



ugr | Universidad
de Granada

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Etiquetado de imágenes en química

Autor

Pedro Bedmar López

Directora

Rocío Celeste Romero Zaliz



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
Granada, julio de 2022

Etiquetado de imágenes en química

Pedro Bedmar López

Palabras clave: palabra_clave1, palabra_clave2, palabra_clave3,

Resumen

Poner aquí el resumen.

Image labelling in chemistry

Pedro Bedmar López

Keywords: Keyword1, Keyword2, Keyword3,

Abstract

Write here the abstract in English.

Granada, a 7 de julio de 2022.

Dña. **Rocío Celeste Romero Zaliz**, Profesora Titular del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado *Etiquetado de imágenes en química*, ha sido realizado bajo su supervisión por **Pedro Bedmar López**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 7 de julio de 2022.

La directora:

Rocío Celeste Romero Zaliz

Agradecimientos

Poner aquí agradecimientos...

Índice general

1. Introducción	1
1.1. Motivación del proyecto	1
1.2. Objetivos	2
2. Gestión y planificación del proyecto	3
2.1. Metodología	3
2.2. Planificación	6
2.3. Gestión de recursos	7
2.3.1. Recursos humanos	7
2.3.2. Recursos materiales	7
2.3.3. Recursos software	8
2.4. Presupuesto	9
2.4.1. Coste de recursos humanos	9
2.4.2. Otros costes	9
2.4.3. Presupuesto final	9
3. Estado del arte	11
3.1. Cheminformatics	11
3.1.1. Compuestos orgánicos y su representación	11
3.1.2. Representación en el ordenador	13
3.1.3. Fuentes y bases de datos	16
3.1.4. Métodos de búsqueda	17
3.1.5. Métodos para análisis de datos	17
3.2. Deep learning	18
3.2.1. Redes neuronales multicapa	19
3.2.2. Redes neuronales convolutivas	20
3.2.3. Autocodificadores	22
3.2.4. Redes generativas antagónicas	24
3.2.5. Transformers	25
3.3. Bibliotecas y paquetes	27
3.3.1. Taming Transformers for High-Resolution Image Synthesis	27
3.3.2. Pytorch	29

4. Metodología de investigación	33
4.1. Balanceo del <i>dataset</i>	34
4.2. Generación de <i>hard negatives</i>	38
4.3. Clasificación de imágenes	40
5. Experimentación	47
5.1. Transformaciones sobre el <i>dataset</i>	48
5.2. Generación de imágenes sintéticas	51
5.2.1. Ruido Perlín	61
5.3. Clasificación de imágenes	68
5.3.1. Clasificador sobre el <i>dataset</i> sin <i>hard negatives</i>	69
5.3.2. Clasificador sobre el <i>dataset</i> con <i>hard negatives</i>	75
6. Conclusiones	79

Índice de figuras

2.1.	Fases de la metodología SCRUM [3]	5
2.2.	Diagrama de Gantt con la temporización de las tareas.	7
3.1.	Estructura de la cafeína	12
3.2.	Estructura del metano	12
3.3.	Dos formas de representar el butano	13
3.4.	Dos posibles codificaciones de la melatonina en SMILES [42]	14
3.5.	Ciclos en SMILES	14
3.6.	Contenido de un archivo MOL [19]	15
3.7.	Modelo de integración y disparo en las neuronas artificiales [13]	19
3.8.	Resultado de convolucionar la entrada I con el kernel K [10] .	21
3.9.	Secciones en las que se divide un autocodificador [33]	22
3.10.	Espacio latente en un autocodificador regular vs en VAE [33]	23
3.11.	Transformer aplicado a la generación de texto [24]	25
3.12.	Los dos tipos de capas que forman el Transformer [24]	26
3.13.	Modelo propuesto por P. Esser et al. [18]	27
3.14.	Ejemplos generados utilizando el modelo basado en transformer [18]	28
3.15.	Logo de PyTorch	29
4.1.	Ejemplos de muestras positivas del <i>dataset</i>	34
4.2.	Ejemplos de muestras negativas del <i>dataset</i>	34
4.3.	Diferentes <i>datasets</i> generados a partir del original	37
4.4.	Límite de decisión separando dos clases. El individuo en rojo representa un <i>hard negative</i> perteneciente a la clase azul, ya que se encuentra muy cercano al límite de decisión.	38
4.5.	Diferentes <i>hard negatives</i> obtenidos entrenando el modelo con diferentes <i>data augmentation</i> y número de épocas.	39
4.6.	Dos <i>datasets</i> para entrenar dos clasificadores.	40
4.7.	La función de entropía cruzada L_{CE} mide la distancia entre la probabilidad representada en el <i>logit S</i> y la probabilidad real representada mediante T [29]. Durante el entrenamiento, el algoritmo tratará de minimizar esta distancia.	44

4.8.	División <i>train-test</i>	45
4.9.	Realizamos una <i>grid search</i> sobre ambos datasets, y elegimos el modelo que mejor resultado da sobre cada uno de ellos.	45
5.1.	Imágenes generadas aplicando <i>data augmentation</i> 1	49
5.2.	Imágenes generadas aplicando <i>data augmentation</i> 2	49
5.3.	Imágenes generadas aplicando <i>data augmentation</i> 3	50
5.4.	Resultados de entrenar los modelos sobre el conjunto de datos sin <i>data augmentation</i> . La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.	53
5.5.	Resultados de entrenar los modelos sobre el conjunto de datos con <i>data augmentation</i> 1. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.	55
5.6.	Resultados de entrenar los modelos sobre el conjunto de datos con <i>data augmentation</i> 2. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.	57
5.7.	Resultados de entrenar los modelos sobre el conjunto de datos con <i>data augmentation</i> 3. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.	59
5.8.	El modelo tiene la capacidad de modificar la imagen para que se parezca a una molécula.	61
5.9.	Interpolación de gradientes en el Ruido Perlín 1D. [30]	62
5.10.	Ejemplos de Ruido Perlín con distinta amplitud y frecuencia.	62
5.11.	Resultados de entrenar con imágenes con <i>data augmentation</i> 2. La primera columna representa la imagen de entrada (ruido Perlín en todos los casos), el resto los diferentes modelos entrenados desde las 70 hasta las 90 épocas tomadas de 5 en 5.	63
5.12.	<i>Hard negatives</i> finales tras ser post-procesados.	65
5.13.	Resultados de entrenar con ejemplos negativos. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde las 70 hasta las 170 épocas tomadas de 20 en 20.	66
5.14.	Dos <i>datasets</i> para entrenar dos clasificadores.	67
5.15.	Variación del error durante el entrenamiento utilizando la configuración LeNet5-Xavier-0.05-SGD. Las iteraciones se corresponden con las producidas durante la validación cruzada.	70
5.16.	Más ejemplos de cómo varía el error durante el entrenamiento utilizando diferentes configuraciones sobre LeNet5.	71

5.17. Modelos entrenados con la configuración LeNet5-He-Adam-0.00005, utilizando <i>datasets</i> de diferente tamaño (desde 25 hasta 700 imágenes).	72
5.18. Evolución del error de <i>training</i> durante el entrenamiento de un modelo con configuración VGG16-Xavier-Adam-0.00005 sobre un <i>dataset</i> sin <i>hard negatives</i>	72
5.19. Más ejemplos de cómo varía el error durante el entrenamiento de diferentes modelos utilizando configuraciones basadas en LeNet5, sobre un <i>dataset</i> sin <i>hard negatives</i>	73
5.20. Evolución del error de <i>training</i> durante el entrenamiento del modelo final sobre el <i>dataset</i> sin <i>hard negatives</i>	74
5.21. Evolución del error de <i>training</i> durante el entrenamiento de un modelo con configuración AlexNet-Xavier-Adam-0.00005 sobre un <i>dataset</i> con <i>hard negatives</i>	77
5.22. Evolución del error de <i>training</i> durante el entrenamiento del modelo final sobre el <i>dataset</i> con <i>hard negatives</i>	77

Índice de tablas

5.1.	<i>Grid search</i> utilizando la arquitectura LeNet5, entrenamiento sobre <i>dataset</i> sin <i>hard negatives</i>	69
5.2.	<i>Grid search</i> utilizando la arquitectura AlexNet, entrenamiento sobre <i>dataset</i> sin <i>hard negatives</i>	69
5.3.	<i>Grid search</i> utilizando la arquitectura VGG16, entrenamiento sobre <i>dataset</i> sin <i>hard negatives</i>	70
5.4.	Porcentaje de error del modelo final sin <i>hard negatives</i>	74
5.5.	<i>Grid search</i> utilizando la arquitectura LeNet5, entrenamiento sobre <i>dataset</i> con <i>hard negatives</i>	75
5.6.	<i>Grid search</i> utilizando la arquitectura AlexNet, entrenamiento sobre <i>dataset</i> con <i>hard negatives</i>	75
5.7.	<i>Grid search</i> utilizando la arquitectura VGG16, entrenamiento sobre <i>dataset</i> con <i>hard negatives</i>	76
5.8.	Porcentaje de error del modelo final con <i>hard negatives</i>	77

Capítulo 1

Introducción

(Aún por actualizar)

1.1. Motivación del proyecto

Desde hace décadas, en el mundo de la química ha estado presente la necesidad de almacenar, gestionar y procesar la gran cantidad de información que se genera. Con el tiempo se fueron desarrollando técnicas de tratamiento de ésta, pero no fue hasta hace algunos años cuando se acuñó el nombre de cheminformatics o chemoinformatics.

En la literatura existen diferentes definiciones para este término, discutidas en [5]. “Chem(o)informatics es un término genérico que encierra el diseño, creación, gestión, recuperación, análisis, diseminación, visualización y el uso de información química” es una de las definiciones recogidas. Otra más abierta es “La aplicación de métodos informáticos para resolver problemas de química”.

Desde la Universidad de Granada, la tutora de este TFG trabaja en este ámbito. Colabora con químicos de la Universidad de Negev y es consciente de los problemas que tienen para manejar la gran cantidad de datos que aparecen en publicaciones científicas. Un tipo de datos muy valioso son las imágenes, pero clasificarlas no es trivial: pueden ser sobre cualquier temática, algunas pueden referirse a esquemas explicando cómo funciona un modelo, otras pueden contener resultados de algún experimento, pueden ser representaciones de compuestos químicos, etc. Clasificarlas manualmente por un operario no es una opción viable, ya que se consumirían demasiados recursos en una tarea que actualmente se puede automatizar.

Es por ello que en este Trabajo de Fin de Grado vamos a crear un modelo que permita clasificar imágenes. En concreto, aquellas que contienen

representaciones de moléculas del resto. Este tipo de imágenes de moléculas en ocasiones son difíciles de clasificar, debido al parecido que presentan con otras estructuras. Por ello, también entrenaremos un clasificador con un conjunto de datos (*dataset*) que contiene *hard negatives*, ejemplos negativos que están en el límite de lo que es una imagen de una molécula química y lo que no. Crearemos estos *hard negatives* a partir de un modelo generativo capaz de sintetizar imágenes.

Aunque existen diferentes opiniones sobre el alcance de las *cheminformatics*, se puede considerar que este proyecto está dentro de sus fronteras, ya que vamos a crear una herramienta de clasificación de imágenes químicas, es decir, una herramienta que procesa y analiza este tipo de información.

1.2. Objetivos

El proyecto tiene como fin cumplir los siguientes objetivos:

- **[OBJ1]** Refinar un *dataset* de clasificación de imágenes de compuestos químicos ya existente.
- **[OBJ2]** Clasificar imágenes que presentan compuestos químicos de aquellas que no.

Capítulo 2

Gestión y planificación del proyecto

2.1. Metodología

En los primeros años del desarrollo de software, este se creaba sin seguir ningún enfoque formal. Muchos de los proyectos que se iniciaban terminaban fracasando por los retrasos en la entrega, el mal funcionamiento del producto o por no cumplir los requisitos del cliente. Además, la complejidad requerida en el software iba aumentando con el tiempo.

Por ello, era necesario crear un marco que permitiera metodizar el desarrollo. Es así como surge la Ingeniería del Software, que acompaña al software durante todo su ciclo vital.

En este proyecto se va a aplicar este enfoque a la hora de desarrollar el producto, apoyándome sobre una metodología de desarrollo para asegurarnos que se consiguen los objetivos en el tiempo previsto, detectando posibles amenazas y problemas a tiempo.

Las metodologías se pueden clasificar en dos grandes bloques [35], tradicionales y ágiles. Las tradicionales son las que primero surgieron, se caracterizan por definir rígidamente los requisitos al inicio del proyecto. En ellas, se aplica una serie de etapas de forma lineal, y una vez alcanzada una de ellas no se puede volver atrás. Por todo esto no se adaptan bien a los cambios.

En general, los proyectos de software tienden a ser cada vez más complejos. Las metodologías ágiles surgieron con el objetivo de hacer los proyectos de desarrollo más dinámicos, de forma que se adaptaran mejor al entorno y a los cambios. Se basan en una metodología incremental, donde se van construyendo prototipos del producto poco a poco, añadiendo funcionalidades hasta obtener la aplicación final. Los equipos se reúnen cada poco tiempo

para intercambiar ideas y repartir las tareas a realizar.

A la hora de elegir la metodología que se va a utilizar, se deben tener en cuenta los siguientes factores relativos a la naturaleza del proyecto:

- Se trata de un proyecto de investigación, donde se van a aplicar diferentes técnicas de Aprendizaje Automático a la resolución de un problema. Por tanto, a priori no se conoce la calidad de los resultados que se van a obtener y el número y tipo de experimentos que va a ser necesario realizar.
- La intervención del experto en química va a ser fundamental durante el desarrollo del proyecto. Aportará información y feedback esencial durante todas las fases.

Por estas razones, creo que la metodología de desarrollo que mejor se adapta a nuestras necesidades es una metodología ágil, ya que nos provee de gran flexibilidad, permitiendo el desarrollo del proyecto de una forma incremental donde obtengo feedback del cliente y experto en cada iteración.

Revisando las distintas metodologías [8], creo que una buena candidata es SCRUM. Es posiblemente una de las más utilizadas en la actualidad, y los proyectos que la aplican cuentan con las siguientes características [3]:

- **Entregable flexible:** Su contenido viene dado por lo que demanda el entorno.
- **Calendario flexible:** El entregable puede ser requerido antes o después de lo previsto.
- **Equipos pequeños:** Los equipos están formados por pocas personas, de forma que la comunicación y sincronización entre sus miembros es alta.
- **Revisiones frecuentes:** El progreso del equipo se evalúa de forma periódica y frecuentemente, de forma que se ponen en común las dificultades encontradas y se intentan resolver con la ayuda de todos los miembros.
- **Colaboración:** La colaboración entre todos los miembros del equipo es muy alta, así como entre el equipo y entidades externas como el cliente.

Cuando se trabaja con esta metodología, en cada equipo existe un miembro conocido como SCRUM manager. Es el encargado de guiar al equipo en el desarrollo y en la aplicación de la metodología. En SCRUM el equipo es muy importante y todos sus miembros participan con su opinión. Esta metodología consta de las siguientes fases [3]:

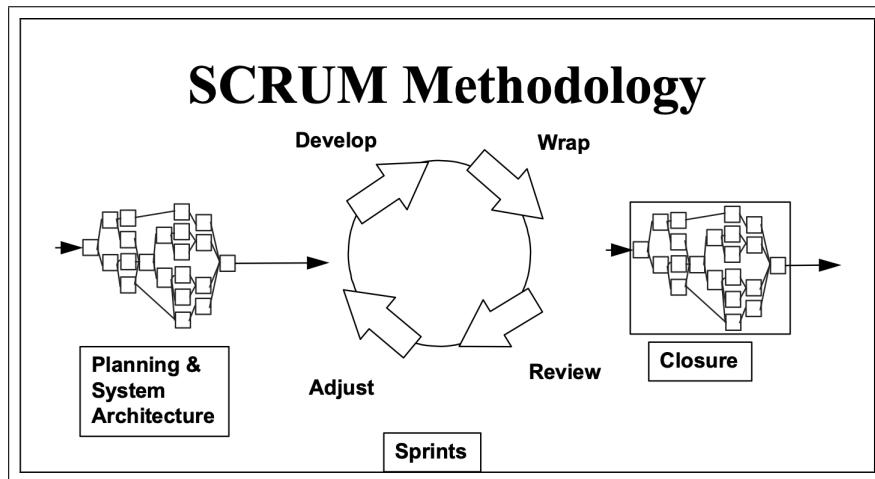


Figura 2.1: Fases de la metodología SCRUM [3]

■ **Pregame:**

- *Planificación:* En esta fase se crea la lista de tareas a realizar (backlog list), se fija la fecha de entrega del producto y su funcionalidad, se forma el equipo (o equipos) de trabajo, se valoran los riesgos que puedan surgir, los costes y finalmente se revisa todo y se aprueba el proyecto.

- *Arquitectura/Diseño de alto nivel:* Se revisan los elementos en el backlog, se realizan cambios si es necesario para poder implementarlos, se perfila la arquitectura del sistema teniendo en cuenta estos cambios y una reunión es organizada para revisar el diseño, donde cada equipo presenta una propuesta para implementar cada backlog.

■ **Game:** Corresponde con el conocido Sprint de la metodología SCRUM. Es la parte iterativa de la metodología, que se ejecuta varias veces hasta conseguir el producto final y está formada por las siguientes subtareas:

- *Desarrollo:* Lo primero que se realiza es definir los cambios que hay que realizar en los backlogs para poder implementarlos. Se dividen las tareas presentes en el backlog en paquetes, y se completan estos paquetes diseñando, desarrollando, implementando, testeando y documentando los cambios.
- *Envoltura:* Se cierran los paquetes, creando una ejecutable que incorpora los cambios y se explica como cumplen lo especificado en los backlogs.
- *Revisión:* Todos los equipos se reunen para presentar el trabajo. El desarrollo obtenido se evalúa y se añaden nuevas tareas que puedan surgir al backlog. Se evalúa el riesgo y se realizan propuestas en base a este.
- *Ajuste:* Se consolida la información recibida durante la revisión.

■ **Postgame:**

- *Cierre:* Cuando el gestor del proyecto considera que el producto está terminado y cumple con los requisitos solicitados por el cliente se entra en esta fase, donde se prepara el producto para su despliegue. Integración, generación de la documentación, testeo y marketing son algunas de las actividades que se realizan en este paso.

Estas características y fases que hemos mencionado son las especificadas en la definición original de SCRUM. Aún así, en la práctica cada proyecto las adapta a sus necesidades. En general, los sprints suelen tener una duración de 2 o 3 semanas.

2.2. Planificación

Para lograr cumplir con los objetivos, se planifican las siguientes tareas a realizar:

- [T1] Estudio del estado del arte del problema.
 - [T1.1] Lectura de publicaciones introductorias a las Cheminformatics.
 - [T1.2] Repaso de la literatura sobre aprendizaje profundo, comprendiendo los principales modelos de generación y clasificación de imágenes.
 - [T1.3] Conocimiento de las principales herramientas que se han desarrollado en ambos ámbitos.
- [T2] Estudio del *dataset* de imágenes químicas existente para su adecuación a tareas de clasificación.
 - [T2.1] Mejorar el balanceo del *dataset*.
 - [T2.2] Adaptación de un modelo de síntesis de imágenes para generar *hard negatives* y añadirlos al *dataset*, de forma que el clasificador sea capaz de trabajar con ejemplos difíciles.
- [T3] Implementación de dos modelos clasificadores de imágenes, capaces de distinguir moléculas de ejemplos negativos, uno entrenado sobre un *dataset* sin *hard negatives* y otro entrenado sobre un conjunto de datos con ellos.
 - [T3.1] Realizar pruebas con diferentes arquitecturas e hiperparámetros para comprobar cuáles se adaptan mejor al problema.
 - [T3.2] Escoger la configuración que tendrán los modelos finales.
- [T4] Documentación del trabajo realizado.

Para el éxito del proyecto es necesario asignar una temporización razonable a cada tarea. Este se realiza en el marco de un Trabajo de Fin de Grado (TFG): en la Universidad de Granada está cuantificado en 12 créditos, lo

que significan 300 horas de trabajo. Creemos que los objetivos propuestos son adecuados para ser realizados en ese número de horas, sin ser excesivos ni escasos.

Los TFGs se suelen realizar en un único cuatrimestre, que normalmente coincide con el último del grado. Esto significa cuatro meses en los que repartir las 300 horas, desde marzo hasta junio. Bien es cierto que este proyecto se empezó con algo de anterioridad, en febrero, para comenzar a profundizar en la literatura y conocer de forma más precisa las necesidades de los investigadores de Negev.

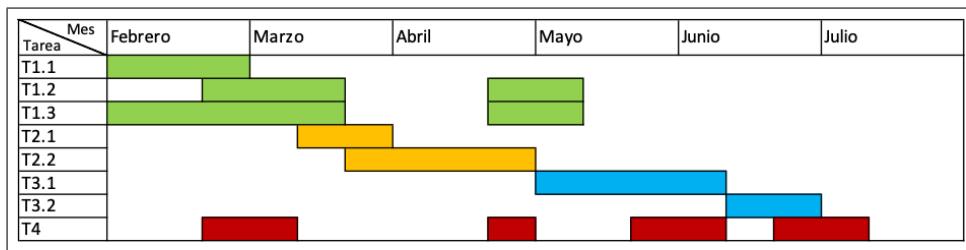


Figura 2.2: Diagrama de Gantt con la temporización de las tareas.

2.3. Gestión de recursos

2.3.1. Recursos humanos

El proyecto cuenta con dos personas trabajando sobre él:

- **Rocío Romero Zaliz.** Profesora del Departamento de Ciencias de la Computación e Inteligencia Artificial tutora del proyecto. Encargada de supervisar al alumno, guiando su trabajo y aconsejándole cuando lo necesite. Se reunirá con él de forma periódica para comentar los avances e indicar posibles pasos a seguir.
- **Pedro Bedmar López.** Estudiante del Grado en Ingeniería Informática que realiza el TFG. Se encargará de su desarrollo y de la elaboración de la memoria.

2.3.2. Recursos materiales

Una ventaja del sector tecnológico es que muchos proyectos no necesitan una gran infraestructura para llevarse a cabo. En este caso se hace uso de lo siguiente:

- **Ordenador personal.** Equipo desde el que trabajará el estudiante. Lo utilizará para consultar la literatura, implementar el código y re-

dactar la memoria. En concreto, se trata de un MacBook Pro 15' 2015, que monta un procesador Intel Core i7 a 2.5 GHz y 16 GB de RAM.

- **Monitor.** Utilizado como apoyo a la pantalla del portátil. El modelo utilizado ha sido fabricado por la marca Benq, y presenta una resolución 1920x1080.
- **Clúster GPU.** Hoy en día, la mayoría del entrenamiento de modelos en Aprendizaje Automático y Deep Learning no se lleva a cabo de forma local, sino en un clúster habilitado para ello. La Universidad de Granada, y en concreto el Instituto DaSCI (Instituto Andaluz Interuniversitario en Data Science and Computational Intelligence), cuenta con uno de ellos donde mediante SLURM se pueden enviar trabajos que se ejecutaran en una GPU, acelerando en gran medida los entrenamientos.

2.3.3. Recursos software

Se utilizan distintos programas que apoyan en distintas tareas:

- **macOS Monterey 12.3.1.** Por encima del resto de programas, se encuentra el Sistema Operativo, encargado de administrar los recursos hardware y controlar la ejecución de procesos.
- **PyCharm 2021.3.3.** IDE desarrollado por la compañía JetBrains, ideal para trabajar con proyectos en Python e incluso con Jupyter Notebooks. Además, se integra directamente mediante SSH con el clúster GPU, permitiendo la sincronización de ficheros entre el espacio de trabajo local y el remoto. Aunque es un software de pago, los estudiantes pueden descargarlo de forma gratuita.
- **Git y GitHub.** Git es un software de control de versiones que permite ir almacenando versiones intermedias del código, de forma que se puede volver hacia atrás si es necesario. También facilita el trabajo en equipo, ya que puede combinar el trabajo de múltiples personas de forma automática (siempre y cuando no hayan trabajado sobre la misma línea). GitHub es una plataforma de alojamiento de proyectos que utilizan Git. Nos es útil, ya que actúa como almacenamiento de seguridad de nuestro trabajo.
- **Google Meet.** Producto de videoconferencias de Google. Es la plataforma oficial de comunicación síncrona en la Universidad de Granada. Será utilizado en las reuniones entre los integrantes del equipo.
- **VSCODE y LATEX.** VSCODE es un editor de código desarrollado por Microsoft. Cuenta con multitud de extensiones creadas por la comunidad, entre ellas destacan algunas que permiten el formateo y compilado de ficheros LATEX. Este software es un sistema de composición de textos utilizado para escribir esta memoria.

2.4. Presupuesto

En esta sección se presenta el presupuesto del proyecto.

2.4.1. Coste de recursos humanos

La mayor parte del coste vendrá generado por las horas de trabajo dedicadas al proyecto por el alumno.

Según la Universidad Europea, el salario medio de un informático recién egresado ronda los 18000 - 22000€ al año [23]. Si contamos el número de días hábiles para este año 2022, obtenemos 253 días [21], a los que hay que restarle 22 días de vacaciones [25]. En total trabajaría 231 días al año ocho horas por día, lo que hace un total de 1848 horas. Si tomamos como salario medio los 20000 euros anuales y los dividimos por el número de horas que se trabajan al año, obtenemos un coste por hora de 10.82€.

2.4.2. Otros costes

A la cantidad total hay que añadirle los impuestos y tasas correspondientes, en nuestro caso el IVA (21 %). En mi caso, por ser mi primer año como autónomo, el tipo de IRPF que me corresponde es del 7 %.

2.4.3. Presupuesto final

El presupuesto queda desglosado en la tabla inferior, siendo el importe total a percibir 3700.44€.

Artículo	Coste (€)	Cantidad	Importe
Hora de trabajo autónomo	10.82€	300	3246€
Base imponible:			3246€
IVA 21 %			681.66€
Ret. 7 % IRPF			227.22€
Total:			3700.44€

Capítulo 3

Estado del arte

Para poder organizar este proyecto debemos conocer el estado del arte en su ámbito. Hablaremos del estado de arte en cheminformatics, en Deep Learning y sobre las herramientas que se han creado en estas áreas del conocimiento y que nos van a facilitar desarrollar este TFG.

3.1. Cheminformatics

Como comentamos en el capítulo 1 (Introducción), cheminformatics es un tema muy amplio y engloba subtópicos diferentes. Muchos de ellos surgieron en la década de 1960 y principios de 1970, y desde esa época muchos grupos de investigación siguen trabajando en ellos y nuevos grupos han surgido para aplicar nuevas tecnologías en nuevos ámbitos. A continuación discutimos algunos de los temas que históricamente han sido de interés para esta ciencia. [5]

3.1.1. Compuestos orgánicos y su representación

Un compuesto orgánico es un compuesto químico que contiene átomos de carbono, formando enlaces carbono-carbono y carbono-hidrógeno [38]. En el conjunto de datos sobre el que trabajamos, los ejemplos positivos son concretamente compuestos químicos organometálicos, en los que átomos de carbono forman enlaces covalentes con átomos metálicos [39].

En las publicaciones de química encontramos numerosas representaciones gráficas de estos compuestos, lo que se conocen como fórmulas estructurales. Éstas muestran la disposición en el espacio de los átomos que forman el compuesto. Un ejemplo es la siguiente figura:

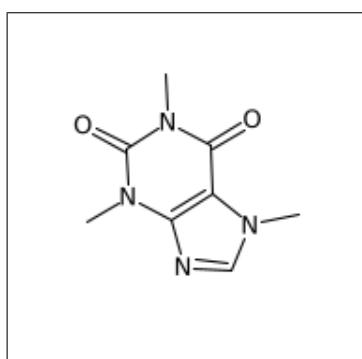


Figura 3.1: Estructura de la cafeína

En ellas pueden aparecer multitud de elementos, apareciendo siempre:

- **Átomos:** Se sitúan en los extremos de los enlaces. Representados con letras que indican el elemento químico del que se trata, tal y como aparece en la tabla periódica.
- **Enlaces:** Unen dos átomos entre sí.

En algunas ocasiones, sobre los átomos pueden mostrarse cargas positivas o negativas representando iones. Aparte, puede mostrarse lo que se conoce como información estereoquímica, que indica la disposición de los átomos en el espacio. Ésta es importante ya que afecta a las propiedades y reactividad de las moléculas [40, 36].

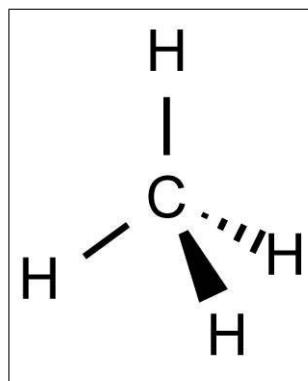


Figura 3.2: Estructura del metano

- Las líneas sólidas representan enlaces en el plano.
- Las discontinuas representan enlaces que están más alejados.
- Aquellas con forma de cuña indican que uno de los átomos se encuentra más cerca del espectador.

Además, un mismo compuesto se puede representar de distintas formas:

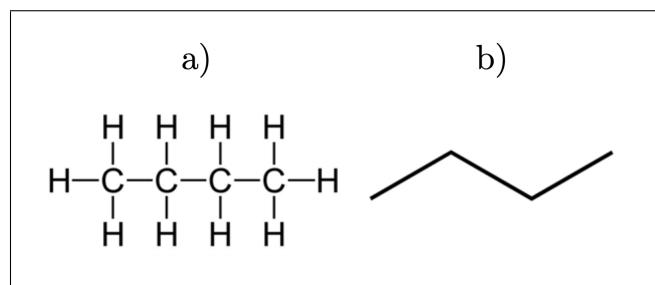


Figura 3.3: Dos formas de representar el butano

En a) se detalla la definición de todos los átomos, en cambio en b) encontramos lo que se conoce como fórmula de esqueleto, donde se omiten los átomos de carbono e hidrógeno. Se sabe que hay un átomo de carbono en los vértices que quedan libres en la intersección de dos enlaces o en las terminaciones donde no aparece ningún otro elemento. Se supone a la vez que cada átomo de carbono tiene cuatro enlaces, por tanto el número de enlaces que faltan por indicar explícitamente se corresponden con enlaces a moléculas de hidrógeno [41, 36].

3.1.2. Representación en el ordenador

El poder almacenar representaciones de compuestos químicos de manera eficiente en un ordenador requiere de la creación de métodos y formatos específicos para ello. Además, hay que tener en cuenta qué datos vamos a codificar, si solo la estructura básica del compuesto, si también queremos guardar información estereoquímica o si queremos añadir notas auxiliares sobre los compuestos. Es importante tener esto claro, ya que la complejidad de la representación influirá en la cantidad de almacenamiento que ocupe en disco y en los recursos necesarios para procesarla.

Utilizando notación lineal, se representa la estructura del compuesto como una secuencia lineal de caracteres y números. Esta es una codificación adecuada para las computadoras, ya que la pueden procesar con facilidad. Algunos formatos que utilizan esta notación son WLN (Wiswesser Line Notation), ROSDAL (Rp) o SMILES (Simplified Molecular Input Line Entry Specification). Aunque WLN y ROSDAL han quedado obsoletos, SMILES se sigue utilizando con mucha frecuencia en la actualidad. [5]

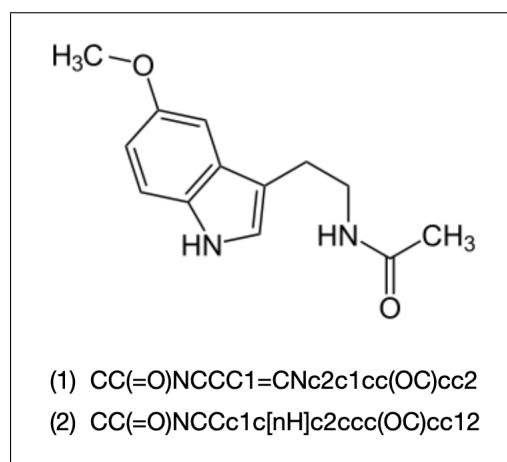


Figura 3.4: Dos posibles codificaciones de la melatonina en SMILES [42]

Aunque no vamos a entrar en detalle de cómo funciona ya que para ello hay que tener nociones avanzadas de química, diremos que utiliza las siglas de cada elemento de la tabla periódica para representar los átomos. La primera letra del elemento se escribe en mayúscula, a no ser que se trate de un átomo perteneciente a un anillo aromático¹, ya que en ese caso se escribe en minúscula. Si el elemento tiene dos caracteres, el segundo se escribe siempre en minúscula. Además, se pueden representar cargas.

Los enlaces se representan con -, =, # y :, según el tipo. Bajo algunas circunstancias, se pueden omitir estos símbolos, ya que por su contexto se deducen. También se pueden codificar ramas, situando elementos entre paréntesis, y ciclos, utilizando un número para indicar el inicio y el fin del ciclo en la cadena de texto. [1]

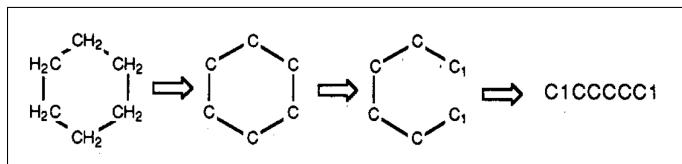


Figura 3.5: Ciclos en SMILES

Otras características como la aromaticidad o estructuras inconectas también pueden ser representadas. Entre las ventajas de esta codificación, destaca su facilidad de comprensión por los humanos. Cualquier químico puede aprender sus reglas de codificación fácilmente y diseñar sus propios com-

¹La aromaticidad es una propiedad presente en enlaces dobles de moléculas cíclicas, donde sus electrones pueden circular libremente. Esto mejora la estabilidad del compuesto. [20, 37]

puestos. Un problema que tiene SMILES es que un mismo elemento se puede representar de diferentes formas. Pero sobre todo, el mayor problema es que un porcentaje significativo de las cadenas no se corresponden con moléculas válidas, ya sea porque son sintácticamente inválidas, no se corresponden con un grafo molecular o no cumplen reglas químicas básicas. [1]

Uno de los principales objetivos de la química computacional es el diseño de nuevas moléculas. Para ello, la utilización de modelos generativos puede ayudar a los investigadores, pero si el espacio de estados de SMILES no es completamente válido se dificulta la tarea. Para ello han surgido otras codificaciones como SELFIES con un espacio 100 % robusto. [15]

Además de estos formatos de notación lineal, es necesario mencionar otros. El lanzamiento en 1982 de MDL MOLfile llevó a su aceptación como principal formato para codificar compuestos químicos. Se han realizado distintas adaptaciones de este para añadir información extra a las moléculas, dando lugar a SDfile (puede agrupar más de un MOLfile y almacenar información estructural), RXNfile (anota los reactivos y productos de una única reacción química), etc [2]. El formato PDB se utiliza principalmente para almacenar información 3D de macromoléculas biológicas, como son las proteínas o los polinucleótidos. CIF también es un formato para almacenar información 3D. En espectroscopia encontramos JCAMP. Finalmente, CML (chemical markup language), una extensión de XML, es una propuesta que intenta aglutinar toda la información disponible. Es compatible con moléculas, reacciones, espectroscopia y otra información. [5]

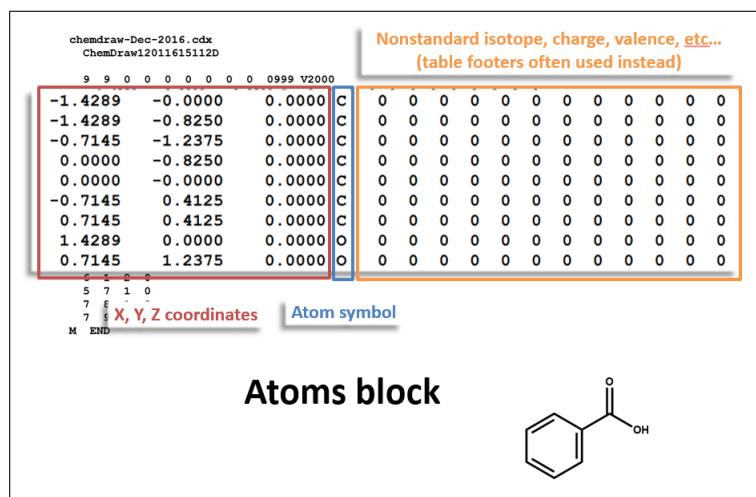


Figura 3.6: Contenido de un archivo MOL [19]

3.1.3. Fuentes y bases de datos

El gran número de facetas que presenta la información química necesita sistemas de almacenamiento a la altura. La química fue una de las primeras ramas científicas en utilizar bases de datos para almacenar: la cantidad de datos que se generaba creció rápidamente, y sigue creciendo hoy en día.

Aunque es complicado clasificarlas, vamos a separarlas en tres grandes grupos según el tipo de información que almacenan: [5]

- **Bases de datos de publicaciones:** Pueden ser bibliográficas, guardando solamente los metadatos y la referencia a la publicación, o de texto completo, donde recogen la publicación de forma íntegra.
- **Bases de datos fácticas:** Al contrario que las anteriores, que almacenan publicaciones de la literatura primaria, estas pueden guardar propiedades físicas de compuestos, información de espectroscopia, información legal, etc.
- **Bases de datos de estructuras y reacciones:** Recogen estructuras químicas, tanto individualmente como formando parte de reacciones. No se almacenan como imágenes, sino en formatos interpretables por la máquina.
- **Bases de datos de biología molecular:** Contienen secuencias de aminoácidos y nucleótidos.

Pero, ¿cómo podemos llenarlas con datos? ¿De dónde podemos extraerlos? Durante décadas se han publicado un gran número de artículos. Se podrían utilizar como una fuente muy amplia de información, ya que contienen todo el progreso científico. Específicamente para extraer datos relativos a estructuras entran en juego utilidades conocidas como OCSR (Optical Chemical Structure Recognition).

Son capaces de transformar una imagen en un formato compatible con la máquina, como podría ser SMILES. En muchos casos incluso es posible introducirles una publicación completa y ellas mismas localizan las imágenes de moléculas. Con el paso del tiempo se han ido perfeccionando, y algunas son capaces de detectar información estereoquímica o de relacionar el compuesto con el texto de la publicación. ChemEx [7], ChemGrapher [16] y DECIMER [17] son ejemplos de sistemas OCSR publicados en la última década.

Por último, mencionar una base de datos que merece la pena conocer. Creada por el National Institute of Health (NIH), PubChem es una base de datos abierta que cada mes sirve a millones de usuarios en todo el mundo. Es una base de datos de estructuras y aunque contiene mayoritariamente moléculas pequeñas, también almacena nucleótidos, carbohidratos, lípidos, péptidos y macromoléculas modificadas químicamente. Para cada compuesto almacena su estructura, identificadores, propiedades físicas y químicas,

toxicidad, patentes, etc. Los datos que aglutina provienen de diversas fuentes, como son agencias del gobierno estadounidense, editores de revistas científicas o proveedores químicos, aunque hay muchas más. [26]

3.1.4. Métodos de búsqueda

Almacenar información en las bases de datos no sirve de nada si no se desarrollan métodos eficientes para extraerla. En aquellas bases de datos donde se almacenan estructuras químicas, una de las principales formas de obtener información es buscar similitudes entre una molécula dada como entrada y otras que se encuentran almacenadas, de forma que comparten una subestructura específica o tengan otras características en común. Para ello, es clave la codificación de los compuestos.

También, si se almacenan metadatos y la base de datos está indexada sobre ellos se podría buscar por su nombre, etiquetas, etc. [5]

3.1.5. Métodos para análisis de datos

En química, grandes cantidades de datos son producidas. Una vez que hemos conseguido limpiarlos y ordenarlos, tenemos un conocimiento muy valioso en nuestras manos. La información es muy interesante en sí misma, pero también lo son las relaciones que se esconden en su interior. Para ello, se crean modelos que puedan interiorizarlas.

El análisis de datos no solo se enfrenta a la extracción de la información principal, sino que también intenta generar nueva información secundaria. [5]

Este TFG se desarrolla dentro de este ámbito, ya que, como describiré más adelante, entreno un modelo capaz de detectar que imágenes contienen moléculas.

3.2. Deep learning

En las últimas décadas, la Inteligencia Artificial (I.A.) ha vivido un periodo de crecimiento brutal. Dentro del ámbito de la I.A. se engloban técnicas muy diferentes que tienen como objetivo simular el comportamiento intelectual de los seres vivos, siendo uno de sus rasgos la capacidad de aprendizaje. Los sistemas expertos fueron su primer éxito comercial. En ellos se codificaban una serie de reglas manualmente, emulando el razonamiento de un experto en un problema específico.

El aprendizaje automático es un subconjunto dentro de la Inteligencia Artificial. Aglutina aquellas técnicas que permiten a un ordenador construir modelos a partir de conjuntos de datos, aprendiendo con la experiencia. No hace falta definir reglas de forma explícita. Pero tiene un inconveniente, y es que en muchos casos un especialista humano tiene que elegir manualmente qué características de los datos son relevantes en el problema.

El aprendizaje profundo o deep learning va un paso más allá. Es un subconjunto del aprendizaje automático, pero en las técnicas que engloba no es necesario indicar al modelo las características útiles, ya que él mismo las descubre automáticamente. Y no se queda ahí, va un aún más allá, puede generar nuevas características a partir de las existentes.

El desarrollo temporal de estas técnicas coincide con el orden en el que las menciono. Como se puede apreciar, el proceso de aprendizaje consta de diferentes fases, de las que cada vez se encuentra automatizado un mayor número. [13]

Casi todos los algoritmos de deep learning son una adaptación particular de un proceso que se puede resumir en los siguientes cuatro pasos:

- Extraer un conjunto de datos asociado al problema (un conjunto de gran tamaño, cuanto más grande mejor).
- Diseñar una función de coste apropiada (loss function).
- Crear un modelo y definir sus hiperparámetros (tamaño, tasa de aprendizaje...), asignándole los valores más adecuados.
- Aplicar un algoritmo de optimización para minimizar la función de coste.

Siendo esta la estrategia que se utiliza en otras muchas técnicas de aprendizaje automático. La diferencia entre el deep learning y estas otras es la capacidad de abstracción que tiene el primero, que viene dada por la capacidad de jerarquizar la información en distintas capas utilizando múltiples niveles de representación. Esta abstracción permite separar la esencia de lo prescindible, de forma que los modelos pueden aprender qué relaciones entre los datos son importantes y cuáles son simples accidentes, y consecuentemente generalizar de forma adecuada. [13]

3.2.1. Redes neuronales multicapa

Nuestro cerebro contiene un tipo especial de células llamadas neuronas. Estas reciben impulsos eléctricos que son capaces de atenuar o amplificar para posteriormente enviarlos a otras neuronas. La unión de millones de estas crea una estructura neuronal que es la que nos permite pensar y actuar como seres inteligentes.

En la informática se ha intentado emular este comportamiento, creando lo que se conoce como redes neuronales artificiales. Las neuronas artificiales omiten muchas características que definen a las biológicas, como su localización espacial, los retardos en la propagación de señales o los mecanismos químicos de sodio y potasio que permiten la transmisión de potenciales eléctricos. Al final, un modelo es una representación de la realidad, y por tanto contiene información más limitada que esta. El modelo de integración y disparo define la base de la neurona que se utiliza habitualmente en Inteligencia Artificial:

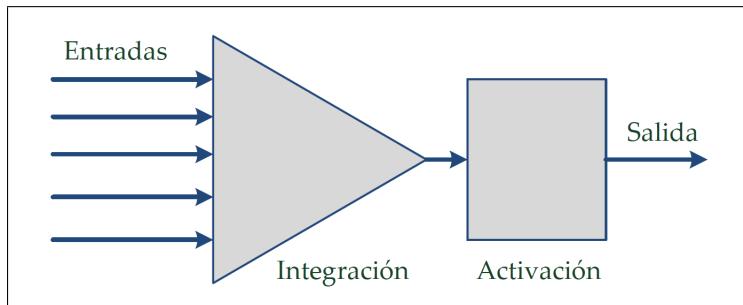


Figura 3.7: Modelo de integración y disparo en las neuronas artificiales
[13]

En la fase de integración se unifican todas las señales x_i entrantes a la neurona j , cada una influyendo en mayor o menor medida según el peso w_{ij} , peso de la conexión de la neurona i hasta la neurona j :

$$z_j = b_j + \sum_{i=1}^n x_i w_{ij} \quad (3.1)$$

Además, en muchas ocasiones se añade la componente b_j que actúa como sesgo, permitiendo que la salida de la neurona se desplace a la izquierda o a la derecha en el eje X. Este sesgo también se puede representar de forma implícita, de forma que i parte de 0 en vez de 1, siendo $x_0 = 1$ y $w_{0j} = b_j$. [13]

Pero esta fase de integración no es suficiente para simular el comportamiento del cerebro humano, su resultado es lineal. Una red neuronal se

genera encadenando capas de estas neuronas, donde la salida de las neuronas de una capa actúan de entrada a las neuronas de la siguiente capa. La combinación de funciones lineales es una función lineal, y por tanto el resultado final de la red seguiría siendo una función lineal de las entradas. En definitiva, tener n capas equivaldría a utilizar una única capa.

Es por ello que existe una segunda fase en este proceso, la fase de activación. Al resultado de integración en cada neurona se le aplica una función de activación no lineal (σ). A partir de ahora, nuestra red podrá aproximar funciones no lineales: cuantas más capas se utilicen en una red, con mayor precisión se podrá llevar a cabo esta aproximación.

$$f_j = \sigma(z_j) = \sigma(b_j + \sum_{i=1}^n x_i w_{ij}) \quad (3.2)$$

f_j es la salida final de la neurona. [13]

3.2.2. Redes neuronales convolutivas

Si hay una técnica del aprendizaje profundo que funcione especialmente bien en la práctica, son las redes neuronales convolutivas o convolutional neural networks (CNN). Estas se utilizan para procesar señales, como son las imágenes.

Una ventaja frente a las redes multicapa reside en que pueden trabajar con entradas y salidas estructuradas. Esto supone que, en vez de recibir un vector de entradas correspondientes a diferentes variables, pueden trabajar directamente con un vector 1D, una matriz 2D o con un tensor con múltiples dimensiones. [13]

En estas redes, las capas realizan la operación de convolución en vez de la multiplicación de las entradas por pesos. En el caso de imágenes (señal 2D), esta operación se define como:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \quad (3.3)$$

Donde I representa una imagen y K un kernel de dos dimensiones [10]. El tipo de convolución que describimos aquí es la discreta y por ello en la fórmula aparecen sumatorias, en vez de las integrales típicas de dominios continuos. En la siguiente figura se puede observar un ejemplo de convolución:

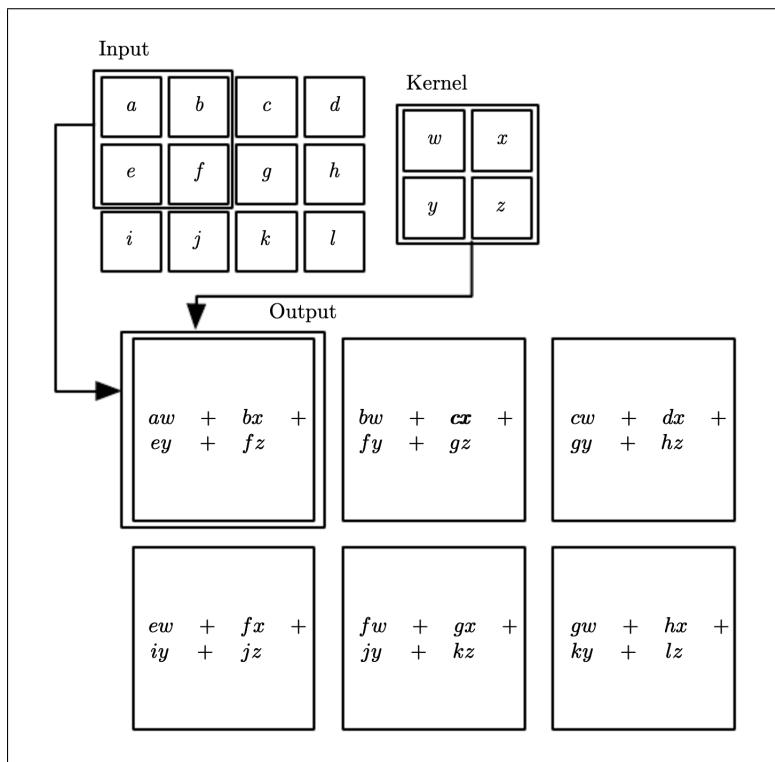


Figura 3.8: Resultado de convolucionar la entrada I con el kernel K [10]

El kernel se va desplazando por la imagen, solapándose sobre cierta área de esta. En cada una de las posiciones que toma genera un valor de salida, fruto de la multiplicación del valor de los píxeles de la imagen sobre los que se encuentra por los pesos del kernel. Todos estos valores de salida forman el resultado final, que mantiene la misma estructura 2D (aunque normalmente con un menor tamaño).

Así es como funcionan las redes neuronales convolutivas, donde cada capa de la red aplica este operador a la entrada que recibe. Con ellas:

- Se reduce el número de parámetros necesarios. Este depende del tamaño de los kernel utilizados, que suele ser muy inferior al de las imágenes de entrada.
- Se mantienen las relaciones espaciales/temporales de las señales. Muy beneficioso en análisis de imágenes, donde dos píxeles cercanos contendrán información similar. Los capas convolutivas son capaces de interpretar estas relaciones y mantenerlas para las siguientes fases del modelo. [10]

3.2.3. Autocodificadores

Los autocodificadores o autoencoders son un modelo de aprendizaje no supervisado que permite encontrar una representación latente no lineal para una distribución de datos dada.

Para comprenderlos correctamente, definamos primero lo que es una representación latente. Esta es una representación subyacente para una determinada distribución de datos, es decir, una forma “comprimida” de representar la distribución que siguen estos. En esta representación, se reduce el número de dimensiones necesarias para definir la distribución.

Existe un algoritmo conocido como análisis de componentes principales (*Principal Component Analysis*, PCA) que encuentra una transformación de la representación original a otra latente. Pero tiene un problema, y es que como su transformación es lineal. [33]

Los autocodificadores realizan esta tarea pero encontrando una transformación no lineal, por lo que pueden trabajar con datos más complejos. Para ello constan de dos partes, el codificador y el decodificador:

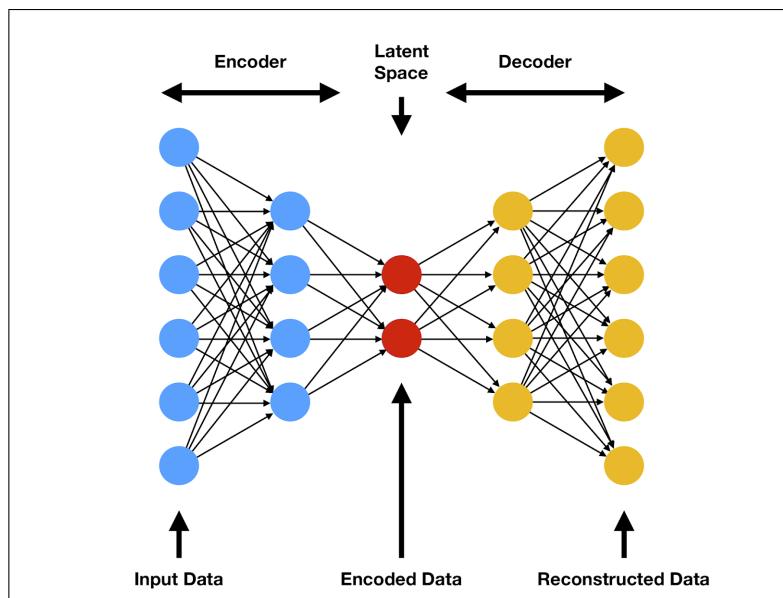


Figura 3.9: Secciones en las que se divide un autocodificador [33]

El objetivo tras el entrenamiento del modelo es que los datos reconstruidos por el decodificador sean lo más parecidos posible a los de entrada.

Un tipo especial de autocodificador es el autocodificador variacional (Variational Autoencoder, VAE). La principal diferencia viene dada por la estructura que sigue la representación latente, donde se fuerza al autocodifica-

dor a organizarla de forma que puntos de datos similares se encuentren cerca y puntos diferentes se encuentren separados. En un autocodificador regular esto no se exige, por lo que no podríamos desplazarnos por el espacio latente y observar cambios graduales que se corresponden con cambios graduales en la representación original. [33]

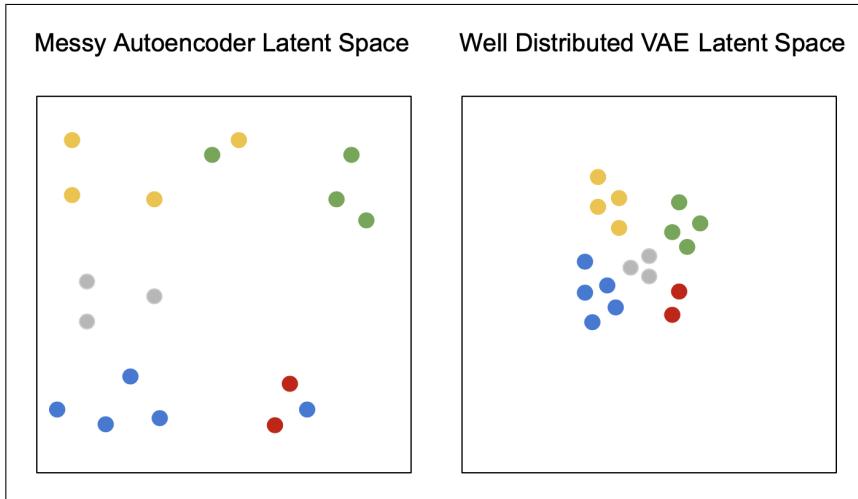


Figura 3.10: Espacio latente en un autocodificador regular vs en VAE [33]

Vector Quantized Variational Autoencoder

La representación latente por defecto en los VAE es continua. Pero existe una variante en la que se discretiza esta representación. Esto es útil cuando trabajamos con datos como imágenes: podría utilizarse una variable discreta para el color, otra para el tipo de objeto, su tamaño, etc. Muchos otros elementos del mundo real son susceptibles de representarse de forma discreta. Hay modelos como los transformers, que vamos a estudiar más adelante, que están diseñados para trabajar sobre datos con esta codificación. [33]

Los Vector Quantized Variational Autoencoders (VQVAE) utilizan este tipo de codificación, donde la representación latente consiste en lo que se conoce como libro de código (codebook). Este codebook no deja de ser un conjunto de vectores discretos asociados a un índice.

Cuando el codificador recibe un dato, este se transforma a la representación latente. A continuación, se introduce al decodificador el vector del codebook con el que existe una menor distancia euclídea. Este vector es procesado por el decodificador devolviendo la salida final. Dicho esto, se podría pensar que la potencia expresiva del autoencoder es muy baja, ya que el número de vectores del codebook es mucho más pequeño que el conjunto de posibles datos de entrada, y siempre que se introdujese el mismo vector al

decodificador obtendríamos la misma salida.

Esto no es así, ya que normalmente no se utiliza un único vector: en datos como imágenes, es probable que se escoja un conjunto de vectores que se introducen en el decodificador. Estos son procesados conjuntamente y finalmente se devuelve la imagen sintética. De esta forma, el número de posibilidades crece enormemente. [33]

3.2.4. Redes generativas antagónicas

En inglés conocidas como Generative Adversarial Networks (GANs), son un tipo de modelos utilizados para generar datos sintéticos, que por ejemplo pueden servir para ampliar nuestro dataset. Contienen dos módulos que trabajan simultáneamente, cada uno de ellos intentando ser mejor que el otro. Estos dos módulos son el generador G y el discriminador D: el generador intenta crear datos sintéticos parecidos a los reales y el discriminador actuará de juez, tendrá que decidir si dado un dato este es real o sintético. Estamos en un juego en el que el generador intentará mejorar todo lo posible para generar imágenes indistinguibles a ojos del discriminador, desarrollándose en dos escenarios: [13]

- Se muestran datos x del conjunto de entrenamiento y se introducen en D. En este caso, $D(x)$ será un valor cercano a 1, ya que las muestras son datos reales.
- Se introduce en G una muestra z generada aleatoriamente de acuerdo a una distribución de probabilidad, siendo $G(z)$ una muestra sintética. $D(G(z))$ deberá tomar un valor cercano a 0, detectando que no se trata de una muestra real. Sin embargo, aunque el discriminador actúa para que $D(G(z))$ sea cercano a 0, el generador hace todo lo contrario y pretende que $D(G(z))$ se aproxime a 1.

El entrenamiento se lleva a cabo simultáneamente en G y D, actualizando los pesos de ambos en cada iteración. Este termina cuando se ha alcanzado un equilibrio en el que ni al generador ni al discriminador les interesa modificar sus pesos, llamado equilibrio de Nash.

Las GANs son muy populares actualmente y se pueden utilizar con diferentes tipos de datos, entre ellos las imágenes. Originalmente las GANs se presentaron utilizando capas completamente conectadas (fully connected layers), pero la técnica de las redes neuronales convolutivas también se aplicó a las GANs dando lugar a las Deep Convolutional GAN (DCGAN). [13]

3.2.5. Transformers

En los últimos años, se han desarrollado diferentes estrategias que permiten el modelado de secuencias. Algunas de ellas son las Redes Neuronales Recurrentes (RNN), los LSTM o modelos basados en redes convolutivas, como Wavenet. Aunque han dado buenos resultados en campos como la generación de texto, son mejorables, ya que las técnicas de memoria que implementan solo les permiten recordar los últimos datos de la secuencia. Así, en sucesiones de mayor longitud, no podrán mantener en memoria datos alejados del punto actual.

Para resolver este problema, los investigadores han creado lo que se conoce como mecanismos de atención. En ellos, el modelo es capaz de comprender qué datos de la secuencia son importantes para la tarea que estamos desarrollando. Estos datos son los que memoriza, descartando el resto. Funcionan de forma similar a nuestro cerebro: cuando vemos una imagen, aunque parece que estamos contemplando todos sus detalles, solo estamos prestando atención a fragmentos concretos de esta y obviando los detalles en el resto de áreas. Algunos autores aplicaron esta técnica a arquitecturas ya existentes.

Pero en 2017, Vaswani et al. [12] propone un modelo basado únicamente en mecanismos de atención, prescindiendo de técnicas como la recurrencia o las convoluciones: a esta arquitectura la llamó el transformer. El modelo está formado por 6 codificadores y 6 decodificadores:

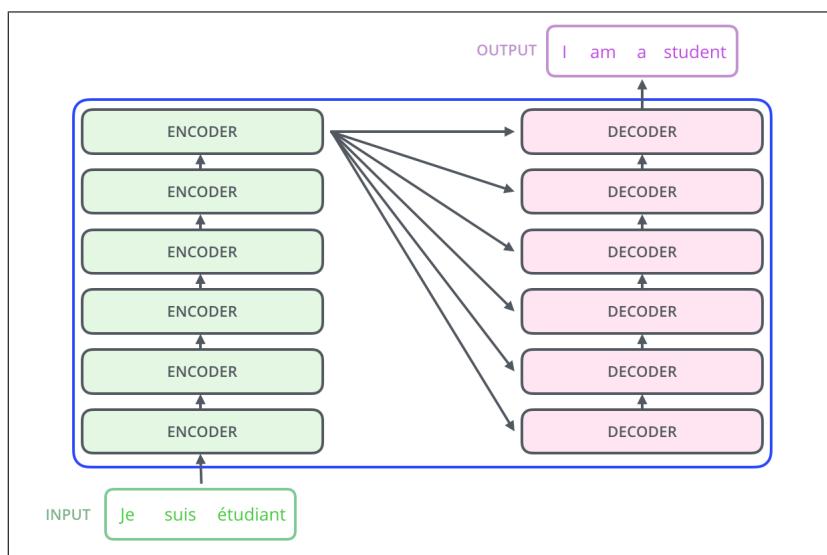
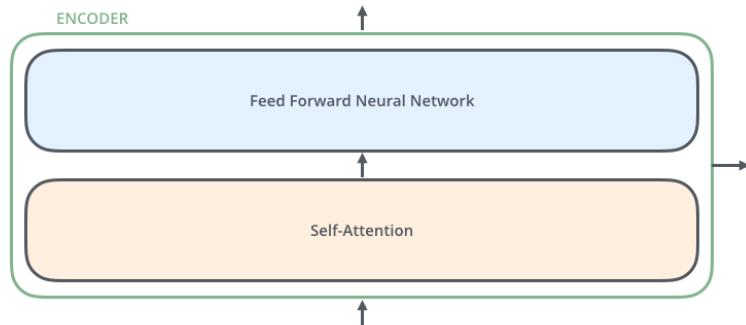


Figura 3.11: Transformer aplicado a la generación de texto [24]

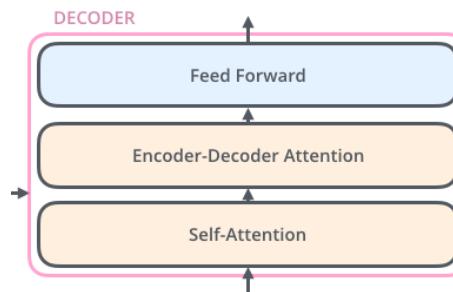
En cada codificador, las características de entrada primero pasan por una capa de auto-atención, en la que interactúan entre sí y se descubre

a cuáles hay que prestar mayor atención. La salida de esta primera capa serán los datos agregados resultantes de esas interacciones y las puntuaciones de atención. A continuación, se introduce esta salida en una red neuronal completamente conectada. [28]

Los decodificadores también poseen estas dos capas, pero entre ambas existe otra capa de atención que ayuda al decodificador a concentrarse en las características importantes de la entrada.



(a) Estructura interior del codificador



(b) Estructura exterior del decodificador

Figura 3.12: Los dos tipos de capas que forman el Transformer [24]

Los transformer son actualmente los modelos más efectivos en tareas como la generación de lenguaje natural, y también se han aplicado a generación de imágenes. Estos son capaces de memorizar datos de cualquier punto de la secuencia, al contrario que otros modelos recurrentes o convolutivos, que sólo almacenan las interacciones locales. [24]

3.3. Bibliotecas y paquetes

Existen productos desarrollados por investigadores que nos van a facilitar cumplir nuestro objetivo.

3.3.1. Taming Transformers for High-Resolution Image Synthesis

Científicos de la Universidad de Heidelberg (Alemania) publicaron a finales de 2020 un modelo capaz de generar imágenes sintéticas de alta resolución utilizando transformers. En publicaciones previas ya se habían expuesto modelos generativos basados en transformers [14], pero estos solo podían funcionar con imágenes de pequeño tamaño. El coste computacional de un transformer aplicado directamente en imágenes crece cuadráticamente con el tamaño de esta, así que no es viable utilizarlos en imágenes de más de 100x100 píxeles, y mucho menos en aquellas del orden del megapixel.

P. Esser et al. [18] desarrolla un método capaz de combinar la potencia generativa de los transformers con la eficiencia que aportan las redes convolutivas.

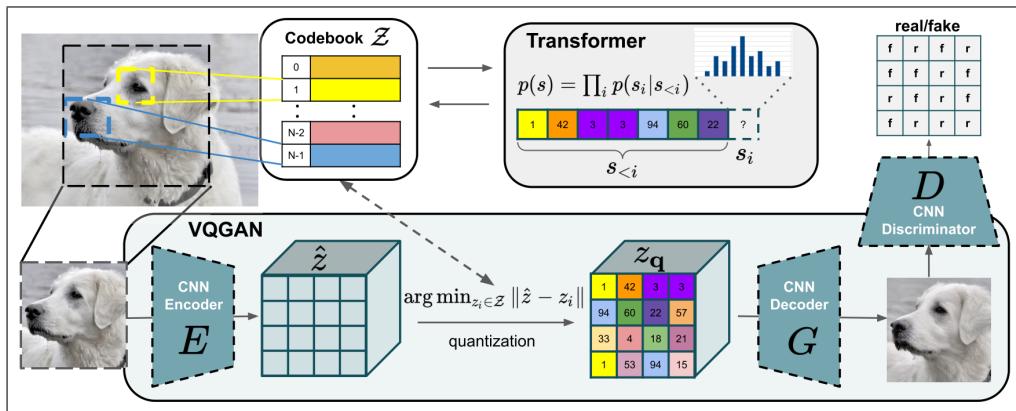


Figura 3.13: Modelo propuesto por P. Esser et al. [18]

La arquitectura convolutiva de la que hablamos es un Vector Quantized Generative Adversarial Network (VQGAN), una variante de VQVAE propuesta por los autores de la publicación donde la estructura codificador-decodificador del VAE se entrena mediante un procedimiento adversarial, utilizando un discriminador.

Tras entrenar el modelo, se habrá generado un codebook que comprime la información de las imágenes. Será sobre este codebook sobre el que se entrene el transformer que posteriormente nos permitirá generar imágenes

sintéticas. Ahora tendremos un modelo que es capaz de aprovechar la gran capacidad generativa de los transformers, manteniendo tiempos de entrenamiento razonables gracias a VQGAN.

En la publicación original los autores muestran ejemplos de los resultados que se pueden obtener dada una imagen condicional de entrada:

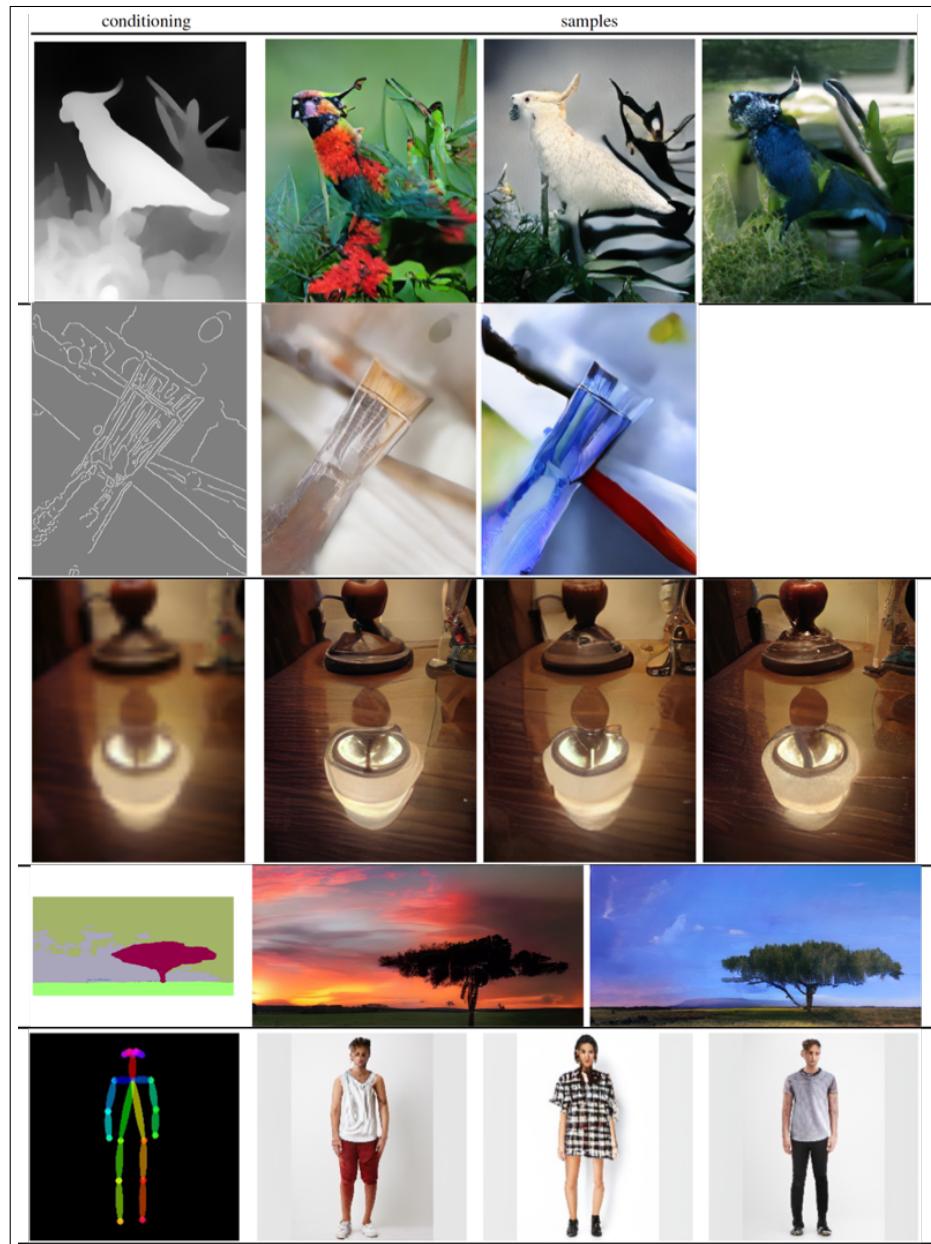


Figura 3.14: Ejemplos generados utilizando el modelo basado en transformer [18]

Se observa que funciona correctamente en diversas situaciones, por lo que puede ser una buena opción para nuestra la de síntesis de imágenes.

3.3.2. Pytorch

PyTorch es una biblioteca basada en tensores optimizada para crear modelos de aprendizaje profundo que se entrenen y ejecuten tanto en GPUs como en CPUs.



Figura 3.15: Logo de PyTorch

Es adecuada para ser utilizada en proyectos de investigación, ya que convierte el desarrollar y experimentar con nuevas arquitecturas en algo relativamente sencillo. Su núcleo proporciona la capacidad de definir funciones matemáticas y calcular sus gradientes.

Al compararla con otras bibliotecas de aprendizaje profundo como Tensorflow encontramos diferencias. Esta sigue el paradigma definición-compilación-ejecución, mientras que PyTorch presenta una estructura definición-ejecución, donde no existe una etapa de compilación. El usuario puede definir una expresión matemática y calcular su gradiente directamente. [11]

A continuación, voy a comentar algunas características y módulos incluidos en esta biblioteca que merece la pena conocer.

Dataset y DataLoader

El código para gestionar los datos con los que se trabaja se puede volver complejo y difícil de mantener. PyTorch proporciona los módulos `torch.utils.data.DataLoader` y `torch.utils.data.Dataset` para automatizar la carga y la manipulación de los datos, tanto procedentes de `datasets` ya existentes (por ejemplo, MNIST) como de datos propios.

`torch.utils.data.Dataset` es una clase heredada por una clase hija en la que se implementan ciertos métodos encargados de cargar los datos. Un da-

taset puede ser de tipo *iterable* o de tipo *map*: los primeros implementan *_iter_()*, y permiten que el objeto sea iterable, de forma que *iter(dataset)* devuelve un flujo de datos que están siendo leídos de una base de datos o incluso se están produciendo en tiempo real. Por otra parte, un *dataset* de tipo *map* implementa los métodos *_getitem_()* y *_len_()*. Esto permite establecer una relación clave-dato, donde si *CompoundDataset* es un *dataset* con imágenes que pueden ser o no compuestos químicos, *CompoundDataset[idx]* devolvería la imagen número *idx* junto a su etiqueta. Este último tipo de *dataset* es el que se utiliza en este proyecto.

El constructor de un objeto de tipo *torch.utils.data.DataLoader* recibe como principal argumento un objeto *torch.utils.data.Dataset*. También recibe otros parámetros, como el tamaño del batch o si queremos que los datos sean devueltos en un orden aleatorio. Devuelve un iterable que cumple estas características, lo que facilita en gran medida el entrenamiento de modelos.

Utilizando estos módulos separamos los datos del código de entrenamiento, obteniendo una mayor modularidad y un código más fácil de leer. [32]

Módulos *torch.nn* y *torch.optim*

El primero contiene los principales componentes utilizados en la construcción de redes neuronales y otras arquitecturas de aprendizaje profundo. Algunos de estos son:

- ***torch.nn.Module*:** Es la clase padre de la que heredan todas las implementaciones de redes neuronales. En su constructor, entre otros elementos, se pueden inicializar diferentes capas que serán utilizadas en el método *forward()*, donde se especifica la estructura de la red y las conexiones entre ellas.
- ***torch.nn.functional*:** Recoge funciones de activación, de error así como capas de la red que no tienen un estado asociado.
- ***torch.nn.Conv2d*, *torch.nn.MaxPool2d*, *torch.nn.Linear*** son ejemplos de capas contenidas en este módulo. Existen diferentes tipos de capas lineales, convolutivas, recurrentes, transformer, etc. previamente implementadas y listas para ser utilizadas.

Por otro lado, *torch.optim* contiene optimizadores como son SGD o Adam, encargados de actualizar los parámetros del modelo durante el entrenamiento. [32]

Autograd

Al entrenar redes neuronales, el método más utilizado para actualizar los parámetros del modelo es la propagación hacia atrás o *back propagation*.

Estos se ajustan de acuerdo al gradiente de la función de error con respecto al parámetro dado.

PyTorch facilita la aplicación de la propagación hacia atrás a través del módulo *torch.autograd*. Cuando implementamos un modelo, solo es necesario declarar el método *forward()*, ya que PyTorch crea sobre la marcha el grafo de las operaciones necesarias para calcular el gradiente. Durante el entrenamiento del modelo, al llamar a *loss.backward()* se calculará el gradiente utilizando este grafo. [32]

Como conclusión, PyTorch aporta una gran flexibilidad para crear modelos de aprendizaje profundo, su sintaxis y paradigma de programación son relativamente fáciles de aprender y contiene un gran número de módulos que permiten crear modelos complejos. Aparte, es compatible con el entrenamiento en GPU y permite guardar los modelos entrenados. Por todas estas razones puede resultar una librería adecuada para implementar el clasificador de imágenes.

Capítulo 4

Metodología de investigación

Para alcanzar el objetivo final es importante fijar unos pasos concretos a seguir. Por ello, en un primer lugar he estudiado el estado del arte, que nos permite conocer que herramientas se encuentran a nuestra disposición. A continuación tendré que revisar el *dataset* del que se parte, mejorando su balance y generando *hard-negatives* que permitirán entrenar el modelo clasificador de forma que funcione correctamente en casos más difíciles. Tras haber preparado el *dataset*, entrenaré el modelo clasificador sobre este.

En esta sección explico los conceptos más importantes que debemos entender si queremos comprender las decisiones que se han tomado y el funcionamiento de los modelos.

4.1. Balanceo del dataset

El *dataset*, entregado por los científicos de la Universidad de Negev, contiene ejemplos positivos y negativos. Todas las imágenes son diferentes, presentan elementos con distintos trazados de línea, tamaños, colores... Originalmente se encuentran en formato .png.

Los ejemplos positivos son imágenes de moléculas organometálicas extraídas de publicaciones. La mayoría son moléculas completas, aunque algunas parecen recortes de estructuras más grandes. En total tenemos 162 imágenes de este tipo.

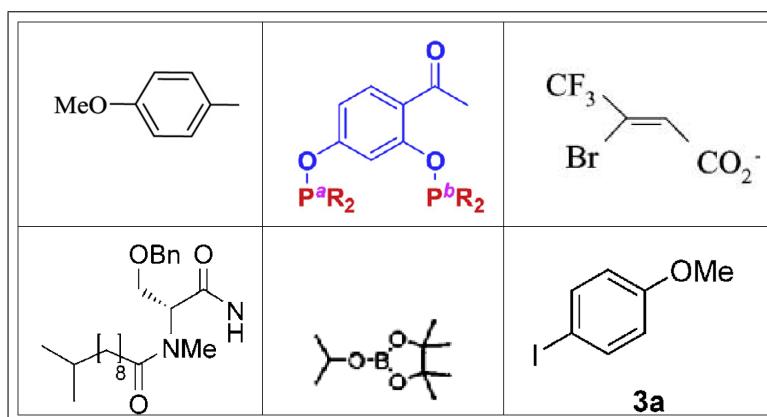


Figura 4.1: Ejemplos de muestras positivas del *dataset*

Los ejemplos negativos son, en cambio, imágenes que contienen rectas, curvas y otras figuras que se parecen a las formas que adquiere una molécula, pero no lo son. La diversidad de estas imágenes es muy alta, como se puede observar en la figura 4.2. En esta categoría hay más imágenes, 800 en total.

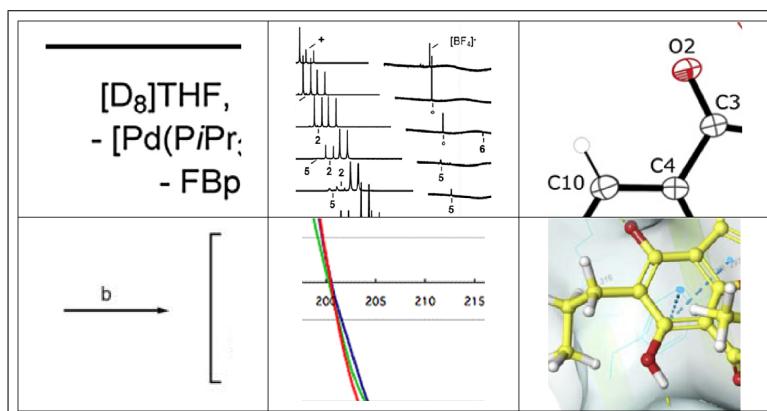


Figura 4.2: Ejemplos de muestras negativas del *dataset*

Estas imágenes necesitan un preprocesamiento, es necesario elegir el formato con el que se va a trabajar y dimensionarlas para que todas tengan el mismo tamaño. Debido a que el canal alfa del formato .png no se está aprovechando, decidimos convertirlas a formato .jpg y así trabajar con tres canales en nuestros modelos. Además, el modelo generativo que utilizamos está diseñado para trabajar con imágenes con este número de canales, así que será lo más adecuado.

Lo más conveniente sería dimensionar todas las imágenes al tamaño de la más grande, así no se perdería información. Pero esto puede suponer un problema, ya que existen algunas que superan los 700x700 píxeles, y cuanto mayor es su resolución más parámetros necesitan almacenar los modelos, dando lugar a altos tiempos de ejecución, y sobre todo, a problemas con la memoria de la GPU. Las GPUs del clúster no tienen capacidad para almacenar modelos con tantos parámetros, por lo que decido reducir el tamaño de las imágenes a 256x256.

También mencionar el limitado número de ejemplos positivos que existen en comparación con negativos: estamos ante un conjunto de datos desbalanceado. En el momento de entrenar el clasificador, tendremos que conseguir que esté más balanceado, ya sea reduciendo el número de ejemplos negativos utilizado o aplicando *data augmentation* sobre los positivos. En este trabajo elijo la segunda opción, ya que en aprendizaje automático cuantos más datos se utilizan en el entrenamiento, mejores modelos se obtienen.

La técnica de *data augmentation* consiste en aumentar el tamaño del conjunto de datos completándolo con imágenes alteradas de este. Existen bibliotecas en lenguajes de programación como Python que facilitan en gran medida esta tarea, ya que permiten indicar, entre una serie de transformaciones ya implementadas, cuáles queremos aplicar [27]. En nuestro caso vamos a crear tres *data augmentation* diferentes, y comprobaremos cuál funciona mejor en los experimentos. La primera realizará transformaciones suaves, la segunda algo más fuertes y la tercera bastante disruptivas.

Data augmentation 1. Transformaciones suaves:

- A_veces(50 %, DesenfoqueGaussiano(sigma=[0, 0.5]))
 - Escalar(x: [80 %, 100 %], y: [80 %, 100 %])
 - Rotar([-25º, +25º])
 - Estirar([-5,5])
-

A cada imagen del conjunto de datos se le aplicarán estas cuatro transformaciones, la primera solamente con un 50 % de probabilidad. Los parámetros de cada transformación vienen dados en un rango, de forma que se elige un valor aleatorio en este.

Data augmentation 2: Transformaciones más fuertes

- A_veces(50 %, DesenfoqueGaussiano(sigma=[0, 0.5]))
 - ContrasteLineal([0.75, 1.5])
 - Escalar(x: [70 %, 100 %], y: [70 %, 100 %])
 - Rotar([-45º, +45º])
 - Estirar([-10,10])
 - Trasladar(x: [-10 %, 10 %], y: [-10 %, 10 %])
 - Multiplicar([0.8, 1.2], por_canal=25 %)
 - RuidoGaussianoAditivo(loc=0, escala=[0.0, 0.05*255])
 - VoltarIzdaDcha(30 %)
-

En este caso, utilice transformaciones como Multiplicar (multiplica el valor de los píxeles de la imagen por un número entre 0.8 y 1.2), RuidoGaussianoAditivo (donde a cada píxel se le añade ruido generado mediante una distribución gaussiana centrada en 0 y con desviación entre 0 y 0.05*255) o VoltarIzqdaDcha, entre otras.

Data augmentation 3: Transformaciones fuertes

- A_veces(50 %, DesenfoqueGaussiano(sigma=[0, 0.6]))
 - ContrasteLineal([0.75, 2])
 - Escalar(x: [70 %, 100 %], y: [70 %, 100 %])
 - Rotar([-45º, +45º])
 - Estirar([-10,10])
 - Trasladar(x: [-20 %, 20 %], y: [-10 %, 10 %])
 - Multiplicar([0.6, 1.4], por_canal=25 %)
 - RuidoGaussianoAditivo(loc=0, escala=[0.0, 0.05*255], por_canal=30 %)
 - VoltarIzdaDcha(20 %)
 - VoltearArribaAbajo(20 %)
 - A_veces(70 %, TransformacionElastica(alfa=[0.75, 3], sigma=(0.2, 0.5)))
 - UnoEntre(Nitidez(alfa=[0, 1], brillo=[0.5, 1.5]), Repujar(alfa=[0, 1], fuerza=[0.75, 2]))
 - Dropout([0.01, 0.15], por_canal=0.5)
-

Finalmente, en esta versión de *data augmentation* añado transformaciones que alteran en gran medida el aspecto de las imágenes. TransformacionElastica desplaza píxeles de una zona de la imagen a otra cercana, donde alfa controla la distancia con la que se produce el desplazamiento y sigma la suavidad de este, un valor bajo de sigma dará lugar a imágenes ruidosas y pixeladas. Nitidez aplica este efecto a la imagen y mezcla el resultado con la imagen original, la intensidad de la mezcla se controla con el parámetro

alfa, mientras que el parámetro brillo controla el brillo de la imagen. Repujar (*Emboss* en inglés) da a la imagen un aspecto metálico, pronunciando las altas luces y sombras. Por último, Dropout da el valor 0 a cada pixel con una probabilidad entre 0.01 y 0.15.

Todas estas transformaciones se aplican en un orden aleatorio, lo que aporta una mayor diversidad. Como resumen, el siguiente gráfico recoge los diferentes conjuntos de datos que obtendremos tras este proceso de *data augmentation*:

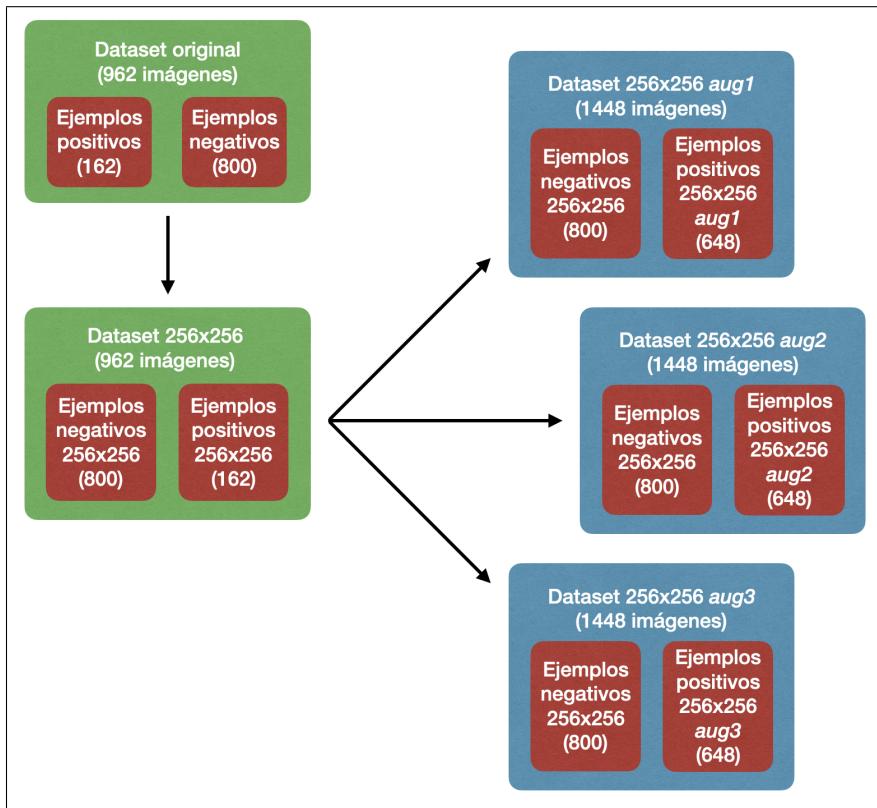


Figura 4.3: Diferentes *datasets* generados a partir del original

Las secuencias *data augmentation* se aplicarán en tres ocasiones sobre el conjunto de ejemplos positivos, por lo que cuadriplican su tamaño inicial: $162 + 3 \times 162 = 648$

Ahora ya tenemos un *dataset* razonablemente balanceado (800 ejemplos negativos vs 648 positivos), bueno, en concreto tres *datasets*. Tras realizar las pruebas experimentales decidiremos cuál es el más adecuado, de forma que las imágenes generadas mediante *data augmentation* aporten una diversidad razonable al conjunto pero sin excederse, para posteriormente utilizarlo para entrenar el clasificador.

4.2. Generación de *hard negatives*

Tras balancear los *datasets* con *data augmentation*, procedemos a generar *hard negatives*. Este tipo de ejemplos negativos son aquellos que se encuentran en el límite de lo que es negativo y positivo, aquellos que a los modelos de aprendizaje automático les cuesta clasificar. Incorporar este tipo de imágenes al conjunto de entrenamiento permite que el modelo sea más preciso en casos extremos y por tanto que funcione mejor.

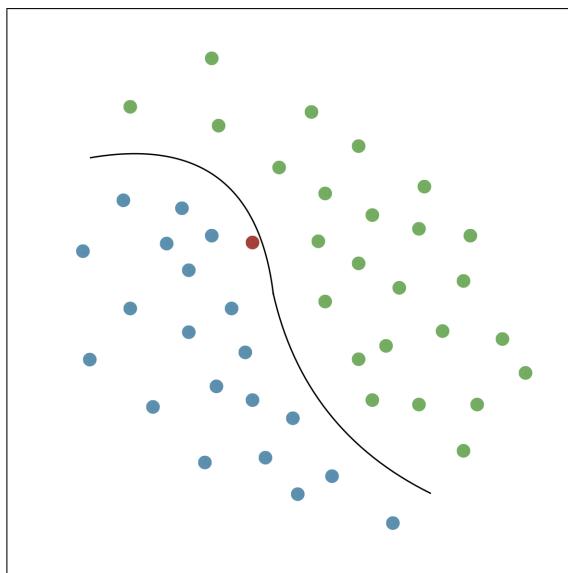


Figura 4.4: Límite de decisión separando dos clases. El individuo en rojo representa un *hard negative* perteneciente a la clase azul, ya que se encuentra muy cercano al límite de decisión.

La generación de estos ejemplos la voy a realizar utilizando el modelo desarrollado por Esser et al. [18], descrito en la revisión del estado del arte de este TFG 3.3.1. Para ello, entreno el modelo sobre los ejemplos positivos del conjunto de datos, de forma que sea capaz de generar imágenes que recuerden a moléculas sin que sean demasiado realistas.

Se entrena el modelo sobre los tres conjuntos de datos aumentados, aunque también se realizarán pruebas sobre el *dataset* original. El entrenamiento se realizará durante diferentes épocas, y se comprobará visualmente cuál es el número adecuado. En concreto, sobre cada conjunto de datos, se prevé probar con 70, 90, 110, 130, 150 y 170 épocas. Si fuera necesario realizar más experimentos con diferentes valores, se realizarían también. Discutiremos esto más a fondo en el capítulo siguiente.

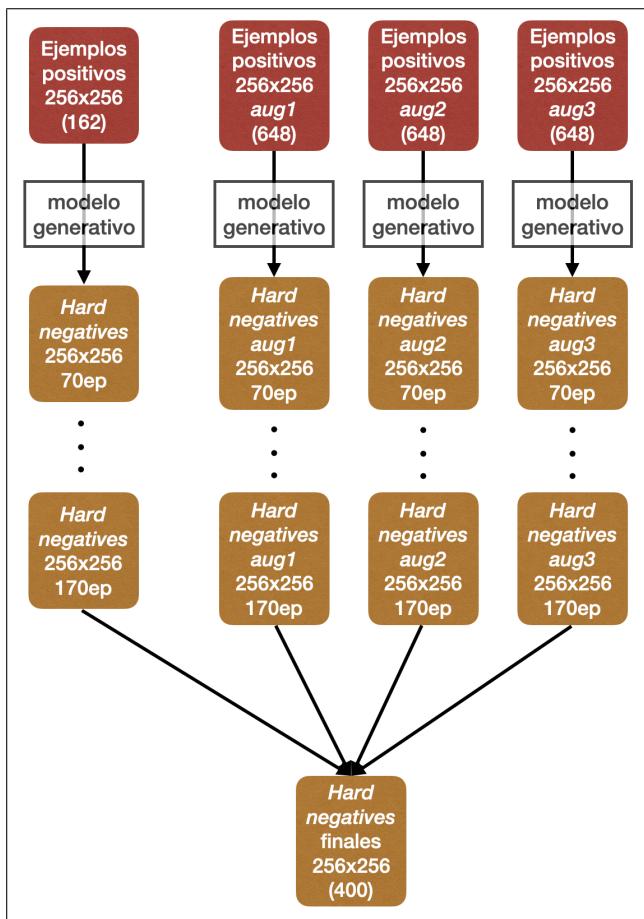


Figura 4.5: Diferentes *hard negatives* obtenidos entrenando el modelo con diferentes *data augmentation* y número de épocas.

Tras realizar los experimentos, se escogerá el modelo (o los modelos) que generen *hard negatives* de mayor calidad. Esta elección se realizará de forma visual. Con los modelos elegidos se creará un conjunto de *hard negatives* finales.

4.3. Clasificación de imágenes

El segundo objetivo de este proyecto consiste en construir un clasificador capaz de separar las imágenes que contienen estructuras de moléculas de aquellas que no. Hasta ahora hemos preparado el *dataset* con el que se va a entrenar este modelo y hemos generado *hard negatives* que pueden ayudar a crear un clasificador más robusto.

En concreto vamos a trabajar con dos *datasets*, no solo con uno: el que ha sido balanceado pero no contiene *hard negatives* y el que sí los contiene, 400 ejemplos negativos originales y 400 ejemplos negativos de los generados de forma sintética. Entregaremos a los científicos de la Universidad de Negev dos clasificadores, cada uno entrenado sobre uno de los dos conjuntos de datos. Para implementarlos utilizo la biblioteca Pytorch ya expuesta en el capítulo estado del arte 3.3.2.

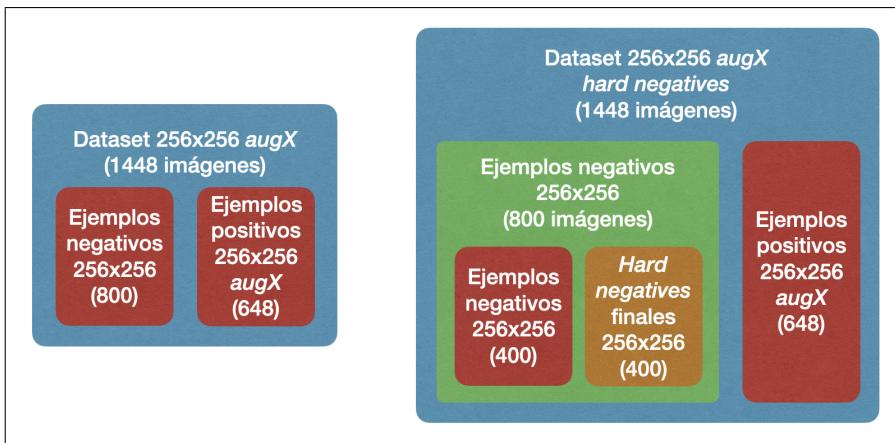


Figura 4.6: Dos *datasets* para entrenar dos clasificadores.

A priori no podemos conocer qué arquitectura funcionará mejor para este problema, por lo que implementaremos varias y compararemos su rendimiento. Voy a elegir una de tamaño pequeño, otra de tamaño mediano y otra de tamaño grande y comparar cuál es la más adecuada:

- **LeNet5:** Modelo propuesto por Yann LeCun en 1998 [4], fue una de las primeras redes convolutivas de la historia. Se diseñó con el propósito de reconocer imágenes de dígitos numéricos, y tras compararse su rendimiento con otros modelos en uso, se demostró que los sobrepassaba. Esto atrajo a muchos investigadores a esta incipiente línea de investigación y sentó las bases para las arquitecturas del futuro.
- **AlexNet:** Propuesto por Alex Krizhevsky en 2012 [6], consiguió mejorar por un alto porcentaje a la siguiente mejor solución en el concurso ImageNet Large Scale Visual Recognition Challenge. La ventaja de es-

te modelo frente a otros fue su profundidad, que le permitía reconocer imágenes más complejas con mayor precisión. Esto collevó un aumento del número de parámetros y del tiempo de entrenamiento, que fue contrarrestado por el uso de GPUs que permitieron la paralelización. Para combatir el sobreajuste se utilizó *dropout*, donde los enlaces entre neuronas se desconectaban con una determinada probabilidad durante cada iteración del entrenamiento.

- **VGG16:** Desarrollado por el Visual Geometry Group (de sus siglas procede el nombre del modelo) de la Universidad de Oxford [9], fue el ganador del concurso ImageNet en su edición de 2014. Una de las características que le permitió mejorar frente a otros modelos fue el uso de *kernels* convolutivos de pequeño tamaño (3x3). Se presentaron varias versiones del algoritmo, cada una con un número de capas diferente. La que nosotros utilizamos, VGG16, cuenta con 16 capas.

En un principio, cada uno de estos modelos se diseñó para trabajar con un tamaño de imágenes y de salida específicos, pero nosotros los adaptamos para funcionar sobre imágenes 256x256 y vectores *one-hot*. Estos vectores se utilizan en problemas de clasificación, de forma que cada coordenada del vector representa una de las clases (en nuestro caso dos clases, imagen de molécula o no). Cada coordenada se conoce como *logit*.

Tras la implementación con estos cambios, el número de parámetros se incrementa. Por ejemplo, LeNet5 contaba originalmente con 62006 parámetros al trabajar con imágenes 28x28 y un vector de salida de tamaño 10, pero tras los cambios cuenta con 7393806. Las versiones modificadas de AlexNet y VGG16 cuentan respectivamente con 58289538 y 153263298 parámetros.

Cada uno de estos modelos tiene diferentes hiperparámetros configurables: el tamaño del lote de entrenamiento (*batch size*), el optimizador que modifica los parámetros en cada iteración, la inicialización de los pesos, etc. Determinados valores en estos hiperparámetros hacen que los modelos produzcan peores o mejores resultados, por lo que es importante elegir valores adecuados. Es imposible probar todos los valores posibles, por lo que seleccionamos algunos que creemos que pueden dar buenos resultados y realizamos una comparativa: esto es lo que se conoce como *grid search*.

Vamos a trabajar con tres hiperparámetros. Aunque podríamos hacer pruebas con más, el número de modelos a entrenar crecería exponencialmente:

- **Tasa de aprendizaje (*Learning rate*):** Es probablemente el hiperparámetro más importante. Si le damos un valor demasiado bajo, los pesos del modelo variarán en muy poca medida en cada iteración y el error decrecerá muy lentamente. Si su valor es muy alto, es posible que el valor de los pesos y del error oscile y el algoritmo no converja. Es conveniente probar con diferentes posibilidades: en nuestro caso vamos

a comprobar el rendimiento de los modelos tomando 0.5, 0.05, 0.005, 0.0005 y 0.00005 como posibles valores. [13]

- **Inicialización de los pesos:** Las redes neuronales dependen en gran medida de la inicialización de los pesos. Si todas las neuronas se inicializasen con el mismo valor, el modelo no sería capaz de aprender nada, ya que la derivada del gradiente sería la misma para todas. Las que voy a utilizar son: [32]

- *He*: utilizada por defecto en PyTorch en sus capas completamente conectadas y convolutivas, He (también conocida como Kaiming) genera valores aleatorios a partir de una distribución uniforme $\mathcal{U}(-\text{bound}, \text{bound})$, donde $\text{bound} = \text{gain} * \sqrt{\frac{3}{\text{fan_in}}}$. El parámetro *fan.in* depende de la entrada a la neurona. Por otro lado, en la inicialización por defecto de PyTorch *gain* depende de $a = \sqrt{5}$.
- *Xavier*: Los valores aleatorios son generados a partir de una distribución uniforme $\mathcal{U}(-a, a)$, donde $a = \sqrt{\frac{6}{\text{fan_in} + \text{fan_out}}}$. Los parámetros *fan.in* y *fan.out* dependen de la entrada y de la salida de la neurona respectivamente.

Una correcta inicialización de los pesos evita el problema de la evanescencia del gradiente (durante el entrenamiento, este se vuelve cada vez menor aproximándose a cero, de forma que los pesos no cambian) y el de la explosión del gradiente (este toma valores cada vez mayores, de forma que los pesos cambian en gran medida y el algoritmo diverge).

- **Algoritmo de optimización:** Como muchos otros algoritmos de aprendizaje automático, los de aprendizaje profundo son problemas de optimización donde se intenta reducir el error de una función objetivo f definida sobre un conjunto de datos de entrenamiento. En muchas ocasiones, este proceso de optimización se realiza de forma numérica, una forma de hacerlo es a partir del cálculo del gradiente descendiente, un proceso costoso por tener que calcular en numerosas ocasiones el valor de la función f y de su derivada. La elección del algoritmo de optimización es importante, ya que de él dependen la convergencia a una buena solución y el actualizar los pesos de forma eficiente: [13]

- *SGD*: En el entrenamiento de redes neuronales, el gradiente se calcula a partir de los datos de entrenamiento. Si estos datos no son un conjunto representativo de la distribución real porque contienen ruido, la estimación no será correcta. El gradiente se puede calcular a partir de todos los datos de entrenamiento (aprendizaje por lotes), a partir de un subconjunto (aprendizaje por minilotes) o a partir de una única muestra (aprendizaje online). Cuantos más datos utilicemos para calcular el gradiente, menor será el error cometido al estimarlo, pero como hemos mencionado, los datos siempre pueden contener ruido y por tanto no podemos asegurar que los resultados sean correctos. El gradien-

te descendiente estocástico (*Stochastic Gradient Descent*, SGD) engloba el aprendizaje por minibatches y el online, y aunque pueda parecer lo contrario por utilizar menos muestras en el cómputo del gradiente, mejora los resultados al permitir al algoritmo escapar de puntos de silla. Además, es mucho más eficiente ya que solo tiene en cuenta un número limitado de datos en el cómputo.

- *AdaDelta*: Es una extensión de AdaGrad (*Adaptive Gradients*), un optimizador que dota a cada parámetro de una tasa de aprendizaje independiente y que va actualizando su valor durante la ejecución del algoritmo. AdaDelta resuelve el principal problema de Adagrad, la disminución prematura de las tasas de aprendizaje. Esto se debe a que Adagrad las calcula a partir del gradiente acumulado de todas las iteraciones, en cambio AdaDelta sólo utiliza las n iteraciones anteriores.
- *Adam*: Se podría considerar una extensión de AdaDelta en la que se utilizan momentos. En concreto, este algoritmo calcula la estimación del primer momento del gradiente (media) y del segundo momento (varianza). A partir de ellos se define la fórmula que da nombre a este algoritmo.

En las ejecuciones que voy a realizar, la función de coste que utilizamos es la función de entropía cruzada (*Cross Entropy Loss*). Esta función integra la función de activación *Softmax*, de forma que cada logit es transformado en un valor probabilístico entre 0 y 1.

El algoritmo de aprendizaje intentará minimizar el valor de la función de entropía cruzada, es decir, tratará de minimizar la distancia entre las probabilidades devueltas por el modelo y la probabilidad real. Tras el entrenamiento, cuando realicemos una predicción, la clase a la que pertenecerá la muestra será aquella que esté representada por el *logit* con mayor probabilidad. [29]

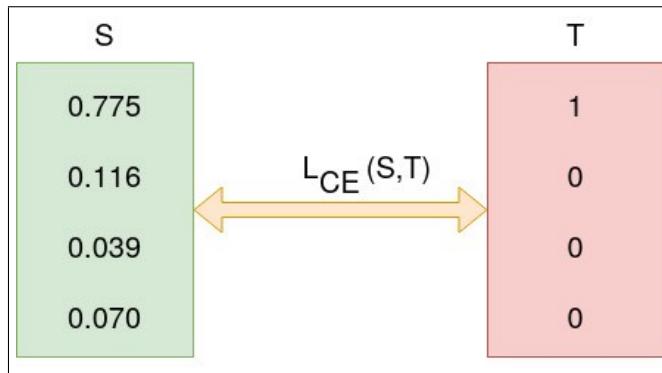


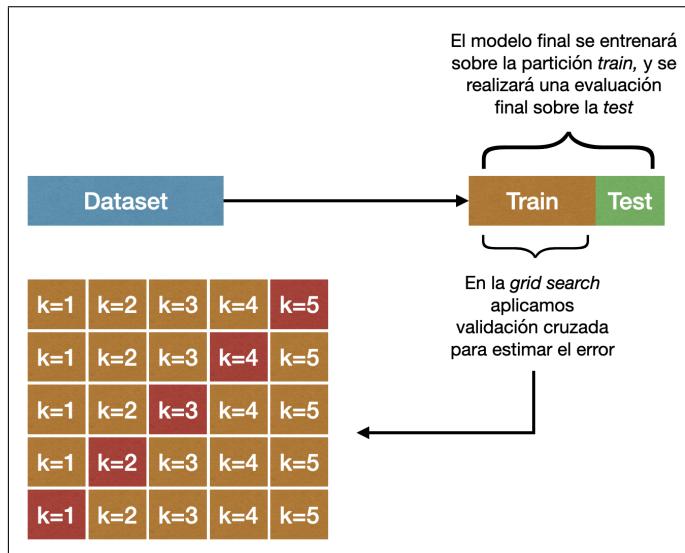
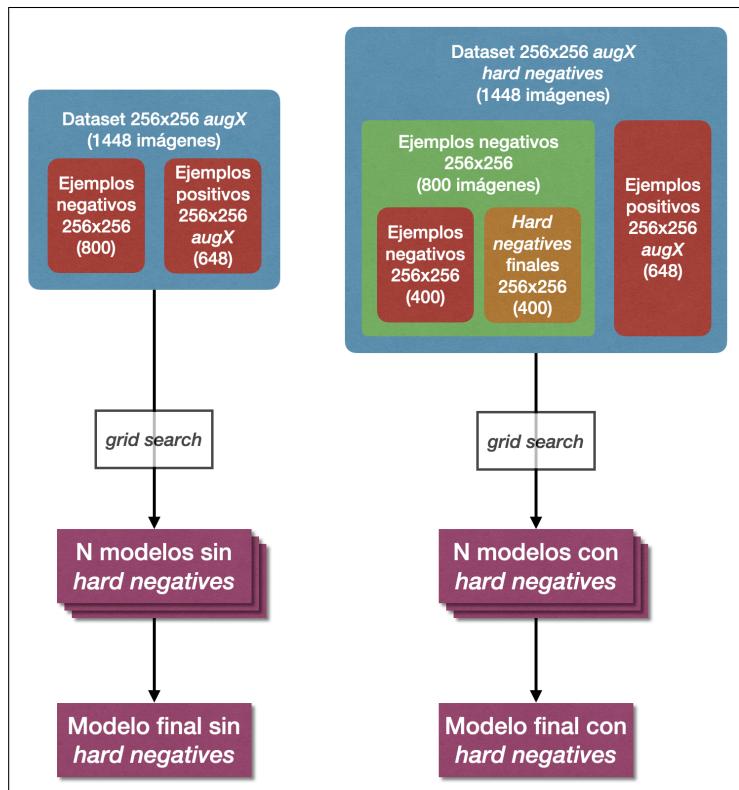
Figura 4.7: La función de entropía cruzada L_{CE} mide la distancia entre la probabilidad representada en el *logit* S y la probabilidad real representada mediante T [29]. Durante el entrenamiento, el algoritmo tratará de minimizar esta distancia.

Hay otro hiperparámetro que fijo desde el inicio que es el tamaño de los minilotes. Podría haber probado con tamaños diferentes, pero de hacerlo la complejidad de la *grid search* crecía en gran medida. Un valor adecuado es aquel que permita que el minilote quepa en la caché de la GPU, de forma que se agilicen los cálculos. Valores como 32, 64 o 128 suelen ser utilizados en la literatura, así que nosotros elegimos fijar 32. [34]

El objetivo es realizar la *grid search* sobre los dos *datasets* que he comentado, de esta forma compararemos el rendimiento de los modelos según los hiperparámetros utilizados y nos quedaremos con aquellos que dan mejores resultados. Estos serán los que utilicen los modelos que entreguemos a los científicos.

Los *datasets* se dividirán en dos secciones, *train* y *test*. *train* se utilizará para entrenar los modelos finales (y por tanto será la mayor división, 85 %) y *test* para evaluar su rendimiento final (15 %). Pero en la *grid search* utilizaremos solamente la división de *train*, a la que aplicaremos validación cruzada. Esta técnica pretende estimar el error de forma más fiable repitiendo el proceso de entrenamiento con diferentes subconjuntos. En concreto, se divide aleatoriamente el conjunto en k partes (en nuestro caso $k = 5$). En cada iteración (k en total), se utiliza un subconjunto como *test* y el resto como *train*. [13]

Finalmente, tras la *grid search*, cuando se haya decidido cuál es la configuración final, se entrenará el modelo final sobre la partición *train* y se dará un valor de error sobre la *test*. (Como ya hemos comentado realmente tendremos dos modelos finales, uno entrenado sobre el *dataset* sin *hard negatives* y otro sobre el que si los tiene)

Figura 4.8: División *train-test*.Figura 4.9: Realizamos una *grid search* sobre ambos datasets, y elegimos el modelo que mejor resultado da sobre cada uno de ellos.

Capítulo 5

Experimentación

En esta sección se explican los experimentos realizados hasta alcanzar la versión final del clasificador siguiendo los pasos propuestos en la metodología. Delimitar las pruebas que se van a realizar en un proyecto es fundamental para concentrar los esfuerzos y cumplir con los objetivos en plazo. Al tratarse de un proyecto de investigación, realizo experimentos iniciales, evalúo los resultados y vuelvo a realizar experimentos hasta alcanzar unos resultados que cumplan con los objetivos.

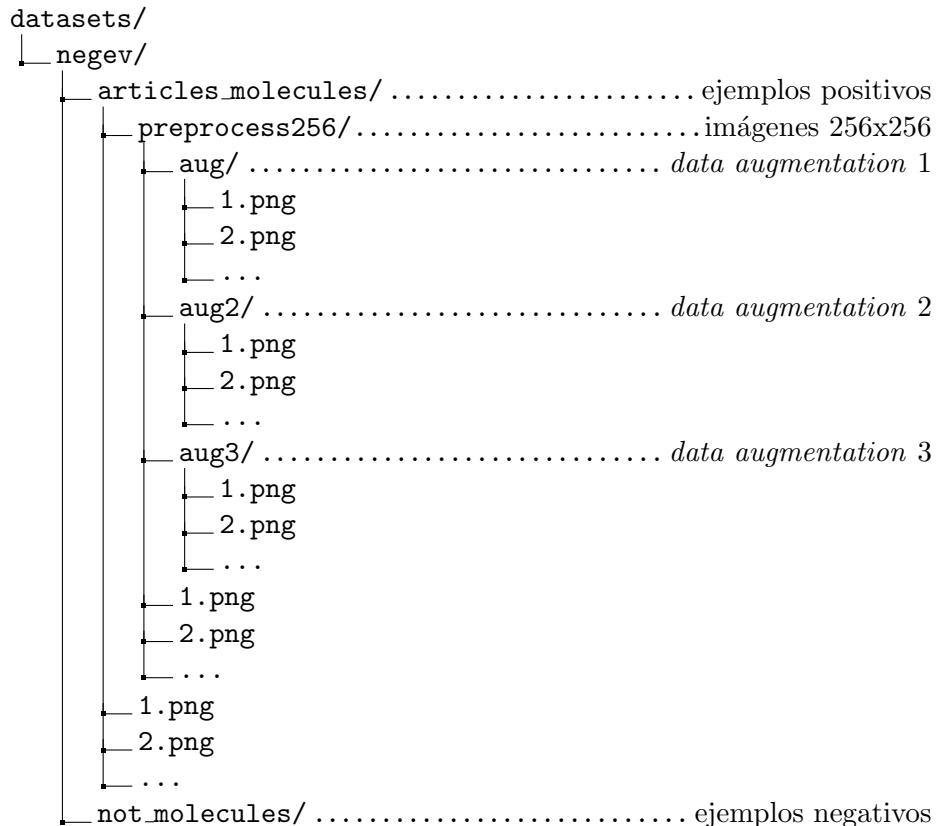
La implementación de los experimentos realizados se encuentra en el repositorio de GitHub del proyecto [31]. La raíz del repositorio contiene los siguientes elementos:

```
/  
└── datasets/  
└── experiments/ ..... implementación del TFG  
└── papers/ ..... algunas de las publicaciones utilizadas en su desarrollo  
└── report/ ..... memoria en formato LATEX  
└── slides/  
└── README.md
```

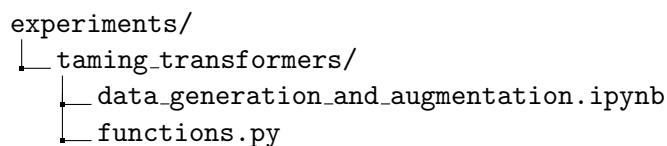
5.1. Transformaciones sobre el *dataset*

Tras recibir el *dataset* y normalizar el tamaño de las imágenes, procedo a aplicar las tres secuencias de *data augmentation* que he explicado en el capítulo anterior.

El *dataset* original y los derivados obtenidos al aplicar *data augmentation* se almacenarán en el directorio *datasets/* del repositorio, como se describe en el siguiente árbol:



El código que nos permitirá realizar generar estas transformaciones se encuentra en los siguientes archivos:



data_generation_and_augmentation.ipynb contiene el código que pre-procesa las imágenes, transformándolas de forma que tengan el mismo tamaño, y que aplica las tres secuencias de *data augmentation* que he comentado.

functions.py declara funciones utilizadas por todos este cuaderno.

Aunque no se mencionan aquí, existen otros directorios y ficheros auxiliares que permiten funcionar a estos.

Comienzo por *data augmentation 1*. Al aplicarla se obtienen los siguientes resultados:

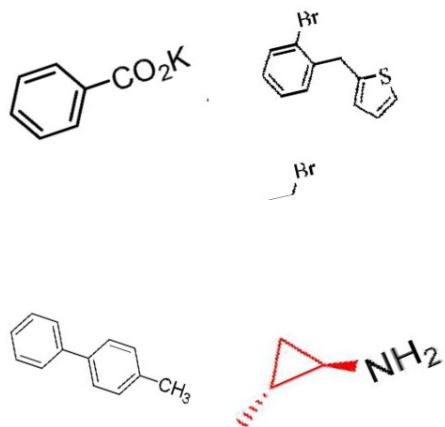


Figura 5.1: Imágenes generadas aplicando *data augmentation 1*

Se aprecian leves rotaciones y estiramientos de las imágenes, sin existir gran diferencia con las originales. Al estar aplicando tres veces la transformación sobre el conjunto de datos, puede que los cambios sean demasiado suaves dando lugar a imágenes muy similares. Observemos los resultados de *data augmentation 2*:

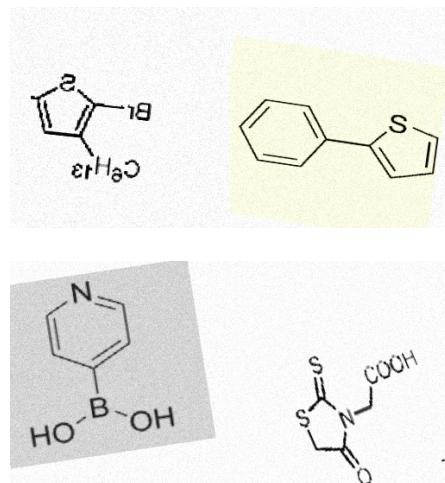


Figura 5.2: Imágenes generadas aplicando *data augmentation 2*

En este caso se observan más cambios fundamentalmente debidos a las variaciones en contraste, al uso de ruido gaussiano y a la multiplicación, responsable de los cambios de color, ya que a veces se aplica a canales independientes. Puede ser una transformación adecuada, ya que añade diversidad al conjunto sin cambios demasiado pronunciados.

Por último, comprobemos los resultados que se obtienen al aplicar *data augmentation* 3:

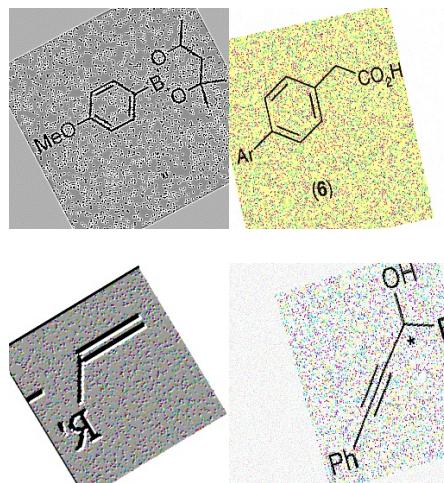


Figura 5.3: Imágenes generadas aplicando *data augmentation* 3

En esta última versión las transformaciones son bastante agresivas. Se introduce mucho ruido en las imágenes y las rotaciones y los cambios en el contraste son fuertes.

Para entrenar el clasificador, seguramente las imágenes más adecuadas sean las generadas por *data augmentation* 2. Son las más equilibradas, ya que la versión 1 apenas introduce cambios y la versión 3 añade demasiado ruido, creando imágenes que no se corresponden con la realidad y por tanto pueden penalizar el rendimiento del modelo.

5.2. Generación de imágenes sintéticas

Sobre los cuatro conjuntos de datos se entrenarán modelos con diferente número de épocas, desde 70 hasta 170, aumentando de 20 en 20. De esta forma tendremos $4 * |[70, 90, 110, 130, 150, 170]| = 4 * 6 = 24$ modelos diferentes.

Una vez entrenados, probamos a generar imágenes a partir de estos para ver cuál se comporta mejor. Para ello, debemos introducirles algún tipo de imagen de entrada.

La generación de imágenes la vamos a realizar utilizando el modelo desarrollado por Esser et al. [18], descrito en la revisión del estado del arte de este TFG . El código en el que se realizan los experimentos con el generador se encuentra en los siguientes archivos:

```
experiments/ ..... implementación del TFG
└── taming_transformers/ ..... generador de imágenes
    ├── taming-transformers/ ..... implementación del generador
    ├── sample_and_clean_molecules.ipynb
    ├── sampling_experiment.ipynb
    └── functions.py
```

El subdirectorio *taming-transformers/* contiene la implementación del modelo generativo proporcionada por sus autores [22].

sampling_experiment.ipynb es un cuaderno que, una vez entrenados los modelos, permite cargarlos y generar imágenes sintéticas a partir de otra imagen de entrada. Se utiliza para comprobar a partir de que *dataset* se generan mejores resultados y con qué número de épocas. Es importante saber que este modelo generativo necesita de una imagen condicionante para generar una imagen sintética: probaremos con distintos tipos, entre ellas ruido, para comprobar con cuáles se obtienen mejores resultados.

sample_and_clean_molecules.ipynb nos permite generar un lote de imágenes sintéticas indicando el modelo que queremos utilizar y las propiedades del ruido, en concreto de ruido Perlín, ruido que explicaremos en los experimentos. También creará el *dataset* final que contiene *hard negatives*.

functions.py declara funciones utilizadas por todos estos cuadernos.

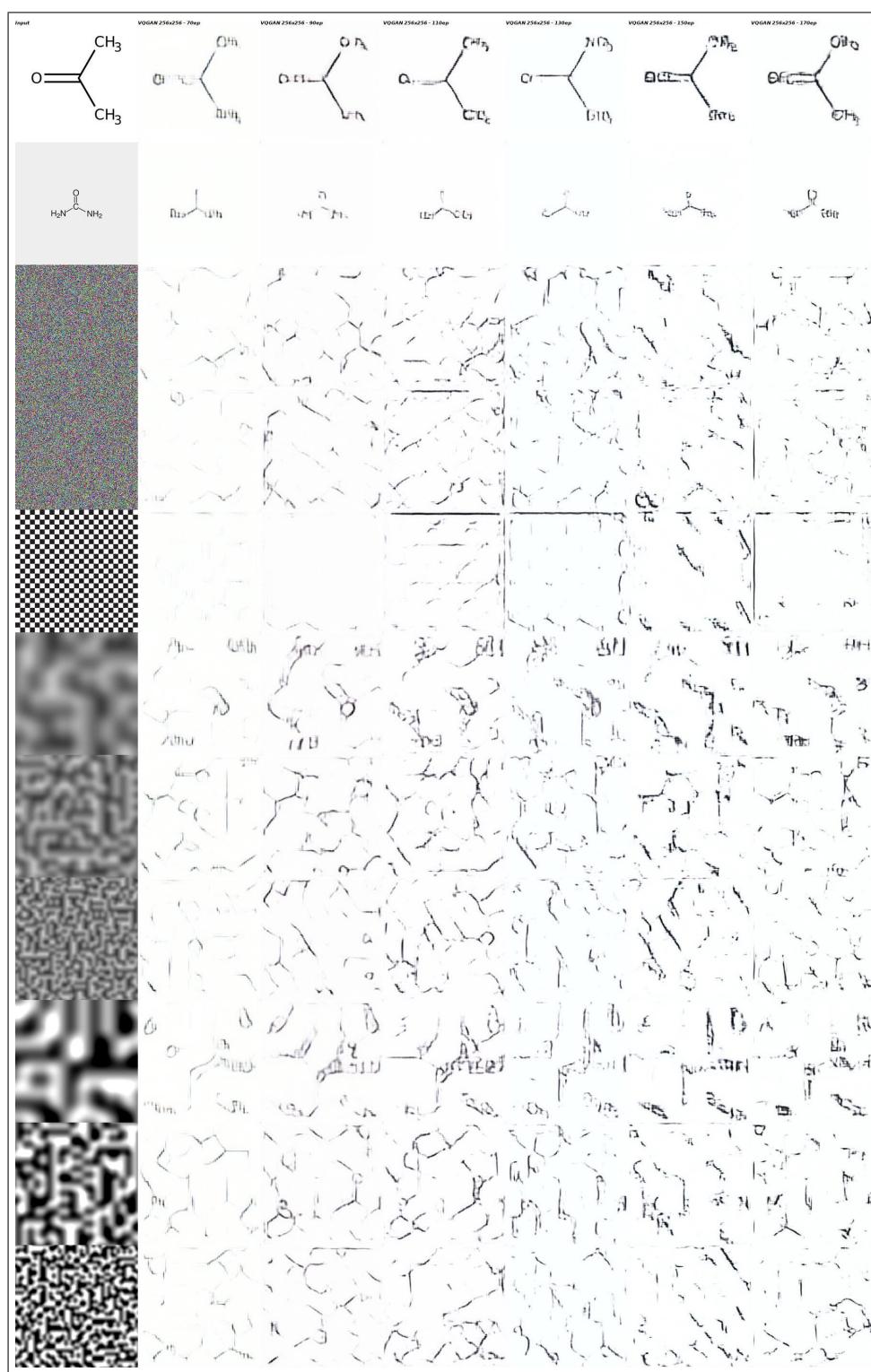
En las páginas siguientes se muestran los resultados. Cada fila corresponde a una imagen de entrada al modelo. En las columnas, la primera muestra la imagen de entrada mientras que cada una de las siguientes corresponde con un modelo entrenado durante x épocas, desde 70 hasta 170 (70, 90, 110, 130, 150, 170).

Se puede observar como la versión sin *data augmentation* no funciona bien, produce imágenes sin apenas estructura. Esto se debe a que solo se utilizan 162 imágenes, cifra muy baja para una arquitectura de deep learning como es un *transformer*. Con *data augmentation* 1 tampoco se observan resultados de buena calidad, puede que algo mejores que en el caso anterior pero sin muchas diferencias.

Es en *data augmentation* 2 cuando ya se empiezan a ver mayores diferencias. El uso de transformaciones más fuertes crea un conjunto heterogéneo de imágenes y el modelo puede aprender más características de estas. Los resultados son aceptables, aunque el cambio de color que las transformaciones producían en algunas imágenes de entrenamiento ha dado lugar a fondos grisáceos en sus homólogas sintéticas.

Con el tercer *data augmentation* estos fondos grisáceos son mucho más marcados. Recordamos que estas transformaciones eran mucho más fuertes, por lo que se traslada a las imágenes sintéticas.

Figura 5.4: Resultados de entrenar los modelos sobre el conjunto de datos sin *data augmentation*. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.



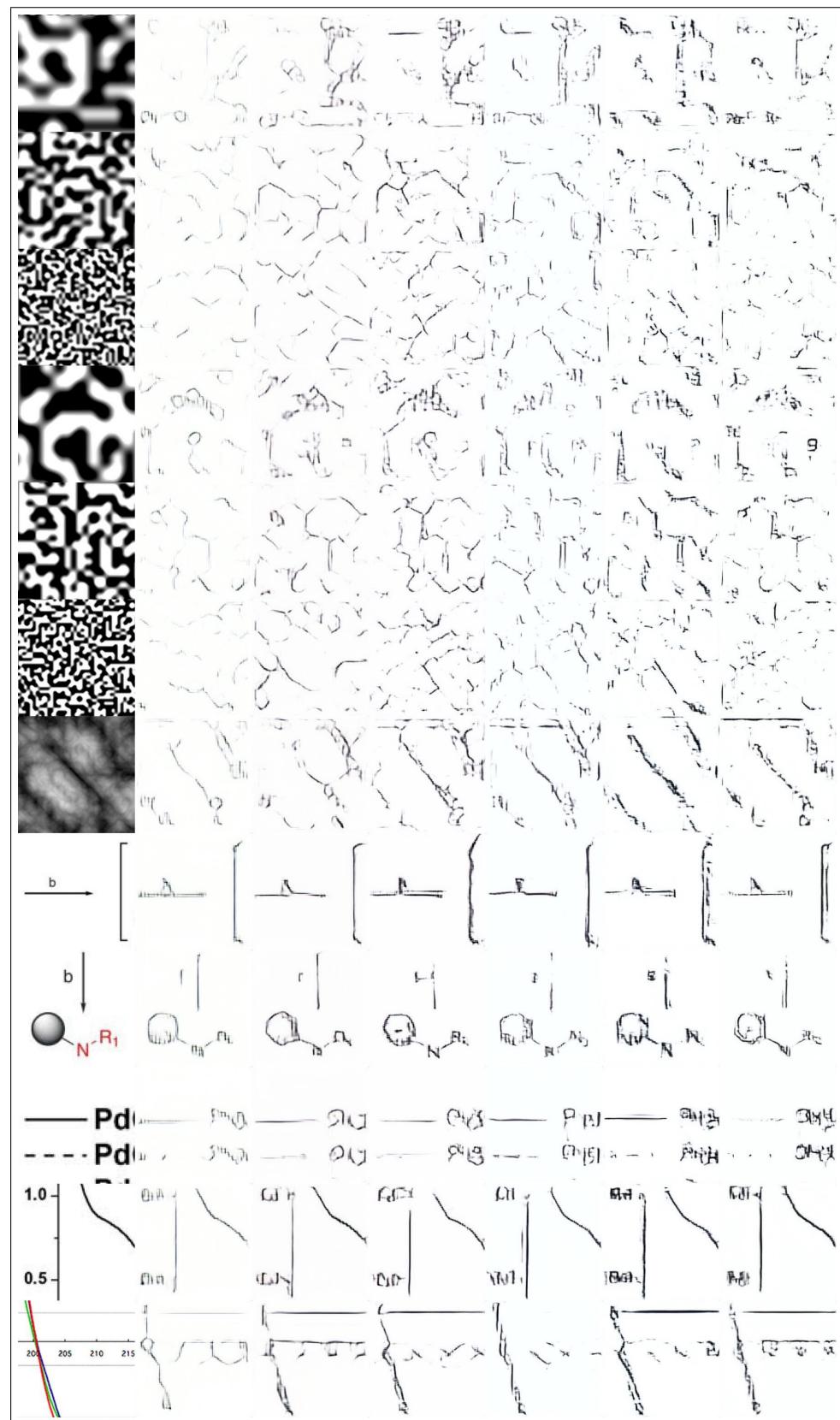
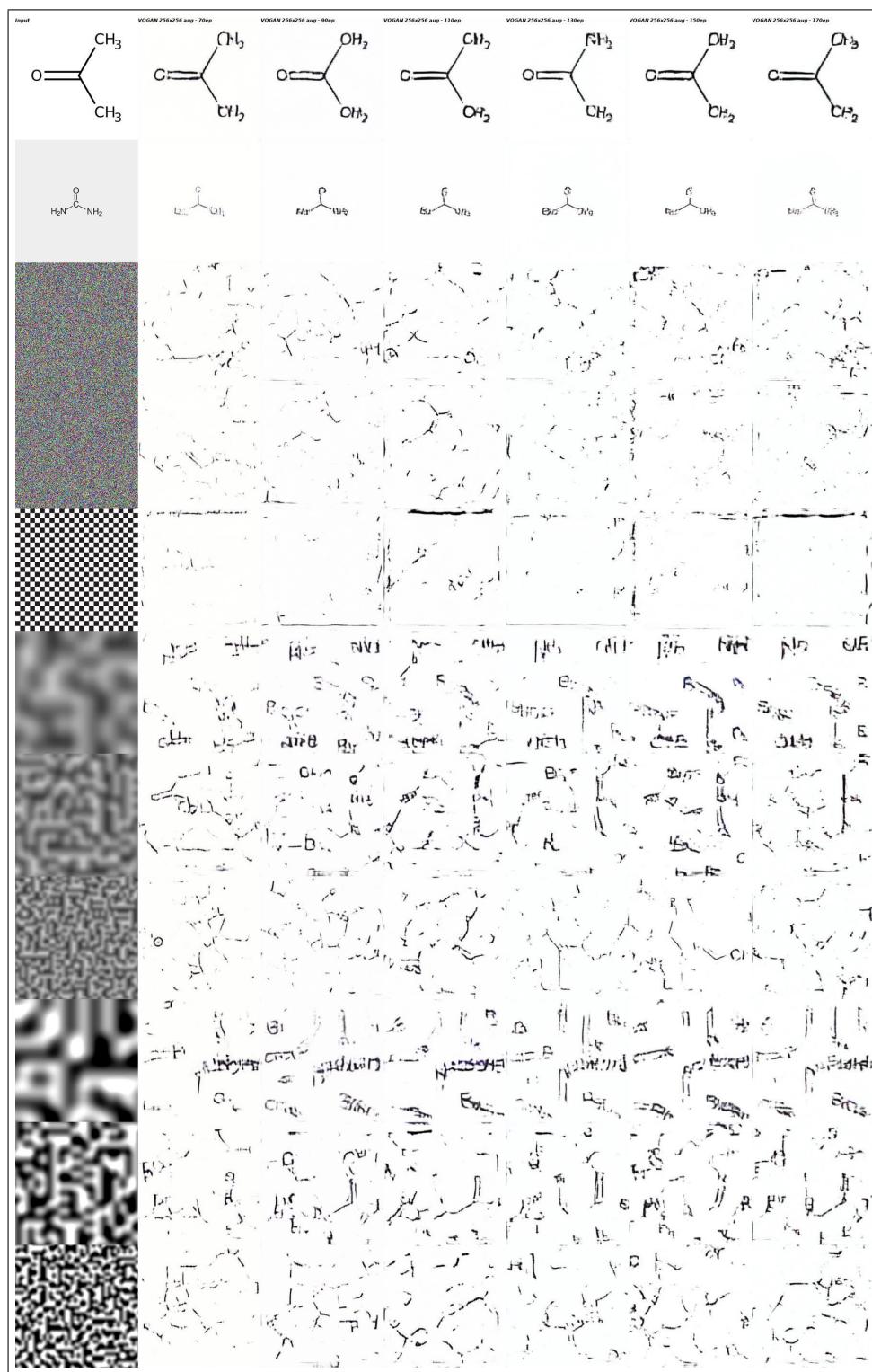


Figura 5.5: Resultados de entrenar los modelos sobre el conjunto de datos con *data augmentation* 1. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.



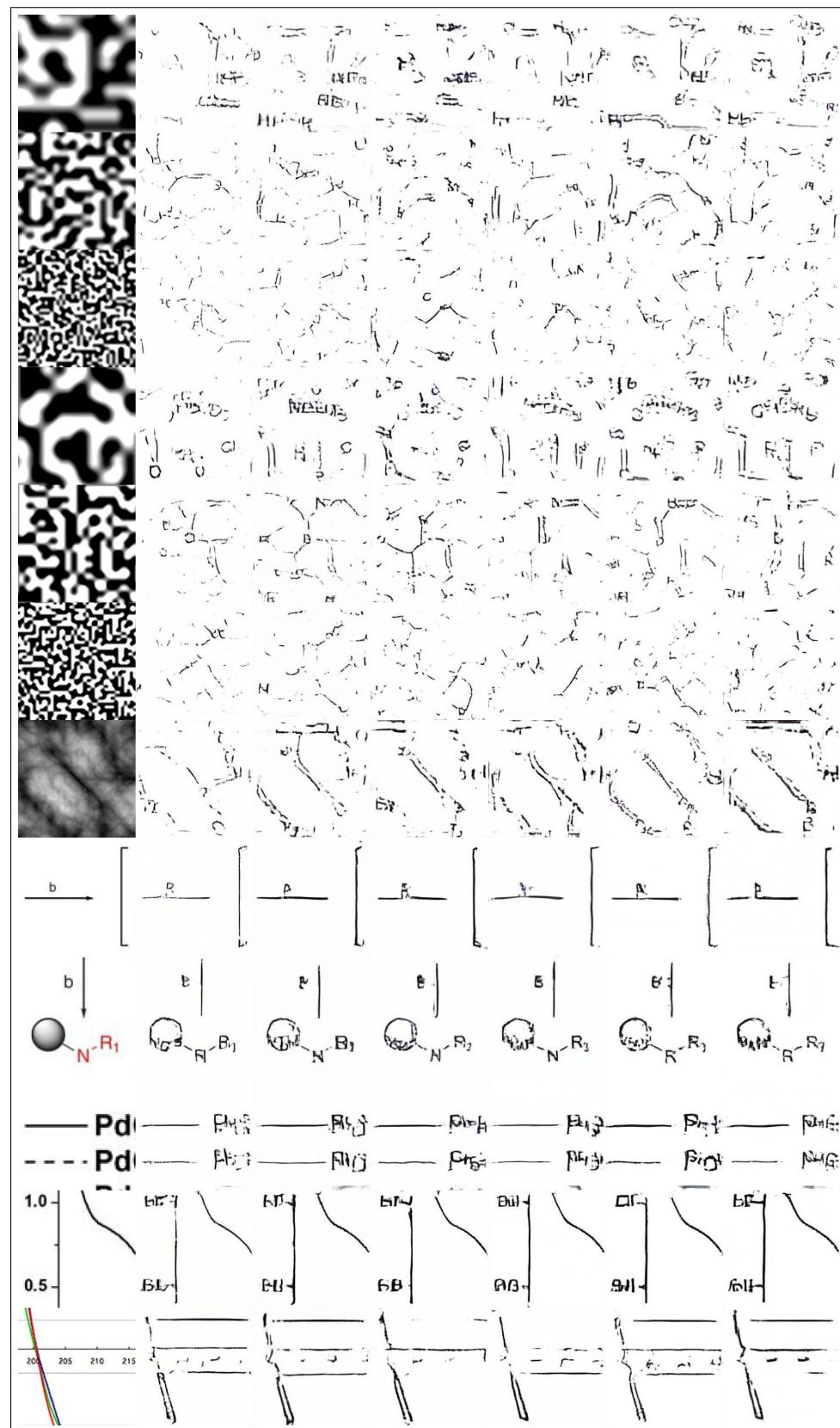
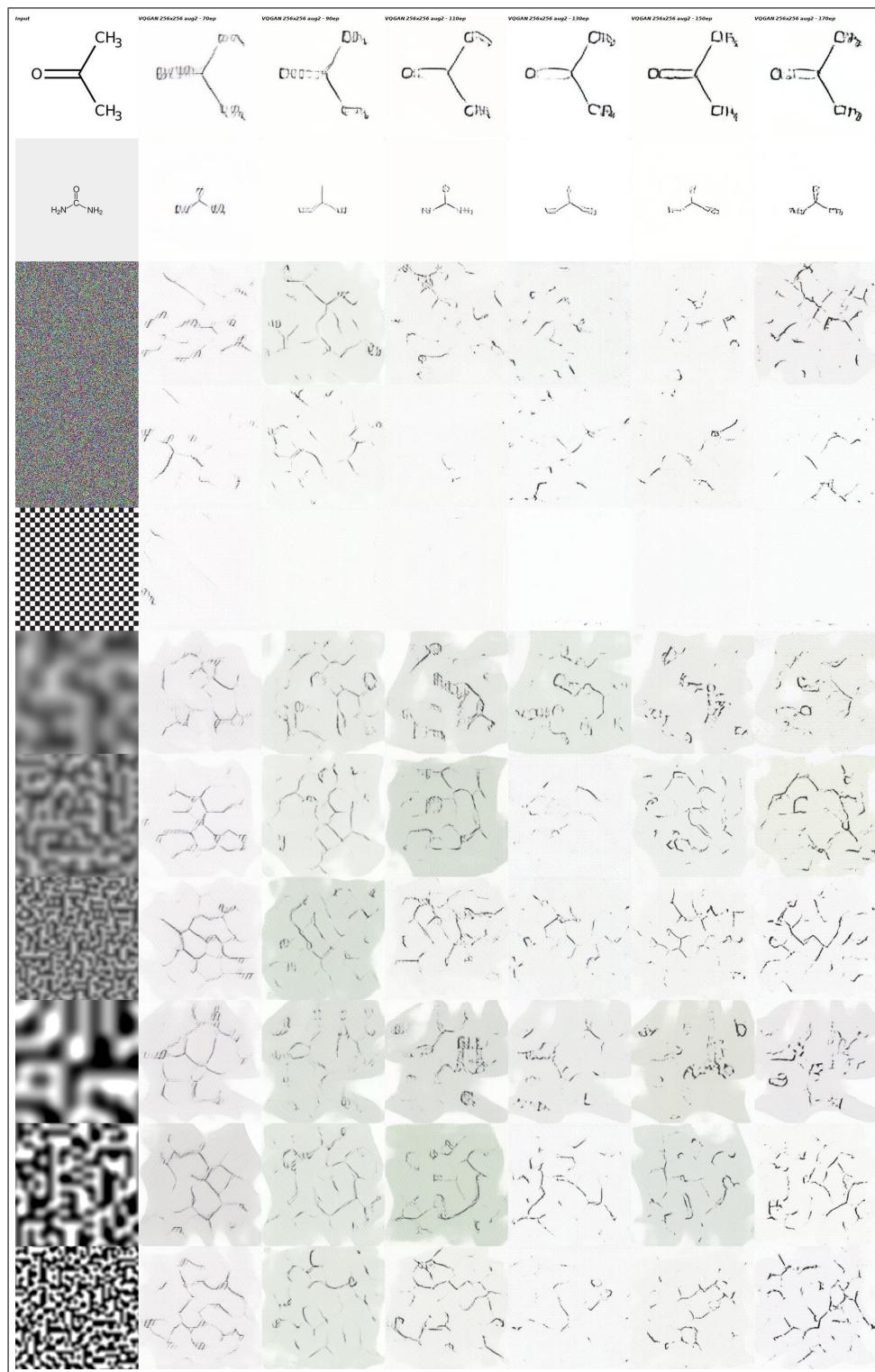


Figura 5.6: Resultados de entrenar los modelos sobre el conjunto de datos con *data augmentation* 2. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.



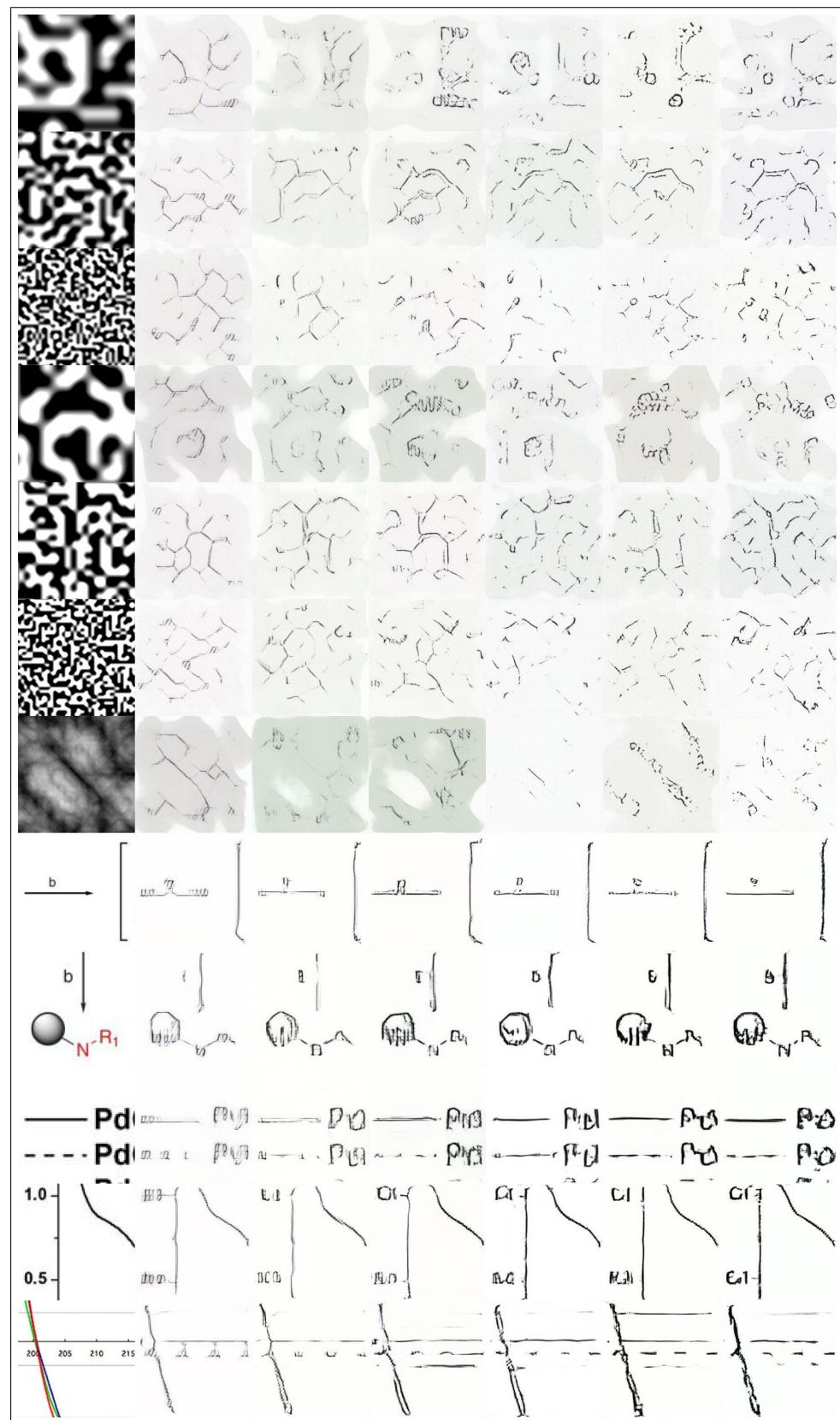
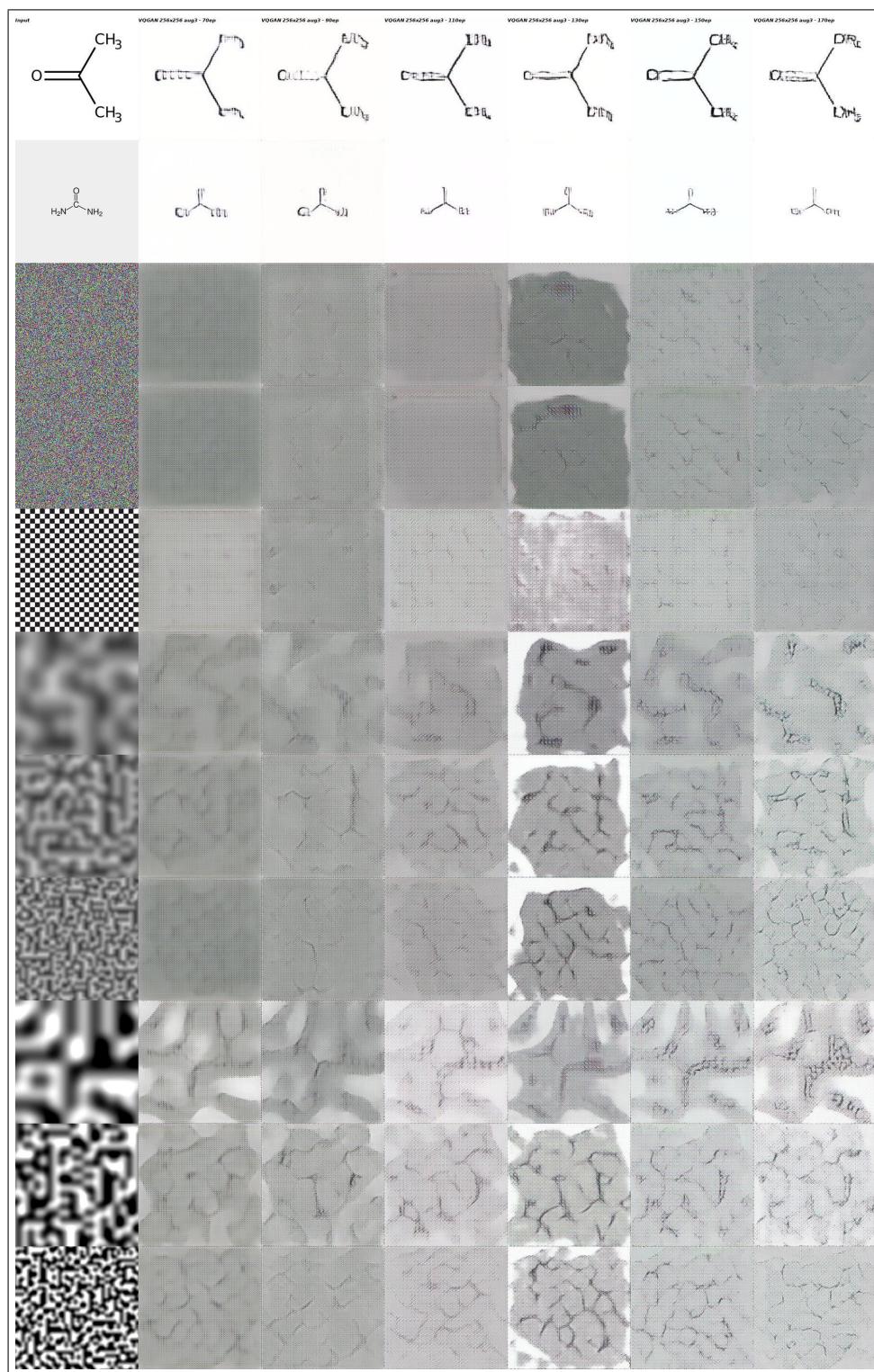
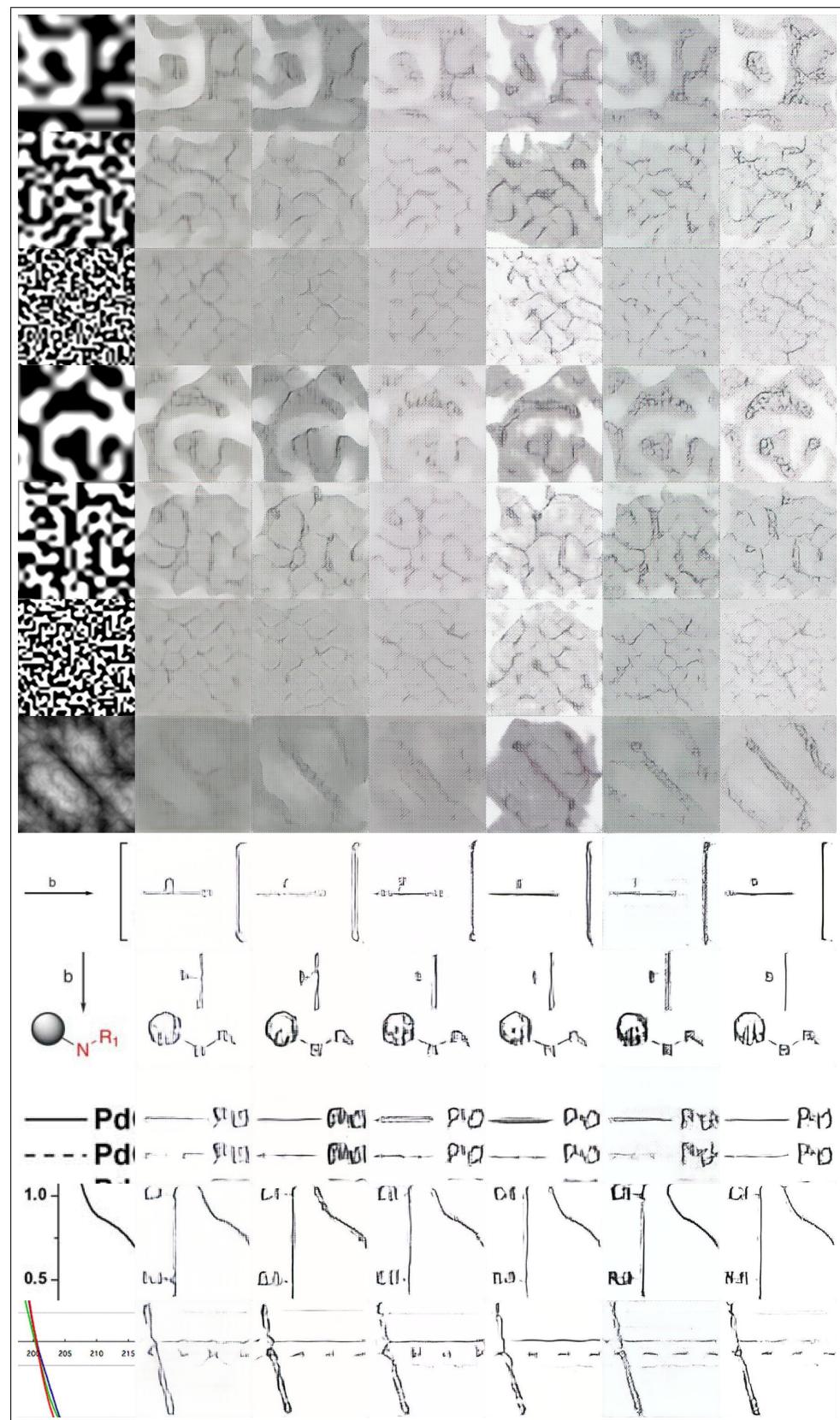


Figura 5.7: Resultados de entrenar los modelos sobre el conjunto de datos con *data augmentation* 3. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde 70 hasta 170 épocas, tomadas de 20 en 20.





¿Qué tipo de datos de entrada se ha utilizado para generar las imágenes sintéticas? En un primer momento probamos a utilizar imágenes de moléculas reales y de objetos que, sin ser moléculas, lo parecen. En el primer caso, la salida es similar a la entrada, con la diferencia de que la imagen sintética pierde información: detalles como letras o números se alteran. En el segundo caso ocurre algo similar, con la particularidad de que objetos circulares son transformados en formas que parecen hexágonos propios de las moléculas. Además, incluso se añaden átomos.

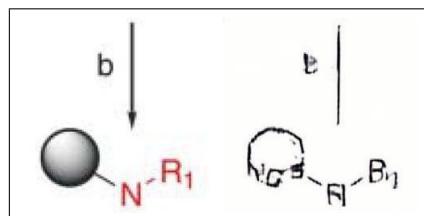


Figura 5.8: El modelo tiene la capacidad de modificar la imagen para que se parezca a una molécula.

Pero queremos generar tantas imágenes como queramos, así que deberíamos poder partir de una imagen de entrada generada aleatoriamente. En un primer momento se utilizó ruido uniforme, pero no funcionó bien en ningún caso. Este tipo de ruido no sigue ninguna estructura, y nuestro modelo necesita una entrada que tenga cierta continuidad. Por ello pruebo con una cuadrícula de ajedrez: ocurre lo mismo, ya que aunque existe un patrón que se repite, los elementos de este no están conectados entre sí.

5.2.1. Ruido Perlín

Se trata de un ruido inventado por Ken Perlín en 1982 que revolucionó el ámbito de la Informática Gráfica. En esta rama de las Ciencias de la Computación, la creación de texturas que imitan materiales de la naturaleza como la madera o el mármol es una tarea importante. Generar estas texturas de una forma manual o mediante un escáner no es una opción escalable.

Este ruido permite simular fenómenos que requieran aleatoriedad a la vez que continuidad. Para ello, genera una serie de gradientes en un *grid* (en nuestro caso 2D). A continuación, interpola el valor de esos gradientes mediante interpolación polinómica de Hermite. La ventaja de este ruido frente a otros que también se generan mediante interpolación es su eficiencia, para calcular la interpolación en un punto solo intervienen los 2^n gradientes más cercanos, donde n es el número de dimensiones. [30]

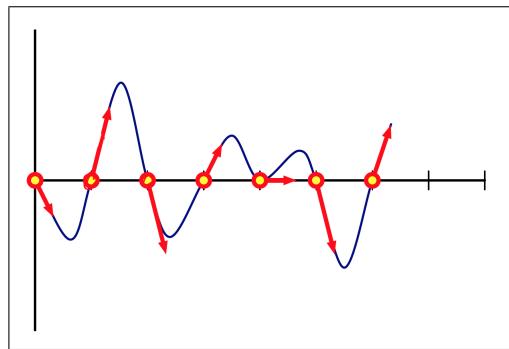


Figura 5.9: Interpolación de gradientes en el Ruido Perlín 1D. [30]

El Ruido Perlín se puede modificar cambiando la amplitud y frecuencia de este. A mayor amplitud, los cambios entre zonas del ruido serán más bruscos, a menor amplitud el ruido será más homogéneo y suave. La frecuencia cambia la granularidad del ruido, a mayor frecuencia encontramos un grano más fino.

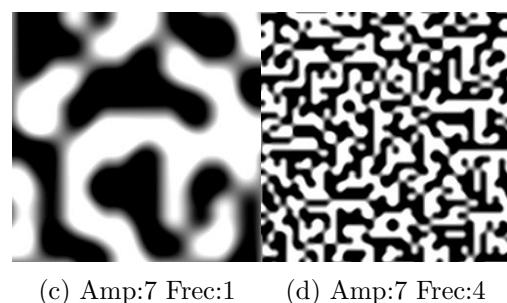
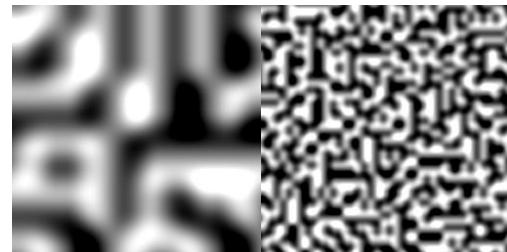
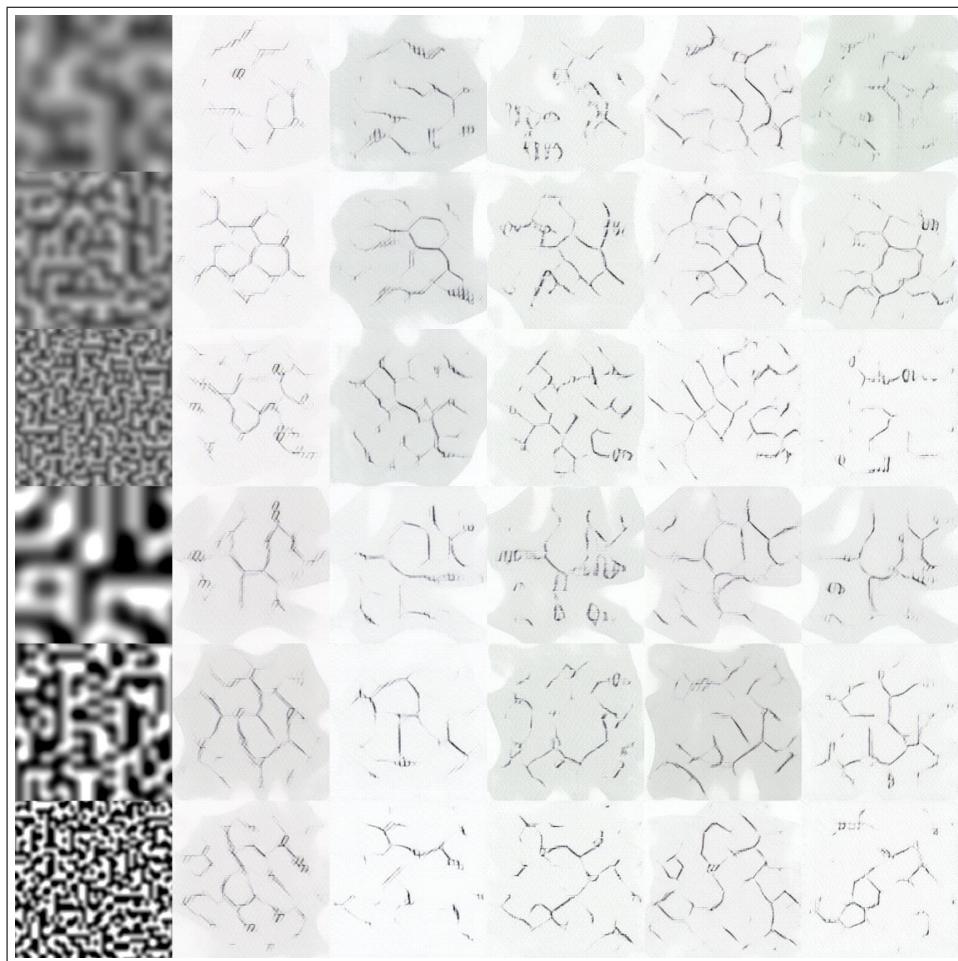


Figura 5.10: Ejemplos de Ruido Perlín con distinta amplitud y frecuencia.

Como se observa en las figuras 5.4, 5.5, 5.6 y 5.7, al contrario que con el ruido uniforme, este tipo de ruido es capaz de generar imágenes que parecen moléculas. Su continuidad permite que el modelo tenga una estructura en la que basarse para realizar la generación.

Tras realizar estos experimentos, se los mostramos al equipo para conocer su opinión: está conforme con la versión *data augmentation* 2 entrenada durante 70 épocas en aquellos casos en los que se utiliza Ruido Perlín como entrada (figural 5.6, segunda columna). De todas formas, como se obtienen buenos resultados entre 70 y 90 épocas, vamos a entrenar con *data augmentation* 2 durante 70, 75, 80, 85 y 90 épocas y a comparar los resultados.

Figura 5.11: Resultados de entrenar con imágenes con *data augmentation* 2. La primera columna representa la imagen de entrada (ruido Perlín en todos los casos), el resto los diferentes modelos entrenados desde las 70 hasta las 90 épocas tomadas de 5 en 5.





En diferentes números de épocas/tipos de ruido Perlín se obtienen buenos resultados, por lo que no vamos a utilizar una única configuración para generar los *hard negatives*, utilizaremos varias hasta alcanzar los 400 que queremos generar.

En concreto, vamos a obtener 50 imágenes a partir de cada una de estas configuraciones (todas ellas obtenidas tras entrenar sobre *data augmentation* 2):

- Modelo entrenado durante 70 épocas al que se le introduce ruido Perlín con amplitud 1 y frecuencia 2 (fila 2 columna 2 de la figura 5.11).
- Modelo entrenado durante 70 épocas al que se le introduce ruido Perlín con amplitud 1 y frecuencia 4 (fila 3 columna 2 de la figura 5.11).
- Modelo entrenado durante 70 épocas al que se le introduce ruido Perlín con amplitud 3 y frecuencia 2 (fila 5 columna 2 de la figura 5.11).
- Modelo entrenado durante 75 épocas al que se le introduce ruido Perlín con amplitud 1 y frecuencia 4 (fila 3 columna 3 de la figura 5.11).

- Modelo entrenado durante 75 épocas al que se le introduce ruido Perlín con amplitud 3 y frecuencia 2 (fila 5 columna 3 de la figura 5.11).
- Modelo entrenado durante 80 épocas al que se le introduce ruido Perlín con amplitud 1 y frecuencia 2 (fila 2 columna 4 de la figura 5.11).
- Modelo entrenado durante 85 épocas al que se le introduce ruido Perlín con amplitud 1 y frecuencia 2 (fila 2 columna 5 de la figura 5.11).
- Modelo entrenado durante 85 épocas al que se le introduce ruido Perlín con amplitud 7 y frecuencia 2 (fila 11 columna 5 de la figura 5.11).

En total, 400 *hard negatives*. Antes de unificarlos con 400 ejemplos negativos del *dataset* original, los tratamos de forma que se reduce el color grisáceo de fondo que presentan tras ser producidos por los modelos generadores. Para ello, de forma independiente en cada una de las configuraciones, fijamos un umbral de color que, si no es superado por un pixel, este es transformado en color blanco. De esta forma, solo los píxeles con un color intenso se mantienen en la imagen, o sea, solo se mantiene la estructura molecular. Antes de eso, aplicamos un aumento de contraste a la imagen y un filtro de apertura (*opening*) que elimina ruido.

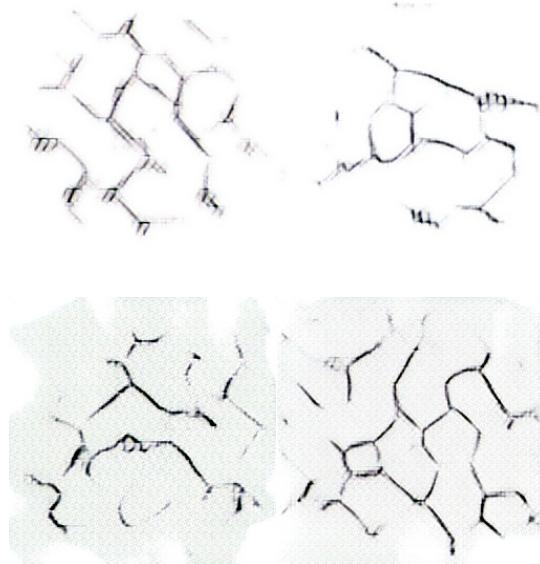
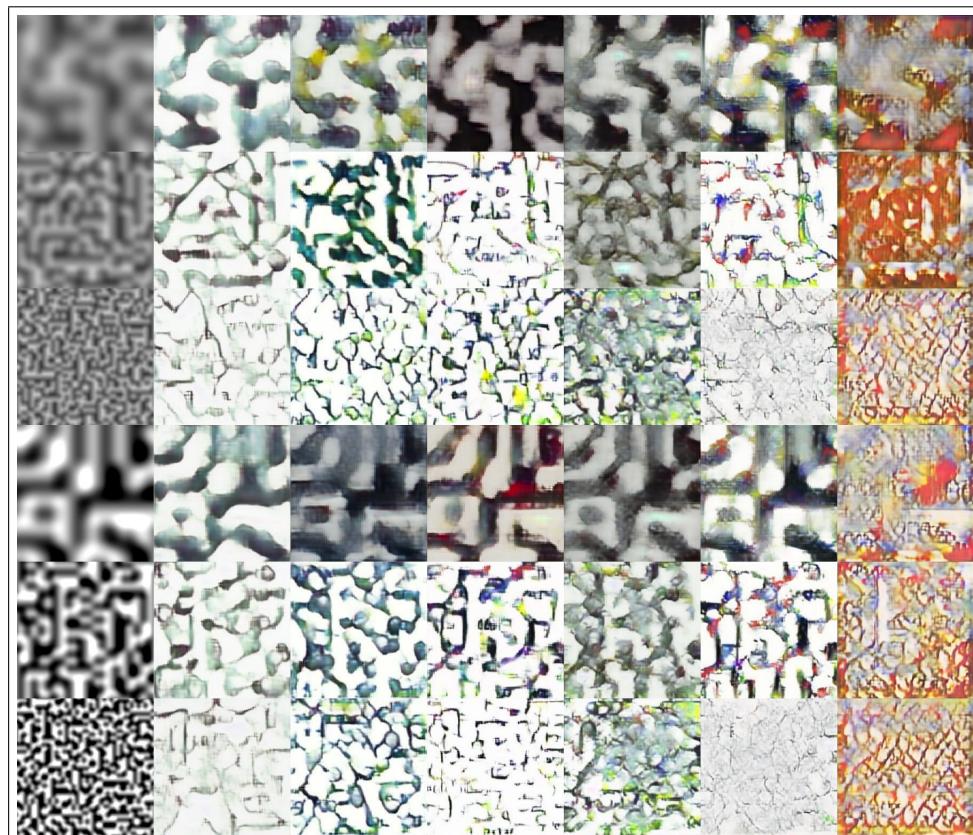


Figura 5.12: *Hard negatives* finales tras ser post-procesados.

Ya puedo crear los dos *datasets* finales para entrenar el clasificador, tal y como se expone en la figura 5.14 del capítulo anterior.

Otra prueba que se hizo y fue descartada consistió en entrenar el modelo generador a partir de ejemplos negativos. Si, en vez de entrenar sobre ejemplos positivos como estábamos haciendo hasta ahora, hacerlo sobre negativos. ¿Qué ocurrió? Como se puede observar en la imagen siguiente, los resultados no fueron buenos:

Figura 5.13: Resultados de entrenar con ejemplos negativos. La primera columna representa la imagen de entrada, el resto los diferentes modelos entrenados desde las 70 hasta las 170 épocas tomadas de 20 en 20.



Esto puede deberse a la alta diversidad que presentan los ejemplos negativos del *dataset*. Esta diversidad hace que el modelo generativo no pueda aprender, ya que las imágenes no tienen apenas características en común.

Por tanto, tenemos dos datasets finales con los que se entrenará cada una de las versiones del clasificador:

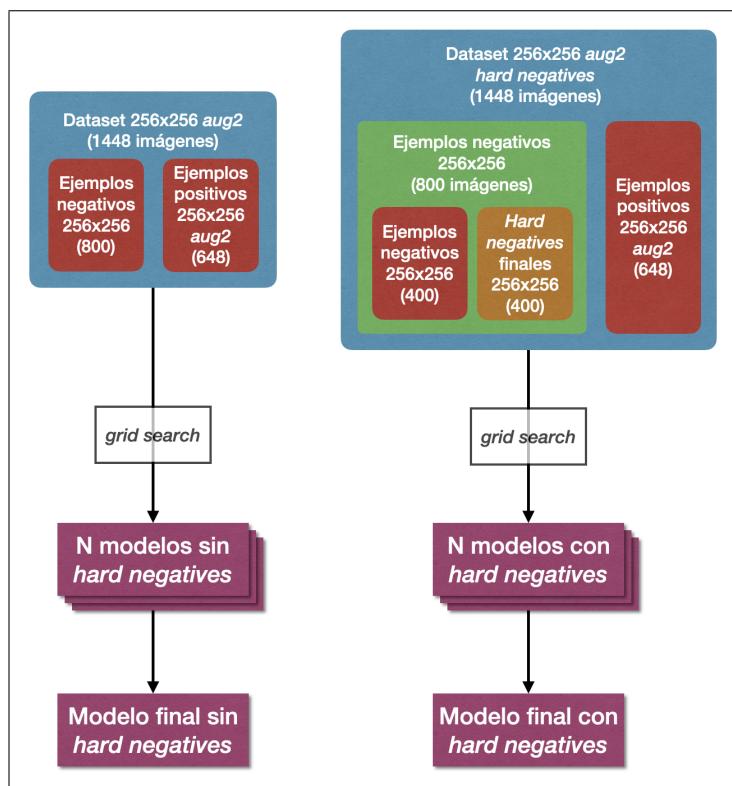


Figura 5.14: Dos *datasets* para entrenar dos clasificadores.

5.3. Clasificación de imágenes

Con los dos *datasets* preparados 5.14, paso a construir los clasificadores. Como mencionamos en el capítulo anterior, realizaré una *grid search* para comprobar que arquitecturas e hiperparámetros funcionan mejor.

El código en el que se realizan los experimentos con el clasificador se encuentra en los siguientes archivos:

```
experiments/ ..... implementación del TFG
└── image_classifier/ ..... clasificador de imágenes
    ├── datasets.py
    ├── models.py
    ├── grid_search.py
    ├── train_final_models.py
    └── functions.py
```

datasets.py declara la clase *CompoundDataset*, una clase que hereda la clase *Dataset* de Pytorch. Este objeto facilita la carga de las imágenes y su uso por las funciones y sentencias incluidas en PyTorch.

models.py implementa cada una de las arquitecturas con las que vamos a trabajar (LeNet5, AlexNet y VGG16). Se declaran cada una de sus capas, funciones de activación y el orden en el que la información fluye por estas.

grid_search.py implementa la *grid_search*, mediante un parámetro podemos indicar si queremos realizarla entrenando los modelos sobre el *dataset* con *hard negatives* o sin ellos.

train_final_models.py entrena los modelos finales con la configuración decidida tras realizar la *grid_search.py*.

functions.py declara funciones utilizadas por todos estos ficheros.

5.3.1. Clasificador sobre el *dataset sin hard negatives*

Tras ejecutar la *grid search*, obtenemos los siguientes resultados:

Tabla 5.1: *Grid search* utilizando la arquitectura LeNet5, entrenamiento sobre *dataset sin hard negatives*.

Ini. de pesos	Tasa de aprendizaje	Optimizadores					
		SGD		Adam		Adadelta	
Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)
He	0.5	44.715	1.971	49.593	6.630	44.715	3.042
	0.05	48.130	6.308	44.715	3.042	44.715	3.042
	0.005	44.715	3.042	44.715	3.042	44.715	3.042
	0.0005	44.715	3.042	44.715	3.042	44.715	3.042
	0.00005	44.715	3.042	44.715	3.042	48.130	6.308
	0.000005	48.130	6.308	44.715	3.042	48.130	6.308
	0.0000005	48.130	6.308	48.130	6.308	48.130	6.308
Xavier	0.5	48.130	6.308	48.943	6.540	26.260	16.213
	0.05	27.967	14.734	44.715	3.042	33.252	11.226
	0.005	43.902	4.378	44.715	3.042	43.740	4.685
	0.0005	44.715	3.042	44.715	3.042	44.715	3.042
	0.00005	44.715	3.042	41.951	6.018	48.130	6.308
	0.000005	48.130	6.308	40.813	6.596	48.130	6.308
	0.0000005	48.130	6.308	44.715	3.042	48.130	6.308

Tabla 5.2: *Grid search* utilizando la arquitectura AlexNet, entrenamiento sobre *dataset sin hard negatives*.

Ini. de pesos	Tasa de aprendizaje	Optimizadores					
		SGD		Adam		Adadelta	
Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)
He	0.5	44.715	3.042	49.268	7.061	5.041	1.691
	0.05	5.528	1.809	53.008	5.732	3.984	2.061
	0.005	3.984	1.091	42.683	6.814	5.366	1.449
	0.0005	44.634	2.937	3.740	1.128	44.715	3.042
	0.00005	44.715	3.042	3.821	0.843	45.691	2.104
	0.000005	48.618	5.782	4.309	1.786	48.699	6.601
	0.0000005	49.675	7.409	5.772	1.298	50.000	7.479
Xavier	0.5	44.715	3.042	46.992	7.281	4.472	0.909
	0.05	5.691	1.437	45.854	7.113	3.984	1.661
	0.005	4.390	1.937	44.634	3.154	4.065	2.370
	0.0005	5.610	2.360	4.146	0.782	5.772	1.505
	0.00005	31.789	2.325	3.333	1.532	34.228	2.625
	0.000005	44.959	2.751	4.228	2.548	45.041	2.687
	0.0000005	46.748	7.157	5.122	1.537	47.805	6.275

Tabla 5.3: *Grid search* utilizando la arquitectura VGG16, entrenamiento sobre *dataset* sin *hard negatives*.

Ini. de pesos	Tasa de aprendizaje	Optimizadores					
		SGD		Adam		Adadelta	
		Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)
He	0.5	44.715	3.042	52.439	5.126	44.715	3.042
	0.05	44.715	3.042	44.797	3.101	44.715	3.042
	0.005	44.715	3.042	44.715	3.042	44.715	3.042
	0.0005	44.715	3.042	44.715	3.042	48.537	7.062
	0.00005	48.618	6.938	3.577	0.668	50.569	5.436
	0.000005	50.650	5.451	5.772	2.020	50.813	5.806
	0.0000005	50.813	5.806	22.764	7.643	50.732	5.766
	0.00005	44.715	3.042	49.350	6.463	44.715	3.042
Xavier	0.05	7.398	1.872	44.715	3.042	3.577	1.849
	0.005	5.041	2.065	44.715	3.042	7.317	1.285
	0.0005	44.309	3.613	44.715	3.042	44.390	3.697
	0.00005	46.260	2.139	3.252	1.113	48.130	4.010
	0.000005	49.593	4.388	4.878	2.802	49.837	4.289
	0.0000005	49.919	4.267	6.667	2.328	50.163	4.514

Es curioso observar que, mientras en AlexNet 5.2 y VGG16 5.3 se reduce el error a menos del 5 % con ciertas combinaciones de hiperparámetros, en LeNet5 5.1 no ocurre esto. En concreto, uno de los mejores resultados que obtiene esta arquitectura es un 27.96 % de error con una desviación media del 14.73 % entre las 5 iteraciones de la validación cruzada. Tras revisar la implementación, no creemos que haya ningún error. ¿A qué puede deberse? ¿Puede ser que el modelo, por su pequeño tamaño, no tenga la capacidad suficiente para aprender este tipo de imágenes?

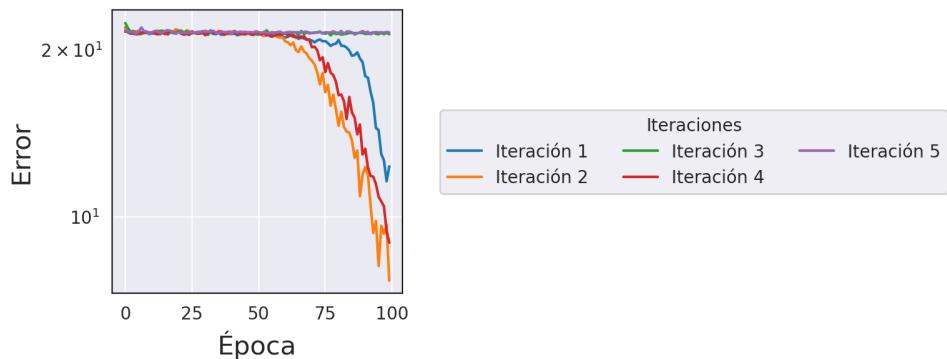


Figura 5.15: Variación del error durante el entrenamiento utilizando la configuración LeNet5-Xavier-0.05-SGD. Las iteraciones se corresponden con las producidas durante la validación cruzada.

Esta gráfica muestra la disminución del error durante el entrenamiento del modelo con la configuración de la que hablamos en el párrafo anterior. Se puede observar como decrece solo en 3 de las 5 iteraciones. Seguramente esta sea la causa de que exista una alta desviación de error entre iteraciones (14.73 %), ya que en algunas el algoritmo tiende a converger pero en otras no.

En la mayoría de configuraciones se observa o bien que el error no decrece o bien una gran diferencia entre iteraciones:

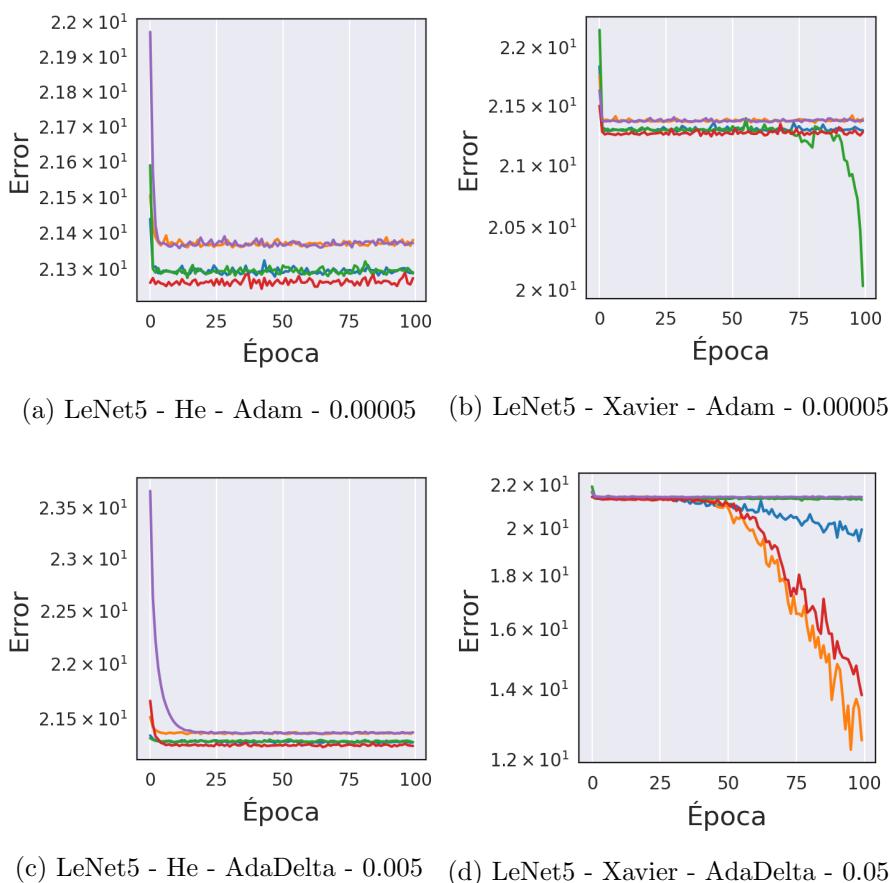


Figura 5.16: Más ejemplos de cómo varía el error durante el entrenamiento utilizando diferentes configuraciones sobre LeNet5.

Es probable que la arquitectura LeNet5 no tenga capacidad para modelar el problema debido a su pequeño tamaño. Vamos a realizar otro experimento, entrenar un modelo LeNet5 con un *dataset* reducido para ver si el error de entrenamiento decrece con mayor facilidad. Este será una muestra aleatoria del original.

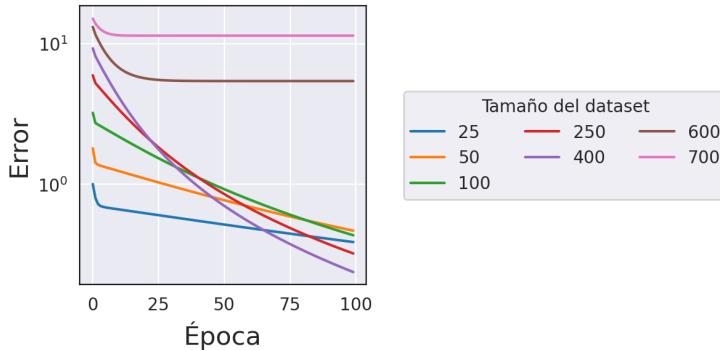


Figura 5.17: Modelos entrenados con la configuración LeNet5-He-Adam-0.00005, utilizando *datasets* de diferente tamaño (desde 25 hasta 700 imágenes).

Se observa como utilizando conjuntos de datos de entrenamiento de menor tamaño se consigue que el error decrezca con mayor facilidad. Hasta un *dataset* de tamaño 400, el error reduce de forma continua durante todo el entrenamiento, pero a partir de 600 el error se estanca. Introducir un mayor número de imágenes de entrenamiento genera mayor diversidad, que tiene que ser capturada por el modelo. Si el modelo es demasiado pequeño no será capaz de capturarla y por tanto el error no se reducirá mientras se entrena.

Por ello AlexNet y VGG16 serán las arquitecturas que tengamos en cuenta a la hora de elegir la configuración del modelo final. Entre ellas, el mejor resultado viene dado por VGG16, con la configuración VGG16-Xavier-Adam-0.00005. Se obtiene un error del 3.252 %, con una desviación entre iteraciones de la validación cruzada del 1.113 %. Vamos a graficar el error de entrenamiento en cada iteración:

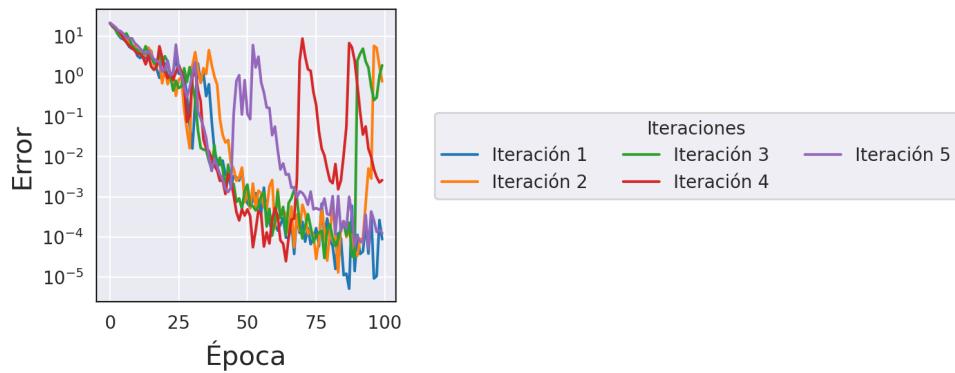


Figura 5.18: Evolución del error de *training* durante el entrenamiento de un modelo con configuración VGG16-Xavier-Adam-0.00005 sobre un *dataset* sin *hard negatives*.

Se observan muchos picos, en algunos casos ascendiendo al valor de error inicial. Esto puede deberse a diversos factores, como son la explosión del gradiente (producida por el valor que toma la tasa de aprendizaje o por su inadecuada actualización en el algoritmo de optimización) o el un tamaño de minibatch demasiado pequeño y que sea culpable de que el gradiente oscile. ¿Qué ocurre en otras configuraciones que también devuelven porcentajes de error razonables en las tablas 5.2 y 5.3?

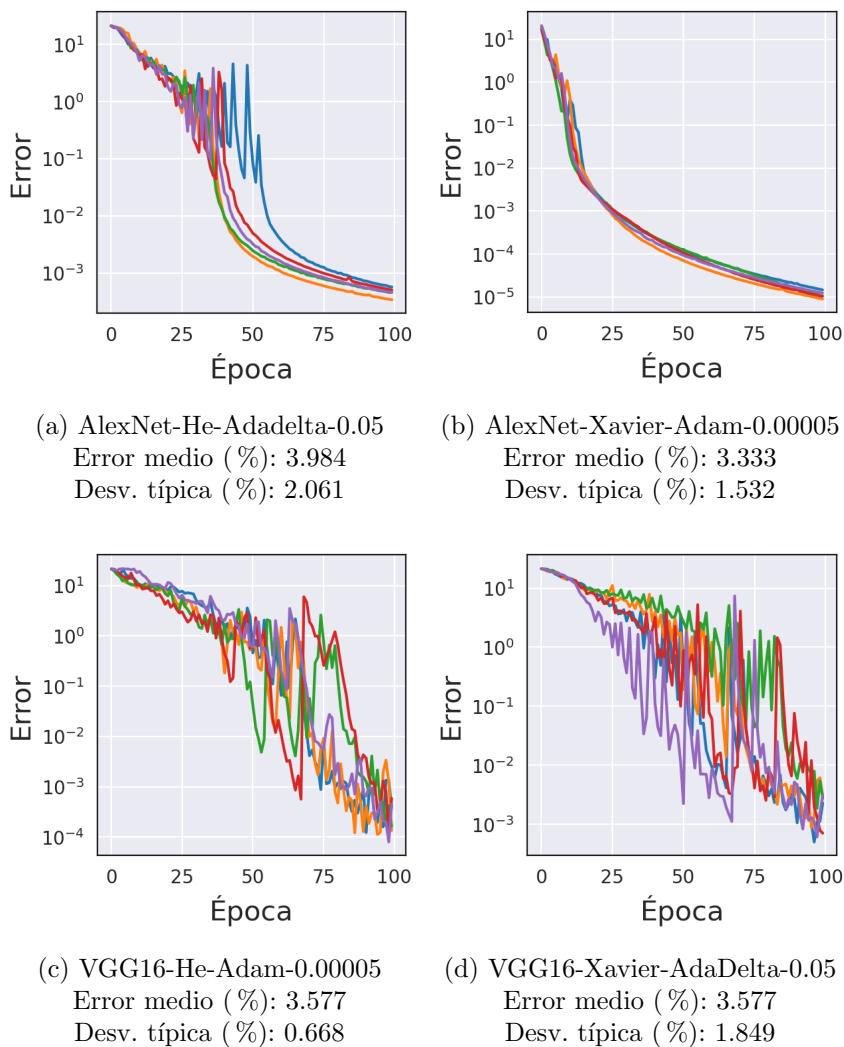


Figura 5.19: Más ejemplos de cómo varía el error durante el entrenamiento de diferentes modelos utilizando configuraciones basadas en LeNet5, sobre un *dataset* sin *hard negatives*.

Se observa como en los modelos basados en VGG16 existe una mayor oscilación del error. Se podría probar a utilizar un tamaño de minibatch mayor

(64, 128 o 256). Por otro lado, en los basados en AlexNet se observa una curva más suave. Por ello, ya que la diferencia de error es mínima entre todos estos modelos, vamos a utilizar como configuración para entrenar el modelo final AlexNet-Xavier-Adam-0.00005, cuya curva de error es muy suave.

El modelo final se entrenará sobre la partición *train* (de 1230 elementos, 85 % del dataset) y se dará una evaluación final sobre la *test* (de 218 elementos, 15 % del dataset), tal y como se representa en la figura 4.8.

Tabla 5.4: Porcentaje de error del modelo final sin *hard negatives*.

Modelo final	Error sobre la partición de <i>test</i>
AlexNet-Xavier-Adam-0.00005	2.752 %

Se mostró este resultado a los científicos israelíes y quedaron conformes. Durante el entrenamiento el error decreció de la siguiente forma:

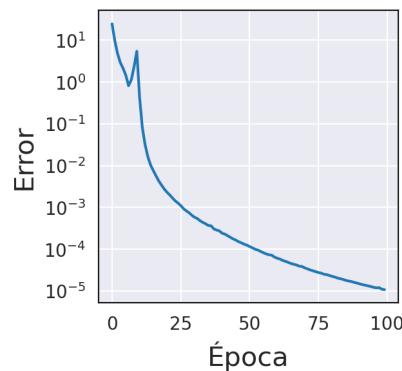


Figura 5.20: Evolución del error de *training* durante el entrenamiento del modelo final sobre el dataset sin *hard negatives*.

5.3.2. Clasificador sobre el *dataset con hard negatives*

Se realiza el mismo proceso que he seguido en la sección anterior, ahora utilizando el *dataset* que contiene un 50 % de ejemplos negativos originales y un 50 % sintéticos creados a partir de los modelos generadores.

Tabla 5.5: *Grid search* utilizando la arquitectura LeNet5, entrenamiento sobre *dataset* con *hard negatives*.

Ini. de pesos	Tasa de aprendizaje	Optimizadores					
		SGD		Adam		Adadelta	
		Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)
He	0.5	44.715	1.971	48.943	6.540	44.715	3.042
	0.05	48.130	6.308	44.715	3.042	44.715	3.042
	0.005	44.715	3.042	44.715	3.042	44.715	3.042
	0.0005	44.715	3.042	44.715	3.042	44.715	3.042
	0.00005	44.715	3.042	44.715	3.042	48.130	6.308
	0.000005	48.130	6.308	44.715	3.042	48.130	6.308
	0.0000005	48.130	6.308	48.130	6.308	48.130	6.308
	0.5	48.130	6.308	48.130	6.308	44.715	3.042
Xavier	0.05	48.130	6.308	44.715	3.042	45.610	2.545
	0.005	45.041	2.672	44.715	3.042	44.715	3.042
	0.0005	44.715	3.042	44.715	3.042	44.715	3.042
	0.00005	44.715	3.042	44.715	3.042	48.130	6.308
	0.000005	48.130	6.308	44.634	2.851	48.130	6.308
	0.0000005	48.130	6.308	44.715	3.042	48.130	6.308

Tabla 5.6: *Grid search* utilizando la arquitectura AlexNet, entrenamiento sobre *dataset* con *hard negatives*.

Ini. de pesos	Tasa de aprendizaje	Optimizadores					
		SGD		Adam		Adadelta	
		Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)
He	0.5	44.715	3.042	50.325	6.768	6.260	1.983
	0.05	5.854	1.240	47.805	6.521	5.203	0.530
	0.005	3.089	0.463	44.715	3.042	3.496	0.680
	0.0005	43.008	3.502	4.065	0.407	43.089	3.931
	0.00005	44.309	3.659	2.927	1.012	47.805	5.644
	0.000005	50.244	7.477	3.008	0.979	50.650	6.884
	0.0000005	51.626	5.603	5.041	1.426	51.951	5.169
	0.5	44.715	3.042	53.740	5.241	7.236	1.894
Xavier	0.05	6.016	1.128	52.927	5.851	4.309	1.537
	0.005	3.496	0.843	44.715	3.042	2.927	0.881
	0.0005	3.577	1.012	4.715	1.171	4.553	1.505
	0.00005	26.179	0.891	2.683	0.936	27.236	0.760
	0.000005	43.089	3.265	2.927	0.971	42.602	3.513
	0.0000005	47.561	6.630	4.228	1.305	47.805	6.540

Tabla 5.7: *Grid search* utilizando la arquitectura VGG16, entrenamiento sobre *dataset* con *hard negatives*.

		Optimizadores					
		SGD		Adam		Adadelta	
Ini. de pesos	Tasa de aprendizaje	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)	Error medio (%)	Desv. típica (%)
He	0.5	44.715	3.042	52.439	5.126	44.715	3.042
	0.05	44.715	3.042	44.634	3.154	44.715	3.042
	0.005	44.715	3.042	44.715	3.042	44.715	3.042
	0.0005	44.715	3.042	44.715	3.042	48.537	7.062
	0.00005	48.618	6.938	3.496	1.537	50.569	5.436
	0.000005	50.650	5.451	3.415	1.482	50.813	5.806
	0.0000005	50.813	5.806	26.748	1.611	50.732	5.766
Xavier	0.5	44.715	3.042	48.130	6.308	36.260	17.262
	0.05	37.236	18.044	44.715	3.042	4.390	0.971
	0.005	5.285	1.574	44.553	3.272	4.715	1.564
	0.0005	44.715	3.239	44.715	3.042	44.390	3.977
	0.00005	46.667	2.325	3.089	2.024	48.618	4.613
	0.000005	49.837	4.211	4.309	1.832	49.756	4.211
	0.0000005	50.163	4.152	10.407	4.132	50.325	4.267

Como se puede observar, los resultados son similares a lo que ocurre al no utilizar *hard negatives*. La arquitectura LeNet5 no tiene capacidad para modelar el problema de clasificación binaria, obteniendo unos porcentajes de error en torno al 40 %-50 %. AlexNet y VGG16 son modelos más modernos y con mayor profundidad, lo que les permite descubrir y modelar relaciones más complejas en las imágenes, y tomar decisiones de clasificación a partir de estas.

Recordar que estos resultados se obtienen a partir de aplicar validación cruzada (ver figura 4.8), de forma que el error medio se calcula como la media de error de las 5 iteraciones que la componen y la desviación es la existente entre iteraciones.

Entrenando sobre este *dataset*, la configuración AlexNet-Xavier-Adam-0.00005 es la que mejores resultados consigue, un 2.683 % de error con una desviación típica del 0.936 % entre iteraciones de la validación cruzada. Si graficamos cómo varía el error de *training* durante el entrenamiento del modelo obtenemos lo siguiente:

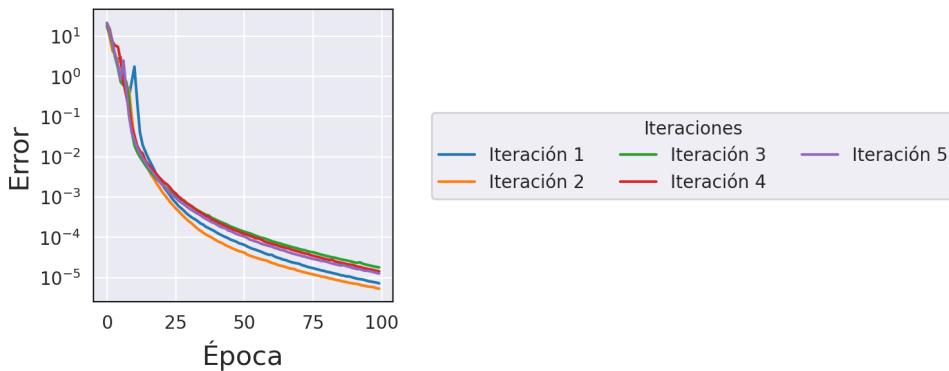


Figura 5.21: Evolución del error de *training* durante el entrenamiento de un modelo con configuración AlexNet-Xavier-Adam-0.00005 sobre un *dataset* con *hard negatives*.

Se observa como el error decrece suavemente. Elijo esta configuración y se procede a entrenar el modelo final, obteniendo los siguientes resultados:

Tabla 5.8: Porcentaje de error del modelo final con *hard negatives*.

Modelo final	Error sobre la partición de <i>test</i>
AlexNet-Xavier-Adam-0.00005	0.917 %

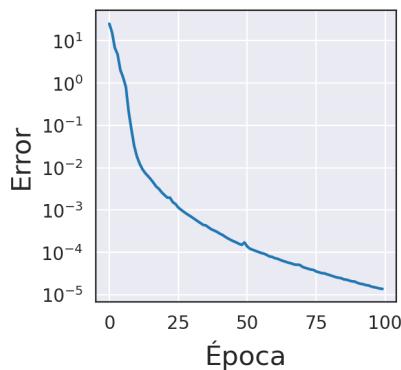


Figura 5.22: Evolución del error de *training* durante el entrenamiento del modelo final sobre el *dataset* con *hard negatives*.

Da la casualidad de que sobre ambos *datasets*, con y sin *hard negatives*, el modelo escogido utiliza la misma configuración. Entregaremos a los científicos de la Universidad de Negev ambos modelos.

Capítulo 6

Conclusiones

Comentar la configuración final y si se han cumplido los objetivos del proyecto.

Bibliografía

- [1] David Weininger. “SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules”. *Journal of chemical information and computer sciences* 28.1 (1988), págs. 31-36.
- [2] Arthur Dalby y col. “Description of several chemical structure file formats used by computer programs developed at Molecular Design Limited”. *Journal of Chemical Information and Computer Sciences* 32.3 (1992), págs. 244-255. DOI: 10.1021/ci00007a012. eprint: <https://doi.org/10.1021/ci00007a012>. URL: <https://doi.org/10.1021/ci00007a012>.
- [3] Ken Schwaber. “Scrum development process”. *Business object design and implementation*. Springer, 1997, págs. 117-134.
- [4] Yann LeCun y col. “Gradient-based learning applied to document recognition”. *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324.
- [5] Thomas Engel. “Basic Overview of Chemoinformatics”. *Journal of Chemical Information and Modeling* 46.6 (2006). PMID: 17125169, págs. 2267-2277. DOI: 10.1021/ci600234z. eprint: <https://doi.org/10.1021/ci600234z>. URL: <https://doi.org/10.1021/ci600234z>.
- [6] Alex Krizhevsky, Ilya Sutskever y Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. *Advances in neural information processing systems* 25 (2012).
- [7] Atima Tharatipyakul y col. “ChemEx: information extraction system for chemical data curation”. *BMC bioinformatics*. Vol. 13. 17. Springer. 2012, págs. 1-11.
- [8] Mihai Liviu Despa. “Comparative study on software development methodologies”. *Database Systems Journal* 5.3 (2014), págs. 37-56.
- [9] Karen Simonyan y Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: <https://arxiv.org/abs/1409.1556>.
- [10] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.

- [11] Nikhil Ketkar. “Introduction to deep learning”. *Deep learning with Python*. Springer, 2017, págs. 1-5.
- [12] Ashish Vaswani y col. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].
- [13] Fernando Berzal. *Redes neuronales & deep learning*. <https://deeplearning.ikor.org>. Independently published, 2018.
- [14] Mark Chen y col. “Generative pretraining from pixels”. *International Conference on Machine Learning*. PMLR. 2020, págs. 1691-1703.
- [15] Mario Krenn y col. “Self-referencing embedded strings (SELFIES): A 100 % robust molecular string representation”. *Machine Learning: Science and Technology* 1.4 (oct. de 2020), pág. 045024. DOI: 10.1088/2632-2153/aba947. URL: <https://doi.org/10.1088%2F2632-2153%2Faba947>.
- [16] Martijn Oldenhof y col. “ChemGrapher: optical graph recognition of chemical compounds by deep learning”. *Journal of chemical information and modeling* 60.10 (2020), págs. 4506-4517.
- [17] Kohulan Rajan, Achim Zielesny y Christoph Steinbeck. “DECIMER: towards deep learning for chemical image recognition”. *Journal of Cheminformatics* 12.1 (2020), págs. 1-9.
- [18] Patrick Esser, Robin Rombach y Bjorn Ommer. “Taming transformers for high-resolution image synthesis”. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, págs. 12873-12883.

Otras fuentes

- [19] Robert Belford. *Anatomy of a MOL file*. URL: [https://chem.libretexts.org/Courses/University_of_Arkansas_Little_Rock/ChemInformatics_\(2017\)%3A_Chem_4399_5399/2.2%3A_Chemical_Representations_on_Computer%3A_Part_II/2.2.2%3A_Anatomy_of_a_MOL_file](https://chem.libretexts.org/Courses/University_of_Arkansas_Little_Rock/ChemInformatics_(2017)%3A_Chem_4399_5399/2.2%3A_Chemical_Representations_on_Computer%3A_Part_II/2.2.2%3A_Anatomy_of_a_MOL_file) (visitado 26-04-2022).
- [20] curiosoando.com. *¿Qué es la aromaticidad?* URL: <https://curiosoando.com/que-es-la-aromaticidad> (visitado 25-04-2022).
- [21] dias-laborables.es. *¿Cuántos días laborables en el año 2022?* URL: https://www.dias-laborables.es/cuantos_dias_laborables_en_anio_2022_Aandaluc%C3%ADa.htm (visitado 25-05-2022).
- [22] Patrick Esser, Robin Rombach y Bjorn Ommer. *Taming Transformers for High-Resolution Image Synthesis - GitHub*. URL: <https://github.com/CompVis/taming-transformers> (visitado 23-02-2022).
- [23] Universidad Europea. *Cuánto gana un ingeniero informático*. URL: <https://universidadeuropea.com/blog/cuanto-gana-un-ingenero-informatico/> (visitado 25-05-2022).
- [24] Giuliano Giacaglia. *How Transformers Work*. URL: <https://towardsdatascience.com/transformers-141e32e69591> (visitado 04-06-2022).
- [25] Rocío González. *¡Comienza el verano! Cuántos días de vacaciones me corresponden*. URL: <https://www.sage.com/es-es/blog/dias-vacaciones/> (visitado 25-05-2022).
- [26] National Institute of Health (NIH). *PubChem Docs - About PubChem*. URL: <https://pubchemdocs.ncbi.nlm.nih.gov/about> (visitado 26-04-2022).
- [27] *Imgaug documentation*. URL: <https://imgaug.readthedocs.io/en/latest/> (visitado 04-06-2022).
- [28] Raimi Karim. *Illustrated: Self-Attention*. URL: <https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a> (visitado 04-06-2022).

- [29] Kiprono Elijah Koech. *Cross-Entropy Loss Function*. URL: <https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e> (visitado 23-06-2022).
- [30] Geometric Computing Laboratory. *Introduction to Computer Graphics - EPFL*. URL: <https://github.com/CompVis/taming-transformers>.
- [31] Pedro Bedmar López. *TFG GitHub repository*. URL: <https://github.com/pbedmar/tfg> (visitado 03-06-2022).
- [32] Pytorch documentation. URL: <https://pytorch.org/docs/stable/index.html> (visitado 04-06-2022).
- [33] Charlie Snell. *Understanding VQ-VAE (DALL-E Explained Pt. 1)*. URL: <https://ml.berkeley.edu/blog/posts/vq-vae/> (visitado 04-06-2022).
- [34] Ayush Thakur. *What's the Optimal Batch Size to Train a Neural Network?* URL: <https://wandb.ai/ayush-thakur/dl-question-bank/reports/What-s-the-Optimal-Batch-Size-to-Train-a-Neural-Network---Vm1ldzoyMDkyNDU> (visitado 26-05-2022).
- [35] Santander Universidades. *Metodologías de desarrollo de software: ¿qué son?* URL: <https://www.becas-santander.com/es/blog/metodologias-desarrollo-software.html> (visitado 04-03-2022).
- [36] Aditya Virani y col. *Structural Representations of Organic Compounds*. URL: <https://brilliant.org/wiki/structural-representations-of-organic-compounds/> (visitado 21-04-2022).
- [37] Wikipedia. *Aromaticidad*. URL: <https://es.wikipedia.org/wiki/Aromaticidad> (visitado 25-04-2022).
- [38] Wikipedia. *Compuesto orgánico*. URL: https://es.wikipedia.org/wiki/Compuesto_org%C3%A1nico (visitado 21-04-2022).
- [39] Wikipedia. *Compuesto organometálico*. URL: https://es.wikipedia.org/wiki/Compuesto_organomet%C3%A1lico (visitado 21-04-2022).
- [40] Wikipedia. *Estereoquímica*. URL: <https://es.wikipedia.org/wiki/Estereoqu%C3%ADmica> (visitado 21-04-2022).
- [41] Wikipedia. *Fórmula estructural*. URL: https://es.wikipedia.org/wiki/F%C3%B3rmula_structural (visitado 21-04-2022).
- [42] Wikipedia. *Simplified molecular-input line-entry system*. URL: https://en.wikipedia.org/wiki/Simplified_molecular_input_line-entry_system (visitado 24-04-2022).