Salina, Mauro David

Deep Learning aplicado al procesamiento de imágenes para la detección de objetos reciclables

2021

Instituto: Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Ingeniería en Informática

TRABAJO FINAL DE LA PRÁCTICA PROFESIONAL SUPERVISADA

Deep Learning aplicado al procesamiento de imágenes para la detección de objetos reciclables

Estudiante:

Mauro David Salina

Tutores:

Prof. Lía Lavigna

Dr. Ing. Cappelletti Marcelo

Mg. Ing. Osio Jorge

Buenos Aires, 2021

PRÁCTICA PROFESIONAL SUPERVISADA (PPS)
Deep Learning aplicado al procesamiento de imágenes para la detección de objetos reciclables
Informe Final

DATOS DEL ESTUDIANTE

Apellido y Nombres: Salina, Mauro David

DNI: 31.604.940

Nº de Legajo: 27439

Correo electrónico: mauro_salina@hotmail.com

Cantidad de materias aprobadas al comienzo de la PPS: 43

PPS enmarcada en artículo (4 ó 7) de la Resolución (CS) 103/16.

DOCENTE SUPERVISOR

Apellido y Nombres: Dr. Ing. Cappelletti, Marcelo

Correo electrónico: mcappelletti@unaj.edu.ar

DOCENTE TUTOR DEL TALLER DE APOYO PARA LA PRODUCCIÓN DE TEXTOS ACADÉMICOS DE LA UNAJ

Apellido y Nombres: Prof. Lavigna, Lía

Correo electrónico: lialavigna@gmail.com

DATOS DE LA ORGANIZACIÓN DONDE SE REALIZA LA PPS

Nombre o Razón Social: Universidad Nacional Arturo Jauretche

Dirección: Av. Calchaquí 6200, Florencio Varela, (1888) Buenos Aires, Argentina

Teléfono: +54 11 4275 6100

Sector: Programa Tecnologías de la Información y la Comunicación (TIC) en aplicaciones de interés social, Instituto de Ingeniería y Agronomía

TUTOR DE LA ORGANIZACIONAL

Apellido y Nombres: Mg. Ing. OSIO, Jorge

Correo electrónico: josio@unaj.edu.ar

FIRMA DEL COORDINADOR DE LA CARRERA

Índice

1. Introducción 8

1.1. Contexto 8

1.2. Motivación 9

1.3. Objetivos 12

1.3.1. Objetivos generales 12

1.3.2. Objetivos específicos 13

1.3.3. Organización del informe 14

2. Marco Teórico 16

2.1. Inteligencia Artificial 16

2.2. Machine Learning 17

2.2.1. Aprendizaje supervisado 19

2.2.2. Aprendizaje no supervisado 20

2.2.3. Aprendizaje por refuerzo 20

2.3. Deep Learning 21

2.4. Redes neuronales Artificiales 23

2.4.1. Arquitectura 24

2.4.2. Sesgo – Bias 26

2.4.3. Función de activación 27

2.4.4. Entrenamiento de una red neuronal 28

2.4.4.1. Proceso de aprendizaje 28

2.4.4.1.1 Método del gradiente descendiente 31

2.4.4.1.2 Hiperparámetros 31

2.4.4.1.3 Optimizadores 34

2.5. Redes neuronales convolucionales 34

2.5.1. Arquitectura 35

2.5.2. Convolución 36

2.5.3. Función de activación 38

2.5.4. Pooling o agrupamiento 39

2.5.5. Clasificación 40

2.5.5.1. Flatten o aplanamiento 40

2.5.5.2. Red completamente conectada 41

2.5.5.3. Activación Softmax 41
2.5.5.4. *Entropía* cruzada 42
2.6. Transferencia de aprendizaje – Transfer Learning 43
2.6.1. VGG16 45
2.6.2. VGG19 46
2.6.3. ResNet50 46
2.6.4. InceptionV3 47
2.6.5. Xception 48
2.6.6. Mobilenet 48
2.7. Preparación del set de datos 49
2.7.1. Recolección de imágenes 50
2.7.2. Prevención de ataques adversarios 50
2.7.3. Normalización 53
2.7.4. División en entrenamiento, validación y testeo 54
2.7.5. Underfitting 54
2.7.6. Overfitting 55
2.7.6.1. Data Augmentation 56
2.7.6.2. Dropout 57
2.7.6.3. Early stopping 58
2.8. Matriz de confusión 59
3. Herramientas utilizadas 62
3.1. Python 62
3.2. Anaconda 63
3.3. Google colab 63
3.4. Tensorflow 64
3.5. Keras 65
3.6. Otras librerías 65
3.7. Hardware 66
4. Modelos propuestos 68
4.1. Modelo con 2 clases 68
4.1.1. Entrenamiento desde cero 68
4.1.2. Entrenamiento con transfer learning 70
4.2. Modelo con 6 clases 74

4.2.1. Entrenamiento desde cero 74
4.2.2. Entrenamiento con transfer learning 75
5. Análisis de resultados 78
5.1 Análisis de gráficos de pérdida y acierto para los 4 modelos propuestos 79
5.2. Análisis de Matriz de confusión para los 4 modelos propuestos 86
5.3. Aplicación de predicción primera versión 90
6. Conclusiones 94
6.1. Líneas futuras 95
Reflexión sobre la práctica Profesional Supervisada como espacio de formación: 96
7. Bibliografía 97
Anexo A – Tutorial de instalación de entorno de trabajo 99
Anexo B – Ejemplo simple de red neuronal convolucional paso a paso 109

Índice de Figuras

Figura 1 - Gráfico de distribución de residuos según su tipo. 10
Figura 2 - Evolución temporal de los campos de IA. 16
Figura 3 - Comparación paradigma clásico vs. Machine Learning. 18
Figura 4 - Diagrama de Venn Inteligencia Artificial. 18
Figura 5 - Enfoque de ML vs. Enfoque de DL. 23
Figura 6 - a) Neurona biológica b) Neurona artificial. 24
Figura 7 - Arquitectura red neuronal artificial. 25
Figura 8 - estructura de una neurona artificial. 26
Figura 9 - Funciones de activación más comunes. 28
Figura 10 - Representación gráfica del proceso de entrenamiento. 30
Figura 11 - Descenso por gradiente. 31
Figura 12 - Gráficos de entrenamiento con LR muy chico VS LR muy grande. 33
Figura 13 - Arquitectura de red neuronal convolucional. 36
Figura 14 - Proceso de convolución. 37
Figura 15 - Stacking de filtros. 38
Figura 16 - Pooling o subsampling. 40
Figura 17 - Aplanamiento de los mapas de características. 41
Figura 18 - pasaje de valores a probabilidades con Softmax. 42
Figura 19 - Entrenamiento desde cero vs. Transfer Learning. 43
Figura 20 - Arquitectura VGG16. 45
Figura 21 - Arquitectura VGG19. 46
Figura 22 - Arquitectura Resnet50. 47
Figura 23 - Arquitectura Inception V3. 48
Figura 24 - Arquitectura MobileNet V2. 49
Figura 25 - Ejemplo adversario generado por un ataque. 52
Figura 26 - Simio Gibón. 52
Figura 27 - División del set de datos para entrenamiento, validación y testeo. 54
Figura 28 - Comparación entre ajustes de datos. 56
Figura 29 - Ejemplo de Image Augmentation. 57
Figura 30 - Red normal vs. Red con Dropout. 58
Figura 31 - Entrenamiento con Parada Temprana. 59
Figura 32 - Matriz de Confusión. 61
Figura 33 - Herramientas utilizadas. 62
Figura 34 - Representación de un tensor. 65
Figura 35 - Comparación entre núcleos de CPU y núcleos de GPU. 67
Figura 36 - Modelo con 2 clases entrenamiento desde cero. 69
Figura 37 - Modelo con 2 clases utilizando Transfer Learning. 73
Figura 38 - Modelo con 6 clases entrenamiento desde cero. 75
Figura 39 - Modelo con 6 clases utilizando Transfer Learning. 77

Figura 40 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.1.1. 80
Figura 41 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.1.1. 80
Figura 42 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.1.1. 81
Figura 43 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.1.2. 81
Figura 44 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.1.2. 82
Figura 45 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.1.2. 82
Figura 46 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.2.1. 83
Figura 47 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.2.1. 83
Figura 48 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.2.1. 84
Figura 49 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.2.2. 84
Figura 50 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.2.2. 85
Figura 51 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.2.2. 85
Figura 52 - Matriz de confusión modelo 4.1.1. 86
Figura 53 - Matriz de confusión modelo 4.1.2. 87
Figura 54 - Matriz de confusión modelo 4.2.1. 88
Figura 55 - Matriz de confusión modelo 4.2.2. 89

1. Introducción

En este capítulo se presentará el proyecto y se definirán sus objetivos.

1.1. Contexto

La Práctica Profesional Supervisada (PPS) se desarrolló en el marco del proyecto de Investigación de la Universidad Nacional Arturo Jauretche UNAJ INVESTIGA 2017 (Código del Proyecto 80020170200025UJ y Resolución Rectoral N° 148/18 de fecha 29/06/2018), cuyo título es “Tecnologías de la información y las comunicaciones mediante IoT para la solución de problemas en el medio socio productivo” dirigida por el Mg. Ing. Jorge Osio.
La propuesta de trabajo se enfocó en el desarrollo de una aplicación de software encargada de realizar una clasificación de imágenes en tiempo real, mediante la cual se busca detectar la presencia de objetos reciclables contenidos en la imagen. El desarrollo se basó específicamente en el uso de redes neuronales convolucionales, las cuales han demostrado ser las más eficientes en el área del procesamiento de imágenes.

Una de las problemáticas actuales se centra en la importancia del reciclaje y cuidado del medio ambiente. Reciclar es de suma importancia para la sociedad, debido a que, supone la reutilización de elementos u objetos ya utilizados, los que de otro modo serían desechados contribuyendo al aumento de la formación de basura y al daño ambiental permanente.

Para llevar a cabo la implementación del presente trabajo se utilizó una herramienta que existe desde mediados del siglo pasado, pero que ha tenido un avance significativo en la última década, la Inteligencia Artificial (IA).

Una de las áreas en donde se avanzó notablemente fue en la de detección de objetos y clasificación de imágenes. Esto se debe en su mayor parte al desarrollo de nuevas técnicas de Machine Learning (Aprendizaje Automático) como el Deep Learning (Aprendizaje Profundo), además de las innovaciones en el manejo de Big Data (datos a gran escala) y el aumento en la capacidad de cómputo mediante el uso de diferentes tecnologías como cloud computing (computación en la nube) o el uso de GPU (unidad de procesamiento gráfico) para el análisis de información. Este avance puede verse en distintas áreas como: medicina, seguridad, turismo, finanzas, robótica, entre otras. Algunos ejemplos de dichos avances en el área se

aplican en: control de vehículos autónomos, detección de rostros, detección de matrículas, diagnóstico de enfermedades, realidad aumentada, etc.

Machine Learning es un subcampo de la IA en el que se utilizan diferentes algoritmos para recolectar datos, y con estos realizar un aprendizaje para luego hacer una predicción o sugerencia sobre algo. De esta manera se permitirá resolver problemas de forma intuitiva y automatizada, sin que el mecanismo de elección se encuentre previamente programado. En la práctica esto se traduce en una función matemática en la que se parte de una entrada y se obtiene una salida, por lo que el desafío reside en construir un modelado automático de esta función matemática.
Deep Learning es un subcampo de Machine Learning, pero existen técnicas de Machine Learning que no utilizan Deep Learning. Este último es utilizado para realizar procesos de Machine Learning empleando redes neuronales artificiales compuestas por varios niveles jerárquicos. En el nivel inicial la red aprende patrones simples, y esta información se envía al siguiente nivel de la jerarquía. Este segundo nivel toma la información obtenida en el primero y la combina con nuevos patrones aprendidos en este, generando información un poco más compleja, la cual es pasada a un tercer nivel, y así sucesivamente.

Dentro de Deep Learning existen varias ramas como redes neuronales, redes neuronales convolucionales o redes neuronales recurrentes. Todas estas arquitecturas son utilizadas en diferentes campos.

1.2. Motivación

En la actualidad existen normativas e iniciativas estatales y privadas que apoyan la recuperación de residuos, cooperativas de recicladores urbanos, contenedores, puntos verdes y también recicladores informales. Pero, aun así, el problema de la basura sigue siendo un desafío colectivo.

Es por ello que se considera sumamente importante el reciclado de desechos para el cuidado del medioambiente. En nuestro país cada dos segundos se produce una tonelada de basura y una fracción grande de ella termina en rellenos sanitarios que están al borde del colapso. Por lo tanto, la clave es profundizar el cambio cultural para dejar de pensar al residuo como un desecho y entenderlo como un recurso. La Figura 1 muestra como es la distribución de residuos según su tipo.

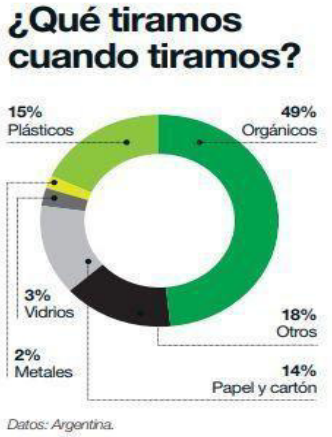


Figura 1 - Gráfico de distribución de residuos según su tipo.
Fuente: Recuperado de https://www.infonueve.com/public/upload/noticias/6150/news_6150_1549028180.jpg (2020)

Según datos del año 2018, el mundo produce alrededor de 1500 millones de toneladas de residuos al año, para establecer una comparación se puede decir que esta cantidad alcanzaría a cubrir la Ciudad Autónoma de Buenos Aires (CABA) hasta una altura equivalente a un edificio de siete pisos, estos datos fueron recolectados de un informe del Estado del Ambiente elaborado por el Ministerio de Ambiente y Desarrollo Sustentable (MAyDS) de la Nación en el año 2018.

Este volumen aporta casi el 5 por ciento de las emisiones de gases de efecto invernadero que producen el cambio climático y se prevé que para el año 2025 la cantidad de residuos se incremente en un 50 por ciento, lo cual acrecentaría aún más la problemática.

Según la Dirección Nacional de Gestión Integral de Residuos (DNGIR), dependiente del MAyDS, la Argentina está ubicada entre los países de rango medio en generación per cápita diaria de residuos sólidos urbanos (RSU), basura proveniente del ámbito comercial, industrial y residencial. El promedio diario de desechos por habitantes es de 1,03 kilos, equivalente a casi 45.000 toneladas diarias para el total de la población (una tonelada cada dos segundos como ya se mencionó) y alrededor de 16,5 millones cada año. O lo que puede compararse con una pirámide de 85 metros de base y la altura del Aconcagua, de 6.960 metros, como afirma el citado Informe del estado del ambiente.

No obstante, las cifras varían de acuerdo a cada jurisdicción, ya que no existe a nivel nacional una política que regule la recolección de residuos. Se trata de una responsabilidad municipal, por lo que cada distrito decide cómo abordar la cuestión, en base a sus capacidades y recursos económicos.

Se estima que los RSU son la mayoría de los desechos. Entre ellos, la basura doméstica encarna la problemática más significativa: aproximadamente la tercera

parte está formada por papel y derivados, mientras que el resto se compone por plásticos, vidrio, metales y pilas.

Desafortunadamente, la Argentina no cuenta con un estudio global sobre la composición de los RSU. Un documento realizado en el marco de la Estrategia Nacional para la Gestión Integral de Residuos Sólidos Urbanos (ENGIRSU) sostiene que, en el período 2005-2010, los orgánicos, el papel y el vidrio redujeron su participación, mientras que la proporción de plástico aumentó, en concordancia con la tendencia mundial.

Por su parte, el Estudio de Calidad de los RSU del Área Metropolitana de Buenos Aires (AMBA), elaborado por la Coordinación Ecológica Área Metropolitana Sociedad del Estado (Ceamse) y la Facultad de Ingeniería de la Universidad de Buenos Aires (FIUBA), concluye que CABA recicla el 46 por ciento de las 6000 toneladas diarias de residuos que genera, pero que el número podría ser superior, ya que el 40 por ciento del material que llega a los rellenos es potencialmente reciclable.

Según el informe, de ese 40 por ciento, un 17 por ciento corresponde a papel y a cartón, un 19 por ciento a plástico, un 3 por ciento a vidrio y un 1 por ciento a metales. El 60 por ciento restante está representado por un 41 por ciento de alimentos, un 5 por ciento de textiles, un 4 por ciento de pañales y apósitos descartables, más un 10 por ciento de materiales catalogados como “otros”, debido a que cada categoría no supera el 1 por ciento.

Los residuos son percibidos como uno de los principales problemas ambientales que tiene el país y, de su mano, la puesta en práctica de acciones tendientes a su reducción y reciclado encarna otro de los grandes retos. Pese a este esfuerzo realizado, Argentina continúa con problemas de logística de recolección y los tratamientos de residuos actuales continúan siendo deficientes. Todas las concesiones deberían ofrecer una recolección diferenciada.

No existe a nivel nacional una ley que impulse la práctica de separar en origen y, mucho menos, dividir los residuos en tres partes:

- basura (no reciclables: residuos sanitarios, pañales y pilas)
- reciclables (papel, cartón, vidrio, plásticos y latas)
- orgánicos (resto de comida, hojas y ramas)

Es fundamental la separación en origen, puesto que discriminar una vez que los residuos están mezclados es poco práctico y costoso.

Como ya se mencionó en párrafos anteriores el reciclado o el reciclaje es un acto de suma importancia para la sociedad ya que el mismo supone la reutilización de elementos y objetos de distinto tipo que de otro modo serían desechados,

contribuyendo a formar más cantidad de basura y, en última instancia, dañando de manera continua al medio ambiente.

Cuando se habla de reciclar o de reciclaje se hace referencia entonces a un acto mediante el cual un objeto que ya ha sido usado es llevado por un proceso de renovación en lugar de ser desechado. Los expertos en la materia consideran que casi todos los elementos que nos rodean pueden ser reciclados o reutilizados en diferentes situaciones, aunque algunos de ellos, por ser extremadamente descartables o por ser tóxicos no pueden ser guardados.

Entre las ventajas del reciclaje hay que destacar que este contribuye a evitar el deterioro del planeta por sobreproducción. La destrucción de gran cantidad de bosques o el deterioro progresivo de la capa de ozono ocurren fundamentalmente por la intención de producir muy por encima de las necesidades de las personas. El reciclaje es una suerte de salida a esa situación y permitiría ahorrar gran cantidad de la energía que se utiliza para esos fines.

En cuanto a los beneficios financieros y económicos del reciclaje, puede decirse que el costo de la energía, que en la actualidad es tan alto, se reduciría fuertemente. Por ejemplo, reciclar una tonelada de papel de periódico ahorra unos 4000 KW de electricidad, aproximadamente la electricidad necesaria para una casa de tres dormitorios a lo largo de un año entero.

En nuestro país, aunque se han tomado medidas para fomentar el reciclado, solo un 24% de la población se esfuerza por separar los residuos para minimizar su generación y la contaminación. Gran parte del problema radica en el esfuerzo que requiere clasificar y separar los residuos inorgánicos, es por eso que la propuesta busca desarrollar un sistema de visión por computadora que permita detectar y clasificar objetos reciclables para minimizar la cantidad de residuos que se generan diariamente en las grandes ciudades.

1.3. Objetivos

1.3.1. Objetivos generales

El objetivo principal de la PPS fue el de desarrollar una aplicación a través de la cual se realizó procesamiento de imágenes utilizando Machine Learning aplicado a la detección de objetos reciclables y no reciclables. Para concretar este objetivo se investigó el uso de Inteligencia Artificial en ciencias de la computación.

Otro de los objetivos fue la formación de recursos humanos en un área en constante crecimiento y con características multidisciplinarias, donde se relacionan los sistemas digitales, el cuidado del medio ambiente y la inteligencia artificial.

1.3.2. Objetivos específicos

La hipótesis de trabajo es que a partir de mayores investigaciones y estudios en la especialidad de inteligencia artificial se podrán desarrollar e implementar nuevas propuestas de sistemas basados en aprendizaje automático para el cuidado del medioambiente, más avanzados en cuanto a diseño y tecnología, que permitirán obtener importantes beneficios respecto a la prevención y detección de la contaminación ambiental.

Para alcanzar el objetivo principal mencionado previamente, con el fin de optimizar el reciclado de residuos, primero se deben tener en cuenta los objetos reciclables más comunes, (papel, cartón, botellas, latas, etc.) y luego los materiales con que están hechos para su clasificación (plástico, vidrio, metal, papel). Con un sistema diseñado a medida, se buscó lograr una aplicación de software encargada de realizar la clasificación de imágenes en tiempo real, mediante la cual se detecta la presencia de objetos reciclables contenidos en la imagen. En este punto se puede deducir cuál es la base para el estudio que se propone, en donde se tiene la necesidad de identificar objetos reciclables en imágenes mediante un algoritmo de aprendizaje eficiente, que permita conseguir el comportamiento deseado y realizar la correcta clasificación para su posterior separación. A continuación, se describen las tareas desarrolladas durante el proyecto y la metodología empleada correspondiente:

(i) **Investigar, analizar y estudiar conceptos de Inteligencia Artificial, Machine Learning, Deep Learning y temas en cuestión.** La primera tarea consiste en la búsqueda de material bibliográfico actualizado, con el propósito de adquirir conocimientos y capacidades específicas, sobre los últimos avances referidos a la temática relacionada con técnicas de reciclado (fundamentos, evolución, características, aplicaciones, estructura, etc.). Además, se realizó un estudio minucioso de las técnicas de inteligencia artificial y aprendizaje profundo, para posteriormente seleccionar la más adecuada para el sistema a implementar.

(ii) **Investigar y analizar distintos métodos de procesamiento de imágenes para la detección de objetos.** Recolectar información sobre las nuevas

tecnologías de sistemas de procesamiento de imágenes y las técnicas actuales usadas en sistemas de detección de objetos.

(iii) **Crear dataset de imágenes que se utilizaron para el entrenamiento, validación y testeo del modelo desarrollado.** Se creó una base de datos con miles de imágenes que contienen objetos reciclables para el entrenamiento de la red. Además, se consideró importante poder detectar objetos no reciclables para aportar información durante la toma de decisión.

(iv) **Modelar y desarrollar una red neuronal capaz de detectar objetos con una tasa de éxito superior al 70%.** El desarrollo se enfocó, entre otras cosas, en el uso de redes neuronales convolucionales, las cuales han demostrado ser las más eficientes en el área del procesamiento de imágenes.

(v) **Testear y analizar el funcionamiento de la red contrastando resultados con el conjunto de datos de prueba.** Para determinar la eficiencia real del sistema, se aplicó el algoritmo entrenado, sobre imágenes que no formaron parte del dataset usado para el entrenamiento.

1.3.3. Organización del informe

En el capítulo 2 se presenta el marco teórico que fundamenta la investigación realizada durante este proyecto. Comenzando con los conceptos y evolución de la Inteligencia Artificial y sus subcampos, continuando con la definición y funcionamiento de las redes neuronales artificiales y las redes neuronales convolucionales y, por último, se describen los pasos a seguir y posibles problemas que se pueden encontrar durante la preparación de un set de datos y el entrenamiento de la red.

En el capítulo 3 se presentan las herramientas utilizadas durante el desarrollo del proyecto. Se realiza una breve descripción del lenguaje de programación, herramientas de software y hardware utilizadas para la creación de los modelos.

El capítulo 4 está destinado a la presentación de los modelos desarrollados que presentaron mejores resultados durante todo el proceso. Se detalla cada una de las arquitecturas propuestas y se indica cómo fue compuesto cada conjunto de datos.

En el capítulo 5 se analizan los resultados obtenidos detallando el análisis realizado a cada uno de los modelos y comparando dichos resultados.

Se completa el informe con las conclusiones que se encuentran en el capítulo 6, en donde, además se sugieren nuevas líneas de investigación futuras.

Para complementar este informe, fueron agregados como anexos: un tutorial de instalación del entorno de trabajo y un ejemplo simple de red convolucional detallada paso a paso.

2. Marco Teórico

Durante este capítulo se presentarán los principales conceptos y se explicará su contenido teórico.

2.1. Inteligencia Artificial

La Inteligencia Artificial (IA) está cada vez más presente en los ámbitos de nuestra vida cotidiana y es incorporada en muchas herramientas de uso común para múltiples propósitos.

Si bien los términos “Inteligencia Artificial”, “Machine Learning” y “Deep Learning” frecuentemente se utilizan de forma equivalente, no se refieren a lo mismo, pero sí se puede establecer una relación entre ellos.

El término más general es la IA y tiene como objetivo que las máquinas tengan la capacidad de imitar el proceso de pensamiento humano, es decir, comportamiento inteligente. A partir de la IA surgieron nuevos subcampos como la robótica, los sistemas expertos, la visión por computadora, el procesamiento de lenguaje natural, procesamiento de voz, machine learning, entre otras.

La IA surge en la década de 1950 como un subcampo dentro la ciencia de la computación, que para ese entonces era una disciplina nueva. En sus inicios la IA buscaba alcanzar la capacidad de procesamiento inteligente mediante sentencias y reglas lógicas, que eran programadas manualmente, hasta el final de la década de 1980 este fue el paradigma predominante en el campo de la IA. A continuación, en la Figura 2, se muestra una línea de tiempo de la evolución de los campos de la IA.

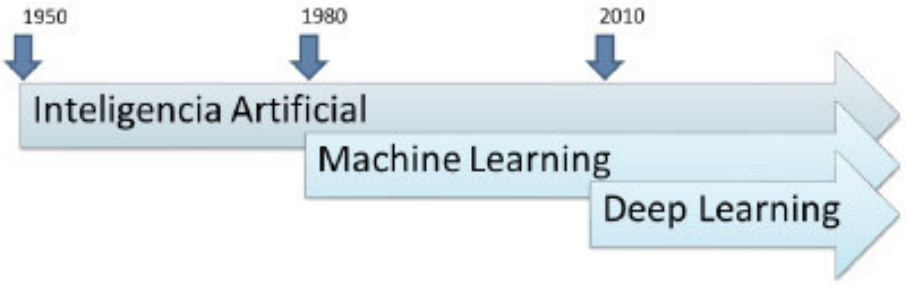


Figura 2 - Evolución temporal de los campos de IA.
Fuente: Elaboración propia

2.2. Machine Learning

El Machine Learning o Aprendizaje Automático (ML) se basa en una serie de algoritmos, que se ejecutan en una máquina y permiten la elaboración de modelos que “aprendan” de manera automática sin estar específicamente programadas para tal fin. El “aprendizaje” de las computadoras se refiere a la capacidad para identificar patrones en grandes conjuntos de datos y a través de ellos tomar decisiones, o hacer una predicción acerca de comportamientos futuros de una situación utilizando un análisis estadístico. ML está estrechamente relacionado con las estadísticas matemáticas, pero presenta algunas diferencias como ser: por una parte, el ML tiende a trabajar con conjuntos de datos tan grandes y complejos que para el análisis estadístico clásico sería muy poco práctico de llevar cabo; por otra parte, el aprendizaje automático está orientado a la ingeniería en donde las ideas se prueban empíricamente con mucha más frecuencia que teóricamente. ML experimentó un crecimiento exponencial durante la última década, esto en gran parte se debe al crecimiento de la capacidad de cómputo y al Big Data, procesamiento de datos a gran escala.

Al aplicar técnicas de ML se presenta un notable cambio en el paradigma de programación como se puede ver en la Figura 3. El paradigma de programación tradicional o clásico tiene como entrada las distintas reglas y los datos, con ellos el programa arroja como resultado una respuesta; en cambio, en el paradigma propuesto por ML las entradas son los datos y las respuestas esperadas y el programa con estas entradas devolverá las reglas, que podrán ser aplicadas a nuevos datos para producir respuestas originales. Un sistema de aprendizaje automático es “entrenado” en lugar de ser “programado”, se presentan muchos ejemplos relevantes de una tarea y el sistema encuentra estructuras estadísticas con estos ejemplos, con lo que eventualmente el sistema crea reglas para automatizar la tarea.

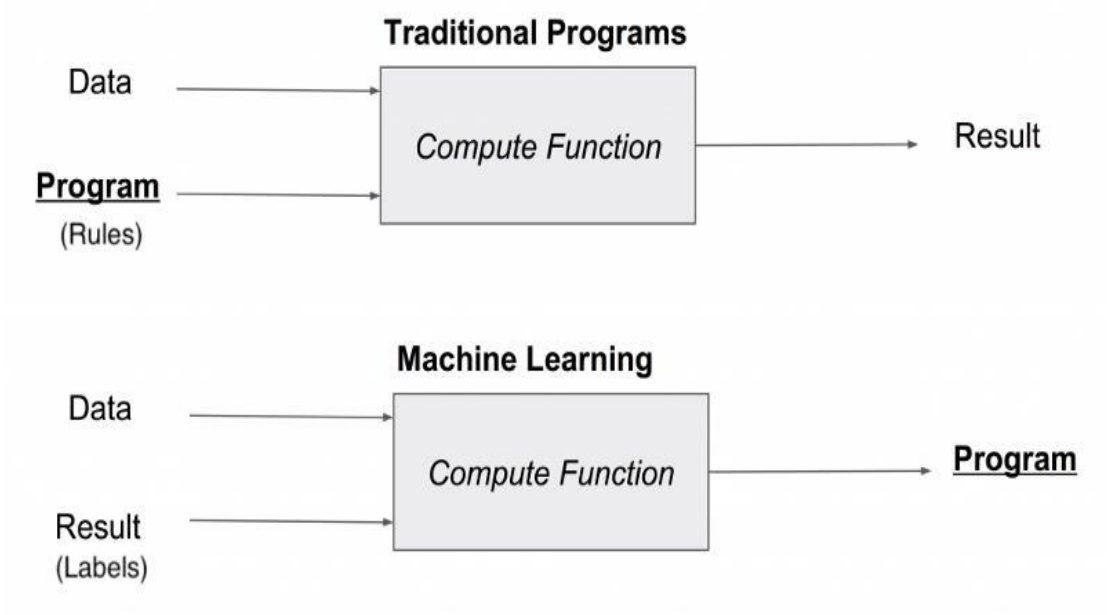


Figura 3 - Comparación paradigma clásico vs. Machine Learning.
Fuente: Recuperado de https://miro.medium.com/max/756/1*LxI5pSQyXI9npG4sfXRPKw.png (2019)

Figura 4 - Diagrama de Venn Inteligencia Artificial.
Fuente: Recuperada de https://www.bravent.net/wp-content/uploads/2019/06/IA.jpg (2019)

Como se puede ver en la imagen (Figura 4) la IA comprende un ámbito muy grande. Un sistema de IA es aquel que tenga la capacidad de aprender, actuar y adaptarse a las situaciones que se le presenten. Dentro del ámbito de la IA se

encuentra el ML como un subconjunto y este está compuesto de algoritmos cuyo rendimiento mejora cuando se procesa una mayor cantidad de datos. El aprendizaje automático es utilizado para modelar, identificar, optimizar, predecir, pronosticar y controlar el comportamiento dinámico de diferentes sistemas reales. Con la técnica de aprendizaje automático se están consiguiendo resultados que antes no era posible obtener con los métodos tradicionales. Sin embargo, no existe una única técnica óptima para todos los problemas, pues cada caso debe analizarse por separado y de acuerdo con los requisitos del problema, en el que se debe aplicar la técnica adecuada. Las técnicas de ML más utilizadas son los algoritmos genéticos, los árboles de decisión, sistemas de regresión, de clusterización, de clasificación, redes neuronales artificiales, entre otras.

Los algoritmos de ML sirven para modelar una gran cantidad de problemas, sin embargo, no funcionan para problemas complejos como lo es el análisis de imágenes para hacer una clasificación o detección de objetos. Poder resolver este tipo de problemas fomentó el desarrollo de nuevos campos dentro de ML, en este informe solo se hace foco en el subconjunto denominado Deep Learning o Aprendizaje Profundo. Este subconjunto junto con las redes neuronales convolucionales son la base de este proyecto, cuyos temas serán retomados en las secciones 2.3, 2.4, y 2.5.

Las técnicas de aprendizaje automático se pueden clasificar en 3 categorías: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo.
Se denomina aprendizaje al proceso por el cual la máquina mejora sus capacidades para resolver un problema. Existen diferentes tipos de aprendizaje en función del tipo de problema al que se enfrenta y de la técnica que se utilice, a continuación, se describirá brevemente cada una de las categorías mencionadas.

2.2.1. Aprendizaje supervisado

En el *aprendizaje supervisado* se empieza con un conjunto de datos etiquetados, para cada vector de variables de entrada se dispone del vector de variables de resultados esperado. Es decir, para cada vector x_i se dispone de la variable de respuesta y_i, en donde i = 1, 2, 3,…, n. El modelo intentará ajustar sus coeficientes para que la diferencia entre la estimación y el valor de respuesta original sea lo menor posible. Este es el tipo de aprendizaje que se utilizará durante este proyecto.

Para aclarar cómo es el funcionamiento del *aprendizaje supervisado* se pone un ejemplo en donde se desea clasificar fotos de perros y gatos. Se debe contar

con un conjunto de imágenes de estos animales y se debe identificar previamente cuál es el animal que se encuentra en cada una de las imágenes y etiquetar dicha imagen. Una vez terminado el proceso anterior este set de datos será utilizado por el algoritmo durante su entrenamiento para aprender las relaciones existentes entre las imágenes y el tipo de animal contenido en ella y, así posteriormente, clasificar nuevas imágenes no etiquetadas que contengan alguno de estos animales.

Con este tipo de aprendizaje se pueden realizar tareas de clasificación y de regresión.

La tarea será de "clasificación" cuando el tipo de valor esperado en la predicción sea discreto, por ejemplo, cuando el modelo debe decidir entre dos categorías como pueden ser objetos "reciclables" y "no reciclables". En caso de haber más de dos clases se trataría de un modelo de clasificación multiclase y este proyecto está basado en este tipo de tarea.

Con respecto a la tarea de "regresión" esta corresponde a modelos en donde el valor de salida es continuo, es decir, no se pueden clasificar los resultados en clases.

2.2.2. Aprendizaje no supervisado

A diferencia del tipo de aprendizaje anterior, en el caso del *aprendizaje no supervisado,* los datos de entrenamiento no incluyen las etiquetas, por lo tanto, será el algoritmo el que buscará realizar la clasificación por sí mismo. Es decir, cada vector de observación x_i no cuenta con la variable de respuesta y_i, el modelo debe autoajustarse buscando patrones y relaciones de dependencia entre las variables observadas. Siguiendo con el ejemplo de la clasificación de imágenes de perros y gatos, el sistema será capaz de distinguir si la imagen contiene un perro o un gato, pero no sabrá a cuál categoría pertenece hasta que se le indique.

Este tipo de aprendizaje presenta una de las principales fronteras de la Inteligencia artificial. Es un tipo de aprendizaje que busca modelar una estructura subyacente o una distribución de datos teniendo como premisa "aprender" más sobre estos.

2.2.3. Aprendizaje por refuerzo

Existen en la actualidad varios modelos de algoritmos que utilizan *aprendizaje por refuerzo*, la metodología de trabajo es la siguiente: los modelos

deberán buscar los resultados sin conocerlo previamente, ante una acción o resultado determinado habrá una recompensa, si este resultado es el esperado se otorgará una recompensa positiva y en caso contrario se otorgará una recompensa negativa.

La metodología de trabajo que utilizan estos algoritmos está compuesta por dos elementos, un entorno y un agente. El entorno le envía un estado al agente, este basándose en su conocimiento realizará una acción para responder a ese estado y esta acción es devuelta al entorno. Posteriormente, el entorno enviará al agente un nuevo estado, pero esta vez acompañado de una recompensa referente a la acción recibida del estado anterior, con esta recompensa el agente actualizará su conocimiento y luego evaluará el nuevo estado. Este bucle continuará hasta que el entorno envíe la señal indicando la finalización del proceso.

2.3. Deep Learning

El aprendizaje profundo o Deep Learning (DL) está comprendido dentro del campo del Machine Learning y se denomina profundo por el modelo de representación de capas de procesamiento no lineal. Estas capas aprenden automáticamente al ser alimentadas por una gran cantidad de datos y son estas capas las encargadas de descifrar patrones ocultos, formando una jerarquía de características desde un nivel de abstracción más bajo a uno más alto. Una de las principales herramientas del DL son las redes neuronales artificiales, entre las que se destacan las redes neuronales artificiales profundas (DNN), las redes neuronales convolucionales (CNN) y las redes neuronales recurrentes (RNN).

El DL contrasta con los algoritmos de aprendizaje poco profundos por el número de transformaciones aplicadas a la señal mientras se propaga desde la capa de entrada hacia la capa de salida. Si bien no existe un número mínimo de capas ocultas para que el modelo sea considerado como aprendizaje profundo, en la mayoría de los casos se espera que el modelo al menos cuente con más de dos capas intermedias.

Estos modelos de DL han mejorado notablemente el estado de las técnicas de reconocimiento de voz, clasificación de objetos visuales, detección de objetos, entre otros.

El DL es una técnica de ML con la que se busca que una máquina pueda aprender a realizar tareas que son naturales para las personas, como ser aprender mediante ejemplos. Los modelos de DL pueden obtener una precisión que, en ocasiones, supera el rendimiento humano. Este avance se dio prácticamente

durante esta última década, aunque las primeras teorías sobre el Deep Learning fueron desarrolladas en la década de 1980, con el comienzo del ML, existen dos razones por lo que su utilidad creció de manera exponencial durante los últimos diez años. En primer lugar, el Deep Learning requiere contar con grandes cantidades de datos y esto era complejo de obtener en 1980. Y, en segundo lugar, para realizar modelos de Deep Learning se requiere una gran potencia de cálculo computacional. El avance de las GPU (unidades de procesamiento gráfico) de alto rendimiento, las cuales cuentan con una arquitectura paralela para la realización de un proceso eficiente del procesamiento de imágenes, esto combinado con los avances de la computación en la nube, permite reducir considerablemente los tiempos necesarios para el entrenamiento de una red de DL.

Por último, para resumir, el DL es una forma especializada de ML, en este la extracción de características relevante se hace de forma manual y con ellas se crea el modelo para lograr, por ejemplo, una clasificación de objetos y, en los modelos de DL, las características relevantes son extraídas directamente por el modelo. El DL lleva a cabo un “aprendizaje completo”, se proporcionan datos sin procesar y la tarea a realizar, como una clasificación y el modelo aprende a hacerlo automáticamente. Otra diferencia clave es que, en los algoritmos de DL, generalmente, la precisión aumenta proporcionalmente con el aumento de la cantidad de datos que se alimenta el modelo, mientras que, en el aprendizaje superficial existen problemas de convergencia. El aprendizaje superficial se refiere a métodos de ML que pueden llevar a un punto muerto en el nivel de rendimiento cuando se agregan más datos y ejemplos al entrenamiento de una red. Por lo tanto, es una ventaja fundamental de los modelos de DL poder continuar mejorando a medida que aumenta el tamaño de los datos.

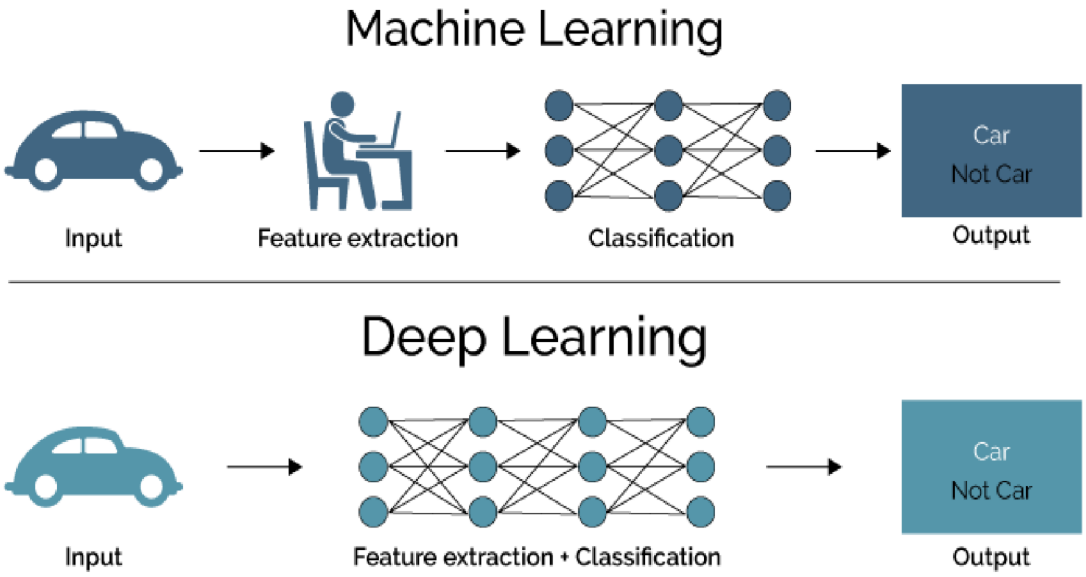


Figura 5 - Enfoque de ML vs. Enfoque de DL.
Fuente: Recuperada de https://www.houseofbots.com/images/news/3620/cover.png (2018)

En la imagen anterior (Figura 5) se representa en la parte superior el proceso de aprendizaje de un modelo de Machine Learning tradicional y, en la parte inferior, el aprendizaje de un modelo de Deep Learning para resolver un mismo problema. Como ya se mencionó en el primer caso se realiza una selección manual de características y en el segundo caso tanto la extracción como la modelización se realizan de manera automática.

La elección de un modelo de u otro dependerá de las necesidades que presente el problema a resolver, dentro de ML se cuenta con diversas técnicas y modelos que se pueden seleccionar en función de la aplicación, el tamaño del conjunto de datos con el que se cuenta y el tipo de problema a resolver, generalmente si se opta por la utilización de un modelo de Deep Learning se requiere una mayor cantidad de datos y un mayor poder computacional.

2.4. Redes neuronales Artificiales

Como se mencionó previamente, las redes neuronales artificiales son un caso especial de algoritmos de Machine Learning. Estos algoritmos datan desde mediados del siglo XX, aunque cobraron popularidad durante la última década debido a que requieren una gran cantidad de recursos computacionales y una enorme cantidad de datos para su correcto funcionamiento. Actualmente, las redes neuronales artificiales son consideradas como una de las mejores técnicas para

llevar a cabo diversas tareas, entre las que se destacan la visión por computador o el procesamiento del lenguaje natural, entre otros.

Las redes neuronales artificiales son un modelo computacional basado en las neuronas del sistema biológico (ver Figura 6) y tienen como finalidad tratar de reproducir el comportamiento de estas. El algoritmo consiste en un número elevado de unidades, llamadas *neuronas*, interconectadas entre sí para transmitir información unas a otras. La neurona es la unidad más básica de una red neuronal y su funcionamiento se basa en recibir datos de entradas que son pasados por una función que produce un resultado; o sea, una red neuronal se puede entender como una gran función que produce una salida dependiente de los datos de entrada de la red. El objetivo final de esta función es realizar predicciones correctas para los datos de entrada (es decir, que la salida obtenida por la red sea la esperada).

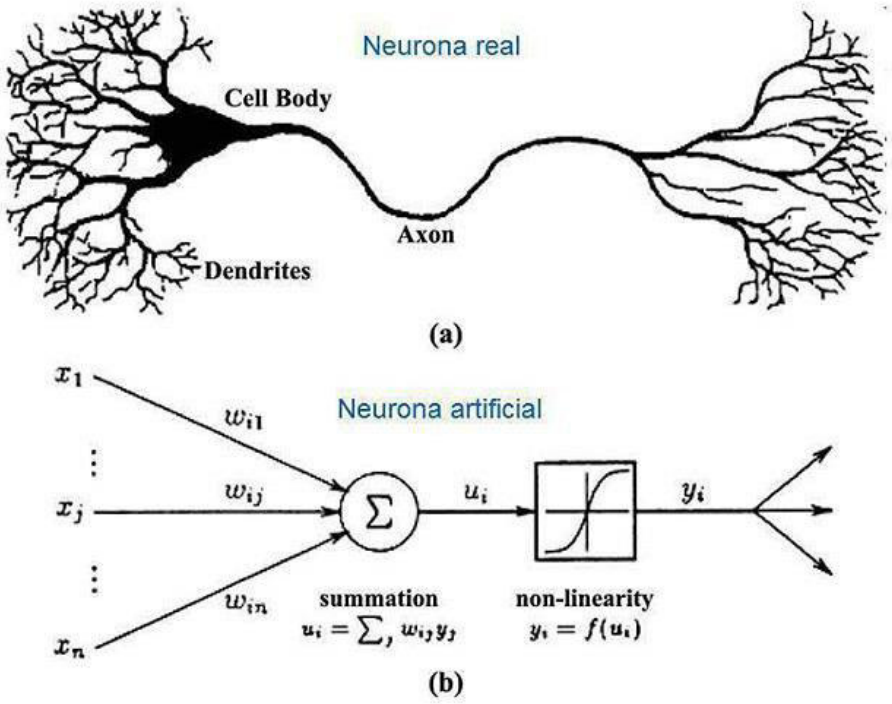


Figura 6 - a) Neurona biológica b) Neurona artificial.
Fuente: Recuperado de https://i.pinimg.com/originals/32/f2/a8/32f2a8c437bf1fb2f84620713d3e44b8.jpg

2.4.1. Arquitectura

En Deep Learning, los algoritmos mencionados permiten representar modelos compuestos por múltiples capas de procesamiento para aprender distintas representaciones de datos, permitiendo múltiples niveles de abstracción, realizando transformaciones no-lineales y partiendo de las entradas se generan salidas

próximas a las esperadas. La red neuronal está organizada por tres capas: la capa de entrada (input layer), que es la que recibe los datos de entrada; la capa oculta (hidden layer), que puede estar compuesta por una o más capas de neuronas (pueden tener distinta cantidad de neuronas en cada una de ellas). Si la red solo contiene una capa de neuronas en la capa hidden se está ante una red neuronal simple y en caso de encontrarse dos o más capas en la capa hidden se dice que la red neuronal es profunda (ver Figura 7); por último, se encuentra la capa de salida (output layer) que es la encargada de devolver la predicción realizada.

La arquitectura más tradicional de las redes neuronales es la que se presenta en la Figura 7 en donde normalmente cada neurona tiene una conexión con todas las neuronas de la siguiente capa, esto se conoce como capas densamente conectadas.

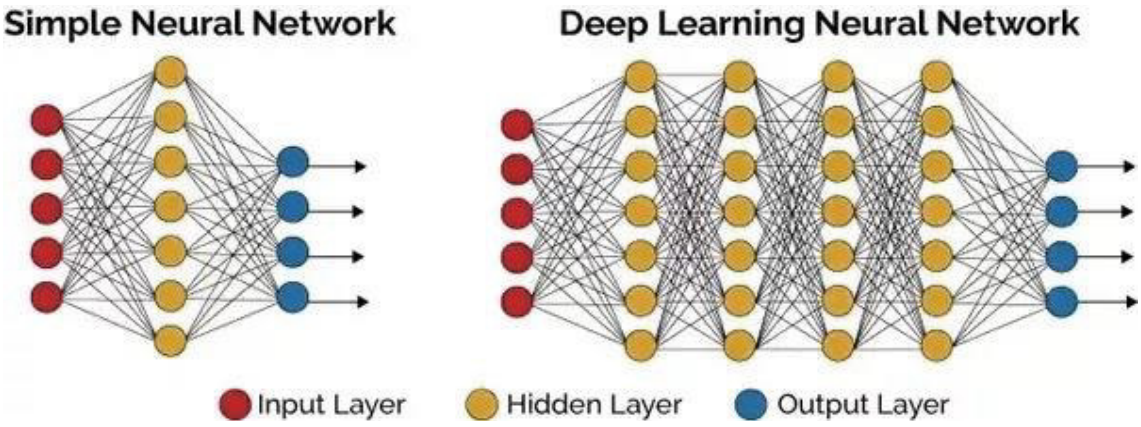


Figura 7 - Arquitectura red neuronal artificial.
Fuente: Recuperado de https://www.iartificial.net/wp-content/uploads/2019/02/tipos_redes-1024x432.jpg (2019)

Para analizar el funcionamiento de una red neuronal se utiliza la arquitectura más simple, o sea, una única neurona también llamada perceptrón. (Ver Figura 8).

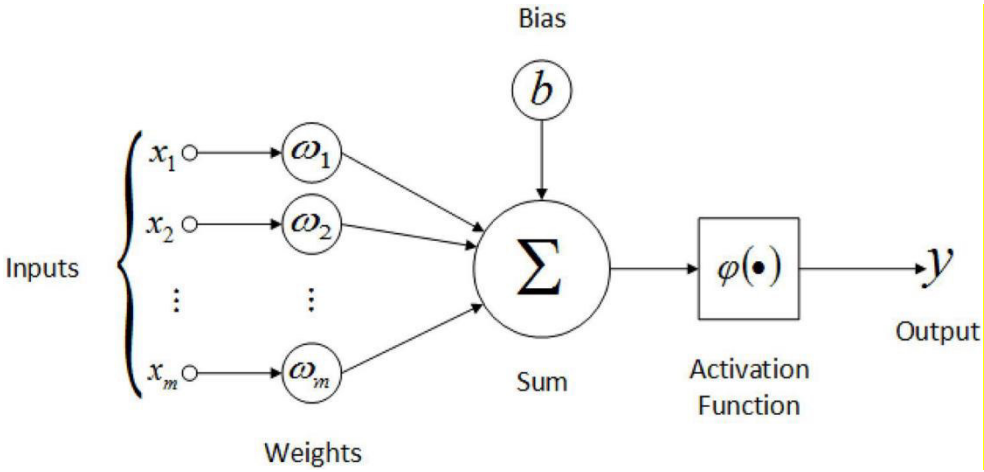


Figura 8 - estructura de una neurona artificial.
*Fuente: Recuperado de https://miro.medium.com/max/3000/0*jtG_gm7tGNoFmbTy (2019)*

Como se aprecia en la Figura 8 cada entrada de la neurona tiene un valor (x_i) que se asocia a un peso (w_i) y este peso se multiplica por el valor de la entrada. El concepto de peso es de suma importancia ya que son los valores que se irán ajustando durante el entrenamiento de la red. Como paso siguiente se realiza la suma de todos los valores de entrada multiplicados por sus respectivos pesos y luego a esta suma se le aplica una función de activación (f). Este cálculo es representado por:

$$y = f(\sum_{i=1}^{N} x_i . w_i)$$

(ec.1)

2.4.2. Sesgo – Bias

Si se observa la Figura 8 se puede ver que antes de aplicar la función de activación a la suma de las entradas por los pesos, se le agrega un bias o sesgo (b), este término puede verse como una falsa neurona que siempre toma el valor 1 y se agrega a la capa de la red con su correspondiente peso. Este bias da a la función la capacidad de desplazarse hacia la derecha o izquierda permitiendo un mejor ajuste de la función para lograr un aprendizaje exitoso, ya que con el ajuste de los pesos solo se modifica la pendiente o inclinación de la función. Al agregar el sesgo la ecuación 1 se corrige a:

$$y = f(b + \sum_{i=1}^{N} x_i . w_i)$$

(ec.2)

2.4.3. Función de activación

La función de activación es una operación que realizan todas las neuronas de la red, esta función recibe como entrada el resultado de la sumatoria anterior y le realiza una transformación, que produce un nuevo valor y que será el valor de salida de la neurona. Este valor será el valor de entrada de la próxima capa y ayuda a determinar cuál neurona se activará en la siguiente capa.

Para el caso de Deep Learning las funciones de activación son no lineales, ya que si se opta por una función de activación lineal la red se comporta como una sola neurona y solo es capaz de resolver problemas muy simples.

Entre las funciones de activación no lineales existen varios tipos (ver Figura 9), las más tradicionales encontramos la función sigmoide y la función tangente hiperbólica, pero en la actualidad se utilizan funciones de activación como Relu (unidad lineal rectificada) o algún derivado de esta como la función Elu y Leaky Relu. En la imagen siguiente (Figura 9) vemos el comportamiento de cada una de estas funciones de activación.

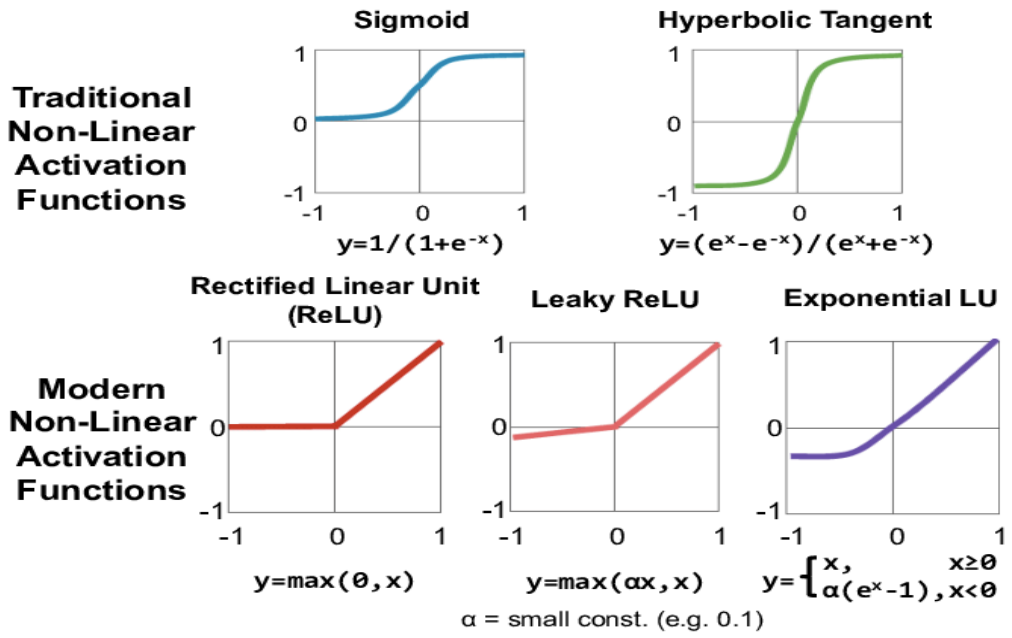


Figura 9 - Funciones de activación más comunes.
Fuente: Recuperada de https://ignaciogavilan.com/wp-content/uploads/2020/05/Various-forms-of-non-linear-activation-functions.png (2020)

2.4.4. Entrenamiento de una red neuronal

Este proceso corresponde al aprendizaje de la red neuronal y consiste en ir ajustando los pesos para que los valores de entrada de la red permitan obtener la salida esperada. Al tratarse de aprendizaje supervisado se parte de la premisa que para cada valor de entrada se conoce el valor de la salida esperada, de esta forma en cada iteración del entrenamiento se ajustarán poco a poco los pesos de cada neurona de modo que, ante todas las entradas, las salidas obtenidas sean las correctas.

El proceso de aprendizaje se puede ver como un proceso iterativo de ida y vuelta por las distintas capas de la red neuronal, siendo sus pasos más destacados la propagación hacia adelante (forward propagation) y la propagación hacia atrás (backpropagation).

2.4.4.1. Proceso de aprendizaje

La fase de *forward propagation* consiste en pasar los datos de entrada a través de toda la red neuronal hasta la capa de salida donde se calcula el valor de la predicción, es decir, cada dato de entrada es pasado por las capas intermedias

de la red donde se realiza la transformación. El resultado obtenido es pasado como entrada de la siguiente capa, cuando los datos cruzan toda la capa oculta se pasan los valores de las transformaciones de la última capa oculta a la capa de salida para realizar la predicción.

El paso siguiente consiste en utilizar una función de pérdida (loss) o costo para estimar el error y así poder comparar y medir si el resultado obtenido fue bueno/malo comparado con el resultado esperado (al tratarse de aprendizaje supervisado los datos están etiquetados por lo tanto se conoce el valor esperado). En el caso ideal se espera que el error sea cero, o sea que coincida el valor predicho con el esperado, para ello a medida que se entrena el modelo se irán ajustando los pesos de las interconexiones de las neuronas de manera automática hasta obtener buenas predicciones. Entre las funciones de error más comunes para este tipo de arquitectura de redes neuronales se tiene el error cuadrático medio (MSE) siendo una de las más utilizadas, error absoluto medio (MAE) y el error cuadrático logarítmico medio (MSLE).

Después de realizar el cálculo de error la información se propaga hacia atrás por la red y durante la fase de backpropagation la información se propaga desde la capa de salida hacia todas las neuronas de las capas ocultas, que contribuyeron directamente en la obtención del valor de salida. Se debe tener en cuenta que cada neurona de la capa oculta sólo recibe una fracción del valor total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona al valor de salida, este proceso se repite a lo largo de todas las capas. En la siguiente imagen (Figura 10) se puede apreciar cómo es este proceso descrito en los párrafos anteriores.

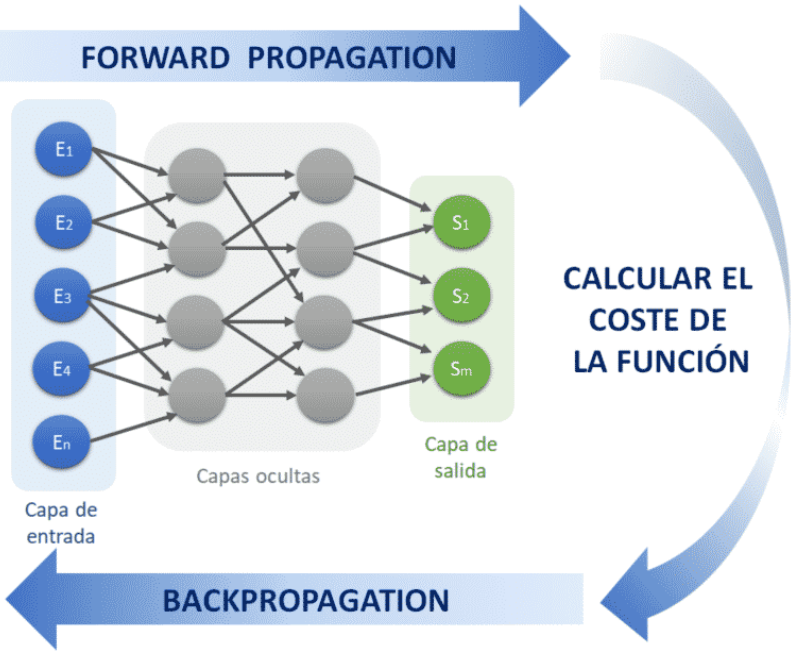


Figura 10 - Representación gráfica del proceso de entrenamiento.
Fuente: Recuperado de https://www.diegocalvo.es/wp-content/uploads/2017/07/esquema-funcionamiento-red-neuronal.png (2017)

Por último, después de terminada la propagación hacia atrás se ajustan los pesos de las conexiones entre las neuronas. Lo que se busca con esto, como ya se mencionó con anterioridad, es que el error sea lo más cercano a cero en la próxima predicción que realice el modelo. Una de las técnicas más utilizadas para lograrlo es la llamada “gradient descent” o método del gradiente descendiente, con esta técnica los pesos se modifican en pequeños pasos basados en la tasa de entrenamiento (learning rate) con la ayuda de la derivada parcial o gradiente de la función de pérdida. Esto permite ver en qué dirección crece la función, por lo tanto, si se utiliza el negativo del gradiente se puede saber la dirección en la que la función decrece hacia un mínimo. Se busca llegar al mínimo global de la función de error, este trabajo se realiza generalmente en lotes de datos (batches) durante las sucesivas iteraciones (epochs) de entrenamiento con todos los datos que le son pasados a la red como entradas. Estos conceptos mencionados se presentan con un poco más de detalle en los siguientes incisos.

2.4.4.1.1 Método del gradiente descendiente

El optimizador “gradient descent” es la base de muchos otros optimizadores y uno de los algoritmos de optimización más comunes en Machine Learning y Deep Learning.

El gradiente es la generalización de la derivada y es el conjunto de todas las derivadas parciales de una función. En el caso de ML se hace foco en el gradiente de la función de pérdida o costo.

El proceso consiste en encadenar las derivadas de la función de pérdida de cada capa oculta a partir de las derivadas de la función de su capa superior, incorporando su función de activación en el cálculo, motivo por el cual las funciones de activación utilizadas deben ser derivables. En cada iteración luego de que todas las neuronas obtienen el valor del gradiente de la función de pérdida se actualizan los valores de los parámetros. Se debe tener en cuenta que el gradiente siempre apunta en el sentido en el que se incrementa el valor de la función, por lo tanto, se utiliza el negativo del gradiente para obtener el sentido en el que la función de pérdida tiende a disminuir(ver Figura 11). De manera que, el sentido viene dado por el negativo del gradiente y la magnitud de este cambio está dada por el valor del gradiente multiplicado por un hiperparámetro llamado Learning Rate. Este hiperparámetro se presentará con más detalle en el próximo apartado.

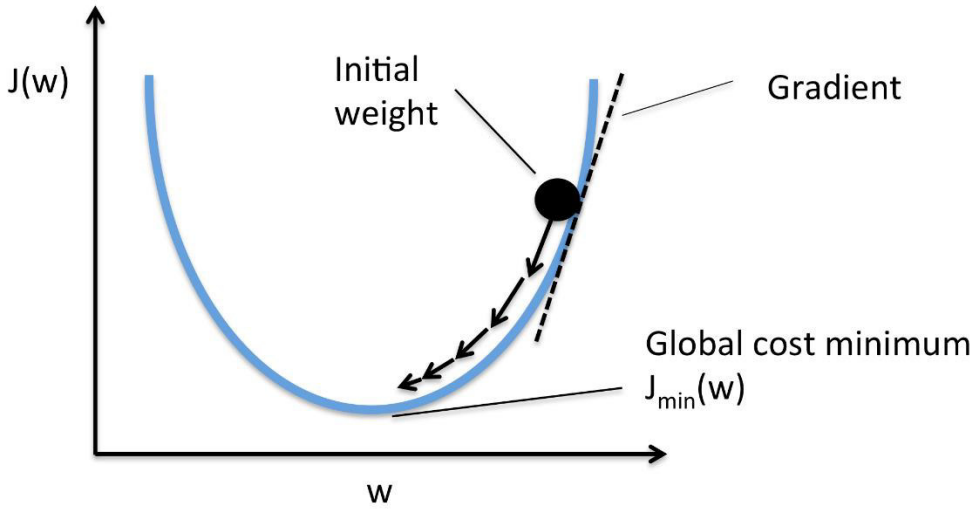


Figura 11 - Descenso por gradiente.
*Fuente: Recuperado de https://miro.medium.com/proxy/0*rBQI7uBhBKE8KT-X.png (2019)*

2.4.4.1.2 Hiperparámetros

Para comenzar esta sección se debe hacer la aclaración de la diferencia entre parámetros e hiperparámetros de la red neuronal.

Los parámetros son los valores que se estiman durante el proceso de entrenamiento partiendo del conjunto de datos, no son definidos manualmente. La estimación de los parámetros es una de las partes más importantes y clave en los modelos de aprendizaje automático, ya que es la parte del modelo que aprende de los datos y son necesarios para realizar predicciones correctas, además definen la capacidad del modelo para resolver un cierto problema. La forma habitual de estimar estos parámetros es la utilización de un optimizador como “gradient descent”, un ejemplo de parámetro son los pesos de las interconexiones.

Mientras que los hiperparámetros del modelo son los valores de las configuraciones utilizados durante el proceso de entrenamiento, estos valores no pueden ser estimados a partir de los datos y son ingresados manualmente por el desarrollador. El valor óptimo de un hiperparámetro no se conoce *a priori* para un problema dado, por lo tanto, se utilizan valores basados en reglas genéricas o valores que han tenido buenos resultados en problemas similares o, también, se suele optar por buscar la mejor opción mediante prueba y error. Una buena opción para encontrar estos hiperparámetros es la utilización de la validación cruzada.

Existen hiperparámetros tanto a nivel de estructura y topología de la red (como el número de capas o el número de neuronas de cada capa, la función de activación elegida, entre otras) e hiperparámetros a nivel de algoritmos de aprendizaje como la tasa de aprendizaje (learning rate), el número de épocas de entrenamiento (epochs), tamaño de lote (batch size), entre otros.

En resumen, al entrenar un modelo de aprendizaje automático se fijan los valores de los hiperparámetros para que con estos se obtengan los parámetros.

- Batch size: cuando el set de datos es muy grande es conveniente pasarlos a la red en lotes más reducidos, se suele dividir el set de datos original en subconjuntos que serán pasados por la red para el entrenamiento. Generalmente se emplean potencias de 2 como tamaño de batches. El tamaño óptimo de este hiperparámetro depende de varios factores entre ellos la memoria del CPU/GPU que se utilice para hacer los cálculos.

- Epochs: indica cada una de las veces en la que todos los datos de entrenamiento pasarán por la red neuronal en el proceso de aprendizaje, cuando todos los paquetes o batches han pasado una vez por la red se cumple una época de entrenamiento. Como el proceso de minimización de la función de error es iterativo se necesitan varias épocas para entrenar la red. Para definir el número adecuado de épocas de entrenamiento se suele ir incrementando el número de épocas hasta que el porcentaje de acierto con los datos de validación empieza a decrecer.

- Learning rate: este es un hiperparámetro que controla la velocidad con la que se avanza en la optimización de la función de error durante el entrenamiento. El vector del gradiente tiene una dirección y una magnitud, el algoritmo de "gradient descent" multiplica la magnitud del gradiente por un escalar, este escalar es la tasa se aprendizaje o "learning rate", para determinar cuál es el siguiente punto en la función determinando el paso que realiza el algoritmo en una iteración del entrenamiento dada. El valor de este hiperparámetro difiere en cada problema a analizar y su correcta elección marca la diferencia entre un buen aprendizaje y uno no tan bueno, se busca que el modelo logre una convergencia óptima lo más rápido posible.

  Si se elige un learning rate (LR) demasiado grande los pasos de optimización serán grandes (Figura 12 – derecha), lo que es bueno para ir rápido en el proceso de aprendizaje, pero es posible que al acercarse al valor mínimo de la función de error el algoritmo oscile entorno al punto de error mínimo y dificulte así la convergencia. Por el contrario, si se elige un valor muy chico para el LR los avances serán constantes, pero muy pequeños (Figura 12 – izquierda) provocando un aprendizaje muy lento o peor aún se podría caer en un mínimo local de la función y nunca converger hacia el mínimo global.

  Existen algunas técnicas que permiten que el valor del LR se reduzca conforme se acerca al mínimo de la función.

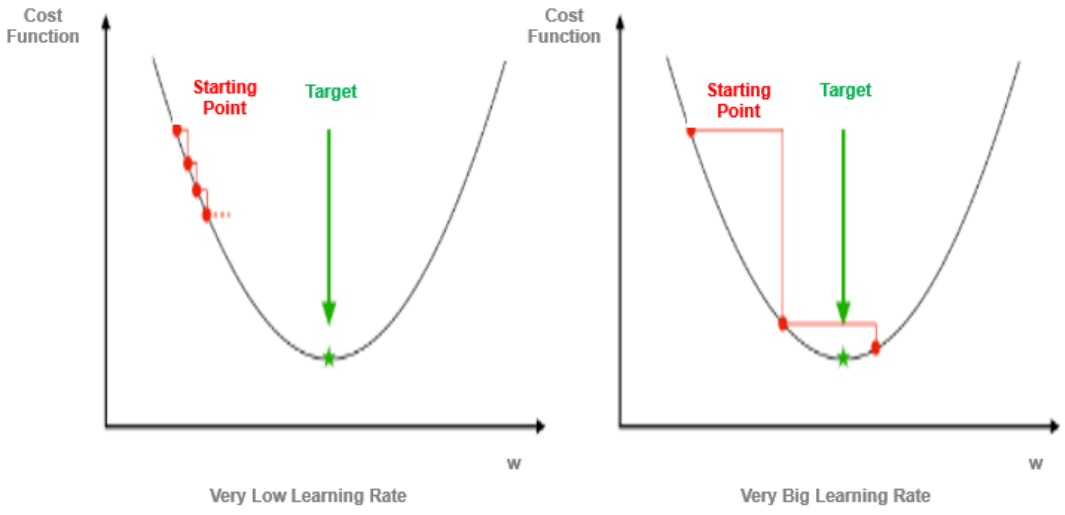


Figura 12 - Gráficos de entrenamiento con LR muy chico VS LR muy grande.
Fuente: Recuperada de https://i2.wp.com/analyticsboomerang.com/wp-content/uploads/2020/06/GD_image3-2.png?w=1170 (2020)

2.4.4.1.3 Optimizadores

Si bien el método de "gradient descent" es el más clásico de los optimizadores, no es el único existente para optimizar la función de error dentro del mundo del Deep Learning, estos optimizadores presentan ciertas ventajas respecto al uso del algoritmo del gradiente descendiente, pero a costa de un mayor costo computacional en su implementación, a continuación, se describen algunos de ellos.

- SGD: Stochastic Gradient Descent (descenso estocástico del gradiente) este es un caso particular del método del gradiente descendiente, que consiste en aplicar un descenso por gradiente a un solo ejemplo por iteración, o sea, aplicar gradient descent a lotes de datos de tamaño 1. Luego al comenzar la siguiente época de entrenamiento se mezclan aleatoriamente los datos de entrenamiento.

- Momentum: es una modificación del descenso del gradiente que da la posibilidad de incluir inercia o momento. Recuerda el incremento aplicado a los pesos en cada iteración y determina la actualización siguiente basándose en una combinación lineal entre el gradiente y esos incrementos anteriores, logrando de esta forma suavizar las oscilaciones en torno al mínimo durante la convergencia del algoritmo gradient descent.

- RMSprop: este optimizador al igual que el anterior pretende suavizar las oscilaciones durante la convergencia del método. Esto lo lleva a cabo ajustando el LR de manera automática. En cada iteración cuando se realiza la actualización de los pesos se escoge un valor diferente para la tasa de aprendizaje. Es un método adaptativo ya que el LR se ajusta durante el entrenamiento.

- Adam: es un optimizador que combina los dos métodos anteriores, Momentum y RSMprop. Este es el optimizador utilizado en los modelos propuestos en este proyecto.

2.5. Redes neuronales convolucionales

Las redes neuronales tradicionales son redes completamente conectadas (fully connected), esto significa que todas las neuronas de una capa oculta están conectadas con todas las neuronas de la capa siguiente y de la capa anterior. Y en

el caso de trabajar con imágenes como datos de entrada se presenta una problemática, para una computadora una imagen es representada como una matriz de píxeles, por lo tanto, cada píxel de la imagen de entrada estaría conectado a cada neurona de la primer capa oculta de la red, y el problema se encuentra en que la cantidad de conexiones necesarias es demasiado grande y esto hace que el uso redes "fully connected" sea prácticamente inviable (incluso para imágenes de tamaño relativamente pequeño en redes muy profundas). Para mitigar este problema se presentan las redes neuronales convolucionales (CNN).

Las CNN son un tipo de redes neuronales artificiales que procesa capas imitando al córtex visual del cerebro humano para identificar distintas características en las entradas que, en definitiva, hacen que pueda identificar objetos y "ver".

Su principal ventaja es que cada parte de la red es entrenada para realizar una tarea específica, aprendiendo diferentes niveles de abstracción, por lo que se reduce significativamente el número de conexiones en las capas ocultas y el entrenamiento es más rápido. Las CNNs son muy potentes para todo lo que tiene que ver con el análisis de imágenes, debido a que la CNN contiene varias capas ocultas especializadas y con una jerarquía. Esto significa que las primeras capas pueden detectar características básicas como líneas, curvas o bordes y se van especializando hasta llegar a capas más profundas capaces de reconocer formas complejas como un rostro, una silueta o un objeto.

2.5.1. Arquitectura

La arquitectura de una red neuronal convolucional (CNN) se puede separar en dos grandes etapas, por un lado, la etapa de extracción de características que a su vez está compuesta por una o más capas convolucionales en donde se realizan pasos fundamentales: convolución, activación y pooling, y por otro lado, la etapa de clasificación, también llamada capa completamente conectada (fully connected layer), que comienza con un aplanamiento y continua con una red neuronal tradicional *fully connected*, en donde se realizará la clasificación final. En la Figura 13 se muestra un ejemplo completo de la arquitectura de una CNN.

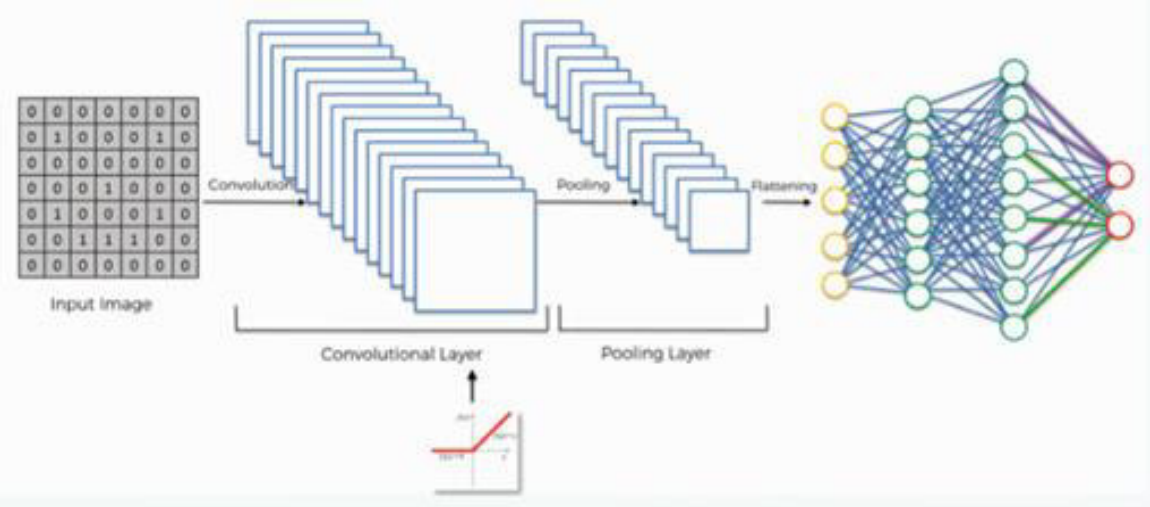


Figura 13 - Arquitectura de red neuronal convolucional.
Fuente: Recuperado de https://frogames.es/wp-content/uploads/2020/02/image-25.png (2020)

2.5.2. Convolución

La convolución es el pilar de la arquitectura de las redes convolucionales. La diferencia fundamental entre una capa densamente conectada (ANN) y una capa especializada en la operación de convolución, llamada capa de convolución, es que la capa densa aprende patrones globales tomados de todo el dato de entrada, mientras que las capas convolucionales aprenden patrones locales en pequeñas ventanas de dos dimensiones.

La capa de entrada de la CNN tiene tantas neuronas como pixeles tiene la imagen de entrada (cada píxel se considera una neurona). En el proceso de convolución se transforman los datos de entrada usando un producto escalar entre una región de las neuronas (sub-matriz de la imagen) de la capa de entrada y la matriz de pesos asignados (kernel). Por lo general, como salida se obtiene una nueva imagen convolucionada con dimensiones espaciales menores o iguales a las de la imagen de entrada, la profundidad de la capa convolucional está dada por la cantidad de filtros que se apliquen a la imagen de entrada.

Una convolución es una operación matemática que describe cómo fusionar dos conjuntos de información. En la siguiente imagen (Figura 14) se muestra esta operación que también es conocida como “detector de características” de una CNN. Como salida se obtiene una capa convolucionada que se denomina mapa de características.

En la Figura 14 se ilustra cómo el filtro se desplaza a través de los datos de entrada para producir la capa convolucionada. Esta ventana va deslizándose (de

izquierda a derecha y de arriba a abajo) a lo largo de toda la capa de neuronas realizando el producto escalar entre las matrices, cada valor que se obtiene como resultado corresponderá a un elemento de la nueva matriz generada por la convolución (o pixel de la imagen filtrada).
El tamaño de la matriz de salida (M_{sal}) viene dado por:

$$M_{sal} = M_{ent} - K + 1 \qquad \text{(ec.3)}$$

Siendo K el tamaño del filtro y M_{ent} el tamaño de la matriz de entrada.

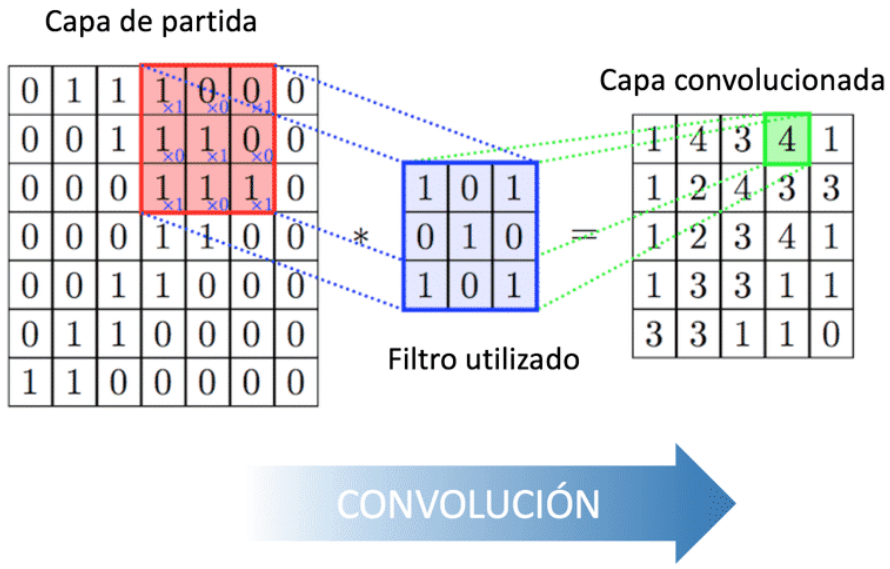


Figura 14 - Proceso de convolución.
Fuente: Recuperada de https://www.diegocalvo.es/wp-content/uploads/2017/07/convoluci%C3%B3n.png (2017)

Es necesario destacar que un filtro es definido por una matriz *K* y un sesgo b y serán los pesos (valores) del filtro y el bias los que se ajustarán durante el entrenamiento de la red, lo que presenta una reducción considerable en la cantidad de parámetros a optimizar por la red. Cada filtro sólo permite detectar una característica concreta en una imagen, por lo tanto, para realizar el reconocimiento de imágenes se utilizan tantos filtros como características se deseen detectar. Por ello, cada capa convolucional en una CNN incluye varios filtros, esto se conoce como “stacking” y se puede ver en la siguiente figura (Figura 15).

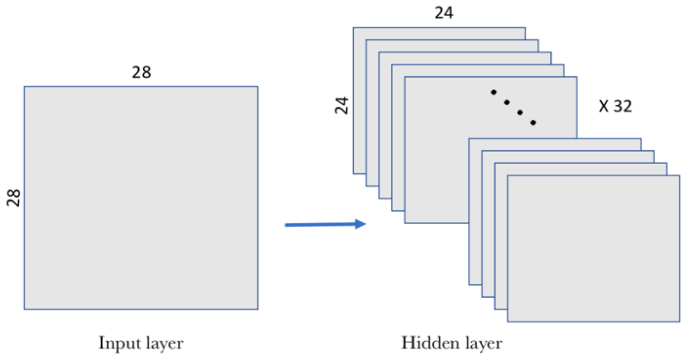


Figura 15 - Stacking de filtros.
*Fuente: Recuperado de https://miro.medium.com/max/1532/1*kOpW3y2veZtoPD7VveLctQ.png (2018)*

Existen hiperparámetros específicos que se deben determinar en una CNN para establecer la disposición espacial y el tamaño del volumen de salida de una capa convolucional.

- Tamaño de kernel: hace referencia a las dimensiones de la matriz de los filtros a utilizar, generalmente este tamaño es mucho más chico que el tamaño de la imagen de entrada.

- Stride o paso: este hiperparámetro refiere a la separación con la que se desplazan los filtros a través de la imagen. Mientras más grande sea el valor del stride más chica será la salida resultante.

- Zero-Padding: Cantidad de "anillos" de ceros agregados alrededor de la imagen de entrada para evitar reducir el tamaño de la salida, es decir, que la matriz de salida tenga las mismas dimensiones que la matriz de entrada.

2.5.3. Función de activación

Como función de activación de cada una de las capas convolucionales se puede utilizar cualquiera de las propuestas que cumpla con la no linealidad, específicamente, para los modelos desarrollados durante el proyecto se utilizó la función Relu. Esta es una de las más utilizadas en los modelos de aprendizaje profundo, ya que permite evitar lo que conoce como desvanecimiento del gradiente. Este es un problema común al realizar backpropagation en redes profundas, ya que este desvanecimiento ocasiona que el entrenamiento de la red no tenga el efecto esperado, o sea, que el entrenamiento no mejore iteración a iteración. La función

devuelve 0 si se recibe una entrada negativa, pero para cualquier valor positivo de *x* devuelve ese mismo valor, por lo que se representa de la siguiente manera:

$$f(x) = \max(0, x) \quad \text{(ec.4)}$$

2.5.4. Pooling o agrupamiento

Una vez que se realiza la convolución y se aplica la transformación no lineal el siguiente paso es realizar un pooling o agrupamiento, también conocido como subsampling (submuestreo). Esta capa tiene como funcionalidad reducir las dimensiones de la salida de la capa convolucional aunque no modifica su profundidad, es decir, se reduce el tamaño de las imágenes filtradas, pero se mantienen las características más importantes que detecto cada filtro. De esta manera se disminuye el número de parámetros y, por lo tanto, el tiempo y capacidad computacional requeridos para entrenar el modelo; además, ayuda a controlar el sobreajuste (overfitting). Esta capa, también, le brinda a la arquitectura una de las características claves que la representan, la capacidad de contar con invariabilidad espacial, es decir que el modelo será capaz de realizar buenas predicciones independientemente de que las imágenes de entrada tengan pequeñas transformaciones como traslaciones y/o rotaciones (por ejemplo, el modelo será capaz de detectar un objeto sin importar si este se encuentra en el centro de la imagen o en alguno de sus bordes).

Existen distintas formas de realizar el pooling, se puede realizar un “average pooling” o un “max pooling” (ver Figura 16). En ambos casos se aplica un filtro de tamaño *m x m (por lo general de 2x2)* con un stride generalmente igual a 2 (el stride puede ser distinto), lo que da como resultado una matriz de salida de la mitad del tamaño de la matriz original, el recorrido del filtro por la matriz al igual que en la convolución se realiza desde izquierda a derecha y de arriba a abajo.

- Max pooling: se aplica un filtro de tamaño *m x m* a través de cada mapa de característica y se selecciona el valor más grande de dicha porción de la matriz, este valor será el valor que se colocará en un pixel de la matriz de salida (ver Figura 16 – a).
- Average pooling: al igual que en el anterior caso se aplica un filtro de tamaño *m x m* a través de cada mapa de característica, pero esta vez el valor que será colocado en un pixel de la matriz de salida corresponde al valor de la media calculada entre todos los valores de la porción de matriz, en donde se encuentra el filtro en ese momento (ver Figura 16 – b).

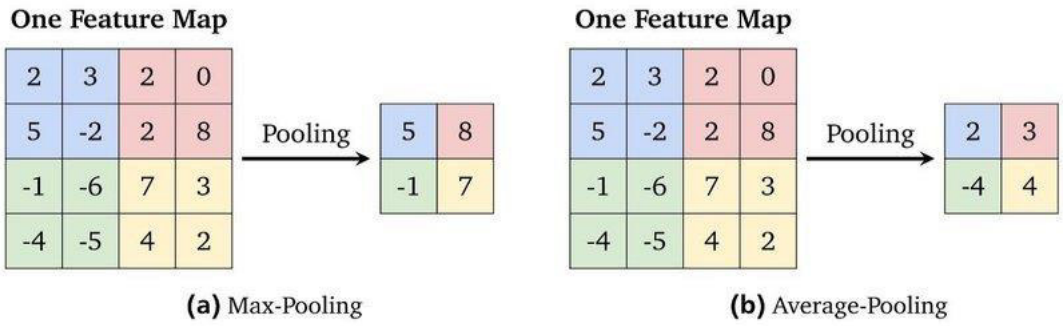


Figura 16 - Pooling o subsampling.
Fuente: Recuperado de https://www.researchgate.net/figure/Example-for-the-max-pooling-and-the-average-pooling-with-a-filter-size-of-22-and-a_fig15_337336341 (2019)

2.5.5. Clasificación

La etapa de clasificación es la etapa final de una CNN. Esta fase consiste, en primer lugar, en realizar un aplanamiento o "flatten" de los mapas de características y luego pasarlos por una red fully connected, que se encargará de realizar la clasificación para determinar la clase a la que pertenece la imagen de entrada.

2.5.5.1. Flatten o aplanamiento

El aplanamiento consiste en tomar todos los mapas de características obtenidos en la última capa convolucional y pasarlos por una función que transformará estos datos tridimensionales en un vector de una dimensión. Este vector será utilizado como entrada de una red *fully connected*.

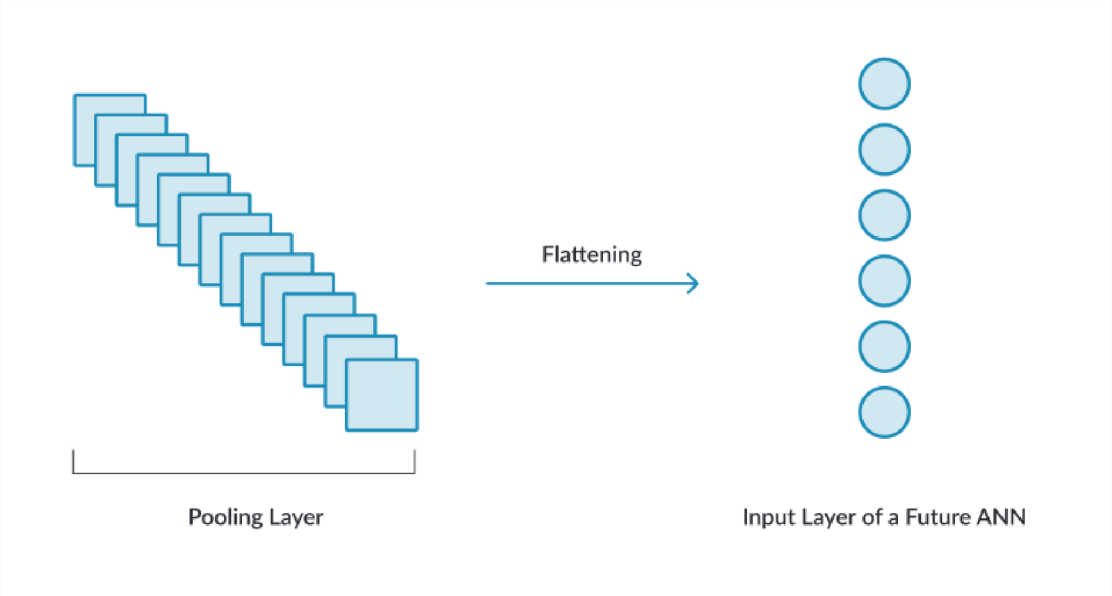


Figura 17 - Aplanamiento de los mapas de características.
Fuente: Recuperado de https://frogames.es/wp-content/uploads/2020/02/image-16.png (2020)

2.5.5.2. Red completamente conectada

Como lo indica su nombre cada neurona de esta capa está conectada a todas las neuronas de la siguiente capa, esto es una arquitectura de redes neuronales artificiales tradicional igual a la presentada en el apartado 2.4.1. Esta red es la encargada de realizar la clasificación final para obtener la predicción del modelo.

2.5.5.3. Activación Softmax

La función de activación utilizada para una clasificación binaria es la activación sigmoide y la empleada en una clasificación multiclase es la activación "softmax" (esta última función también sirve para clasificaciones binarias), durante este proyecto se optó por utilizar para todos los modelos la activación softmax.

La función Softmax es una función que toma como entrada un vector con *k* números reales y lo normaliza en una distribución de probabilidad de que pertenezca a una de las posibles clases. Antes de aplicar softmax cada elemento del vector tiene un determinado valor que no necesariamente suman uno, posteriormente, al aplicar la función de activación cada elemento del vector estará en el intervalo (0,1); por lo tanto, la suma de todos los componentes será igual a 1,

logrando de esta manera que puedan ser interpretados como probabilidades, elemento con valor más grande será el de mayor probabilidad y su posición en el vector determinará la clase predicha. La expresión matemática para calcular dicha función es la siguiente. En la Figura 18 se muestra como queda la transformación, de los datos de salida, en probabilidades.

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{k} e^{x_j}} \qquad i = 0,1,2,\dots,k \qquad \text{(ec.5)}$$

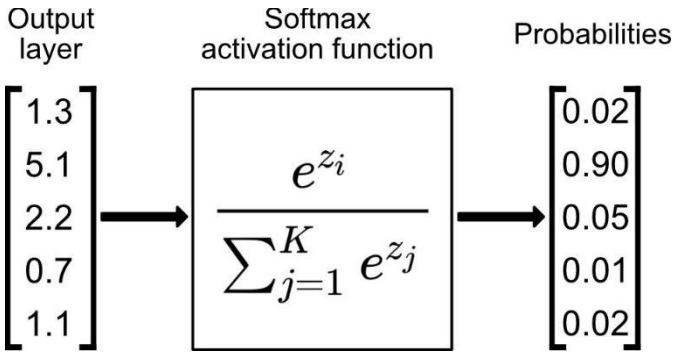


Figura 18 - pasaje de valores a probabilidades con Softmax.
*Fuente: Recuperado de https://miro.medium.com/max/1906/1*ReYpdIZ3ZSAPb2W8cJpkBg.jpeg (2020)*

2.5.5.4. *Entropía* cruzada

La función de pérdida utilizada durante el desarrollo de los modelos propuestos en este trabajo es la función de entropía cruzada (cross entropy loss). La entropía cruzada entre dos distribuciones de probabilidad *p* y *q* mide el número promedio de bits necesarios para identificar un evento de un conjunto de posibilidades. Si un esquema de codificación utilizado para el conjunto es optimizado para una distribución de probabilidad dada *q*, en lugar de la distribución auténtica *p.* la entropía cruzada para dos distribuciones *p* y *q* sobre el mismo espacio de probabilidad, siendo *p* y *q* variables discretas, se define como sigue:

$$H_{(p,q)} = -\sum_x p(x) \log q(x) \qquad \text{(ec.6)}$$

2.6. Transferencia de aprendizaje – Transfer Learning

El aprendizaje por transferencia o *Transfer Learning* se centra en almacenar el conocimiento adquirido mientras se resuelve un problema para luego aplicarlo a un problema diferente. Muchos modelos existentes de clasificación de imágenes pueden obtener un alto rendimiento con un set de datos bien distribuido y características diferenciadas. Pero, lograr una muy buena recolección de datos de alta calidad es una tarea sumamente difícil, debido al tiempo que consume y el alto costo computacional que requiere entrenar modelos desde cero con set de datos que contengan una cantidad elevada de imágenes. Para lograr un buen modelo se debería contar con miles de imágenes que representen a cada una de las clases, en la Figura 19 se muestra una comparación entre el entrenamiento desde cero y el aprendizaje por transferencia.

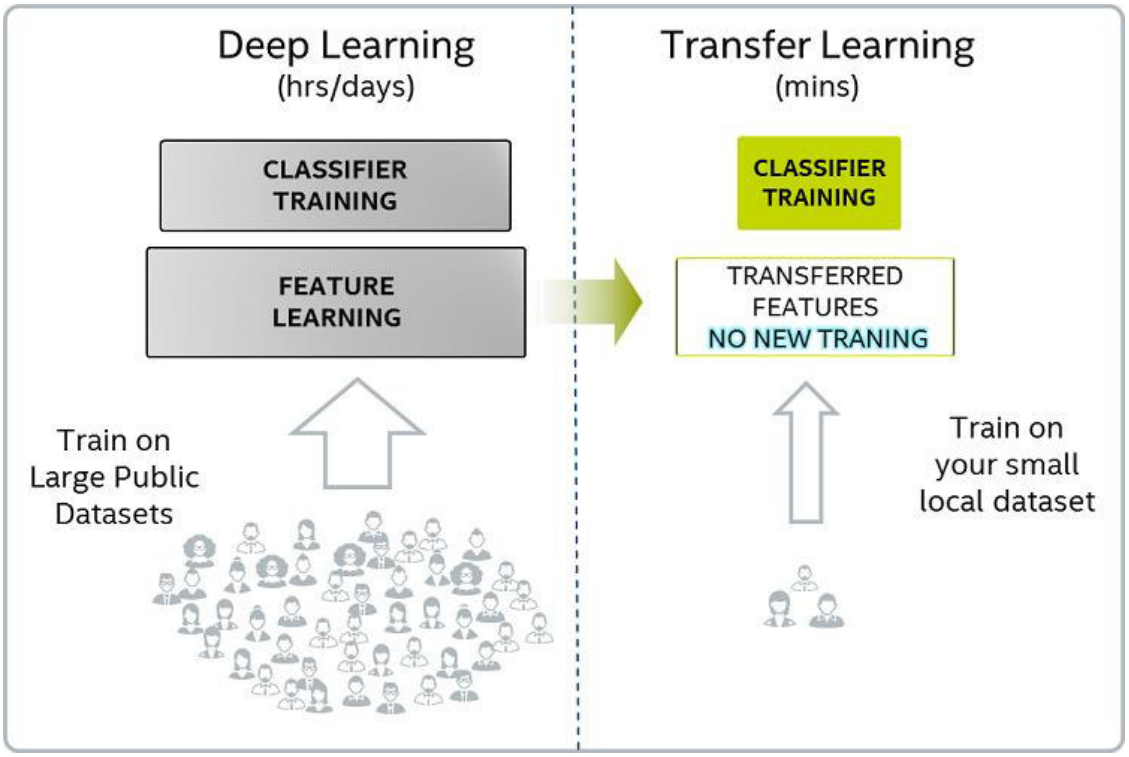


Figura 19 - Entrenamiento desde cero vs. Transfer Learning.
Fuente: Recuperado de https://cdn-media-1.freecodecamp.org/images/nhxsEn9S-VwNdFKCClwfeKhKmTd1buwzF3pR (2019)

Entre las ventajas que presenta la utilización de la transferencia de aprendizaje se pueden destacar los buenos resultados obtenidos con set de datos pequeños, ya que se reduce notablemente el tiempo empleado para el

entrenamiento debido a que se disminuye el número de iteraciones (épocas) y, dependiendo del modelo elegido para hacer la transferencia de aprendizaje, también se reduce la capacidad de cómputo requerida.

Se puede realizar Transfer Learning de tres formas distintas: la primera y la utilizada durante este proyecto, consiste en utilizar el modelo pre entrenado en la etapa de extracción de características y entrenar desde cero la etapa de clasificación. La idea de esta técnica es aprovechar todo el ajuste de los pesos o valores de los filtros que se utilizan en el modelo pre entrenado y solo ajustar el peso de las neuronas de la capa completamente conectada. Además, se reajusta la capa de salida para que se puedan clasificar las nuevas clases del modelo. Esta es una técnica utilizada frecuentemente cuando existe cierta relación entre el set de datos original y el que se desea entrenar.

La segunda técnica consiste en reutilizar la arquitectura de un modelo pre existente, pero realizar un entrenamiento de todas sus capas sin olvidar que se debe modificar la capa de salida para ajustar el modelo a las nuevas clases que se clasificarán. Esta es empleada cuando no existe una relación entre los sets de datos original y el actual; por otro lado, hay investigaciones que aseguran que, aunque no haya relación entre los sets de datos, se obtienen excelentes resultados utilizando la técnica de Transfer Learning mencionada en el párrafo anterior.

La tercera y última de las técnicas consiste en reutilizar parcialmente el pre entrenamiento de la red, por ejemplo, conservando los pesos de las primeras capas convolucionales en donde solo se extraen características muy generales como líneas o bordes y re entrenando las capas convolucionales más profundas, donde se extraen características puntuales y la capa de clasificación. Esta última técnica es muy útil cuando se cuenta con un conjunto pequeño de datos o cuando alguna de las nuevas clases no guarda similitud con los datos de entrenamiento del modelo original.

A continuación, se realiza una breve descripción de las arquitecturas utilizadas para desarrollar modelos basados en transfer learning, que se emplearon durante este proyecto. Todas ellas fueron diseñadas originalmente para el set de datos “Imagenet”.

ImageNet es un proyecto que proporciona una gran base de datos de imágenes con sus correspondientes etiquetas indicando el contenido de las imágenes. Este enorme dataset cuenta con imágenes de 1000 clases diferentes y debido a su gran tamaño es fundamental en el ámbito de investigación en visión artificial. De hecho, cada año, el equipo de ImageNet organiza una competición en la que diferentes equipos de investigación prueban sus algoritmos de reconocimiento visual. En dicha competición se usan más de un millón de imágenes de las 1000 clases que tiene el dataset (es un subconjunto del dataset el que se

usa, ya que el dataset contiene aún más imágenes). Gracias a esta competición podemos saber qué redes son las mejores clasificando imágenes. Y es que, muchas de las arquitecturas conocidas de Redes Neuronales Convolucionales se hicieron famosas a raíz de la competición. Tan importante es el dataset ImageNet que la mayoría de frameworks en Deep Learning almacenan los pesos obtenidos de haber entrenado cada modelo con el dataset (al menos lo hacen para las principales arquitecturas). De este modo, se puede en cualquier momento cargar esos pesos y ya se tiene el modelo entrenado sin necesidad de realizar el entrenamiento desde cero.

2.6.1. VGG16

Esta red fue desarrollada por Visual Geometry Group (VGG) de la universidad de Oxford. Existen varias redes bajo el nombre VGG que difieren en el número de capas que poseen. Es una de las arquitecturas más utilizadas para la extracción de características en imágenes.

La arquitectura VGG16 utiliza filtros convolucionales siempre del mismo tamaño 3x3 y stride=1, padding = same y filtros de agrupación max pooling sobre ventanas de 2x2 y stride=2. Consta de un total de 16 capas, 13 de ellas convolucionales, 2 capas completamente conectadas y la capa final utiliza una activación “softmax” para la clasificación. Además, su arquitectura es muy uniforme y emplea ventanas pequeñas, pero muchos filtros. En la Figura 20 se muestra su arquitectura.

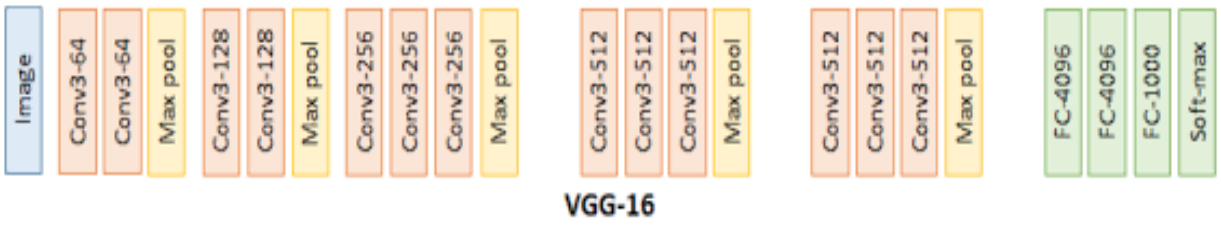


Figura 20 - Arquitectura VGG16.
Fuente: Elaboración propia, basada en https://programmersought.com/article/64016026959/

2.6.2. VGG19

Arquitectura variante del modelo VGG16 desarrollado por el mismo equipo de desarrolladores, en este caso el modelo VGG19 también utiliza filtros convolucionales siempre del mismo tamaño 3x3 y stride=1, padding = same y filtros de agrupación max pooling sobre ventanas de 2x2 y stride=2. Consta de un total de 19 capas, 16 de ellas convolucionales, 2 capas completamente conectadas y la capa final utiliza una activación “softmax” para la clasificación.

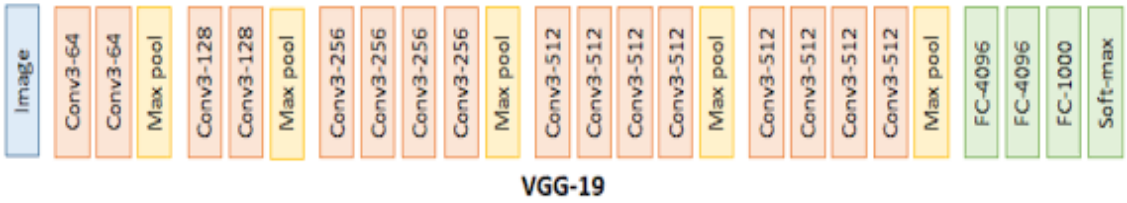


Figura 21 - Arquitectura VGG19.
Fuente: Elaboración propia, basada en https://programmersought.com/article/64016026959/

2.6.3. ResNet50

La arquitectura ResNet es una de las más novedosas, ResNet proviene de “residual networks” o redes residuales y este tipo de redes es capaz de aprender funciones más complejas y en consecuencia obtener un mejor rendimiento. Sin embargo, en muchas ocasiones agregar más capas tuvo eventualmente un efecto negativo en el resultado final, a este fenómeno se lo conoce como el problema de degradación, aludiendo al hecho de que, si bien las mejores técnicas de inicialización de parámetros y la normalización de lotes permiten que las redes más profundas converjan, en ocasiones convergen a tasas de error más altas que las menos profundas.

Lo que propone ResNet es incorporar saltos en las conexiones entre capas. Es decir, permitir una conexión directa entre la entrada de una capa *n* y una capa

n+x, permite saltos en las conexiones de la red. Se ha demostrado que entrenar estas redes es más sencillo que entrenar redes simples.

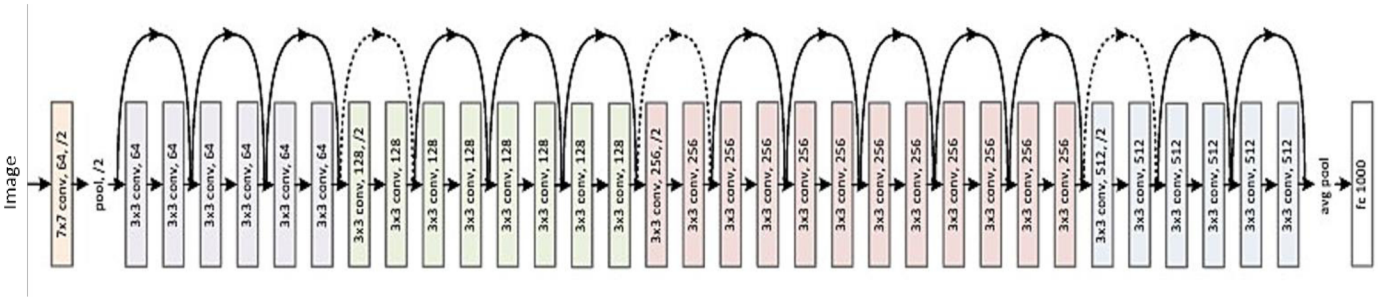


Figura 22 - Arquitectura Resnet50.
Fuente: Recuperado de https://storage.googleapis.com/kaggle-datasets-images/6209/9122/9dbea1f498f8d02e6c26ead7f130af04/data-original.png?t=2017-12-06-02-54-47 (2017)

2.6.4. InceptionV3

Es una red neuronal convolucional propuesta por Google en el año 2015, la Inception V3 no posee muchas capas ocultas, en su lugar se compone de diferentes "módulos inception" que son pequeñas operaciones en paralelo dentro de la red. (Ver Figura 23)

Esta arquitectura presenta como novedad la idea de concatenar los resultados obtenidos de aplicar diferentes filtros de convolución de diferente tamaño, ventanas de pooling a una misma entrada.

Esto permite al modelo beneficiarse de la extracción de características a múltiples niveles en un único paso y al, mismo tiempo, además extrae características generales y específicas a la vez. La filosofía de esta arquitectura es la de por qué tener que elegir entre diferentes opciones si se pueden todas. Esto hace que el diseño de la inception sea uno de los más complicados, pero a su vez el más eficiente. También, se agregan unas capas clasificadoras auxiliares que ayudan a asegurar que los parámetros que se extraen en las capas ocultas son de utilidad para realizar la predicción, esto tiene un efecto de regularización sobre toda la red y previene el sobre entrenamiento (overfitting).

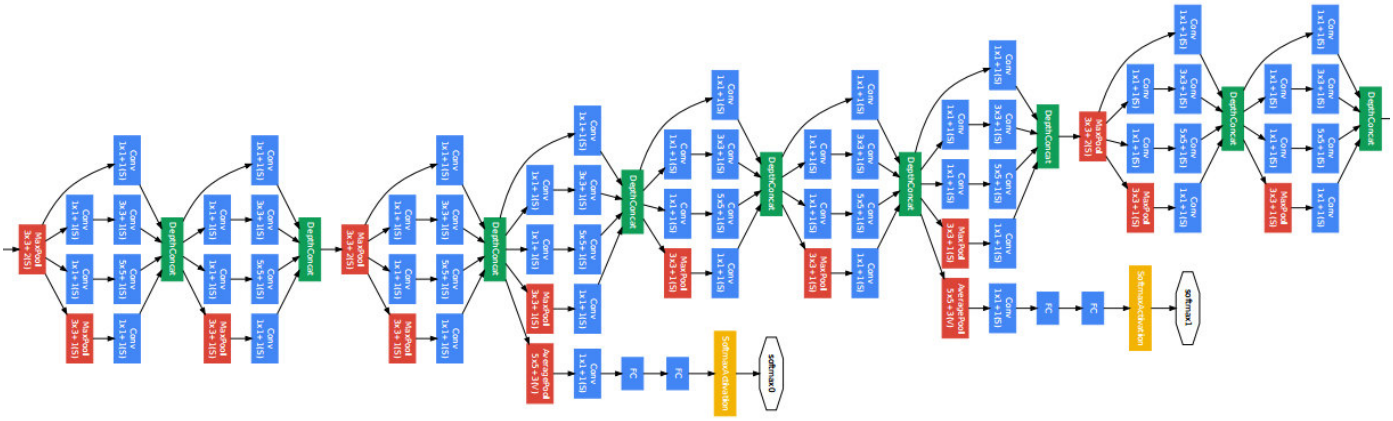


Figura 23 - Arquitectura Inception V3.
Fuente: Recuperado de https://www.deeplearningitalia.com/wp-content/uploads/2018/06/1.png (2018)

2.6.5. Xception

Esta arquitectura es una extensión de la anterior (Inception), también desarrollada por Google. La principal diferencia es que introduce el concepto de convolución separable por profundidad. Es decir, en lugar de afrontar la convolución como un único paso en el que se obtiene un único mapa de características resultado de convolucionar un filtro con la entrada, se obtiene un mapa por cada canal, se realiza una convolución independiente de cada canal de la entrada con cada canal de filtro.

2.6.6. Mobilenet

MobileNet es una arquitectura creada por Google pensada y optimizada para aplicaciones de visión móviles. Su principal característica es reducir lo más posible la potencia computacional necesaria para el algoritmo. Es una arquitectura muy sencilla a pesar de que esto le haga sacrificar algo de precisión y rendimiento. Esto lo consigue en gran parte gracias a la utilización de convoluciones separables por profundidad.

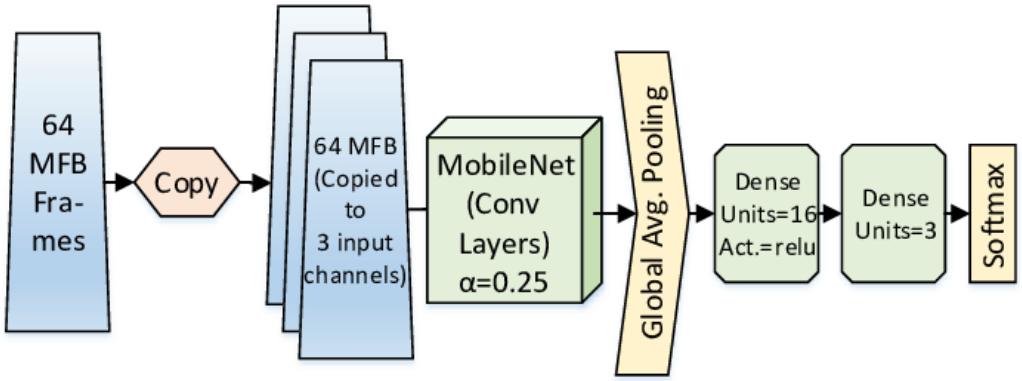


Figura 24 - Arquitectura MobileNet V2.
Fuente: Recuperado de https://www.researchgate.net/figure/Network-Architecture-of-MobileNet_fig1_328654938 (2018)

2.7. Preparación del set de datos

Una de las tareas más importantes al modelar una red neuronal artificial es la preparación del conjunto de datos. La correcta elección de la base de datos para el entrenamiento es esencial para llegar a los resultados deseados. La red debe conocer de igual manera todos los elementos que se quieren predecir, por lo que los prototipos de entrenamiento deben ser equilibrados y balanceados para todas las clases a detectar y además se debe disponer de una enorme cantidad de datos.

Para preparar un dataset se debe tener en cuenta la selección o recolección y luego el procesamiento de los mismos para que al entrenar el modelo se pueda obtener una convergencia óptima. Hay que tener en cuenta el formato de los datos, ya que serán usados como entradas para el modelo, eliminar ruido o redundancia, normalizar las dimensiones.

Los resultados del entrenamiento tienen varios resultados posibles:

- No convergencia: el algoritmo no es capaz de encontrar una solución, por lo tanto, durante las iteraciones del entrenamiento nunca mejora el acierto del modelo.
- Convergencia: el algoritmo encuentra una solución al problema, pero esta solución tiene tres resultados posibles, underfitting (subajuste), overfitting (sobreajuste) y entrenamiento óptimo.

El desafío central es que el algoritmo tenga buen rendimiento sobre entradas nuevas no observadas previamente, esta habilidad de poder desempeñarse correctamente sobre entradas nuevas se denomina “generalización”. En el caso de una convergencia con entrenamiento óptimo la red genera predicciones con mucha

precisión, pero a su vez ha desarrollado la capacidad de generalización que le permite predecir correctamente ante nuevos datos de entrada.

Existen diversas técnicas para evitar las patologías típicas de un mal entrenamiento. Sin embargo, con una correcta elección de parámetros y una base de datos equilibrada, los algoritmos actuales son capaces de alcanzar buenas soluciones.

Se debe encontrar un punto medio en el aprendizaje del modelo en el que no se esté incurriendo en underfitting y tampoco en overfitting, para eso se debe ir ajustando poco a poco los hiperparametros de la red. Tanto los casos de underfitting como los de overfitting se detallan en los siguientes apartados.

2.7.1. Recolección de imágenes

La recolección de datos se puede realizar de forma manual buscando información en distintos medios como internet teniendo en cuenta el origen de los datos, bases de datos relacionales, planillas de cálculos, conjuntos de imágenes, archivos de texto plano o archivos ".csv". Luego se procesa la información para dejar solamente las características (features) correctas, solo se dejan las necesarias para el modelo que se desea implementar. En este punto es un buen momento para realizar la normalización de los datos.

Aunque, también existen técnicas y algoritmos que permiten realizar esta tarea de forma automática, estas son las técnicas de "scraping". Donde se puede destacar el web scraping e image scraping, en ambos casos se aplican algoritmos que realizan la búsqueda de determinados parámetros dentro de un sitio web determinado, en el primer caso para recolectar información y en el segundo para recolectar imágenes.

2.7.2. Prevención de ataques adversarios

Como ya se ha mencionado durante la última década se lograron numerosos avances en el campo del aprendizaje automático. Esto se debe fundamentalmente a dos motivos, por un lado, a la gran cantidad de datos disponibles, los cuales son utilizados para entrenar los modelos y, por otro lado, las mejoras en el hardware que posibilitan un mayor poder computacional. Estos avances permitieron obtener excelentes resultados utilizando redes neuronales profundas en distintas áreas.

Estas técnicas son cada vez más aplicadas a distintas problemáticas de la vida real, como vehículos autónomos, traductores automáticos, publicidad, entre otras.

A pesar del buen desempeño alcanzado por las redes neuronales, se ha descubierto que son altamente sensibles a los denominados "ejemplos adversarios", datos de entrada ligeramente alterados con el objetivo de obtener resultados incorrectos por parte de la red. Entre los escenarios más críticos se encuentran los sistemas de reconocimiento facial y los vehículos autónomos, en donde un atacante podría manipular la percepción de los modelos para causar que se detecte a un rostro diferente al real o modificar la toma de decisiones de un vehículo autónomo.

Se llama "*Ataques Adversarios*" a los distintos algoritmos diseñados para generar ejemplos adversarios. Este tipo de ataque se comenzó a estudiar a partir del año 2014 y, originalmente se detectó en redes de clasificación de imágenes donde se descubrió que era posible para un atacante generar imágenes con perturbaciones casi imperceptibles a la vista, pero que aun así generaban una clasificación errónea por parte de los modelos de redes neuronales. Estos ataques demuestran una contradicción en la gran capacidad de generalización que presentan las redes neuronales. Además, se ha demostrado que los ataques adversarios no solo afectan a la red para la cual fueron generados, sino que también son transferibles a otras redes con distinta arquitectura, entrenadas con distintos hiperparámetros e incluso sobre otro conjunto de datos.

Existen dos maneras de introducir estos ataques a una red, por un lado, se puede abordar como un problema de optimización, en este caso se parte de un conjunto de datos de entrada "limpios", que pueden interpretarse como puntos en un espacio multidimensional, y lo que busca el ataque es desplazar estos puntos en la dirección donde se maximice el error de la red. Logrando de esta manera ejemplos capaces de engañar a la red y al mismo tiempo se asemejan a los datos naturales. Por otro lado, también se ha demostrado que es posible trasladar los ataques al mundo físico, es decir, la perturbación primero es llevada al mundo físico, por ejemplo, pegando distintas etiquetas en una señal de tránsito y luego sacar la foto que será pasada como entrada del modelo o también sacando una foto, luego imprimiéndola para posteriormente volver a sacar una foto y presentarla al modelo, encontrándose sujeta a distorsiones y diversos factores externos. Esto eleva aún más el riesgo propuesto por los ataques adversarios, ya que se extienden los posibles escenarios en los que pueden ser utilizados.

Figura 25 - Ejemplo adversario generado por un ataque.
Fuente: Recuperado de https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT4ruAwwwXT-rMjw4ZFbCLFxLVb1JE8amcTWQ&usqp=CAU (2019)

Figura 26 - Simio Gibón.
Fuente: Recuperado de https://www.monkeyworlds.com/wp-content/uploads/Gibon_informacion.jpg

Existen estrategias proactivas y reactivas para prevenir estos ataques. La estrategia proactiva consiste en entrenar los modelos no solo con imágenes limpias o reales, sino también, hacerlo con imágenes que contengan ruido para así indicar al modelo cuáles corresponden a ejemplos correctos y cuáles no. Pero, este tipo de estrategia no tiene buenos resultados debido a lo que ya mencionamos, los ataques adversarios presentan la característica de ser transferibles; por lo tanto, un atacante podría armar su propio modelo basado solo en ejemplos adversarios y una vez que obtenga los resultados esperados utilice estas imágenes para atacar cualquier otro modelo que desee.

La estrategia reactiva consiste en ocultar los detalles del modelo a los atacantes, por ejemplo, si el modelo desarrollado en un modelo en donde se realiza una detección de objetos ya sea en imágenes o en tiempo real (video), el resultado que visualiza el usuario de dicha aplicación sólo debería indicar qué tipo de objeto es el que se identificó y no mostrar la distribución de probabilidades de dicha predicción. Saber el porcentaje de la predicción le daría a un posible atacante la ventaja de saber cuánto debe manipular una imagen de entrada para lograr

confundir al sistema. Aunque, este tipo de estrategia parezca tener mejores resultados, es vulnerable a ataques de fuerza bruta, en donde solo será cuestión de tiempo lograr confundir al modelo.

Las últimas novedades en este campo demuestran que todo modelo de clasificación de imágenes durante su entrenamiento aprende dos tipos de características, robustas y no robustas. Las características no robustas (detalles más finos) son las que para el ojo humano aparentemente no contienen información relevante, pero para el modelo DL esta información es altamente predictiva y esencial para la clasificación, por ejemplo, si se tratase de extraer características de una imagen de un animal, una característica no robusta sería la orientación o grosor del pelo. Mientras que las características robustas sí son perceptibles a primera vista, siguiendo el ejemplo anterior una característica robusta sería el contorno de la cara, una oreja o una nariz, etc. Los ataques de ejemplos adversarios tienden a afectar las características no robustas en la imagen y, por lo tanto, si nuestro modelo tiene una alta dependencia con este tipo de características será más vulnerable ante estos ataques.

Entonces, para prevenir el efecto causado por los ataques adversarios se debe buscar que el set de datos contenga imágenes en las que predominen las características robustas logrando un modelo con mayor eficiencia y menor vulnerabilidad.

2.7.3. Normalización

Antes de comenzar con el entrenamiento de la red neuronal es conveniente realizar un análisis y normalización de los datos para que todos estén en un mismo rango de valores. Este procedimiento ayuda a mejorar la convergencia del modelo. Una manera de realizar la normalización de los datos consiste en dividir cada dimensión por su desviación estándar una vez que haya sido centrada en cero, logrando que las dimensiones tengan aproximadamente la misma escala. En el caso de las CNN se trabaja con imágenes como datos de entrada y estas imágenes son representadas mediante una matriz de píxeles cuyos valores van de 0 a 255, por lo que sus dimensiones ya se encuentran en una escala parecida y no es estrictamente necesario normalizar los datos de entrada, aunque se suele dividir el valor de cada pixel por 255 para conseguir que los valores estén todos dentro del rango que va de 0 a 1.

2.7.4. División en entrenamiento, validación y testeo

Los algoritmos de entrenamiento necesitan realizar una separación en la base de datos, una parte se utiliza para el entrenamiento y otra para la validación, siendo esta la división más común, aunque también se puede optar por hacer una división más como se muestra en la siguiente imagen (Figura 27) en donde se guarda una parte del set de datos para el testeo o prueba del modelo. El conjunto de pruebas puede ser un conjunto aparte de datos ya que este no debe estar ni siquiera etiquetado y es utilizado para comprobar la eficacia real del modelo, en cambio, el set de entrenamiento debe estar etiquetado y es el conjunto de datos que se utilizará para realizar el aprendizaje (extracción de características). A su vez, el set de validación también se encuentra etiquetado y es utilizado luego de cada iteración de entrenamiento para comprobar si el modelo está realizando una buena generalización y/o detectar problemas de subajuste o sobreajuste del modelo. Tanto los datos de validación como los de testeo son datos que el modelo no conoce.

Si bien no existe una regla que indique cómo dividir el dataset, es conveniente que el conjunto de entrenamiento sea mayor que el conjunto de validación siendo una división muy usual la de 80-20% o 70-30% siendo 80% y 70% lo correspondiente al set de entrenamiento y 20% y 30% lo correspondiente al set de validación en cada caso.

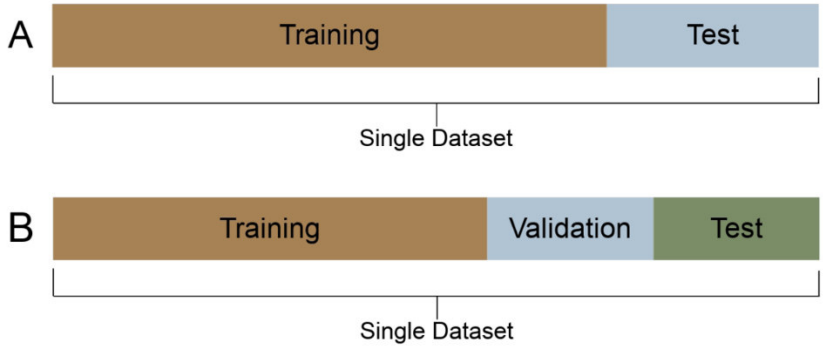


Figura 27 - División del set de datos para entrenamiento, validación y testeo.
Fuente: Recuperado de https://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/ML_dataset_training_validation_test_sets.png/1200px-ML_dataset_training_validation_test_sets.png

2.7.5. Underfitting

El modelo presenta underfitting o subajuste cuando no es capaz de detectar las características que hay detrás de los datos. Esto se refleja en un alto error tanto para el set de entrenamiento como para el set de validación y es un modelo incapaz

de generalizar en ningún caso. Esto puede ocurrir por no contar con la cantidad y variedad de datos suficientes, o que el modelo no sea lo suficientemente complejo como para detectar las características particulares de los datos de entrada (se debe aumentar al número de capas y/o el número de neuronas del modelo), o las iteraciones de entrenamiento no son las suficientes como para obtener buenos resultados, o bien puede ocurrir que la arquitectura elegida no sea la más adecuada para analizar y resolver el problema en cuestión (ej. utilización de una RNA para el análisis de imágenes en vez de una CNN).

2.7.6. Overfitting

El sobreajuste u overfitting ocurre cuando el modelo clasifica muy bien los datos de entrenamiento, pero no es capaz de clasificar bien los datos que no ha visto nunca, es decir, el modelo no generaliza lo aprendido durante el entrenamiento. El objetivo del algoritmo de aprendizaje es aprender a partir de los datos de entrenamiento a predecir la categoría de los datos que no ha visto nunca antes.

El overfitting es un problema habitual en el aprendizaje automático y especialmente en el caso de las redes neuronales con muchas capas (Deep learning).

Entre los motivos por los que un modelo presenta overfitting se encuentran los siguientes: el modelo propuesto es demasiado complejo (contiene un exceso de capas y/o neuronas en las capas) y el set de datos no contiene la cantidad suficiente de datos para realizar un buen entrenamiento. Existen varias técnicas para evitar o minimizar el sobreajuste, entre las que se destacan “data augmentation”, “dropout” y “early stopping”, técnicas que se describen en los siguientes apartados.

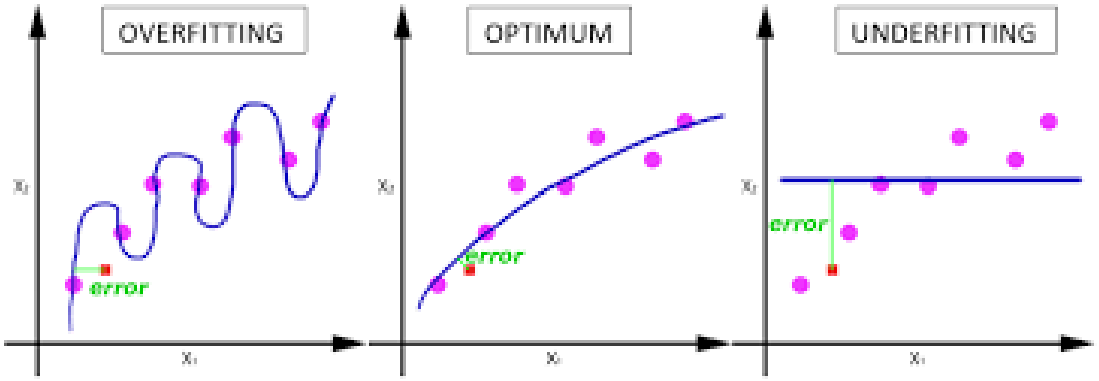


Figura 28 - Comparación entre ajustes de datos.
Fuente: Recuperado de https://docs.paperspace.com/machine-learning/wiki/overfitting-vs-underfitting (2020)

2.7.6.1. Data Augmentation

La mejor manera en la que una red neuronal generalice bien es contar con la construcción de una buena base de datos con la mayor cantidad de datos posible, pero en la práctica esto es una tarea muy difícil de realizar ya que se cuenta generalmente con una cantidad limitada de datos. Para resolver este problema existen técnicas para aumentar los datos artificialmente, esta técnica se conoce como *data augmentation* (o *image augmentation*). Consiste en aumentar la base partiendo de prototipos conocidos, modificándolos ligeramente para generar nuevos a partir de ellos. Cuando se trata de redes neuronales aplicadas al reconocimiento de imágenes es común generar variaciones de los prototipos realizando rotaciones, traslaciones, aplicando aumentos, invirtiéndolas, variando resolución, ruido, brillo y modificando el espacio de color para crear nuevas imágenes con las que se entrenará la red.

En general la clase correcta con la que se identifica cada imagen no cambia al aplicar estas modificaciones, pero se debe tener cuidado para que esto no suceda (ej. si tuviéramos una imagen con el número 6 y se le aplica una rotación de 180° este se convertiría en un 9, cambiando la clase original).

La eficacia de esta técnica reside en la forma en que las redes neuronales entienden las imágenes y sus características, si una imagen se modifica ligeramente es percibida por la red como una imagen completamente distinta perteneciente a la misma clase. Esto disminuye la probabilidad de que la red se centre en orientaciones o posiciones mientras mantiene la relevancia de las características deseadas.

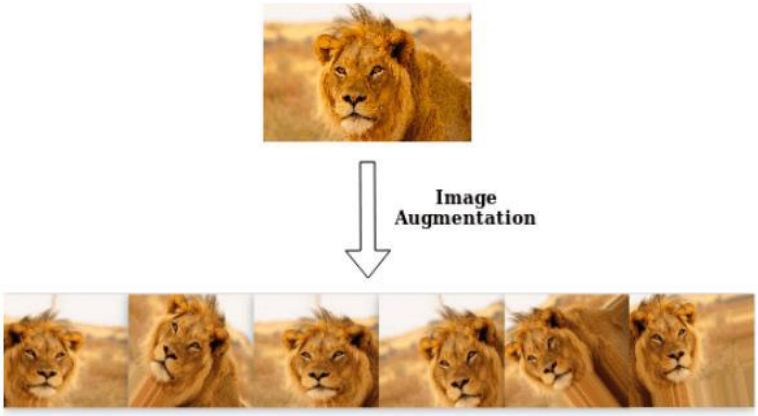


Figura 29 - Ejemplo de Image Augmentation.
*Fuente: Recuperado de https://miro.medium.com/max/605/0*Utma-dS47hSoQ6Zt (2019)*

2.7.6.2. Dropout

La técnica de *dropout* consiste en desactivar un número determinado de neuronas de una red neuronal de forma aleatoria. Las neuronas desactivadas no se tienen en cuenta durante la etapa de alimentación hacia adelante (*feedforward*), ni en la propagación hacia atrás (*backpropagation*), lo que obliga a las neuronas cercanas a no depender de las neuronas desactivadas. Esto reduce el overfitting ya que las neuronas necesitan trabajar de mejor forma individualmente para no depender tanto de las relaciones con las neuronas vecinas.

Al aplicar *dropout* a una capa de neuronas se indica con un parámetro que va de 0 a 1, el cual representa el porcentaje de neuronas que se desactivarán en cada iteración. Cuando los valores son cercanos a 0 el *dropout* desactivará menos neuronas y cuando el valor se acerque a 1 ocurre lo contrario.

Cabe destacar que el *dropout* solo se usa durante la fase de entrenamiento de la red. En un entrenamiento sin *dropout*, como se ve en la parte izquierda de la Figura 30, la red actualiza todos los pesos de sus neuronas en cada iteración, mientras que utilizando *dropout* se desactivan neuronas aleatoriamente y se utiliza una subred de la original lo que impide a las neuronas generar una dependencia fija entre sí, como se puede ver en la parte derecha de la Figura 30. Esta dependencia ocurre cuando dos o más neuronas consecutivas dependen mucho entre ellas para detectar features, en vez de que cada neurona busque un tipo particular de feature como ya se había mencionado en párrafos anteriores. Entre las ventajas de aplicar dropout también se encuentra el bajo costo computacional que tiene aplicarla y,

además, no limita significativamente el modelo o procedimiento de entrenamiento que se utiliza.

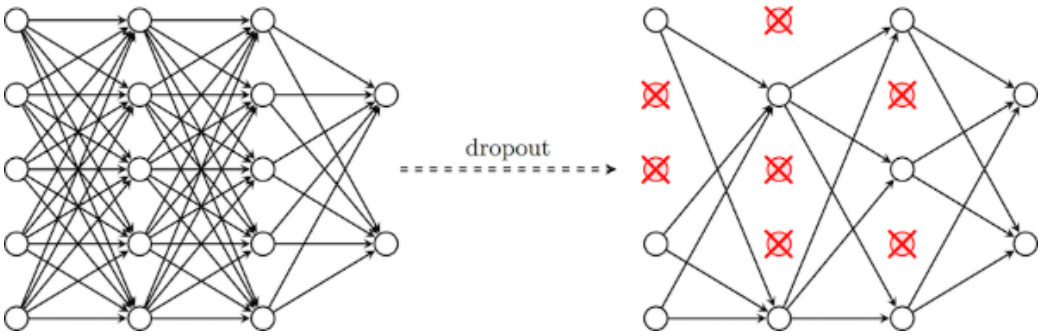


Figura 30 - Red normal vs. Red con Dropout.
Fuente: Recuperado de https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcS9uGLJbIh_S2NPnH4yvjmKnnVueqWYZiEd1w&usqp=CAU

2.7.6.3. Early stopping

Cuando se realiza el entrenamiento de grandes modelos compuestos por dataset con la suficiente representación, suele ocurrir que el error de entrenamiento decrece de manera constante durante las iteraciones, pero en algún punto el error de validación deja de disminuir y comienza a aumentar. Cuando esto ocurre sería conveniente poder volver atrás las modificaciones que sufrieron los hiperparámetros, regresando los valores en donde el error de validación era menor. Esto se logra guardando una copia de los valores de los hiperparámetros de la red en cada iteración siempre y cuando el error de validación vaya disminuyendo y, posteriormente, si el error de validación no mejora luego de un número prefijado de iteraciones, el modelo restaurará automáticamente los valores de los hiperparámetros guardados. Esta estrategia se conoce como "*early stopping*" y es uno de los métodos de regularización comúnmente usados en redes neuronales, tanto por su simplicidad como por su eficacia.

Esta técnica es una forma de seleccionar y determinar de manera automática el parámetro "épocas de entrenamiento" y, además, es una buena forma de controlar la capacidad efectiva del modelo, determinando cuántas iteraciones puede tomar el algoritmo para ajustar de mejor manera el set de entrenamiento.

Una de las ventajas de esta técnica utilizada como forma de regularización es que no requiere cambios en el procedimiento de entrenamiento, la función objetivo, o la red, por lo que es fácil de utilizar sin modificar la dinámica del

aprendizaje. También puede ser utilizada en conjunto con otras técnicas de regularización.

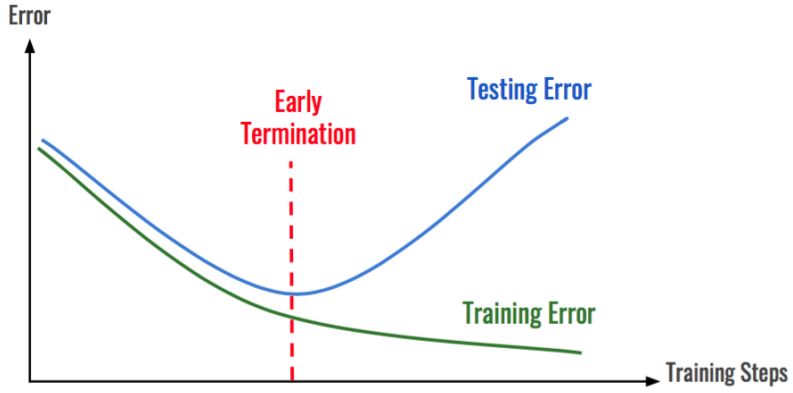


Figura 31 - Entrenamiento con Parada Temprana.
*Fuente: Recuperado de https://miro.medium.com/max/973/1*nhmPdWSGh3ziatQKOmVq0Q.png (2020)*

2.8. Matriz de confusión

Existen diferentes maneras de medir o evaluar el desempeño de un modelo de clasificación, una de las métricas más intuitivas y sencillas es la matriz de confusión. La matriz de confusión es una matriz cuadrada que nos permite visualizar de forma rápida y efectiva lo bien o mal que ha clasificado el modelo. En un ejemplo de clasificador binario la matriz de confusión tiene la estructura que se puede ver en la Figura 20.

Una vez identificado el problema, la matriz de confusión, es una tabla con dos dimensiones ("Valor real" y "Predicción") y conjunto de "clases" en ambas dimensiones, las clasificaciones reales son las filas y las clasificaciones predichas son las columnas. Para generar la matriz de confusión, los resultados obtenidos por el modelo de aprendizaje se dividen en:

- Verdaderos Positivos (VP): es la cantidad de positivos que fueron clasificados correctamente como positivos.
- Verdaderos Negativos (VN): es la cantidad de negativos que fueron clasificados correctamente como negativos.
- Falsos Negativos (FN): es la cantidad de positivos que fueron clasificados incorrectamente como negativos.
- Falsos Positivos (FP): es la cantidad de negativos que fueron clasificados incorrectamente como positivos.

En un escenario ideal no deberían existir ni falsos positivos ni falsos negativos, pero esto no sucede en la realidad, si se encuentra un modelo de ML que presente resultados ideales seguramente se esté ante un problema de sobre ajuste de datos. En realidad, la matriz de confusión en sí misma no es una medida de rendimiento como tal, pero casi todas las métricas de rendimiento están basadas en ella y los valores que contiene. A partir de la matriz de confusión se pueden obtener las siguientes métricas:

Recall: da información sobre el desempeño de un clasificador con respecto a falsos negativos, cuanto falla el modelo.

$$R = Recall = \frac{VP}{VP+FN} \qquad \text{(ec. 7)}$$

Precision: da información sobre el desempeño respecto a los falsos positivos.

$$P = Precision = \frac{VP}{VP+FP} \qquad \text{(ec. 8)}$$

Especificity: da información sobre la tasa de verdaderos negativos. Expresa cuán bien puede el modelo detectar esa clase.

$$S = Specificity = \frac{VN}{VN+FP} \qquad \text{(ec. 9)}$$

Accuracy: es el número de predicciones correctas realizadas por el modelo sobre todo tipo de predicciones realizadas.

$$A = Accuracy = \frac{VP+VN}{VP+FP+FN+VN} \qquad \text{(ec. 10)}$$

F1-Score: es el promedio ponderado de la precisión y el recall, por lo tanto, tiene en cuenta los FP y los FN (considera el desbalanceo de las clases), aportando así una generalización de los resultados.

$$F1 = F1 - Score = \frac{2\ X\ P\ X\ R}{P+R} \qquad \text{(ec. 11)}$$

Matriz de confusión para una variable de salida dicotómica (clasificación)		Resultado de la predicción	
		Sí	No
Valor real de la clase	Sí	Verdadero positivo	Falso Negativo
	No	Falso positivo	Verdadero negativo

Figura 32 - Matriz de Confusión.
Fuente: Recuperado de https://micromedix.files.wordpress.com/2018/01/matriz-de-confusic3b3n.jpg (2018)

3. Herramientas utilizadas

En este capítulo se presentan todas las herramientas utilizadas durante el desarrollo y modelado de todas las redes convolucionales implementadas durante el proyecto.

Figura 33 - Herramientas utilizadas.
Fuente: Elaboración propia.

3.1. Python

Python es actualmente el lenguaje de programación de más rápido crecimiento, tiene una serie de características que lo hacen muy particular y que, sin dudas, le aportan muchas ventajas. Es un lenguaje de programación multiparadigma soportando la orientación a objetos, programación imperativa y programación funcional. Es interpretado, usa tipado dinámico, open source y es multiplataforma.

Este lenguaje está ganando usuarios que se dedican al campo de la inteligencia artificial. El crecimiento en el uso se debe fundamentalmente a las nuevas tecnologías de inteligencia artificial, campo donde junto con el lenguaje R dominan. Aunque, R es un lenguaje que proviene del mundo de la estadística y Python, por otro lado, es un lenguaje de propósito general y su uso está más extendido. Además, esta popularidad se debe a su alto nivel y gran número de librerías.

Como se verá más adelante, es un lenguaje que facilita la depuración del código y, además, es uno de los lenguajes de programación oficiales de Google. Por todas estas características es un lenguaje con muchísimo soporte y relativamente fácil de aprender.

El framework Tensorflow y la librería de alto nivel Keras utilizados para el entrenamiento de redes neuronales son APIs que pueden ser integradas en Python, siendo esta la razón de la elección para la integración en el proyecto.

3.2. Anaconda

Anaconda es una distribución libre y abierta de los lenguajes Python y R, utilizada en ciencia de datos y aprendizaje automático (machine learning). Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos. Está orientado a simplificar el despliegue y administración de los paquetes de software.

Las diferentes versiones de los paquetes se administran mediante el sistema de gestión de paquetes Conda, el cual lo hace bastante sencillo de instalar, correr, y actualizar software de ciencia de datos y aprendizaje automático como ser Scikit-team, TensorFlow y SciPy.

3.3. Google colab

Colab es un servicio cloud, basado en los Notebooks de Jupyter, que permite el uso gratuito de las GPUs y TPUs de Google, con librerías como: Scikit-learn, PyTorch, TensorFlow, Keras y OpenCV. Todo ello bajo Python 2.7 y 3.6, aún no está disponible para R y Scala.

Aunque tiene algunas limitaciones, que pueden consultarse en su página de FAQ, es una herramienta ideal, no solo para practicar y mejorar los conocimientos en técnicas y herramientas de Data Science, sino también para el desarrollo de aplicaciones (pilotos) de machine learning y deep learning, sin tener que invertir en recursos hardware o del Cloud.

Con Colab se pueden crear notebooks o importar los que ya se tengan creados, además de compartirlos y exportarlos cuando se desee. Esta fluidez a la hora de manejar la información también es aplicable a las fuentes de datos que se use en nuestros proyectos (notebooks), de modo que se puede trabajar con

información contenida en nuestro propio Google Drive, unidad de almacenamiento local, github e incluso en otros sistemas de almacenamiento cloud, como el S3 de Amazon.

3.4. Tensorflow

TensorFlow es una librería de código abierto para Machine Learning desarrollada originalmente por científicos e ingenieros del equipo Google Brain del departamento de Machine Learning de Google y fue liberado como software de código abierto a finales del año 2015. Es una plataforma con soportes para construir gran variedad de redes neuronales, aunque gracias a su flexibilidad y gran rendimiento para computación numérica también es usada en otros campos científicos.

Esta librería de computación matemática ejecuta de forma rápida y eficiente gráficos de flujo. Estos gráficos están formados por operaciones matemáticas representadas sobre nudos y cuyas entradas y salidas son vectores multidimensionales de datos, conocidos como tensores. Con un tensor se hace referencia a un conjunto de valores primitivos, por ejemplo, números enteros o flotantes, organizados por un arreglo de 1 o N dimensiones, donde el rango del tensor sería el número de dimensiones (ver Figura 34).

Google usa TensorFlow para implementar ML en casi todas las aplicaciones, por ejemplo, al utilizar Google fotos o búsqueda por voz de Google se utiliza indirectamente TensorFlow que trabaja en modelos utilizados por grandes grupos de hardware de Google y son poderosos en tareas de percepción.

Keras y TensorFlow se combinan tan bien ya que Keras permite abstraerse de estas operaciones vectoriales gracias a su interfaz sencilla y sintaxis homogénea.

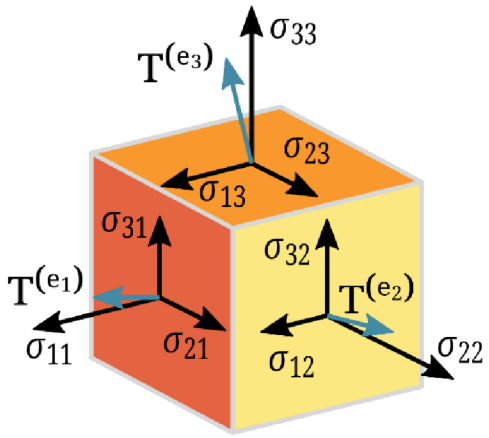


Figura 34 - Representación de un tensor.
Fuente: Recuperado de https://upload.wikimedia.org/wikipedia/commons/thumb/4/45/Components_stress_tensor.svg/1200px-Components_stress_tensor.svg.png

3.5. Keras

Keras es una Interfaz de Programación de Aplicaciones (API) de alto nivel y de código abierto escrita en Python. Está especialmente diseñada para facilitar una rápida experimentación en Deep Learning. Permite crear prototipos rápidos de redes profundas y llevar a cabo su entrenamiento de forma cómoda y rápida, a través de la facilidad de uso, la modularidad y la extensibilidad. Está diseñado para correr sobre otros frameworks de más bajo nivel como TensorFlow, CNTK o Theano como motores de backend. Funciona perfectamente con CPU o GPU, también soporta CNNs, redes neuronales recurrentes (RNN) o una combinación de ambas.

3.6. Otras librerías

- Jupyter Notebook: (anteriormente IPython Notebooks) es un entorno informático interactivo basado en la web para crear documentos de Jupyter notebook. El término "notebook" puede hacer referencia coloquialmente a muchas entidades diferentes, principalmente la aplicación web Jupyter, el servidor web Jupyter Python o el formato de documento Jupyter según el contexto. Un documento de Jupyter Notebook es un documento JSON, que sigue un esquema versionado y que contiene una lista ordenada de celdas de entrada/salida que pueden incluir código, texto (usando Markdown), matemáticas, gráficos y texto enriquecidos, generalmente terminado con la extensión ".ipynb".

El Jupyter Notebook es una aplicación web de código abierto que permite crear y compartir documentos que contienen código vivo, ecuaciones, visualizaciones y texto narrativo. Entre sus usos se destacan la limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización de datos y aprendizaje automático.

- Numpy: librería de Python orientada al cálculo vectorial y matricial. Permite trabajar con matrices N-dimensionales, además de implementar algebra lineal, transformada de Fourier y muchas funciones relacionadas con estadísticas.

- Pandas: es una librería de Python concebida como una extensión Numpy, que facilita el procesamiento de tablas, series temporales y otras estructuras de datos complejas.

- ScikitLearn: es una biblioteca de código abierto, que provee de herramientas científicas para el análisis de datos y el data mining.

- Pillow: es una librería para el manejo de todo tipo de imágenes.

- Matplotlib: es una librería específica para la generación de gráficos en Python, incluye la representación 2D y 3D de funciones e imágenes.

- OS: Permite acceder a funcionalidades dependientes del sistema operativo. Sobre todo, aquellas que refieren información sobre el entorno del mismo y facilitando la manipulación de la estructura de directorios.

3.7. Hardware

El uso de técnicas de Machine Learning y en especial de Deep Learning como es el caso de las redes neuronales convolucionales implica un costo computacional muy elevado, debido a que están compuestas por un gran número de capas y neuronas y, además, trabajan con gran cantidad de imágenes que en muchos casos tienen una alta resolución.

El CPU de una computadora realiza operaciones matemáticas con una velocidad muy elevada, pero las ejecuta de a una por vez, o al menos una operación por núcleo. En cambio, las tarjetas gráficas o GPU como se puede apreciar en la Figura 35 disponen de muchos más núcleos por lo que pueden realizar muchas más operaciones en el mismo tiempo.

Una GPU (Unidad de Procesamiento Gráfico) es un coprocesador dedicado al procesamiento de imágenes y operaciones de coma flotante. Básicamente, al ser otro procesador añadido, su función es la de liberar la carga del CPU, aumentando el rendimiento de nuestro ordenador. En realidad, los procesadores principales ya tienen una GPU integrada, pero de potencia reducida y son las tarjetas gráficas

(placas de video) las que tienen unas GPUs potentes. Toda esta potencia de cálculo aritmético puede emplearse para la computación de los algoritmos de Deep Learning.

La diferencia entre una CPU y una GPU está en la cantidad de núcleos. Una CPU tiene menos cantidad de núcleos y está enfocada al trabajo en tareas secuenciales que pueden ser más o menos complejas. Por otra parte, la GPU tiene cientos o incluso miles de núcleos optimizados para trabajar en paralelo con operaciones sencillas, por lo que está especialmente preparada para el cálculo matricial.

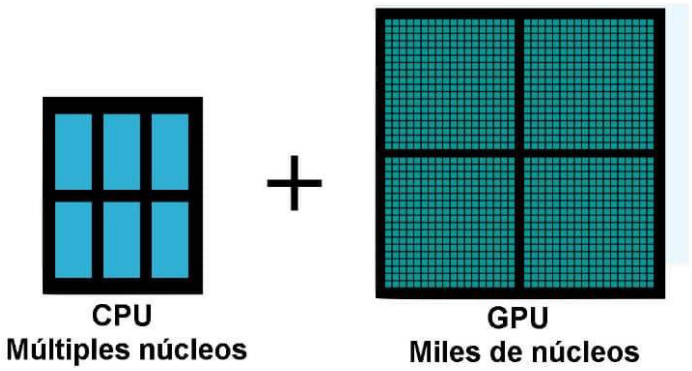


Figura 35 - Comparación entre núcleos de CPU y núcleos de GPU.
Fuente: Recuperado de https://miracomosehace.com/wp-content/uploads/2020/05/CPU-mas-GPU.jpg (2020)

4. Modelos propuestos

Durante el desarrollo del proyecto se implementaron aproximadamente 80 modelos de redes neuronales convolucionales, 50 de estos modelos fueron implementados con aprendizaje desde cero y el resto de los modelos fueron desarrollados utilizando técnicas de transferencia de aprendizaje. A su vez, las pruebas se implementaron con 2 sets de datos, el primero con una división de dos clases, reciclables y no reciclables y, el segundo conjunto de datos, fue dividido en 6 clases, papel/cartón, vidrio, plástico, metal, orgánico y no reciclables. A continuación, se detallará mejor el contenido de cada set de datos y se mostrará la arquitectura de los modelos con los cuales se obtuvo un mejor resultado, tanto para los modelos entrenados desde cero como para los modelos en donde se utilizó aprendizaje por transferencia.

4.1. Modelo con 2 clases

En este caso los datos fueron divididos en 2 clases (reciclable y no reciclable), la cantidad total de imágenes de este set es de 8340 imágenes muy bien balanceadas en ambas clases, a su vez, se dividieron los datos en tres partes separando 100 imágenes para el testeo del modelo, 1708 imágenes para la validación y 6532 imágenes utilizadas para el entrenamiento. Antes de ser enviadas al modelo como entradas, se realizó un redimensionamiento por software a cada una de las imágenes para que todas tengan un tamaño de 200 x 200 píxeles. El mismo proceso de redimensionamiento se debe realizar a las nuevas imágenes que el modelo deberá clasificar luego del entrenamiento.

En los siguientes apartados se expondrán las arquitecturas que mejores resultados arrojaron para cada tipo de entrenamiento.

4.1.1. Entrenamiento desde cero

En la tabla a continuación (Figura 36) se detalla la arquitectura de la red convolucional modelada, en la que se utilizó un modelo secuencial en donde, además de la capa de entrada, se cuenta con 4 capas convolucionales con sus

respectivos pooling o agrupamiento y una función de activación “RELU” para cada una de ellas, luego de estas capas convolucionales se generan 128 mapas de características cada uno de un tamaño de 25 x 25 píxeles. Como siguiente paso estos mapas de características son pasados por una función de aplanamiento que genera un vector de 80000 elementos, que serán utilizados como entradas para la capa completamente conectada. En esta capa se modeló una red neuronal profunda con 3 capas densas cada una con dropout de un 75% de las neuronas en la primera capa y un total de 512 neuronas y un dropout de 50% en otras dos capas con 256 neuronas cada una. La función de activación para estas capas también es “RELU” y para finalizar el modelo cuenta con una capa de salida con 2 neuronas y activación “SoftMax” para realizar la clasificación final. El optimizador utilizado fue “Adam” y la función de costo a optimizar “categorical cross entropy”.

El modelo inicialmente se programó para realizar 150 épocas de entrenamiento con una tasa de aprendizaje inicial de 0.0005.

Model: "sequential"

Layer (type)	Output	Shape Param #
conv2d (Conv2D)	(None, 200, 200, 32)	896
max_pooling2d (MaxPooling2D)	(None, 100, 100, 32)	0
conv2d_1 (Conv2D)	(None, 100, 100, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 50, 50, 64)	0
conv2d_2 (Conv2D)	(None, 50, 50, 128)	32896
conv2d_3 (Conv2D)	(None, 50, 50, 128)	65664
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 128)	0
flatten (Flatten)	(None, 80000)	0
dense (Dense)	(None, 512)	40960512
dropout (Dropout 0,75)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131328
dropout_1 (Dropout 0,5)	(None, 256)	0
dense_2 (Dense)	(None, 256)	65792
dropout_2 (Dropout 0,5)	(None, 256)	0
dense_3 (Dense)	(None, 2)	514

Total params: 41,276,098
Trainable params: 41,276,098
Non-trainable params: 0

Figura 36 - Modelo con 2 clases entrenamiento desde cero.
Fuente: Elaboración propia, basada en la práctica.

4.1.2. Entrenamiento con transfer learning

Para este caso la red convolucional modelada se basa en un modelo pre entrenado de una arquitectura llamada XCEPTION, este modelo ya fue presentado en el apartado 2.6.5. El procedimiento para la implementación del modelo consistió en aprovechar todo el entrenamiento de la etapa de extracción de características ya provisto por el modelo Xception y reemplazar por completo la capa completamente conectada por un modelo de desarrollo propio. A la etapa de extracción de características solo se le modificó el tamaño esperado de las imágenes en la capa de entrada pasando de 224 x 224 píxeles, tamaño original del modelo, a 200 x 200 píxeles para que concuerde con el tamaño de imágenes utilizado en el modelo entrenado desde cero.

El proceso consta de 14 bloques convolucionales en donde se realiza en cada uno una serie de convoluciones separadas en paralelo, que posteriormente se concatenan antes de enviar los mapas de características al siguiente bloque convolucional. En el último bloque convolucional del modelo Xception se obtienen como salida 2048 mapas de características con imágenes de 7 x 7 píxeles cada uno. Como siguiente paso estos mapas de características son pasados por una función de aplanamiento que genera un vector de 100352 elementos, que serán utilizados como entradas para la capa completamente conectada. En esta capa se modeló una red neuronal profunda con 3 capas densas cada una con dropout de un 75% de las neuronas en la primera capa y un total de 1024 neuronas y un dropout de 50% en otras dos capas con 512 neuronas cada una. La función de activación empleada en estas capas es “RELU” y para finalizar el modelo cuenta con una capa de salida con 2 neuronas y activación “SoftMax” para realizar la clasificación final. El optimizador utilizado fue “Adam” y la función de costo a optimizar “categorical cross entropy”.

La cantidad total de parámetros del modelo es de 124.411.434 de los cuales solo se entrenaron 103.549.954 y, por lo tanto, solo se optimiza el peso de sus filtros y conexiones. El modelo inicialmente se programó para realizar 150 épocas de entrenamiento con una tasa de aprendizaje inicial de 0.0005. En la Figura 37 se muestra la arquitectura del modelo planteado.

Model: "xception"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 200, 200, 3)	0	
block1_conv1 (Conv2D)	(None, 99, 99, 32)	864	input_1[0][0]
block1_conv1_bn (BatchNormaliza	(None, 99, 99, 32)	128	block1_conv1[0][0]
block1_conv1_act (Activation)	(None, 99, 99, 32)	0	block1_conv1_bn[0][0]
block1_conv2 (Conv2D)	(None, 97, 97, 64)	18432	block1_conv1_act[0][0]
block1_conv2_bn (BatchNormaliza	(None, 97, 97, 64)	256	block1_conv2[0][0]
block1_conv2_act (Activation)	(None, 97, 97, 64)	0	block1_conv2_bn[0][0]
block2_sepconv1 (SeparableConv2	(None, 97, 97, 128)	8768	block1_conv2_act[0][0]
block2_sepconv1_bn (BatchNormal	(None, 97, 97, 128)	512	block2_sepconv1[0][0]
block2_sepconv2_act (Activation	(None, 97, 97, 128)	0	block2_sepconv1_bn[0][0]
block2_sepconv2 (SeparableConv2	(None, 97, 97, 128)	17536	block2_sepconv2_act[0][0]
block2_sepconv2_bn (BatchNormal	(None, 97, 97, 128)	512	block2_sepconv2[0][0]
conv2d (Conv2D)	(None, 49, 49, 128)	8192	block1_conv2_act[0][0]
block2_pool (MaxPooling2D)	(None, 49, 49, 128)	0	block2_sepconv2_bn[0][0]
batch_normalization (BatchNorma	(None, 49, 49, 128)	512	conv2d[0][0]
add (Add)	(None, 49, 49, 128)	0	block2_pool[0][0] batch_normalization[0][0]
block3_sepconv1_act (Activation	(None, 49, 49, 128)	0	add[0][0]
block3_sepconv1 (SeparableConv2	(None, 49, 49, 256)	33920	block3_sepconv1_act[0][0]
block3_sepconv1_bn (BatchNormal	(None, 49, 49, 256)	1024	block3_sepconv1[0][0]
block3_sepconv2_act (Activation	(None, 49, 49, 256)	0	block3_sepconv1_bn[0][0]
block3_sepconv2 (SeparableConv2	(None, 49, 49, 256)	67840	block3_sepconv2_act[0][0]
block3_sepconv2_bn (BatchNormal	(None, 49, 49, 256)	1024	block3_sepconv2[0][0]
conv2d_1 (Conv2D)	(None, 25, 25, 256)	32768	add[0][0]
block3_pool (MaxPooling2D)	(None, 25, 25, 256)	0	block3_sepconv2_bn[0][0]
batch_normalization_1 (BatchNor	(None, 25, 25, 256)	1024	conv2d_1[0][0]
add_1 (Add)	(None, 25, 25, 256)	0	block3_pool[0][0] batch_normalization_1[0][0]
block4_sepconv1_act (Activation	(None, 25, 25, 256)	0	add_1[0][0]
block4_sepconv1 (SeparableConv2	(None, 25, 25, 728)	188672	block4_sepconv1_act[0][0]
block4_sepconv1_bn (BatchNormal	(None, 25, 25, 728)	2912	block4_sepconv1[0][0]
block4_sepconv2_act (Activation	(None, 25, 25, 728)	0	block4_sepconv1_bn[0][0]
block4_sepconv2 (SeparableConv2	(None, 25, 25, 728)	536536	block4_sepconv2_act[0][0]
block4_sepconv2_bn (BatchNormal	(None, 25, 25, 728)	2912	block4_sepconv2[0][0]
conv2d_2 (Conv2D)	(None, 13, 13, 728)	186368	add_1[0][0]
block4_pool (MaxPooling2D)	(None, 13, 13, 728)	0	block4_sepconv2_bn[0][0]
batch_normalization_2 (BatchNor	(None, 13, 13, 728)	2912	conv2d_2[0][0]
add_2 (Add)	(None, 13, 13, 728)	0	block4_pool[0][0] batch_normalization_2[0][0]
block5_sepconv1_act (Activation	(None, 13, 13, 728)	0	add_2[0][0]
block5_sepconv1 (SeparableConv2	(None, 13, 13, 728)	536536	block5_sepconv1_act[0][0]
block5_sepconv1_bn (BatchNormal	(None, 13, 13, 728)	2912	block5_sepconv1[0][0]
block5_sepconv2_act (Activation	(None, 13, 13, 728)	0	block5_sepconv1_bn[0][0]
block5_sepconv2 (SeparableConv2	(None, 13, 13, 728)	536536	block5_sepconv2_act[0][0]
block5_sepconv2_bn (BatchNormal	(None, 13, 13, 728)	2912	block5_sepconv2[0][0]
block5_sepconv3_act (Activation	(None, 13, 13, 728)	0	block5_sepconv2_bn[0][0]
block5_sepconv3 (SeparableConv2	(None, 13, 13, 728)	536536	block5_sepconv3_act[0][0]
block5_sepconv3_bn (BatchNormal	(None, 13, 13, 728)	2912	block5_sepconv3[0][0]
add_3 (Add)	(None, 13, 13, 728)	0	block5_sepconv3_bn[0][0] add_2[0][0]

block6_sepconv1_act (Activation	(None, 13, 13, 728)	0	add_3[0][0]
block6_sepconv1 (SeparableConv2	(None, 13, 13, 728)	536536	block6_sepconv1_act[0][0]
block6_sepconv1_bn (BatchNormal	(None, 13, 13, 728)	2912	block6_sepconv1[0][0]
block6_sepconv2_act (Activation	(None, 13, 13, 728)	0	block6_sepconv1_bn[0][0]
block6_sepconv2 (SeparableConv2	(None, 13, 13, 728)	536536	block6_sepconv2_act[0][0]
block6_sepconv2_bn (BatchNormal	(None, 13, 13, 728)	2912	block6_sepconv2[0][0]
block6_sepconv3_act (Activation	(None, 13, 13, 728)	0	block6_sepconv2_bn[0][0]
block6_sepconv3 (SeparableConv2	(None, 13, 13, 728)	536536	block6_sepconv3_act[0][0]
block6_sepconv3_bn (BatchNormal	(None, 13, 13, 728)	2912	block6_sepconv3[0][0]
add_4 (Add)	(None, 13, 13, 728)	0	block6_sepconv3_bn[0][0] add_3[0][0]
block7_sepconv1_act (Activation	(None, 13, 13, 728)	0	add_4[0][0]
block7_sepconv1 (SeparableConv2	(None, 13, 13, 728)	536536	block7_sepconv1_act[0][0]
block7_sepconv1_bn (BatchNormal	(None, 13, 13, 728)	2912	block7_sepconv1[0][0]
block7_sepconv2_act (Activation	(None, 13, 13, 728)	0	block7_sepconv1_bn[0][0]
block7_sepconv2 (SeparableConv2	(None, 13, 13, 728)	536536	block7_sepconv2_act[0][0]
block7_sepconv2_bn (BatchNormal	(None, 13, 13, 728)	2912	block7_sepconv2[0][0]
block7_sepconv3_act (Activation	(None, 13, 13, 728)	0	block7_sepconv2_bn[0][0]
block7_sepconv3 (SeparableConv2	(None, 13, 13, 728)	536536	block7_sepconv3_act[0][0]
block7_sepconv3_bn (BatchNormal	(None, 13, 13, 728)	2912	block7_sepconv3[0][0]
add_5 (Add)	(None, 13, 13, 728)	0	block7_sepconv3_bn[0][0] add_4[0][0]
block8_sepconv1_act (Activation	(None, 13, 13, 728)	0	add_5[0][0]
block8_sepconv1 (SeparableConv2	(None, 13, 13, 728)	536536	block8_sepconv1_act[0][0]
block8_sepconv1_bn (BatchNormal	(None, 13, 13, 728)	2912	block8_sepconv1[0][0]
block8_sepconv2_act (Activation	(None, 13, 13, 728)	0	block8_sepconv1_bn[0][0]
block8_sepconv2 (SeparableConv2	(None, 13, 13, 728)	536536	block8_sepconv2_act[0][0]
block8_sepconv2_bn (BatchNormal	(None, 13, 13, 728)	2912	block8_sepconv2[0][0]
block8_sepconv3_act (Activation	(None, 13, 13, 728)	0	block8_sepconv2_bn[0][0]
block8_sepconv3 (SeparableConv2	(None, 13, 13, 728)	536536	block8_sepconv3_act[0][0]
block8_sepconv3_bn (BatchNormal	(None, 13, 13, 728)	2912	block8_sepconv3[0][0]
add_6 (Add)	(None, 13, 13, 728)	0	block8_sepconv3_bn[0][0] add_5[0][0]
block9_sepconv1_act (Activation	(None, 13, 13, 728)	0	add_6[0][0]
block9_sepconv1 (SeparableConv2	(None, 13, 13, 728)	536536	block9_sepconv1_act[0][0]
block9_sepconv1_bn (BatchNormal	(None, 13, 13, 728)	2912	block9_sepconv1[0][0]
block9_sepconv2_act (Activation	(None, 13, 13, 728)	0	block9_sepconv1_bn[0][0]
block9_sepconv2 (SeparableConv2	(None, 13, 13, 728)	536536	block9_sepconv2_act[0][0]
block9_sepconv2_bn (BatchNormal	(None, 13, 13, 728)	2912	block9_sepconv2[0][0]
block9_sepconv3_act (Activation	(None, 13, 13, 728)	0	block9_sepconv2_bn[0][0]
block9_sepconv3 (SeparableConv2	(None, 13, 13, 728)	536536	block9_sepconv3_act[0][0]
block9_sepconv3_bn (BatchNormal	(None, 13, 13, 728)	2912	block9_sepconv3[0][0]
add_7 (Add)	(None, 13, 13, 728)	0	block9_sepconv3_bn[0][0] add_6[0][0]
block10_sepconv1_act (Activatio	(None, 13, 13, 728)	0	add_7[0][0]
block10_sepconv1 (SeparableConv	(None, 13, 13, 728)	536536	block10_sepconv1_act[0][0]
block10_sepconv1_bn (BatchNorma	(None, 13, 13, 728)	2912	block10_sepconv1[0][0]
block10_sepconv2_act (Activatio	(None, 13, 13, 728)	0	block10_sepconv1_bn[0][0]
block10_sepconv2 (SeparableConv	(None, 13, 13, 728)	536536	block10_sepconv2_act[0][0]
block10_sepconv2_bn (BatchNorma	(None, 13, 13, 728)	2912	block10_sepconv2[0][0]
block10_sepconv3_act (Activatio	(None, 13, 13, 728)	0	block10_sepconv2_bn[0][0]
block10_sepconv3 (SeparableConv	(None, 13, 13, 728)	536536	block10_sepconv3_act[0][0]
block10_sepconv3_bn (BatchNorma	(None, 13, 13, 728)	2912	block10_sepconv3[0][0]
add_8 (Add)	(None, 13, 13, 728)	0	block10_sepconv3_bn[0][0] add_7[0][0]

block11_sepconv1_act (Activatio	(None, 13, 13, 728)	0	add_8[0][0]
block11_sepconv1 (SeparableConv	(None, 13, 13, 728)	536536	block11_sepconv1_act[0][0]
block11_sepconv1_bn (BatchNorma	(None, 13, 13, 728)	2912	block11_sepconv1[0][0]
block11_sepconv2_act (Activatio	(None, 13, 13, 728)	0	block11_sepconv1_bn[0][0]
block11_sepconv2 (SeparableConv	(None, 13, 13, 728)	536536	block11_sepconv2_act[0][0]
block11_sepconv2_bn (BatchNorma	(None, 13, 13, 728)	2912	block11_sepconv2[0][0]
block11_sepconv3_act (Activatio	(None, 13, 13, 728)	0	block11_sepconv2_bn[0][0]
block11_sepconv3 (SeparableConv	(None, 13, 13, 728)	536536	block11_sepconv3_act[0][0]
block11_sepconv3_bn (BatchNorma	(None, 13, 13, 728)	2912	block11_sepconv3[0][0]
add_9 (Add)	(None, 13, 13, 728)	0	block11_sepconv3_bn[0][0] add_8[0][0]
block12_sepconv1_act (Activatio	(None, 13, 13, 728)	0	add_9[0][0]
block12_sepconv1 (SeparableConv	(None, 13, 13, 728)	536536	block12_sepconv1_act[0][0]
block12_sepconv1_bn (BatchNorma	(None, 13, 13, 728)	2912	block12_sepconv1[0][0]
block12_sepconv2_act (Activatio	(None, 13, 13, 728)	0	block12_sepconv1_bn[0][0]
block12_sepconv2 (SeparableConv	(None, 13, 13, 728)	536536	block12_sepconv2_act[0][0]
block12_sepconv2_bn (BatchNorma	(None, 13, 13, 728)	2912	block12_sepconv2[0][0]
block12_sepconv3_act (Activatio	(None, 13, 13, 728)	0	block12_sepconv2_bn[0][0]
block12_sepconv3 (SeparableConv	(None, 13, 13, 728)	536536	block12_sepconv3_act[0][0]
block12_sepconv3_bn (BatchNorma	(None, 13, 13, 728)	2912	block12_sepconv3[0][0]
add_10 (Add)	(None, 13, 13, 728)	0	block12_sepconv3_bn[0][0] add_9[0][0]
block13_sepconv1_act (Activatio	(None, 13, 13, 728)	0	add_10[0][0]
block13_sepconv1 (SeparableConv	(None, 13, 13, 728)	536536	block13_sepconv1_act[0][0]
block13_sepconv1_bn (BatchNorma	(None, 13, 13, 728)	2912	block13_sepconv1[0][0]
block13_sepconv2_act (Activatio	(None, 13, 13, 728)	0	block13_sepconv1_bn[0][0]
block13_sepconv2 (SeparableConv	(None, 13, 13, 1024)	752024	block13_sepconv2_act[0][0]
block13_sepconv2_bn (BatchNorma	(None, 13, 13, 1024)	4096	block13_sepconv2[0][0]
conv2d_3 (Conv2D)	(None, 7, 7, 1024)	745472	add_10[0][0]
block13_pool (MaxPooling2D)	(None, 7, 7, 1024)	0	block13_sepconv2_bn[0][0]
batch_normalization_3 (BatchNor	(None, 7, 7, 1024)	4096	conv2d_3[0][0]
add_11 (Add)	(None, 7, 7, 1024)	0	block13_pool[0][0] batch_normalization_3[0][0]
block14_sepconv1 (SeparableConv	(None, 7, 7, 1536)	1582080	add_11[0][0]
block14_sepconv1_bn (BatchNorma	(None, 7, 7, 1536)	6144	block14_sepconv1[0][0]
block14_sepconv1_act (Activatio	(None, 7, 7, 1536)	0	block14_sepconv1_bn[0][0]
block14_sepconv2 (SeparableConv	(None, 7, 7, 2048)	3159552	block14_sepconv1_act[0][0]
block14_sepconv2_bn (BatchNorma	(None, 7, 7, 2048)	8192	block14_sepconv2[0][0]
block14_sepconv2_act (Activatio	(None, 7, 7, 2048)	0	block14_sepconv2_bn[0][0]
flatten (Flatten)	(None, 100352)	0	block14_sepconv2_act[0][0]
dense (Dense)	(None, 1024)	102761472	flatten[0][0]
dropout (Dropout)	(None, 1024)	0	dense[0][0]
dense_1 (Dense)	(None, 512)	524800	dropout[0][0]
dropout_1 (Dropout)	(None, 512)	0	dense_1[0][0]
dense_2 (Dense)	(None, 512)	262656	dropout_1[0][0]
dropout_2 (Dropout)	(None, 512)	0	dense_2[0][0]
dense_3 (Dense)	(None, 2)	1026	dropout_2[0][0]

Total params: 124,411,434
Trainable params: 103,549,954
Non-trainable params: 20,861,480

Figura 37 - Modelo con 2 clases utilizando Transfer Learning.
Fuente: Elaboración propia, basada en la práctica.

4.2. Modelo con 6 clases

Para el set de datos número 2 se incrementó significativamente la cantidad de imágenes y se realiza una división en 6 clases (orgánico, papel/cartón, metal, vidrio, plástico, y no reciclable). La cantidad total de imágenes de este set es de 15105 que se buscó balancear entre las 6 clases disponibles, a su vez, se dividieron los datos en tres partes separando 90 imágenes para el testeo del modelo, 3039 imágenes para la validación y 11976 imágenes utilizadas para el entrenamiento. Antes de ser enviadas al modelo como entradas, se realizó un redimensionamiento por software a cada una de las imágenes para que todas tuvieran un tamaño de 200 x 200 píxeles. El mismo proceso de redimensionamiento se realiza a las nuevas imágenes que el modelo deberá clasificar luego del entrenamiento.

En los siguientes apartados se expondrán las arquitecturas que mejores resultados arrojaron para cada tipo de entrenamiento.

4.2.1. Entrenamiento desde cero

En la siguiente imagen se detalla la arquitectura de la red convolucional modelada, en la que se utilizó un modelo secuencial, además de la capa de entrada se cuenta con 10 capas convolucionales, aplicando un pooling después de la segunda, cuarta, séptima y décima capa y una función de activación “RELU” para cada una de ellas. Luego de estas 10 capas convolucionales se generan 256 mapas de características cada uno de un tamaño de 12 x 12 píxeles. Estos mapas de características son aplanados por la función “flatten” generando un vector de 36864 elementos, que serán utilizados como entradas para la capa completamente conectada. En esta capa se modeló una red neuronal profunda con 3 capas densas cada una con dropout de un 75% de las neuronas en la primera capa y un total de 1024 neuronas y un dropout de 50% en otras dos capas con 512 neuronas cada una. La función de activación para estas capas también es “RELU” y para finalizar el modelo cuenta con una capa de salida con 6 neuronas y activación “SoftMax” para hacer la clasificación final. El optimizador utilizado fue “Adam” y la función de costo a optimizar “categorical cross entropy”.

El modelo inicialmente se programó para realizar 150 épocas de entrenamiento con una tasa de aprendizaje inicial de 0.0005 y su arquitectura final es representada en la Figura 38.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 200, 200, 32)	896
conv2d_1 (Conv2D)	(None, 200, 200, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 100, 100, 32)	0
conv2d_2 (Conv2D)	(None, 100, 100, 64)	18496
conv2d_3 (Conv2D)	(None, 100, 100, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 50, 50, 64)	0
conv2d_4 (Conv2D)	(None, 50, 50, 128)	32896
conv2d_5 (Conv2D)	(None, 50, 50, 128)	65664
conv2d_6 (Conv2D)	(None, 50, 50, 128)	65664
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 128)	0
conv2d_7 (Conv2D)	(None, 25, 25, 256)	131328
conv2d_8 (Conv2D)	(None, 25, 25, 256)	262400
conv2d_9 (Conv2D)	(None, 25, 25, 256)	262400
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 256)	0
flatten (Flatten)	(None, 36864)	0
dense (Dense)	(None, 1024)	37749760
dropout (Dropout 0,75)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
dropout_1 (Dropout 0,5)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262656
dropout_2 (Dropout 0,5)	(None, 512)	0
dense_3 (Dense)	(None, 6)	3078

Total params: 39,426,214
Trainable params: 39,426,214
Non-trainable params: 0

Figura 38 - Modelo con 6 clases entrenamiento desde cero.
Fuente: Elaboración propia, basada en la práctica.

4.2.2. Entrenamiento con transfer learning

Para este caso la red convolucional modelada se basa en un modelo pre entrenado de una arquitectura llamada VGG19, el cual se presentó en el apartado 2.6.2. Para este modelo se transfiere el aprendizaje de la etapa de extracción de características provisto por el modelo VGG19 y se modeló una nueva capa completamente conectada. A la etapa de extracción de características solo se le

modificó el tamaño esperado de las imágenes en la capa de entrada pasando de 224 x 224 píxeles, tamaño original del modelo, a 200 x 200 píxeles para que concuerde con el tamaño de imágenes utilizado en el modelo entrenado desde cero.

La última capa convolucional del modelo VGG19 deja como salida 512 mapas de características con imágenes de 6 x 6 píxeles cada uno. Como siguiente paso estos mapas de características son pasados por una función de aplanamiento que genera un vector de 18432 elementos que serán utilizados como entradas para la capa completamente conectada. En esta capa se modeló una red neuronal profunda con 3 capas densas cada una con dropout de un 50%, la primera capa con 2048 neuronas, la segunda con un total de 1024 neuronas y la tercera capa con 512 neuronas. La función de activación utilizadas en estas capas es “RELU” y para finalizar el modelo cuenta con una capa de salida con 6 neuronas y activación “SoftMax” para realizar la clasificación final. El optimizador utilizado fue “Adam” y la función de costo a optimizar “categorical cross entropy”.

Al final de la tabla se muestra la cantidad de parámetros del modelo y corresponde a un total de 60.401.222 parámetros, pero gracias a una de las ventajas que brinda la transferencia de aprendizaje solo se entrenaron los 40.376.838 parámetros correspondiente a la capa completamente conectada del modelo, para los parámetros restantes no se ajustaron los pesos durante las iteraciones de la etapa de entrenamiento (ver Figura 39).

El modelo inicialmente se programó para realizar 150 épocas de entrenamiento con una tasa de aprendizaje inicial de 0.0005.

Model: "model-VGG19"

Layer (type)	Output	Shape Param #
input_1 (InputLayer)	[(None, 200, 200, 3)]	0
block1_conv1 (Conv2D)	(None, 200, 200, 64)	1792
block1_conv2 (Conv2D)	(None, 200, 200, 64)	36928
block1_pool (MaxPooling2D)	(None, 100, 100, 64)	0
block2_conv1 (Conv2D)	(None, 100, 100, 128)	73856
block2_conv2 (Conv2D)	(None, 100, 100, 128)	147584
block2_pool (MaxPooling2D)	(None, 50, 50, 128)	0
block3_conv1 (Conv2D)	(None, 50, 50, 256)	295168
block3_conv2 (Conv2D)	(None, 50, 50, 256)	590080
block3_conv3 (Conv2D)	(None, 50, 50, 256)	590080
block3_conv4 (Conv2D)	(None, 50, 50, 256)	590080
block3_pool (MaxPooling2D)	(None, 25, 25, 256)	0
block4_conv1 (Conv2D)	(None, 25, 25, 512)	1180160
block4_conv2 (Conv2D)	(None, 25, 25, 512)	2359808
block4_conv3 (Conv2D)	(None, 25, 25, 512)	2359808
block4_conv4 (Conv2D)	(None, 25, 25, 512)	2359808
block4_pool (MaxPooling2D)	(None, 12, 12, 512)	0
block5_conv1 (Conv2D)	(None, 12, 12, 512)	2359808
block5_conv2 (Conv2D)	(None, 12, 12, 512)	2359808
block5_conv3 (Conv2D)	(None, 12, 12, 512)	2359808
block5_conv4 (Conv2D)	(None, 12, 12, 512)	2359808
block5_pool (MaxPooling2D)	(None, 6, 6, 512)	0
flatten (Flatten)	(None, 18432)	0
dense (Dense)	(None, 2048)	37750784
dropout (Dropout)	(None, 2048)	0
dense_1 (Dense)	(None, 1024)	2098176
dropout_1 (Dropout)	(None, 1024)	0
dense_2 (Dense)	(None, 512)	524800
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 6)	3078

Total params: 60,401,222
Trainable params: 40,376,838
Non-trainable params: 20,024,384

Figura 39 - Modelo con 6 clases utilizando Transfer Learning.
Fuente: Elaboración propia, basada en la práctica.

5. Análisis de resultados

Durante todo el proyecto se modelaron aproximadamente 80 redes neuronales convolucionales, de los cuales los primeros 60 modelos fueron entrenados desde el inicio y los restantes 20 modelos fueron entrenados utilizando transfer learning basados en varias redes pre entrenadas que se detallaron del inciso 2.6.1 al 2.6.6.

Para el entrenamiento de los distintos modelos se necesitaron alrededor de dos meses de horas máquina, ya que los modelos más simples llevaron un tiempo promedio de entrenamiento de 16 horas y los modelos más complejos tuvieron un tiempo promedio de entrenamiento de 60 horas, llegando a un promedio general de 30 horas de entrenamiento por modelo propuesto.

Uno de los principales problemas que se detectó durante el desarrollo de los primeros modelos fue la presencia de overfitting, logrando buenos resultados para el set de entrenamiento y malos para el set de validación. Para los primeros modelos los resultados rondaban el 60% de acierto para la predicción de nuevas imágenes, por lo tanto, para lograr mejorar estos resultados se fueron probando pequeñas variaciones en los hiperparámetros de la red y la profundidad de los modelos. Además, se implementaron algunas de las técnicas mencionadas en el inciso 2.7.6 para lograr reducir el sobreajuste del modelo, siendo de mucha utilidad las funciones de “callbacks” que permite utilizar la librería de alto nivel Keras.

Las funciones “callbacks” son funciones que se ejecutan luego de cada iteración de entrenamiento, permitiendo realizar modificaciones al LR (learning rate) o guardar los pesos del modelo en ese momento, por ejemplo, si se cumple alguna condición predefinida. Esto se consigue debido a que estas funciones tienen acceso a todas las variables involucradas durante el entrenamiento como, por ejemplo, el porcentaje de acierto, ya sea para el set de entrenamiento como para el set de validación. Las funciones elegidas fueron tres, en primer lugar, se utilizó la función “EarlyStopping” a la que se le indica la cantidad de épocas (iteraciones) que se debe esperar para terminar el entrenamiento siempre y cuando se cumpla una condición aplicada a una variable que se va monitoreando; en este caso se monitorea el valor de la función de pérdida de los datos de validación y la función corta el entrenamiento si este dato no disminuye luego de las iteraciones indicadas. En segundo lugar, se utilizó la función “ModelCheckpoint”, que permite guardar los valores de los pesos del modelo obtenido en cada iteración y una vez terminado el entrenamiento restaurar los mejores pesos, en donde el modelo se desempeña de mejor manera. Por último, la función utilizada fue “ReduceLROnPlateau”, que

permite reducir el hiperparámetro LR en un factor que se define cuando se llama a la función y debe ser mayor a 0 y menor a 1, permitiendo una mayor reducción cuando el número es más cercano al nivel inferior permitido. Además, se debe indicar cuántas iteraciones se debe esperar para aplicar este cambio, permitiendo también el monitoreo de una variable para aplicarlo; también permite descartar todos los cambios realizados luego de cierta cantidad de iteraciones, esto se realiza con el parámetro "cooldown" de la función. Para el caso de los modelos planteados la función monitoreaba el valor del porcentaje de acierto del set de validación y los mejores resultados se obtuvieron cuando el factor por el que se multiplicaba el LR era "0.9" y el porcentaje de épocas a esperar para realizar el cambio era de alrededor de un 15% a un 20% de la cantidad total de épocas definidas para el entrenamiento.

Por otra parte, también se aplicó la técnica de dropout a la capa completamente conectada del modelo, variando el porcentaje de neuronas al que se le aplicaba dependiendo de la cantidad de neuronas que contenía cada capa, (este porcentaje se fue variando entre un 25% y un 75% en los distintos modelos). Todas estas modificaciones permitieron una mejora considerable en el porcentaje de acierto de los modelos propuestos llegando a valores superiores al 70% para el set de datos que estaba dividido en seis clases y valores superiores al 84% para el set de datos dividido en dos clases. Una vez obtenidos estos resultados se comenzaron a aplicar los modelos pre entrenados con los cuales se logró obtener resultados mayores al 85% de acierto para el set de dos clases y valores cercanos al 80% para el set de seis clases. En el caso de los modelos que utilizaron transfer learning también se aplicaron las funciones "callbacks".

A continuación, se hace un análisis de los resultados obtenidos durante el entrenamiento para cada uno de los modelos presentados en los apartados anteriores y también se realiza el análisis de los resultados que se obtuvieron con los datos reservados para el testeo de los modelos desarrollados.

5.1 Análisis de gráficos de pérdida y acierto para los 4 modelos propuestos

A continuación, se presentan tres figuras para cada uno de los modelos, la primera contiene los valores de la función de pérdida tanto para el set de entrenamiento como para el set de validación y el valor del acierto también para ambos sets de datos, todo en función de las épocas de entrenamiento del modelo. La segunda figura muestra una comparación entre los valores de la función de

pérdida y acierto del set de entrenamiento. Por último, la tercera figura detalla una comparación entre las mismas funciones, pero para el set de validación.

El primer modelo que se analizará es el modelo presentado en el apartado 4.1.1.

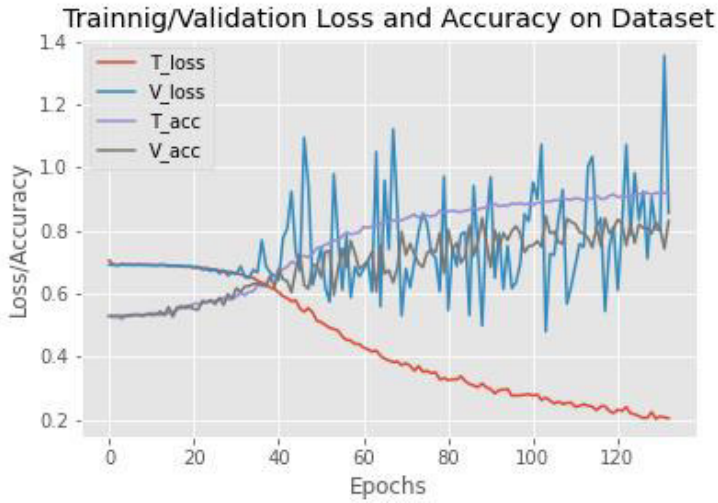


Figura 40 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.1.1.
Fuente: Elaboración propia, basada en la práctica.

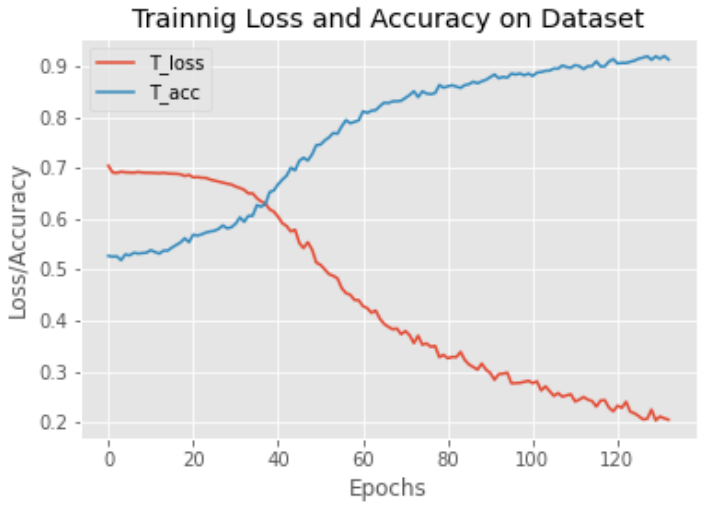


Figura 41 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.1.1.
Fuente: Elaboración propia, basada en la práctica.

Para el caso del set de entrenamiento, tanto el valor de la función de acierto como el de la función de pérdida, presentan valores dentro de lo esperado para el modelo, ya que a medida que el acierto crece la función de pérdida disminuye.

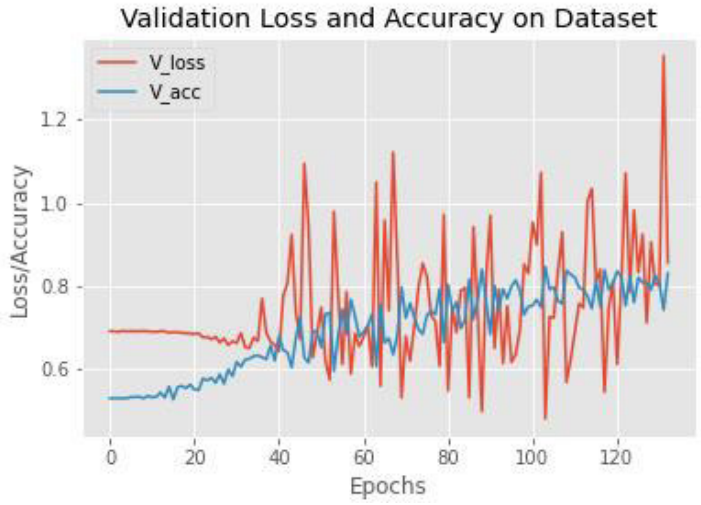


Figura 42 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.1.1.
Fuente: Elaboración propia, basada en la práctica.

Como se puede apreciar en la Figura 42, si bien el valor del acierto se fue incrementando a lo largo de las iteraciones, este presenta cierta fluctuación. Pero, en el caso de la función de pérdida, la dispersión de los valores obtenidos es aún mayor, lo que indica que a este modelo aún se le pueden aplicar ciertas modificaciones para mejorar su performance.

El segundo modelo es el presentado en el inciso 4.1.2

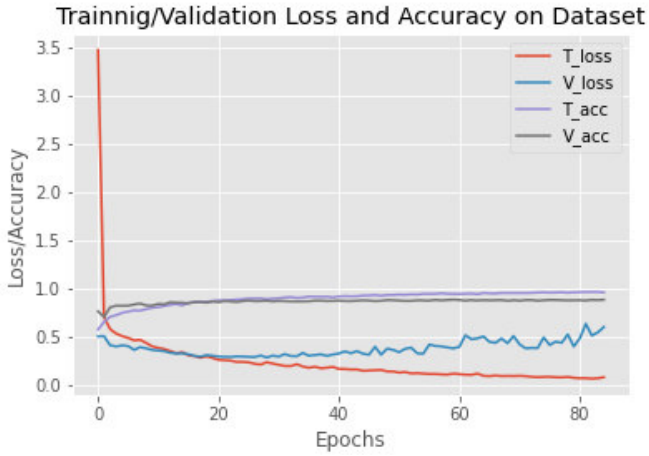


Figura 43 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.1.2.
Fuente: Elaboración propia, basada en la práctica.

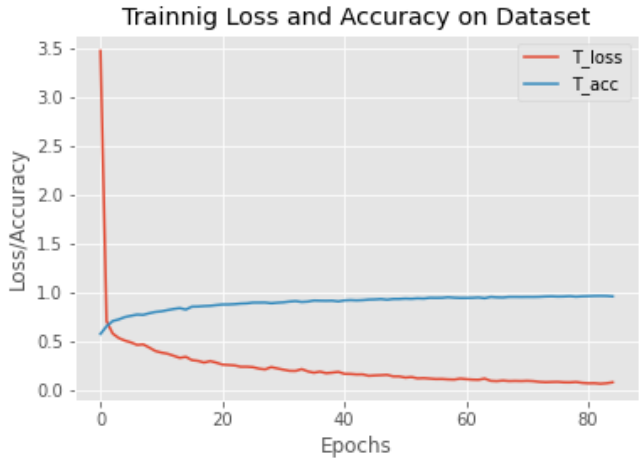


Figura 44 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.1.2.
Fuente: Elaboración propia, basada en la práctica.

Para este segundo modelo al tratarse de la reutilización de una red pre entrenada, las funciones de acierto y pérdida crecen y decrecen con mayor estabilidad respectivamente. Además, se nota que en muy pocas iteraciones el modelo ya presenta una buena tasa de acierto para el set de entrenamiento.

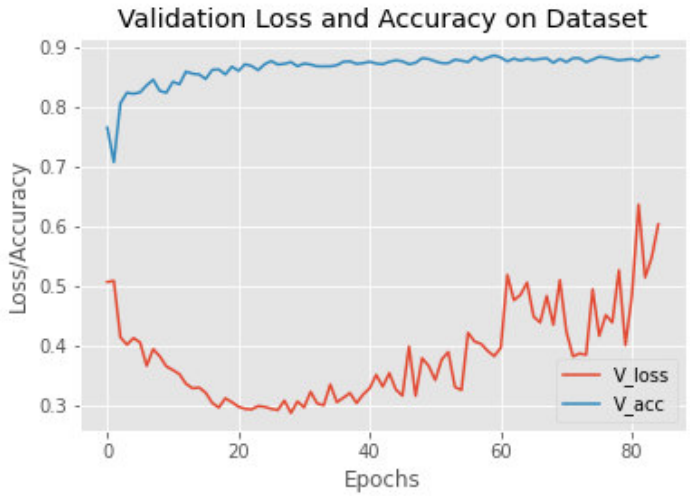


Figura 45 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.1.2.
Fuente: Elaboración propia, basada en la práctica.

En el caso del conjunto de datos de validación, los resultados obtenidos concuerdan con los del set de entrenamiento, con la única diferencia que a partir de la iteración 30 aproximadamente la función de pérdida dejó de disminuir y comenzó a aumentar. Este inconveniente fue resuelto gracias a la utilización de la función de parada temprana (early stopping), pudiendo regresar el modelo a valores en donde se mostraba una mejor generalización.

El tercer modelo corresponde a lo propuesto en el apartado 4.2.1.

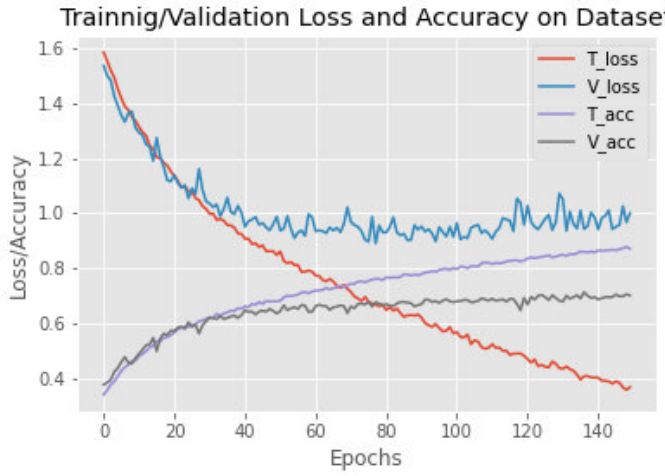


Figura 46 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.2.1.
Fuente: Elaboración propia, basada en la práctica.

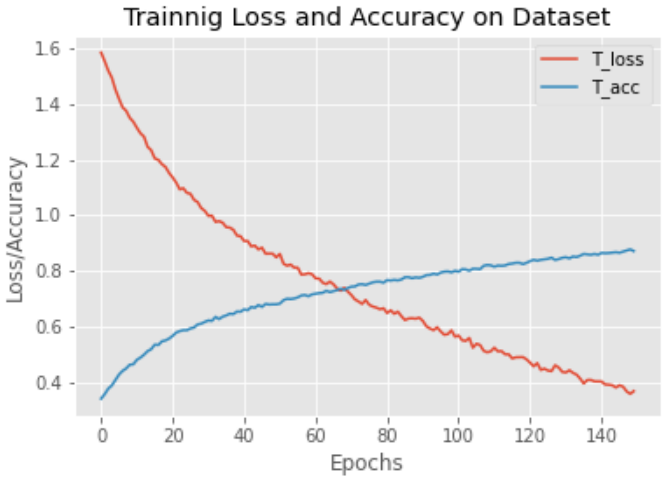


Figura 47 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.2.1.
Fuente: Elaboración propia, basada en la práctica.

En la figura anterior (Figura 47) se puede apreciar el buen comportamiento, tanto para la función de pérdida como para la tasa de aciertos. Estos son resultados comparables con los obtenidos con los modelos entrenados con aprendizaje por transferencia, pero con la necesidad de más iteraciones para iguales resultados.

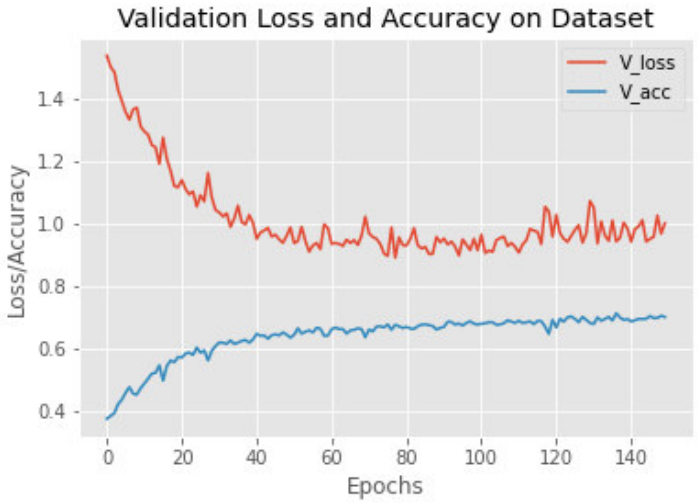


Figura 48 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.2.1.
Fuente: Elaboración propia, basada en la práctica.

Para el set de validación los valores de las funciones demuestran un buen desempeño del modelo durante las primeras 100 iteraciones.

Por último, se analiza el modelo propuesto en el inciso 4.2.2.

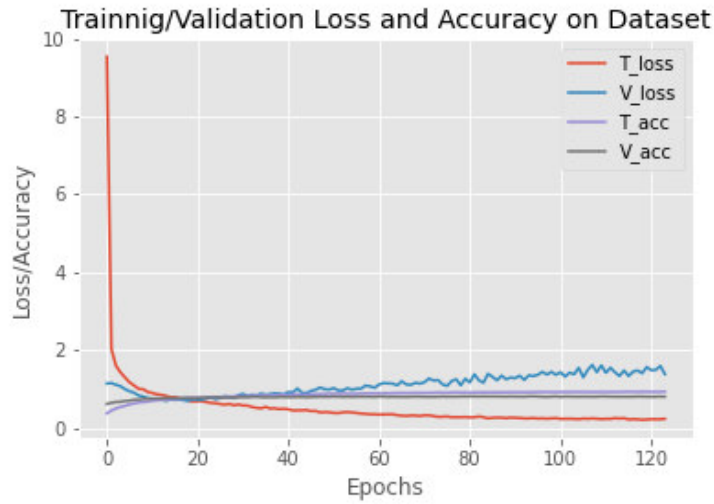


Figura 49 - T_loss, V_loss, T_acc, V_acc en función de las épocas de entrenamiento modelo 4.2.2.
Fuente: Elaboración propia, basada en la práctica.

Figura 50 - T_loss, T_acc en función de las épocas de entrenamiento modelo 4.2.2.
Fuente: Elaboración propia, basada en la práctica.

Al igual que en el caso del dataset de dos clases y modelo pre entrenado, este modelo presenta valores esperados para el conjunto de entrenamiento.

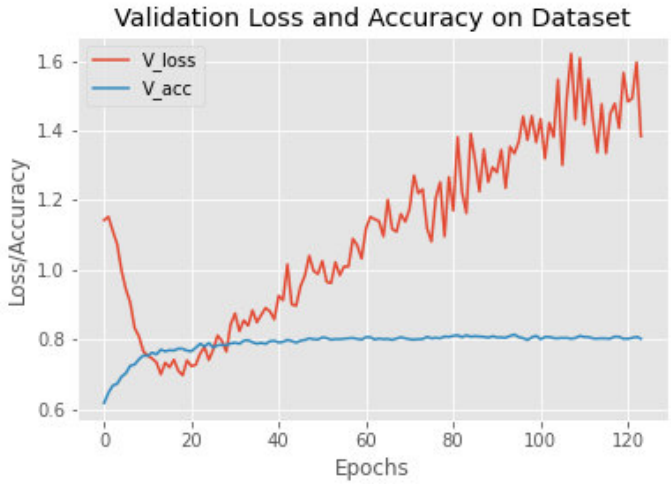


Figura 51 - V_loss, V_acc en función de las épocas de entrenamiento modelo 4.2.2.
Fuente: Elaboración propia, basada en la práctica.

Curiosamente para el caso de los valores de las funciones de pérdida y tasa de acierto del set de validación, se logró un buen resultado en las primeras 25 iteraciones y posteriormente la función de pérdida incrementó sostenidamente sus valores. En este caso también se aprovechó el uso de las funciones “callbacks”.

5.2. Análisis de Matriz de confusión para los 4 modelos propuestos

En este apartado se presentan los análisis de las matrices de confusión de cada uno de los modelos ya vistos. Se calculó en cada caso el recall "R", la precisión "P", el accuracy o acierto "A" y el F1-score "F1" con las fórmulas presentadas en el inciso 2.8 para el cálculo de cada valor. La primera matriz corresponde al modelo del inciso 4.1.1.

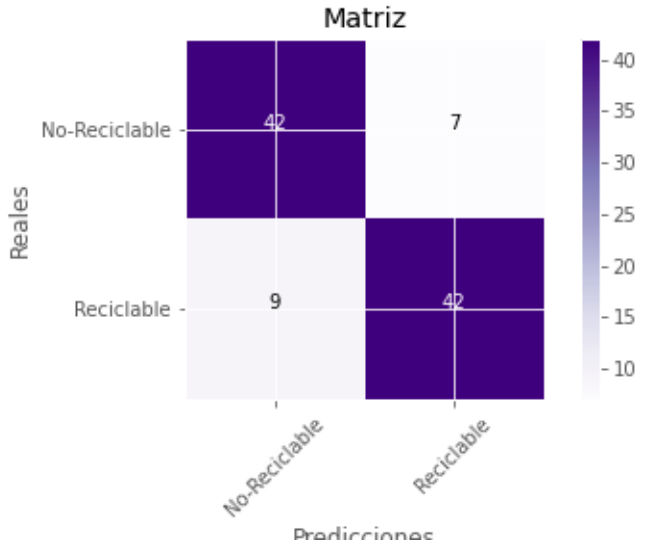


Figura 52 - Matriz de confusión modelo 4.1.1.
Fuente: Elaboración propia, basada en la práctica.

$$R = \frac{VP}{VP+FN} = \frac{42}{42+7} = \frac{42}{49} = 0{,}85 \qquad P = \frac{VP}{VP+FP} = \frac{42}{42+9} = \frac{42}{51} = 0{,}82$$

$$A = \frac{VP + VN}{VP + FP + FN + VN} = \frac{42 + 42}{42 + 9 + 7 + 42} = \frac{84}{100} = 0{,}84$$

$$F1 = \frac{2\ x\ P\ x\ R}{P + R} = \frac{2\ x\ 0{,}82\ x\ 0{,}85}{0{,}82 + 0{,}85} = \frac{1{,}394}{1{,}67} = 0{,}83$$

Para este modelo los resultados obtenidos son bastante similares, tanto para el acierto como para la métrica F1 y esto se debe principalmente a que solo se dividen los datos en dos clases; además, estas clases contienen una cantidad muy bien balanceada de datos cada una. En este caso se obtuvo un 84% de acierto y un 83% para F1.

La siguiente matriz pertenece a los resultados del modelo presentado en el apartado 4.1.2.

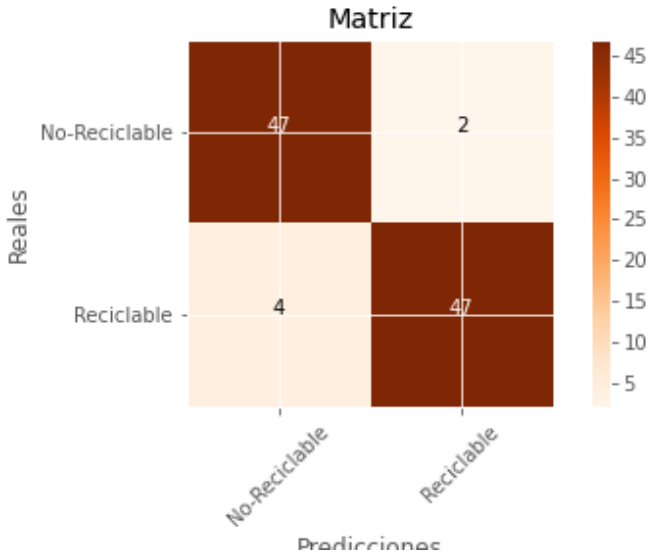


Figura 53 - Matriz de confusión modelo 4.1.2.
Fuente: Elaboración propia, basada en la práctica.

$$R = \frac{VP}{VP+FN} = \frac{47}{47+2} = \frac{47}{49} = 0{,}95 \qquad P = \frac{VP}{VP+FP} = \frac{47}{47+4} = \frac{47}{51} = 0{,}92$$

$$A = \frac{VP + VN}{VP + FP + FN + VN} = \frac{47 + 47}{47 + 4 + 2 + 47} = \frac{94}{100} = 0{,}94$$

$$F1 = \frac{2\ x\ P\ x\ R}{P + R} = \frac{2\ x\ 0{,}92\ x\ 0{,}95}{0{,}92 + 0{,}95} = \frac{1{,}748}{1{,}87} = 0{,}93$$

Para el segundo modelo los resultados son consistentes con lo mencionado en el modelo anterior, con la diferencia que al tratarse de un modelo basado en transfer learning los porcentajes de acierto y F1 fueron mayores llegando a un 94% para el acierto y un 93% para la métrica F1.

A continuación, se presenta la matriz de confusión para el modelo de seis clases visto en el apartado 4.2.1.

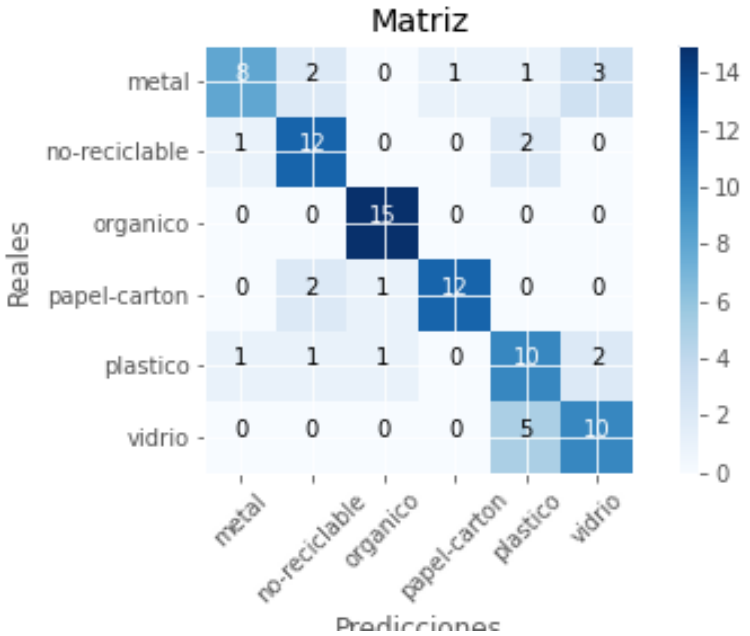


Figura 54 - Matriz de confusión modelo 4.2.1.
Fuente: Elaboración propia, basada en la práctica.

$$A = \frac{VP + VN}{VP + FP + FN + VN} = \frac{67}{90} = 0{,}74$$

F1 – Score calculado para la clase “vidrio”

$$R = \frac{VP}{VP+FN} = \frac{10}{10+5} = \frac{10}{15} = 0{,}66 \qquad P = \frac{VP}{VP+FP} = \frac{10}{10+5} = \frac{10}{15} = 0{,}66$$

$$F1 = \frac{2\ x\ P\ x\ R}{P + R} = \frac{2\ x\ 0{,}66\ x\ 0{,}66}{0{,}66 + 0{,}66} = \frac{0{,}88}{1{,}32} = 0{,}66$$

Este modelo presenta una clara diferencia entre el resultado del acierto y el resultado de la métrica F1 (en este informe solo se muestra el cálculo de F1 para la clase “vidrio”), siendo para algunas clases mayor al acierto y para otras menor, debido al desbalanceo que presentan las clases. Por otro lado, al analizar la matriz se puede denotar en los falsos negativos que el modelo suele confundir, ya que presenta la mayoría de las veces una relación estrecha entre ciertas clases. Un ejemplo de este caso se ve claramente en la clase “vidrio”, en donde todos los falsos negativos son tomados por el modelo como imágenes pertenecientes a la clase “plástico”. La tasa de acierto lograda por el modelo es del 74% sobre el set de datos de testeo.

Por último, la siguiente figura (Figura 55) es la correspondiente a los resultados obtenidos del modelo visto en el inciso 4.2.2.

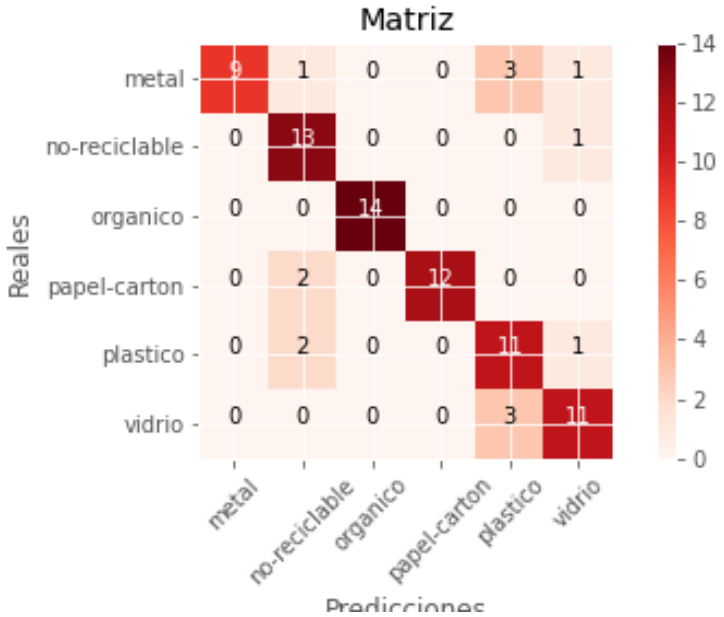


Figura 55 - Matriz de confusión modelo 4.2.2.
Fuente: Elaboración propia, basada en la práctica.

$$A = \frac{VP + VN}{VP + FP + FN + VN} = \frac{70}{84} = 0{,}83$$

F1 – Score calculado para la clase “metal”

$$R = \frac{VP}{VP+FN} = \frac{9}{9+5} = \frac{9}{14} = 0{,}64 \qquad P = \frac{VP}{VP+FP} = \frac{9}{9+0} = \frac{9}{9} = 1$$

$$F1 = \frac{2\ x\ P\ x\ R}{P + R} = \frac{2\ x\ 1\ x\ 0{,}64}{1 + 0{,}64} = \frac{1{,}28}{1{,}64} = 0{,}78$$

F1 – Score calculado para la clase “organico”

$$R = \frac{VP}{VP+FN} = \frac{14}{14+0} = \frac{14}{14} = 1 \quad P = \frac{VP}{VP+FP} = \frac{14}{14+0} = \frac{14}{14} = 1$$

$$F1 = \frac{2\ x\ P\ x\ R}{P + R} = \frac{2\ x\ 1\ x\ 1}{1 + 1} = \frac{2}{2} = 1$$

Nuevamente para el caso en donde se utiliza aprendizaje por transferencia el porcentaje de acierto aumentó respecto al modelo con entrenamiento desde cero, obteniendo un 83% de acierto sobre el set de datos de testeo. Para este caso se muestra el cálculo de la métrica F1, tanto para la clase “metal” como para la clase “orgánico”, verificando en los resultados que existe un desbalanceo grande entre estas dos clases, ya que ambos valores presentan una diferencia grande con la tasa

de acierto. Para el resto de las clases la diferencia entre las métricas es menor y se encuentran dentro de los valores esperados. El resultado visto en estas dos clases es consistente con lo esperado, debido a la formación del set de datos, en donde la clase “metal” es la que menos ejemplos tiene y la clase “orgánico” así como la clase “no-reciclable” son las clases que cuentan con la mayor cantidad de ejemplos para el entrenamiento y la validación.

5.3. Aplicación de predicción primera versión

A continuación, se presenta el código de la primera versión de la aplicación para realizar la clasificación en tiempo real de imágenes.

```
#se importan las librerías a utilizar
import tensorflow as tf
import numpy as np
from keras.preprocessing.image import load_img, img_to_array
from keras.models import load_model
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

#se carga el modelo guardado ya entrenado
altura, ancho= 200,200
models='C:/temporal pps/modelo-8.4.1.vgg19/modelo.h5'
weights='C:/temporal pps/modelo-8.4.1.vgg19/pesos.h5'
cnn=tf.keras.models.load_model(models)
cnn.load_weights(weights)

#funcion que realiza la prediccion
def predict(file):

    image=load_img(file, target_size=(altura, ancho))
    image=img_to_array(image)
    image=np.expand_dims(image, axis=0)
    prediction=cnn.predict(image)
    result=prediction[0]
    reply=np.argmax(result)
    if reply==0:
        clase='0 - METAL'
    elif reply==1:
        clase='1 - NO-RECICLABLE'
```

```
elif reply==2:
    clase='2 - ORGANICO'
elif reply==3:
    clase='3 - PAPEL-CARTON'
elif reply==4:
    clase='4 - PLASTICO'
elif reply==5:
    clase='5 - VIDRIO'
image=mpimg.imread(file)
plt.imshow(image, interpolation=None)
plt.title('Imagen analizada - Categoría predicha: ' + clase)
plt.axis('off')

return
```

La función definida realiza la predicción sobre la imagen que recibe como parámetro e imprime el resultado de la predicción junto con la imagen analizada. A continuación, se presentan tres ejemplos de predicciones tomadas con un celular y pasadas a la función, además se presenta un ejemplo de un falso positivo en donde se fotografió un envase de plástico y fue tomado por la red como uno de vidrio, aunque al mirar la imagen no es tan sencillo para el ojo humano hacer una buena predicción sobre la misma.

```
predict('C:/temporal pps/real-test/imagen1.jpg')
```

Imagen analizada - Categoría predicha: 4 - PLASTICO

```
predict('C:/temporal pps/real-test/imagen2.jpg')
```

Imagen analizada - Categoría predicha: 0 - METAL

```
predict('C:/temporal pps/real-test/imagen3.jpg')
```

Imagen analizada - Categoría predicha: 1 - NO-RECICLABLE

La siguiente predicción presenta el caso del falso positivo.

```
predict('C:/temporal pps/real-test/imagen4.jpg')
```

Imagen analizada - Categoría predicha: 5 - VIDRIO

6. Conclusiones

Durante el desarrollo e implementación de este trabajo se estudiaron algunas técnicas de Machine Learning aplicadas a la clasificación de imágenes y particularmente se buscó colaborar con el cuidado del medio ambiente utilizando la inteligencia artificial aplicada a la clasificación de objetos reciclables. Además, se realizó un recorrido por la teoría que fundamenta las redes neuronales y se han descrito los pasos necesarios para el diseño y desarrollo de una red neuronal que permita realizar la clasificación deseada. Posteriormente, se plantearon los resultados obtenidos y en esta sección se presentan las conclusiones, posibles mejoras y líneas de investigación futuras.

También, han ido surgiendo diferentes problemas, debido a las razones de público conocimiento (pandemia COVID-19) se tuvo que replantear uno de los objetivos propuestos al inicio, como ser la realización de pruebas en un ambiente real. Se utilizó el tiempo destinado para ese objetivo a la mejora de los restantes y principalmente se buscó optimizar el acierto de los modelos de redes neuronales propuestos. De por sí la implementación del sistema se encuentra atada a la complejidad de los altos requerimientos de una red neuronal convolucional pero, de todas formas, se logró implementar una red funcional que supera los objetivos planteados durante las primeras etapas del trabajo.

Se ha elegido Deep Learning para llevar a cabo este trabajo debido a la alta capacidad que presenta para el análisis de imágenes. Sin embargo, al programar algoritmos de DL, específicamente redes neuronales convolucionales, se presentaron ciertas dificultades que se irán especificando en estas conclusiones. Por la gran cantidad de usos que hoy en día se da a las redes neuronales, resulta de mucho interés comprender sus ventajas, limitaciones y vulnerabilidades, con el fin de poder garantizar la robustez que los sistemas actuales necesitan.

Es muy importante y complejo determinar los parámetros e hiperparámetros que mejor se adecuen al modelo propuesto impactando directamente en el resultado final. Este ajuste “fine tuning” muchas veces se considera un arte y no una ciencia para los que se inician en el campo del DL, debido a que se requiere cierta experiencia e intuición para encontrar los valores óptimos de estos hiperparámetros. En este caso, la investigación ha jugado un papel fundamental, además los parámetros e hiperparámetros se deben especificar antes de iniciar el proceso de entrenamiento. Otro aspecto que se debe tener en cuenta es la profundidad y la cantidad de capas de los modelos propuestos.

Por otro lado, la correcta elección y confección del set de datos que se utilizará para entrenar los modelos es vital para llegar a los resultados deseados. La

red debe conocer de igual manera todos los elementos o clases que se quieren predecir o clasificar, por lo que, las distintas clases deben estar correctamente balanceadas, contener ejemplos representativos y, además, contar con la mayor cantidad posible de datos para lograr buenas soluciones, aunque esto se traduce a un mayor tiempo de entrenamiento. Todo esto ayuda a reducir los problemas de subajuste y sobreajuste que presentan habitualmente los modelos de redes neuronales. Durante el trabajo se demostró que las redes neuronales convolucionales presentan excelentes resultados en el campo de la "computer vision", aunque uno de los inconvenientes encontrados ha sido el tiempo, ya que, a medida que los modelos se hacen más complejos crecen los tiempos de entrenamiento, llegando en el caso de este trabajo a un promedio de 36 horas para el entrenamiento de cada uno de los modelos propuestos.

Se debe tener en cuenta también que trabajar con CNN y grandes conjuntos de datos implica un alto costo computacional, por lo tanto, es muy importante contar con equipos que presenten altas prestaciones e incluso contar con GPUs (unidades de procesamiento gráfico). Asimismo, es recomendable el re escalado de las imágenes del set de datos para disminuir los tiempos de entrenamiento.

Otro punto a destacar es la elección del Framework Tensorflow y la API de alto nivel Keras, así como también el lenguaje Python para trabajar con DL, puesto que, cuentan con una multitud de funciones y librerías que facilitan el trabajo en muchas ocasiones.

En referencia a los resultados obtenidos para el conjunto de datos de prueba se puede decir que fueron satisfactorios, si bien no se pudieron realizar pruebas en ambientes reales, estos superaron las expectativas propuestas al inicio del trabajo. Además, todas las pruebas realizadas sirven como aprendizaje para futuros proyectos. Se considera que la Inteligencia Artificial, el Machine Learning y el Deep Learning serán la base para los proyectos venideros, independientemente del campo en el que se esté trabajando. Destacándose puntualmente en el campo de la medicina y robótica, entre otros.

6.1. Líneas futuras

A continuación, se proponen algunas líneas de trabajo futuras con el fin de continuar y mejorar este proyecto.

En primer lugar, se puede mejorar y aumentar el set de datos permitiendo el re entrenamiento de los modelos planteados con estas mejoras en los datos, lo que presentará una mayor robustez y mejor balanceo en las clases a clasificar. Además, se puede optar por incluir en los nuevos dataset imágenes que contengan ejemplos

adversarios logrando en futuros entrenamientos una mayor defensa ante este tipo de ataques.

En segundo lugar, y luego de haber mejorado el conjunto de datos, se debería proceder a implementar nuevos modelos con distintas variaciones en los parámetros e hiperparámetros buscando una mejora en los resultados para las distintas predicciones.

Por último, se podrían realizar pruebas en un ambiente real verificando así la verdadera efectividad de los modelos propuestos, esta es una de las tareas que ha quedado pendiente. También se podría implementar esta aplicación en algún sistema robótico como, por ejemplo, un cesto inteligente en donde se pueda colocar un objeto al cual se le tomará una foto que será la entrada de la red neuronal y se le realizaría la posterior clasificación para determinar el lugar donde se almacenará, según su tipo.

Reflexión sobre la práctica Profesional Supervisada como espacio de formación:

La realización de este trabajo sirvió para introducirme al mundo de las redes neuronales y Machine Learning, lo que es muy importante a nivel profesional. Si bien, durante el transcurso de la formación académica no se hace demasiado hincapié en este tema, los conocimientos adquiridos en las cursadas de las distintas materias de la carrera aportaron significativamente a la curva de aprendizaje transitada durante el proyecto.

Seleccioné este proyecto entre otras opciones que me presentaron debido a la temática que trataba, pero sin haber tenido experiencia previa sobre el desarrollo practico de estas técnicas. Considero que en la actualidad la Inteligencia Artificial y el Deep Learning son herramientas que todo ingeniero informático o futuro ingeniero debe conocer o comenzar a conocer, ya que día a día cada vez más aplicaciones hacen uso de ellas.

A nivel personal consideré esta instancia formativa como una excelente oportunidad para incorporar estos nuevos conocimientos e introducirme en el mundo de la investigación académica. Esta investigación me llevó a descubrir numerosas aplicaciones y proyectos relacionados con el tema, los cuales me han llamado la atención y motivado a continuar con esta línea de desarrollo.

7. Bibliografía

- Andreas Muller, Sarah Guido, Introduction to Machine Learning with Python, Editorial O'reilly, 2016.
- Andrew W.Trask, Grokking Deep Learning, MEAP edition, Manning Publications 2017.
- Aprende Machine Learning (2020), https://www.aprendemachinelearning.com/, recuperado 25 de abril 2020.
- Aurelien Gerón, Hands-On Machine Learning withs cikitLearn & TensorFlow, Editorial O'reilly, 2017
- C. Jay Kuo, Understanding Convolutional Neural Networks with A Mathematical Model, Department of Electrical Engineering University of Southern California 2016.
- Charu C. Aggarwal, Neural Network and Deep Learning, Springer International Publishing AG part of Springer Nature 2018.
- Diario El Cronista (2018), Producción de basura: cuál es la realidad en Argentina y que se podría hacer, https://www.cronista.com/responsabilidad/Produccion-de-basura-cual-es-la-realidad-en-Argentina-y-que-se-podria-hacer-20180302-0075.html, recuperado 10 de mayo 2020.
- Dominik Scherer, Andreas Muller, Sven Behnke, Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition, University of Bonn, Institute of Computer Science VI ,2010.
- Francois Chollet, Deep Learning with Python, MEAP edition, Manning Publications 2017.
- Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, MIT 2017.
- Keras io (2020), Keras API references, https://keras.io/api/, recuperado 01 octubre de 2020.
- Kevin Murphy, Machine Learning - A probabilistic perspective, University of Cambridge, 2012
- Maquituls España (2017), La importancia del reciclaje. Cuidemos el Medio Ambiente, https://www.maquituls.es/noticias/la-importancia-del-reciclaje-cuidemos-el-medio-ambiente/#:~:text=El%20reciclar%20o%20el%20reciclaje,de%20manera%20continua%20al%20planeta., recuperado 10 de mayo 2020.
- Miroslav Kubat, An Introduction to Machine Learning, second edition, Springer International Publishing AG 2017.
- Nikhil Buduma, Fundamentals of Deep Learning, Editorial O'reilly. 2017
- Prateek Joshi, John Hearty, Bastiaan Sjardin, Luca Massaron, Alberto Boschetti, Python: Real World Machine Learning, Editorial Packt Publishing Ltd. 2016.
- Sandro Skansi, Introduction to Deep Learning – From Logical Calculus to Artificial Intelligence, Springer International Publishing AG part of Springer Nature 2018.
- TensoFlow (2020), TensorFlow API documentation, https://www.tensorflow.org/api_docs/, recuperado 01 octubre de 2020.
- Tom Hope, Yehezkel Resheff, Itay Lieder, Learning TensorFlow, Editorial O'reilly. 2017

- Transfer Learning Wikipedia (2020), Transfer Learning, https://en.wikipedia.org/wiki/Transfer_learning, recuperado 01 julio de 2020.
- Wolfgang Ertel, Introduction to Artificial Intelligence, second edition, Springer International Publishing AG 2017.
- Zoubin Gharamani, Automatic Machine Learning, University of Cambridge, 2018

Anexo A – Tutorial de instalación de entorno de trabajo

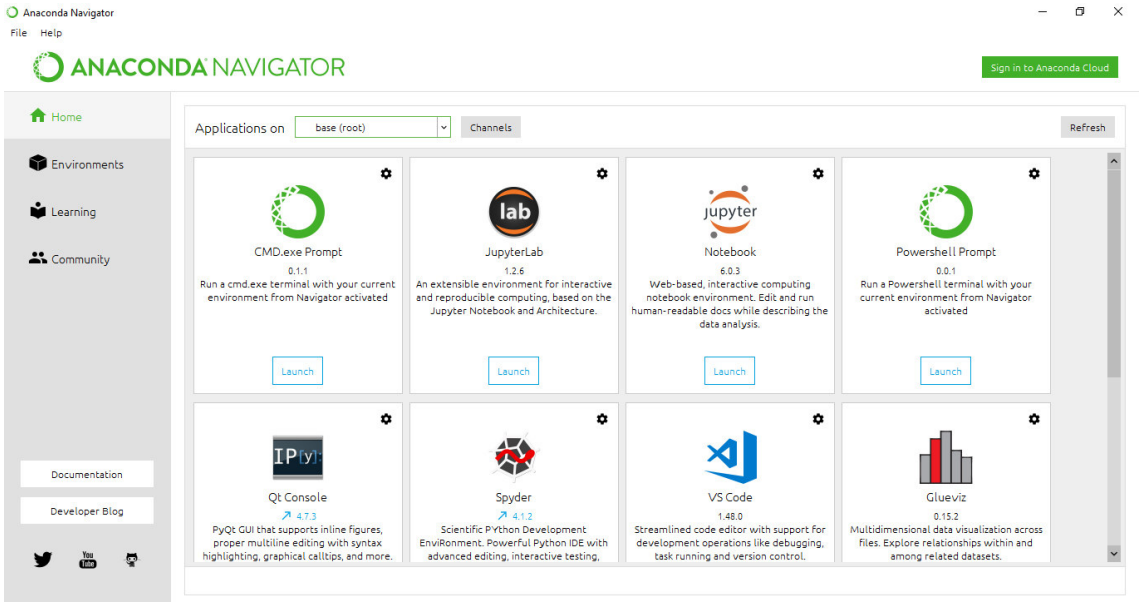
Este anexo tiene como objetivo guiar al lector para poder instalar y configurar el entorno de desarrollo utilizado durante la realización de este trabajo. A continuación, se detallan los pasos a seguir para realizar este proceso:

1- Desde https://www.anaconda.com/download/ ir hasta download y descargar la versión que corresponda según nuestro sistema operativo

2- Una vez descargado, ejecutar y seguir los pasos del Wizard, solo tener en cuenta que en uno de los pasos se podrá elegir la opción de agregar “anaconda” al path de variables de entorno (recomendado). Esto permitirá que se pueda ejecutar los comandos de “conda” desde cualquier directorio del sistema.

3- Finalizada la instalación probamos que todo esté correctamente instalado abriendo la herramienta gráfica de anaconda “Anaconda Navigator” que debe verse como la imagen a continuación.

4- Luego abrimos la consola de Linux/Mac/Windows y procedemos a instalar y actualizar las librerías que utilizaremos.



5- Escribimos:

```
conda –V     #devuelve la versión de conda instalada
python -V    #devuelve la versión de Python instalada
```

6- Ahora creamos un nuevo entorno de trabajo en donde se instalarán las librerías y dependencias necesarias para comenzar el desarrollo.

```
conda create -n <nombre del entorno a crear> python=<versión> tensorflow=<versión> keras=<versión>
```

Se debe indicar las versiones de Python, tensorflow y keras que se desean instalar en el nuevo entorno, ej. Python=3.6.7

7- Una vez creado el entorno virtual procedemos a activarlo para actualizarlo

```
conda actívate <nombre del entorno>
```

8- Luego actualizamos anaconda con los siguientes comandos:

```
conda update conda
```

y

```
conda update anaconda
```

9- Por último, ejecutamos los archivos versiones.py y versiones_deep.py que se adjuntan con este documento.

```
Python versiones.py
python versiones_deep.py
```

Esto nos debería devolver algo como lo siguiente:

```
scipy: 0.18.1
numpy: 1.12.1
matplotlib: 1.5.3
pandas: 0.19.2
statsmodels: 0.8.0
sklearn: 0.18.1
```

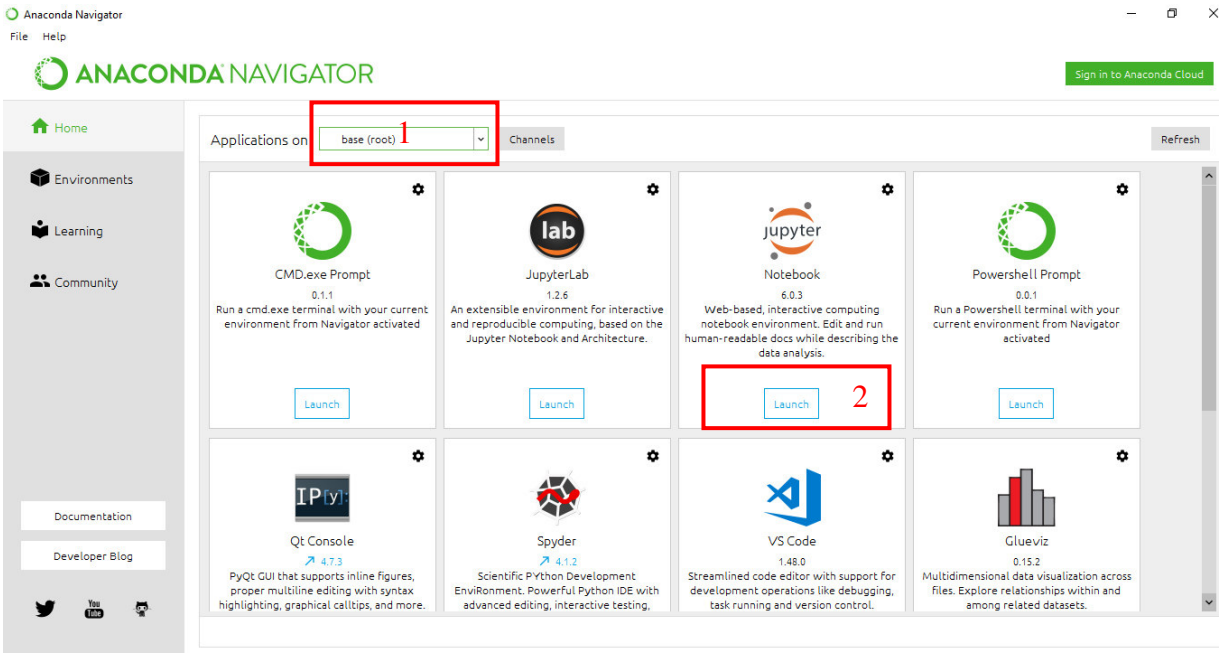
y

```
tensorflow: 2.1.0
Using TensorFlow backend.
keras: 2.3.1
```

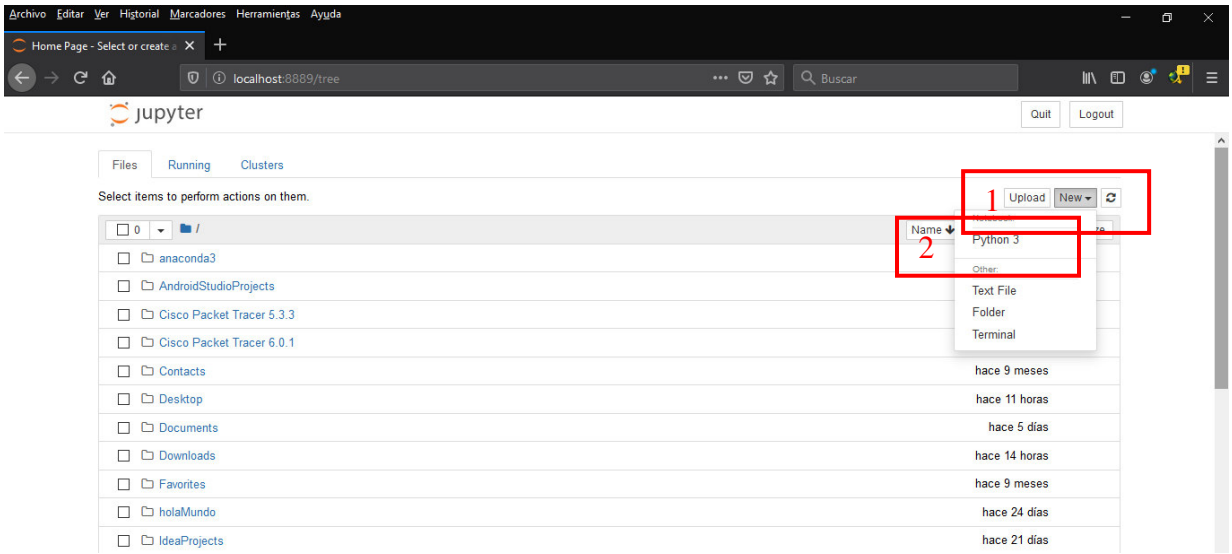
10- En caso de que alguna de las dependencias o librerías nos falte la instalaremos con el comando “pip install” seguido del nombre de la librería a instalar, ej. pip install numpy

```
pip install <nombre de la librería>
```

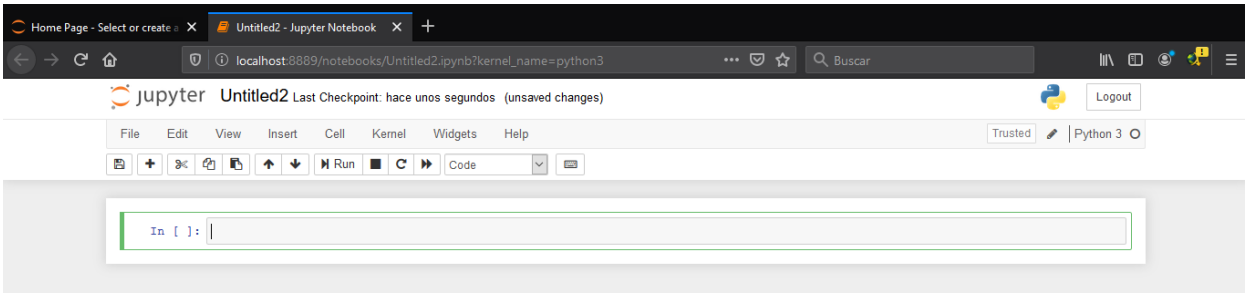
11- Después de realizar todas las actualizaciones abrimos nuevamente el “anaconda navigator” y seleccionamos el nuevo entorno creado. Y, en segundo lugar, hacemos clic en “launch” en la solapa de jupyter notebook.



12- Esto abrirá el navegador predeterminado que tengamos y se mostrará de la siguiente manera. Aquí hacer clic en “new” y luego en “Python 3”.



13- Se abrirá una nueva pestaña del navegador en donde podremos empezar a desarrollar nuestro código.

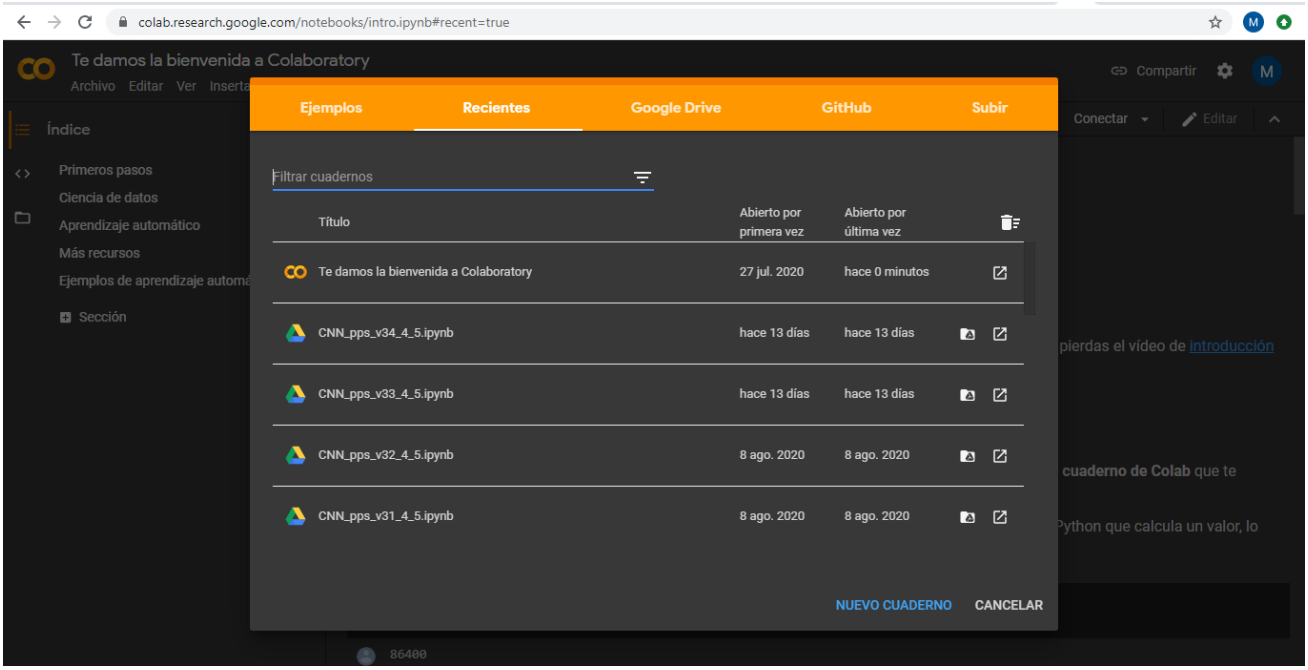


Estos cuadernos se pueden exportar a un archivo Python o con el formato “.ipynb” los cuales se pueden importar y ejecutar en google colab. Esta última es una plataforma que google brinda para realizar pruebas, además google nos brinda la posibilidad de utilizar “GPU” lo que hace que los tiempos de entrenamiento se reduzcan considerablemente. Aunque tiene como limitación que cada 12 horas aproximadamente se resetea el entorno de trabajo, lo que limita la realización de proyectos complejos y si estos proyectos se realizan utilizando GPU el tiempo proporcionado se reduce aún más e incluso luego se debe esperar un tiempo aleatorio para volver a conectarse al entorno de trabajo.

A continuación, se detallan algunas configuraciones que se pueden utilizar si optamos por realizar las pruebas en google colab

1- En primer lugar, debemos tener una cuenta de google para poder loguearnos en Colab. https://colab.research.google.com/

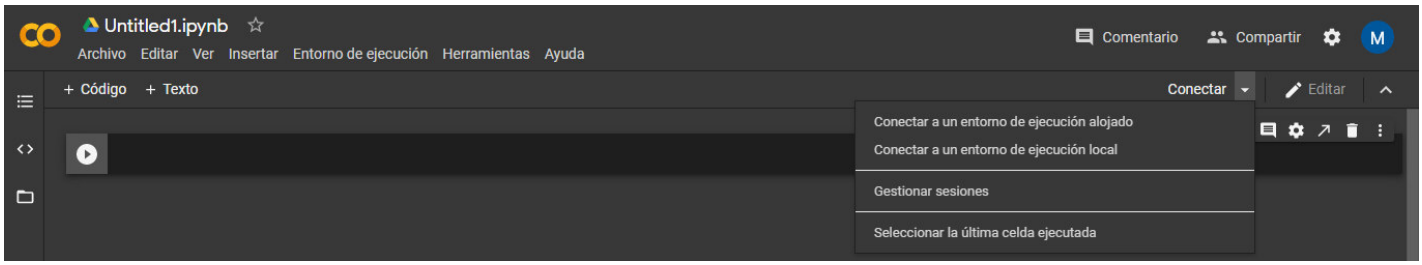
2- Se iniciará una pestaña en el navegador en la que podremos comenzar a desarrollar, la interfaz gráfica es muy parecida y compatible con Jupyter Notebook de Anaconda.



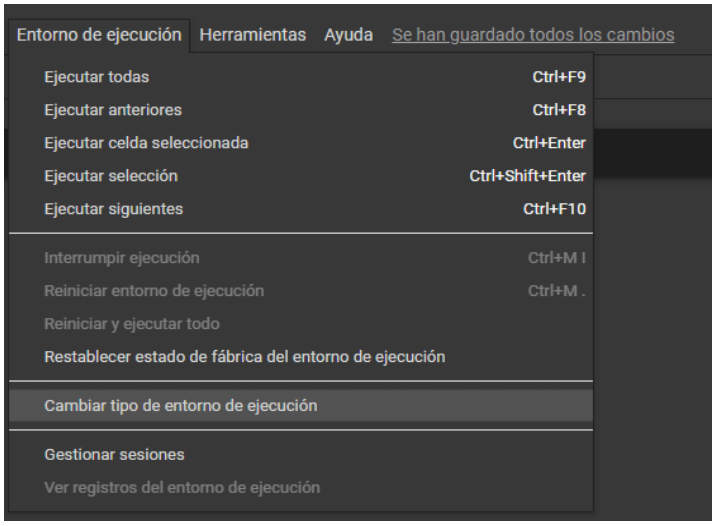
La imagen anterior es la primera ventana que nos mostrará colab

3- Para iniciar un nuevo cuaderno hacemos clic en “Archivo” y luego en “Nuevo Cuaderno”

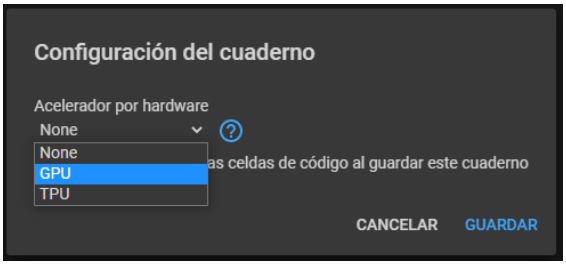
4- El paso siguiente es conectar el entorno de trabajo como se muestra en la siguiente imagen, acá podemos optar por elegir entorno local (propio de nuestro equipo, usa nuestro hardware) o alojado (entorno de hardware proporcionado por google, permite el uso de GPU y TPU).

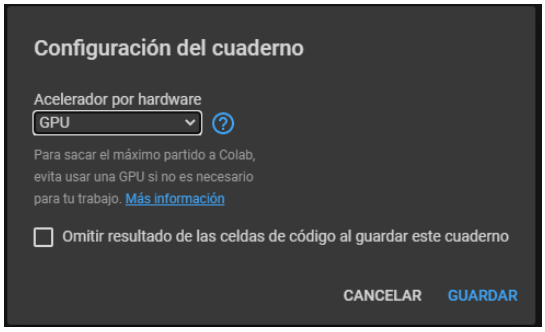


5- Para la utilización del GPU proporcionado por google debemos hacer clic en “entorno de ejecución” y luego en “cambiar tipo de entorno de ejecución” como se muestra a continuación.

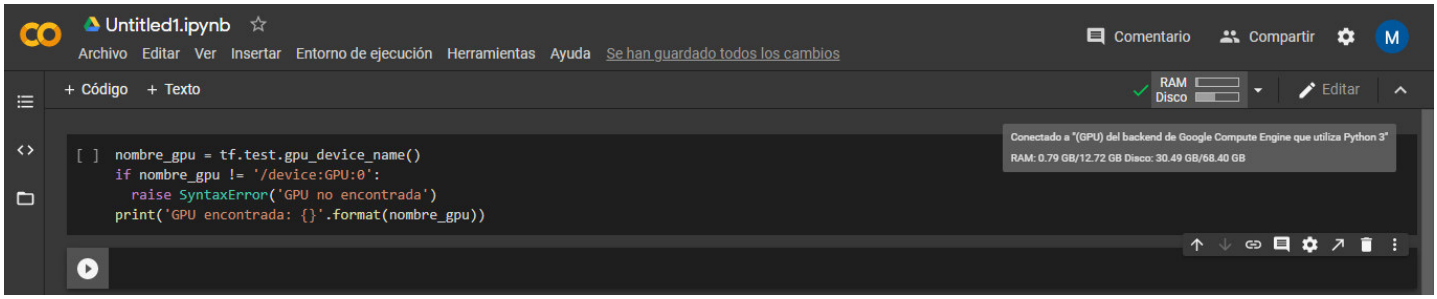


6- Para completar la elección del uso de GPU o TPU se debe seleccionar, en la ventana que se abrió en el paso anterior, la opción a elegir y luego confirmar la elección.



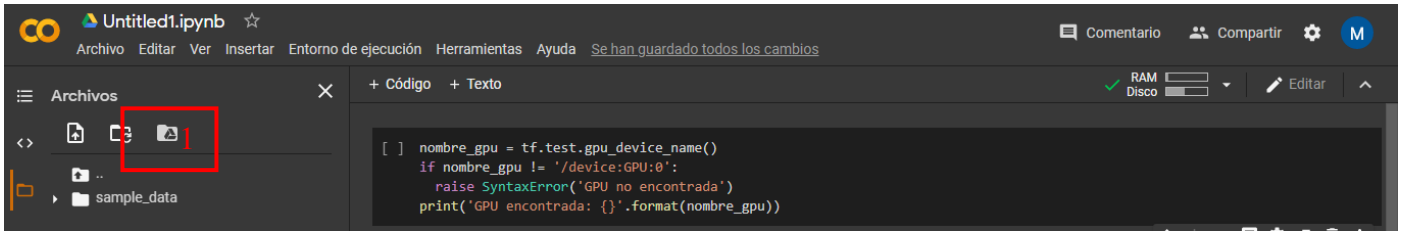


7- Podemos utilizar el siguiente código para comprobar que estamos utilizando “GPU”

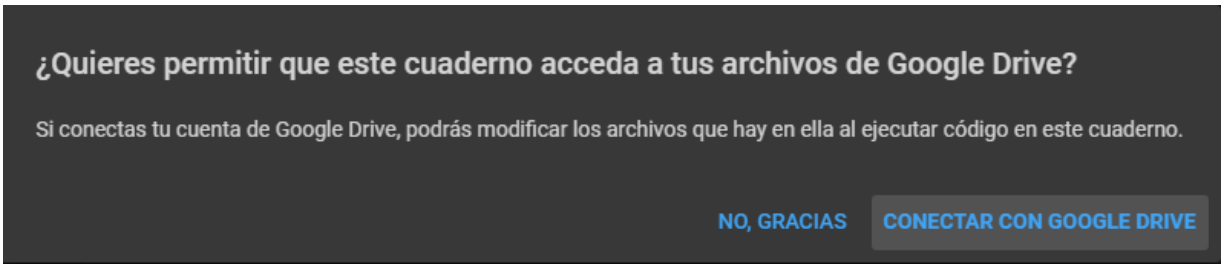


8- Por último, pero no menos importante, al utilizar google colab podemos sincronizar el cuaderno con nuestro google drive para poder hacer uso de los archivos y carpetas que tengamos ahí. Esto es muy útil para subir nuestro

dataset y poder accederlo desde cualquier equipo. En las siguientes imágenes se enumeran los pasos a seguir.

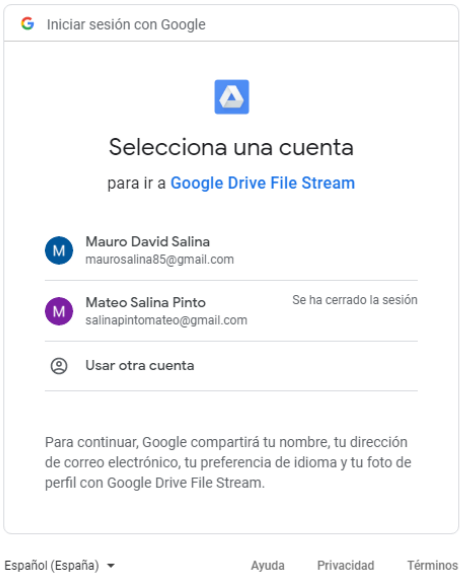


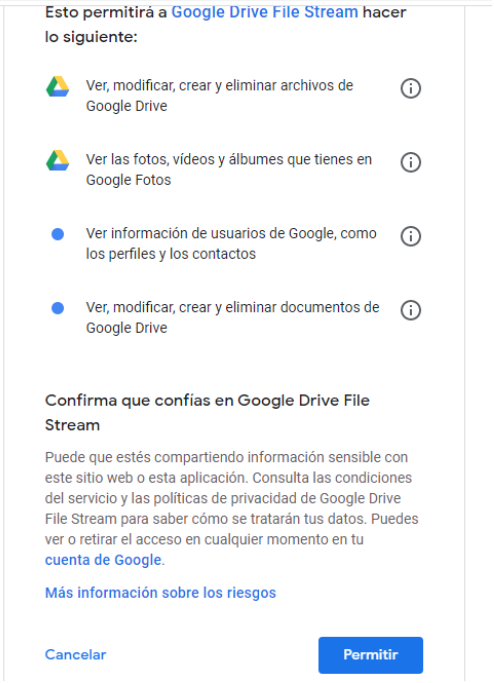
Luego de hacer click en donde indica la imagen anterior aparecerá una ventana emergente en la que se debe confirmar el acceso a las carpetas y documentos (imagen a continuación)



9- En el caso que la PC que se esté utilizando para acceder a google drive no tenga acceso directo a nuestra cuenta de google tenemos otro método para realizar la conexión anterior, esta configuración se detalla en las siguientes imágenes.

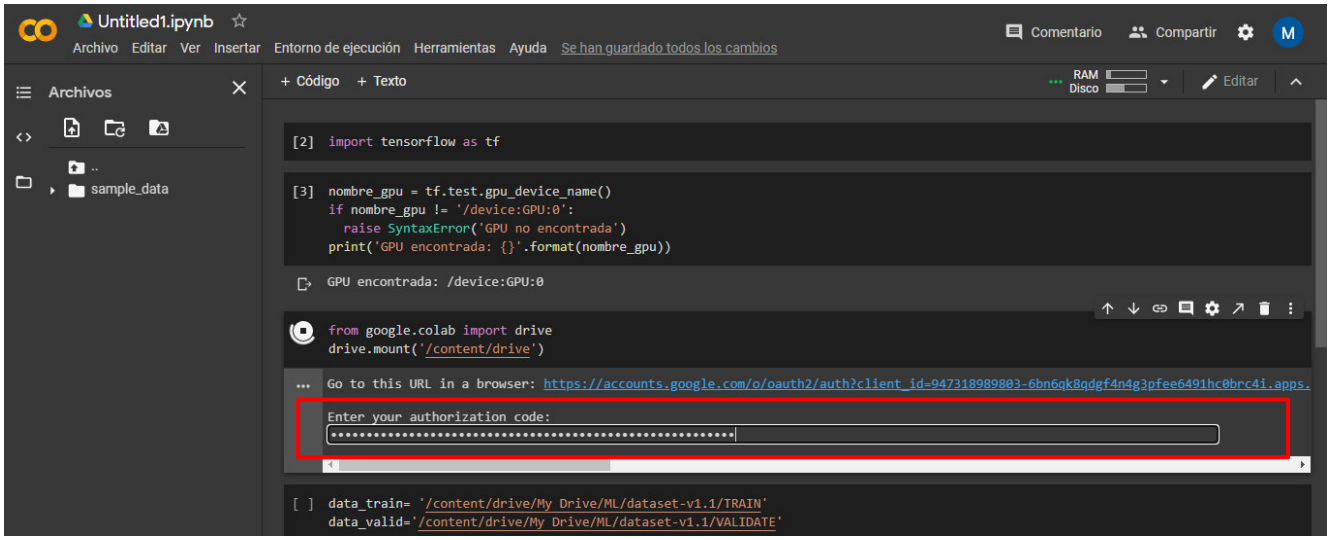
Como se ve en la imagen, se puede generar la conexión con google drive a través de código en nuestro cuaderno, ahí mismo nos mostrará un link el cual nos redirigirá a una nueva ventana para seleccionar la cuenta de google a la que se desea acceder.

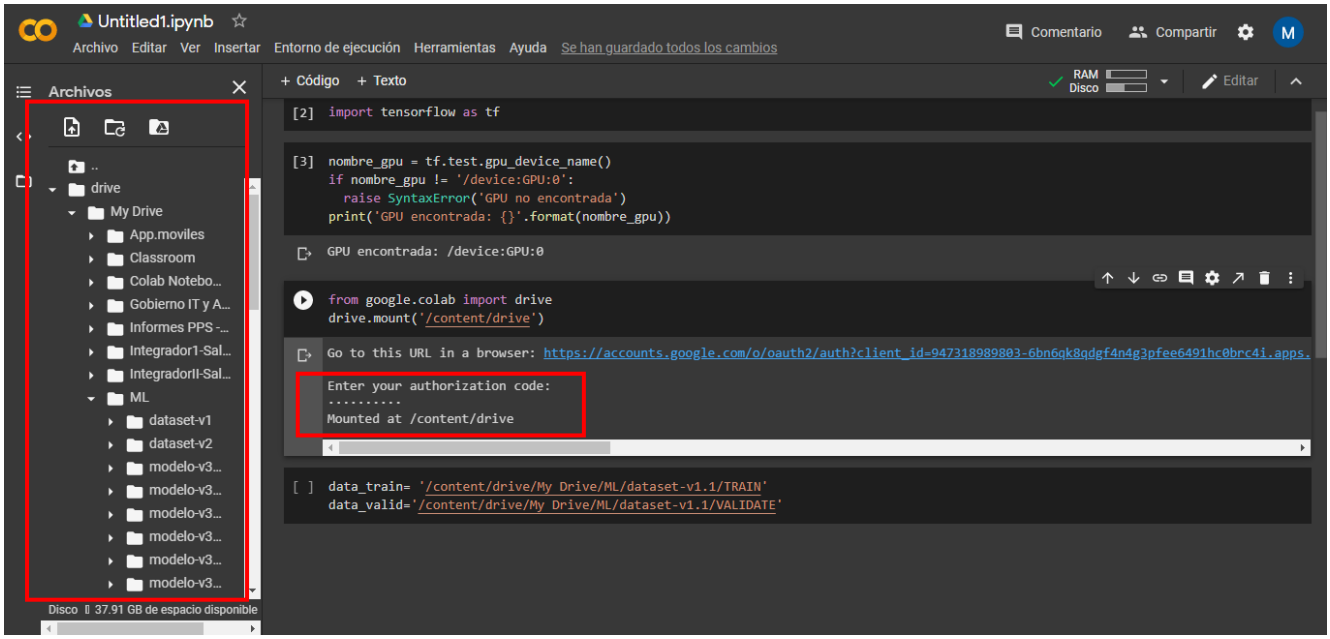




Primero seleccionamos la cuenta y luego hacemos clic en permitir.
Esto abrirá una nueva pestaña en la que se nos brindará un código que debe copiarse para luego pegarlo en nuestro cuaderno y así confirmar el acceso a google drive.

10- Por último, ingresamos el código copiado en el cuaderno y presionamos “enter” para confirmar. Luego veremos el mensaje que confirma que se montó correctamente el acceso a google drive, además en la izquierda de la pantalla veremos y tendremos acceso al árbol de directorios de google drive.





Anexo B – Ejemplo simple de red neuronal convolucional paso a paso

En este apartado se implementa una red neuronal convolucional simple, para permitir al lector comprender como es el paso a paso en la creación, entrenamiento y predicción de una CNN, esta red neuronal es creada con TensorFlow y la librería de alto nivel Keras. Además, el set de datos utilizado en este ejemplo corresponde al dataset “MNIST”, que está compuesto por 70000 imágenes de 28 x 28 pixeles monocromáticas, de dígitos numéricos que van entre cero y nueve. Del total de imágenes se utilizan 60000 para el entrenamiento del modelo y las restantes 10000 para la validación del mismo.
La parte siguiente del anexo se desarrolló a modo de tutorial paso a paso para facilitar la compresión.

1- Lo primero que debe hacerse es instalar y configurar el entorno de desarrollo, ver Anexo A, en este caso en particular se utilizó el entorno proporcionado por Google Colab.
2- Una vez que estemos en el nuevo cuaderno procedemos a importar todas las librerías que utilizaremos durante el desarrollo de la CNN como se muestra a continuación.

```
import tensorflow as tf
import os
import numpy as np
np.random.seed(2)
from keras.datasets import mnist
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers.core import Flatten, Dense
from keras.optimizers import SGD
from tensorflow.python.keras import backend as K
import pandas as pd
import mlxtend
from mlxtend.evaluate import confusion_matrix
from mlxtend.plotting import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import itertools
```

3- Luego se realiza una limpieza del entorno para evitar datos residuales en memoria.

```
K.clear_session()
```

4- Como paso siguiente se importan los datos a utilizar por el modelo, para ello se hace uso de una función provista por Keras llamada “load_data()”.

```
(x_train, y_train), (x_test,y_test) = mnist.load_data()
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
Output:
Downloading data from
https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

En la salida se corrobora que los datos fueron cargados a los tensores “x_train”, “y_train”, “x_test”, “y_test”, estos tensores son los que se utilizan desde acá en adelante.

5- Ahora se establecen algunos hiperparámetros y se realiza la normalización de los datos. Recordemos que una imagen es representada en una computadora por una matriz de pixeles y el valor de cada pixel va de 0 a 255, al normalizar estos valores cada pixel tendrá un valor entre 0 y 1.

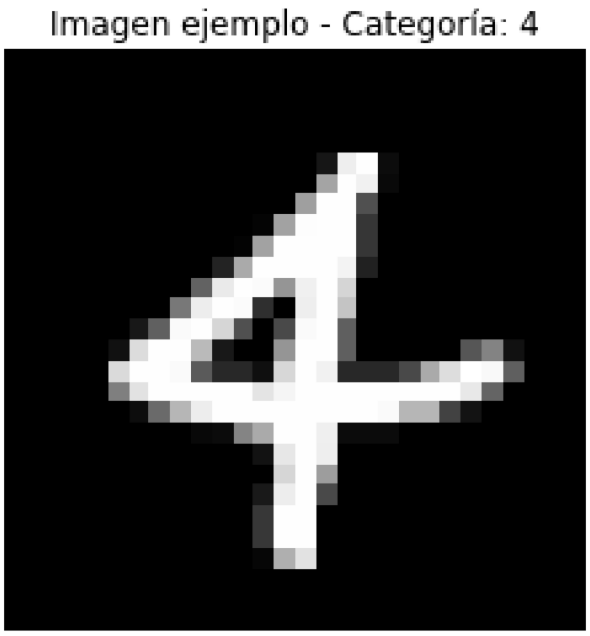
```
#Normalizacion
x_train = x_train/255.0
x_test = x_test/255.0

#Hiperparametros
nclases = 10
nepochs = 50
tam_lote = 128

y_train = np_utils.to_categorical(y_train,nclases)
y_test = np_utils.to_categorical(y_test,nclases)
```

En los hiperparámetros se define la cantidad de clases del modelo, el número de épocas o iteraciones de entrenamiento y el tamaño del lote de imágenes a procesar en cada paso.
Por último, en este paso se divide en categorías las etiquetas de las imágenes para separar las distintas clases.

6- Este paso es opcional, pero nos sirve para visualizar el tipo de imágenes con las que estamos trabajando, se utiliza la librería "matplotlib" para graficar una imagen del dataset.

```
nimagen = 150
plt.imshow(x_train[nimagen,:].reshape(28,28), cmap='Greys_r')
plt.title('Imagen ejemplo - Categoría: ' + str(np.argmax(y_train
[nimagen])))
plt.axis('off')
```

Output:



7- Se prosigue realizando una redimensión de los tensores que contienen las imágenes agregándoles una nueva dimensión, la cual es requerida por la función que se utiliza posteriormente para el entrenamiento del modelo

```
x_train = x_train.reshape(60000,28,28,1)
x_test = x_test.reshape(10000,28,28,1)
```

Esta nueva dimensión corresponde a la cantidad de colores que contienen las imágenes del conjunto de datos, en este caso se trata de imágenes monocromáticas por eso el valor de esta dimensión se establece en 1, si estuviésemos trabajando con imágenes a color “RGB” dicho valor sería 3.

8- Ahora sí estamos listos para realizar uno de los pasos más importantes, que es la creación del modelo que luego se entrenará.

```
modelo = Sequential()

# CONV1 Y MAX-POOLING1
modelo.add(Conv2D(filters=8, kernel_size=(5,5), activation='relu
', input_shape=(28,28,1)))
modelo.add(MaxPooling2D(pool_size=(2,2)))

# CONV2 Y MAX-POOLING2
modelo.add(Conv2D(filters=16, kernel_size=(5,5), activation='rel
u'))
modelo.add(MaxPooling2D(pool_size=(2,2)))

# Aplanar, FC1, FC2 y salida
modelo.add(Flatten())
modelo.add(Dense(128,activation='relu'))
modelo.add(Dense(96,activation='relu'))
modelo.add(Dense(nclases,activation='softmax'))
```

En las líneas anteriores se crea el nuevo modelo del tipo “secuencial”, luego se agregan dos capas convloucionales, la primera con 8 filtros de 5x5, una función de activación no lineal “Relu” y un agrupamiento utilizando “maxpooling” con filtros de 2x2, la segunda capa es muy similar a la primera, con la diferencia que en esta los filtros aplicados son 16.
Como se puede ver en la primera convolucion se incluye el parámetro “input_shape” es este el que define la capa de entrada indicando el tamaño de las imágenes y si se trata de imágenes monocromáticas o a color.
Luego se continúa con la capa completamente conectada que comienza con la operación de aplanamiento “Flatten” y luego se agregan dos capas ocultas

densas con activación “Relu”, 128 y 96 neuronas respectivamente. Por último, se agrega la capa de salida con 10 neuronas correspondientes a cada una de las clases y activación “Softmax” para realizar la clasificación multiclase.

9- Para divisar la arquitectura del modelo creado y la cantidad de parámetros a entrenar de utiliza la siguiente línea:

```
# Arquitectura del modelo creado
modelo.summary()
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 8)         208
max_pooling2d (MaxPooling2D) (None, 12, 12, 8)         0
conv2d_1 (Conv2D)            (None, 8, 8, 16)          3216
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 16)          0
flatten (Flatten)            (None, 256)               0
dense (Dense)                (None, 128)               32896
dense_1 (Dense)              (None, 96)                12384
dense_2 (Dense)              (None, 10)                970
=================================================================
Total params: 49,674
Trainable params: 49,674
Non-trainable params: 0
```

El total de parámetros a entrenar es de 49674 (cuarenta y nueve mil seiscientos setenta y cuatro).

10- El paso siguiente es definir el optimizador y paso de entrenamiento (learning rate) y luego compilar el modelo indicando cual será la métrica a analizar y la función de pérdida a optimizar, que para este modelo son “accuracy” y “categorical_crossentropy” respectivamente.

```
sgd = SGD(lr=0.1) #rate_learn
modelo.compile(loss='categorical_crossentropy', optimizer=sgd, m
etrics=['accuracy'])
```

11- Luego de compilar el modelo estamos listos para realizar el entrenamiento del mismo.

```
history = modelo.fit(x_train,y_train,epochs=nepochs,batch_size=t
am_lote, verbose=1, validation_data=(x_test,y_test))
```

Con el llamado a la función “fit()” comienza el entrenamiento, esta función recibe como parámetros los datos de entrenamiento y sus etiquetas, además la cantidad de épocas, el tamaño del lote y los datos para la validación.

Output:

```
Epoch 1/50
469/469 [==============================] - 9s 5ms/step - loss:
0.8317 - accuracy: 0.7350 - val_loss: 0.0986 - val_accuracy: 0.9677
Epoch 2/50
469/469 [==============================] - 2s 4ms/step - loss:
0.1057 - accuracy: 0.9670 - val_loss: 0.0764 - val_accuracy: 0.9762
Epoch 3/50
469/469 [==============================] - 2s 4ms/step - loss:
0.0742 - accuracy: 0.9759 - val_loss: 0.0635 - val_accuracy: 0.9789
…….

Epoch 18/50
469/469 [==============================] - 2s 4ms/step - loss:
0.0089 - accuracy: 0.9970 - val_loss: 0.0418 - val_accuracy: 0.9876
Epoch 19/50
469/469 [==============================] - 2s 4ms/step - loss:
0.0074 - accuracy: 0.9974 - val_loss: 0.0393 - val_accuracy: 0.9878
Epoch 20/50
469/469 [==============================] - 2s 4ms/step - loss:
0.0066 - accuracy: 0.9980 - val_loss: 0.0355 - val_accuracy: 0.9898
…….

Epoch 35/50
469/469 [==============================] - 2s 4ms/step - loss:
0.0022 - accuracy: 0.9993 - val_loss: 0.0491 - val_accuracy: 0.9887
Epoch 36/50
469/469 [==============================] - 2s 4ms/step - loss:
9.4767e-04 - accuracy: 0.9997 - val_loss: 0.0427 - val_accuracy:
0.9897
Epoch 37/50
469/469 [==============================] - 2s 4ms/step - loss:
8.0860e-04 - accuracy: 0.9998 - val_loss: 0.0487 - val_accuracy:
0.9895
…….

Epoch 48/50
```

```
469/469 [==============================] - 2s 4ms/step - loss:
1.8616e-04 - accuracy: 1.0000 - val_loss: 0.0438 - val_accuracy:
0.9902
Epoch 49/50
469/469 [==============================] - 2s 4ms/step - loss:
1.8984e-04 - accuracy: 1.0000 - val_loss: 0.0435 - val_accuracy:
0.9904
Epoch 50/50
469/469 [==============================] - 2s 4ms/step - loss:
1.3179e-04 - accuracy: 1.0000 - val_loss: 0.0441 - val_accuracy:
0.9902
```

Luego de 50 iteraciones de entrenamiento se logra un acierto del 100% para el set de entrenamiento y un acierto de 99,02% para el set de validación.

12- Con el paso anterior tendremos entrenado el modelo, por lo tanto, procedemos a guardarlo juntos con los pesos para su posterior uso en futuras predicciones.

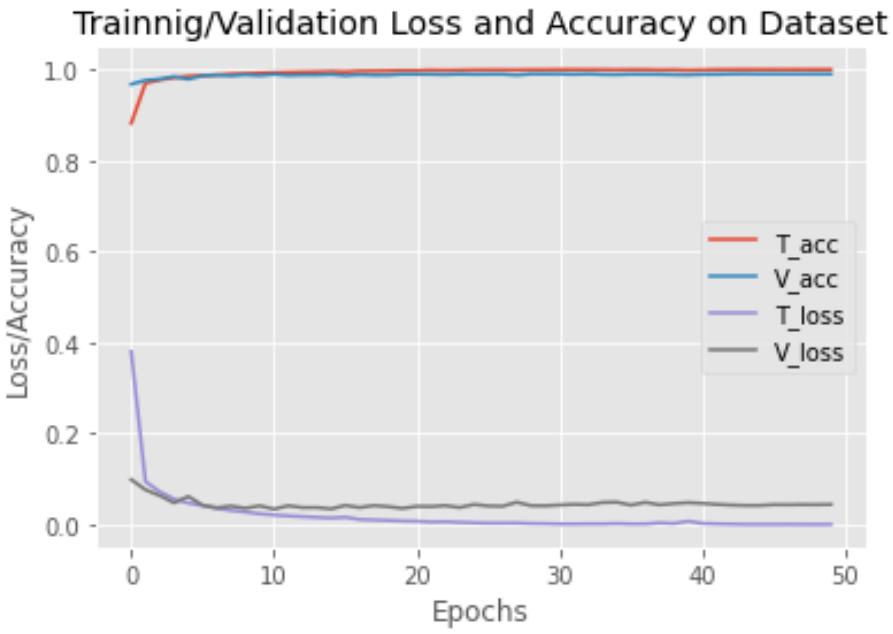
```
##Guardo el modelo y los pesos
dir='/content/drive/My Drive/ML/modelo-anexoB/'
if not os.path.exists(dir):
      os.mkdir(dir)

modelo.save('/content/drive/My Drive/ML/modelo-
anexoB/modelo.h5')
modelo.save_weights('/content/drive/My Drive/ML/modelo-
anexoB/pesos.h5')
```

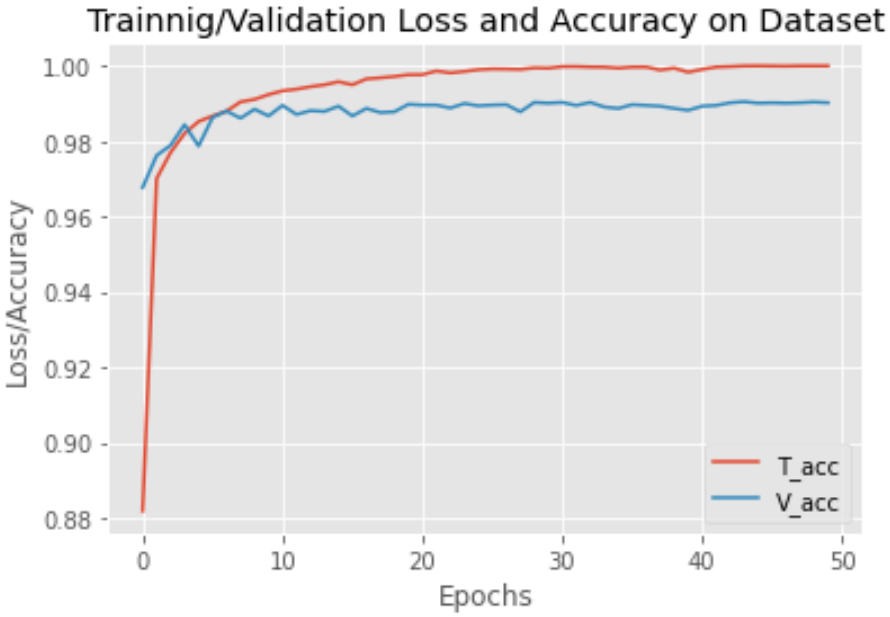
13- Este paso muestra cómo graficar las métricas del entrenamiento del modelo, es opcional, aunque permite una mejor interpretación de las métricas. Se graficará el acierto y la pérdida tanto del set de entrenamiento como del set de validación, estas líneas se pueden ir combinando para obtener distintos gráficos.

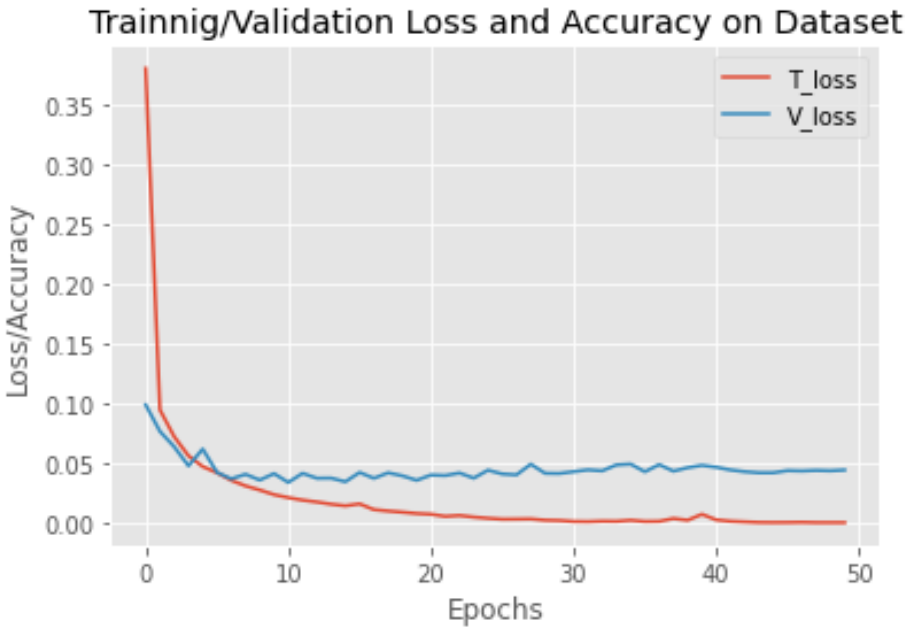
```
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, nepochs), history.history["accuracy"], lab
el="T_acc")
plt.plot(np.arange(0, nepochs), history.history["val_accuracy"],
 label="V_acc")
plt.plot(np.arange(0, nepochs), history.history["loss"], label="
T_loss")
```

```
plt.plot(np.arange(0, nepochs), history.history["val_loss"], lab
el="V_loss")
plt.title("Trainnig/Validation Loss and Accuracy on Dataset")
plt.xlabel("Epochs")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="best")
plt.savefig("/content/drive/My Drive/ML/modelo-anexoB/anexoB-
e.png")
```

A continuación, se presentan algunos de los gráficos obtenidos.







Para finalizar con este anexo se muestra cómo realizar la predicción y representar los datos en una matriz de confusión para que puedan ser analizados de una manera más clara.

1- Se realiza la predicción:

```
prediction=modelo.predict(x_test, verbose=1)
y_pred = np.argmax(prediction, axis=1)
```

```
y_ref = np.argmax(y_test,axis=1)
```

2- Luego se realiza la matriz de confusión.

```
matriz=confusion_matrix(y_ref, y_pred)
```

3- Este paso es opcional, se define una función para representar en una imagen a color la matriz definida en el paso anterior.

```
def plot_matriz(matriz, clases, normalize=False, title='Matri
z de confusion', cmap=plt.get_cmap('Oranges')):
    plt.imshow(matriz, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks=np.arange(len(clases))
    plt.xticks(tick_marks, clases, rotation=90)
    plt.yticks(tick_marks, clases)

    if normalize:
        matriz=matriz.astype('float') / matriz.sum(axis=1)[:,
 np.newaxis]
        print('Matriz normalizada')
    else:
        print('Matriz sin normalizacion')

    print(matriz)
    print()

    thresh=matriz.max()/2
    for i, j in itertools.product(range(matriz.shape[0]), ran
ge(matriz.shape[1])):
        plt.text(j, i, matriz[i,j], horizontalalignment="cent
er", color="white" if matriz[i,j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel("Reales")
    plt.xlabel("Predicciones")
    plt.savefig("/content/drive/My Drive/ML/modelo-
anexoB/matriz.png")
```

4- Por último, se definen las etiquetas del gráfico y se imprime la matriz de confusión con dos representaciones distintas, primero una clásica y luego una con gráfica a color.

```
matriz_plot_labels=['numero 0','numero 1','numero 2','numero
3','numero 4','numero 5','numero 6','numero 7','numero 8','nu
mero 9'] #[0,1,2,3,4,5,6,7,8,9] #mnist.test.labels

plot_matriz(matriz, mtriz_plot_labels)
```

Output:

```
Matriz sin normalizacion
[[ 976    0    0    0    0    0    2    2    0    0]
 [   0 1128    2    1    0    0    0    2    2    0]
 [   0    0 1026    1    1    0    0    3    1    0]
 [   0    0    0 1002    0    3    0    0    4    1]
 [   0    1    1    0  973    0    0    2    1    4]
 [   2    0    1    4    0  881    1    0    2    1]
 [   3    2    0    0    2    5  945    0    1    0]
 [   0    4    4    1    1    0    0 1016    2    0]
 [   4    0    1    1    0    3    1    0  962    2]
 [   0    0    0    3    6    1    0    4    2  993]]
```