UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"CLASIFICACIÓN AUTOMÁTICA DE FRUTAS A PARTIR DE IMÁGENES CON PHYTON. TÉCNICAS DE TRANSFERENCIA DE APRENDIZAJE A PARTIR DE CONJUNTOS DE DATOS PEQUEÑOS"

TRABAJO FIN DE GRADO

Septiembre
2022

AUTOR: Manuel Ribes Escámez

DIRECTOR/ES: Ramon Pedro Ñeco García

RESUMEN

En el presente trabajo se ha realizado el diseño e implementación de un sistema de reconocimiento y clasificación automática de frutas a partir de imágenes usando arquitecturas de redes convolucionales, analizando su efectividad usando distintos valores de los parámetros de aprendizaje.

Además, se han estudiado técnicas de transferencia del aprendizaje son útiles cuando se dispone de conjuntos pequeños de datos el entrenamiento de las redes convolucionales como pueden ser, por ejemplo, técnicas de extracción de características o de sintonizado fino.

AGRADECIMIENTOS

En primer lugar, me gustaría agradecer a mi tutor, Ramon Pedro Ñeco García, la oportunidad de realizar este trabajo, con el cual he aprendido muchas cosas y de la que voy a guardar un gran recuerdo, y la dedicación que me ha prestado durante todo este proceso.

En segundo lugar, me gustaría agradecer a mis amigos, que han estado en todo momento y me han ayudado a generar ideas para este trabajo.

Por último, a mi familia, que no ha dejado que me desanimara y me ha apoyado a seguir en todo momento.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN

La inteligencia artificial (IA) hace posible que las máquinas aprendan por prueba y error, se ajusten a nuevas entradas y realicen tareas de forma "similar" a como pueden realizarlas los seres humanos. La mayoría de los ejemplos de inteligencia artificial sobre los que oye hablar hoy día recurren al *Deep Learning* (aprendizaje profundo) y al procesamiento del lenguaje natural. Empleando estas tecnologías, las computadoras pueden ser entrenadas para realizar tareas específicas procesando grandes cantidades de datos y reconociendo patrones en los datos.

1.1. MOTIVACIÓN

El *Deep Learning* (aprendizaje profundo) es una rama de *Machine Learning* (aprendizaje automático) formada por un conjunto de algoritmos que intentan modelar abstracciones de alto nivel en datos usando grafos profundos con múltiples capas de procesamiento. Estas capas de procesamiento pueden estar compuestas por transformaciones tanto lineales como no lineales.

El *Deep Learning*, que tuvo sus inicios en los años 80, no es más que un ejemplo de implementación de *Machine Learning*. Debido a las recientes investigaciones sobre el mundo de la Inteligencia Artificial, esta tecnología se encuentra ahora mismo en constante desarrollo, ya que se ha demostrado que es la forma más eficiente de hacer con un computador una simulación de lo que nuestro cerebro puede llegar a hacer. Por medio del presente TFG descubriremos que estas tecnologías, que a simple vista parecen futuristas o sacadas de una película de ciencia ficción, están accesibles casi para cualquier programador.

1.2. ¿QUÉ PODEMOS HACER MEDIANTE *DEEP LEARNING*?

Las técnicas de *Deep Learning* nos han abierto una puerta a muchos tipos de problemas que hasta hace relativamente poco eran difícilmente programables. De entrada, podemos resolver problemas de clasificación y predicción de una manera mucho más eficiente y precisa. Algunos ejemplos concretos de aplicaciones que utilizan redes neuronales podrían ser:

1. **Coloreado de imágenes en blanco y negro:** Tradicionalmente, esta tarea ha sido llevada a cabo por el ser humano de manera manual, hasta que mediante el uso de *Deep Learning*, se han utilizado los objetos y contextos de las propias imágenes para ser coloreadas.

2. **Adición de sonido a películas mudas:** Pongamos el ejemplo de querer recrear el sonido que hace un palo al golpear con una superficie. Si entrenamos el sistema utilizando una gran cantidad de vídeos donde se muestra el sonido que hace un palo al ser golpeado contra diferentes superficies, nuestra red neuronal asociará

los frames del vídeo mudo con la información ya aprendida, y seleccionará el sonido que mejor se adapte a la escena.

3. **Traducciones automáticas:** Pese a que la traducción de palabras, frases o textos lleva siendo posible desde hace muchos años, mediante la utilización de redes neuronales se han alcanzado resultados mucho mejores sobre todo en dos áreas: traducción de texto, y traducción de imágenes. En el caso de los textos, por ejemplo, se ha pasado de traducir palabras sueltas, a analizar y entender la gramática y la conexión entre palabras, haciendo así deducciones mucho más precisas del significado global de la frase. En el caso de traducción de textos en imágenes ya es posible identificar letras, con ellas formar palabras, y a su vez formar un texto. En muchos contextos a esto se le llama traducción visual, y ya ha sido implementado en aplicaciones como Google Translator.

4. C**lasificación de objetos en fotografías:** Esta funcionalidad consiste en detectar y clasificar uno o más objetos de la escena de una fotografía para después crear una palabra o frase que describa el contenido de la imagen. En 2014, hubo un ”boom“ de algoritmos que alcanzaron resultados impresionantes a la hora de resolver este problema.

5. **Generación de textos:** Esta tarea consiste en, dado una recopilación de textos, generar nuevos textos a partir de una palabra o una frase. Una aplicación podría ser la generación de textos escritos a mano. Mediante *Machine Learning*, podemos aprender la relación entre los movimientos del bolígrafo y las letras escritas con la idea de generar nuevos ejemplos. Esta funcionalidad puede ser utilizada por médicos forenses o especialistas en análisis de manuscritos, ya que es posible aprender una cantidad impresionante de estilos de escritura. Otra posible aplicación sería la de generar nuevos textos con nuevas historias.

6. **Inteligencia artificial en videojuegos**: Por todos es sabido que la inteligencia artificial en los videojuegos es una tecnología con la que llevamos conviviendo mucho tiempo. En un shooter, la IA sabe identificar donde está tu jugador, y con esa información, sabe a quién y donde disparar. Pero... ¿Y si fuese posible analizar, por ejemplo, todos los pixeles de la pantalla? Mediante *Deep Learning* esto es relativamente sencillo, permitiendo a la IA rival analizar mucha más información para tomar así mejores decisiones.

Otros ejemplos de aplicaciones de redes neuronales para resolver problemas actuales podrían ser: reconocimiento y traducción de discursos y charlas en tiempo real, foco automático en objetos en movimiento en fotografías, conversión automática de objetos en fotografías, respuestas automáticas a preguntas sobre objetos en fotografías, etc. Como se puede observar, casi todas estas tareas se tratan de automatizar algún proceso abstracto, y son tareas que el ser humano puede hacer manualmente, pero una maquina es capaz de aprender a hacerlo en mucho menos tiempo y con mucha más eficiencia y precisión.

1.3. OBJETIVOS

En este trabajo se ha diseñado e implementado un sistema de reconocimiento y clasificación automática de frutas a partir de imágenes usando arquitecturas de redes convolucionales, analizando su efectividad usando distintos valores de los parámetros de aprendizaje. Además, se estudian técnicas de *Transfer Learning* especialmente útiles en los casos en los que se dispone de conjuntos pequeños de datos para el entrenamiento de las redes como pueden ser, por ejemplo, técnicas de extracción de características o de sintonizado fino.

1.4. ORGANIZACIÓN DEL TRABAJO

El presente trabajo se divide en 6 capítulos los cuáles tratarán los siguientes aspectos:

- Capítulo 1: En este primer capítulo se trata una breve introducción trabajo, la organización y los objetivos del mismo.
- Capítulo 2: Contiene explicaciones de los diferentes conceptos necesarios que se verán en este TFG para trabajar con una red neuronal.
- Capítulo 3: Trataremos las herramientas usadas durante la realización del trabajo y un pequeño ejemplo previo al mismo.
- Capítulo 4: En este punto comentaremos cómo funciona la red de ejemplo en la que nos hemos basado para realizar el trabajo. Después se explicará cómo se ha adaptado la misma para el objetivo de este trabajo.
- Capítulo 5: En este capítulo realizaremos un comentario general sobre todas las pruebas realizadas en el trabajo y llegaremos a una conclusión. Se explicará la obtención de los datos y de los algoritmos empleados durante el proceso.
- Capítulo 6: En este capítulo final se comentarán las conclusiones obtenidas a través de este trabajo y posibles trabajos futuros de ampliación para este TFG.

CAPÍTULO 2. ESTADO DEL ARTE. REDES NEURONALES

Desde hace cientos de años, se ha intentado estudiar el cerebro humano. Sin embargo, a partir de los años 50, cuando se empezó a desarrollar las bases de la tecnología actual, hubo un fuerte avance en este estudio desde la perspectiva de la computación. Las redes neuronales artificiales, como su propio nombre indica, pretenden imitar la forma de funcionamiento de las neuronas que forman el cerebro humano.

- A principios de los años 40, el neurólogo Warren McCulloch y el matemático Walter Pitts propusieron los primeros modelos matemáticos y eléctricos de redes neuronales. Describieron como funcionaban las neuronas, y modelaron una red neuronal simple utilizando circuitos eléctricos.
- A finales de década, Donald Hebb definió dos conceptos muy importantes:
    - El proceso de aprendizaje se localiza principalmente en la sinapsis, también conocida como conexión entre neuronas.
    - La información se representa en el cerebro humano mediante un conjunto de neuronas activas o inactivas.

Estas dos reglas se sintetizan en la regla de aprendizaje de Hebb, que sigue siendo aplicada en algunos modelos actuales. Esta regla afirma que los cambios en los pesos de la sinapsis se basan en la interacción entre las neuronas pre y post sintácticas, y en el número de veces que se activan de manera simultánea.

- En 1956 en la ciudad de Dartmouth, se llevó a cabo la primera conferencia sobre Inteligencia Artificial donde se discutió sobre la capacidad de las máquinas para simular el aprendizaje humano.
- A finales de los años 50, el neurobiólogo Frank Rosenblatt empezó a trabajar en el concepto de Perceptrón. Esto dio lugar a la regla de aprendizaje Delta, que permitía emplear señales continuas de entrada y de salida.
- En 1959 Bernard Widrow y Marcian Hoff desarrollaron los algoritmos Adaline (Adaptative Linear Neuron) y Madaline (Multiple Adaline). Estos modelos fueron empleados para desarrollar la primera solución a un problema real aplicando redes neuronales: un filtro que eliminaba el eco en las llamadas telefónicas.
- En 1982, el interés por las redes neuronales se renovó. John Hopfield presentó varios artículos a la Academia Nacional de Ciencias en los que no solo describía el modelado del cerebro humano, si no que demostró que aplicaciones podría tener su estudio a la hora de crear nuevos dispositivos.
- En 1984, Kohonen desarrolló una familia de redes de memoria asociativa y mapas autoorganizativos, actualmente llamadas redes de Kohonen.
- En 1985, Paul Werbos desarrolló el algoritmo Backpropagation que resolvió el problema de entrenar redes neuronales multicapa de forma efectiva.
- En 1986, los investigadores Hinton y Sejnowski desarrollaron la máquina de Boltzmann, un tipo de red neuronal recurrente estocástica.

2.1. LA NEURONA BIOLÓGICA

El cerebro, visto a un alto nivel y simplificando enormemente su estructura, se podría definir como un conjunto de millones y millones de células, llamadas neuronas, interconectadas entre ellas mediante sinapsis. La sinapsis se lleva a cabo en la zona donde 2 neuronas se conectan, y las partes de la célula que se encargan de realizarla son las dendritas y las ramificaciones del axón.

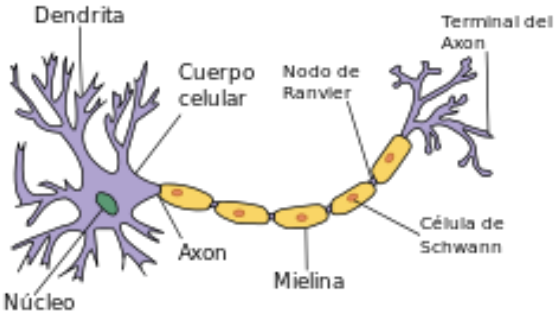


Figura 2.1. Estructura de una neurona

Cada neurona desarrolla impulsos eléctricos que se trasmiten a lo largo de esta mediante su axón. Este, al final se divide en ramificaciones axionales, que conectan con otras neuronas mediante sus dendritas. El conjunto de elementos que hay entre la ramificación axional y la dendrita forman la sinapsis, que regula la transmisión del impulso eléctrico mediante unos elementos químicos llamados neurotransmisores.

Los neurotransmisores liberados en la sinapsis pueden tener un efecto negativo o positivo sobre la transmisión del impulso eléctrico en la neurona que los recibe en sus dendritas. Esta neurona recibe varias señales de las distintas sinapsis, y las combina consiguiendo un cierto nivel de estimulación. En función de este nivel de activación, la neurona emite señales eléctricas mediante impulsos, con una intensidad determinada y con una frecuencia llamada tasa de disparo. En resumen, si consideramos que la información del cerebro está codificada en impulsos eléctricos que se transmiten entre neuronas, y que los impulsos se ven modificados básicamente en la sinapsis, podemos intuir que la codificación del aprendizaje estará en la sinapsis y en la forma en la que las neuronas dejan pasar o inhiben las señales segregando neurotransmisores.

2.2. LA NEURONA ARTIFICIAL

Las neuronas artificiales son modelos que tratan de simular el comportamiento de las neuronas biológicas. Cada neurona se representa como una unidad de proceso que forma parte de una entidad mayor, la red neuronal.

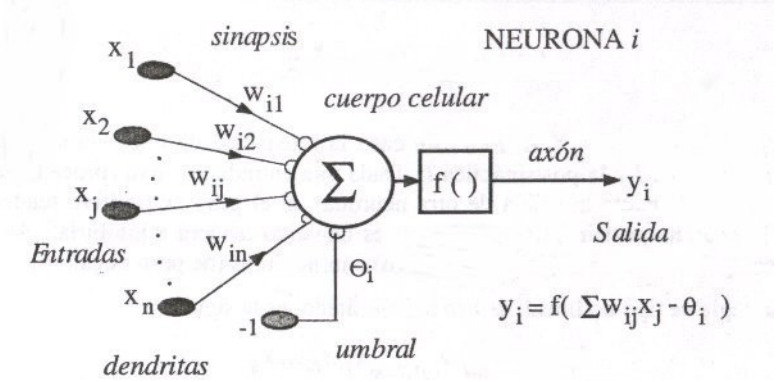


Figura 2.2: Esquema de una red neuronal artificial

La neurona artificial se comporta en cierto modo como una biológica, pero de forma simplificada:

- Entradas (x1, x2, ...): estos valores pueden ser enteros, reales o binarios y equivaldrían a los impulsos que envían otras neuronas a través de sus dendritas.

- Pesos (w1, w2, ...): equivaldrían a los mecanismos de sinapsis necesarios para transmitir el impulso.

- El producto de los valores *xi, wi* equivaldría a las señales químicas inhibidoras y excitadoras que se dan en la neurona. Estos valores son la entrada de la función de activación, que convierte todo el conjunto de valores en uno solo llamado potencial. La función de ponderación suele ser la suma ponderada de las entradas y los pesos sinápticos.

- La salida de la función de ponderación llega a la función de activación que transforma este valor en otro en el dominio que trabajen las salidas de las neuronas. Suele ser una función no lineal como la función paso o la función sigmoide, aunque también se usa funciones lineales.

2.3. FUNCIÓN DE PROPAGACIÓN

Esta función se encarga de transformar las diferentes entradas que provienen de la capa anterior, en el potencial de la neurona actual. Normalmente, las neuronas han de tratar simultáneamente con varios valores de entrada, y han de hacerlo como si se tratase de uno solo. Esta función viene a resolver el problema de combinar las entradas simples (x1, x2, x3...) en una sola entrada global.

Multiplicando las entradas por los pesos, se permite que un valor muy grande de entrada pueda tener una influencia pequeña, si sus pesos son pequeños

2.4. FUNCIÓN DE ACTIVACIÓN

La función de activación combina el potencial que nos proporciona la función de propagación con el estado actual de la neurona para conseguir el estado futuro de ´esta (activada/desactivada). Esta función es normalmente creciente y monótona. Se va a hablar de 4 tipos de función de activación.

2.4.1. ACTIVACIÓN RELU

Esta función es la más utilizada actualmente, ya que la mayoría de las demás están acotadas entre 0 y 1. Esta función es muy simple, se limita a enviar a la siguiente neurona el mayor valor entre el cálculo previo o 0.

$$F(X) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

2.4.2. ACTIVACIÓN SOFTMAX

Convierte los valores de las neuronas de salida a tanto por 1 sobre el total de neuronas de salida. Con esto se consigue una gran diferenciación entre valores muy grandes y pequeños.

2.4.3. ACTIVACIÓN LOGÍSTICA/SIGMOID

Acota los resultados entre 0 y 1, genera una curva suave.
Suele dar problemas en números muy grandes o pequeños porque la curva se va rectificando más cuanto más se aleje del 0. Este problema se llama desvanecimiento de gradiente.

$$F(x) = \frac{1}{1+e\hat{}-x}$$

2.4.4. ACTIVACIÓN TANGENTE HIPERBÓLICA/TANH

Esta función acota los valores entre -1 y 1. Tiene mayor derivada que la función Sigmoid, esto genera un aprendizaje más rápido ya que genera cambios mayores en la red. También tiene problema de desvanecimiento de gradiente y tiene mucho coste computacional.

$$F(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

2.5. TIPOS DE APRENDIZAJE EN REDES NEURONALES

2.5.1. APRENDIZAJE SUPERVISADO

Los modelos de aprendizaje supervisado son aquellos utilizados cuando el problema nos presenta de antemano tanto el conjunto de datos de entrada como los atributos que queremos predecir. Existen dos categorías:

- Regresión: los valores de salida son una o más variables continuas. Un ejemplo sería predecir el valor de una casa en función de sus metros cuadrados, el número de habitaciones, tamaño de la piscina...

- Clasificación: los datos pertenecen a dos o más clases, y se busca aprender como clasificar nuevas entradas en esas clases a partir de datos que ya conocemos. Uno de los ejemplos más conocidos es el Iris dataset, donde se intenta clasificar los datos en 4 tipos de flores según la longitud y la anchura de sus pétalos y sépalos. Otro de ellos es el problema del dataset MNIST, que busca clasificar imágenes de números en los 10 tipos de caracteres numéricos.

2.5.2 APRENDIZAJE NO SUPERVISADO

Este problema se da cuando no hay información previa de salidas para el conjunto de entradas. En estos casos, el objetivo es encontrar grupos mediante clustering, o determinar una distribución de probabilidad sobre un conjunto de entrada.

2.6. PERCEPTRÓN

En los años 50, F.Rosenblatt desarrolló por primera vez la idea de Perceptrón basándose en la regla de aprendizaje de Hebb y en los modelos de neuronas biológicas de McCulloch y Pitts. Una de las características que más interés despertó de este modelo fue su capacidad para aprender a reconocer patrones. El perceptrón se basa en una arquitectura monocapa. Está constituido por un conjunto de células de entrada, que reciben los patrones a reconocer o clasificar, y una o varias células de salida que se ocupan de clasificar estos patrones de entrada en clases. Cada célula de entrada está conectada con todas las células de salida.

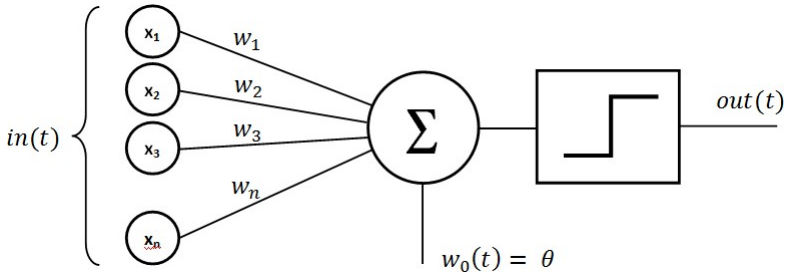


Figura 2.3: Perceptrón simple

El umbral (θ) es un parámetro utilizado como factor de comparación a la hora de generar la salida de una neurona. En esta arquitectura, la salida de la neurona se calcula multiplicando los valores de las entradas, por los pesos de las conexiones:

$$y' = \sum_{i=1}^{n} wi * xi$$

Sin embargo, la salida definitiva depende del umbral, por lo que se obtiene aplicándole la función de salida al nivel de activación de la célula, es decir:

$$y = F(y', \theta)$$

$$y = \begin{cases} si\ y' \geq \theta\ \ 1 \\ si\ y' < \theta\ \ 0 \end{cases}$$

Con estas fórmulas, podemos deducir que la salida de la célula (y) será 1 (neurona activada) si $y\ 0 > \theta$, o 0 (neurona desactivada) en caso contrario.
La función de salida (F) produce una salida binaria, por lo que es un diferenciador en dos categorías. En el caso de tener únicamente dos dimensiones, se podría llegar a la ecuación:

$$w1x1 + w2x2 + \theta = 0$$

que es la ecuación de una recta con pendiente

$$-\frac{w1}{w2}$$

y cuyo corte en la abscisa en el eje x pasa por

$$-\frac{\theta}{w2}$$

Podemos imaginarnos el perceptrón como una recta, que en una gráfica de dos dimensiones deja las dos categorías a separar a un lado y a otro de la misma.

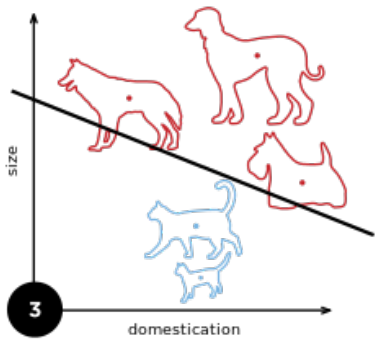


Figura 2.4: Separación lineal del perceptrón

El perceptrón es un tipo de red basado en aprendizaje supervisado, es decir, necesita conocer los valores esperados para cada una de las entradas antes de construir el modelo. Este proceso se lleva a cabo mediante la inserción de patrones de entrada del dataset con el que se quiere entrenar la red.

Entrenar la red no es más que encontrar los pesos sinápticos y el umbral que mejor haga predecir a nuestra red los resultados esperados. Para la determinación de estas variables, se sigue un proceso adaptativo. Se comienza con unos valores aleatorios, y se van modificando según la diferencia entre los valores deseados y los obtenidos por la red. En resumen, el perceptrón simple aprende iterativamente siguiendo estos pasos:

1. Inicialización de variables

2. Bucle de iteraciones:

    a) Bucle para todos los ejemplos

        1) Leer valores de entrada

        2) Calcular error

        3) Actualizar pesos según el error
            a´ Actualizar pesos de entradas
            b´ Actualizar el umbral

    b) Comprobar que el vector de pesos es correcto

3. Salida

Hasta ahora, hemos descrito el perceptrón como un clasificador. Sin embargo, también pueden ser usados para emular funciones lógicas elementales como AND, OR y NAND. Consideremos las funciones AND y OR. Dado que son funciones linealmente separables, pueden ser aprendidas por un perceptrón.
Sin embargo, la función XOR no puede ser aprendida por un único perceptrón, ya que requiere el uso de al menos dos rectas para separar las clases. Si se quiere alcanzar esta funcionalidad, es necesario utilizarse al menos una capa más. Aquí es donde entra en juego el perceptrón multicapa.

2.7. PERCEPTRÓN MULTICAPA

El perceptrón multicapa tuvo su origen en los años 80, con la idea de solucionar el mayor problema que presenta el perceptrón simple: no ser capaz de aprender funciones no linealmente separables (como es el caso de la función XOR). Se ha demostrado que es un aproximador universal de funciones
Éste, es un modelo de red neuronal con alimentación hacia delante. Está compuesto por varias capas ocultas entre la entrada y la salida, y se caracteriza por tener salidas disjuntas pero relacionadas entre sí (siendo la salida de una neurona la entrada de la siguiente).

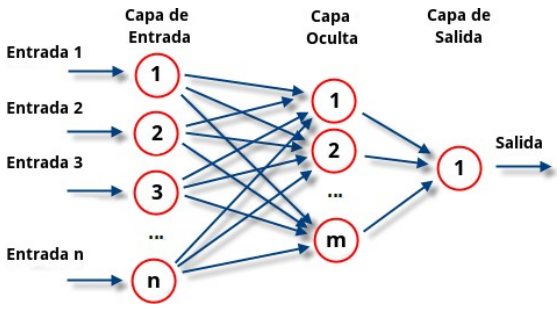


Figura 2.5: Perceptrón multicapa

En el perceptrón multicapa, podemos diferenciar una fase de propagación de los valores de entrada hacia delante, y una fase de aprendizaje en la que los errores obtenidos a la salida del perceptrón se van propagando hacia atrás con el objetivo de actualizar los pesos de las conexiones mediante el gradiente de la función de error. Este algoritmo se llama backpropagation o retropropagación.

A la hora de diseñar un perceptrón multicapa para abordar un problema, el primer paso es determinar la arquitectura de la red. Esto implica determinar tres variables, ya explicadas anteriormente:

- **Función de activación:** se elige según el comportamiento y el recorrido deseado, pero no influye en la capacidad de la red para resolver el problema.
- **Número de neuronas de entrada y de salida:** viene dado por la definición del problema.
- **Número de capas ocultas:** pese a ser una variable que ha de definir el diseñador, no existe un método que determine el número óptimo de capas ocultas necesarias para resolver un problema, ya que existen muchos tipos de arquitectura distintos capaces de resolver un mismo ejercicio.

2.7.1. ALGORITMO DE RETROPROPAGACIÓN O BACKPROPAGATION

El proceso de aprendizaje del perceptrón multicapa consiste en propagar hacia atrás cierta información desde la salida del perceptrón hasta su entrada. Esta información será el error entre la salida obtenida y la salida esperada, con el objetivo de que en las siguientes iteraciones la salida obtenida se aproxime lo máximo posible a la salida esperada. Por tanto, nos encontramos ante un algoritmo de aprendizaje supervisado. Hay muchas funciones de error que se pueden utilizar, pero el perceptrón multicapa usa habitualmente la función del error cuadrático medio, que se define como:

$$e(n) = \frac{1}{2}\sum_{i=1}^{nc}\big(si(n) - yi(n)\big)\text{^}2$$

siendo *n* cada una de las iteraciones del aprendizaje. El error obtenido *e*(*n*) es el error de una sola iteración. En resumen, el perceptrón multicapa aprende encontrando el mínimo de la función de error.
Aunque el aprendizaje debe llevarse a cabo minimizando el error total, los métodos más utilizados son los basados en métodos de gradiente estocástico, que consisten en

minimizar el error en cada patrón e(n). Por tanto, aplicando el método de gradiente descendiente, cada peso de la red *w* se actualiza para cada patrón de entrada *n*, y siendo α la tasa de aprendizaje, de acuerdo con la siguiente ecuación:

$$w(n) = w(n - 1) - \alpha \frac{\partial e(n)}{\partial w}$$

Dado que las neuronas de la red están agrupadas en capas ocultas según niveles, es posible aplicar el método del gradiente descendente de forma eficiente en todas las neuronas, llevando a cabo el algoritmo de retropropagación o regla delta generalizada.

La tasa de aprendizaje es un parámetro que determina la velocidad a la que van a cambiar los pesos de las conexiones de la red. Tiene generalmente un rango [0,1], siendo los valores más cercanos a cero los que hacen que los pesos cambien poco a poco, acercándose lentamente a la convergencia, y los cercanos a uno los que hacen que la red converja más rápidamente al principio, pero siendo posible que los pesos oscilen demasiado al encontrar el peso óptimo final. Por esta razón es importante encontrar una tasa de aprendizaje óptima.
Aquí es donde entra en juego un segundo término llamado momento (η), que pondera cuánto queremos que influya lo que los pesos han cambiado en la anterior iteración. Con ello, sabremos que si los pesos han cambiado mucho es que estamos aún lejos del valor de la tasa de aprendizaje óptimo, por lo que se avanzará en su búsqueda más rápidamente. Por otro lado, si los pesos no han cambiado apenas, sabremos que el valor óptimo de α está cerca.

$$w(n) = w(n - 1) = -\alpha \frac{\partial e(n)}{\partial w} + \eta \Delta w(n - 1)$$

2.7.2. SOBREAJUSTE Y EARLY STOPPING

El sobreajuste o overfitting, es el efecto secundario de sobreentrenar un algoritmo de aprendizaje con ciertos datos para los que ya se conoce el resultado deseado. El algoritmo debe alcanzar un estado en el cual sea capaz de predecir otros casos a partir de lo ya aprendido mediante datos de entrenamiento, generalizando para poder resolver distintas situaciones a las ya presentes durante el entrenamiento. No obstante, cuando un sistema se entrena demasiado o se entrena con datos extraños, el algoritmo tiende a quedarse ajustado a unas características muy específicas de los datos de entrenamiento que no representan un estado general del problema. En este estado, nuestro diseño es más eficaz al responder a muestras del conjunto de entrenamiento, pero ante nuevas entradas el resultado empeora. Una manera de resolver este problema es extraer un conjunto de entradas del dataset de entrenamiento, y utilizarlo de manera auxiliar a modo de validación. Ya que este subconjunto se deja al margen durante el entrenamiento, su objetivo será evaluar el error en la red tras cada época para determinar cuándo este empieza a aumentar. Cuando aumente, se procedería a detener el entrenamiento y se conservarían los valores de los pesos del ciclo anterior. A este método se le conoce como early-stopping.

2.8. REDES NEURALES CONVOLUCIONALES

Las redes neuronales convolucionales son muy similares a las redes neuronales ordinarias, como el perceptrón multicapa. Las neuronas tienen pesos, sesgos, reciben una entrada con la que realiza un producto escalar y sobre la que aplica una función de activación, tienen una función de perdida...
Sin embargo, se utilizan principalmente para resolver el mayor problema que tienen las redes neuronales ordinarias: el tratamiento de imágenes. Pese a que con las redes neuronales estándar es posible manejar imágenes, en cuanto las imágenes aumentan su tamaño y calidad esto se vuelve intratable. Al tratarse de neuronas totalmente conectadas, provocaríamos un desperdicio de recursos, así como un rápido sobreajuste.
Las redes neuronales convolucionales trabajan dividiendo y modelando la información en partes más pequeñas, y combinando esta información en las capas más profundas de la red. Por ejemplo, en el caso del tratamiento de una imagen, las primeras capas tratarían de detectar los bordes de las figuras. Las siguientes capas buscarían combinar los patrones de detección de bordes para conseguir formas más simples, y aplicar patrones de posición de objetos, iluminación... Por ´último, en las últimas capas se intentará hacer coincidir la imagen con todos los patrones descubiertos, para conseguir una predicción final de la suma de todos ellos. Así es como las redes neuronales convolucionales consiguen modelar una gran cantidad de datos, dividiendo previamente el problema en partes para conseguir predicciones más sencillas y precisas.

En general, todas las redes neuronales convoluciones están formadas por una estructura compuesta por 3 tipos de capas:

- **Capa convolucional:** Esta capa le da nombre de la red. En vez de utilizar la multiplicación de matrices como en las redes neuronales estándar, se aplica una operación llamada convolución. Esta operación recibe como entrada una imagen, y sobre ella aplica un filtro que nos devuelve su mapa de características, reduciendo así el tamaño de sus parámetros.

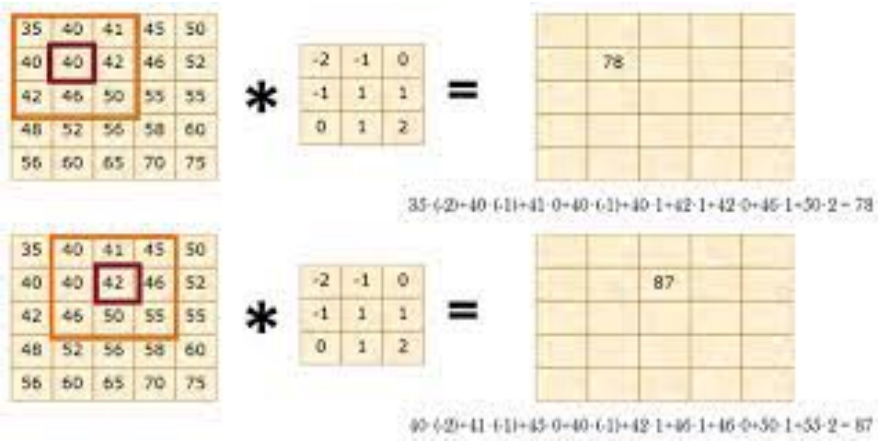


Figura 2.6. Proceso de convolución 3x3

- **Capa de pooling:** Esta capa se coloca generalmente después de la capa convolucional. Es la encargada de reducir la cantidad de parámetros a analizar reduciendo las dimensiones espaciales, quedándose de esta forma con las características más comunes.

La operación que se suele aplicar en esta capa es “max-pooling”, que divide la imagen de entrada en un conjunto de rectángulos, y respecto a cada uno de ellos, se queda con el valor máximo.
También existe la operación “average-pooling”, que en vez de quedarse con el valor máximo del rectángulo se queda con la media de sus valores.

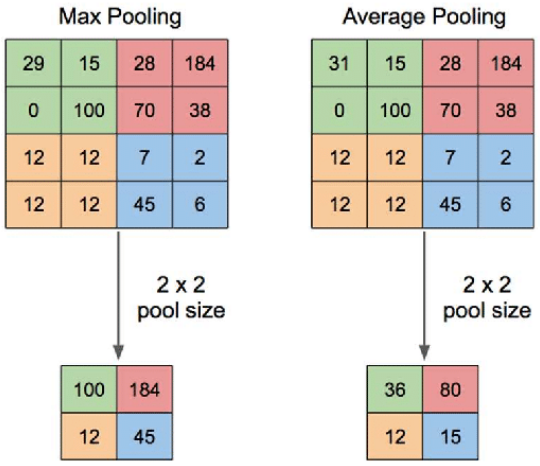


Figura 2.7. Agrupación máxima

- **Capa clasificadora totalmente conectada:** Una vez que la imagen ha pasado tanto por las capas convolucionales como las de pooling y se han extraído sus características más destacadas, los datos llegan a la fase de clasificación. Para ello, las redes convolucionales normalmente utilizan capas completamente conectadas en las que cada píxel se trata como una neurona independiente. Las neuronas de esta fase funcionan como las de un perceptrón multicapa, donde la salida de cada neurona se calcula multiplicando la salida de la capa anterior por el peso de la conexión, y aplicando a este dato una función de activación.

CAPÍTULO 3. HERRAMIENTAS UTILIZADAS

En este capítulo se describen las herramientas que se han usado para el desarrollo del trabajo.

3.1. KERAS

Keras es una biblioteca de redes neuronales de código abierto escrita en Python. Es capaz de ejecutarse sobre TensorFlow, Microsoft Cognitive Toolkit o Theano.

Está especialmente diseñada para posibilitar la experimentación en poco tiempo con redes de Deep Learning. Es amigable para el usuario, modular y extensible

Su autor principal y mantenedor ha sido el ingeniero de Google François Chollet.
Chollet explica que Keras ha sido concebido para actuar como una interfaz de machine learning. Ofrece abstracciones más intuitivas y de alto nivel haciendo más sencillo el desarrollo de modelos de aprendizaje profundo independientemente del backend computacional utilizado.

Contiene varias implementaciones de los bloques constructivos de las redes neuronales como por ejemplo los layers, funciones objetivo, funciones de activación, optimizadores matemáticos.

3.1.1. INSTALACIÓN

Para la instalación de Keras, son necesarias las siguientes dependencias:

- **numpy y scipy:** librerías científicas para Python numéricas.
- **pyyaml**: parseador YAML para Python.
- **HDF5 y h5py:** paquete para el manejo de grandes cantidades de datos. (Opcional, pero requerido si se usan funciones para cargar/guardar modelos)
- **cuDNN - Nvidia CUDA:** librería para tarjetas gráficas NVIDIA que permite la aceleración de la GPU en temas de redes neuronales. (Opcional pero recomendado si se usan CNNs)
- **Tensorflow o Theano:** librerías Python enfocadas a Machine Learning.

Existen 2 modos de instalación:

- **Desde PyPI:** sudo pip install keras
- **Desde repositorio Git:** sudo python setup.py install

Por defecto, Keras usa Tensorflow como motor backend. Sin embargo, esta opción puede ser configurada. La primera vez se ejecuta Keras, se crea un fichero de configuración en /.keras/keras.json", que puede ser modificado.

```
"image_dim_ordering": "tf",
"epsilon": 1e-07,
 "floatx": "float32",
"backend": "tensorflow"
```

3.1.2. MODELOS

Hay dos tipos de modelos disponibles para su implementación en Keras: el modelo secuencial, y la clase Model (más general). Ambos modelos presentan varios métodos en común, como summary (), get_config() o to_json() que devuelven información básica sobre el modelo, o get_weights() y set_weights(), los métodos getter y setter propios de los pesos del modelo.

3.1.2.1. MODELO SECUENCIAL

Los métodos básicos del modelo secuencial son:

- compile(self, optimizer, loss, metrics=[], sample_weight_mode=None) : Calcula automáticamente la función gradiente y prepara el modelo para una forma concreta de entrenamiento.
- fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[], validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None) : Entrena el modelo para un número fijado de ciclos (nb_epoch), con actualizaciones de gradiente cada batch_size ejemplos.
- evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None) : Calcula la función de perdida dados unos datos de entrada, lote por lote.
- predict(self, x, batch_size=32, verbose=0) : Genera predicciones para unos valores de entrada.

3.1.2.1. CLASE MODEL

Con la clase Model, dadas una entrada y una salida, podemos inicializar nuestro modelo de la siguiente forma:

```
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(input=a, output=b)
```

En caso de que tengamos múltiples conjuntos de entradas y de salidas, también podríamos inicializarlo de la siguiente manera:

```
model = Model(input=[a1, a2], output=[b1, b3, b3])
```

3.1.3. CAPAS

A la hora de diseñar las capas de nuestra red neuronal, tenemos una serie de funciones disponibles:

- Dense: Para capas regulares totalmente conectadas
- Activation: Se aplica una función de activación a una salida.
- Dropout: Consiste en establecer al azar una fracción de unidades de entrada a 0 en cada actualización durante la fase de entrenamiento. Esto ayuda a evitar el sobreajuste.
- Reshape: Transforma la salida en una forma concreta.

Algunos ejemplos de uso podrían ser:

- model.add(Dense(32, input_dim=16)) : modelo que contaría con una array de entrada de tamaño 16, y de salida de 32.
- model.add(Activation('tanh')) : aplica la función de activación tangencial.
- model.add(Dropout(0.2)) : establecería entradas a 0 en el 20 % de los casos.
- model.add(Reshape((3, 4), input_shape=(12,))) : define una capa de salida de 3x4 para nuestro modelo.

3.1.4. CAPAS CONVOLUCIONALES

Keras presenta diversas funciones para desarrollar capas convolucionales según las dimensiones de su entrada.
Un ejemplo de uso donde se define una capa convolucional de dos dimensiones, con 32 mapeadores de tamaño 5x5 y una función de activación rectificadora, sería:

```
model.add(Convolution2D(32, 5, 5, border_mode='valid', activation='relu'))
```

Además de la versión en 2D, Keras nos ofrece funciones para entradas de una dimensión o tres dimensiones. Aparte, tenemos funcionalidades para desarrollar capas convolucionales dilatadas (AtrousConvolution2D), hacer deconvolución (Deconvolution2D), etc.

3.1.5. CAPAS POOLING

Al igual que ocurre con las capas convolucionales, Keras ofrece un buen número de funciones para desarrollar capas pooling en redes convolucionales, dependiendo de las dimensiones de la entrada. Como en el caso anterior, una función representativa sería:

```
MaxPooling2D(pool_size=(2, 2), strides=None, border_mode='valid', dim_ordering='default')
```

Si en vez de utilizar la función MaxPooling se requiere que nuestra capa utilice otro tipo de algoritmos, Keras también nos ofrece funciones como: AveragePooling2D, GlobalMaxPooling2D... Así como sus variantes para 1D y 3D.

3.1.6. INICIALIZADORES

Los inicializadores definen la manera de inicializar los pesos de las capas de nuestro modelo. El argumento usado en para definir este inicializador en las capas es init. Las opciones más comunes son:

- uniform
- normal
- identity
- zero
- glorot
- normal
- glorot
- uniform

3.1.7. ACTIVACIONES

El modo de activación puede ser implementado mediante una capa Activation (model.add (Dense(64)); model.add(Activation('tanh'))), o mediante un parámetro que soportan todas las capas (model.add(Dense(64, activation='tanh'))). Las funciones más comunes ya comentadas son:

- **sigmoid:** función sigmoide / logística.
- **tanh:** función tangente hiperbólica.
- **relu:** función rectificadora.
- **softmax:** aproximación suavizada de la función rectificadora.

3.1.8. FUNCIÓN DE PÉRDIDA

La función de perdida es uno de los dos parámetros necesarios para compilar un modelo. Algunas de las funciones más conocidas que nos proporciona Keras son:

- **mean_squared_error(y_true, y_pred):** Calcula el error cuadrático medio.
- **mean_absolute_error / mae:** Calcula el error medio absoluto.
- **binary_crossentropy:** Calcula la entropía cruzada en problemas de clasificación binaria. También conocida como perdida logarítmica.
- **categorical_crossentropy:** se utiliza en tareas de clasificación multiclase. Estas son tareas en las que un ejemplo solo puede pertenecer a una de muchas categorías posibles, y el modelo debe decidir cuál.

3.1.9. OPTIMIZADORES

El optimizador es otro de los dos valores necesarios para compilar un modelo (junto con la función de pérdida). Los principales optimizadores disponibles son:

- **SGD:** aproximación estocástica del método de gradiente descendiente.

- **Adagrad:** algoritmo basado en gradiente que adapta el learning rate a los parámetros, realizando grandes actualizaciones cuando éstos son infrecuentes, y pequeñas actualizaciones cuando son frecuentes.
- **Adadelta:** extensión del algoritmo Adagrad
- **Adam:** algoritmo basado en el gradiente de primer orden de funciones estocásticas objetivas. Este método es sencillo de implementar, es eficiente computacionalmente hablando, necesita poca capacidad de memoria, y es muy bueno en problemas con una gran cantidad de datos y/o parámetros. Es utilizado por defecto en los modelos de Keras.

3.1.10. CALLBACKS

Los callbacks son una serie de funciones que pueden ser aplicadas en ciertos momentos del proceso de entrenamiento. Estos callbacks pueden ser utilizados para echar un vistazo a los estados internos, así como a las estadísticas del modelo que estamos entrenando. Para utilizarlos, basta con pasar como argumento a la función fit(...) una lista de métodos, que serán llamados en cada fase del entrenamiento. Las principales funciones son:

- BaseLogger(): Se aplica en todo modelo de Keras, y acumula la media de las métricas que se han definido para ser monitorizadas por ´época.
- Callback(): Clase base abstracta utilizada para crear nuevos callbacks. Como parámetros, recibe los propios para configurar el entrenamiento (número de ciclos, verbosidad...), así como la instancia del modelo que queremos entrenar.
- ProgbarLogger(): Callback que muestra métricas en la salida estándar (stdout).
- History(): Callback aplicado en cada modelo Keras que almacena eventos en un objeto de tipo History.
- ModelCheckpoint(...): Almacena el modelo después de cada ´época. Como argumentos recibe el fichero donde dejar la información, la cantidad de elementos a monitorizar, el modo de verbosidad, un booleano para decidir si sobrescribir o no el mejor modelo, el modo, y otro booleano para guardar solo pesos o información de todo el modelo.

3.1.11. PREPROCESADO

Keras cuenta con una serie de funciones para procesar tanto modelos secuenciales, como texto, como imágenes. La que se usa en este trabajo es ImageDataGenerator(...). Define la configuración para la preparación y el aumento de una imagen a la hora de ser procesada.

3.2. TENSORFLOW

Creado por el equipo de Google Brain, TensorFlow es una biblioteca de código abierto basada en Phyton para la computación numérica y Machine Learning a gran escala.
El propósito principal de TensorFlow es permitir que los ingenieros e investigadores manipulen expresiones matemáticas sobre tensores numéricos. Pero TensorFlow va mucho más allá del alcance de NumPy de las siguientes maneras:

- Puede calcular automáticamente el gradiente de cualquier expresión diferenciable, lo que lo hace muy adecuado para el aprendizaje automático.
- Puede ejecutarse no solo en CPU, sino también en GPU y TPU.
- La computación definida en TensorFlow se puede distribuir fácilmente entre muchas máquinas.
- Los programas de TensorFlow se pueden exportar a otros tiempos de ejecución, como C++, Java-Script (para aplicaciones basadas en navegador) o TensorFlow Lite (para aplicaciones que se ejecutan en dispositivos móviles o dispositivos integrados), etc. Esto hace que las aplicaciones de TensorFlow sean fáciles de implementar en ajustes prácticos.

TensorFlow es mucho más que una sola biblioteca. Es una plataforma con muchos componentes, algunos desarrollados por Google y otros desarrollados por terceros. Por ejemplo, hay TF-Agents para la investigación de reinforcement-learning, TensorFlow Serving para production deployment y también está el repositorio TensorFlow Hub de modelos preentrenados. Juntos, estos componentes cubren una amplia gama de aplicaciones, desde investigación de vanguardia hasta aplicaciones de producción a gran escala.

TensorFlow escala bastante bien: por ejemplo, Google ha utilizado Tensor-Flow para desarrollar aplicaciones de aprendizaje profundo muy intensivas en computación, como AlphaZero, el agente de juegos de ajedrez y Go.

3.3 KERAS Y TENSORFLOW: UNA BREVE HISTORIA

Keras es anterior a TensorFlow por ocho meses. Se lanzó en marzo de 2015 y TensorFlow se lanzó en noviembre de 2015. Si Keras se construye sobre TensorFlow, ¿cómo podría existir antes de que se lanzara TensorFlow? Keras se creó originalmente sobre Theano, otra biblioteca de manipulación de tensores que proporcionaba diferenciación automática y compatibilidad con GPU, la primera de su tipo. Theano, desarrollado en el Montréal Institute for Learning Algorithms (MILA) de la Université de Montréal, fue en muchos sentidos un precursor de TensorFlow. Fue pionero en la idea de usar gráficos de computación estáticos para la diferenciación automática y para compilar código tanto para CPU como para GPU.

A finales de 2015, después del lanzamiento de TensorFlow, Keras se refactorizó a una arquitectura multibackend: se hizo posible usar Keras con Theano o TensorFlow, y cambiar entre los dos fue tan fácil como cambiar una variable de entorno. Para septiembre de 2016, TensorFlow había alcanzado un nivel de madurez técnica en el que fue posible convertirlo en la opción de back-end predeterminada para Keras. En 2017, se agregaron

dos nuevas opciones de backend adicionales a Keras: CNTK (desarrollado por Microsoft) y MXNet (desarrollado por Amazon). Hoy en día, tanto Theano como CNTK están fuera de desarrollo y MXNet no se usa mucho fuera de Amazon. Keras ha vuelto a ser una API de back-end único, además de TensorFlow.

Keras y TensorFlow han tenido una relación simbiótica durante muchos años. A lo largo de 2016 y 2017, Keras se hizo conocido como la forma fácil de usar para desarrollar aplicaciones de TensorFlow, canalizando nuevos usuarios al ecosistema de TensorFlow. A fines de 2017, la mayoría de los usuarios de TensorFlow lo usaban a través de Keras o en combinación con Keras. En 2018, el liderazgo de TensorFlow eligió a Keras como la API oficial de alto nivel de TensorFlow. Como resultado, la API de Keras está al frente y al centro en TensorFlow 2.0, lanzado en septiembre de 2019, un amplio rediseño de TensorFlow y Keras que tiene en cuenta más de cuatro años de comentarios de los usuarios y progreso técnico.

3.4. COLAB

Todo este trabajo se ha codificado en su totalidad a través de colab.
Colab, también conocido como Colaboratory, es un entorno de programación que permite programar y ejecutar Python en el navegador sin necesitar ninguna configuración. Da acceso a GPUs sin coste y permite compartir el contenido de manera sencilla. Solo se necesita tener un usuario en Google.
https://colab.research.google.com/

CAPÍTULO 4. SISTEMA DE CLASIFICACIÓN USANDO REDES CONVOLUCIONALES

En este capítulo primero se va a explicar el código de ejemplo sobre el que se ha realizado el clasificador de imágenes. Después se explicará brevemente el cambio que se ha realizado sobre el código original para adaptarlo mejor a la realidad.Finalmente se explicarán las técnicas de Transfer Learning y Fine Tuning y como se han aplicado al modelo.

4.1. CÓDIGO INICIAL

En esta sección se describe y explica el diseño del código base para la implementación del reconocedor de imágenes de frutas [6, 7], a partir del cual se ha realizado el trabajo. A continuación se explica detalladamente su funcionamiento.

```
[ ]  !pip install wget
```

```
[ ]  import numpy as np
     import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras import layers
     import pandas as pd
     import seaborn as sns
     import matplotlib.pyplot as plt
     import wget
     import os

     from tensorflow.keras.utils import to_categorical
     from tensorflow.keras.preprocessing.image import load_img, img_to_array
     from tensorflow.keras.preprocessing.image import ImageDataGenerator
     from sklearn.metrics import classification_report, log_loss, accuracy_score
     from sklearn.model_selection import train_test_split
```

```
[ ]  directory = "fruits/train/train"
```

```
     !wget -N "https://cainvas-static.s3.amazonaws.com/media/user_data/AmrutaKoshe/fruits.zip"
     !unzip -qo fruits.zip
```

Figura 4.1. Código ejemplo 1

Primero se instala el paquete wget para poder descargar la base de datos de imágenes de internet y las librerías necesarias para que funcione el clasificador, después creamos un directorio donde se buscarán las imágenes para entrenar con directory y la ruta. Finalmente descargamos la base de datos con wget y la renombramos de manera que coincida con el directorio creado.

```
Name=[]
for file in os.listdir(directory):
    Name+=[file]
print(Name)
print(len(Name))
Name.sort()

fruit_map = dict(zip(Name, [t for t in range(len(Name))]))
print(fruit_map)
r_fruit_map=dict(zip([t for t in range(len(Name))],Name))
```

```
def mapper(value):
    return r_fruit_map[value]
```

```
[ ] img_datagen = ImageDataGenerator(rescale=1./255,
                                     vertical_flip=True,
                                     horizontal_flip=True,
                                     rotation_range=40,
                                     width_shift_range=0.2,
                                     height_shift_range=0.2,
                                     zoom_range=0.1,
                                     validation_split=0.2)

    test_datagen = ImageDataGenerator(rescale=1./255)
```

Figura 4.2. Código ejemplo 2

Se crea un vector vacío (Name) sobre el que se va guardando a través del bucle for y una variable FILE el nombre de cada tipo de fruta según esta ordenada en el directorio, donde están guardadas las imágenes en carpetas con el nombre de la clase.
Después se ordenan estos nombres por orden alfabético con Name.sort y creamos los diccionarios fruit_map y r_fruit_map para asociar los nombres de las frutas a números con dict.

A continuación, seguimos creando la función mapper, que a través de un número devuelve el tipo de fruta asociado guardado en r_fruit_map para comprobar que está bien ordenado. Por ejemplo, si Apple es la primera clase, al enviarle un 0 nos debería devolver el nombre Apple.

Continuamos reescalando las imagénes y ampliando la cantidad de datos añadiendo nuevas imágenes que son alteraciones de las que ya tenemos, ya sea giradas o con tamaños distintos. Esto se denomina *Data Augmentation*.

Con el parámetro validation_split elegimos el porcentaje de imágenes que se van a usar de validación y no en entrenamiento, para comprobar que la red se está entrenando bien.

```
test_datagen = ImageDataGenerator(rescale=1./255)
```

```
train_generator = img_datagen.flow_from_directory(directory,
                                                  shuffle=True,
                                                  batch_size=32,
                                                  subset='training',
                                                  target_size=(100, 100))

valid_generator = img_datagen.flow_from_directory(directory,
                                                  shuffle=True,
                                                  batch_size=16,
                                                  subset='validation',
                                                  target_size=(100, 100))
```

```
test_generator = img_datagen.flow_from_directory("fruits/train/train",
                                                  shuffle=True,
                                                  batch_size=16,
                                                  subset='validation',
                                                  target_size=(100, 100))
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Conv2D,MaxPooling2D,Dropout,Flatten,Activation,BatchNormalization
from tensorflow.keras.models import model_from_json
from tensorflow.keras.models import load_model
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(100,100,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))
```

Figura 4.3. Código ejemplo 3

Creamos lotes de datos que se usarán durante el entrenamiento para entrenar la IA y comprobar. Train_generator serán las imágenes de entrenamiento en lotes de 32, y Valid_generator y Test_generator serán imágenes de validación en lotes de 16.

Añadimos más librerías que se usarán en el modelo de la red neuronal y creamos el modelo, este modelo se ha hecho con 6 convoluciones con sus agrupaciones máximas correspondientes.

Después añadimos un clasificador compuesto por capa flatten para unidimensionar la red multidimensional y añadimos una capa dense con 256 neuronas y un dropout del 50%.

La capa final será otra capa densa de tantas neuronas como tipos de fruta tengamos, acompañada de una función de activación softmax.

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

```
[ ]  callbacks = [
         keras.callbacks.ModelCheckpoint(
             filepath="red_entrenada.keras",
             save_best_only=True,
             monitor="val_loss")
     ]
```

```
[ ]  history = model.fit(train_generator, validation_data=valid_generator,
                         steps_per_epoch=train_generator.n//train_generator.batch_size,
                         validation_steps=valid_generator.n//valid_generator.batch_size,
                         epochs=10,
                          callbacks=callbacks)
```

```
test_model = keras.models.load_model("red_entrenada.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
208/208 [==============================] - 26s 123ms/step - loss: 0.0303 - accuracy: 0.9919
Test accuracy: 0.992
```

Figura 4.4. Código ejemplo 4

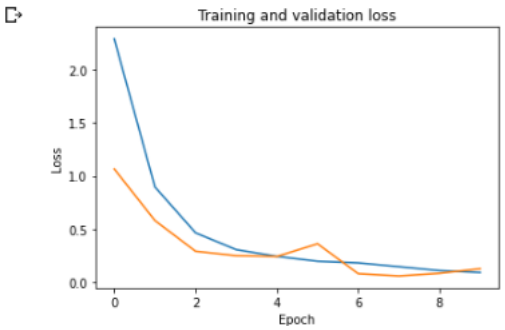
Compilamos el modelo con el optimizador Adam y la función de perdida categorical_crossentropy para las clases.

Uando callbacks hacemos que los datos de entrenamiento se guarden en "red_entrenada.keras" y comenzamos el entrenamiento con 10 épocas.

Finalmente se comprueba el % de acierto de la red cargando el callback y usando evaluate para evaluarlo. En este caso concreto la red alcanza un 99,2%.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and validation loss')
plt.show()

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and validation accuracy')
plt.show()
```



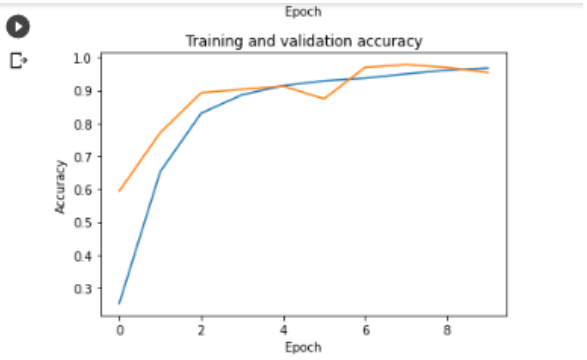


Figura 4.5. Código ejemplo 5

Podemos observar las gráficas para ver como ha ido mejorando el % de acierto y la pérdida de datos según los ciclos de entrenamiento. Para comprobar si la red funciona intentamos una predicción.

```
load_img("fruits/test/test/0120.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/0120.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
[ ]  prediction=model.predict(prediction_image)
     value=np.argmax(prediction)
     move_name=mapper(value)
     print("Prediction is {}.".format(move_name))

     Prediction is Pineapple.
```

Figura 4.6. Código ejemplo 6

Cargamos la imagen de una piña y hacemos la predicción, se puede observar que la acierta correctamente.

4.2. REDUCCIÓN DE IMÁGENES DE ENTRENAMIENTO

Esta red es muy avanzada y la base de datos tiene muchas imágenes de entrenamiento, más de 190000.
En la realidad no es normal tener tantas imágenes de entrenamiento puesto que todas estas imágenes deben ser hechas manualmente, se hizo un pequeño cambio que redujo la cantidad de imágenes para entrenar para que se adaptara más a la realidad.
El cambio consistió en cambiar el % de validation_split de 0,2 a 0,8. De este modo solo se entrenaría el modelo con un 20% de las imágenes totales, con esto bajamos la precisión al **89,5%**, también se redujo el tiempo de entrenamiento de 52 minutos a 12 minutos. Con este pequeño cambio ya hubo margen de mejora para aplicar técnicas de Transfer Learning y Fine Tuning para mejorar dicha precisión.

4.3. TRANSFER LEARNING Y FINE TUNING

Un enfoque común y efectivo para Deep Learning en conjuntos de datos de imágenes pequeños es usar un modelo previamente entrenado. Un modelo preentrenado es un modelo que se entrenó previamente en un gran conjunto de datos, generalmente en una tarea de clasificación de imágenes a gran escala. Si este conjunto de datos original es lo suficientemente grande y general, las aprendidas por el modelo preentrenado pueden actuar como un modelo genérico y, por lo tanto, sus características pueden resultar útiles para muchos problemas diferentes de visión por computador. aunque estos nuevos problemas pueden involucrar clases completamente diferentes a las de la tarea original. Por ejemplo, se puede entrenar un modelo en ImageNet y luego reutilizar este modelo entrenado para algo tan remoto como identificar muebles en imágenes. Esta portabilidad de las funciones aprendidas en diferentes problemas es una ventaja clave del Deep Learning en comparación con muchos enfoques de aprendizaje superficial más antiguos, y hace que el Deep Learning sea muy efectivo para problemas de datos pequeños.

Existen básicamente dos formas de utilizar un modelo preentrenado: Transfer Learning y Fine Tuning.

4.3.1. DEFINICIÓN DE TRANSFER LEARNING

El Transfer Learning consiste en utilizar las representaciones aprendidas por un modelo previamente entrenado para extraer características interesantes de nuevas muestras. Estas características luego se ejecutan a través de un nuevo clasificador, que se entrena desde cero.

Las redes convolucionales utilizadas para la clasificación de imágenes constan de dos partes: comienzan con una serie de capas de agrupación y convolución, y terminan con un clasificador denso conectado. La primera parte se llama *convolutional base* del modelo. En el caso de las redes convolucionales, el Transfer Learning consiste en tomar la base convolucional de una red previamente entrenada, ejecutar los nuevos datos a través de ella y entrenar un nuevo clasificador encima de la salida.
Se debe evitar reutilizar los clasificadores densos. La razón es que las representaciones aprendidas por la base convolucional probablemente sean más genéricas y, por lo tanto, más reutilizables: los mapas de características de una red convolucional son mapas de presencia de conceptos genéricos sobre una imagen, que probablemente sean útiles. independientemente del problema de visión por computadora en cuestión. Pero las representaciones aprendidas por el clasificador serán específicas para el conjunto de clases en las que se entrenó el modelo: solo contendrán información sobre la probabilidad de las clases del modelo original.
Si su nuevo conjunto de datos difiere mucho del conjunto de datos en el que se entrenó el modelo original, es mejor que se use solo las primeras capas del modelo para realizar Transfer Learning, en lugar de usar toda la base convolucional.

Su forma más habitual de empleo sigue 4 fases:

1. Se añaden capas de otro modelo ya entrenado al modelo donde se quiere emplear.

2. Se congelan estas capas preentrenadas.
3. Se añaden capas nuevas despúes de las que están congeladas, que serán capaces de adaptar los datos ya entrenados a una nueva base de datos. El clasificador denso conectado.
4. Se entrenan estas nuevas capas.

Cuando hablamos de congelar una capa, nos referimos a que esta capa deja de poder entrenarse durante los ciclos de entrenamiento, asi no alterará los pesos que ya ha aprendido previamente.

Con este método estamos sobreentrenando el modelo casi desde el principio a pesar de usar dropout con una tasa bastante grande. Esto es porque este método no
usa Data Augmentation, que es esencial para evitar el sobreentrenamiento con bases de datos de imágenes pequeñas,

4.3.2. DEFINICION DE FINE TUNING

Otra técnica que se usa para la reutilización de modelos preentrenados, complementario al Transfer Learning, es el Fine Tuning.
El Fine Tuning consiste en descongelar algunas de las capas superiores de la base congelada utilizada, y entrenar conjuntamente tanto la parte nueva del modelo y estas capas superiores. Se llama Fine Tuning porque ajusta ligeramente las representaciones más abstractas del modelo siendo reutilizadas para que sean más relevantes para nuestro problema.
Solo es posible ajustar las capas superiores de la base convolucional una vez que el clasificador en la parte superior ya ha sido entrenado. Si el clasificador aún no está entrenado, la señal de error que se propagará a través de la red durante el entrenamiento será demasiado grande, y las representaciones previamente aprendidas se perderán. Por lo tanto, los pasos para ajustar una red son los siguientes:

1. Agregar nuestra red encima de una base ya entrenada.
2. Congelar la base.
3. Entrenar la parte nueva.
4. Descongele algunas capas en la red base.
5. Entrenar estas capas y la parte que agregamos, pero con un learning rate muy bajo.

4.4. APLICANDO TRANSFER LEARNING Y FINE TUNING AL MODELO

Veremos como, de manera sencilla, hemos aplicado los métodos a nuestro modelo.

4.4.1. BASE DE DATOS IMAGENET

Imagenet es una base de datos pública que contiene 14 millones de imágenes en 21 mil categorías diferentes.
En 2007, una investigadora llamada Fei-Fei Li empezó a trabajar en la idea de crear un conjunto de datos de este tipo. Los datos fueron recogidos y etiquetados desde la web por humanos. Por lo tanto, es de código abierto y no pertenece a ninguna empresa en particular.

4.4.2. RED VGG16

VGG es una red neuronal convolucional propuesta por K. Simonyan y A. Zisserman en 2014, de la Universidad de Oxford, y adquirió notoriedad al ganar el Desafío de Reconocimiento Visual a Gran Escala de ImageNet (ILSVRC) en 2014. El modelo alcanzó una precisión del 92,7% en Imagenet. Supone una mejora respecto a los modelos anteriores al proponer núcleos de convolución más pequeños (3×3) en las capas de convolución de lo que se había hecho anteriormente.

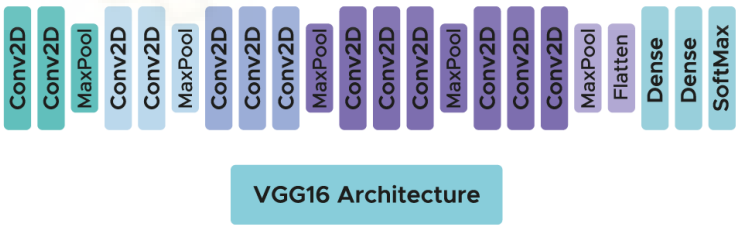


Figura 4.7. Arquitectura VGG16

El modelo VGG16 sólo requiere un pre-procesamiento específico que consiste en restar a cada píxel el valor RGB medio, calculado en el conjunto de entrenamiento.
Durante el entrenamiento del modelo, el input a la primera capa de convolución es una imagen RGB de tamaño 224 x 224. Para todas las capas de convolución, el núcleo de convolución es de tamaño 3×3.

Estas capas de convolución van acompañadas de capas Max-Pooling, cada una de ellas de tamaño 2×2, para reducir el tamaño de los filtros durante el entrenamiento.
A la salida de las capas de convolución y agrupación, tenemos 3 capas de neuronas totalmente conectadas. Las dos primeras están compuestas por 4096 neuronas y la última

por 1000 neuronas con una función de activación softmax para determinar la clase de imagen.

Esta red tiene una serie de características óptimas para ser una red preentrenada para transfer learning:

- Tiene una arquitectura fácil de comprender e implementar.
- Consigue buenos resultados cuando se usa con Imagenet, entre el 96% y 97%.
- Contienen relativamente pocas capas convolucionales: 13 capas convolucionales y 3 densas, por eso se llama VGG16.
- La red está disponible en Keras.

4.4.3. APLICANDO TRANSFER LEARNING

```
[23] from keras.applications.vgg16 import VGG16

     vgg_model = VGG16(input_shape=[100,100,3], weights='imagenet', include_top=False)
```

```
[24] for layer in vgg_model.layers:
        layer.trainable = False
```

```
[25] vgg_model.summary()
```

```
transfer_learning_model_vgg = Sequential()

transfer_learning_model_vgg.add(vgg_model)

transfer_learning_model_vgg.add(Flatten())
transfer_learning_model_vgg.add(Dense(256))
transfer_learning_model_vgg.add(Activation('relu'))
transfer_learning_model_vgg.add(Dropout(0.5))
transfer_learning_model_vgg.add(Dense(len(fruit_map)))
transfer_learning_model_vgg.add(Activation('softmax'))
```

```
from tensorflow.keras.optimizers import SGD, Adam, RMSprop

optimizer = Adam()
transfer_learning_model_vgg.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

Figura 4.8. Transfer learning 1

Primero importamos la red VGG16 y le pasamos 3 argumentos:

- **Weights:** especifica el punto de control de peso desde el cual inicializar el modelo.
- **Include_top:** se refiere a incluir o no el clasificador denso situado en la parte superior de la red. De forma predeterminada, este clasificador denso conectado corresponde a las clases de ImageNet. Debido a que pretendemos usar nuestro propio clasificador no necesitamos incluirlo
- **Input_shape:** es la forma de los tensores de imagen que enviaremos a la red. Este argumento es puramente opcional: si no lo pasamos, la red podrá procesar entradas de cualquier tamaño.

Obligamos a que no se entrener las capas de su base convolucional a través del bucle for y trainable = False.
Creamos un nuevo modelo y le añadimos nuestra red y despúes la red VGG congelada. Como antes, añadimos el clasificador.
Importamos librerías y compilamos el modelo con el optimizador Adam.

```
callbacks_2 = [
    keras.callbacks.ModelCheckpoint(
        filepath="2_red_LT_VGG.keras",
        save_best_only=True,
        monitor="val_loss")
]

inicio = time.time()

history_2 = transfer_learning_model_vgg.fit(train_generator, validation_data=valid_generator,
                  steps_per_epoch=train_generator.n//train_generator.batch_size,
                  validation_steps=valid_generator.n//valid_generator.batch_size,
                  epochs=10,
                   callbacks = callbacks_2)

fin = time.time()
print(fin-inicio)
```

```
test_model = keras.models.load_model("2_red_LT_VGG.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
831/831 [==============================] - 65s 78ms/step - loss: 0.2170 - accuracy: 0.9501
Test accuracy: 0.950
```

Figura 4.9. Transfer learning 2

Creamos un nuevo callback y creamos la variable time para ver cuánto dura el entrenamiento.
Finalmente vemos que hemos conseguido aumentar la precisión al 95% solo con Transfer Learning.
Esto es sólo una pequeña mejora en comparación con la precisión de la prueba anterior.
La precisión de un modelo depende del conjunto de muestras que lo evalúe.
Algunas bases de datos pueden ser más difíciles de entrenar que otras, y buenos resultados en un conjunto no necesariamente será igual en todos los demás conjuntos.

4.4.4. APLICANDO FINE TUNING

```
transfer_learning_model_vgg.trainable = True

transfer_learning_model_vgg.compile(loss='categorical_crossentropy',
                                    optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),metrics=['accuracy'])

callbacks_8 = [
    keras.callbacks.ModelCheckpoint(
        filepath="8_red_FT_VGG16_afinada.keras",
        save_best_only=True,
        monitor="val_loss")
]

historia = transfer_learning_model_vgg.fit(train_generator, validation_data=valid_generator,
                  steps_per_epoch=train_generator.n//train_generator.batch_size,
                  validation_steps=valid_generator.n//valid_generator.batch_size,
                  epochs=10, callbacks = callbacks_8)
```

```
test_model = keras.models.load_model("8_red_FT_VGG16_afinada.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
831/831 [==============================] - 44s 53ms/step - loss: 0.0281 - accuracy: 0.9930
Test accuracy: 0.993
```

Figura 4.10. Fine Tuning

Primero descongelamos las capas del modelo y lo compilamos de nuevo, pero con un learning rate de 1e-4. Creamos el callback y entrenamos.
Finalmente podemos observar que la precisión ha conseguido subir a más del 99% desde el 89,5% original a través de estos dos métodos.
Al aprovechar estas técnicas de Deep Learning, hemos logrado llegar a un resultado similar al original usando solo una pequeña fracción de los datos de entrenamiento que estaban disponibles, con un tiempo de computación casi 5 veces inferior, de 50 a 12 minutos.

CAPÍTULO 5. EXPERIMENTOS REALIZADOS Y ANÁLISIS DE RESULTADOS

En este capítulo, se comentarán algunas pruebas que se realizaron sobre el código original, sin reducción de imágenes, para intentar optimizar el entrenamiento que tardaba unos 52 minutos en ejecutarse.
Despúes se comentarán pruebas que se hicieron sobre el modelo convolucional, también sin la reducción de imágenes.
Finalmente se comentara como se usaron otras redes preentrenadas aparte de VGG16 en Transfer Learning y Fine Tuning para observar si se obtenían mejores resultados.

5.1. PRUEBAS SOBRE DATA AUGMENTATION

5.1.1. ELIMINACIÓN DEL DATA AUGMENTATION

En esta prueba se le dijo al programa que no generara nuevas imágenes a partir de las que ya teníamos anulando los parámetros salvo el % de imágenes de validadción.

```
img_datagen = ImageDataGenerator(rescale=1./255,
                                 vertical_flip=False,
                                 horizontal_flip=False,
                                 rotation_range=0,
                                 width_shift_range=0,
                                 height_shift_range=0,
                                 zoom_range=0,
                                 validation_split=0.2)
```

Figura 5.1. Anulación de la Data Augmentation

Se obtuvo una precisión similar (99,8%) y se redujo el tiempo de ejecución a 43 minutos, pero esta precisión no era real. Con este cambio el modelo solo aprendió a identificar imágenes con la misma disposición a las que usó durante el entrenamiento, si tuviera que predecir una imagen al revés, este no sabría identificarla bien.

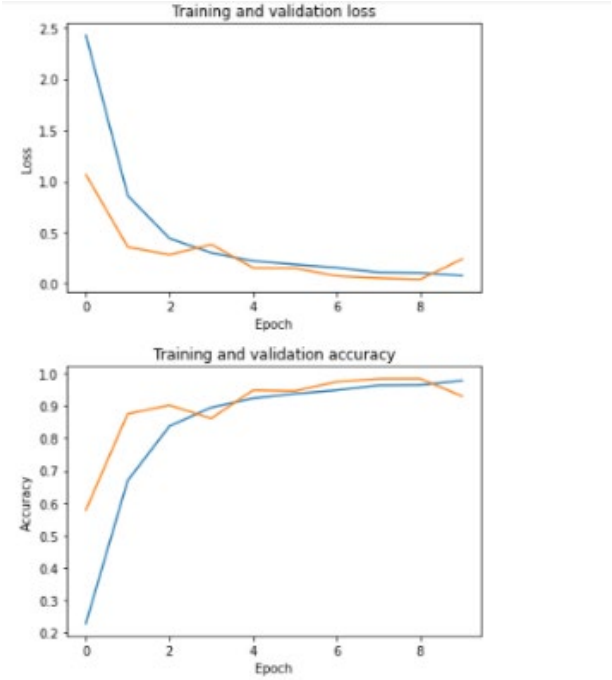


Figura 5.2. Gráfico de la anulación de la data augmentation

5.1.2. MODIFICACIÓN DE LOS ÁNGULO DE ROTACIÓN

En esta prueba se cambió el ángulo de rotación a 20º y 60º desde el 40º original, para observar cómo respondía el modelo al tener imágenes en disposiciones diferentes a las originales.

```
img_datagen = ImageDataGenerator(rescale=1./255,
                                 vertical_flip=True,
                                 horizontal_flip=True,
                                 rotation_range=20,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range=0.1,
                                 validation_split=0.2)
```

```
img_datagen = ImageDataGenerator(rescale=1./255,
                                 vertical_flip=True,
                                 horizontal_flip=True,
                                 rotation_range=60,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range=0.1,
                                 validation_split=0.2)
```

Figura 5.3. Izquierda: parámetros de rotación de 20º. Derecha: parámetros de rotación de 60º

Este experimento obtuvo menor precisión., pero se pudo confirmar igualmente que cuantas más imágenes generadas tengamos mejor precisión obtendremos, ya que al ir generando una imagen cada 20º obtuvimos un 97,2%% de acierto mientras que haciéndolo cada 60º solo un 95,27%. También se observó que hubo mayor % de pérdida de datos al tener menos imágenes. Este experimento no varió el tiempo de ejecución en ninguno de los dos casos.

Rotation range 20º	**Rotation range 60º**
Precisión 97’2 %	Precisión 95’27 %

Tabla 5.1. Comparativa ángulo de rotación

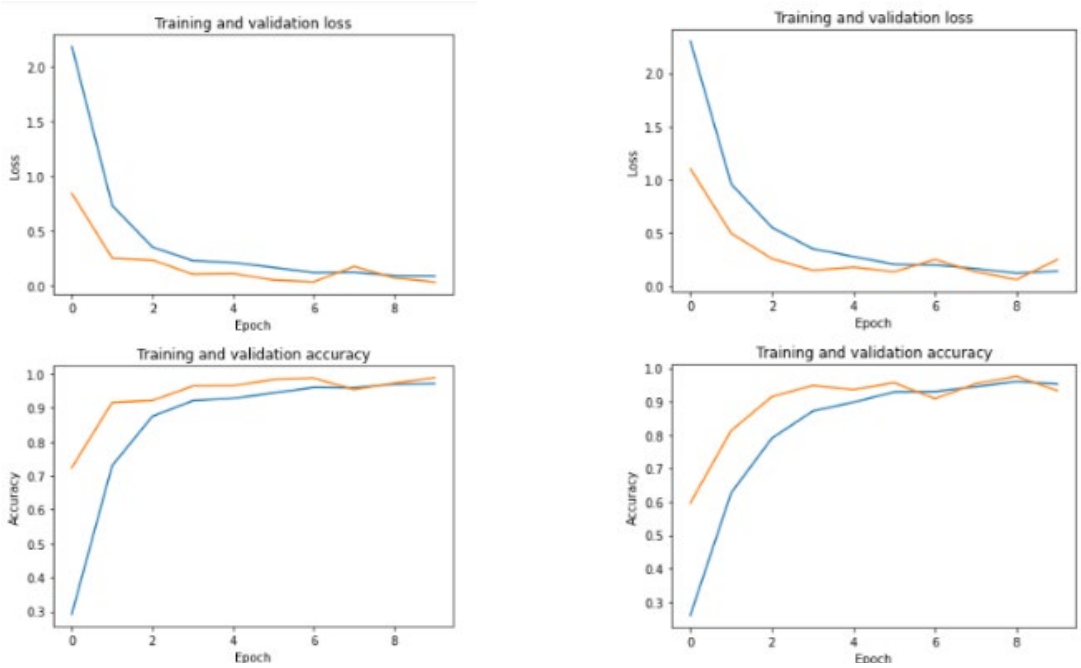


Figura 5.4. Izquierda: gráfico de rotación de 20º. Derecha: gráfico de rotación de 60º

5.1.3. MODIFICACIÓN DEL DESPLAZAMIENTO DE LAS IMÁGENES

En este experimento, se aumento el rango que desplazaba la imagen original a 0.4 desde el 0.2 original, hacia los lados o arriba y abajo, antes de generar la nueva imagen.

```
img_datagen = ImageDataGenerator(rescale=1./255,
                                 vertical_flip=True,
                                 horizontal_flip=True,
                                 rotation_range=40,
                                 width_shift_range=0.4,
                                 height_shift_range=0.4,
                                 zoom_range=0.1,
                                 validation_split=0.2)
```

Figura 5.5. Variación de shifts.

Este experimento redujo la precisión al 95,45% como se esperaba al tener menos imágenes. También obtuvo una mayor pérdida de datos que el modelo original, así que se concluyó que a menor cantidad de imágenes más datos se perdieron. No varió el tiempo de ejecución.

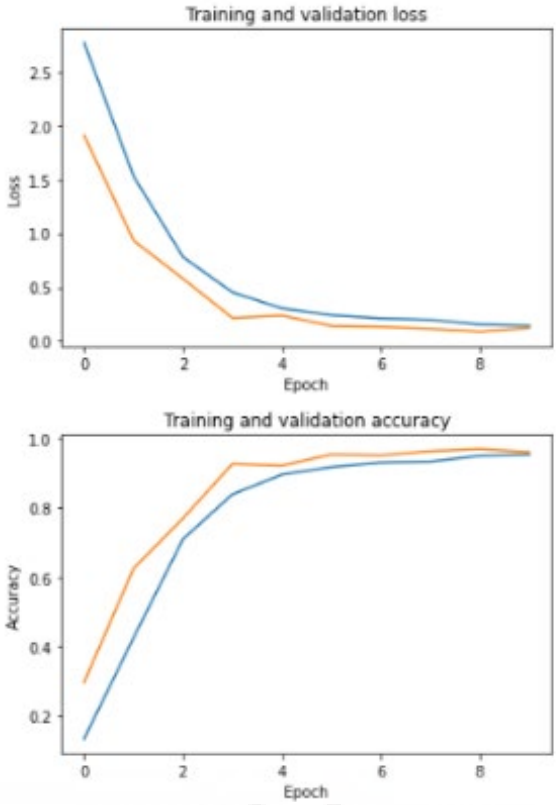


Figura 5.6. Gráfico de la variación de shifts.

5.1.4. MODIFICACIÓN DEL ZOOM

En este experimento se incremento la cantidad de zoom en las nuevas imágenes generadas a 0.3 desde 0.1.

```
img_datagen = ImageDataGenerator(rescale=1./255,
                                 vertical_flip=True,
                                 horizontal_flip=True,
                                 rotation_range=40,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range=0.3,
                                 validation_split=0.2)
```

Figura 5.7. Variación de zoom

En este experimento la precisión descendió al 95,72% pero se consiguió reducir el tiempo de entrenamiento a 48 minutos, también se incrementó la perdida de datos. Con esto se concluyó que el modelo ejecuta imágenes de mayor tamaño ligeramente más deprisa.

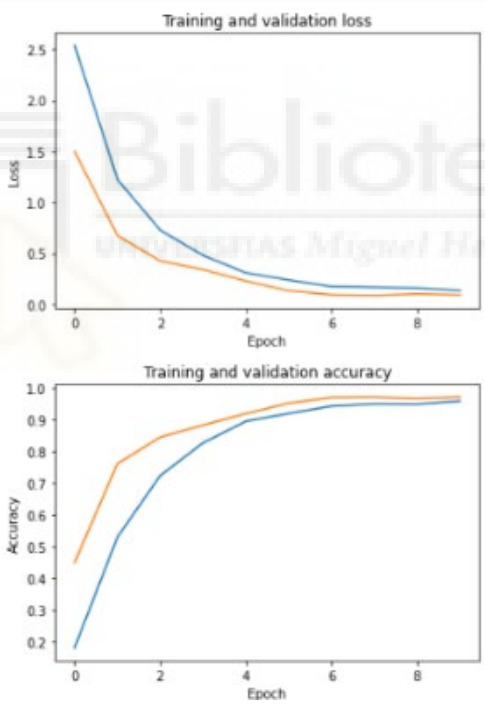


Figura 5.8. Gráfica de la variación de zoom

5.2. PRUEBAS SOBRE EL MODELO CONVOLUCIONAL

5.2.1. VARIACIÓN DEL NÚMERO DE FILTROS APLICADOS

Con esta prueba se quiso estudiar como evolucionaba el modelo dependiendo de si aplicaba los filtros más o menos veces durante su ejecución.

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(100,100,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```

```
model = Sequential()
model.add(Conv2D(filters=64, kernel_size=(3,3),input_shape=(100,100,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```


Figura 5.9. Arriba: Modelo de 32 filtros. Abajo: Modelo de 64 filtros

Al finalizar el experimento se observó que en ambos casos la precisión era muy similar, siendo ligeramente superior a menor número de filtros. También se comprobó que el

número de filtros influye más en el tiempo de ejecución que la data augmentation, ya que teniendo 32 filtros se redujo el tiempo a 38 minutos y con 64 se incrementó a 68 minutos.

Filters a 32	**Filters a 64**
Precisión 95’37 %	Precisión 95’1 %

Tabla 5.2. Comparativa variación filtros

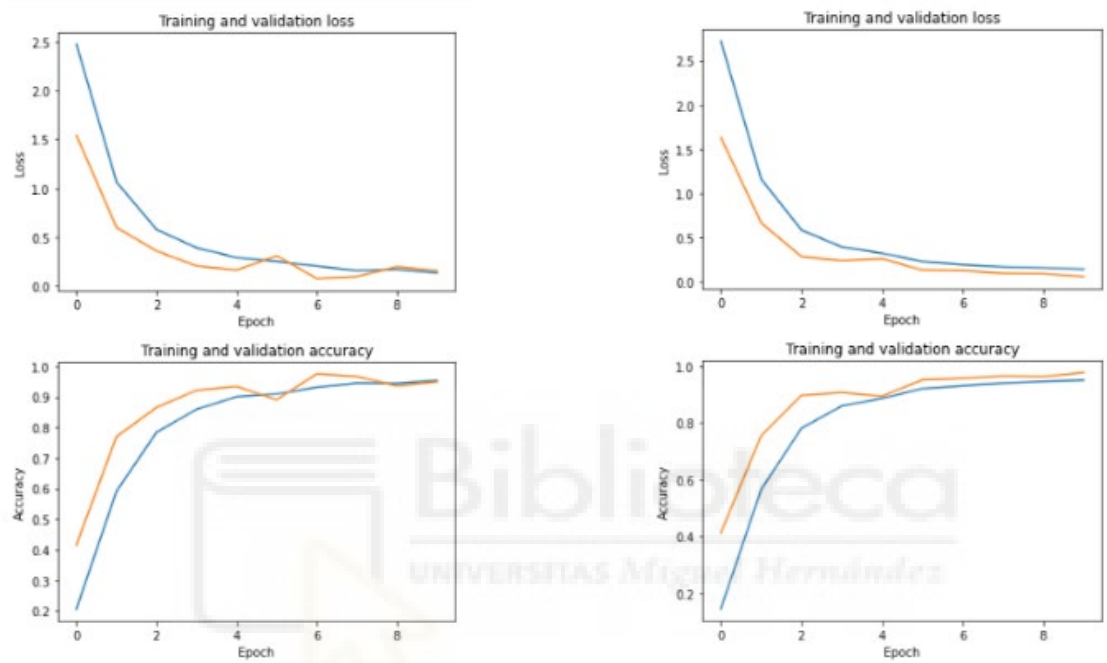


Figura 5.10. Izquierda: Gráfica de modelo de 32 filtros. Derecha: Gráfica de modelo de 64 filtros

5.2.2. VARIACIÓN DE LA DIMENSION DE LOS KERNEL DE CONVOLUCIÓN

En este experimento se quiso comprobar como reaccionaba el modelo cuando los kernels de las capas convolucionales tenían dimensiones distintas.

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5),input_shape=(100,100,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(5,5), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(5,5), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(5,5), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(5,5), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(5,5), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```



```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(1,1),input_shape=(100,100,3), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(1,1), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(1,1), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(1,1), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(1,1), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(1,1), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```

Figura 5.11. Arriba: Modelo de kernel 5x5. Abajo: Modelo de kernel 1x1

Con este experimento se observó que cuanto más pequeña es la dimensión de un kernel más precisión se pierde y hay menor tiempo de ejecución, 28 minutos con 1x1. Con kernels más grandes se pierde menos precisión, pero se incrementa mucho el tiempo de ejecución, hasta 98 minutos con 5x5.

Kernels a 5x5	Kernels a 1x1
Precisión 93’5 %	Precisión 87%

Tabla 5.3. Comparativa dimensión kernels

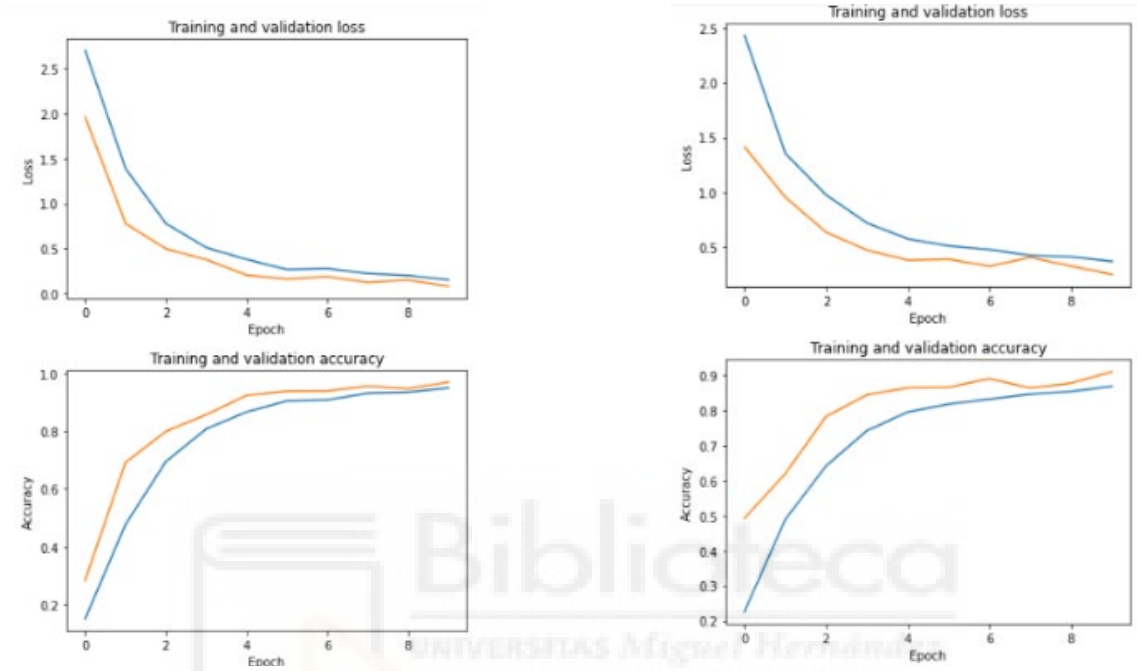


Figura 5.12. Izquierda: Gráfica de modelo de kernel 5x5. Derecha: Gráfica de modelo de kernel 1x1

5.2.3. USO DE OTRAS FUNCIONES DE ACTIVACIÓN

Se probaron dos funciones de activación diferentes a relu para ver la reacción del modelo.

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(100,100,3), activation='sigmoid', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='sigmoid', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='sigmoid', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='sigmoid', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='sigmoid', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='sigmoid', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```



```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(100,100,3), activation='tanh', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='tanh', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='tanh', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='tanh', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='tanh', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='tanh', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```

Figura 5.13. Arriba: Activación sigmoid. Abajo: Activación tanh

Utilizando la función Sigmoid se obtuvo muy poca precisión (73,86%), sin embargo , con la función de activación tanh se obtuvo resultado bastante bueno (96,31%).

Activación Sigmoid	Activación Tanh
Precisión 73’86%	Precisión 96’31 %

Tabla 5.4. Comparativa funciones de activación

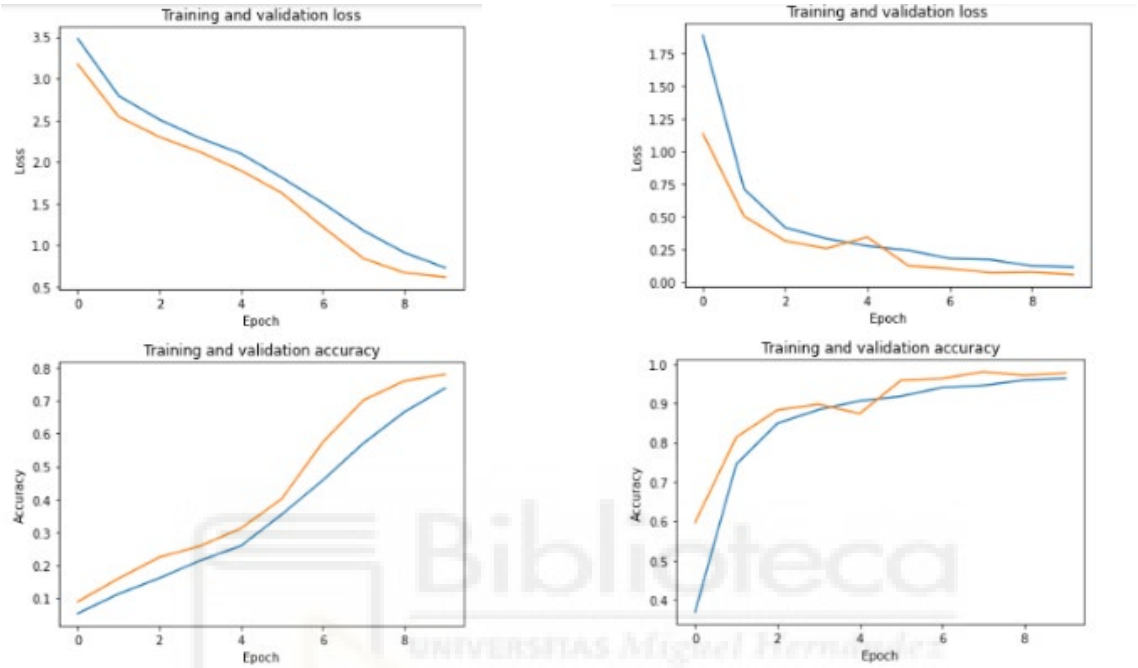


Figura 5.14. Izquierda: Gráfica activación sigmoid. Derecha: Gráfica activación tanh

5.2.4. CAMBIO DEL TIPO DE PADDING

El modelo está construido con el tipo de padding "same", este parón opera sobre todos los números de la matriz de pixeles independientemente de su posición. Se quiso cambiar el padding al tipo "valid", este padding solo realiza cálculos dentro de los bordes de la matriz. Es decir, si el kernel es 3x3 y la matriz 10x10, solo se calcularía sobre los pixeles dentro del 8x8, para que ni el kernel ni el número coincidan con los bordes.

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3),input_shape=(100,100,3), activation='relu', padding = 'valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding = 'valid'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(len(fruit_map)))
model.add(Activation('softmax'))

model.summary()
```


Figura 5.15. Padding Valid

Este experimento redujo el tiempo de entrenamiento a 41 minutos, pero se obtuvo una reducción drástica de la precisión al 12%.

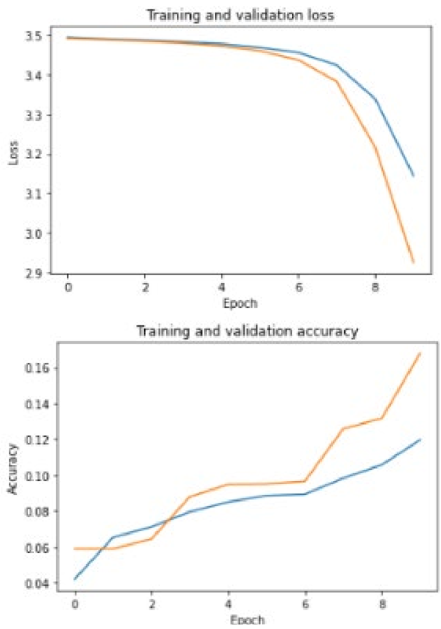


Figura 5.16. Gráfica de padding Valid

5.3. PRUEBAS CON INCEPTION V_3 Y XCEPTION

Veremos ahora que resultados se obtuvieron al cambiar el tipo de red preentrenada que se usó en el modelo. Usando Xcpetion e Inception V_3 para Transfer Learning y Fine Tuning

5.3.1. INCEPTION V_3

Al contrario que la arquitectura con la que hemos trabajado hasta ahora, en la que hemos apilado capas convolucionales, Inception es resultado de un importante trabajo de ingeniería para lograr capas más anchas y menos profundas, introduciendo en una misma capa varios operadores de convolución y luego concatenando sus resultados:

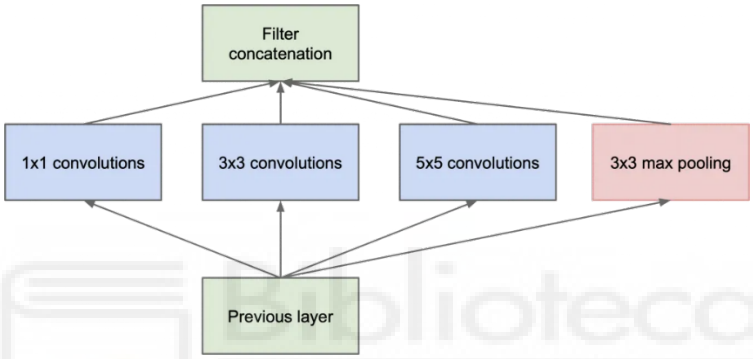


Figura 5.17. Arquitectura Inception

De este modo, se consiguen modelos que son capaces de tener un alto poder predictivo, pero más eficientes que redes más profundas. Este modelo ha sido revisado en varias ocasiones con algunas mejoras, dando lugar a diferentes versiones de la arquitectura.

La versión Inception V_3 está disponible en TensorFlow y esta preentrenada con Imagenet. Tiene una arquitectura compleja, de 48 capas:

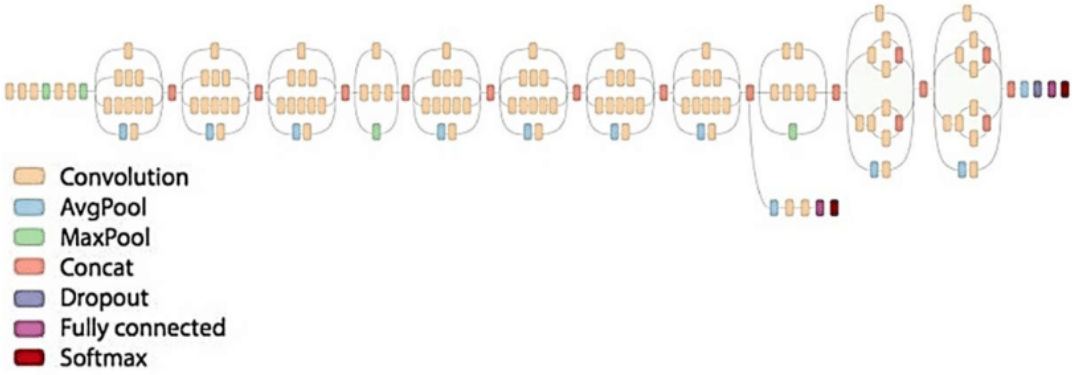


Figura 5.18. Arquitectura Inception V_3

5.3.1.1. TRANSFER LEARNING CON INCEPTION V_3

Primero veremos como cambia el modelo con Transfer Learning y Inception V_3.

```
from tensorflow.keras.applications.inception_v3 import InceptionV3
```

```
conv_base_Inception = InceptionV3(input_shape=[100,100,3],weights = 'imagenet',
                  include_top = False)
conv_base_Inception.trainable = False
```

```
transfer_learning_model_Inc = Sequential()

transfer_learning_model_Inc.add(conv_base_Inception)

transfer_learning_model_Inc.add(Flatten())
transfer_learning_model_Inc.add(Dense(256))
transfer_learning_model_Inc.add(Activation('relu'))
transfer_learning_model_Inc.add(Dropout(0.5))
transfer_learning_model_Inc.add(Dense(len(fruit_map)))
transfer_learning_model_Inc.add(Activation('softmax'))
```

```
from tensorflow.keras.optimizers import SGD, Adam, RMSprop

optimizer = Adam()
transfer_learning_model_Inc.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

Figura 5.19. Código Inception 1

Primero cargamos Inception y congelamos las capas de su base.
Después construimos un nuevo modelo con nuestra red, la red preentrenada y el clasificador.
Luego compilamos todo el modelo con el optimizador Adam y la función de pérdida categorical_crossentropy.

```
callbacks_3 = [
    keras.callbacks.ModelCheckpoint(
        filepath="3_red_LT_INC.keras",
        save_best_only=True,
        monitor="val_loss")
]

inicio = time.time()

history_3 = transfer_learning_model_Inc.fit(train_generator, validation_data=valid_generator,
                  steps_per_epoch=train_generator.n//train_generator.batch_size,
                  validation_steps=valid_generator.n//valid_generator.batch_size,
                  epochs=10,
                   callbacks = callbacks_3)

fin = time.time()
print(fin-inicio)
```

```
test_model = keras.models.load_model("3_red_LT_INC.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
831/831 [==============================] - 48s 57ms/step - loss: 0.2587 - accuracy: 0.9202
Test accuracy: 0.920
```

Figura 5.20. Código Inception 2

Elegimos el callback y entrenamos. Podemos comprobar que la precision aumenta de 89,5% a 92%, un poco menos que usando VGG16, que llegaba al 95%.

5.3.1.2. FINE TUNING CON INCEPTION V_3

Vamos a comprobar ahora cómo funciona el Fine Tuning con esta red.

```
conv_base_Inception.trainable = True

from tensorflow.keras.optimizers import SGD
transfer_learning_model_Inc.compile(optimizer=SGD(learning_rate=0.0001, momentum=0.9), loss='categorical_crossentropy',metrics=['accuracy'])

callbacks_6 = [
    keras.callbacks.ModelCheckpoint(
        filepath="6_red_FT_INC_afinada.keras",
        save_best_only=True,
        monitor="val_loss")
]

historia_FT_INC_afinada = transfer_learning_model_Inc.fit(train_generator, validation_data=valid_generator,
                  steps_per_epoch=train_generator.n//train_generator.batch_size,
                  validation_steps=valid_generator.n//valid_generator.batch_size,
                  epochs=10, callbacks = callbacks_6)
```

```
test_model = keras.models.load_model("6_red_FT_INC_afinada.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
831/831 [==============================] - 49s 57ms/step - loss: 0.2663 - accuracy: 0.9573
Test accuracy: 0.957
```

Figura 5.21. Código Inception 3

Descongelamos las capas de la red preentrenada y compilamos de nuevo el modelo, esta vez con el optimizador SGD que acabamos de importar. Le aplicamos un learning rate bajo, del 0,0001. Elegimos el callback y entrenamos.
Con esta red aumentamos la precisión casi al 96%. Sigue siendo menor que VGG16 que llegaba al 99,3%.

5.3.2. XCEPTION

Una de las arquitecturas de redes de Deep Learning que ha demostrado funcionar bastante bien para el reconocimiento de imágenes es Xception. Fue presentada por Google en la conferencia más importante sobre visión por computador y reconocimiento de patrones del 2017. Se inspira en la arquitectura Inception. Xception, introduce un elemento novedoso en la arquitectura: las convoluciones separables en profundidad. Este operador realiza una consolación de forma separada en cada canal de la entrada, y luego aplica una convolución 1×1 para proyectar los canales generados como salida en un nuevo espacio de canales. La arquitectura total resultante tiene esta forma:

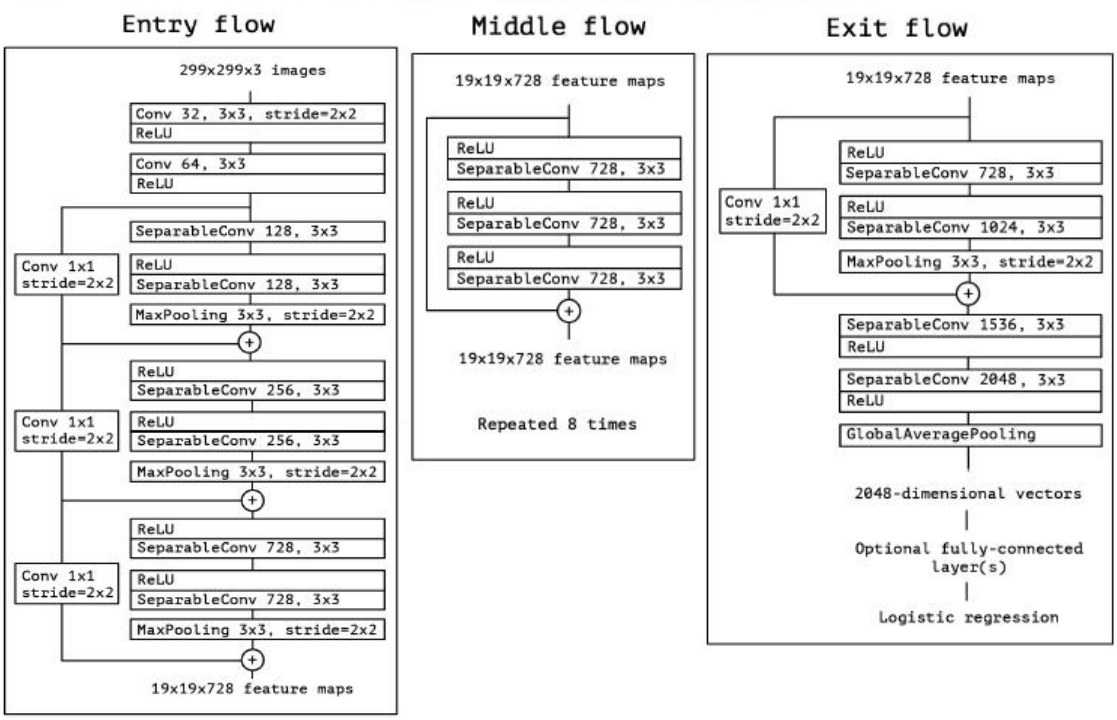


Figura 5.22. Arquitectura Xception

Esta arquitectura obtuvo muy buenos resultados en ImageNet.
Fue entrenada con más de un millón de imágenes durante tres días, y luego evaluada sobre 50.000 imágenes. El resultado fue una tasa de aciertos del 79%, que ascendía al 94,5% cuando se consideraba el *top-5 accuracy*, donde al modelo se le dan cinco oportunidades para acertar la clase.
Keras nos permite cargar la arquitectura directamente, y además empleando los parámetros que se aprendieron cuando se entrenó con ImageNet.

5.3.2.1. TRANSFER LEARNING CON XCEPTION

Aplicamos Transfer Learning con la red Xception.

```
conv_base_Xception = keras.applications.Xception(
    weights='imagenet',
    input_shape=(100, 100, 3),
    include_top=False)

conv_base_Xception.trainable = False
```

```
transfer_learning_model_Xception = Sequential()

transfer_learning_model_Xception.add(conv_base_Xception)

transfer_learning_model_Xception.add(Flatten())
transfer_learning_model_Xception.add(Dense(256))
transfer_learning_model_Xception.add(Activation('relu'))
transfer_learning_model_Xception.add(Dropout(0.5))
transfer_learning_model_Xception.add(Dense(len(fruit_map)))
transfer_learning_model_Xception.add(Activation('softmax'))
```

```
from tensorflow.keras.optimizers import SGD, Adam, RMSprop
optimizer = Adam()
transfer_learning_model_Xception.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

Figura 5.23. Código Xception 1

El proceso es igual al anterior, cargar la base de datos, congelar capas, crear el nuevo modelo y compilar.

```
callbacks_4 = [
    keras.callbacks.ModelCheckpoint(
        filepath="4_red_LT_Xc.keras",
        save_best_only=True,
        monitor="val_loss")
]

inicio = time.time()

history_4 = transfer_learning_model_Xception.fit(train_generator, validation_data=valid_generator,
                  steps_per_epoch=train_generator.n//train_generator.batch_size,
                  validation_steps=valid_generator.n//valid_generator.batch_size,
                  epochs=10,
                   callbacks = callbacks_4)

fin = time.time()
print(fin-inicio)
```

```
test_model = keras.models.load_model("4_red_LT_Xc.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
831/831 [==============================] - 45s 53ms/step - loss: 0.1152 - accuracy: 0.9616
Test accuracy: 0.962
```

Figura 5.24. Código Xception 2

Después de realizar el entrenamiento la precisión alcanza el 96,2%, ligeramente más que con Inception.

5.3.2.2. FINE TUNING CON XCEPTION

```
conv_base_Xception.trainable = True

transfer_learning_model_Xception.compile(loss='categorical_crossentropy',
                                         optimizer=keras.optimizers.RMSprop(learning_rate=1e-5),metrics=['accuracy'])

callbacks_9 = [
    keras.callbacks.ModelCheckpoint(
        filepath="10_red_FT_Xception_afinada.keras",
        save_best_only=True,
        monitor="val_loss")
]

historia = transfer_learning_model_Xception.fit(train_generator, validation_data=valid_generator,
                  steps_per_epoch=train_generator.n//train_generator.batch_size,
                  validation_steps=valid_generator.n//valid_generator.batch_size,
                  epochs=10, callbacks = callbacks_9)
```

```
test_model = keras.models.load_model("10_red_FT_Xception_afinada.keras")
test_loss, test_acc = test_model.evaluate(test_generator)
print(f"Test accuracy: {test_acc:.3f}")
```

```
831/831 [==============================] - 46s 54ms/step - loss: 0.0715 - accuracy: 0.9822
Test accuracy: 0.982
```

Figura 5.25. Código Xception 3

Descongelamos las capas de Xception, compilamos con el optimizador RMSprop y entrenamos el modelo.
Obtenemos un 98,2 % de acierto, muy similar al % obtenido con VGG16.

CAPÍTULO 6. CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se resumen los resultados obtenidos a lo largo de la investigación, primero centrándonos en el Data Augmentation para después centrarnos en la parte de Transfer Learning. Además, se expondrán posibles ampliaciones o experimentos con este trabajo que se pudieran realizar en un futuro.

6.1. CONCLUSIONES RESPECTO A LA CONSTRUCCIÓN DE UN MODELO DE RED NEURONAL

Como se ha visto en los experimentos realizados, existen muchas maneras de obtener más datos de entrenamiento a partir de los que ya se tiene usando el método de *Data Augmentation* (aumento de datos) y sus variaciones. Así, siempre que se tenga un número reducido de imágenes de entrenamiento, es posible la generación de nuevas imágenes a través de la alteración de las mismas. Evidentemente, esto redunda en la mayoría de casos en una mejora del entrenamiento de las redes.

También se ha observado que, en redes convolucionales, cuanto mayor es el número de filtros o de mayor tamaño sean los kernels, más aumentará la precisión, pero esto va a tener un impacto muy grande en el tiempo de computación de cualquier red neuronal.

6.2. CONCLUSIONES RESPECTO REDES PREENTRENADAS

También se ha comprobado que las redes preentrenadas con ImageNet suelen dar muy buenos resultados, en muchos casos alcanzando prácticamente el 100% de acierto al usarse con las técnicas de *Transfer Learning* (transferencia de aprendizaje) y *Fine Tuning* (sintonizado fino).

6.3. CONCLUSIONES GLOBALES

Después de haber realizado este trabajo, se puede llegar a la conclusión de que no es necesario tener muchos datos de entrenamiento para realizar determinadas aplicaciones de inteligencia artificial industrial o de cualquier tipo si se aplican convenientemente técnicas de aumento de datos, transferencia del aprendizaje o de sintonizado fino. En el trabajo se ha comprobado que, incluso cuando las técnicas de aumento de datos no son suficientes para conseguir la precisión deseada, es posible recurrir a técnicas de transferencia del aprendizaje para aprovechar el conocimiento guardado en otras redes. Esto ahorra tiempo de trabajo, ya que elimina la necesidad de recabar más datos para el entrenamiento. Además, elimina tiempo de computación ya que hay capas que no se entrenarán y por lo tanto no consumirán tiempo de ejecución.

6.4. POSIBLES AMPLIACIONES

Este trabajo se ha realizado usando solo unas pocas redes neuronales. Hay muchos tipos distintos de redes con las que no se ha experimentado, sobre todo las que no fueron entrenadas en ImageNet. Un posible trabajo futuro puede será la realización de estudios y análisis con redes que fueron testeadas en otras bases de datos y que podría dar lugar a resultados muy diferentes.

Del mismo modo se pueden testear el uso de más de una red preentrenada durante el proceso, una para la transferencia del aprendizaje y otra para el sintonizado fino, y observar qué resultados se obtienen al aprender de dos redes neuronales diferentes.

Otra posibilidad de ampliación es la exploración del uso de la red implementada como punto de partida para la realización de trabajos de simulación 3D y visión artificial, a través de los cuales diseñar un proceso industrial funcional de clasificación de frutas.

BIBLIOGRAFÍA

[1] Artículo sobre la Inteligencia Artificial: https://www.sas.com/es_es/insights/analytics/what-is-artificial-intelligence.html

[2] Artículo sobre Deep Learning: https://www.sas.com/es_es/insights/analytics/deep-learning.html

[3] Artículo sobre Transfer Learning: https://www.tokioschool.com/noticias/transfer-learning/

[4] Lista de reproducción sobre IAs con phyton en el canal Ringa Tech de youtube: https://www.youtube.com/watch?v=iX_on3VxZzk&list=PLZ8REt5zt2Pn0vfJjTAPaDVSACDvnuGiG

[5] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola (2021). Dive Into Deep Learning.

[6] François Chollet (2020). Deep Learning con Phyton. Capítulo 8.

[7] A. Koshe (2021). *Fruits Classification using Deep Learning*. AI Technology & Systems (enlace: https://medium.com/ai-techsystems/fruits-classification-using-deep-learning-f8261b0ee0ca)

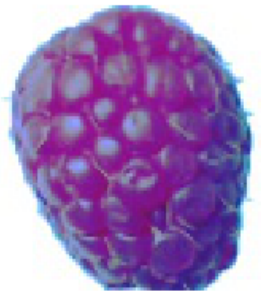
[8] https://datascientest.com/es/vgg-que-es-este-modelo-daniel-te-lo-cuenta-todo

[9] https://blog.noaxacademy.com/deep-learning-sobre-hombros-de-gigantes/

ANEXO. EJEMPLOS DE IMÁGENES Y PREDICCIONES

En este anexo se muestran otras predicciones de la red neuronal donde aparecen otras clases de frutas que se han usado en el proyecto.

```
load_img("fruits/test/test/0010.jpg",target_size=(180,180))
```



```
image=load_img("fruits/test/test/0010.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)


prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Raspberry.
```

Anexo 1. Frambuesa

```
load_img("fruits/test/test/0013.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/0013.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Avocado.
```

Anexo 2. Aguacate

```
load_img("fruits/test/test/0005.jpg",target_size=(180,180))
```


```
image=load_img("fruits/test/test/0005.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Watermelon.
```

Anexo 3. Sandía

```
load_img("fruits/test/test/0095.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/0095.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Grape Blue.
```

Anexo 4. Uva

```
load_img("fruits/test/test/1009.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/1009.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Papaya.
```

Anexo 5. Papaya

```
load_img("fruits/test/test/2705.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/2705.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Pepper Green.
```

Anexo 6. Pimiento verde

```
load_img("fruits/test/test/0056.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/0056.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Pepper Red.
```

Anexo 7. Pimiento rojo

```
load_img("fruits/test/test/5639.jpg",target_size=(180,180))
```

```
image=load_img("fruits/test/test/5639.jpg",target_size=(100,100))

image=img_to_array(image)
image=image/255.0
prediction_image=np.array(image)
prediction_image= np.expand_dims(image, axis=0)
```

```
prediction=model.predict(prediction_image)
value=np.argmax(prediction)
move_name=mapper(value)
print("Prediction is {}.".format(move_name))
```

```
Prediction is Apricot.
```

Anexo 8. Albaricoque