

Título

Antonio Molner Domenech

Trabajo de Fin de Grado
Ingeniería Informática

Supervisado por:
Alberto Guillén Perales



Universidad de Granada, España
Junio 2020

Índice general

Listado de figuras	4
Listado de tablas	4
1. Objetivos	5
1.1. Alcance de los objetivos	6
2. Introducción	8
2.1. El problema de la reproducibilidad	8
2.2. Clasificación de partículas	10
3. Fundamentos y estado del arte	11
3.1. Nomenclatura	11
3.1.1. Acrónimos	13
3.2. Reproducibilidad	14
3.2.1. Tipos de reproducibilidad	16
3.2.2. Aspectos críticos	18
3.3. Proceso de ciencia de datos y deuda técnica	20
3.3.1. Deuda técnica	21
3.3.2. Anti-patrones	24
3.4. <i>Machine Learning Operations (MLOps)</i>	25
3.4.1. DevOps. Definición	27
3.4.2. <i>DevOps</i> aplicado al <i>Machine Learning</i>	28
3.4.3. <i>MLOps</i> y Reproducibilidad	32
3.5. Estado del Arte	33
3.5.1. Herramientas para la reproducibilidad	33
3.5.2. Herramientas para <i>MLOps</i>	34
3.5.3. Análisis de rayos gammas	37

3.6.	Modelos utilizados: Redes neuronales	37
3.6.1.	Algoritmo de propagación hacia atrás	39
3.6.2.	Autoencoders	41
3.6.3.	Autoencoders apilados	48
3.6.4.	Aplicaciones de los autoencoders	49
4.	Planificación del trabajo	51
5.	Presupuesto	52
6.	Diseño y desarrollo del <i>framework</i>	53
6.1.	Herramientas utilizadas	55
6.2.	Estructura general	56
6.3.	Instrumentalización de código y ejecución del código	57
6.3.1.	Ejecución de experimentos en Docker o Ray	60
6.4.	Tracking de experimentos	60
6.5.	Hiperparametrización y entrenamiento distribuido	62
6.6.	Sistema de notificaciones y callbacks	64
6.7.	Interfaz Web	64
6.8.	Desarrollo del framework	64
6.9.	Futuro desarrollo	64
7.	Experimentos	65
7.1.	Definición del problema	65
7.1.1.	Historia	65
7.1.2.	Definición formal del problema	68
7.2.	Procedimiento	70
7.3.	Modelos considerados	74
7.3.1.	SVM	75
7.3.2.	Xgboost	76
7.3.3.	Autoencoder V1	79
7.3.4.	Autoencoder V2	81
7.3.5.	Autoencoder Variacional	83
7.3.6.	Construcción del modelo	86
7.4.	Resultados	88
7.4.1.	SVM	89

7.4.2.	Xgboost	91
7.4.3.	Autoencoder v1	95
7.4.4.	Autoencoder v2	100
7.4.5.	Autoencoder Variacional	104
8.	Anexo 1. APSAC: ml-experiment: A Python framework forreproducible data science	108
9.	Anexo 2. Manual de Usuario	114
	Referencias	140

Listado de figuras

Índice de figuras

3.1.	Resultados de la encuesta sobre reproducibilidad. Fuente: [24]	15
3.2.	Solamente una fracción pequeña es dedicada al código de ML. El resto de código de arquitectura es necesario, y complejo. Fuente: [36]	24
3.3.	El desarrollo de sistemas de ML es complejo e implica varios pasos bien diferenciados. MLOps tiene como objetivo mejorar cada uno de los pasos, pero especial aquellos que corresponden a la etapa de Operaciones. Fuente: [45]	28
3.4.	Tabla que resume los aspectos claves de la adopción de MLOps en la industria a diferentes niveles según el modelo de madurez descrito en esta sección. Fuente: [46]	31
3.5.	Diagrama de una neurona, también llamada unidad. Fuente: [59]	37
3.6.	Arquitectura básica de un autoencoder de una sola capa para el codificador y el decodificador. Fuente: [65]	42
3.7.	Ilustración de como el truco de la reparametrización hace el proceso de muestreo de \mathbf{z} entrenable. Fuente: Dispositiva 12 en el workshop de Kingma para NIPS 2015	47
6.1.	Diagrama de la estructura general de <i>ml-experiment</i>	56

7.1.	Cascada atmosférica extensa. (Observatorio Pierre Auger)	66
7.2.	Mapa del observatorio de Pierre Auger. Cada punto negro representa un detector WCD	67
7.3.	Todos los experimentos ejecutados con sus parámetros, métricas, artefactos, y otros metadatos, se almacenan en un servidor de MLFlow en local	71
7.4.	Arquitectura del autoencoder variacional implementado.	
	Fuente propia	83
7.5.	SVM. C vs Validation accuracy	90
7.6.	SVM. Kernel vs Validation accuracy	90
7.7.	SVM. Gamma vs Validation accuracy	91
7.8.	SVM. Parameters comparison	91
7.9.	Xgboost. Gamma vs Validation accuracy	92
7.10.	Xgboost. Number of trees vs Validation accuracy	93
7.11.	Xgboost. Learning rate vs Validation accuracy	93
7.12.	Xgboost. Min child weight vs Validation accuracy	94
7.13.	Xgboost. Max depth vs Validation accuracy	94
7.14.	Xgboost. Subsample vs Validation accuracy	95
7.15.	Xgboost. Parameters comparison	95
7.16.	Autoencoder v1. Dimensión del código vs Validation accuracy	96
7.17.	Autoencoder v1. Tied-weights vs Validation accuracy	97
7.18.	Autoencoder v1. Probabilidad de descarte (dropout) vs Validation accuracy	97
7.19.	Autoencoder v1. Restricción de ortogonalidad vs Validation accuracy	98
7.20.	Autoencoder v1. Restricción UnitNorm vs Validation accuracy	98
7.21.	Autoencoder v1. Ratio de aprendizaje vs Validation accuracy	99
7.22.	Autoencoder v1. OneCycle vs Validation accuracy	99
7.23.	Autoencoder v1. Parameters comparison	100
7.24.	Autoencoder v2. Dimensión del código vs Validation accuracy	101
7.25.	Autoencoder v2. Probabilidad de descarte (dropout) vs Validation accuracy	101
7.26.	Autoencoder v2. Pesos ortogonales vs Validation accuracy . .	102
7.27.	Autoencoder v2. Restricción UnitNorm vs Validation accuracy	102
7.28.	Autoencoder v2. Ratio de aprendizaje vs Validation accuracy	103

7.29. Autoencoder v2. Tied-weights vs Validation accuracy	103
7.30. Autoencoder v2. One-cycle vs Validation accuracy	104
7.31. Autoencoder v2. Parameters comparison	104
7.32. Autoencoder Variacional. Dimensión de la capa intermedia vs Validation accuracy	105
7.33. Autoencoder Variacional. Dimensión del espacio latente vs Validation accuracy	105
7.34. Autoencoder Variacional. Ratio de aprendizaje vs Validation accuracy	106
7.35. Autoencoder Variacional. One-cycle vs Validation accuracy .	106
7.36. Autoencoder Variacional. Activation vs Validation accuracy	107
7.37. Autoencoder Variacional. Parameters comparison	107

Listado de tablas

Índice de cuadros

7.3.	Información general sobre el experimento SVM	75
7.4.	Espacio de hiperparámetros para el experimento SVM . . .	75
7.5.	Información general sobre el experimento XGBoost.	77
7.6.	Espacio de hiperparámetros para el experimento XGBoost. .	77
7.7.	Información general sobre el experimento Autoencoder V1 .	80
7.8.	Espacio de hiperparámetros para el experimento Autoencoder V1	81
7.9.	Información general sobre el experimento Autoencoder V2 .	82
7.10.	Espacio de hiperparámetros para el experimento Autoencoder V1	82
7.11.	Información general sobre el experimento Autoencoder Varia- cional	84
7.12.	Espacio de hiperparámetros para el experimento Autoencoder Variacional	84
7.13.	Resultados para los mejores modelos de cada tipo sobre el conjunto de test. (Las métricas están redondeadas a 5 deci- males).	88

Capítulo 1

Objetivos

El objetivo de este proyecto es el de desarrollar un marco de trabajo para *Machine Learning* enfocado en la reproducibilidad y buenas prácticas. Por otro lado, como objetivo secundario tenemos la aplicación de dicho framework para resolver un problema real.

De un modo más específico, los principales objetivos son:

- Diseño e implementación de un framework de reproducibilidad: El desarrollo de una herramienta que permita instrumentalizar proyectos de *Machine Learning* con mínimo esfuerzo, orientada a mantener unas buenas prácticas de desarrollo y seguir una filosofía *MLOps* (ver *Fundamentos*). Dentro de este objetivo, de manera secundaria, incluimos una contribución de código a uno de los proyectos de código libre que componen el módulo central de nuestra herramienta, *Mlflow*.
- Especificación de buenas prácticas: La creación de una lista de pautas y requisitos necesarios para hacer reproducible un proyecto. Desde la recolección de datos hasta la gestión de experimentos.
- Aplicación de la herramienta a la resolución de un problema real: Aplicación de diferentes técnicas de *Machine Learning* tradicional y Deep learning para la resolución de un problema importante en física, la clasificación de partículas. El problema consiste en la detección del tipo de partícula primaria de una *cascada de partículas extensa* a partir de

una señal registrada por un detector de partículas que almacena una mezcla de señal electromagnética y muónica. El objetivo es encontrar un buen modelo para el dominio en cuestión, y hacer un uso extensivo de la herramienta y para valorar los beneficios para este caso concreto.

1.1. Alcance de los objetivos

Para el primer objetivo, el alcance incluye el desarrollo integral de una herramienta en Python que permita cumplir con la mayoría de requisitos que consideramos necesarios para que un proyecto sea reproducible fácilmente por la comunidad científica. Esta herramienta debe ser flexible y permitir integrarse con frameworks de *Machine Learning* tradicional o *Deep learning* existentes, así como con proyectos orientados al análisis de datos exclusivamente en lugar de al modelado.

En relación con el primer objetivo, se debe desarrollar una especificación de buenas prácticas basadas en problemas existentes, con el objetivo de reducir aquella deuda técnica que concierne a este tipo de proyectos, tanto durante el desarrollo o experimentación, como en el momento de compartir el trabajo con otras personas. Estas buenas prácticas son bastante comunes en el desarrollo de software, pero no tanto en ciencia de datos, debido, entre otros motivos, a la heterogeneidad de perfiles que componen este campo. Dentro de esta relación entre el desarrollo de software y el desarrollo de proyectos de *Machine Learning* o ciencia de datos en general, se van tener en cuenta también aspectos relacionados con el despliegue e integración de software, lo que se conoce como DevOps, cuya aplicación al machine learning es más bien conocida como MLOps.

El tercer y último objetivo comprende el desarrollo de un proyecto de *Machine Learning* real, enfocado al modelado y a la experimentación. El alcance comprende la parte de análisis de modelado del *proceso de ciencia de datos* (ver Fundamentos) - entendimiento del problema, procesado de datos, modelado, etc. Como objetivo secundario, se profundiza en el desarrollo de los autoencoders, atajando el problema de clasificación desde un enfoque de aprendizaje no supervisado, para finalmente compararlo con el resto de

métodos tradicionales.

Capítulo 2

Introducción

2.1. El problema de la reproducibilidad

Hoy en día, los proyectos de ciencia de datos se desarrollan de una forma desestructurada en la mayoría casos, lo cual lo hacen muy difícil de reproducir [1] . Siendo conscientes de las dificultades que conlleva ser rigurosos con el desarrollo de este tipo de trabajos para asegurar la reproducibilidad, este trabajo presenta un framework que facilita el rastreo de experimentos y la operacionalización del machine learning, combinando tecnologías open source existentes y apoyadas fuertemente por la comunidad. Estas tecnologías incluyen Docker [2], MLFlow [3], y Ray [4], entre otros.

El framework ofrece un flujo de trabajo concreto para el diseño y ejecución de experimentos en un entorno local o remoto. Para facilitar la integración con código existente, se ofrece además un sistema de *tracking* automático para los frameworks de Deep Learning más famosos: Tensorflow [5] , Keras [6], Fastai [7], además de otros paquetes de Machine Learning como Xgboost [8] y Lightgdm [9]. Por otro parte, se ofrece un soporte de primera clase para el entrenamiento de modelos y la hiperparametrización en entornos distribuidos. Todas estas características se hacen accesibles al usuario por medio de un paquete de Python con el que instrumentalizar el código existente, y un Interfaz de Linea de Comandos (CLI) con el que empaquetas y ejecutar trabajos.

La reproducibilidad es un reto en la investigación moderna y produce bastante debate ???, [10]–[12]. Entre los diferentes tipos de trabajos reproducibles, este trabajo se centra en trabajos computacionales, desarrollando un flujo de trabajo específico basado en los principios de Control de Versiones, Automatización, Tracking y Aislamiento del entorno .

- El control de versiones permite rastrear los diferentes ficheros del proyecto y sus cambios, así como facilitar la colaboración.
- Automatizar los procesos, desde ficheros de shell hasta pipelines de alto nivel, permite que otra persona puede reproducir los pasos del trabajo fácilmente. Estos pasos incluyen: creación de ficheros, preprocesado de datos, ajuste de modelos, etc.
- *Tracking* o recolección de información: Durante la ejecución de estos pasos, se generan gráficos, artefactos, nuevos datos, etc. Por este motivo, es necesario proporcionar una forma sistemática de recolectar toda esa información generada y mostrarla de manera accessible desde un único lugar (*Knowledge Center*).

Finalmente, el aislamiento del sistema anfitrión mediante el uso de contenedores o máquinas virtuales, permite ampliar el ámbito de control sobre los experimentos, proporcionando un “escenario común” para la ejecución de los mismo. De otra forma, los factores externos al proyecto, como las versiones de las paquetes de análisis, los drivers de la Unidad de Procesado Gráfico (GPU), o la propia versión del sistema operativo donde se ejecuten pueden incrementar la incertidumbre del experimento [13]. Otra ventaja de aislar las dependencias y la imagen del sistema operativo (entre otros factores), combinado con la automatización de los diferentes procesos, es que que facilita enormemente la ejecución de los experimentos y los hace independientes de la plataforma, evitando tener que instalar las diferentes dependencias, modificar ficheros de configuración, etc. Por no decir que las dependencias del proyecto pueden ser incompatibles con las globales instaladas en el sistema.

2.2. Clasificación de partículas

Uno de los misterios de Astrofísica a día de hoy es la forma en la que se generan los rayos cósmicos de ultra alta energía (UHECRs). Para comprender mejor el comportamiento de estas partículas, el observatorio de Pierre Auger [14] fue construido. Supone un proyecto muy ambicioso y el experimento de mayor magnitud a día de hoy. Un área de 3000 kilómetros cuadrados se ha diseñado y construido para alojar detectores de agua Cherenkov (WCD) [15]. Estos detectores son unos tanques grandes de agua ultra-pura donde se detecta la radiación de Cherenkov, normalmente utilizando [16], [17]. Estos detectores son capaces de medir la señal generada por las partículas mientras viajan a través del agua. Las interacciones de los UHECRs con las moléculas de aire de la atmósfera producen lo que se conoce como *Cascada atmosférica extensa* [14]. Esto ocurre cuando la partícula primaria colisiona con la parte superior de la atmósfera y genera una cascada de partículas secundarias como protones, electrones y muones. Utilizando la señal recogida, los científicos pueden tratar de responder a varias cuestiones: qué tipo de partícula llegó a la atmósfera, de dónde procede, y cómo se originó.

La respuesta a la primera pregunta es uno de los objetivos de este trabajo. Tradicionalmente, la clave para conocer el tipo de primario es el número muones generados en la cascada. Cuando una partícula colisiona en la atmosférica y llega al suelo, esta genera una señal en cada WCD, la cual es una combinación de la señal electromagnética y la muónica de la cascada. Estimar la naturaleza de la partícula incidente utilizando la señal muónica es un desafío con los dispositivos disponibles actualmente. El objetivo en este caso, es el de aplicar técnicas de Machine Learning y Deep Learning para la detección del tipo de Partícula primaria a partir de señales de generadas dentro de los WCDs y capturadas por los .

Capítulo 3

Fundamentos y estado del arte

A lo largo de este capítulo, describiremos principalmente los fundamentos del trabajo y el estado del arte. Para los fundamentos, destacaremos la nomenclatura utilizada, añadiendo un pequeño glosario de términos. Posteriormente, se define el concepto de *reproducibilidad* en los proyectos de investigación basados de *Machine Learning* (ML), y los aspectos críticos de dictaminan cuando un proyecto es reproducible o no. Por otro lado, se define el proceso de ciencia de datos, con una descripción de cada uno de los pasos, y la deuda técnica asociada a dicho proceso.

En las sección de *MLOps* se describe un concepto novedoso sobre un conjunto de buenas prácticas para el desarrollo de proyectos de ciencia de datos que la industria está implementando progresivamente. En las posteriores secciones, se describen varios de los algoritmos de *Machine Learning* y *Deep Learning* utilizados en la experimentación, con especial atención a los *autoencoders*. Finalmente, se hace un repaso del estado del arte para las herramientas para reproducibilidad, *MLOps* y para los algoritmos implementados.

3.1. Nomenclatura

El área de la ciencia de datos, *Machine Learning* y *MLOps*, se hace uso de una terminología concreta [18]–[20], basada principalmente en la terminología de

Aprendizaje estadístico, Desarrollo Software, y DevOps en el caso de MLOps.

En esta sección se desarrollan algunos de los términos más utilizados:

- **Canalización o Pipeline:** Consiste en una definición e implementación exhaustiva de las diferentes etapas de un proceso. Un pipeline se puede definir como un script, conjunto de scripts, ficheros de configuración, etc. Además, permite la ejecución del proceso de manera automatizada .
- **Conjuntos de datos** – Colección de datos estructurada que se utiliza para entrenar modelos de Machine Learning (ML), para análisis, o para inferencia. Aunque los conjuntos de datos pueden contener información de diferentes fuentes, el conjunto en sí tiene una sola tesis central (propósito).
- **Experimento** – Un proceso o actividad que permite testear una hipótesis y validarla iterativamente. Los resultados de una cierta iteración deben ser almacenados para poder ser evaluados, comparados, y monitoreados para propósitos de auditoría.
- **Artefacto:** Pieza de información generada en un experimento. Incluye modelos entrenados, datos generados, imágenes, documentación auto-generada, etc.
- **Modelo** – Es un caso concreto de artefacto que permite predecir valores en un sistema ML o bien, permite ser usado como pieza de otro modelo (mediante *ensamblado* o *transferencia de conocimiento*).
- **Repositorio:** Fuente de código común para la organización. Se entiende por repositorio aquel directorio gestionado por un control de versiones (como Git). Este repositorio puede contener implementaciones de pipelines, modelos, datos, ficheros de configuración, decisiones de dependencias, entre otras cosas.
- **Registro de modelos** – Almacén centralizado donde se almacenan los diferentes modelos generados en el ciclo de vida de un proyecto de ciencia de datos.

- **Espacio de trabajo:** Los científicos de datos desarrollan sus actividades de manera colaborativa o individual. Un entorno de trabajo comprime aquellas herramientas e información necesarias para el desempeño de un rol específico. Un entorno típico de ciencia de datos consiste en un IDE donde escribir código, y un conjunto de herramientas locales o en línea que permiten acceder a los datos, modelos, etc. Un ejemplo de espacio de trabajo típico puede ser *Jupyter* [21] para desarrollo de código, *Amazon Sagemaker* [22] para la gestión de modelos, y *DataGrip* [23] para el acceso a base de datos.
- **Entorno objetivo:** El entorno de despliegue de los sistemas de ML, es decir, el entorno donde el modelo va a generar información (en forma de predicciones) para el consumo por el usuario. Alguno de los entornos objetivos más comunes son:
 - Servicio web, como parte de un backend propio, o como microservicio. Se implementa a partir de una API REST, GRPC o cualquier otro protocolo web.
 - Dispositivos finales. El modelo se integra dentro del dispositivo y se hacen las predicciones localmente. Útil para dispositivos con conectividad limitada, IoT, etc.
 - Parte de un sistema de predicción por lotes.

3.1.1. ACRÓNIMOS

- **ML:** Machine Learning
- **DL:** Deep Learning
- **PCA:** Análisis de Componentes Principales
- **WCD:** Water Cherenkov Detector
- **GCP:** Google Cloud Platform
- **GPU:** Unidad de Procesado Gráfico
- **CI:** Integración Continua

- **CD:** Entrega Continua
- **IA:** Inteligencia Artificial
- **CLI:** Interfaz de Linea de Comandos
- **VAE:** Autoencoder variacional
- **SVM:** Máquina de Soporte Vectorial
- **TPE:** Tree-structured Parzen Estimator
- **HPO:** Optimización de Hiperparámetros

3.2. Reproducibilidad

Según una encuesta realizada por Nature, una de las revistas científicas más prestigiosas a nivel mundial, más del 70 por ciento de los 1,576 investigadores encuestados no han podido reproducir alguno de sus propios experimentos. Además, los datos son claros, la mayoría piensa que existe una *crisis de reproducibilidad* (ver Figura 3.1).

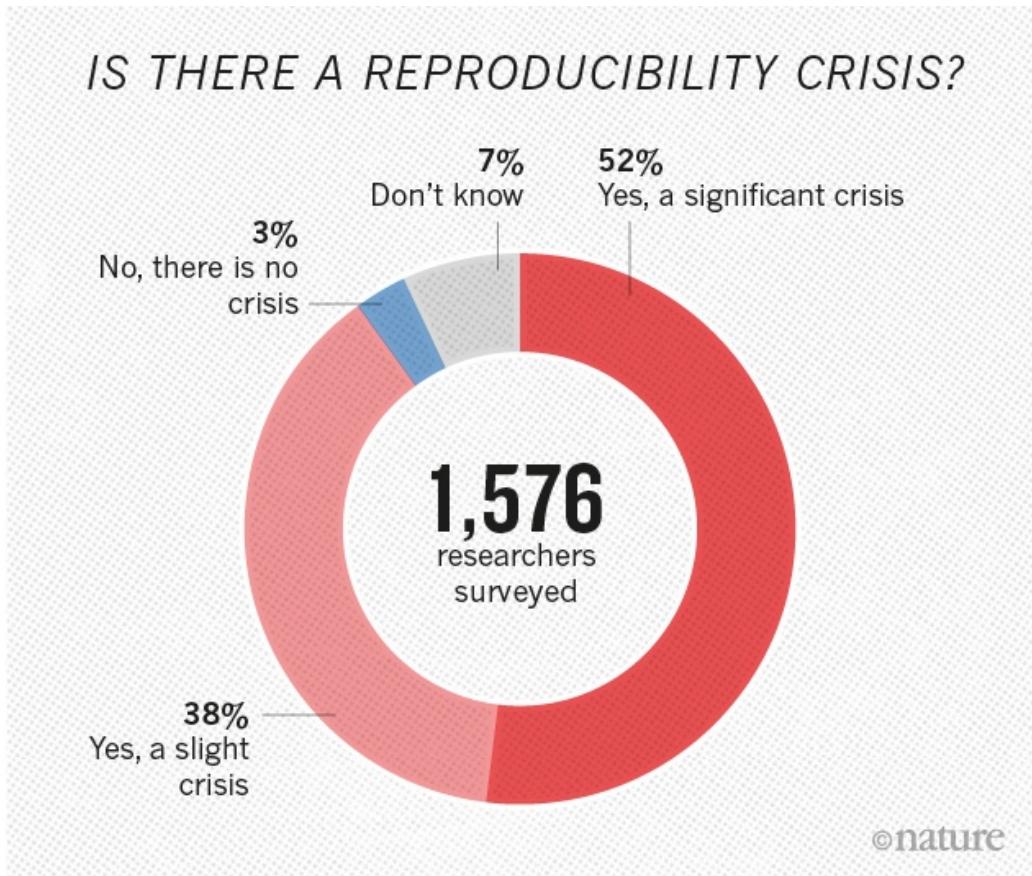


Figura 3.1: Resultados de la encuesta sobre reproducibilidad. Fuente: [24]

A día de hoy, los estudios suelen ofrecer los resultados en forma de gráficas y tablas, pero en muchos casos carecen de la información necesaria para poder contrastar los resultados. Esta información suele ser, el entorno de ejecución, los datos originales y la implementación de los propios métodos (modelos, algoritmos, etc) entre otros. Para aumentar la accesibilidad de los estudios, los investigadores deben asegurarse de ofrecer esta información además de las gráficas y tablas.

La verificación independiente tiene como objetivo la confirmación de credibilidad y la extensión del conocimiento en un área. La investigación relativa al *Machine Learning* o a otras áreas donde se haga uso del mismo, no está exenta de este requisito de la investigación científica. Por tanto, adoptando un flujo de trabajo reproducible, estamos ofreciendo a la audiencia las herramientas necesarias que demuestran las decisiones tomadas y que permiten

validar nuestros resultados. Por otro lado, para que un estudio computacional pueda ser reproducido correctamente por un investigador independiente es necesario el acceso completo a los datos, código, parámetros de los experimentos, información sobre el entorno de ejecución, etc.

Otro motivo de interés para la búsqueda de la reproducibilidad es el de facilitar el uso de nuestros métodos por el resto de la comunidad científica o incluso en aplicaciones comerciales. Ofreciendo acceso a los datos y al código, como se ha comentado antes, permitimos que nuestros métodos se puedan aplicar a otros problemas, tanto en investigación como para fines comerciales, así como facilita la extensión de nuestro trabajo.

En los últimos años nos hemos encontrado con muchos casos de publicaciones científicas que muestran resultados difíciles o incluso imposibles de reproducir. Este fenómeno se conoce como la crisis de la reproducibilidad, donde incluso estudios prominentes no se pueden reproducir [24], [25]. Este fenómeno ha estudiado de manera extensiva en otros campos, pero en el área del *Machine Learning* está tomando últimamente mucha importancia. Esto es debido a que tradicionalmente, los experimentos científicos se deben describir de tal forma que cualquiera pueda replicarlos, sin embargo, los experimentos computacionales tienen varias complicaciones que los hacen particularmente difíciles de replicar: versiones de software, dependencias concretas, variaciones del hardware, etc.

Con motivo de esta crisis de la reproducibilidad que afecta en gran medida a AI/ML, conferencias como NeurIPS han optado por añadir este factor en su proceso de revisión, e implementan políticas para alentar el código compartido [26]. Por otro lado, algunos autores (incluido nosotros) han propuesto herramientas para facilitar la reproducibilidad, mientras que otros han propuesto una serie de reglas o heurísticas que para evaluar este aspecto [10], [27], [28].

3.2.1. TIPOS DE REPRODUCIBILIDAD

Para poder atajar de una manera directa y eficiente el problema de la reproducibilidad es necesario separarla en diferentes niveles [29]. Esta separación

nos permite desarrollar una serie de buenas prácticas y herramientas específicas para cada nivel, así como ver, de una manera clara, cuales aspectos se pueden recoger en un *framework* común, y cuales son inherentes del estudio científico en cuestión. Entre los niveles de reproducibilidad podemos destacar:

- **Reproducibilidad computacional.** Cuando se provee con información detallada del código, software, hardware y decisiones de implementación.
- **Reproducibilidad empírica.** Cuando se provee información sobre experimentación empírica no computacional u observaciones.
- **Reproducibilidad estadística.** Cuando se provee información sobre la elección de los test estadísticos, umbrales, p-valores, etc.

Una vez hecha separación del problema en tres capas, podemos ver claramente que la reproducibilidad computacional debe ser nuestro objetivo a la hora de desarrollar el *framework*. Mientras que la reproducibilidad empírica se puede conseguir en mayor medida, haciendo los datos accesibles, la reproducibilidad estadística se consigue mediante el desarrollo de un diseño inicial del estudio. En este diseño se especifica la hipótesis base, las asunciones del problema, los test estadísticos a realizar, y los p-valores correspondientes. El establecer las bases estadísticas sobre las que se va a desarrollar el estudio de antemano, nos puede ayudar además a evitar problemas como el *p-hacking* [30].

Por otro lado, el término reproducibilidad además de poder descomponerse según la información o parte del trabajo que se esté tratando, llamémosla la escala o eje horizontal, también se puede descomponer en otro eje, llamémosle vertical, que indica como de replicable y reproducible es un estudio en su conjunto. Los niveles de esta nueva escala son los siguientes [31]:

- **Investigación revisable.** Las descripciones de los métodos de investigación pueden ser evaluados de manera independiente y los resultados juzgados. Esto incluye tanto los tradicionales *peer-review*, *community-review*, y no implica necesariamente reproducibilidad.

- **Investigación replicable.** Se ponen a disposición del público las herramientas que necesarias para replicar los resultados, por ejemplo se ofrece el código de los autores para producir las gráficas que se muestran en la publicación. En este caso, las herramientas pueden tener un alcance limitado, ofreciendo los datos ya procesados y esenciales, así como ofreciéndolas mediante petición exclusivamente.
- **Investigación confirmable.** Las conclusiones del estudio se pueden obtener sin el uso del software proporcionado por el autor. Pero se debe ofrecer una completa descripción de los algoritmos y la metodología usados en la publicación y cualquier material complementario necesario.
- **Investigación auditabile.** Cuando se registra la suficiente información sobre el estudio (incluidos datos y programas informáticos) para que la investigación pueda ser defendida posteriormente si es necesario o para llevar a cabo una resolución en caso de existir diferencias entre confirmaciones independientes. Esta información puede ser privada, como con los tradicionales cuadernos de laboratorio.
- **Investigación abierta o reproducible.** Investigación auditabile disponible abiertamente. El código y los datos se encuentran lo suficientemente bien documentados y accesibles al público para que la parte computacional se pueda auditar, y los resultados del estudio se puedan replicar y reproducir de manera independiente. También debe permitir extender los resultados o aplicar el método desarrollado a nuevos problemas.

3.2.2. ASPECTOS CRÍTICOS

Una vez hemos definido los diferentes niveles de reproducibilidad, vamos a definir los aspectos que consideramos críticos para lograr una investigación *abierta o reproducible* [1], [10], [27], [28], [31], [32].

- **Conjunto de datos:** La información sobre la localización y el proceso de extracción de los datos. Este factor es determinante a la hora de

hacer un estudio reproducible. El objetivo es el de facilitar los datos y/o la forma de extraerlos. En caso de que los datos no sean accesibles públicamente, o que los datos que se ofrezcan no sean los extraídos en crudo, estaríamos ante un *estudio replicable*, pero no reproducible.

- **Preprocesado de datos:** En este aspecto se recogen los diferentes pasos del proceso de transformación de los datos. Un investigador independiente debería ser capaz de repetir los datos de preprocesado fácilmente. Sería también interesante incluir datos ya preprocesados con los que comparar y validar que las transformaciones se han realizado correctamente. Estos procedimientos no son sencillos de documentar ni de compartir. En algunas ocasiones, las transformaciones se realizan en software privativos o utilizando una interfaz gráfica. En esos casos, en lugar de ofrecer los scripts de preprocesado, sería más interesante dar una descripción detallada de como los datos se han transformado. Además, sugerimos favorecer las herramientas de código libre en caso de que existan como alternativa a algunas de las herramientas privadas.
- **Partición de los datos:** En caso de que los datos se separen, por ejemplo para ajustar un modelo y validarla, es necesario proporcionar los detalles de como se ha realizado esta separación. En el caso de que dicha separación sea aleatoria, como mínimo se debe proporcionar la semilla y el tipo de muestreo (estratificado o no, por ejemplo). Aunque preferiblemente, todo este procedimiento debe estar recogido en un script.
- **Ajuste del modelo:** Corresponde a toda la información relativa al ajuste de un modelo. En este caso, es necesario hacer disponible toda la información posible en relación a este proceso y a las decisiones tomadas. La información mínima que se debe proporcionar es:
 1. Parámetros del experimento
 2. Métodos propuestos: detalles de implementación, algoritmos, código, etc (si es aplicable).

- **Evaluación del modelo:** Información sobre como se evalúa un modelo entrenado. Información similar al punto anterior se aplica aquí.
- **Control de la estocásticidad:** La mayoría de operaciones en *Machine Learning* tienen un factor de aleatoriedad. Por tanto, es esencial establecer los valores de las semilla que controlar dichos procesos. La mayoría de herramientas de cálculo científico ofrecen algún método para establecer la semilla del generador de números aleatorios.
- **Entorno software:** Debido al hecho de que los paquetes/módulos de software están en continuo desarrollo y sufren posibles alteraciones de los algoritmos internos, es importante que los detalles del entorno de software utilizado: módulos, paquetes y números de versión..., estén disponible.
- **Entorno hardware:** Algunos estudios, sobre todo los que contienen grandes cantidades de datos, son reproducibles exclusivamente cuando se ejecutan en una cierta máquina, o al menos, cuando se cumplen unos requisitos de hardware determinados. Otro problema que surge en algunos casos y que está estrechamente relacionado con el punto anterior, es el de las versiones de los drivers. Por este motivo, se requiere una correcta documentación de los recursos utilizados, tanto GPU como CPU, así como de las versiones de sus drivers correspondientes.

3.3. Proceso de ciencia de datos y deuda técnica

La mayoría de proyectos de ciencia de datos recogen una serie de pasos distinguidos. Una vez definido el caso comercial (el producto), y la métrica que mide el éxito, los pasos para llevar a cabo un proyecto de *Machine Learning* son los siguientes [33], [34]:

- **Extracción de datos:** Se seleccionan e integran datos de diferentes fuentes que sean relevantes para el problema.
- **Análisis de datos:** En este paso se realiza un análisis exploratorio (EDA) con el fin de comprender el modelo de datos, realizar asunciones,

identificar posibles características relevantes, y preparar un plan para la ingeniería de características y el preprocesado de datos.

- **Preparación de los datos:** Se preparan los datos para la tarea en cuestión. Se realizan las particiones de datos, se limpian y transforman los mismos para adaptarlos al problema, y se lleva a cabo la ingeniería de características. El resultado de este proceso es una serie de conjuntos de datos listos para entrenar, evaluar y validar modelos.
- **Ajuste de modelos:** Aquí se lleva a cabo el entrenamiento de modelos. Se implementan diferentes algoritmos y se realiza un ajuste de hiperparámetros con el fin de obtener el mejor modelo posible.
- **Evaluación de modelos:** Se evalúa el modelo utilizando los conjuntos de validación y/o test.
- **Validación de modelos:** Se realiza una confirmación del rendimiento del modelo para comprobar que es adecuado para la implementación. Para ello se compara su rendimiento predictivo con un modelo de referencia determinado, denominado **baseline**.
- **Entrega o despliegue del modelo:** Se implementa el modelo final en el *entorno de destino* para hacer las predicciones disponibles a los usuarios.
- **Monitorización del modelo:** Se supervisa el rendimiento del modelo con el fin de planificar las siguientes iteraciones.

3.3.1. DEUDA TÉCNICA

Como se puede observar, los pasos de este proceso siguen un orden estricto, lo cual lo hace resonar a un modelo de desarrollo en cascada. Al igual que el resto de aplicaciones del desarrollo software, la deuda técnica es un factor vital a tener en cuenta. Un factor que puede ralentizar enormemente las iteraciones, y que se va acumulando en cada paso del proceso. Además, la alta dependencia que hay en el orden de los pasos del proceso de ciencia de datos, hace muy difícil la refactorización.

La deuda técnica [35] es un concepto acuñado en el desarrollo software para describir aquellas decisiones, que se toman por falta de tiempo o conocimiento, que provocan un coste adicional sobre los nuevos cambios conforme pasan el tiempo. Este término está basado en el concepto de *deuda monetaria*, y al igual que este tipo de deuda, si no se paga temprano, el coste adicional aumenta de manera exponencial (*intereses compuestos*). Algunas de las causas de deuda técnica son: falta de tests, falta de documentación, falta de conocimiento, presión comercial (deadlines irreales), refactorización tardía, entre otros.

Además de la deuda técnica originada por el propio desarrollo software, existe unos elementos particulares al proceso de ciencia de datos que pueden aumentar drásticamente esta deuda [36], [37]:

- **Bucles de retroalimentación:** Este problema ocurre cuando, de manera indirecta, la salida del modelo influencia la entrada al mismo. De esta forma, los sistemas de ML modifican su propio comportamiento conforme pasa el tiempo. Este tipo de errores parecen sencillos de resolver, pero en la práctica, conforme se integran diferentes sistemas la probabilidad de que estos se retroalimenten entre sí es muy alta. Incluso si dos sistemas de ML parecen no estar relacionados, este problema puede surgir. Imagínese dos sistemas que predicen del valor de acciones de un mismo mercado para dos compañías distintas. Mejoras o peor aún, bugs de un sistema, pueden influir en el comportamiento del otro sistema.
- **Cascadas de corrección:** Este problema ocurre cuando el modelo de ML no aprende lo que se esperaba, y se terminan aplicando una serie de parches (heurísticas, filtros, calibraciones, etc.) sobre la salida del modelo. Añadir un parche de este tipo puede ser tentador incluso cuando no hay restricciones de tiempo. El problema principal es que la métrica que el modelo intenta optimizar se descorrelaciona con la métrica general del sistema. Conforme esta capa de heurísticas se vuelve más grande, es difícil reconocer cambios sobre el modelo de ML que mejoraren la métrica final, dificultando de esta forma la iteración y mejora continua.

▪ **Características basura:** Características que no aportan nada al sistema, incluso pueden perjudicar el rendimiento. Algunas de las características basura que podemos encontrar son:

- Características agrupadas: Cuando se agrupan varias características y se evalúan en conjunto, es difícil saber si todas las características aportan, o si simplemente hay algunas que son beneficiosas y otras no.
- e-Características: Algunas características que se añaden mejoran muy poco el rendimiento del modelo. Aunque es tentador añadir este tipo de características, el problema emerge cuando dichas características dejan de mejorar el modelo o incluso lo empeoran cuando los datos cambian mínimamente.
- Características obsoletas: Conforme pasa el tiempo, algunas características se vuelven obsoletas, porque o bien no aportan la información correcta, o bien la información que aportan ya se recoge en otras variables. Para evitar este problema, reevaluar la importancia de las características con el paso del tiempo.

▪ **Deuda de configuración:** Sistemas de ML están compuestos por diferentes partes, cada una con un configuración específica. Los modelos y pipelines en general, deben de ser fácilmente configurables. Además, la organización de ficheros y el sistema de configuración debe facilitar lo siguiente:

- Modificar configuraciones existentes fácilmente
- Comparar y ver claramente la diferencias entre configuraciones de modelos
- Detectar configuraciones redundantes
- Revisión de código sobre las configuración y su inclusión en un control de versiones.

▪ **Deuda de reproducibilidad:** Como se verá en la sección siguiente, es importante que como investigadores, podamos reproducir experimentos y obtener los mismos resultados fácilmente. Aunque en los sistemas ML reales es realmente difícil conseguirlo; debido principalmente a la

naturaleza no determinística de los algoritmos, del entrenamiento en paralelo, y de las interacciones con el mundo exterior.

3.3.2. ANTI-PATRONES

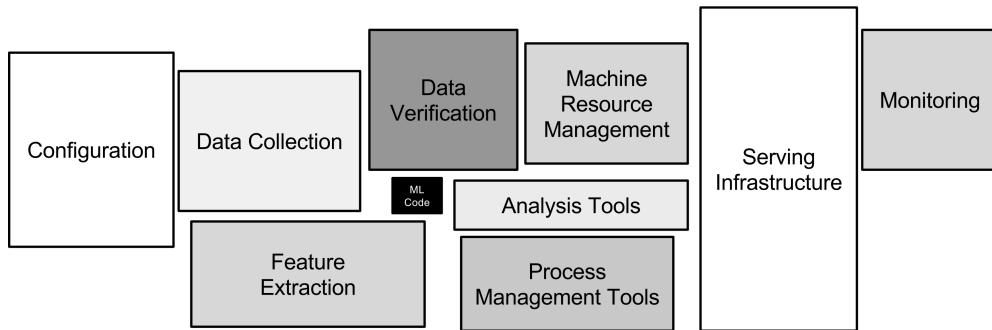


Figura 3.2: Solamente una fracción pequeña es dedicada al código de ML. El resto de código de arquitectura es necesario, y complejo. Fuente: [36]

Sorprendentemente, en la mayoría de sistemas de ML, solamente una pequeña fracción del código está dedicado al entrenamiento y predicción. El resto de código, conocido como *plumbing*, es susceptible a una serie de anti-patrones que se describen a continuación:

- **Código pegamento:** A pesar de que en la comunidad existen numerosos paquetes y soluciones para ML. El utilizar herramientas genéricas puede hacer que el sistema dependa mayoritariamente de ellas. Eso provoca que en algunos casos haya una gran cantidad de código solamente para introducir y extraer datos de estas soluciones *open source*. Si nuestro sistema tienen una gran proporción del código dedicado a adaptar los datos y algoritmos a un paquete de propósito general, deberíamos plantearnos crear una solución propia.
- **Junglas de pipelines:** La mayoría de sistema integran multiples fuentes de información. Estas fuentes de información, así como las transformaciones pertinentes sobre los datos, suelen evolucionar a lo largo del desarrollo. Esto induce a un caso particular de *código pegamento* donde se hace muy complicado poder testear, recuperarse de errores, etc.

Una forma de subsanar este problema, es diseñando el sistema holísticamente (teniendo en cuenta todo el pipeline), en lugar de enfocarse en los pasos intermedios. Además, también sería beneficioso, en la medida de lo posible, aplicar los conceptos de *programación funcional*.

- **Código muerto:** Los proyectos de ML se basan en la experimentación. Al cabo del tiempo, estos sistemas pueden acabar con una gran cantidad de código dedicados experimentos que nunca han visto la luz.
- **Deuda de abstracción:** Los problemas anteriores reflejan una falta de abstracción para los sistemas de ML, como puede ser un lenguaje común de alto nivel para definir las fuentes de datos, modelo y predicciones.
- **Code-smells más comunes:** Algunos de los indicadores de *peligro* en la implementación de sistemas de ML son los siguientes:
 - *Tipos de datos planos:* En un sistema robusto, la información producida en el mismo se almacena enriquecida. Se debe saber si un parámetro de un modelo es un *umbral* o no, si una variable está en escala logarítmica, etc. Así como debe haber claras indicaciones de cómo se ha producido la información y cómo se debe ser consumida.
 - *Multiples lenguajes:* Es tentador utilizar diferentes lenguajes para un mismo sistema de ML cuando hay soluciones o sintaxis conveniente para cada componente. Sin embargo, esto limita la movilidad del capital humano, así como complica el testing.
 - *Prototipos:* Todo sistema de ML parte de un prototipo. Sin embargo, es necesario un código bien testeado y listo para producción en cualquier parte de estos sistemas. Aunque es complicado llevarlo a la práctica cuando existen unas restricciones de tiempo fuertes.

3.4. *Machine Learning Operations (MLOps)*

Durante los últimos años, el papel de la ciencia de datos y del *Machine Learning* ha tomado gran relevancia en la industria. En la actualidad, la

ciencia de datos se utiliza para resolver problemas complejos, y ofrecer una gran variedad de productos de datos: traductores automáticos [38], sistemas de recomendación [39], sistemas de trading de alta frecuencia [40], [41], etc. La ciencia de datos ha podido ser aplicada a una variedad muy amplia de campos, ha aportado valor en cada uno de ellos, incluso ha revolucionado algunas industrias . Para que esto haya sido posible, y para que siga siendo posible, es necesario una gran cantidad de datos, recursos de computación (CPU y GPU) accesibles, hardware optimizado para cálculo científico, así como una activa comunidad de investigadores.

El hecho de que cada vez más industrias estén implementando sistemas de ML como productos o parte de productos comerciales, hace indispensable unos flujos de desarrollo orientados a la industria. La ciencia de datos parte originalmente de la experimentación, no obstante, conforme los sistemas de ML se integran con el resto de componentes de una organización, es necesario aplicar las técnicas y buenas prácticas conocidas en el desarrollo software, con el fin de ofrecer a los usuarios sistemas predictivos con valor comercial y mínimo coste. Los científicos de datos pueden implementar y entrenar modelos localmente, sin conexión a Internet incluso, pero el verdadero desafío consiste en implementar un sistema ML completo, y operarlo en producción de manera continua [42]–[44].

Como se ha detallado en la sección anterior el ciclo de desarrollo de un producto de un sistema ML implica diferentes fases. El código relacionado con la propia implementación y entrenamiento de modelos es mínimo comparado con el resto de código necesario para el desarrollo de estos sistemas (ver Figura 3.2). Además, debido a la necesidad de grandes cantidades de datos y de recursos computacionales amplios, estos sistemas deben incluir otros módulos relativos a la infraestructura: manejos de recursos, monitorización, automatización, etc. Por lo que el desarrollo de sistemas de ML en la industria no consiste solamente en entrenar modelos, o recolectar y procesar datos, sino que requiere de una amplia base de desarrollo software, la cuál, es carente en muchos equipos de ciencia de datos multidisciplinares.

Para poder lidiar con los problemas inherentes a la aplicación de ciencia de datos a la industria, y para poder abstraer a los científicos de datos sobre

la infraestructura, en los últimos años se ha ido desarrollando el concepto de *MLOps*. Las prácticas de la filosofía *MLOps* se fundamentan en DevOps, una filosofía de buenas prácticas para el desarrollo de software. Es por eso que es necesario hacer una breve introducción ha dicho concepto antes de profundizar en *MLOps*.

3.4.1. DEVOPS. DEFINICIÓN

Para poder desarrollar sistemas software complejos, la tendencia actual es utilizar las técnicas de *DevOps*. *DevOps* es un conjunto de prácticas en el desarrollo y operacionalización. Estas prácticas aumentan la velocidad de implementación, reducen los ciclos de desarrollo, y facilitan la entrega de actualizaciones. Entre las prácticas recogidas en este concepto se incluyen:

- **Integración Continua (CI):** Esta práctica de desarrollo software permite a los desarrolladores ejecutar versiones y pruebas automáticas cuando se combinan cambios de código en el repositorio del proyecto. Esto permite validar y corregir errores con mayor rapidez, mejorando así la calidad del software.
- **Integración Continua (CI):** Esta práctica de desarrollo software se basa en la compilación, prueba y preparación automática de artefactos. Estos artefactos se generan automáticamente cuando se producen cambios en el código y se entregan a la fase de producción. De esta forma, las actualizaciones a los usuarios finales se entregar con mínimo esfuerzo. Travis o CircleCI son algunos de los servicios que ofrecen tanto *Integración Continua* como *Entrega Continua*.
- **Microservicios:** La arquitectura de microservicios es un enfoque de diseño que permite crear una aplicación a partir de un conjunto de servicios pequeños. Cada servicio se ejecuta de manera independiente y se comunica con los otros servicios a través una interfaz ligera, normalmente HTTP. Recientemente, algunos otros protocolos de nivel superior como gRPC o GraphQL se están utilizando para la interconexión de estos servicios.

- **Infraestructura como código:** Aprovisionar y administrar infraestructura con técnicas de desarrollo de programación y desarrollo software, como el control de versiones. Algunos servicios como AWS, CloudFormation o Terraform permiten aprovisionar y gestionar infraestructuras utilizando lenguajes de programación o ficheros de configuración.
- **Monitorización y registro:** Monitorizar métricas y registros para analizar el desempeño de las aplicaciones y la infraestructura sobre la experiencia usuario.
- **Comunicación y colaboración:** Uno de los aspectos clave en la filosofía *DevOps* es el incremento de la comunicación y la colaboración en las organizaciones.

3.4.2. *DEVOPS* APlicado al *MACHINE LEARNING*

MLOps se fundamenta en los principios y prácticas de *DevOps*. Nociones, como se ha comentado previamente, orientadas a la eficiencia en el desarrollo: integración y entrega continuos, monitorización, etc. *MLOps* aplica estos principios para la entrega de sistemas de ML a escala, resultando en:

- Tiempo de comercialización de soluciones basadas en ML menor.
- Ratio de experimentación mayor que fomenta la innovación.
- Garantía de calidad, confidencialidad e *IA ética*.

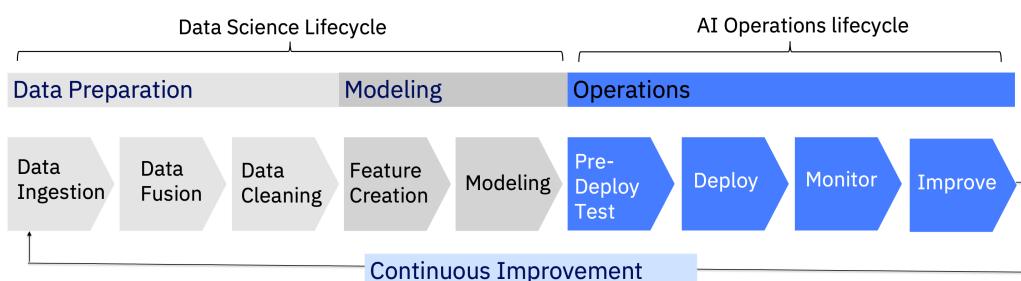


Figura 3.3: El desarrollo de sistemas de ML es complejo e implica varios pasos bien diferenciados. MLOps tiene como objetivo mejorar cada uno de los pasos, pero especial aquellos que corresponden a la etapa de Operaciones. Fuente: [45]

Para poder analizar la interacción entre DevOps y el desarrollo de sistemas de ML, es necesario destacar las tareas claves de este proceso. Teniendo en cuenta el proceso de ciencia de datos descrito en la sección anterior, (también representado en la Figura 3.3) podemos destacar las siguientes tareas:

- **Recolectar y preparar datos:** Generar y preparar los conjuntos de datos para el entrenamiento.
- **Aprovisionar y gestionar la arquitectura:** Establecer los entornos de computación donde se entrenan los modelos y despliegan los modelos.
- **Entrenar modelos:** Desarrollar el código de entrenamiento y evaluación, y ejecutarlos en la infraestructura aprovisionada.
- **Registrar modelos:** Después de la ejecución de un experimento, el modelo resultado se almacena en el *registro de modelos*.
- **Desplegar el modelo:** Validar los resultados del modelo, desplegarlo en el *entorno objetivo*.
- **Operar el modelo:** Operar el modelo en producción monitorizándolo para conocer su rendimiento, detectar *desfases de datos*, alerta de fallos, etc.

Esta secuencia de actividades se corresponde con un *pipeline*. La dificultad principal en el diseño de este pipeline es que cada paso es altamente iterable. Es decir, los modelos necesitan ser modificados, los resultados testeados, se añaden nuevas fuentes de información, etc. El poder iterar de una manera eficiente es fundamental para este tipo de sistemas. Además, existen ciertos requisitos que solamente se conocen una vez que el modelo se monitoriza. Como pueden ser el *desfase de datos*, sesgo inherente o fallos del sistema.

Para responder a estos desafíos de manera exitosa, los equipos de ML deben implementar las siguientes prácticas [46].

- **Reproducibilidad:** Como se ha explicado en el principio del capítulo, este aspecto es fundamental y es uno de los objetivos de *MLOps*. Cuando se automatizan los diferentes pasos del proceso de ciencia de datos,

es necesario que cada paso sea determinista, para evitar resultados indeseables.

- **Reusabilidad:** Para poder ajustarse a los principios de *entrega continua*, la *pipeline* necesita empaquetar y entregar modelos y código de una manera consistente, tanto a los entornos locales de entrenamiento como a los *entornos objetivos*, de forma que una misma configuración pueda arrojar los mismos resultados.
- **Manejabilidad** – La habilidad de aplicar regulación, rastrear los cambios en los modelos y código a lo largo del ciclo de vida, y permitir a los managers y gestores de equipo medir el progreso del proyecto y el valor comercial.
- **Automatización** – Al igual que en DevOps, para aplicar integración y entrega continua se require automatización. Los *pipelines* deben ser fácilmente repetibles, especialmente cuando se aplica gobernanza, o testing. Desarrolladores y científicos de datos pueden adoptar *MLOps* para colaborar y asegurar que las iniciativas de ML están alineadas con el resto de entrega del software, así como con el negocio en general.

CATEGORY	LEVEL 0	LEVEL 1	LEVEL 2	LEVEL 3	LEVEL 4
STRATEGY	<ul style="list-style-type: none"> - No data scientists hired - Skeptical of value of ML among executive team 	<ul style="list-style-type: none"> - Small and siloed data science and data engineering teams - A small number of ML champions in executive team 	<ul style="list-style-type: none"> - Small Data science, data engineers and software development teams starting to be coordinated - ML development efforts still unstructured and discrete 	<ul style="list-style-type: none"> - Large, well-integrated teams across data science, engineering and software development - Chief Data Officer, and C-suite level sponsorship exists - New team members brought up to speed in weeks - Project checkpoints to ensure ML is considered for major projects 	<ul style="list-style-type: none"> - ML seen as strategic, driving company initiatives - Well-governed process for ML delivery - Engineers & researchers are embedded on the same team
ARCHITECTURE	<ul style="list-style-type: none"> - Data Silos with one-off integration - Data not prepared nor ready for ML 	<ul style="list-style-type: none"> - Basic Enterprise data ready for ML - Data architecture still immature - Tacit commitment to meaningful enterprise data in the cloud. 	<ul style="list-style-type: none"> - Data architecture is mature - Most enterprise data ready for ML in the cloud - Overt commitment to cloud 	<ul style="list-style-type: none"> - Enterprise data is well cataloged and managed - Automated data pipelines in place - ML configuration and infrastructure is managed - ML models automatically provisioned as microservices 	<ul style="list-style-type: none"> - Comprehensive architecture to effectively govern all data - Consistent data storage and consumption pipeline across projects - Target ML infrastructure monitored for cost-effectiveness and optimal utilization
MODELING	<ul style="list-style-type: none"> - Manual process for model training - Limited pilot studies 	<ul style="list-style-type: none"> - Manual ML model training and live pilots - Basic experiment tracking, no model management 	<ul style="list-style-type: none"> - Experiment tracking and model management in place - Models dependencies not well understood 	<ul style="list-style-type: none"> - Models cataloged through lifecycle, supporting reproducibility and reuse - Output from ML is predictable and consistent, with auditable and reproducible outcomes 	<ul style="list-style-type: none"> - Interdependencies between models are monitored and managed - Impact of small changes to ML models can be measured
PROCESSES	<ul style="list-style-type: none"> - No DevOps practices adopted - No clearly defined success criteria for ML projects 	<ul style="list-style-type: none"> - DevOps practices like CI/CD have been adopted for non-ML components - No consistency in measures for ML or MLOps success 	<ul style="list-style-type: none"> - Development iterative but CI/CD not in place for models - ML infrastructure expertise not broadly available - ML configuration is an afterthought - Metrics/ measures in place but not consistent across projects 	<ul style="list-style-type: none"> - Data tested for model applicability and monitored for changes in distribution - All artifacts (data sets, tests, models) under version control - DevOps practices like CI/CD, code reviews in place for ML code - Production MLOps pipeline flow includes packaging, deployment, serving and operational monitoring 	<ul style="list-style-type: none"> - Comprehensive MLOps pipeline supporting frequent model updates - New algorithmic approaches can be tested at full scale - Automatic metrics gathering, alerts, issues analysis (such as data drift) and automated retraining of systems is in place
GOVERNANCE	<ul style="list-style-type: none"> - Not considered 	<ul style="list-style-type: none"> - Not considered, though the organization may have broader views - No notion of the concept of bias in models 	<ul style="list-style-type: none"> - Model explainability not considered - Models may harbor prediction bias - Model releases are tracked 	<ul style="list-style-type: none"> - Security policies applied to models, data - Ethics and explainability consideration for models and ML Systems - Good faith attempts to remove biased variables from models - Potential for malicious use of ML considered in ML lifecycle 	<ul style="list-style-type: none"> - Cybersecurity experts engaged in ML operations - ML systems protected from external manipulation. - End to end audit trail for ML - who, why, when



Figura 3.4: Tabla que resume los aspectos claves de la adopción de MLOps en la industria a diferentes niveles según el modelo de madurez descrito en esta sección. Fuente: [46]

Las prácticas anteriores son un indicador de la madurez del equipo de ciencia de datos, así como de las relaciones con el resto de equipos de desarrollo, y la compañía. Cada compañía puede implementar estas prácticas a diferentes niveles. El modelo de madurez de *MLOps* se denomina *MLOps Maturity*

Model. En la figura 3.4) se muestra un resumen de cada nivel según este modelo. Las categorías recogidas en él son las siguientes:

- **Estrategia:** Como la compañía puede alinear las actividades de *MLOps* con las prioridades ejecutivas, de organización y culturales.
- **Arquitectura** – La habilidad para manejar datos, modelos, entornos de despliegue y otros artefactos de manera unificada.
- **Modelado** – Habilidades de ciencia de datos y experiencia, que sumados al conocimiento de dominio, permitan el desarrollo y entrega de sistema de ML para dicho dominio.
- **Procesos** – Entrega y despliegue de actividades de manera eficiente, efectiva y mensurable, que impliquen científicos, ingenieros y administradores.
- **Gobernanza** – En general, la habilidad para construir soluciones de inteligencia artificial seguras, responsables y justas.

3.4.3. MLOPS Y REPRODUCIBILIDAD

Como se puede observar, *MLOps* y el problema de la reproducibilidad están estrechamente relacionados. Para poder implementar correctamente las buenas prácticas de *MLOps*, es necesario que cada paso del proceso sea lo más reproducible y determinista posible. Esto es un requisito necesario para poder implementar las prácticas de integración y entrega continua, ya que se fundamentan en la automatización. Por tanto, las prácticas descritas en la sección de *Reproducibilidad* sobre el control de las particiones de datos, estocásticidad, parámetros del experimento, etc. deben ser aplicados también para el desarrollo de sistemas de ML en la industria.

Por la razón anterior, la mayoría de software y plataformas orientadas a *MLOps* ofrecen herramientas para la gestión y control de experimentos, así como control sobre el entorno software y/o hardware. Además, las herramientas de *MLOps* están orientadas en su mayoría a la ejecución de trabajos en la nube y la colaboración. Esto puede ser de utilidad para la investigación, cuando se estén tratando con datos o algoritmos que requieran de una capacidad de cómputo superior a los ordenadores locales. Es por eso que

nuestro objetivo principal va a ser el estudio de las diferentes herramientas para *MLOps* y el desarrollo de nuestra propia herramienta con foco en la reproducibilidad.

3.5. Estado del Arte

En esta sección vamos a analizar el estado del arte para las herramientas de *MLOps*, herramientas orientadas a la reproducibilidad exclusivamente, así como el trabajo realizado hasta la fecha en relación al problema a resolver.

3.5.1. HERRAMIENTAS PARA LA REPRODUCIBILIDAD

Existen herramientas dedicadas a facilitar la reproducibilidad de experimentos en el campo de la investigación. A continuación, se resumen algunas de las herramientas más utilizadas:

- **Reprozip:** *Reprozip* [47] es una utilidad de código libre cuyo objetivo es el de empaquetar todo el trabajo con sus respectivas dependencias, variables de entorno, etc, en un paquete autocontenido. Una vez creado ese paquete, *Reprozip* puede restablecer el entorno tal y como se originó para que se pueda reproducir en una máquina distinta, ahorrando al usuario de la instalación de dependencias y la configuración del entorno. *Reprozip* puede utilizarse con cualquier lenguaje de programación y con una gran variedad de herramientas de análisis, incluidos los cuadernos de *Jupyter*.
- **Sacred:** Sacred [48] es una herramienta en Python, cuyo objetivo es el de facilitar la configuración, organización y registro de experimentos. Está diseñada para añadir una sobrecarga mínima y permitir la modularidad y configuración de experimentos. Las funcionalidades principales de esta herramienta son:
 - Registrar los parámetros de los experimentos
 - Facilitar la ejecución de experimentos con diferente configuración

- Almacenar la información sobre los experimentos en una base de datos
- Reproducir los resultados Además, se integra fácilmente con herramientas de visualización de monitorización de experimentos como *Tensorboard*.

3.5.2. HERRAMIENTAS PARA *MLOPS*

- **MLFlow:** MLFlow [3] es una herramienta de código abierto para el manejo del ciclo de vida completo de un proyecto de ML, incluida la experimentación, reproducibilidad y despliegue. Actualmente, este proyecto ofrece tres módulos principales: Tracking, Projects, Models.
 - *Tracking*: La API de Tracking permite registrar experimentos, parámetros, métricas, artefactos, y otros metadatos.
 - *Projects*: El módulo de Projects permite empaquetar y distribuir los proyectos usando un formato simple como YAML. En este fichero se le especifican las dependencias, el entorno, los parámetros, y el punto de entrada del proyecto.
 - *Models*: El módulo Models permite empaquetar modelos de los frameworks más conocidos - Tensorflow, Pytorch, Sklearn, MXNet, etc, en un formato genérico, almacenarlos en un *Registro de modelos* (ver Nomenclatura), y desplegarlos. Soporta múltiples lenguajes y ofrece una API REST para la consulta de información por servicios externos.
- **CometML:** Comet [49] ofrece una plataforma para el registro, rastreo, comparación y optimización de experimentos y modelos. Esta plataforma está basada en cloud (aunque con soporte para alojarlo en servidores propios). Algunas de las características a destacar son: soporte para cuadernos de *Jupyter*, optimización de hiperparámetros nativa (*meta-learning* [50]), y un potente sistema de visualización. Además, permite recoger métricas del sistema - uso de CPU, memoria, etc, a lo largo de la ejecución de los experimentos. Soporta múltiples lenguajes

y ofrece una *API REST* para la consulta de información por servicios externos.

- **Polyaxon:** *Polyaxon* [51] es una herramienta enfocada también al ciclo de vida completo de un proyecto de ML. La plataforma utiliza Kubernetes [52] para hacer los proyectos reproducibles, escalables y portables. Esta herramienta permite definir experimentos, almacenar información (métricas, parámetros, etc), así como desplegar modelos. Una funcionalidad que ofrece esta herramienta, que no se encuentra en las dos anteriores, es soporte propio para optimización de hiperparámetros. Además, ofrece un completo sistema de manejo de usuarios y un marketplace de integraciones. Esta plataforma es ideal para organizaciones de tamaño medio-grande que requieran una gestión de usuarios y roles completa, escalabilidad, y gobernanza sobre los modelos desplegados.
- **Kubeflow:** El objetivo de *Kubeflow* [53] no es implementar una plataforma para el ciclo de vida ni para el manejo de modelos, el objetivo principal es el de despliegue de flujos de trabajo completos en Kubernetes. Esta herramienta permite desplegar modelos en diferentes infraestructuras de forma sencilla, portable, y escalable. Por otro lado, con *Kubeflow Pipelines* se pueden desplegar *pipelines* completas usando *Argo* como motor.
- **Amazon SageMaker:** *SageMaker* [22] es la plataforma de ML de Amazon Web Services (AWS) . Esta plataforma integra herramientas que cubren todo el proceso de ciencia de datos. Incluye servicios de gestión de datos y etiquetado, cuadernos de Jupyter en la nube, registro y seguimiento de experimentos, despliegue, monitorización, y optimización de hiperparámetros. Hay varias características que hacen única esta plataforma, entre ellas: ofrece un IDE orientado a ML (Amazon SageMaker Studio), ofrece herramientas de depuración (Amazon SageMaker Debugger), y una integración con el servicio de etiquetado humano Amazon Mechanical Turk.
- **Google AI Platform:** La nube de Google Cloud Platform (GCP) ofrece un conjunto de herramientas que cubren todo el proceso de cien-

cia de datos. A este conjunto de herramientas se le conoce como *Google AI Platform* [54], aunque cada herramienta se puede utilizar por separado. Para la gestión y procesado de datos Google Cloud ofrece bases de datos a escala (*BigQuery*), un y un servicio de etiquetado automático (*Data Labelling Service*). Para la construcción y entrenamiento de modelos, GCP ofrece imágenes de máquinas virtuales, servicios de *cuadernos de Jupyter* en la nube, y otras herramientas para la ejecución de trabajos en la nube. Además, todos los trabajos se pueden ejecutar tanto en máquinas de GCP, como en servidores propios gracias al soporte para *Kubeflow Pipelines*.

- **Azure Machine Learning:** El conjunto de servicios y herramientas para ciencia de datos de Azure se llama *Azure Machine Learning* [55]. Al igual que la GCP, Azure ofrece herramientas para todas las etapas del ciclo de vida del proceso de ciencia de datos. Azure Machine Learning ofrece soporte para pipelines reproducibles, imágenes de máquinas virtuales, gestión del código y datos, etc. Además, ofrece soporte para seguimiento de experimentos e hiperparametrización. Una característica interesante es que ofrece la posibilidad de empaquetar modelos en formato ONNX [56] y desplegarlos en diferentes entornos objetivos ofrecidos por Azure, incluido instancias con FPGA [57].
- **Neptune:** *Neptune* [58] ofrece una biblioteca de código libre para Python con la que poder registrar y hacer un seguimiento de experimentos. Neptune ofrece una gestión de proyectos y un sistema de usuarios y roles completo. Además, cada experimento puede ser visualizado, compartido y debatido entre los diferentes miembros del equipo. *Neptune* es un framework ligero pero se integra fácilmente con diferentes herramientas, como MLFlow. En lugar de enfocarse en todo el proceso de ciencia de datos, el objetivo principal de esta herramienta es el de gestionar experimentos y registrar toda la información de una manera sencilla.

3.5.3. ANÁLISIS DE RAYOS GAMMAS

3.6. Modelos utilizados: Redes neuronales

Para el experimento llevado a cabo se ha propuesto la aplicación de una técnica clásica para la clasificación de partículas. Se ha trabajado con redes neuronales, en concreto con *autoencoders*, debido principalmente a su eficacia, robustez, flexibilidad, entre otras características.

Las redes neuronales son algoritmos de aprendizaje automático que han adquirido una gran popularidad en los últimos años, y que han sido desarrollados y utilizados en una gran variedad de problemas: aprendizaje supervisado, no supervisado, aprendizaje por refuerzo, y reducción de la dimensionalidad, entre otros.

Para describir una red neuronal vamos a empezar por la arquitectura más básica, una sola neurona. Una forma de representar dicha neurona en un diagrama es la siguiente:

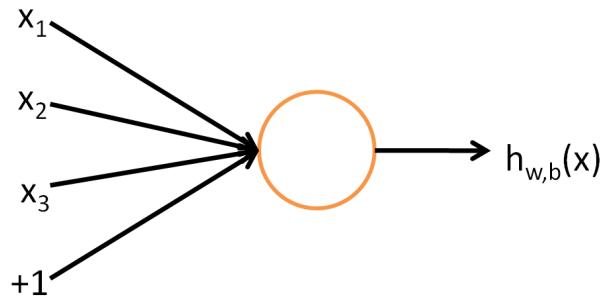


Figura 3.5: Diagrama de una neurona, también llamada unidad. Fuente: [59]

Una neurona no es más que una unidad computacional que toma como entrada un vector x (más un elemento a 1 para el sesgo), y cuya salida es $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^3 W_i x_i + b)$, donde $f : \mathbb{R} \mapsto \mathbb{R}$ es la llamada **función de activación**. Entre las funciones de activación más comunes se encuentran: sigmoide, tanh, RELU, LeakyRELU y Swish [60].

Una red neuronal se construye juntando varias neuronas, de forma que las salidas de unas neuronas son las entradas de otra, como se muestra en la

figura 3.5). En la figura, los círculos representan una neurona, y aquellos con etiqueta +1 son las **unidades de bias**. Por otro lado, las unidades o neuronas se agrupan en capas, una capa está representada como una columna de círculos. Dentro de estas capas, podemos diferenciar tres tipos: la capa de entrada (más a la izquierda), la capa interna, y la capa de salida que solamente contiene una neurona (a la derecha).

Vamos a denotar, n_l como el numero de capas de nuestra red, $n_l = 3$ en nuestro ejemplo. A la capa de entrada la denotamos como L_1 , y la capa de salida por tanto sería L_{n_l} . Nuestra red neuronal tiene como parámetros $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, donde cada elemento $W_{ij}^{(l)}$ corresponde con el parámetro asociado a la conexión entre la neurona j de la capa l y la neurona i de la capa $l + 1$. Por otro lado, $b_i^{(l)}$ es el *bias* asociado a la unidad i de la capa $l + 1$.

Podemos denotar a la activación (valor de salida) de una neurona i de la capa l como $a_i^{(l)}$. En el caso de la capa de entrada ($l = 1$), es obvio que $a_i^{(1)} = x_i$, es decir, la activación de la capa de entrada es el mismo vector de entrada. Gracias a la notación vectorial, podemos definir el vector de activaciones de una capa como:

$$\begin{aligned} z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\ a^{(l+1)} &= f(z^{(l+1)}) \end{aligned}$$

Finalmente, la función hipótesis, o salida de la red, se puede definir como:

$$h_{W,b}(x) = a^{(n_l)} = f(z^{(n_l)})$$

Teniendo en cuenta esta nomenclatura, la función de salida de la red mostrada en la figura 3.5, corresponde con la siguiente ecuación:

$$\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}$$

Donde x es el vector de entrada, es decir, un ejemplo en el conjunto de entrenamiento.

Una de las ventajas principales de usar la notación vectorial es que a la hora de implementarlo, podemos aprovechar bibliotecas y rutinas de álgebra lineal con implementaciones eficientes como BLAS [61] o LAPACK [62].

3.6.1. ALGORITMO DE PROPAGACIÓN HACIA ATRÁS

Suponiendo que tenemos un conjunto de datos $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ con m ejemplos. Podemos entrenar una red neuronal usando gradiente descendiente. La función de coste a optimizar para un ejemplo es la siguiente:

$$J(W, b; x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2$$

Dado un conjunto de entrenamiento de m ejemplos, el coste total se define como:

$$\begin{aligned}
J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2 \\
&= \left[\frac{1}{m} \sum_{i=1}^m \left(\frac{1}{2} \|h_{W,b}(x^{(i)}) - y^{(i)}\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2
\end{aligned}$$

El primer término de $J(W, b)$ es la media de los cuadrados de los residuos (errores). El segundo término corresponde con la regularización. El término λ controla la importancia relativa de la regularización. Esta función de coste se utiliza tanto para regresión como para clasificación. En el caso de la

clasificación, y toma los valores 0 o 1 según la clase que corresponda. Si usamos \tanh como función de activación en la salida en lugar de la sigmoide, usaríamos los valores -1 y 1 en su lugar.

El objetivo es minimizar $J(W, b)$ como función de W y b . Para llevar a cabo esta optimización, debemos inicializar W y b con valores aleatorios próximos a cero, por ejemplo, con valores muestrados de $\mathcal{N}(0, \epsilon^2)$. El motivo por el que es importante inicializar aleatoriamente los pesos, es para **romper la simetría**. Posteriormente, aplicamos un algoritmo de optimización, como puede ser *gradiente descendiente*. Una iteración de gradiente descendiente actualizaría los pesos de la siguiente forma:

$$\begin{aligned} W_{ij}^{(l)} &:= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\ b_i^{(l)} &:= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \end{aligned}$$

El parámetro α corresponde al ratio de aprendizaje. El algoritmo de propagación hacia atrás [63] nos ofrece una forma eficiente de calcular las derivadas parciales necesarias para actualizar los pesos mediante gradiente descendiente. Para calcular las derivadas parciales, es necesario formular dichas derivadas.

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \end{aligned}$$

El motivo por el que ambas ecuaciones difieren, es que la regularización no se aplica al *bias*. El algoritmo que nos permite calcular dichas derivadas de manera eficiente es el siguiente:

1. Una pasada hacia adelante computando los valores de todas las neuronas a partir de la segunda capa.

2. Para cada neurona i de la capa n_l de salida, calculamos:

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = - \left(y_i - a_i^{(n_l)} \right) \cdot f' \left(z_i^{(n_l)} \right)$$

3. Para cada capa $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$ y para cada neurona i en l , calcular:

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f' \left(z_i^{(l)} \right)$$

4. Finalmente, las derivadas parciales vienen dadas por:

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) &= \delta_i^{(l+1)} \end{aligned}$$

3.6.2. AUTOENCODERS

Autoencoders [64] son redes neuronales entrenadas para reconstruir la entrada, es decir, para copiar la entrada en la salida. Internamente, estas arquitecturas contienen un capa interna llamada **código**. Este código es una representación de los datos de entrada en un espacio vectorial de dimensión igual o distinta a los mismo. La red puede plantearse como la suma de dos partes bien diferenciadas: un codificador (encoder), que representa una función $h = f(x)$, y un decodificador (decoder) que produce una reconstrucción de la salida $r = g(h)$. Esta arquitectura se puede ver fácilmente en la figura 3.6.

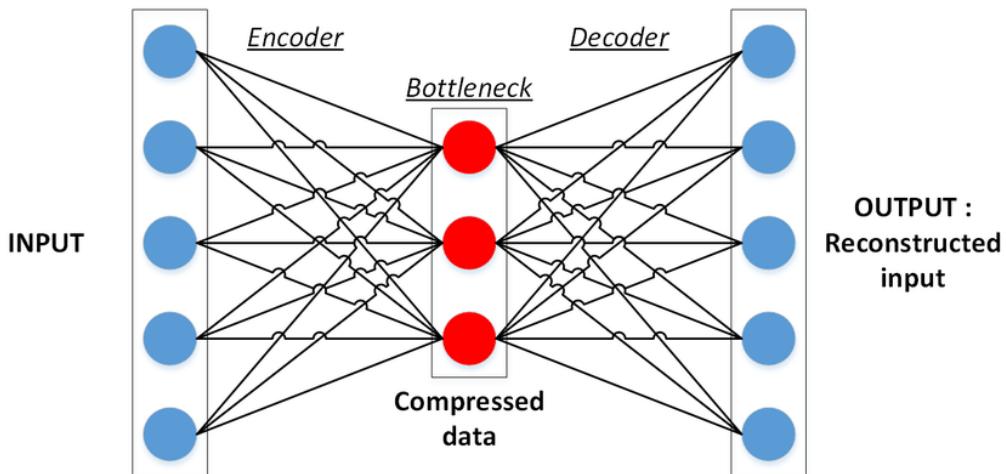


Figura 3.6: Arquitectura básica de un autoencoder de una sola capa para el codificador y el decodificador. Fuente: [65]

Si diseñamos un autoencoder que únicamente se encargue de copiar la entrada en la salida, es decir, si simplemente es capaz de mapear $g(f(x)) = x$ para todos los valores de x , no es especialmente útil. Sin embargo, podemos diseñar autoencoders que no se limiten a copiar la información de entrada, sino que aprendan patrones de los datos y los utilicen para la reconstrucción. Este es el objetivo de los autoencoders. Cuando restringimos de alguna forma una arquitectura de este tipo, el error de reconstrucción $e = L(g(f(x)), x)$, donde L puede ser cualquier métrica de distancia, va a ser mayor que 0 en mayoría de casos (y positivo siempre). Debido a que solamente podemos reconstruir los datos de entrada de manera aproximada. Dichas restricciones, fuerzan al modelo a priorizar partes de información que deben ser copiadas, encontrando así patrones útiles en los datos.

Tradicionalmente, este tipo de arquitecturas se han utilizado para reducción de dimensionalidad o aprendizaje de características [66]. La reducción de la dimensionalidad es posible debido que la capa interna (*código*) contiene información relevante que permite reconstruir los datos originales a partir de ella. Por ese motivo, si utilizamos una capa de código con un número de neuronas menor que la dimensión de los datos de entrada, podemos conseguir una representación aproximada de dichos datos en un espacio de dimension inferior. Para el aprendizaje de características, un uso interesante que se le ha dado a esta arquitectura es el de preentrenar arquitecturas o partes de

ellas a partir de datos sin etiquetas ??. Esto se consigue entrenando un autoencoder, y transfiriendo los pesos de dicha arquitectura, normalmente de la parte del codificador, hay otra arquitectura diseñada para un problema supervisado. De esta forma, si disponemos de datos no etiquetados, podemos aprovecharlos también para un problema supervisado.

3.6.2.1. Autoencoders según la dimensión del código

Según el tamaño del código existen dos categorías de autoencoders. Cuando el código tiene un tamaño menor que la dimensión del vector de entrada (columnas del conjunto de entrenamiento), se le conocen como autoencoders ***undercomplete***. Si por el contrario, el código tiene una dimensión mayor, esos autoencoders reciben el nombre de ***overcomplete***.

Una de las formas más importantes para hacer que el encoder extraiga características relevantes de los datos, en lugar de meramente copiarlos, es restringir h para que tenga una dimensión menor que x . Es decir, tener un autoencoder *undercomplete*. De esta forma, el encoder es forzado a aprender las características más importantes que van a permitir restaurar la mayoría de información.

El proceso de aprendizaje de los autoencoders se puede resumir en la optimización de la siguiente función de coste:

$$L(\mathbf{x}, g(f(\mathbf{x})))$$

Para todos los ejemplos x del conjunto de entrenamiento. L corresponde, como se ha mencionado anteriormente, a la métrica de similaridad. La métrica más común es el error cuadrático medio. Un aspecto interesante de esta métrica, es que cuando se usa con un autoencoder *undercomplete* cuyo decodificador sea lineal (aquel cuya función de activación para todas sus neuronas sea $f(x) = x$), este aprende a generar un subespacio equivalente al de Análisis de Componentes Principales (PCA).

Por otro lado, los autoencoders *overcomplete* no suelen ser muy útiles en

la práctica. Debido principalmente a que si el código es mayor o igual que el tamaño de los datos de entrada, no hay nada que impida al autoencoder aprender a copiar la información, ya que si $x \in \mathbb{R}^N$, cualquier espacio vectorial $\mathbb{R}^{N'}$ donde N' sea mayor que N puede generar todos los datos de entrada. Para poder utilizar este tipo de autoencoders es necesario el uso de **regularización**.

3.6.2.2. Autoencoders regularizados

Como se ha descrito anteriormente, los autoencoders *undercomplete*, cuya dimensión del código es menor que la de la entrada, pueden aprender las características o patrones mas relevantes de la distribución de los datos. El problema principal de este tipo de arquitecturas, tanto *undercomplete* como *overcomplete*, es que el autoencoder sea demasiado potente como para no tener aprender nada útil y simplemente se encarguen de copiar la información. Este problema se hace obvio cuando en el caso de los autoencoders *overcomplete*, (incluso en aquellos con una dimensión del código igual que la entrada). En esos casos, hasta un autoencoder lineal puede aprender a copiar la entrada en la salida [67].

El objetivo de la regularización es permitir entrenar cualquier arquitectura de autoencoder de manera que esta aprenda correctamente, donde el tamaño del código y la profundidad de la red no esté limitada por el aprendizaje, sino por la complejidad de la distribución de datos. En lugar de restringir la arquitectura, los autoencoders regularizados utilizan una función de coste que penaliza la copia de datos, o al menos, favorece características intrínsecas del modelo. Entre estas características se encuentra, la dispersión, robustez frente a ruido, etc. Al hacer uso de ese tipo de funciones de coste con regularización, incluso autoencoders no lineales y *overcomplete* pueden aprender patrones útiles sobre los datos. Incluso si la capacidad del modelo es suficiente como para aprender la función identidad.

3.6.2.2.1. Autoencoders dispersos Un autoencoder disperso ???, [59] es simplemente un autoencoder cuya función de coste contiene una penaliza-

ción por dispersión. La nueva función de coste es la siguiente:

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h})$$

Donde $\Omega(\mathbf{h})$ es la penalización por dispersión. El objetivo es el de maximizar la dispersión del vector de activaciones en la *capa oculta o interna* (código). Para ello, la penalización que se propone es la siguiente:

$$\Omega(\mathbf{h}) = \beta \sum_{j=1}^{s_2} \text{KL}(\rho \parallel \hat{\rho}_j)$$

Donde β es el parámetro que controla el peso de la penalización, ρ es el **parámetro de dispersión** y $\hat{\rho}_j$ es la activación media de la neurona j de la capa interna, cuya expresión viene dada por:

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^m \left[a_j^{(2)}(x^{(i)}) \right]$$

Básicamente, se calcula la salida o activación de una misma neurona para todos los ejemplos de entrenamiento, y se hace la media. Por otro lado, la función Kullback-Leibler $\text{KL}(\rho \parallel \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j}$ mide la divergencia entre una variable aleatoria de Bernoulli con media p y una variable aleatoria de Bernoulli con media $\hat{\rho}_j$. Esta función es un estándar a la hora de medir la similitud de dos distribuciones.

Los autoencoders dispersos se suelen usar para aprender características útiles para otra tarea, como puede ser clasificación. Un autoencoder disperso debe encontrar patrones inherentes a la distribución de datos, en lugar de actuar como una simple función identidad.

3.6.2.3. Denoising Autoencoders

Para este tipo de autoencoders, en lugar de añadir una penalización a la función de coste, se modifican los datos de entrada. Siguiendo la formulación del problema de apartados anteriores, tenemos la siguiente función a optimizar:

$$L(\mathbf{x}, g(f(\tilde{\mathbf{x}})))$$

Donde $\tilde{\mathbf{x}}$ es una copia de los datos de entrada a la que se le ha añadido ruido o algún otro tipo de corrupción. De esta forma, no basta con aprender la función identidad, es necesario además aprender patrones interesantes que permitan eliminar el ruido. Una forma sencilla de implementar este arquitecturas, es añadiendo una capa de **Dropout** como capa de entrada.

3.6.2.4. Autoencoders variacionales

Los autoencoders variacionales [68] tienen dos enfoques, el enfoque de *Deep Learning* o el enfoque probabilístico. En nuestro caso, este tipo de arquitecturas se describen desde el enfoque del *Deep Learning*.

El principal uso de este tipo de arquitecturas es como *modelos generacionales*. Se utilizan para producir nuevos datos (especialmente imágenes) a partir de unos datos de entrenamiento. Desde el punto de vista de los modelos generativos, un autoencoder regular es ineficiente para este tipo de problemas. El motivo es que el espacio de representación intermedias (código), también conocido como **espacio latente**, tiene discontinuidades. Una forma de generar un nuevo ejemplo es aplicar el codificador y obtener la representación en el espacio latente. Posteriormente, ese vector se modifica ligeramente en una dirección deseada y se aplica el decodificador sobre el nuevo vector, generando así un nuevo ejemplo similar al anterior. El nuevo dato resultante es una combinación de aquellos ejemplos cercanos al nuevo vector. Si el espacio latente tiene discontinuidades, y el vector a reconstruir resulta estar en alguna de esas discontinuidades, el resultado va a ser muy poco realista. El objetivo de los autoencoders variacionales (*VAE*) es el de generar un espacio latente continuo para suavizar las interpolaciones.

Para entrenar este tipo de autoencoders necesitamos modificar la función de coste original. La nueva función de coste es la siguiente:

$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_{\theta}(z|x_i)} [\log p_{\phi}(x_i|z)] + \text{KL}(q_{\theta}(z|x_i) \| p(z))$$

Donde los parámetros θ y ϕ representan la matriz de pesos y el vector de *bias*, $q_\theta(z|x)$ denota el codificador, $p_\phi(x|z)$ denota el decodificador, y $p_\phi(x|z)$ representa el error de reconstrucción.

3.6.2.5. Truco de la reparametrización

El término de la esperanza en la función de coste implica la generación de ejemplos de la distribución $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$. Muestrear es un proceso estocástico, por tanto, no podemos aplicar la propagación hacia atrás. Para poder optimizar dicha función de coste, se aplica el truco de la reparametrización (**reparameterization trick**) [69].

Una variable aleatoria \mathbf{z} se puede expresar como una variable determinística $\mathbf{z} = \mathcal{T}_\phi(\mathbf{x}, \epsilon)$, donde ϵ es una variable aleatoria independiente, y la función de transformación \mathcal{T}_ϕ parametrizada por ϕ convierte ϵ a \mathbf{z} .

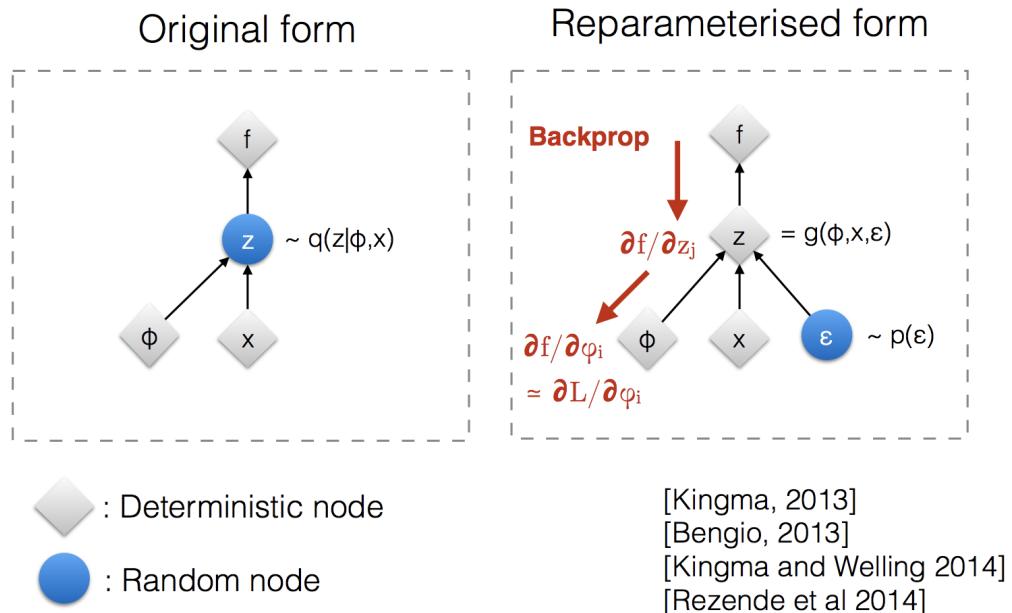


Figura 3.7: Ilustración de como el truco de la reparametrización hace el proceso de muestreo de \mathbf{z} entrenable. Fuente: Dispositiva 12 en el workshop de Kingma para NIPS 2015

Como ejemplo, una forma común para esto $q_\phi(\mathbf{z}|\mathbf{x})$ es una Gaussiana multivariable con estructura de covarianza diagonal.

$$\begin{aligned}\mathbf{z} &\sim q_{\phi}(\mathbf{z} | \mathbf{x}^{(i)}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)} \mathbf{I}) \\ \mathbf{z} &= \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \text{ where } \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})\end{aligned}$$

Donde \odot corresponde al producto elemento a elemento.

El truco de la reparametrización funciona también para otro tipo de distribuciones, no solo la Gaussiana. En el caso de la Gaussiana multivariable, se hace posible entrenar el modelo aprendiendo la media y la varianza de la distribución. μ y σ , usando explícitamente este truco, mientras que la estocásticidad permanece en la variable aleatoria $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$.

3.6.3. AUTOENCODERS APILADOS

Aunque en las secciones anteriores tanto el codificador como el decodificador se han tratado como dos capas dentro de una red de 3 capas. La realidad es que en muchos casos se necesita más capas tanto a un lado como a otro. Aquellos autoencoders donde o bien el codificador o bien el decodificador tienen más de una capa, se les conoce con el nombre de Autoencoders apilados (**Stacked Autoencoders**) ??? . Al igual que para el resto de arquitecturas de *Deep Learning*, añadir más capas permite reducir la linealidad y aprender patrones más complejos.

El principal factor a tener en cuenta en estos casos, es que los autoencoders son muy potentes de por sí, en relación con la función que tienen que modelar (identidad). Es por esto, por lo que la regularización se vuelve esencial a la hora de apilar diferentes capas a un lado u a otro.

En cuanto a diseño, la forma más común de diseñarlos es de manera simétrica - el mismo número de capas y unidades para el encoder y el decoder. Además, las capas suelen tener un número de neuronas decrecientes para el encoder y crecientes para el decoder. Esto permite aplicar una técnica conocida como **Tied Weights** . Esta técnica consiste en compartir los pesos entre el codificar y el decodificador, haciendo que los pesos de este último correspondan con la transpuesta del primero:

$$\theta_d = \theta_e^T$$

Esta técnica mejora el rendimiento en el entrenamiento, ya que se entranan menos parámetros, pero además, sirve como método de regularización [70].

3.6.4. APLICACIONES DE LOS AUTOENCODERS

Las aplicaciones principales de los autoencoders han sido la **reducción de la dimensionalidad** y **recuperación de información**. Autoencoders no lineales pueden ofrecer un error de reconstrucción menor que PCA, y al no estar limitados a una proyección lineal, pueden aprender una representación más fácil de interpretar. En el caso de clasificación, los autoencoders pueden encontrar una representación donde los datos estén agrupados en clusters y las categorías estén bien diferenciadas. Además, encontrar una proyección a un espacio de dimensión inferior que mantenga la mayoría de la información, permite mejorar el rendimiento de modelos, ya que estos en espacios inferiores tiene un menor coste de cómputo y memoria.

Otra aplicación que se ha ido desarrollando en los últimos años es la de **detección de anomalías**. Los autoencoders pueden utilizarse para modelar la distribución de datos, y el error de reconstrucción se puede utilizar como indicador para detectar anomalías. Cuando un autoencoder se ha entrenado correctamente, el error de reconstrucción sobre datos de entrenamiento es bajo. Así como el error otros datos de la misma distribución que no hayan usado para entrenar (conjunto de validación y test por ejemplo). Pero en el caso de utilizar datos de una distribución distinta, al no poder extraer las características más importantes eficazmente, el error de reconstrucción es mayor. Por este motivo, se puede entrenar un autoencoder sobre los datos “no anómalos”, y establecer un umbral sobre el error de reconstrucción que indique si el ejemplo que se ha pasado por la red es una anomalía.

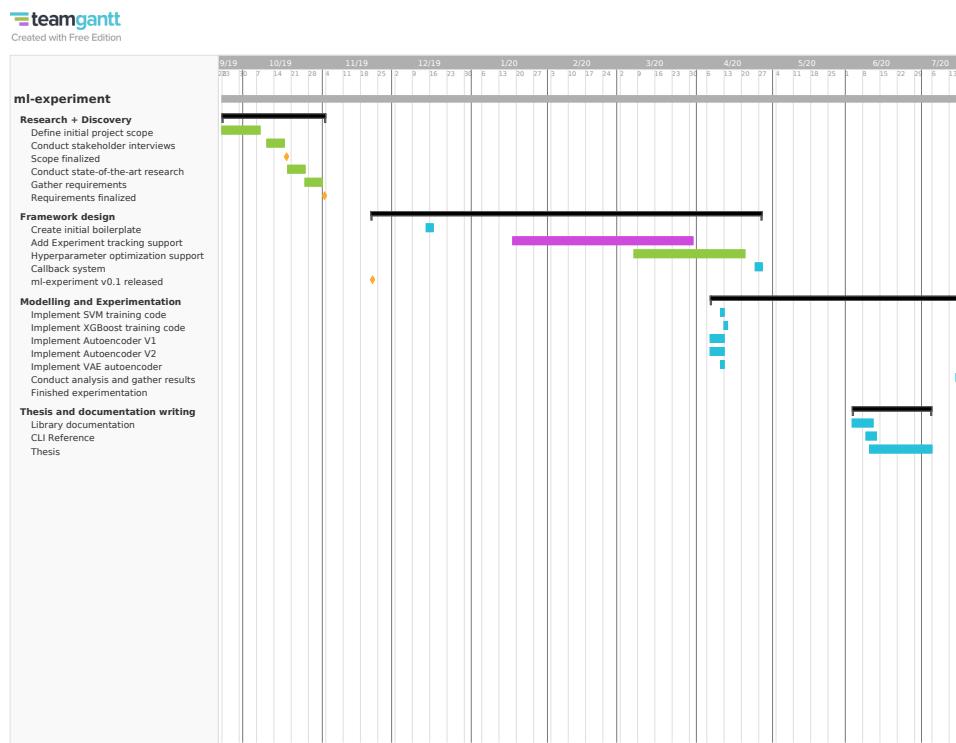
Por otro lado, cabe destacar el uso de los autoencoders variacionales como modelos generativos, aunque se ven opacados en su mayoría por GAN y similares.

Una última aplicación que cabe destacar es la de **clasificación**. Los autoencoders, pese a modelos de aprendizaje no supervisado, pueden usarse para problemas de clasificación [71]. Si se entrena un autoencoder para cada clase (con ejemplos exclusivos de esa clase), el error de reconstrucción de cada autoencoder se puede utilizar para decidir la clase. Presuntamente, aquellos ejemplos cercanos a una determinada clase, tendrán un error de reconstrucción menor en su autoencoder correspondiente. De esta forma, podemos aplicar este tipo de arquitecturas a problemas de clasificación. No obstante, los autoencoders se entranan de manera independiente minimizando el error de reconstrucción y la regularización (si aplica), esto implica que no hay una optimización directa del error de clasificación. Al perder esa relación directa con la métrica objetiva, este aplicación puede dar lugar a resultados subóptimos.

Capítulo 4

Planificación del trabajo

- Planificación optimista
- Planificación real



Capítulo 5

Presupuesto

- Comparativa cluster propio vs AWS, Azure, GDC
- Coste de titulado superior (36€)

Capítulo 6

Diseño y desarrollo del *framework*

El diseño y desarrollo de un framework orientado a la reproducibilidad es el objetivo principal de este trabajo. Un framework abierto que soporte cualquier biblioteca de Machine Learning o Deep Learning, y que se fundamente en los principios de reproducibilidad detallados en *Fundamentos*.

Aunque existen bastantes herramientas de MLOps que cubren en mayor o menor medida la reproducibilidad, muchas de ellas son privadas (*Amazon Sagemaker*, *Google AI Platform*, *CometML*, etc). Y las que son de código libre (MLFlow, Kubeflow) o híbridas (Polyaxon), no tienen algunas características importantes como optimización de hiperparámetros *out-of-the-box*, o bien, son complejas de utilizar o configurar (véase Polyaxon). En cuanto a las herramientas exclusivas de reproducibilidad, tanto *Sacred* como *Reprozip* son buenas soluciones cuando se realizan análisis en local, pero carecen de soporte para la gestión de trabajos en la nube o en un *cluster* remoto.

INSERTAR TABLA COMPARATIVA

El objetivo de nuestra herramienta es el de ofrecer un marco de trabajo completo, que incluya las características esenciales de MLOps, pero con un enfoque especial en la reproducibilidad. Como objetivo secundario, la herramienta está pensando para facilitar un flujo de trabajo tanto en remoto como

en local, con una instrumentalización del código mínima. Las características fundamentales de *ml-experiment* son:

- **Registro de experimentos y seguimiento de experimentos:** Uno de los pilares fundamentales de la reproducibilidad y de MLOps. La capacidad para almacenar en una *centro de conocimiento* (base de datos, sistema de directorios, etc) parámetros, métricas, artefactos, y otros metadatos.
- **Control de estocásticidad:** Recoger información sobre la semilla utilizada para los diferentes generadores de números aleatorios.
- **Optimización de hiperparámetros:** Soporte para la ejecución de multiples experimentos en paralelo con el fin de optimizar una o varias métricas. Además, se pueden aplicar aplicar diferentes algoritmos optimización - Bayesiana, GridSearch, etc.
- **Ejecución de experimentos de manera distribuida:** Una de las características esenciales a la hora de llevar a cabo optimización de hiperparámetros, es la posibilidad de poder ejecutar los experimentos de manera paralela y/o distribuida. En este sentido, *ml-experiment* permite la ejecución de los diferentes experimentos en paralelo utilizando los diferentes núcleos de la CPU (*multiprocessing*), así como ejecutarlos de manera distribuida en un cluster remoto de Ray (ver ??).
- **Almacenamiento y gestión de modelos:** Esta característica propia de la filosofía MLOps también es interesante desde el punto de vista de la investigación. Primeramente, el llevar un seguimiento de los modelos entrenados durante la fase de experimentación permite, entre otras cosas, seleccionar y aplicar los modelos que se consideren adecuados para la resolución del problema. De otra forma, se debería seleccionar el mejor experimento y replicar todo el proceso hasta obtener el modelo. Por otro lado, si se almacenan los modelos, y por algún motivo los datos y procedimientos no se pueden compartir con la comunidad científica, al menos se pueden compartir los modelos acercando el estudio a la *Investigación Replicable* (ver ??)

- **Configuración de experimentos flexible y sencilla:** Con el fin de reducir la *Deuda de configuración*, *ml-experiment* ofrece una manera sencilla de definir experimentos y trabajos de optimizar de hiperparámetros utilizando ficheros YAML o JSON.
- **Instrumentalización mínima:** Como se ha detallado en la sección de ??, uno de los *anti patrones* a evitar en los proyectos de ciencia de datos es el uso *código pegamento*. Por este motivo, nuestro objetivo a la hora de diseñar *ml-experiment* es el de evitar grandes cambios en el código existente para entrenamiento o análisis, es decir, reducir la instrumentalización. Gracias a esto, evitamos dicho *anti patrón*.

6.1. Herramientas utilizadas

ml-experiment se fundamenta en un pequeño conjunto de herramientas de código abierto muy potentes y activas. Entre las principales herramientas utilizadas, cabe destacar:

- Docker [72]: Docker es una plataforma de contenedores software que ayuda a empaquetar aplicaciones junto con sus dependencias en forma de contenedores para asegurar que la aplicación se ejecuta independientemente al sistema operativo anfitrión. Un contenedor docker es una unidad software estandarizada que se crea en tiempo real para desplegar una aplicación o entorno particular. Los contenedores pueden ser entornos - Ubuntu, CentOs, Alpine, etc - o puede ser aplicaciones enteras - contenedor de NodeJS-Ubuntu por ejemplo.
- Optuna [73]: Optuna es un framework para optimización de hiperparámetros automatizada, diseñado especialmente para ML. La característica principal que diferencia a esta herramienta de otras como Hyperopt, sk-opt, etc, es la API *define-by-run*, la cuál es una API imperativa con la que se pueden construir espacios de búsqueda de hiperparámetros de manera dinámica. Además, soporta diferentes algoritmos de optimización: TPE, Hyperband, GridSearch, etc [74].

- MLFlow [3]: Ver sección de *Fundamentos*.
- Ray [75]: Ray es un framework orientado al desarrollo y ejecución de aplicaciones distribuidas. Originalmente, este proyecto fue propuesto para el entrenamiento de modelos de Aprendizaje por Refuerzo (RL) [76] distribuido, pero posteriormente se adaptó para cualquier aplicación distribuida en Python. De esta forma, Ray se puede considerar un framework de propósito general para la computación en clusters. Algunos experimentos requieren de una preprocesado de datos costoso, o de un entrenamiento de larga duración. Para satisfacer estos requisitos, *Ray* propone una interfaz unificada con la que se pueden definir dos tipos de tareas: Tareas paralelas, tareas basadas en el modelo *actor* [77]. Las tareas paralelas permiten distribuir la computación de manera balanceada, procesar grandes cantidades de datos, y recuperarse de errores. Por otro lado, el uso de Actores permite manejar computaciones con estado, y compartir ese estado entre diferentes nodos de manera sencilla.

6.2. Estructura general

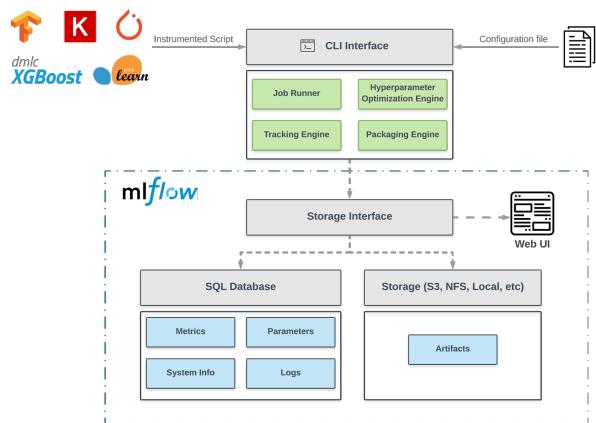


Figura 6.1: Diagrama de la estructura general de *ml-experiment*

El núcleo principal de *ml-experiment* está dividido en 4 módulos (ver Figura 6.1) con los que el usuario interacciona indirecta o directamente a través

de una CLI y de una biblioteca de *Python*. El módulo *JobRunner* es el encargado de ejecutar un código instrumentalizado¹ en un cluster de *Ray*, en un contenedor de Docker, o en local (utilizando varios procesos si se emplea HPO). El módulo de seguimiento (*Tracking*) ofrece una abstracción sobre *MLFlow* para el registro de experimentos, parámetros, métricas, y artefactos. Además, recoge automáticamente otra información relevante sobre el experimento como el entorno software y hardware, las semillas de los generadores aleatorios, entre otros. El módulo de empaquetado *Packaging* se encarga de generar una versión del proyecto que se pueda distribuir fácilmente utilizando la tecnología **Docker** para generar un *imagen* de todas las dependencias. El objetivo principal de este módulo es el de simplificar la reproducibilidad de trabajo científico entre diferentes entornos heterogéneos mediante el uso de *contenedores*. Los cuatro módulos principales que componen el *framework* dependen de una “centro de conocimiento” (*Knowledge Center*). El *Knowledge Center* se compone de una base de datos SQL y un sistema de ficheros (local, compartido, o en la nube). Las métricas, parámetros, y otros metadatos se almacenan en la base de datos SQL, mientras que los artefactos se almacenan en el sistema de ficheros. La gestión de ambos sistemas de información se delega a *MLFlow*.

6.3. Instrumentalización de código y ejecución del código

El primer paso para poder utilizar el framework es el de instrumentalización. *ml-experiment* ofrece una biblioteca de Python minimalista con el que se pueden definir *trabajos* a ejecutar. En Listing 6.1 se muestra la forma de instrumentalizar un código de Python. Básicamente, se define el punto de entrada del programa *main*² con los parámetros del experimento, y se envuelve con el decorador *@job*. Toda la información sobre el uso del framework

¹Un código instrumentalizado comprende un código de Python al que se le aplican los cambios pertinentes para poder utilizarse con el framework. En concreto, estos cambios se basan en utilizar la biblioteca de Python del *framework*. El objetivo de *ml-experiment* es que estos cambios sean mínimos.

²El nombre no es relevante. Simplemente tiene que ser la función que se llame al ejecutar el script

(biblioteca y CLI incluida) se encuentra recogida en el *Manual de Usuario* (ver *Anexo 2*).

Listing 6.1: Ejemplo de código instrumentalizado.

```
1 from ml_experiment import job
2
3 @job
4 def main(param1, param2, param3):
5     # Training code goes here
6
7 if __name__ == '__main__':
8     main()
```

El uso del decorador `@job` en el punto de entrada del programa hace imposible poder ejecutar el script directamente. Para poder ejecutar este código es necesario utilizar las CLI de *ml-experiment* (ver *CLI Reference* en *Manual de Usuario*). Existen tres formas diferentes de ejecutar el código con la CLI. La forma principal, la cuál recomienda encarecidamente, es mediante un fichero de configuración YAML o JSON (ver *YAML/JSON Specification* en *Manual de Usuario*). En Listing 6.7 se muestra un ejemplo de fichero de configuración. En ese mismo fichero de configuración se definen el nombre, parámetros, y el script a ejecutar. Además, se puede definir en el la configuración de *Docker* o *Ray* en caso de querer ejecutar en un contenedor o cluster. Una vez definido ese fichero, para ejecutar el código mediante la CLI se utiliza el siguiente comando:³

Listing 6.2: Ejemplo de comando para la ejecución del código definido en Listing 6.1 usando el fichero de configuración de Listing 6.7

```
1 ml-experiment --config_file=my_experiment.yaml
```

³El directorio donde se ejecuta el comando debe tener acceso al script. Es decir, la ruta definida en el campo *run* del fichero de configuración debe ser una ruta absoluta o una ruta relativa al directorio donde se ejecuta el comando.

Listing 6.3: Ejemplo de fichero de configuración para Listing 6.1

```
1 name: My Experiment
2
3 params:
4   param1: "hello"
5   param2: "ml-experiment"
6   param3: 1
7
8 run:
9   - path/to/script.py
```

La segunda opción para ejecutar el código es utilizar directamente la CLI. Esta opción es menos recomendable que la anterior, principalmente porque los ficheros de configuración facilitan la *replicabilidad* de experimentos, mientras que si se usa la CLI, para poder replicar los experimentos es necesario o bien compartir el comando ejecutado, lo cuál es poco legible y susceptible a cambios en la CLI, o compartir los parámetros con los que se ha ejecutado manualmente, sin disfrutar de las ventajas del control de versiones por ejemplo. En Listing 6.4 se muestra un ejemplo de ejecución de experimentos manualmente.

Listing 6.4: Ejemplo de comando para la ejecución del código definido en Listing 6.1 usando el fichero de configuración de Listing 6.7

```
1 ml-experiment \
2   --name="My experiment" \
3   --params="{'param1': 'hello', 'param2': 'world', 'param3':
4   ': 1}"
5   path/to/script.py
```

La tercera opción consiste en una combinación de las dos anteriores. La CLI de *ml-experiment* permite utilizar un fichero de configuración como base y modificar ciertos campos manualmente. En Listing 6.5 se muestra un ejemplo donde se utiliza el fichero de configuración definido en 6.7, y se sobreescribe el parámetro “param3”. El diccionario definido tanto en *-params* como *-param_space* se combina con aquel especificado en el fichero de configuración, tomando prioridad el de la CLI en caso de colisión entre claves.

Listing 6.5: Ejemplo de comando donde se combina la especificación de un experimento de un fichero YAML con parámetros introducidos por el usuario mediante la CLI.

```
1 ml-experiment \
2   --config_file=my_experiment.yaml
3   --params="{'param3': 20}"
```

6.3.1. EJECUCIÓN DE EXPERIMENTOS EN DOCKER O RAY

6.4. Tracking de experimentos

El registro y seguimiento de experimentos es un aspecto fundamental de la reproducibilidad, y uno de los pilares de *MLOps* (ver *Fundamentos*). *MLFlow* ofrece un potente interfaz para el “logging” de parámetros, métricas, y artefactos. La forma de registrar métricas en un experimento con *MLFlow* es la siguiente:

```
1 with mlflow.start_run():
2     mlflow.log_metric(key="quality", value=)
```

Como vemos, la interfaz es fácil de utilizar, el problema principal no es la dificultad de uso de *MLFlow*, sino la mantenibilidad del código. Los parámetros de un experimento pueden cambiar con el tiempo, haciendo que los parámetros que se registren sean obsoletos, o que los nuevo parámetros no se registren. Estos problemas se hacen acentúan al tener que mezclar código ML con código de relativo al seguimiento. El objetivo de *ml-experiment* es el de ofrecer una abstracción sobre la API de *Tracking* de *MLFlow* con el fin de asegurar unas buenas prácticas de desarrollo y reducir la deuda técnica como se ha comentado anteriormente. La forma en la actúa *ml-experiment* es automatizando el registro de parámetros, métricas, y artefactos. Cuando un código se instrumentaliza (como se muestra en Listing 6.6), se realiza un seguimiento automático de los los parámetros de la función de entrada (*main*). Es decir, cuando un experimento se ejecuta, los parámetros de entrada se almacenan en el servidor de *MLFlow*. Además de los parámetros, la función principal puede devolver una tupla de dos diccionarios, un primer

diccionario para las métricas,⁴ y el segundo diccionario para los artefactos.⁵

Listing 6.6: Ejemplo de código instrumentalizado donde se devuelve un diccionario el diccionario de métricas y el de artefactos.

```
1 from ml_experiment import job
2
3 @job
4 def main(param1, param2, param3):
5     {'metric1': 1.0}, {'artifact1': 'path/to/artifact'}
6
7 if __name__ == '__main__':
8     main()
```

Cuando se ejecuta un experimento, MLFlow por su parte almacena ciertos metadatos, como el nombre el identificador del ultimo *commit* antes de la ejecución del código, el fichero de Python ejecutado, nombre de usuario en el sistema, etc. Además de esta información, *ml-experiment* almacena otros metadatos teniendo en cuenta los aspectos críticos de la reproducibilidad descritos en *Fundamentos*. Estos metadatos son los siguientes:

- **Información hardware:** Especificaciones de la CPU y de la GPU, entre los que se incluye, nombre del procesador y de la tarjeta gráfica, cantidad de memoria de la GPU, versión de los drivers de CUDA [78], entre otros.
- **Información software:** Se hace una captura de todos los paquetes de Python disponibles en la ejecución del experimento. Esa información se almacena en un fichero *requirements.txt* y se sube al servidor de *MLFlow*. Por otro lado, se recoge información sobre la versión del interprete de Python utilizado.
- **Logs:** La salida de la consola se almacena en un fichero y se sube también a *MLFlow*.
- **Semilla de generadores aleatorios:** Por último, cuando un meneador de números aleatorios se alimenta con una determinada semilla,

⁴La clave es el nombre de la métrica y el valor es un número entero o flotante

⁵La clave es un identificador o nombre arbitrario para el artefacto, y la clave es la ruta relativa o absoluta al mismo

ml-experiment recoge esa semilla y la almacena como metadatos. El framework soporta los generadores aleatorios de Tensorflow, Pytorch, Numpy [79], y el nativo de Python (módulo *random*)

6.5. Hiperparametrización y entrenamiento distribuido

ml-experiment tiene soporte de primera clase para Optimización de Hiperparámetros (HPO). El framework permite ejecutar multiples experimentos simultáneamente en un entorno local o remoto. Además, permite la asignación de recursos de CPU y GPU para el trabajo de optimización, es decir, podemos especificar el número o fracción de cores y GPUs (si hay disponible), y el framework distribuirá las tareas entre los recursos disponibles (ver *Hyperparameter Tuning* en *Manual de Usuario*). Como ejemplo, Listing ?? muestra una especificación de un trabajo de HPO donde se ejecutan 12 experimentos. Los parametros *param1* y *param2* *se muestran de una distribución categórica*⁶ y *de una distribución uniforme respectivamente, mientras que el parámetro param3** tiene un valor fijo. Además, de los recursos disponibles en el sistema, se utilizan 4 núcleos exclusivamente. A nivel de ejecución, el campo *ray_config* permite añadir la dirección del cluster de Ray para ejecutar en remoto, en caso de no especificar ninguna dirección, o utilizar “localhost”, *ml-experiment* crea un cluster de Ray en local.⁷

A la hora de definir trabajos de HPO, el término utilizado en el framework es el de **grupo**. Un grupo es un tipo de trabajo en el que se ejecutan varios experimentos a la vez. Para poder definir un *grupo* utilizando la especificación YAML/JSON, es necesario añadir el campo *kind* e igualarlo a “group”. De este forma, *ml-experiment* reconoce que el trabajo corresponde a un grupo y configura la infraestructura correspondiente. El objetivo de un trabajo de HPO (grupo) es el de optimizar un métrica específica, ya sea minimización o maximización, en otras palabras, encontrar el conjunto de parámetros que maximice o minimice la métrica del experimento.⁸

⁶Se selecciona un elemento de la lista de manera aleatoria con la misma probabilidad.

⁷Un cluster de Ray en local implica que se levantan varios procesos (*num_cpus*), y la los experimentos se divide entre cada proceso (*num_trials / num_cpus*).

⁸Un experimento puede tener varias métricas y todas ellas quedan recogidas por el

Por otro lado, *ml-experiment* permite definir el *sampler* y *pruner* de **Optuna**. Un *pruner* es un algoritmo que indica cuando un experimento debe ser “podado” o no. Podar un experimento significa terminar su ejecución antes de tiempo. Cuando un experimento no es prometedor, es decir, cuando tiene muy poca probabilidad de mejorar la mejor métrica registrada previamente, los *pruners* permiten terminar la ejecución ahorrando recursos de computación y tiempo. Un *sampler* es una implementación de una estrategia de muestreo, es decir, una forma concreta con la que generar los parámetros de un experimento a partir de un espacio de hiperparámetros. Ese espacio de hiperparámetros viene definido por el campo *param_space*. *ml-experiment* soporta varias distribuciones de parámetros que cubren la mayoría de necesidades de un experimento de ML (ver *YAML/JSON Specification* en *Manual de Usuario*). Además, *ml-experiment* soporta todos los *samplers* y *pruners* de *Optuna*.

Listing 6.7: Ejemplo de fichero de configuración para un grupo de experimentos sobre Listing 6.1

```

1 name: My Experiment
2 kind: group
3 num_trials: 12
4 sampler: TPE
5
6 param_space:
7   param1: choice(["hello", "world"])
8   param2: uniform(0.001, 100)
9
10 params:
11   param3: 1
12
13 metric:
14   name: metric1
15   direction: maximize
16
17 ray_config:
18   num_cpus: 4
19
20 run:
21   - path/to/script.py

```

módulo de *Tracking*, pero de momento no hay soporte para optimización multiobjetivo.

6.6. Sistema de notificaciones y callbacks

6.7. Interfaz Web

6.8. Desarrollo del framework

1. Calidad del código
2. Python Avanzado.

[80]

6.9. Futuro desarrollo

Capítulo 7

Experimentos

En este último capítulo de la tesis se recogen los experimentos llevados a cabo para la resolución del problema (Objetivo 2). Primeramente, se describe detalladamente el problema: origen, tipo y estructura de los datos, tipo de problema, asunciones y/o restricciones, etc. Posteriormente, se detallan los diferentes algoritmos y arquitecturas de ML/DL empleados - se especifican los parámetros, biblioteca empleada, detalles de implementación, y otra información relevante. Finalmente, se muestran los resultados obtenidos para cada tipo de modelo, y algunos trabajos a posteriori que pueden ser interesantes.

7.1. Definición del problema

7.1.1. HISTORIA

Los rayos cósmicos son fragmentos de átomos (electrones, protones, y núcleos atómicos) que bombardean la tierra desde todas direcciones. La mayoría de fragmentos corresponden a núcleos atómicos o electrones. Las partículas de rayos cósmicos viajan a prácticamente la velocidad de la luz, lo que significa que tienen una gran energía. Algunas de ellas incluso contienen más energía que cualquier otra partícula observada en la naturaleza. Los rayos cósmicos de mayor energía contiene cientos millones de veces más energía que la

partícula con mayor energía hoyada en la naturaleza.

Este fenómeno de la Física fue descubierto en 1912 por Hess y Kohlhorster [81], y algunas de sus propiedades siguen siendo un misterio después de más de un siglo. Un ejemplo es el origen de los rayos, la mayoría de los científicos sospechan que el origen de los rayos cósmicos está relacionado con las *supernovas*, aunque no descartan otro tipo de fuentes [82]. Además, no es del todo claro como las supernovas pueden generar estos rayos cósmicos tan rápido.

Para aprender más sobre la naturaleza de este fenómeno, los científicos miden la energía y la dirección de los rayos conforme llegan a la tierra. Los rayos cósmicos de baja energía se miden utilizando globos aerostáticos y satélites situados por encima de la atmósfera terrestre, mientras que para los rayos cósmicos de alta energía, es más eficiente medirlos indirectamente observando la cascada de partículas que produce.

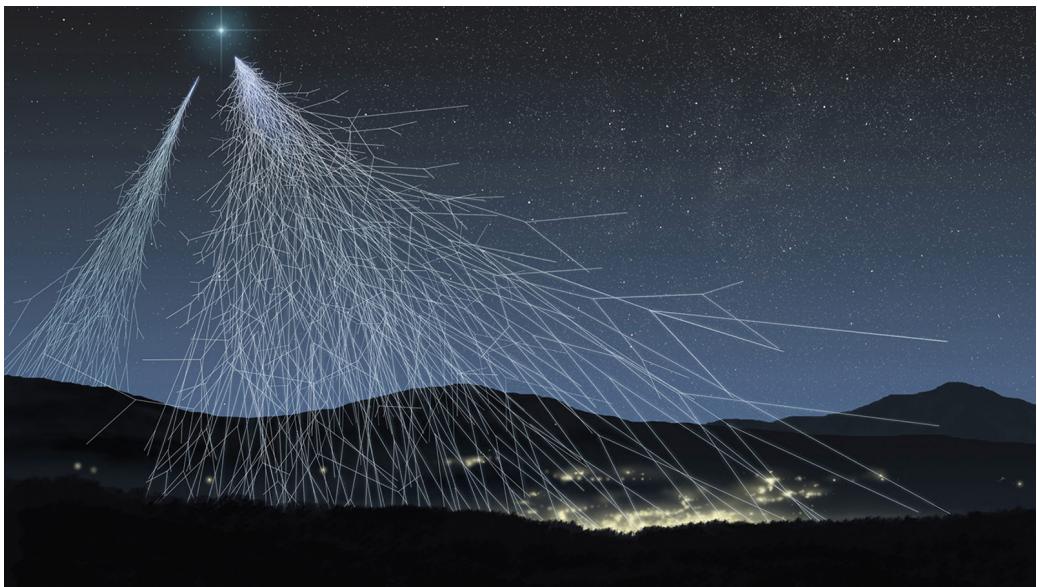


Figura 7.1: Cascada atmosférica extensa. (Observatorio Pierre Auger)

Una *cascada atmosférica extensa* [83] se produce cuando un rayo cósmico de alta energía (y de alta velocidad) penetra en la atmósfera. Cuando una partícula colisiona violentamente con las moléculas de aire, se fragmenta generando hadrones. Los fragmentos desprendidos a su vez colisionan con otras partículas del aire, produciendo así una cascada donde la energía de la

partícula original se dispersa entre millones de partículas que caen hacia la tierra (ver figura 7.1). Al estudiar las *cascadas atmosféricas*, los científicos pueden medir algunas propiedades de las partículas originales que llegaron a la atmósfera, también llamadas *primarios*.

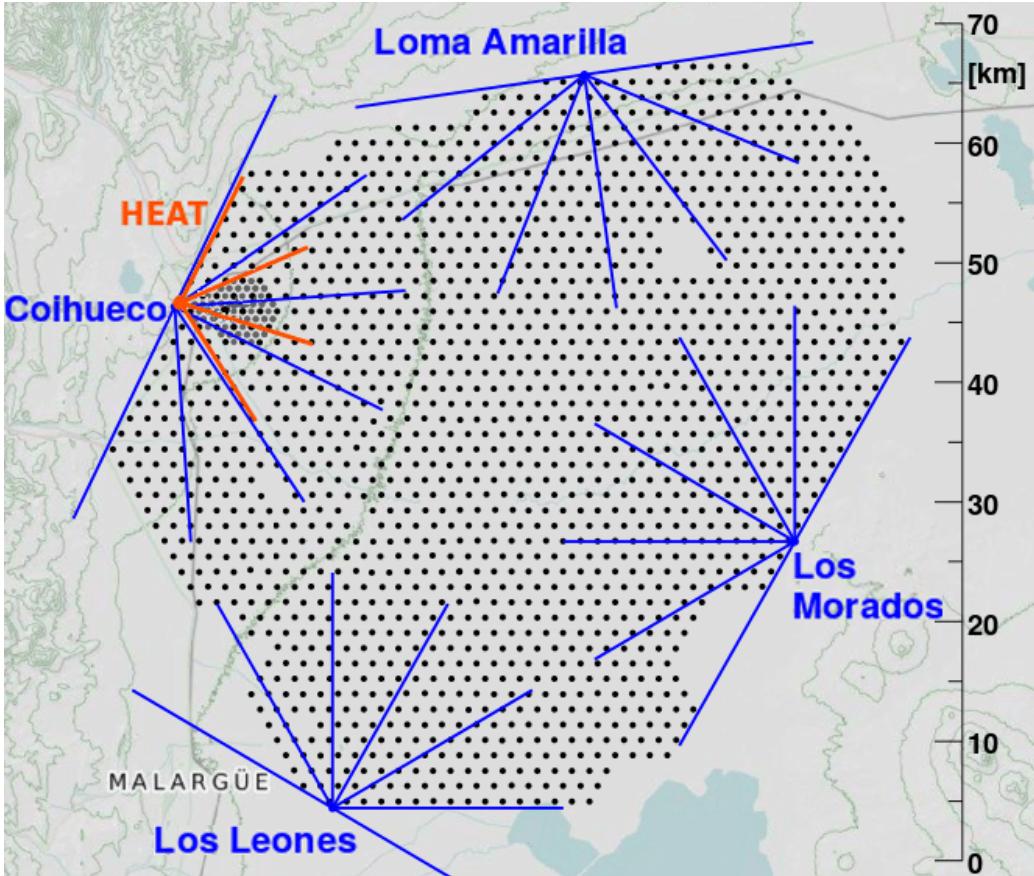


Figura 7.2: Mapa del observatorio de Pierre Auger. Cada punto negro representa un detector WCD

El Observatorio de Pierre Auger [14] se propuso para descubrir y entender la fuentes de los rayos cósmicos de energía más altas. El observatorio, situado en la ciudad de Malargüe, en la provincia de Mendoza, Argentina, es una colaboración única entre 18 países, cuya construcción empezó en 2002 y finalizó en 2008. El observatorio es un detector híbrido, utiliza un detector de gran superficie (SD) y un detector de fluorescencia (FD). El SD se compone de 1660 WCDs situados estratégicamente formando una malla triangular. En esta malla, los detectores están separados con una distancia de 1500 metros. Además, existe otra malla más pequeña cuyos detectores están separados 750

metros. En la figura 7.2 se muestra la distribución de los detectores.

Los WCDs del Observatorio de Pierre Auger consisten en tanques de agua de 3.6 metros de diámetro, que contienen 12,000 litros de agua ultrapura cada uno. En estos tanques están colocados tres distribuidos simétricamente, los cuales se encargan de medir la radiación Cherenkov. La señal de estos corresponden a la combinación de la señal muónica y electromagnética de la *cascada atmosférica extensa*. Como se puede intuir, una sola partícula primaria puede producir una señal en multiples , incluso en múltiples WCDs. Lo cual complica el análisis de la naturaleza de la partícula al tener que estudiar las relaciones entre las señales de los diferentes detectores.

7.1.2. DEFINICIÓN FORMAL DEL PROBLEMA

Los experimentos recogidos en este trabajo están basados en datos de simulaciones, en lugar de los datos reales. En concreto, se componen diferentes herramientas para la simulación de *cascadas atmosférica extensas*. El flujo de generación de los datos se muestra en la figura ???. CORSIKA [84] se utilizada para la simulación detallada de como se desarrolla la *cascada atmosférica extensa* en la atmósfera. Las interacciones hadrónicas se modelan utilizando QGSJET-II [85] o EPOS-LHC [86]. La señales de los WCDs producidas por las partículas se generan utilizando el software *offline* de Auger [87]. Finalmente, los datos de las simulaciones se almacenan en formato ROOT [88] para su procesamiento.

Como es de intuir, este tipo de simulaciones requieren una cantidad de recursos de espacio y computacionales enorme, por este motivo, se utilizada una fracción reducida de los datos. Para esta fracción de los datos, se han recogido alrededor de 20000 muestras para cada tipo de primario. En este caso, los tipos de primarios disponibles son: helio, hierro, proton, oxígeno. Para cada primario se han separado los datos en un conjunto de entrenamiento y otro de test. Siendo la distribución de los datos según el número de ejemplos la siguiente:

Primary	Training set	Test set	Total
Helium	16007	4001	20008
Iron	16019	4004	20023
Proton	16026	4006	20032
Oxygen	16021	4005	20026

Como se ha descrito anteriormente, los datos se basan en la simulación de la señal recogida en los WCDs. Esta señal recoge tanto la parte muónica μ , como la parte electromagnética em . A simple vista, la señal muónica se puede utilizar para separar entre las diferentes tipos de primario. Pero al existir varios en un mismo detector, pueden existir relaciones entre las señales de cada fotomultiplicador. Para poder atajar este problema con ML/DL es necesario encontrar una representación del problema tal que nos permita utilizar las señales de muónicas capturadas por los para clasificar entre primarios. Una representación utilizada en trabajos previos, consiste en integrar la señal muónica, obteniendo un único valor real para cada . Disponiendo así de un vector con N valores reales, tantos como haya en el WCD.

La representación que se propone en este trabajo consiste en utilizar la media de las señales de los . Utilizando la media de las tres señales podemos profundizar en la información recogida en la señal, en lugar de condensar toda esa información en un solo número real (la integral). Elegir la granularidad con la que se analizan los datos es uno de los retos más importantes de este problema. Como se ha mencionado al principio del capítulo, una *cascada atmosférica extensa* puede producir una señal en multiples dentro de un mismo detector WCD, pero además, puede afectar a varios detectores vecinos. Teniendo esto en cuenta, el problema se puede modelar a nivel de , a nivel de WCD, o a nivel de estación. En nuestro caso lo vamos a analizar a nivel de WCD.

Como trabajar con la señal en crudo puede ser muy costosa en términos de memoria, y recursos de CPU, los valores de cada vector se extraen a partir de la salida de *Offline de Auger*. La señal recogida está discretizada por lo que se puede utilizar como un vector de tamaño fijo. Sin embargo, según los expertos, la traza puede ser caracterizada en gran medida mediante un

conjunto más pequeño de variables que se describen a continuación:

Variables extraídas directamente de las simulaciones

- Energía Monte Carlo · E · : La energía total (en EeV, Exaelectron Voltios) del rayo cósmico primario (transformada con \log_{10})
- Ángulo de Zenit Monte Carlo Θ : Ángulo en grados entre el zenith y la trayectoria del rayo cósmico primario.
- Distancia al núcleo r : Distancia entre cada estación y la posición estimada del núcleo de la cascada, medida en metros.
- Señal total S_{total} : Número real en muones equivalentes verticales (VEMs) de la señal capturada por los WCD.
- Longitud de la traza: Tamaño del vector de la señal recogida. La señal está discretizada en *bins* de 25 nanosegundos.

Variables generadas mediante ingeniería de atributos:

- Ángulo Azimuth ζ : medido en radianes.
- Tiempo de subida $t_{1/2}$: medido en nanosegundos.
- Tiempo de caída: Tiempo en el que la señal empieza a descender.
- Área sobre el punto máximo de la señal: Suma de todas las señales en cada traza dividida por el máximo valor en cada traza.

Estas son las variables utilizadas para todos los experimentos llevados a cabo en este capítulo.

7.2. Procedimiento

Para el desarrollo de los diferentes experimentos se utilizado el framework *ml-experiment* descrito en el capítulo anterior. Para cada algoritmo de ML o arquitectura de DL se ha implementado un script de entrenamiento y un fichero de configuración asociado. Como la cantidad de datos o la dimensionalidad de los mismos no son relativamente grandes, se ha podido aplicar optimización de hiperparámetros. Para ello, el fichero de configuración asociado

a cada algoritmo define un Grupo de experimentos (ver *Diseño y desarrollo del framework*) con un espacio de hiperparámetros diseñado especialmente para cada tipo de modelo. Estos grupos de experimentos se han ejecutado de manera local aprovechando todos los núcleos de la CPU. Por otra parte, el proyecto de experimentación se llevado a cabo teniendo en cuenta los aspectos críticos de la reproducibilidad descritos en *Fundamentos*, y aplicando las buenas prácticas de MLOps.

The screenshot shows the MLflow interface for the XGBoost experiment. At the top, there's a navigation bar with 'mlflow' and links to 'GitHub' and 'Docs'. Below it, a sidebar lists various experiments: Default, Autoencoder V1, Stacked Autoencoder, Autoencoder V2, MLP, SVM, XGBoost (which is selected and highlighted in blue), and Variational Autoencoder. The main content area is titled 'XGBoost' and shows the following details:

- Experiment ID:** 98
- Artifact Location:** ./miruna/98
- Description:** [empty]
- Search Runs:** metrics.rmse < 1 and params.model = "tree"
- Filter Params:** alpha, lr
- Filter Metrics:** rmse, r2
- State:** Active
- Search** button

Below this, a table displays the 'Showing 100 matching runs' for the XGBoost experiment. The columns are: Date, User, Run Name, Source, Version, Tags, Parameters, and Metrics. Two rows of data are visible:

Date	User	Run Name	Source	Version	Tags	Parameters	Metrics
2020-04-18 19:32:56	antonio	Trial 9	default...	257d90	CPU: Intel(R) Core(TM) i7-97... Numpy seed: 1234 Python: 3.7.7 (default, Mar 10...	early_stopping_r... None gamma: 0 learning_rate: 0.473511224221... max_depth: 5 maximize: False min_child_weight: 0.412616292407... n_estimators: 3000 num_boost_round: 3000 num_class: 4 objective: multi:softmax random_state: 1234 subsample: 0.940403106541... verbose_eval: False	eval-merror: 0.123424 train-merror: 0 val_accuracy: 0.876576
2020-04-18 19:32:56	antonio	Trial 6	default...	257d90	CPU: Intel(R) Core(TM) i7-97... Numpy seed: 1234 Python: 3.7.7 (default, Mar 10...	early_stopping_r... None gamma: 0 learning_rate: 0.464122382827... max_depth: 5 maximize: False min_child_weight: 0.135409190052... n_estimators: 3500 num_boost_round: 3500 num_class: 4 objective: multi:softmax random_state: 1234 subsample: 0.934417888844... verbose_eval: False	eval-merror: 0.121613 train-merror: 0 val_accuracy: 0.878387

Figura 7.3: Todos los experimentos ejecutados con sus parámetros, métricas, artefactos, y otros metadatos, se almacenan en un servidor de MLFlow en local

Por una parte, el procedimiento de partición y procesado de los datos se realiza desde una interfaz compartida por todos los scripts de entrenamiento utilizando las *DataLoader* (ver Manual). En concreto, se ha definido dos *DataLoaders* uno para los autoencoders y otro para el resto de algoritmos. Para los autoencoders, se genera una pareja de entrenamiento-validación para cada tipo de primario (ver Listing 7.1). Para los modelos de aprendizaje supervisado, los datos de entrenamiento de todos los primarios se unifican en un solo conjunto de datos y posteriormente se extrae el conjunto de validación (ver Listing 7.2). Finalmente, tanto para el conjunto unificado como para los conjuntos separados, se aplica un *reescalado* mediante *estandarización* (*z-score*).

Por otro lado, las semillas para la partición, procesado, y entrenamiento de

modelos se establecen y quedan almacenadas como metadatos en cada experimento. Esto nos asegura que todos los modelos son entrenados y validados con los mismos datos, así como facilita la *replicabilidad* del experimento. Además, los parámetros, métricas y artefactos de cada experimento están almacenados en el servidor de *MLFlow* (ver figura 7.3), permitiendo visualizar y comparar entre los modelos y entre las diferentes configuraciones de hiperparámetros para cada algoritmo. Finalmente, la información relativa al hardware y el software donde se han ejecutado los experimentos también queda almacenada, en concreto, la información relativa al hardware que se ha recogido es la siguiente:

Tag	Value
CPU Info	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Python	3.7.7 (default, Mar 10 2020) [Clang 11.0.0
Version	(clang-1100.0.33.17)]
GPU Info	-

En cuanto a la métrica utilizada para evaluar los modelos, después de conocer que no existe ninguna preferencia respecto a un primario u otro, y que el conjunto de datos está balanceado, se ha decidido utilizar la precisión (*accuracy*) como métrica. Para evaluar los modelos se utiliza una partición aleatoria con un ratio 80-20 para entrenamiento y validación respectivamente. Por otro lado, el conjunto de test se utiliza exclusivamente para evaluar el modelo final seleccionado.

Listing 7.1: Implementación del DataLoader para autoencoders separados.

```
1 Dataset = Tuple[np.ndarray, np.ndarray]
2 VALIDATION_SPLIT = 0.2
3 SEED = 1234
4
5
6 class SplitDataLoader(DataLoader):
7     @classmethod
8     def load_data(cls) -> Tuple[
9         List[Dataset],
10        List[Dataset]]:
11         train_files = glob.glob('data/raw/QGSJet--train.txt'
12                                )
12         train_datasets, val_datasets, = [], []
13         datasets = [np.genfromtxt(file, delimiter=',')
14                     for file in train_files]
15
16         for i, dataset in enumerate(datasets):
17             dataset = dataset[:, 3:-1]
18             class_vector = np.full(dataset.shape[0], i)
19
20             X_train, X_val, y_train, y_val =
21                 train_test_split(
22                     dataset, class_vector,
23                     test_size=VALIDATION_SPLIT,
24                     random_state=SEED)
25
26             scaler = MinMaxScaler()
27             X_train = scaler.fit_transform(X_train)
28             X_val = scaler.transform(X_val)
29             train_datasets.append((X_train, y_train))
30             val_datasets.append((X_val, y_val))
31
31     return train_datasets, val_datasets
```

Listing 7.2: Implementación de DataLoader para los algoritmos de aprendizaje supervisado.

```
1  class UnifiedDataLoader(DataLoader):
2      @classmethod
3      def load_data(cls) -> Tuple[np.ndarray,
4                                     np.ndarray,
5                                     np.ndarray,
6                                     np.ndarray]:
7          train_files = glob.glob('data/raw/QGSJet--train.txt
8                                  ')
9          datasets = [np.genfromtxt(file, delimiter=',')
10                     for file in train_files]
11          X, y = [], []
12          for i, dataset in enumerate(datasets):
13              X.append(dataset[:, 3:-1])
14              y.append(np.full(dataset.shape[0], i))
15
16          X = np.concatenate(X, axis=0)
17          y = np.concatenate(y, axis=0)
18
19          X_train, X_val, y_train, y_val = train_test_split(
20              X, y, test_size=VALIDATION_SPLIT, random_state=
21              SEED)
22          scaler = StandardScaler()
23          X_train = scaler.fit_transform(X_train)
24          X_val = scaler.transform(X_val)
25          return X_train, X_val, y_train, y_val
```

7.3. Modelos considerados

Entre los algoritmos candidatos, se han empleado Support Vector Machine (SVM), Xgboost, y Autoencoder (simple, profundo, y variacional). Como se ha comentado anteriormente, para cada tipo de modelo se utiliza un espacio de hiperparámetros y se ejecuta un grupo de experimentos (entre 50-100 configuraciones distintas). En las siguientes secciones se especifican el el espacio de hiperparámetros, detalles de implementación, así como otra información relativa a cada experimento.

7.3.1. SVM

El primer algoritmo con el que se experimentó fue SVM [89]. SVM es un algoritmo de aprendizaje supervisado cuya función de coste tienen como objetivo maximizar el margen entre clases. Además, permite utilizar el *kernel trick* para entrenar modelos con funciones de decisión complejas mapeando los datos a un espacio de dimensión superior. La implementación de SVM utilizada es LinearSVC de *sklearn* como se puede ver en Listing 7.3. En concreto, se utiliza un SVM linear con un transformador basado en *aproximación del kernel*, una forma de mapear los datos más eficiente que el *kernel trick*, pero también inexacta. Para la aproximación del kernel se utiliza el método de Nystroem, el cual es un método genérico para aproximaciones de bajo rango de kernels. A rasgos generales, Nystroem muestrea un número limitado de ejemplos de entrenamiento (100 por defecto en el caso de *sklearn*) y les aplica el kernel real. Con esos ejemplos transformados construye una matriz de transformación de bajo rango que aproxima el kernel aplicado [90]. Esto permite poder entrenar SVMs de manera más eficiente, incluso utilizar *online learning*, en nuestro caso, nos permite aumentar el número de configuraciones distintas para la optimización de hiperparámetros. Los detalles del experimento y de la configuración de hiperparámetros son las siguientes:

Cuadro 7.3: Información general sobre el experimento SVM

Detalle	Valor
Nombre del experimento	SVM
Número de configuraciones	50
Optuna Sampler	TPE

Cuadro 7.4: Espacio de hiperparámetros para el experimento SVM

Parámetro	Distribución
C	loguniform(0.001, 1000)
kernel	choice(['rbf', 'poly', 'linear'])
gamma	choice(['scale', 'auto'])
degree	range(2, 5) (solo aplica si kernel = 'poly')

Listing 7.3: Implementación del script de entrenamiento para SVM.

```
1 from typing import *
2 from modelling.utils.data import UnifiedDataLoader, SEED
3 import numpy as np
4 from ml_experiment import job
5 from sklearn.svm import LinearSVC
6 from sklearn.kernel_approximation import Nystroem
7
8
9 @job(data_loader=UnifiedDataLoader())
10 def main(C: float = 1.0, kernel: str = 'rbf',
11         degree: int = 3, gamma: Any = 'scale'):
12     np.random.seed(SEED)
13     X_train, X_val, y_train, y_val = \
14         UnifiedDataLoader.load_data()
15
16     if isinstance(gamma, str):
17         if gamma == 'scale':
18             gamma = 1.0 / (X_train.shape[1] * X_train.var())
19         if gamma == 'auto':
20             gamma = 1.0 / X_train.shape[1]
21
22     if kernel != 'linear':
23         degree = degree if kernel == 'poly' else None
24         feature_map = Nystroem(
25             kernel=kernel, gamma=gamma, degree=degree)
26         X_train = feature_map.fit_transform(X_train)
27         X_val = feature_map.transform(X_val)
28
29     model = LinearSVC(C=C)
30     model.fit(X_train, y_train)
31     accuracy = model.score(X_val, y_val)
32     return {'val_accuracy': accuracy}
33
34
35 if __name__ == '__main__':
36     main()
```

7.3.2. XGBOOST

El segundo modelo con el que se ha experimentado a sido XGBoost [8]. XGBoost es un algoritmo *gradient boosting* [91] flexible, escalable, y portable. Permite *boosting* de arboles en paralelo, así como definir funciones de coste propias [91]. Uno de los inconvenientes a la hora de realizar HPO con XGBoost (y con los ensamblados de árboles en general), es la cantidad de hi-

perparámetros distintos que se pueden modificar. Esto hace que el espacio de hiperparámetros sea demasiado grande como para cubrirlo uniformemente de manera acertada. Por este motivo, es de especial importancia el uso de algoritmos de HPO más novedosos que GridSearch para poder explorar zonas del espacio más interesantes. En concreto, se utiliza TPE (*Tree-structured Parzen Estimator*) al igual que en el resto de modelos. A rasgos generales, TPE ajusta un modelo Modelo Gaussiano de Mezcla (*GMM*) $l(x)$ al conjunto de hiperparámetros asociados a los mejores valores de la métrica hasta el momento, y luego otro GMM $g(x)$ con el resto de valores de hiperparámetros. Finalmente, se coge el parámetro x que maximice el ratio $l(x)/g(x)$.

Cuadro 7.5: Información general sobre el experimento XGBoost.

Detalle	Valor
Nombre del experimento	XGBoost
Número de configuraciones	100
Optuna Sampler	TPE

Cuadro 7.6: Espacio de hiperparámetros para el experimento XGBoost.

Parámetro	Distribución
num_boost_round	choice([10, 100, 250, 500, 1000, 2500])
learning_rate	loguniform(0.0001, 1)
max_depth	range(2, 15)
gamma	choice([0, 0.5, 2, 10, 20])
min_child_weight	loguniform(0.01, 1)
subsample	uniform(0.1, 0.9)

Además, para mejorar el rendimiento del trabajo de HPO, se hace uso de varios *callbacks* de XGBoost para *podar* configuraciones no prometedoras (ver Listing 7.4). De esta forma, conforme se van probando configuraciones, los modelos menos prometedores no terminan de construirse, ahorrando así cómputo. El primero de ellos es *XGBoostPruningCallback* de *Optuna* que permite abortar configuraciones no prometedoras según el espacio de

hiperparámetros ya explorado y las métricas recogidas anteriormente. El segundo de ellos es un *callback* de *Early Stopping*, se utiliza para parar el entrenamiento cuando no existe mejora (o incluso empeora) en sucesivas iteraciones. En este caso, el número de iteraciones mínimo donde debe haber mejora es de *numero_estimadores*/10. Por último, se utiliza el *callback record_evaluation* para guardar los resultados del entrenamiento a lo largo de las iteraciones y evitar el cómputo de evaluar el modelo al final del entrenamiento (ya lo hace XGBoost internamente).

Listing 7.4: Implementación del script de entrenamiento para Xgboost.

```
1 @job(data_loader=UnifiedDataLoader(),
2       autologging_backends=AutologgingBackend.XGBOOST)
3 def main(n_estimators: int, learning_rate: float,
4          max_depth: int, gamma: float,
5          subsample: float, min_child_weight: float):
6     np.random.seed(SEED)
7
8     X_train, X_val, y_train, y_val = \
9         UnifiedDataLoader.load_data()
10    train_data = xgb.DMatrix(X_train, label=y_train)
11    val_data = xgb.DMatrix(X_val, label=y_val)
12    params = dict(random_state=SEED,
13                  learning_rate=learning_rate,
14                  max_depth=max_depth, gamma=gamma,
15                  subsample=subsample,
16                  min_child_weight=min_child_weight,
17                  num_class=len(np.unique(y_train)),
18                  objective='multi:softmax')
19
20    trial = Trial.get_current()
21    evallist = [(val_data, 'eval'), (train_data, 'train')]
22    eval_result = {}
23    patience = max(10, n_estimators // 10)
24    callbacks = [
25        XGBoostPruningCallback(trial, 'eval-merror'),
26        xgb.callback.early_stop(patience, verbose=False),
27        xgb.callback.record_evaluation(eval_result)
28    ]
29
30    xgb.train(params, train_data,
31               num_boost_round=n_estimators,
32               evals=evallist,
33               callbacks=callbacks,
34               verbose_eval=False)
35
36    accuracy = 1 - eval_result['eval']['merror'][-1]
37    return {'val_accuracy': accuracy}
38
39 if __name__ == '__main__':
40     main()
```

7.3.3. AUTOENCODER V1

Cuadro 7.7: Información general sobre el experimento Autoencoder V1

Detalle	Valor
Nombre del experimento	Autoencoder V1
Número de configuraciones	100
Optuna Sampler	TPE
Épocas de entrenamiento	200

Para los algoritmos basados en autoencoders, se han propuesto dos experimentos por separado. El primero de ellos, consiste en un autoencoder *denoising* con probabilidad de descarte p y con soporte para *Tied-weights* (ver Tabla 7.8). Además, tiene las siguientes características:

- La red consta de únicamente tres capas, la capa de entrada, la capa del código cuyo número de neuronas viene dado por el parámetros *encoding_dim*, y la capa de salida.
- Permite dos funciones de activación distintas: *RELU* y *SELU*. *SELU* es un función de activación moderna (2017), que junto con *Swish* y *ELU*, suele arrojar resultados mejores en comparación con *RELU*. Aunque la mayoría de investigación al respecto está enfocada a problemas de clasificación de imágenes, lo cual hace realmente difícil la comparativa entre ambas aplicadas a *autoencoders*.
- PCA y autoencoders (sobre todo lineales) tienen muchas similaridades, pero interesantes de PCA. PCA y autoencoders (sobre todo lineales) tienen muchas similaridades, pero no pueden ofrecer ortogonalidad, ni vectores unitarios en la base. En este trabajo se han implementado dos restricciones a nivel de capa, para asegurar la ortonormalidad, una restricción para ortogonalidad propia (ver Listing 7.5), y una restricción para el tamaño unitario de las componentes del espacio latente utilizando *UnitNorm* de *Keras*. Existen además otras propiedades de PCA que se han adaptado a autoencoders [92], como por ejemplo, forzar a que los componentes del espacio latente sean estadísticamente independientes. Dichas características se salen del alcance de este trabajo.

Cuadro 7.8: Espacio de hiperparámetros para el experimento Autoencoder V1

Parámetro	Distribución
encoding_dim	range(1, 20)
ps	uniform(0, 0.5)
lr	loguniform(0.001, 0.1)
activation	choice(['selu', 'relu'])
tied_weights	choice([false, true])
unit_norm_constraint	choice([false, true])
weight_orthogonality	choice([false, true])

Listing 7.5: Implementación del la restricción para ortogonalidad en las componentes del espacio latente en autoencoders. Fuente: <https://towardsdatascience.com/build-the-right-autoencoder-tune-and-optimize-using-pca-principles-part-ii-24b9cca69bd6>

```

1  class WeightsOrthogonalityConstraint(constraints.Constraint):
2      :
3      def __init__(self, encoding_dim, weightage=1.0, axis=0):
4          self.encoding_dim = encoding_dim
5          self.weightage = weightage
6          self.axis = axis
7
8      def weights_orthogonality(self, w):
9          if self.axis == 1:
10              w = K.transpose(w)
11          if self.encoding_dim > 1:
12              m = K.dot(K.transpose(w), w) - K.eye(self.
13                  encoding_dim)
14              return self.weightage * K.sqrt(K.sum(K.square(m
15                  )))
16          else:
17              m = K.sum(w ** 2) - 1.
18          return m
19
20      def __call__(self, w):
21          return self.weights_orthogonality(w)

```

7.3.4. AUTOENCODER V2

Para la segunda versión del autoencoder se han implementado dos características principales con respecto a la versión 1:

- El número de capas para el codificador y decodificador son mayores que 1. Es decir, el autoencoder V2 corresponde a un autoencoder *denoising* apilado. El número de capas a cada lado del código varía entre 2 a 4, y el número de neuronas por capa entre 2 a 20. Tanto el número de capas como el de neuronas se controla con el parámetro *encoding_dim* (ver Tabla 7.10). Por ejemplo, si *encoding_dim* = [5, 5, 4] el autoencoder tendrá la siguiente arquitectura: *Input* + *Dense(5)* + *Dense(5)* + *Dense(4)* + *Dense(5)* + *Dense(5)* + *Output*. Es decir, el vector *encoding_dim* representa el codificador entero incluido el código. y como el autoencoder es simétrico, el decodificador también esta definido con ese mismo vector.
- Se ha utilizado una política de entrenamiento con un esquema de actualización del ratio de aprendizaje. En concreto, se ha utilizado OneCycle [93] para adaptar el ratio de aprendizaje a lo largo de las épocas hasta llegar ha un máximo definido por el parámetro del experimento *lr* (ver Tabla 7.10). El motivo principal por el que se ha decidido utilizar una política de actualización del ratio de aprendizaje, es porque al aumentar la complejidad del autoencoder, el riesgo de sobreajuste aumenta, con OneCycle* conseguimos hacer que el propio ratio de aprendizaje actué de regularizador a lo largo del entrenamiento [93].

Cuadro 7.9: Información general sobre el experimento Autoencoder V2

Detalle	Valor
Nombre del experimento	Autoencoder V1
Número de configuraciones	90
Optuna Sampler	TPE
Épocas de entrenamiento	200

Cuadro 7.10: Espacio de hiperparámetros para el experimento Autoencoder V1

Parámetro	Distribución
encoding_dim	range(2, 20) × range(2, 5)
ps	uniform(0, 0.5)

Parámetro	Distribución
lr	loguniform(0.001, 0.1)
activation	choice(['selu', 'relu'])
tied_weights	choice([false, true])
one_cycle	choice([false, true])
unit_norm_constraint	choice([false, true])
weight_orthogonality	choice([false, true])

7.3.5. AUTOENCODER VARIACIONAL

El tercer y último tipo de autoencoder con el que se ha experimentado ha sido el variacional. En este caso, se ha implementado un VAE puro, es decir, sin capa de *Dropout*, *Tied-weights*, características de PCA, etc. Además, se ha utilizado una arquitectura sencilla para evitar el sobreajuste que consiste en 5 capas, capa de entrada, capa de salida, la capa del código, y una capa para el codificador y otra para el decodificador (ver Figura 7.4).

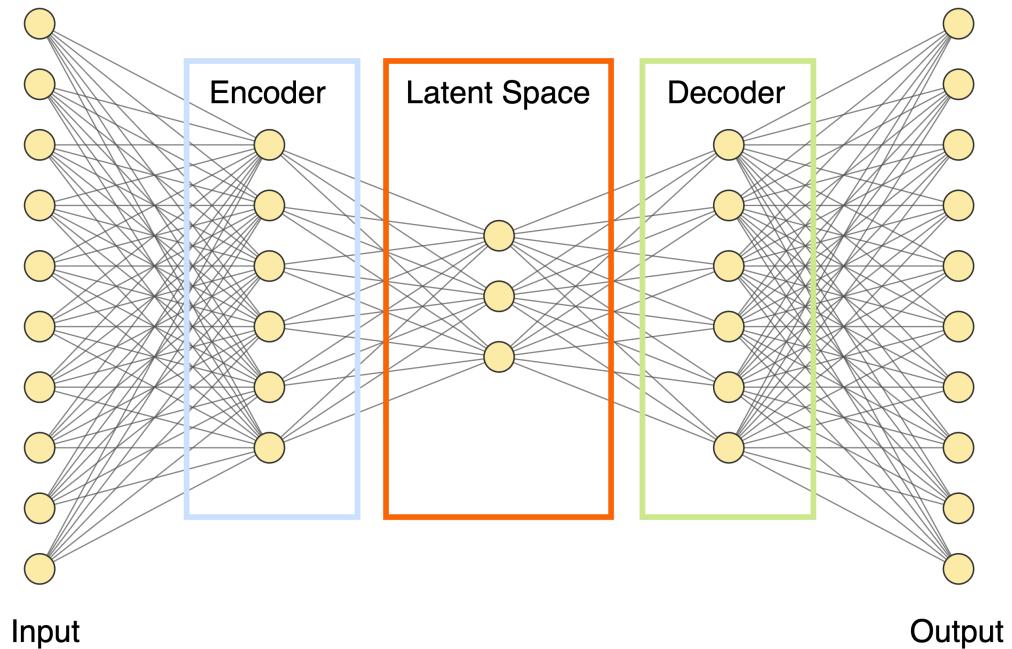


Figura 7.4: Arquitectura del autoencoder variacional implementado. Fuente propia

La arquitectura es siempre simétrica, es decir, las capas a un lado y a otro del código coinciden. Los únicos parámetros que se ajustan mediante HPO son: el número de neuronas en la capa del codificador y decodificador, el tamaño del código, y si se hace uso o no de *OneCycle* para entrenar el modelo (ver detalles en Tabla 7.12).

Cuadro 7.11: Información general sobre el experimento Autoencoder Variacional

Detalle	Valor
Nombre del experimento	Variational Autoencoder
Número de configuraciones	50
Optuna Sampler	TPE
Épocas de entrenamiento	200

Cuadro 7.12: Espacio de hiperparámetros para el experimento Autoencoder Variacional

Parámetro	Distribución
encoding_dim	range(2, 15)
latent_dim	range(2, 5)
activation	choice(['selu', 'relu'])
one_cycle	choice([false, true])

7.3.5.1. Truco de la reparametrización

Como se ha descrito en *Fundamentos*, para poder entrenar sobre una variable aleatoria es necesario hacer el *truco de la reparametrización*. Para ello, hay que definir una capa Lambda de Keras que permita ejecutar una función de Tensorflow o Pytorch arbitraria. En este caso, se definen dos capas densas μ y σ con número de neuronas igual al tamaño del código, y se utiliza el *Lambda* para transformar una variable aleatoria z en una combinación de μ y σ con un ruido ϵ (ver Listing 7.6). Recordemos que el truco de la reparametrización hace básicamente lo siguiente:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \text{ where } \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$$

Para:

$$\mathbf{z} \sim q_\phi(\mathbf{z} | \mathbf{x}^{(i)}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)} \mathbf{I})$$

Listing 7.6: Implementación del truco de la reparametrización para Keras. Fuente: Documentación oficial de Keras.

```

1 import keras.backend as K
2
3 def sampling(args):
4     """Reparameterization trick by sampling from an
       isotropic unit Gaussian.
5
6     # Arguments
7     args (tensor): mean and log of variance of Q(z|X)
8
9     # Returns
10    z (tensor): sampled latent vector
11    """
12    z_mean, z_log_var = args
13    batch = K.shape(z_mean)[0]
14    dim = K.int_shape(z_mean)[1]
15    # by default, random_normal has mean = 0 and std = 1.0
16    epsilon = K.random_normal(shape=(batch, dim))
17    return z_mean + K.exp(0.5 * z_log_var) * epsilon

```

7.3.5.2. Función de coste

Por otro lado, hay que cambiar la función de coste para este tipo de autoencoder, la cuál debe incluir el término de divergencia KL (Kullback–Leibler). Para ello, se hace uso de una función de conste personalizada para Keras (ver Listing 7.7).

Listing 7.7: Función de coste para VAE. Fuente: Documentación oficial de Keras.

```
1 import keras.backend as K
2
3 def create_loss(input_dim, inputs, outputs, mu, sigma):
4     # VAE loss = mse_loss or xent_loss + kl_loss
5     reconstruction_loss = mse(inputs, outputs) * input_dim
6     kl_loss = 1 + sigma - K.square(mu) - K.exp(sigma)
7     kl_loss = K.sum(kl_loss, axis=-1)
8     kl_loss *= -0.5
9     return K.mean(reconstruction_loss + kl_loss)
```

Básicamente se define la función de coste de un autoencoder básico (MSE normalmente), y se le suma $\mathbb{KL}(q_\theta(z | x_i) \| p(z))$.

7.3.6. CONSTRUCCIÓN DEL MODELO

Finalmente, la construcción del modelo se implementa de manera sencilla (ver Listing 7.8):

1. Se define la capa de entrada
2. Se define la capa (o capas) del encoder
3. Se define μ, σ y z
4. Se crea el decodificador
5. Por último, se construye el autoencoder conectando todas las partes de la arquitectura y añadiendo la función de coste personalizada.

Listing 7.8: Construcción del autoencoder VAE.

```
1 def create_model(
2     input_size: int,
3     encoding_dim: Union[List[int], int],
4     latent_dim: int,
5     lr: float,
6     activation: str = 'relu'):
7
8     np.random.seed(SEED)
9     decoding_dim = [input_size] + encoding_dim[:-1]
10
11    inputs = Input(shape=(input_size, ),
12                    name='encoder_input')
13    encoder = inputs
14
15    for i, units in enumerate(encoding_dim):
16        kwargs = {'input_shape': (input_size,)} if i == 0
17        else {}
18        encoder = Dense(units, activation=activation, **
19                         kwargs)(encoder)
20
21    mu = Dense(latent_dim, name='mu')(encoder)
22    sigma = Dense(latent_dim, name='log_var')(encoder)
23    z = Lambda(sampling,
24                output_shape=(latent_dim, ),
25                name='z')([mu, sigma])
26    encoder = Model(
27        inputs,
28        [mu, sigma, z],
29        name='encoder')
30
31    latent_inputs = Input(
32        shape=(latent_dim, ),
33        name='z_sampling')
34    decoder = latent_inputs
35
36    for i, units in enumerate(decoding_dim[::-1]):
37        layer_activation = 'sigmoid' if i == len(
38            encoding_dim) - 1 else activation
39        decoder = Dense(
40            units,
41            activation=layer_activation)(decoder)
42
43    decoder = Model(latent_inputs, decoder, name='decoder')
44    outputs = decoder(encoder(inputs)[2])
45    autoencoder = Model(inputs, outputs, name='vae_mlp')
46    loss = create_loss(input_size, inputs, outputs, mu,
47                       sigma)
48    autoencoder.add_loss(loss)
49    autoencoder.compile(SGD(learning_rate=lr), loss='mse')
50
51    return autoencoder, encoder, decoder
```

7.4. Resultados

En esta sección se analizan los resultados obtenidos para los diferentes modelos. En cada subsección aparecen recogidas las gráficas que relacionan los diferentes hiperparámetros con la métrica de validación. Además, en la tabla 7.13 se muestran los resultados de el mejor modelo para cada algoritmo.

Cuadro 7.13: Resultados para los mejores modelos de cada tipo sobre el conjunto de test. (Las métricas están redondeadas a 5 decimales).

Algoritmo	Parámetros	Métrica
SVM	C: 1e-4 degree: no aplica gamma: auto kernel: lineal	0.64036
XGBoost	gamma: 0 learning_rate: 0.42 max_depth: 5 min_child_weight: 0.4133 n_estimators: 4500 subsample: 0.9924	0.88519
Autoencoder V1	activation: selu encoding_dim: 10 learning_rate: 0.001 optimizer_name: Adam ps: 0.031 tied_weights: True unit_norm_constraint: False weight_orthogonality: False	0.69825

Algoritmo	Parámetros	Métrica
Autoencoder V2	activation: selu encoding_dim: [9, 9, 7] learning_rate: 0.00338 optimizer_name: SGD ps: 0.002 only_cycle: True tied_weights: True unit_norm_constraint: True weight_orthogonality: False	0.56267
VAE	activation: selu encoding_dim: 13 latent_dim: 3 learning_rate: 1e-04 optimizer_name: SGD only_cycle: True	0.38315

7.4.1. SVM

En general, el rendimiento de SVM para este problema es pobre. Exceptuando una configuración sin transformación de datos (kernel lineal), y con un parámetro de regularización C muy bajo, el cual fuerza al clasificador a tener margen entre clases grande. En cuanto al parámetro C , se ve claramente como los valores bajos ofrecen los mejores resultados. En cuanto al resto de parámetros, no se ve una tendencia clara sobre si un *kernel* o *gamma* determinado ofrece mejores resultados que el resto. De hecho, la configuración con mejor rendimiento se podría considerar un *outlier*. Esto no implica que el modelo no sea correcto, significa que deberíamos probar un número más alto de configuraciones para poder sacar conclusiones más exactas.

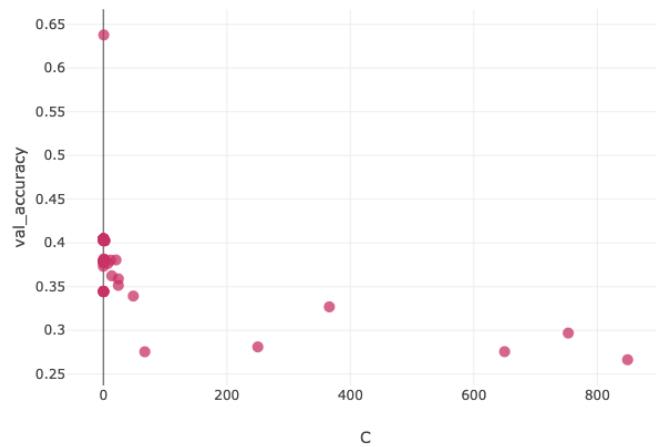


Figura 7.5: SVM. C vs Validation accuracy

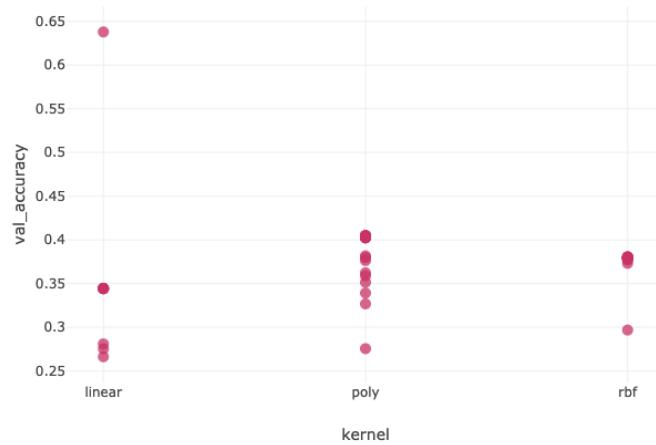


Figura 7.6: SVM. Kernel vs Validation accuracy

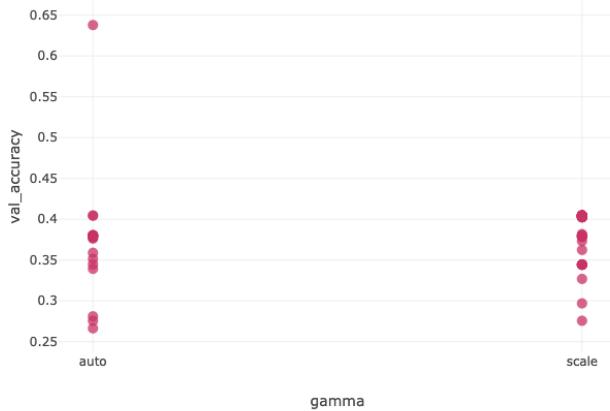


Figura 7.7: SVM. Gamma vs Validation accuracy

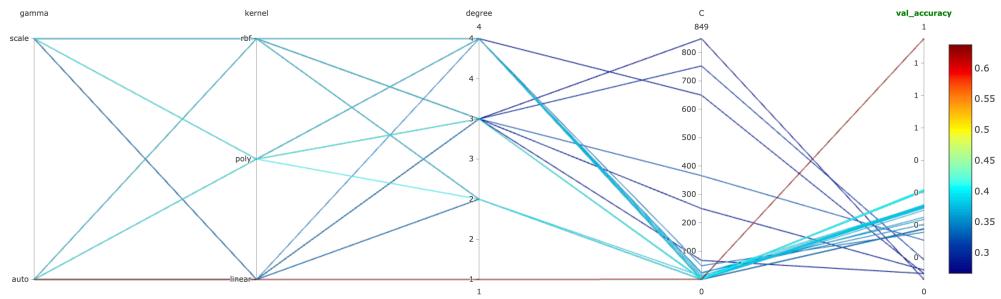


Figura 7.8: SVM. Parameters comparison

7.4.2. XGBOOST

En cuanto a XGBoost, todos los modelos funcionan muy bien de manera general. Viendo los datos recogidos, se puede argumentar que existe una correlación positiva entre el número de arboles del ensamblado (también llamado rondas) y la precisión en validación (Figura 7.10). Por otro lado, también existe una correlación positiva entre *subsample* (ratio de columnas con las que se entrena cada árbol) y la precisión (Figura 7.14). En cuanto a los parámetros del ratio de aprendizaje y *min_child_weight*, se puede apreciar que valores muy bajos deterioran el rendimiento, mientras que un *gamma* de 0 suele ofrecer los mejores resultados. (Figuras 7.11, 7.12, 7.9) El motivo por el que en el parámetro *gamma*, por ejemplo, los puntos no están uniformemente distribuidos entre cada valor del espacio de hiperparámetros,

es que al aplicar el algoritmo de optimización probabilístico TPE, el espacio de hiperparámetros se explora siguiendo las regiones más prometedoras, en lugar de explorarse uniformemente. Esto mismo ocurre con el parámetro *max_depth* (Figura 7.13), donde el valor de 5 ofrece los mejores resultados y es por eso que hay más configuraciones con ese valor que con otras.

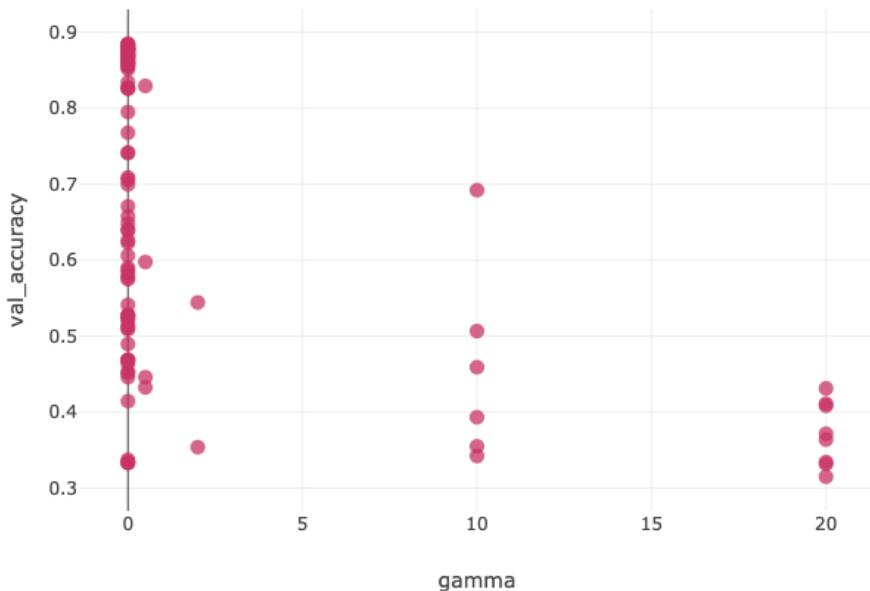


Figura 7.9: Xgboost. Gamma vs Validation accuracy

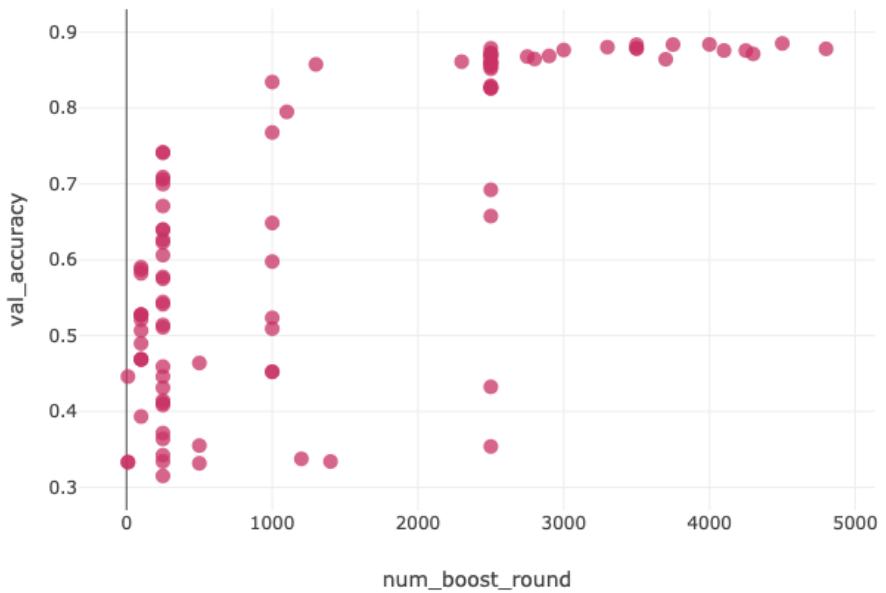


Figura 7.10: Xgboost. Number of trees vs Validation accuracy

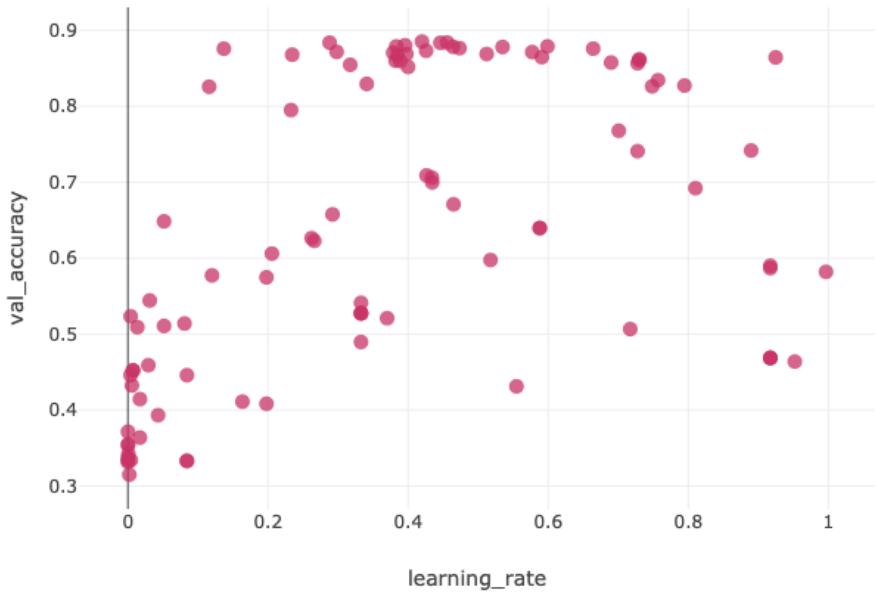


Figura 7.11: Xgboost. Learning rate vs Validation accuracy

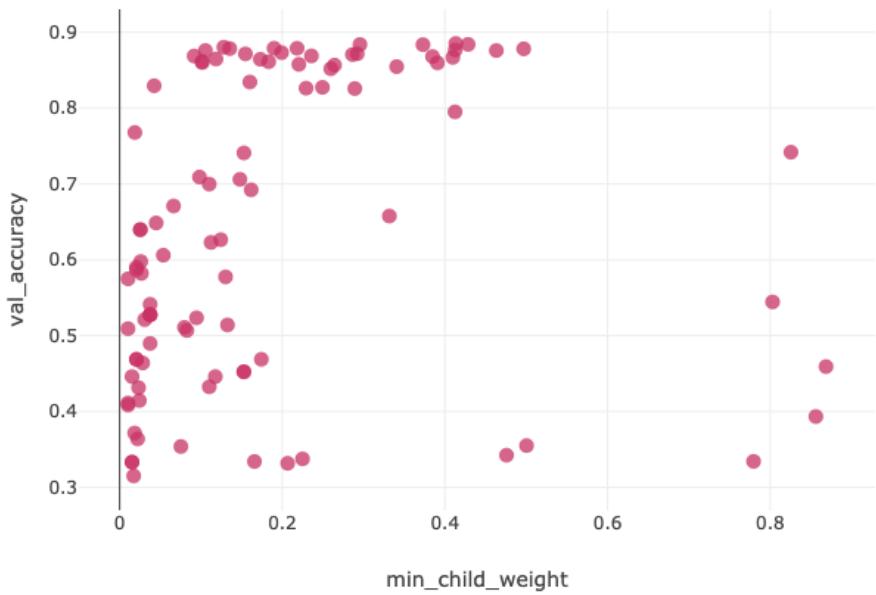


Figura 7.12: Xgboost. Min child weight vs Validation accuracy

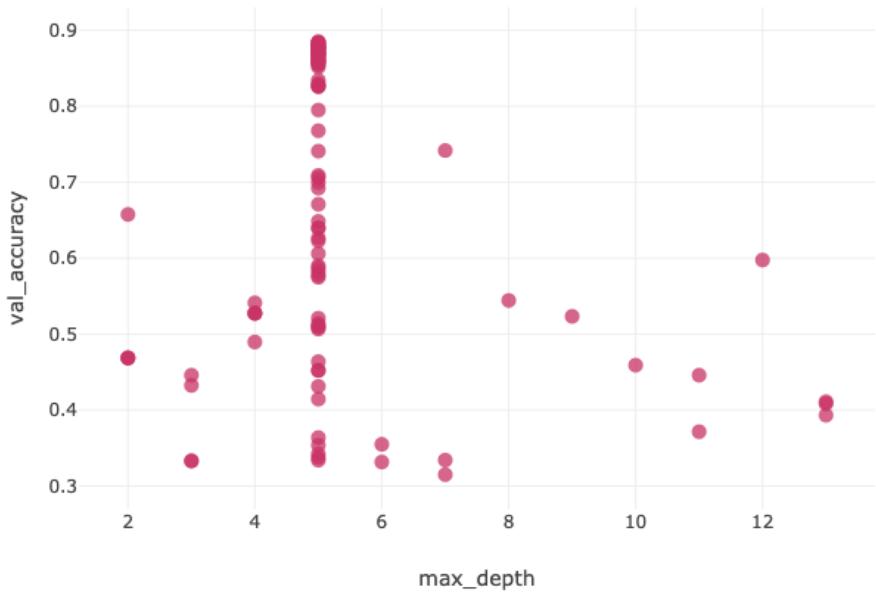


Figura 7.13: Xgboost. Max depth vs Validation accuracy

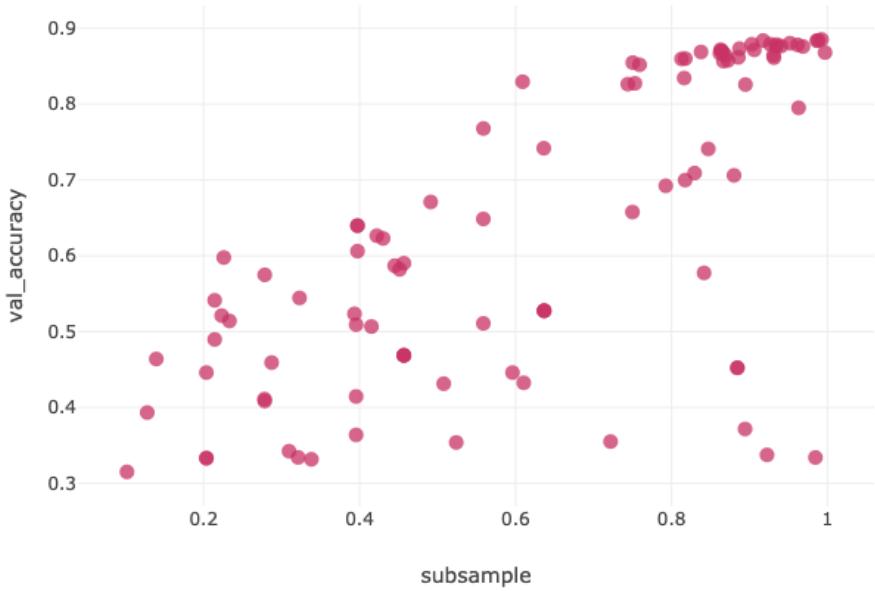


Figura 7.14: Xgboost. Subsample vs Validation accuracy

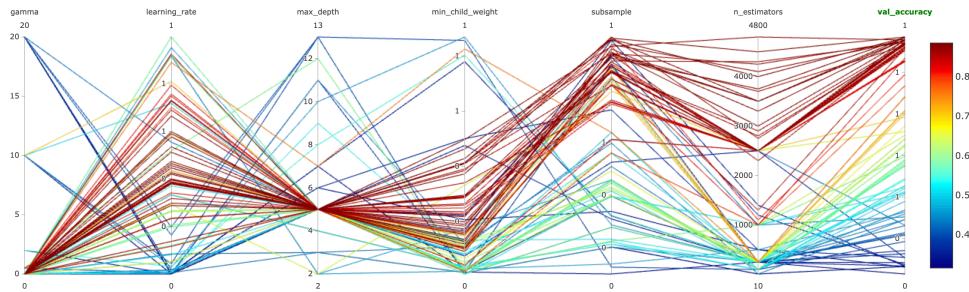


Figura 7.15: Xgboost. Parameters comparison

7.4.3. AUTOENCODER v1

Para la primera versión de autoencoder, el cuál sorprendentemente tiene un rendimiento mejor que SVM, podemos decir que las restricciones tanto de ortonormalidad no mejoran la precisión en validación. Por otro lado, vemos que el nivel de ruido (ps) influye positivamente cuando se aplica ligeramente (Figura 7.18). El ratio de aprendizaje no sigue un patrón bien definido, pero el algoritmo de HPO ha considerado como configuraciones más prometedoras aquellas con un ratio de aprendizaje más bajo (Figura 7.21). Por otro

lado, emplear la técnica de *Tied-weights* parece mejorar el rendimiento considerablemente (Figura 7.17). Finalmente, la dimensión del código óptima se encuentra entre 10 y 12, que coincide prácticamente con la dimensión de los datos de entrada (Figura 7.16).

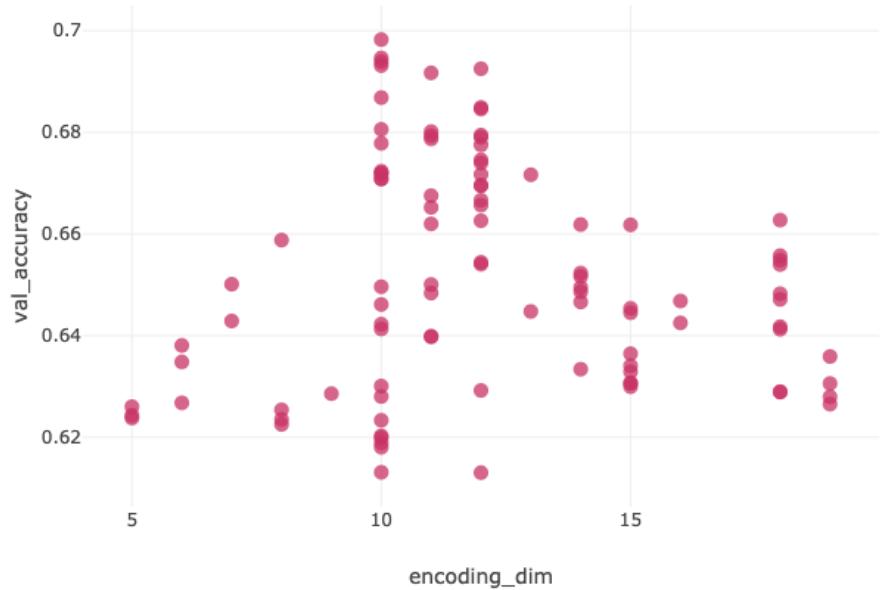


Figura 7.16: Autoencoder v1. Dimensión del código vs Validation accuracy

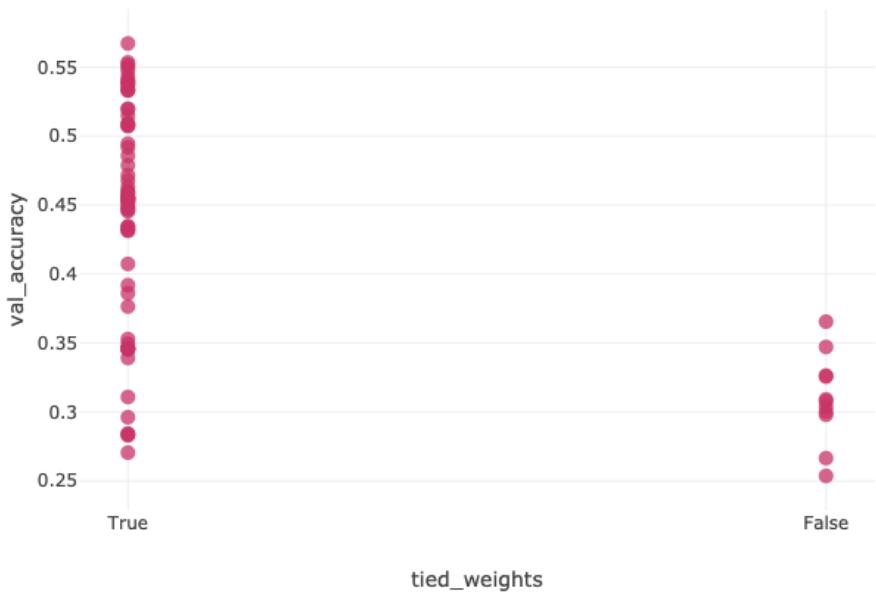


Figura 7.17: Autoencoder v1. Tied-weights vs Validation accuracy

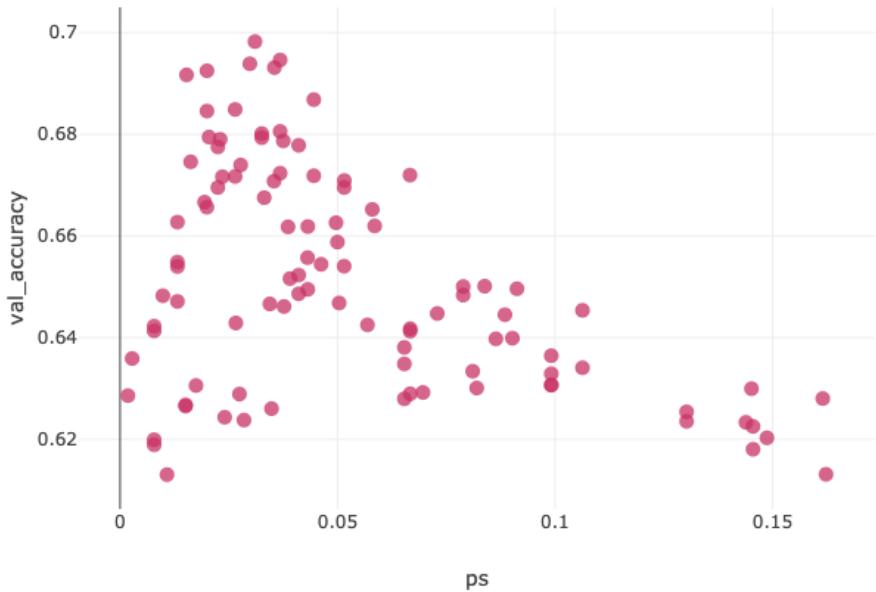


Figura 7.18: Autoencoder v1. Probabilidad de descarte (dropout) vs Validation accuracy



Figura 7.19: Autoencoder v1. Restricción de ortogonalidad vs Validation accuracy

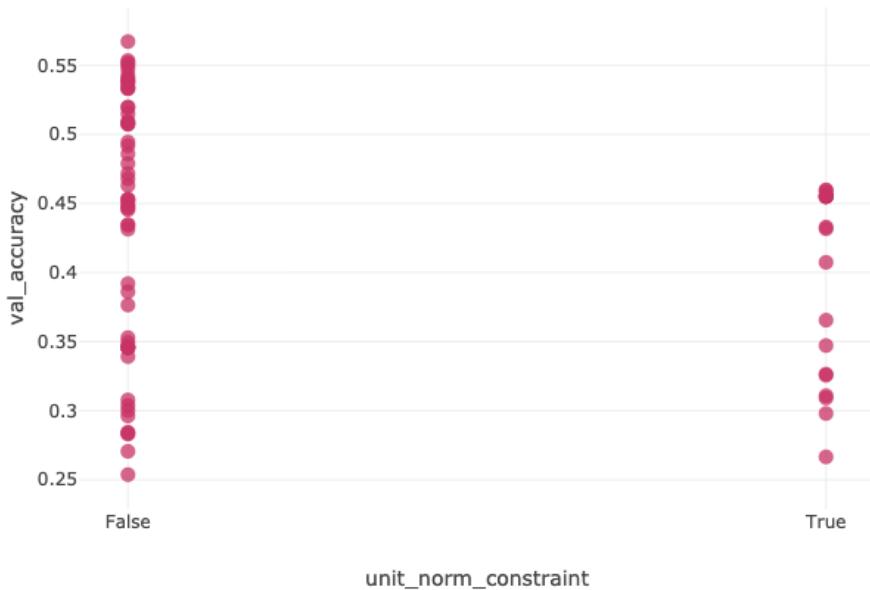


Figura 7.20: Autoencoder v1. Restricción UnitNorm vs Validation accuracy

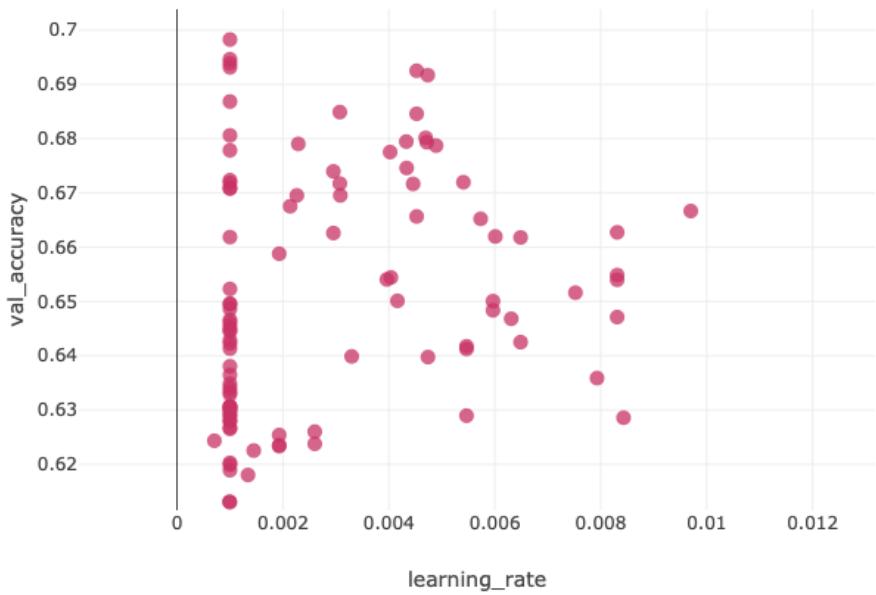


Figura 7.21: Autoencoder v1. Ratio de aprendizaje vs Validation accuracy

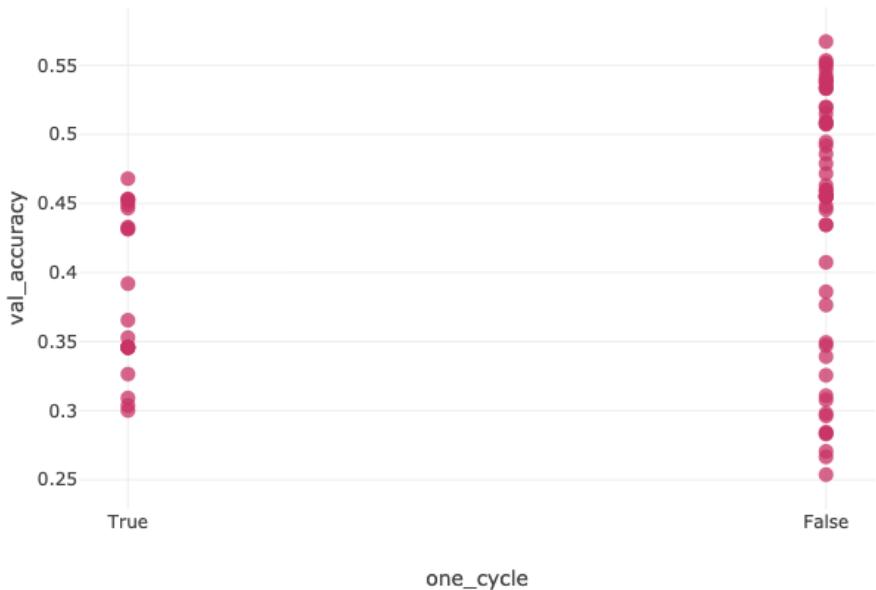


Figura 7.22: Autoencoder v1. OneCycle vs Validation accuracy

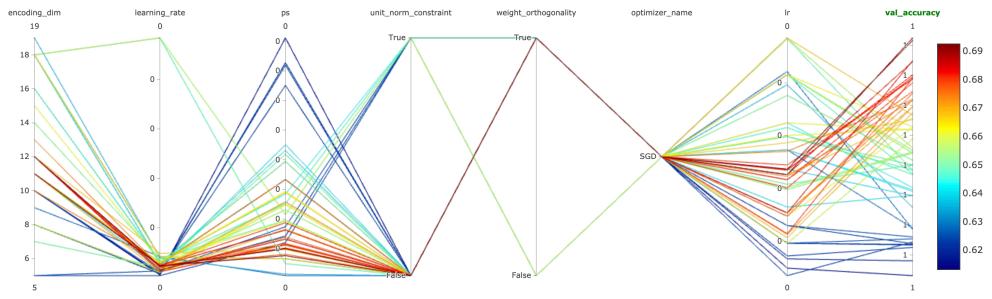


Figura 7.23: Autoencoder v1. Parameters comparison

7.4.4. AUTOENCODER V2

Para la segunda versión de autoencoder, la restricción de ortogonalidad no mejora el rendimiento (Figura 7.26), pero la restricción de pesos unitarios si que mejora ligeramente (Figura 7.27). También, podemos ver que tanto Tied-weight como OneCycle mejoran considerablemente el rendimiento (Figuras 7.29 7.30). Esto puede ser debido a que al incrementar la complejidad de la arquitectura, ambas técnicas permitan mantener el sobreajuste bajo mientras se aumenta el poder de predicción. Por otro lado, tanto el porcentaje de ruido como el ratio de aprendizaje parecen ser óptimos en niveles muy bajos (Figuras 7.25 7.28). Por último, se puede ver como tres capas por delante y por detrás de la capa del código es el número óptimo para esta arquitectura (Figura 7.24). A la luz de estos datos, podríamos argumentar que las arquitecturas menos profundas funcionan mejor para este problema en concreto (véase *Autoencoder V1*).

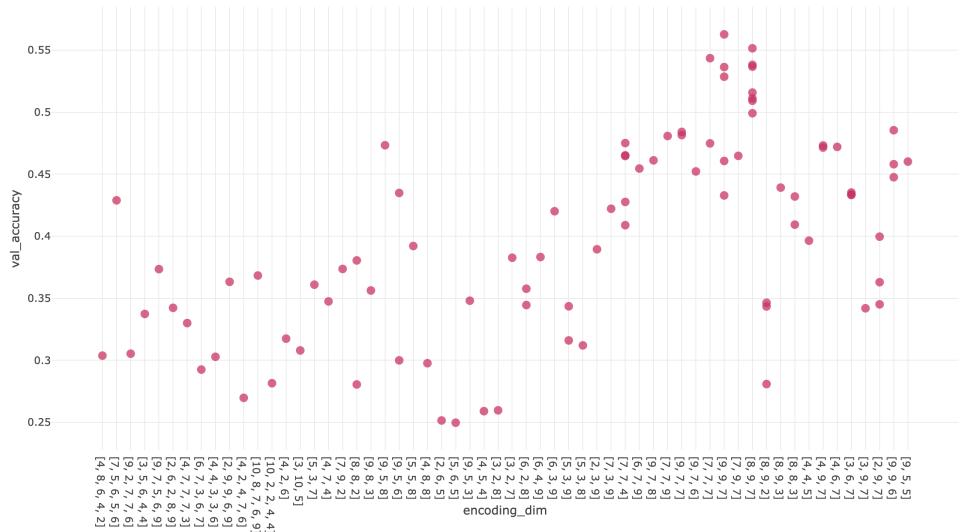


Figura 7.24: Autoencoder v2. Dimensión del código vs Validation accuracy

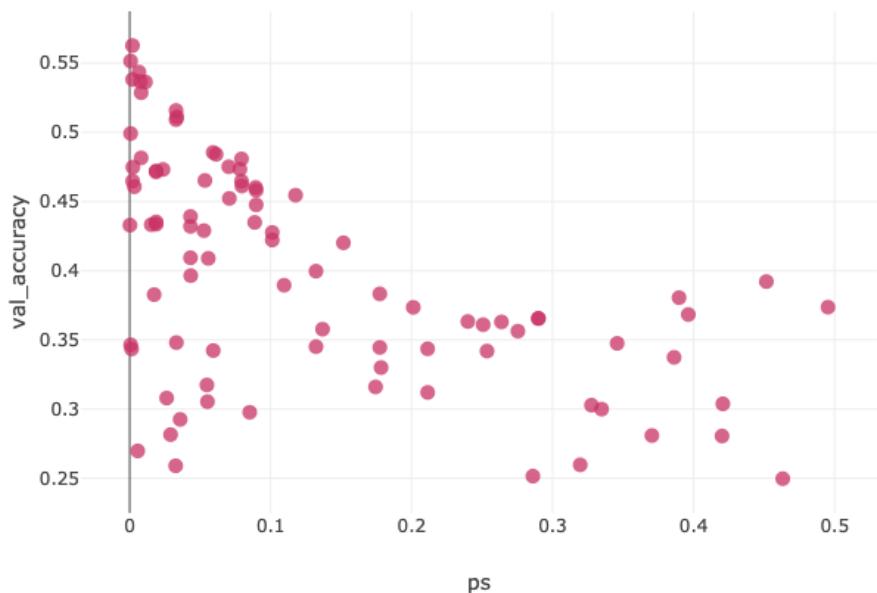


Figura 7.25: Autoencoder v2. Probabilidad de descarte (dropout) vs Validation accuracy

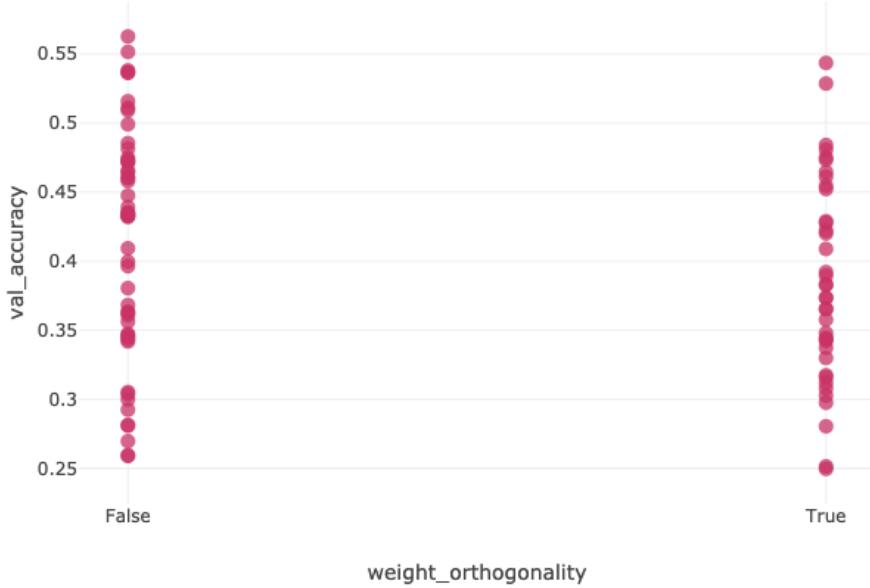


Figura 7.26: Autoencoder v2. Pesos ortogonales vs Validation accuracy

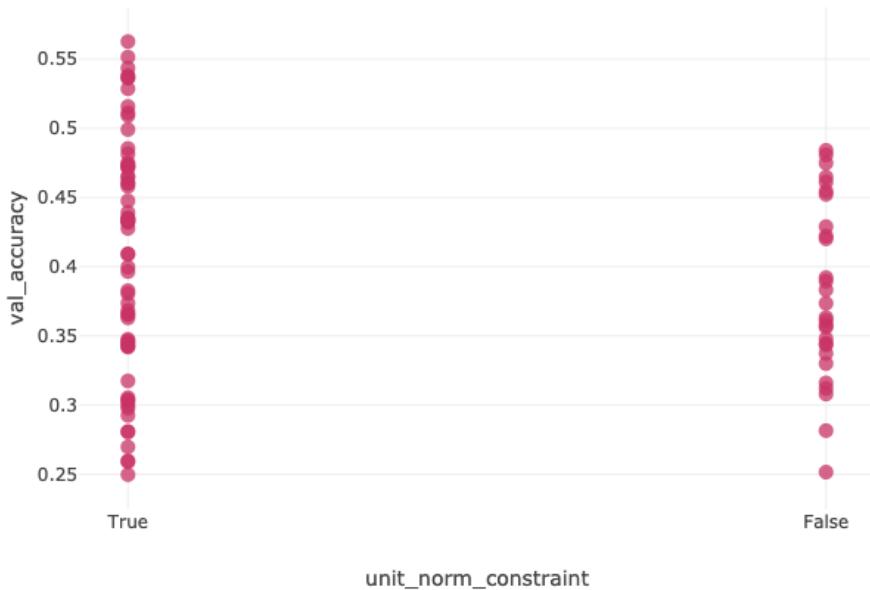


Figura 7.27: Autoencoder v2. Restricción UnitNorm vs Validation accuracy

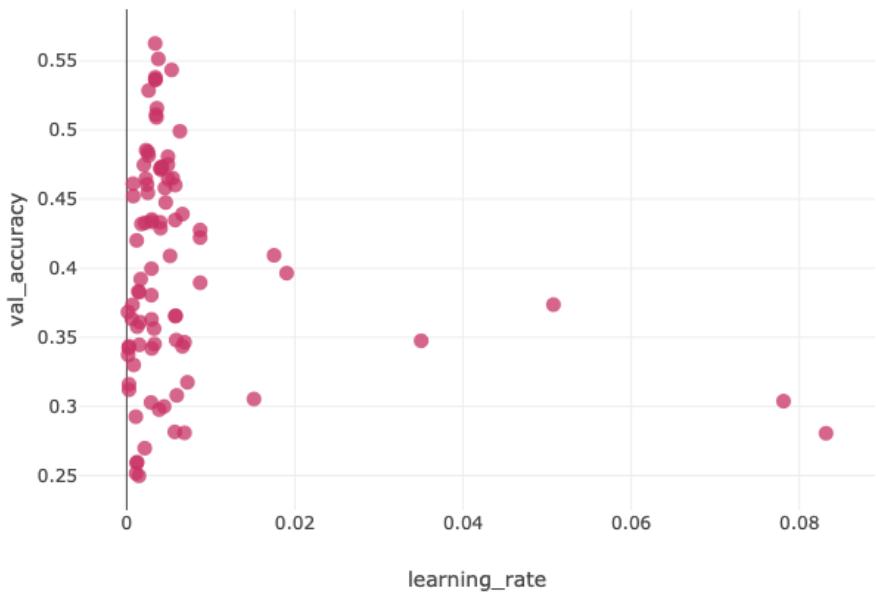


Figura 7.28: Autoencoder v2. Ratio de aprendizaje vs Validation accuracy

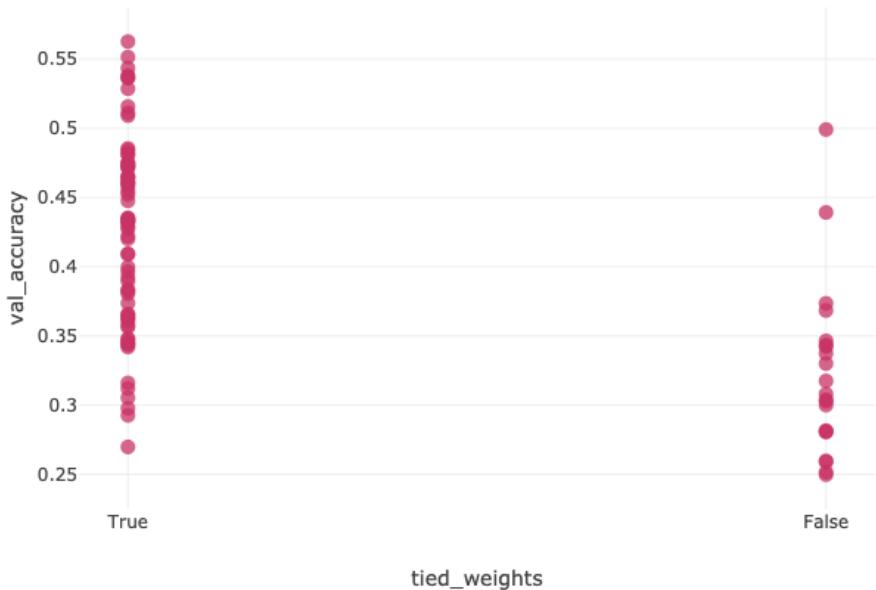


Figura 7.29: Autoencoder v2. Tied-weights vs Validation accuracy

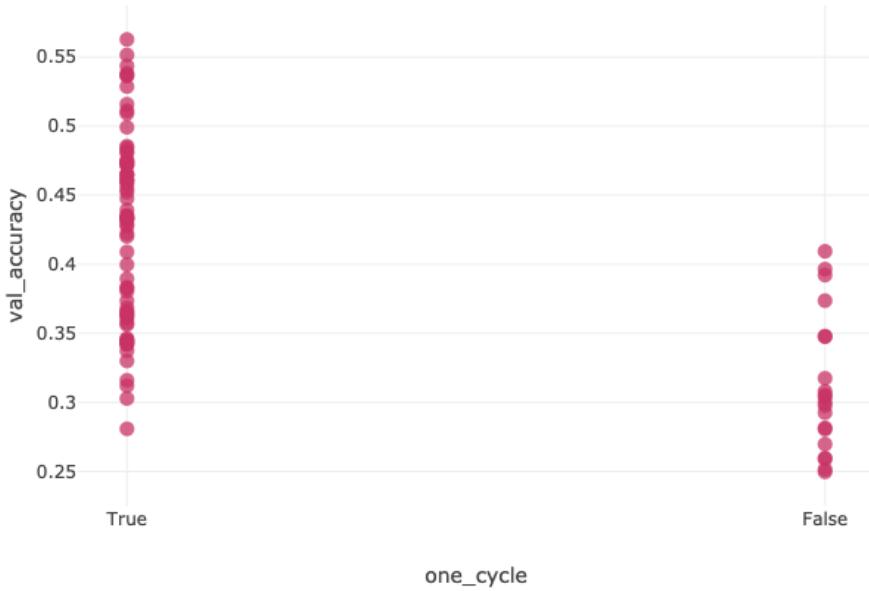


Figura 7.30: Autoencoder v2. One-cycle vs Validation accuracy

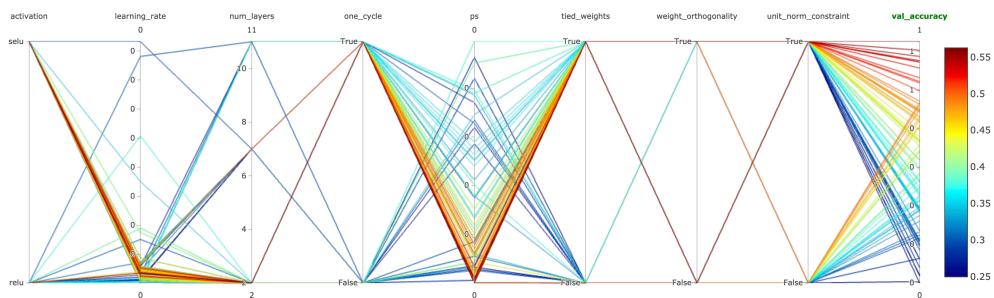


Figura 7.31: Autoencoder v2. Parameters comparison

7.4.5. AUTOENCODER VARIACIONAL

Para la arquitectura VAE, vemos que el rendimiento es muy pobre. Dentro del rango de valores de la métrica para este modelo, vemos como un ratio de aprendizaje bajo, y la aplicación de OneCycle favorecen el rendimiento (ver Figuras 7.34 7.35). Aún así, esta arquitectura solamente arroja unos resultados ligeramente mejores que un clasificador aleatorio.¹ Para el resto de

¹Un clasificador aleatorio tendría una precisión del 25 %

parámetros no podemos destacar ningún patrón en la precisión (ver Figuras 7.32 7.36 7.33).

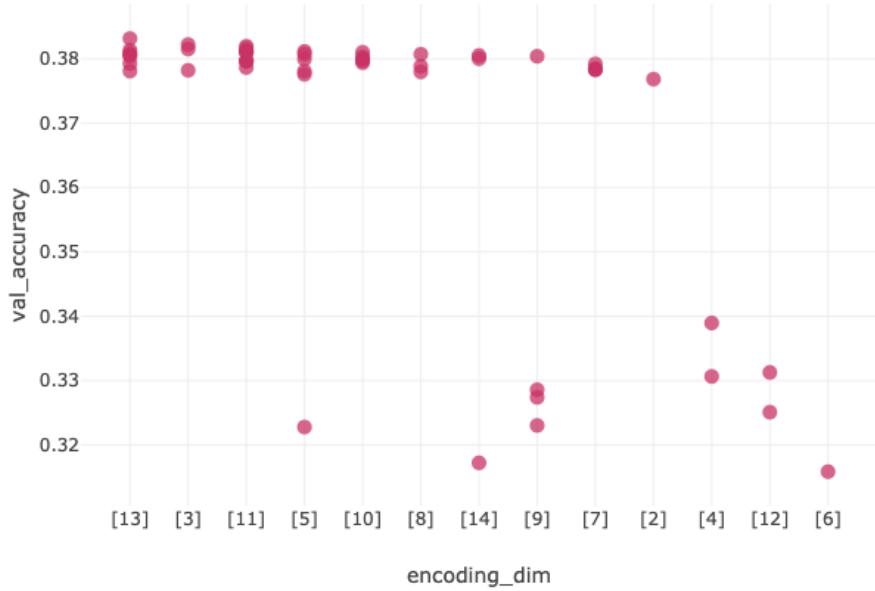


Figura 7.32: Autoencoder Variacional. Dimensión de la capa intermedia vs Validation accuracy

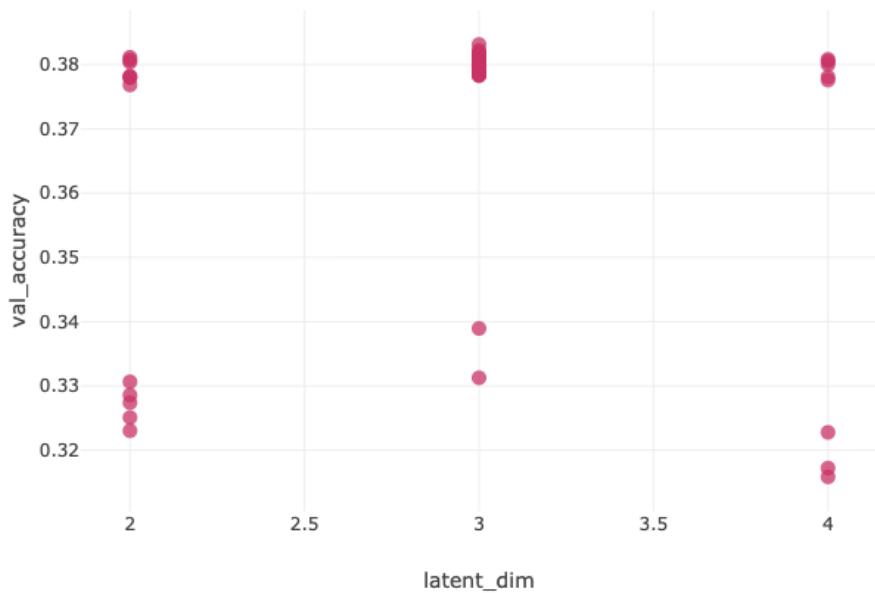


Figura 7.33: Autoencoder Variacional. Dimensión del espacio latente vs Validation accuracy

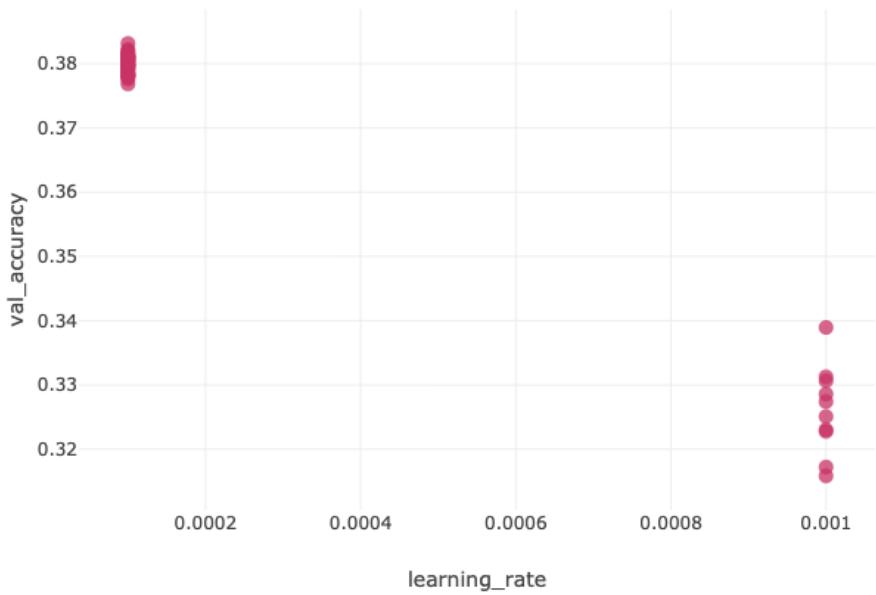


Figura 7.34: Autoencoder Variacional. Ratio de aprendizaje vs Validation accuracy

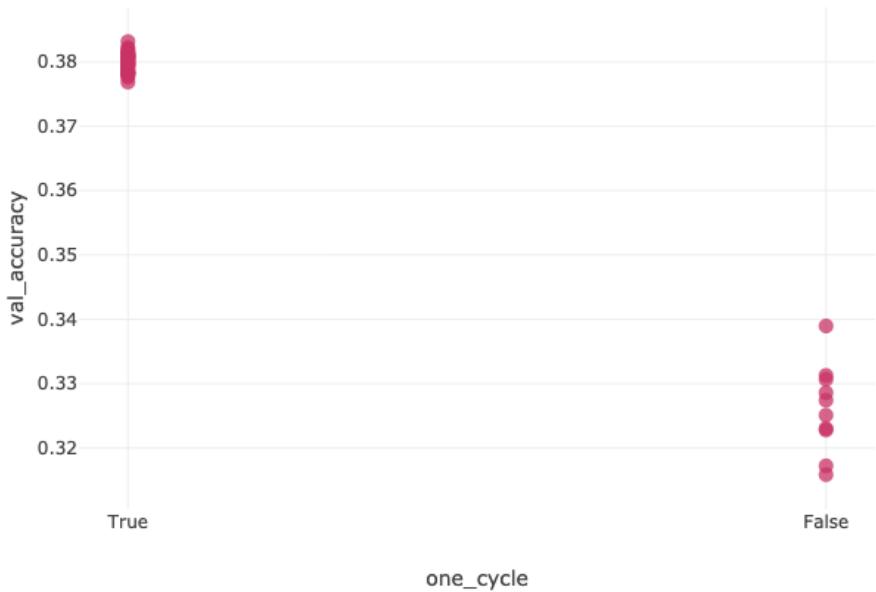


Figura 7.35: Autoencoder Variacional. One-cycle vs Validation accuracy



Figura 7.36: Autoencoder Variacional. Activation vs Validation accuracy

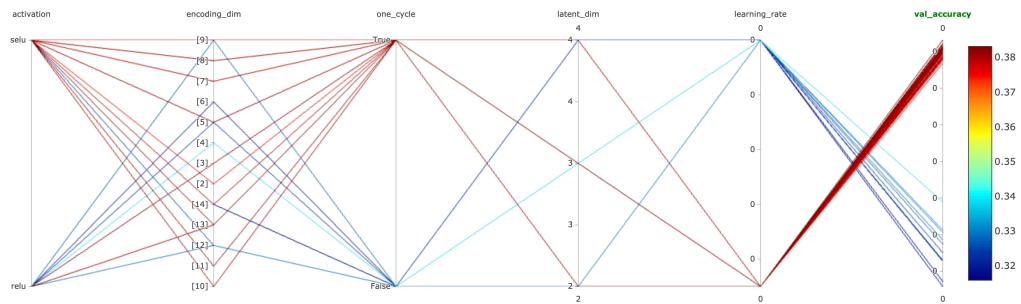


Figura 7.37: Autoencoder Variacional. Parameters comparison

Capítulo 8

**Anexo 1. APSAC:
ml-experiment: A Python
framework forreproducible
data science**

ml-experiment: A Python framework for reproducible data science

Antonio Molner Domenech

E-mail: antoniomolner@correo.ugr.es

Alberto Guillén

Computer Architecture and Technology Department, University of Granada, Spain.

E-mail: aguillen@ugr.es

Abstract. Nowadays, data science projects are usually developed in an unstructured way, which makes it difficult to reproduce. It is also hard to move from an experimental environment to production. Operational workflows such as containerization, continuous deployment, and cloud orchestration allow data science researchers to move a pipeline from a local environment to the cloud. Being aware of the difficulties of setting those workflows up, this paper presents a framework to ease experiment tracking and operationalizing machine learning by combining existent and well-supported technologies. These technologies include Docker, Mlflow, Ray, among others. The framework provides an opinionated workflow to design and execute experiments either on a local environment or the cloud. ml-experiment includes: an automatic tracking system for the most famous machine learning libraries: Tensorflow, Keras, Fastai, Xgboost and Lightgdm, first-class support for distributed training and hyperparameter optimization, and a Command Line Interface (CLI) for packaging and running projects inside containers.

1. Introduction

Reproducibility is a challenge in modern research and produces a lot of discussion [1, 2, 3]. Among all types of reproducible research, the work presented in this paper focuses on computational research [4]: building a set of tools and a specific workflow based on the principles of version control, automation, tracking, and environment isolation [5, 6]. Version control enables keeping track of files changes through time and facilitate the collaboration. Automation, from shell scripts to fully defined pipelines, allows your audience to reproduce each step of your work easily. These steps include creating files, processing data, fitting models, etc. Tracking includes a set of techniques to record the research objects such as figures, data artifacts, results, etc, systematically. Finally, environment isolation makes it easier to install all dependencies of the analytical tools with its specific versions separately from the host machine. As long as we create an isolated environment that represents a "common scenario" for researchers, computational reproducibility will not be jeopardized.

On the other hand, computational reproducibility is not an issue specific to research, all these principles we are looking forward to applying in research can be used in the industry as well

[7]. Applied data science is a field heavily focused on research, so the same ideas should apply for both fields. Our motivation to develop this framework is to both facilitate the experimental work and to fill the gap between research and deployment in the industry.

2. Structure

The framework main core is divided into four modules (depicted in Figure 1) that interact with the user through a Command-Line Interface (CLI) and a Python library. The objective of the library is to minimize the code changes required to instrument scripts to be executed by the Job Runner and to provide the abstractions to interact with the Tracking and Hyperparameter Optimization engines. On the other hand, the CLI is in charge of executing scripts either in a local environment or a remote environment.

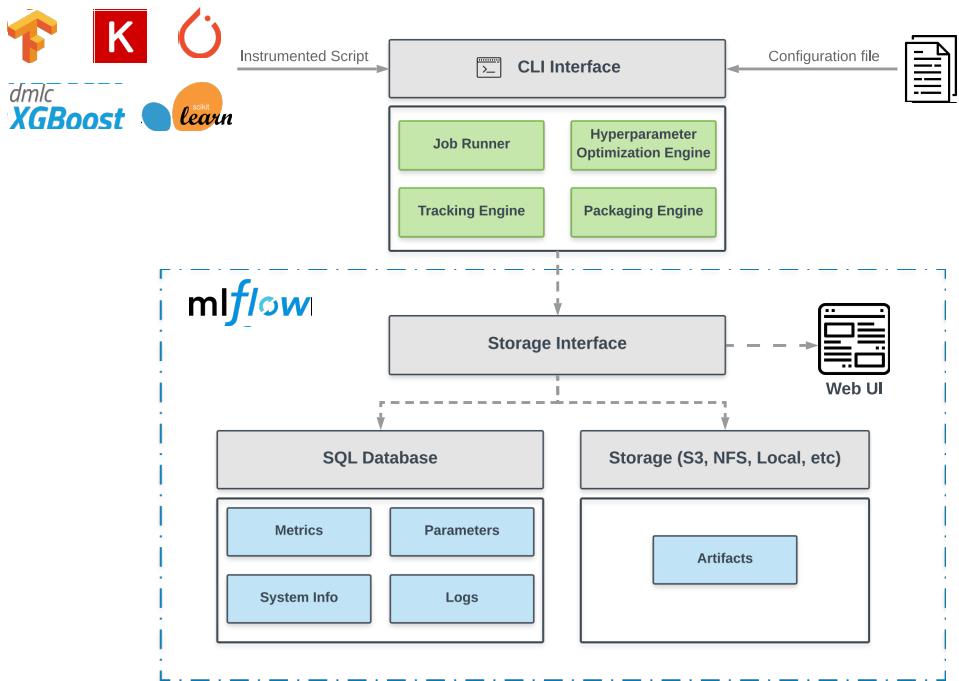


Figure 1. ml-experiment framework overview

The main idea around this tool is the concept of Job. A Job is an entity defined in a configuration file written in YAML or JSON which includes: the name of the job, the script, or list of scripts to execute, parameters, and other optional fields for containerized execution, Hyperparameter optimization, and Ray [8] cluster setup. This configuration file serves as input for the CLI which reads the file and executes the job according to the content. The only requirement for a Python script to be used in a job is to be instrumented in the following form:

```

from ml_experiment import job

@job # Add this decorator
def main(param1, param2, ...):
    # your code goes here

if __name__ == '__main__':
    main()

```

Listing 1: Example of an instrumented python script

A job can represent any computation, from a data processing task to a set of experiments executed in parallel, so to differentiate between tasks that need to use the tracking engine or the hyperparameter optimization engine, two more entities have been derived from Job: Experiment and Group. An experiment is a job that uses the tracking engine to log the parameters, metrics, and artifacts generated during its execution. While a Group is a set of parallel experiments executed with different parameters using the hyperparameter optimization engine.

Packaging Engine is the last module, which is responsible for generating a shareable version of the project using Docker [9] technology to create an image of all dependencies. This module focus on simplifying the reproduction of scientific work across multiples and heterogeneous environments by providing an abstraction layer to create *Docker* images easily.

Finally, all these different modules rely on a common knowledge center. This knowledge center is composed of an SQL database for metrics, parameters, and system logs; and artifact storage - local environment, S3 bucket, NFS, etc - where heavy files are stored.

3. Experiment tracking

Experiments and groups of experiments use the tracking engine to keep record of the information produced during the execution. By instrumenting a python script we allow the framework to automatically record the following information:

- *Parameters*. The arguments used to execute the experiment.
- *System information*: CPU information, GPU drivers, hostname, etc.
- *Version control metadata*. Git commit hash.
- *Metrics and artifacts*. If the main function returns a dictionary of metrics and a dictionary of artifact paths optionally.
- *Logs*. The console output is saved in a file and uploaded to the server.

```
name: "SVM with RBF kernel"
kind: 'experiment'

params:
    C: 0.5
    kernel: 'rbf'
    gamma: 'auto'

run:
    - modelling/train_svm.py
```

Listing 2: Experiment configuration file example

4. Hyperparameter optimization and distributed learning

ml-experiment has first-class support for hyperparameter optimization and distributed learning. The framework allows executing multiples experiments simultaneously in a local or remote environment. It also enables the configuration of the resources distribution among experiments. Thus, we can specify the number of CPUs and GPUs (if available) to use and it will distribute the tasks between the different resources. As an example, 4 reflects the definition of a group of twelve experiments, where its parameters C and gamma are sampled from some distributions. The objective of the job is to maximize the metric called val_accuracy and the experiments will be distributed across four processes, it means three experiments will be executed per CPU core.

```

from ml_experiment import job

@job
def main(C, kernel, gamma='scale'):
    np.random.seed(1234)
    X_train, X_val, y_train, y_val = load_data()
    model = SVC(C=C, gamma=gamma, kernel=kernel)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
    return {'val_accuracy': accuracy}

if __name__ == '__main__':
    main()

```

Listing 3: Experiment python script example

Another important aspect of any hyperparameter optimization engine is the efficiency of the sample space evolution. To achieve competitive results in a limited amount of time, we need a set of tools and algorithms to construct parameter spaces that maximize the objective function in the minimum amount of time. Thanks to *Optuna* [?] we have designed an optimization module that supports state-of-the-art algorithms such as *TPE*, *CMA-ES*, and traditional ones like *GridSearch* and *RandomSearch* [10].

```

name: "SVM"
kind: 'group'
num_trials: 12
sampler: TPE

param_space:
    C: loguniform(0.01, 1000)
    gamma: choice(['scale', 'auto'])

metric:
    name: val_accuracy
    direction: maximize

ray_config:
    num_cpus: 4

run:
    - modelling/train_svm.py

```

Listing 4: Group configuration file example

5. Web User Interface

Arguably the most relevant module of ml-experiment would be the Web UI (see Figure 2). All the experiments and group of experiments with its metrics, logs, parameters and artifacts are logged in the storage service described in 1. But is the web user interface what allows researchers or machine learning practitioners to examine and compare between different experiments. For this module, we opted for using Mlflow [11] as it is a well-supported and rapidly growing tool used by companies like Microsoft or Databricks.

6. Conclusions

This paper presents a simple interface to generate a container based application that allows to scale experiments and to easily deploy the models obtained in a secure and controlled

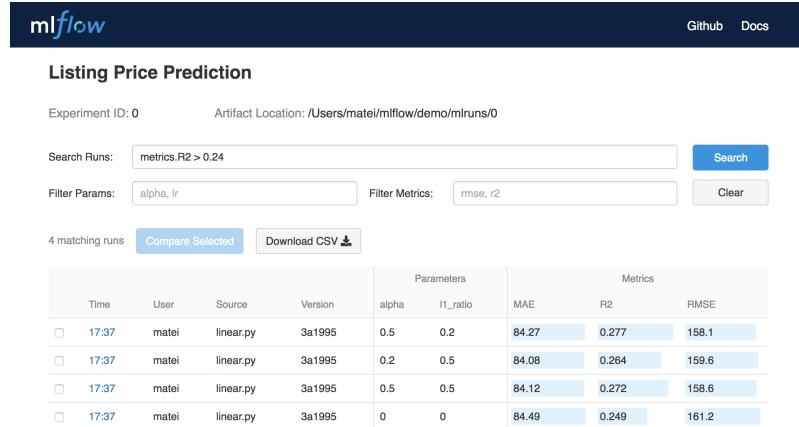


Figure 2. Mlflow Web UI screenshot

manner. Thus, reproducibility is achieved by carefully logging all the stages of the machine learning pipeline. Another important aspect is the homogeneity in the interface that will allow practitioners to share and reuse their pipelines in different scenarios without rewriting code.

Acknowledgements

This research has been possible thanks to the support of projects: FPA2017-85197-P (Spanish Ministry of Economy and Competitiveness –MINECO– and the European Regional Development Fund. –ERDF).

References

- [1] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018.
- [2] Juliana Freire, Norbert Fuhr, and Andreas Rauber. Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.
- [3] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational reproducibility: State-of-the-art, challenges, and database research opportunities. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’12, page 593–596, New York, NY, USA, 2012. Association for Computing Machinery.
- [4] Victoria Stodden, David H. Bailey, Jonathan M. Borwein, Randall J. LeVeque, William J. Rider, and William Stein. Setting the default to reproducible reproducibility in computational and experimental mathematics. 2013.
- [5] Babatunde Kazeem Olorisade, Pearl Brereton, and Peter Andras. Reproducibility in machine learning-based studies: An example of text mining. 2017.
- [6] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6):1–20, 06 2017.
- [7] Grigori Fursin, Herve Guillou, and Nicolas Essayan. Codereef: an open platform for portable mlops, reusable automation actions and reproducible benchmarking, 2020.
- [8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2017.
- [9] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, Jan 2015.
- [10] James Bergstra, R. Bardenet, Balázs Kégl, and Y. Bengio. Algorithms for hyper-parameter optimization. 12 2011.
- [11] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.

Capítulo 9

Anexo 2. Manual de Usuario

ml-experiment

Release 0.1

Antonio Molner, Alberto Guillen

Jul 03, 2020

CONTENTS:

1	Installation	1
2	Running your first experiment	3
2.1	Quickstart	3
2.2	Running jobs in a Docker container	4
2.3	Adding callbacks	4
3	Hyperparameter Tuning	7
3.1	From experiments to group of experiments	7
3.2	Sharing data across multiples processes	8
3.3	Accessing the Trial instance to model a complex parameter space	9
4	YAML/JSON Specification	11
4.1	Experiment Definition	11
4.2	Group Definition	12
4.3	Parameter space distributions	12
4.4	Models Reference	13
5	Package Reference	15
5.1	ml_experiment	15
5.2	ml_experiment.callbacks	16
5.3	ml_experiment.integrations	19
5.4	ml_experiment.config.models	23
6	CLI Reference	25

**CHAPTER
ONE**

INSTALLATION

ml-experiment supports Python 3.6 and above.

We use [Ray](#) for hyperparameter optimization, so as Ray currently supports MacOS and Linux only, Windows support will be available as soon as this framework supports it.

We recommend installing ml-experiment using Pip:

```
pip install ml-experiment
```


RUNNING YOUR FIRST EXPERIMENT

2.1 Quickstart

2.1.1 1. Instrumenting a python script

```
from ml_experiment import job

import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

SEED = 1234

@job # ADD THIS DECORATOR
def main(C = 1.0, kernel = 'rbf', degree = 3, gamma = 'scale'):
    np.random.seed(SEED)
    iris = load_iris()
    X_train, X_val, y_train, y_val = train_test_split(iris.data, iris.target, random_
    ↵state=SEED)
    model = SVC(C=C, gamma=gamma, kernel=kernel, degree=degree)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
    return {'val_accuracy': accuracy}

if __name__ == '__main__':
    main()
```

2.1.2 2. Defining a configuration file

Defining a configuration file is straightforward, we just need to create a YAML/JSON file and specify the name of the experiment, its parameters, and the file to execute.

```
name: "SVM #1"

params:
  C: 10
  kernel: poly
  gamma: auto
  degree: 3
```

(continues on next page)

(continued from previous page)

```
run:  
- examples/scripts/train_svm.py
```

2.1.3 3. Executing the experiment

Finally, once we have an instrumented Python script and a config file, we can execute the job as follows:

```
ml-experiment --config_file examples/experiments/svm.yaml
```

2.2 Running jobs in a Docker container

```
docker_config:  
  image: image_name:tag
```

```
docker_config:  
  dockerfile: path/to/dockerfile
```

2.3 Adding callbacks

Callbacks are an important aspect in almost any ML/DL framework. Callbacks allow one to hook into the process and react to some event happening.

ml-experiment offers a simple callback system based on the class `Callback`. In order to implement a custom callback, all it takes is implementing that abstract class.

Once we've implemented a custom callback class, we can add an instance of it as `callbacks` argument of `job`

The module `notifiers` contains some predefined callbacks for notifications. It

```
from ml_experiment import job
from ml_experiment.callbacks.notifiers import DesktopNotifier

import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

SEED = 1234

# ADD THIS
@job(callbacks=[DesktopNotifier()])
def main(C = 1.0, kernel = 'rbf', degree = 3, gamma = 'scale'):
    np.random.seed(SEED)
    iris = load_iris()
    X_train, X_val, y_train, y_val = train_test_split(iris.data, iris.target, random_
    ↪state=SEED)
    model = SVC(C=C, gamma=gamma, kernel=kernel, degree=degree)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
```

(continues on next page)

(continued from previous page)

```
return {'val_accuracy': accuracy}

if __name__ == '__main__':
    main()
```

HYPERPARAMETER TUNING

3.1 From experiments to group of experiments

```
name: "SVM"
kind: 'group'
num_trials: 10

param_space:
    C: loguniform(0.01, 1000)
    kernel: choice(['rbf', 'poly', 'linear'])
    gamma: choice(['scale', 'auto'])
    degree: range(2, 5)

params:
    C: 1.0

metric:
    name: val_accuracy
    direction: maximize

run:
    - examples/scripts/train_svm.py
```

3.1.1 Configuring the Ray cluster

```
resources_per_worker:
    cpu: 0.25
    gpu: 0.5

ray_config:
    num_cpus: 4
    num_gpus: 1
```

NOTE: Docker integration and Ray integration are incompatible for the moment. So, Docker is not supported for running groups of experiments.

3.1.2 Pruning unpromising trails

```
sampler: tpe
pruner: hyperband
```

3.2 Sharing data across multiples processes

When we are executing a group of experiments multiples processes are created (one for each experiment), as a consequence of this, when we load, compute, or transform some data, we need to execute that computation multiples times. To avoid that, we propose a solution inspired by Pytorch data loaders.

In our case, we create a class inherited from ml_experiment.DataLoader and define the load_data method. This method will be executed **only** by the master node all of the other processes will have a copy of the data generated by that method.

After we have created a custom DataLoader, we need to pass it as *data_loader* argument to job and after that, we can use it as it we used any other class.

The shared data will be stored in the Plasma Object Store of ray, so you should take into account its limitations: Ray Serialization

```
from ml_experiment import job, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

SEED = 1234

class MyDataLoader(DataLoader):
    @classmethod
    def load_data(cls):
        iris = load_iris()
        X_train, X_val, y_train, y_val = train_test_split(iris.data, iris.target, ↵
random_state=SEED)
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)
        return X_train, X_val, y_train, y_val

@job(data_loader=MyDataLoader)
def main(C = 1.0, kernel = 'rbf', degree = 3, gamma = 'scale'):
    X_train, X_val, y_train, y_val = MyDataLoader.load_data()
    model = SVC(C=C, gamma=gamma, kernel=kernel, degree=degree)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
    return {'val_accuracy': accuracy}
```

3.3 Accessing the Trial instance to model a complex parameter space

Sometimes, it may be necessary to access the optuna.Trial object of the current experiment so we can generate a more complex hyperparameter space. To do so, we just need to the following:

```
from ml_experiment import job, Trial

@job
def main(param1, ..., paramN):
    trial = Trial.get_current()
    ...
```

If you're running a single experiment instead of a group, Trial.get_current will raise an exception.

CHAPTER
FOUR

YAML/JSON SPECIFICATION

Configuration files are validated and parsed into Python objects. Therefore, config files have the same structure as the Python models defined on experiment_specification/Models Reference. Feel free to check them in case of doubt.

4.1 Experiment Definition

```
name: str # Required
kind: experiment # Optional. This is the default value

# Optional. Defaults to an empty dict
params:
    param1: int | str | list | dict
    param2: ...
    ...
    paramN: ...

# Optional. If not specified, the job will be run in the host environment (without Docker).
docker_config:
    image: str
    dockerfile: path/to/dockerfile
    context: path/to/context/directory
    args: dict

# Optional. If not specified, the job will be run in a local environment (without Ray).
# In any case, only one process will be spawned.
# Any other entry of this dictionary will be passed as it is to Ray.init,
# so you can fully configure the job execution.
# More information about the parameters you can use here:
# https://docs.ray.io/en/master/package-ref.html#ray.init
ray_config:
    address: localhost | master_node_address

run: # Required
- path/to/script1.py
...
- path/to/scriptN.py
```

4.2 Group Definition

```
name: str    # Required.
kind: group # Required. This line should be specified for ml-experiment CLI to know what type of job is this
sampler: str # Optional.
pruner: str # Optional.
timeout_per_trial: positive float # Optional
resources_per_worker: # Optional
  cpu: positive float # Required (only if the parent is specified)
  gpu: positive float # Optional.

# Optional. Defaults to an empty dict
param_space:
  param1: distribution(x1, x2, ..., xN)
  ...
  paramN: distribution(x1, x2, ..., xN)

# Distributions can also be used inside a list.
# The behavior is to sample a random value for each position
otherParam:
  - distribution(x1, ..., xN]
  ...
  - distribution(x1, ..., xN)

# Optional. Defaults to an empty dict
params:
  otherParam1: int | str | list | dict
  ...
  otherParamN: ...

# Optional. If not specified, the job will be run in a local Ray cluster.
# Any other entry of this dictionary will be passed as it is to Ray.init,
# so you can fully configure the job execution.
# More information about the parameters you can use here:
ray_config:
  address: localhost | master_node_address

run:
  - path/to/script1
  ...
  - path/to/scriptN
```

4.3 Parameter space distributions

Generates a random integer from min to max with an specific step.

```
range(min: int, max: int, step: int)
```

Generates a random integer from min to max (same as range with step = 1)

```
randint(min: int, max: int)
```

A uniform distribution in the log domain.

```
loguniform(min: int, max: int)
```

A uniform distribution in the linear domain.

```
uniform(min: int, max: int)
```

A categorical distribution based on the values provided. The sample parameter will be selected randomly (with uniform probability) among the provided values.

```
choice([value1: any, value2: any, ..., valueN: int])
```

4.4 Models Reference

```
class ml_experiment.config.models.ExperimentConfig(**data)
    Bases: ml_experiment.config.models.JobConfig

    docker_config = None
    kind = None
    name = None
    params = None
    ray_config = None
    run = None

class ml_experiment.config.models.GroupConfig(**data)
    Bases: ml_experiment.config.models.JobConfig

    metric: Optional[Metric] = None
    num_trials: PositiveInt = None
    param_space: Dict[str, Union[ParamDistribution, List[ParamDistribution]]] = None
    pruner: Optional[PrunerEnum] = None
    resources_per_worker: WorkerResourcesConfig = None
    sampler: Optional[SamplerEnum] = None
    timeout_per_trial: Optional[PositiveFloat] = None

class ml_experiment.config.models.RayConfig(**data)
    Bases: pydantic.main.BaseModel

    class Config:
        Bases: object
        extra = 'allow'
        address: Optional[str] = None
        classmethod convert_localhost(v)

            Parameters v(str)-

```

Parameters v(str)-

```
class ml_experiment.config.models.JobTypes
    Bases: enum.Enum

    An enumeration.
```

```
EXPERIMENT = 'experiment'
GROUP = 'group'
JOB = 'job'

class ml_experiment.config.models.Metric(**data)
    Bases: pydantic.main.BaseModel
        direction: OptimizationDirection = None
        name: str = None

class ml_experiment.config.models.DockerConfig(**data)
    Bases: pydantic.main.BaseModel
        args: Dict = None
        @classmethod check_dockerfile_and_image(values)
        context: Optional[DirectoryPath] = None
        dockerfile: Optional[FilePath] = None
        image: Optional[str] = None

class ml_experiment.config.models.WorkerResourcesConfig(**data)
    Bases: pydantic.main.BaseModel
        cpu: PositiveFloat = None
        gpu: PositiveFloat = None

class ml_experiment.config.models.PrunerEnum
    An enumeration.
        hyperband = 'hyperband'
        median = 'median'
        percentile = 'percentile'
        sha = 'sha'

class ml_experiment.config.models.SamplerEnum
    An enumeration.
        random = 'random'
        skopt = 'skopt'
        tpe = 'tpe'
```

PACKAGE REFERENCE

5.1 ml_experiment

```
@ml_experiment.job(func=None, *, callbacks=None, autologging_backends=None, optimization_metric=None, data_loader=None, log_seeds=True, log_system_info=True, delete_if_failed=False, **kwargs)
```

Experiment decorator.

This decorator must be used as a wrapper for the main function of the experiment. It handles the tracking of the following information:

- Parameters
- Metrics
- Artifacts
- System information
- Randomness (Numpy, Pytorch and TF seeds)

It also handles the job execution on clusters and the hyperparameter optimization logic.

Parameters

- **func** (*Optional[Callable]*) – Experiment main function
- **callbacks** (*Optional[Iterable[ml_experiment.callbacks.core.Callback]]*) – List of callbacks to notify when some event occur
- **autologging_backends** (*Union[List[ml_experiment.mlflow.AutologgingBackend], ml_experiment.mlflow.AutologgingBackend, None]*) – List of frameworks whose autologging functionality will be enabled. To specify a supported framework you need to use the AutologgingBackend enum.
- **optimization_metric** (*Union[ml_experiment.config.models.Metric, str, None]*) – Metric to optimize. It is mandatory when running a group job, and it will be ignored when running a single experiment. The name of the metric should be the same as one of the keys that the experiment function returns.
- **data_loader** (*Optional[ml_experiment.experiments.DataLoader]*) – Custom data loader class (not instance). It is necessary to specify this argument when using a DataLoader to share data across multiples processes.
- **log_seeds** (*bool*) – If true, it will log the seed of Numpy, Pytorch, or Python random's generator when the corresponding function to set the seed is called. Eg. when calling `numpy.random.seed(...)`
- **log_system_info** (*bool*) – Whether or not the system information, CPU, GPU, installed packages..., etc, should be logged
- **delete_if_failed** (*bool*) – If true, the experiment information will be removed in case of failure.

Returns The wrapped function

```
class ml_experiment.DataLoader
    Base class for Data loaders.
```

The main purpose of data loaders is to provide an easy way to share data across processes when running a group of experiments, also known as hyperparameter tuning.

When this abstract class is implemented (using a subclass) and that subclass is added as the argument *data_loader* to the experiment main function decorator, a shared resource will be created. This shared resource is the result of executing the implemented function, *load_data*. The key point here is that the *load_data* function will be only called once by the master process and then its result will be shared among the rest workers. In this way, we can avoid expensive computation being duplicated for each worker.

The shared data will be stored in the Plasma Object Store of ray, so you should take into account its limitations: <https://docs.ray.io/en/latest/serialization.html>

```
classmethod load_data()
```

```
class ml_experiment.Trial
```

This class makes the current Optuna Trial object accessible.

It can be used to model complex hyperparameter spaces. More information here: <https://optuna.readthedocs.io/en/latest/tutorial/configurations.html>

```
classmethod get_current()
```

Return type optuna.trial.Trial

```
class ml_experiment.AutologgingBackend
```

An enumeration.

```
FASTAI = 'fastai'
```

```
KERAS = 'keras'
```

```
LIGHTGBM = 'lightgbm'
```

```
TENSORFLOW = 'tensorflow'
```

```
XGBOOST = 'xgboost'
```

5.2 ml_experiment.callbacks

```
class ml_experiment.callbacks.Callback
```

Base class for callbacks that want to react to fired events.

To create a new type of callback, you'll need to inherit from this class, and implement one or more methods as required for your purposes. Arguably the easiest way to get started is to look at the source code for some of the pre-defined ones.

```
on_info_logged(config, metrics, artifacts, **kwargs)
```

This method will be execute every time the metrics and artifacts are logged. It will be called at least one time for every experiment.

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) – The configuration object contains all the information regarding how the experiment is executed
- **metrics** (*Dict [str, Any]*) – The metrics dictionary returned by the main function
- **artifacts** (*Dict [Dict, Any]*) – The artifacts dictionary returned by the main function

on_job_end(*config, exception*)

This method will be execute once the experiment has ended :param config: The configuration object contains all the information regarding how the experiment is executed :param exception: If not none, it means that the experiment has finished due to an error. This param contains that error.

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) –
- **exception** (*Optional[Exception]*) –

on_job_start(*config, **kwargs*)

This method will be execute once the experiment has started :param config: The configuration object contains all the information regarding how the experiment is executed

Parameters config(*ml_experiment.config.models.JobConfig*) –**on_trial_end**(*config, trial, metric, exception*)

Only for groups of experiments.

This method will be execute once a trial has been finished. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param metric: If everything when fine, it will contain the value of the optimization metric :param exception: If not none, it means that the trial has finished due to an error. This param contains that error.

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) –
- **trial** (*optuna.trial.Trial*) –
- **metric** (*Optional[float]*) –
- **exception** (*Optional[Exception]*) –

on_trial_start(*config, trial, sampled_params*)

Only for groups of experiments.

This method will be execute once a trial has been started. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param sampled_params: The randomly selected parameters

Parameters

- **config** (*ml_experiment.config.models.JobConfig*) –
- **trial** (*optuna.trial.Trial*) –
- **sampled_params** (*dict*) –

class ml_experiment.callbacks.notifiers.NotifierBase

Base class for notifiers

To create a new type of notifier, you'll need to inherit from this class, and implement one or more methods as required for your purposes. Specifically, the send_message method should be override. Arguably the easiest way to get started is to look at the source code for some of the pre-defined ones.

Parameters send_message_for_trials – If true, messages will be send for trial-related events.

on_job_end(*config, exception*)

This method will be execute once the experiment has ended :param config: The configuration object contains all the information regarding how the experiment is executed :param exception: If not none, it means that the experiment has finished due to an error. This param contains that error.

on_job_start (*config*, ***kwargs*)

This method will be execute once the experiment has started :param config: The configuration object contains all the information regarding how the experiment is executed

on_trial_end (*config*, *trial*, *metric*, *exception*)

Only for groups of experiments.

This method will be execute once a trial has been finished. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param metric: If everything when fine, it will contain the value of the optimization metric :param exception: If not none, it means that the trial has finished due to an error. This param contains that error.

on_trial_start (*config*, *trial*, *sampled_params*, ***kwargs*)

Only for groups of experiments.

This method will be execute once a trial has been started. :param config: The configuration object contains all the information regarding how the experiment is executed :param trial: The object of the class optuna.Trial that correspond to this trial :param sampled_params: The randomly selected parameters

abstract send_message (*msg*)

This method should be override by your custom logic

Parameters **msg** (*str*) – The preprocessed messaged generated from the experiment information

Return type None

class ml_experiment.callbacks.notifiers.TelegramNotifier (*token*, *chat_id*)

Callback that sends a message through a Telegram Bot.

It requires a token and chat_id which can be get following these instructions: <https://core.telegram.org/bots>

Parameters

- **token** – Telegram Bot Token.
- **chat_id** – Telegram group id. More info on how to get this id [here](#).

class ml_experiment.callbacks.notifiers/DesktopNotifier

Callback that sends a desktop notification when an event happens.

It works out-of-the-box for Linux and OS X. For Windows it is necessary to install the `win10toast` package.

class ml_experiment.callbacks.notifiers.SlackNotifier (*webhook_url*, *channel*, *username*)

Callback that sends a message to an specific Slack channel when an event happens.

It requires a webhook URL, in order to generate that URL you should follow these instructions

Parameters

- **webhook_url** – Theb webhook generated from Slack Apps
- **channel** – Slack channel name
- **username** – The username that will appear on the slack messages

class ml_experiment.callbacks.notifiers.EmailNotifier (*sender_email*, *recipient_emails*)

Callback that sends a email to a recipient or list of recipients when an event happens.

This service relies on `Yagmail`, so you'll need to setup a Gmail account and follow the library instructions.

5.3 ml_experiment.integrations

All these classes are imported from the Optuna package. For more information of how to use, please take a look at the official documentation [here](#).

```
class ml_experiment.integrations.KerasPruningCallback(trial, monitor)
    Keras callback to prune unpromising trials.
```

Example

Add a pruning callback which observes validation losses.

```
model.fit(X, y, callbacks=[KerasPruningCallback(trial, 'val_loss')])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g., `val_loss` and `val_acc`. Please refer to [keras.Callback reference](#) for further details.

```
class ml_experiment.integrations.TensorFlowPruningHook(trial, estimator, metric, run_every_steps, is_higher_better=None)
TensorFlow SessionRunHook to prune unpromising trials.
```

Example

Add a pruning SessionRunHook for a TensorFlow's Estimator.

```
pruning_hook = TensorFlowPruningHook(
    trial=trial,
    estimator=clf,
    metric="accuracy",
    is_higher_better=True,
    run_every_steps=10,
)
hooks = [pruning_hook]
tf.estimator.train_and_evaluate(
    clf,
    tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=500, hooks=hooks),
    eval_spec
)
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **estimator** – An estimator which you will use.
- **metric** – An evaluation metric for pruning, e.g., `accuracy` and `loss`.
- **run_every_steps** – An interval to watch the summary file.
- **is_higher_better** – Please do not use this argument because this class refers to `StudyDirection` to check whether the current study is minimize or maximize.

```
class ml_experiment.integrations.TFKerasPruningCallback(trial, monitor)
    tf.keras callback to prune unpromising trials.
```

This callback is intend to be compatible for TensorFlow v1 and v2, but only tested with TensorFlow v1.

Example

Add a pruning callback which observes validation losses.

```
model.fit(x, y, callbacks=[TFKerasPruningCallback(trial, 'val_loss')])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g., val_loss or val_acc.

```
class ml_experiment.integrations.XGBoostPruningCallback(trial, observation_key)
    Callback for XGBoost to prune unpromising trials.
```

Example

Add a pruning callback which observes validation errors to training of an XGBoost model.

```
pruning_callback = XGBoostPruningCallback(trial, 'validation-error')
bst = xgb.train(param, dtrain, evals=[(dtest, 'validation')],
                 callbacks=[pruning_callback])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **observation_key** – An evaluation metric for pruning, e.g., validation-error and validation-merror. Please refer to eval_metric in [XGBoost](#) reference for further details.

```
class ml_experiment.integrations.LightGBMPruningCallback(trial, metric,
                                                          valid_name='valid_0')
    Callback for LightGBM to prune unpromising trials.
```

Example

Add a pruning callback which observes validation scores to training of a LightGBM model.

```
param = {'objective': 'binary', 'metric': 'binary_error'}
pruning_callback = LightGBMPruningCallback(trial, 'binary_error')
gbm = lgb.train(param, dtrain, valid_sets=[dtest], callbacks=[pruning_callback])
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **metric** – An evaluation metric for pruning, e.g., binary_error and multi_error. Please refer to [LightGBM](#) reference for further details.

- **valid_name** – The name of the target validation. Validation names are specified by `valid_names` option of `train` method. If omitted, `valid_0` is used which is the default name of the first validation. Note that this argument will be ignored if you are calling `cv` method instead of `train` method.

```
class ml_experiment.integrations.PyTorchIgnitePruningHandler(trial, metric,  
                                         trainer)
```

PyTorch Ignite handler to prune unpromising trials.

Example

Add a pruning handler which observes validation accuracy.

```
evaluator = create_supervised_evaluator(model,  
                                         metrics={'accuracy': Accuracy()},  
                                         device=device)  
handler = PyTorchIgnitePruningHandler(trial, 'accuracy', trainer)  
evaluator.add_event_handler(Events.COMPLETED, handler)  
  
@trainer.on(Events.EPOCH_COMPLETED)  
def log_validation_results(engine):  
    evaluator.run(val_loader)
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **metric** – A name of metric for pruning, e.g., `accuracy` and `loss`.
- **trainer** – A trainer engine of PyTorch Ignite. Please refer to `ignite.engine.Engine` reference for further details.

```
class ml_experiment.integrations.PyTorchLightningPruningCallback(trial, moni-  
                                                               tor)
```

PyTorch Lightning callback to prune unpromising trials.

Example

Add a pruning callback which observes validation accuracy.

```
trainer.pytorch_lightning.Trainer(  
    early_stop_callback=PyTorchLightningPruningCallback(trial, monitor='avg_val_-  
→acc'))
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g., `val_loss` or `val_acc`. The metrics are obtained from the returned dictionaries from e.g. `pytorch_lightning.LightningModule.training_step` or `pytorch_lightning.LightningModule.validation_end` and the names thus depend on how this dictionary is formatted.

```
class ml_experiment.integrations.FastAIPruningCallback(learn, trial, monitor)
    FastAI callback to prune unpromising trials for fastai.
```

Note: This callback is for fastai<2.0, not the coming version developed in fastai/fastai_dev.

Example

Add a pruning callback which monitors validation loss directly to Learner.

```
# If registering this callback in construction
from functools import partial

learn = Learner(
    data, model,
    callback_fns=[partial(FastAIPruningCallback, trial=trial, monitor='valid_loss
    ↴')])
```

Example

Register a pruning callback to learn.fit and learn.fit_one_cycle.

```
learn.fit(n_epochs, callbacks=[FastAIPruningCallback(learn, trial, 'valid_loss')])
learn.fit_one_cycle(
    n_epochs, cyc_len, max_lr,
    callbacks=[FastAIPruningCallback(learn, trial, 'valid_loss')])
```

Parameters

- **learn** – fastai.basic_train.Learner.
- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **monitor** – An evaluation metric for pruning, e.g. valid_loss and Accuracy. Please refer to fastai.Callback reference for further details.

```
class ml_experiment.integrations.MXNetPruningCallback(trial, eval_metric)
    MXNet callback to prune unpromising trials.
```

Example

Add a pruning callback which observes validation accuracy.

```
model.fit(train_data=X, eval_data=Y,
          eval_end_callback=MXNetPruningCallback(trial, eval_metric='accuracy'))
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **eval_metric** – An evaluation metric name for pruning, e.g., cross-entropy and accuracy. If using default metrics like mxnet.metrics.Accuracy, use it's default metric name. For custom metrics, use the metric_name provided to constructor. Please refer to mxnet.metrics reference for further details.

```
class ml_experiment.integrations.ChainerPruningExtension(trial, observation_key,
                                                               pruner_trigger)
```

Chainer extension to prune unpromising trials.

Example

Add a pruning extension which observes validation losses to Chainer Trainer.

```
trainer.extend(
    ChainerPruningExtension(trial, 'validation/main/loss', (1, 'epoch')))
```

Parameters

- **trial** – A Trial corresponding to the current evaluation of the objective function.
- **observation_key** – An evaluation metric for pruning, e.g., main/loss and validation/main/accuracy. Please refer to [chainer.Reporter](#) reference for further details.
- **pruner_trigger** – A trigger to execute pruning. pruner_trigger is an instance of [IntervalTrigger](#) or [ManualScheduleTrigger](#). IntervalTrigger can be specified by a tuple of the interval length and its unit like (1, 'epoch').

5.4 ml_experiment.config.models

```
class ml_experiment.config.models.Metric(**data)

direction: OptimizationDirection = None
name: str = None
```

CHAPTER
SIX

CLI REFERENCE

ml-experiment CLI allows you to execute jobs from a configuration file. It can also be used to run a job without having to create a configuration file. And ultimately, it can be used to execute jobs combining input arguments with a configuration file, so those config files can work as a template.

Usage:

```
$ ml-experiment [OPTIONS] [SCRIPTS]...
```

Options:

- `--config_file FILE`
- `--name TEXT`: Name of the job. Overrides the config file field if specified.
- `--kind [job|experiment|group]`: Type of job. Overrides the config file field if specified.
- `--params FILE | DICT`: Job parameters. If config file is specified, these parameters In case of overlap, the values of this dictionary will take precedence over the rest
- `--param_space FILE | DICT`: Job parameter space. Only applies for groups of experiments. In case of overlap, the values of this dictionary will take precedence over the rest
- `--num_trials POSITIVE_INT`: Number of experiments to execute in parallel. Only applies for groups of experiments. Overrides the config file field if specified.
- `--timeout_per_trial POSITIVE_FLOAT`: Timeout per trial. In case of an experiment taking too long, it will be aborted.Only applies for groups of experiments. Overrides the config file field if specified.
- `--sampler [random|tpe|skopt]`: Sampler name. Only applies for groups of experiments. Overrides the config file field if specified.
- `--pruner [hyperband|sha|percentile|median]`: Pruner name. Only applies for groups of experiments. Overrides the config file field if specified.
- `--metric_key TEXT`: Name of the metric to optimize. It must be one of the keys of the metrics dictionary returned by the main function. Only applies for groups of experiments. Overrides the config file field if specified.
- `--metric_direction [minimize|maximize]`: Whether Hyperparameter Optimization Engine should minimize or maximize the given metric. Only applies for groups of experiments. Overrides the config file field if specified.
- `--docker_image TEXT`: If specified, the job will be run inside a docker contained based on the given image
- `--dockerfile FILE`
- `--docker_context DIRECTORY`: A directory to use as a docker context.Only applies when dockerfile is specified. Overrides the config file field if specified.

- `--docker_build_args FILE | DICT`: A dictionary of build arguments. Only applies when the Dockerfile is specified.
 - `--ray_config FILE | DICT`: A dictionary of arguments to pass to Ray.init. Here you can specify the cluster address, number of cpu, gpu, etc. In case of overlap, the values of this dictionary will take precedence over the rest
 - `--install-completion`: Install completion for the current shell.
 - `--show-completion`: Show completion for the current shell, to copy it or customize the installation.
 - `--help`: Show this message and exit.
- **User Guide:**
 - *Installing the package*
 - *Running your first experiment*
 - *Hyperparameter tuning*
 - *Experiment Specification*
 - *Package Reference*
 - *CLI Reference*

Referencias

- [1] B. K. Olorisade, P. Brereton, and P. Andras, “Reproducibility in Machine Learning-Based Studies: An Example of Text Mining,” 2017.
- [2] C. Boettiger, “An introduction to Docker for reproducible research, with examples from the R environment,” *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015, doi: 10.1145/2723872.2723882.
- [3] M. Zaharia *et al.*, “Accelerating the Machine Learning Lifecycle with MLflow,” *IEEE Data Eng. Bull.*, 2018.
- [4] P. Moritz *et al.*, “Ray: A Distributed Framework for Emerging AI Applications,” *arXiv:1712.05889 [cs, stat]*, Sep. 2018, Accessed: Jun. 21, 2020. [Online]. Available: <http://arxiv.org/abs/1712.05889>.
- [5] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” *arXiv:1603.04467 [cs]*, Mar. 2016, Accessed: Jun. 21, 2020. [Online]. Available: <http://arxiv.org/abs/1603.04467>.
- [6] A. Gulli and S. Pal, *Deep Learning with Keras*. Packt Publishing Ltd, 2017.
- [7] J. Howard and S. Gugger, “Fastai: A Layered API for Deep Learning,” *Information*, vol. 11, no. 2, Art. no. 2, Feb. 2020, doi: 10.3390/info11020108.
- [8] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, California, USA, Aug. 2016, pp. 785–794, doi: 10.1145/2939672.2939785.
- [9] G. Ke *et al.*, “LightGBM: A Highly Efficient Gradient Boosting Decision Tree,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 3146–3154.
- [10] C. Collberg, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, “Measuring Reproducibility in Computer Systems Research,” p. 37.

- [11] J. Freire, N. Fuhr, and A. Rauber, “Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041),” *Dagstuhl Reports*, vol. 6, no. 1, pp. 108–159, 2016, doi: 10.4230/DagRep.6.1.108.
- [12] J. Freire, P. Bonnet, and D. Shasha, “Computational reproducibility: state-of-the-art, challenges, and database research opportunities,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, Scottsdale, Arizona, USA, May 2012, pp. 593–596, doi: 10.1145/2213836.2213908.
- [13] P. Nagarajan, G. Warnell, and P. Stone, “Deterministic Implementations for Reproducibility in Deep Reinforcement Learning,” *arXiv:1809.05676 [cs]*, Jun. 2019, Accessed: Jun. 18, 2020. [Online]. Available: <http://arxiv.org/abs/1809.05676>.
- [14] “The Pierre Auger Cosmic Ray Observatory,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 798, pp. 172–213, Oct. 2015, doi: 10.1016/j.nima.2015.06.058.
- [15] M. Aglietta *et al.*, “Response of the Pierre Auger Observatory water Cherenkov detectors to muons,” Fermi National Accelerator Lab. (FNAL), Batavia, IL (United States), FERMILAB-CONF-05-282-E-TD, Jul. 2005. Accessed: Jun. 21, 2020. [Online]. Available: <https://www.osti.gov/biblio/15020240>.
- [16] B. K. Lubsandorzhiev, “On the history of photomultiplier tube invention,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 567, no. 1, pp. 236–238, Nov. 2006, doi: 10.1016/j.nima.2006.05.221.
- [17] J.-W. Shi, *Photodiodes: Communications, Bio-Sensings, Measurements and High-Energy Physics*. BoD – Books on Demand, 2011.
- [18] P. Warden, *Big Data Glossary*. O'Reilly Media, Inc., 2011.
- [19] M. Hutson, “AI Glossary: Artificial intelligence, in so many words,” *Science*, vol. 357, no. 6346, pp. 19–19, Jul. 2017, doi: 10.1126/science.357.6346.19.
- [20] F. Provost and R. Kohavi, “Glossary of terms,” *Journal of Machine Learning*, vol. 30, nos. 2–3, pp. 271–274, 1998.
- [21] “Project Jupyter.” <https://www.jupyter.org> (accessed Jun. 30, 2020).
- [22] “Amazon SageMaker,” *Amazon Web Services, Inc.* <https://aws.amazon.com/es/sagemaker/> (accessed Jun. 22, 2020).
- [23] “DataGrip: el IDE multiplataforma para bases de datos y SQL, de JetBrains,” *JetBrains*. <https://www.jetbrains.com/es-es/datagrip/> (accessed Jun. 30, 2020).
- [24] M. Baker, “1,500 scientists lift the lid on reproducibility,” *Nature News*, vol. 533, no.

7604, p. 452, May 2016, doi: 10.1038/533452a.

- [25] R. Peng, “The reproducibility crisis in science: A statistical counterattack,” *Significance*, vol. 12, no. 3, pp. 30–32, 2015, doi: 10.1111/j.1740-9713.2015.00827.x.
- [26] E. Gibney, “This AI researcher is trying to ward off a reproducibility crisis,” *Nature*, vol. 577, no. 7788, Art. no. 7788, Dec. 2019, doi: 10.1038/d41586-019-03895-5.
- [27] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal, “Good enough practices in scientific computing,” *PLOS Computational Biology*, vol. 13, no. 6, p. e1005510, Jun. 2017, doi: 10.1371/journal.pcbi.1005510.
- [28] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, “Ten Simple Rules for Reproducible Computational Research,” *PLOS Computational Biology*, vol. 9, no. 10, p. e1003285, Oct. 2013, doi: 10.1371/journal.pcbi.1003285.
- [29] “Edge.org.” <https://www.edge.org/response-detail/25340> (accessed Jun. 21, 2020).
- [30] M. L. Head, L. Holman, R. Lanfear, A. T. Kahn, and M. D. Jennions, “The Extent and Consequences of P-Hacking in Science,” *PLOS Biology*, vol. 13, no. 3, p. e1002106, Mar. 2015, doi: 10.1371/journal.pbio.1002106.
- [31] V. Stodden, D. H. Bailey, J. M. Borwein, R. J. LeVeque, W. J. Rider, and W. Stein, “Setting the Default to Reproducible Reproducibility in Computational and Experimental Mathematics,” *undefined*, 2013. /paper/Setting-the-Default-to-Reproducible-Reproducibility-Stodden-Bailey/992647adcc7e3626768841acb039d2b4a70d5c95 (accessed Jun. 21, 2020).
- [32] R. D. Peng, “Reproducible Research in Computational Science,” *Science*, vol. 334, no. 6060, pp. 1226–1227, Dec. 2011, doi: 10.1126/science.1213847.
- [33] “STANDARD DATA SCIENCE TASKS,” in *Data Science*, The MIT Press, 2018.
- [34] “MLOps: Continuous delivery and automation pipelines in machine learning,” *Google Cloud*. <https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines> (accessed Jun. 21, 2020).
- [35] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical Debt: From Metaphor to Theory and Practice,” *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov. 2012, doi: 10.1109/MS.2012.167.
- [36] D. Sculley *et al.*, “Hidden technical debt in Machine learning systems,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, Montreal, Canada, Dec. 2015, pp. 2503–2511, Accessed: Jun. 18, 2020. [Online].
- [37] D. Sculley *et al.*, “Machine Learning: The High Interest Credit Card of Technical Debt,” 2014.

- [38] Y. Bar-Hillel, “The Present Status of Automatic Translation of Languages**This article was prepared with the sponsorship of the Informations Systems Branch, Office of Naval Research, under Contract NR 049130. Reproduction as a whole or in part for the purposes of the U. S. Government is permitted.” in *Advances in Computers*, vol. 1, F. L. Alt, Ed. Elsevier, 1960, pp. 91–163.
- [39] L. Lü, M. Medo, C. H. Yeung, Y.-C. Zhang, Z.-K. Zhang, and T. Zhou, “Recommender systems,” *Physics Reports*, vol. 519, no. 1, pp. 1–49, Oct. 2012, doi: 10.1016/j.physrep.2012.02.006.
- [40] R. R. Trippi and J. K. Lee, *Artificial Intelligence in Finance and Investing: State-of-the-Art Technologies for Securities Selection and Portfolio Management*, 1st ed. USA: McGraw-Hill, Inc., 1995.
- [41] M. Kearns and Y. Nevyvaka, “Machine learning for market microstructure and high frequency trading,” *High Frequency Trading: New Realities for Traders, Markets, and Regulators*, 2013.
- [42] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, “Data Management Challenges in Production Machine Learning,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago, Illinois, USA, May 2017, pp. 1723–1726, doi: 10.1145/3035918.3054782.
- [43] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, “Data Lifecycle Challenges in Production Machine Learning: A Survey,” *SIGMOD Rec.*, vol. 47, no. 2, pp. 17–28, Dec. 2018, doi: 10.1145/3299887.3299891.
- [44] S. Schelter *et al.*, “On challenges in machine learning model management.” *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 5–15, 2018.
- [45] M. Arnold *et al.*, “Towards Automating the AI Operations Lifecycle,” *arXiv:2003.12808 [cs]*, Mar. 2020, Accessed: Jun. 18, 2020. [Online]. Available: <http://arxiv.org/abs/2003.12808>.
- [46] J. Collins, “Delivering on the Vision of MLOps,” p. 33.
- [47] R. Rampin, F. Chirigati, D. Shasha, J. Freire, and V. Steeves, “ReproZip: The Reproducibility Packer,” *Journal of Open Source Software*, vol. 1, no. 8, p. 107, Dec. 2016, doi: 10.21105/joss.00107.
- [48] *IDSIA/sacred*. IDSIA, 2020.
- [49] “Comet – Build better models faster!” <https://www.comet.ml/site/> (accessed Jun. 22, 2020).
- [50] D. Seita, “Learning to Learn,” *The Berkeley Artificial Intelligence Research Blog*. <http://bair.berkeley.edu/blog/2017/07/18/learning-to-learn/> (accessed Jun. 22, 2020).

- [51] “Polyaxon - machine learning at scale,” *Polyaxon*. <https://polyaxon.com/> (accessed Jun. 22, 2020).
- [52] “Production-Grade Container Orchestration,” *Kubernetes*. <https://kubernetes.io/> (accessed Jun. 22, 2020).
- [53] “Kubeflow,” *Kubeflow*. <https://www.kubeflow.org/> (accessed Jun. 22, 2020).
- [54] “AI Platform,” *Google Cloud*. <https://cloud.google.com/ai-platform> (accessed Jun. 22, 2020).
- [55] “Azure Machine Learning | Microsoft Azure.” <https://azure.microsoft.com/es-es/services/machine-learning/> (accessed Jun. 22, 2020).
- [56] *onnx/onnx*. Open Neural Network Exchange, 2020.
- [57] A. Shawahna, S. M. Sait, and A. El-Maleh, “FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review,” *IEEE Access*, vol. 7, pp. 7823–7859, 2019, doi: 10.1109/ACCESS.2018.2890150.
- [58] “Data science collaboration hub.” *neptune.ai*. <https://neptune.ai/> (accessed Jun. 22, 2020).
- [59] A. Ng and others, “Sparse autoencoder,” *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.
- [60] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, “Activation Functions: Comparison of trends in Practice and Research for Deep Learning,” *arXiv:1811.03378 [cs]*, Nov. 2018, Accessed: Jun. 18, 2020. [Online]. Available: <http://arxiv.org/abs/1811.03378>.
- [61] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for Fortran usage,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.
- [62] J. Demmel, “LAPACK: a portable linear algebra library for supercomputers,” in *IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, Dec. 1989, pp. 1–7, doi: 10.1109/CACSD.1989.69824.
- [63] R. Hecht-nielsen, “III.3 - Theory of the Backpropagation Neural Network**Based on ‘nonindent’ by Robert Hecht-Nielsen, which appeared in Proceedings of the International Joint Conference on Neural Networks 1, 593–611, June 1989. © 1989 IEEE.” in *Neural Networks for Perception*, H. Wechsler, Ed. Academic Press, 1992, pp. 65–93.
- [64] P. Baldi, “Autoencoders, unsupervised learning, and deep architectures,” in *Proceedings of ICML workshop on unsupervised and transfer learning*, 2012, pp. 37–49.
- [65] J. Sublime and E. Kalinicheva, “Automatic Post-Disaster Damage Mapping Using

Deep-Learning Techniques for Change Detection: Case Study of the Tohoku Tsunami,” *Remote Sensing*, vol. 11, p. 1123, May 2019, doi: 10.3390/rs11091123.

[66] Y. Wang, H. Yao, and S. Zhao, “Auto-encoder based dimensionality reduction,” *Neurocomputing*, vol. 184, pp. 232–242, Apr. 2016, doi: 10.1016/j.neucom.2015.08.104.

[67] Y. Bengio, “Regularized Autoencoders,” in *Deep Learning*, Cambridge, Massachusetts: MIT Press, 2017, p. 501.

[68] D. P. Kingma and M. Welling, “An Introduction to Variational Autoencoders,” *FNT in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019, doi: 10.1561/2200000056.

[69] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” *arXiv:1312.6114 [cs, stat]*, May 2014, Accessed: Jun. 22, 2020. [Online]. Available: <http://arxiv.org/abs/1312.6114>.

[70] P. Vincent, “A Connection Between Score Matching and Denoising Autoencoders,” *Neural Computation*, vol. 23, no. 7, pp. 1661–1674, Apr. 2011, doi: 10.1162/NECO_a_00142.

[71] Y. Ju, J. Guo, and S. Liu, “A Deep Learning Method Combined Sparse Autoencoder with SVM,” in *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, Sep. 2015, pp. 257–260, doi: 10.1109/CyberC.2015.39.

[72] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[73] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A Next-generation Hyperparameter Optimization Framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage, AK, USA, Jul. 2019, pp. 2623–2631, doi: 10.1145/3292500.3330701.

[74] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for Hyper-Parameter Optimization,” in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554.

[75] P. Moritz *et al.*, “Ray: A Distributed Framework for Emerging AI Applications,” *arXiv:1712.05889 [cs, stat]*, Sep. 2018, Accessed: Jun. 24, 2020. [Online]. Available: <http://arxiv.org/abs/1712.05889>.

[76] R. S. Sutton, A. G. Barto, and others, *Introduction to reinforcement learning*, vol. 135. MIT press Cambridge, 1998.

[77] C. Hewitt, “Actor Model of Computation: Scalable Robust Information Systems,” *arXiv:1008.1459 [cs]*, Jan. 2015, Accessed: Jun. 25, 2020. [Online]. Available: <http://arxiv.org/abs/1008.1459>.

- [78] “CUDA Toolkit,” *NVIDIA Developer*, Jul. 02, 2013. <https://developer.nvidia.com/cuda-toolkit> (accessed Jul. 02, 2020).
- [79] “NumPy.” <https://numpy.org/> (accessed Jul. 02, 2020).
- [80] A. M. Domenech, *antoniomdk/seminars_and_talks*. 2020.
- [81] T. Stanev, “Overview,” in *High Energy Cosmic Rays*, Springer Science & Business Media, 2010.
- [82] A. S. Burrows, “Baade and Zwicky: ‘Super-novae,’ neutron stars, and cosmic rays,” *PNAS*, vol. 112, no. 5, pp. 1241–1242, Feb. 2015, doi: 10.1073/pnas.1422666112.
- [83] T. Stanev, *High Energy Cosmic Rays*. Springer Science & Business Media, 2010.
- [84] D. Heck, J. Knapp, J. N. Capdevielle, G. Schatz, and T. Thouw, “CORSIKA: A Monte Carlo code to simulate extensive air showers,” Feb. 1998, Accessed: Jun. 18, 2020. [Online]. Available: <https://inspirehep.net/literature/469835>.
- [85] S. Ostapchenko, “QGSJET-II: towards reliable description of very high energy hadronic interactions,” *Nuclear Physics B - Proceedings Supplements*, vol. 151, no. 1, pp. 143–146, Jan. 2006, doi: 10.1016/j.nuclphysbps.2005.07.026.
- [86] T. Pierog, I. Karpenko, J. M. Katzy, E. Yatsenko, and K. Werner, “EPOS LHC : test of collective hadronization with LHC data,” *Phys. Rev. C*, vol. 92, no. 3, p. 34906, Sep. 2015, doi: 10.1103/PhysRevC.92.034906.
- [87] S. Argiro *et al.*, “The Offline Software Framework of the Pierre Auger Observatory,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 580, no. 3, pp. 1485–1496, Oct. 2007, doi: 10.1016/j.nima.2007.07.010.
- [88] R. Brun and F. Rademakers, “ROOT — An object oriented data analysis framework,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1, pp. 81–86, Apr. 1997, doi: 10.1016/S0168-9002(97)00048-X.
- [89] C. Cortes and V. Vapnik, “Support-vector networks,” *Mach Learn*, vol. 20, no. 3, pp. 273–297, Sep. 1995, doi: 10.1007/BF00994018.
- [90] C. K. I. Williams and M. Seeger, “Using the Nyström Method to Speed Up Kernel Machines,” in *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds. MIT Press, 2001, pp. 682–688.
- [91] L. Mason, J. Baxter, P. L. Bartlett, and M. R. Frean, “Boosting Algorithms as Gradient Descent,” in *Advances in Neural Information Processing Systems 12*, S. A. Solla, T. K. Leen, and K. Müller, Eds. MIT Press, 2000, pp. 512–518.

- [92] S. Ladjal, A. Newson, and C.-H. Pham, “A PCA-like Autoencoder,” *arXiv:1904.01277 [cs]*, Apr. 2019, Accessed: Jun. 29, 2020. [Online]. Available: <http://arxiv.org/abs/1904.01277>.
- [93] L. N. Smith, “A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay,” *arXiv:1803.09820 [cs, stat]*, Apr. 2018, Accessed: Jun. 18, 2020. [Online]. Available: <http://arxiv.org/abs/1803.09820>.