Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aprendizaje automático para la identificación de razas caninas

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Marcos Rodríguez Loriente

Tutor: Carlos David Martínez Hinarejos

Curso 2021-2022

Resumen

Dada la imagen de un perro, se pretenden utilizar técnicas de Aprendizaje Automático para determinar su raza entre un conjunto de 120 razas totales. Para ello se dispone de un *dataset* con un total de 10222 imágenes etiquetadas que actuarán como conjunto de entrenamiento para nuestro modelo y disponemos también de un conjunto de 10357 imágenes no etiquetadas que pretendemos clasificar. El objetivo del trabajo será implementar un clasificador con el mínimo error posible empleando para ello diferentes técnicas de Aprendizaje Automático, como Máquinas de Vector Soporte y Redes Neuronales. Para poder implementar dicho clasificador se recurrirá al uso de librerías Python como Keras y Tensorflow.

Palabras clave: Inteligencia Artificial, Aprendizaje Automático, Redes Neuronales, Máquinas de Vector Soporte, Clasificación de imágenes.

Abstract

Given the image of a dog, it is intended to use Machine Learning techniques to determine its breed among a set of 120 total breeds. For this, we have a dataset with a total of 10222 labeled images that will act as a training set for our model and we also have a set of 10357 unlabeled images that we intend to classify. The objective of the work is to implement a classifier with the minimum possible error, using different Machine Learning techniques such as Support Vector Machines and Neural Networks. In order to implement this classifier we will use some Python specialized libraries such as Keras and Tensorflow.

Keywords: Artificial Intelligence, Machine Learning, Neural Networks, Support Vector Machines, Image classification.

Resum

A partir de la imatge d´un gos, tractarem d´ identificar la seva raça entre un conjunt total de 120 races possibles. Per a realizar aquesta labor es disposa d´un *dataset* amb un total de 10222 imatges etiquetades amb la raça corresponent que actuaran com a conjunt d´entrenament per al nostre model i a la vegada disposarem també d´un conjunt de 10357 imatges no etiquetades que tractarem de clasificar. L´objectiu del treball serà implementar un classificador amb un mínim error possible utilitzant per a aquest fi diferents mètodes d´Aprenentatge Automàtic com Màquines de Vectors Support i Xarxes Neuronals. Per a poder implementar aquest classificador s´utilitzaran llibreries especialitzades de Python com Keras i Tensorflow.

Paraules clau: Intel·ligència Artificial, Aprenentatge Automàtic, Xarxes Neuronals, Màquines de Vectors Support, Classificació d´imatges.

Tabla de contenidos

Capítulo 1 Introducción

Fue en el verano del 1956, en la conferencia de Dartmouth (Hanover, New Hampshire, EEUU) donde escuchamos por primera vez el término Inteligencia Artificial (IA). Dicha conferencia reunió a un conjunto de 10 investigadores (Minsky, McCarthy, Newell, Simon, Samuel, Rochester, Shannon, Solomonoff, Selfridge, More) que definieron las directrices y las líneas de actuación futuras para el desarrollo de la IA mediante la proposición de la siguiente hipótesis: “todo aspecto de aprendizaje o cualquier otra característica de inteligencia puede ser definido de forma tan precisa que puede construirse en una máquina para simularlo” [1].

A partir de esta idea surge un entonces revolucionario sistema inteligente conocido como Logic Theorist. Se trataba de un programa que podía probar teoremas en lógica simbólica de la *Principia Mathematica* de Whitehead y Russell (un conjunto de tres libros con las bases de la matemática).

Desde entonces se siguen investigando nuevas utilidades de la IA y avanzando cada vez más en el desarrollo de tecnologías inteligentes. Actualmente, la IA está integrada en casi todos los ámbitos de nuestra sociedad, realizando labores de reconocimiento y detección (como, por ejemplo, el detector facial de nuestro *smartphone* o el lector de huellas del mismo), labores de predicción y ayuda (como, por ejemplo, la ayuda a la detección del cáncer mediante el reconocimiento de tumores malignos) [7], etc.

La principal herramienta utilizada para dichas labores de detección, predicción y reconocimiento es el Aprendizaje Automático o *Machine Learning* en inglés.

El Aprendizaje Automático es un conjunto de algoritmos y técnicas utilizados para diseñar sistemas que aprenden de los datos que reciben como entrada.

1.1 Motivación

Una de las motivaciones principales de este documento es profundizar en los diferentes aspectos y modalidades del aprendizaje automático. Concretamente se pretende realizar un estudio de las principales metodologías empleadas en el ámbito de la clasificación de imágenes.

En adición, se emplearán los resultados de este estudio para implementar un sistema automático capaz de identificar la raza de nuestra mascota. Dado que el animal de compañía por excelencia (tal y como lo indica su nombre) es el *canis familiaris*, vulgarmente conocido como perro, nuestro sistema se enfocará en la detección de la raza de este animal. Con ello no solo se pretende ayudar a los propietarios de dicha mascota, sino que también resulta de gran interés para todo aquel que busque adoptar uno de estos animales, ya que la raza no influye solo en el aspecto físico, sino que también influye en el comportamiento y carácter del perro en cuestión.

Además, se espera que la creación de dicho clasificador sea de gran utilidad para quien consulte este documento con la finalidad de resolver problemas de clasificación similares; por ello se detallarán en todo momento los pasos realizados en el proceso de implementación del mismo.

1.2 Objetivos

El objetivo principal de este proyecto es la creación de un sistema automático que reciba como *input* una imagen de un perro y que nos devuelva como *output* su correspondiente raza, determinada a su vez por un modelo de Aprendizaje Automático que se construirá a partir de un conjunto o *dataset* de imágenes clasificadas de perros de distintas razas (120 en total).

Dicho objetivo principal se dividirá a su vez en múltiples objetivos secuenciales o etapas de menor dimensión:

1. Estudio y comprensión de nuestro problema, examinando tanto la varianza como el formato como la calidad de nuestro *dataset* de imágenes de entrada.

2. Estudio de las técnicas de Aprendizaje Automático empleadas para la clasificación de imágenes.

3. Implementación y comparación de las técnicas examinadas con anterioridad con el fin de crear un clasificador lo más efectivo posible.

1.3 Impacto esperado

Tras lograr alcanzar el objetivo principal se espera quien use el sistema desarrollado sea capaz de identificar la raza de cualquier perro, que entienda tanto el funcionamiento de dicho sistema como la construcción del clasificador que lo caracteriza y que pueda realizar tareas similares para problemas de clasificación de imágenes, utilizando las metodologías más efectivas para ello.

1.4 Estructura

La estructura de este documento se divide en 7 capítulos:

Capítulo 1. Introducción: Es el apartado en que nos encontramos y que nos introducirá brevemente el tema que vamos a tratar, la estructura de este documento, los objetivos y las finalidades por las cuales se escribe el mismo.

Capítulo 2. Estado del arte: Apartado en el que trataremos de identificar las metodologías empleadas actualmente en problemas de clasificación de imágenes.

Capítulo 3. Fundamentos teóricos: Apartado que nos servirá para explicar de manera teórica las técnicas utilizadas en la construcción del clasificador.

Capítulo 4. Materiales y métodos: Apartado donde explicaremos todo el entorno de programación utilizado, así como sus principales librerías y sus métodos.

Capítulo 5. Experimentación: En dicho apartado abordaremos los aspectos relativos a la programación de las diferentes metodologías vistas con anterioridad de manera teórica, y posteriormente se realizará un análisis de la efectividad de cada una de ellas.

Capítulo 6. Conclusiones: En este apartado presentaremos las conclusiones obtenidas de este trabajo.

Bibliografía: Último apartado del trabajo, donde se indicarán las referencias de interés que han sido utilizadas para redactar este documento.

1.5 Convenciones

A continuación se describirán todas las normativas de marcado empleadas en este documento, así como su correspondiente significado:

Comillas simples – Se utilizarán siempre que se haga una cita a un texto de otro autor, indicando con posterioridad y entre corchetes el índice de dicho texto dentro de la bibliografía.

Cursiva – Se empleará siempre que se utilice un término proveniente de la lengua inglesa.

Courier New, Cursiva – Se utilizará para indicar las partes del texto que pertenezcan al código de nuestro clasificador, además de emplearse al citar los métodos de las librerías empleadas en Python.

Capítulo 2 Estado del arte

Nuestro caso concreto a resolver es la clasificación de la raza de un perro entre un total de 120 razas posibles, es decir, tenemos entre manos un problema de clasificación de imágenes en varias clases.

En este capítulo se investigarán los métodos empleados en la actualidad para resolver este tipo de problemas, así como la precisión obtenida por los mismos.

En el artículo [8] se detalla la efectividad de los modelos de *deep learning* empleados en problemas de clasificación de especies. El objetivo de dicho estudio es el de implementar un clasificador de imágenes que ayude a identificar diferentes especies de animales que han sido fotografiadas en la naturaleza mediante cámaras de monitoreo. Para ello se cuenta con un total de 4 *datasets* de imágenes de animales de diferentes especies (todos ellos incluyen imágenes donde no hay ningún animal): Snapshot Serengeti (SS), Camera CATalogue (CC), Elephant Expedition (EE) y Snapshot Wisconsin (SW). Se pretende comparar la efectividad que nos ofrecen los modelos basados en el uso de *transfer learning* [3] frente a la efectividad que nos ofrecen los modelos basados en el entrenamiento de redes neuronales convolucionales (CNN)[3] entrenadas desde cero.

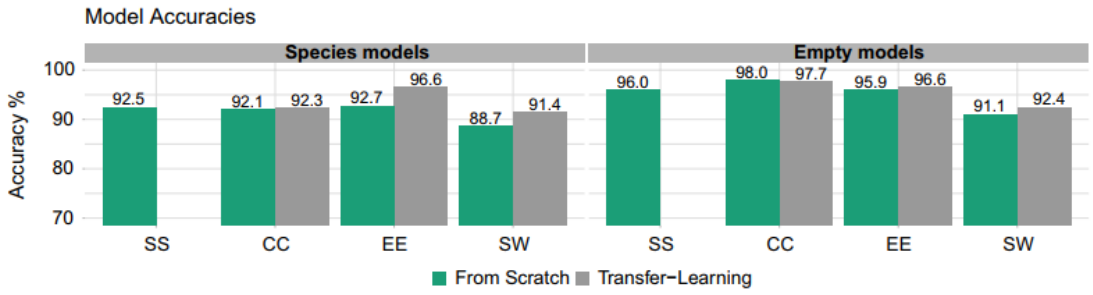


Imagen 1. Comparativa de la efectividad del entrenamiento de CNN desde cero (verde) frente al entrenamiento de un modelo basado en *transfer learning* (gris) [8]

En la imagen 1 se muestran los resultados de este estudio, comparando la precisión de ambas metodologías en los diferentes *datasets* (el *dataset* SS se utiliza para realizar la técnica de *transfer learning*, por ello no tenemos su precisión en la gráfica). Se puede observar como los modelos basados en *transfer learning* nos ofrecen una mayor precisión tanto a la hora de detectar imágenes donde no hay ningún animal, como a la hora de detectar la especie del animal que aparece en la imagen. En esta misma imagen

Además, también existen artículos que abordan directamente el problema de clasificación de imágenes de perros, como es el caso del artículo [10]. En dicho documento se estudian diferentes formas de implementar un clasificador de razas de perros, con un conjunto de 8350 imágenes y un total de 133 razas distintas. El clasificador se compone de dos capas o estructuras:

- Una primera capa encargada de extraer las características más importantes de cada imagen, que nos permitan identificar la raza del perro de manera correcta.

- Una segunda capa o estructura que cumple el papel de clasificador de las imágenes a partir de las características extraídas en la capa anterior.

Dicho artículo trata de descubrir cuáles son las técnicas y algoritmos de extracción de características y de clasificación que mayor efectividad ofrecen en la tarea en cuestión. Primero se estudia la efectividad que ofrecen varias de las técnicas de clasificación empleadas, como máquinas de vectores soporte (SVM)[4], algoritmo de los k-vecinos[13], algoritmo de *Bag of Words*[11], etc. Se utiliza para ello descriptores SIFT[12] como método de extracción de características clave en las imágenes de los perros; se tratan de detectar aspectos como los ojos, las orejas o la nariz, tal y como se muestra en la imagen 2.

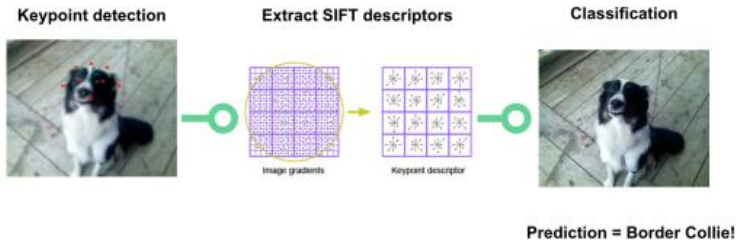


Imagen 2. Detección de características clave mediante descriptores SIFT [10].

Una vez estudiada la efectividad de los clasificadores, se estudia la efectividad de los algoritmos de extracción de características en las imágenes. Principalmente se estudian los descriptores SIFT mencionados con anterioridad, junto con otros algoritmos como los de creación de histogramas de color de cada imagen.

Los resultados de este estudio se muestran en la imagen 3.

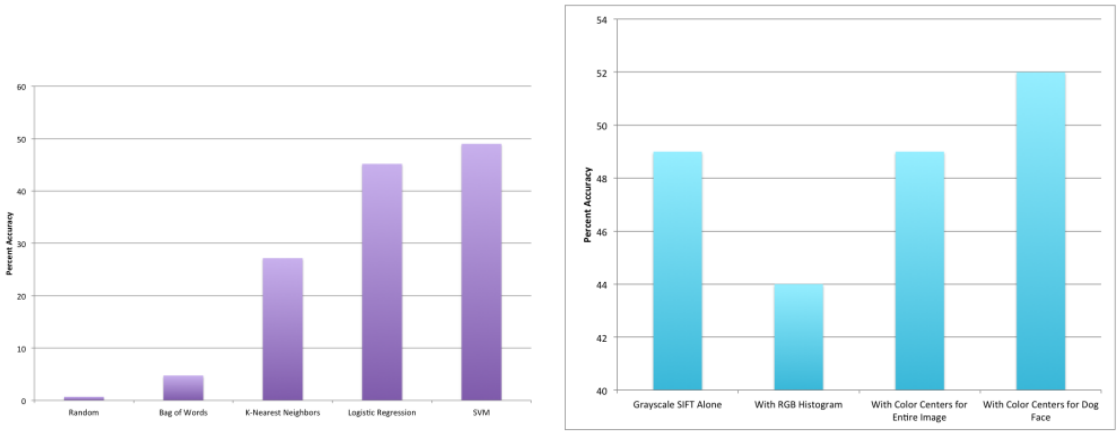


Imagen 3. Comparación de mecanismos de extracción de características y de clasificación de imágenes [10].

Como se observa en esta imagen, se logra alcanzar una precisión máxima cercana al 50%, utilizando máquinas de vectores soporte para clasificar los descriptores SIFT extraídos por cada imagen.

Capítulo 3 Fundamentos teóricos

Como bien se ha explicado con anterioridad, el Aprendizaje Automático o *Machine Learning* es un conjunto de algoritmos y técnicas empleados para el diseño de sistemas que aprenden de los propios datos que reciben. Estos sistemas son capaces de realizar predicciones o deducciones de patrones de los datos suministrados.

Algunos de los principales tipos de problema resueltos mediante *Machine Learning* son:

- Problemas de clasificación. Se trata de identificar a qué categoría o clase pertenece la nueva observación en función del conjunto de datos de entrenamiento que contiene dichas categorías. Si dicha clasificación solo emplea dos clases (como, por ejemplo, predecir si un tumor es canceroso o no), decimos que es de tipo binaria.
- Problemas de regresión. A diferencia de la clasificación, la regresión se utiliza para predecir el futuro mediante las relaciones entre las variables numéricas de entrada (capaces de representar conjuntos numéricos potencialmente infinitos), dando como resultado un valor decimal. Un ejemplo de dicho tipo de problema es la predicción de temperaturas para la semana siguiente.
- Problemas de agrupamiento o *clustering*. En el caso de los problemas de *clustering* tratamos de descubrir cómo se organiza el conjunto de datos dado mediante la agrupación de dichos datos. Un ejemplo de dicho tipo de problema es la agrupación de un conjunto de espectadores atendiendo al género de películas que les gusta.

Además, los algoritmos de *Machine Learning* empleados se pueden subdividir en dos grupos diferentes:

- Algoritmos de aprendizaje supervisado. En el aprendizaje supervisado se usan *datasets* etiquetados, es decir, se usan datos con una etiqueta que proporciona un significado a cada dato como puede ser, por ejemplo, un *dataset* de imágenes cuyas etiquetas sean 1 (gato) o 0 (perro). Usaremos estos datos etiquetados para predecir una nueva etiqueta en las imágenes sin etiquetar. La mayoría de los algoritmos utilizados en el aprendizaje automático son supervisados.
- Algoritmos de aprendizaje no supervisado. En el caso del aprendizaje no supervisado el conjunto de datos que nos sirve para entrenar nuestro modelo de aprendizaje automático no está etiquetado. En estos casos el objetivo es el de encontrar relaciones entre dichos datos; se trata de problemas de *clustering*.

En las siguientes secciones se pasará a explicar los principales algoritmos supervisados utilizados para la clasificación de imágenes.

3.1 Máquinas de vector soporte (SVM)

Las máquinas de vector soporte, o *Support Vector Machines* (SVM) en inglés, son un tipo de algoritmo supervisado empleado en problemas de clasificación [2]. La idea principal consiste en establecer una recta (caso bidimensional) que separe las diferentes clases de la mejor forma posible, tal y como se observa en la imagen 4.

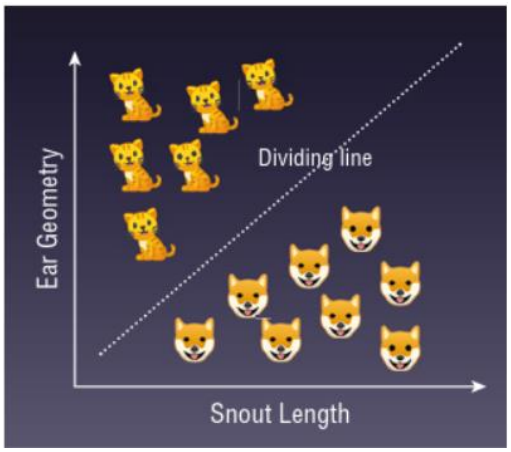


Imagen 4. Ejemplo de uso de SVM para la clasificación de muestras que pertenecen a perros o gatos en función de características faciales de ambos grupos, como la geometría de las orejas y la longitud del hocico.

Una vez hemos definido dicha recta, también conocida como hiperplano de decisión (usamos el término hiperplano porque habitualmente se trata con más de dos dimensiones), la podemos utilizar para predecir datos. En el ejemplo de la imagen 4 podremos determinar si una futura muestra pertenece a la clase de los perros o a la clase de los gatos fijándonos en la geometría de las orejas y la longitud del hocico que presente la misma.

Sin embargo, existen múltiples hiperplanos que cumplen con la capacidad de separar linealmente las muestras, con lo que se plantea el problema de cuál elegir.

La forma correcta de elegir el hiperplano guarda relación con la amplitud de los márgenes entre las clases: el hiperplano correcto será aquel que presente los márgenes con mayor amplitud (con cada uno de los márgenes tocando al menos a un punto de cada clase). Así pues, para el ejemplo indicado en la imagen 5, la frontera de decisión correcta encargada de separar las dos clases indicadas será la frontera de color verde, con un margen d2 > d1 (siempre y cuando d2 sea el mayor margen posible).

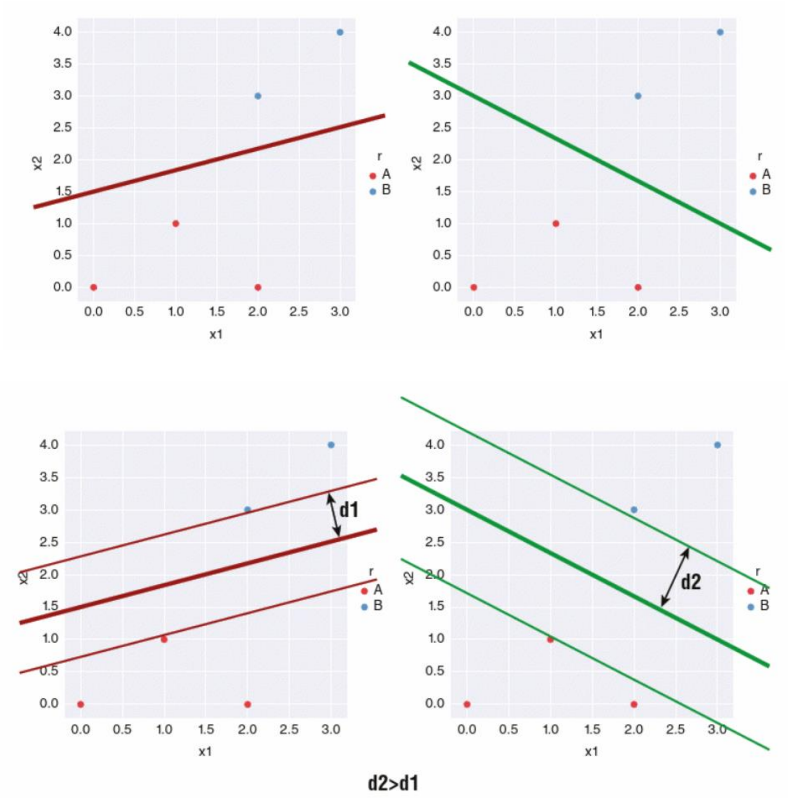


Imagen 5. Ejemplo de selección de hiperplano correcto.

Un término clave en los SVM son los vectores soporte. Los vectores soporte son los puntos o muestras de cada clase que se sitúan sobre los márgenes. Como ejemplo en la imagen 5 tenemos dos vectores soporte, uno situado en el margen de la clase A y otro que se encuentra sobre el margen de la clase B.

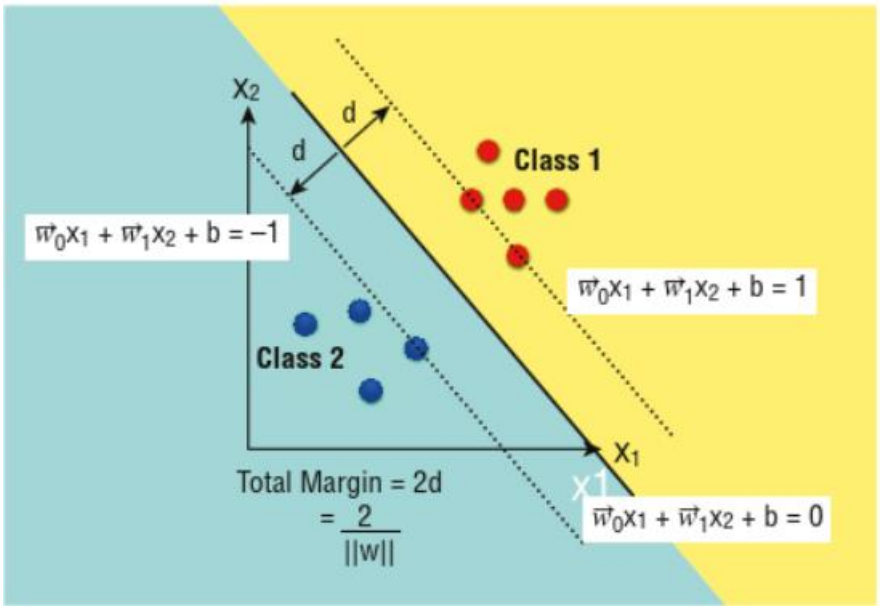


Imagen 6. Fórmula para el hiperplano y los correspondientes márgenes en un espacio bidimensional.

Tal y como se indica en la imagen 6, la fórmula del hiperplano en un espacio bidimensional viene dada por la expresión $G(x) = \overrightarrow{\omega_0}x_1 + \overrightarrow{\omega_1}x_2 + b$, donde el vector

$x = (x_1, x_2)$ representa los *inputs* del modelo, $\overrightarrow{\omega_1}$ y $\overrightarrow{\omega_0}$ son los vectores de peso y el término *b* es conocido como *bias*. Si el valor de la función G es >= 0, entonces la muestra especificada pertenecerá a la clase 1; en cambio, si G < 0 la muestra pertenecerá a la clase 2. Por otro lado, tal y como hemos mencionado con anterioridad, el objetivo del SVM es el de encontrar los márgenes más amplios que sean capaces de dividir las clases; el margen total lo definimos mediante la expresión $\frac{2}{||w||}$, donde $||w||$ es la normalización de los vectores $\overrightarrow{\omega_1}$ y $\overrightarrow{\omega_0}$.

A partir del *dataset* de entrenamiento el objetivo es minimizar el valor de $||w||$ con el fin de obtener la máxima separabilidad entre clases. Una vez hecho esto, seremos capaces de obtener los valores de $\overrightarrow{\omega_0}, \overrightarrow{\omega_1}$ y *b*.

Encontrar el margen es un problema de optimización que se resuelve aplicando la técnica de los multiplicadores de Lagrange [4].

3.1.1 Funciones kernel

Las funciones kernel (K) se utilizan para encontrar el hiperplano de separación óptimo en el caso de que los datos de entrada de nuestro modelo SVM sean linealmente no separables. En dicho caso, la función tratará de convertir el espacio de los datos de entrada en un espacio con mayor dimensionalidad, tal y como se observa en la imagen 7.

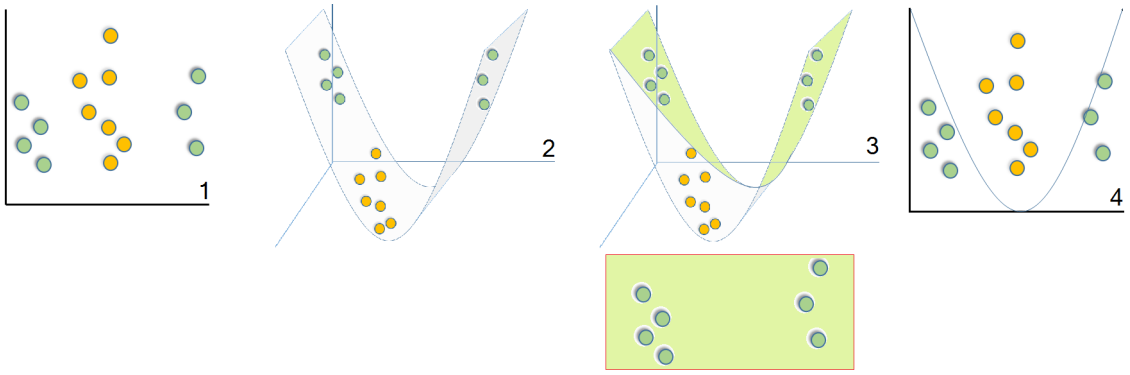


Imagen 7. Funcionamiento del kernel en un modelo basado en SVM.

Las funciones kernel nos permiten convertir problemas no linealmente separables en problemas linealmente separables, aumentando la efectividad de nuestro modelo SVM.

Las principales funciones kernel son:

- kernel lineal $K(x, x') = x * x'$
- kernel polinómico $K(x, x') = (x * x' + c)^d$
- kernel gaussiano $K(x, x') = \exp\left(-\Upsilon\left|\left|x - x'\right|\right|^2\right)$

Donde los valores Υ , c y d son los parámetros de cada kernel.

3.2 Redes Neuronales Artificiales

Las redes neuronales artificiales o *Artificial Neural Networks* (ANNs) están inspiradas en el funcionamiento de las neuronas biológicas que residen en nuestro cerebro y están compuestas de colecciones de unidades mínimas llamadas neuronas artificiales, que a su vez se organizan en tres tipos de capas (capas de entrada, capa oculta y capa de salida).

El funcionamiento básico de una neurona artificial es el cálculo del producto escalar entre los pesos del *input* y los pesos internos de la propia neurona, tal y como se muestra en la imagen 8.

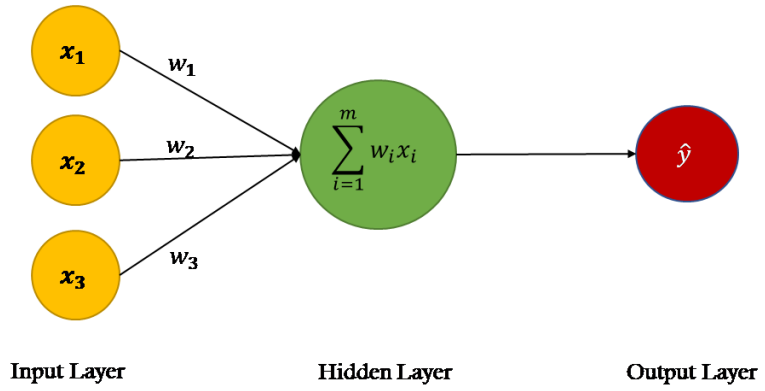


Imagen 8. Funcionamiento de una neurona artificial.

Cada una de estas neuronas se conecta con otras para formar una red, cuyo propósito es el de encontrar el conjunto adecuado de pesos que nos ayuden a realizar la tarea para la cual hemos programado la red.

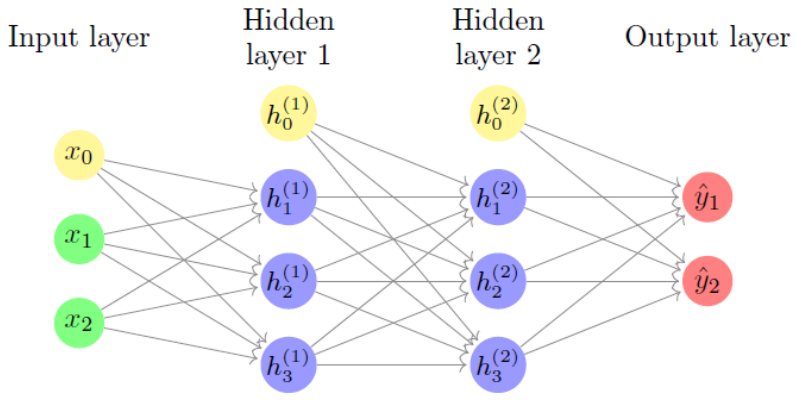


Imagen 9. Ejemplo red neuronal multicapa con 3 capas.

Como se observa en la imagen 9, la red cuenta con 4 capas; sin embargo, decimos que es una red con 3 capas, dado que por convención solo se cuentan las capas con pesos entrenables (por tanto, descontamos la capa 0 o capa de entrada).

Las redes neuronales con más de una capa son aproximadores universales capaces de aproximar cualquier función continua [3].

Las ANN están formadas por capas llamadas densamente conectadas, completamente conectadas o simplemente capas lineales. Algunas librerías de *deep learning* como Caffe las consideran como una operación de producto escalar que pueden estar seguidas o no de una capa no lineal [1]. Su principal parámetro es el tamaño de la capa de salida, que a su vez es el número de neuronas que hay en el *output* de la misma.

3.2.1 Funciones de activación

Con tal de permitir a los modelos basados en el uso de redes neuronales la resolución de problemas más complejos se necesita añadir un bloque no lineal al final del cálculo del producto escalar de cada neurona.

Si no usamos estas funciones no lineales de activación, nuestro modelo tendrá siempre un comportamiento lineal, sin importar cuántas capas haya (dado que la combinación lineal de funciones lineales da como resultado otra función lineal).

Existen múltiples funciones de activación. En esta sección vamos a describir las más utilizadas:

- ReLU(Rectified Linear Unit) $f(x) = \max(x, 0)$

  La función ReLU es ampliamente utilizada para extraer mapas de características sobre los datos de entrada. Es por ello que dicha función se sitúa en las capas ocultas de nuestra red, donde se aprenden las características más relevantes de dichos datos de entrada para poder resolver el problema en cuestión.

- Sigmoid $f(x) = \frac{1}{1+e^{-x}}$

  La función sigmoid se utiliza para obtener la probabilidad (número decimal entre 0 y 1) de cada clase (suma total 1). Es por ello que dicha función se sitúa en las capas finales de nuestra red.

- Softmax $f(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e_j^x}$

  La función softmax tiene el mismo propósito que la función sigmoid pero, a diferencia de esta, solo devuelve la clase con mayor probabilidad. Es por ello que esta función es ampliamente utilizada para problemas de clasificación multiclase.

3.2.2 Entrenamiento de redes neuronales

Para que nuestra red neuronal sea capaz de aprender, necesitaremos una cantidad de datos que la entrene; este conjunto de datos recibe el nombre de conjunto de entrenamiento y consta de una serie de *inputs* (X) junto con sus correspondientes *outputs* deseados (Y).

La red aprende de dichos datos, es decir, los parámetros de nuestra red se actualizarán de tal forma que para cada dato de entrada X la red es capaz de obtener el *output* correcto Y dentro de los datos de entrenamiento. Tras entrenar nuestra red se espera que la misma sea capaz de generalizar y clasificar de forma correcta los *inputs* que no han sido vistos durante el entrenamiento.

Con el fin de que dicha generalización tenga éxito, nuestro *dataset* de entrenamiento debe ser lo bastante extenso y representativo como para contemplar la mayor cantidad de casos posibles y extrapolarlos a los casos que queremos clasificar. Por ejemplo, si queremos entrenar un modelo que clasifique los diferentes tipos de monedas que existen, deberemos tener en cuenta las monedas de todos los países en la misma proporción.

Como se observa en la imagen 10, durante el entrenamiento la red ejecuta dos procesos básicos distintos:

- **Propagación hacia atrás o *backward propagation***. Se calcula el impacto que tiene cada peso sobre la producción de errores o pérdidas en la red y se actualizan los pesos con el fin de minimizar el valor de dichas pérdidas.
- **Propagación hacia delante o *forward propagation***. La red obtiene el *output* del correspondiente *input* y, a través de una función de pérdida o *loss function,* se evalúa el resultado, indicándonos si la red está actuando de manera correcta o no.

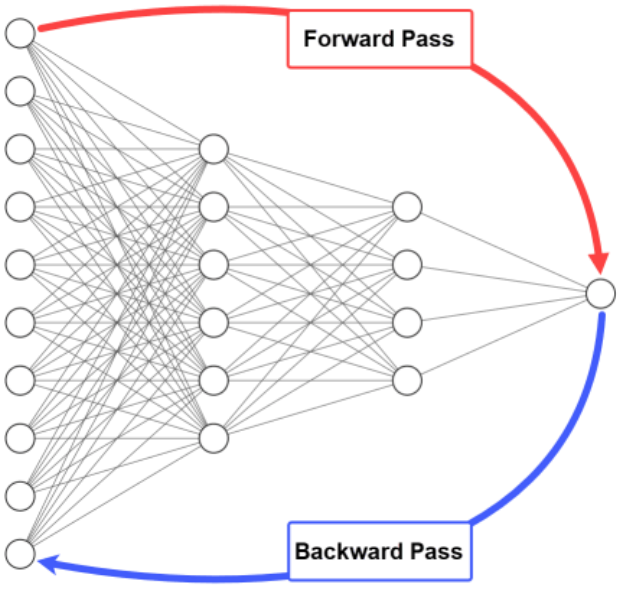


Imagen 10. *Forward propagation* y *Backward propagation*.

3.2.3 Funciones de pérdida

Las funciones de pérdida o *loss functions* se encargan de cuantificar (devuelven un escalar) la efectividad de nuestro conjunto de pesos de la red a la hora de predecir correctamente los *outputs* esperados (Y) para nuestro conjunto de *inputs* (X) de nuestro *dataset* de entrenamiento.

Las funciones de pérdida principalmente usadas para tareas de clasificación son dos: ***log loss*** (clasificación con solo dos posibles resultados, normalmente binarios) y ***cross-entropy loss*** (para clasificación con más de dos resultados)[3].

3.2.4 Optimizadores

El algoritmo de propagación hacia atrás nos devuelve la derivada de cada parámetro con respecto a la función de pérdida. De esta manera se descubre cómo afecta el cambio de cada parámetro en el valor de la función de pérdida. Los encargados de minimizar el valor de la función de pérdida e ir minimizando el error de entrenamiento mientras se entrena la red son los algoritmos de optimización [5].

La labor de los optimizadores es actualizar los pesos de la red con el fin de minimizar el error de pérdida en el entrenamiento. Éstos a su vez disponen de parámetros que sirven para modificar su comportamiento y que se ajustan en función del problema.

El parámetro más importante es la **tasa de aprendizaje o *learning rate,*** que se encarga de controlar la rapidez con la que el optimizador minimiza la función de pérdida. Si se pone a un valor muy alto tendremos problemas para llegar al mínimo deseado. mientras que si se pone a un valor muy bajo tendremos problemas con la convergencia y puede que nos devuelva un mínimo local.

Otro aspecto a tener en cuenta es que mientras el entrenamiento va avanzando y el error disminuye, el valor de la tasa de aprendizaje puede ser demasiado grande y se puede llegar a sobrepasar el mínimo.

Para solventar este inconveniente se recurre a realizar un proceso llamado **planificación de la tasa de aprendizaje o *learning rate scheduling,*** que se encarga de disminuir la tasa de aprendizaje a medida que avanza el entrenamiento con un determinado valor de descenso.

Los dos optimizadores más empleados en la actualidad son los optimizadores SGD y Adam:

- ***SGD (Stochastic Gradient Descent).***

  El optimizador SGD es el optimizador de gradiente descendente más popular. Generalmente logra alcanzar el mínimo valor para la función de pérdida, pero requiere realizar una configuración de sus parámetros (tasa de aprendizaje y

momentum) en función del problema; de lo contrario, se puede quedar atascado en un mínimo local [5].

- ***Adam (Adaptative Moment Estimation).***

El optimizador Adam es uno de los optimizadores con tasa de aprendizaje adaptativa más empleados.

Este tipo de optimizadores destacan por ofrecer una alta efectividad con sus parámetros por defecto, dado que ajustan automáticamente la tasa de aprendizaje a medida que avanza el entrenamiento de la red. Por lo tanto, nos ofrecen mejores tiempos de convergencia, tal y como se observa en la imagen 11.

Este tipo de optimizador es ampliamente utilizado en el entrenamiento de redes neuronales profundas. Además, también se recomienda su uso para problemas donde los datos de entrada son escasos [5].

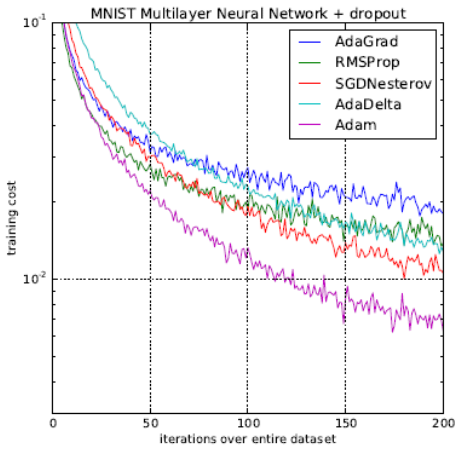


Imagen 11. Comparativa coste de entrenamiento de optimizadores (MNIST NN + *dropout*).

3.2.5 *Feature scaling*

Antes de pasar a entrenar nuestra red es necesario normalizar o estandarizar nuestros datos. Si, por ejemplo, estamos tratando con un *dataset* sobre casas donde nuestro vector de entrada (X) contiene características como número de habitaciones (2,4, …) y área de la casa (1000, 10000, …) por cada muestra, decimos que ambas características no están a la misma escala. Será necesario realizar una operación de estandarizado con el fin de evitar problemas de convergencia y retrasos con el algoritmo de descenso por gradiente (como se puede apreciar en la imagen 12).

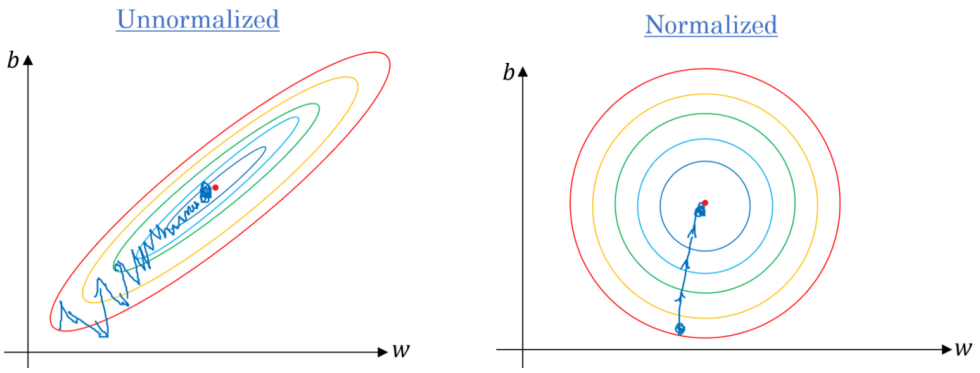


Imagen 12. Tiempo de convergencia para datos escalados vs no escalados.

Es importante destacar que, por este mismo motivo, el preproceso aplicado a nuestro conjunto de entrenamiento debe ser aplicado también a nuestro conjunto de datos utilizado para testear nuestra red.

3.2.6 *Batches y epoch*

La idea de mantener el *dataset* entero en memoria mientras se entrena la red es inviable para *datasets* extensos. Por ello, durante el entrenamiento de la misma es habitual dividir dicho conjunto de datos en partes pequeñas llamadas mini lotes (comúnmente lotes o *batches*). De esta manera, cada *batch* es cargado en memoria y utilizado por la red, la cual se encargará de aplicar los algoritmos de propagación hacia atrás y de descenso por gradiente para calcular los pesos de la misma. Este proceso se repite para todos los *batches*, hasta que se entrena el *dataset* completo.

Durante el entrenamiento de cada *batch* se realiza una estimación aproximada del gradiente de nuestra red; sin embargo, al recibir aproximaciones de manera repetida la red es capaz de llegar a un mínimo lo suficientemente bueno en la función de pérdida.

A mayor tamaño de *batch* se obtiene mejor tasa de aprendizaje y, por tanto, mejor estimación del verdadero gradiente. Sin embargo, esto requiere mayor capacidad de memoria para poder mantener dicho *batch* en la etapa de entrenamiento [3].

Cuando la red ha visto todos los *batches*, es decir, se ha entrenado para el *dataset* completo, decimos que ha completado una etapa o *epoch*. Como se verá en el capítulo 5 de este documento, se necesitan múltiples *epoch* para que el valor de la función de pérdida pueda converger.

3.2.7 *Overfitting* y *underfitting*

Como se ha comentado con anterioridad, durante la etapa de entrenamiento nuestra red busca generalizar y clasificar de manera correcta los datos de entrada, utilizando para ello los pares (valor, etiqueta) obtenidos a partir de los conjuntos de entrada X e Y. Si la red es efectiva, ésta será capaz de clasificar correctamente un nuevo conjunto de entrada X_{val} y obtener el conjunto Y_{val} correspondiente.

La precisión de nuestra red depende en gran parte de nuestro *dataset* de entrada X, como se ha explicado de forma previa. Sin embargo, otro de los elementos más importantes que afectan a la precisión de la misma es su estructura, o lo que es lo mismo, las capas y las neuronas que la forman. Elegir una estructura es un factor clave para poder obtener los resultados esperados.

Los principales problemas derivados de una mala elección de la estructura de nuestra red son los siguientes:

- *Underfitting*. El *underfitting* surge cuando la red no es lo suficientemente grande como para lograr captar la complejidad de los datos de entrada. En este caso la red no será capaz de clasificar correctamente el conjunto X, puesto que no ha sido capaz de aprender las características de los datos de entrenamiento.

- *Overfitting*. El *overffiting*, en contraposición al *underfitting*, surge cuando nuestra red es demasiado grande para los datos de entrada y, por tanto, ésta aprende al detalle las características de los mismos. Como resultado, la red no será capaz de clasificar correctamente el conjunto X_{val} (diferente del conjunto X), puesto que ha aprendido demasiado bien del conjunto X y no reconoce datos que no sean exactamente igual.

3.3 Redes Neuronales Convolucionales

A continuación pasaremos a ver otro tipo de red neuronal, la cual está especialmente diseñada para trabajar con datos que tienen propiedades espaciales como las imágenes. Este tipo de red se conoce como Red Neuronal Convolucional o *Convolutional Neural Network* (CNN) [3].

Las CNN están formadas principalmente por capas llamadas capas convolucionales o *convolution layers,* que se encargan de filtrar los *inputs* de la capa para encontrar como resultado características útiles entre dichos datos de entrada. Este tipo de filtrado se llama convolución.

3.3.1 Convoluciones

La convolución es un tipo especializado de operación lineal [3].

Se utilizan dos convoluciones bidimensionales en el caso del procesamiento de imágenes, con el objetivo de implementar filtros para las mismas que nos sirven para reconocer patrones específicos de la imagen o para extraer sus características.

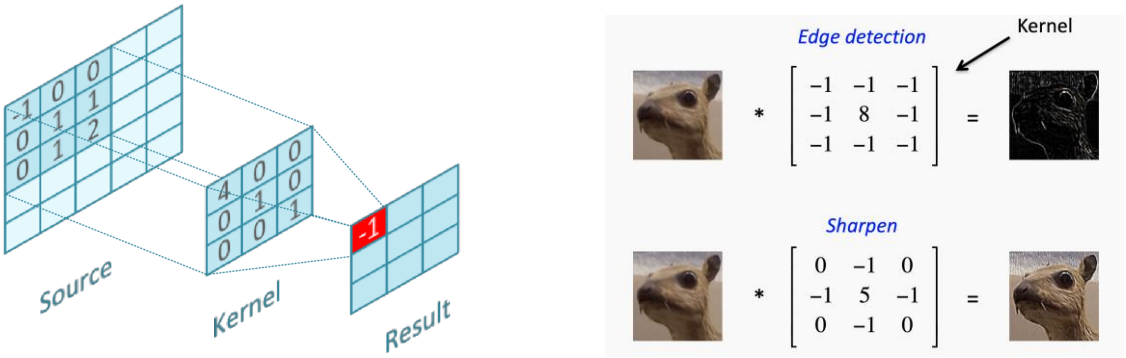


Imagen 13. Proceso de convolución.

Tal y como se aprecia en la imagen 13, las capas convolucionales filtran la entrada mediante una pequeña ventana llamada *kernel*. Este *kernel* es el que define exactamente qué cosas queremos filtrar en la operación de convolución y producirá una respuesta cada vez que se detecte lo que esta buscando. Los *kernels* se pueden utilizar para extraer aspectos específicos de la imagen como, por ejemplo, las esquinas.

Los parámetros que se pretenden aprender mediante las capas convolucionales son los pesos del/de los *kernel/kernels* de la capa. Durante el entrenamiento de las CNNs, los valores de estos filtros se ajustarán de manera automática con el fin de extraer la información más útil para la tarea en cuestión.

En las capas convolucionales cada *kernel* se desliza por toda la entrada buscando patrones específicos. Los *kernels* son pequeños en tamaño e independientes del tamaño de lo que están convolucionando.

Los hiperparámetros principales encargados de controlar el comportamiento de una capa convolucional son los siguientes:

- **Tamaño del *kernel* (K)**. Cómo de grande es la ventana de *kernel* (en píxeles). Generalmente se utilizan ventanas de tamaño impar y reducido (1, 3, 5 y 7).
- **Paso o *stride* (S).** Cuántos píxeles se va a desplazar la ventana *kernel* en cada paso del proceso de convolución. Normalmente suele tener un valor de 1 para evitar perder información.
- ***Zero padding* (pad).** Número de ceros que se colocan en el borde de la imagen (nos permite filtrar completamente cada ubicación de la imagen de entrada incluyendo bordes, tal y como se observa en la imagen 14).

- **Número de filtros(F).** Cuántos filtros tendrá nuestra capa de convolución o, lo que es lo mismo, la cantidad de patrones a buscar.

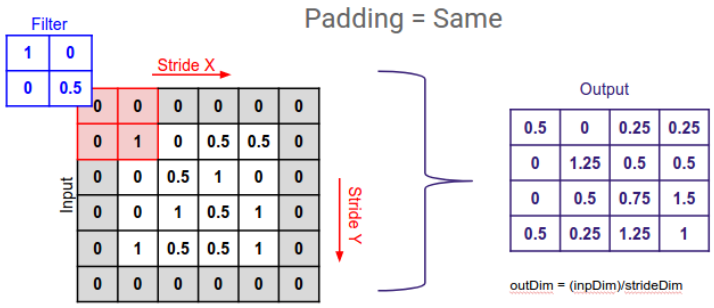


Imagen 14. Hiperparámetro *Zero Padding*.

Para realizar el cálculo del tamaño del *output* de una operación de convolución se utiliza la siguiente fórmula:

$$(W_{out}, H_{out}) = \frac{(W_{in},\ H_{in}) - K + 2P}{S}$$

Donde *W* y *H* hacen referencia a la anchura (*width*) y altura (*heigth*) de la entrada/salida, *K* es el tamaño de los *kernels*, *P* hace referencia al *padding* y *S* es el *stride*.

Para calcular el número de parámetros o pesos dentro de una capa convolucional se utiliza la siguiente fórmula:

$$n_{params} = (m \cdot n \cdot d_{in}) \cdot k$$

Donde *m·n* es el tamaño del kernel, d_{in} es la profundidad de la entrada (corresponde al número de canales de color de la imagen) y *k* es el número de *kernels*.

Por ejemplo, si tenemos un *input* con formato 32 x 32 x 3 y una capa convolucional con 10 kernels de tamaño 5 x 5, el número de parámetros sería $(5 \cdot 5 \cdot 3)\ \cdot 10 = 750$.

Por otro lado, para las capas completamente conectadas se emplea la siguiente fórmula:

$$n_{params} = input_{shape} \cdot num_{outputs}$$

Si utilizamos una red neuronal artificial directamente en la imagen perderíamos información espacial de la misma, además de tener muchos más parámetros dado que tendríamos un parámetro por cada píxel para cada neurona. Si utilizamos el ejemplo anterior para realizar el cálculo del número de parámetros en una capa completamente conectada, obtendremos un valor de 30720 = $(32 \cdot 32 \cdot 3)\ \cdot 10$.

Las capas convolucionales dentro de las arquitecturas actuales de CNN son responsables del 90% del coste computacional de todo el modelo.

El coste computacional de una capa neuronal medida en MACs (*Multiply Add Acumulators*) por operaciones viene dado por la siguiente fórmula:

$$MAC = [F\ \cdot F\ \cdot C\ \cdot (H + 2 \cdot P - FS + 1) \cdot (W + 2\ \cdot P - FS + 1) \cdot M] \cdot B$$

Donde *F* es el tamaño del *kernel* convolucional, *M* el número de filtros, *H* y *W* son la altura y amplitud del mapa de características de entrada, *B* es el tamaño del *batch* de entrada, *C* es la profundidad del *input* (número de canales) y *S* es el *stride* de la capa convolucional.

Por otro lado, las capas completamente conectadas tienen el siguiente número de operaciones:

$$MAC = [H \cdot W \cdot C \cdot OutputNeurons] \cdot B$$

Donde la variable $OutputNeurons$ hace referencia al número de unidades de salida de la capa. Generalmente, las capas convolucionales del inicio de la CNN representan la mayoría del coste de computación, pero tienen menos parámetros, mientras que, por el contrario, las últimas capas tienen mayor número de parámetros y menor coste computacional [3].

3.3.2 La capa *pooling*

La capa *pooling* o capa de agrupación se utiliza con la finalidad de reducir la dimensionalidad espacial de la entrada (pero no la profundidad del volumen de la CNN). Con la reducción de la dimensionalidad se gana rendimiento computacional.

Una de las principales características de las capas *pooling* es su ausencia de parámetros (dicha capa no tiene pesos que aprender).

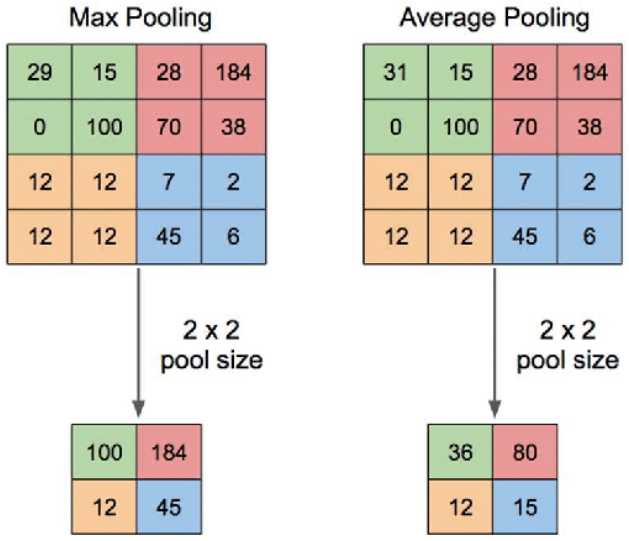


Imagen 15 – Funcionamiento de las capas *max pooling* y *average pooling*.

Aunque hay múltiples capas de *pooling*, las más utilizadas son las capas de *max pooling* y la capa de *average pooling*.

Como se observa en la imagen 15 la capa de *max pooling* se desliza por cada ubicación de la ventana extrayendo el valor más grande como *output*, mientras que la capa de *average pooling* extrae el valor promedio de los píxeles.

Por otro lado, para ajustar la profundidad (manteniendo la información espacial) se emplean convoluciones 1 x 1, en un proceso de expansión o contracción, tal y como se aprecia en la imagen 16.

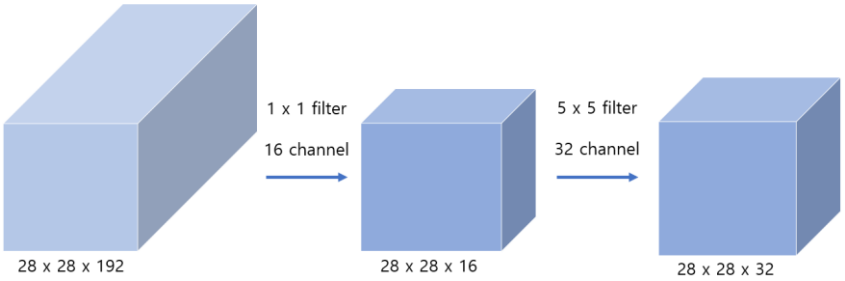


Imagen 16. Capa convolucional 1x1.

Una de las principales técnicas empleadas actualmente para la reducción de los parámetros de nuestro modelo es el uso de pequeñas convoluciones en cascada. Dicha técnica recibe el nombre de *cascading* [3].

3.4 Principales arquitecturas CNN empleadas

En esta sección vamos a explicar algunas de las arquitecturas más utilizadas que han mostrado una gran efectividad en muchas tareas comunes como la detección o la clasificación de imágenes.

3.4.1 VGG *net*

Creada por el Grupo de Visual Geometry (VGGroup)[1], fue una de las primeras redes en introducir la idea de apilar un gran número de capas pequeñas. VGGNet utiliza pequeños filtros de tamaño 3 x 3 intercalados normalmente con capas de *max pooling* de tamaño 2 x 2.

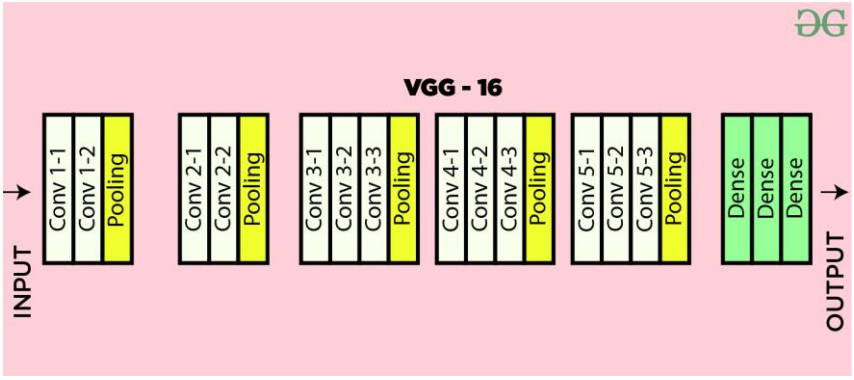


Imagen 17. Ejemplo de red VGG, concretamente VGG 16 con un total de 16 capas.

Como se observa en la imagen 17, la red presenta un diseño modular simple basado en apilar tres capas convolucionales seguidas por las correspondientes capas de *max pooling*, lo cual permite que sea fácilmente ampliada o reducida en cuanto a tamaño, dando lugar a modelos como VGG11, VGG13 o VGG16.

En 2014, VGG recibió el segundo premio en la competición de *Imagenet Clasification* y el primer premio en la competición de *Imagenet Localization* [3].

El modelo VGG ha demostrado ser útil en muchas tareas [3] y, debido a su simple arquitectura, es un modelo perfecto para empezar a experimentar. No obstante, sus principales inconvenientes son:

- El uso de filtros 3 x 3 , especialmente en la primera capa, hace que la cantidad de cómputo no sea apta para soluciones móviles.
- Los modelos VGG más profundos (con más capas) no funcionan tan bien debido a problemas con los gradientes [3].
- La gran cantidad de capas completamente conectadas en el diseño original se traduce en demasiados parámetros, lo que nos puede llevar a problemas de sobreentrenamiento [3].
- El uso de muchas capas *pooling* no se considera un buen diseño [3].

[1] https://www.robots.ox.ac.uk/~vgg/

3.4.2 Inception *net*

El bloque Inception (conjunto de capas) tiene como objetivo cubrir una área amplia de la imagen, a la vez que se pretende mantener una buena resolución para poder observar información local importante dentro de la misma [3].

Además de crear redes más profundas, el bloque Inception introduce la idea de convoluciones paralelas. Las convoluciones (de diferentes tamaños) se realizan de forma paralela en la salida de la capa anterior a la misma.

La idea básica de Inception es usar todos los tamaños posibles de los *kernels* y tipos de operaciones para abarcar la mayor cantidad de información posible y dejar que el algoritmo de propagación hacia atrás decida cuál de esta información utilizar en base a los datos. El problema de esto es el coste computacional que conlleva; es por ello que en la práctica se aplican filtros de 1 x 1 antes de aplicar el *kernel*, con el fin de reducir la profundidad de la entrada.

En la imagen 18 se muestra la estructura de un bloque Inception y el modelo InceptionV3, basado en la concatenación de múltiples bloques Inception.

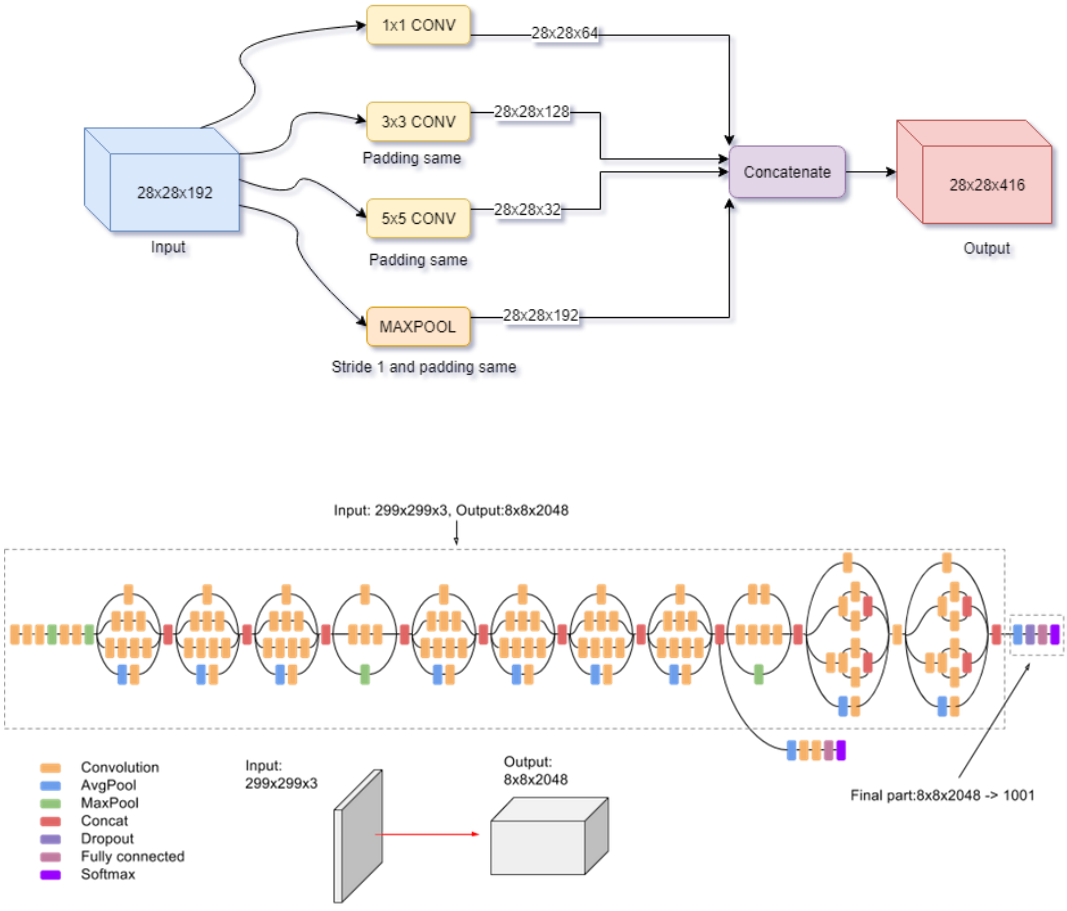


Imagen 18. Estructura de un bloque Inception y descripción el modelo Inception-V3.

Aunque InceptionV3 tenga mayor precisión que el modelo VGG [3], al apilar tantas capas Inception continuamos teniendo problemas con los algoritmos de gradiente.

3.4.3 *Residual net*

Como hemos visto en secciones anteriores, la profundidad de una red es un factor determinante que contribuye a mejorar la precisión. Dicho esto, cabría plantearse si a mayor número de capas mejor precisión.

Según los autores del estudio [6], se encontró que la precisión se satura una vez el modelo llega a alcanzar las 30 capas de profundidad.

Para solventar dicho problema se decidió recurrir al uso de un nuevo bloque de capas llamado bloque residual, que, tal y como se observa en la imagen 19, añade la salida de la capa anterior a la salida de la capa actual.

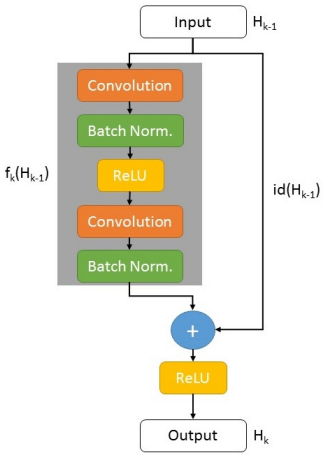


Imagen 19. Funcionamiento básico de una capa residual.

La red residual, *Residual Net* o simplemente *ResNet* ha mostrado resultados excelentes con redes muy profundas (con incluso más de 100 capas), superando incluso a las redes Inception. En la imagen 20 se muestra su estructura.

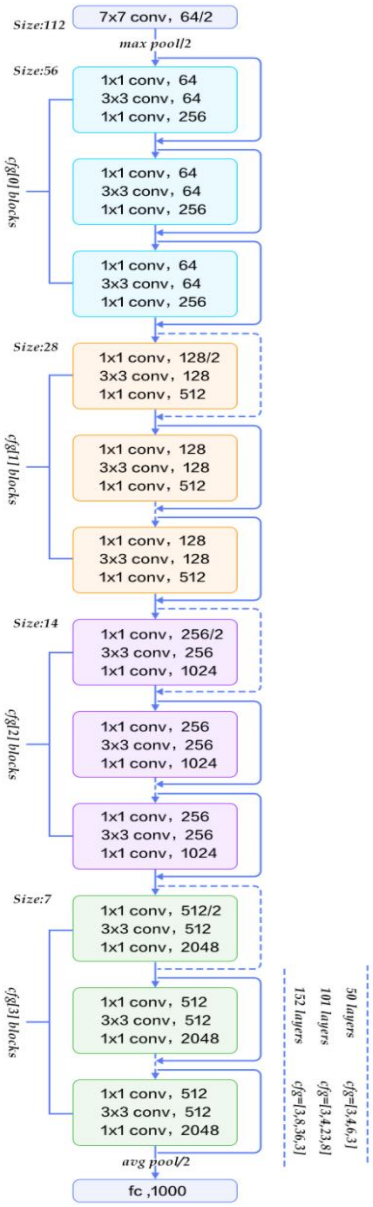


Imagen 20. Estructura básica de una red residual.

3.4.4 Inception-*ResNet*

El modelo Inception-*ResNet*, tal y como su nombre indica, surge como combinación de las redes Inception y *ResNet*. Dicho modelo propone un nuevo modelo de red con hasta 164 capas de profundidad, destacándose principalmente por el uso de nuevos bloques de capas llamados bloques Inception residuales o *residual Inception Blocks*. En la imagen 21 se detalla la estructura de este modelo.

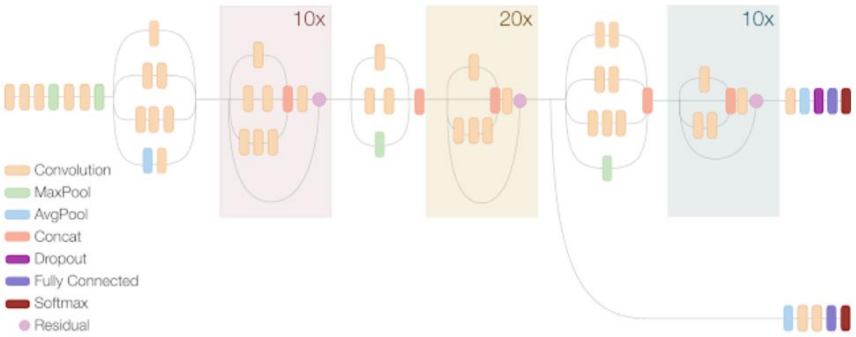


Imagen 21. Estructura del modelo Inception-ResNetV2.

3.4.5 NAS *net*

La red NAS (*Neural Architecture Search*) o red de búsqueda de arquitectura neuronal se caracteriza por el uso de un método de búsqueda de aprendizaje por refuerzo que determina la estructura de dicha red, definiendo de forma automática los bloques o células dependiendo del problema. La red NAS a su vez se compone de **células normales** (células convolucionales que devuelven un mapa de características de la misma dimensión) y de **células de reducción** (células convolucionales que devuelven un mapa de características donde la amplitud y la profundidad de dicho mapa se reducen por un factor de dos).

Dicha red utiliza un controlador (red neuronal recurrente o RNN) que se encarga de predecir recursivamente el resto de la estructura para las células convolucionales. En la imagen 22 se detalla el funcionamiento de dicho controlador; además, en la imagen 23 se detalla la arquitectura de la red NAS.

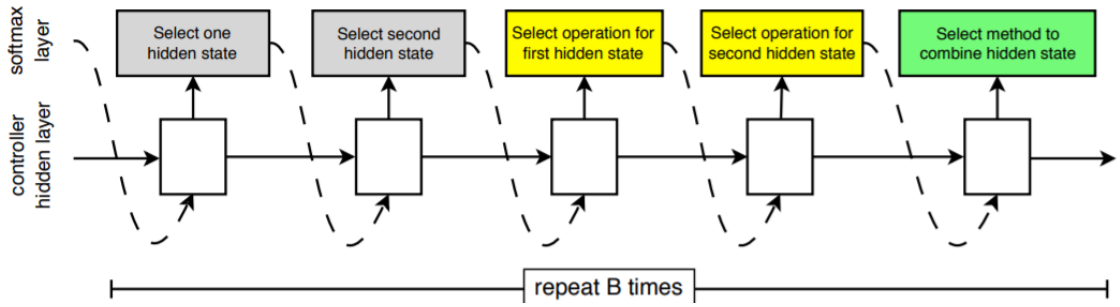


Imagen 22. Funcionamiento del controlador RNN en las redes NAS.

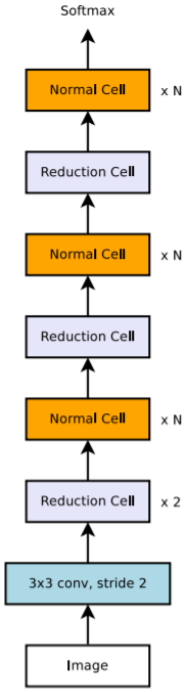


Imagen 23. Arquitectura básica de una red NAS entrenada mediante ImageNet.

3.5 *Transfer learning*

La transferencia de aprendizaje o *transfer learning,* tal como su nombre indica, es una técnica que se emplea para transferir el conocimiento aprendido en una tarea y aplicarlo a otra tarea diferente.

El *transfer learning* es una técnica muy empleada, dado que entrenar un modelo desde cero es muy costoso y su efectividad depende de varios factores. Además, en algunas ocasiones no hay suficientes datos para entrenar una arquitectura profunda como *ResNet*, lo que se traduce en problemas de sobreentrenamiento.

La idea principal en la que se fundamenta el *transfer learning* es el hecho de que las capas finales en redes con arquitecturas profundas ya entrenadas para una tarea determinada se pueden reutilizar en otra tarea diferente, dado que al congelar el resto de capas se obtienen pesos muy similares. Así pues, se puede utilizar una arquitectura

profunda ya entrenada con un conjunto de datos extenso (por ejemplo, ImageNet) [3] que nos permita generalizar de forma adecuada, de tal manera que sus pesos convolucionales actúen como extractores de características para nuestro clasificador. En la imagen 24 se detalla la diferencia entre el entrenamiento de una red convencional y el entrenamiento de un modelo basado en *transfer learning*.

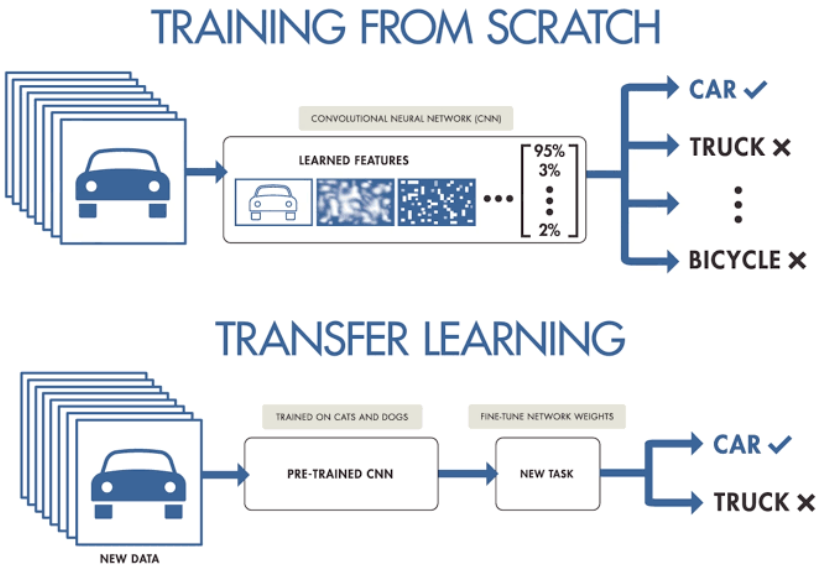


Imagen 24. Entrenamiento desde cero vs *transfer learning*.

La extracción de características mediante los pesos de una red convolucional entrenada con el conjunto de datos de ImageNet es uno de los métodos de extracción más efectivos, superando a métodos tradicionales como Histogramas de Gradientes Orientados (HOG), Bag of Words (BoW), … entre otros [3].

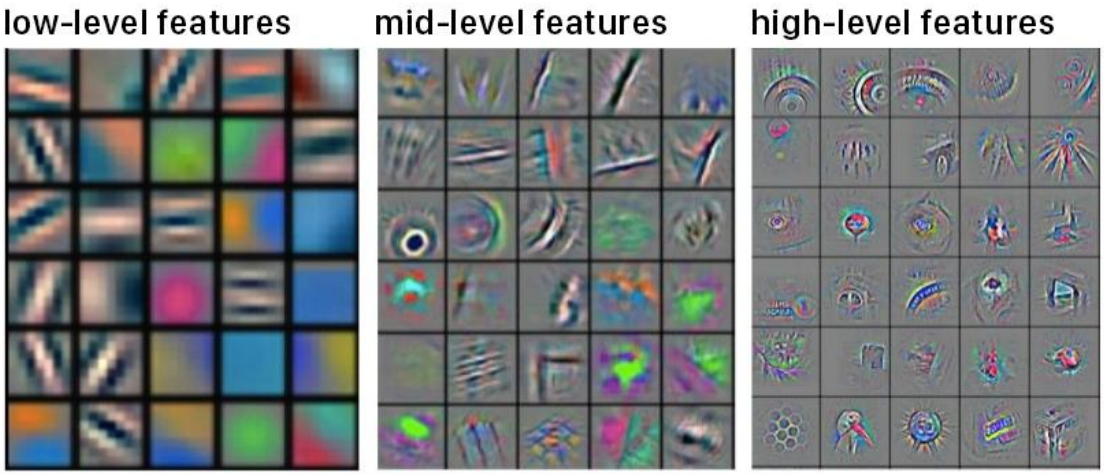


Imagen 25. Filtros convolucionales en las diferentes capas de una red neuronal convolucional entrenada con ImageNet.

Como se observa en la imagen 25, los filtros convolucionales de las primeras capas de la CNN entrenada con ImageNet aprenden patrones o características de bajo nivel, mientras que los filtros convolucionales aplicados en las últimas capas de la red se encargan de reconocer características a más alto nivel, capturando la información específica a nivel de cada clase.

Esto explica el poder que tiene la red de generalizar muy bien en niveles superiores, llegando a utilizar dicho aprendizaje para reconocer incluso casos nunca antes vistos por el modelo.

Se recomienda hacer uso de técnicas de *transfer learning* cuando tenemos un conjunto de datos pequeño para entrenar nuestro modelo. En estos casos podemos entrenar nuestro modelo con un conjunto de datos de mayor tamaño que contenga información semántica similar para después reentrenar la última capa del modelo (clasificador lineal) con nuestro conjunto de datos de menor tamaño. En el caso de contar con suficientes datos y disponer a su vez de otro *dataset* similar con mayor tamaño, podemos preentrenar nuestro modelo con este último *dataset* para obtener una mayor robustez.

Capítulo 4 Materiales y métodos

En este capítulo se describirán todas las herramientas empleadas para realizar el proyecto.

Primero se hará una descripción de todos los materiales utilizados, incluyendo lenguajes de programación, entornos de desarrollo, editores de código, características del equipo utilizado, etc.

En segundo lugar, se describirán los métodos empleados para realizar el entrenamiento de los diferentes modelos utilizados en la experimentación, así como el significado de cada uno de sus parámetros, los valores recomendados para los mismos y, por supuesto, las librerías empleadas.

4.1 Materiales

Para poder realizar este proyecto se ha recurrido al uso del lenguaje de programación Python por dos motivos principales:

1 – Librerías específicas para *Machine Learning*. Una de las principales características de Python es la amplia disponibilidad de librerías con las que cuenta, librerías como Numpy [2] o Pandas [3] nos serán útiles para realizar las tareas más comunes, mientras que otras librerías como Keras [4] nos permitirán aplicar los conceptos de Aprendizaje Automático explicados con anterioridad. Python es uno de los principales lenguajes de programación utilizados para labores de aprendizaje automático.

2 – Soporte de la comunidad. Python es reconocido por ser un lenguaje sencillo de utilizar, fácilmente legible y ampliamente utilizado, contando con una amplia comunidad que lo soporta. Debido a la gran cantidad de público que utiliza este lenguaje de programación, es fácil encontrar información sobre métodos o posibles problemas que nos puedan surgir recurriendo al uso de motores de búsqueda como Google.

Además de lo anteriormente mencionado, esta facilidad de uso permite a quienes lean este documento que tengan unas mínimas nociones básicas sobre programación realizar proyectos similares de la forma más sencilla posible.

Una vez tenemos claro el lenguaje de programación que vamos a utilizar, el siguiente paso será descargar un intérprete de dicho lenguaje que nos permita utilizarlo en nuestro sistema Windows (también está disponible para Linux y para Mac). El intérprete utilizado será Anaconda Navigator (concretamente la versión 2020.11). El primer paso a realizar

[2] https://numpy.org/
[3] https://pandas.pydata.org/
[4] https://keras.io/

será la descarga e instalación de dicho programa (de código abierto) mediante la web oficial [5].

Una vez descargado e instalado correctamente pasaremos a crear un nuevo entorno. Para ello accederemos al apartado *Enviroments* (en el menú de la izquierda) y seleccionaremos la opción *Create* tal y como se observa en la imagen 26. Pondremos el nombre que queramos y la versión de Python correspondiente (en este caso 3.8).

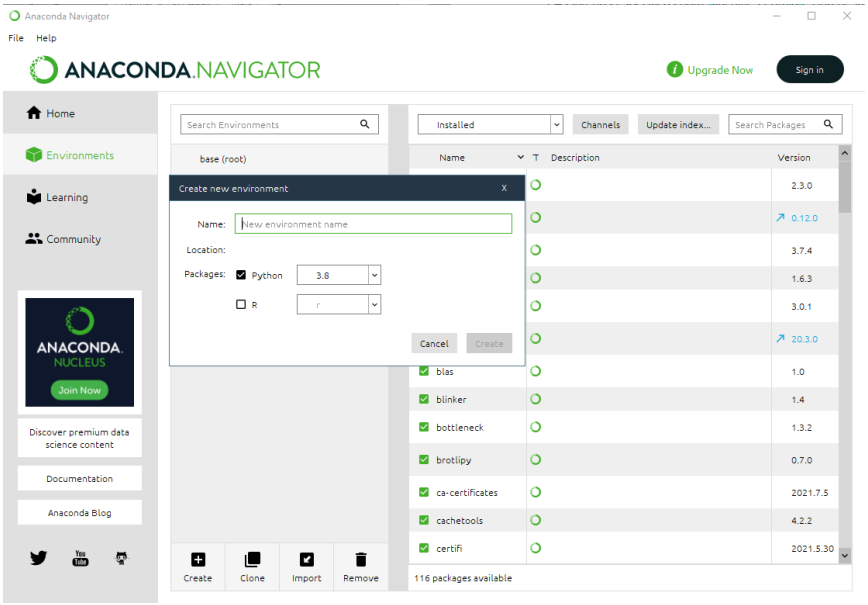


Imagen 26. Creación de nuevo entorno.

Una vez creado nuestro entorno tendremos acceso a una interfaz que nos permitirá abrir un terminal y ejecutar desde aquí nuestro código, pero no sin antes instalar las librerías necesarias haciendo uso de la siguiente orden:

conda install [NombreLibrería]

Como editor de texto se utilizará Visual Studio Code [6], un editor sencillo e intuitivo que nos permite instalar extensiones para marcar la sintaxis de Python.

Además de todo lo anterior, el equipo empleado para la realización del proyecto cuenta con un procesador de 2 núcleos y 4 hilos a una velocidad base de 2.5 GHz con una memoria RAM de 8 GB, todo ello funcionando con el sistema operativo Windows 10.

[5] https://anaconda.org
[6] https://code.visualstudio.com/

4.2 Métodos

Una vez indicados los materiales necesarios pasaremos a explicar los métodos empleados en este proyecto.

Las librerías empleadas para la realización de este trabajo son las siguientes:

- `os`. Librería encargada del manejo de ficheros (lectura/escritura).
- `numpy`. Librería de propósito general.
- `pandas`. Librería especializada en *DataFrames*.
- `tqdm`. Librería utilizada para mostrar barras de progreso en las estructuras de control (en nuestro caso bucles `for`) mediante un método que tiene el mismo nombre.
- `sklearn`. Librería especializada en *Machine Learning*, principalmente utilizada para la creación y entrenamiento de modelos basados en SVM.
- `keras`. Librería especializada en *Machine Learning*, es la librería principal de nuestro proyecto. Ésta, a su vez, se extrae desde la librería `tensorflow`.

A continuación pasaremos a definir todos los métodos empleados haciendo una breve explicación de su funcionamiento y el significado de los parámetros. Dado que hay multitud de métodos los vamos a organizar por su propósito general creando un total de 3 grupos de métodos:

- Bloque 1. Métodos utilizados para extraer los datos y realizar un preproceso de los mismos.
- Bloque 2. Métodos utilizados para crear modelos ya existentes (ej: ResNet, Inception …).
- Bloque 3. Métodos utilizados para entrenar y clasificar nuestro modelo.

4.2.1 Bloque 1 (lectura y procesado)

```
keras.preprocessing.image.load_img(path,
                                   color_mode='rgb',
                                   target_size=None,
                                   interpolation="nearest")
```

Este método se encarga de leer la imagen almacenada en la ruta `path` y devolver un objeto de tipo `Image` con el tamaño especificado.

Parámetros:

- `path`: ruta absoluta (completa) donde se encuentra la imagen que se quiere leer.
- `color_mode`: determina el modo de color con el que se pretende leer cada imagen, de manera que si elegimos el modo `"rgb"`, por cada píxel de la imagen se almacenaran 3 valores correspondientes a cada canal o color (rojo, verde y azul), mientras que si optamos por escoger un modo `"grayscale"` solo tenemos en cuenta un color (gris), por tanto, cada pixel se puede almacenar con un solo valor.

- *target_size*: tamaño deseado para la imagen leída. Por defecto este tamaño viene determinado por el modo de color elegido, de manera que, si tenemos una imagen de 224 x 224 *píxeles* leída en modo `grayscale`, el tamaño resultante será 224 x 224 x 1, y por lo tanto este parámetro tendrá el valor `[224, 224, 1]`.
- *interpolation*: hace referencia al método de interpolación empleado para volver a muestrear la imagen en el caso de que el tamaño objetivo sea diferente al tamaño de la imagen.

```
keras.utils.to_categorical(y, num_classes=None, dtype="float32")
```

Este método se utiliza para transformar el vector de etiquetas de nuestros datos (*y*) a formato *one-hot*. Si, por ejemplo, disponemos de un vector *y* con un total de 3 clases, el formato *one-hot* para la etiqueta 2 sería [0, 1, 0], para la 1 [0, 0, 1] y para la 3 [1, 0, 0].

El parámetro *num_classes* indica el número de clases de nuestro vector *y;* por defecto toma el máximo valor de dicho vector más 1.

```
keras.preprocessing.image.ImageDataGenerator(
rotation_range=0,
width_shift_range=0.0,
height_shift_range=0.0,
brightness_range=None,
shear_range=0.0,
zoom_range=0.0,
fill_mode="nearest",
horizontal_flip=False,
vertical_flip=False,
rescale=None,
preprocessing_function=None,
data_format=None,
validation_split=0.0,
dtype=None)
```

Clase utilizada para extraer un subconjunto de validación sobre los datos, aplicar una función de preproceso y generar transformaciones aleatorias sobre los mismos.

Parámetros:

- `rotation_range, width_shift_range, height_shift_range, brigthness_range, shear_range y zoom_range.`
  Rango de rotación (valor que indica el número de grados a rotar la imagen), de desplazamiento de ancho (valor en forma de porcentaje, decimal entre 0 y 255 o conjunto de valores [inicial, final] que indica el rango que queremos que se desplace la imagen, siendo a la izquierda si el valor es negativo y a la derecha si no lo es), de desplazamiento de alto (actúa de la misma forma que el rango de desplazamiento horizontal pero desplazando la imagen arriba o abajo), de brillo (vector con un conjunto de posibles valores o tupla con los valores límite que

puede tomar el brillo), de deslizamiento (se fija un eje y se estira la imagen en un ángulo determinado también conocido como ángulo de deslizamiento creando así un estiramiento en la imagen) y de zoom (valor o conjunto de datos que delimitan que rango de zoom se puede aplicar a la imagen leída), respectivamente. Todos estos campos toman valores aleatorios determinados por el rango especificado.

- *fill_mode*. Modo de relleno, que indica cómo se deben completar los puntos fuera de los límites de la entrada. Por defecto tiene un valor "nearest", lo que indica que los puntos de fuera de la entrada heredan el valor anterior a estos, es decir, el último valor que sí esta dentro de la entrada.
- *horizontal_flip, vertical_flip y rescale*. Volteo horizontal (dar la vuelta a la imagen horizontalmente), volteo vertical (dar la vuelta a la imagen verticalmente) y reescalado (valor del factor de escalado que multiplica los datos), respectivamente. Es importante destacar que el reescalado es la última transformación que se aplica.
- *preprocessing_function*. Función de preproceso aplicada a cada imagen. Indica el preproceso que se le aplica a los datos.
- *data_format*. Formato de la imagen. Este argumento puede tomar dos valores diferentes: "channels_last" (si queremos que la imagen tenga una formato altura x anchura x canales_de_color) o "channels_first" (si por el contrario queremos que la imagen tenga un formato canales_de_color x altura x anchura). Por defecto toma el valor "channels_last".
- *validation_split*: Valor decimal de 0 a 1 que representa el tamaño de validación que se empleará para substraer el correspondiente conjunto de validación sobre los datos de entrada, encargado de validar nuestro modelo.
- *dtype*: tipo de datos utilizados para los *arrays* generados.

```
keras.preprocessing.image.ImageDataGenerator.flow_from_dataframe(
dataframe,
directory=None,
x_col="filename",
y_col="class",
target_size=(256,256),
color_mode="rgb",
class_mode="categorical",
batch_size=32,
shuffle=True,
seed=None,
subset=None)
```

El método *flow_from_dataframe* se aplica sobre un objeto *ImageDataGenerator* y se encarga de leer los datos de entrada mediante un directorio y un *dataframe* que contiene la ruta relativa de imagen y su etiqueta. Además, dicho método aplica todas las transformaciones y funciones indicadas por la clase *ImageDataGenerator* sobre la que se aplica.

Parámetros:

- *dataframe*. *Dataframe* (tipo de dato procedente de la librería `pandas` utilizado para representar tablas de datos, con filas y columnas) donde se encuentra la ruta (relativa o absoluta) de cada imagen (columna X) y sus correspondiente etiqueta (columna Y).
- *y_col* y *x_col*. Nombre que reciben la columnas X e Y dentro del *dataframe* indicado anteriormente.
- *directory*. Ruta que indica el directorio de donde se leen las imágenes. Si dicho parámetro no se especifica en la columna x (*x_col*), correspondiente al id de cada imagen, se debe indicar una ruta absoluta de las mismas.
- *target_size*. Tupla de enteros `(altura, amplitud)` que indican el tamaño deseado para nuestras imágenes (se redimensionará el tamaño de las mismas en el caso que no coincida con el tamaño deseado).
- *color_mode*. Modo de color que queremos que tengan las imágenes leídas (`"rgb","grayscale"` o `"rgba"`).
- *class_mode*. Parámetro utilizado para indicar el formato de la columna de etiquetas (*x_col*) de nuestro *dataframe*. Por defecto tiene el valor `"categorical"`, lo que indica que las clases vienen dadas por un vector de dos dimensiones `[imagen, clase formato one-hot]`.
- *batch_size*. Tamaño del *batch* o, lo que es lo mismo, el número de imágenes por *batch*. Como se ha explicado con anterioridad, el *batching* es una técnica empleada en *Machine Learning* que nos permite cargar un conjunto deteminado de imágenes en memoria de manera secuencial (véase apartado 3.2.6).
- *shuffle* y *seed*. Parámetros utilizados para mezclar o barajar nuestros datos de manera aleatoria (el parámetro *shuffle* tendrá un valor *True* en caso de que queramos realizar este proceso y para realizar otro *shuffle* adicional elegiremos un valor entero para el parámetro *seed*).
- *subset*. Subconjunto que vamos a cargar en memoria. Puede ser el conjunto de validación (`"validation"`) o el conjunto de entrenamiento (`"train"`).

```
keras.preprocessing.image.ImageDataGenerator.flow(
x,
y=None,
batch_size=32,
shuffle=True,
seed=None,
subset=None)
```

Este método cumple la misma función que el método anterior, con la diferencia de que los datos no se extraen de un *dataframe*, sino que se pasan como parámetros de dicho método (*x*, *y*).

4.2.2 Bloque 2 (arquitecturas CNN predefinidas)

```
keras.applications.resnet_v2.ResNet50V2
keras.applications.NASNetLarge
keras.applications.vgg16.VGG16
keras.applications.inception_v3.InceptionV3
keras.applications.inception_resnet_v2.InceptionResNetV2
```

Métodos empleados para crear una estructura CNN ya definida. Habitualmente utilizados como capas en modelos basados en *transfer learning*.

Los métodos mencionados en este bloque comparten los siguientes parámetros:

- *include_top*. Como se ha indicado en la sección 3.4 de este documento, las arquitecturas de red más populares y efectivas empleadas para la clasificación de imágenes (VGG16, ResNet, Inception, ...) tienen como capas finales capas densamente conectadas, con el fin de realizar tareas de clasificación. Este parámetro sirve para indicar si se quiere añadir estas capas densamente conectadas en nuestro modelo o no (por defecto tiene valor `True`).
- *weights*. Pesos iniciales de nuestro modelo. Si dicho parámetro toma un valor `None`, incializará los pesos de forma aleatoria. En cambio, también tenemos la opción de inicializar mediante los pesos preentrenados del *dataset* ImageNet (`weights="imagenet"`) o indicar los pesos que se deseen de forma manual mediante la ruta absoluta de los mismos.
- *input_tensor*. Entrada del modelo (por defecto `None`). Se puede seleccionar una capa `layers.Input` para el mismo.
- *input_shape*. Tamaño de entrada de nuestro modelo (solo cuando `include_top=False`). Se debe especificar el tamaño de nuestras imágenes, que actuarán como entrada de nuestro modelo, en forma de tupla `[ancho, alto, canales]` (debe tener 3 canales).
- *pooling*. Este parámetro solo se aplica cuando `include_top = False` . Puede tomar los valores `None` (la salida de nuestro modelo devuelve un vector de 4 dimensiones, este es el valor que se aplica por defecto), `"avg"` (se aplica un *average pooling* a la salida de nuestro modelo) o `"max"` (se aplica un *max pooling* a la salida de nuestro modelo).
- *classes*. Número de clases totales a clasificar en las capas densamente conectadas situadas al final de nuestro modelo. Este parámetro solo se utiliza cuando incluimos la capa densamente conectada al final del modelo (`include_top=True`) y por defecto tiene un valor de `1000` (número de clases empleadas en ImageNet).

Cabe destacar que estas redes CNN predefinidas pueden actuar como capas para nuestro modelo; por lo tanto, al igual que los objetos de tipo capa (`keras.layers`), cuentan con un atributo *trainable*. Este atributo se utiliza para indicar si se quiere entrenar la capa o aplicación o, por el contrario, se pretende usar unos pesos preentrenados (pesos iniciales de la capa, en este caso indicados por el parámetro

weights) sin entrenar nuevamente dicha capa. Por defecto, el atributo `trainable` tiene valor *True*.

4.2.3 Bloque 3 (creación y entrenamiento del modelo)

```
keras.models.Sequential(layers=None)
```

Clase utilizada para crear un modelo secuencial, es decir, un modelo creado mediante una pila simple de capas, donde cada capa tiene exactamente un vector de entrada y un tensor de salida. Los principales métodos aplicables a las instancias de la clase `keras.models.Sequential` son:

```
model.add(layer)
```

Hay dos formas diferentes de especificar las capas que conformaran nuestro modelo sequencial: bien en forma de parámetro al crear el modelo o bien mediante este método *add* que recibe como parámetro la capa a añadir al modelo.

```
model.compile(optimizer="rmsprop", loss=None, metrics=None)
```

El método *compile* nos permite compilar nuestro modelo previamente diseñado e instanciarlo con todos los parámetros y ajustes que hemos indicado con anterioridad. Este método dispone, a su vez, de tres parámetros:

- *optimizer*. Optimizador que queremos utilizar para nuestro modelo (por ejemplo, `"adam"` o `"SGD"`).
- *loss*: hace referencia a la función de pérdida o *loss function* que queremos utilizar en el entrenamiento de nuestro modelo (como por ejemplo, `"categorical_crossentropy"`).
- *metrics:* lista de métricas que va a utilizar nuestro modelo para evaluar su efectividad (ejemplo: `['accuracy']`).

```
model.summary()
```

El método *summary* sirve para imprimir por pantalla un resumen del modelo que hemos creado en tiempo de ejecución. Este resumen incluye el nombre de cada capa con su correspondiente formato de salida y número de parametros (o pesos entrenables) .

```
model.fit(
x=None,
y=None,
batch_size=None,
epochs=1,
verbose="auto",
callbacks=None,
steps_per_epoch=None,
validation_split=0.0,
validation_data=None)
```

El método `fit` inicia el entrenamiento de nuestro modelo. A continuación vamos a explicar sus parámetros:

- `x e y`. Conjuntos X e Y (en formato de vector o de *dataframe*).
- `validation_split , validation_data.` Tamaño requerido para el subconjunto de validación (en formato decimal de 0 a 1) y el conjunto de validación en sí (excluyentes entre sí).
- `batch_size` y `epoch`: Tamaño del *batch* y cantidad de *epoch* con la que se quiere entrenar el modelo (véase apartado 3.2.6).
- `callbacks`. Como se verá más adelante en este mísmo capítulo, en `keras` los `callbacks` son clases que se utilizan para realizar funciones específicas, tanto a lo largo del entrenamiento (incluyendo inicio y fin de los *batch/epoch*) como en la parte de pruebas y predicciones.
- `verbose`. El parámetro `verbose` sirve para indicar la cantidad de información que queremos recibir por consola. Tiene tres modos o tres posibles valores: `0` (no se indica nada por pantalla), `1` (muestra una barra de progreso) y `2` (muestra una línea por *epoch*). Por defecto este parámetro tiene valor `'auto'` (se utiliza el modo `1`).
- `steps_per_epoch`: Número de pasos por cada *epoch*. Por cada *epoch* se realizan N iteraciones, con N = X/número de *batch*; este número de iteraciones se puede modificar con este parámetro.

```
predict(x, batch_size=None, verbose=0)
```

El método `predict` genera las predicciones (conjunto Y) para el conjunto de entrada `x` mediante el modelo previamente entrenado sobre el cual se aplica. Además, dicho método nos permite realizar dicha predicción mediante `batches`.

```
keras.callbacks.EarlyStopping(
monitor="val_loss",
 min_delta=0,
 patience=0,
restore_best_weigths=False)
```

Dicho *callback* tiene la función de abortar el proceso de entrenamiento si el valor a monitorizar (parámetro `monitor`) no mejora (si aumenta un número menor al valor del parámetro `min_delta` se cuenta como no mejoría) despúes de un número determinado de *epoch* (indicados en el parámetro `patience`). A su vez, este método tambien dispone de un parámetro `restore_best_weights` que permite que nuestro modelo se quede con los pesos que tengan un valor más alto del parámetro `monitor` en el caso de que se aborte la ejecución por falta de mejoría del mismo.

Una vez descritos los métodos encargados de crear y configurar los parámetros de nuestro modelo, pasaremos a describir los métodos utilizados para crear cada una de las capas del mismo.

```
keras.layers.Convolution2D(filters,
                           kernel_size,
                           padding,
                          activation=None,
                          input_shape)
```

La clase `Convolution2D` se utiliza para crear una capa convolucional bidimensional, que, tal y como se ha indicado en el capítulo 3, permite extraer las características de la imagen de entrada. Para poder crear dicha capa, se necesita indicar el tamaño de entrada de la misma, utilizando el parámetro *`input_shape`*. A su vez, también será necesario configurar el número de kernels (`filters`), tamaño de los kernels (`kernel_size`), el tipo de *padding* (`padding='Same'` o `padding='valid'`) y el tipo de función de activación (`activation='relu', activation='softmax',` etc).

```
keras.layers.MaxPooling2D(pool_size)
keras.layers.GlobalAveragePooling2D()
```

Estas capas se utilizan para realizar un proceso de *pooling* (*max pooling* o *average pooling*, respectivamente) sobre los datos de entrada, con el fin de reducir su dimensionalidad. Deberemos indicar el tamaño de la ventana *pooling* (*`pool_size`*) para nuestra capa de *max pooling*. Estas capas no cuentan con pesos entrenables.

```
keras.layers.Dense(units, activation=None)
```

La clase `Dense` se utiliza para crear capas densamente conectadas. Para ello se debe indicar el tamaño del *output* o número de unidades de salida de la misma (`units`) y el tipo de función de activación que se desea emplear (`activation`).

```
keras.layers.Dropout(rate)
```

La capa de `Dropout` se utiliza para eliminar aleatoriamente un porcentaje determinado (`rate`) de sus datos de entrada para evitar problemas de *overfitting*. Esta capa no cuenta con pesos entrenables.

```
keras.layers.Flatten()
```

La capa `Flatten` , o capa de aplanado , se encarga de reducir el volumen de los datos de entrada, devolviendo como salida un vector con una sola dimension, es decir, una lista de valores. Esta capa no cuenta con pesos entrenables.

Finalmente pasaremos a describir los métodos empleados para crear y entrenar un modelo basado en el uso de SVM.

```
sklearn.svm.SVC(kernel='rbf', C=1.0,  gamma='scale')
```

La clase SVC implementa el modelo SVM utilizado para tareas de clasificación de datos.

Parámetros:

- *kernel*. Tipo de kernel empleado (por ejemplo, `'linear'`).
- *C*. Parámetro de regularización o penalización de los términos mal clasificados. Este parámetro determina cuánto queremos evitar los errores en la clasificación de cada muestra de entrenamiento y afecta directamente a la anchura del margen del hiperplano; valores más altos de *C* conformarán márgenes más pequeños, mientras que valores más pequeños de *C* conforman márgenes más amplios.
- *gamma*. El parámetro *gamma* representa el coeficiente de las funciones `'rbf'`, `'poly'` y `'sigmoid'`.

  Esta clase cuenta, a su vez, con los siguientes métodos:

  ```
  SVC.fit(X, y)
  ```

  El método `fit` inicia el entrenamiento del clasificador SVM, para el conjunto de datos de entrada dado (`X`) y su correspondiente vector de etiquetas (`y`).

  ```
  SVC.predict(X)
  ```

  El método `predict` sirve para clasificar el conjunto de muestras dado (`X`). Se emplea sobre un clasificador SVM ya entrenado.

```
sklearn.model_selection.GridSearchCV(estimator,
                                      param_grid,
                                      scoring=None,
                                      cv=None,
                                      verbose=0)
```

Al igual que el método `fit`, este método también se emplea para ajustar el modelo SVM Sin embargo, se utiliza la técnica de validación cruzada para mejorar la precisión del mismo. Además, se realizan diferentes ejecuciones con diferentes combinaciones de parámetros, con la finalidad de obtener la combinación de valores que mayor precisión obtengan. Todo esto conlleva un elevado coste computacional.

Parámetros:

- *`estimator`*. Modelo SVM sobre el cual se aplica el método.
- *`param_grid`*. Diccionario con los diferentes valores de los parámetros del modelo SVM. Dado que el objetivo es encontrar la mejor combinación de parámetros, nuestro modelo se ejecutará por cada combinación posible de los mismos, de manera consecutiva.
- *`cv`*. Número de particiones empleadas por la técnica de validación cruzada. Esta técnica consiste en particionar el conjunto de datos de entrada del modelo (X) y su correspondiente conjunto de etiquetas (Y) en N partes iguales. De esta forma, el modelo se ejecuta un total de N veces (por cada combinación de parámetros), y sobre cada una de estas ejecuciones se seleccionará una partición diferente para evaluar la efectividad del mismo, de manera que al comprobar las N particiones, nuestro modelo se adaptará mejor a los datos de entrada mejorando los resultados de la clasificación.
- *`scoring`*.Valor (o conjunto de valores) que describe la estrategia empleada para evaluar el rendimiento de nuestro modelo en el conjunto de test correspondiente, obtenido mediante la técnica de validación cruzada.
- *`verbose`*. Como se ha comentado con anterioridad, es el parámetro utilizado para manejar el número de información que se muestra por consola mientras se entrena el modelo.

```
sklearn.metrics.accuracy_score(y_true, y_pred)
```

El método `accuracy_score` se utiliza para calcular la efectividad de nuestro modelo SVM. Para ello se comparan las etiquetas que dicho modelo genera (*`y_pred`*) con las etiquetas correctas del conjunto que se quiere clasificar (*`y_true`*).

Capítulo 5 Experimentación

Como se ha comentado en el capítulo 1 de este documento, el principal objetivo de este trabajo es la implementación de un clasificador que reciba como entrada la imagen de un perro y sea capaz de determinar su raza de forma correcta con la mayor efectividad posible.

En esta sección se hará uso tanto de los conceptos teóricos explicados con anterioridad en el capítulo 3 así como de los métodos descritos en el capítulo 4.

Nuestra tarea inicial consistirá en la búsqueda de un conjunto de datos representativo con imágenes de perros de diferente raza que nos permitan entrenar adecuadamente nuestro modelo de Aprendizaje Automático elegido.

Para ello se recurrirá al uso de un *dataset* con 20580 imágenes de perros de diferentes razas extraído de la web de Kaggle [7], a su vez basado en el famoso *dataset* de perros de Stanford (*Stanford Dogs Dataset*) [8].

El archivo zip descargado desde la web mencionada contiene dos directorios de imágenes y dos ficheros csv:

- Directorio *train*. Contiene 10222 imágenes.
- Directorio *test*. Contiene 10357 imágenes.
- Fichero *labels.csv*. Lista con el nombre de cada imagen del directorio de entrenamiento y su correspondiente etiqueta o raza en formato CSV.
- Fichero `sample_submission.csv`. Plantilla donde se muestra un ejemplo de salida para la clasificación correcta de las muestras del directorio de test (para cada una de las imágenes de test se nos pide indicar la probabilidad que tiene la misma de pertenecer a cada una de las diferentes razas).

La existencia del directorio `test` y el fichero `sample_submission` se debe a que originalmente en la web de Kaggle se nos pide la construcción de un modelo capaz de clasificar las imágenes de test en el formato pedido. En este proyecto no tenemos ningún interés en clasificar el conjunto de test; más bien nuestro objetivo, como ya se ha mencionado, es la creación de un sistema automático capaz de clasificar nuestras imágenes de forma precisa. Esto requiere tener las etiquetas de las muestras que se usan tanto para entrenar el clasificador como para estimar su rendimiento, algo que no está disponible para las imágenes de test proporcionadas. Es por ello que tanto el directorio `test` como el fichero `sample_submission` no van a ser utilizados a lo largo de la experimentación.

Así pues, utilizaremos las imágenes de entrenamiento para construir nuestro modelo de Aprendizaje Automático, y haremos uso del fichero `labels.csv` para obtener la etiqueta

[7] https://www.kaggle.com
[8] http://vision.stanford.edu/aditya86/ImageNetDogs/

o raza de cada imagen de dicho fichero. A continuación vamos a analizar con más detalle cada uno de estos archivos.

Por un lado tenemos el archivo `labels.csv`, con el identificador único (id) de cada imagen de entrenamiento contenida en el directorio `train` y su correspondiente raza:

```
id,breed
000bec180eb18c7604dcecc8fe0dba07,boston_bull
001513dfcb2ffafc82cccf4d8bbaba97,dingo
001cdf01b096e06d78e9e5112d419397,pekinese
00214f311d5d2247d5dfe4fe24b2303d,bluetick
0021f9ceb3235effd7fcde7f7538ed62,golden_retriever
…
```

Como hay un total de 10222 imágenes en el directorio de entrenamiento, el fichero `labels.csv` tendrá un total de 10222 filas (además de la cabecera). Respecto a las razas, contamos con un total de 120 razas, listadas en el Anexo 2 de este documento.

Por otra parte, en el directorio `train` nos encontramos las imágenes de entrenamiento. Dichas imágenes toman como nombre el id (indicado en el csv `labels`) acompañado de el formato de la imagen (p.ej: `000bec180eb18c7604dcecc8fe0dba07.jpg`).

Imagen 27. Ejemplo de imágenes del conjunto de entrenamiento usado.

Como observamos en la imagen 27, no todas las imágenes tienen la misma calidad y luminosidad, cosa que, como se verá más adelante en este mismo capítulo, nos ocasionará problemas en algunos modelos de *Machine Learning*.

Por otra parte el tamaño de cada imagen también varía; por ello, tal como hemos visto en el apartado 3.2.5, será necesario normalizar los *inputs* de nuestro modelo, con la finalidad de evitar problemas de escalado que nos introduzcan retrasos y problemas en la computación.

Además del tamaño y la calidad de las imágenes, el número de imágenes por clase también puede llegar a ser un factor determinante que puede llegar a causarnos problemas, pero tal y como observamos en la imagen 28, la distribución de las muestras por clase no nos representará ningún problema.

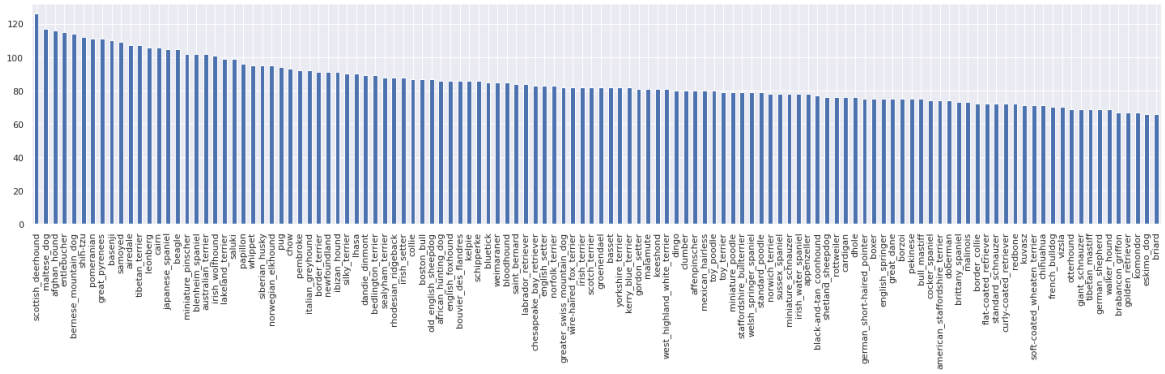


Imagen 28. Muestras por clase.

Una vez hemos estudiado nuestro conjunto de imágenes de entrenamiento, el siguiente paso a realizar será la construcción del vector de *inputs* o datos (X) y su correspondiente vector de etiquetas (Y), que nos servirán para indicar a nuestro modelo qué etiqueta esperamos por cada *input* dentro de nuestros datos de entrenamiento. Como hemos indicado en el apartado 3.2.2 de este documento, el modelo que usemos tratará de predecir la etiquetas de los nuevos datos mediante la generalización de cada X visto en el entrenamiento y su correspondiente Y.

5.1 Etapas de la implementación

Antes de pasar a profundizar en la implementación de cada uno de los diferentes modelos vamos a explicar el esquema general a seguir mostrado en la imagen 29.

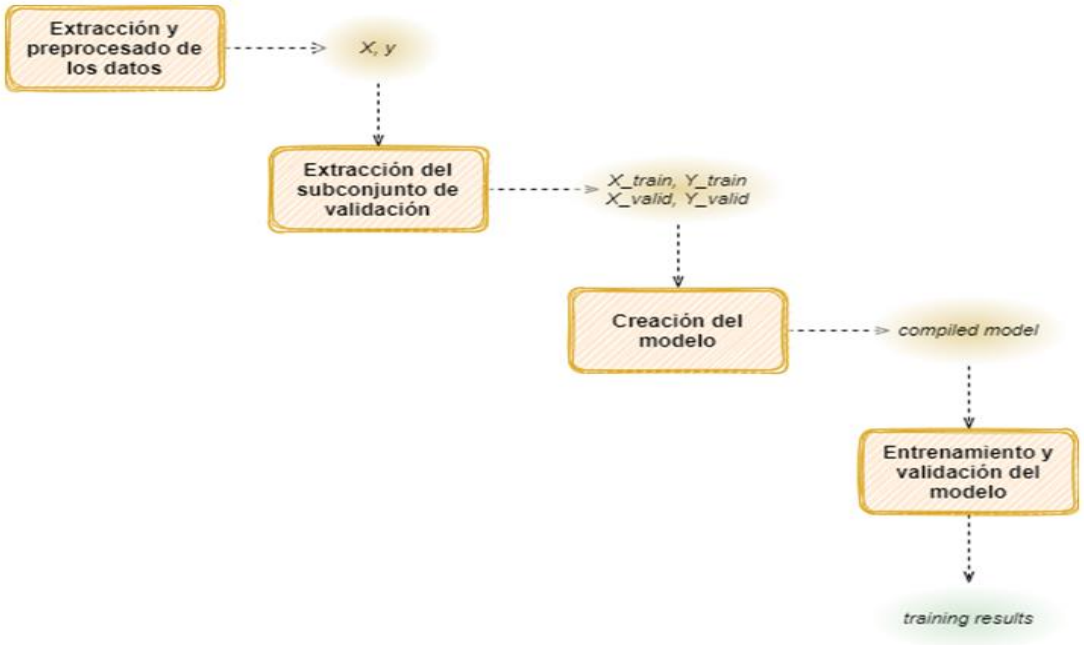


Imagen 29. Esquema general empleado para el desarrollo de los modelos.

Etapa 1. Extracción y procesado de los datos

El objetivo de esta primera etapa es la creación del conjunto X (que representa las imágenes de entrenamiento) y su correspondiente conjunto Y (que indica la clase a la que pertenecen las mismas) utilizados como *input* de nuestro modelo.

El conjunto X debe describir las imágenes con el mayor detalle posible para permitir al clasificador realizar su función y asociar las características de cada muestra X con su correspondiente etiqueta Y. Para ello se puede recurrir a almacenar cada uno de los píxeles de la imagen en formato RGB (el color en este caso es importante dado que puede ser un factor diferencial entre distintas razas) o a almacenar un conjunto determinado de características de cada una de dichas imágenes, realizando un preproceso previo sobre las mismas. El contenido del conjunto X puede afectar a la efectividad del modelo, como se verá más adelante en este documento.

Por otra parte, el conjunto Y debe contener la clase de cada muestra del conjunto X en formato de lista. Cabe destacar que dentro de la lista Y no almacenaremos las etiquetas

en formato de cadena de texto, sino que convertiremos cada etiqueta a formato numérico y, dependiendo del modelo empleado, se requerirá también transformar dicha etiqueta numérica a formato *one-hot*.

Tanto para la creación del conjunto X como el conjunto Y utilizaremos el fichero `labels.csv` con el id de cada imagen (que nos permitirá identificarla dentro del directorio `train`) y su etiqueta.

Etapa 2. Extracción del conjunto de validación

Una vez obtenidos los conjuntos X e Y, pasaremos a crear el conjunto de validación o test. La creación de dicho conjunto se basa en la extracción de un porcentaje de muestras del conjunto X (y sus correspondientes etiquetas del conjunto Y) con la finalidad de evaluar la efectividad de nuestro modelo, comparando las etiquetas reales de dicho subconjunto con las etiquetas obtenidas por el modelo después de realizar el proceso de entrenamiento.

Es importante resaltar que el porcentaje de datos empleado para crear este conjunto debe ser el adecuado, evitando extraer un porcentaje demasiado elevado (cosa que perjudicaría al entrenamiento del modelo por la falta de datos) o un porcentaje demasiado pequeño (cosa que perjudicaría al proceso de evaluación del modelo dado que no contamos con datos suficientes para evaluar su capacidad de generalización). Por ello, utilizaremos un tamaño de evaluación de 20% (`0.2`).

Etapa 3. Creación del modelo

Una vez ya tenemos todos los datos listos, se especificará la estructura de nuestro modelo, indicando el tipo del modelo, las estructuras utilizadas (redes neuronales convolucionales, máquinas de vector soporte y/o arquitecturas CNN ya definidas), el número de capas que conformarán al mismo y los parámetros de cada una.

Como es de esperar esta etapa variará en función de cada modelo.

Etapa 4. Entrenamiento y validación del modelo

Finalmente, el útlimo paso a realizar es el entrenamiento del modelo. En este paso entrenaremos nuestro modelo y evaluaremos su efectividad.

A continuación, pasaremos a explicar cada uno de los modelos utilizados en la experimentación y el proceso seguido para implementar cada una de las etapas anteriormente indicadas.

5.2 Uso de máquinas de vectores soporte

El primer modelo de aprendizaje automático que vamos a construir se basará en el uso de máquinas de vectores soporte (SVM). Como se ha visto en el apartado 3.1 de este documento, el entrenamiento de un modelo basado en SVM se basa en la creación de un hiperplano separador de las diferentes clases o etiquetas contenidas en el vector Y. Es decir, a partir de las imágenes de entrada "X" y de sus etiquetas "Y", el modelo agrupará las muestras del vector X según sus etiquetas con el objetivo de crear un hiperplano que las separe y que sirva para poder clasificar nuevas muestras no vistas por nuestro modelo. En la imagen 30 se observa el esquema empleado para la creación y el entrenamiento de este primer modelo.

Tal y como se observa en dicha ilustración, el primer paso a realizar es la lectura de las muestras (imágenes) y sus correspondientes etiquetas o clases (razas). Para llevar a cabo dicha tarea se recurre al uso del método `images_to_array` (Apéndice 1, Figura 1), método que se encarga de recorrer el directorio `train_dir` en busca de las imágenes de entrenamiento (mediante el identificador de cada imagen almacenado en la columna `id` del fichero `labels.csv`) con el objetivo de almacenar las mismas en el vector X, a la vez que almacenamos la etiqueta correspondiente (indicada en la columna `breed` del fichero `labels.csv`) en el vector Y.

Dado que este primer modelo utiliza máquinas de vectores soporte como herramienta principal para la creación del clasificador, necesitaremos un conjunto de características por cada una de las imágenes, tal y como se ha explicado en el apartado 3.1. Por ello, en nuestro método `images_to_array` se necesitará aplicar la función `flatten` sobre cada muestra del conjunto X, convirtiendo la matriz de píxeles resultado de la lectura de cada imagen en el vector de características deseado. En este caso se utilizará un tamaño de 85x85 pixeles, lo que nos dará como resultado un vector X con 21.675 características por imagen (85 x 85 x 3). Por otro lado, el vector Y se utiliza para almacenar la etiqueta de cada imagen, por lo tanto tendremos un valor numérico por cada una de las 10222 muestras.

Una vez obtenidos los vectores X e Y, el siguiente paso a realizar será la extracción del subconjunto de validación. Esta labor será desempeñada por el método `train_test_split` (Apéndice 1, Figura 4), que recibirá como entrada dichos vectores y nos devolverá los conjuntos `X_valid` e `Y_valid` (empleados para la fase de validación) y `X_train` e `Y_train` (empleados para la fase de entrenamiento). Para crear los conjuntos de validación, se agrupan las muestras del conjunto X mediante su etiqueta indicada en el conjunto Y, de manera que se extrae un mismo porcentaje (20%) de muestras de cada etiqueta o clase, dando como resultado un subconjunto `X_valid` lo suficientemente representativo, con imágenes de perros de todas las razas.

Tras realizar la extracción del subconjunto de validación ya tendremos listos todos los datos de entrada de nuestro modelo y por tanto la siguiente fase será la fase de entrenamiento y validación del mismo. Este modelo está conformado únicamente por

una capa SVM que será la encargada de implementar el clasificador de imágenes deseado, generando un hiperplano separador de clases a partir de las características de cada imagen (en este caso solo se tiene en cuenta el color de los píxeles en formato RGB) almacenadas en el vector *X_train* y las correspondientes etiquetas almacenadas en el vector *Y_train*. No obstante, debido a la simpleza de este primer modelo, se empleará la técnica de validación cruzada (véase apartado 4.2.3) y se entrenará dicho modelo con varias combinaciones de parámetros con el fin de obtener mejores resultados. Así pues, para llevar a cabo esta tarea utilizaremos el método *GridSearchCV* (véase apartado 4.2.3) con 2 tipos de kernels diferentes, 4 valores posibles para el parámetro '*C*' y otros 4 valores posibles para el parámetro 'gamma' (Apéndice 1, Figura 5). Además, también será necesario indicar de forma paramétrica el número de particiones empleado para la validación cruzada, en este caso 4.

Tal y como se observa en la imagen 30, el entrenamiento del modelo consistirá en encontrar la combinación de parámetros que mejor se adapte a nuestros datos de entrenamiento y que, por tanto, nos ofrezca una mayor precisión. Por consiguiente, al finalizar la fase de entrenamiento obtendremos el conjunto de parámetros más efectivo sobre los datos de entrenamiento y utilizaremos dicha combinación para predecir las etiquetas de las imágenes de validación almacenadas en el conjunto *X_valid*, utilizando para ello el método *predict* sobre nuestro modelo. Finalmente compararemos las etiquetas obtenidas por nuestro modelo *Y_pred* con las etiquetas reales de los datos de validación (*Y_valid*), para determinar la efectividad del modelo.

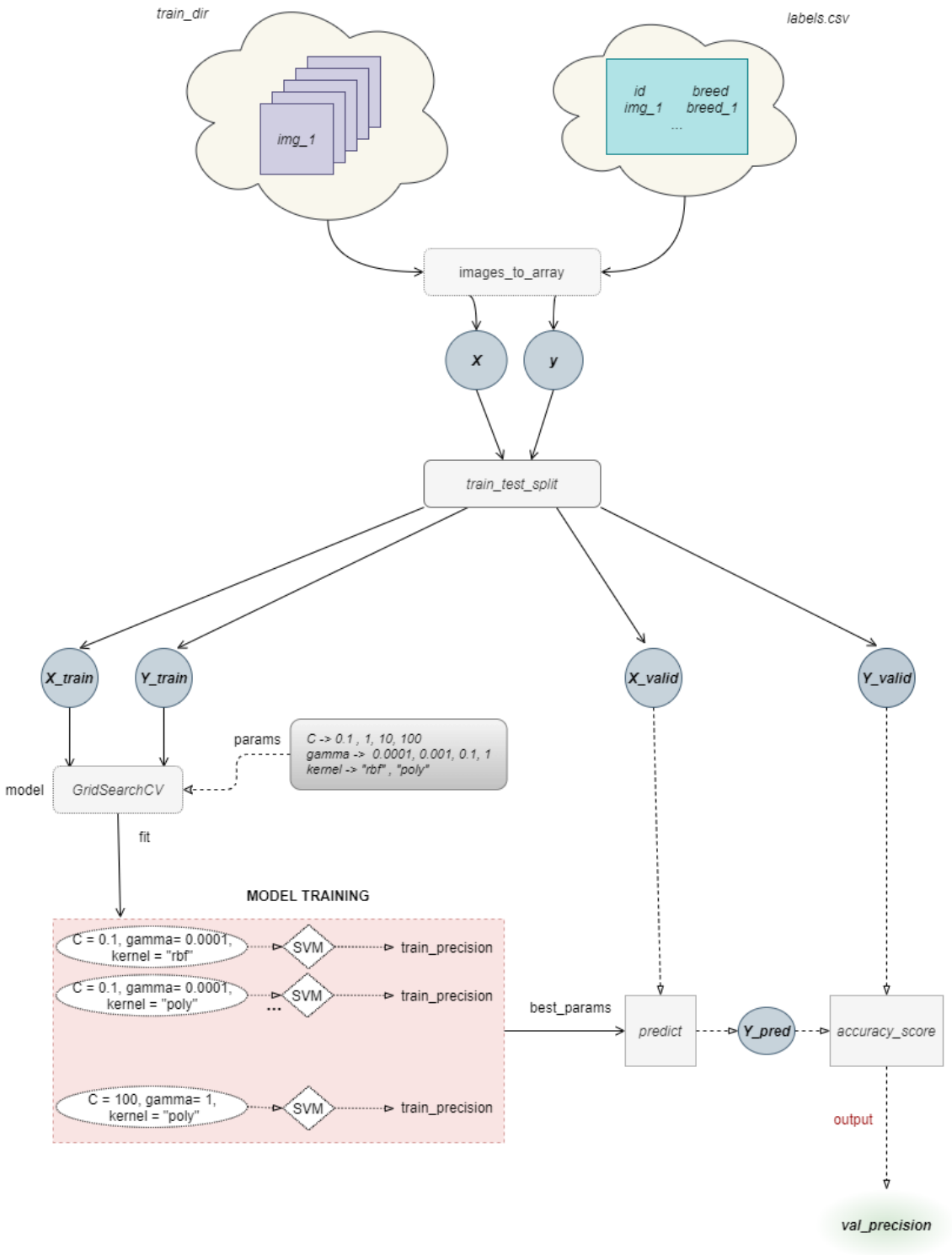


Imagen 30. Esquema de creación y entrenamiento del modelo SVM.

La precisión obtenida para este primer modelo es de **5.58%.**

Como hemos explicado con anterioridad en este mismo apartado, este primer modelo trata de diferenciar cada raza utilizando como característica de cada imagen una lista

con los colores de cada píxel. Sin embargo, como se observa, este modelo no resulta ser muy eficaz. Las causas de esta baja efectividad son principalmente dos:

- Por un lado, las imágenes, además de contener los colores de cada perro, contienen también los colores del fondo de la imagen y de otros elementos que aparecen en la misma (como personas o muebles), como se observa en la imagen 27. Además de esto, la calidad de las imágenes no es la adecuada, dado que algunas presentan una mala iluminación o unos colores muy saturados, lo que hace que la lista de píxeles extraídos por cada imagen no sea de gran utilidad como elemento diferencial de cada raza.

- En adición al motivo mencionado, también podemos señalar que el uso exclusivo del color de cada imagen con el objetivo de diferenciar cada raza (aunque las imágenes tuvieran una máxima calidad y no hubiera elementos que interfirieran) puede no ser suficiente para implementar un clasificador válido para las 120 razas, dado que no solo por el color se distinguen las razas, ya que también hay que tener en cuenta factores como las formas.

5.3 Entrenamiento de una CNN desde cero

Como se ha observado con anterioridad, no se pueden utilizar solo los colores de cada píxel de la imagen como único factor diferencial de cada raza. En esta ocasión, además de utilizar el color como factor diferencial trataremos también de identificar patrones en las imágenes de cada clase con el objetivo de poder reconocer formas entre las distintas razas de perros como las orejas, el hocico, la cola, etc. Es por ello que este segundo modelo hará uso de uno de los algoritmos más utilizados para la clasificación de imágenes; utilizaremos redes neuronales convolucionales.

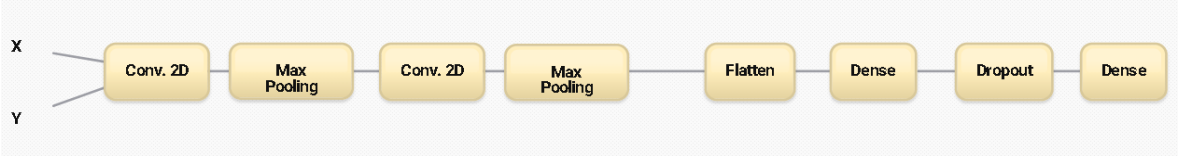


Imagen 31. Estructura CNN básica.

El segundo modelo con el que vamos a experimentar contará con la estructura indicada en la imagen 31, donde:

- El uso de una capa convolucional bidimensional (`Conv 2D`) se utiliza con el objetivo de reconocer patrones en la imagen y extraer sus características, mientras que el uso de la capa `Max Pooling` nos permite reducir la dimensionalidad espacial extraída por la capa convolucional (no se reduce el volumen o la profundidad del *input*) sin perder información espacial importante, de manera que obtenemos un mejor coste computacional.
- La capa `Flatten` o capa de aplanado se utiliza para reducir el volumen de las características extraídas, obteniendo como resultado un único vector de características por imagen.

- La capa `Dense` o capa densamente conectada es la capa empleada por las redes neuronales artificiales y, junto con la capa de `dropout`, desempeña las labores de clasificación.
- La capa de `Dropout` elimina un porcentaje de los datos que recibe como entrada de manera aleatoria. Normalmente se emplea con el objetivo de evitar problemas de sobreentrenamiento.

El procedimiento empleado para crear y entrenar este segundo modelo, se detalla en la imagen 32.

Como se observa en dicha ilustración, al igual que en el modelo anterior, el primer paso consiste en leer las imágenes y sus etiquetas y almacenarlas en los vectores X e Y, respectivamente. Para realizar dicha tarea volveremos a hacer uso del método `images_to_array` (Apéndice 1, Figura 2); sin embargo, deberemos tener en cuenta varios aspectos importantes en relación al modelo anterior:

- El tamaño de las imágenes en esta ocasión será de 224 x 224, con el objetivo de facilitar a nuestra red la identificación de los patrones en cada imagen (aunque esto también supondrá un aumento en el coste computacional del modelo).
- Las redes neuronales convolucionales reciben como entrada una matriz de píxeles por cada imagen; por lo tanto, será menester que las muestras almacenadas en el vector X tengan un formato matricial y, en consecuencia, no será necesario utilizar la función `flatten` en el método `images_to_array`. Dado que las imágenes tienen tamaño 224 x 224 y se leen en formato RGB (3 canales de color), cada matriz almacenada en el vector X tendrá un tamaño de 224 x 224 x 3.

El siguiente paso a realizar es la extracción del subconjunto de validación y, para ello, volveremos a utilizar el método `train_test_split` (Apéndice 1, Figura 4) sobre los vectores X e Y. En adición, deberemos aplicar la función `to_categorical` sobre los conjuntos `Y_valid` e `Y_train` obtenidos mediante dicho método, para convertir las etiquetas de cada imagen a formato `one-hot`, tal y como requiere la red.

Con el fin de evitar problemas de memoria, realizaremos un entrenamiento por *batches* o lotes de imágenes, de tal manera que se irán cargando los *batches* en memoria de forma secuencial durante el proceso de entrenamiento (véase apartado 3.2.6). Además, se aplicará un conjunto de transformaciones aleatorias sobre nuestras imágenes de entrada por cada *epoch* de entrenamiento, aumentando así la capacidad de generalización de nuestro modelo.

Por lo tanto, tal y como se observa en la imagen 32, una vez listos los conjuntos de entrenamiento y de validación y, antes de pasar a la fase de entrenamiento del modelo, definiremos dos miembros de la clase `ImageDataGenerator` para crear los generadores `train_generator` y `valid_generator`. Estos generadores nos permitirán indicar tanto las transformaciones aleatorias que queremos emplear sobre las imágenes (rotaciones, translaciones, cambios en el brillo de la imagen, etc.) como el proceso de escalado que pretendemos aplicar sobre las mismas. Las transformaciones

aleatorias se aplicarán solo sobre el conjunto de entrenamiento `train_generator` dado que el conjunto de validación debe permanecer invariable ya que se empleará únicamente para calcular la precision obtenida al finalizar la etapa de entrenamiento (como se verá con posterioridad). Sin embargo, el escalado se aplicará en ambos conjuntos y consistirá en dividir los datos de entrada de cada conjunto por el valor 255, de manera que cada píxel de la imagen almacenado en formato entero con un rango de 0 a 255 (modo RGB), pasará a ser almacenado en formato decimal con rango de 0 a 1. Este proceso de normalización ayudará a disminuir el tiempo de convergencia de nuestro modelo (véase apartado 3.2.5)

Por otra parte, para poder realizar el entrenamiento mediante *batches*, utilizaremos el método `flow` sobre ambos generadores, indicando los conjuntos sobre los cuales vamos a extraer las imágenes y el tamaño del *batch* para cada uno (Apéndice 1, Figura 6). En este caso utilizaremos un tamaño de 32 imágenes por *batch*.

Una vez definidos ambos generadores pasaremos a definir nuestro modelo. Tal y como se ha descrito con anterioridad, nuestro modelo contará con dos bloques conformados por la concatenación de capas convolucionales y capas de *max pooling* al principio del mismo con el objetivo de extraer características de las imágenes, y otro bloque conformado por capas densamente conectadas, situado al final del mismo, para clasificar las imágenes.

Las capas convolucionales contarán con 64 filtros de tamaño 4x4, con una función de activación '`relu`' y con la opción de *padding* habilitada (véase apartado 4.2.3). Además, utilizaremos ventanas de tamaño 2x2 para realizar el proceso de *max pooling*. Todo esto irá seguido de una función `flatten` que se encargará de reducir la dimensionalidad de la matriz de características extraída por las capas anteriores, generando un vector de características por cada imagen (véase Apéndice 1, Figura 7).

Por otra parte, las capas ANN finales del modelo contarán con un total de 120 unidades de salida (correspondiente al número de clases), dado que nuestro modelo devolverá un vector en formato *one-hot* por cada imagen de entrada, donde se indicará la etiqueta con mayor probabilidad calculada para dicha muestra. La primera de estas capas densamente conectadas utilizará una función de activación de tipo '`relu`' para filtrar las características más importantes, mientras que la segunda utilizará una función de activación de tipo '`softmax`', con el objetivo de devolver como output el vector en formato *one-hot* mencionado.

Finalmente, indicaremos la función de optimización ('`Adam`') y la función de pérdida ('`categorical_crossentropy`') que utilizaremos en la etapa de entrenamiento.

Como se ha explicado en el capítulo 3.3, el entrenamiento de una red neuronal convolucional se basa en el aprendizaje de los pesos de los kernels de cada una de sus capas. Este proceso requiere de varias iteraciones dependientes entre sí (*epoch*) y, además, en este caso, se realiza por *batches*. En la imagen 33 se detalla el procedimiento seguido en esta etapa de entrenamiento.

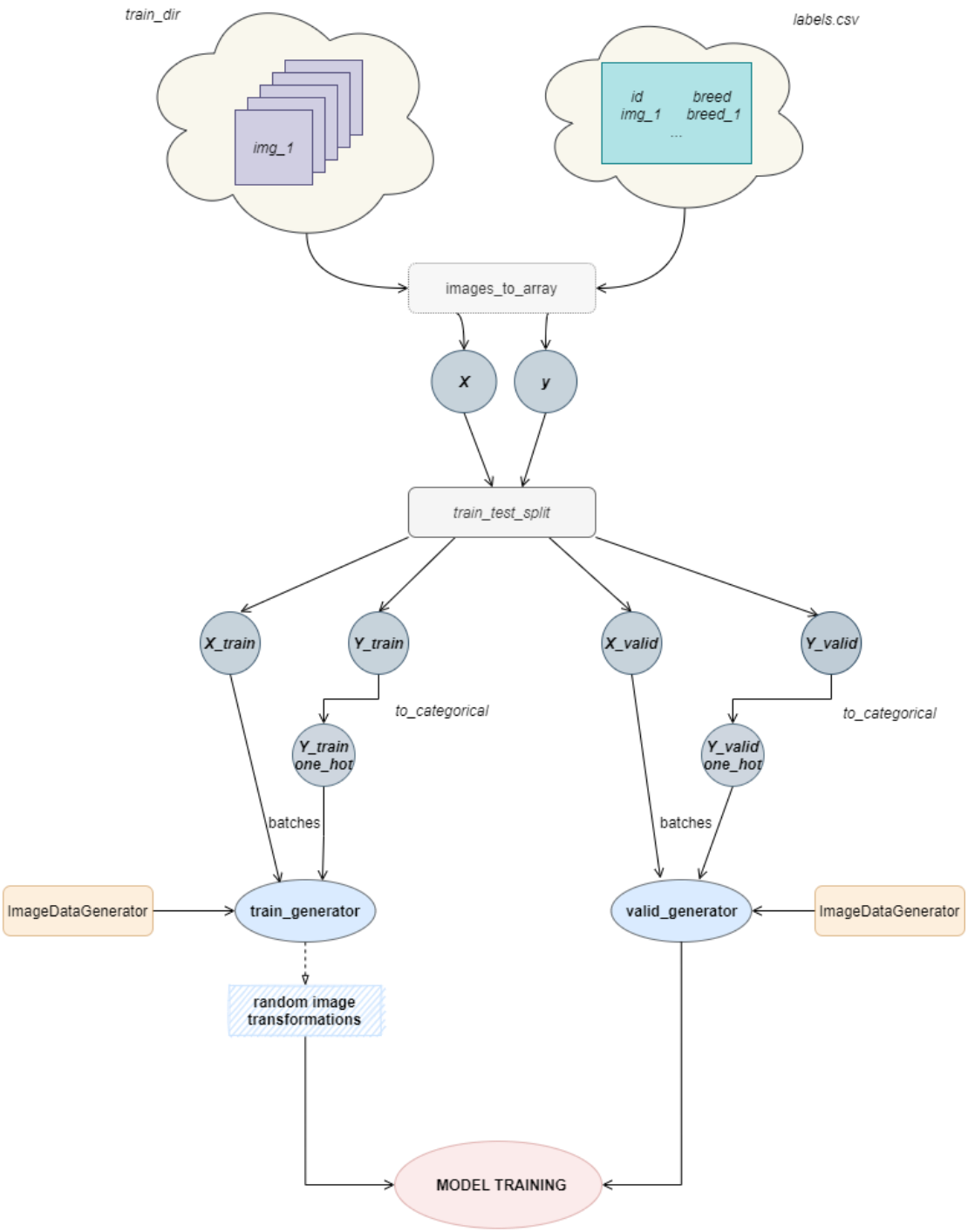


Imagen 32. Esquema empleado para la creación y entrenamiento de la red CNN.

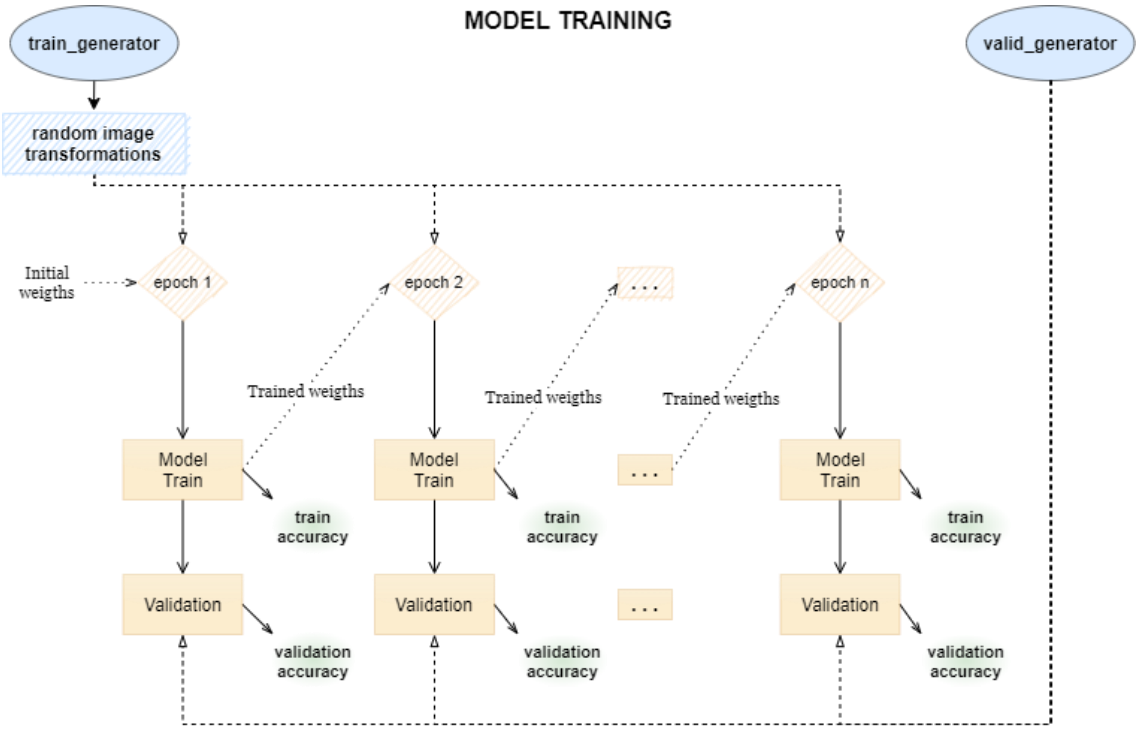


Imagen 33. Descripción de la fase de entrenamiento de los modelos basados en el uso de CNNs.

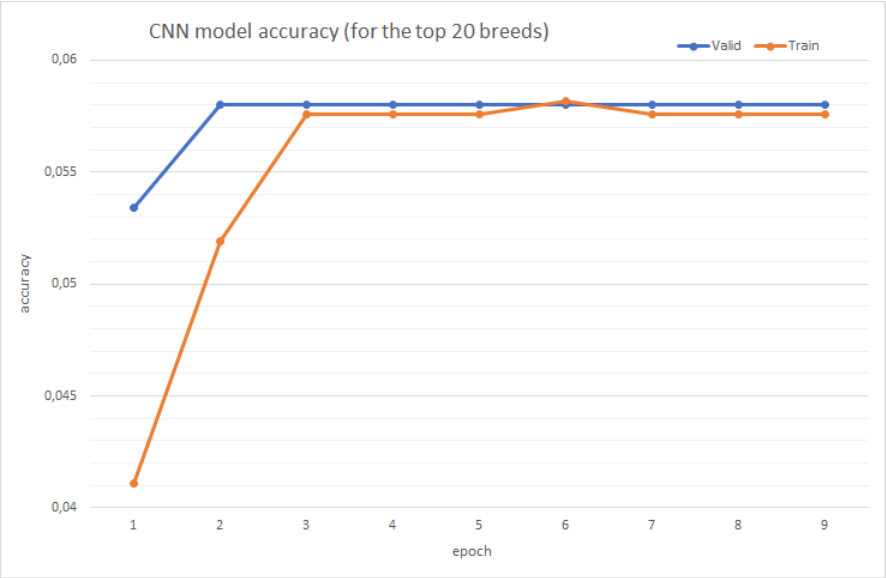
Como se observa en esta imagen, el entrenamiento se realiza para una cantidad de *epoch* determinada. En cada *epoch* o iteración, nuestra red convolucional recibe como entrada un conjunto de *batches* de imágenes de entrenamiento sobre los cuales se les ha aplicado unas transformaciones aleatorias. Esto implica que por cada *epoch* tendremos un conjunto de datos de entrada diferentes (aunque todos parten de un conjunto común de entrenamiento). A su vez, cada iteración tiene una fase de entrenamiento y otra fase de validación:

- La fase de entrenamiento recibe como entrada los lotes de imágenes provenientes del conjunto de entrenamiento y un conjunto de pesos inicial, que se modificará en base a dichos datos de entrada. Tras ajustar los pesos a los datos de entrada, éstos se transferirán como pesos inicales en la siguiente iteración o *epoch*, de manera que se volverán a recalcular atendiendo a los datos de entrada de esta iteración y se transferirán a la siguiente, repitiendo este proceso de forma sucesiva.
  Tras el cálculo del conjunto de pesos de cada iteración, la red tratará de clasificar los datos que ha recibido como entrada en esta misma iteración y comparará el resultado de la clasificación con el conjunto de etiquetas de entrenamiento reales, obteniendo así la precisión sobre los datos de entrenamiento (`train accuracy`).
- La fase de validación se aplica tras finalizar la fase de entrenamiento y antes de transmitir los pesos calculados al *epoch* siguiente. En esta fase se clasifican las muestras de validación contenidas en el generador `valid_generator` y se compara el resultado obtenido con las etiquetas reales de dichos datos, obteniendo así la precisión sobre los datos de validación (`valid accuracy`).

Esta fase obtiene un cálculo más realista de la precisión real de nuestro modelo, dado que los datos utilizados para calcular dicha precisión no son los mismos que se han utilizado para calcular los pesos del modelo (aunque guardan cierta relación con dichos datos, pues parten de un mismo conjunto de imágenes X).
Por lo tanto, el objetivo principal del entrenamiento de la red será calcular el conjunto de pesos que ofrezca una mayor precisión en los datos de validación, sin llegar a sobreentrenar el modelo (véase capítulo 3.2.7).
Cabe destacar que tanto en la fase de entrenamiento como en la fase de validación también se calculan los valores de la función de pérdida, lo que nos dará mayor información acerca del error de clasificación en las muestras (véase apartado 3.2.3).

Para el caso que nos ocupa, iniciaremos el entrenamiento de la red mediante el método `fit_generator`, con un total de 9 *epoch* y un conjunto de pesos inicial generado de forma aleatoria (véase Apéndice 1, Figura 7).

El entrenamiento de una CNN desde cero es una tarea computacionalmente costosa, por lo tanto en este segundo modelo se optará por realizar una primera ejecución eligiendo tan solo las muestras pertenecientes a las 20 clases con mayor cantidad de datos y acto seguido se tratará de ejecutar el modelo para el conjunto de imágenes de entrada completo.



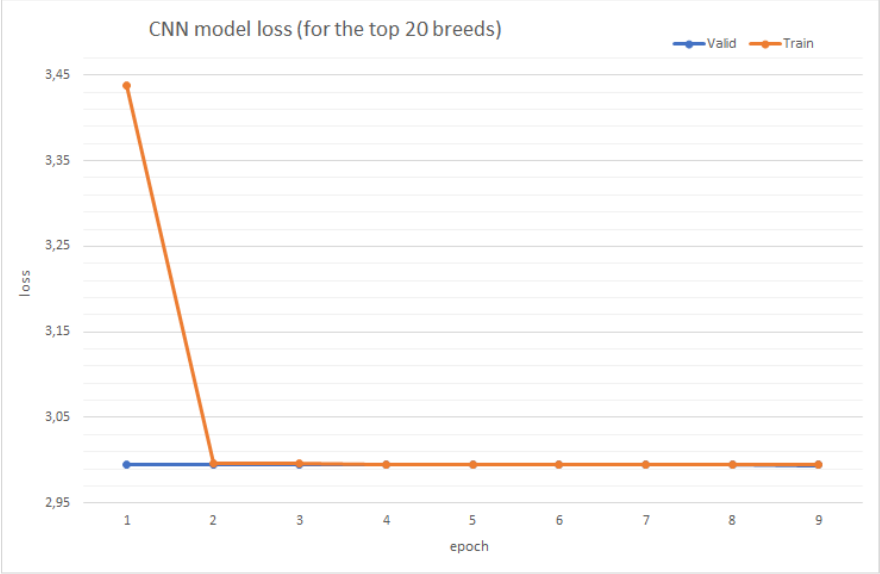


Imagen 34. Resultados de la red CNN entrenada para las 20 razas con mayor número de muestras.

Como se observa en la imagen 34, tras entrenar nuestro modelo para las 20 razas con mayor cantidad de datos, apenas se obtiene mejoría en el valor de la función de pérdida para los datos de entrenamiento; lo mismo ocurre con la función de pérdida para los datos de validación. Ambas funciones convergen en el valor **2.995**. Además, sucede lo mismo con el valor de la precisión para ambos conjuntos, obteniendo una precisión del **5.8%** para los datos de validación y una precisión de 5.76% para los datos de entrenamiento.

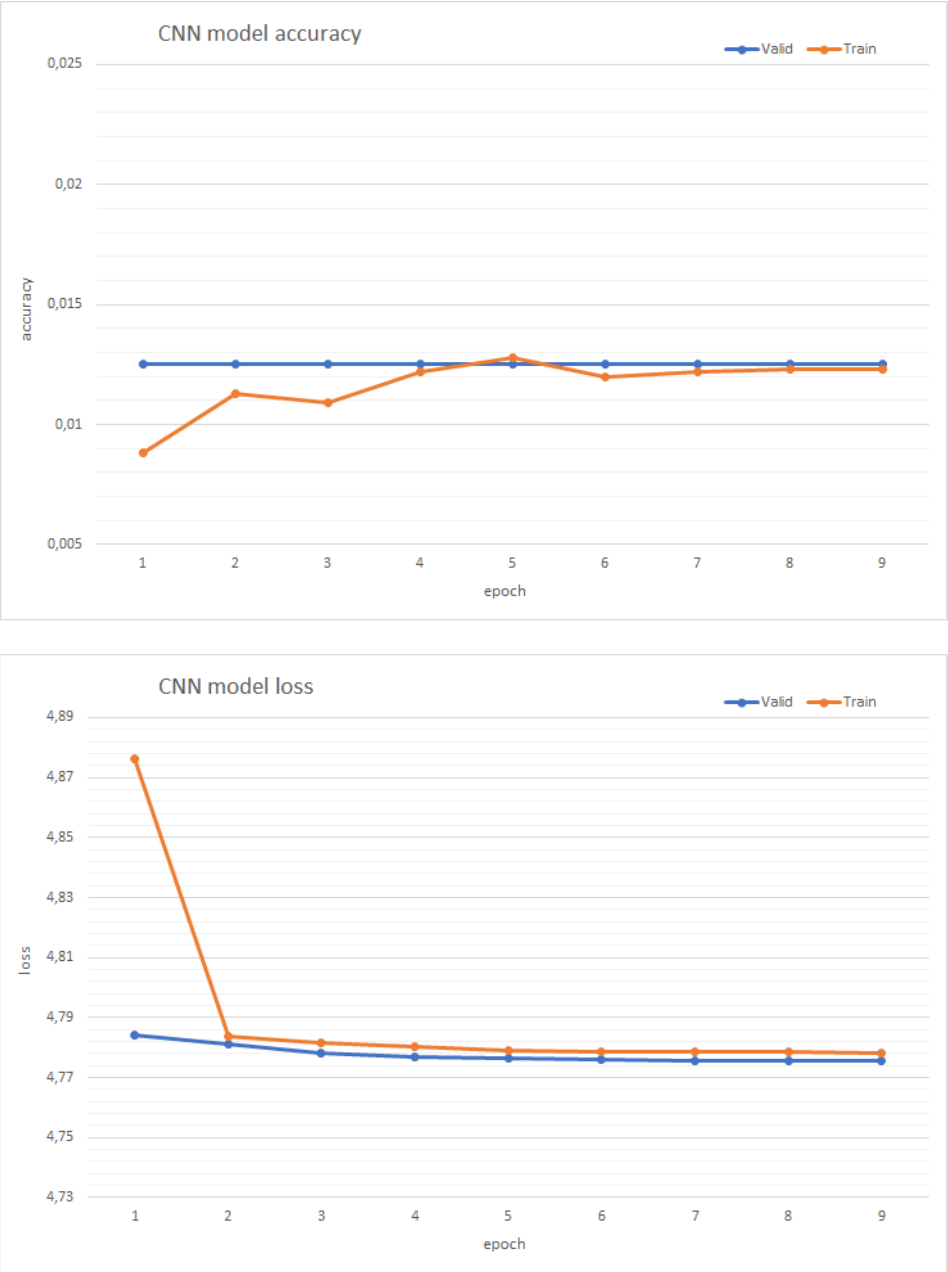


Imagen 35. Resultados de la red CNN entrenada para el conjunto completo de razas.

Como se observa en la imagen 35, sucede lo mismo cuando se entrena la red para todas las muestras disponibles. Por ello se puede afirmar que el modelo no es capaz de aprender las características de las imágenes con éxito. Los motivos principales que han llevado a este modelo a obtener resultados tan bajos son dos:

1 Calidad de los datos de entrada. Como se ha observado en la imagen 24, la calidad y la luminosidad de las imágenes no es igual para todo el conjunto de datos X; hay imágenes con mala resolución que no permiten a nuestro modelo CNN reconocer con éxito las características de cada clase y además introducen cierto ruido en la red, haciendo aún más difícil que una red entrenada desde cero (sin los pesos preentrenados) pueda realizar las labores de clasificación de forma esperada.

2 Volumen de datos de entrada. No hay suficiente cantidad de datos como para que nuestra CNN sea capaz de generalizar con éxito los tipos de entradas X y sus correspondientes salidas Y.

Además de los motivos comentados con anterioridad, la simpleza del modelo puede tener también un efecto negativo en los resultados de la red, pudiendo aumentar su efectividad al añadir más capas y aumentar la profundidad de la misma; pero debido al alto coste computacional que esto conlleva y a los problemas ya comentados se ha decidido elaborar un primer modelo de CNN sencillo que compruebe qué magnitud de resultados podemos esperar. En este caso son demasiado bajos como para añadir más capas, aumentar el coste y realizar otras optimizaciones, dado que aunque se realizaran dichos cambios, no se esperan grandes aumentos en la precisión total obtenida por el modelo.

5.4 Entrenamiento basado en *transfer learning*

Como hemos visto, el uso de modelos basados en SVM (aplicados sobre las imágenes sin preproceso) y el entrenamiento de una arquitectura CNN desde cero no resultan ser efectivos para el caso que nos ocupa; es por ello que en este apartado pasaremos a experimentar con modelos basados en el uso de *transfer learning*.

Mediante el empleo de la técnica de *transfer learning* se pretende dar solución a los problemas de insuficiencia de datos para el entrenamiento de arquitecturas CNN complejas desde cero. Además, se pretende también solventar o aminorar los problemas relacionados con la calidad de los datos de entrada.

Como se ha estudiado en la sección 3.5 de este documento, la idea principal del *transfer learning* es el uso del conocimiento adquirido en la resolución de una tarea con el fin de resolver una tarea similar. Partimos de un conjunto de entrenamiento B extenso con diferentes imágenes y clases que a su vez contienen o guardan relación con las clases del *dataset* A original que se pretende clasificar. Se utiliza una arquitectura CNN compleja con el objetivo de clasificar los datos del conjunto B, de manera que se obtienen unos pesos como resultado que permiten que los kernels de dicha red sean capaces de extraer los patrones comunes entre las imágenes de la misma clase, actuando como extractores de características. Aprovechando la similitud entre los conjuntos A y B, se pueden utilizar los pesos de la CNN ya entrenada y tratar así de extraer las características de las imágenes del conjunto A que nos serán de gran utilidad para realizar su clasificación.

La efectividad de dicha técnica viene determinada tanto por la efectividad de la arquitectura CNN empleada para clasificar el *dataset* B como por el grado de similitud de los conjuntos A y B. Este último factor es determinante, dado que no sirve de nada utilizar la red entrenada para el conjunto B si éste no guarda relación alguna con el conjunto A, debido a que la red será capaz de reconocer los patrones de las imágenes de las clases de B pero no podrá reconocer los patrones de las imágenes de las clases de A, dado que dichos patrones no estaban presentes en las imágenes que se han

utilizado para entrenar dicha red. Así pues, para evitar este problema se busca que el conjunto B sea una buena generalización del conjunto A.

Para el caso que nos ocupa, pretendemos utilizar un conjunto B cuyas imágenes y clases guarden relación con nuestro problema original de clasificación de razas caninas. Para ello utilizaremos el famoso *dataset* de ImageNet [9] un *dataset* extenso de uso general con un total de 1000 clases diferentes, entre ellas 118 pertenecientes a distintas razas de perros. En este caso, ambos *datasets* tienen clases en común y no tendremos problemas de generalización y reconocimiento de patrones en nuestro conjunto A.

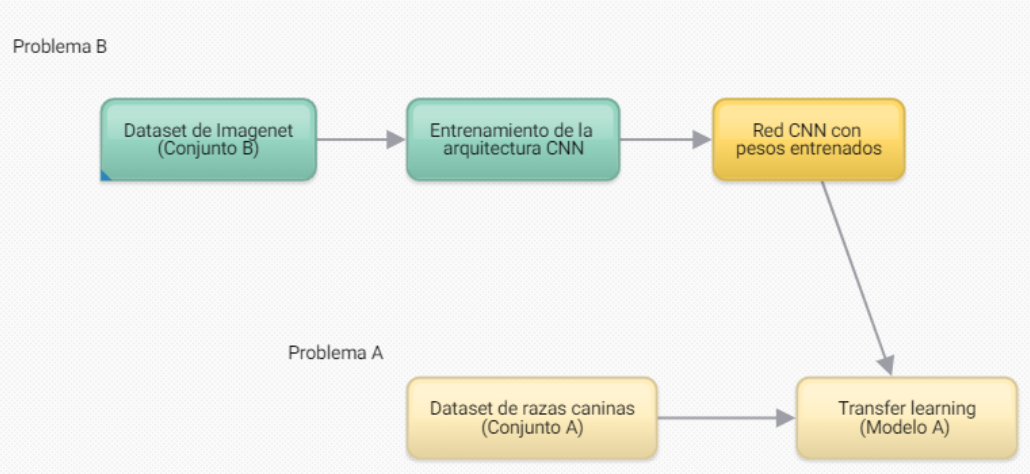


Imagen 36. Esquema de nuestro modelo de *Machine Learning*.

Una vez elegido el *dataset* B que se utilizará para realizar esta transferencia de conocimiento, el siguiente paso será elegir la arquitectura CNN que se aplicará para la clasificación del mismo, tal y como se observa en la imagen 36.

Existen diferentes arquitecturas CNN utilizadas para la clasificación de imágenes, como ya se ha visto en la sección 3.4 de este documento. El objetivo de este tercer modelo será experimentar con algunas de ellas y tratar de determinar cuáles son las más efectivas para nuestro problema.

Así pues, este tercer modelo que vamos a desarrollar tendrá la estructura indicada en la imagen 37.

Tal y como se observa en dicha imagen, los modelos que emplean técnicas de *transfer learning* cuentan con una primera capa donde se define la arquitectura CNN previamente entrenada que se utilizará para extraer un conjunto de características de cada imagen. Dicha arquitectura CNN está a su vez seguida por un conjunto de capas (principalmente capas densamente conectadas) que se encargan de clasificar nuestras imágenes basándose en las características que hemos extraído.

[9] https://image-net.org/

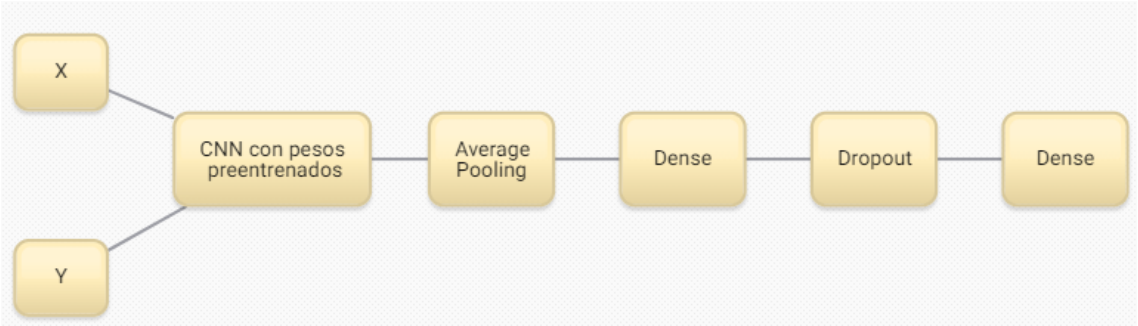


Imagen 37. Estructura del modelo basado en *transfer learning*.

Además de emplear dos capas densamente conectadas para clasificar nuestro *dataset*, también utilizaremos una capa de *Average Pooling* a la salida de la red CNN con el objetivo de disminuir el tamaño de las características de las imágenes extraídas por la misma. Cabe destacar que, de forma similar al modelo CNN anteriormente implementado, también utilizaremos una capa de `Dropout` para eliminar un porcentaje de los datos de salida de la primera capa densamente conectada y evitar problemas de sobreentrenamiento.

El procedimiento seguido para crear y entrenar este tercer modelo es el mismo que el que se ha descrito en el modelo anterior (véase imagen 32 y 33); sin embargo, deberemos tener en cuenta algunos aspectos importantes:

- El tamaño de las imágenes de entrada cambiará en función de la arquitectura CNN empleada en cada ejecución, dado que cada una de las arquitecturas utiliza un tamaño de imagen diferente. Así pues, para realizar una ejecución del modelo utilizando la arquitectura `VGG16` será necesario que las imágenes de entrada tengan un tamaño de 224 x 224 pixeles, mientras que, si utilizamos una arquitectura `NasNetLarge`, necesitaremos imágenes de tamaño 331 x 331 pixeles.
- Tanto en el generador de entrenamiento (`train_generator`) como en el de validación (`valid_generator`) se aplicará una función de preprocesado que se encargará de realizar un conjunto de operaciones para normalizar los datos en función de la arquitectura que se use. Por ejemplo, en el caso de la arquitectura `NasNetLarge`, se empleará una función de `preprocess_input` que escalará los píxeles de entrada entre -1 y 1.
  Por consiguiente, no será necesario utilizar ningún tipo de escalado en los generadores.
- Al momento de definir el modelo deberemos tener en cuenta la estructura que se muestra en la imagen 37, compuesta por una primera capa CNN encargada de extraer las características de las imágenes y un conjunto de capas densamente conectadas que se encargan de clasificar las imágenes atendiendo a las características extraídas. Sin embargo, al utilizar la técnica de *transfer learning*, la primera capa CNN no se tiene que volver a entrenar, dado que dicho procedimiento ya se ha realizado para los datos del *dataset* de ImageNet.
  Además, tampoco necesitaremos cargar las capas densamente conectadas finales presentes en esta estructura CNN, dado que solo nos interesa utilizar las capas encargadas de extraer las características de las imágenes. Por lo tanto, deberemos configurar esta capa como una capa no entrenable, utilizar como

pesos iniciales los obtenidos a partir de ImageNet y evitar cargar las capas finales de esta CNN extractora.
Por otro lado, las capas densamente conectadas finales irán precedidas por la capa de *`Average Pooling`* y tendrán una función de activación de tipo '*`relu`*' y '*`softmax`*', respectivamente, y un total de 120 unidades de salida, al igual que en el modelo anterior (véase Apéndice 1, Figura 8).

Teniendo en cuenta los aspectos anteriores y una vez definido nuestro modelo, procederemos con el entrenamiento del mismo.

Como se ha mencionado con anterioridad, este modelo se va a ejecutar para diferentes estructuras CNN; concretamente ejecutaremos el modelo un total de 5 veces empleando las estructuras *VGG16*, InceptionV3, *Resnet50*, *Inception_Resnet* y *NasNetLarge*.

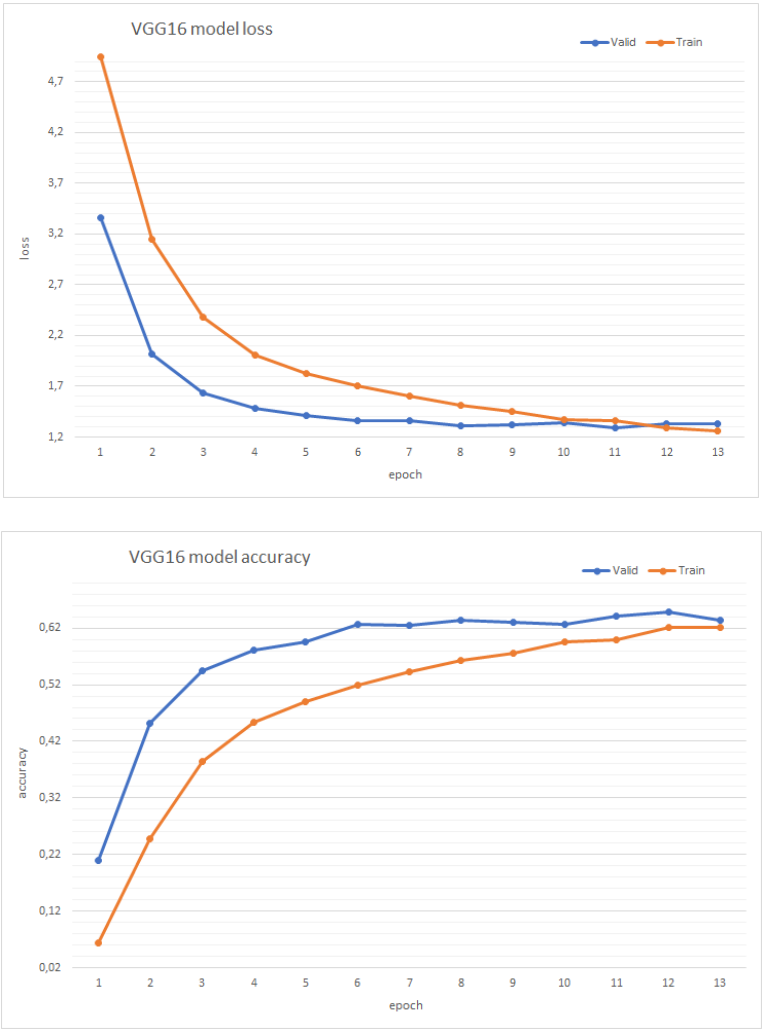


Imagen 38. Resultados VGG16.

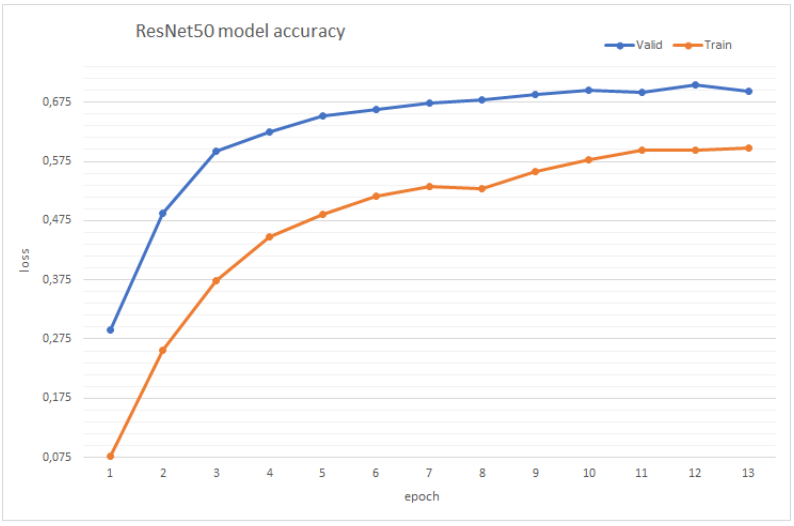
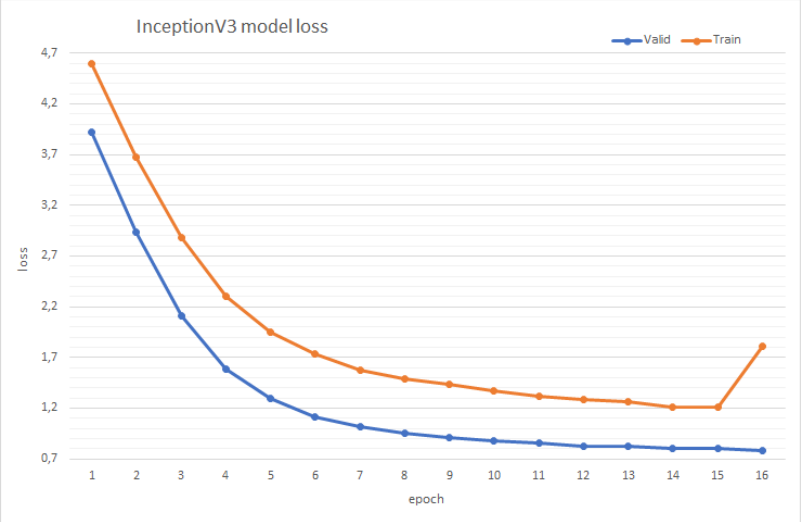


Imagen 39. Resultados Resnet50V2.



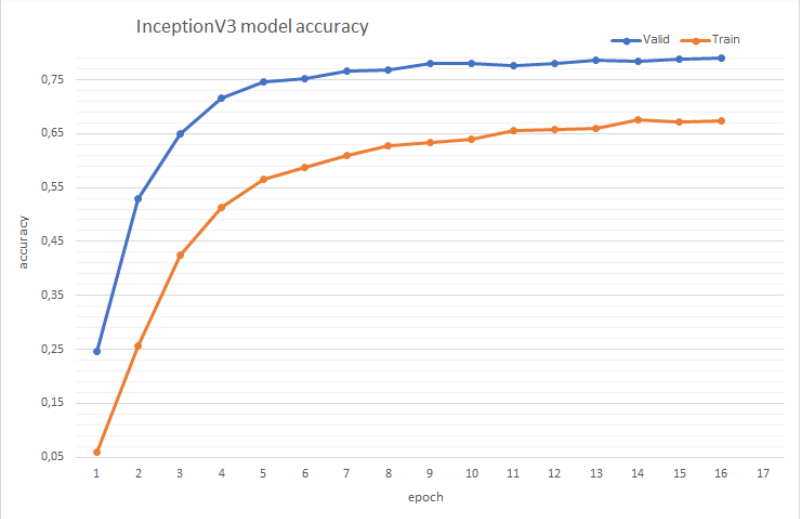


Imagen 40. Resultados InceptionV3.

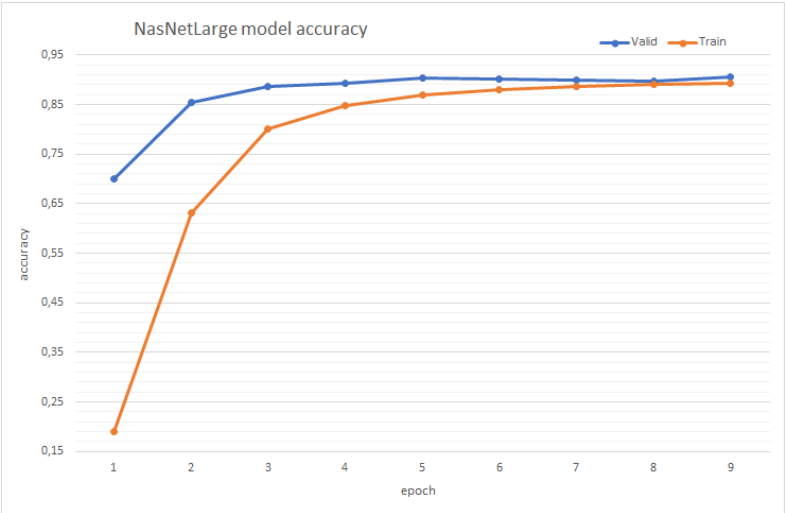


Imagen 41. Resultados NasNet Large.

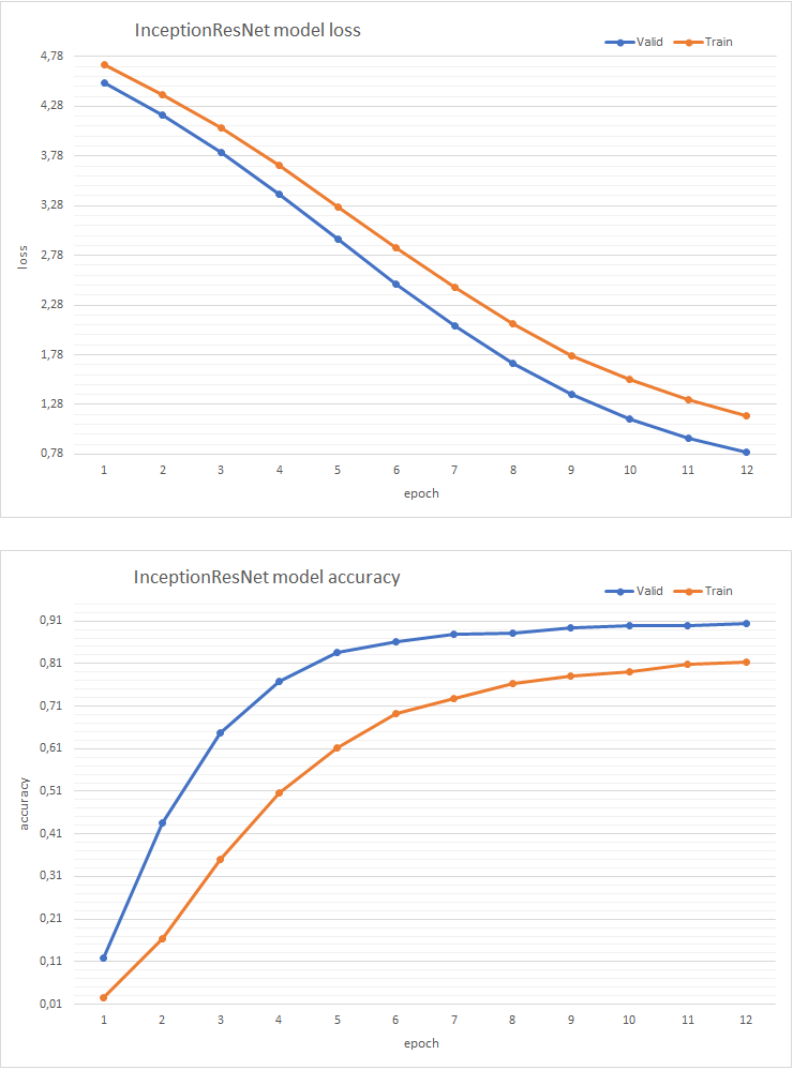


Imagen 42. Resultados Inception_Resnet.

Como se observa en las imágenes 38, 39, 40, 41 y 42, este tercer modelo basado en el uso de *transfer learning* resulta ser el modelo más efectivo entre los modelos vistos hasta el momento, llegando a obtener valores máximos de **90%** de precisión mediante estructuras como InceptionResnet o NasNetLarge y valores mínimos de **60%** de precisión para estructuras como VGG16.

Como se puede apreciar, en este tercer modelo se ha logrado solventar el problema de la calidad de los datos de entrada, dado que no solo utilizamos el color de los píxeles de cada imagen para detectar al perro en cuestión, si no que, a su vez, la red busca por patrones específicos capaces de identificar la forma del perro en la imagen y distinguirlo entre los diferentes objetos y el fondo de la misma.

Así pues, podemos afirmar, que la técnica de *transfer learning* presenta gran efectividad a la hora de extraer las características de las imágenes. Además, presenta un menor coste computacional que las CNN profundas entrenadas desde cero, dado que la arquitectura CNN utilizada por el modelo ya dispone de un conjunto de pesos

entrenados, y por lo tanto el único coste computacional de nuestro modelo recae sobre las capas finales del mismo, es decir, depende de las capas densamente conectadas empleadas para clasificar las imágenes.

5.5 Extractor de características (modelo propuesto)

Finalmente, en vista de los resultados obtenidos por los modelos anteriores, vamos a implementar un último modelo que pretende mejorar la precisión empleando las técnicas que han resultado ser más efectivas.

Indiscutiblemente, el modelo que ha obtenido mejores resultados ha sido el modelo que ha empleado las técnicas de *transfer learning*. Sin embargo, la precisión obtenida por este modelo puede ser más o menos elevada dependiendo de la arquitectura CNN preentrenada que se utilice, llegando a variar hasta un 30% entre el modelo basado en la arquitectura VGG16 y el basado en la arquitectura NasNet.

Por ello, este último modelo pretende utilizar diferentes arquitecturas CNN a la vez, permitiendo obtener la mayor precisión posible mediante una única ejecución. La idea principal es concatenar las características extraídas por diferentes redes preentrenadas para una misma imagen, tal y como se observa en la imagen 43. Cada red obtendrá un conjunto de características diferentes o, equivalentemente, se reconocerá una mayor cantidad de patrones por imagen, lo cual implica que el vector de características de cada imagen será un vector más completo, permitiéndonos mejorar la precisión del modelo anterior.

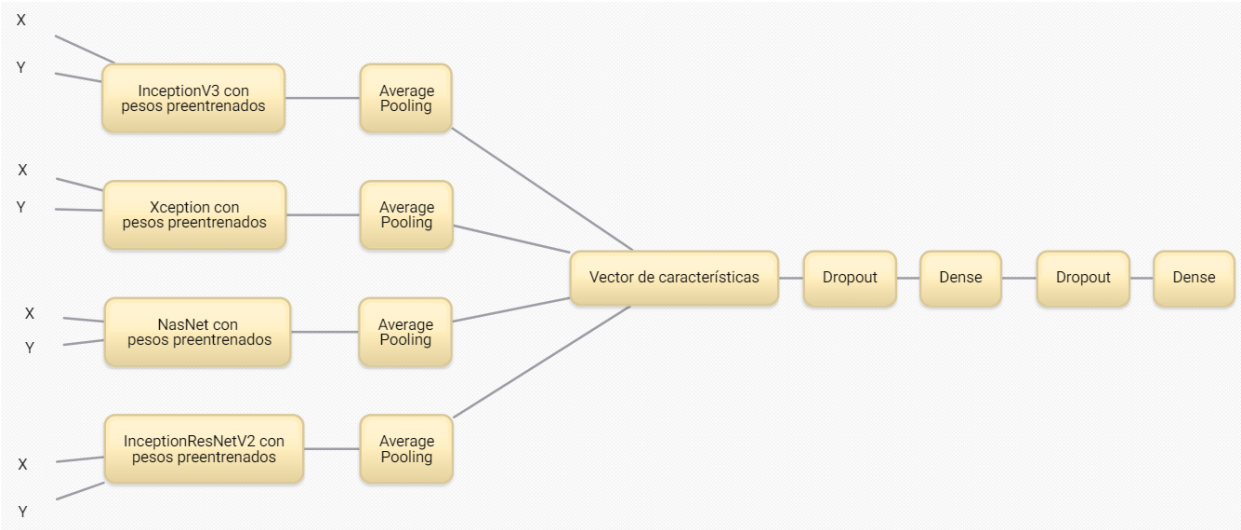


Imagen 43. Estructura del modelo propuesto.

Como se observa en la imagen 43, el modelo realizará un proceso de extracción de características seguido de un proceso de *average pooling* por cada una de las cuatro arquitecturas CNN preentrenadas. Además, después de concatenar los resultados de las diferentes arquitecturas, realizaremos un proceso de *dropout* sobre las mismas, dado que al extraer características de diferentes modelos para una misma imagen existe riesgo de sufrir sobreentrenamiento en el modelo. Finalmente, al igual que en el modelo anterior, utilizaremos dos capas densamente conectadas para clasificar nuestras imágenes y se realizará un proceso de *dropout* entre ambas para disminuir aún más el riesgo de sufrir sobreentrenamiento.

El procedimiento empleado para crear y entrenar este último modelo se detalla en la imagen 44.

En dicha ilustración se observa un esquema similar al empleado en los modelos anteriores; como vemos primero se crean los conjuntos X e Y mediante el método `images_to_array` y posteriormente se utiliza el método `train_test_split` para extraer los subconjuntos de validación y entrenamiento. Sin embargo, existen varias diferencias con respecto a los modelos anteriores:

1. En esta ocasión también tenemos presentes varias arquitecturas CNN en un mismo modelo, de forma que el tamaño de las imágenes de entrada variará en función de la arquitectura empleada. No obstante, esta vez no se va a ejecutar un mismo modelo en repetidas ocasiones para comprobar la efectividad de cada estructura CNN, sino que, se va a tratar de usar un conjunto de redes CNN distintas en una misma ejecución. Por lo tanto, el tamaño de las imágenes de entrada deberá ser el máximo tamaño de los tamaños requeridos por las diferentes arquitecturas, en este caso 331 x 331 píxeles (arquitectura `NasNetLarge`).
2. Tras realizar la etapa de extracción de los subconjuntos de validación (`X_valid` e `Y_valid`) y entrenamiento (`X_train` e `Y_train`), y antes de pasar a convertir las etiquetas a formato *one-hot*, se vuelven a unir dichos conjuntos creando dos nuevos vectores `new_X` y `new_Y`. Esto es debido a que el método `fit` que utilizaremos para entrenar nuestro modelo extraerá los subconjuntos de validación a partir de los últimos elementos del vector de entrada, atendiendo al porcentaje indicado en el parámetro `validation_split` (véase apartado 4.2.3). De esta forma nos aseguraremos de que dicho conjunto de validación contenga muestras de todas las clases en una misma proporción (véase Apéndice 1, Figura 9).
3. Tras realizar el procedimiento anterior, pasaremos a extraer las características de cada una de las imágenes de entrada almacenadas en el vector `new_X`, mediante el método `get_features` (Apéndice 1, Figura 3). Este método se encargará de extraer los patrones de cada imagen de entrada mediante una arquitectura CNN indicada y de aplicar un proceso de *average pooling* sobre la salida de dicha CNN, tal y como se muestra en la imagen 40.
   Por lo tanto, deberemos aplicar el método `get_features` por cada una de las estructuras CNN preentrenadas que se pretenda utilizar, obteniendo un vector de características diferente por cada una de estas estructuras utilizadas, tal y como se observa en la imagen 41.

4. Finalmente, concatenaremos los diferentes conjuntos de características obtenidos para crear el vector de características final deseado `train_features` (véase Apéndice 1, Figura 10).
   A su vez, las etiquetas almacenadas en el nuevo vector `new_Y` serán convertidas a formato *one-hot* y podremos pasar a iniciar la fase de entrenamiento de este modelo.

La etapa de entrenamiento seguirá el mismo esquema empleado en los modelos anteriores (véase imagen 33). Sin embargo, esta etapa recibirá como entrada un vector de características por cada imagen (`train_features`) y su correspondiente etiqueta en formato one-hot almacenada en el vector `new_Y`, de manera que solo se necesitará ajustar los pesos de las capas densamente conectadas finales del modelo (véase Apéndice 1, Figura 11).

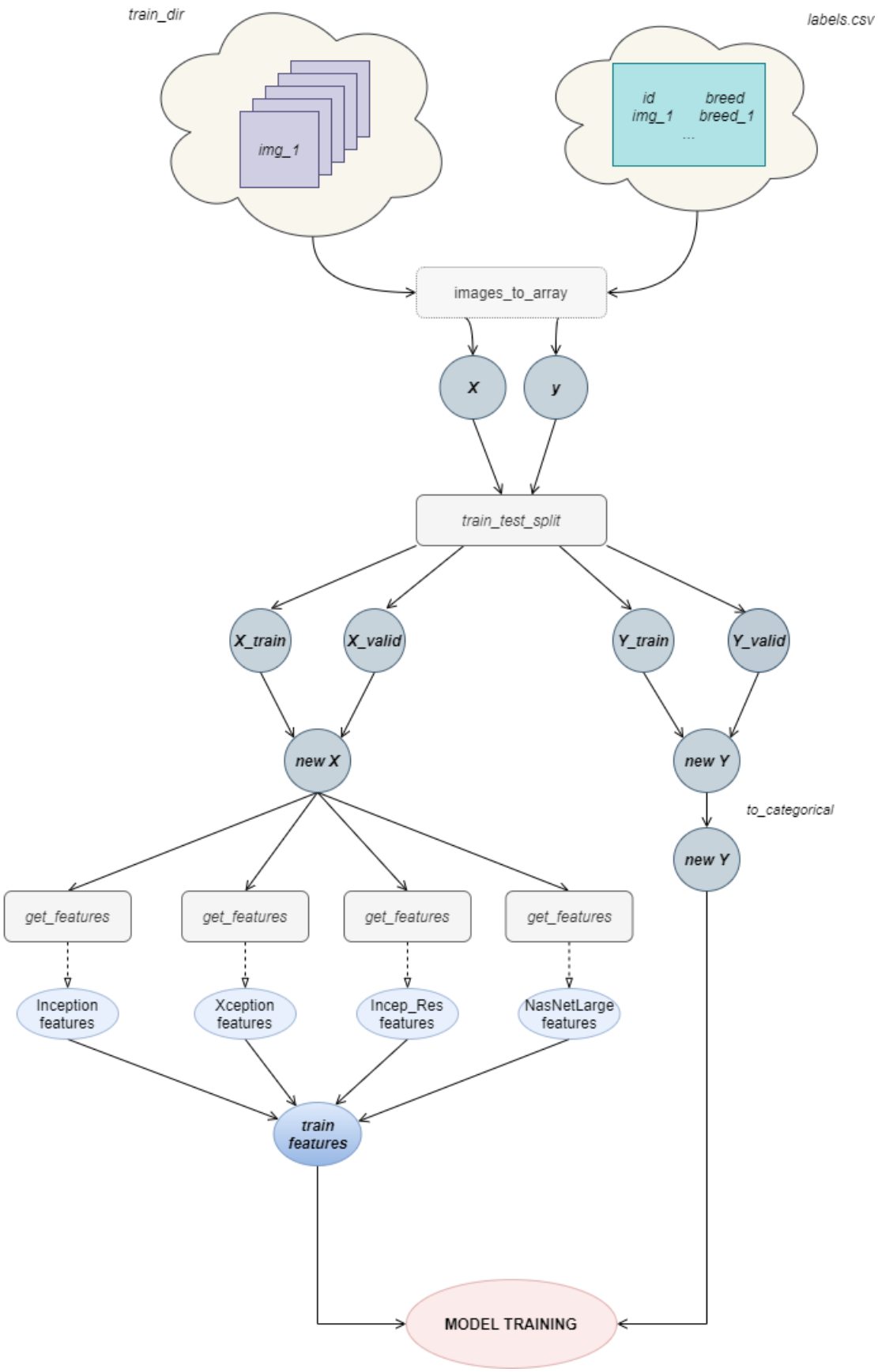


Imagen 44. Esquema empleado para la creación del extractor de características.

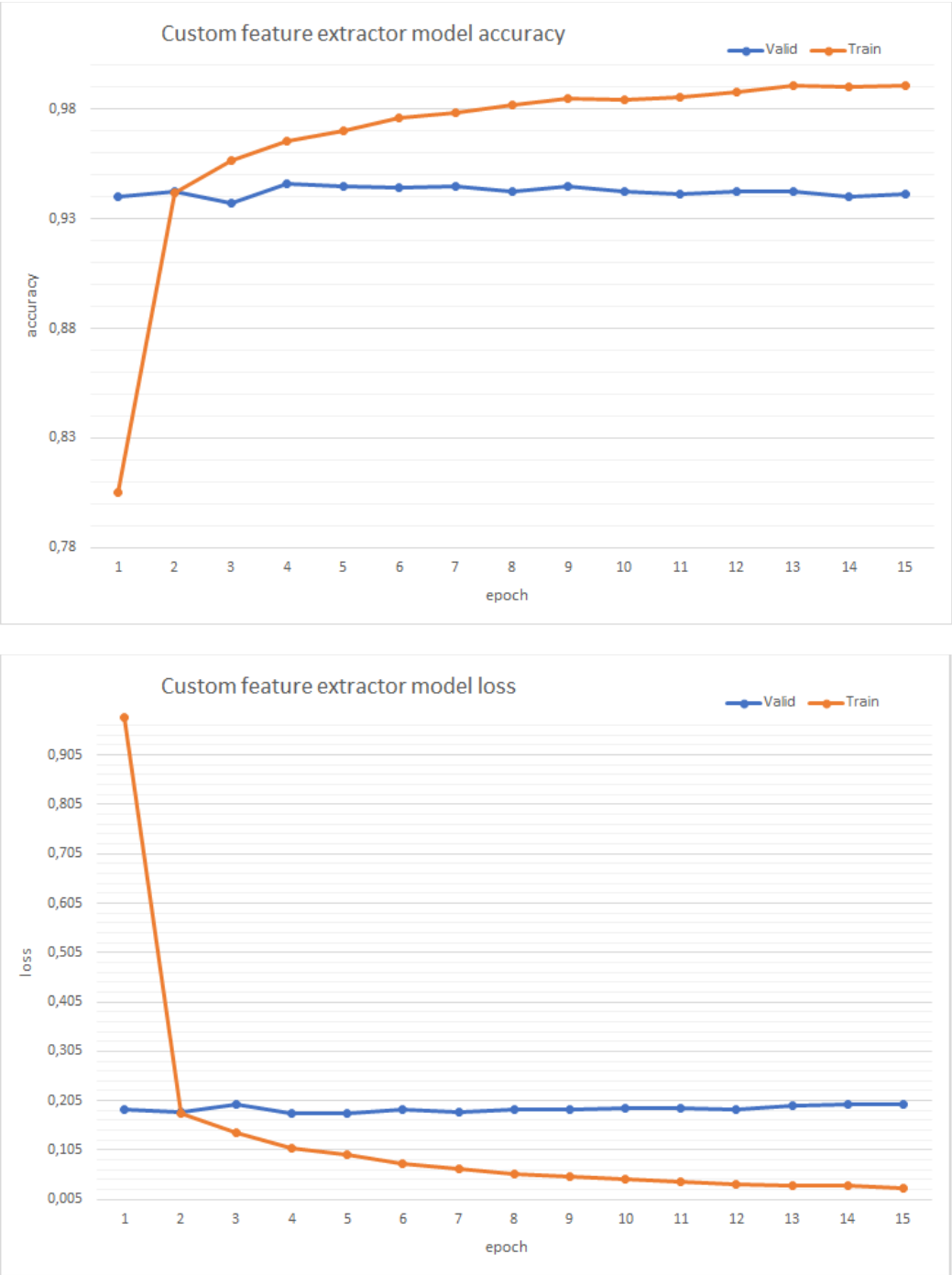


Imagen 45. Resultados obtenidos mediante el modelo propuesto.

Como se observa en la imagen 45, a partir del segundo *epoch* la función de pérdida calculada para el conjunto de validación empieza a converger, mientras que la función de pérdida del conjunto de entrenamiento continúa decreciendo, dando posibles signos de sobreentrenamiento. Esto significa que nuestro modelo se está adaptando a los datos de entrenamiento vistos conforme avanzan los *epoch*, pero sin embargo la precisión

calculada para el conjunto de validación se mantiene más o menos estable (el aumento de precisión que se observa en la gráfica 45 se debe a la relación que guardan los conjuntos de validación y de entrenamiento, cuanto más similares sean más aumentará dicho valor), lo que indica que el modelo ya ha llegado a su capacidad máxima de extracción de las características relevantes para las imágenes no vistas en el proceso de entrenamiento y, por lo tanto, no tiene sentido entrenar dicho modelo con más *epoch*.

Por lo tanto, se puede afirmar que dicho modelo obtiene una precisión máxima de un **94,5%** a partir del segundo *epoch* de entrenamiento. Esto supone una mejora con respecto al modelo anterior, ya que no es necesario entrenar el mismo modelo de *transfer learning* para diferentes arquitecturas con la finalidad de encontrar aquella que ofrece mayor precisión. Además, como se ha visto, este modelo converge en tan solo 2 *epoch* de entrenamiento, ofreciéndonos no solo una mejora con respecto a la precisión, sino también una mejora con respecto al coste de entrenamiento del modelo.

6. Conclusiones

A lo largo de este documento se han descrito las principales metodologías que se emplean en la actualidad para la clasificación de imágenes. A su vez, se ha experimentado con cada una de ellas, y se ha testeado la precisión que nos ofrecen. En la imagen 46 se indica de forma resumida, las precisiones obtenidas por cada modelo en comparación a su coste de entrenamiento o coste computacional.

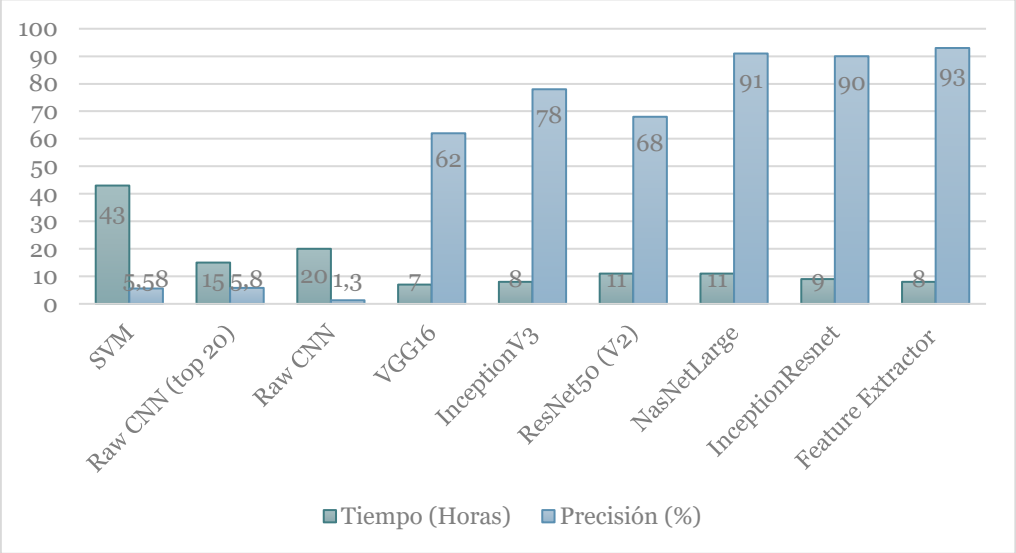


Imagen 46. Comparación coste precisión de los modelos.

Atendiendo a los algoritmos empleados en cada modelo podemos realizar una clasificación en tres categorías diferentes.

1. Modelos basados en la clasificación de los datos de entrada sin extracción de características.

Dentro de este grupo, tenemos a aquellos modelos que tratan de clasificar las muestras de entrada a partir de los datos disponibles sobre las mismas, sin realizar ningún tipo de operación sobre estos datos que se encargue de filtrar los aspectos más importantes de los mismos según su clase. Destacan algoritmos que realizan exclusivamente tareas de clasificación como por ejemplo máquinas de vectores soporte (SVM) o redes neuronales artificiales (ANN).

En la sección 5.2 de este documento se ha implementado un modelo basado en el uso de SVM para clasificar nuestro *dataset* de imágenes de perros, utilizando el color de los píxeles de cada imagen como elemento diferencial de cada una de las razas. Como se observa en la imagen 46, este primer modelo ha tenido resultados pésimos tanto de precisión como de coste computacional.

Estos tipos de algoritmos no están diseñados para clasificar imágenes por sí solos, necesitan de un mecanismo que sea capaz de extraer un vector de características por cada imagen, de manera que se pueda realizar la clasificación de las muestras de forma óptima.

2. Modelos basados en la clasificación de los datos de entrada mediante extracción de características.

Dentro de este grupo se engloban todos aquellos modelos que tratan de extraer las características más relevantes de los datos de entrada, con la finalidad de emplear dichas características para la posterior tarea de clasificación. En el caso de tareas de clasificación de imágenes, destacan estructuras como las redes neuronales convolucionales (CNN).

En la sección 5.3 de este documento se ha experimentado su efectividad a la hora de clasificar nuestro dataset de imágenes de perros. Sin embargo, aunque obtiene un menor coste computacional con respecto al tipo de modelo anterior, sigue sin ser un modelo efectivo.

Las redes neuronales convolucionales son efectivas para realizar tareas de clasificación de imágenes; sin embargo, son muy sensibles al ruido en los datos de entrada y necesitan un conjunto de datos de entrada extenso y un alto poder computacional para que funcionen de manera efectiva.

3. Modelos basados en el uso de *transfer learning*.

Los modelos de este grupo destacan por reutilizar otros modelos ya entrenados para tareas de clasificación similares a la tarea que se pretende realizar. Estos modelos también realizan un proceso de extracción de características sobre las imágenes y un posterior proceso de clasificación de las mismas, pero a diferencia de los modelos anteriores, la tarea de extracción de características se realiza mediante una red neuronal convolucional ya entrenada. A efectos prácticos, podemos decir que solo realizan tareas de clasificación, ya que no implementan ningún proceso de extracción de características, sino que usan como extractores otros modelos ya entrenados.

En la sección 5.4 de este documento se ha experimentado la efectividad de estos modelos en nuestra tarea de clasificación de perros, siendo el modelo más efectivo de los vistos en este documento y presentando además costes computacionales menores, tal y como se observa en la imagen 46.

Además, en la sección 5.5 se ha implementado un modelo basado en el uso de *transfer learning*, capaz de disminuir el coste computacional del modelo visto en la sección 5.4 y de aumentar la precisión de la clasificación, utilizando varias estructuras CNN entrenadas mediante el dataset de ImageNet como extractores de características.

Por lo tanto podemos concluir afirmando que los modelos basados en el uso de *transfer learning* tienen una mayor efectividad para tareas de clasificación con ruido en los datos de entrada o con insuficiencia de datos y/o poder computacional.

7. Apéndice

```python
def images_to_array(data_dir, labels_dataframe, img_size):
    images_names = labels_dataframe['id']
    images_labels = labels_dataframe['breed']
    data_size = len(images_names)
    # inicializar los arrays de salida.
    X = np.zeros([data_size, img_size[0]* img_size[1]* img_size[2]], dtype=np.uint8)
    y = np.zeros([data_size,1], dtype=np.uint8)

    # leer las muestras y sus etiquetas
    for i in tqdm(range(data_size)):
        image_name = images_names[i]
        img_dir = os.path.join(data_dir, image_name+'.jpg')
        img_pixels = load_img(img_dir, target_size=img_size)
        X[i] = np.array(img_pixels).flatten()

        image_breed = images_labels[i]
        y[i] = class_to_num[image_breed] # de etiqueta a numero

    return X, y
```

Figura 1. Método *images_to_array* empleado en el modelo I (SVM)

```python
def images_to_array(data_dir, labels_dataframe, img_size):
    images_names = labels_dataframe['id']
    images_labels = labels_dataframe['breed']
    data_size = len(images_names)
    # inicializar los arrays de salida.
    X = np.zeros([data_size, img_size[0], img_size[1], img_size[2]], dtype=np.uint8)
    y = np.zeros([data_size,1], dtype=np.uint8)

    # leer las muestras y sus etiquetas
    for i in tqdm(range(data_size)):
        image_name = images_names[i]
        img_dir = os.path.join(data_dir, image_name+'.jpg')
        img_pixels = load_img(img_dir, target_size=img_size)
        X[i] = img_pixels

        image_breed = images_labels[i]
        y[i] = class_to_num[image_breed] # convertir de etiqueta a numero

    return X, y
```

Figura 2. Método *images_to_array*

```python
def get_features(model_name, data_preprocessor, input_size, data):

    # Preparamos el pipeline.
    input_layer = Input(input_size)
    preprocessor = Lambda(data_preprocessor)(input_layer)
    base_model = model_name(weights='imagenet', include_top=False,
                            input_shape=input_size)(preprocessor)
    avg = GlobalAveragePooling2D()(base_model)
    feature_extractor = Model(inputs = input_layer, outputs = avg)

    # Extraemos las caracteristicas.
    feature_maps = feature_extractor.predict(data, batch_size=128, verbose=1)
    print(model_name,' -> feature_maps shape: ', feature_maps.shape)
    return feature_maps
```

Figura 3. Método *get_features* empleado en el modelo IV (Feature Extractor)

```python
# función lambda que selecciona las posiciones del array xs cuyo valor es x
# ( utilizada para realizar la agrupación de muestras por cada clase )
get_indexes = lambda x, xs: [n for (f, n) in zip(xs, range(len(xs))) if x == f]

def train_test_split(num_clases,X,Y,split_tam):
    X_train, X_valid, y_train, y_valid = [],[],[],[]

    for i in tqdm(range(num_clases)):
        label_set = get_indexes(i,Y) # conjunto de indices del array y de cada clase i (0 a 119)
        test_size = int(split_tam * len(label_set))
        for j in range(len(label_set)):
            current_label = label_set[j] # indice de cada label dentro del conjunto label_set
            if j > test_size:
                X_train.append(X[current_label])
                y_train.append(i)
            else:
                X_valid.append(X[current_label])
                y_valid.append(i)

    X_train, y_train = np.array(X_train), np.array(y_train)
    X_valid, y_valid = np.array(X_valid), np.array(y_valid)
    return X_train,y_train,X_valid,y_valid
```

Figura 4. Método *train_test_split*

```
# variables globales
train_dir = 'train'
img_shape = (85,85,3)

# Leemos las etiquetas de las muestras de entrenamiento
dataframe = pd.read_csv('labels.csv')

# Creamos la lista de razas ordenadas de manera alfabetica
dog_breeds = sorted(list(set(dataframe['breed'])))
n_classes = len(dog_breeds)

# Mapeamos cada etiqueta/raza de String a Integer
class_to_num = dict(zip(dog_breeds, range(n_classes)))

X,y = images_to_array(data_dir=train_dir, labels_dataframe=dataframe, img_size=img_shape)
print(X.shape, y.shape)
trainX, trainY, validX, validY = train_test_split(num_clases=n_classes, X=X, Y=y, split_tam=0.2)

from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV

params = {'C':[0.1,1,10,100], 'gamma':[0.0001,0.001,0.1,1], 'kernel':['rbf','poly']}
svc = svm.SVC()
model = GridSearchCV(estimator=svc, param_grid=params, scoring='accuracy', cv=4 ,verbose=2 )

model.fit(trainX, trainY)
print('model best score: ',model.best_score_)
print('model best params: ',model.best_params_)

pred = model.predict(validX)
print('accuracy: ', accuracy_score(pred, validY))
```

Figura 5. Definición y entrenamiento del modelo I (SVM)

```
# variables globales
NUM_EPOCH = 9
train_dir = 'train'
img_shape = (299, 299, 3)

# Leemos las etiquetas de las muestras de entrenamiento
dataframe = pd.read_csv('labels.csv')

# Creamos la lista de razas ordenadas de manera alfabetica
dog_breeds = sorted(list(set(dataframe['breed'])))
n_classes = len(dog_breeds)

# Mapeamos cada etiqueta/raza de String a Integer
class_to_num = dict(zip(dog_breeds, range(n_classes)))

X,y = images_to_array(labels_dataframe=dataframe, img_size=img_shape)
trainX, trainY, validX, validY = train_test_split(clases=top_clases,X=X, Y=y, split_tam=0.2)

# one hot encoder
trainY = to_categorical(trainY)
validY = to_categorical(validY)

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

test_datagen=ImageDataGenerator(rescale=1./255)

training_set = train_datagen.flow(trainX, y=trainY, batch_size=32)
testing_set = test_datagen.flow(validX, y=validY, batch_size=32)
```

Figura 6. Definición de generadores para los modelos II y III

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

training_set = train_datagen.flow(trainX, y=trainY, batch_size=32)
testing_set = test_datagen.flow(validX, y=validY, batch_size=32)

from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Convolution2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten

model=Sequential()
model.add(Convolution2D (filters = 64, kernel_size = (4,4), padding = 'Same',
                         activation ='relu', input_shape = img_shape))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Convolution2D (filters = 64, kernel_size = (4,4), padding = 'Same',
                         activation ='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(units = NUM_CLASES, activation = 'relu'))
model.add(Dropout(0.2))
model.add(Dense(units = NUM_CLASES, activation = 'softmax')) # output layer

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

history=model.fit_generator(training_set,
                            validation_data = testing_set,
                            epochs = NUM_EPOCH,
                            verbose = 2)
```

Figura 7. Definición y entrenamiento del modelo II (CNN)

```
inceptionResnet = InceptionResNetV2(input_shape = (224,224,3),
                                    include_top = False,
                                    weights = 'imagenet')


from keras.losses import categorical_crossentropy
from keras.models import Sequential
from keras.layers import GlobalAveragePooling2D, Dense, Dropout

model=Sequential()
model.add(inceptionResnet)
model.add(GlobalAveragePooling2D())
model.add(Dense(256,activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(120,activation='softmax'))

model.layers[0].trainable=False

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

model.summary()

callback=keras.callbacks.EarlyStopping(monitor='loss', patience=3,
                                       min_delta=0.01,mode='auto',
                                       restore_best_weights=False,
                                       baseline=None)

history=model.fit(
                  train_generator,
                  epochs=13,
                  validation_data=valid_generator,
                  callbacks=[callback]
                 )
```

Figura 8. Definición y entrenamiento del modelo III (CNN con *transfer learning*)

```
# variables globales
train_dir = 'train'
img_shape = (331,331,3)

# Leemos las etiquetas de las muestras de entrenamiento
dataframe = pd.read_csv('labels.csv')

# Creamos la lista de razas ordenadas de manera alfabetica
dog_breeds = sorted(list(set(dataframe['breed'])))
n_classes = len(dog_breeds)

# Mapeamos cada etiqueta/raza de String a Integer
class_to_num = dict(zip(dog_breeds, range(n_classes)))

X,y = images_to_array(data_dir=train_dir, labels_dataframe=dataframe, img_size=img_shape)
trainX, trainY, validX, validY = train_test_split(num_clases=n_classes, X=X, Y=y, split_tam=0.2)

newX = np.concatenate([trainX, validX], axis=0)
newY = np.concatenate([trainY, validY], axis=0)
newY = to_categorical(newY)

# Liberamos memória RAM ...
del X
del y
del trainX
del trainY
del validX
del validY
```

Figura 9. Creación de los conjuntos `newX` y `newY` del modelo IV (*Feature Extractor*)

```python
# InceptionV3
from keras.applications.inception_v3 import InceptionV3, preprocess_input
inception_preprocessor = preprocess_input
inception_features = get_features(InceptionV3,
                                  inception_preprocessor,
                                  img_shape, newX)

# Xception
from keras.applications.xception import Xception, preprocess_input
xception_preprocessor = preprocess_input
xception_features = get_features(Xception,
                                 xception_preprocessor,
                                 img_shape, newX)

# NASNetLarge
from keras.applications.nasnet import NASNetLarge, preprocess_input
nasnet_preprocessor = preprocess_input
nasnet_features = get_features(NASNetLarge,
                               nasnet_preprocessor,
                               img_shape, newX)

# InceptionResNetV2
from keras.applications.inception_resnet_v2 import InceptionResNetV2, preprocess_input
inc_resnet_preprocessor = preprocess_input
inc_resnet_features = get_features(InceptionResNetV2,
                                   inc_resnet_preprocessor,
                                   img_shape, newX)

# no vamos a necesitar mas la variable X por lo tanto la eliminamos para liberar RAM
del newX

# concatenamos todas las caracteristicas extraidas a partir de los diferentes modelos
train_features = np.concatenate([inception_features,
                                 xception_features,
                                 nasnet_features,
                                 inc_resnet_features,], axis=-1)
# train_features[0] = [inception_features[0],xception_features[0],...]
```

Figura 10. Creación del vector de características `train_features` del modelo IV(*Feature Extractor*)

```python
from keras.callbacks import EarlyStopping
# EarlyStop_callback -> (Paramos el entrenamiento si el valor de val_loss
# no mejora tras 10 epoch)
EarlyStop_callback = keras.callbacks.EarlyStopping(monitor='val_loss',
                                                   patience=10,
                                                   restore_best_weights=True)

# Finalmente Preparamos el modelo DNN (deep neural network)
dnn = keras.models.Sequential([
    InputLayer(train_features.shape[1:]),
    Dropout(0.7),
    Dense(n_classes, activation='softmax')
])

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

# Entrenamos un DNN simple con las caracteristicas extraidas
h = dnn.fit(train_features, newY,
            batch_size=32,
            epochs=16,
            validation_split=0.2,
            callbacks=EarlyStop_callback)
```

Figura 11. Definición y entrenamiento del modelo IV (*Feature Extractor*)

8. Bibliografía

[1] Francisco Escolano, Miguel Ángel Cazorola, María Isabel Alfonso, Otto Colomina y Miguel Ángel Lozano. Inteligencia Artificial; Modelos, Técnicas y Áreas de Aplicación. Departamento de Ciencia de la Computación e Inteligencia Artificial. Universidad de Alicante (Nov. 2003)

[2] Wei-Meng Lee. Python Machine Learning . Ed Wiley; Nº1 edition (Apr 2019)

[3] Iffat Zafar, Giounona Tzandiou, Richard Burton, Nimesh Patel and Leonardo Araujo. Hands-On Convolutional Neural Networks with TensorFlow. Ed Packt Publishing (Aug 2018)

[4] Enrique J. Carmona Suárez. Tema 8 - Tutorial sobre Máquinas de Vectores Soporte (SVM). Dpto. de Inteligencia Artificial, ETS de Ingeniería Informática, Universidad Nacional de Educación a Distancia (UNED), Madrid (2016)

[5] Sebastian Ruder. An overview of gradient descent optimization algorithms. Insight Centre for Data Analytics, NUI Galway Aylien Ltd., Dublin (Jun 2017)

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun. Deep Residual Learning for Image Recognition. Microsoft Research (Dec 2015)

[7] Padideh Danaee and Reza Ghaeini. A deep learning aproach for cancer detection and relevant gene identification. School of Electrical Engineering and Computing Science. Oregon State University, Corvallis, USA (2017).

[8] Marco Will, Roos T.Pitman, Anabelle W.Cardoso, Christina Locke, Alexandra Swanson, Amy Boyer, Marten Veldthuis and Lucy Fortson. Identifying animal species in camera trap images using deep leraning and citizen science. British Ecological Society (2018)

[9] Jiayi Fan, jangHyeon Lee and Yong Keun Lee. A Transfer Learning Architecture Based on a Support Vector Machine for Histopathology Image Clasification. *Appl. Sci.* 2021, *11*, 6380

[10] Whitney LaRow, Brian Mittl, Vijay Singh. Dog Breed Identification. Standford University (2016)

[11] N. Dalal and B. Triggs,."Histograms of oriented gradients for human detection," 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005, pp. 886-893 vol. 1, doi: 10.1109/CVPR.2005.177.

[12] Desolneux, Agnés; Leclaire, Arthur. Stochastic Image Models from SIFT-like descriptors. Society for Industrial and Applied Mathematics SIAM journal on imaging sciences (2018)

[13] László Kozma. T-61-6020 Special Course In Computer Information Science – K Nearest Neighbors algorithm. Helsinki University Technology (2008)