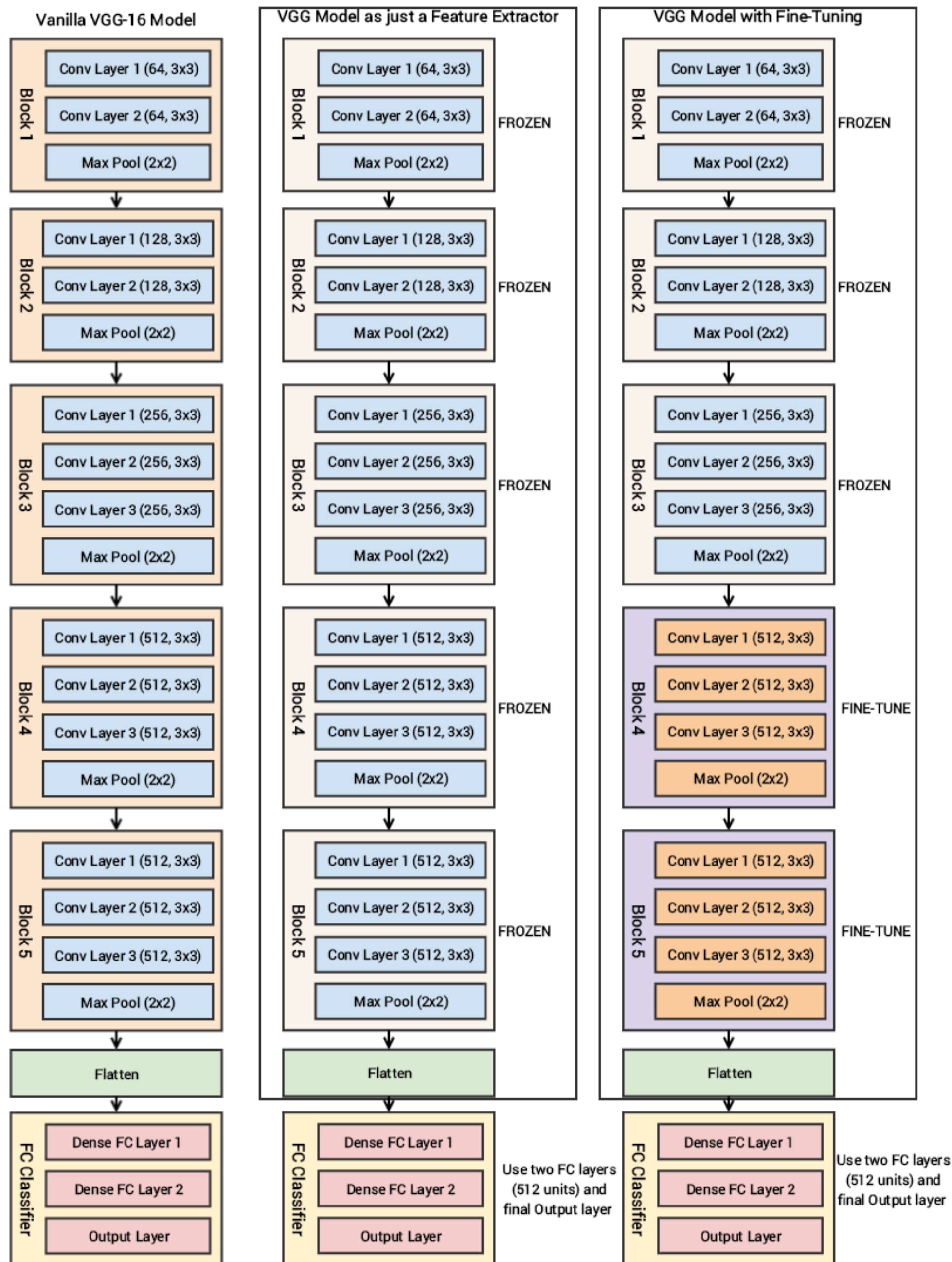

Figura 4.2: Diferentes modificaciones del Modelo VGG16. En towardsdatascience.com [44]

Para este modelo, la última parte de la arquitectura perteneciente a la parte de Clasificación formado por las *Capas Full Connected* se corresponde con la *Tabla 4.2.* En este caso, el número de parámetros a entrenar es de 249172994 que corresponde con el 99% de todos los parámetros que componen el modelo.

4.4 CONCLUSIONES Y DISCUSIÓN DE LOS RESULTADOS

En primer lugar, hay que mencionar que, en un principio, y tal y como se ha explicado antes, se quería obtener un modelo que fuera independiente de la referencia y que fuera capaz de discernir entre todos los defectos descritos en la *Sección 3.1.* Sin embargo, y debido a la poca cantidad de datos disponibles, se ha optado, en primer lugar, por crear modelos separados con el fin de obtener un buen rendimiento en una referencia concreta. Una vez observados los resultados de las referencias por separado, se valorará la creación de un modelo que agrupe a todas las referencias para resolver el problema descrito. Por ello, se va a empezar explicando los resultados obtenidos con la referencia que contenía mayor número de ejemplos (aunque bastante limitada, no obstante): REFERENCIA 1. En concreto, se ha utilizado un 75% de las imágenes disponibles para train, un 10% para validación y un 15% para test siguiendo una técnica de hold-out con estratificación para asegurarnos de que la distribución entre los conjuntos era balanceada y representativa.

- **REFERENCIA 1 (TORTILLA DE PATATA CON CEBOLLA):**

  En primer lugar, se ha creado un modelo con la referencia con mayor número de imágenes disponibles (3255). De estas imágenes, 2442 imágenes (2198 OK y 244 NOK) han sido para train, 325 para validación (291 OK y 34 NOK) y 488 para test (430 OK y 58 NOK). Con las imágenes obtenidas del preprocesamiento (*Sección 4.2)* se han probado los modelos descritos en la *Sección 4.3* y estos han sido los resultados obtenidos.

  - **<u>CNN BÁSICA:</u>** En la *Figura 4.3* se muestra un gráfico que muestra el valor del coste tanto en train como en validación obtenido para 48 épocas.

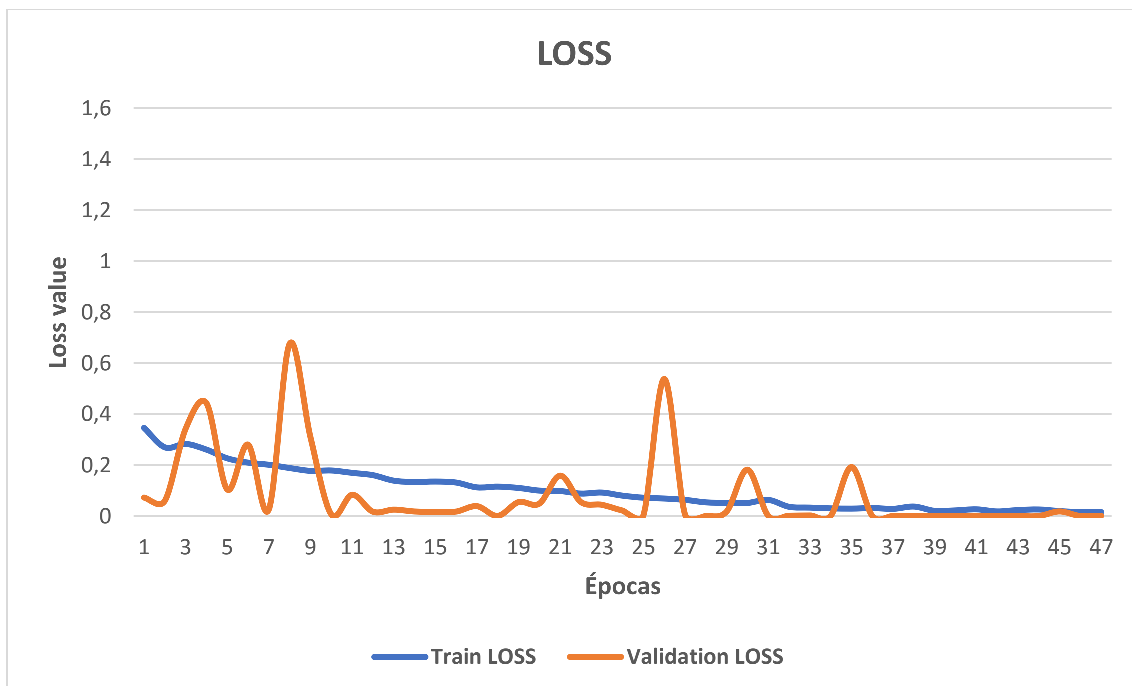

Figura 4.3: Gráfica del coste para CNN Básica

Observamos como el coste tanto en train como en validación se va reduciendo conforme pasan las épocas, indicando que el modelo va aprendiendo y, por tanto, aumentando su tasa de acierto sobre los ejemplos proporcionados. Los saltos en el coste en validación pueden ser debidos a la escasez de datos para validar lo que puede hacer que ciertos ejemplos no muy comunes en train aparezcan en algunos lotes al validar y al modelo le cueste aprender sobre dichos ejemplos. No obstante, el rendimiento que se observa es bastante aceptable.

A continuación, en la *Tabla 4.3* se muestra la matriz de confusión obtenida para los ejemplos de test.

		PREDICCIÓN	
		NOK	**OK**
REAL	**NOK**	61	2
	OK	0	425

Tabla 4.3: Matriz de Confusión en test para CNN Básica

Por último, en la *Tabla 4.4* se muestran los valores de Precision, Recall y F1-score tanto para la clase OK como para la clase NOK. El Accuracy obtenido en test ha sido de 0.996.

	PRECISION	**RECALL**	**F1-SCORE**
NOK	1.00	0.968	0.983
OK	0.995	1.00	0.997

Tabla 4.4: Valores de Precision, Recall y F1-Score en test para CNN Básica

Con todo ello, vemos como el modelo es capaz de acertar prácticamente todos los ejemplos en test tanto para ejemplos OK como NOK a excepción de unos pocos ejemplos NOK que son dados como OK los cuales queremos evitar siempre. Esto puede ser debido a que sean ejemplos que aparezcan en test, pero no en train (o que aparezcan con poca frecuencia) o a que el modelo no es capaz de encontrar la frontera que separe bien a dichos ejemplos.

- **TRANSFER LEARNING (VGG16):** Con este modelo pretendemos mejorar los resultados del modelo anterior (que ya eran muy buenos) y ver cómo se comporta un modelo con Transfer Learning en nuestro problema concreto.
  En la *Figura 4.4* se muestra un gráfico que muestra el valor del coste tanto en train como en validación obtenido para 42 épocas.

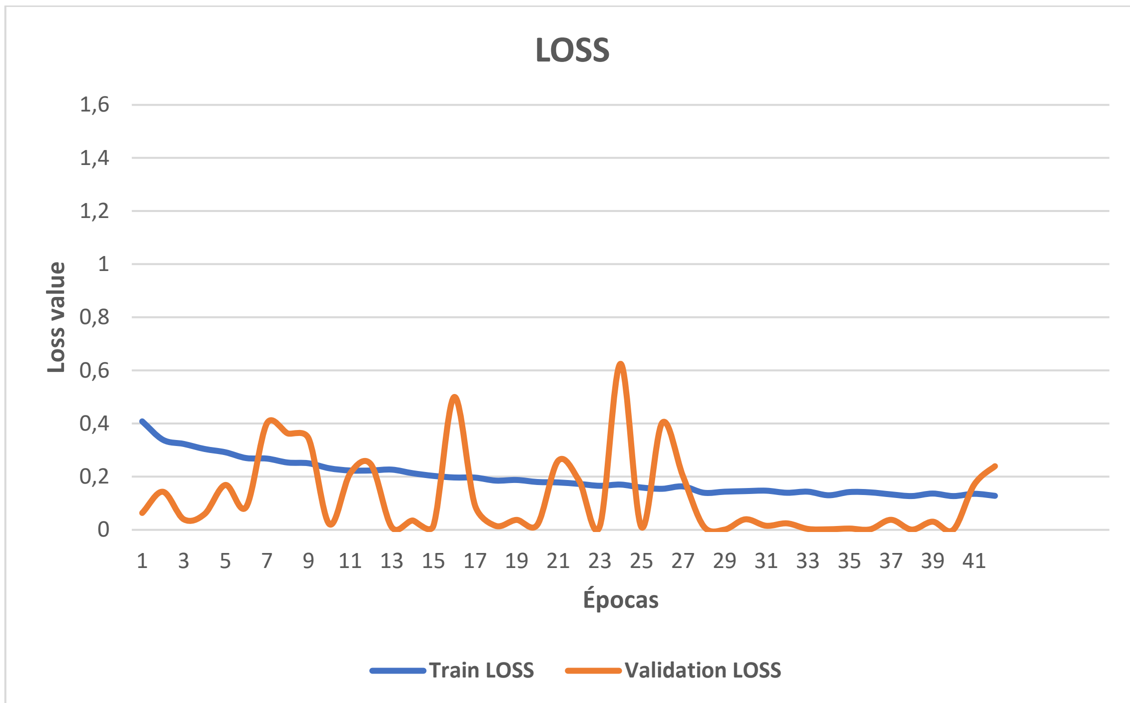

Figura 4.4: Gráfica del coste para VGG16

En este caso, observamos como el coste en train va decreciendo con el paso de las épocas, así como el coste en validación. Sin embargo, podemos observar como la tasa de reducción es pequeña y al modelo le cuesta converger hacia una buena solución. Además, los picos que observamos en validación nos hacen intuir que no está aprendiendo bien las características necesarias para nuestro problema. Cabe destacar que quedándonos con el modelo de la época 35 (menor coste en validación) el resultado probablemente fuera mejor que el obtenido para las 42 épocas.

A continuación, en la *Tabla 4.5* se muestra la matriz de confusión obtenida para los ejemplos de test.

	PREDICCIÓN		
REAL		**NOK**	**OK**
	NOK	43	19
	OK	3	423

Tabla 4.5: Matriz de Confusión en test para VGG16

Por último, en la *Tabla 4.6* se muestran los valores de Precision, Recall y F1-score tanto para la clase OK como para la clase NOK. El Accuracy obtenido en test ha sido de 0.955.

	PRECISION	RECALL	F1-SCORE
NOK	0.934	0.693	0.796
OK	0.957	0.992	0.974

Tabla 4.6: Valores de Precision, Recall y F1-Score en test para VGG16

En base a los resultados anteriores, podemos concluir que este modelo obtiene unos resultados bastante peores que el modelo anterior. Vemos como el valor del *RECALL* en la clase NOK nos indica una tasa de acierto de 0.693 lo que refleja que el modelo no es capaz de discernir bien los ejemplos defectuosos que son los que más nos interesa minimizar. Estos resultados nos dejan entrever que el modelo no se adecua correctamente a nuestro problema. Además, podemos observar como en las últimas épocas el modelo parece sobreentrenar lo que indica que no está generalizando bien con el paso de las iteraciones.

- **TRANSFER LEARNING (VGG16) CON FINE-TUNING:** Vistos los dos modelos anteriores, vamos a probar el rendimiento del modelo VGG16 con Fine-Tuning. A priori, los resultados deberían ser mejores que los modelos anteriores ya que juntamos el poder del Transfer Learning para obtener características relevantes de las imágenes, con la modificación de los pesos de las ultimas capas que hacen adecuarse a nuestro problema concreto.
  En la *Figura 4.5* se muestra un gráfico que muestra el valor del coste tanto en train como en validación obtenido para 43 épocas.

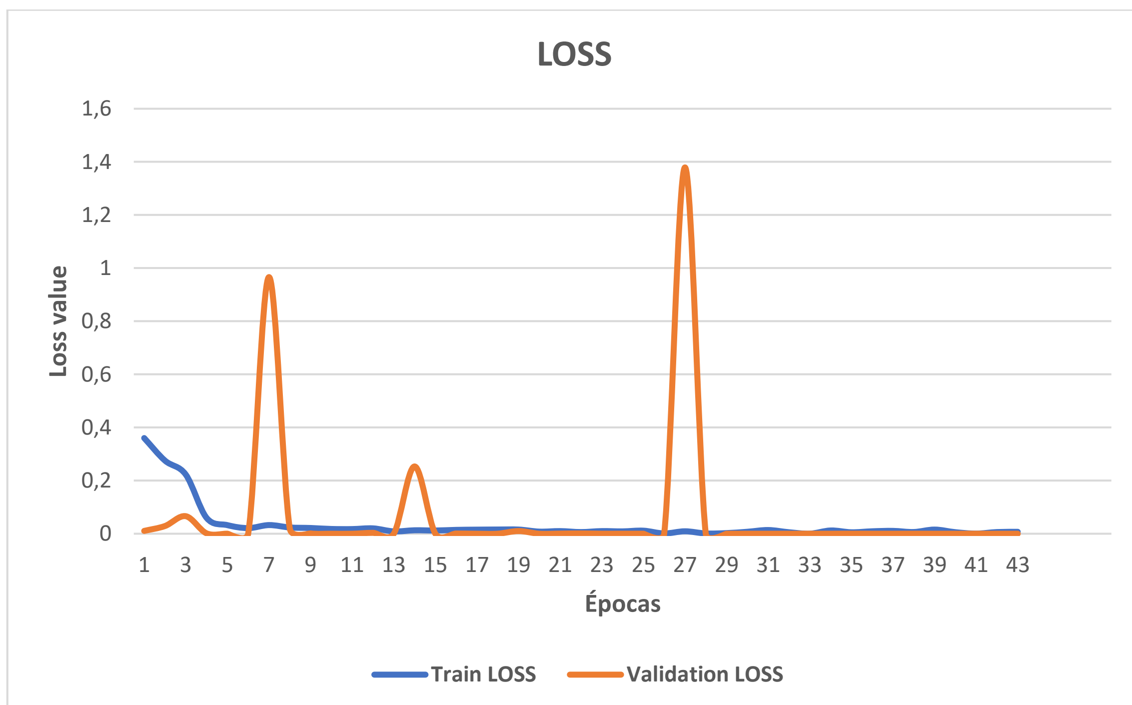

Figura 4.5: Gráfica del coste para VGG16 con Fine-Tuning

Vemos como el coste tanto en train como en validación se reduce prácticamente a cero desde las primeras épocas. Lo único que resalta son los picos en validación ocasionado, probablemente otra vez, por ciertos ejemplos que solo aparezcan en validación y no en train (o que aparezcan con poca frecuencia). Independientemente de ello, vemos como los resultados son bastante prometedores.

En la *Tabla 4.7* se muestra la matriz de confusión obtenida para los ejemplos de test.

	PREDICCIÓN		
REAL		**NOK**	**OK**
	NOK	62	1
	OK	0	425

Tabla 4.7: Matriz de Confusión en test para VGG16 con Fine-Tuning

En la *Tabla 4.8* se muestran los valores de Precision, Recall y F1-score tanto para la clase OK como para la clase NOK. El Accuracy obtenido en test ha sido de 0.998.

	PRECISION	**RECALL**	**F1-SCORE**
NOK	1.00	0.984	0.991
OK	0.997	1.00	0.998

Tabla 4.8: Valores de Precision, Recall y F1-Score en test para VGG16 con Fine-Tuning

En este modelo podemos ver como los resultados son casi perfectos a excepción de un ejemplo NOK que ha sido dado como OK. Por tanto, podemos concluir que este modelo que permite el ajuste de ciertos pesos para nuestro problema concreto, junto con la arquitectura previa VGG16, es el mejor de los tres probados para esta referencia.

Vistos los resultados tan positivos obtenidos para la referencia 1 se optó por afrontar directamente el problema inicial: crear un modelo capaz de clasificar correctamente las tortillas de cualquier referencia. Sabiendo que la cantidad de imágenes de algunas referencias es muy limitada, es probable que el resultado no sea tan bueno como un modelo especializado en una referencia concreta. A continuación, se mostrarán los resultados obtenidos para el modelo que engloba a todas las referencias.

- **MODELO GLOBAL:**

  De la misma manera que antes, se realizó el preprocesamiento de todas las imágenes de todas las referencias (7559 imágenes) y se han probado los tres modelos descritos anteriormente con la siguiente distribución: 5668 imágenes para train (5055 OK y 613 NOK), 756 imágenes para validación (674 OK y 82 NOK) y 1135 imágenes para test (1012 OK y 123 NOK).

➢ **<u>CNN BÁSICA:</u>** En la *Figura 4.6* se muestra un gráfico que muestra el valor del coste tanto en train como en validación obtenido para 25 épocas.

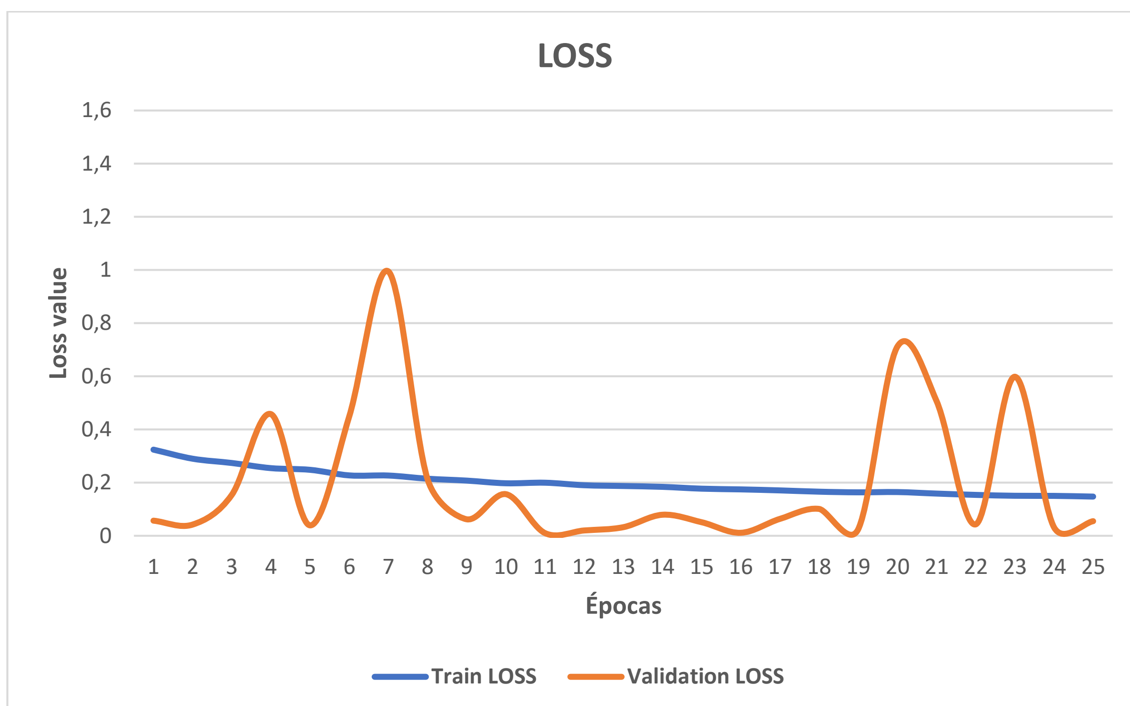

Figura 4.6: Gráfica del coste para CNN Básica en el modelo global

Podemos observar cómo al modelo le cuesta converger tanto en train como en validación. Es cierto que el número de épocas es bastante reducido debido a la gran cantidad de imágenes que se evalúan por época y que hace que el entrenamiento se demore considerablemente. No obstante, los valores del coste son bastantes parecidos a los obtenidos con la misma arquitectura para el modelo de la referencia 1 lo cual es bastante satisfactorio.

A continuación, en la *Tabla 4.9* se muestra la matriz de confusión obtenida para los ejemplos de test.

	PREDICCIÓN		
REAL		**NOK**	**OK**
	NOK	75	48
	OK	10	1002

Tabla 4.9: Matriz de Confusión en test para CNN Básica en el modelo global

Por último, en la *Tabla 4.10* se muestran los valores de Precision, Recall y F1-score tanto para la clase OK como para la clase NOK. El Accuracy obtenido en test ha sido de 0.948.

	PRECISION	RECALL	F1-SCORE
NOK	0.882	0.609	0.720
OK	0.954	0.990	0.971

Tabla 4.10: Valores de Precision, Recall y F1-Score en test para CNN Básica en el modelo global

Vemos como el modelo no obtiene malos resultados considerando el contexto del problema, aunque hay que destacar que, tal y como se observaba en la *Figura 4.6,* el algoritmo parece no generalizar del todo bien y, por tanto, la solución a encontrar dista mucho de ser una buena solución. Se observa, además, como el RECALL es bajo tal y como ocurría en el modelo de la referencia 1, valor que nos interesa minimizar con el fin de evitar falsos negativos.

- **TRANSFER LEARNING (VGG16):** Vamos a probar los modelos con Transfer Learning para intentar mejorar el modelo presentado en el punto anterior. Empezaremos, al igual que para la referencia 1, describiendo los resultados obtenidos para el modelo con la arquitectura base VGG16.
  En la *Figura 4.7* se muestra un gráfico que muestra el valor del coste tanto en train como en validación obtenido para 21 épocas.

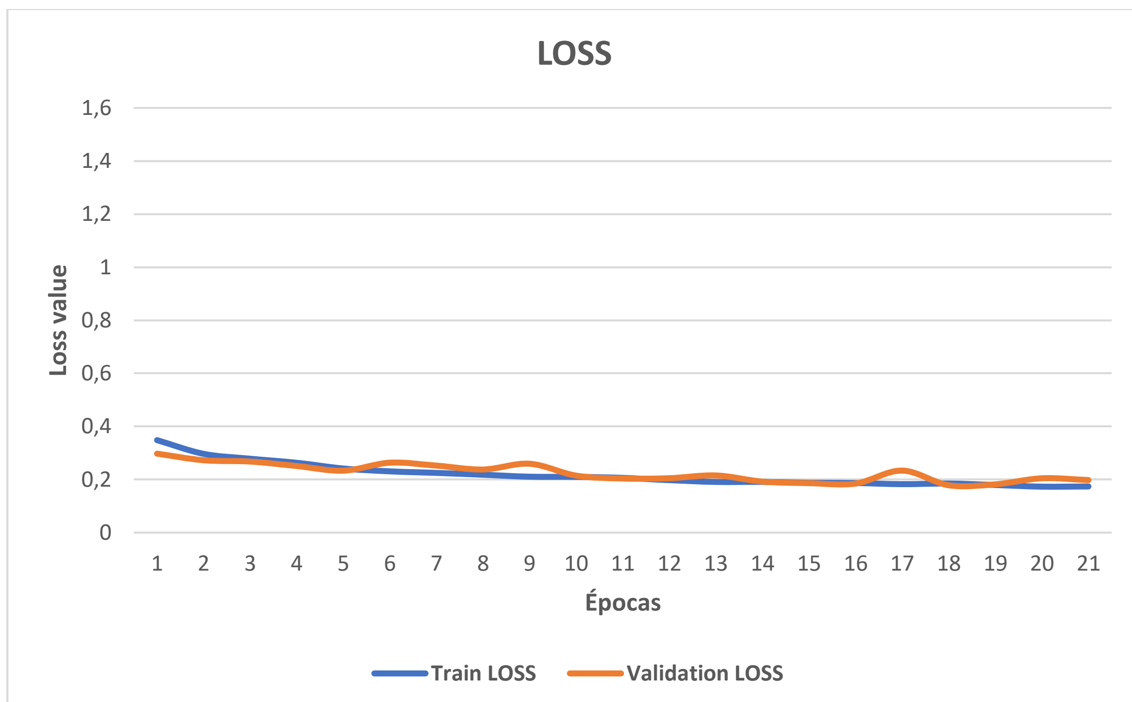

Figura 4.7: Gráfica del coste para VGG16 en el modelo global

En este caso observamos como el coste en train va decreciendo con el paso de las épocas, así como el coste en validación. Vemos como desde el principio el coste es bastante bajo, pero parece que el modelo está bastante estancado indicando que no es capaz de aprender las características concretas de nuestro problema.

A continuación, en la *Tabla 4.11* se muestra la matriz de confusión obtenida para los ejemplos de test.

	PREDICCIÓN		
REAL		**NOK**	**OK**
	NOK	47	76
	OK	0	1012

Tabla 4.11: Matriz de Confusión en test para VGG16 con el modelo global

Por último, en la *Tabla 4.12* se muestran los valores de Precision, Recall y F1-score tanto para la clase OK como para la clase NOK. El Accuracy obtenido en test ha sido de 0.933.

	PRECISION	**RECALL**	**F1-SCORE**
NOK	1.00	0.382	0.552
OK	0.930	1.00	0.963

Tabla 4.12: Valores de Precision, Recall y F1-Score en test para VGG16 en el modelo global

En vista de estos resultados, podemos concluir que el modelo con VGG16 no es capaz de obtener unos buenos resultados para las imágenes del conjunto de test. Especialmente relevante es el 0.382 obtenido en RECALL para la clase NOK que es en la que nos interesa obtener buenos resultados. Al igual que en el modelo para la referencia 1, la arquitectura VGG16 sin modificar no es capaz, en un número pequeño de épocas, de converger hacia la solución, y, por tanto, de no ser capaz de maximizar los resultados.

- **TRANSFER LEARNING (VGG16) CON FINE-TUNING:** Por último, vamos a probar con el último modelo con el fin de obtener, al igual que antes, un rendimiento mejor que con los modelos previos.
  En la *Figura 4.8* se muestra un gráfico que muestra el valor del coste tanto en train como en validación obtenido para 19 épocas.

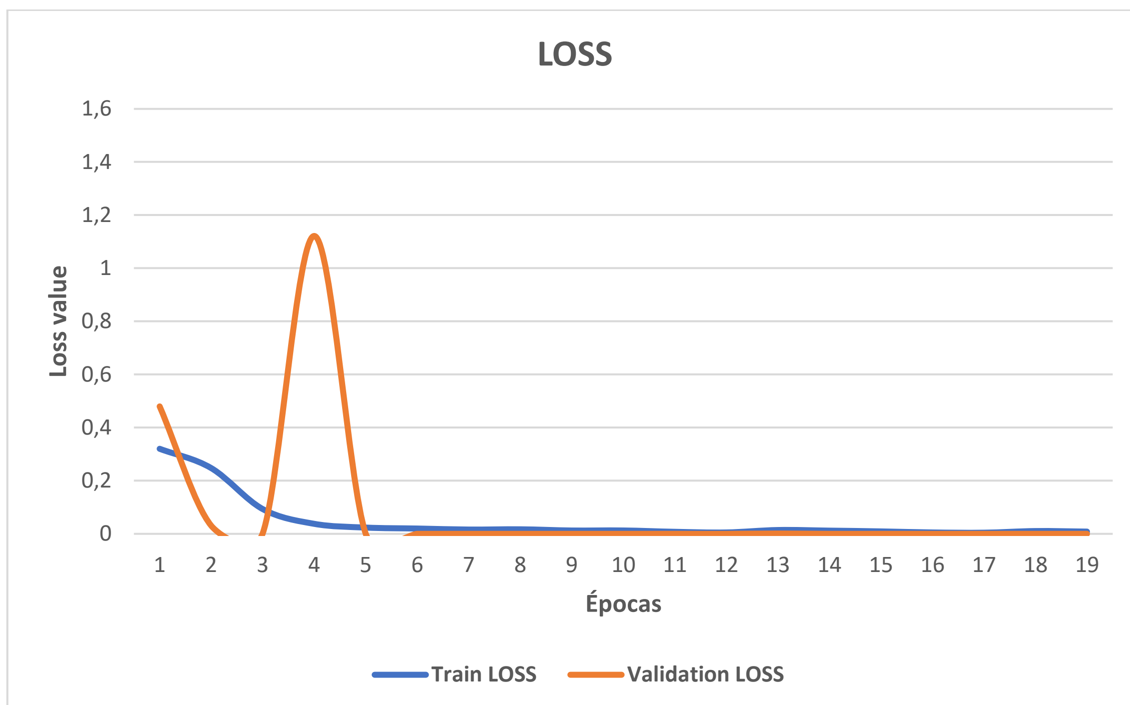

Figura 4.8: Gráfica del coste para VGG16 con Fine-Tuning en el modelo global

En este caso, vemos como el modelo tras unas pocas épocas es capaz de ajustarse de manera precisa a los ejemplos tanto de train como de validación y por tanto el coste que se obtiene tiende a cero prácticamente.

En la *Tabla 4.13* se muestra la matriz de confusión obtenida para los ejemplos de test.

	PREDICCIÓN		
REAL		**NOK**	**OK**
	NOK	120	3
	OK	0	1012

Tabla 4.13: Matriz de Confusión en test para VGG16 con Fine-Tuning en el modelo global

En la *Tabla 4.14* se muestran los valores de Precision, Recall y F1-score tanto para la clase OK como para la clase NOK. El Accuracy obtenido en test ha sido de 0.997.

	PRECISION	**RECALL**	**F1-SCORE**
NOK	1.00	0.975	0.987
OK	0.997	1.00	0.998

Tabla 4.14: Valores de Precision, Recall y F1-Score en test para VGG16 con Fine-Tuning en el modelo global

Podemos ver como este modelo, al igual que con la misma arquitectura que para la referencia 1, obtiene unos resultados prácticamente perfectos lo que nos indica que esta configuración ha sido la mejor para nuestro problema.

Para poder visualizar y entender mejor que ejemplos se están fallando, en la *Tabla 4.15* se muestran los ejemplos fallados por el modelo con mejor rendimiento: VGG16 con Fine-Tuning. Los siguientes ejemplos corresponden a la clase NOK y han sido etiquetados como OK.

Tabla 4.15: Imágenes falladas por VGG16 con Fine-Tuning para los ejemplos de test en el modelo global

Podemos observar cómo estos ejemplos son casos poco frecuentes o excepcionales dentro del conjunto de entrenamiento. La imagen superior izquierda corresponde a un ejemplo con doble film superpuesto y que no aparece en train, la imagen superior derecha es uno de los defectos presentados en la *Sección 3.1* y que corresponde a la ausencia de producto, aunque es cierto que su presencia es muy minoritaria y la imagen central pertenece a una etiqueta con fecha de caducidad/lote, pero en la que la calidad de la impresión es muy baja y prácticamente no se puede apreciar nada. Es importante recalcar que el modelo sí que parece que ha detectado bien la presencia o no de la fecha de caducidad/lote ya que es probable que estos ejemplos NOK hayan sido predichos como OK por la presencia de la banda negra que corresponde al lugar donde debe ir la fecha de caducidad/lote.

5. DESPLIEGUE DEL SISTEMA

A continuación, se pasará a explicar cuál es la configuración necesaria para montar el sistema (*Sección 5.1)* así como el funcionamiento de la aplicación *(Sección 5.2).*

5.1 CONFIGURACIÓN NECESARIA

La configuración elegida para el proyecto es importante debido a que el sistema no funciona con cualquier configuración ni con cualquier versión de los programas. Es clave tener esto en cuenta si queremos replicar el sistema y que todo funcione correctamente.

5.1.1 VISUAL STUDIO COMMUNITY

Es importante tener instalado previamente Visual Studio Community ya que tanto QTCreator, que hace uso de los compiladores MSVC, como CUDA, que hace uso de la integración de Visual, requieren de ello para su funcionamiento. En nuestro caso hemos instalado Visual Studio Community 2017 junto a todos sus componentes para C++.

5.1.2 QTCREATOR

La versión elegida de QTCreator es la 5.13.2. En el momento de inicio del proyecto, ésta era la última versión disponible. A día de hoy se ha lanzado la 5.14.0 y se ha llegado a cargar el proyecto con esta nueva versión, pero ha dado errores de incompatibilidad.

En cuanto al Kit utilizado se ha optado por *Desktop Qt 5.13.2 MSVC 2017 64 bit* que incluye el compilador de MSVC para nuestra plataforma de 64 bits.

5.1.3 PYLON

Para hacer uso de la cámara Basler como de sus librerías de desarrollo es imprescindible tener instalado Pylon 6.0. Es necesario instalar los drivers para GigE y en modo Developer para poder acceder a las funcionalidades de desarrollo.

5.1.4 PYTHON

Para este proyecto, se ha optado por utilizar Python 3.7 para librar errores de compatibilidad con las librerías Tensorflow y Keras. Se llegó a probar la versión 3.8 pero se descartó por ser demasiado reciente.

En un principio, se pensó en introducir la librería OPENCV para el tratamiento de las imágenes en conjunción con Python. Sin embargo, esta librería solo admite la compilación con MINGW y Python lo hace con el compilador MSVC. Por tanto, se descartó la opción de utilizar OPENCV para no desviarnos del objetivo del proyecto.

5.1.5 CUDA Y CUDNN

Junto a todo lo anterior, necesitamos toda la configuración relacionada con la tarjeta gráfica para hacer correr el programa en GPU. En nuestro caso hemos instalado CUDA 10.2 junto con CUDNN 7.6.5. Importante no tener otros drivers de Nvidia instalados para evitar errores en la instalación.

5.1.6 VARIABLES DE ENTORNO

No hay que olvidar que debemos añadir todas las variables de entorno necesarias: Python, Pylon, QTCreator y Nvidia GPU.

5.1.7 LIBRERÍAS PYTHON

En este punto es necesario tener instalado las siguientes librerías de Python: *numpy (1.19.1), matplotlib (3.2.2), pillow (7.2.0), pandas (1.0.5) y scikit-learn (0.23.1).*

Además, debemos instalar tensorflow en su versión 2.2.0 y posteriormente Keras (2.4.3).

5.2 FUNCIONAMIENTO DE LA APLICACIÓN

El objetivo de la aplicación creada se puede resumir en los siguientes puntos tal y como se ha explicado en la *Sección 3.3*:

- *Conexión y configuración de la cámara:* Modificar los parámetros de la cámara para obtener una imagen de calidad.
- *Creación de una base de datos de imágenes:* Clasificar imágenes con el fin de obtener el dataset que se utilizará para entrenar y validar un modelo.
- *Aplicación de un modelo creado sobre una imagen:* Para cada imagen obtenida se aplica un modelo creado clasificando cada imagen como correcta o incorrecta.

A continuación, se desgranarán los puntos anteriores junto con ejemplos visuales de su funcionamiento. La interfaz principal se puede visualizar en la *Figura 5.1.*

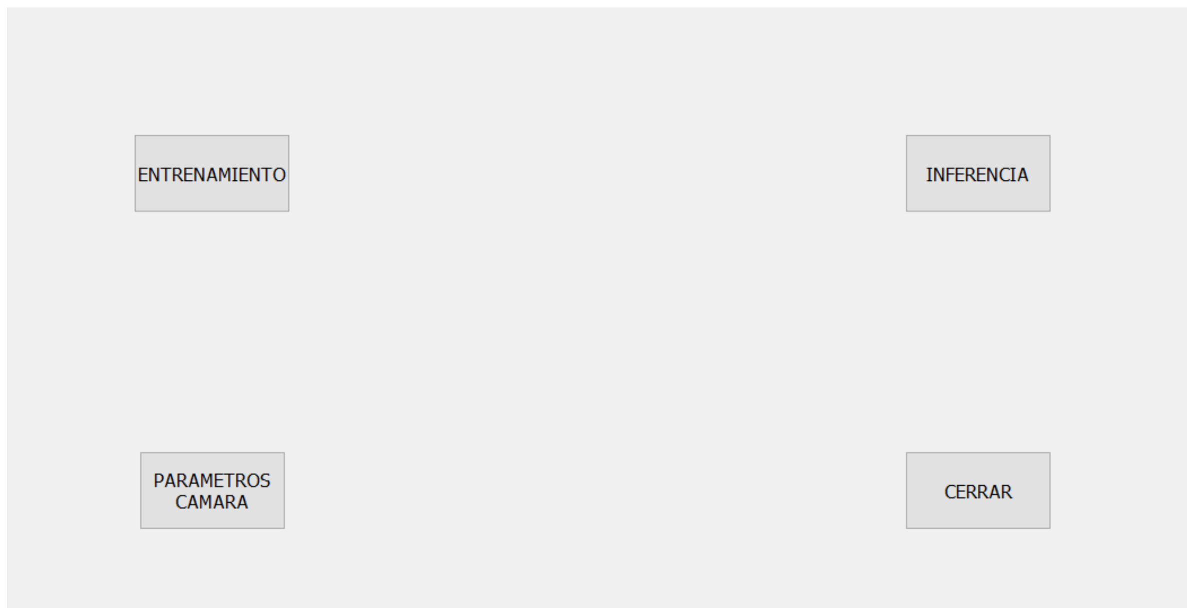

Figura 5.1: Interfaz principal de la aplicación

5.2.1 CONEXIÓN Y CONFIGURACIÓN DE LA CÁMARA

La primera de las funciones de la aplicación es la conexión con la cámara Basler y la configuración de la imagen. Este apartado es clave debido a que es necesario tener una buena imagen para poder trabajar posteriormente en ella. Recordemos que si no tenemos datos de calidad es difícil obtener resultados aceptables.

Una imagen podemos considerarla de calidad si posee un nivel de brillo mínimo, un buen nivel de contraste, un nivel de blancos equilibrado, etc.

Dentro de la propia aplicación si clicamos sobre *"PARÁMETROS CÁMARA"* nos aparecerá la *Figura 5.2.*

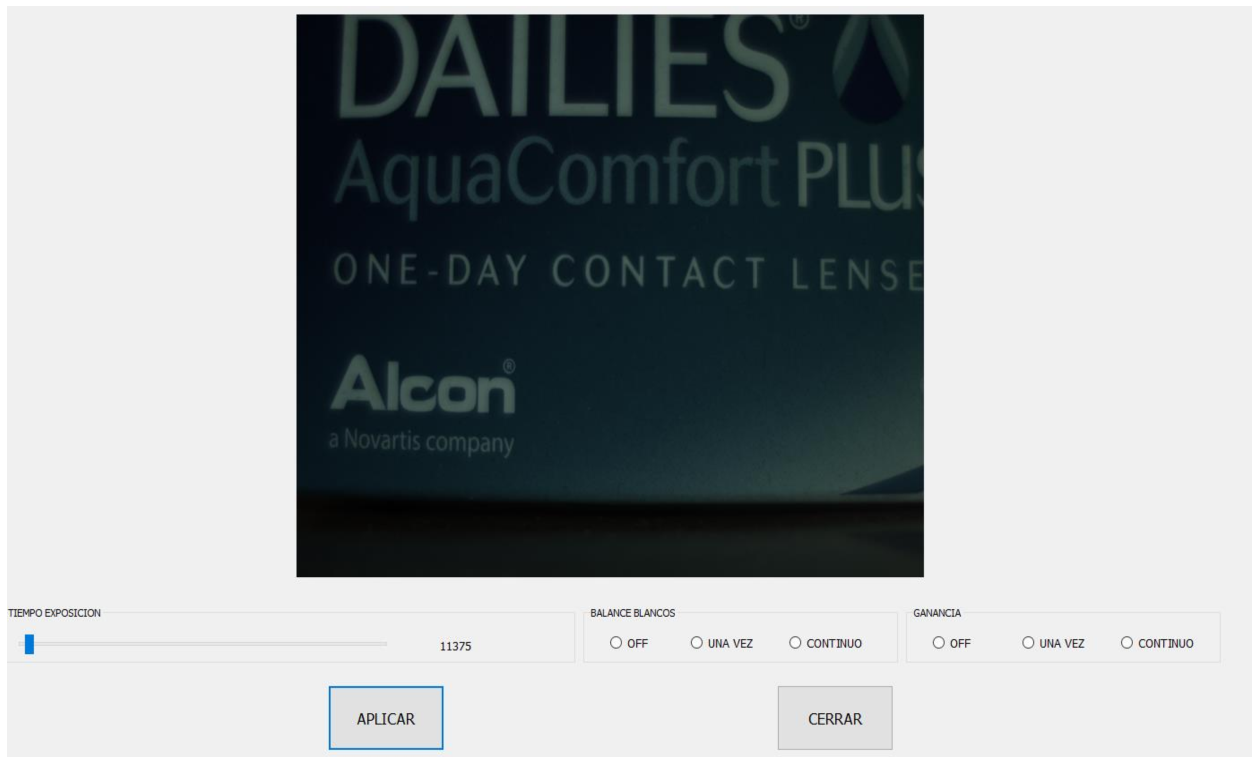

Figura 5.2: Ventana de "PARÁMETROS CÁMARA"

En este caso podemos ver tres de los parámetros que más repercusión tienen a la hora de obtener una buena imagen: Tiempo de Exposición, Balance de Blancos y Ganancia.

- *Tiempo de Exposición:* Es el tiempo que permanece abierto el obturador de la cámara dejando pasar la luz hasta el sensor. Normalmente se mide en fracciones de segundo o en microsegundos. A mayor Tiempo de Exposición mayor luz entra en el sensor y por tanto más clara nos sale la imagen. Por el contrario, esto puede hace que, si el objeto que queremos capturar está en movimiento, la imagen salga borrosa al tener el obturador demasiado tiempo abierto. Por tanto, es clave el buscar un buen Tiempo de Exposición que dependerá de cada aplicación en concreto.
- *Balance de Blancos:* Se trata de un ajuste de software que intenta mostrar la imagen sin dominancias de ningún color. Básicamente trata de compensar el color dominante en la imagen y le añade una tonalidad más fría o más cálida según convenga. No siempre es necesario su uso, ya que, dependiendo del resultado que queramos obtener puede ser mejor no utilizarlo o hacerlo solo en determinadas condiciones.
- *Ganancia:* Es el factor que convierte el número de fotoelectrones recogidos al capturar la imagen en el valor numérico que toma cada píxel y que por tanto determina su nivel de brillo. Amplifica digitalmente la señal pudiendo ganar más o menos luz, pero a costa de perder calidad (aumento del ruido). Hay que utilizarlo con precaución, pero puede ser una buena herramienta en algunos casos.

Cuando entramos por primera vez en este modo, obtenemos una imagen con los parámetros que tenía la cámara por defecto. Para modificar el Tiempo de Exposición tenemos que arrastrar la barra disponible y nos irá actualizando el valor. En cuanto al Balance de Blancos podemos elegir tres configuraciones: OFF (no se aplica ninguna corrección), UNA VEZ (se aplica la corrección hasta que alcanza un nivel estable de blancos) o CONTINUO (se aplica la corrección del balance de blancos en cada imagen que se captura). Del mismo modo, para la Ganancia se pueden aplicar los mismos modos (OFF, UNA VEZ o CONTINUO). De esta forma, elegimos cuando se aplica una corrección sobre la Ganancia.

Para la imagen que estamos visualizando vemos como está bastante oscura. Modificando el Tiempo de Exposición obtenemos el resultado de la *Figura 5.3.*

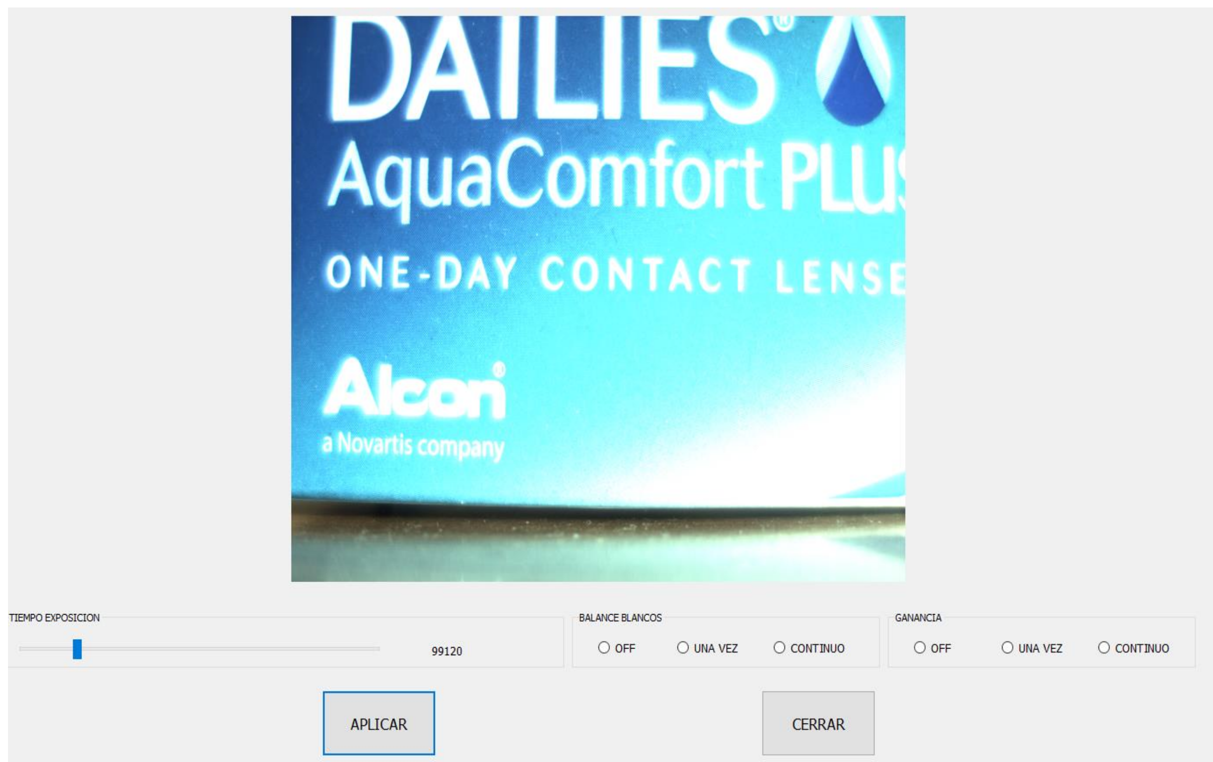

Figura 5.3: Ventana de "PARÁMETROS CÁMARA" modificado el Tiempo de Exposición

Vemos como ahora el Tiempo de Exposición es excesivo y obtenemos una imagen sobreexpuesta. Este es uno de los peores escenarios debido a que una imagen de este tipo es muy difícil de corregir a posteriori debido a que los píxeles quemados ya no se pueden recuperar. Con una imagen subexpuesta y mediante técnicas de postprocesado es posible recuperar más información. Reduciendo un poco el Tiempo de Exposición llegamos a un resultado más óptimo (*Figura 5.4).*

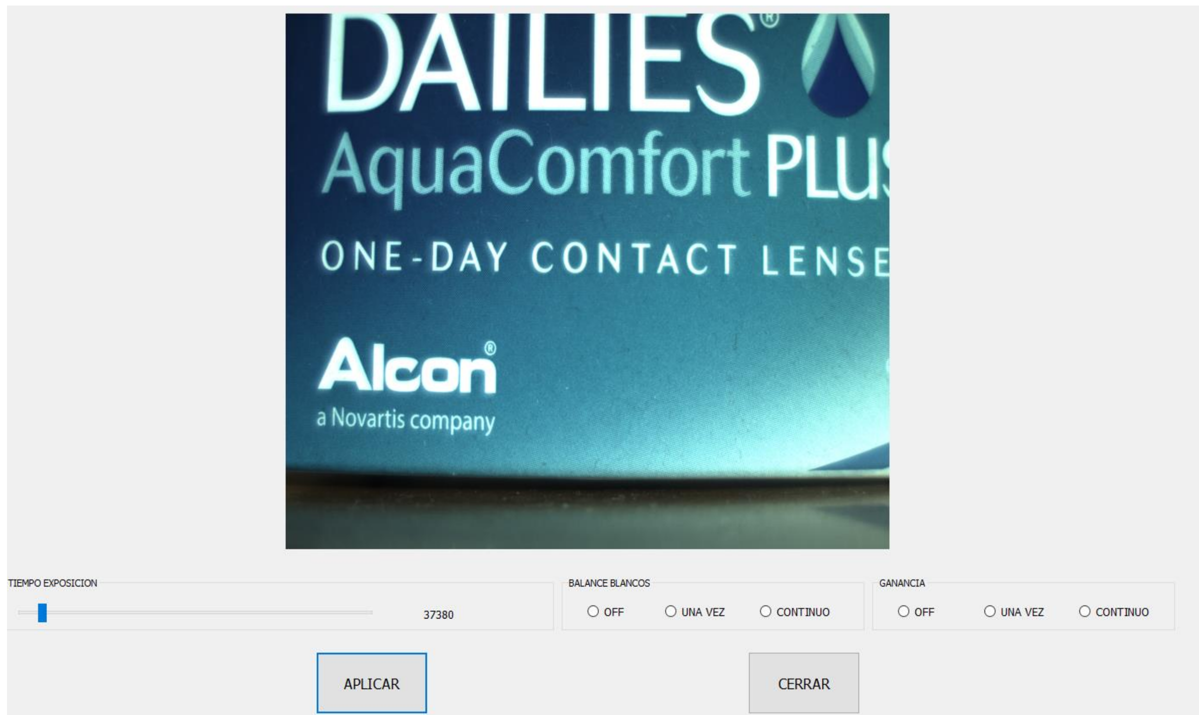

Figura 5.4: Ventana de "PARÁMETROS CÁMARA" con Tiempo de Exposición correcto

Tras esta corrección, pasaríamos a aplicar una corrección de blancos para ver si podemos mejorar la imagen. Elegimos el modo "UNA VEZ" y observamos el resultado de la *Figura 5.5.*

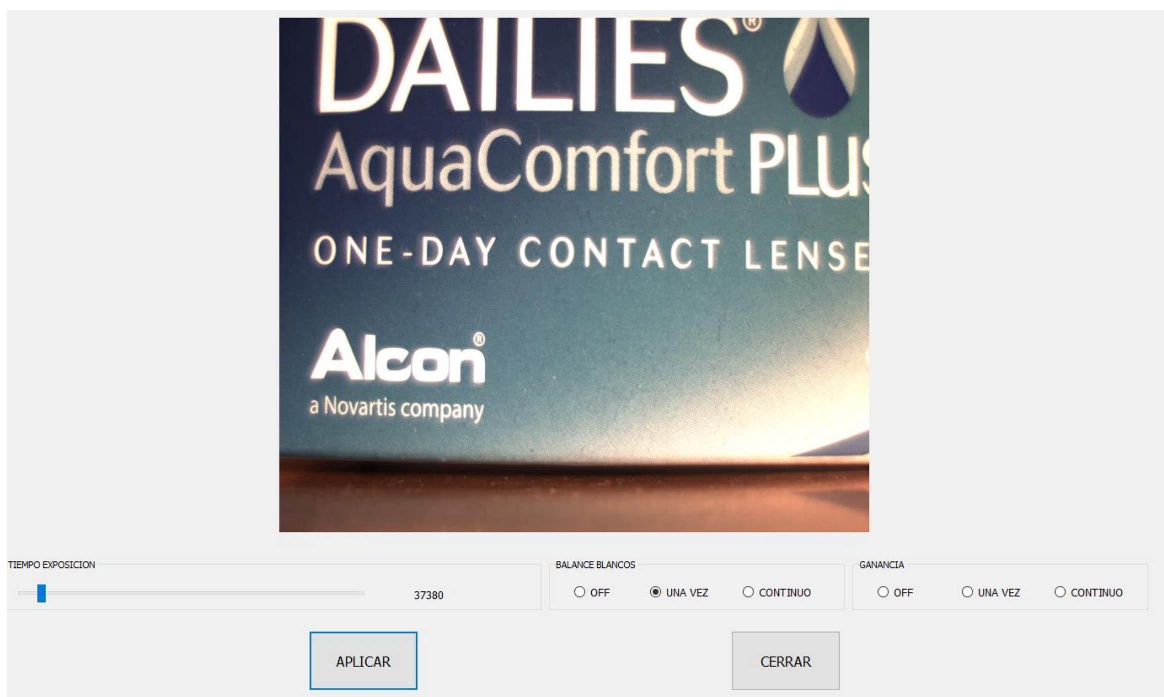

Figura 5.5: Ventana de "PARÁMETROS CÁMARA" con Balance Blancos activado

Podemos ver como la imagen ha pasado a mostrar unos tonos más cálidos que dependiendo de la aplicación que vayamos a darle nos puede venir bien o, por el contrario, nos desfavorece a la hora de encontrar ciertos detalles. Como mencionamos anteriormente, cada aplicación requiere un tratamiento especializado.

Ahora veremos cómo influye una corrección en la Ganancia. Seleccionamos “UNA VEZ” y observamos el resultado de la *Figura 5.6.*

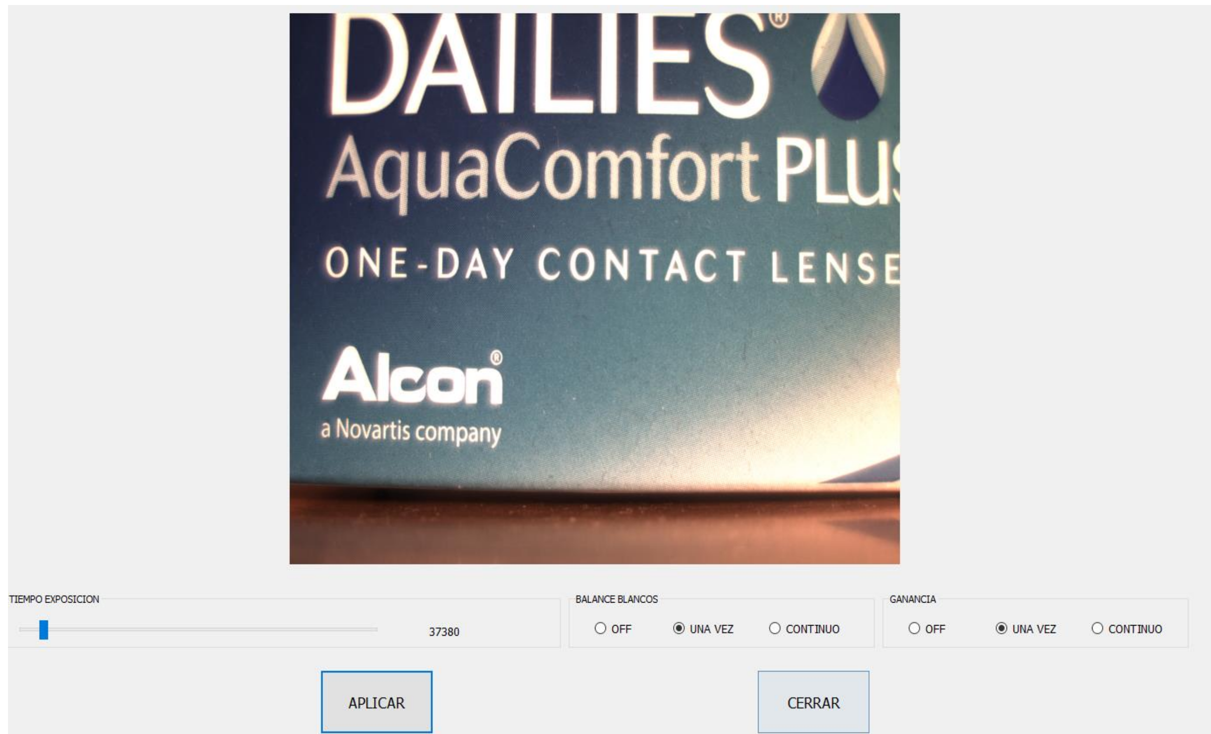

Figura 5.6: Ventana de “PARÁMETROS CÁMARA” con Ganancia activada

En este caso, vemos como no hay mucho cambio debido a que la imagen está bastante bien configurada. Apreciamos un ligero descenso del brillo en la parte inferior derecha (donde más refleja la luz) pero nada muy relevante.

En este punto, podríamos seguir realizando pruebas para ajustar aún más la imagen o si, por el contrario, estamos bastante convencidos del resultado, cerraríamos la ventana y ya podríamos proceder a los apartados de ENTRENAMIENTO o INFERENCIA.

5.2.2 CREACIÓN DE UNA BASE DE IMÁGENES PARA ENTRENAMIENTO

Una vez que hemos configurado correctamente la imagen para poder utilizarla, es imprescindible el disponer de una buena base de ellas para posteriormente aplicar algoritmos sobre las mismas. Uno de los requisitos del Deep Learning es tener una cantidad suficiente de imágenes de diferente índole para que sea capaz de aprender un modelo que generalice bien.

Para ello, si clicamos sobre "ENTRENAMIENTO" en la pantalla principal de la aplicación nos llevará a la *Figura 5.7.*

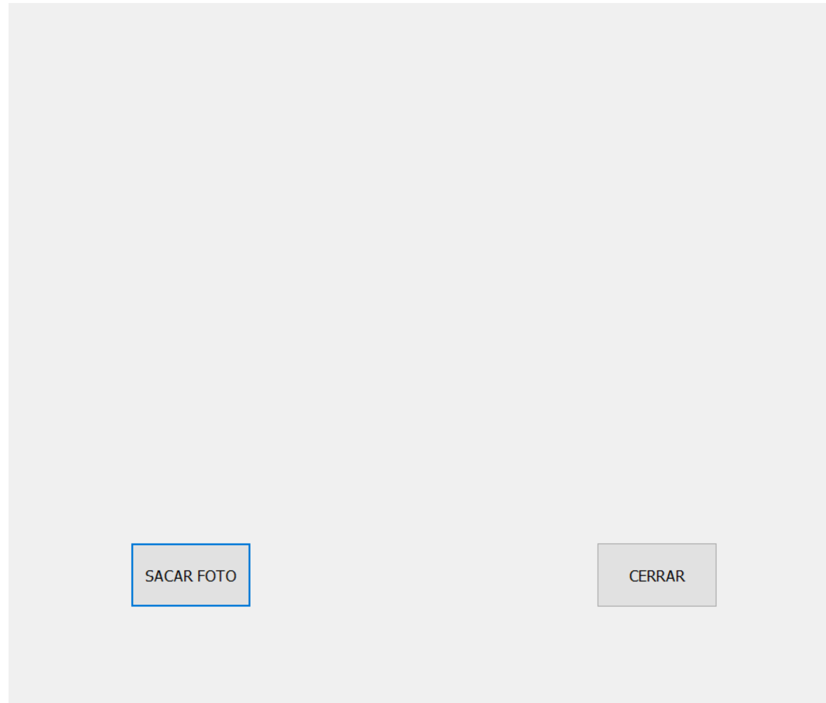

Figura 5.7: Ventana de "ENTRENAMIENTO"

En este punto existe la posibilidad de sacar una foto o de volver a la ventana principal. Cuando clicamos sobre "SACAR FOTO", nos aparecerá la ventana representada en la *Figura 5.8.*

Figura 5.8: Ventana de “ENTRENAMIENTO” una vez capturada la imagen

En este punto podemos realizar tres acciones, clasificar esta imagen como OK, como NOK o repetir la imagen. Si elegimos tanto OK como NOK la imagen se guardará en sus respectivas carpetas y volveremos a la pantalla anterior (*Figura 5.7*). Si elegimos repetir (debido a que la imagen no ha salido bien, hemos clicado sin querer en el botón “SACAR FOTO”, etc.), obtendremos la misma ventana, pero con la nueva imagen adquirida.

5.2.3 EJECUCIÓN DEL MODELO CREADO SOBRE LA IMAGEN

La última funcionalidad de la aplicación, y la más interesante, consiste en aplicar un modelo creado con las imágenes del apartado anterior y aplicarlo sobre una imagen capturada por la aplicación para así poder discernir si es OK o NOK (no olvidemos que estamos considerando un problema de clasificación).

Centrándonos en la aplicación, cuando clicamos sobre “INFERENCIA” en la ventana principal nos aparecerá la misma ventana de la *Figura 5.7.*

Cuando clicamos sobre “SACAR FOTO” obtenemos el resultado de la *Figura 5.9.*

APLICAR REPETIR

Figura 5.9: Ventana de "INFERENCIA" una vez capturada la imagen

En este punto ocurre igual que en "ENTRENAMIENTO" en donde podíamos continuar con el proceso o repetir la imagen si ésta no ha salido como quisiéramos.

Cuando procedemos a aplicar el modelo sobre la imagen podemos obtener los dos siguientes resultados: IMAGEN CORRECTA (*Figura 5.10)* o IMAGEN INCORRECTA (*Figura 5.11)*.

IMAGEN CORRECTA

CERRAR

Figura 5.10: Ventana de "INFERENCIA" una vez aplicado el modelo sobre una imagen correcta

IMAGEN INCORRECTA

CERRAR

Figura 5.11: Ventana de "INFERENCIA" una vez aplicado el modelo sobre una imagen incorrecta

Cabe destacar que en un funcionamiento del sistema en producción este proceso sería automático, pero se ha realizado de esta forma para poder explicar su funcionamiento paso a paso.

6. CONCLUSIONES Y LÍNEAS FUTURAS

Tras haber realizado una introducción al Machine Learning y Deep Learning, con especial hincapié en las Redes Neuronales Convolucionales, y haber presentado la problemática a resolver por parte del cliente de tortillas, hemos desgranado el sistema embebido capaz de resolver la necesidad del mismo.

Para ello, hemos descrito tanto los componentes hardware como software, así como la integración de las diferentes librerías y lenguajes de programación necesarios. Además, hemos probado y evaluado diferentes arquitecturas CNN para conseguir un modelo en Python capaz de clasificar correctamente las imágenes obtenidas por el sistema.

Por último, hemos visualizado el funcionamiento de la aplicación que engloba todo el sistema.

Tras haber montado el sistema embebido y haber sido capaces de realizar la integración de todos los componentes, podemos llegar a ciertas conclusiones generales.

En primer lugar, hay que comentar que la problemática a resolver era, de antemano, bastante complicada. La poca cantidad de imágenes disponibles junto con la sutileza y disparidad de defectos a encontrar hacía que, de antemano, se presupusiera complicado encontrar un modelo que funcionara bien ante la situación descrita.

Además, la integración de todos los elementos correspondientes tanto software como hardware dentro de una misma aplicación embebida predispone a encontrar muchos problemas de compatibilidad entre programas y librerías, tal y como ha ocurrido finalmente en el proyecto.

No obstante, los resultados finales son bastante satisfactorios ya que se ha hecho funcionar la aplicación con todos los elementos descritos en el proyecto y consiguiendo un modelo bastante fiable para todo el conjunto de referencias disponibles.

Más concretamente, el modelo Global que abarca a todas las referencias y cuya arquitectura VGG16 con Fine-Tuning ha dado los mejores resultados arroja una fiabilidad que nos puede dar una idea de los rendimientos que se pueden llegar a obtener en la industria con técnicas de Deep Learning. No hay que olvidar, por otro lado, que ha sido necesario un preprocesamiento de las imágenes para poder trabajar con ellas y que han sido clave en los resultados obtenidos. Además, si comparamos los resultados obtenidos por dicha arquitectura respecto a las otras también analizadas en esta memoria vemos como, para este problema en concreto, el ajuste de los pesos entre capas resulta relevante para obtener un buen rendimiento. Esto puede ser debido a que los defectos mostrados en las imágenes no siguen unas características globales y es, por ello, necesario ajustar el modelo a nuestras características particulares.

En relación con lo anterior, se ha podido observar como el número de épocas para el entrenamiento de cada modelo es reducido debido a la gran dimensionalidad de las imágenes y que hace que el tiempo empleado para realizar una iteración se demore mucho en el tiempo lo que repercute tanto en los rendimientos obtenidos (sobre todo cuando no hay una convergencia rápida como en las arquitecturas VGG16) y en la velocidad para validar un modelo con los datos del conjunto de test.

En cuanto a futuras mejoras sobre el sistema actual podríamos destacar, en primer lugar, que debido a un conjunto de datos tan diverso y desbalanceado la aplicación de técnicas tanto de *oversamping* como *undersampling* u otras técnicas similares probablemente hicieran mejorar los resultados obtenidos.

Por otro lado, el uso de otro tipo de arquitecturas CNN tales como *Inception* o *ResNet* podrían ser de utilidad a la hora de intentar mejorar los resultados vistos hasta ahora.

Además, y no menos importante, disponer de más cantidad de imágenes especialmente de aquellas pertenecientes a la clase NOK facilitaría mucho el aprendizaje. Unido a ello, la disponibilidad de hardware especializado para el entrenamiento de modelos de Deep Learning (GPUs de gran computabilidad) aceleraría mucho el proceso de aprendizaje y, por tanto, reduciría los tiempos a la hora de validar los modelos, ajustar parámetros y volver a entrenar encontrando soluciones óptimas mucho más rápido y con posibilidad de realizar múltiples pruebas en un período de tiempo mucho más corto.

También es importante mencionar que sería interesante poder tener el modelo cargado en memoria a la hora de realizar las predicciones ya que de esta forma se aumentaría la velocidad a la hora de aplicar la inferencia y, por ende, obtendríamos una mejora considerable en el rendimiento del sistema.

7. BIBLIOGRAFÍA

[1] “Regresión Lineal con ejemplos en Python - IArtificial.net.” https://iartificial.net/regresion-lineal-con-ejemplos-en-python/ (accessed Apr. 10, 2020).

[2] “Regresión Polinómica en Python con scikit-learn - IArtificial.net.” https://iartificial.net/regresion-polinomica-en-python-con-scikit-learn/ (accessed Apr. 10, 2020).

[3] “Support Vector Machine vs Logistic Regression - Georgios Drakos - Medium.” https://medium.com/@george.drakos62/support-vector-machine-vs-logistic-regression-94cc2975433f (accessed Apr. 10, 2020).

[4] “KNN Classification using Scikit-learn - DataCamp.” https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn (accessed Apr. 10, 2020).

[5] M. R. Arbabshirani, K. A. Kiehl, G. D. Pearlson, and V. D. Calhoun, “Classification of schizophrenia patients based on resting-state functional network connectivity,” *Front. Neurosci.*, vol. 7, no. 7 JUL, p. 133, Jul. 2013, doi: 10.3389/fnins.2013.00133.

[6] M. Wieland, W. Liu, and F. Yamazaki, “Learning Change from Synthetic Aperture Radar Images: Performance Evaluation of a Support Vector Machine to Detect Earthquake and Tsunami-Induced Changes,” *Remote Sens.*, vol. 8, no. 10, p. 792, Sep. 2016, doi: 10.3390/rs8100792.

[7] “Redes Neuronales artificiales: Qué son y cómo se entrenan - Parte I | Blog Xeridia.” https://www.xeridia.com/blog/redes-neuronales-artificiales-que-son-y-como-se-entrenan-parte-i (accessed Apr. 10, 2020).

[8] “Redes Neuronales: una visión superficial - Fernando Sancho Caparrini.” http://www.cs.us.es/~fsancho/?e=72 (accessed Apr. 10, 2020).

[9] “Red neuronal artificial - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Red_neuronal_artificial (accessed Apr. 10, 2020).

[10] “The difference between Artificial Intelligence, Machine Learning and Deep Learning – Data Catchup.” https://datacatchup.com/artificial-intelligence-machine-learning-and-deep-learning/ (accessed Apr. 10, 2020).

[11] “Why Deep Learning over Traditional Machine Learning?” https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063 (accessed Apr. 10, 2020).

[12] “It’s Deep Learning Times: A New Frontier of Data - Towards Data Science.” https://towardsdatascience.com/its-deep-learning-times-a-new-frontier-of-data-a1e9ef9fe9a8 (accessed Apr. 19, 2020).

[13] M. Mostavi, Y.-C. Chiu, Y. Huang, and Y. Chen, “Convolutional neural network models for cancer type prediction based on gene expression,” *BMC Med. Genomics*, vol. 13, no. S5, p. 44, Apr. 2020, doi: 10.1186/s12920-020-0677-2.

[14] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, “Face recognition: A convolutional neural-network approach,” *IEEE Trans. Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997, doi: 10.1109/72.554195.

[15] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, “SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving,” 2017. Accessed: Apr. 09, 2020. [Online]. Available: https://blogs.nvidia.com/blog/2016/09/28/.

[16] P. Constante, A. Gordon, O. Chang, E. Pruna, F. Acuna, and I. Escobar, “Artificial Vision Techniques to Optimize Strawberrys Industrial Classification,” *IEEE Lat. Am. Trans.*, vol. 14, no. 6, pp. 2576–2581, Jun. 2016, doi: 10.1109/TLA.2016.7555221.

[17] C. Sandoval Niño, Z. Liliana, P. Ortiz, and F. Augusto, “Revista Facultad Nacional de Agronomía-Medellín,” *Rev. Fac. Nac. Agron.*, vol. 60, no. 2, pp. 4105–4127, 2007, Accessed: Apr. 09, 2020. [Online]. Available: http://www.redalyc.org/articulo.oa?id=179914078015.

[18] I. Aleksander, Ed., *Artificial Vision for Robots*. Boston, MA: Springer US, 1983.

[19] “Artificial Vision for the Blind by Connecting a Television C... : ASAIO Journal: Artificial Organ Research and Development.” https://journals.lww.com/asaiojournal/Fulltext/2000/01000/Artificial_Vision_for_the_Blind_by_Connecting_a.2.aspx (accessed Apr. 09, 2020).

[20] Y. Huang, C. Qiu, X. Wang, S. Wang, and K. Yuan, “A Compact Convolutional Neural Network for Surface Defect Inspection,” *Sensors*, vol. 20, no. 7, p. 1974, Apr. 2020, doi: 10.3390/s20071974.

[21] K. Gopalakrishnan, S. K. Khaitan, A. Choudhary, and A. Agrawal, “Deep Convolutional Neural Networks with transfer learning for computer vision-based data-driven pavement distress detection,” *Constr. Build. Mater.*, vol. 157, pp. 322–330, Dec. 2017, doi: 10.1016/j.conbuildmat.2017.09.110.

[22] T. Estienne *et al.*, “Deep Learning-Based Concurrent Brain Registration and Tumor Segmentation,” *Front. Comput. Neurosci.*, vol. 14, p. 17, Mar. 2020, doi: 10.3389/fncom.2020.00017.

[23] “3x3 convolution filters — A popular choice - IceCream Labs - Medium.” https://medium.com/@icecreamlabs/3x3-convolution-filters-a-popular-choice-75ab1c8b4da8 (accessed May 13, 2020).

[24] “MNIST Handwritten Digits Classification using a Convolutional Neural Network (CNN) – mc.ai.” https://mc.ai/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn/ (accessed May 16, 2020).

[25] Y. Lecun, L. Eon Bottou, Y. Bengio, and P. H. Abstract|, “Gradient-Based Learning Applied to Document Recognition.”

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks.” Accessed: May 16, 2020. [Online]. Available: http://code.google.com/p/cuda-convnet/.

[27] “Fig. A1. The standard VGG-16 network architecture as proposed in [32].... | Download Scientific Diagram.” https://www.researchgate.net/figure/Fig-A1-The-standard-VGG-16-

network-architecture-as-proposed-in-32-Note-that-only_fig3_322512435 (accessed May 16, 2020).

[28] "Visualizing the Features of a Convolutional Neural Network (for ResNet architecture) – On Data Science | Machine Learning | Deep Learning." https://mpeker.com/index.php/2019/11/02/visualize-features-of-a-convolutional-neural-network-for-resnet-architecture/ (accessed May 16, 2020).

[29] "Residual Networks (ResNets) - engMRK." https://engmrk.com/residual-networks-resnets/ (accessed May 16, 2020).

[30] "Common architectures in convolutional neural networks." https://www.jeremyjordan.me/convnet-architectures/ (accessed May 16, 2020).

[31] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware."

[32] "Basler ace acA2440-20gc - Area Scan Camera." https://www.baslerweb.com/en/products/cameras/area-scan-cameras/ace/aca2440-20gc/ (accessed Apr. 18, 2020).

[33] "Gigabit Ethernet - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Gigabit_Ethernet (accessed Apr. 08, 2020).

[34] "Distorsión óptica - Wiki." http://autores.uy/wiki/Distorsión_óptica (accessed Apr. 18, 2020).

[35] "FJN-HF9HA-1S - Iberoptics." http://www.iberoptics.com/es/lentes-montura-c-15mpx-12-y-23/fujifilm-europe_fjn-hf9ha-1s-2966.html (accessed Apr. 18, 2020).

[36] "Welcome to Protech Systems - Interactive kiosk(KIOSK), POS System(POS), Industrial Personal Computer(IPC)." https://www.protech.com.tw/Products/product_us.asp?PM_ID=594 (accessed Apr. 18, 2020).

[37] "Nvidia - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Nvidia (accessed Apr. 08, 2020).

[38] "Tarjeta gráfica GeForce RTX 2060 | NVIDIA." https://www.nvidia.com/es-es/geforce/graphics-cards/rtx-2060/ (accessed Apr. 08, 2020).

[39] "Qt Creator - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Qt_Creator (accessed Apr. 08, 2020).

[40] "Python - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Python (accessed Apr. 08, 2020).

[41] "Introduction — Python 3.8.2 documentation." https://docs.python.org/3/c-api/intro.html (accessed Apr. 08, 2020).

[42] "C++ - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/C%2B%2B (accessed Apr. 08, 2020).

[43] "pylon SDK Samples Manual | Basler." https://www.baslerweb.com/en/sales-support/downloads/document-downloads/pylon-sdk-samples-manual/ (accessed Apr. 08, 2020).

[44] "A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning | by Dipanjan (DJ) Sarkar | Towards Data Science." https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a (accessed Jul. 30, 2020).