Minería de imágenes en repositorios de proyectos de Software soportada por Machine Learning

Sergio Andrés Rodríguez Torres

Estudiante magíster en informatica

Escuela colombiana de ingeniería Julio Garavito

Director

Luis Daniel Benavides Navarro

Codirectores

Wilmer Garzón A., Héctor Cadavid R.

Bogotá, Colombia

sergio.rodriguez-tor@mail.escuelaing.edu.co

11 de abril de 2023

Índice general

Resumen

Este documento presenta un análisis enfocado a los diagramas de software en repositorios Git. Para el análisis se construyó un dataset con 5.981 imágenes con las categorías none, diagrama de actividades, diagrama de secuencia, diagrama de clases, diagrama de componentes, diagrama de casos de uso y diagramas cloud. Dicho dataset se usó para el entrenamiento de una red convolucional DenseNet169 pre-entrenada con el dataset ImageNet usando la técnica de transfer learning. La red alcanzó una exactitud en la predicción del 98.6 % y un f1-score de 98.3 %. Luego se usaron técnicas de minería de repositorios para analizar 2'469.206 imágenes equivalentes a 231 GB en datos, obtenidas de 287.201 repositorios. Con el fin de conocer que tan común es cargar imágenes de diagramas, cómo se distribuyen en los repositorios de software y cada cuanto se actualizan.

Keywords — Repository mining, Neural Network, Deep Learning, Multiclass Classification, Diagrams.

Capítulo 1

Introducción

Durante la última década, la colaboración entre equipos cada vez más grandes y dispersos geográficamente, se ha vuelto más importante en la ingeniería de software. Como resultado, surgieron herramientas que facilitan flujos de trabajo distribuidos en proyectos comerciales y proyectos de código abierto, por ejemplo, *GitHub*, *GitLab*, *Bitbucket*, etc. La adopción generalizada de estas plataformas por parte de la comunidad de software ha abierto nuevas oportunidades en el área de investigación, como el análisis empírico de los datos generados por los equipos de desarrollo a lo largo del tiempo. Los datos no solo muestran la evolución de los artefactos de software, sino también de las interacciones de la comunidad y el equipo, mediante *issue trackers*, *logs*, etc. Al proceso de minería de datos aplicada a los proyectos de software se le denomina minería de repositorios, este campo de investigación busca entender y analizar hipótesis relacionadas con el desarrollo, diseño e ingeniería de software.

En este proyecto, nos enfocaremos en el análisis de las imágenes que se

encuentran en los repositorios de software, estas son relevantes debido a que contienen conocimiento técnico del proyecto, como diagramas de software, los cuales son útiles para trabajar de forma efectiva como un equipo. A medida que las aplicaciones crecen, se hace más complejo su diseño y arquitectura, es necesario ser capaz de expresar el funcionamiento de los sistemas y la arquitectura de forma rápida entre el equipo. En esto juega un rol importante los diagramas que permiten visualizar y documentar los artefactos y sus interacciones en un sistema de software, facilitando así la comunicación de ideas. Los desarrolladores de software, arquitectos y diseñadores han creado diversos estándares para los diagramas, entre esos el lenguaje de modelado unificado (UML) que ofrece un conjunto universal de notaciones para describir conceptos y comportamientos de un sistema [1]. Sin embargo, hay otros estándares usados para crear diagramas de otros tipos, cómo los diagramas de arquitectura o diagramas de componentes, estos a su vez incorporan métodos de comunicación específicos. Por ejemplo, los diagramas de arquitectura construidos con iconos específicos que representan recursos y herramientas de proveedores de servicios *cloud* (AWS, Microsoft Azure, GCP).

Los diagramas por lo general se encuentran en formato de imagen, pero no son las únicas imágenes en un repositorio de software, es común que los proyectos contengan *assets* e iconos que no son diagramas de documentación. Suele ser difícil extraer información de imágenes o contenido multimedia, ya que la información está embebida en el contenido de este. Una de las técnicas comúnmente usadas para expresar de forma sintetizada el contenido de una imagen es etiquetándola basándose en su contenido, de esta forma se clasifican las imágenes. El problema de clasificación de imágenes es un

problema estudiado que suele resolverse por medio de técnicas de *machine learning*. Los avances en distintos campos, como la algoritmia y el poder de computación, han facilitado la predominancia de redes neuronales profundas, en concreto para el problema de clasificaciones de imágenes predominan las redes neuronales convolucionales (CNN) [37]. Además, se ha complementado con otras técnicas como *transfer learning*, que consiste en transferir de forma parcial o total el conocimiento pre-aprendido por una red al resolver un problema, para resolver otro problema diferente pero relacionado [38].

1.1. Problemática

Se desea entender cómo los ingenieros de software hacen uso de imágenes en el ciclo de vida del software. Bajo la hipótesis de que el conocimiento del diseño se encuentra en varios componentes, de los cuales el código fuente es de la representación más fiel, debido a que otras formas como los diagramas de software suelen estar desactualizados y no corresponder a la última versión del código.

1.2. Motivación

Son varias las razones que motivaron este estudio, a continuación se listan algunas:

- Clasificar los diagramas de software de forma automática nos permite realizar nuevas investigaciones en campos más específicos, debido a que cada tipo de diagrama puede ser útil para estudiar distintos aspectos

de un proyecto de software y el equipo que los construye. Por ejemplo, se puede realizar estudios enfocados en diagramas de clases, para entender cuantos componentes hay y estimar el tamaño o complejidad del proyecto o recrear la arquitectura del sistema como se realizó en [23]. Además, la ausencia o exceso de diagramas de un tipo, pueden indicar la presencia de ciertos patrones de diseño o uso de ciertas tecnologías.

- Hay algunos estudios que presentan herramientas para clasificar diagramas UML de forma automática, pero la mayoría usan técnicas de *machine learning* empleando algoritmos tradicionales, por lo cual los resultados dependen de las características de las imágenes que consideraron adecuadas para identificar los diagramas [21]. Además, hay muy pocos estudios que usan redes neuronales profundas, en concreto redes neuronales convolucionales (CNN).

- Son escasos los estudios de clasificadores de diagramas de software que incorporen otros tipos de diagramas no UML y ninguno usa redes neuronales convolucionales.

- Son muy escasos los estudios que incorporan herramientas de clasificación automatizadas que usan CNN en conjunto con técnicas de minería de repositorios para realizar análisis en repositorios de software.

1.3. Preguntas de investigación

Con este artículo se busca resolver las siguientes preguntas de investigación:

Q1 ¿Cuál es el impacto del uso de *transfer learning* y *fine-tuning* en el desempeño de un clasificador de imágenes enfocado en diagramas?

Q2 ¿Qué tan común es cargar imágenes en repositorios Git, si son diagramas, qué tipo de diagramas son?

Q3 ¿Cuál es la proporción de proyectos Git que contienen diagramas y cómo se distribuyen por tipo de diagrama?

Q4 Con el fin de saber si los diagramas están desactualizados, ¿Cuál es la diferencia en tiempo entre la última actualización del diagrama y el repositorio?

1.4. Contribuciones

El artículo contribuye en el área de investigación y plantea algunas preguntas alrededor:

- Diseño de un flujo de trabajo para el análisis de imágenes en repositorios de ingeniería de software usando una red neuronal convolucional (CNN) pre-entrenada con ImageNet y reentrenada usando la técnica de *transfer learning* y complementada con *fine-tuning*.
- Creación de un *dataset* extenso, con casi 6 mil imágenes distribuidas entre las categorías: *none*, diagrama de actividades, diagrama de secuencia, diagrama de clases, diagrama de componentes, diagrama de casos de uso y diagrama *cloud*.

- Creación de un *dataset* extenso etiquetado por inteligencia artificial con categorías de diagramas. Se usó el clasificador sobre 2'469.206 imágenes obtenidas de 287.201 repositorios, mediante técnicas de minería de repositorios.

- Validación y experimentos analizando el uso de diagramas en formato de imagen en repositorios de software.

Lo restante del documento contiene cinco capítulos más. En el capítulo 2, se presenta el marco teórico en cuál se presentan algunos conceptos que facilitan la comprensión del documento. En el capítulo 3, discutimos el estado del arte, los trabajos realizados previamente relacionados con el documento. En el capítulo 4, exploramos el desarrollo y detalles del experimento realizado. En el capítulo 5, realizamos el análisis de los resultados obtenidos y respondemos la preguntas de investigación. Por último, en el capítulo 6 tenemos la conclusión y trabajo futuro.

Capítulo 2

Marco Teórico

En este capítulo se espera proveer una introducción a algunos conceptos que serán útiles para entender el artículo. Primero, se explicará como funcionan las redes convolucionales y otros conceptos relacionados con las redes neuronales, incluido como se puede aplicar *transfer learning* sobre una red convolucional y como es posible la transferencia de conocimiento entre dos problemas similares. Luego veremos otras definiciones importantes de conceptos que se menciona repetidamente en el documento, como minería de repositorios y los tipos de diagramas que se van a estudiar, siendo estos tipos especificos de UML y diagramas de arquitectura *cloud*.

2.1. Red convolucional

Una red convolucional es un tipo de red profunda [25], que se especializa en tareas de visión por computador, como la clasificación y segmentación de imágenes. La red aprovecha la estructura espacial de la imagen, bajo la pre-

misa de que el valor de un pixel está ligado a otros píxeles adyacentes, lo que compone estructuras, formas y patrones que facilitan entender el contenido de la imagen [8]. La red convolucional aprende y asigna valores numéricos o pesos a aspectos u objetos de una imagen, con el fin de distinguirlos. La red requiere un menor preprocesamiento en comparación a otros algoritmos de clasificación, además se caracterizan por aprender los filtros para identificar características de forma automática, evitando el proceso de *feature engineering*, el cual consiste en seleccionar manualmente las características que se desean identificar en la imagen [33].

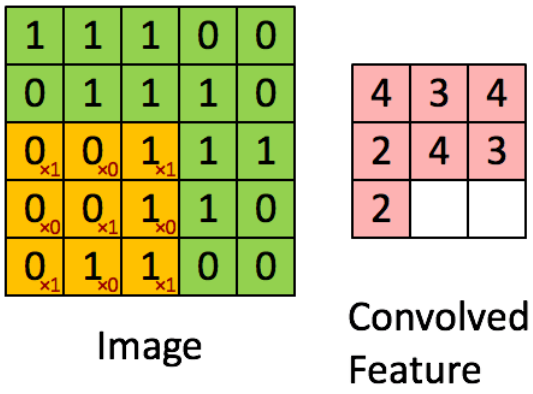

Figura 2.1: Representación del proceso de convolución sobre una imagen de 5x5 aplicando un kernel de 3x3 cuyos valores son el multiplicador en rojo, resultando en un mapa de características de 3x3. Tomado de [33].

La red convolucional se caracteriza por incorporar una o varias capas en las cuales se realiza la operación de convolución, la cual busca reducir la imagen a una forma que sea más fácil de procesar, sin perder características que son críticas para obtener una buena predicción, esta consiste en aplicar un filtro usando un grupo de píxeles adyacentes, a este filtro también se

le conoce como *kernel*. La red aprenderá los valores adecuados para cada filtro durante el proceso de entrenamiento. Al aplicar el filtro se generará una nueva representación de la imagen, por lo general más pequeña, a la cual se le denomina mapa de características, en esta las áreas brillantes son las regiones activadas que pasaron el filtro, indicando que se ha detectado la característica que busca identificar ese filtro, en la figura 2.1 se muestra el proceso de convolución, en cuál se aplica un kernel sobre los valores de una imagen de entrada, para generar un mapa de características.

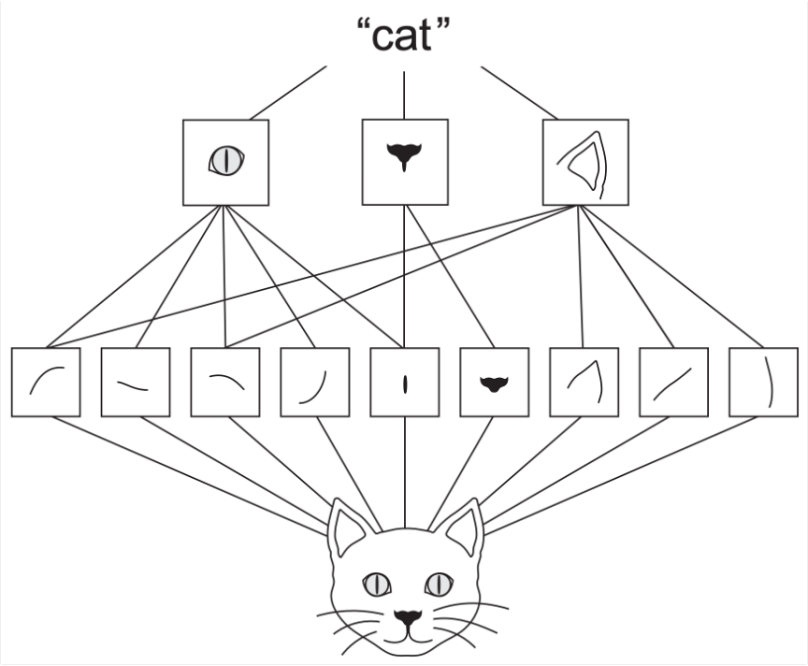

Figura 2.2: Representación de un patrón de jerarquía espacial de módulos visuales: los bordes hiperlocales se combinan en objetos locales como ojos u oídos, que se combinan en conceptos de alto nivel como "gato". Tomado de [8].

La red se compone de múltiples capas secuenciales, cada una con múltiples filtros que usan como entrada todas las salidas de la capa directamente

anterior. Se generan filtros cada vez más complejos por composición, debido a que patrones detectados por filtros en las primeras capas, como bordes o segmentos de recta, pueden componer circunferencias y otras estructuras más complejas, que a su vez componen conceptos de nivel medio como un ojo, una nariz, orejas, etc. Estos a su vez componen conceptos de alto nivel, como un rostro, un carro o un paisaje, a esto se le conoce como patrones de jerarquías espaciales, podemos observar un ejemplo en la figura 2.2 [39].

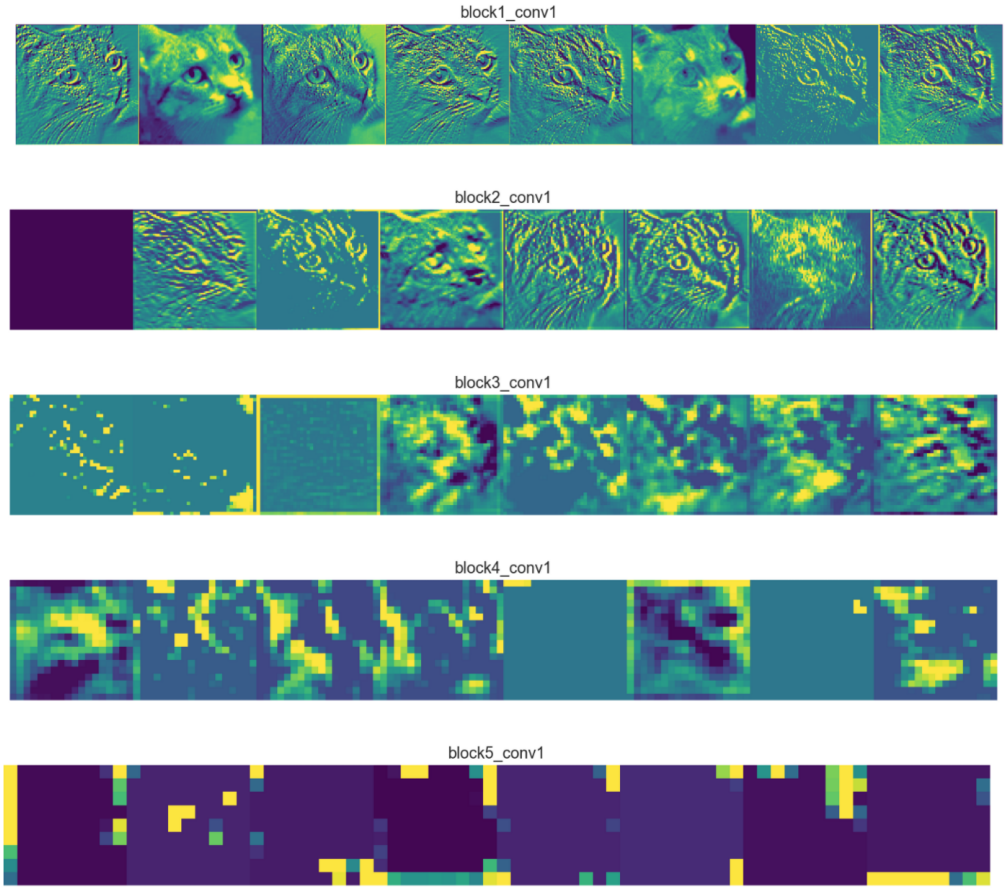

Figura 2.3: Representación gráfica de los primeros 8 mapas de características por cada capa, perteneciente a una red convolucional entrenada para clasificar entre gatos y perros. Los mapas fueron generados a partir de la imagen de un gato. Tomado de [11].

Podemos ver un ejemplo en la figura 2.3 en la cual se muestran los primeros 8 mapas de características, generados por una red con 5 bloques de convolución de tamaño 64, 128, 256, 512 y 512 respectivamente, esta red está entrenada para clasificar imágenes en las categorías gato o perro. Podemos detectar que la primera capa *block1_ conv1* retiene la mayoría de la información de la imagen de entrada y se encarga de detectar bordes. A medida que avanzamos a las siguientes capas, los mapas de características se parecen menos a la imagen original y más a una representación abstracta de la misma. Esto se debe a que los mapas de características más profundos codifican conceptos de alto nivel como "nariz de gato" u "oreja de perro", mientras que los mapas de características de nivel inferior detectan bordes y formas simples. Debido a lo específico de los filtros pertenecientes a capas más profundas, vemos una menor activación representada con imágenes más oscuras en las capas *block4_ conv1* y *block5_ conv1* [11].

2.2. Overfitting

El *Overfitting* o sobreajuste es el resultado de sobreentrenar un algoritmo de aprendizaje automático con datos para los que se conoce el resultado deseado. Este tipo de algoritmos buscan llegar a un estado en el cual sean capaces de predecir el resultado en nuevos casos a partir de lo aprendido. Sin embargo, cuando se sobreentrena el algoritmo, este se ajusta a las características aleatorias muy específicas de los datos de entrenamiento que no tienen relación con la función objetivo. Una definición más precisa de *overfitting*, este se da cuando se rompe el principio de parsimonia o navaja de Ockham, el

cual indica que “en igualdad de condiciones, la explicación más simple suele ser la más probable” [3]. Esto aplica a un modelo que realiza predicciones, cuando se tiene más parámetros de los necesarios para describir el patrón de comportamiento de los datos. Por ejemplo, como se muestra en la figura 2.4 los datos siguen un patrón que puede ser descrito por una función lineal, pero se usa una función polinomial que se ajusta perfectamente a los datos, se puede esperar que la función lineal generalice mejor el comportamiento, por lo tanto, si las dos funciones se usaran para extrapolar nuevos datos, la función lineal debería hacer mejores predicciones. En otras palabras, el modelo recuerda los ejemplos en lugar de aprender los patrones y características del problema [14].

El sobreajuste es más común en los casos en que el aprendizaje se realizó durante demasiado tiempo o donde los ejemplos de entrenamiento son escasos.

2.3. Transfer learning

Transfer learning es la técnica para transferir el conocimiento aprendido por una red a otra que busca solucionar un problema diferente, pero similar.

Consiste en tomar una red pre-entrenada y congelar algunas o todas sus capas, impidiendo que estas modifiquen los parámetros pre-aprendidos. Opcionalmente, se pueden agregar nuevas capas que, en conjunto con las capas no congeladas, van a aprender las características específicas del nuevo problema. Podemos dividir una red en cuatro secciones, la entrada, las capas iniciales en las cuales se aprenden características generales del problema que

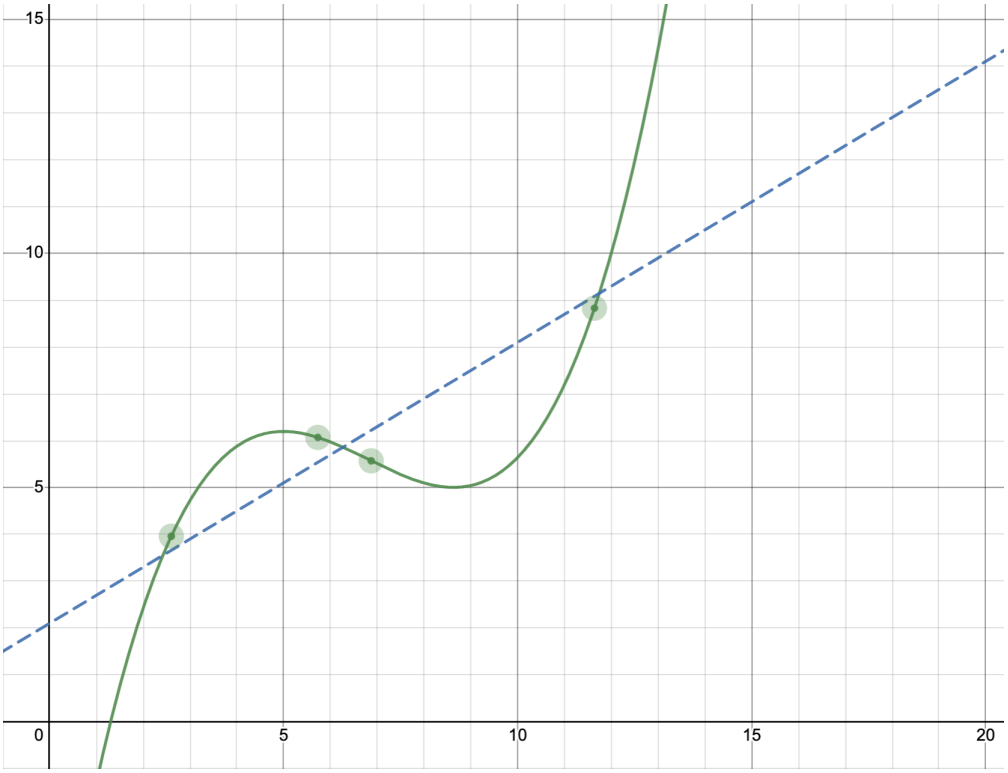

Figura 2.4: Gráfico creado con Desmos, en este se muestra una función polinomial de cuarto grado y una función lineal que buscan describir el comportamiento de los datos.

son fáciles de transferir, las capas finales que aprenden las características específicas del problema y el *top* en el cual se realiza la clasificación final, como se describe en la figura 2.5. La red, al tener pre-aprendidos muchos de los patrones básicos para la identificación de bordes y formas simples aprendidos en las capas iniciales, solo requiere aprender patrones compuestos específicos del nuevo problema. Por lo tanto, se puede obtener un buen rendimiento con *datasets* de entrenamiento reducidos y además disminuye el tiempo de entrenamiento de la red.

Por ejemplo, las características aprendidas en las capas iniciales de un modelo para la clasificación de letras del abecedario, pueden ser útiles en un

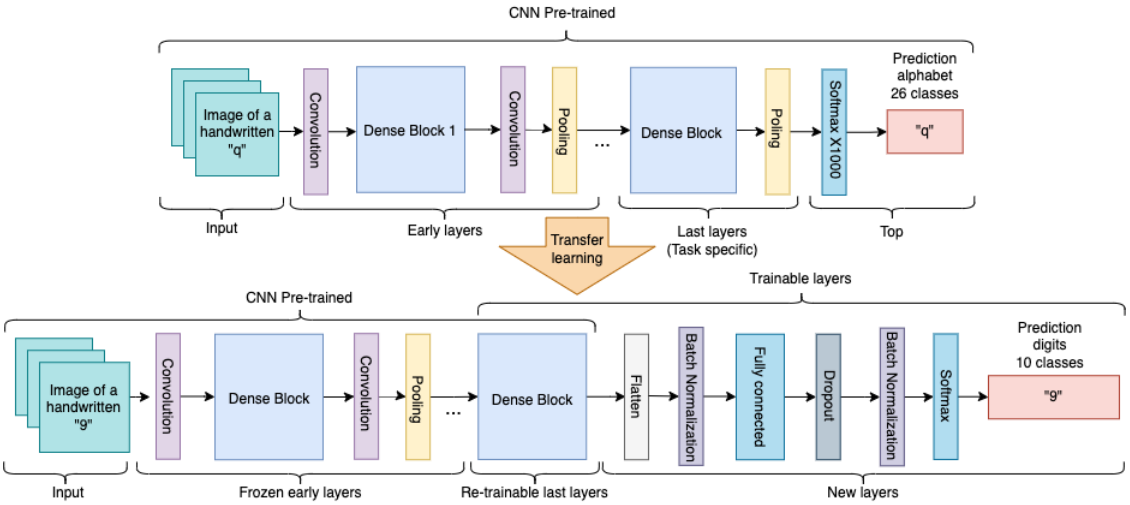

Figura 2.5: Diagrama que describe como se aplica transfer learning sobre una CNN pre-entrenada para la detección de letras escritas a mano del abecedario, para entrenar una nueva red para la detección de dígitos escritos a mano.

modelo de clasificación de dígitos. Debido a que los patrones necesarios para identificar una letra o un dígito son similares, comparten patrones de bajo nivel como segmentos de recta y bordes; también patrones de medio nivel como circunferencias y rectas que componen una letra o un dígito [38].

Por último, se puede complementar *transfer learning* con *fine-tuning*, que consiste en descongelar todo el modelo que se obtuvo anteriormente aplicando *transfer learning* (o parte de él) y volver a entrenarlo con los nuevos datos con una tasa de aprendizaje muy baja. Esto puede potencialmente lograr mejoras significativas, al adaptar de manera incremental las características pre-entrenadas a los nuevos datos.

2.4. Minería de repositorios

Minería de repositorios de software es un campo de la ingeniería de software que analiza los datos disponibles en los repositorios de software, como el control de versiones, comentarios, incidencias, solicitudes de mezcla, entre otros. Con el fin de extraer información útil. Esta información se puede usar para mejorar el proceso de desarrollo y ayudar a los desarrolladores. Por ejemplo, mediante la predicción de problemas realizada a partir de los errores pasados, reconociendo los patrones presentes cuando hay un error o como se realizará en este documento, nos permite extraer información de los repositorios para hacer análisis e identificar patrones indeseados u oportunidades de mejora [31].

La minería de repositorios se compone de herramientas y técnicas propias, como modelos para almacenar y recuperar métricas de software, puede abarcar aspectos avanzados como el análisis de datos y ser complementada con otras técnicas como *machine learning* [22].

2.5. UML

El lenguaje de modelado unificado (UML) es un lenguaje de modelado de sistemas de software, que proporciona herramientas para el análisis, diseño e implementación de sistemas basados en software, así como para el modelado de negocios y procesos similares. UML ofrece un estándar para describir un "plano" del sistema o modelo, incluyendo aspectos conceptuales tales como procesos, funciones del sistema, y aspectos concretos como expresiones de len-

guajes de programación, esquemas de bases de datos y compuestos reciclados [1].

2.6. Diagrama de arquitectura cloud

Los diagramas de arquitectura *cloud* se utilizan para documentar visualmente los servicios de computación en la nube de una organización. La infraestructura de estos servicios puede ser compleja, por lo que crear un diagrama de arquitectura *cloud* es una buena manera de describir el entorno de la nube para la documentación de la organización, hacer planes para actualizaciones o solucionar problemas [36].

Una arquitectura en la nube típicamente se compone de: recursos de computación, como máquinas virtuales, recursos de almacenamiento, como bases de datos o sistemas de caché, componentes de red, como nubes privadas virtuales o servicios enrutadores y otros tipos de servicios complejos como servicios de seguridad, administración de contenidos, monitoreo de servicios y recursos, comunicación y dispersión de mensajes [30]. Los diagramas suelen especificar la tecnología usada, por lo general la solución concreta de un proveedor *cloud*, como vemos en la figura 2.6 donde encontramos un ejemplo de un diagrama de arquitectura *cloud*, en esta podemos ver el uso de múltiples servicios del proveedor *cloud* AWS, el primer servicio que interactúa con las solicitudes del usuario es un *WAF* el cual es un componente de seguridad, luego ingresamos a una VPC la cual es una nube privada virtual, donde tenemos unidades de cómputo administradas por un balanceador de carga y acompañadas de RDS, los cuales son sistemas de bases de datos relaciona-

les, adicionalmente hay un servicio de monitoreo *CloudWath*, otro servicio de almacenamiento estático S3 y un servicio de respaldo AWS Backup.

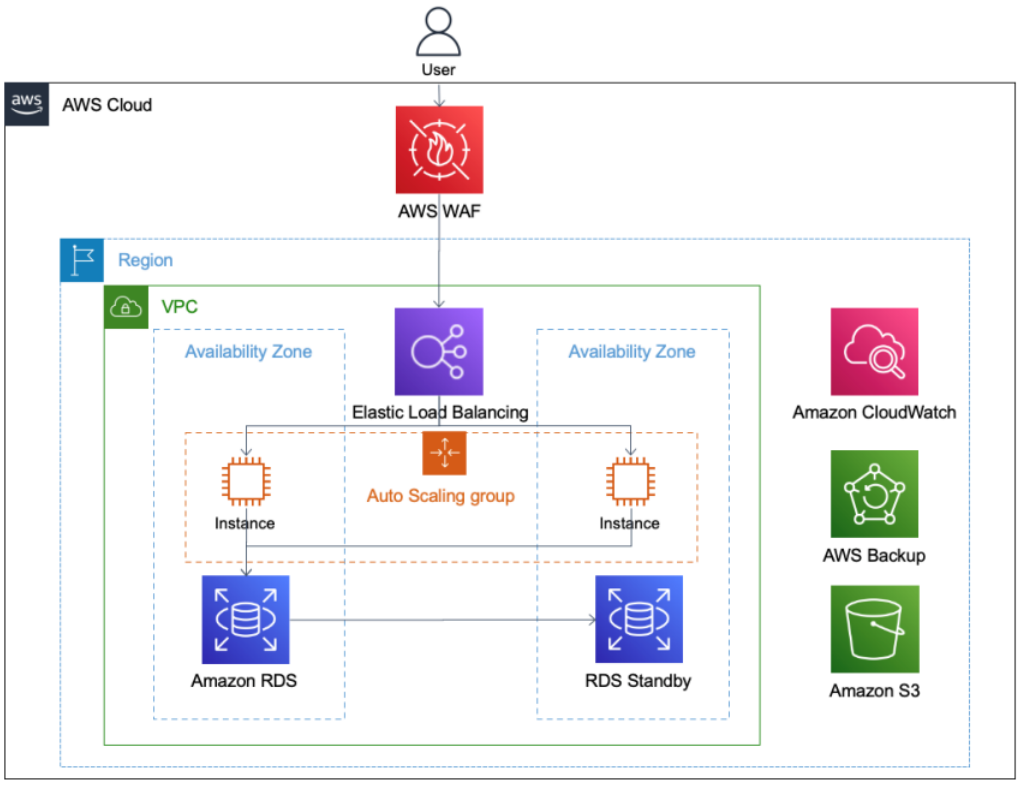

Figura 2.6: Diagrama de arquitectura cloud empleada por MPM una empresa de tecnológica líder en soluciones de gestión para la distribución de seguros. Tomado de [12].

Adicionalmente, los diagramas de arquitectura *cloud* suelen ser usados como representación gráfica de soluciones de infraestructura como código (IaaC), incluso hay herramientas *open-source* y de pago que permiten auto-generarlos.

Capítulo 3

Estado del arte

En esta sección se explorarán algunos trabajos relacionados realizados en las áreas de minería de repositorios con imágenes de diagramas y clasificación de imágenes de diagramas. Se detectó que la mayoría de estudios están enfocados a diagramas UML, dejando de lado diagramas de arquitectura de componentes de los proveedores de servicios *cloud*. Además, la mayoría usan algoritmos clásicos de *machine learning*, solo se encontraron pocos estudios que usa redes convolucionales que son el actual estado del arte en la clasificación de imágenes.

3.1. Minería de repositorios aplicada a imágenes de diagramas

Se han realizado varios estudios en el área, por ejemplo, en [32] se busca crear un *dataset* de diagramas UML usando la técnica de minería de repositorios. Para ello, extrajeron repositorios de *GitHub* con ayuda de *GHTorrent*

y *Github API*. Luego recorrieron el contenido del repositorio en busca de archivos que pudieran ser diagramas, para esto realizaron un proceso de filtrado a partir de la extensión de los archivos (jpg, jpeg, png, bmp, svg, gif) y el nombre del archivo (*diagram, architecture, design, uml*). Después filtraron las imágenes basándose en diversos parámetros, como el tamaño de estas y se eliminaron las repetidas. Por último, usaron un clasificador de diagramas UML propuesto en [29] el cual usa *logistic Regression* y un proceso de extracción de características. Otro estudio [15] se enfoca en la construcción de una herramienta para poder determinar si un proyecto contiene un diagrama y poder seguir el proceso de evolución de ese diagrama en el proyecto, usan *GHTorrent* y *GitHub API* para la extracción de la *metadata* de los repositorios, al igual que [26] usan un proceso de pre-filtrado a partir de la extensión de los archivos y su nombre, utilizan *Random Forest* como algoritmo de *machine learning* para realizar la clasificación final.

3.2. Clasificación de imágenes de diagramas

Hay estudios que se enfocan en la clasificación de diagramas a partir de imágenes, en su mayoría enfocados en diagramas UML y buscan clasificar qué tipo de diagrama UML representa una imagen. En [17] no usan *machine learning* de forma directa para la clasificación, se apoya en la herramienta *Img2UML* [18] publicada por ellos, la cual usa MODI, una librería para extraer texto de imágenes, luego con un conjunto fijo de criterios realizan la clasificación. Por otro lado, [29, 26, 24] se enfocan en estrategias para la extracción de características y mejoras de la eficacia del clasificador, en estos se

usan algoritmos clásicos de *machine learning*, como *Support Vector Machines* (SVM), *Logistic regression*, *Random forest*, entre otros.

Otros estudios se enfocaron en detectar diagramas no UML, la investigación realizada en este campo es bastante escasa y solo unas pocas utilizan *machine learning*. La mayoría utilizan métodos de procesamiento de imágenes y reconocimiento de patrones, modificados para detectar dominios específicos, como diagramas arquitectónicos [23] o diagramas de autómatas finitos [5].

Los recientes avances en hardware y software han permitido que las redes neuronales convolucionales den mejores resultados en la tarea de la clasificación de imágenes, convirtiéndose en el estado del arte en ese problema [37]. Uno de los primeros trabajos que usaron redes neuronales convolucionales y *transfer learning* fue [6] donde se muestra la viabilidad de usar una red pre-entrenada con el *dataset* ImageNet y transferir el conocimiento general para reconocimiento de imágenes a la detección de diagramas, a pesar de que las imágenes sean de dominios diferentes. Luego otros estudios siguieron esa línea de investigación como [35] en el cual se busca clasificar tres tipos específicos de diagramas UML e imágenes no UML. Para esto se construyó un *dataset* con 3231 imágenes repartidas entre las 4 categorías y se buscó probar diferentes arquitecturas (*MobileNet, DenseNet, NasNet, ResNet* e *Inception* y una propia) de las cuales *DenseNet169* y la arquitectura propia obtuvieron los mejores resultados. En este estudio se aplicó la técnica de *transfer learning* a este problema; sin embargo, no se detalla la implementación, dejando incierto si se realizó *fine-tuning* o no. Además, se dejan de lado otras arquitecturas como *Xception, InceptionResNetV2, NASNetLarge* y *Ef-*

ficientNet que en algunas tareas se desempeñan mejor que las arquitecturas estudiadas. Otro estudio que aplica redes convolucionales es [7] que usa un dataset obtenido de otro artículo de conferencia [9], el cual cuenta con 3298 imágenes, realizan una clasificación binaria para determinar si las imágenes son UML o no.

En [27] propusieron usar redes profundas simples en combinación con una estrategia *low-short* de aprendizaje para reducir los parámetros de aprendizaje al clasificar diagramas de clase y secuencia UML; sin embargo, se limitan solo dos tipos de diagramas y no se aplica *transfer learning*.

3.3. Otros estudios con minería de repositorios

Otros estudios se encargaron de crear *datasets* con imágenes de diagramas UML, para esto [15] ofrece un listado de repositorios que contienen imágenes UML, en [19] se creó un repositorio en línea de diagramas UML en formato XMI. Por otro lado, en [32] combinaron trabajo manual y algoritmos de detección para reconocer 93.596 diagramas UML de 24.717 repositorios. Además, de los diagramas detectados, el 61.8 % se encontraba en formato de imagen, esto demuestra que si hay una cantidad considerable de diagramas dentro de los repositorios en formato de imagen.

Se detectó la oportunidad de investigar en el área de minería de repositorios, enfocándonos en las imágenes de diagramas que se encuentran dentro de los repositorios. Se usarán redes convolucionales con *transfer learning* y *fine-tuning* e incluirán otros tipos de diagramas, además de UML durante la clasificación.

Capítulo 4

Desarrollo

En este capítulo se va a presentar el proceso empleado durante la construcción del *dataset* usado para entrenar la red del clasificador de imágenes de diagramas. También veremos la arquitectura y el proceso de refinamiento de la red. Por último, se presenta el proceso de extracción de datos y predicción para el análisis realizado sobre las imágenes de repositorios *Git*.

Podemos encontrar el código fuente usado para obtener los resultados presentados en este capítulo en *Figshare*[1].

4.1. Construcción del Dataset

Para el *dataset* de entrenamiento se eligieron 7 categorías, 3 diagramas estructurales (clases, componentes y arquitectura *cloud*), 3 diagramas de comportamiento (actividades, casos de uso y secuencia) y una categoría *none* para imágenes no relacionadas con diagramas. Los diagramas se eligieron

[1]Rodríguez Torres, S. A. (2022): An analysis of diagram images on Git repositories. Figshare. Software. https://doi.org/10.6084/m9.figshare.20438661.v5

por su popularidad y los de arquitectura *cloud* se incluyeron para evaluar su frecuencia en los repositorios de software, ya que no se han estudiado.

El *dataset*[2] creado se compone de imágenes obtenidas de diversas fuentes, la mayoría de imágenes fueron obtenidas mediante *scripts* tipo *scrapper* que extraen imágenes de los resultados de búsquedas en Google, simulando una interacción humana con la página web usando *Selenium*. Otras imágenes se extrajeron del *dataset* de *Lindholmen* provisto en [32], e imágenes obtenidas de repositorios *Git*, la mayoría de imágenes de la categoría *none* provienen de este último, son *assets* e imágenes varias que se pueden encontrar en repositorios de software.

4.1.1. Scrapper a Google search

El *scrapper* a *Google search* consiste de un *script* que descarga las imágenes en los *thumbnails* de los resultados de búsqueda entregados por Google en el *tab* de imágenes al buscar una palabra. Este funciona usando *Selenium*, *Selenium* es un conjunto de herramientas de código abierto para la automatización de pruebas de navegadores web, estas permite escribir scripts con los cuales se puede automatizar una interacción simulada de un usuario con el navegador web [34]. En este caso primero se construye la URL del sitio web incluyendo como un *queryParam* la palabra de la cual se quieren descargar las imágenes. *Selenium* mediante el *driver* del navegador abre una nueva ventana y de forma automatizada simula la interacción de un usuario con los resultados de búsqueda.

[2]Rodríguez Torres, S. A. (2022): Image diagram dataset. Figshare. https://doi.org/10.6084/m9.figshare.20399283.v2

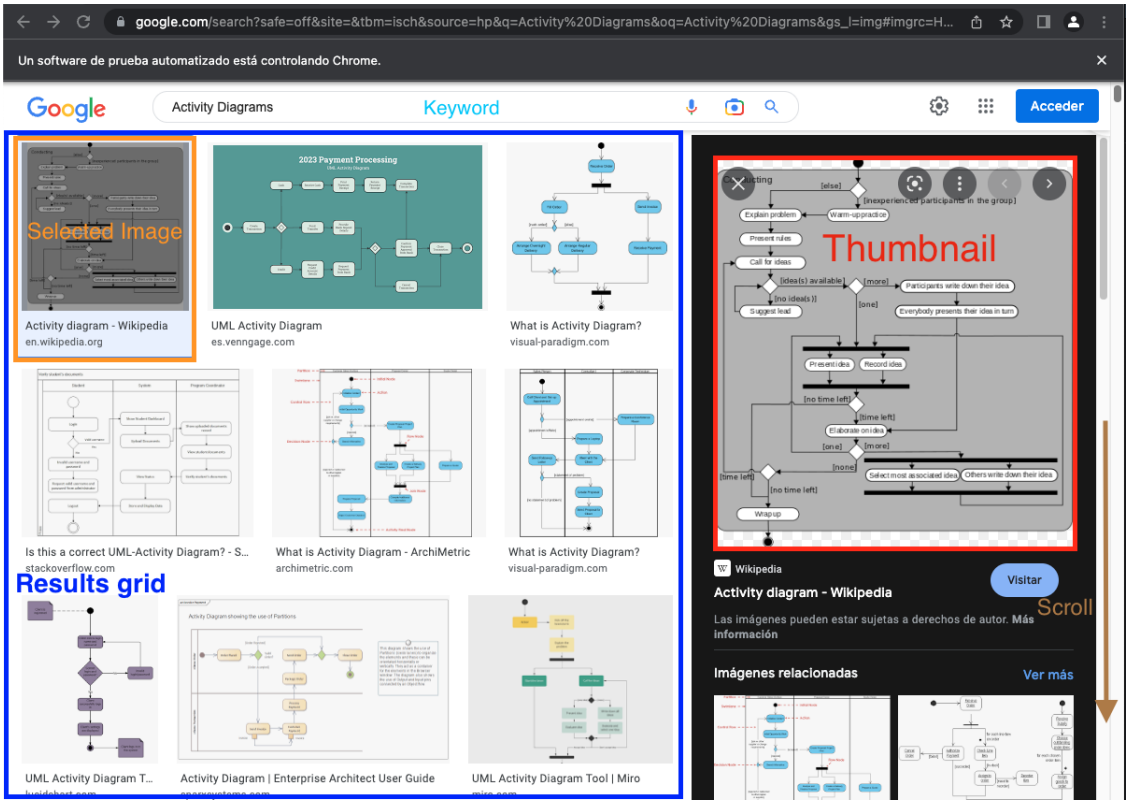

Figura 4.1: Pantallazo de los resultados de la búsqueda en Google para "Activity Diagrams", con las secciones de la página resaltadas.

Lo primero que se realiza es un *scroll* vertical hacia abajo, para cargar todos los resultados posibles, Google página los resultados de la búsqueda. Una vez se llega al final de los resultados, se identifican los componentes en el DOM HTML que conforman la matriz de resultados y la sección de vista previa, estos componentes se resaltan en la figura 4.1. Para ello se buscan los componentes que tengan cierto estilo CSS, no se usan etiquetas o propiedades de las etiquetas HTML debido a que esos valores están ofuscados. Una vez identificados los elementos HTML que contiene cada resultado, se itera sobre ellos haciendo clic en cada uno para que nos muestre la vista previa agrandada (*thumbnail*) de la imagen, se extrae la URL perteneciente a la imagen y esa

se almacena para luego ser descargada. Los resultados descargados se van a almacenar renombrados con un *hash* generado a partir del contenido de la imagen en una carpeta con el nombre de la palabra buscada.

4.1.2. Normalización de las imágenes

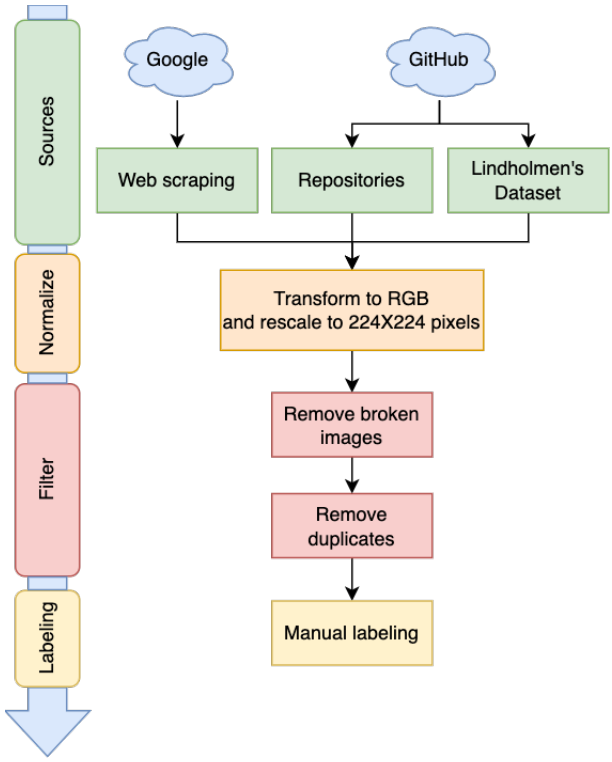

Figura 4.2: Proceso de extracción, normalización, filtrado y etiquetado realizado para la construcción del dataset.

Sobre las imágenes del *dataset* de entrenamiento se realizó un proceso de normalización, filtrado y etiquetado descrito en la figura 4.2. Este proceso es necesario debido a que el modelo requiere que las imágenes usadas para el proceso de entrenamiento y evaluación se encuentren en un formato específico. El proceso de normalización consiste en convertir las imágenes a JPG en

formato RGB y re-escalar las imágenes a 224x224 píxeles usando el algoritmo de escalado de muestreo de *Lanczos* [20]. Se seleccionó este algoritmo, aun siendo el más lento, debido a su habilidad de retener detalles y conservar la información de la imagen original [28], lo cual es importante en imágenes de diagramas porque las líneas que conectan los componentes son finas y con algunos algoritmos de re-escalado desaparecían. Luego, las imágenes se verificaron manualmente para detectar imágenes dañadas por el proceso de normalización. Las imágenes normalizadas se renombraron usando un *hash* de 15 dígitos generado a partir del contenido de la imagen. Esto permitió detectar imágenes repetidas, además se usaron programas de detección de imágenes similares para descartar más imágenes duplicadas. Se analizaron aproximadamente 14.000 imágenes, de las cuales 5.981 se seleccionaron y etiquetaron manualmente, resultando en la distribución descrita en el cuadro 4.1. Las imágenes descartadas no pertenecían claramente a una de las categorías propuestas, o tenían una resolución demasiado baja, o se dañaron durante el proceso de conversión a RGB, como era el caso para algunas imágenes *png* con fondo transparente. También se descartaron imágenes que después del proceso de re-escalado quedaban irreconocibles, debido a la pérdida de muchos detalles en la imagen o las proporciones de las dimensiones de la imagen original diferían mucho al formato cuadrado 1:1 de la imagen final de 224x224 píxeles.

Etiqueta	Categoría	Número
0	None	1.451
1	Activity Diagram	591
2	Sequence Diagram	800
3	Class Diagram	972
4	Component Diagram	360
5	Use Case Diagram	844
6	Cloud Diagram	963
-	Total	5.981

Cuadro 4.1: Distribución del dataset de entrenamiento creado.

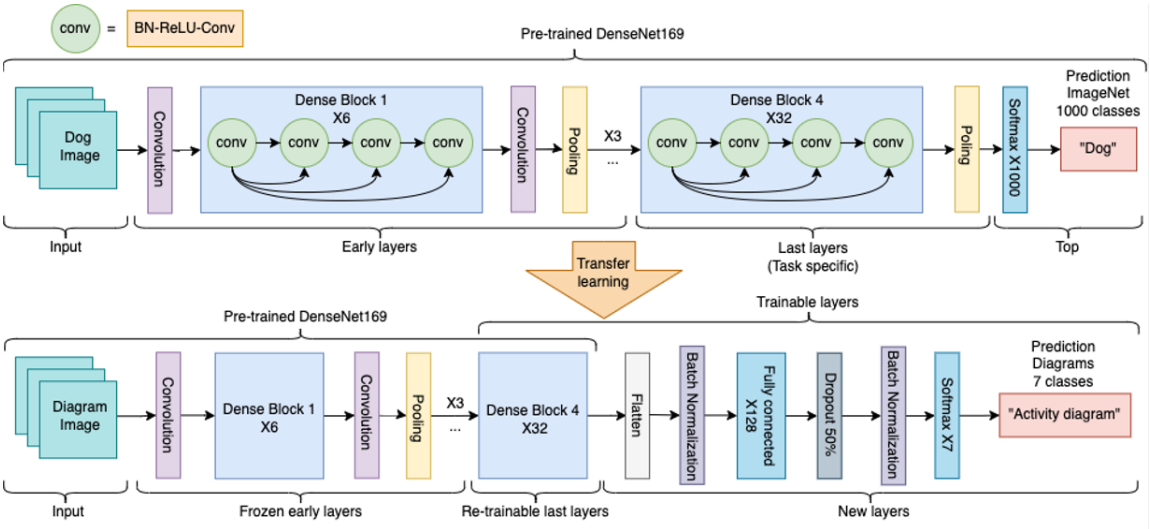

Figura 4.3: Transfer learning aplicado a la red DenseNet169.

4.2. Clasificador de imágenes

La arquitectura de red convolucional usa como base a *DenseNet169* [16] debido a su alta eficacia en este problema [35]. La figura 4.3 describe la arquitectura de la red y cómo se aplicó la técnica de *transfer learning*. En la parte superior de la figura se ve la arquitectura de una red *DenseNet169*, esta se compone de *dense blocks* y capas de transición entre medias que cambian el tamaño de la salida de la capa mediante capas de convolución y agrupamiento

o *pooling*. Cada *dense block* se compone de varias capas *conv* interconectadas entre sí (*fully connected*), cada *conv* corresponden a una secuencia *Batch Normalization-ReLU-Convolution*.

La red se presenta pre-entrenada con el *dataset ImageNet* [10], podemos dividir la red en 4 secciones, las capas de entrada que reciben la imagen, las capas iniciales en las cuales se aprende a identificar características generales en las imágenes, las capas finales en las cuales se aprenden representaciones más específicas del problema con el que se entrenó la red y por último el top donde se realiza la clasificación.

A la red se le agregaron algunas capas nuevas con el fin de mejorar la asertividad del modelo, para cada capa se refinaron los parámetros y el contenido de cada una. En el proceso de *transfer learning* no se agregaron nuevas capas de entrada, por lo cual las imágenes deben estar en el formato de *ImageNet*, es decir, de 224X244 píxeles y pre-procesadas con un re-escalando de los valores de los píxeles de entrada entre 0 y 1 y cada canal se normaliza respecto al *dataset ImageNet*. Las capas iniciales se dejan intactas y se congelan marcándolas como no entrenables, por otro lado, el último *dense block* se reentrenará y al final se agregaron nuevas capas a la red.

4.2.1. Refinamiento de la red

El proceso de refinamiento de la red consiste en buscar los valores óptimos para las variables de cada componente en la red. Esto nos permite maximizar el rendimiento de la red y validar el impacto que tiene cada componente agregado. Para esto se evaluarán distintas configuraciones de la red y se

medirá el rendimiento de la red para determinar cuál es la óptima.

Empecemos por especificar cómo se va a medir el rendimiento de la red para esta sección. Se va a emplear la técnica de validación iterada de K-pliegues aleatorios, esta consiste en dividir de forma aleatoria el *dataset* en dos subconjuntos, entrenamiento y validación, en este caso la distribución es 80/20 respectivamente. Luego se entrena el modelo con el subconjunto de entrenamiento y la métrica de eficacia que usamos es el porcentaje de asertividad, determinando la categoría de los elementos en el subconjunto de validación. Para evitar sesgos ocasionados por particularidades en la distribución de los subconjuntos, este proceso se realiza K veces con semillas aleatorias diferentes para obtener una distribución distinta en cada ocasión, el valor usado es el promedio de las K ejecuciones, todas las pruebas realizada se hicieron con un K=4.

Lo primero que se agregó a la red fue una capa para aplanar la red pre-entrenada, presentando todas las representaciones o características pre-aprendidas en una sola capa. Luego se normalizó y seguido se agrega una capa *dense* que buscan extender la capacidad de aprendizaje de la red para aprender patrones específicos del problema de clasificación de diagramas y disminuir de forma suave el número de salidas del modelo original. Vemos el impacto en la figura 4.4, donde vemos que agregar una capa extra *dense* mejora la exactitud del modelo en casi dos puntos porcentuales y mejora la función de perdida que se busca minimizar. Por otro lado, agregar más de una capa no tiene un impacto significativo.

Otro valor que es importante analizar es el tamaño adecuado de la capa *dense* que se va a agregar a la red pre-entrenada, debido a que las capas *dense*

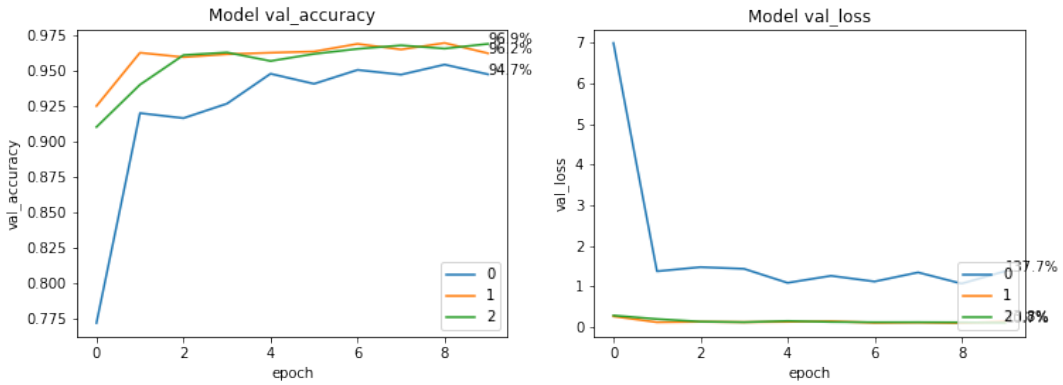

Figura 4.4: Gráficos de exactitud y función de perdida del modelo agregando 0, 1 o 2 capas Dense extra.

están interconectadas con todos los nodos de la capa anterior, el número de parámetros o tamaño de la red depende de este valor, con una capa de 512 aumentamos el número de parámetros de la red en 41 millones, por otro lado, una capa de 64 aumentará el tamaño de la red en solo 5 millones. Una red más grande implica una mayor capacidad de aprendizaje, pero por lo general requiere de *datasets* de entrenamiento más grandes y es susceptible a aprender cosas específicas del *dataset*, memorizando las imágenes y no aprendiendo patrones generales del problema. Además, el proceso de entrenamiento y predicción tomará más tiempo en función del tamaño de la red. En la figura 4.5 se ve que el peor rendimiento se da con una capa *dense* de tamaño 512, los tamaños inferiores tiene rendimientos similares entre sí, pero no es deseable tener una red tan grande, por lo que se eligió la de 128.

Luego se agregó a la red una capa de *dropout* para evitar *overfitting* en la red, esta aleatoriamente cambia a 0 los valores de activación de algunos nodos, en la figura 4.6 vemos que aumenta la exactitud del modelo y disminuye los valores de la función de pérdida.

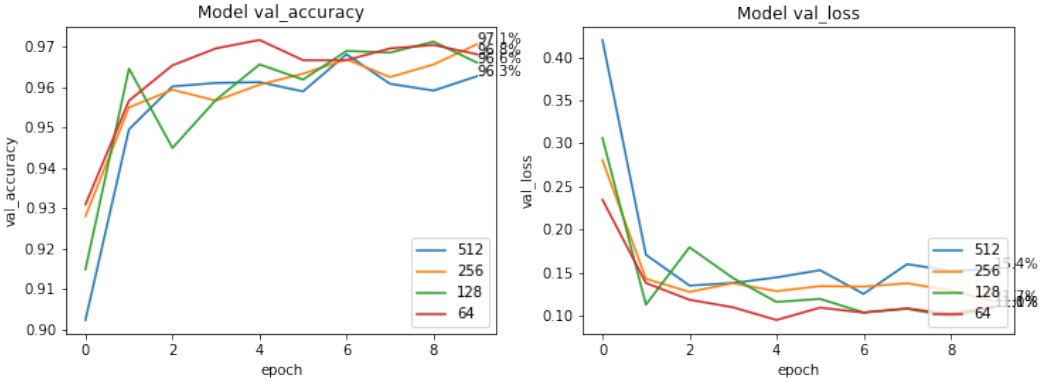

Figura 4.5: Gráficos de exactitud y función de perdida del modelo usando una capa Dense inicial de diferente tamaño.

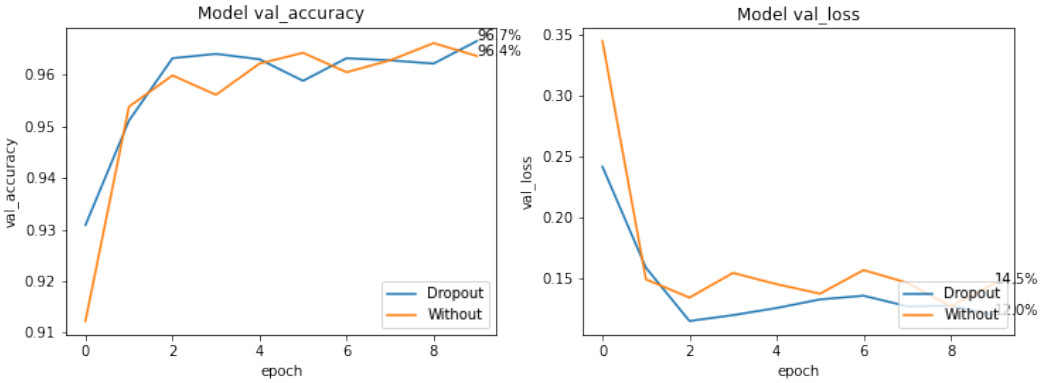

Figura 4.6: Gráficos de exactitud y función de perdida del modelo con y sin dropout.

También podemos elegir reentrenar o no el último *dense block* de la red original. En la figura 4.7 vemos que esto mejora el rendimiento del modelo debido a que seguramente en esa capa ha aprendido patrones específicos para identificar las clases en *ImageNet* que no nos son útiles en la clasificación de diagramas.

Por último, podemos aplicar *fine-tuning*, lo cual consiste en tomar el modelo reentrenado usando *transfer learning* y descongelar todas sus capas.

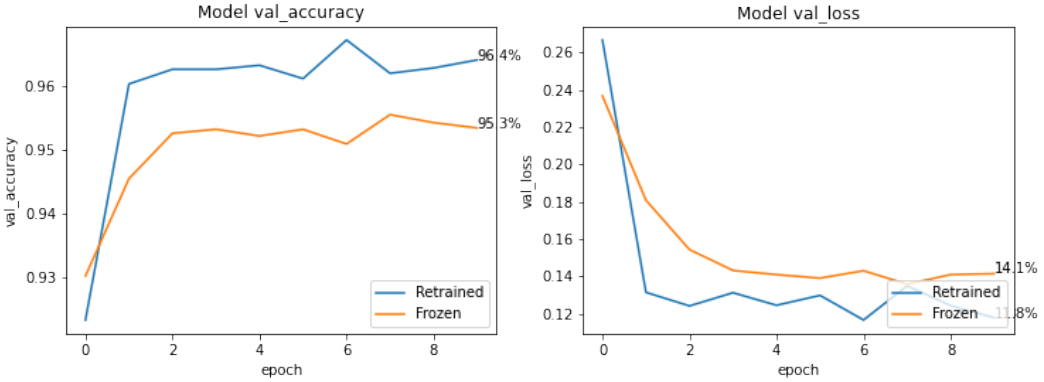

Figura 4.7: Gráficos de exactitud y función de perdida del modelo con y sin re-entrenamiento del último Dense Block.

Luego reentrenarlo por completo nuevamente, pero con una tasa de aprendizaje muy baja, con el objetivo de reentrenar ligeramente todo el modelo, mejorando el rendimiento de la red. Hay que ser muy cuidadosos con la tasa de aprendizaje, ya que se puede generar fácilmente *overfitting* en la red. En este caso se usó una tasa de aprendizaje de 1×10^{-6}, lo que permitió aumentar otros dos puntos porcentuales la precisión del modelo.

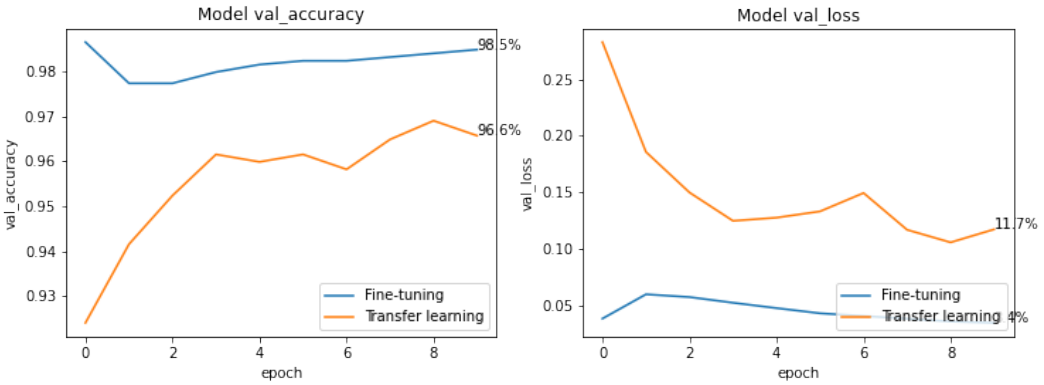

Figura 4.8: Gráficos de exactitud y función de perdida del modelo con y sin fine-tuning.

Del proceso de refinamiento pudimos encontrar que para este caso, con

este *dataset* para el problema de clasificación de diagramas, la configuración más óptima aplicando la técnica de *transfer learning* es reentrenar el último *dense block*, agregando una capa *dense* de tamaño 128, seguido de una capa de *dropout*, como está descrito en la figura 4.3.

4.3. Extracción y predicción de imágenes

Para el análisis se tomó una muestra aleatoria de *GHTorrent* [13] compuesta de 510.068 repositorios, de los cuales, 287.201 se clonaron exitosamente, sobre estos se analizó el contenido de cada uno en búsqueda de archivos con extensiones de imagen (*jpg*, *png*, *jpeg*). Una vez identificadas las imágenes se les aplicaron otros filtros, como un tamaño con un mínimo de 100x100 píxeles. Se usó el *.git* local para extraer información de la última fecha de modificación de la imagen y del repositorio. Por último, se creó un CSV[3] con la información de las imágenes y el proyecto al que pertenecen. Este proceso duró una semana y se obtuvieron, 2'469.206 imágenes equivalentes a 231 GB en datos.

4.3.1. Predicción

El proceso de predicción de los datos se realizó por batches de 1000 imágenes, en cada batch se cargaban las imágenes, normalizaban a RGB y reescalaban a 224x224 píxeles usando el algoritmo de escalado de muestreo de *Lanczos*, esto era necesario para que las imágenes estuvieran en el formato

[3]Rodríguez Torres, S. A. (2022): Git labeled dataset of image diagrams. Figshare. https://doi.org/10.6084/m9.figshare.20400999.v2

adecuado para ser analizadas por la red neuronal. Con la imagen resultado del proceso anterior se predecía el tipo de imagen usando la red previamente entrenada[4] con la mejor configuración. Los resultados de la predicción son los valores de activación de la función *softmax* en la última capa, este valor de activación es un número entre 0 y 1 por cada categoría, lo cual describe la distribución categórica de la imagen. Es decir, el valor asociado a una categoría representa la probabilidad de que una imagen pertenezca a dicha categoría, por lo tanto, la sumatoria de los valores de todas las categorías es 1. La categoría cuya probabilidad sea la más alta es la seleccionada como la categoría predicha, esta es almacenada junto con todas las otras probabilidades en el CSV[3], el cual luego se usó para el análisis posterior.

4.3.2. Validación de resultados

Se comprobó la calidad de las predicciones realizadas que se encontraban en el CSV[3]. Para esto se usó un *script* que buscaba las imágenes con una probabilidad máxima menor al 30 %. Este tipo de imágenes indican que la red no tiene certeza sobre la categoría de la imagen. Esto indica que el *dataset* utilizado para entrenar la red seguramente carece de imágenes similares que permitan aprender a qué categoría pertenecen la imagen. Se agregaron 800 imágenes de este tipo al *dataset* de entrenamiento y se clasificaron de forma manual, la mayoría de las imágenes agregadas pertenecían a la categoría *none* y se componían de diagramas de circuitos, planos de obras, pantallazos de interfaces gráficas, iconos, entre otros. Esto permitió enrique-

[4]Rodríguez Torres, S. A. (2022): TensorFlow/Keras convolutional neural network DenseNet169 based for diagram image classification. Figshare. https://doi.org/10.6084/m9.figshare.20401074.v3

cer la variedad de imágenes, incrementando la probabilidad promedio de las predicciones realizadas por el modelo en un 1.2 % de 98.1 % a 99.3 %. Sin embargo, vemos un mayor incremento entre las categorías que contiene un diagrama, pasando de 80.4 % a 83.4 %, y se redujo la cantidad de imágenes detectadas como diagramas en casi 8.000, revisando esas imágenes eran en su mayoría imágenes similares a las que se agregaron al *dataset* y se habían clasificado erróneamente en la primera iteración.

Capítulo 5

Resultados y análisis

En este capítulo, se presentan los resultados obtenidos y se responderán las preguntas planteadas en el capítulo 1. Para ello, se presentan los resultados del análisis realizado sobre las imágenes de los repositorios *Git*, también se mostrará el rendimiento final de la red obtenido del proceso de entrenamiento y refinamiento.

5.1. Rendimiento del clasificador de imágenes

La clasificación de una imagen en una categoría puede resultar en uno de cuatro casos posibles:

- Las imágenes que han sido acertadamente clasificadas se consideran un Verdadero positivo (VP).
- Las imágenes que pertenecían a una categoría, pero fueron clasificadas con otra, se consideran Falsos positivos (FP).

- Las imágenes de otra categoría que han sido identificadas correctamente se consideran como Verdadero negativo (VN).

- Las imágenes de una categoría identificadas incorrectamente como imágenes de otras clases se consideran como Falso Negativo (FN).

Para medir el rendimiento de la red se usaron las siguientes métricas:

- La exactitud (*accuracy*): Es la proporción del conjunto predicho adecuadamente a partir de todas las muestras.

- La precisión o valor positivo predicho (*precision*): Es la relación entre las imágenes correctamente identificadas que pertenecen a una categoría y todas las imágenes identificadas como esta categoría.

- La recuperación o exhaustividad (*recall*): Es la fracción que indica cuántos de los verdaderos positivos fueron encontrados de una categoría.

- Valor F1 (*F1-score*): Es la media armónica entre los valores de la precisión y la exhaustividad.

Estas métricas normalmente se utilizan en la evaluación del rendimiento de los clasificadores de imágenes y fueron calculadas usando las siguientes fórmulas:

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN}$$

$$Precision = \frac{VP}{VP + FP}$$

$$Recall = \frac{VP}{VP + FN}$$

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Category	Precision	Recall	F1-score	Support
0	0.9700	0.9864	0.9782	295
1	0.9828	0.9661	0.9744	118
2	0.9866	0.9932	0.9899	148
3	0.9911	0.9911	0.9911	225
4	0.9839	0.9531	0.9683	64
5	0.9941	0.9941	0.9941	169
6	0.9943	0.9831	0.9887	178
Accuracy			0.9850	1197
Macro avg	0.9861	0.9810	0.9835	1197
Weighted avg	0.9850	0.9850	0.9850	1197

Cuadro 5.1: Rendimiento del modelo evaluado para una muestra aleatoria que divide el *dataset* en dos subconjuntos, entrenamiento y validación en una proporción 80/20 respectivamente.

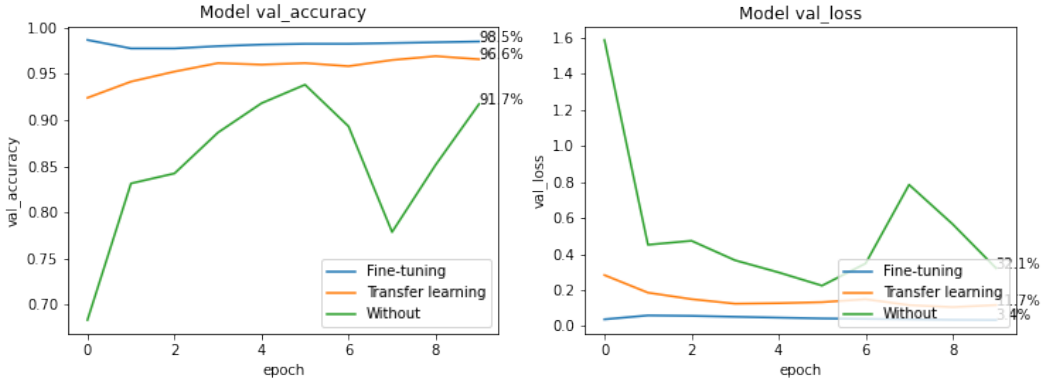

Figura 5.1: Gráficos de exactitud y función de perdida del modelo al usar transfer learning con fine-tuning, solo transfer learning o entrenar la red completamente.

Q1 ¿Cuál es el impacto del uso de *transfer learning* y *fine-tuning* en el desempeño de un clasificador de imágenes enfocado en diagramas?

RQ1 En la Figura 5.1 se puede ver el impacto de usar *transfer learning* con *fine-tuning*. Podemos notar que al aplicar *transfer learning* la red tiene

una curva de aprendizaje menos empinada y el resultado final es mejor, además *fine-tuning* nos permite mejorar ligeramente el rendimiento de la red. Es posible usar *transfer learning* tomando como base una red pre-entrenada de propósito general como *ImageNet* para resolver el problema de clasificación de diagrama a pesar de contener imágenes poco similares. Puede ver los detalles de rendimiento del clasificador utilizando *transfer learning* con *fine-tuning* en el cuadro 5.1. Para llegar a un resultado similar entrenando la red por completo deberíamos alargar el proceso de entrenamiento, posiblemente más allá de la capacidad de nuestro *dataset* de solo 5 mil imágenes, generando *overfitting*. Esto hace necesario un *dataset* y un proceso de entrenamiento más extenso.

5.2. Análisis de imágenes en proyectos Git

Q2 ¿Qué tan común es cargar imágenes en repositorios Git, si son diagramas, qué tipo de diagramas son?

RQ2 De los repositorios analizados, 98.207 contenían una imagen, siendo cerca del 34 % del total, es común subir imágenes a los repositorios. Sin embargo, como era de esperarse, la mayoría de imágenes son de la categoría *none*, conformando estas el 97 % de las imágenes. Como era de esperarse el tipo de diagrama más común es el diagrama de clases, los diagramas menos comunes son los diagramas de secuencia, seguidos de los diagramas *cloud*, por lo que los diagramas *cloud* no son muy comunes en los repositorios, se encuentran en solo el 1.7 % de los repositorios y representan solo el 0.17 % de las imágenes. La distribución completa y la probabilidad de asertividad

promedio de cada categoría están descritas en el cuadro 5.2.

Categoría	**Número**	**Porcentaje**	**Probabilidad**
None	2412201	97.89	0.9941
Activity Diagram	4567	0.19	0.7994
Sequence Diagram	3287	0.13	0.8512
Class Diagram	26975	1.09	0.8555
Component Diagram	8177	0.33	0.8068
Use Case Diagram	4906	0.20	0.8432
Cloud Diagram	4161	0.17	0.7556
Any Diagram	52073	2.11	0.8335

Cuadro 5.2: Distribución de las imágenes por categoría.

Q3 ¿Cuál es la proporción de proyectos Git que contienen diagramas y como se distribuyen por tipo de diagrama?

RQ3 En el cuadro 5.3 se presenta la distribución por categoría de imagen en los repositorios. Cerca del 10 % de los repositorios con imágenes tenían algún tipo de diagrama, lo que corresponde al 3,7 % del total de repositorios analizados. No es una práctica común agregar diagramas en forma de imagen directamente en los repositorios.

Categoría	**Repositorios**	**Porcentaje**
None	96555	98.32
Activity Diagram	1810	1.84
Sequence Diagram	1355	1.38
Class Diagram	5774	5.88
Component Diagram	3351	3.41
Use Case Diagram	1997	2.03
Cloud Diagram	1757	1.79
Any Diagram	10642	10.84
Total with image	98207	100
Total	287201	-

Cuadro 5.3: Distribución de las imágenes por repositorio.

Q4 Con el fin de saber si los diagramas están desactualizados, ¿Cuál

es la diferencia en tiempo entre la última actualización del diagrama y el repositorio?

RQ4 La diferencia en el tiempo entre la publicación del último *commit* que modificó la imagen y el último *commit* del repositorio, se presenta en el cuadro 5.4. Podemos ver que, la media para las imágenes identificadas como diagramas es de 554 días, por lo que podemos suponer que es muy probable que los diagramas se encuentren desactualizados, debido a cambios en el código fuente realizados durante este periodo de tiempo.

Categoría	**Días**
None	351
Activity Diagram	513
Sequence Diagram	482
Class Diagram	561
Component Diagram	526
Use Case Diagram	666
Cloud Diagram	530
Any Diagram	554

Cuadro 5.4: Delta en días entre la última actualización de la imagen y el repositorio.

Capítulo 6

Conclusiones y trabajo futuro

En este proyecto, se construyó un *dataset*[2] de imágenes etiquetado con 7 categorías, compuesto de casi de 6 mil imágenes, alrededor del problema de la clasificación de imágenes para diagramas de software. Este se usó para entrenar una red convolucional *DenseNet169* que fue reentrenada y modificada para usar la técnica de *transfer learning*, obteniendo resultados de exactitud del 98.6 % y un *f1-score* de 98.3 % contra el subconjunto de validación. Esto comprueba que es validó usar el conocimiento aprendido en la clasificación de un *dataset* de uso general como *ImageNet* en una tarea distante, como es la clasificación de diagramas de software.

Se aplicaron técnicas de minería de repositorios para analizar con ayuda del modelo entrenado[4] más de medio millón de repositorios Git, extrayendo 2'469.206 imágenes. Estas se clasificaron y se creó el *dataset*[3] que contiene información de las imágenes, la categoría, la predicción y el repositorio al que pertenecen.

Del análisis se identificó que solo el 3,7 % de los repositorios contaban con

algún tipo de diagrama. No es una práctica común subir imágenes de diagramas de software en los repositorios de código. Además, los diagramas son propensos a desactualizarse, la diferencia media entre la última actualización del repositorio y de la imagen es de 554 días, por lo que podemos suponer que es muy probable que haya habido cambios durante ese periodo de tiempo en la base de código que no se reflejaron en los diagramas, soportando la hipótesis planteada en la problemática.

La minería de repositorios es un área de estudio con mucho potencial, nos permite entender cómo los ingenieros de software interactúan con los artefactos de software durante el ciclo de desarrollo, y así validar hipótesis sobre comportamientos y tendencias de la comunidad de software, indicándonos que tipo de herramientas o prácticas pueden ser útiles a futuro. En este caso se detectó que los diagramas son propensos a estar desactualizados, por lo que se detectó la necesidad de incorporar prácticas durante ciclo de desarrollo para incentivar la actualización de los diagramas luego de realizar un cambio, por otro lado, vemos la necesidad de usar y desarrollar herramientas que asistan en el ciclo de vida de los diagramas para mantenerlos al día. Por ejemplo, generadores de diagramas a partir del código fuente o herramientas que sean capaces de identificar que una modificación en el código fuente afecta un diagrama y este debe ser actualizado.

Del desarrollo de este estudio destaca la importancia de los datos de entrenamiento para obtener buenos resultados con las predicciones de la red. Las redes neuronales profundas tienen una mayor capacidad de aprendizaje, pero requieren de un *dataset* de mayor tamaño para sacar todo su potencial. Sin embargo, técnicas como *transfer learning* nos permiten entrenar la red con

datasets más pequeños y reducir la tasa de error y obtener una mejor precisión en las predicciones. Al trabajar con *datasets* reducidos juega un rol crucial la calidad de los datos, para evitar sesgos se debe evitar la recurrencia de patrones no asociados a la clase que queremos identificar, mediante muestras variadas. Por ejemplo, todas las imágenes de una clase tienen fondo gris, la red puede asociar cualquier imagen con fondo gris a esa clase ignorando la característica que distingue esa clase.

En este estudio nos centramos en el procesamiento de imágenes, pero hay otra aproximación al problema utilizando el procesamiento del lenguaje natural (NLP) para la comprensión semántica de los diagramas. Las características reconocidas a partir del texto nos pueden permitir enriquecer el modelo actual y mejorar la calidad de las predicciones usando ambos métodos en conjunto.

En futuros estudios se puede usar el clasificador propuesto para realizar la identificación de artefactos de documentación tipo diagrama, lo cual facilita futuras investigaciones en el área de minería de repositorios. Por ejemplo, el problema de *Traceability link recovery* (TLR) que busca determinar las relaciones entre grupos de artefactos de un repositorio, recuperando los enlaces entre artefactos de código, documentación o pruebas [2]. Nuevas investigaciones en ese campo como [4] incorporan estrategias de *machine learning* para automatizar el proceso de reconocimiento del lenguaje natural, pero no se han aplicado a diagramas de software en imágenes.

Bibliografía

[1] *About the Unified Modeling Language Specification Version 2.5 — omg.org*. https://www.omg.org/spec/UML/2.5/.

[2] G. Antoniol et al. «Recovering traceability links between code and documentation». En: *IEEE Transactions on Software Engineering* 28.10 (2002), págs. 970-983. DOI: 10.1109/TSE.2002.1041053.

[3] Robert Audi. *Cambridge Dictionary of Philosophy*. Cambridge University Press, 1999.

[4] Thazin Win Win Aung, Huan Huo y Yulei Sui. «A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis». En: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, págs. 14-24. ISBN: 9781450379588. DOI: 10.1145/3387904.3389251. URL: https://doi.org/10.1145/3387904.3389251.

[5] Olusola Tope Babalola. *Automatic recognition and interpretation of finite state automata diagrams*. Dic. de 2015. URL: https://scholar.sun.ac.za/handle/10019.1/97814.

[6] Natalie Best, Jordan Ott y Erik Linstead. «Exploring the efficacy of transfer learning in mining image-based software artifacts». En: *Journal Of Big Data* 7 (ago. de 2020). DOI: 10.1186/s40537-020-00335-4.

[7] Gosala Bethany et al. «Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network». En: *Applied Sciences* (mayo de 2021). DOI: 10.3390/app11094267.

[8] Francois Chollet. *Deep learning with python*. en. New York, NY: Manning Publications, 2017. ISBN: 9781617294433.

[9] *Dataset of the Paper "Automatically Classifying UML Class Diagrams from Images using Deep Learning"*. Zenodo, ene. de 2021. DOI: 10.5281/zenodo.4252890. URL: https://doi.org/10.5281/zenodo.4252890.

[10] Jia Deng et al. «ImageNet: a Large-Scale Hierarchical Image Database». En: jun. de 2009, págs. 248-255. DOI: 10.1109/CVPR.2009.5206848.

[11] Arden Dertat. *Applied Deep Learning - Part 4: Convolutional neural networks*. en. 2017. URL: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.

[12] Marta de Esteban Belzuz, Carlos Martin y Francisco Morillo. *MPM Construyendo una arquitectura Segura y altamente Disponible en AWS*. URL: https://aws.amazon.com/es/blogs/aws-spanish/mpm-construyendo-una-arquitectura-segura-y-altamente-disponible-en-aws/.

[13] Georgios Gousios. «The GHTorrent dataset and tool suite». En: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, págs. 233-236. ISBN: 978-1-4673-2936-1. URL: http://dl.acm.org/citation.cfm?id=2487085.2487132.

[14] Douglas M. Hawkins. «The problem of overfitting». En: *ChemInform* 35.19 (2004). DOI: 10.1002/chin.200419274.

[15] Regina Hebig et al. «The quest for open source projects that use UML: mining GitHub». En: oct. de 2016, págs. 173-183. DOI: 10.1145/2976767.2976778.

[16] Gao Huang et al. «Densely Connected Convolutional Networks». En: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Jul. de 2017, págs. 2261-2269. DOI: 10.1109/CVPR.2017.243.

[17] B. Karasneh y Michel Chaudron. «Extracting UML models from images». En: mar. de 2013, págs. 169-178. DOI: 10.1109/CSIT.2013.6588776.

[18] B. Karasneh y Michel Chaudron. «Img2UML: A system for extracting UML models from images». En: sep. de 2013, págs. 134-137. DOI: 10.1109/SEAA.2013.45.

[19] B. Karasneh y Michel Chaudron. «Online Img2UML Repository: An Online Repository for UML Models». En: oct. de 2013.

[20] C. Lanczos. «An iteration method for the solution of the eigenvalue problem of linear differential and integral operators». En: *Journal of*

research of the National Bureau of Standards 45.4 (1950), pág. 255. ISSN: 0091-0635. DOI: 10.6028/jres.045.026. URL: http://dx.doi.org/10.6028/jres.045.026.

[21] D. Lu y Q. Weng. «A survey of image classification methods and techniques for improving classification performance». En: *International Journal of Remote Sensing* 28.5 (2007), págs. 823-870. DOI: 10.1080/01431160600746456. eprint: https://doi.org/10.1080/01431160600746456. URL: https://doi.org/10.1080/01431160600746456.

[22] Ramírez Luna et al. «Modelo para Almacenar y Recuperar Métricas de Software». Español. En: *Conciencia Tecnológica* (2010). ISSN: 1405-5597. URL: https://www.redalyc.org/articulo.oa?id=94415753006.

[23] Emmanuel Maggiori et al. «Towards Recovering Architectural Information from Images of Architectural Diagrams». En: sep. de 2014.

[24] Kaushil Mangaroliya y Het Patel. «Classification of Reverse-Engineered Class Diagram and Forward-Engineered Class Diagram using Machine Learning». En: (nov. de 2020).

[25] Amitha Mathew, Amudha Arul y S. Sivakumari. «Deep Learning Techniques: An Overview». En: ene. de 2021, págs. 599-608. ISBN: 978-981-15-3382-2. DOI: 10.1007/978-981-15-3383-9_54.

[26] Valentín Moreno et al. «Automatic Classification of Web Images as UML Static Diagrams Using Machine Learning Techniques». En: *Applied Sciences* 10 (abr. de 2020), pág. 2406. DOI: 10.3390/app10072406.

[27] Jordan Ott, Abigail Atchison y Erik Linstead. «Exploring the applicability of low-shot learning in mining software repositories». En: *Journal Of Big Data* 6 (mayo de 2019), pág. 35. DOI: 10.1186/s40537-019-0198-z.

[28] Mr Pankaj S. Parsania y Dr Paresh V. Virparia. «A comparative analysis of image interpolation algorithms». En: *nternational journal of advanced research in computer and communication engineering* 5.1 (2016), págs. 29-34. ISSN: 2319-5940. DOI: 10.17148/ijarcce.2016.5107. URL: http://dx.doi.org/10.17148/ijarcce.2016.5107.

[29] Truong Ho-Quang et al. «Automatic Classification of UML Class Diagrams from Images». En: dic. de 2014. DOI: 10.1109/APSEC.2014.65.

[30] George Reese. *Cloud application architectures: Building Applications and infrastructure in the cloud*. O'Reilly, 2009.

[31] Romain Robbes. «Minería de repositorios de software para ayudar a los desarrolladores». En: *Revista Bits de Ciencia* 5 (2011), págs. 2-7. ISSN: 0718-8013. URL: https://revistasdex.uchile.cl/index.php/bits/issue/view/130.

[32] Gregorio Robles et al. «An Extensive Dataset of UML Models in GitHub». En: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Abr. de 2017, págs. 519-522. DOI: 10.1109/MSR.2017.48.

[33] Sumit Saha. *A comprehensive guide to Convolutional Neural Networks - the eli5 way*. Nov. de 2022. URL: https://towardsdatascience.com/

a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[34] *Selenium: Definition, how it works and Why you need it*. Jun. de 2022. URL: https://www.browserstack.com/selenium.

[35] Sergei Shcherban et al. «Multiclass Classification of Four Types of UML Diagrams from Images Using Deep Learning». En: mayo de 2021. DOI: 10.18293/SEKE2021-185.

[36] Crystal Song. *Software architecture diagramming and patterns*. Abr. de 2022. URL: https://www.educative.io/blog/software-architecture-diagramming-and-patterns.

[37] Farhana Sultana, Abu Sufian y Paramartha Dutta. «Advancements in Image Classification using Convolutional Neural Network». En: *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*. 2018, págs. 122-129. DOI: 10.1109/ICRCICN.2018.8718718.

[38] Warnick Sean West Jeremy Ventura Dan. «A Theoretical Foundation for Inductive Transfer». En: *Spring Research* (2007).

[39] Rikiya Yamashita et al. «Convolutional neural networks: an overview and application in radiology». En: *Insights into Imaging* 9 (jun. de 2018). DOI: 10.1007/s13244-018-0639-9.