UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

Uso de Redes Neuronales Convolucionales para el Reconocimiento Automático de Imágenes de Macroinvertebrados para el Biomonitoreo Participativo

Autor:

Carlos Andrés Quintero Samaniego

Supervisor:

Dr. Javier E. Sánchez-Galán

Trabajo de graduación para optar al título de

Licenciado en Ingeniería de Sistemas y Computación

2018

A Dios Todopoderoso quien me bendijo con una fuente de fortaleza, mi familia. Bendiciones a mis padres Eugenio Quintero y Neyra Samaniego; a mi hermano Javier Sánchez; y a Yeniffer Ortega. Esto va dedicado por su amor y apoyo.

Reconocimientos

Primero y más importante deseo agradecer a Dios por darme la fuerza, conocimiento, habilidad y oportunidad para culminar esta investigación de manera satisfactoria, sin su bendición este logro no habría sido posible.

En mi camino a través de este proyecto he encontrado un maestro, amigo y pilar como asesor, el Dr. Javier Sánchez-Galán. Por su continuo apoyo en la investigación, por su paciencia, motivación e inmenso conocimiento. Su apoyo me ayudó en cada paso de investigación y escritura de esta tesis. No puedo imaginar un mejor mentor para este proyecto de investigación.

Aparte de mi asesor debo dar un agradecimiento especial al resto del comité: Dr. Fernando Merchán y Mgter. Aydée Cornejo, por sus comentarios y motivaciones, pero también por las preguntas que incentivaban a ampliar la investigación en varias perspectivas.

Debo agradecer al grupo de científicos del centro de investigación del Instituto Conmemorativo Gorgas de Estudios de la Salud (ICGES), por brindarme acceso al laboratorio y por su apoyo en el conocimiento entomológico.

Al Dr. Víctor López por vincularme al equipo de trabajo del Dr. Fernando Merchán luego de expresarle mis intereses académicos.

Índice general

Índice de figuras

Índice de tablas

Resumen Descriptivo

El reconocimiento de organismos macroinvertebrados en ríos es utilizado como una técnica de medición para la calidad del agua. Esta técnica es un método aplicable en comunidades rurales que hacen uso de afluentes naturales para consumo. Para aplicarlo se requiere de entrenamiento en la captura y reconocimiento de los organismos. Se han desarrollado algunos métodos para solucionar este problema, entre estos se incluyen una guía con imágenes para teléfonos y un algoritmo que relaciona preguntas sobre las características del mismo para facilitar el reconocimiento. El uso de técnicas de inteligencia artificial para reconocimiento por imágenes ha sido estudiado con detalle, el método hace uso de redes neuronales y mediante un algoritmo de entrenamiento se genera un modelo capaz de reconocer diferentes clases. El desarrollo de estos sistemas requiere de la construcción de una base de datos para entrenar el algoritmo, que en el caso de este proyecto incluye 81 familias de macroinvertebrados. Para esta tesis se experimenta con hasta 14 familias cuyos resultados mostraron un porcentaje por encima del 75 % de precisión y sobre el 93 % al considerar las 3 clases con mayor puntaje. Un factor importante para lograr estos resultados fue la implementación de algoritmos de aumentación de datos y regularización. Estos resultados se mantienen durante la experimentación con diferentes conjuntos de datos lo cual es indicio de que el proyecto puede funcionar una vez se proceda con el reconocimiento de las

81 familias que abarca el proyecto. Para lograr abarcar una completa extensión del problema es necesario generar la base de datos, proceso limitado por la disponibilidad de imágenes de algunas familias. Los resultados del proyecto así como la metodología utilizada han sido presentados de una manera en que pueden ser utilizados para la generación de un modelo de redes neuronales capaz del reconocimiento de todas las familias de macroinvertebrados que considera el proyecto una vez se cuente con la base de datos.

Capítulo 1

Introducción

Cuando miramos una imagen de una multitud, nuestro cerebro es capaz de reconocer rostros de conocidos y extraños, determinar género, etnia y edad aproximada en fracciones de segundo. Además somos capaces de detectar elementos como la hora del día o el clima solo con la iluminación de una escena. Pero cuando una computadora "observa" la misma imagen no ve nada, al menos mientras no apliquemos algoritmos de visión por computadora. A la fecha se han generado algoritmos que brindan la capacidad de ubicar los rostros de cada persona, hacer un conteo, calcular sus edades y encontrar diferentes elementos en la imagen.

El campo de la inteligencia artificial ha cambiado considerablemente en los últimos años, principalmente desde la publicación de los resultados de ILSVRC (ImageNet Large Scale visual Recognition Competition) 2012 (Krizhevsky, Sutskever e Hinton 2012) donde por primera vez una red neuronal profunda lograba vencer y por un amplio margen, esta fue llamada AlexNet. Estos resultados cambiaron radicalmente el estado del arte y constituyeron el inicio de una nueva era de la inteligencia artificial. El uso de las Redes Neuronales Profundas permitió la solución de múltiples problemas relacionados

con la clasificación de patrones.

Muchas investigaciones se enfocaron en el desarrollo de sistemas capaces de resolver problemas como el reconocimiento del habla, traducción de lenguajes (Zhang y Zong 2015), procesamiento de imágenes (Szegedy, Toshev y Erhan 2013), audio (Owens y col. 2016) y video (Suwajanakorn, Seitz y Kemelmacher-Shlizerman 2017).

El objetivo de este trabajo es desarrollar una herramienta de reconocimiento de imágenes para el Instituto Conmemorativo Gorgas de Estudios de la Salud (ICGES), el cual está interesado en el monitoreo de macroinvertebrados presentes en las tomas de agua por medio del proyecto de Biomonitoreo Participativo de la calidad del Agua con Juntas Administradoras de Acueductos Rurales (JAAR); dirigido por la investigadora Aydée Cornejo (ICGES) en colaboración con la UTP.

Los macroinvertebrados son bioindicadores de la calidad de agua en ríos. En efecto, las diferentes especies requieren de condiciones especializadas y debido a esto presentan una manera sencilla de evaluar la calidad de estas aguas. Con la implementación de visión por computadora podemos brindar la posibilidad a los miembros de las comunidades rurales de identificar las especies desde su teléfono celular.

1.1. Inteligencia Artificial y las Redes Neuronales

En la actualidad existe un gran interés en los algoritmos de aprendizaje automático (conocido en inglés como «machine learning»). Estos algoritmos tienen como objetivo el manejo y análisis de cantidades masivas de datos

que permiten su uso para entrenamiento y prueba en tareas de clasificación, predicción, asociación y agrupación. El aprendizaje automático se divide en aprendizaje supervisado y no supervisado.

Aprendizaje supervisado es aquel donde se utiliza una función mapeo desde una entrada con respecto a una salida, ambas conocidas. Su objetivo es aproximar esta función de manera precisa para cuando se ingresen datos nuevos el algoritmo suministre una salida certera. Este aprendizaje se encarga de las tareas de clasificación y predicción.

Aprendizaje no supervisado es aquel donde se tienen los datos de entrada pero no su correspondiente salida. En estos no se tienen respuestas correctas y no hay instructor que valide, los algoritmos son dejados por su cuenta en el descubrimiento de las estructuras de datos. Este aprendizaje se encarga de las tareas de asociación y agrupación.

Las redes neuronales artificiales (RNA) son un modelo computacional que utiliza una amplia colección de funciones matemáticas llamadas neuronas artificiales. Típicamente las neuronas se conectan en capas y las señales transitan desde la primera (entrada) hasta la última (salida), pasando por capas intermedias (ocultas), ver Figura 1.1. La conexión entre cada neurona es definida por un valor numérico llamado peso, adicionalmente cada neurona contiene un valor de tendencia llamado «bias» que agrega flexibilidad al modelo.

Como tal, el entrenamiento de una red neuronal tiene como función objetivo minimizar un valor error en el aprendizaje supervisado o en maximizar un valor recompensa en el no supervisado, modificando los pesos y tendencias de las neuronas. Generalmente este aprendizaje se realiza de manera iterativa, entrenando la red y ajustando los pesos de las conexiones hasta que se obtenga

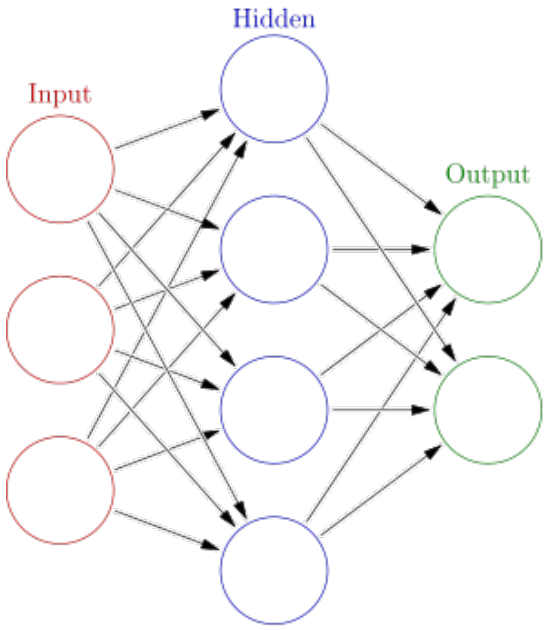


FIGURA 1.1: Esquema de una red neuronal con 3 capas: capa de entradas, capa de salidas (objetivos) y la capa oculta que aprende el modelo (*Colored_neural_network*).

la salida deseada.

1.1.1. Redes Neuronales Profundas

Redes Neuronales Profundas o «Deep Neural Networks» es el nombre que se da a las nuevas arquitecturas de las redes neurales y algoritmos de «machine learning». Estas nuevas arquitecturas de redes han logrado cambios importantes en la dirección en la que se desarrolla la inteligencia artificial al proveer estructuras más eficientes a nivel de memoria y computación; además del desarrollo de nuevos y mejores algoritmos para el aprendizaje, que permiten reducir el tiempo de entrenamiento y mejorar los resultados para el reconocimiento de patrones(Goodfellow, Bengio y Courville 2016).

FIGURA 1.2: Reconocimiento de objetos en imágenes utilizando CNN basado en regiones (R-CNN) (Szegedy, Toshev y Erhan 2013).

Muchos investigadores de inteligencia artificial han volcados sus esfuerzos al desarrollo de esta tecnología y se han creado sistemas para resolver problemas que hasta la fecha parecían imposibles (Brownlee 2016), por ejemplo: sistemas capaces de reconocer el habla e incluso traducirlo de manera precisa, reconocimiento de objetos en videos e imágenes (figura 1.2), generación de música (Coutinho y col. 2014), generación de poesía (Karpathy 2015), entre otros.

En el capítulo 2 se explicará con detalle la estructura de una red neuronal profunda.

1.1.2. Redes Neuronales Convolucionales

Para el reconocimiento de imágenes se hace uso de las redes neuronales convolucionales (CNNs por sus siglas en inglés), las cuales son un modelo donde las neuronas corresponden a campos receptivos de una manera muy similar a las neuronas de la corteza visual primaria de un cerebro biológico.

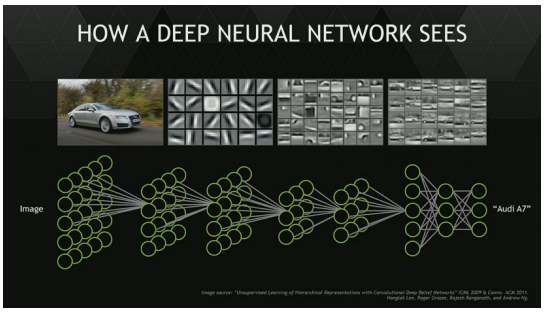


FIGURA 1.3: Esquema de una red neuronal convolucional. La imagen original en pixeles es subdividida en sus características, por ejemplo bordes, formas geométricas y partes de un carro (*CS231n Convolutional Neural Networks for Visual Recognition*).

La red se compone de múltiples capas, en el principio se encuentra la fase de extracción de características compuesta de neuronas convolucionales y de reducción; a medida que se avanza en la red se disminuyen las dimensiones activando características cada vez más complejas; al final se encuentran neuronas sencillas para realizar la clasificación. En la Figura 1.3, se muestra la arquitectura de Red Convolucional aplicada a imágenes.

Ya desde los 80's existían ideas sobre la implementación de CNNs, pero es hasta 1998 con un artículo seminal de Yann LeCun (LeCun y col. 1998a) que se indica la propuesta de una de las ideas más importantes de esta tecnología para el reconocimiento de imágenes utilizando redes neuronales convolucionales y retropropagación.

El éxito de estos sistemas se debe en gran parte a la evolución de la capacidad computacional (Cireşan y col. 2010). La inteligencia artificial actualmente permite la solución de problemas en diversos campos que requieren un alto costo computacional que van desde el diseño de asistentes personales, traductores, vehículos autónomos, hasta sistemas de recomendaciones de productos multimedia y compras en línea entre otros (Adams 2017).

Esto queda claramente demostrado en 2012 cuando gracias al desarrollo de AlexNet, una red neuronal que logró un 15 % de error en la identificación de imágenes para el concurso ILSVRC (ImageNet Large Scale Visual Recognition Competition) donde su rival más cercano marcó 25 % (Krizhevsky, Sutskever e Hinton 2012). El desarrollo de técnicas de aprendizaje profundo permitió que en la edición 2015 del concurso se lograra 4.8 % de error, excediendo la precisión humana de 5.1 % en esta misma prueba (Ioffe y Szegedy 2015).

En el capítulo 2 se explica la estructura y se explican los beneficios que se obtienen de la misma.

1.2. Aplicaciones en el Mundo

Las grandes compañías de la computación han hecho sus apuestas a la inteligencia artificial y en la actualidad podemos disfrutar de muchos de los beneficios mediante los productos que son lanzados al mercado. Diariamente utilizamos directa o indirectamente sistemas como los de etiquetado de imágenes en la red, traducción de idiomas, reconocimiento de voz y servicios de recomendación de productos como los que utiliza Netflix para mostrarte su contenido o los anuncios de Google.

El verdadero potencial de los sistemas «deep learning» está más allá de estas aplicaciones y es posible verlos en desarrollo por empresarios y científicos. Con el desarrollo algoritmos y las mejoras continuas de hardware computacional nuevas aplicaciones van surgiendo en áreas tan diversas como la energía, medicina, física, clima y economía.

Esto explica porqué compañías como Intel ha concretado adquisiciones de hardware y software de compañías como Nervana Systems y Movidius; porqué Nvidia ha definido el futuro empresarial a la aceleración de los sistemas «deep learning», y la razón del buen crecimiento de miles de empresas «startup» en Sillicon Valley (*The Next Wave of Deep Learning Applications* 2016).

Para el área de la medicina se desarrolló un sistema para la visualización y detección de melanomas (Jafari y col. 2016), permitiendo tratar una de las formas de cáncer más agresivas en etapas tempranas. Otro ejemplo muy similar fue utilizado para la detección de cáncer cerebral (Havaei y col. 2016).

En el área de predicciones contamos con sistemas desarrollados para detección de eventos climáticos al utilizar redes neuronales y dinámicas de fluidos en conjunto con algoritmos genéticos para la detección de ciclones (Safikhani 2016). Y en el área de economía se hicieron predicciones en las fluctuaciones de oferta y demanda de una red eléctrica y así predecir los patrones de consumo más eficientes y ayudar con la reducción de costos (Azimi, Ghofrani y Ghayekhloo 2016).

1.3. Librerías para Machine Learning

Los sistemas de redes neuronales requieren de múltiples operaciones numéricas para realizar el procesamiento de matrices las cuales usualmente son tratadas en paralelo beneficiandose del uso de algoritmos de operaciones en paralelo. Para facilitar la tarea de programación podemos utilizar librerías especializadas, algunas de las más fuertes a la fecha son Tensorflow y Theano. La selección de Tensorflow como plataforma para el proyecto es basada en

la calidad y cantidad de contenido relacionado que son utilizados como referencias.

1.3.1. Tensorflow

Tensorflow es una librería de código abierto desarrollada por Google que facilita el desarrollo de proyectos en el área de redes neuronales profundas. Su modelo basado en arreglos de datos multidimensionales (llamados tensores) y nodos que representan operaciones matemáticas facilita la tarea de implementación de sistemas de aprendizaje automático. Tensorflow ofrece facilidad para realizar computación en plataformas CPU o GPU y el uso de lenguajes como Python, C++, Haskell, Java y Go al utilizar la API.

1.3.2. TensorBoard

Adicionalmente provee una herramienta llamada TensorBoard que permite la visualización del proceso de aprendizaje al proyectar datos generados durante el entrenamiento. Algunos de esos valores son el coeficiente de error, la precisión de predicción o valores más especializados como la kappa de Cohen; para luego evaluarlos con respecto al tiempo o a diferentes etapas o ciclos en el aprendizaje.

Es también posible visualizar contenido multimedia como imágenes, audio y video; además permite comparar el comportamiento de diferentes versiones del modelo. En la Figura 1.4 se muestra la estructura de una capa del modelo y el flujo de los datos de los tensores de peso y tendencia (bias).

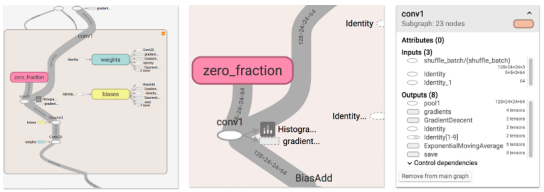


FIGURA 1.4: Visualización de modelo utilizando complemento TensorBoard.

1.3.3. Keras

Keras es la librería de alto nivel para el desarrollo de redes neuronales, es escrita en Python y es capaz de correr sobre otras librerías como Tensorflow o Theano. El enfoque de esta librería es facilitar una rápida implementación siendo la clave para muchas investigaciones científicas.

La librería es recomendada cuando se desea hacer una rápido prototipado utilizando algoritmos de redes convolucionales y redes recurrentes, así como combinaciones de ambas. Al correr sobre Tensorflow y Theano esta soporta computación por CPU y GPU.

Keras es una API amigable, diseñada para su fácil uso al manejar ciertos procesos y declaraciones de manera automática y transparente, reduciendo líneas de código y carga cognitiva en el programador. Keras ofrece un sistema es modular, donde los modelos son definidos como secuencias o grafos de ejecución. Estos módulos son completamente configurables y pueden ser interconectados de manera sencilla. La librería ofrece múltiples herramientas para el manejo de redes neuronales, funciones de costo, optimizadores, esquemas de inicialización, funciones de activación y esquemas de regularización que trabajan como módulos independientes y al ser combinados permiten la creación de nuevos modelos.

1.3.4. Implementación en Móviles (Android, iOS, Raspberry Pi)

Tensorflow fue desarrollado con la mentalidad de permitir utilizar la inteligencia artificial en dispositivos móviles y es por eso que brinda soporte a tres de las mayores plataformas como lo son Android, iOS y Raspbian. (*Tensorflow Mobile*).

El proceso para implementación en Android y iOS se lleva a cabo mediante el uso de librerías desarrolladas para esta finalidad y la configuración adecuada en el modelo generado en un computador y exportado como un archivo de datos. En el blog de Patrick Rodriguez se explica detalladamente algunas de las modificaciones que estos modelos tienen con respecto a la contraparte exclusiva para computador(Rodriguez 2017).

En el Git de Siraj Raval (Raval 2017) podemos encontrar un código bien detallado y un tutorial en formato de video sobre las principales funciones que permiten la implementación en Android. De igual manera podemos encontrar un demo implementación publicado por Tensorflow el cual puede ser modificado para hacer uso de cualquier modelo que necesitemos (Jarvis 2017)

Para el desarrollo de esta tesis se hace uso de la librería Tensorflow para la manipulación del modelo, permitiendo así la posibilidad de exportarlo a la plataforma Android. Con la librería Keras podemos modificar sus capas y entrenarlo de manera sencilla e intuitiva.

1.4. Descripción del problema a solucionar

La problemática de la contaminación de ríos ha llevado a los especialistas a explorar herramientas innovadoras para la evaluación de la calidad de los afluentes superficiales. Una de las alternativas que está tomando cada vez más relevancia en Panamá y otras partes del mundo, es el empleo de los macroinvertebrados como bioindicadores. Esto es debido a su amplia distribución, sedentarismo, sensibilidad a perturbaciones ambientales, largos ciclos de vida dentro del agua, gran tamaño y a que existen numerosos métodos de evaluación, como índices bióticos y de diversidad (Resh, David y Vh 1993).

El principal objetivo de este proyecto es el desarrollo de un software capaz de realizar el reconocimiento de animales macroinvertebrados utilizando solamente imágenes de los mismos. La iniciativa surge con la expansión de un proyecto en desarrollo por el Instituto Conmemorativo Gorgas de Estudios de la Salud (ICGES), institución interesada en el monitoreo de la población de macroinvertebrados presentes en las tomas de aguas por medio del proyecto "Biomonitoreo Participativo de la calidad del Agua con Juntas Administradoras de Acueductos Rurales (JAAR): Herramienta para sostenibilidad de los Recursos Hídricos en Panamá"; dirigido por la investigadora Aydée Cornejo (ICGES) en colaboración con la UTP.

El censo de las poblaciones de macroinvertebrados es de gran importancia como indicador de calidad de agua en ríos. En efecto, las diferentes especies de estos organismos presentan condiciones especializadas y debido a esto presentan una manera sencilla de evaluar la calidad ecológica del agua(Fabela y col. 2001). Basándose en la observación de estos organismos se puede realizar

un estudio ambiental en múltiples comunidades remotas lo cual, en contraste a alternativas más costosas como el uso de sensores, es un aporte significativo.

El sistema con el que cuentan actualmente hace uso de una serie de preguntas sobre la caracterización de los organismos logrando reducir las opciones de selección, facilitando la identificación del organismo. La propuesta es brindar a las comunidades esta herramienta en un dispositivo y por medio de un esquema cliente-servidor almacenar los distintos formularios, validarlos y analizar los datos generados en el instituto accediendo a un servidor web.

El problema de reconocimiento de macroinvertebrados basado en imágenes se ha estudiado anteriormente. En (Doyle 2009) se extraen diferentes características de las imágenes y se utilizan redes neuronales de una manera jerárquica y particionada, pero con la desventaja de seleccionarlas manualmente, lo cual conlleva de no realizarse correctamente. En (Sarpola y col. 2008) se propone un sistema con reconocimiento de patrones en 2-D y 3-D utilizando ciertos equipos especializados incluyendo sensores infrarojos y un sistema de clasificación binario (solamente se comparaban dos clases).

También en (Tirronen y col. 2009) se hace una aproximación, esta vez utilizando la técnica de «Support Vector Machine - SVM», la cual es una representación de los datos como puntos en el espacio, esparcidos de una manera que facilite una clasificación binaria. Sin embargo en este último algoritmo cuentan con algunas carencias, destacando algunas de sus mayores desventajas en la escogencia de los híper-parámetros y la función kernel, que de no ser correctos afectan negativamente el desempeño del modelo. Debido a la complejidad y a lo difícil que es encontrar la mejor configuración se suelen probar diferentes valores y realizar pruebas hasta encontrar una configuración con

resultados satisfactorios. Por último la computación del algoritmo aumenta de manera lineal al agregar nuevas clases para identificación, porque cada clase adicional requiere del entrenamiento de un nuevo clasificador debido a la naturaleza binaria del algoritmo.

1.5. Relación de la calidad del agua y los macroinvertebrados

Los indicadores biológicos son una herramienta útil, rápida y de bajo costo que pude ser utilizada en programas de monitoreo rutinario. Es posible obtener información que integra elementos de corto plazo y responden rápidamente a las variaciones ambientales coincidiendo con los análisis químicos (Fabela y col. 2001).

El concepto de especie indicadora es de fundamental importancia en el monitoreo biológico al definirse como aquel organismo que tiene requerimientos particulares de variables físicas o químicas tales, que cambios en su presencia o ausencia, número, morfología, fisiología o conducta, indican que las condiciones físicas y químicas del agua, así como biológicas, están fuera de sus límites aceptables.

Los organismos indicadores ideales son aquellas especies que tienen tolerancias ambientales estrechas específicas y que con su presencia reflejan la calidad de su ambiente. Por el contrario, los organismos que tienen una amplia tolerancia a diferentes condiciones ambientales no son buenos indicadores (Resh, David y Vh 1993).

En efecto, las diferentes especies de estos organismos presentan condiciones especializadas y debido a esto presentan una manera sencilla de evaluar la calidad ecológica de estas aguas (Fabela y col. 2001). Utilizando como base la observación de estos organismos se puede realizar un estudio ambiental en múltiples comunidades remotas lo cual, en contraste a alternativas más costosas como el uso de sensores, es un aporte significativo. Esta metodología de biomonitoreo ofrece múltiples ventajas para la vigilancia rutinaria de la calidad del agua en las cuencas y ríos en general, ya que son simples, de bajos costos de aplicación, además de que los resultados se obtienen con gran rapidez y con una alta confiabilidad.

Los organismos son los primeros en detectar las modificaciones que se generan en su entorno producto de la contaminación y el uso de procedimientos de análisis de compuestos brindan información de la concentración de los diferentes agentes contaminantes.

Para la recolecta de la comunidad de macroinvertebrados se seleccionan estaciones de monitoreo específicas y se realiza un análisis del conteo de los macroinvertebrados identificados.

1.5.1. Especies de macroinvertebrados

En la región de Panamá existen una gran cantidad de macroinvertebrados acuáticos de interés, el ICGES propone para su estudio 81 familias. Algunos ejemplos pueden ser vistos en la figura 1.5.

De acuerdo a las condiciones necesarias para los macroinvertebrados se puede cuantificar en una escala de tolerancias. Uno de los índices más usados

"Calopterygidae" "Heptageniidae" "Perilestidae" "Gomphidae"

FIGURA 1.5: Algunos macroinvertebrados pertenecientes al proyecto (*Libélulas*).

es el BMWP (Biological Monitoring Working Party), desarrollado para Gran Bretaña y que ha sido adaptado a varios países, entre ellos Costa Rica (BMWP-CR), Colombia (BMWP-COL) y Panamá (BMWP-PAN).

El índice BMWP asigna puntuaciones a las familias de macroinvertebrados, de 10 a 1. Las familias que habitan en zonas no perturbadas se clasifican con 10; a aquellas que tienen un rango de tolerancia más amplio, se les asigna valores de 8 a 4 y los números 3, 2 y 1 indican familias de aguas contaminadas.

En nuestro estudio se tomaron 10 familias de conchas elegidas debido a que engloban un conjunto de macro invertebrados muy beneficioso para la medición del coeficiente de contaminación de los ríos. En la tabla 1.1 se muestra un extracto mostrando las familias de conchas utilizadas en el proyecto.

Familia	Puntaje
Ampullariidae	8
Ancylidae	4
Corbiculidae	7
Hydrobiidae	5
Lymnaeidae	5
Neritidae	3
Physidae	3
Planorbidae	3
Sphaeriidae	7
Thiaridae	4

TABLA 1.1: Valores de tolerancia según BWMP-PAN de las 10 familias elegidas (Cornejo y col. 2017).

Capítulo 2

Técnicas y Algoritmos

Para solucionar el problema debemos diseñar un modelo de redes neuronales. Este es conformado por una serie de componentes ordenados a manera de grafo y en este capítulo se explicarán esos componentes que le conforman.

El modelo parte como una modificación del algoritmo «Inception-v3», el cual hace uso de los módulos «inception», estructuras de pequeño tamaño conformada por segmentos de redes convolucionales y operaciones de «pooling». Este modelo es sacado de los repositorios de Tensorflow con sus valores preparados para el concurso ImageNet 2012, mediante un proceso conocido como transferencia de conocimiento, para posteriormente ser modificado en sus capas finales y entrenado para cumplir con la tarea de identificar los macroinvertebrados.

Para comprender el funcionamiento de este modelo y las redes neuronales convolucionales en general, hace falta comprender los conceptos básicos que conforman las redes neuronales y a la vez es necesario describir la base matemática presente en el perceptrón y los algoritmos de entrenamiento como «softmax», «cross entropy», «gradient descent» y los optimizadores de entrenamiento.

2.1. Redes neuronales

Las redes neuronales artificiales son sistemas inspirados en las estructuras neuronales biológicas de los cerebros animales. Estos sistemas son capaces de aprender y realizar tareas.

Este sistema se basa en la colección de unidades conocidas como neuronas artificiales que se conectan y transmiten mensajes entre ellas. Cada conexión es una sinapsis y posee un modelo matemático conformado por coeficientes (ver figura 2.1).

Para que este modelo pueda funcionar correctamente es necesario un proceso de entrenamiento. Este es posible mediante el algoritmo de retropropagación. Este consiste en aplicar algoritmos de optimización a los coeficientes de las conexiones entre unidades, partiendo desde la capa de salida hasta llegar a la capa de entrada.

2.1.1. Componentes de las redes neuronales

Para comprender la estructura de una red neuronal es beneficioso comprender la base matemática de sus componentes. Es por eso que debemos remontarnos a los años 1950s y 1960s con el surgimiento del perceptrón y comprender la evolución que ha tenido.

Perceptrón

Las redes neuronales de hoy en día preserva muchas de las características del perceptrón. A continuación describiremos su base matemática y las limitaciones que sufre.

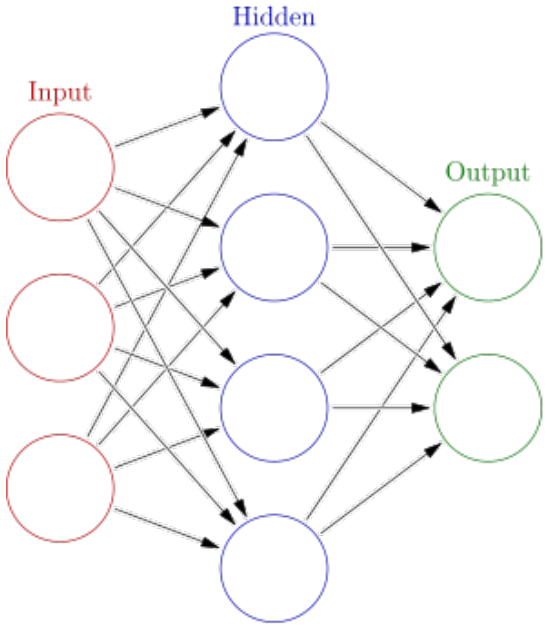


FIGURA 2.1: Esquema de una red neuronal con 3 capas: capa de entrada, capa de salida (objetivos) y la capa oculta. Los círculos son las unidades y las flechas son los coeficientes matemáticos. (*Colored_neural_network*)

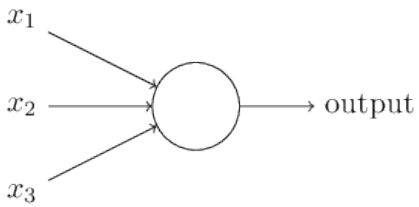


FIGURA 2.2: Diagrama de una neurona artificial («Perceptron Picture»).

El funcionamiento de un perceptrón es sencillo, toma entradas x_i y produce una salida (ver figura 2.2). Para calcular el valor de salida se propuso la función de regresión lineal. La regresión lineal es una aproximación sencilla del aprendizaje supervisado. El modelo de regresión lineal es utilizado para predecir un valor salida de acuerdo a una o más variables x. Este modelo hace una suposición en que existe una relación entre estos valores.

$$z = \beta_0 + W_0 x_0 + W_1 x_1 + ... + W_n x_n \tag{2.1}$$

$$z = \beta_0 + \sum_{i=0}^{M} W_i x_i \tag{2.2}$$

En la ecuación, z corresponde a la predicción; W_i a los coeficientes del modelo; y 'β_0' es el interceptor o también conocido como tendencia o «bias», valor independiente de la x que ofrece un mejor ajuste al modelo al ser constante. W_i y β_0 son aprendidos durante el proceso de entrenamiento.

Para hacer uso del perceptrón debemos evaluar z como una función con valores positivos o negativos, esto permite una clasificación binaria.

Regresión logística

En las redes neuronales solo una función no lineal permite que estas puedan computar problemas no triviales como X-OR. Estas funciones determinan la activación de las neuronas, decidiendo si la información que contienen debe ser utilizada o simplemente ignorada.

Una red neuronal sin una función de activación solo puede resolver problemas lineales como AND, OR y NOR. No importa con cuantas capas cuente el modelo, al sumar esas capas lineales solo es posible resultar con otra función lineal.

Una función no lineal que soluciona este problema es la función sigmoide. Estas neuronas sigmoides son utilizadas para describir datos y predecir la probabilidad de que un determinado valor pertenezca o no a una clase. Es utilizado para hacer una clasificación binaria principalmente. Hace uso de la función sigmoide, dada por la fórmula descrita en 2.3, en donde z es la función 2.1:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

La forma que provee la función sigmoide agrega no linealidad (ver figura 2.3). La neurona sigmoide es un perceptrón mejorado, con bordes suaves, este es el factor fundamental para el éxito de estas neuronas, ya no contamos con una salida binaria, sino un valor de punto flotante que brinda flexibilidad al modelo.

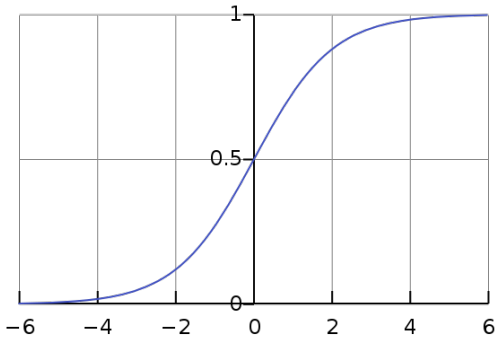


FIGURA 2.3: Función sigmoide. Esta figura debe tener un punto medio en 0,5 para asignar la determinación de la clase 0 o 1. De Wikimedia Commons.

Perceptrón multicapa

El perceptrón multicapa está conformado de por lo menos tres capas. En la parte inicial tenemos la capa de entrada y en la parte final la capa de salida. La capa en medio es conocida como capa oculta debido a que estas no proveen información que nos interese saber y son para uso de la misma red neuronal.

El perceptrón multicapa en verdad puede tener más de una capa oculta (ver figura 2.4). Aunque parezca confuso, el perceptrón multicapa en verdad es conformado por neuronas sigmoides en vez de perceptrones.

Cuando hablamos de redes neuronales profundas solo nos referimos a modelos que cuentan con al menos tres capas ocultas presentes (Bengio y Others 2009), con esto podemos afirmar que el perceptrón multicapa es parte de la tecnología «deep learning».

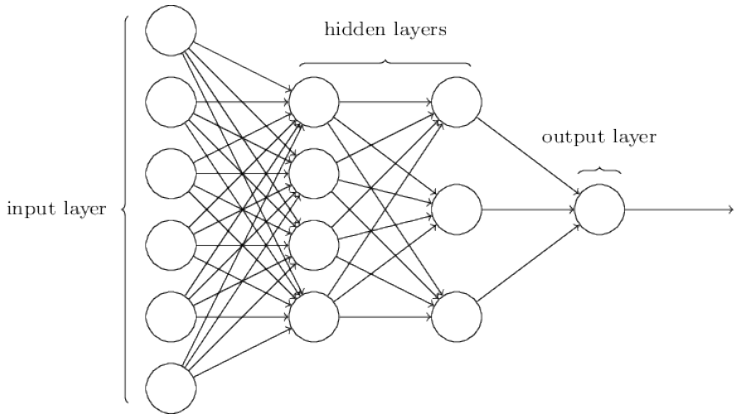


FIGURA 2.4: Ejemplo de perceptrón multicapa (*Multilayer perceptron*).

Regresión logística multiclase

La función sigmoide es fundamental para agregar no linealidad al modelo, pero sigue siendo una función de respuestas binarias, es por eso que es necesario aplicar en la capa final del modelo una versión modificada con capacidad multiclase.

La función Softmax es precisamente una generalización del algoritmo de la función sigmoide siendo un clasificador logístico multiclase. Esta es su ecuación:

$$S(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{2.4}$$

El valor obtenido de la función Softmax S es equivalente a la distribución de probabilidad categórica y nos permite descifrar la probabilidad de cualquier clase de ser cierta. Es utilizada en las capas de salida de nuestro modelo de redes neuronales. Softmax puede ser utilizado para cualquier cantidad de clases desde los cientos hasta los miles y esto lo hace especialmente útil para

nuestro problema de reconocimiento de macroinvertebrados.

2.1.2. Algoritmos y funciones de entrenamiento

Para el funcionamiento de estos modelos neuronales es necesario un proceso de entrenamiento. Para esto existe ciertos algoritmos que permiten precisamente este proceso.

En regla básica para lograr el entrenamiento es necesario calcular el desempeño del algoritmo y luego realizar las modificaciones que se deben hacer a los coeficientes *weigth* y *bias* de cada neurona.

Función error de la entropía cruzada

La entropía cruzada es un método comúnmente usado para cuantificar la diferencia entre dos distribuciones probabilistas p y q. La distribución veraz p se presenta en un formato de distribución codificada «one-hot», el cual es una forma de representar etiquetas mediante un arreglo de valores binarios, donde el valor 1 indicará la etiqueta dependiendo de su posición en el arreglo (ver tabla 2.1). Y otra distribución q es generada por el modelo en valores flotantes entre 0 y 1.

La proximidad entre la distribución predicha q y la veraz p es lo que se conoce como la función error. Para calcularla se debe hacer uso de la siguiente ecuación:

$$H(p,q) = -\sum_{i} p_i \log(q_i) \tag{2.5}$$

Nombre	Etiqueta Categórica	Etiqueta one-hot
Elefante	0	100
Gato	1	010
Perro	2	001

TABLA 2.1: Equivalencia de etiquetas categóricas y one-hot.

Esa ecuación nos brinda el error de solamente un par de distribuciones, pero para nuestro proceso de entrenamiento un promedio de varios pares de distribuciones es mucho más conveniente, para eso se modifica la ecuación a la siguiente:

$$L = \frac{1}{N} \sum_{n=1}^{N} H(p_n, q_n) \tag{2.6}$$

Ecuación donde L representa el valor «loss» o error de nuestro modelo. Este valor es de gran utilidad para el proceso de entrenamiento, pues determinará en gran medida como se harán las modificaciones en los valores W y β.

Algoritmo de optimización del gradiente descendiente

Los métodos algorítmicos desarrollados en «machine learning» aseguran su eficiencia a través del uso de optimizaciones. En esta sección se explica uno de los algoritmos más sencillos para entender e implementar. Este es la base de otros algoritmos más complejos.

El algoritmo del gradiente descendiente permite la búsqueda de valores de los coeficientes de una función con el objetivo de disminuir la función costo. Este algoritmo es el que nos permite la optimización de los vectores W y β.

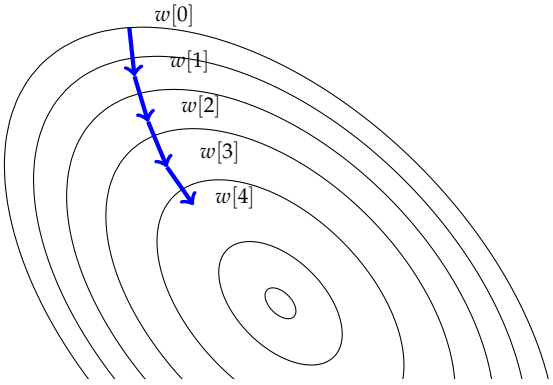


FIGURA 2.5: El gradiente descendiente sigue el valor negativo de el gradiente en su posición actual a medida que describe la dirección de el punto más bajo. El ratio de aprendizaje α describe al ser multiplicada con el gradiente el tamaño del desplazamiento en cada iteración.

Es posible observarlo como un plato hondo, donde la superficie del mismo es un plano cartesiano de la función costo donde el centro es el punto más bajo. Al iniciar el proceso de entrenamiento nos encontraremos en una posición aleatoria de la superficie del plato y el función costo será definida por la distancia hasta el centro. Al repetir el procedimiento suficientes veces nos llevará al fondo del plato logrando el costo mínimo (ver figura 2.5).

Para encontrar los nuevos coeficientes de W y β primero debemos utilizar la derivada de la función costo. El tamaño de cada salto depende el valor de la función costo, mientras que la dirección es determinada por el signo de la pendiente. Una pendiente positiva indica que los coeficientes se están alejando del valor óptimo y se debe hacer el salto en la dirección contraria, si es negativa se debe seguir con la misma dirección.

En conjunto con un coeficiente α que indica la tasa de aprendizaje. Seleccionar la tasa correcta es fundamental, un coeficiente elevado nos puede hacer pasar el mínimo, mientras que una muy pequeña hará que se necesiten más

iteraciones.

Estos valores conforman la ecuación que determina el nuevo valor de los coeficientes W y β.

$$step = -\alpha \Delta L(W_1, W_2, ..., W_N) \tag{2.7}$$

$$step_W = -\alpha \Delta \frac{1}{N} \sum_{n=1}^{N} H(p_n, q_n) \tag{2.8}$$

Retropropagación

La retropropagación del error es un algoritmo del aprendizaje supervisado de las redes neuronales que hace uso del gradiente descendiente. El nombre surge de que los cálculos del gradiente proceden en retroceso a través de la red neuronal, empezando con el gradiente de la última capa hasta llegar a la primera (Rumelhart, Hinton y Williams 1986).

El cálculo parcial del cómputo del gradiente de la última capa es reutilizado en la computación de la capa anterior. Este flujo en retroceso permite una computación eficiente del gradiente en cada capa frente a hacer cada capa de manera separada.

El algoritmo es muy popular hoy en día a pesar de tener sus orígenes en los años 80's. Es considerado un algoritmo eficiente e implementaciones modernas sacan provecho de procesadores GPU permitiendo máximo rendimiento.

El algoritmo cuenta con tres fases:

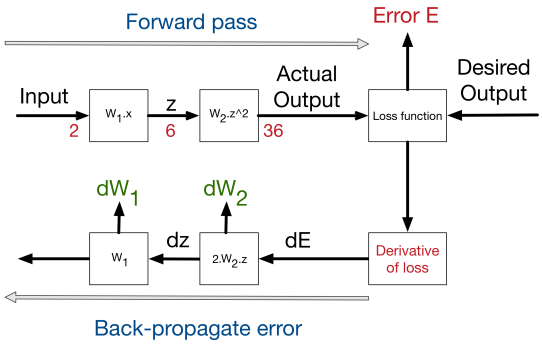


FIGURA 2.6: Diagrama de retropropagación.

1. Fase hacia adelante. Se calculan los valores individuales de cada capa de neuronas en manera progresiva desde la capa inicial hasta la capa final y se calcula la función error.

2. Fase hacia atrás. Se hace uso de la derivativa de la función de error y obtener el gradiente. Luego se calcula el nuevo valor de coeficientes W y β sobre la derivada de sus respectivas funciones.

3. Actualización de los coeficientes. De acuerdo a la tasa propuesta por α se hacen las modificaciones pertinentes a la W y la β.

2.2. Redes neuronales profundas

Las redes neuronales profundas son de extrema importancia en la generación de modelos de «machine learning» de hoy en día. Una sucesión suficientemente larga de la cadena da origen a la profundidad del modelo que es donde surge el nombre de «deep learning».

Muchos de los conceptos presentes en «deep learning» ya existían en los años 1980s y 1990s, pero la diferencia radica en que hoy en día se disponen de datasets masivos y computadores GPU.

Los algoritmos son poderosos pero requiere de grandes cantidades de datos y procesamiento, y finalmente en la actualidad podemos contar con ambos elementos. La tecnología de hardware juega un papel importante permitiendo la generación de modelos más complejos, grandes y robustos.

Otro factor importante es el desarrollo de la función de activación ReLu, la cual alivia el problema del desvanecimiento del gradiente, permitiendo la construcción de modelos mucho más profundos.

La suma de estos factores ha provocado un profundo interés en la comunidad científica lo que significa la creación de nuevos modelos, plataformas, técnicas y algoritmos de optimización.

2.2.1. Beneficios de una arquitectura profunda

Las redes neuronales se pueden incrementar en tamaño y complejidad en anchura o profundidad. En anchura se agregan más nodos a cada capa, mientras que en profundidad se agregan más capas al modelo. Resulta que un incremento a lo ancho no es eficiente, el incremento de parámetros no justifica los beneficios obtenidos. Es por eso que se prefieren los incrementos en profundidad. Esto sucede debido a un fenómeno jerárquico que los modelos profundos captan de manera natural. Si visualizamos los resultados entre capas de nuestro modelo podemos apreciar que en las primeras capas el modelo capta elementos sencillos como líneas y bordes, una vez que se avanza

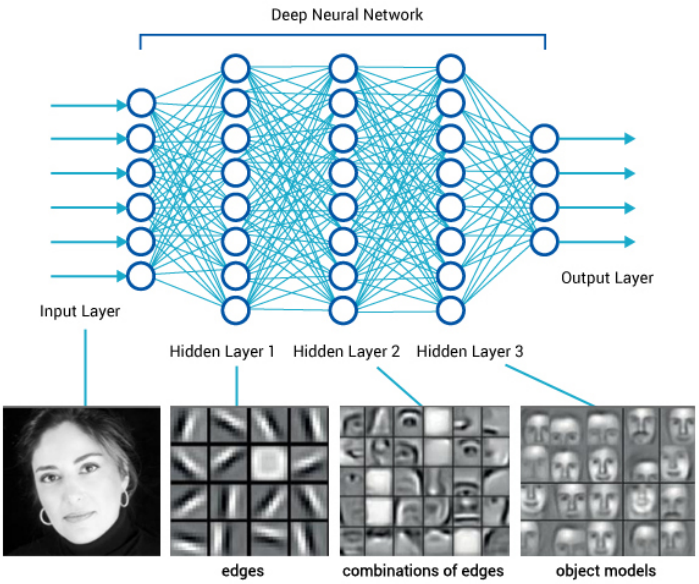


FIGURA 2.7: Esquema de abstracción de las redes neuronales profundas («Hierarchy CNN»).

en la red se pueden apreciar formas más complejas como figuras geométricas y de seguir se pueden empezar a ver objetos complejos (ver figura 2.7).

2.2.2. Nuevos algoritmos

ReLu

Una red neuronal sin una función de activación solo puede resolver problemas lineales. No importa con cuantas capas cuente el modelo, al sumar esas capas lineales solo es posible resultar con otra función lineal.

Algunas de las funciones de activación más comunes fueron la función de activación sigmoide (ver figura 2.3) y la tangente hiperbólica tanh x en un rango de -2 a 2. La función sigmoide sufre de varios problemas el desvanecimiento del gradiente, dificultades para optimizar y baja convergencia; mientras que

la función hiperbólica soluciona estos problemas menos el desvanecimiento del gradiente.

En cambio la función ReLu se desenvuelve de mejor manera brindando beneficios adicionales como una computación más eficiente y una convergencia más rápida al compararla con otras funciones. Esta función debe utilizarse exclusivamente entre las capas ocultas de nuestro modelo de redes neuronales.

No es de extrañarse que la función ReLu se haya hecho muy popular en los últimos años. Su ecuación es sencilla $R(x) = max(0, x)$, esto significa que se deshace de los valores negativos, una función simple y eficiente.

Optimización del gradiente descendiente

Basado en el algoritmo del gradiente descendiente han surgido una gran variedad de ecuaciones que aportan mejoras. En general los algoritmos deben definir una tasa de aprendizaje dinámica que se ajuste gradualmente para lograr saltos más largos cuando es necesario y reducirlos cuando ya no es necesario. También es común reducir el tamaño del conjunto a evaluar, eligiendo solamente una fracción del conjunto de entrenamiento aleatoriamente, para hacer cálculos más rápidos aunque ligeramente más imprecisos de los saltos.

Otra variación incluye el uso del método de momento, donde este valor almacena la diferencia entre los W de cada iteración y determina un valor que suavice el desplazamiento entre iteraciones.

Algoritmos más complejos como AdaGrad (Duchi, Hazan y Singer 2011), RMSProp (Tieleman e Hinton 2012), Adam (Kingma y Ba 2017), kSGD (Patel 2016), entre otros se ofrecen en las librerías Tensorflow y Keras, listos para una fácil implementación. Estos ofrecen una convergencia de la gráfica de

aprendizaje más rápida, una mayor robustez ante mínimos o máximos locales y un mayor control híperparámetros para configurar el modelo.

2.2.3. Técnicas de regularización

Uno de los problemas centrales en «machine learning» es lograr que los algoritmos se desenvuelvan bien. No solo en la base de datos de entrenamiento, sino en campo con nuevas fuentes de datos. Algunas de las estrategias que se siguen son la creación de una base de datos de pruebas a expensas de una reducción en la de entrenamiento.

Las estrategias que se utilizan para lograr este objetivo se llaman regularización y existen una serie de estos disponibles producto de muchas de las grandes investigaciones en el campo.

La regularización consiste en aplicar restricciones artificiales a nuestros algoritmos, lo que de manera implícita reduce el número de parámetros libres. Esto permite que el modelo sea preciso, pero sin caer en el sobreajuste.

En esta sección se hace una introducción a los conceptos como el sobreajuste, la sobregeneralización, la tendencia, la varianza y algunos de las principales técnicas utilizadas.

Terminación Temprana

El primer método para prevenir el sobreajuste es simplemente detener el entrenamiento inmediatamente se detienen las mejoras en la base de datos de validación.

Es un método sencillo que deja en claro que más entrenamiento no siempre es mejor, es necesario analizar correctamente los resultados que ofrece nuestro modelo.

Regularización L_2

La regularización L_2 surge en la optimización o cálculo de errores mínimos cuadráticos, su implementado en redes neuronales forma parte de los algoritmos originarios de «deep learning» (Ng 2004). La idea es agregar un nuevo término a la función error de entropía cruzada, esto es logrado al agregar la norma L_2 al parámetro W y luego multiplicándolo por una constante. La nueva ecuación queda así:

$$L_2 = L + \beta \frac{1}{2} \|W\|_2^2 \tag{2.9}$$

Dropout

Otra técnica importante de regularización es llamada «dropout», surge recientemente y tiene un rendimiento muy bueno (Srivastava y col. 2014). Dropout es un método de regularización poderoso de bajo costo. En 2014 Szegedy y su equipo presentaron una red neuronal en el ILSVRC con seis capas «dropout» que le permitieron la victoria. Sin embargo utilizar esta técnica en exceso puede causar efectos adversos (Szegedy y col. 2014).

La técnica es utilizada al ser colocada en medio de dos capas de neuronas. Los valores que van de una capa a la otra se llaman activaciones, ahora «dropout» selecciona de manera aleatoria un porcentaje de estas activaciones y las convierte en valores cero, destruyendo su información (ver imagen 2.8).

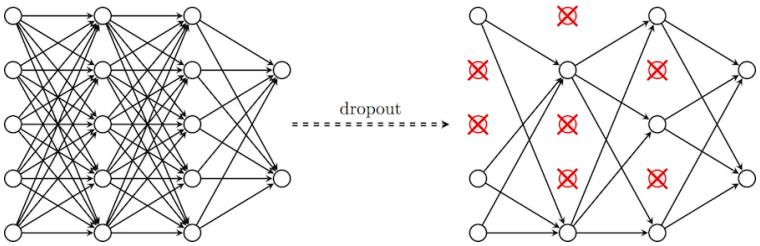


FIGURA 2.8: A la izquierda red neuronal sin alteraciones, a la derecha luego de aplicarse la técnica «dropout» («DropoutDiagram»).

Esta técnica permite que la red neuronal nunca dependa de un activación o característica específica permitiendo una mayor generalización en los modelos.

2.3. Redes neuronales convolucionales

Conocidas como «CNNs» por sus siglas en inglés (Convolutional Neural Networks), son una red especializada de redes neuronales para el procesamiento de datos en forma de malla. Pueden ser mallas de 1 dimensión como las muestras de una sucesión en un intervalo de tiempo, o mallas de 2 y 3 dimensiones como imágenes (LeCun y col. 1989). El nombre proviene de la operación matemática llamada convolución, la cual es un tipo especializado de operación lineal.

Las redes convolucionales han jugado un papel muy imporante en la historia del «deep learning». Son una de las primeras aplicaciones de modelos profundos en obtener resultados satisfactorios. También las redes convolucionales son unas de las primeras en lograr resolver problemas para aplicaciones comerciales. Por ejemplo en los años 1990s un grupo de investigaciónñ de

AT&T desarrollo un modelo para la lectura de cheques (LeCun y col. 1998b). Para finales de la década este sistema procesaba un 10 % de todos los cheques en Estados Unidos. Posteriormente una serie de sistemas OCR y de reconocimiento de manuscrito fueron desarrollados por Microsoft («Transformation invariance in pattern recognition: Tangent distance and propagation»).

Las redes neuronales se especializan en trabajar con datos de tipo bidimensional como imágenes, aunque también pueden trabajar con unidimensionales como secuencia de datos o tridimensionales como videos, pero para la primera existe una especialización mucho mas poderosa conocida como redes neuronales recurrentes, mientras que para videos se han realizado pruebas con modelos híbridos entre redes convolucionales y recurrentes (Xu, Hu y Deng 2016).

En esta sección describiremos el concepto de convolución, la motivación para su uso, el uso en conjunto de la operación «pooling» y la estructura que las redes neuronales convolucionales normalmente adoptan.

2.3.1. Operación de convolución

Una convolución es una integral que expresa la cantidad de solapamiento de una función g sobre otra función f. Se puede decir que combina una función con otra. Matemáticamente es expresado como:

$$s(t) = \int_0^t f(\tau)g(t-\tau)d\tau \tag{2.10}$$

La operación de convolución es escrita con un asterisco:

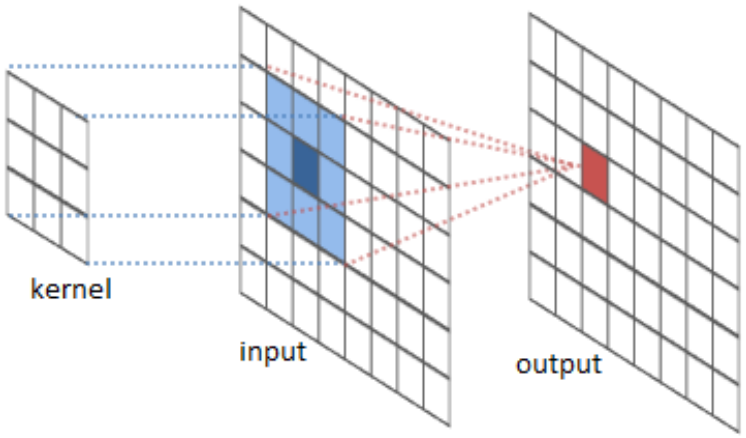


FIGURA 2.9: A la izquierda kernel aplicado sobre matriz entrada sobre un punto (x,y) para producir uno de los valores de la matriz salida («ConvolutionDiagram»).

$$s(t) = (x * w)(t) \tag{2.11}$$

La convolución presenta tres elementos que ayudan en los sistemas de «machine learning»: dispersión, parámetros compartidos y representaciones equivariantes.

Tradicionalmente las capas de redes neuronales usan multiplicación de matrices, donde cada unidad de entrada está conectada con cada unidad de salida. En las redes convolucionales sin embargo hay interacciones dispersas, esto se logra al generar un kernel más pequeño que el valor de entrada. Por ejemplo al procesar una imagen, esta puede contener miles o millones de pixeles, pero para la detección de características como los bordes solo se necesitan cientos de pixeles, esto significa que debemos almacenar menos parámetros reduciendo los requerimientos de memoria del modelo y mejorando la eficiencia computacional. Estas mejoras son significativas, al tener m valores de

entrada y n valores de salida el algoritmo será $O(m \times n)$, mientras que si lo limitamos a el tamaño de un kernel k solo requeriremos $O(k \times n)$ operaciones.

Los parámetros compartidos hacen referencia a la reutilización de los parámetros en más de una función del modelo. En una red neuronal tradicional cada nodo de sus capas es usado una sola vez durante el cómputo de los valores de salida, sin embargo en la red convolucional cada parámetro del kernel es utilizado múltiples veces. Esto significa que en vez de aprender parámetros separados para cada ubicación de una imagen, solo se aprende un conjunto. Es por esto que las operaciones convolucionales son mucho más eficientes en términos de memoria que la multiplicación de matrices.

En el caso de la convolución, la propiedad de parámetros compartidos causa a la capa una nueva propiedad llamada equivarianza al desplazamiento. Una función equivariante significa que si los valores de entrada cambian los de salida cambian de igual manera. Si movemos el objeto en la función de entrada, su representación se moverá de la misma manera en la salida. Esto es útil por ejemplo en la detección de bordes de una imagen, los mismos bordes aparecen más o menos de la misma manera en toda la imagen. Sin embargo hay casos donde no deseamos esta equivarianza, por ejemplo al detectar las cejas y la barbilla de un rostro, deseamos que estas estén presentes en lugares específicos de nuestra imagen.

2.3.2. Pooling

Una capa típica utilizada en modelos de redes neuronales convolucionales es el «pooling». Usualmente en los modelos se cuenta con una etapa inicial

donde se realizan una serie de convoluciones para producir una serie de activaciones lineales. En la segunda etapa cada activación lineal es procesada por activaciones no lineales como funciones rectificadoras (ReLU, tanh, sigmoide) y en la tercera etapa se tienen las funciones «pooling» que permiten modificar las capas de salida un poco más.

Una función «pooling» reemplaza los valores de salida de una red con una versión resumida, por ejemplo, «max pooling» obtiene los valores máximos dentro de una región definida como vecindario. Otras funciones comunes utilizan la función promedio, normalización L^2 o el promedio basado en distancia al punto central del vecindario.

La capa «pooling» ayuda a crear una representación invariante a la traslación. Por ejemplo para determinar si una imagen contiene un rostro no necesitamos saber exactamente donde están ubicados los ojos, solo necesitamos saber habrá un ojo a la izquierda y otro a la derecha del rostro.

Como el «pooling» genera una versión resumida de las capas es posible mejorar la eficiencia computacional de la red, generando capas que son k veces más pequeña, que ocupan menos memoria y son más sencillas de procesar al reducir la cantidad de parámetros.

2.3.3. Redes convolucionales y la neurociencia

Las redes convolucionales son probablemente el mayor éxito de inteligencia artificial inspirado por la biología. La historia de las redes convolucionales inicia con experimentos en neurociencia cuando los científicos David Hubel y Torsten Wiesel colaboraron para determinar como funciona la visión en los

mamíferos (Hubel y Wiesel 1959; Hubel y Wiesel 1962; Hubel y Wiesel 1968). Durante un experimento donde registraron el funcionamiento neuronal de gatos observaron como responden a imágenes proyectadas en una pantalla frente al animal. El gran descubrimiento fue cuando las secciones iniciales de la visión reaccionaban fuertemente a patrones específicos como barras, pero al cambiar los patrones su respuesta disminuía.

De manera simplificada podemos enfocarnos en la parte del cerebro conocida como corteza visual primaria o V1. Este es el primer área del cerebro que ejecuta acciones de reconocimiento visual. En un esquema simplificado podemos describir cómo las imágenes son formadas por la luz recibida en los ojos estimulando la retina, la cual tiene un tejido sensible a la luz. Las neuronas en la retina hacen algunas transformaciones sencillas a la señal pero sin alterar significativamente la forma. Luego la imagen pasa por una región llamada núcleo lateral genicular o «Lateral Genicular Nucleous (LGN)», cuya principal función es transportar la señal desde el ojo hasta la corteza visual primaria ubicada en la parte posterior del cerebro.

La corteza visual primaria contienen células especializadas con tres características definidas que fueron capturadas en el diseño de las redes neuronales convolucionales:

1. Su estructura bidimensional que refleja la estructura de imagen en la retina. Por ejemplo la luz recibida en la parte superior de la retina solo afecta a las neuronas correspondientes en V1.

2. Contiene células simples. Una célula es capaz de reconocer patrones en el campo receptivo de una manera muy similar a como lo hacen las

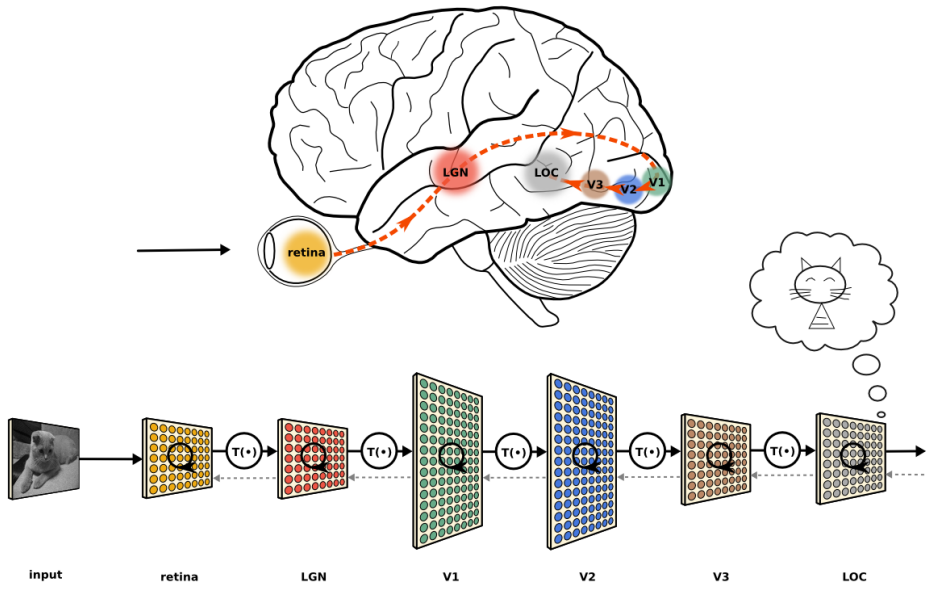


FIGURA 2.10: Comparación de cerebro humano y modelo de red neuronal. («VisualCortexDiagram»).

convoluciones.

3. Contiene células complejas. Estas células responden de manera invariante a pequeños cambios de posición de la característica, también son invariantes a la luz. Podemos ver estas características en las capas «pooling».

En el cerebro existen varias capas nombradas numéricamente desde V1 hasta V5 que se encargan de la detección de diferentes elementos y al ser combinados pueden identificar un elemento (ver figura 2.10).

2.4. Arquitecturas

En la actualidad se han desarrollado una gran variedad de modelos basados en las operaciones convolucionales para el reconocimiento de imágenes. Algunos de estos modelos son:

AlexNet. Uno de las primeras arquitecturas profundas y pionera de las redes neuronales profundas al hacer uso de la tecnología GPU para el entrenamiento superó con creces cualquier prueba de visión por computadora de la fecha.

VGG. Se caracterizó por su estructura piramidal. Es bastante poderosa, pero es lenta de entrenar, lo que aún con buen hardware GPU puede tomar semanas para lograr resultados decentes.

GoogLeNet. Haciendo uso de los módulos Inception, esta arquitectura probó ser realmente poderosa y eficiente computacionalmente. Con la arquitectura se introdujo un sistema multicapas en paralelo, brindando más herramientas al modelo para solucionar el problema.

ResNet. Una arquitectura gigante que redefine el término profundo en las redes neuronales profundas. Consiste en una secuencia de módulos residuales, que pueden estar entre los cientos hasta los miles de capas sin necesariamente reducir el desempeño.

ResNeXt. Es una combinación de los módulos inception y la arquitectura de módulos residuales. Pertenece a las investigaciones en el estado del arte.

SqueezeNet. Es una de las arquitecturas más poderosas en escenarios con bajo ancho de banda como en las plataformas móviles. La arquitectura apenas ocupa 4.9MB de espacio, a comparación de GoogleNet que ronda en

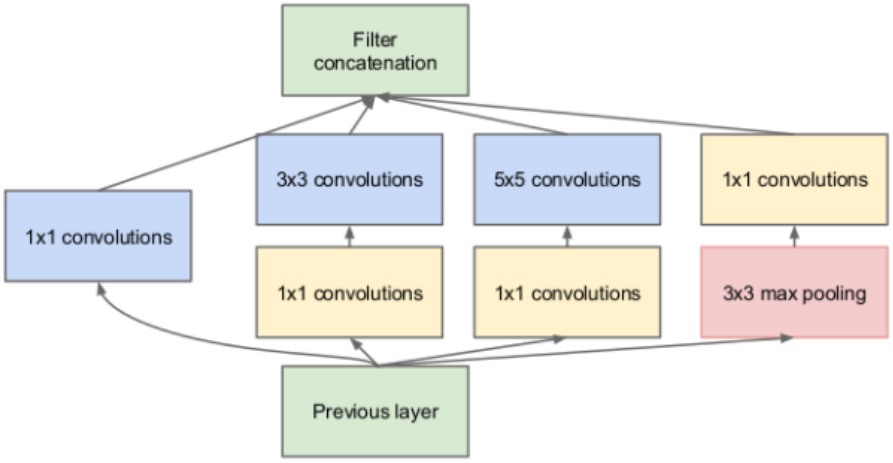


FIGURA 2.11: Diagrama de capas del módulo Inception (Szegedy y col. 2014).

los 100MB. Este cambio se le logra gracias a los módulos fuego.

2.4.1. Módulos Inception

Para comprender el buen funcionamiento de GoogleNet, es necesario comprender la pieza fundamental que le conforma, el módulo Inception.

Los módulos Inception actúan como múltiples filtros aplicados a un mismo valor de entrada mediante capas convolucionales y de pooling. Esto permite sacar provecho de la extracción de patrones que brindan diferentes tamaños en los filtros. Luego el resultado de estos filtros es concatenado y utilizado como el valor de salida del módulo. Este modelo aumenta el número de parámetros entrenables y la computación requerida, pero mejora considerablemente la precisión (ver figura 2.11).

Todas las operaciones de convolución, incluyendo aquellas dentro de los módulos Inception hacen uso del rectificador ReLu.

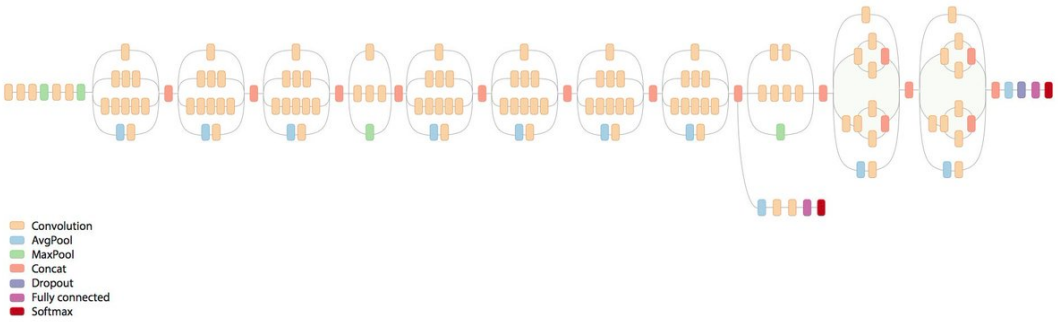


FIGURA 2.12: Esquema de entrenamiento y clasificación de Inception-v3, modelo mejorado del original GoogLeNet (Szegedy y col. 2015).

2.4.2. Estructura

GoogLeNet acepta como su campo de entrada imágenes de un tamaño 224×224 de color RGB, luego es reducido haciendo uso de capas convolucionales y pooling, para luego aplicar una sucesión de módulos Inception y culminar con una capa completamente conectada y la capa de predicciones «softmax» (ver figura reffig:inceptionv3).

2.4.3. Eficiencia del modelo Inception-v3

Una red neuronal más grande significa un incremento en el uso de memoria y la carga computacional. La red Inception fue diseñada para ser computacionalmente eficiente y tomando en cuenta equipos con recursos limitados de memoria.

Para hacer un cálculo del tamaño de memoria necesarios para entrenar una red neuronal es necesario hacer un conteo de los parámetros que están presentes en la red y multiplicarlos por la cantidad de espacio requerido por

el tipo de dato, aunque el valor real de memoria dependerá ligeramente de la implementación.

Memoria

La memoria que requiere un modelo se calcula en el conteo de parámetros de las diferentes capas en producto con el tipo de datos de cada uno. En nuestro modelo debemos considerar las capas convolucionales y las completamente conectadas.

El tamaño de una capa convolucional dependerá exclusivamente del tamaño del kernel y la profundidad de la capa anterior y actual. Esto puede ser calculado en la ecuación 2.12 donde *weights* es la cantidad de parámetros de los kernels en la capa convolucional, $depth_n$ es la profundidad de la capa actual, $depth_{n-1}$ la profundidad de la capa anterior y el ancho y alto del kernel son definidos por $kernel_w$ y $kernel_h$ respectivamente. Adicional en la ecuación 2.13 se calculan los parámetros de tendencia representados por el término *biases*.

$$weights = depth_n \times (kernel_w \times kernel_h) \times depth_{n-1} \tag{2.12}$$

$$biases = depth_n \tag{2.13}$$

Las capas «pooling» no presentan parámetros, pero las capas completamente conectadas son calculadas dependiendo del tamaño de los nodos. En la ecuación 2.14 y 2.15 se obtiene el tamaño de los parámetros para esta capa donde $outputs$ es el tamaño de la capa de salida y el término $inputs$ el tamaño

de la capa de entrada.

$$weights = outputs \times inputs \tag{2.14}$$

$$biases = inputs \tag{2.15}$$

Una vez calculado estos valores hacemos una sumatoria en todas las capas y obtendremos la cantidad de memoria necesaria para el entrenamiento con una imagen.

De manera ilustrativa podemos calcular la cantidad de parámetros de diferentes modelos como AlexNet (Krizhevsky, Sutskever e Hinton 2012), VGG-16 (Simonyan y Zisserman 2015), GoogLeNet (Szegedy y col. 2014) e Inception-v3 (Szegedy y col. 2015).

En AlexNet podemos contar aproximadamente 60 millones de parámetros (240 MB por imagen); siendo este un modelo de apenas 8 capas. Luego en VGG-16 podemos hallar cerca de 138 millones de parámetros (553 MB por imagen) en un modelo de 22 capas. En ambos modelos la mayoría de los parámetros son encontrados en las capas completamente conectadas.

Por otro lado GoogLeNet contiene apenas 5 millones de parámetros (20 MB por imagen) siendo la clave una reducción de las dimensiones al final de la fase de convoluciones para reducir la carga en las capas completamente conectadas. Igualmente Inception-v3 ofrece una reducción con respecto a AlexNet y VGG-16 con menos de 22 millones de parámetros (87 MB por imagen)

Complejidad Computacional

Es de gran utilidad estimar la complejidad computacional de una arquitectura a medida que es diseñada. Esta es determinada en las redes neuronales como las multiplicaciones necesarias en las capas completamente conectada y las capas convolucionales; ignorando operaciones no lineales, «dropout» y las capas de normalización por su reducido costo computacional.

Para el cálculo de la complejidad computacional una medida útil es la estimación del número de «FLOP (floating point operations)» (operaciones de punto flotante).

Para las capas completamente conectadas solo debemos calcular el número de parámetros *Weight* que se multiplican entre sí. La ecuación queda así:

$$FLOPs = parametros\_entrada \times parametros\_salida \tag{2.16}$$

Mientras que para las capas convolucionales el cálculo es más complejo y es necesario considerar el ancho, alto y profundidad de las capas que participan en la operación. La ecuación es la siguiente:

$$\begin{aligned} FLOPs(n) = {} & ancho\_capa_n \times altura\_capa_n \times ancho\_kernel_n \times \\ & altura\_kernel_n \times profundidad\_capa_n \times profundidad\_capa_{(n-1)} \end{aligned} \tag{2.17}$$

Como con el consumo de memoria podemos aplicar el mismo ejercicio de calcular la cantidad de FLOPs en los diferentes modelos.

En AlexNet tenemos 725 millones de FLOP, donde más de 650 millones

están en las capas convolucionales; en VGG-16 la computación aumenta mucho más, hasta los 15.8 billones de FLOP donde el peso de la computación está en las capas convolucionales totalmente. Por otro lado en GoogLeNet hace consumo de 2 billones de FLOP e Inception-v3 cuesta por imagen 6 billones de FLOP; ambos modelos muestran una mejoría con respecto a VGG-16 debido al uso inteligente de las convoluciones y la reducción de parámetros (Szegedy y col. 2014) (Szegedy y col. 2015).

2.5. Transferencia de conocimiento entre redes neuronales

Ahora bien, en práctica es muy difícil entrenar un modelo desde cero y esto se debe a que es difícil encontrar conjuntos de datos lo suficientemente grandes como para lograr buena precisión en las predicciones debido al sobreajuste que sufren las redes neuronales. Aquí es cuando debemos aplicar una técnica conocida como transferencia de conocimiento, esta se basa en el uso de modelos previamente entrenados (Oquab y col. 2014).

Como vemos en las redes neuronales en las primeras capas obtenemos características de bajo nivel como los bordes para al final capturar las de alto nivel. Al utilizar modelos previamente entrenados aprovechamos las características de bajo nivel y resolvemos el problema del sobreajuste; a la vez reducimos la carga de entrenamiento que para los modelos más complejos requieren de mucha computación.

En este trabajo se creó una instancia de la red convolucional Inception-v3

(Szegedy y col. 2016) junto con sus valores de entrenamiento usados para la competencia ImageNet, donde se deben clasificar las imágenes en 1000 diferentes clases. Para nuestro estudio se reemplazó únicamente la capa completamente conectada con una que conecte solo a nuestras categorías y se procede a entrenar exclusivamente esa capa final bloqueando el acceso de escritura en las capas inferiores.

2.6. Ajuste Fino

El proceso de ajuste fino o «fine tuning» por otro lado consiste en aprovechar y realizar pequeñas modificaciones a los valores del modelo al modificar capas más internas. Por ejemplo durante el proceso de transferencia de conocimiento se bloquearon las capas internas, en ajuste fino se desbloquean n capas y se procede al entrenamiento logrando una adaptación más detallada .

El ajuste fino es reservado para las etapas finales del entrenamiento, se debe configurar con un ratio de aprendizaje bajo. Usualmente los modelos inician con un estado de conocimiento muy malo y tienen un ratio de aprendizaje alto, sin embargo cuando se hace ajuste fino el modelo debe ser bastante robusto. El proceso puede ser dividido en varias fases donde se aumente el número de capas desbloqueadas y se reduzca la razón de aprendizaje. Es importante mencionar que este proceso puede llevar al sobreajuste en algunos modelos si el tamaño de la base de datos no es lo suficientemente amplio.

2.7. Plataformas de desarrollo

A la fecha se han desarrollado algunas plataformas como tensorflow (www.tensorflow.org), theano (deeplearning.net/software/theano/), chainer (www.chainer.org) y mxnet (mxnet.incubator.apache.org/) que proveen muchos de los algoritmos de las redes neuronales profundas, además de las optimizaciones para trabajar en entornos GPU. Estas plataformas permiten al desarrollador enfocarse en la creación de las estructuras del modelo sin tener que preocuparse por detalles de bajo nivel como el gradiente o la administración de la GPU.

2.7.1. Tensorflow

La selección de Tensorflow como plataforma para el proyecto es basada en la calidad y disponibilidad de contenido relacionado. TensorFlow es una librería de código abierto para la computación numérica de flujo de datos a través de un grafo. Los nodos del grafo representan operaciones matemáticas, mientras que las conexiones representan arreglos de datos multidimensionales (tensores) que se comunican entre sí. La flexibilidad de la arquitectura permite el desarrollo de computación en múltiples CPUs e incluso en GPU en computadores de escritorio, servidores e incluso dispositivos móviles a través del uso de un API.

Tensorflow fue desarrollado originalmente por investigadores e ingenieros de «Google Brain Team» con el propósito de conducir investigaciones de «machine learning» y «deep neural networks», pero el sistema es lo suficientemente general para ser aplicable a una amplia variedad de dominios.

La librería está escrita en Python y C++, siendo el manejo de computación en C++ se puede hacer uso de la librería mediante el uso de API's. Con el uso de la API obtenemos una interfaz de desarrollo más amigable y podemos utilizar la velocidad y eficiencia de su código compilado en C++. La librería está disponible como un API en Python, C++, Haskell, Java, Go y Rust. Algunos paquetes terceros están disponibles para C#, Julia, R y Scala.

Capítulo 3

Pruebas de la investigación

El siguiente capítulo contendrá el trabajo realizado para la creación de la red neuronal aplicada al problema de reconocimiento de macroinvertebrados. Se explicará el proceso de generación de la base de datos de imágenes, la aumentación de datos, la arquitectura del modelo y los resultados obtenidos en las diferentes pruebas.

3.1. Base de datos de imágenes

La base de datos de imágenes es uno de los elementos principales para el algoritmo de entrenamiento. La cantidad y calidad de estas determinará en gran parte la precisión de predicción del sistema; entre más imágenes mejores resultados. La cantidad mínima suele variar de problema a problema y depende de una serie de factores como la cantidad de clases, el algoritmo a utilizar, la similaridad de las clases y otros más.

El sistema final debe ser capaz de identificar las 81 familias con precisión, pero para finalidad de esta fase de pruebas se establecieron 10 familias, disminuyendo la complejidad de la tarea; aunque estos modelos se han visto con

muy buen desempeño en hasta 1000 clases diferentes (Szegedy y col. 2014), como en el concurso ImageNet.

Para esta fase de prueba centraremos nuestra selección de familias al grupo de las conchas. En este caso la similaridad es un factor importante a tomar en cuenta. Esto representa un incremento en la complejidad, especialmente en familias como "Corbiculidae", "Ancylidae", "Sphaeriidae"; o "Ampullaridae", "Hydrobiidae", "Lymnaeidae", "Physidae", "Thyaridae"; donde la mayoría de las diferencias reside en factores como su tamaño, dureza y distribución de características específicas como agujeros y patrones de coloración (ver figura 3.1).

Durante el proceso de investigación se aplicó el algoritmo a otros conjuntos de datos, incluyendo experimentos con "Calopterygidae" y "Heptageniidae" con publicación en 3er Congreso Científico de la Universidad Autónoma de Chiriquí y en el VI Congreso Internacional de Ingeniería, Ciencias y Tecnología de la Universidad Tecnológica de Panamá.

Posteriormente se hicieron pruebas abarcando todas las familias del proyecto pertenecientes a la orden "Ephemeropterans" y "Odonata".

3.1.1. Generación de la base de datos

Para el entrenamiento del algoritmo es indispensable la generación de una base de datos de imágenes que contenga ficheros de las familias de macroinvertebrados que se eligieron. Dentro deberán estar las imágenes en formato JPG con tamaños al menos superior a los 150 pixeles de ancho y alto, donde 300 pixeles sería el tamaño óptimo.

Nuestra base de datos en su mayoría es conformada por imágenes provenientes de búsquedas en la internet, con ayuda del plugin de Google Chrome «Fatkun Batch Download Image», que permite automatizar el proceso de descarga y etiquetado de las imágenes. Cada carpeta contiene entre 70 y 90 imágenes para la fase de pruebas de este proyecto, con posibilidad a ser ampliado una vez el proyecto se encuentre en marcha.

3.1.2. Procesamiento de la base de datos

Posteriormente estas deben ser procesadas de manera automatizada para normalizar ciertos valores. Las imágenes primero son redimensionadas a un tamaño definido en el modelo como el tamaño de entrada, este valor para Inception v3 ronda entre 224x224 y 299x299; en nuestro caso lo definimos en 299x299. Luego los pixeles de cada imágenes deben ser normalizados desde valores enteros 0-255 a de punto flotante entre -1.0 y 1.0 con el objetivo de reducir considerablemente el tiempo de convergencia en el proceso de entrenamiento.

Por último las imágenes son empaquetadas en tres conjuntos diferentes: entrenamiento, validación y prueba. El primer conjunto, el más grande, es definido arbitrariamente a 60 % de las imágenes. Este conjunto es el que permite al modelo modificar sus valores internos para lograr la predicción de las imágenes. Esto es realizado comparando el resultado de la predicción y la etiqueta correcta de cada imagen mediante el algoritmo de retropropagación, explicado en el la sección 2.1.2. En el conjunto de validación contamos con un 30 % de las imágenes que nos permiten determinar la precisión de los modelos

que se generan durante el entrenamiento. En esta fase el algoritmo no se le proporciona la etiqueta de las imágenes. Por último, en el conjunto de prueba que representa un 10 % de las imágenes, el modelo con mayor puntaje en el conjunto de validación es evaluado para determinar la utilidad.

Aumentación de Datos

La mejor manera para ayudar a un modelo de «machine learning» a generalizar mejor es incrementar la cantidad de datos. Esto en práctica no siempre es posible cuando se trabaja con bases de datos limitadas. Una solución alternativa para este problema es generar datos falsos y agregarlos al conjunto de entrenamiento.

La aumentación de datos es una técnica efectiva en los problemas de reconocimiento de objetos. Operaciones como la traslación de algunos pixeles en cada dirección ofrecen grandes beneficios a la generalización incluso si el modelo es invariante al utilizar convolución y «pooling». Otras operaciones de gran ayuda son rotación, volteado, variaciones de brillo, contraste y saturación.

Es importante tener mucho cuidado al aplicar transformaciones que puedan afectar la correcta clasificación de las imágenes. Por ejemplo para el entrenamiento del número “6” y “9” las transformaciones de volteado horizontal o el giro de 180 grados no son adecuadas. Un ejemplo en el proyecto de los macroinvertebrados donde muchas de las familias son asimétricas y su característica diferenciadora es la orientación del agujero de su caparazón (ver figura 3.1).

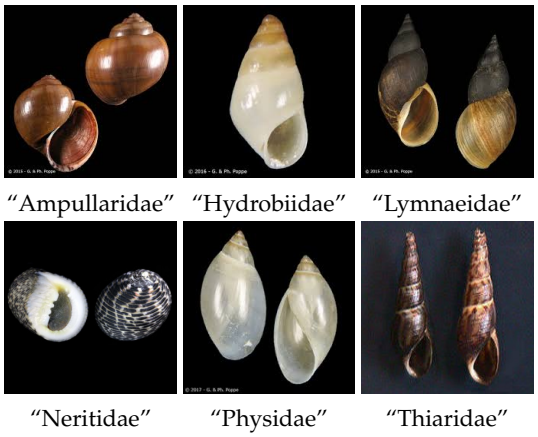


FIGURA 3.1: Familias asimétricas a tener en consideración a la hora de aplicar aumentación de datos (*Conchas - Macro Invertebrados*).

En el proyecto se utilizó «imgaug», una librería de python que ayuda en el proceso de aumentación de datos en imágenes para proyectos de «machine learning» creada por Alexander Jung («imgaug»). Esta convierte un conjunto de imágenes con diferentes técnicas de alteración (ver imagen 3.2). Para el proyecto se utilizaron las siguientes modificaciones:

- Crop
- Pad
- Fliplr
- Flipud
- Superpixels
- Add
- Multiply
- GaussianBlur
- AverageBlur
- MedianBlur
- Sharpen
- Emboss
- SimplexNoiseAlpha
- AdditiveGaussianNoise
- Dropout
- CoarseDropout
- Invert
- AddToHueAndSaturation
- FrequencyNoiseAlpha
- ContrastNormalization

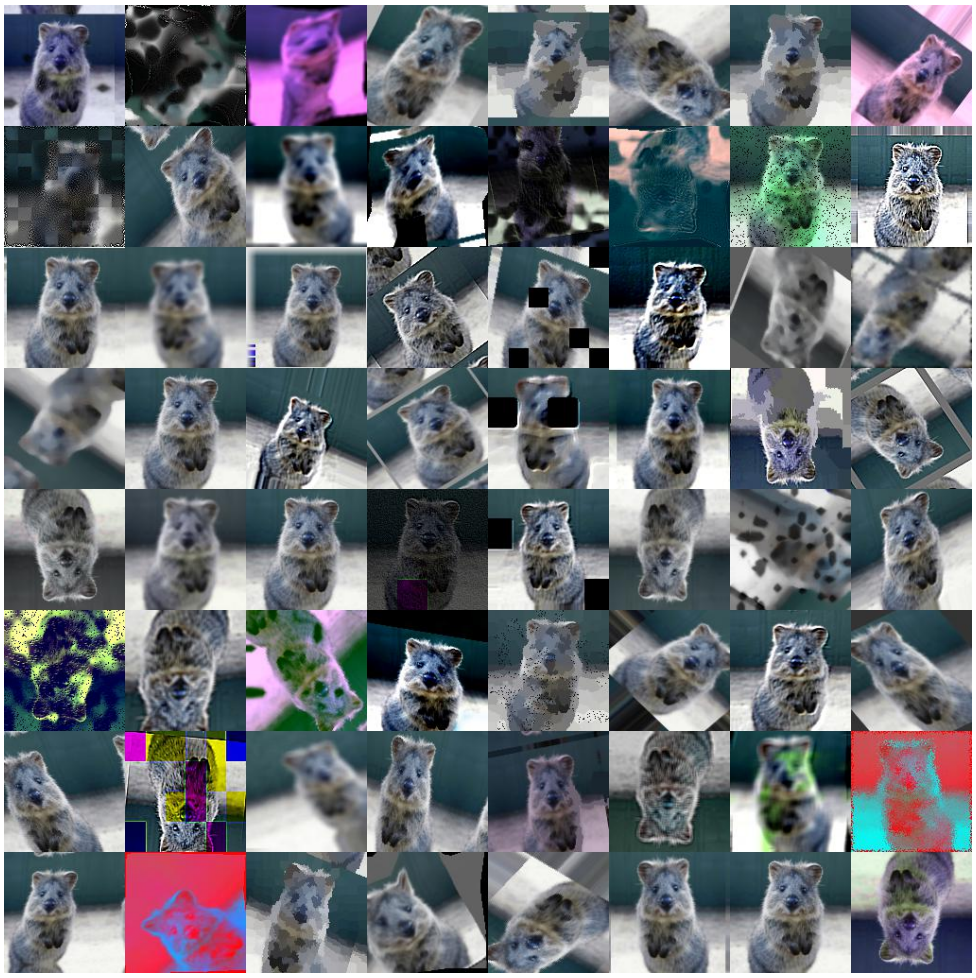


FIGURA 3.2: Algunas de las transformaciones que se pueden aplicar utilizando «imgaug».

- Grayscale
- ElasticTransformation
- PiecewiseAffine
- PerspectiveTransform

Para mayor detalle sobre los parámetros utilizados ver Anexo A.

3.1.3. Bases de datos utilizadas

Familias "Calopterygidae" y "Heptageniidae"

En las fases tempranas del proyecto se aplicó el algoritmo a un conjunto de datos que solamente incluía dos familias "Calopterygidae" y "Heptageniidae" (ver figura 3.3). Los resultados de este experimento fuero presentados en el 3er Congreso Científico de la Universidad Autónoma de Chiriquí y en el VI Congreso Internacional de Ingeniería, Ciencias y Tecnología de la Universidad Tecnológica de Panamá.

En estas pruebas se utilizaron 47 y 50 imágenes para "Calopterygidae" y "Heptageniidae" respectivamente. A estas imágenes no se les llegó a aplicar ninguna técnica de aumentación de datos.

Familias de la Orden "Ephemeropterans" y "Odonata"

Posteriormente se probó ampliar el número de clases y se decidió incluir el resto de las familias de las ordenes "Ephemeropterans" y "Odonata", las cuales incluyen las siguientes familias: "Calopterygidae", "Gomphidae", "Heptageniidae", "Heteragrionidae", "Leptophlebiidae", "Perilestidae" y "Polythoridae" (ver figura 3.3).

Sin embargo se descubrió que la disponibilidad de imágenes de algunos de estos grupos era limitada y conllevaría a bajos resultados de precisión, siendo muy difícil hacer una evaluación del modelo de redes neuronales y fueron removidas de la base de datos. Entre esas familias están "Heteragrionidae", "Perilestidae" y "Polythoridae".

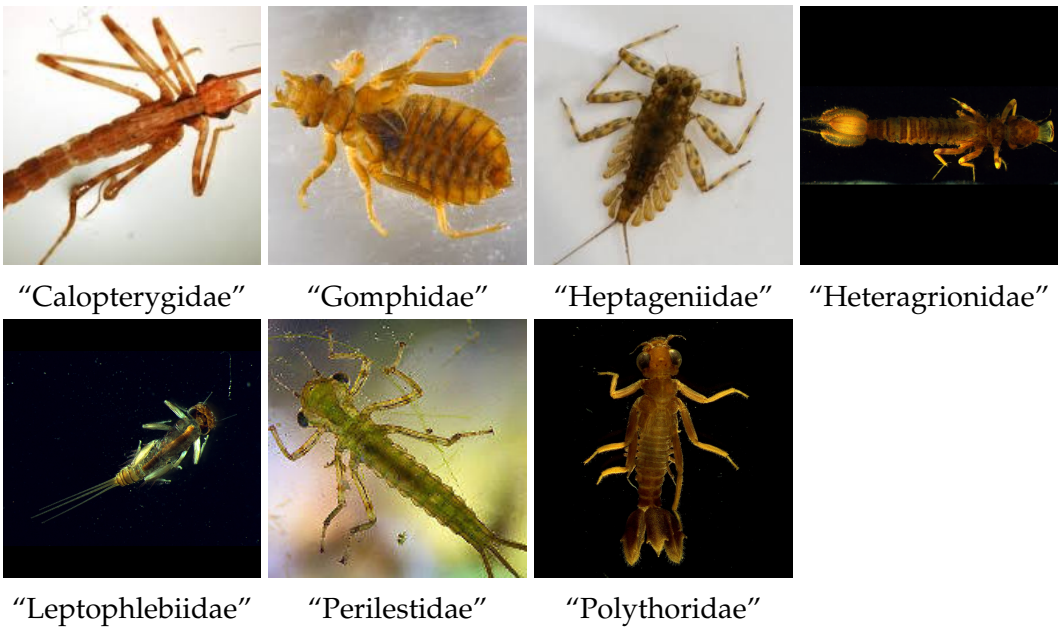


FIGURA 3.3: Familias pertenecientes a la orden "Odonata" y "Ephemeropterans" (*Libélulas*).

Para el caso de estas familias en particular el proyecto tiene contemplado alimentar la base de datos de manera gradual con imágenes del laboratorio y las aportaciones de campo por parte de las comunidades.

El conjunto final de imágenes consiste de "Calopterygidae", "Gomphidae", "Heptageniidae" y "Leptophlebiidae", con 47, 334, 50 y 76 imágenes respectivamente.

Familias de Conchas

En nuestro estudio se eligieron 10 familias de conchas, estas son: "ampullariidae", "ancylidae", "corbiculidae", "hydrobiidae", "lymnaeidae", "neritidae", "physidae", "planorbidae", "sphaeriidae" y "thiaridae" (ver figura 3.4). Para el caso particular de estas familias se pudo lograr armar una base de datos con 80, 69, 82, 84, 89, 86, 70, 86, 70 y 85 imágenes respectivamente.

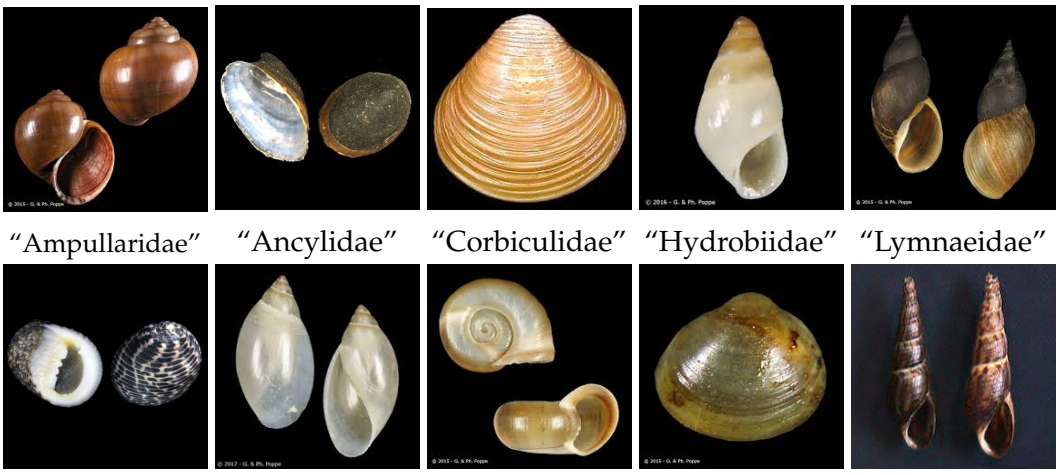


FIGURA 3.4: Familias elegidas para el reconocimiento automático (*Conchas - Macro Invertebrados*).

Estas fueron elegidas debido a que engloban un conjunto de macroinvertebrados muy beneficioso para la medición del coeficiente de contaminación de los ríos y por su naturaleza común, permitiendo hallar una cantidad de imágenes suficientes para satisfacer los requerimientos de entrenamiento en el modelo de redes neuronales profundas.

Es importante resaltar que para el entrenamiento de estas familias se tuvo que desarrollar un módulo de aumentación de datos selectivo, donde se omitían ciertas transformaciones debido a la naturaleza asimétrica de las conchas.

3.2. El modelo

Para el proyecto se decidió trabajar utilizando el modelo Inception-V3 entrenado para ILSVRC-2012 sin incluir las capas finales.

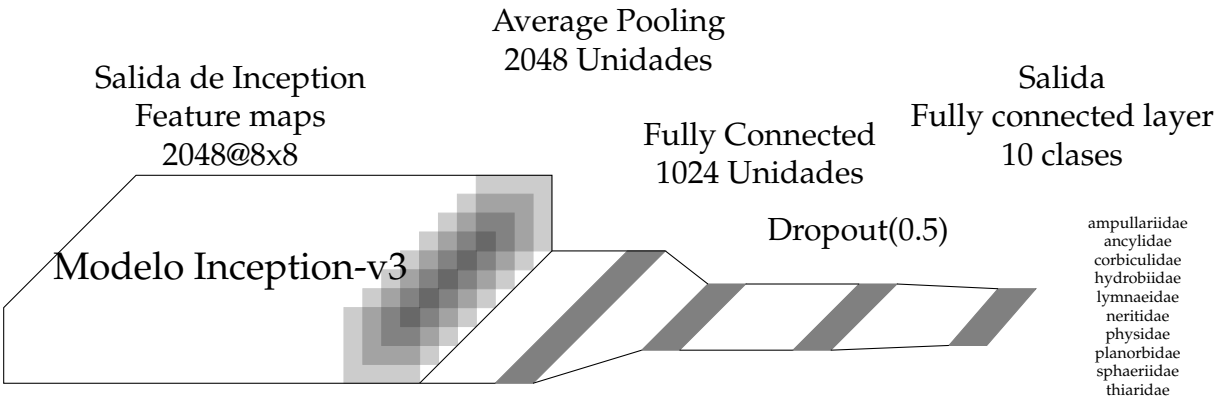


FIGURA 3.5: La salida del modelo Inception-V3 es reducida mediante pooling y luego se conecta con la capa de salida.

En la capa inicial se le define para aceptar imágenes de entrada de tamaño 299x299 con los tres canales de color RGB.

Luego las capas del modelo son congeladas, esto quiere decir que cuando se hace el proceso de entrenamiento los valores de los coeficientes W y β no pueden ser alterados.

Este modelo aún está incompleto, pues no contiene una capa de salida. En nuestro modelo agregamos una capa de «pooling» bidimensional a la capa convolucional para reducir la carga computacional y de memoria. Conectamas la salida a una capa completamente conectada de 1024 nodos y luego agregamos una capa de regularización Dropout a un 50 % de su valor. Luego conectamos a la capa de salida con una cantidad de nodos definidos por la cantidad de clases que debe reconocer nuestro sistema. Como se puede visualizar en la figura 3.5.

3.3. Metodología de Evaluación

Existen diferentes parámetros por los cuales podemos evaluar la efectividad de un modelo en la tarea de clasificación. Podemos hacer uso de múltiples de estos y hacer un análisis sobre las necesidades del modelo dependiendo de los valores alcanzados. En esta sección hablaremos sobre como son calculados estos valores y cómo deben ser analizados.

Estos métodos de evaluación son aplicados a los tres conjuntos de imágenes de manera separada, un análisis importante es la comparación de los resultados en el conjunto de entrenamiento y el de validación, debido a que pueden señalar problemas en el modelo, en la base de datos y en los algoritmos de aumentación de datos.

3.3.1. Precisión top-n

El método por defecto de evaluación es calcular el valor top-1 o el porcentaje de aciertos del modelo. Se considera la clase con mayor puntaje en la capa softmax final del modelo y se compara con la clasificación correcta, de ser igual se dice que es una predicción correcta y se agrega a un contador, de esa manera se evalúan todas las imágenes del subconjunto y se calcula un promedio. El valor resultante será el porcentaje de precisión del modelo.

Sin embargo esta metodología se puede ampliar a una evaluación top-n, donde se consideran los n valores con mayor puntaje en la capa softmax de salida y si alguna de estas clases es la acertada se toma como respuesta correcta. Por ejemplo si se evalúa una imagen de “Ampullaridae” y el modelo genera en su capa softmax puntajes donde esta clase recibe el segundo mayor

puntaje en una evaluación top-2 será tomado como una respuesta correcta, mientras que en top-1 no lo es.

Este tipo de evaluación es más flexible y beneficioso cuando se tienen clases que son muy similares entre sí, y es tolerable fallar, por ejemplo en un modelo de clasificación de animales se pueden tener clases para perros y lobos, los cuales comparten características muy similares si los comparamos con otras especies como elefantes, gatos o ballenas.

3.3.2. Coeficiente kappa de Cohen

El coeficiente kappa de Cohen es una medida estadística que permite la medición azar categórica de las clases. Es generalmente una medida más robusta que simplemente el porcentaje, ya que la kappa toma en consideración las posibilidades de acertar por azar.

Este valor es de gran utilidad cuando las base de datos de imágenes son desiguales en cuanto a cantidad. Cuando se evalúa un modelo con una base de datos desbalanceada la clase con mayor número de imágenes es probabilísticamente más común y el modelo de redes neuronales se aprovecha de esta característica para maximizar sus resultados. Por ejemplo si una clase contiene 90 imágenes y otra 10 imágenes solamente, la primera clase es 90 % más probable cuando se va a evaluar el conjunto.

Para calcular este coeficiente primero debemos colocar las predicciones del modelo junto con las respuestas en una matriz de confusión.

En la tabla 3.1 podemos seguir un ejemplo y encontrar el coeficiente. Para armar la matriz de confusión necesitamos cruzar el arreglo de predicciones

con las respuestas correctas. Luego necesitamos sumar los valores coincidentes entre las predicciones y las respuestas correctas, estos están ubicados en la matriz principal. Posteriormente haremos una sumatoria de la diagonal, en el ejemplo la matriz principal la componen los valores $[4, 5, 3]$, dando como resultado 12 los valores coincidentes.

Posteriormente debemos calcular las concordancias por probabilidad, en este caso debemos sacar el producto de la sumatoria de las filas y columnas. En este caso multiplicaremos los arreglos $[4, 8, 3]$ y $[5, 5, 5]$, lo cual da como resultado $[20, 40, 15]$. A estos valores los dividimos con el total de elementos, en este caso 15 y nos queda el siguiente arreglo [1.33, 1.67, 1]. Estos son los valores de concordancia por probabilidad de las tres clases, para calcular el coeficiente hacemos una sumatoria y nos queda 5.

Mediante la siguiente ecuación podremos calcular el coeficiente kappa de Cohen:

$$kappa = \frac{(concordancia - concordancias\_probabilidad)}{(total\_elementos - concordancias\_probabilidad)} \tag{3.1}$$

$$kappa = \frac{(12 - 5)}{(15 - 5)} = 0{,}7$$

El coeficiente kappa de Cohen es siempre un valor menor o igual a 1. Cualquier valor menor a 0 indica que el clasificador es inútil. No existe una manera estandarizada para la interpretación de los valores, pero Landis y Koch (Landis y Koch 1977) provee la siguiente escala. De acuerdo a el esquema un valor menor a 0 indica que no hay coincidencias, valores entre 0-0.20 indica una ligera coincidencia, entre 0.21-0.40 como ligera, 0.41-0.60 como razonable,

		Respuestas			
		Calopterygidae	Gomphidae	Heptageniidae	
Predicciones	Calopterygidae	4	1	0	5
	Gomphidae	0	5	0	5
	Heptageniidae	0	2	3	5
		4	8	3	15

TABLA 3.1: Matriz de confusión.

0.61-0.80 como substancial y 0.81-1 casi perfecta coincidencia.

Para maximizar este valor durante el entrenamiento podemos asignar ponderaciones a las clases que tienen menos imágenes y de esa manera castigar cuando falla estas de manera más severa.

3.4. Resultados

Para esta sección se consideran los resultados de la ejecución del modelo en 4 «datasets» diferentes aplicando o no los algoritmos de aumentación de datos y ajuste fino. Los «datasets» están conformados por:

1. Familias “Calopterygidae” y “Heptageniidae”
2. Orden “Ephemeropterans” y “Odonata”
3. Conchas
4. Todas las anteriores

Los algoritmos de aumentación de datos dependen de cada familia y para mayor detalle se recomienda ver el anexo A.

3.4.1. Experimentos

En esta sección se mostrarán los resultados de precisión top-1, top-3, top-5, el coeficiente kappa de Cohen y el valor de error de entropía cruzada (ver capítulo 2.1.2) del modelo para cada «dataset». También se harán comparaciones entre los resultados de los modelos al aplicar los algoritmos de aumentación de datos y al hacer uso de ajuste fino.

En las tablas y figuras posteriores el proceso de transferencia de conocimiento estará con las siglas TL, del inglés Transfer Learning; ajuste fino será FT, de sus siglas en inglés Fine Tuning; y la aumentación de datos será DA, del inglés Data Augmentation.

En esta sección se mostrarán las gráficas generadas por los puntajes de evaluación durante el proceso de entrenamiento y también se genera una tabla que muestra los mejores modelos de acuerdo a el puntaje obtenido en la clasificación top-1 en el subconjunto de evaluación.

Familias “Calopterygidae” y “Heptageniidae”

El primer paso para evaluar los resultados es comparar los métodos en términos de rendimiento en las predicciones. La tabla 3.2 muestra los resultados de las predicciones sobre los tres subconjuntos del «dataset». Este primer conjunto nos permite hacer una prueba rápida debido al reducido volumen de imágenes que suman ambas familias.

De las gráficas 3.6, 3.7 y 3.8 podemos apreciar que el modelo aprende rápidamente a predecir el conjunto de entrenamiento obteniendo puntajes

	Entrenamiento			Validación		
Modelo	top-1	κ de Cohen	error	top-1	κ de Cohen	error
TL	0.9167	0.8295	0.3265	0.8333	0.6606	0.4794
TL-DA	0.8906	0.7582	0.2798	0.8000	0.5982	0.3941
FT-DA	0.9922	0.9844	0.0150	0.9000	0.7982	0.3882

	Prueba		
Modelo	top-1	κ de Cohen	error
TL	0.7000	0.4000	0.5012
TL-DA	0.9000	0.8000	0.2914
FT-DA	0.9000	0.8000	0.1951

TABLA 3.2: Comparación de métodos para el primer «dataset».

cercanos al 100 % de precisión, en parte debido al reducido tamaño de este. Sin embargo la validación se estanca por debajo del 90 %.

Adicional podemos apreciar una progresión de mejora en los resultados de validación y prueba dependiendo del método utilizado, siendo la aplicación de Data Augmentation y Fine Tuning la mejor implementación.

Un factor importante que podemos notar es la variación del coeficiente κ de Cohen, el cual se mantiene en un rango considerado como substancial, indicando que no existe una tendencia a predicciones por azar excepto en el método sin aumentación de datos, el cual marca un 0.40 en el conjunto de prueba.

Los resultados son buenos, pero podrían ser mejores al aumentar el número de imágenes, permitiendo aprender más características diferenciadoras de las diferentes especies, esto es posible deducirlo al considerar el reducido valor de error en el conjunto de entrenamiento, sugiriendo una memorización del conjunto impidiéndole mejorar los resultados de validación y prueba.

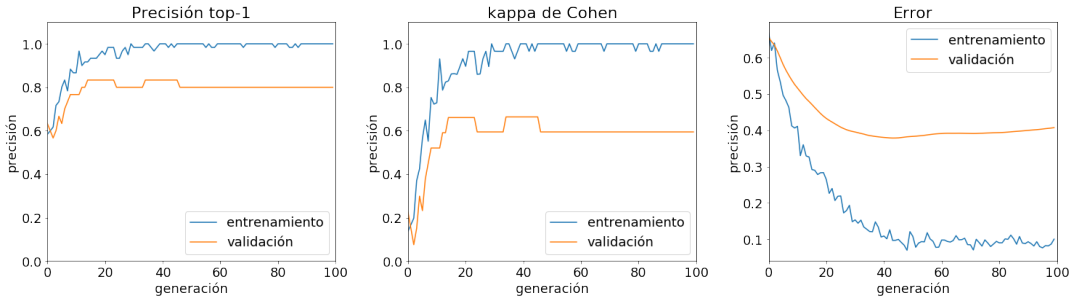


FIGURA 3.6: Evaluación para el primer «dataset» utilizando «Transfer Learning».

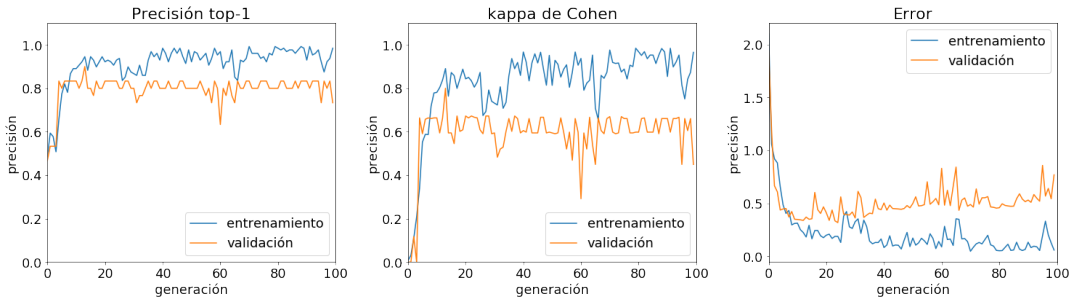


FIGURA 3.7: Evaluación para el primer «dataset» utilizando «Transfer Learning with Data Augmentation»..

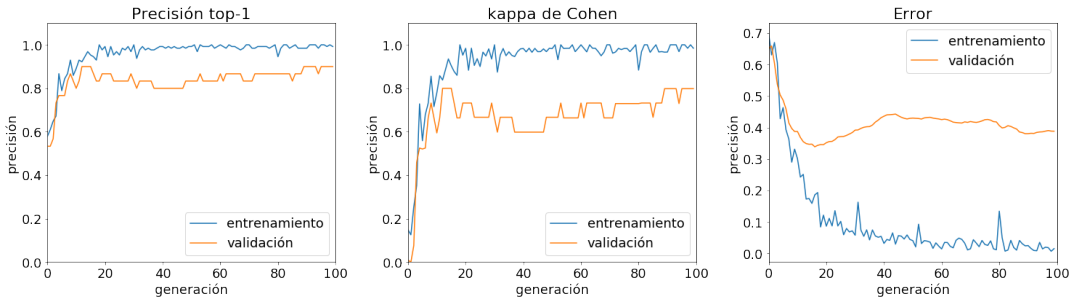


FIGURA 3.8: Evaluación para el primer «dataset» utilizando «Transfer Learning with Data Augmentation and Fine Tuning».

Orden "Ephemeropterans" y "Odonata"

En el segundo experimento comparamos los métodos aplicados sobre el segundo «dataset», el cual consiste en la orden "Ephemeropterans" y "Odonata". El tamaño de este conjunto es mucho más grande que el anterior permitiendo resultados más interesantes si lo comparamos con el sistema final. Los resultados los podemos ver en la tabla 3.3.

De las gráficas 3.9, 3.10 y 3.11 podemos ver que con al no aplicar aumentación de datos los resultados entre el conjunto de entrenamiento y validación difieren grandemente y se sigue presentando al final una memorización del conjunto al obtener 100 % de precisión y valores cercanos a cero en el error de entropía cruzada.

Posteriormente al aplicar aumentación de datos corregimos el problema con el conjunto de entrenamiento y logramos un incremento en la precisión, pero la precisión top-1 de ambos conjuntos se estanca cerca de la generación 50, a pesar que el error se reduce en el conjunto de entrenamiento. Adicionalmente los valores presentan una gran fluctuación entre generaciones.

Luego al aplicar técnicas de ajuste fino y liberar las capas finales del modelo aumentando la capacidad del mismo logramos una gráfica más estable y un incremento en la precisión de manera considerable logrando el mejor modelo posible para la tarea de predicción en este conjunto de datos. Sin embargo podemos notar que durante la generación 60 en adelante hay un incremento en el error de validación y un una reducción en el error de entrenamiento siendo una señal de sobreajuste indicando que el proceso de entrenamiento debe ser detenido y que para lograr mejores resultados se deben aplicar técnicas de

	Entrenamiento				Validación			
Modelo	top-1	top-3	κ de Cohen	error	top-1	top-3	κ de Cohen	error
TL	1.0000	1.0000	1.0000	0.0432	0.7191	0.9775	0.5979	0.7657
TL-DA	0.8750	1.0000	0.8246	0.3867	0.7416	0.9438	0.6350	0.7241
FT-DA	0.8828	1.0000	0.8326	0.3133	0.8090	0.9888	0.7354	0.5880

	Pruebas			
Modelo	top-1	top-3	κ de Cohen	error
TL	0.7742	1.0	0.6303	0.5671
TL-DA	0.8387	1.0	0.7359	0.4539
FT-DA	0.8710	1.0	0.7963	0.3352

TABLA 3.3: Comparación de métodos para el segundo «dataset».

regularización más agresivas o un incremento de la base de datos.

Nuevamente el coeficiente κ de Cohen se mantiene en un rango substancial, lo cual indica que el azar en las pruebas no influyen de manera marcada en el éxito de las predicciones.

Conchas

En el tercer experimento comparamos los métodos aplicados sobre el tercer «dataset», el cual consiste en familias de conchas. Estos resultados los podemos ver en la tabla 3.4 utilizando transferencia de conocimiento y luego ajuste fino, de igual manera comparamos los resultados para los datos sin alterar y los aumentados. El conjunto consiste en 10 clases de las cuales hay varias que presentan similaridades muy grandes y es necesario considerar factores como la orientación para clasificación de la especie correcta, por lo que en la fase de aumentación de datos se efectúa con modificadores de volteado selectivo según la familias de macro invertebrado.

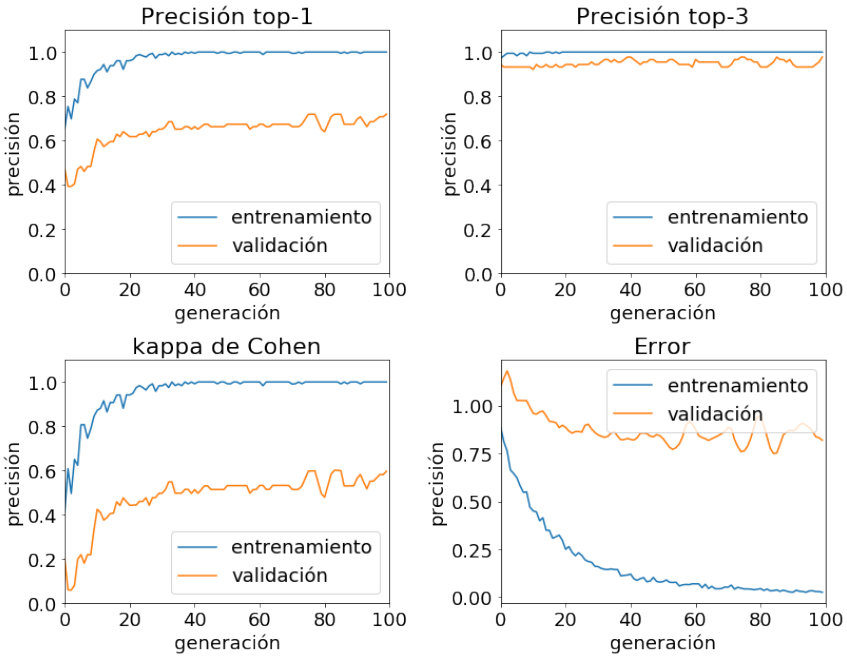


FIGURA 3.9: Evaluación para el segundo «dataset» utilizando «Transfer Learning».

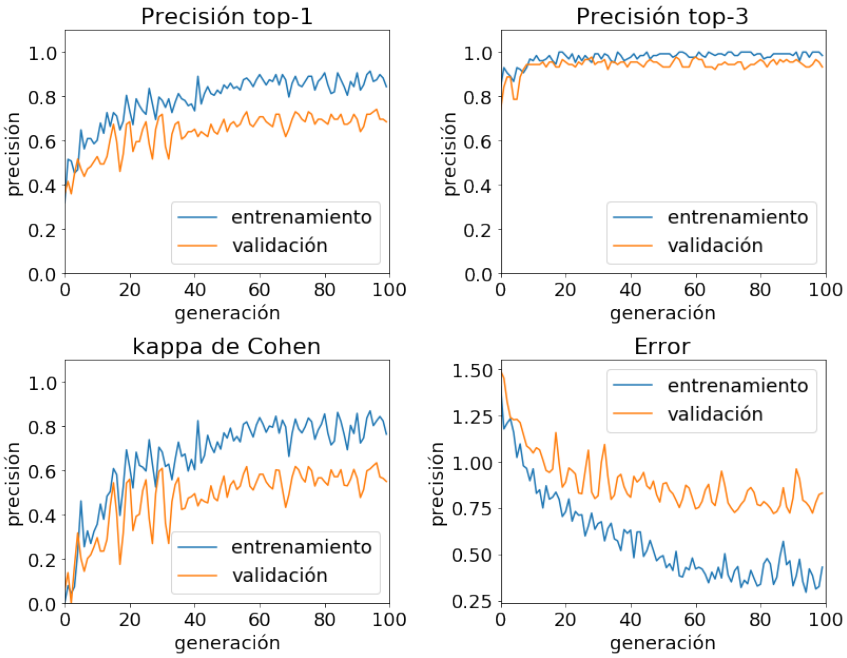


FIGURA 3.10: Evaluación para el segundo «dataset» utilizando «Transfer Learning with Data Augmentation»..

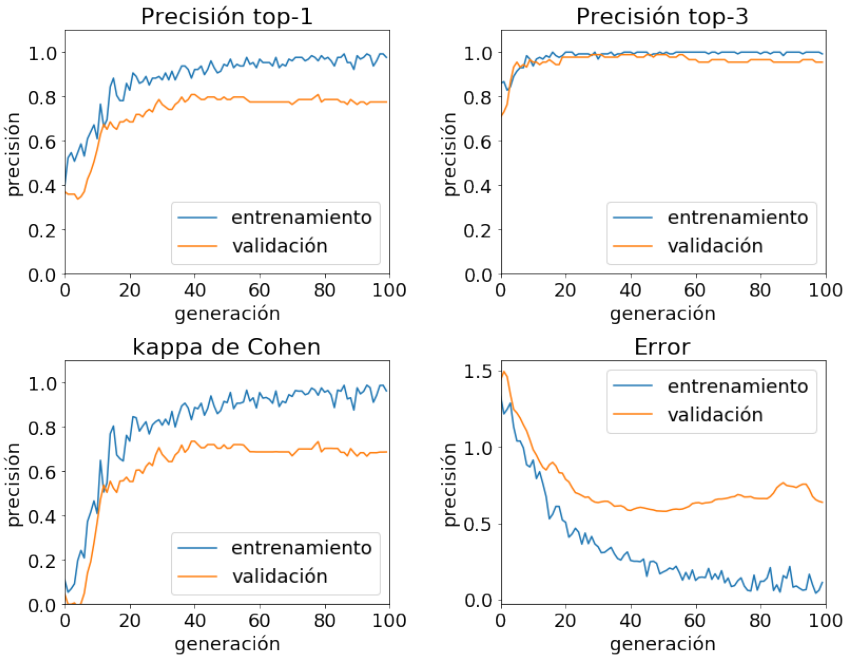


FIGURA 3.11: Evaluación para el segundo «dataset» utilizando «Transfer Learning with Data Augmentation and Fine Tuning».

De las gráficas 3.12, 3.13 y 3.14 podemos ver que con al no aplicar aumentación de datos los resultados entre el conjunto de entrenamiento y validación difieren grandemente y aunque requiera de más generaciones se sigue llegando al punto donde el modelo se memoriza el conjunto de entrenamiento y alcanza 100 % de precisión y valores cercanos a cero en el error de entropía cruzada. Al contar con más clases es importante resaltar que podemos evaluar el modelo por sus predicciones top-3 y top-5, las cuales para el primer método son bastantes satisfactorias según la tabla.

Al aplicar aumentación de datos selectiva a las clases se obtiene nuevamente una gráfica con valores inestables, pero esta vez sin mejoras aparentes en los resultados frente a los conjuntos de validación y prueba. Y las predicciones en el conjunto de entrenamiento se detienen cerca del 70 %, esto es un claro

	Entrenamiento					Validación				
Modelo	top-1	top-3	top-5	κ de Cohen	error	top-1	top-3	top-5	κ de Cohen	error
TL	0.9860	1.0000	1.0000	0.9843	0.2041	0.6682	0.8925	0.9486	0.6278	0.9586
TL-DA	0.7188	0.9219	0.9766	0.6848	0.8281	0.6776	0.9019	0.9766	0.6306	0.9697
FT-DA	0.9062	0.9922	1.0000	0.8932	0.3480	0.7523	0.9579	0.9860	0.7160	0.7772

	Prueba				
Modelo	top-1	top-3	top-5	κ de Cohen	error
TL	0.6667	0.9306	0.9861	0.6189	1.0537
TL-DA	0.6528	0.9028	0.9722	0.6079	0.9759
FT-DA	0.7500	0.9305	0.9722	0.7094	0.8196

TABLA 3.4: Comparación de métodos para el tercer «dataset».

indicativo de que se alcanzó el límite de aprendizaje del modelo y debemos probar con una mayor amplitud.

Utilizando un entrenamiento de ajuste fino logramos superar la barrera impuesta en el método anterior logrando maximizar el aprendizaje logrando resultados cerca de 10 % mejores y con una reducción del error de casi un 30 % en el conjunto de prueba.

Al igual que en el experimento anterior la κ de Cohen se mantiene en un rango substancial, lo cual es beneficioso para considerar el modelo para el sistema final.

Todas las anteriores

Último experimento, que busca replicar las condiciones del sistema final al tener familias de tipos completamente diferentes, aquí comparamos los métodos aplicados sobre un «dataset» que contiene las 10 familias de conchas y las 4 de “Ephemeropterans” y “Odonata”. Estos resultados los podemos ver en la tabla 3.5. Se evalúan los métodos de transferencia de conocimiento y

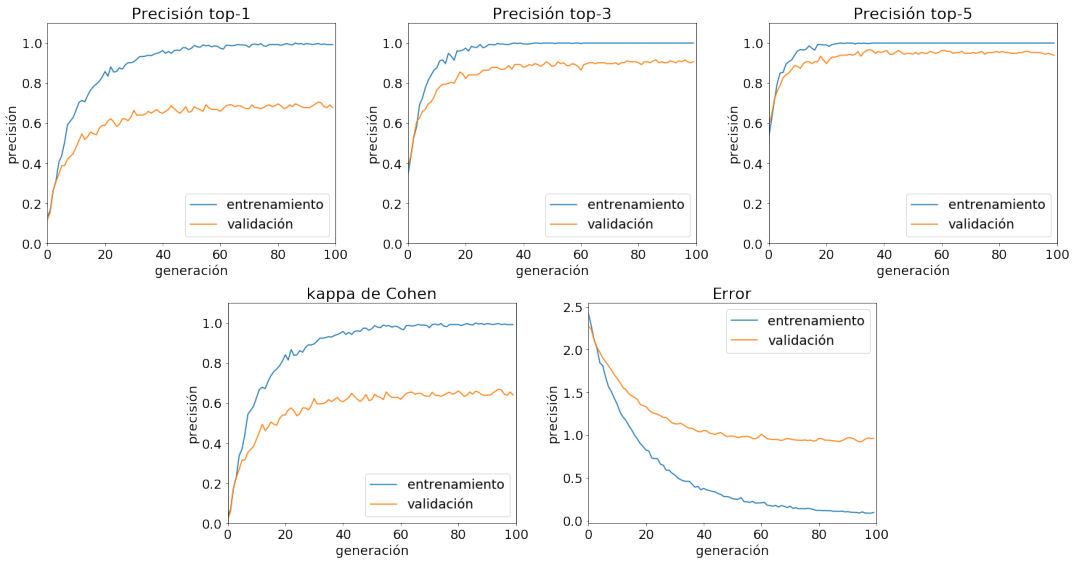


FIGURA 3.12: Evaluación para el tercer «dataset» utilizando «Transfer Learning».

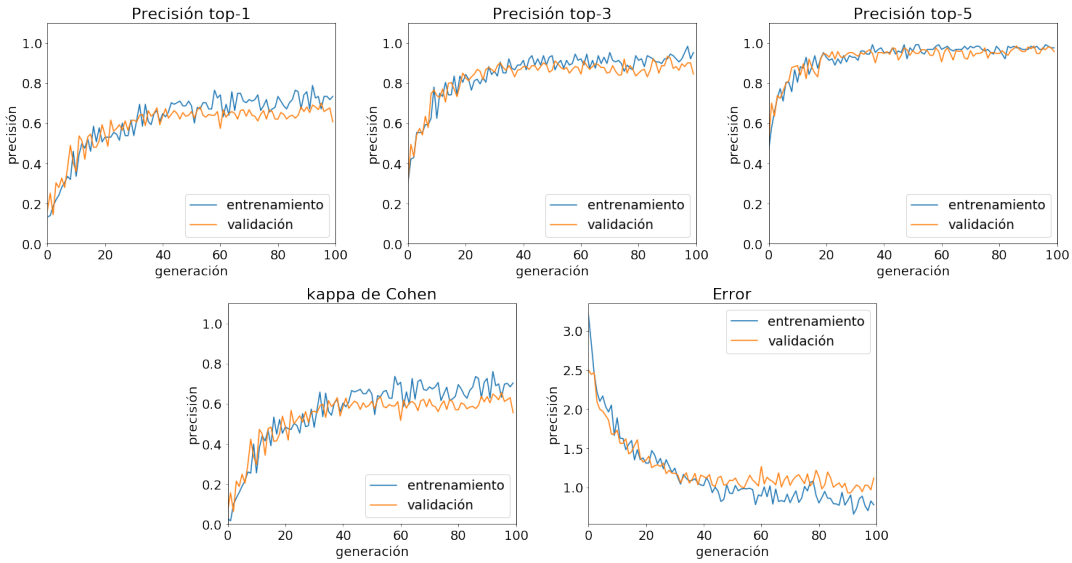


FIGURA 3.13: Evaluación para el tercer «dataset» utilizando «Transfer Learning with Data Augmentation»..

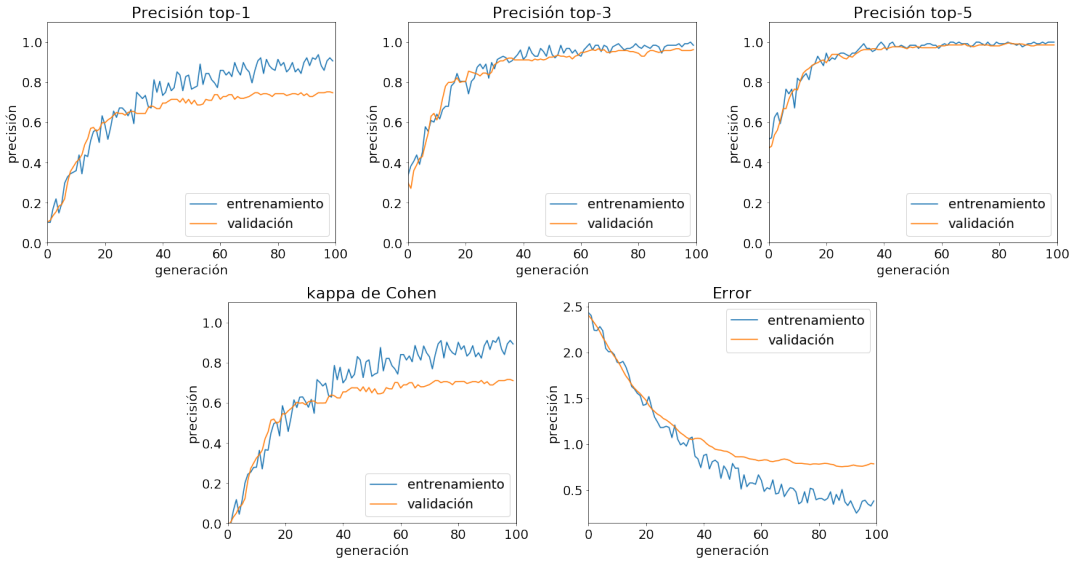


FIGURA 3.14: Evaluación para el tercer «dataset» utilizando «Transfer Learning with Data Augmentation and Fine Tuning».

luego ajuste fino, de igual manera comparamos los resultados para los datos sin alterar y los aumentados.

De las gráficas 3.15, 3.16 y 3.17 podemos encontrar un comportamiento bastante similar con los experimentos anteriores al utilizar los tres diferentes métodos que abarca esta investigación. A pesar de contar con un mayor número de clases se mantienen muy buenos resultados en los todos los criterios de evaluación.

	Entrenamiento					Validación				
Modelo	top-1	top-3	top-5	κ de Cohen	error	top-1	top-3	top-5	κ de Cohen	error
TL	0.9901	1.0000	1.0000	0.9892	0.0401	0.7007	0.8849	0.9309	0.6734	1.1625
TL-DA	0.6641	0.9062	0.9766	0.6299	0.9673	0.6809	0.9046	0.9605	0.6486	0.9643
FT-DA	0.8594	0.9844	1.0000	0.8436	0.5727	0.7270	0.9013	0.9737	0.6993	0.8500

	Prueba				
Modelo	top-1	top-3	top-5	κ de Cohen	error
TL	0.6863	0.8529	0.9706	0.6542	1.2423
TL-DA	0.7059	0.8922	0.9608	0.6760	0.9835
FT-DA	0.8235	0.9804	0.9902	0.8044	0.6856

TABLA 3.5: Comparación de métodos para el cuarto «dataset».

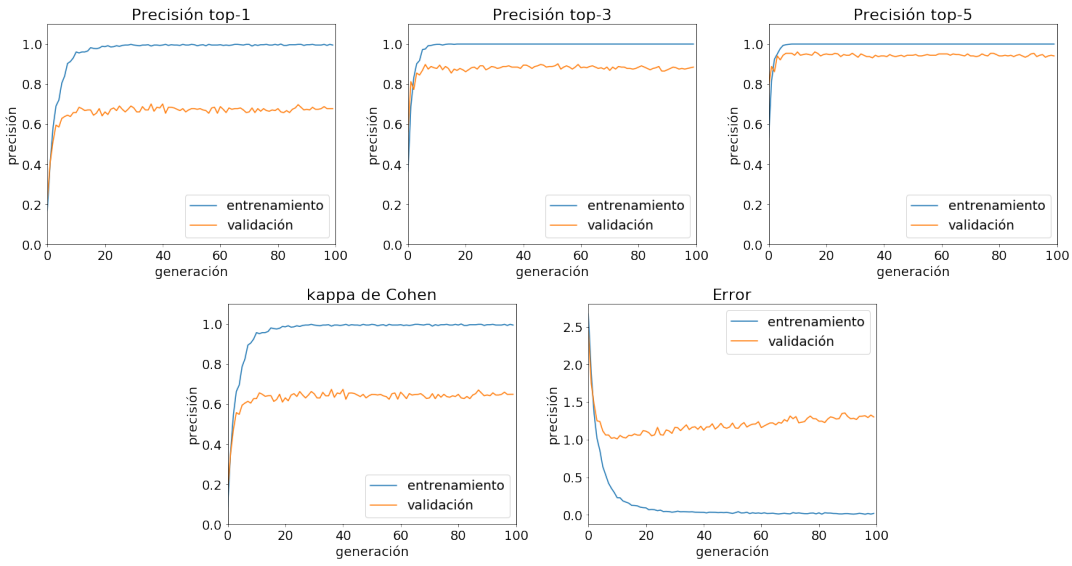


FIGURA 3.15: Evaluación para el cuarto «dataset» utilizando «Transfer Learning».

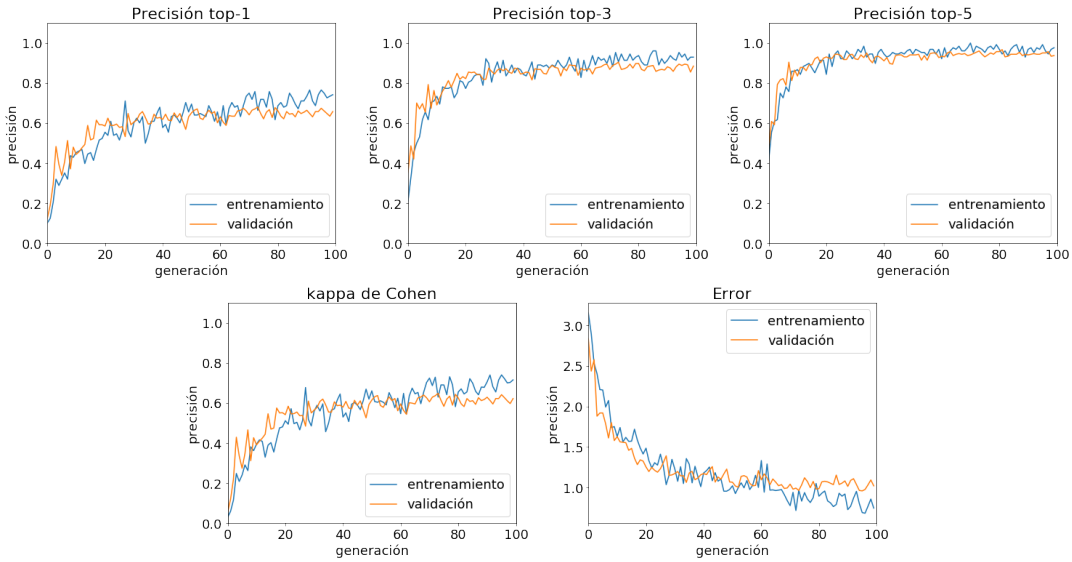


FIGURA 3.16: Evaluación para el cuarto «dataset» utilizando «Transfer Learning with Data Augmentation»..

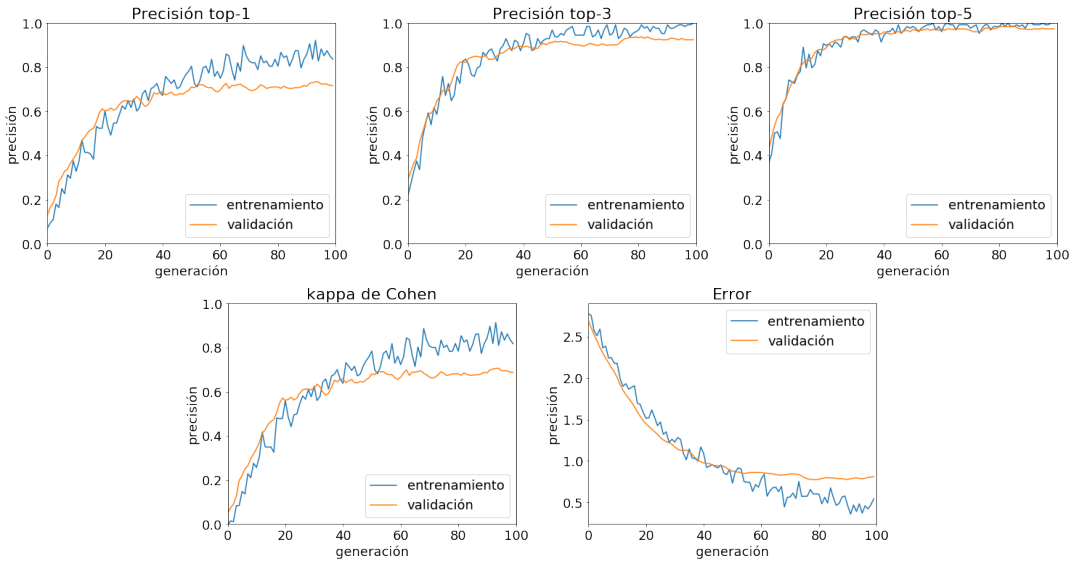


FIGURA 3.17: Evaluación para el cuarto «dataset» utilizando «Transfer Learning with Data Augmentation and Fine Tuning».

Capítulo 4

Aplicación en Android

4.1. NDK y Compilación en Android

Las aplicaciones en Android son desarrolladas típicamente en el lenguaje Java, haciendo uso de un diseño orientado a objetos. Sin embargo hay veces en que existen limitaciones como el rendimiento y manejo de memoria, para estos casos el entorno Android ofrece una alternativa y es su soporte al desarrollo nativo en los lenguajes C/C++.

El kit de desarrollo nativo en inglés «Native Development Kit» (NDK) es un conjunto de herramientas que permite el uso de los lenguajes de programación C/C++ dentro del entorno Android. Este kit permite el uso de librerías para el manejo nativo de actividades y accesar a componentes físicos del dispositivo. Quizás el kit no sea el apropiado para los programadores novatos y solo es recomendado cuando se cumpla algunas de las siguientes condiciones:

1. Se necesite extraer el máximo desempeño del dispositivo para adquirir una latencia más baja en los proceso o desarrollar computación intensiva como la ejecución de juegos y simulaciones físicas.

2. Reutilizar librerías desarrolladas en los lenguajes C/C++.

Para tener acceso a estas capacidades es necesario ejecutar la instalación del kit en el entorno de trabajo de preferencia, para este proyecto se hizo la instalación en Android Studio 2.3.3.

4.2. Librerías y funciones para ejecutar modelo de redes neuronales

Para agregar Tensorflow a una de nuestras aplicaciones Android la manera más sencilla es incluir en el archivo Gradle de nuestra aplicación las siguientes líneas:

```
allprojects {
    repositories {
        jcenter()
    }
}
dependencies {
    compile 'org.tensorflow:tensorflow-android:+'
}
```

CÓDIGO 4.1: Inclusión de Tensorflow en aplicación Android

En la figura 4.1 podemos ver un diagrama sobre la interacción entre los diferentes comoponentes desarrollados en SDK y los de NDK. Por un lado tenemos una serie de clases encargadas del manejo de la interfaz gráfica y las actividades de nuestra aplicación desarrolladas utilizando la SDK, luego se llega a un punto en el que se hace el llamado a la computación de la predicción de una imagen, esta es enviada al clasificador el cual tomara los valores RGB de la imagen y los colocará como el valor de entrada del modelo de redes neuronales, luego se tomará el valor de salida del modelo y este será enviado

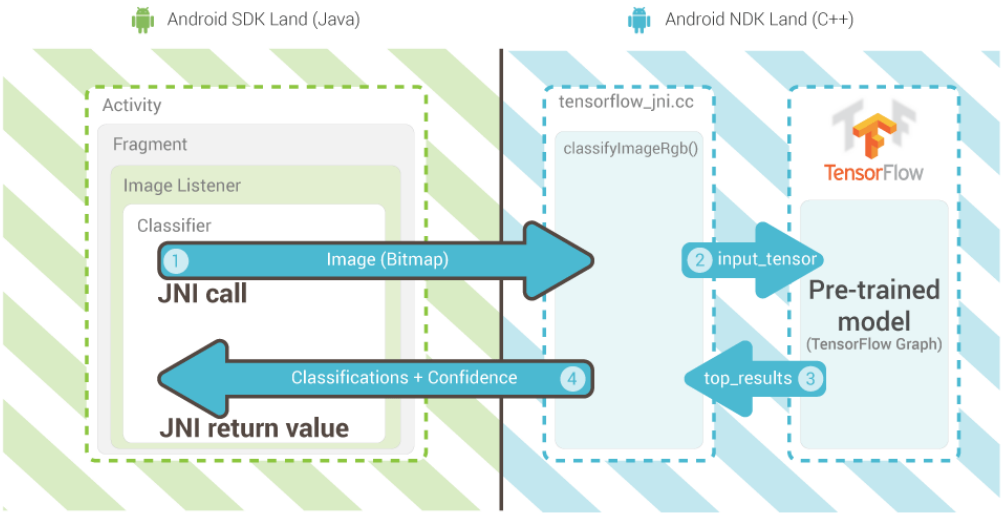


FIGURA 4.1: Diagrama de conexiones entre SDK y NDK para nuestro proyecto. («Diagrama SDK NDK»)

como un arreglo de valores de certeza por cada una de las clases para luego ser trabajado en el lado del SDK.

Posteriormente necesitaremos una clase que se encargue del manejo del programa clasificador. En esta clase importaremos la librería «TensorFlowInferenceInterface», la cual como el nombre lo indica nos brinda una interfaz para inferencias del modelo Tensorflow. Para hacer uso de la misma es recomendable crear una clase manejadora del clasificador e instanciarla, uno de sus componentes debe ser un objeto de la clase «TensorFlowInferenceInterface». El método constructor de esta clase recibe como parámetros el manejador de componentes o «assetManager» y una cadena con la dirección del modelo, el primero es obtenido con el llamado del método getAssets() en la clase «Activity» y el segundo será definido, es recomendable que esté en la carpeta «assets».

```
1 TensorFlowImageClassifier c = new TensorFlowImageClassifier();
2
```

```
3 c.inferenceInterface = new TensorFlowInferenceInterface(
    assetManager, modelFilename);
```

CÓDIGO 4.2: Creación del objeto manejador y la definición del la interfaz para hacer uso del modelo.

Para hacer uso de la interfaz de inferencia hay tres métodos que nos permiten la interacción.

El primer método es "feed", el cual recibe el nombre de la capa de entrada, sus dimensiones y los valores que serán alimentados; para el caso particular de nuestro modelo la capa de entrada es llamada «input_1» y tiene dimensiones $1 \times 299 \times 299 \times 3$.

```
1 inferenceInterface.feed("input_1", floatValues, 1, 299, 299, 3);
```

CÓDIGO 4.3: Copia los valores de entrada al modelo Tensorflow.

El segundo método es "run", este recibe una lista de la o las capas de salida y ejecuta inferencias entre la capa de entrada registrada anteriormente en "feed" y la o las capas de salidas solicitadas; en el caso particular de nuestro modelo esta es la capa "dense_1/Softmax".

```
1 inferenceInterface.run(new String[] {"dense_1/Softmax"});
```

CÓDIGO 4.4: Ejecuta el llamado de inferencias entre la capa entrada y la de salida.

Por último, "fetch" permite la extracción de los resultados de una capa de salida por medio del nombre de la capa que deseemos.

```
1 inferenceInterface.fetch("dense_1/Softmax", outputs);
```

CÓDIGO 4.5: Copia la salida del Tensor a la variable provista en este método.

4.3. Exportación de modelo en Keras

Para exportar un modelo en Keras debemos tener cuidado sobre la plataforma destino en la cual será utilizado, esto se debe a que por lo menos en la actualidad la versión móvil de Tensorflow no cuenta con todas las funciones causando errores difíciles de depurar. Esta sección brinda recomendaciones para exportar un modelo en plataformas PC y móviles.

El método más sencillo es llamar la función «save» una vez finalizado el entrenamiento. Sin embargo esta aproximación es limitada y no permite aprovechar el máximo potencial del entrenamiento.

```
model.save('my_model.h5')
```

Nuestra recomendación es hacer uso de la técnica de regularización, terminación temprana. Al terminar el entrenamiento a tiempo podemos evitar que exista sobre ajuste y guardemos el mejor modelo que la arquitectura y la base de datos puede brindarnos. Para hacer uso de esta técnica debemos definir una función de terminación temprana y una para el almacenamiento del mejor modelo, luego enviarlas a la función de entrenamiento.

En el caso de Keras la librería nos ofrece la opción de usar la función "EarlyStopping" y "ModelCheckpoint", la primera establece monitoreo sobre una de las variables de evaluación y si no hay mejoría en n iteraciones el entrenamiento se da por concluido. En la segunda función al definir una variable de evaluación y la dirección de un directorio podemos almacenar el mejor modelo generado durante el entrenamiento dependiendo de la variable.

```
from keras.callbacks import EarlyStopping, ModelCheckpoint
top_weights_path = os.path.join(os.path.abspath(model_path), '
    top_model_weights.h5')
callbacks_list = [
    EarlyStopping(monitor='val_acc', patience=5, verbose=0),
```

```
    ModelCheckpoint(top_weights_path, monitor='val_acc', verbose=1,
    save_best_only=True),
]
model.fit(...,
      callbacks=callbacks_list,
      ...)
```

CÓDIGO 4.6: Lista de funciones «callback» que permiten almacenar el mejor modelo según un parámetro de evaluación y detener el entrenamiento cuando se observe que no hay mejoras.

El modelo generado puede ser utilizado mediante el uso de la función "load_model" de la librería Keras en cualquier programa desarrollado en Python. Este modelo es completamente funcional, puede ser entrenado, utilizado para calcular predicciones e incluso modificado en su arquitectura.

```
from keras.models import load_model
model = load_model('models/top_model_weights.h5')
```

CÓDIGO 4.7: Carga de un modelo generado en Keras de formato h5 para su uso en Python.

Para utilizar el modelo en un entorno móvil debemos hacer algunos cambios antes de poder exportar el modelo, y esto se debe a que el model Inception-v3 en Keras hace uso de algunas capas con funciones no soportadas en la librería móvil. Primero debemos definir la etapa de aprendizaje a modo de prueba, esto se hace mediante la función "set_learning_phase(0)", luego debemos inicializar una sesión y cargar el modelo generado en el entrenamiento.

```
K.set_learning_phase(0)

sess = tf.Session()
K.set_session(sess)

model = load_model('models/top_model_weights.h5')
```

CÓDIGO 4.8: Definición de etapa de prueba inicialización de sesión y carga de modelo entrenado.

Como los modelos en la versión móvil de Tensorflow no pueden ser entrenados y no tienen soporte para cargar modelos en entrenamiento también

debemos hacer nuestro modelo con valores constantes, a este proceso en Tensorflow se le llama «freeze». Para hacer esto debemos hacer uso de la función "convert_variables_to_constants" de Tensorflow. Una vez el modelo ha sido convertido a constante podemos exportarlo como un archivo.

```
output_graph_def = graph_util.convert_variables_to_constants(sess,
    sess.graph.as_graph_def(), ['dense_1/Softmax'])

with tf.gfile.GFile('android_model.pb', "wb") as f:
    f.write(output_graph_def.SerializeToString())
```

CÓDIGO 4.9: Conversión del modelo a constante y exportación del mismo.

4.4. Visualización de resultados

El enfoque de este proyecto de tesis es generar un modelo y ejecutarlo en la plataforma Android. Para ello es necesario desarrollar una interfaz gráfica que permita la visualización de los resultados generados por el sistema de inferencias de Tensorflow utilizando el modelo de redes neuronales y la cámara.

En la imagen 4.2 podemos ver de manera sencilla como mediante la cámara del dispositivo generamos predicciones de las familias de macroinvertebrados en tiempo real.

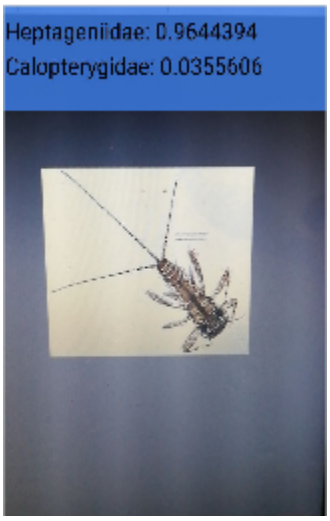


FIGURA 4.2: Pantalla de prototipo de prueba en Android.

Capítulo 5

Discusión General: Contribuciones y Trabajos Futuros

5.1. Contribuciones

En esta tesis se exploró la aplicación de las redes neuronales convolucionales en un problema de reconocimiento de organismos macroinvertebrados agrupados en cuatro conjuntos. Se escogieron los organismos con el objetivo de hallar los posibles resultados en conjuntos diferentes. Adicionalmente se estudió de manera comparativa el uso de dos técnicas (aumentación de datos y ajuste fino) en los diferentes conjuntos de datos y comprobar los beneficios que ofrecen. Los mayores descubrimientos obtenidos durante la práctica fueron los siguientes:

1. Es posible desarrollar una herramienta de reconocimiento de imágenes de macroinvertebrados utilizando una base de datos de reducido volumen. Esto es posible al aplicar técnicas de regularización.

2. Los mejores resultados en la herramienta de reconocimiento fueron obtenidos al usar ajuste fino en la capas finales del modelo convolucional

y un módulo de aumentación de datos.

3. Para obtener un buen modelo es necesario balancear el volumen de imágenes entre las diferentes clases en lo posible o ponderar el entrenamiento en las clases con menor volumen, esto se puede apreciar en el elevado valor obtenido en el coeficiente κ de Cohen.

4. Las técnicas de regularización aumentan las capacidades del modelo, pero no son suficientes para lograr máxima precisión con una base de datos pobre.

Tomando estos resultados queda demostrado que nuestro sistema es funcional a un nivel que facilita en gran medida el trabajo de las Juntas Comunales para la identificación de los macroinvertebrados. Podemos incluso sugerir que la aplicación puede ser de gran ayuda en los centros de investigación.

El esfuerzo realizado en esta tesis está lejos de brindar soporte al proyecto finalizado y se requiere de dedicación para la formación de la base de datos con todas las familias de macroinvertebrados y la integración a la aplicación Android que busca utilizar el Instituto en el proyecto con las Juntas Comunales.

Los resultados de esta tesis vistos en contexto pueden ser utilizados a futuro como la base para el desarrollo de otros proyecto de reconocimiento similares dirigidos a dispositivos móviles como Android, iOS y RaspberryPi.

Posteriormente también es interesante conocer el impacto ambiental en los sistemas hídricos en la geografía panameña, lo cual ahora es facilitado mediante el uso de las Tecnologías de Información y Comunicación.

5.2. Limitaciones

Los modelos generados durante la fase de entrenamiento demostraron buenos porcentajes en muchos casos, pero aún así al compararlo con el conocimiento de un experto se quede corto y no se logren resultados superhumanos. Una de las limitaciones que impide este logro está en el reducido tamaño de la base de datos, pero otra de las limitaciones proviene del tipo de datos, muchas de las familias que comprende el estudio se diferencian con datos como su tamaño, en caso de las conchas por la dureza de sus caparazones.

Otro factor que reduce en cierto porcentaje la precisión es la calidad de la base de datos. Esto incluye baja resolución en algunas imágenes, la presencia de elementos no relacionados como hojas, manos o rocas; también es posible encontrar imágenes que no presentan datos suficientes sobre el macroinvertebrado y que no es posible hacer una predicción con un alto grado de certeza.

5.3. Trabajos Futuros

Finalizado el prototipo generado para este documento se trabajará en la integración con la aplicación Android dedicada al proyecto de las Juntas Comunales, la cual incluye funciones para el registro y geolocalización de las familias de macroinvertebrados conectándose a un servidor web.

A largo plazo debido a lo exitoso del trabajo prototipo realizado en esta investigación este proyecto seguirá su desarrollo con el apoyo del equipo del Instituto Conmemorativo Gorgas de Estudios de la Salud (ICGES) con la

generación de una base de datos robusta que incluya todas las familias de macroinvertebrados que considera el proyecto (81 familias). Este documento servirá de apoyo para facilitar el uso del algoritmo de entrenamiento a medida que se escala el proyecto.

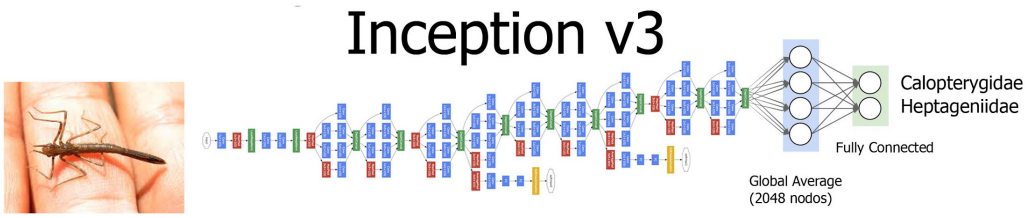
Parte de la estrategia a futuro es generar un «script» de fácil uso para la ejecución inmediata y posiblemente automatizada de la fase de entrenamiento a medida que se van agregando nuevos elementos a la base de datos. Y cada cierto tiempo actualizar el modelo en los teléfonos que utilicen el sistema.

Apéndice A

Estructura de la Aplicación

A.1. Grafo del modelo de red convolucional

El modelo utilizado para el proyecto consiste en una modificación en las capas finales de la arquitectura Inception-v3. El modelo original para ILSVRC se le recorta la capa completamente conectada y se reemplaza por una capa completamente conectada de tamaño N, haciendo referencia al número de clases a identificar. Adicionalmente se aplica una capa «pooling» a 2048 nodos, para reducir la carga computacional de entrenamiento entre la salida del modelo convolucional y la capa completamente conectada.



A.2. Código del modelo

A.2.1. Módulo de carga y generación de «dataset»

```
import cv2
from skimage import img_as_float

import os
import glob

import numpy as np
import tables
import datetime
from random import shuffle
```

CÓDIGO A.1: Librerías de mayor utilidad para el manejo de las imágenes y la generación de los archivos del «dataset»

```

class LocalDataLoader:

    def __init__(self, img_dim=(224,224), dataset_path = '',
    label_path = '',raw_path = '.data/', channels = 3, val_ratio
    =0.3, test_ratio=0.1):
        d = datetime.datetime.today()
        self.log_date = d.strftime('%d-%m-%Y_%H:%M:%S')

        self.img_dim = img_dim
        self.img_dim_str = '{}x{}'.format(img_dim[0], img_dim[1])
        if not dataset_path:
            self.dataset_path = './dataset/{}_{}.hdf5'.format(self.
    log_date, self.img_dim_str)
        else:
            self.dataset_path=dataset_path

        if not label_path:
            self.label_path = './labels/' + self.log_date
        else:
            self.label_path = label_path


        self.raw_path = raw_path
        self.img_dtype = tables.Float32Atom()
        self.data_shape = (0, img_dim[0], img_dim[0], channels)
        self.val_ratio = val_ratio
        self.test_ratio = test_ratio
        self.train_ratio = 1 - val_ratio - test_ratio

```

```
28      # Función para cargar un dataset o iniciar proceso de generació
    n
29      def load_data(self):
30          if not os.path.isfile(self.dataset_path):
31              print("Dataset not found.")
32              print("Creating dataset ...")
33              self.create_dataset()
34          else:
35              print("Using existing dataset.")
36
37          with open(self.label_path) as f:
38               labels_name = f.read().split()
39          nb_classes = len(labels_name)
40
41          # Carga del archivo
42          h5f = tables.open_file(self.dataset_path, mode='r')
43          # Cargando conjunto de entrenamiento
44          # Carga de las imágenes
45          train_X = h5f.root.train_X[:]
46          train_y = h5f.root.train_y[:]
47
48          batch_size = h5f.root.train_y.shape[0]
49
50          # Conversión de las etiquetas a su formato one hot
51          train_y_one_hot = np.zeros((batch_size, nb_classes), dtype=
    np.int)
52          train_y_one_hot[np.arange(batch_size), train_y] = 1
53
54          # Carga de conjunto de validación
55          val_X = h5f.root.val_X[:]
56          val_y = h5f.root.val_y[:]
57
58          batch_size = h5f.root.val_y.shape[0]
59          val_y_one_hot = np.zeros((batch_size, nb_classes), dtype=np
    .int)
60          val_y_one_hot[np.arange(batch_size), val_y] = 1
61
62          # Carga de conjunto de prueba
63
64          test_X = h5f.root.test_X[:]
65          test_y = h5f.root.test_y[:]
66
67          batch_size = h5f.root.test_y.shape[0]
68          test_y_one_hot = np.zeros((batch_size, nb_classes), dtype=
    np.int)
69          test_y_one_hot[np.arange(batch_size), test_y] = 1
70
71          return train_X, train_y_one_hot, val_X, val_y_one_hot,
    test_X, test_y_one_hot, labels_name
```

```
72
73      # Módulo de creación de dataset utilizando un directorio con
        subdirectorios para cada clase
74      def create_dataset(self):
75          class_names = os.listdir(self.raw_path)
76
77          # Evaluando cantidad de imágenes por clase. De ser muy baja
         se ignora
78          def directory_files_filter(class_name, len_limit=2):
79              l = len(os.listdir(self.raw_path + class_name + '/'))
80              if l * self.test_ratio > len_limit and l * self.
        val_ratio > len_limit:
81                  return True
82              else:
83                  print("{} too few images. only {}".format(
        class_name, l))
84              return False
85
86          print("Filtering...")
87          class_names = list(filter(lambda x: directory_files_filter(
        x), class_names))
88          print(class_names)
89
90          # Obtención de dirección de todas las imágenes
91          files = []
92          labels = []
93
94          for i, class_name in enumerate(class_names):
95              class_dir = '{}{}/'.format(self.raw_path, class_name)
96              class_files = glob.glob(class_dir + '*.jpg')
97
98              files.extend(class_files)
99              labels.extend([i for f in class_files])
100
101         c = list(zip(files, labels))
102         # Mezcla aleatoria de las imágenes para distribución en el
        dataset
103         shuffle(c)
104         files, labels = zip(*c)
105
106
107         # División a subconjuntos
108         train_nb = int(self.train_ratio * len(files))
109         val_nb = train_nb + int(self.val_ratio * len(files))
110
111         train_paths  = files[0:train_nb]
112         train_labels = labels[0:train_nb]
113
114         val_paths    = files[train_nb:val_nb]
```

```
115         val_labels   = labels[train_nb:val_nb]
116
117         test_paths    = files[val_nb:]
118         test_labels   = labels[val_nb:]
119
120         # Generación de archivo dataset
121         FILTERS = tables.Filters(complib='zlib', complevel = 5)
122         h5f = tables.open_file(self.dataset_path, mode='w', filters
    =FILTERS)
123
124         # Inclusión de las etiquetas
125         h5f.create_array(h5f.root, 'train_y', train_labels)
126         h5f.create_array(h5f.root, 'val_y', val_labels)
127         h5f.create_array(h5f.root, 'test_y', test_labels)
128
129         # Función para procesamiento de imágenes
130         def path2data(paths, storage):
131             dataset_size = float(len(paths))
132             for i in range(len(paths)):
133                 if i %50 == 0:
134                     print('image {}/{}'.format(i, len(paths)))
135                 path = paths[i]
136                 img = cv2.imread(path)
137                 img = cv2.resize(img, self.img_dim, interpolation=
    cv2.INTER_CUBIC)
138                 img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
139                 img = img_as_float(img)
140                 storage.append(img[None])
141
142         # Inclusión de las imágenes
143         train_X = h5f.create_earray(h5f.root, 'train_X', self.
    img_dtype, shape=self.data_shape)
144         val_X = h5f.create_earray(h5f.root, 'val_X', self.img_dtype
    , shape=self.data_shape)
145         test_X = h5f.create_earray(h5f.root, 'test_X', self.
    img_dtype, shape=self.data_shape)
146
147         print("Saving dataset... ")
148         print("--Training set--")
149         path2data(train_paths, train_X)
150         print("--Validation set--")
151         path2data(val_paths, val_X)
152         print("--Testing set--")
153         path2data(test_paths, test_X)
154
155         # Se cierra el archivo dataset
156         h5f.close()
157
158         print("Dataset saved... ")
```

```
159
160         # Se genera un archivo que contiene el nombre de las clases
        según los subdirectorios
161         with open(self.label_path, 'w') as f:
162             for name in class_names:
163                 f.write(name + '\n')
```

CÓDIGO A.2: Módulo para creación de «dataset»

```
1 # Para cargar un dataset definido.
2 loader = LocalDataLoader(dataset_path='./dataset/29-12-2017_17
      :35:28_299x299.hdf5', label_path='./label_names/29-12-2017_17
      :35:28')
3
4 # Generación de nuevo dataset
5 loader = LocalDataLoader(img_dim=(DIM, DIM), raw_path=raw_path,
      dataset_path=dataset_path, label_path=label_path)
6
7 X_train, y_train, X_validation, y_validation, X_test, y_test,
      class_labels, dataset_mean = loader.load_data()
8 nb_classes = len(class_labels)
```

CÓDIGO A.3: Utilización del módulo.

A.2.2. Generación del modelo

```
1 from keras.layers import Dense, GlobalAveragePooling2D, Flatten,
      Dropout
2 from keras.preprocessing.image import array_to_img, img_to_array,
      load_img
3 from keras.applications.inception_v3 import InceptionV3
4 from keras.models import Model
```

CÓDIGO A.4: Carga de librerías para la generación del modelo de redes neuronales.

```
1 def generate_model(classes, shape = (299,299,3) ):
2     # Modelo InceptionV3 entrenado para imagenet, sin las capas
      finales
3     base_model = InceptionV3(input_shape=shape, weights='imagenet',
       include_top=False)
4     # Agregando capas finales
5
6     # Usamos GlobalAveragePooling2D en vez de Flatten para reducir
      complejidad y prevenir sobreajuste
7     # Con Flatten el modelo sería 64 veces más grande
8     x = base_model.output
9     x = GlobalAveragePooling2D()(x)
10    # Agregamos capa completamente conectada con su activador relu
```

```
    x = Dense(1024, activation='relu')(x)
    # Agregamos regularización on un Dropout al 50%
    x = Dropout(0.5)(x)
    # Y nuestro clasificador
    predictions = Dense(classes, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=predictions)

    # Congelamos los valores en las capas que conforman el modelo
    for layer in base_model.layers:
        layer.trainable = False
    return model
```

CÓDIGO A.5: Función para la creación de modelo utilizando Keras y el modelo previamente entrenado InceptionV3 para imagenet.

```
model = generate_model(classes=nb_classes, shape=(DIM, DIM, 3))
model.summary()
```

CÓDIGO A.6: Generación del modelo e impresión de las capas que lo conforman.

A.2.3. Compilación y ejecución de modelo básica

```
from keras.metrics import top_k_categorical_accuracy
from sklearn.metrics import cohen_kappa_score
import keras.backend as K
import tensorflow as tf

# Precisión para 3 mejores elementos en predicción
def top_3_accuracy(y_true, y_pred):
    return top_k_categorical_accuracy(y_true, y_pred, k=3)

# Precisión para 5 mejores elementos en predicción
def top_5_accuracy(y_true, y_pred):
    return top_k_categorical_accuracy(y_true, y_pred, k=5)

# Coeficiente kappa de Cohen
def ckm(y_true, y_pred):
    labels = K.argmax(y_true, axis=-1)
    pred = K.argmax(y_pred, axis=-1)

    confusion_matrix = tf.confusion_matrix(labels, pred)

    total = tf.to_float(tf.reduce_sum(confusion_matrix))

    agreement = tf.diag_part(confusion_matrix)
```

```
    agree_sum = tf.to_float(tf.reduce_sum(agreement))

    row_sum = tf.reduce_sum(confusion_matrix, 0)
    column_sum = tf.reduce_sum(confusion_matrix, 1)

    prod_sum = tf.to_float(tf.multiply(row_sum, column_sum))
    by_chance = prod_sum / total
    bch_sum = tf.reduce_sum(by_chance)

    kappa = (agree_sum - bch_sum) / (total - bch_sum)
    return kappa
```

CÓDIGO A.7: Generación de métodos de evaluación personalizados.

```
from keras.optimizers import Adam
# Al compilar se pueden definir el valor de error, el optmizador y las metricas de evaluación.
model.compile(loss="categorical_crossentropy",
                  optimizer="adam",
                  metrics=[ckm, 'accuracy', top_3_accuracy, top_5_accuracy])
```

CÓDIGO A.8: Función para compilación del modelo.

```
from keras.callbacks import ModelCheckpoint
from sklearn.utils import class_weight

# Definición de ponderación por clases según número de instancias
decoded_y = np.array(
    [np.where(r==1)[0][0] for r in y_train])
class_weight = class_weight.compute_class_weight('balanced', np.unique(decoded_y), decoded_y)

# Función de almacenamiento de los modelos según mayor valor de precisión
callbacks_list = [
    ModelCheckpoint(top_weights_path, monitor='val_acc', verbose=1, save_best_only=True),
]
# Nombramiento de los modelos
model_path = './models'

top_weights_path = os.path.join(os.path.abspath(model_path), 'top-noDA-exp{}.h5'.format(exp))
top_weights_path = os.path.join(os.path.abspath(model_path), "top-noDA-exp"+exp+"-{epoch:02d}-{val_acc:.2f}.hdf5")

batch_size = 128
epochs = 100
```

```
21
22 # Entrenamiento del modelo
23 # Las metricas son almacenadas en la variable history para luego
       graficar
24 history = model.fit(X_train, y_train,
25           batch_size=batch_size,
26           epochs=epochs,
27           verbose=1,
28           validation_data=(X_validation, y_validation),
29           class_weight = class_weight,
30           callbacks = callbacks_list)
```

CÓDIGO A.9: Entrenamiento del modelo.

A.2.4. Aumentación de datos

```
1 # Librería utilizada está en https://github.com/aleju/imgaug
2 import imgaug as ia
3 from imgaug import augmenters as iaa
4
5 # Definición de %de probabilidad
6 sometimes = lambda aug: iaa.Sometimes(0.5, aug)
7
8 # Lista de aumentadores
9 augmenters = [
10
11     # Corta las imágenes a un -5% y 10% de su anchura y altura
12     sometimes(iaa.CropAndPad(
13         percent=(-0.05, 0.1),
14         pad_mode=ia.ALL,
15         pad_cval=(0, 255)
16     )),
17     # Distorciones
18     sometimes(iaa.Affine(
19         scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
20         translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)},
21         rotate=(-45, 45),
22         shear=(-16, 16),
23         order=[0, 1],
24         cval=(0, 255),
25         mode=ia.ALL
26     )),
27     # Utiliza una de las trasnformaciones en la lista a la vez. Se
       puede aumentar para aumentación más agresiva
28     iaa.SomeOf((0, 1),
29         [
30             sometimes(iaa.Superpixels(p_replace=(0, 0.25),
       n_segments=(20,200))), # aplica algoritmo superpixel
31             # Aplica uno de los algoritmos de difuminado
```

```
32            iaa.OneOf([
33                iaa.GaussianBlur((0, 3.0)),
34                iaa.AverageBlur(k=(2, 7)),
35                iaa.MedianBlur(k=(3, 11)),
36            ]),
37
38            iaa.Sharpen(alpha=(0, 1.0), lightness=(0.75, 1.5)), # resalta los bordes
39
40            iaa.Emboss(alpha=(0, 1.0), strength=(0, 2.0)), # emboss images
41
42            iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.05*255), per_channel=0.5), # aplica algoritmo de ruido gaussiano
43
44            iaa.OneOf([
45                iaa.Dropout((0.01, 0.1), per_channel=0.5), # Remueve hasta 10% de los pixeles de la imagen
46                iaa.CoarseDropout((0.03, 0.15), size_percent=(0.02, 0.05), per_channel=0.2), # Remueve pixeles en paquetes cuadrados
47            ]),
48
49            iaa.Add((-10, 10), per_channel=0.5), # modificador de brillo
50
51            iaa.AddToHueAndSaturation((-20, 20)), # modificador de tono y saturación
52
53            # Cambia el brillo de toda la imagen o en subareas
54            iaa.OneOf([
55                iaa.Multiply((0.5, 1.5), per_channel=0.5),
56                iaa.FrequencyNoiseAlpha(
57                    exponent=(-4, 0),
58                    first=iaa.Multiply((0.5, 1.5), per_channel=True),
59                    second=iaa.ContrastNormalization((0.5, 2.0))
60                )
61            ]),
62
63            iaa.ContrastNormalization((0.5, 2.0), per_channel=0.5), # aumenta o reduce el contraste
64
65            iaa.Grayscale(alpha=(0.0, 1.0)), # reduce colores
66
67            sometimes(iaa.ElasticTransformation(alpha=(0.5, 3.5), sigma=0.25)), # mueve pixeles individualmente alrededor de campos de distorción
68
```

```
69              sometimes(iaa.PiecewiseAffine(scale=(0.01, 0.05))), #
    coloca una malla de  puntos y luego mueve los puntos formando
    distorción
70
71              sometimes(iaa.PerspectiveTransform(scale=(0.01, 0.1)))
    # Aplica transformación de perspectiva 4 puntos de manera
    aleatoria
72          ],
73          random_order=True
74      )
75
76 ]
77
78 # Secuenciador encargado de aumentar las imágenes
79 seq_no_flip = iaa.Sequential(augmenters,
80     random_order=True)
81 # Se agregan las transformaciones de volteado
82 augmenters.extend(
83     [
84         iaa.Fliplr(0.5), # horizontally flip 50% of all images
85         iaa.Flipud(0.2), # vertically flip 20% of all images
86     ]
87 )
88 # Secuenciador con volteado
89 seq_with_flip = iaa.Sequential(augmenters,
90     random_order=True)
```

CÓDIGO A.10: Utilización de librería de aumentación imgaug por Alexander Jung.

```
1 # Para esta lista de clases
2 #['heptageniidae', 'ancylidae', 'thiaridae', 'lymnaeidae', '
    corbiculidae', 'planorbidae', 'gomphidae', 'leptophlebiidae', '
    hydrobiidae', 'physidae', 'ampullariidae', 'neritidae', '
    calopterygidae', 'sphaeriidae']
3 flip_aug_per_class = [1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1]
4 seq_per_class = list(map(lambda flip: seq_with_flip if flip else
    seq_no_flip, flip_aug_per_class))
```

CÓDIGO A.11: Definición de aumentación con volteado.

```
1 import time
2 # Generación de bloque de imágenes usando o no aumentación
3 def batch_generator(X, y, batch_size, seq=None):
4     start = time.time()
5
6     # Elige de manera aleatoria las imágenes en el dataset
7     batch_indices = np.random.randint(0, X.shape[0], (batch_size))
8
9     X_batch = np.array(X[batch_indices])
```

```
    y_batch = np.array(y[batch_indices])

    decoded_y = np.array([np.where(r==1)[0][0] for r in y_batch])

    # Bloque de aumentación
    X_augmented_batch = []
    y_augmented_batch = []

    for i in range(CLASSES):
        class_indices = np.where(decoded_y == i)
        if not np.any(class_indices):
            continue

        X_batch_class = X_batch[class_indices]
        y_batch_class = y_batch[class_indices]

        # Si se tiene aumentación se aplica
        if seq:
            X_batch_class = img_as_ubyte(X_batch_class)
            X_batch_class = seq[i].augment_images(X_batch_class)
            X_batch_class = img_as_float(X_batch_class)

        X_augmented_batch.extend(X_batch_class)
        y_augmented_batch.extend(y_batch_class)

    # Se convierte a arraglo numpy
    X_augmented_batch = np.array(X_augmented_batch)
    y_augmented_batch = np.array(y_augmented_batch)

    end = time.time()
    print('batch_generator: {:f}s'.format(end - start))
    return (X_augmented_batch, y_augmented_batch)
```

CÓDIGO A.12: Función para generar los bloques de entrenamiento.

```
from sklearn.utils import class_weight
NB_BATCHES = 100
history = []

model_path = './models'
max_val = 0.0

# Entrenamiento por bloques
for batch_idx in range(NB_BATCHES):
    print('Training {}/{}'.format(batch_idx + 1, NB_BATCHES))

    X,y = batch_generator(X_train, y_train, BATCH_SIZE,
    seq_per_class)

```

```
    decoded_y = np.array(
        [np.where(r==1)[0][0] for r in y])
    c_weight = class_weight.compute_class_weight('balanced', np.
    unique(decoded_y), decoded_y)

    batch_history = model.fit(X, y, epochs=1, class_weight=c_weight
    , verbose=1,
                        validation_data=(X_validation, y_validation)
    ,)

    val_acc = batch_history.history['val_acc'][0]

    # Si el valor de validación es el mayor se hace una copia de
    los pesos del modelo
    if max_val < val_acc:
        max_val = val_acc
        top_weights_path = os.path.join(os.path.abspath(model_path)
    , "top-DA-exp{}-{}-{:.2f}.hdf5".format(exp, batch_idx, val_acc))
        print(top_weights_path)
        model.save(top_weights_path)


    keys = batch_history.history.keys()

    # Se guarda el historial de entrenamiento
    if batch_idx == 0:
        history = batch_history
    else:
        for k in keys:
            history.history[k].extend(batch_history.history[k])
```

CÓDIGO A.13: a.

A.2.5. Graficado y evaluación del modelo

```
# Muestra el nombre de los elementos en el diccionario
print(history.history.keys())

# Fácil extracción de los valores para almacenamiento en una matriz
def history2matrix(history):
    train_history = [history.history['acc'],
      history.history['top_3_accuracy'],
      history.history['top_5_accuracy'],
      history.history['ckm'],
      history.history['loss']]

    val_history = [history.history['val_acc'],
      history.history['val_top_3_accuracy'],
      history.history['val_top_5_accuracy'],
```

```
      history.history['val_ckm'],
      history.history['val_loss']]

    return train_history, val_history

train_history, val_history = history2matrix(history)

# Almacenamiento y utilización de historial en archivo
import json
with open("noDA-history-exp"+exp+".txt", "w") as outfile:
    json.dump({'training': train_history, 'validation': val_history
    }, outfile)

with open("noDA-history-exp"+exp+".txt", "r") as infile:
    a = json.load(infile)
```

CÓDIGO A.14: Almacenamiento de datos de entrenamiento.

```
import matplotlib.pyplot as plt
# Generación de distribución de gráficos
def ax_generator(n_graphs):
    if n_graphs == 3:
        plt.figure(figsize=(20,5))
        ax1 = plt.subplot2grid(shape=(1,3), loc=(0,0))
        ax2 = plt.subplot2grid((1,3), (0,1))
        ax3 = plt.subplot2grid((1,3), (0,2))

        return [ax1,ax2,ax3]

    if n_graphs == 4:
        plt.figure(figsize=(22,8))
        ax1 = plt.subplot2grid(shape=(2,4), loc=(0,0))
        ax2 = plt.subplot2grid((2,4), (0,1))
        ax3 = plt.subplot2grid((2,4), (1,0))
        ax4 = plt.subplot2grid((2,4), (1,1))

        return [ax1,ax2,ax3,ax4]

    if n_graphs == 5:
        plt.figure(figsize=(20,10))
        ax1 = plt.subplot2grid(shape=(2,6), loc=(0,0), colspan=2)
        ax2 = plt.subplot2grid((2,6), (0,2), colspan=2)
        ax3 = plt.subplot2grid((2,6), (0,4), colspan=2)
        ax4 = plt.subplot2grid((2,6), (1,1), colspan=2)
        ax5 = plt.subplot2grid((2,6), (1,3), colspan=2)

        return [ax1,ax2,ax3,ax4,ax5]

    return []

```

```
# Generación de gráficos dado los valores de entrenamiento y validación
def generate_graph(acc, val_acc, title):
    axarr = ax_generator(len(acc))
    plt.tight_layout(pad=0, w_pad=5.0, h_pad=5.0)
    plt.rcParams['xtick.labelsize'] = 18
    plt.rcParams['ytick.labelsize'] = 18

    for i,ax in enumerate(axarr):
        ax.plot(acc[i])
        ax.plot(val_acc[i])
        ax.set_title(title[i], fontsize=22)
        ax.set_xlim(0,len(acc[i]))
        if i != len(axarr)-1:
            ax.set_ylim(0,1)

        ax.set_ylabel('precisión', fontsize=18)
        ax.set_xlabel('generación', fontsize=18)

        if i != len(axarr)-1:
            ax.legend(['entrenamiento', 'validación'], loc='lower right', fontsize=18)
        else:
            ax.legend(['entrenamiento', 'validación'], loc='upper right', fontsize=18)

    plt.show()

title = ['Precisión top-1', 'Precisión top-3', 'Precisión top-5', 'kappa de Cohen', 'Error']
generate_graph(train_history, val_history, title)
```

CÓDIGO A.15: Funciones para generación de gráficos.

```
# Carga de los valores del mejor modelo
model.load_weights('top-model-exp1.hdf5')

# Evaluación del modelo
scores = model.evaluate(X_test, y_test, verbose=2)
```

CÓDIGO A.16: Evaluación del mejor modelo utilizando el conjunto de prueba.

A.2.6. Exportación del modelo entrenado

```
K.set_learning_phase(0)

sess = tf.Session()
K.set_session(sess)
```

```
5
6 model = load_model('models/top_model_weights.h5')
```

CÓDIGO A.17: Definición de etapa de prueba inicialización de sesión y carga de modelo entrenado.

```
1 output_graph_def = graph_util.convert_variables_to_constants(sess,
      sess.graph.as_graph_def(), ['dense_1/Softmax'])
2
3 with tf.gfile.GFile('android_model.pb', "wb") as f:
4     f.write(output_graph_def.SerializeToString())
```

CÓDIGO A.18: Conversión del modelo a constante y exportación del mismo.

Apéndice B

Ejecución del modelo

B.1. Código de la aplicación Android

B.1.1. Importación de modelo

```
allprojects {
    repositories {
        jcenter()
    }
}
dependencies {
    compile 'org.tensorflow:tensorflow-android:+'
}
```

CÓDIGO B.1: Inclusión de Tensorflow en aplicación Android en el Gradle.

```
@Override
  public void onPreviewSizeChosen(final Size size, final int
   rotation) {
    final float textSizePx = TypedValue.applyDimension(
        TypedValue.COMPLEX_UNIT_DIP, TEXT_SIZE_DIP, getResources().
   getDisplayMetrics());
    borderedText = new BorderedText(textSizePx);
    borderedText.setTypeface(Typeface.MONOSPACE);

    classifier =
        TensorFlowImageClassifier.create(
            getAssets(),
            MODEL_FILE,
            LABEL_FILE,
            INPUT_SIZE,
```

```
14                IMAGE_MEAN,
15                IMAGE_STD,
16                INPUT_NAME,
17                OUTPUT_NAME);
18
19        previewWidth = size.getWidth();
20        previewHeight = size.getHeight();
21
22        final Display display = getWindowManager().getDefaultDisplay();
23        final int screenOrientation = display.getRotation();
24
25        LOGGER.i("Sensor orientation: %d, Screen orientation: %d",
          rotation, screenOrientation);
26
27        sensorOrientation = rotation + screenOrientation;
28
29        LOGGER.i("Initializing at size %dx%d", previewWidth,
          previewHeight);
30        rgbFrameBitmap = Bitmap.createBitmap(previewWidth,
          previewHeight, Config.ARGB_8888);
31        croppedBitmap = Bitmap.createBitmap(INPUT_SIZE, INPUT_SIZE,
          Config.ARGB_8888);
32
33        frameToCropTransform = ImageUtils.getTransformationMatrix(
34            previewWidth, previewHeight,
35            INPUT_SIZE, INPUT_SIZE,
36            sensorOrientation, MAINTAIN_ASPECT);
37
38        cropToFrameTransform = new Matrix();
39        frameToCropTransform.invert(cropToFrameTransform);
40
41        yuvBytes = new byte[3][];
42
43        addCallback(
44            new DrawCallback() {
45              @Override
46              public void drawCallback(final Canvas canvas) {
47                renderDebug(canvas);
48              }
49            });
50    }
```

CÓDIGO B.2: Definición del modelo

```
1 protected void processImageRGBbytes(int[] rgbBytes ) {
2        rgbFrameBitmap.setPixels(rgbBytes, 0, previewWidth, 0, 0,
         previewWidth, previewHeight);
3        final Canvas canvas = new Canvas(croppedBitmap);
4        canvas.drawBitmap(rgbFrameBitmap, frameToCropTransform, null);
5
```

```
    // For examining the actual TF input.
    if (SAVE_PREVIEW_BITMAP) {
      ImageUtils.saveBitmap(croppedBitmap);
    }
    runInBackground(
        new Runnable() {
          @Override
          public void run() {
            final long startTime = SystemClock.uptimeMillis();
            final List<Classifier.Recognition> results = classifier
    .recognizeImage(croppedBitmap);
            lastProcessingTimeMs = SystemClock.uptimeMillis() -
    startTime;
            LOGGER.i("Detect: %s", results);
            cropCopyBitmap = Bitmap.createBitmap(croppedBitmap);
            if (resultsView==null) {
              resultsView = (ResultsView) findViewById(R.id.results
    );
            }
            resultsView.setResults(results);
            requestRender();
            computing = false;
            if (postInferenceCallback != null) {
              postInferenceCallback.run();
            }
          }
        });
  }
```

B.1.2. Ejecución del modelo

```
public static Classifier create(
      AssetManager assetManager,
      String modelFilename,
      String labelFilename,
      int inputSize,
      int imageMean,
      float imageStd,
      String inputName,
      String outputName) {
    TensorFlowImageClassifier c = new TensorFlowImageClassifier();
    c.inputName = inputName;
    c.outputName = outputName;

    // Read the label names into memory.
    // TODO(andrewharp): make this handle non-assets.
    String actualFilename = labelFilename.split("file:///
    android_asset/")[1];
```

```
17      Log.i(TAG, "Reading labels from: " + actualFilename);
18      BufferedReader br = null;
19      try {
20        br = new BufferedReader(new InputStreamReader(assetManager.
        open(actualFilename)));
21        String line;
22        while ((line = br.readLine()) != null) {
23          c.labels.add(line);
24        }
25        br.close();
26      } catch (IOException e) {
27        throw new RuntimeException("Problem reading label file!" , e)
        ;
28      }
29
30      c.inferenceInterface = new TensorFlowInferenceInterface(
        assetManager, modelFilename);
31
32      // The shape of the output is [N, NUM_CLASSES], where N is the
        batch size.
33      final Operation operation = c.inferenceInterface.graphOperation
        (outputName);
34      final int numClasses = (int) operation.output(0).shape().size
        (1);
35      Log.i(TAG, "Read " + c.labels.size() + " labels, output layer
        size is " + numClasses);
36
37      // Ideally, inputSize could have been retrieved from the shape
        of the input operation.  Alas,
38      // the placeholder node for input in the graphdef typically
        used does not specify a shape, so it
39      // must be passed in as a parameter.
40      c.inputSize = inputSize;
41      c.imageMean = imageMean;
42      c.imageStd = imageStd;
43
44      // Pre-allocate buffers.
45      c.outputNames = new String[] {outputName};
46      c.intValues = new int[inputSize * inputSize];
47      c.floatValues = new float[inputSize * inputSize * 3];
48      c.outputs = new float[numClasses];
49
50      return c;
51    }
```

CÓDIGO B.3: Creación del objeto manejador y la definición del la interfaz para hacer uso del modelo.

```
1  @Override
2    public List<Recognition> recognizeImage(final Bitmap bitmap) {
```

```
3     // Preprocess the image data from 0-255 int to normalized float
      based
4     // on the provided parameters.
5     bitmap.getPixels(intValues, 0, bitmap.getWidth(), 0, 0, bitmap.
      getWidth(), bitmap.getHeight());
6
7     imageMean = 0;
8     imageStd = (float) 255.0;
9     for (int i = 0; i < intValues.length; ++i) {
10      final int val = intValues[i];
11      floatValues[i * 3 + 0] = (((val >> 16) & 0xFF) - imageMean) /
      imageStd;
12      floatValues[i * 3 + 1] = (((val >> 8) & 0xFF) - imageMean) /
      imageStd;
13      floatValues[i * 3 + 2] = ((val & 0xFF) - imageMean) /
      imageStd;
14    }
15
16    int val = (intValues[0] >> 16) & 0xFF;
17    float fval = (float) (((val) -0 )/255.0);
18    Log.i("test", "Reading labels from: " + val +", "+ fval);
19
20    // Copy the input data into TensorFlow.
21    inferenceInterface.feed(inputName, floatValues, 1, inputSize,
      inputSize, 3);
22
23    // Run the inference call.
24    inferenceInterface.run(outputNames, logStats);
25
26    // Copy the output Tensor back into the output array.
27    inferenceInterface.fetch(outputName, outputs);
28
29    // Find the best classifications.
30    PriorityQueue<Recognition> pq =
31        new PriorityQueue<Recognition>(
32            3,
33            new Comparator<Recognition>() {
34              @Override
35              public int compare(Recognition lhs, Recognition rhs)
      {
36                // Intentionally reversed to put high confidence at
       the head of the queue.
37                return Float.compare(rhs.getConfidence(), lhs.
      getConfidence());
38              }
39            });
40    for (int i = 0; i < outputs.length; ++i) {
41      if (outputs[i] >= THRESHOLD) {
42        pq.add(
```

```
43              new Recognition(
44                  "" + i, labels.size() > i ? labels.get(i) : "
    unknown", outputs[i], null));
45          }
46      }
47      final ArrayList<Recognition> recognitions = new ArrayList<
    Recognition>();
48      int recognitionsSize = Math.min(pq.size(), MAX_RESULTS);
49      for (int i = 0; i < recognitionsSize; ++i) {
50        recognitions.add(pq.poll());
51      }
52      return recognitions;
53    }
```

CÓDIGO B.4: Hace el manejo del modelo tensorflow.

Bibliografía

Adams, R. L. (ene. de 2017). «10 Powerful Examples Of Artificial Intelligence In Use Today». En: URL: https://www.forbes.com/sites/robertadams/2017/01/10/10-powerful-examples-of-artificial-intelligence-in-use-today/2/\#19215ff83c8b.

Azimi, R., M. Ghofrani y M. Ghayekhloo (2016). «A hybrid wind power forecasting model based on data mining and wavelets analysis». En: *Energy Conversion and Management* 127.Supplement C, págs. 208-225. DOI: https://doi.org/10.1016/j.enconman.2016.09.002. URL: http://www.sciencedirect.com/science/article/pii/S0196890416307804.

Bengio, Yoshua y Others (2009). «Learning deep architectures for AI». En: *Foundations and trends*

textregistered in Machine Learning 2.1, págs. 1-127.

Brownlee, Jason (jul. de 2016). *8 Inspirational Applications of Deep Learning*. URL: https://machinelearningmastery.com/inspirational-applications-deep-learning/.

Cireşan, Dan C. y col. (2010). «Deep, big, simple neural nets for handwritten digit recognition». En: *Neural computation* 22.12, págs. 3207-3220.

Colored_neural_network. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Colored_neural_network.svg/300px-Colored_neural_network.svg.png.

«ConvolutionDiagram». En: (). URL: https://i.stack.imgur.com/6zX2c.png.

Cornejo, A. y col. (2017). *Diagnóstico de la condición ambiental de los afluentes superficiales de Panamá*.

Coutinho, Eduardo y col. (2014). «The Munich LSTM-RNN Approach to the MediaEval 2014.Emotion in Music'"Task.» En: *MediaEval*.

«Diagrama SDK NDK». En: (). URL: https://camo.githubusercontent.com/e3d3b7a9977a2800ff2af09a124a0b51f3bbca6a/68747470733a2f2f6a616c616d6d61722e6769746875622e696f2f696d616765732f616e64726f69642d74656e736f72666c6f772d6170702d737472756374757265322e706e67.

Doyle, Santiago (2009). *Recognition of freshwater macroinvertebrate taxa by image analysis and artificial neural networks*.

«DropoutDiagram». En: (). URL: https://s3-eu-west-1.amazonaws.com/com.cambridgespark.content/tutorials/convolutional-neural-networks-with-keras/figures/drop.png.

Duchi, John, Elad Hazan y Yoram Singer (2011). «Adaptive subgradient methods for online learning and stochastic optimization». En: *Journal of Machine Learning Research* 12.Jul, págs. 2121-2159.

Fabela, Pilar S. y col. (2001). «Utilización de un índice de diversidad para determinar la calidad del agua en sistemas lóticos.» En: *Ingeniería Hidráulica en México*. URL: http://repositorio.imta.mx/bitstream/handle/20.500.12013/762/0242.pdf?sequence=3\&\#38;isAllowed=y.

Goodfellow, Ian, Yoshua Bengio y Aaron Courville (2016). *Deep Learning*. urlhttp://www.deeplearningbook.org. MIT Press.

Havaei, Mohammad y col. (mayo de 2016). «Brain Tumor Segmentation with Deep Neural Networks». En: *Medical Image Analysis* 35, págs. 18-31. ISSN: 13618415. DOI: 10.1016/j.media.2016.05.004. arXiv: 1505.03540. URL: http://dx.doi.org/10.1016/j.media.2016.05.004.

«Hierarchy CNN». En: (). URL: https://media.licdn.com/mpr/mpr/AAIA_wDGAAAAAQAAAAAAAAxnAAAAJGMxYmU5NTUxLWIzOWQtNDJlNi05MmE0LWQ3YTIxODZkN2M2NQ.jpg.

Hubel, David H. y Torsten N. Wiesel (1959). «Receptive fields of single neurones in the cat's striate cortex». En: *The Journal of physiology* 148.3, págs. 574-591.

— (1962). «Receptive fields, binocular interaction and functional architecture in the cat's visual cortex». En: *The Journal of physiology* 160.1, págs. 106-154.

— (1968). «Receptive fields and functional architecture of monkey striate cortex». En: *The Journal of physiology* 195.1, págs. 215-243.

Ioffe, Sergey y Christian Szegedy (mar. de 2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv: 1502.03167.pdf. URL: http://arxiv.org/abs/1502.03167.pdf.

Jafari, Mohammad H. y col. (sep. de 2016). «Extraction of Skin Lesions from Non-Dermoscopic Images Using Deep Learning». En: *International Journal of Computer Assisted Radiology and Surgery* 12.6, págs. 1021-1030. ISSN: 1861-6410. DOI: 10.1007/s11548-017-1567-8. arXiv: 1609.02374.pdf. URL: http://dx.doi.org/10.1007/s11548-017-1567-8.

Jarvis, Dan (feb. de 2017). *Creating an image classifier on Android using TensorFlow (part 1)*. URL: https://medium.com/@daj/creating-an-image-classifier-on-android-using-tensorflow-part-1-513d9c10fa6a.

Jung, Alexander. «imgaug». En: (). URL: https://github.com/aleju/imgaug.

Karpathy, Andrej. *CS231n Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/convolutional-networks/.

— (mayo de 2015). *The Unreasonable Effectiveness of Recurrent Neural Networks*. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Kingma, Diederik P. y Jimmy Ba (ene. de 2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980.pdf. URL: http://arxiv.org/abs/1412.6980.pdf.

Krizhevsky, Alex, Ilya Sutskever y Geoffrey E. Hinton (2012). «ImageNet Classification with Deep Convolutional Neural Networks». En:

Kubilius, Jonas. «VisualCortexDiagram». En: (). URL: http://blog.arimaresearch.com/wp-content/uploads/2017/01/visual_stream_small.png.

Landis, J. Richard y Gary G. Koch (1977). «The measurement of observer agreement for categorical data». En: *biometrics*, págs. 159-174.

LeCun, Yann y col. (1989). «Backpropagation applied to handwritten zip code recognition». En: *Neural computation* 1.4, págs. 541-551.

LeCun, Yann y col. (1998a). «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86, págs. 2278-2324. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

LeCun, Yann y col. (1998b). «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86.11, págs. 2278-2324.

LeDell, Erin. *Multilayer perceptron*. URL: https://github.com/ledell/sldm4-h2o/blob/master/mlp_network.png.

Múltiples, Citaciones. *Conchas - Macro Invertebrados*. URL: https://fr.wikipedia.org/wiki/Thiaridae.

— *Libélulas*. URL: http://www.dec.ny.gov/animals/87939.html.

Ng, Andrew Y (2004). «Feature selection, L 1 vs. L 2 regularization, and rotational invariance». En: *Proceedings of the twenty-first international conference on Machine learning*. ACM, pág. 78.

Oquab, Maxime y col. (2014). «Learning and transferring mid-level image representations using convolutional neural networks». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*, págs. 1717-1724.

Owens, Andrew y col. (abr. de 2016). *Visually Indicated Sounds*. arXiv: 1512.08512.pdf. URL: http://arxiv.org/abs/1512.08512.pdf.

Patel, Vivak (2016). «Kalman-based stochastic gradient method with stop condition and insensitivity to conditioning». En: *SIAM Journal on Optimization* 26.4, págs. 2620-2648.

«Perceptron Picture». En: (). URL: https://cdn-images-1.medium.com/max/800/1*xfimN_ITvZu4CbyQXAZf1A.png.

Raval, Siraj (jun. de 2017). «MNIST Convnet Keras · GitHub». En: URL: https://github.com/llSourcell/A_Guide_to_Running_Tensorflow_Models_on_Android/blob/master/tensorflow_model/mnist_convnet_keras.py.

Resh, Vincent H., M. David y Resh Vh (1993). *Freshwater biomonitoring and benthic macroinvertebrates*. 504.4 FRE.

Rodriguez, Patrick (mar. de 2017). *Creating a Deep Learning iOS App with Keras and Tensorflow*. URL: http://blog.stratospark.com/creating-a-deep-learning-ios-app-with-keras-and-tensorflow.html.

Rumelhart, David E, Geoffrey E Hinton y Ronald J Williams (1986). «Learning representations by back-propagating errors». En: *nature* 323.6088, pág. 533.

Safikhani, Hamed (2016). «Modeling and multi-objective Pareto optimization of new cyclone separators using CFD, ANNs and NSGA II algorithm». En: *Advanced Powder Technology* 27.5, págs. 2277-2284. DOI: https://doi.org/10.1016/j.apt.2016.08.017. URL: http://www.sciencedirect.com/science/article/pii/S0921883116302278.

Sarpola, Matt J. y col. (2008). «An aquatic insect imaging system to automate insect classification». En: *Transactions of the ASABE* 51.6, págs. 2217-2225.

Simard, Patrice Y. y col. «Transformation invariance in pattern recognition: Tangent distance and propagation». En: ().

Simonyan, Karen y Andrew Zisserman (abr. de 2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv: 1409.1556.pdf. URL: http://arxiv.org/abs/1409.1556.pdf.

Srivastava, Nitish y col. (2014). «Dropout: a simple way to prevent neural networks from overfitting.» En: *Journal of machine learning research* 15.1, págs. 1929-1958.

Suwajanakorn, Supasorn, Steven M. Seitz e Ira Kemelmacher-Shlizerman (2017). «Synthesizing obama: learning lip sync from audio». En: *ACM Transactions on Graphics (TOG)* 36.4, pág. 95.

Szegedy, Christian, Alexander Toshev y Dumitru Erhan (2013). «Deep neural networks for object detection». En: págs. 2553-2561. URL: https://scholar.googleusercontent.com/scholar.bib?q=info:vr7GcR7MvhwJ:scholar.google.com/\&\#38;output=citation\&\#38;scisig=AAGBfm0AAAAAWhnLGDutvieNJOWG-9N7wdxLqbD07w7M\&\#38;scisf=4\&\#38;ct=citation\&\#38;cd=-1\&\#38;hl=en.

Szegedy, Christian y col. (sep. de 2014). *Going Deeper with Convolutions*. arXiv: 1409.4842.pdf. URL: http://arxiv.org/abs/1409.4842.pdf.

Szegedy, Christian y col. (dic. de 2015). *Rethinking the Inception Architecture for Computer Vision*. arXiv: 1512.00567.pdf. URL: http://arxiv.org/abs/1512.00567.pdf.

Szegedy, Christian y col. (ago. de 2016). *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*. arXiv: 1602.07261. URL: http://arxiv.org/abs/1602.07261.

Tensorflow Mobile. URL: https://www.tensorflow.org/mobile/.

The Next Wave of Deep Learning Applications (sep. de 2016). URL: http://globalbigdataconference.com/news/129842/the-next-wave-of-deep-learning-applications.html.

Tieleman, Tijmen y Geoffrey Hinton (2012). «Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude». En: *COURSERA: Neural networks for machine learning* 4.2, págs. 26-31.

Tirronen, Ville y col. (2009). «Multiple Order Gradient Feature for Macro-Invertebrate Identification Using Support Vector Machines». En: *Adaptive and Natural Computing Algorithms*. Ed. por Mikko Kolehmainen, Pekka Toivanen y Bartlomiej Beliczynski. Vol. 5495. Lecture Notes in Computer Science. Springer Berlin Heidelberg, págs. 489-497. DOI: 10.1007/978-3-642-04921-7_50. URL: http://dx.doi.org/10.1007/978-3-642-04921-7_50.

Xu, Zhenqi, Jiani Hu y Weihong Deng (2016). «Recurrent convolutional neural network for video classification». En: *Multimedia and Expo (ICME), 2016 IEEE International Conference on*. IEEE, págs. 1-6.

Zhang, Jiajun y Chengqing Zong (2015). «Deep neural networks in machine translation: An overview». En: *IEEE Intelligent Systems* 30.5, págs. 16-25.