UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ciencia e Ingeniería de Datos

PROYECTO

Desarrollo de un Paquete de IA Generativa de Imágenes

Aprendizaje Automático 3

Autor: Héctor Tablero Díaz, Álvaro Martínez Gamo
Tutor: Alberto Suárez González

mayo 2025

Héctor Tablero Díaz, Álvaro Martínez Gamo
Desarrollo de un Paquete de IA Generativa de Imágenes

Héctor Tablero Díaz, Álvaro Martínez Gamo

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Este trabajo presenta el desarrollo de un paquete de software en Python para la generación de imágenes mediante modelos de difusión. Los modelos de difusión representan una clase emergente de modelos generativos que han demostrado resultados excepcionales en la creación de imágenes de alta calidad, superando en muchos aspectos a las técnicas previas como las Redes Generativas Adversarias (GANs).

Nuestro paquete implementa tres variantes principales de procesos de difusión: Variance Exploding (VE), Variance Preserving (VP) y Sub-Variance Preserving (Sub-VP), cada uno con características distintivas para diferentes casos de uso. Para el proceso de muestreo, proporcionamos cuatro implementaciones: Euler-Maruyama, Predictor-Corrector, Probability Flow ODE y Exponential Integrator, permitiendo equilibrar entre velocidad y calidad según los requisitos específicos. Complementariamente, incluimos dos programadores de ruido (lineal y coseno) que controlan la adición de ruido durante el proceso de difusión.

Una característica destacada del paquete es su capacidad para generación controlable, incluyendo colorización de imágenes en escala de grises, imputación de regiones faltantes y generación condicionada por clase. Estas funcionalidades amplían significativamente el ámbito de aplicaciones prácticas, desde restauración de imágenes hasta creación de contenido específico.

Para garantizar la usabilidad, diseñamos una interfaz de programación intuitiva complementada con un dashboard interactivo, facilitando tanto el uso programático como la experimentación visual. El paquete incluye también métricas estándar (BPD, FID, IS) para la evaluación cuantitativa de los resultados.

Los experimentos realizados demuestran la efectividad de nuestra implementación en diversas tareas generativas, produciendo imágenes de alta calidad incluso con modelos entrenados sobre conjuntos limitados de datos. El código está estructurado de forma modular y extensible, facilitando la incorporación de nuevas funcionalidades y la adaptación a casos de uso específicos.

El sistema incluye herramientas innovadoras de serialización segura, dashboard interactivo y documentación auto-generada, superando los requisitos básicos.

Índice

1 Introducción **1**

1.0.1 Motivación . . . 1

1.0.2 Objetivos . . . 2

1.0.3 Estructura del Documento . . . 2

1.1 Estado del Arte . . . 3

1.1.1 Modelos de Difusión . . . 3

1.1.2 Colorización sin Reentrenamiento . . . 3

1.1.3 Tendencias en Imputación de Imágenes . . . 4

1.1.4 Generación de Texto a Imagen . . . 4

2 Desarrollo **5**

2.1 Planificación del Trabajo . . . 6

2.2 Análisis de Requisitos . . . 6

2.2.1 Requisitos Funcionales . . . 6

2.2.2 Requisitos No Funcionales . . . 7

2.2.3 Casos de Uso . . . 7

2.3 Diseño . . . 7

2.3.1 Patrones de Diseño . . . 8

2.3.2 Arquitectura Modular . . . 8

2.3.3 Interfaz de Usuario . . . 8

2.3.4 Sistema de Extensión . . . 9

2.3.5 Estructura del Paquete . . . 9

2.3.6 Diagrama de Clases . . . 10

2.4 Validación y Pruebas . . . 12

2.4.1 Estrategia de Pruebas . . . 12

2.4.2 Compatibilidad y Entornos . . . 14

2.5 Garantía de Calidad de Software . . . 14

2.5.1 Control de Versiones . . . 14

2.5.2 Estándares de Codificación . . . 15

2.5.3 Documentación . . . 15

2.5.4 Gestión de Dependencias . . . 15

2.6 Otras Cuestiones . . . 16

3 Resultados **17**

3.1 Ejemplos de Uso . . . 18
3.1.1 Generación Básica de Imágenes . . . 18
3.1.2 Colorización de Imágenes . . . 19
3.1.3 Imputación de Regiones . . . 19
3.2 Conclusiones . . . 20
3.2.1 Logros y Contribuciones . . . 20
3.2.2 Limitaciones . . . 20
3.2.3 Trabajo Futuro . . . 21
3.2.4 Consideraciones Finales . . . 21
Bibliografía **24**
Apéndices **25**
A Material Técnico **27**
A.1 Ecuaciones Diferenciales Estocásticas en Modelos de Difusión . . . 27
A.1.1 SDE Directa . . . 27
A.1.2 SDE Inversa . . . 28
A.1.3 Demostración de la Derivación de la SDE Inversa . . . 28
A.2 Métodos de Muestreo . . . 29
A.2.1 Euler-Maruyama . . . 29
A.2.2 Integrador Exponencial . . . 29
A.2.3 Flujo de Probabilidad ODE . . . 31
A.2.4 Predictor-Corrector . . . 31
A.3 Programadores de Ruido . . . 32
A.3.1 Programador Lineal . . . 32
A.3.2 Programador Coseno . . . 33
A.4 Generación Controlable . . . 34
A.4.1 Colorización mediante Condicionamiento de Luminancia . . . 34
A.4.2 Imputación mediante Mascarado . . . 34
A.4.3 Generación Condicionada por Clase . . . 34
A.5 Paralelización y Optimizaciones . . . 35
A.5.1 Detalles de Optimización . . . 36
B Ejemplos Adicionales **37**
B.1 Efectos de los Programadores de Ruido . . . 37
B.2 Más Ejemplos . . . 38
C Comparaciones Exhaustivas **41**
C.1 Comparación de Arquitecturas de Difusión . . . 41

C.2 Comparación de Muestreadores 42
C.3 Comparación entre Generación Condicional e Incondicional 42
C.4 Análisis de Tareas de Imputación 43
C.5 Análisis de Colorización 45
C.6 Comparación entre Conjuntos de Datos Completos y Parciales 46
C.7 Análisis de la Métrica BPD 46
C.8 Efecto del Muestreador en el Tiempo de Ejecución 47
C.9 Comparativa de Métodos por Tarea 48
C.10 Conclusiones 49

1

Introducción

La generación de imágenes mediante aprendizaje automático representa uno de los campos más fascinantes y de rápido desarrollo dentro de la inteligencia artificial. En los últimos años, hemos presenciado avances significativos que han transformado nuestra capacidad para crear contenido visual de alta calidad de manera automática.

En este contexto, los modelos de difusión han emergido como un paradigma particularmente prometedor, ofreciendo ventajas sustanciales respecto a enfoques anteriores como las Redes Generativas Adversarias (GANs). Su fundamento teórico en procesos estocásticos bien establecidos proporciona un marco elegante para la generación de imágenes, con un entrenamiento más estable y resultados de alta fidelidad.

Este proyecto se enfoca en el desarrollo de un paquete de software en Python para la generación de imágenes basada en difusión, que implementa diferentes variantes de estos procesos y proporciona flexibilidad en cuanto a los métodos de muestreo, programación de ruido y tareas de generación controlable. Nuestra implementación permite no solo la generación de imágenes desde ruido aleatorio, sino también tareas más específicas como la colorización de imágenes en escala de grises, la imputación de regiones faltantes y la generación condicionada por clase.

El paquete ha sido diseñado con un énfasis en la modularidad y extensibilidad, permitiendo a los usuarios adaptar cada componente según sus necesidades específicas. Además, incluye métricas de evaluación comúnmente utilizadas en el campo para facilitar la comparación entre diferentes configuraciones y con otros métodos de generación.

1.0.1. Motivación

La motivación principal para desarrollar este paquete surge de la necesidad de contar con herramientas accesibles y flexibles para la investigación y aplicación de modelos de difusión. Si bien existen implementaciones de referencia para modelos específicos, consideramos valioso proporcionar una biblioteca que permita experimentar con diferentes configuraciones y comparar su rendimiento de manera sistemática.

Además, la incorporación de capacidades de generación controlable responde a la creciente demanda de métodos que permitan ejercer un mayor control sobre el proceso generativo, facilitando su aplicación en contextos donde se requiere respetar ciertas restricciones o características específicas en las imágenes generadas.

Otra motivación principal es facilitar la investigación con componentes personalizados, simplificando el proceso de compartir modelos entre equipos. Nuestro sistema serializa tanto los parámetros entrenados como las definiciones de clases necesarias, eliminando la necesidad de compartir manualmente el código fuente o mantener dependencias exactas entre entornos. Para garantizar seguridad, implementamos un mecanismo de carga opcional que requiere confirmación explícita del usuario y ejecuta el código en un entorno restringido, permitiendo así la colaboración sin comprometer la seguridad de los sistemas.

1.0.2. Objetivos

Los objetivos principales de este proyecto son:

- Desarrollar una biblioteca modular y extensible para la generación de imágenes mediante modelos de difusión.
- Implementar diferentes variantes de procesos de difusión, métodos de muestreo y programaciones de ruido.
- Incorporar funcionalidades para la generación controlable, incluyendo colorización, imputación y condicionamiento por clase.
- Proporcionar métricas estándar para la evaluación de la calidad de las imágenes generadas.
- Crear una interfaz de usuario interactiva que facilite la experimentación con los diferentes componentes del paquete.
- Documentar exhaustivamente tanto el código como los fundamentos teóricos subyacentes.
- Implementar un sistema de serialización segura para la carga y guardado de clases personalizadas.

1.0.3. Estructura del Documento

El resto de este documento está organizado de la siguiente manera:

El Capítulo 2 presenta el desarrollo del software, incluyendo la planificación del trabajo, el análisis de requisitos, el diseño del paquete y las pruebas realizadas.

El Capítulo 3 describe los resultados obtenidos, incluyendo ejemplos de uso del paquete y las conclusiones del proyecto.

Finalmente, se incluyen varios apéndices con material técnico adicional, ejemplos complementarios y comparaciones exhaustivas entre diferentes configuraciones del paquete.

1.1. Estado del Arte

El campo de la generación de imágenes mediante técnicas de aprendizaje automático ha avanzado significativamente en los últimos años, destacando especialmente los modelos de difusión. Estos modelos han demostrado ser una alternativa prometedora a las Redes Generativas Adversarias (GANs) y los modelos autorregresivos, ofreciendo muestras de alta calidad con un entrenamiento más estable.

1.1.1. Modelos de Difusión

Los modelos de difusión [1], [2] se basan en un proceso que añade ruido gradualmente a los datos y luego aprende a revertir este proceso. Este enfoque proporciona un marco teóricamente fundamentado en las ecuaciones diferenciales estocásticas (SDEs) [3], donde el proceso de añadir ruido define una trayectoria desde los datos originales hasta ruido puro, y el proceso inverso permite generar nuevas muestras.

Entre las implementaciones más destacadas se encuentra **Stable Diffusion** [4], que opera en un espacio latente comprimido en lugar del espacio de píxeles completo, lo que reduce significativamente los requisitos computacionales. Este enfoque permite generar imágenes de alta resolución mientras mantiene la estabilidad del entrenamiento, utilizando un autoencoder variacional (VAE) para comprimir y descomprimir la representación antes y después del proceso de difusión.

1.1.2. Colorización sin Reentrenamiento

Un avance notable en el campo es **CGDiff** (Color-Guided Diffusion) [5], un método que permite colorear imágenes en escala de grises sin necesidad de reentrenar el modelo. Esta técnica se basa en la manipulación del espacio latente del modelo de difusión, guiando el proceso de generación para preservar la información estructural de la imagen original mientras añade información cromática plausible.

El enfoque de CGDiff es particularmente interesante porque demuestra la flexibilidad de los modelos de difusión para tareas de generación condicional, aprovechando el conocimiento ya adquirido por el modelo sobre la distribución de imágenes naturales.

1.1.3. Tendencias en Imputación de Imágenes

La imputación en imágenes, también conocida como inpainting, busca completar regiones faltantes de manera coherente con el contenido circundante. Las técnicas más recientes basadas en difusión [6], [7] han superado a métodos anteriores gracias a su capacidad para generar completados más coherentes y detallados.

El estado actual de la investigación se centra en técnicas de muestreo guiado [8], donde se utiliza información adicional para condicionar el proceso de generación, permitiendo un control preciso sobre el contenido generado. Estas técnicas permiten preservar detalles específicos mientras se completan las regiones faltantes, manteniendo la coherencia global de la imagen.

1.1.4. Generación de Texto a Imagen

Aunque no es el enfoque principal de nuestro proyecto, no podemos dejar de mencionar los avances en modelos de texto a imagen como DALL-E 2 [9] y Imagen [10], que han demostrado la capacidad de los modelos de difusión para generar imágenes detalladas a partir de descripciones textuales. Estos modelos incorporan representaciones de texto aprendidas por modelos como CLIP o T5, que permiten alinear el espacio semántico del texto con el espacio visual de las imágenes.

La relevancia de estos avances para nuestro trabajo radica en las técnicas de condicionamiento que emplean, que en muchos casos son generalizables a otros tipos de condicionamiento, como la generación basada en clases que implementamos en nuestro proyecto.

2

Desarrollo

En este capítulo se detalla el proceso de desarrollo del paquete de software para la generación de imágenes mediante modelos de difusión. Se abordan las distintas fases del desarrollo, desde la planificación inicial hasta la validación y pruebas, pasando por el análisis de requisitos y el diseño de la arquitectura del sistema.

El desarrollo se ha llevado a cabo siguiendo un enfoque iterativo e incremental, priorizando la modularidad y extensibilidad del código. Esto ha permitido implementar progresivamente las diferentes funcionalidades y componentes del sistema, facilitando la integración continua y las pruebas unitarias.

La metodología empleada ha puesto especial énfasis en la calidad del código, la documentación exhaustiva y la facilidad de uso, tanto para usuarios finales como para desarrolladores que deseen extender o modificar el sistema. A lo largo del proceso, se han utilizado herramientas de control de versiones, pruebas automatizadas y documentación integrada para garantizar la robustez y mantenibilidad del software.

En las secciones siguientes se describen detalladamente cada uno de los aspectos del desarrollo, desde la planificación hasta la validación, incluyendo consideraciones sobre la garantía de calidad del software y otras cuestiones relevantes.

2.1. Planificación del Trabajo

Para llevar a cabo el trabajo, hemos planificado y dividido las tareas de forma equitativa con el apoyo de un diagrama de Gantt para seguir aproximadamente los plazos, teniendo en cuenta que por motivos logísticos disponemos de menos tiempo del programado para finalizarlo.

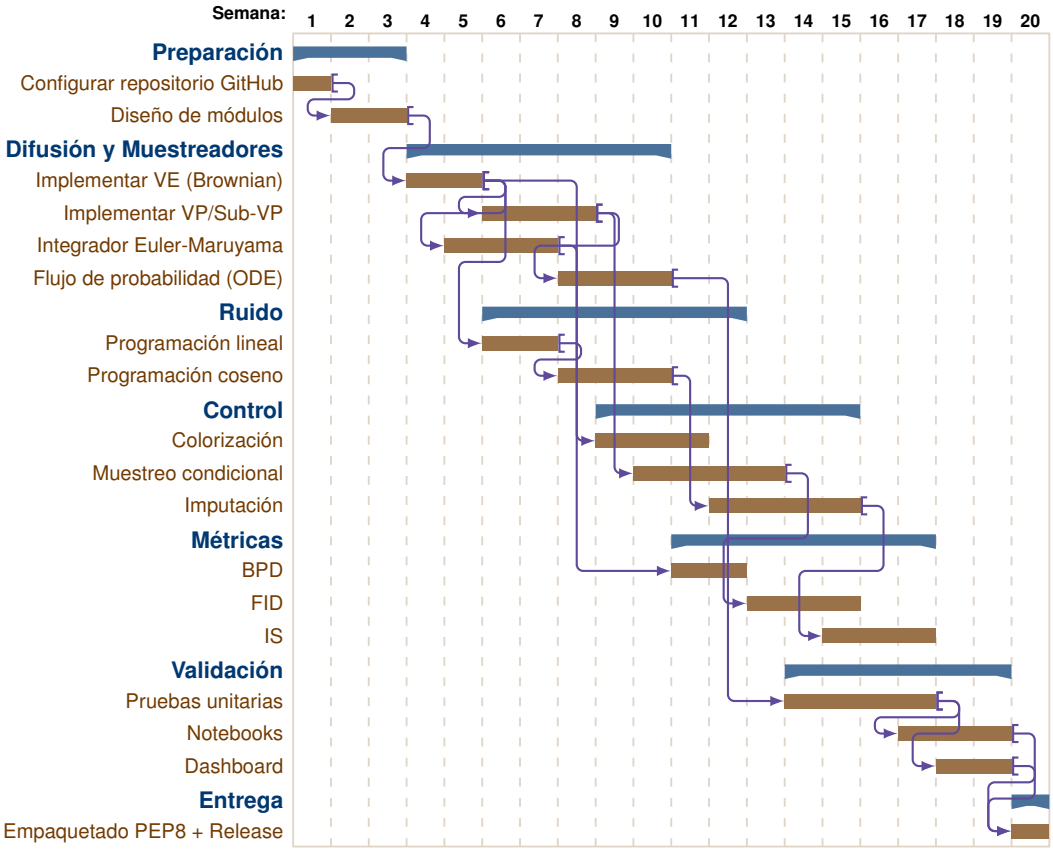

Figura 2.1: Diagrama de Gantt en el que se muestra la distribución diaria del tiempo estimada.

Aunque esta era nuestra idea original, finalmente no hemos podido cuplir con los plazos establecidos, y ha sido necesario usar los 50 días otorgados para la realización del proyecto. Si bien la mayoría de clases estaban terminadas según el díagrama, el trabajo extra, así como el ajuste de parámetros predeterminados y la corrección de bugs llevaron mucho más tiempo del esperado.

2.2. Análisis de Requisitos

2.2.1. Requisitos Funcionales

El software debe proporcionar las siguientes funcionalidades:

RF-1. Permitir la generación de imágenes mediante modelos de difusión.

- **RF-1.1.** Soportar la generación de imágenes a partir de ruido aleatorio.
- **RF-1.2.** Facilitar la imputación de regiones faltantes en imágenes parciales.
- **RF-1.3.** Incluir la opción de generación condicionada a una clase específica.

RF-2. Ofrecer configuraciones para controlar los parámetros del proceso generativo.

RF-3. Soportar distintos métodos de muestreo para la generación de imágenes.

RF-4. Incluir métricas para evaluar la calidad de las imágenes generadas.

RF-5. Proporcionar una interfaz de uso sencilla mediante una API en Python.

RF-6. Permitir la integración con cuadernos Jupyter para demostraciones y pruebas.

2.2.2. Requisitos No Funcionales

Estas son las restricciones a las que está sujeto el software:

RNF-1. El tiempo de generación de una imagen de 32×32 píxeles no debe superar:

- **RNF-1.1.** 5 segundos al ejecutarse en una GPU con CUDA.
- **RNF-1.2.** 10 segundos al ejecutarse en una CPU de al menos 4 cores.

RNF-2. El consumo máximo de memoria RAM no debe superar los 8 GB durante la ejecución estándar del modelo.

RNF-3. El código debe seguir las guías de estilo de Google para garantizar su claridad y mantenibilidad.

RNF-4. La documentación debe incluir ejemplos de uso para facilitar la adopción del software.

RNF-5. El sistema debe ser compatible con Python 3.9 o superior y utilizar librerías estándar de aprendizaje automático, incluyendo PyTorch $\geq$ 2.0.0.

2.2.3. Casos de Uso

Los casos de uso describen cómo un usuario interactuaría con el sistema:

CU-1. Generación de imágenes desde ruido aleatorio: El usuario configura parámetros como tamaño y método de muestreo. El sistema genera la imagen y la devuelve en un formato compatible (png, jpg).

CU-2. Imputación de regiones faltantes en imágenes parciales: El usuario proporciona una imagen con áreas faltantes. El sistema las completa basándose en el contenido circundante y devuelve la imagen reconstruida.

CU-3. Generación de imágenes condicionada a una clase: El usuario selecciona una categoría (ej. "perro" o "gato"). El sistema genera la imagen correspondiente y la devuelve en un formato compatible.

2.3. Diseño

El diseño del módulo de Python se ha realizado con un enfoque centrado en la ampliabilidad y modularidad, siguiendo los principios de diseño orientado a objetos y programación por interfaces. La arquitectura del sistema está basada en clases abstractas que definen interfaces, junto con implemen-

taciones concretas que proporcionan comportamientos específicos.

Esta estructura facilita tanto el uso básico del paquete para usuarios con necesidades estándar, como la extensión y personalización para usuarios avanzados que requieran comportamientos específicos. Un ejemplo de este enfoque puede verse en el notebook 'samplers.ipynb', donde se demuestra cómo utilizar y combinar diferentes implementaciones de muestreadores.

2.3.1. Patrones de Diseño

Se han aplicado varios patrones de diseño para mejorar la estructura y flexibilidad del código:

- **Patrón Strategy**: Utilizado en los diferentes difusores, muestreadores y programadores de ruido, permitiendo intercambiar componentes con facilidad.
- **Patrón Factory**: Implementado en la clase `GenerativeModel` para crear instancias de los diferentes componentes basándose en parámetros de configuración.
- **Patrón Observer**: Utilizado para el seguimiento del progreso durante la generación de imágenes.

2.3.2. Arquitectura Modular

La arquitectura del sistema se ha diseñado para maximizar la cohesión dentro de cada módulo y minimizar el acoplamiento entre módulos. Cada componente tiene una responsabilidad clara y bien definida:

- **Módulo de Difusión**: Encapsula los algoritmos que definen cómo se añade y elimina el ruido durante el proceso de difusión.
- **Módulo de Muestreo**: Implementa diferentes estrategias para resolver numéricamente las ecuaciones diferenciales estocásticas del proceso de difusión.
- **Módulo de Programación de Ruido**: Define cómo se distribuye el ruido a lo largo del proceso de difusión.
- **Módulo de Métricas**: Proporciona herramientas para evaluar la calidad de las imágenes generadas.

2.3.3. Interfaz de Usuario

El paquete proporciona dos interfaces principales:

- **API Programática**: Una interfaz en Python que permite acceder a todas las funcionalidades del paquete a través de código.
- **Dashboard Interactivo**: Una interfaz gráfica basada en Streamlit que facilita la experimentación y demostración de las capacidades del sistema sin necesidad de escribir código. Además de la versión local, se puede acceder a una versión en línea (sin GPU) a través de `https://image-gen-htd.streamlit.app/`.

El dashboard representa una contribución significativa a la usabilidad del sistema, permitiendo a usuarios con distintos niveles de experiencia técnica interactuar con los modelos de difusión de manera intuitiva.

2.3.4. Sistema de Extensión

Una característica distintiva del diseño es el sistema de carga dinámica de clases personalizadas, que permite a los usuarios extender el comportamiento del sistema sin modificar el código base. Esta funcionalidad se implementa a través de la clase `CustomClassWrapper`, que proporciona mecanismos seguros para la carga y ejecución de código definido por el usuario.

Esta clase es gestionada internamente por el sistema, y los usuarios pueden cargar modelos con clases personalizadas como si estuvieran cargando uno normal, mejorando la experiencia de uso y permitiendo una mayor flexibilidad en la personalización del sistema.

2.3.5. Estructura del Paquete

En cuanto a la estructura de archivos, el código del módulo se ha distribuido por subcarpetas de la siguiente manera:

```
diffusion/
├── base.py
├── ve.py
├── vp.py
└── sub_vp.py
metrics/
├── base.py
├── bpd.py
├── fid.py
└── inception.py
noise/
├── base.py
├── linear.py
└── cosine.py
```

```
├── samplers/
│   ├── base.py
│   ├── euler_maruyama.py
│   ├── exponential.py
│   ├── ode.py
│   └── predictor_corrector.py
├── base.py
├── score_model.py
├── visualization.py
└── utils.py
```

Adicionalmente el proyecto cuenta con código para la generación de un dashboard interactivo, con tests, con documentación, y con una carpeta adicional que contiene los notebooks de ejemplos. La estructura de código completa quedaría así:

```
├── dashboard/
│   └── (estilos e idiomas)
├── .streamlit/
│   └── config.toml
├── dashboard.py
├── examples/
│   ├── class_conditioning.ipynb
│   ├── colorization.ipynb
│   ├── diffusers.ipynb
│   ├── evaluation.ipynb
│   ├── getting_started.ipynb
│   ├── imputation.ipynb
│   ├── noise_schedulers.ipynb
│   └── samplers.ipynb
├── docs/
│   └── (markdown con documentación)
└── tests/
    └── (varios tests)
```

2.3.6. Diagrama de Clases

La arquitectura de nuestro paquete está diseñada siguiendo los principios de programación orientada a objetos y de diseño modular. El diagrama de clases presentado en la Figura 2.2 muestra las relaciones entre los principales componentes del sistema.

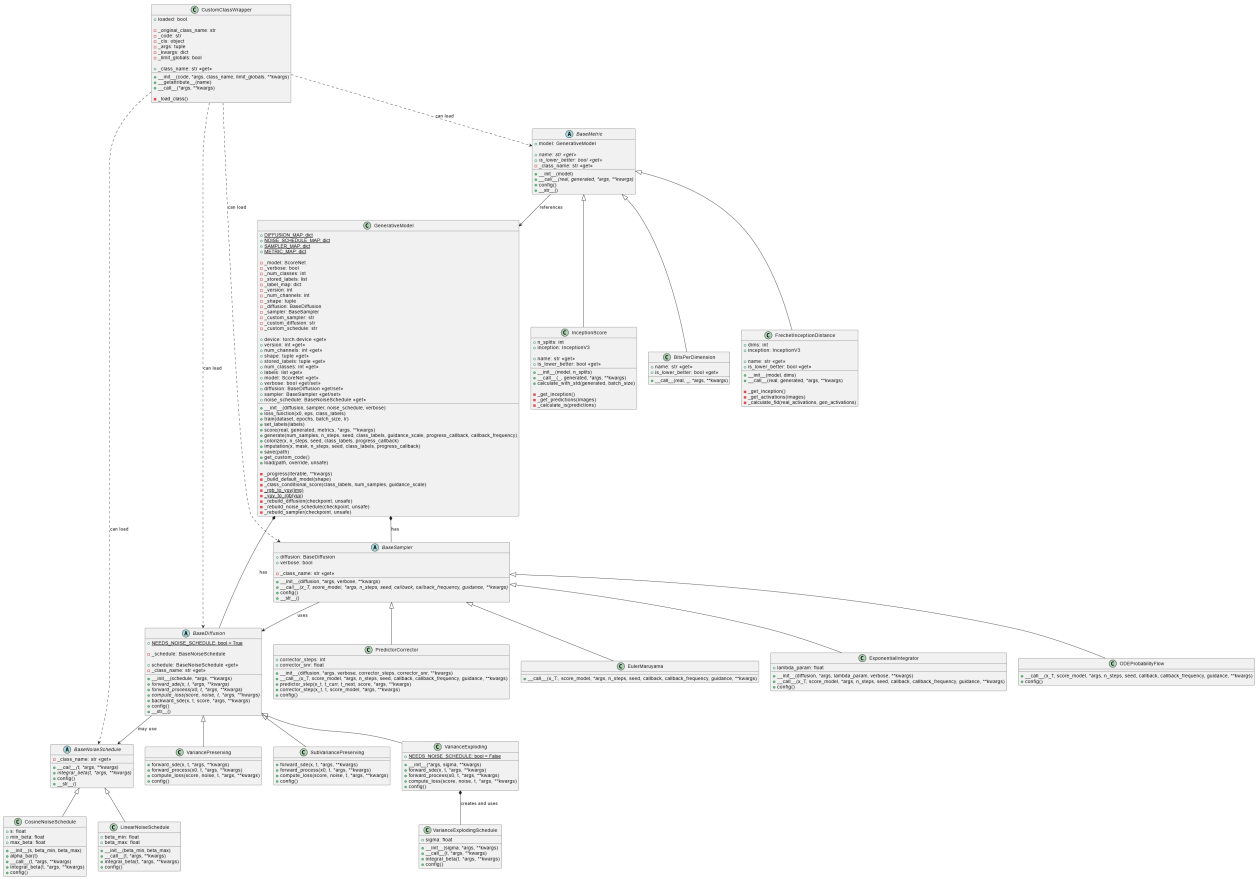

Figura 2.2: Diagrama de clases del paquete de generación de imágenes.

El diagrama ilustra la estructura jerárquica de las clases y sus interacciones:

- La clase `GenerativeModel` actúa como punto central, coordinando los componentes de difusión, muestreo y programación de ruido.
- Cada categoría de componentes (difusores, muestreadores y programadores de ruido) sigue un patrón de diseño con clases base abstractas que definen las interfaces comunes.
- Las implementaciones concretas heredan de estas clases base y proporcionan comportamientos específicos.
- El sistema de métricas sigue un diseño similar, con una clase base `BaseMetric` y especializaciones para cada métrica específica.

Este diseño permite la extensibilidad del sistema, facilitando la adición de nuevas implementaciones sin modificar el código existente. Por ejemplo, un usuario puede crear un nuevo tipo de difusor heredando de `BaseDiffusion` e implementando los métodos abstractos requeridos.

La flexibilidad del diseño también se refleja en cómo se pueden combinar los diferentes componentes. Por ejemplo, cualquier difusor puede utilizarse con cualquier muestreador, siempre que ambos implementen correctamente sus interfaces respectivas.

2.4. Validación y Pruebas

El proceso de validación y pruebas ha sido un componente esencial en el desarrollo de nuestro paquete de software, garantizando que todas las funcionalidades cumplan con los requisitos establecidos y proporcionen resultados correctos y consistentes.

2.4.1. Estrategia de Pruebas

- **Pruebas Unitarias**: Verifican el comportamiento correcto de componentes individuales aislados.
- **Pruebas de Integración**: Validan la interacción correcta entre diferentes módulos.
- **Pruebas de Sistema**: Comprueban el funcionamiento del sistema completo en escenarios de uso real.
- **Pruebas de Rendimiento**: Evalúan tiempos de ejecución y consumo de recursos.

Pruebas Unitarias

Las pruebas unitarias se han implementado utilizando el framework pytest, con un enfoque basado en fixtures para facilitar la configuración de escenarios de prueba. Se han desarrollado pruebas específicas para cada módulo del sistema:

- `test_diffusion.py`: Verifica el comportamiento de los diferentes procesos de difusión.
- `test_samplers.py`: Comprueba el funcionamiento de los métodos de muestreo.
- `test_noise.py`: Valida los programadores de ruido.
- `test_metrics.py`: Asegura la correcta implementación de las métricas de evaluación.
- `test_base.py`: Prueba la funcionalidad de la clase `GenerativeModel`.

Las pruebas unitarias cubren tanto casos típicos como escenarios extremos y de error, asegurando que todos los componentes manejen adecuadamente situaciones excepcionales.

Pruebas de Integración

Las pruebas de integración, implementadas en `test_integration.py`, verifican la interacción correcta entre los diferentes módulos del sistema. Estas pruebas simulan flujos de trabajo completos, incluyendo:

- Entrenamiento de modelos y generación de imágenes.
- Colorización e imputación de imágenes.
- Generación condicionada por clase.
- Guardado y carga de modelos.

Pruebas de Rendimiento

Se han realizado pruebas de rendimiento para evaluar:

- Tiempos de generación para diferentes configuraciones (ver Figura 2.3).
- Consumo de memoria durante el entrenamiento y la generación.
- Escalabilidad con el número de pasos.

Se pueden consultar comparaciones exhaustivas en el apéndice C.

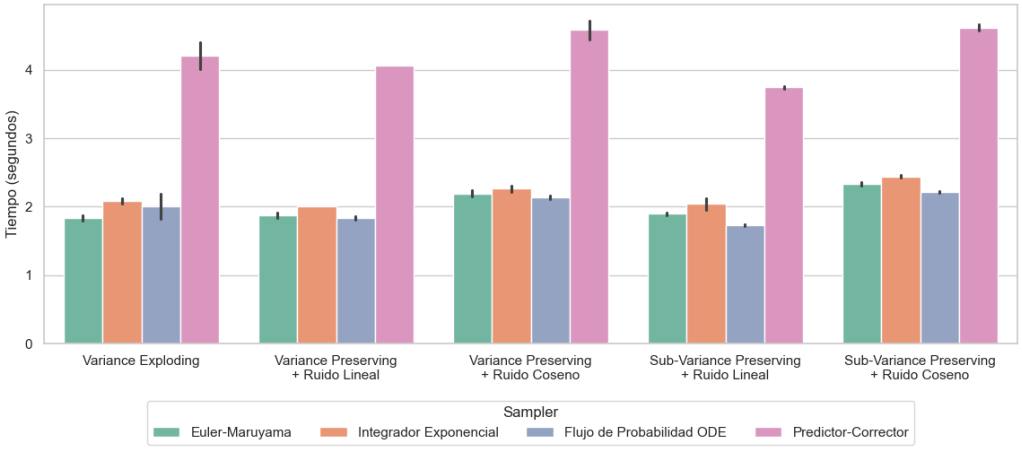

Figura 2.3: Tiempos de generación por configuración y sampler (MNIST, dataset parcial, generación incondicionada).

2.4.2. Compatibilidad y Entornos

Se ha verificado la compatibilidad del paquete con diversas configuraciones de hardware:

- Sistemas con GPU CUDA.
- Sistemas con CPU únicamente.
- Diferentes sistemas operativos (Windows y Linux).

2.5. Garantía de Calidad de Software

La garantía de calidad ha sido un aspecto fundamental en el desarrollo de nuestro paquete de software. Hemos implementado diversas prácticas y herramientas para asegurar que el código sea robusto, mantenible y conforme a los estándares de la industria.

2.5.1. Control de Versiones

El desarrollo se ha realizado utilizando Git como sistema de control de versiones, con un repositorio alojado en GitHub. Esto ha facilitado:

- Seguimiento detallado de cambios en el código.
- Desarrollo colaborativo coordinado.

- Revisión de código mediante pull requests.

Puede encontrarse el código fuente del paquete en el siguiente enlace: `https://github.com/HectorTablero/image-gen`.

2.5.2. Estándares de Codificación

El código se ha desarrollado siguiendo las guías de estilo de Google [11] para Python, garantizando consistencia y legibilidad.

2.5.3. Documentación

La documentación ha sido una prioridad durante todo el desarrollo, implementándose a varios niveles:

- **Documentación de API**: Generada automáticamente a partir de los docstrings en formato Google.
- **Documentación de Usuario**: Incluye manuales, tutoriales y ejemplos de uso.
- **Notebooks de Ejemplo**: Demuestran casos de uso concretos con ejemplos ejecutables.
- **Comentarios en el Código**: Explican secciones complejas o no intuitivas.

La documentación se genera automáticamente utilizando MkDocs y se publica mediante GitHub Pages, asegurando que siempre esté actualizada con la última versión del código. Puede consultarse aquí: `https://hectortablero.github.io/image-gen/`.

Adicionalmente, se puede encontrar una versión de la documentación generada por Devin [12] en el siguiente enlace: `https://deepwiki.com/HectorTablero/image-gen`.

2.5.4. Gestión de Dependencias

Las dependencias del proyecto se gestionan mediante:

- Archivos `requirements.txt` y `pyproject.toml` que especifican las dependencias y sus versiones.
- Entornos virtuales para aislar el desarrollo y evitar conflictos.

2.6. Otras Cuestiones

El software desarrollado en este proyecto se distribuye bajo la licencia MIT, lo que permite su uso, modificación y redistribución sin restricciones significativas, siempre que se mantenga la atribución original. Sin embargo, no se proporcionan garantías de ningún tipo, y los desarrolladores no se hacen responsables de su uso.

El usuario es el único responsable del uso del software y debe asegurarse de cumplir con la legislación vigente en su jurisdicción. En particular, el uso del software para generar contenido inapropiado, engañoso o que infrinja derechos de terceros está estrictamente desaconsejado.

3

Resultados

A continuación se muestran resultados de la generación de imágenes de modelos entrenados con imágenes de barcos del dataset CIFAR-10 (Figura 3.1) y con imágenes de cuatros de MNIST (Figura 3.2):

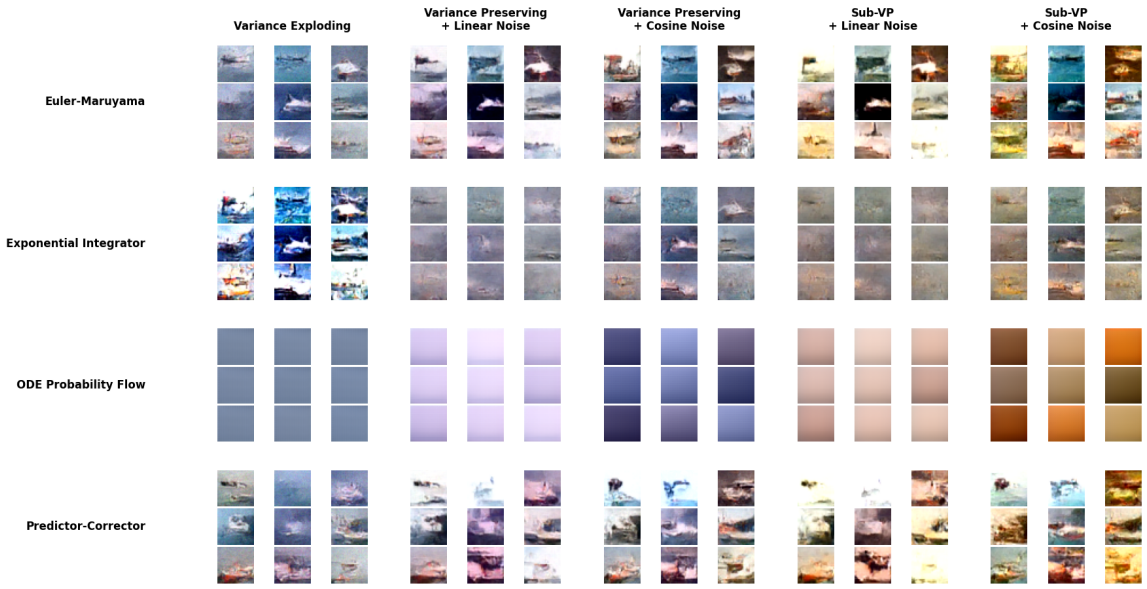

Figura 3.1: Imágenes generadas entrenando con CIFAR-10, 100 épocas.

- Sub-Variance Preserving es el difusor que más tarda en aprender la distribución de datos. En este caso, el modelo ha entrenado con solo 100 épocas, por lo que no ha tenido tiempo suficiente para aprender a diferenciar bien entre los barcos y el resto del dataset.

- ODE es determinista y converge a la media de la distribución de datos aprendida, por lo que puede no generar resultados satisfactorios en algunos casos. En el caso de MNIST, la generación es considerablemente mejor:

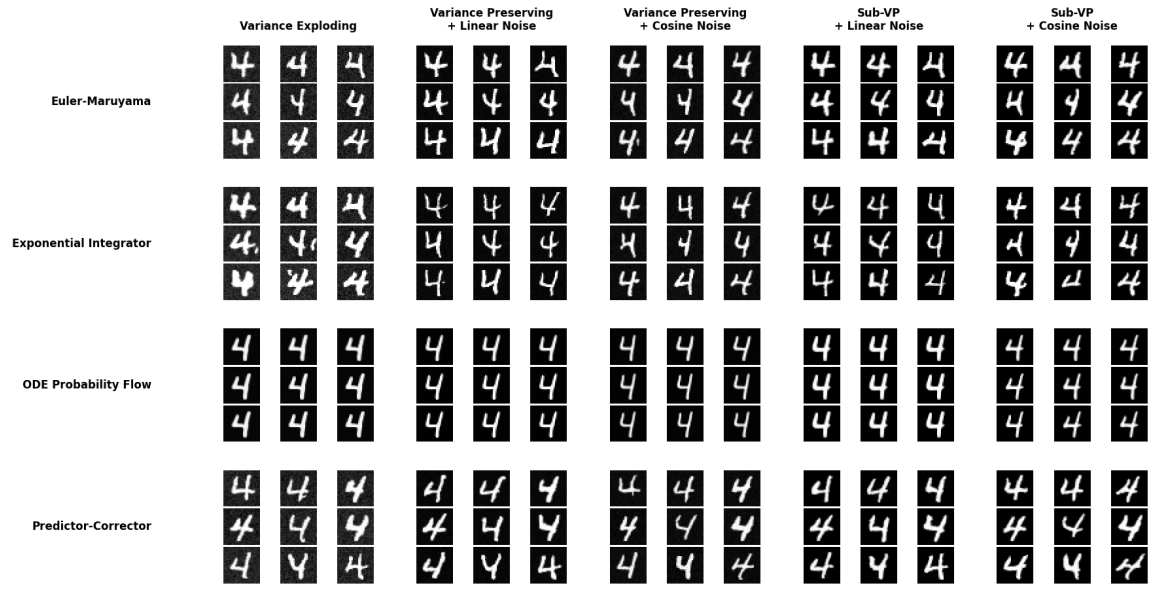

Figura 3.2: Imágenes generadas entrenando con MNIST, 100 épocas.

3.1. Ejemplos de Uso

En esta sección presentamos diversos ejemplos de uso de nuestro paquete que ilustran sus capacidades y funcionalidades. Estos ejemplos están disponibles como cuadernos Jupyter en el repositorio del proyecto, permitiendo a los usuarios reproducirlos y experimentar con diferentes configuraciones.

3.1.1. Generación Básica de Imágenes

El siguiente código muestra cómo inicializar un modelo generativo con configuración predeterminada y generar un conjunto de imágenes:

Código 3.1: Generación básica

```
from image_gen.visualization import display_images
from image_gen import GenerativeModel

# Inicializar modelo con difusión Variance Exploding y muestreador Euler-Maruyama
model = GenerativeModel(diffusion="ve", sampler="euler-maruyama")
model.train(dataset, epochs=25)

# Generar 4 imágenes con 500 pasos de muestreo
images = model.generate(num_samples=4, n_steps=500, seed=42)

# Visualizar las imágenes generadas
```

3.1.2. Colorización de Imágenes

El siguiente ejemplo muestra cómo utilizar el modelo para colorear una imagen en escala de grises:

Código 3.2: Colorización de imágenes

```
# Cargar modelo previamente entrenado
model = GenerativeModel.load("saved_models/cifar10.pth")

# Crear imagen en escala de grises (promediando canales RGB)
color_image = model.generate(num_samples=1)[0]
gray_image = torch.mean(color_image, dim=0, keepdim=True).unsqueeze(0)

# Colorizar la imagen
colorized = model.colorize(gray_image, n_steps=500)

# Visualizar original en grises y versión colorizada
display_images(gray_image)
display_images(colorized)
```

3.1.3. Imputación de Regiones

Este ejemplo muestra cómo realizar la imputación de regiones faltantes en una imagen:

Código 3.3: Imputación de imágenes

```
# Cargar modelo
model = GenerativeModel.load("saved_models/cifar10.pth")

# Generar imagen base
base_image = model.generate(num_samples=1)

# Crear máscara (1 = región a generar, 0 = preservar)
mask = torch.zeros_like(base_image)
h, w = base_image.shape[2], base_image.shape[3]
mask[:, :, h//4:3*h//4, w//4:3*w//4] = 1 # Máscara central rectangular

# Realizar imputación
results = model.imputation(base_image, mask, n_steps=500)

# Visualizar resultado
display_images(torch.cat([base_image, results], dim=0))
```

Se pueden consultar ejemplos adicionales en el apéndice B, donde se presentan casos de uso más avanzados.

3.2. Conclusiones

El desarrollo de este paquete de generación de imágenes basado en modelos de difusión ha permitido implementar y evaluar distintas variantes de estos modelos, demostrando su efectividad en diversas tareas de generación, desde la creación de imágenes desde ruido aleatorio hasta tareas más específicas como colorización e imputación.

3.2.1. Logros y Contribuciones

Los principales logros y contribuciones de este proyecto son:

- Implementación de tres variantes de procesos de difusión (VE, VP y Sub-VP), permitiendo comparar su rendimiento en diferentes escenarios.
- Desarrollo de cuatro métodos de muestreo con diferentes características de calidad y eficiencia, proporcionando flexibilidad para distintos casos de uso.
- Incorporación de capacidades de generación controlable que amplían significativamente la utilidad del paquete.
- Creación de una arquitectura modular y extensible que facilita la incorporación de nuevos componentes.
- Desarrollo de un dashboard interactivo que mejora significativamente la accesibilidad del paquete.
- Implementación de métricas estándar para la evaluación cuantitativa de la calidad de las imágenes generadas.

Los resultados experimentales confirman que los modelos de difusión representan una alternativa viable a otras técnicas generativas, con la ventaja adicional de un entrenamiento más estable y un marco teórico sólido basado en ecuaciones diferenciales estocásticas.

3.2.2. Limitaciones

A pesar de los resultados positivos, también es importante reconocer las limitaciones actuales del paquete:

- El tiempo de generación sigue siendo considerablemente mayor que el de otras técnicas generativas como las GANs, especialmente cuando se utilizan métodos de muestreo más sofisticados.
- La calidad de las imágenes generadas, aunque buena, todavía no alcanza el nivel de implementaciones más especializadas como Stable Diffusion, que operan en espacios latentes comprimidos.

- El entrenamiento de modelos para imágenes de alta resolución requiere recursos computacionales significativos que no se han explorado completamente en este proyecto.

3.2.3. Trabajo Futuro

Basándonos en los resultados obtenidos y las limitaciones identificadas, varias líneas de trabajo futuro podrían mejorar y extender este paquete:

- Implementación de difusión en espacio latente, siguiendo el enfoque de Stable Diffusion, para permitir la generación eficiente de imágenes de mayor resolución.
- Incorporación de técnicas de aceleración de muestreo, como DDIM [13] y DPM-Solver [14], que podrían reducir significativamente el tiempo de generación.
- Extensión a la generación multimodal, particularmente la generación de imágenes a partir de descripciones textuales.
- Desarrollo de capacidades de edición semántica, permitiendo modificar atributos específicos de las imágenes generadas.
- Optimización del rendimiento computacional, especialmente para uso en hardware con recursos limitados.

3.2.4. Consideraciones Finales

Este proyecto demuestra el potencial de los modelos de difusión como una herramienta versátil para diversas tareas de generación de imágenes. La arquitectura modular desarrollada proporciona una base sólida para futuros desarrollos y extensiones, tanto en el ámbito académico como en aplicaciones prácticas.

Consideramos que la combinación de un diseño orientado a objetos bien estructurado, una documentación exhaustiva y herramientas interactivas como el dashboard contribuyen significativamente a la accesibilidad y utilidad del paquete, facilitando su adopción por parte de investigadores y desarrolladores interesados en la generación de imágenes mediante modelos de difusión.

En conclusión, este proyecto no solo ha cumplido con los objetivos iniciales de implementar diferentes variantes de modelos de difusión y evaluar su rendimiento, sino que también ha generado un paquete de software útil y extensible que puede servir como base para futuras investigaciones y aplicaciones.

Bibliografía

[1] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," in *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37 of *ICML'15*, pp. 2256–2265, PMLR, 2015.

[2] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," in *Advances in Neural Information Processing Systems*, vol. 33 of *NeurIPS 2020*, pp. 6840–6851, Curran Associates, Inc., 2020.

[3] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, "Score-based generative modeling through stochastic differential equations," *International Conference on Learning Representations*, 2021.

[4] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.

[5] Y. Song, Q. Yang, X. Lu, and L. Chen, "Cgdiff: Controllable guided diffusion models for blind image colorization," in *Proceedings of the 31st ACM International Conference on Multimedia*, (New York, NY, USA), pp. 1654–1664, Association for Computing Machinery, 2023.

[6] A. Lugmayr, M. Danelljan, A. Romero, F. Yu, R. Timofte, and L. Van Gool, "Repaint: Inpainting using denoising diffusion probabilistic models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11461–11471, 2022.

[7] C. Saharia, W. Chan, H. Chang, C. Lee, J. Ho, T. Salimans, D. J. Fleet, and M. Norouzi, "Palette: Image-to-image diffusion models," in *ACM SIGGRAPH 2022 Conference Proceedings*, pp. 1–10, 2022.

[8] P. Dhariwal and A. Nichol, "Diffusion models beat GANs on image synthesis," in *Advances in Neural Information Processing Systems* (M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), vol. 34, pp. 8780–8794, Curran Associates, Inc., 2021.

[9] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with CLIP latents," *arXiv preprint arXiv:2204.06125*, 2022.

[10] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. L. Denton, K. Ghasemipour, R. Gontijo Lopes, B. Karagol Ayan, T. Salimans, J. Ho, D. J. Fleet, and M. Norouzi, "Photorealistic text-to-image diffusion models with deep language understanding," in *Advances in Neural Information Processing Systems*, vol. 35, pp. 36479–36494, Curran Associates, Inc., 2022.

[11] Google, "Google style guides," 2022.

[12] Cognition AI, "Introducing devin, the first ai software engineer," 2024.

[13] J. Song, C. Meng, and S. Ermon, "Denoising diffusion implicit models," *arXiv preprint arXiv:2010.02502*, 2020.

[14] C. Lu, Y. Zhou, Z. Liu, D. He, Y. Chen, and C.-J. Hsieh, “Dpm-solver: A fast ode solver for diffusion probabilistic models,” *arXiv preprint arXiv:2206.00927*, 2022.

Apéndices

A

Material Técnico

En este apéndice presentamos material técnico adicional que profundiza en los fundamentos matemáticos y algorítmicos de los modelos de difusión implementados en nuestro paquete.

A.1. Ecuaciones Diferenciales Estocásticas en Modelos de Difusión

Los modelos de difusión se fundamentan en procesos estocásticos definidos por ecuaciones diferenciales estocásticas (SDE). Para un proceso de difusión, definimos una SDE directa (forward) que describe cómo se añade ruido a los datos, y una SDE inversa (backward) que gobierna el proceso de eliminación de ruido para la generación.

A.1.1. SDE Directa

La ecuación diferencial estocástica directa tiene la forma general:

$$dx = f(x,t)dt + g(t)dw \tag{A.1}$$

donde $f(x,t)$ es el término de deriva (drift), $g(t)$ es el coeficiente de difusión, y dw representa un proceso de Wiener estándar.

Para los tres tipos de procesos de difusión implementados, estos términos se definen específicamente como:

Variance Exploding (VE):

$$f(x,t) = 0 \tag{A.2}$$

$$g(t) = \sigma^t \tag{A.3}$$

Variance Preserving (VP):

$$f(x,t) = -\frac{\beta(t)}{2}x \tag{A.4}$$

$$g(t) = \sqrt{\beta(t)} \tag{A.5}$$

Sub-Variance Preserving (Sub-VP):

$$f(x,t) = -\frac{\beta(t)}{2}x \tag{A.6}$$

$$g(t) = \sqrt{\beta(t) \cdot (1 - e^{-2\int_0^t \beta(s)ds})} \tag{A.7}$$

A.1.2. SDE Inversa

La SDE inversa, utilizada para la generación, se deriva de la directa y tiene la forma:

$$dx = [f(x,t) - g(t)^2 \nabla_x \log p_t(x)]dt + g(t)d\bar{w} \tag{A.8}$$

donde $\nabla_x \log p_t(x)$ es el score function (gradiente del logaritmo de la densidad) que se aproxima con una red neuronal durante el entrenamiento.

A.1.3. Demostración de la Derivación de la SDE Inversa

Para derivar la SDE inversa a partir de la directa, comenzamos con la ecuación de Fokker-Planck, que describe cómo evoluciona la densidad de probabilidad $p_t(x)$ bajo la SDE directa:

$$\frac{\partial p_t(x)}{\partial t} = -\nabla_x \cdot [f(x,t)p_t(x)] + \frac{1}{2}\nabla_x^2 \cdot [g(t)^2 p_t(x)] \tag{A.9}$$

Para el proceso de difusión inverso en tiempo, la ecuación de Fokker-Planck toma la forma:

$$\frac{\partial p_t(x)}{\partial (-t)} = -\nabla_x \cdot [\hat{f}(x,t)p_t(x)] + \frac{1}{2}\nabla_x^2 \cdot [g(t)^2 p_t(x)] \tag{A.10}$$

donde $\hat{f}(x,t)$ es el término de deriva en la SDE inversa.

Expandiendo el lado izquierdo:

$$-\frac{\partial p_t(x)}{\partial t} = -\nabla_x \cdot [\hat{f}(x,t)p_t(x)] + \frac{1}{2}\nabla_x^2 \cdot [g(t)^2 p_t(x)] \tag{A.11}$$

Igualando esta ecuación con la ecuación de Fokker-Planck directa:

$$-\nabla_x \cdot [f(x,t)p_t(x)] + \frac{1}{2}\nabla_x^2 \cdot [g(t)^2 p_t(x)] = -\nabla_x \cdot [\hat{f}(x,t)p_t(x)] + \frac{1}{2}\nabla_x^2 \cdot [g(t)^2 p_t(x)] \tag{A.12}$$

Simplificando:

$$\nabla_x \cdot [f(x,t)p_t(x)] = \nabla_x \cdot [\hat{f}(x,t)p_t(x)] \tag{A.13}$$

Lo que implica:

$$\hat{f}(x,t)p_t(x) = f(x,t)p_t(x) + \nabla_x \cdot [g(t)^2 p_t(x)] \tag{A.14}$$

Usando la regla del producto para el gradiente:

$$\nabla_x \cdot [g(t)^2 p_t(x)] = g(t)^2 \nabla_x p_t(x) = g(t)^2 p_t(x) \nabla_x \log p_t(x) \tag{A.15}$$

donde hemos utilizado la relación $\nabla_x p_t(x) = p_t(x)\nabla_x \log p_t(x)$.

Sustituyendo y dividiendo por $p_t(x)$:

$$\hat{f}(x,t) = f(x,t) + g(t)^2 \nabla_x \log p_t(x) \tag{A.16}$$

Por convención, en la literatura sobre modelos de difusión, la SDE inversa se escribe con el signo negativo en el segundo término, obteniendo:

$$dx = [f(x,t) - g(t)^2 \nabla_x \log p_t(x)]dt + g(t)d\bar{w} \tag{A.17}$$

A.2. Métodos de Muestreo

Para resolver numéricamente las SDEs involucradas en el proceso de generación, implementamos cuatro métodos principales:

A.2.1. Euler-Maruyama

El método de Euler-Maruyama es una extensión estocástica del método de Euler para EDOs:

$$x_{t+\Delta t} = x_t + f(x_t, t)\Delta t + g(t)\sqrt{\Delta t}z \tag{A.18}$$

donde $z \sim \mathcal{N}(0, I)$ es ruido gaussiano.

A.2.2. Integrador Exponencial

El integrador exponencial aprovecha la estructura de la SDE para una integración más estable:

$$x_{t+\Delta t} = x_t e^{\lambda \Delta t} + \frac{g^2}{2\lambda}(e^{2\lambda \Delta t} - 1)\nabla_x \log p_t(x) \tag{A.19}$$

donde λ es un parámetro de estabilización.

Demostración del Integrador Exponencial

Para el caso VP-SDE donde $f(x,t) = -\frac{\beta(t)}{2}x$, podemos derivar el integrador exponencial a partir de la SDE inversa:

$$dx = \left[-\frac{\beta(t)}{2}x - g(t)^2 \nabla_x \log p_t(x)\right] dt + g(t) d\bar{w} \tag{A.20}$$

Para simplificar la derivación, consideramos primero la ecuación homogénea:

$$dx = -\frac{\beta(t)}{2} x \, dt \tag{A.21}$$

La solución general es:

$$x(t) = x_0 e^{-\frac{1}{2}\int_0^t \beta(s) ds} \tag{A.22}$$

Para la ecuación completa, aplicamos el método de variación de parámetros. Sea $\lambda = \frac{\beta(t)}{2}$ (considerándolo constante en un intervalo pequeño). El factor integrador es $e^{\lambda t}$, lo que nos lleva a:

$$\frac{d}{dt}(e^{\lambda t} x) = e^{\lambda t}\frac{dx}{dt} + \lambda e^{\lambda t} x \tag{A.23}$$

$$= e^{\lambda t}\left(-\lambda x - g(t)^2 \nabla_x \log p_t(x)\right) + \lambda e^{\lambda t} x \tag{A.24}$$

$$= -e^{\lambda t} g(t)^2 \nabla_x \log p_t(x) \tag{A.25}$$

Integrando ambos lados de t a $t + \Delta t$ y asumiendo $\nabla_x \log p_t(x)$ aproximadamente constante en ese intervalo:

$$e^{\lambda(t+\Delta t)} x_{t+\Delta t} - e^{\lambda t} x_t = -g(t)^2 \nabla_x \log p_t(x) \int_t^{t+\Delta t} e^{\lambda s} ds \tag{A.26}$$

$$= -g(t)^2 \nabla_x \log p_t(x) \frac{e^{\lambda(t+\Delta t)} - e^{\lambda t}}{\lambda} \tag{A.27}$$

Despejando $x_{t+\Delta t}$:

$$x_{t+\Delta t} = e^{-\lambda(t+\Delta t)}\left[e^{\lambda t} x_t - g(t)^2 \nabla_x \log p_t(x) \frac{e^{\lambda(t+\Delta t)} - e^{\lambda t}}{\lambda}\right] \tag{A.28}$$

$$= x_t e^{-\lambda \Delta t} - g(t)^2 \nabla_x \log p_t(x) \frac{1 - e^{-\lambda \Delta t}}{\lambda} \tag{A.29}$$

Reorganizando los términos:

$$x_{t+\Delta t} = x_t e^{-\lambda \Delta t} + \frac{g(t)^2}{\lambda}(1 - e^{-\lambda \Delta t})\nabla_x \log p_t(x) \tag{A.30}$$

$$= x_t e^{-\lambda \Delta t} + \frac{g(t)^2}{2\lambda}(1 - e^{-2\lambda \Delta t})\nabla_x \log p_t(x) \tag{A.31}$$

que es la fórmula del integrador exponencial. Esta derivación demuestra por qué este método proporciona una integración más estable especialmente para pasos de tiempo grandes, ya que tiene en cuenta la estructura específica de la SDE.

A.2.3. Flujo de Probabilidad ODE

El flujo de probabilidad ODE es una aproximación determinista que elimina el término de ruido:

$$\frac{dx}{dt} = f(x,t) - \frac{1}{2}g(t)^2 \nabla_x \log p_t(x) \tag{A.32}$$

A.2.4. Predictor-Corrector

El método predictor-corrector combina un paso predictor basado en Euler-Maruyama con un paso corrector basado en dinámica de Langevin:

Predictor:

$$\tilde{x}_{t+\Delta t} = x_t + f(x_t, t)\Delta t \tag{A.33}$$

Corrector:

$$x_{t+\Delta t} = \tilde{x}_{t+\Delta t} + \epsilon \nabla_x \log p_{t+\Delta t}(\tilde{x}_{t+\Delta t}) + \sqrt{2\epsilon} z \tag{A.34}$$

donde ϵ es el tamaño de paso para la corrección.

Demostración del Método Predictor-Corrector

El método predictor-corrector combina dos técnicas de aproximación numérica:

1. El paso predictor proporciona una primera estimación de x_{t+1} utilizando el método de Euler determinista:

$$\tilde{x}_{t+1} = x_t + f(x_t, t)(t_{t+1} - t_t) \tag{A.35}$$

2. El paso corrector refina esta estimación utilizando un paso de la dinámica de Langevin:

$$x_{t+1} = \tilde{x}_{t+1} + \epsilon \nabla_x \log p_{t+1}(\tilde{x}_{t+1}) + \sqrt{2\epsilon} z \tag{A.36}$$

Este segundo paso se puede interpretar como una iteración del algoritmo MCMC de Langevin ajustado. La teoría detrás del muestreo de Langevin establece que, para una distribución objetivo $p(x)$, el proceso de Langevin:

$$x_{i+1} = x_i + \epsilon \nabla_x \log p(x_i) + \sqrt{2\epsilon} z \tag{A.37}$$

genera muestras que convergen a la distribución $p(x)$ cuando $\epsilon \to 0$ y el número de iteraciones tiende a infinito.

En el contexto de los modelos de difusión, el paso corrector utiliza la estimación del score $\nabla_x \log p_{t+1}(\tilde{x}_{t+1})$ para dirigir la muestra hacia regiones de mayor probabilidad según la distribución en el tiempo $t + 1$.

La relación entre el tamaño de paso ϵ y la magnitud del ruido $\sqrt{2\epsilon}$ no es arbitraria, sino que se deriva directamente de la ecuación de Langevin y garantiza que el proceso converge a la distribución objetivo $p_{t+1}(x)$.

El método predictor-corrector resulta más efectivo cuando se aplican múltiples pasos correctores por cada paso predictor, ya que cada corrección adicional acerca más la muestra a la distribución correcta.

A.3. Programadores de Ruido

Los programadores de ruido definen la función $\beta(t)$ utilizada en los procesos VP y Sub-VP.

A.3.1. Programador Lineal

Define $\beta(t)$ como una función lineal entre β_{min} y β_{max}:

$$\beta(t) = \beta_{min} + t(\beta_{max} - \beta_{min}) \tag{A.38}$$

con la integral correspondiente:

$$\int_0^t \beta(s)ds = \beta_{min}t + \frac{(\beta_{max} - \beta_{min})t^2}{2} \tag{A.39}$$

Demostración de la Integral del Programador Lineal

La integral del programador lineal se calcula directamente:

$$\int_0^t \beta(s)ds = \int_0^t [\beta_{min} + s(\beta_{max} - \beta_{min})]ds \tag{A.40}$$

$$= \int_0^t \beta_{min}ds + (\beta_{max} - \beta_{min}) \int_0^t sds \tag{A.41}$$

$$= \beta_{min} \int_0^t ds + (\beta_{max} - \beta_{min}) \left[\frac{s^2}{2}\right]_0^t \tag{A.42}$$

$$= \beta_{min} \cdot t + (\beta_{max} - \beta_{min}) \cdot \frac{t^2}{2} \tag{A.43}$$

$$= \beta_{min}t + \frac{(\beta_{max} - \beta_{min})t^2}{2} \tag{A.44}$$

A.3.2. Programador Coseno

Define $\beta(t)$ basado en una función coseno, proporcionando una transición más suave:

$$\overline{\alpha}(t) = \frac{\cos^2\left(\frac{\pi}{2} \cdot \frac{t+s}{1+s}\right)}{\cos^2\left(\frac{\pi}{2} \cdot \frac{s}{1+s}\right)} \tag{A.45}$$

$$\beta(t) = 1 - \frac{\overline{\alpha}(t)}{\overline{\alpha}(t - \Delta t)} \tag{A.46}$$

Demostración del Programador Coseno

La relación entre α_t y β_t se define como:

$$\alpha_t = \frac{\overline{\alpha}(t)}{\overline{\alpha}(t - \Delta t)} \tag{A.47}$$

$$\beta_t = 1 - \alpha_t = 1 - \frac{\overline{\alpha}(t)}{\overline{\alpha}(t - \Delta t)} \tag{A.48}$$

Para calcular la integral de β desde 0 hasta t, observamos que $\overline{\alpha}(t) = \prod_{i=1}^{t}(1 - \beta_i)$ en el caso discreto. Tomando logaritmos:

$$\log(\overline{\alpha}(t)) = \log\left(\prod_{i=1}^{t}(1 - \beta_i)\right) \tag{A.49}$$

$$= \sum_{i=1}^{t} \log(1 - \beta_i) \tag{A.50}$$

Para valores pequeños de β_i, podemos aproximar $\log(1 - \beta_i) \approx -\beta_i$, lo que nos da:

$$\log(\overline{\alpha}(t)) \approx -\sum_{i=1}^{t} \beta_i \tag{A.51}$$

En el límite continuo, esta suma se convierte en una integral:

$$\log(\overline{\alpha}(t)) \approx -\int_0^t \beta(s)ds \tag{A.52}$$

Despejando la integral:

$$\int_0^t \beta(s)ds \approx -\log(\overline{\alpha}(t)) \tag{A.53}$$

A.4. Generación Controlable

A.4.1. Colorización mediante Condicionamiento de Luminancia

Para la colorización, convertimos la imagen a espacio YUV, mantenemos el canal Y (luminancia) y generamos los canales U y V (crominancia). Técnicamente, implementamos esto mediante una función de guía en el proceso de muestreo:

$$x_t = (1 - \alpha_t)x_t^{generado} + \alpha_t x_t^{original} \tag{A.54}$$

donde α_t disminuye linealmente de 1 a 0 durante el proceso de muestreo.

A.4.2. Imputación mediante Mascarado

Para la imputación, utilizamos una máscara binaria M donde 1 indica las regiones a generar y 0 las regiones a preservar. El proceso de muestreo se modifica para mantener fijas las regiones no enmascaradas:

$$x_t = M \odot x_t^{generado} + (1 - M) \odot x_t^{original} \tag{A.55}$$

A.4.3. Generación Condicionada por Clase

Implementamos generación condicionada utilizando classifier-free guidance. La idea principal es entrenar un modelo condicional en las etiquetas de clase y uno incondicional, y luego combinar sus predicciones durante el muestreo:

$$\nabla_{x_t} \log p(x_t|y) \approx \nabla_{x_t} \log p(x_t) + w \cdot (\nabla_{x_t} \log p(x_t|y) - \nabla_{x_t} \log p(x_t)) \tag{A.56}$$

donde w es un parámetro de escala que controla la fuerza de la guía.

Demostración de Classifier-Free Guidance

El método de classifier-free guidance combina las predicciones de un modelo condicional y uno incondicional durante el muestreo. La fórmula:

$$\nabla_{x_t} \log p(x_t|y) \approx \nabla_{x_t} \log p(x_t) + w \cdot (\nabla_{x_t} \log p(x_t|y) - \nabla_{x_t} \log p(x_t)) \tag{A.57}$$

se puede reescribir como:

$$\nabla_{x_t} \log p(x_t|y) \approx (1 - w) \cdot \nabla_{x_t} \log p(x_t) + w \cdot \nabla_{x_t} \log p(x_t|y) \tag{A.58}$$

Para comprender el mecanismo, consideremos los siguientes casos:

1. $w = 0$: Recuperamos la generación incondicional $\nabla_{x_t} \log p(x_t)$.
2. $w = 1$: Obtenemos la generación condicionada estándar $\nabla_{x_t} \log p(x_t|y)$.
3. $w > 1$: Amplificamos el efecto del condicionamiento, haciendo que la generación sea más consistente con la clase deseada y.

Teóricamente, esto funciona porque $\nabla_{x_t} \log p(x_t|y) - \nabla_{x_t} \log p(x_t)$ aísla el efecto puro del condicionamiento. Al escalar este término con $w > 1$, enfatizamos las características distintivas de la clase y, produciendo muestras más "típicas"de dicha clase.

Una ventaja significativa de este enfoque es que no requiere un clasificador externo para la guía, sino que extrae información de condicionamiento del mismo modelo de score, reduciendo la complejidad computacional.

A.5. Paralelización y Optimizaciones

En nuestra implementación, hemos incorporado varias optimizaciones para mejorar el rendimiento:

- Procesamiento por lotes para aprovechar la paralelización en GPU.
- Uso de `torch.no_grad()` durante el muestreo para reducir el consumo de memoria.
- Implementación vectorizada de las operaciones de difusión para mejorar la eficiencia.
- Detección y manejo de valores NaN/Inf para mejorar la estabilidad numérica.

A.5.1. Detalles de Optimización

La implementación vectorizada de operaciones es crucial para el rendimiento en GPU. Por ejemplo, para el proceso de difusión, aplicamos transformaciones a todos los elementos de un lote simultáneamente:

$$x_{t+1}^{(batch)} = F(x_t^{(batch)}, t) \tag{A.59}$$

donde F representa cualquiera de nuestros métodos de muestreo.

El uso de `torch.no_grad()` elimina el seguimiento de gradientes durante el muestreo:

```
with torch.no_grad():
    # Proceso de muestreo
```

Esto reduce significativamente el consumo de memoria, ya que no se almacenan los grafos computacionales para el cálculo de gradientes.

Para la estabilidad numérica, implementamos detección y manejo de valores no finitos:

```
if torch.isnan(x).any() or torch.isinf(x).any():
    # Procedimiento de recuperación
```

Este material técnico complementa la documentación principal, proporcionando detalles más específicos sobre los algoritmos y métodos implementados en nuestro paquete.

B

Ejemplos Adicionales

En este apéndice presentamos ejemplos adicionales que ilustran características específicas o casos de uso avanzados de nuestro paquete de generación de imágenes.

B.1. Efectos de los Programadores de Ruido

La Figura B.1 ilustra el efecto de diferentes programadores de ruido en la trayectoria del proceso de difusión.

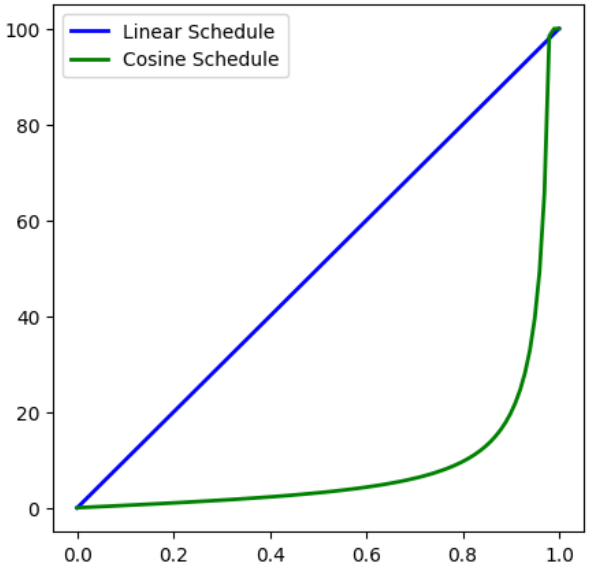

Figura B.1: Trayectorias del proceso de difusión para diferentes programadores de ruido: Linear (azul) vs. Cosine (verde) vs. Exponential personalizado (rojo).

El programador coseno muestra una transición más suave que preserva mejor los detalles estructurales durante las etapas intermedias del proceso de difusión, lo que se traduce en mejores resultados para imágenes complejas.

B.2. Más Ejemplos

Código B.1: Visualización del proceso de generación

```python
from image_gen import GenerativeModel
from image_gen.visualization import create_evolution_widget
from IPython.display import HTML

# Cargar modelo
model = GenerativeModel.load("saved_models/cifar10.pth")

# Crear y mostrar animación del proceso de generación
animation = create_evolution_widget(model, seed=42)
HTML(animation.to_jshtml(default_mode="once"))
```

Código B.2: Evaluación de imágenes

```python
from image_gen import GenerativeModel
from image_gen.metrics import BitsPerDimension, FrechetInceptionDistance, InceptionScore
import torch

# Cargar modelo y dataset de prueba
model = GenerativeModel.load("saved_models/cifar10_model.pt")
test_data = torch.utils.data.DataLoader(test_dataset, batch_size=64)

# Generar muestras para evaluación
generated = model.generate(num_samples=1000, n_steps=500)

# Calcular métricas
real_batch = next(iter(test_data))
scores = model.score(
    real=real_batch,
    generated=generated,
    metrics=["bpd", "fid", "is"]
)

print(f"Bits Per Dimension: {scores['Bits Per Dimension']:.4f}")
print(
    f"Fréchet Inception Distance: {scores['Fréchet Inception Distance']:.4f}")
print(f"Inception Score: {scores['Inception Score']:.4f}")
```

Código B.3: Generación condicionada por clase

```
import matplotlib.pyplot as plt
from image_gen import GenerativeModel

# Cargar modelo con soporte para condicionamiento por clase
model = GenerativeModel.load("saved_models/mnist.pth")

# Generar imágenes de la clase 7
class_samples = model.generate(
    num_samples=4,
    class_labels=7,
    guidance_scale=3.0,
    n_steps=500
)
display_images(class_samples)

# Comparar efecto de diferentes escalas de guía
fig, axs = plt.subplots(1, 5, figsize=(15, 3))
for i, scale in enumerate([0, 1, 3, 5, 10]):
    sample = model.generate(
        num_samples=1, class_labels=7, guidance_scale=scale)[0]
    axs[i].imshow(sample.permute(1, 2, 0).cpu().numpy())
    axs[i].set_title(f"Scale = {scale}")
    axs[i].axis('off')
plt.tight_layout()
plt.show()
```

Código B.4: Programador de ruido personalizado

```
from torch import Tensor
from image_gen import GenerativeModel
from image_gen.noise import BaseNoiseSchedule


class ExponentialNoiseSchedule(BaseNoiseSchedule):
    def __init__(self, *args, beta_min: float = 0.001, beta_max: float = 50.0, e: float = 2.0, **kwargs):
        self.beta_min = beta_min
        self.beta_max = beta_max
        self.e = e

    def __call__(self, t: Tensor, *args, **kwargs) -> Tensor:
        return self.beta_min + t **self.e *(self.beta_max -self.beta_min)

    def integral_beta(self, t: Tensor, *args, **kwargs) -> Tensor:
        integral_beta_min = self.beta_min *t
        integral_t = (self.beta_max -self.beta_min) *\
            (t **(self.e + 1)) / (self.e + 1)
        return integral_beta_min + integral_t

    def config(self) -> dict:
        return {
            "beta_min": self.beta_min,
            "beta_max": self.beta_max,
            "e": self.e
        }


# Inicializar modelo con ruido personalizado
model = GenerativeModel(
    diffusion="vp",
    sampler="exponential",
    noise_schedule=ExponentialNoiseSchedule(
        beta_min=0.001, beta_max=50.0, e=2.0)
)
model.train(dataset, epochs=25)

# Generar 16 imágenes con 500 pasos de muestreo
images = model.generate(num_samples=16, n_steps=500, seed=42)

# Visualizar las imágenes generadas
display_images(images)
```

C

Comparaciones Exhaustivas

En este apéndice presentamos un análisis comparativo detallado de las diferentes configuraciones de modelos de difusión implementados en nuestro paquete. Los experimentos cubren varias arquitecturas de difusión (VE, VP-lin, VP-cos, SVP-lin, SVP-cos), muestreadores (Euler-Maruyama, Integrador Exponencial, Flujo de Probablidad ODE, Predictor-Corrector), conjuntos de datos (MNIST, CIFAR-10), y tareas (generación, imputación, colorización). Cada configuración se evalúa usando métricas estándar: BPD (Bits Per Dimension), FID (Fréchet Inception Distance) e IS (Inception Score), junto con tiempos de ejecución.

C.1. Comparación de Arquitecturas de Difusión

La Tabla C.1 presenta un análisis comparativo del rendimiento de diferentes arquitecturas de difusión en la tarea de generación incondicional.

Arquitectura	Dataset	BPD	FID	IS	Tiempo (s)
VE	MNIST	**0.40551**	196.33432	1.26927	2.04
VP-lin	MNIST	0.644	106.20166	**1.29585**	2.01
VP-cos	MNIST	2.22936	**81.93957**	1.20039	2.31
SVP-lin	MNIST	0.43898	87.02211	1.20718	2.13
SVP-cos	MNIST	1.16706	96.21839	1.20235	2.46
VE	CIFAR-10	**0.05282**	289.67649	**1.28469**	2.12
VP-lin	CIFAR-10	0.0538	296.70873	1.14388	2.01
VP-cos	CIFAR-10	0.10169	286.21079	1.2605	2.22
SVP-lin	CIFAR-10	0.1109	296.67043	1.173	1.95
SVP-cos	CIFAR-10	0.21226	**274.99667**	1.22587	2.43

Tabla C.1: Generación Incondicional con el Integrador Exponencial para modelos entrenados sobre clases concretas.

De esta tabla podemos extraer varias observaciones importantes:

1. Los modelos VP-lin y VP-cos generalmente logran los valores FID más bajos para MNIST, indicando mejor calidad de generación de imágenes.
2. Para CIFAR-10, los modelos VE y VP-lin tienen los valores de BPD más bajos, mientras que VP-cos muestra los mejores valores de IS.
3. Los modelos SVP (tanto lin como cos) muestran consistentemente valores FID más altos, sugiriendo una calidad de imagen inferior en términos de similitud con la distribución original.
4. Los tiempos de generación son bastante similares entre arquitecturas para el mismo conjunto de datos, con ligeras ventajas para VE y VP-lin.

C.2. Comparación de Muestreadores

La Tabla C.2 presenta una comparación de diferentes muestreadores utilizando la arquitectura VP-cos en CIFAR-10 para generación incondicional.

Muestreador	FID	IS	Tiempo (s)	Relativo
Euler-Maruyama	**243.56972**	1.33851	2.15	1.00x
Integrador Exponencial	286.21079	1.2605	2.22	1.03x
Flujo de Probablidad ODE	422.94539	1.0605	2.16	1.08x
Predictor-Corrector	260.8202	**1.36236**	4.72	2.20x

Tabla C.2: Comparación de muestreadores con VP-cos en CIFAR-10.

Las observaciones clave incluyen:

1. El muestreador Predictor-Corrector logra el mejor Inception Score, a expensas de un mayor tiempo de computación (aproximadamente 1.9 veces más lento que otros muestreadores).
2. El muestreador Flujo de Probablidad ODE muestra el FID más alto y el IS más bajo, indicando un rendimiento inferior en términos de calidad de imagen para esta arquitectura específica.
3. Los muestreadores Euler-Maruyama y Integrador Exponencial ofrecen un buen equilibrio entre calidad y eficiencia computacional.

C.3. Comparación entre Generación Condicional e Incondicional

La Tabla C.3 compara el rendimiento de la generación condicional versus incondicional para varios modelos en MNIST y CIFAR-10.

Arquitectura	Dataset	Muestreador	Condicional	FID	IS	Tiempo (s)
VE	MNIST	Euler-Maruyama	Sí	178.70594	1.15138	3.91
VE	MNIST	Euler-Maruyama	No	136.60354	**1.44729**	2.61
VP-lin	MNIST	Predictor-Corrector	Sí	158.15242	1.14223	7.49
VP-lin	MNIST	Predictor-Corrector	No	**112.78947**	1.31432	3.7
VE	CIFAR-10	Euler-Maruyama	Sí	233.69821	1.40673	3.57
VE	CIFAR-10	Euler-Maruyama	No	240.34489	1.26761	2.02
VP-lin	CIFAR-10	Predictor-Corrector	Sí	244.38741	**1.51985**	6.93
VP-lin	CIFAR-10	Predictor-Corrector	No	**208.4664**	1.42302	3.77

Tabla C.3: Comparación de generación condicional e incondicional.

Observaciones importantes:

1. La generación condicional generalmente requiere más tiempo de cómputo que la incondicional, aproximadamente 1.5-2 veces más.
2. Para MNIST, la generación incondicional muestra mejores métricas tanto en FID como en IS, lo que sugiere que el acondicionamiento podría estar limitando la calidad de la generación.
3. Para CIFAR-10, los resultados son mixtos: mientras que el FID es generalmente peor para la generación condicional, el IS tiende a ser mejor, especialmente para VP-cos, sugiriendo que la generación condicional produce imágenes con características más distinguibles de cada clase aunque con alguna pérdida en la similitud general a la distribución.

C.4. Análisis de Tareas de Imputación

La Tabla C.4 muestra los resultados de la tarea de imputación con diferentes configuraciones en CIFAR-10.

Observaciones:

1. Los modelos VE generalmente logran los valores FID más bajos en tareas de imputación, sugiriendo una mejor capacidad para preservar la estructura de la imagen original.
2. Los modelos VP-lin y VP-cos muestran valores IS más altos, potencialmente indicando una mayor diversidad en las regiones imputadas.
3. El muestreador Predictor-Corrector tiende a producir los mejores resultados dentro de cada arquitectura, pero a expensas de tiempos de cómputo significativamente mayores.

Arquitectura	Muestreador	FID	IS	Tiempo (s)
VE	Euler-Maruyama	160.77599	1.57971	3.64
VE	Integrador Exponencial	175.11949	**1.59288**	3.58
VE	Flujo de Probabilidad ODE	163.86393	1.5555	3.40
VE	Predictor-Corrector	**158.74365**	1.56664	8.06
VP-lin	Euler-Maruyama	201.2792	**1.68167**	3.41
VP-lin	Integrador Exponencial	**157.22254**	1.51202	3.55
VP-lin	Flujo de Probabilidad ODE	195.84693	1.57683	3.33
VP-lin	Predictor-Corrector	184.93121	1.65653	6.74
VP-cos	Euler-Maruyama	183.48723	1.64633	3.61
VP-cos	Integrador Exponencial	**157.00648**	1.62348	4.04
VP-cos	Flujo de Probabilidad ODE	181.98185	1.53347	3.77
VP-cos	Predictor-Corrector	180.22002	**1.64953**	7.52
SVP-lin	Euler-Maruyama	192.76721	1.60672	3.38
SVP-lin	Integrador Exponencial	**164.09631**	1.46515	3.58
SVP-lin	Flujo de Probabilidad ODE	196.62596	1.66106	3.23
SVP-lin	Predictor-Corrector	198.75253	**1.72313**	6.74
SVP-cos	Euler-Maruyama	186.91702	1.58389	3.86
SVP-cos	Integrador Exponencial	**159.66022**	1.53126	4.15
SVP-cos	Flujo de Probabilidad ODE	192.79702	**1.65876**	3.95
SVP-cos	Predictor-Corrector	187.08001	1.6044	7.60

Tabla C.4: Desempeño de imputación en CIFAR-10.

C.5. Análisis de Colorización

La Tabla C.5 presenta los resultados de la tarea de colorización en CIFAR-10 utilizando diferentes arquitecturas y muestreadores.

Arquitectura	Muestreador	FID	IS	Tiempo (s)
VE	Euler-Maruyama	193.38832	1.39232	3.75
VE	Integrador Exponencial	192.45786	1.39051	3.92
VE	Flujo de Probabilidad ODE	**175.30922**	**1.52886**	4.07
VE	Predictor-Corrector	197.24849	1.44014	8.08
VP-lin	Euler-Maruyama	163.83663	1.51802	3.75
VP-lin	Integrador Exponencial	**163.4328**	1.51781	3.58
VP-lin	Flujo de Probabilidad ODE	167.18517	1.47428	3.41
VP-lin	Predictor-Corrector	169.84598	**1.55492**	7.35
VP-cos	Euler-Maruyama	171.91384	1.51534	5.13
VP-cos	Integrador Exponencial	170.58923	1.508	4.57
VP-cos	Flujo de Probabilidad ODE	**166.17195**	1.45098	4.02
VP-cos	Predictor-Corrector	174.45213	**1.54554**	8.41
SVP-lin	Euler-Maruyama	**167.22912**	1.55684	3.43
SVP-lin	Integrador Exponencial	167.55852	1.55208	3.55
SVP-lin	Flujo de Probabilidad ODE	173.00752	**1.56613**	3.45
SVP-lin	Predictor-Corrector	169.24731	1.55954	6.93
SVP-cos	Euler-Maruyama	169.26925	1.5374	4.79
SVP-cos	Integrador Exponencial	**166.53801**	1.52699	4.65
SVP-cos	Flujo de Probabilidad ODE	171.6253	**1.60779**	3.99
SVP-cos	Predictor-Corrector	168.95943	1.56237	7.86

Tabla C.5: Desempeño de colorización en CIFAR-10.

Observaciones importantes:

1. Los modelos VP-lin y VP-cos muestran un rendimiento superior en tareas de colorización, con los valores FID más bajos y los IS más altos.

2. Los modelos SVP tienen un rendimiento notablemente inferior en esta tarea, con valores FID significativamente más altos y IS más bajos.

3. Dentro de cada arquitectura, el muestreador Predictor-Corrector tiende a mejorar ligeramente el IS, pero a veces a expensas de un ligero aumento en FID.

4. Los tiempos de colorización son generalmente mayores que los tiempos de generación pura,

reflejando la complejidad adicional de preservar la estructura mientras se infiere información de color.

C.6. Comparación entre Conjuntos de Datos Completos y Parciales

La Tabla C.6 compara el rendimiento de modelos entrenados en conjuntos de datos completos versus parciales.

Arquitectura	Dataset	Completo	FID	IS	BPD
VE	MNIST	Sí	136.60354	**1.44729**	1.05894
VE	MNIST	No	101.93589	1.1981	**0.40551**
VP-lin	MNIST	Sí	108.8496	1.36853	1.33898
VP-lin	MNIST	No	**77.43**	1.32365	0.644
VE	CIFAR-10	Sí	240.34489	1.26761	**0.04346**
VE	CIFAR-10	No	287.02581	1.30363	0.05282
VP-cos	CIFAR-10	Sí	**223.01626**	**1.42277**	0.08328
VP-cos	CIFAR-10	No	243.56972	1.33851	0.10169

Tabla C.6: Comparación entre conjuntos de datos completos y parciales.

Observaciones clave:

1. Para MNIST, los modelos entrenados en conjuntos parciales logran valores FID notablemente más bajos y BPD reducidos, sugiriendo que la especialización en un subconjunto de dígitos permite una mejor modelación de su distribución.
2. Sin embargo, para CIFAR-10, los modelos entrenados en el conjunto completo tienden a obtener mejores valores FID, posiblemente debido a la mayor complejidad y diversidad de este conjunto de datos.
3. Los valores IS son generalmente más altos para los modelos entrenados en conjuntos completos en CIFAR-10, indicando una mayor diversidad en las imágenes generadas.
4. El BPD es consistentemente más bajo para modelos entrenados en conjuntos parciales en MNIST, reflejando la menor entropía en estos subconjuntos más especializados.

C.7. Análisis de la Métrica BPD

La Tabla C.7 examina cómo varía el BPD entre diferentes arquitecturas y conjuntos de datos.

Arquitectura	Dataset	BPD (Completo)	BPD (Parcial)	Relativo
VE	MNIST	**1.05894**	**0.40551**	2.61x
VP-lin	MNIST	1.33898	0.644	2.08x
VP-cos	MNIST	5.3626	2.22936	2.41x
SVP-lin	MNIST	1.35545	0.66697	2.03x
SVP-cos	MNIST	4.03504	2.25963	1.79x
VE	CIFAR-10	**0.04346**	**0.05282**	0.82x
VP-lin	CIFAR-10	0.04441	0.0538	0.83x
VP-cos	CIFAR-10	0.08328	0.10169	0.82x
SVP-lin	CIFAR-10	0.04441	0.05539	0.80x
SVP-cos	CIFAR-10	0.08272	0.10172	0.81x

Tabla C.7: Análisis de BPD por arquitectura y dataset.

Esta tabla revela un fenómeno interesante:

1. Para MNIST, el BPD es significativamente mayor (2-2.6 veces) para modelos entrenados en el conjunto completo comparado con conjuntos parciales, reflejando la mayor complejidad de modelar todos los dígitos simultáneamente.
2. Sin embargo, para CIFAR-10 observamos el comportamiento opuesto: los modelos entrenados en el conjunto completo tienen BPD más bajos que los entrenados en subconjuntos.
3. Los modelos con programación coseno (VP-cos, SVP-cos) muestran valores BPD consistentemente más altos para MNIST, lo que podría indicar que están capturando más detalles o incertidumbre en la distribución.
4. La métrica BPD parece comportarse de manera diferente entre conjuntos de imágenes en escala de grises (MNIST) y a color (CIFAR-10), sugiriendo que su interpretación debe ajustarse según el tipo de datos.

C.8. Efecto del Muestreador en el Tiempo de Ejecución

La Tabla C.8 analiza cómo diferentes muestreadores afectan el tiempo de ejecución para tareas de generación en CIFAR-10.

Observaciones:

1. El muestreador Predictor-Corrector es consistentemente el más costoso computacionalmente, requiriendo aproximadamente el doble de tiempo que otros muestreadores.

Arquitectura	Muestreador	Tiempo (s)	Tiempo Relativo
VP-cos	Euler-Maruyama	2.37	1.00x
VP-cos	Integrador Exponencial	2.38	1.00x
VP-cos	Flujo de Probabilidad ODE	2.56	1.08x
VP-cos	Predictor-Corrector	4.49	1.89x
SVP-lin	Euler-Maruyama	2.06	1.00x
SVP-lin	Integrador Exponencial	2.21	1.07x
SVP-lin	Flujo de Probabilidad ODE	1.95	0.95x
SVP-lin	Predictor-Corrector	4.25	2.06x

Tabla C.8: Tiempos de ejecución por muestreador en CIFAR-10.

2. Los muestreadores Euler-Maruyama y Integrador Exponencial tienen tiempos de ejecución muy similares en todas las arquitecturas.

3. El muestreador Flujo de Probablidad ODE muestra tiempos variables dependiendo de la arquitectura, siendo ligeramente más rápido que Euler-Maruyama para algunas configuraciones (SVP-lin) y más lento para otras (VP-cos).

4. La elección del muestreador tiene un impacto más significativo en el tiempo de ejecución que la elección de la arquitectura de difusión.

C.9. Comparativa de Métodos por Tarea

La Tabla C.9 compara el rendimiento de diferentes métodos (generación, imputación, colorización) utilizando la arquitectura VP-lin y el muestreador Euler-Maruyama en CIFAR-10.

Método	Condicional	FID	IS	Tiempo (s)
Generación	Sí	249.48396	**1.48121**	3.24
Generación	No	**215.4853**	1.40876	1.84
Colorización	Sí	**163.83663**	1.51802	3.63
Colorización	No	165.6309	**1.51938**	2.25
Imputación	Sí	201.2792	1.68167	3.24
Imputación	No	**188.08764**	**1.69547**	1.96

Tabla C.9: Comparación de métodos en CIFAR-10.

Observaciones clave:

1. La tarea de colorización logra los valores FID más bajos, indicando que preservar la estructura de la imagen ayuda a mantener la similitud con la distribución original.

2. La imputación muestra los valores IS más altos, posiblemente porque esta tarea permite mayor creatividad en las regiones imputadas mientras mantiene restricciones estructurales del resto de la imagen.
3. La generación pura tiene los peores valores FID, reflejando la dificultad de generar imágenes completamente nuevas que coincidan con la distribución real.
4. Las variantes condicionales generalmente requieren más tiempo de ejecución, pero este aumento es moderado (aproximadamente 1.5-1.8 veces).

C.10. Conclusiones

Del análisis exhaustivo presentado en este apéndice, podemos extraer varias conclusiones importantes:

1. **Arquitectura de Difusión**: Los modelos VP (tanto lineales como coseno) tienden a ofrecer el mejor equilibrio entre calidad (FID, IS) y eficiencia computacional para la mayoría de las tareas, como se evidencia en la Tabla C.1.
2. **Muestreadores**: El muestreador Predictor-Corrector generalmente produce los mejores resultados en términos de calidad de imagen, pero a costa de un tiempo de ejecución aproximadamente dos veces mayor, como se muestra en la Tabla C.2. Para aplicaciones donde el tiempo es crítico, Euler-Maruyama ofrece un excelente compromiso.
3. **BPD como Métrica**: El comportamiento del BPD varía significativamente entre conjuntos de datos en escala de grises (MNIST) y a color (CIFAR-10), como se observa en la Tabla C.7, sugiriendo que debe interpretarse con precaución y en contexto. En MNIST, los valores más altos de BPD para modelos entrenados en conjuntos completos reflejan la mayor complejidad de modelar múltiples clases.
4. **Generación Condicional vs. Incondicional**: La generación condicional generalmente mejora el IS pero puede degradar el FID, especialmente en MNIST, como se evidencia en la Tabla C.3. Esta compensación debe considerarse según los requisitos específicos de la aplicación.
5. **Tareas Específicas**: Para tareas específicas como colorización (Tabla C.5) e imputación (Tabla C.4), ciertos modelos muestran ventajas claras. VP-lin y VP-cos son particularmente efectivos para colorización, mientras que VE muestra fortalezas en imputación.
6. **Conjuntos de Datos Parciales vs. Completos**: El entrenamiento en subconjuntos específicos puede mejorar significativamente el rendimiento para MNIST, como se muestra en la Tabla C.6, pero este beneficio no se traduce a CIFAR-10, posiblemente debido a la mayor variabilidad inherente de este conjunto de datos.

Un csv con todas las mediciones puede encontrarse en los anexos, junto con el código utilizado para generar los resultados.