Título

Antonio Molner Domenech

Trabajo de Fin de Grado
Ingeniería Informática

Supervisado por:
Alberto Guillén

UNIVERSIDAD
DE GRANADA

Universidad de Granada, España
Junio 2020

Índice general

Listado de figuras

Listado de tablas

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.

A modo de resumen, 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. 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 común en física, la detección de primarios. El problema consiste en la detección del tipo de

primario 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, y el coste en recursos de tiempo y capital 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 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 CLI con el que empaquetas y ejecutar trabajos.

La reproducibilidad es un reto en la investigación moderna y produce bastante debate 10 11 12 13. Entre los diferentes tipos de trabajos reproducibles, este trabajo se centra en trabajos computacionales, desarrollando un flujo de trabajo específico bastado 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 GPU, o la propia versión del sistema operativo donde se ejecuten pueden incrementar la incertidumbre del experimento 14. 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 dependiente 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 primarios

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 15 fue construido. Supone un proyecto muy ambicioso y uno de los experimentos de mayor magnitud a día de hoy. Un area de 3000 kilómetros cuadrados se ha diseñado y construido para alojar detectores de agua Cherenkov (WCDs) **???**. Estos detectores son unos tanques grandes de agua ultra-pura donde se detecta la radiación de Cherenkov, normalmente utilizando fotomultiplicadores (PMTs) **??? ???**. 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* 15. 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 muons. Utilizando la señal recogida, los científicos pueden tratar de responder a varias cuestiones: que tipo de partícula llego a la atmósfera, de donde procede, y como 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 el 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 WCDs.

Capítulo 3

Fundamentos y estado del arte

A lo largo de este capitulo, 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 area de la ciencia de datos, *Machine Learning* y *MLOps*, se hace uso de una terminología concreta [16]–[18], 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 los diferentes pasos 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 ML, para análisis, o para inferencia. Aunque los conjuntos de datos pueden contener información de diferentes fuentes, el conjunto en sí tiene un solo cuerpo de trabajo.
- **Experimento** – Un proceso o actividad que permite testear una hipóstasis y validarla iterativamente. Los resultados de una cierta iteración deben ser almacenados para poder ser evaluados, comparados, y monitorizados para propósitos de auditoría.
- **Artefacto**: Pieza de información generada en un experimento. Incluye modelos entrenados, datos generados, imágenes, documentación autogenerada, 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** – A logical picture of all elements required to support a given ML model, across its development and operational pipeline.

- **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 u online que permiten acceder a los datos, modelos, etc.

- **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.

3.2. 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).

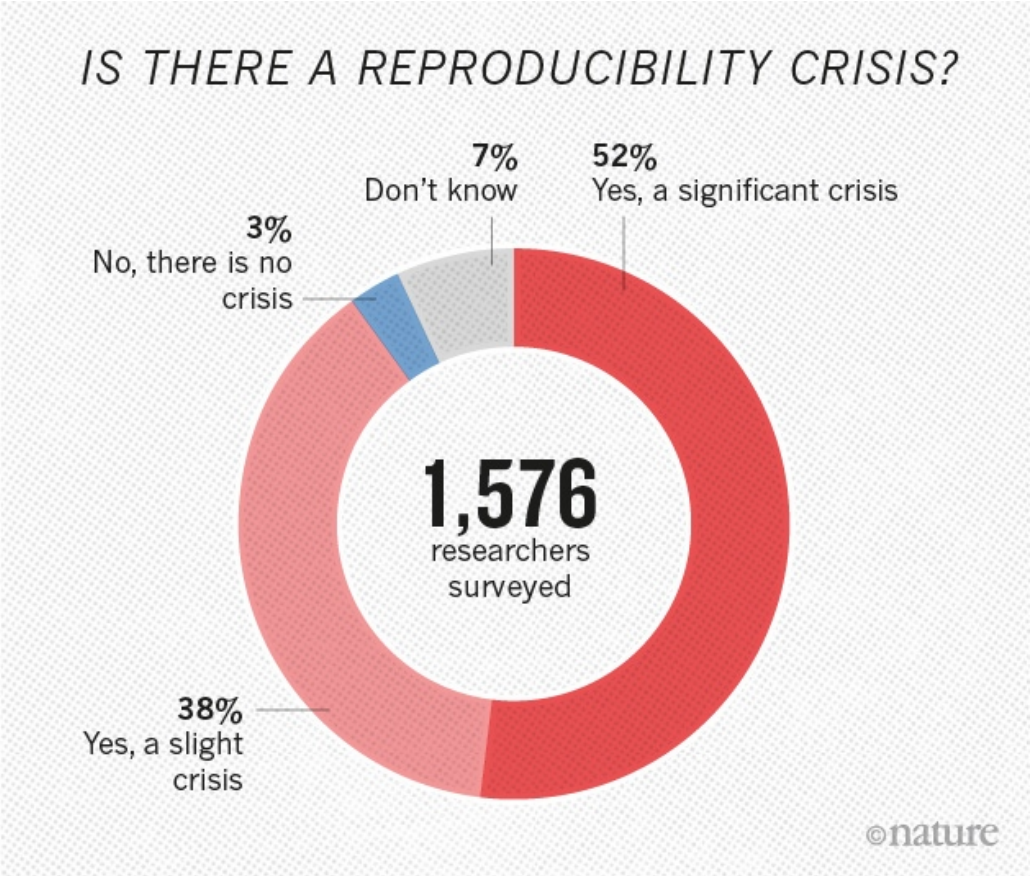


Figura 3.1: Resultados de la encuesta sobre reproducibilidad. 19

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. Está 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 area. La investigación relativa al *Machine Learning* o a otras areas 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 [19], [20] . Este fenómeno ha estudiado de manera extensiva en otros campos, pero en el area 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 tienes 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 [21]. 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 [11], [22], [23].

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 [24]. Esta separación

nos permite desarrollar una serie de buenas prácticas y herramientas específicas para cada nivel, así como ver de una manera clara que 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* [25].

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 [26]:

- **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 publico las herramientas que necesarias para replicar los resultados, por ejemplo se ofrece el 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 auditable**. 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 auditable disponible abiertamente. El código y los datos se encuentran lo suficientemente bien documentados y accesibles al publico para que la parte computacional se pueda auditar, y lo 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], [11], [22], [26], [27, p. @sandveTenSimpleRules2013].

- **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 validarlo, 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 [28], [29]:

- **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 seria 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 [30] 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, etc.

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 [31], [32]:

- **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 si 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 sen 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.
    - -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, revaluar 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, podemos 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

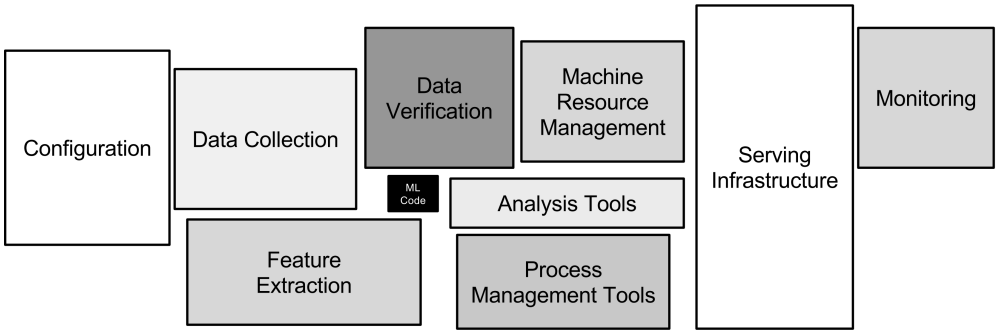


Figura 3.2: Solamente una fracción pequeña es dedicada al código de ML. El reston de código de arquitectura es necesario, y complejo. 31

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 antipatrones 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, algoritmos, etc, 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 diseñado 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 threshold o no, si uen variable esta en escala logarítmica, etc. Así como debe haber claras indicaciones de como se ha producido la información y como 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 [33], sistemas de recomendación [34], sistemas de trading de alta frecuencia [35], [36], 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 haya 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 implementado 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 [37]–[39].

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 computaciones amplios, estos sistemas deben incluir otros módulos relativos a la infraestructura: manejos de recursos, monitorización, automatización, etc.

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.

- **Entrega continua (CD)**: 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 claves 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 y *IA ética*.

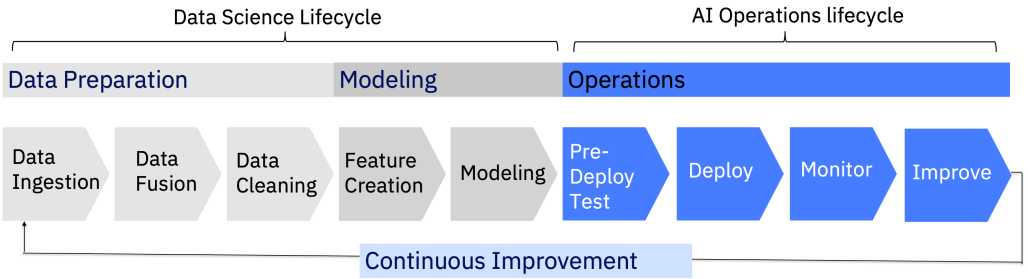


Figura 3.3: El desarrollo de sistemas de ML es complejo e implica varios pasos bien diferenciados. MLOps tiene como objetivo mejorar cada uno de los pasos, pero especial aquellos que corresponden a la etapa de Operaciones. 40

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 se entrenan los modelos y despliegan los modelos.
- **Entrenar modelos**: Desarrollar el código de entrenamiento y evaluación, y ejecutarlos en la infraestructura aprovisionada.
- **Registrar de 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 fallas, 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 iteratable. Es decir, los modelos necesitar 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 fallas del sistema.

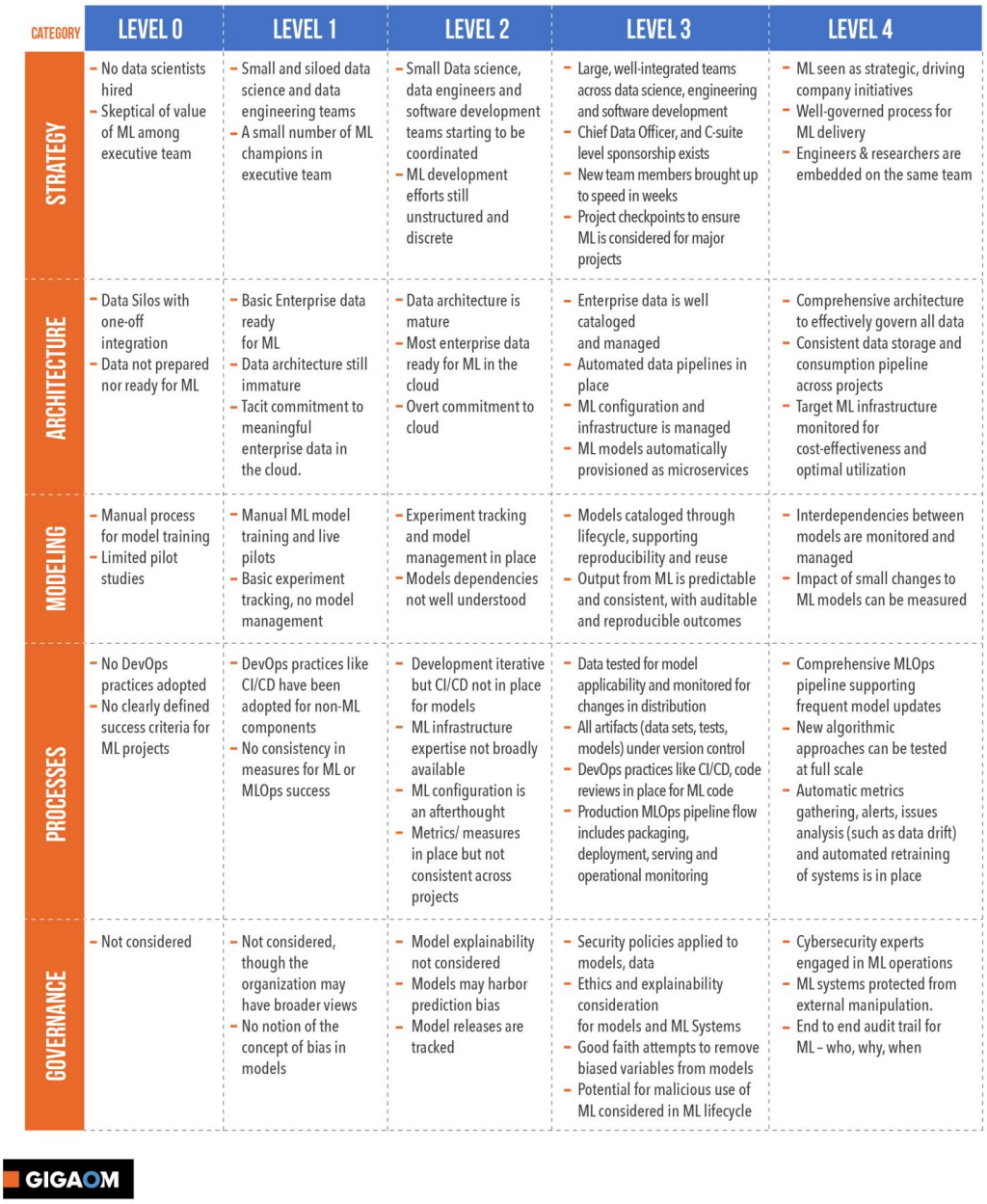
Para responder a estos desafíos de manera exitosa, los equipos de ML deben implementar las siguientes prácticas [41].

- **Reproducibilidad**: Como se ha explicado en al 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	– No data scientists hired – Skeptical of value of ML among executive team	– Small and siloed data science and data engineering teams – A small number of ML champions in executive team	– Small Data science, data engineers and software development teams starting to be coordinated – ML development efforts still unstructured and discrete	– 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	– ML seen as strategic, driving company initiatives – Well-governed process for ML delivery – Engineers & researchers are embedded on the same team
ARCHITECTURE	– Data Silos with one-off integration – Data not prepared nor ready for ML	– Basic Enterprise data ready for ML – Data architecture still immature – Tacit commitment to meaningful enterprise data in the cloud.	– Data architecture is mature – Most enterprise data ready for ML in the cloud – Overt commitment to cloud	– Enterprise data is well cataloged and managed – Automated data pipelines in place – ML configuration and infrastructure is managed – ML models automatically provisioned as microservices	– 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	– Manual process for model training – Limited pilot studies	– Manual ML model training and live pilots – Basic experiment tracking, no model management	– Experiment tracking and model management in place – Models dependencies not well understood	– Models cataloged through lifecycle, supporting reproducibility and reuse – Output from ML is predictable and consistent, with auditable and reproducible outcomes	– Interdependencies between models are monitored and managed – Impact of small changes to ML models can be measured
PROCESSES	– No DevOps practices adopted – No clearly defined success criteria for ML projects	– DevOps practices like CI/CD have been adopted for non-ML components – No consistency in measures for ML or MLOps success	– 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	– 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	– 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	– Not considered	– Not considered, though the organization may have broader views – No notion of the concept of bias in models	– Model explainability not considered – Models may harbor prediction bias – Model releases are tracked	– 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	– 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. 41

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 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.

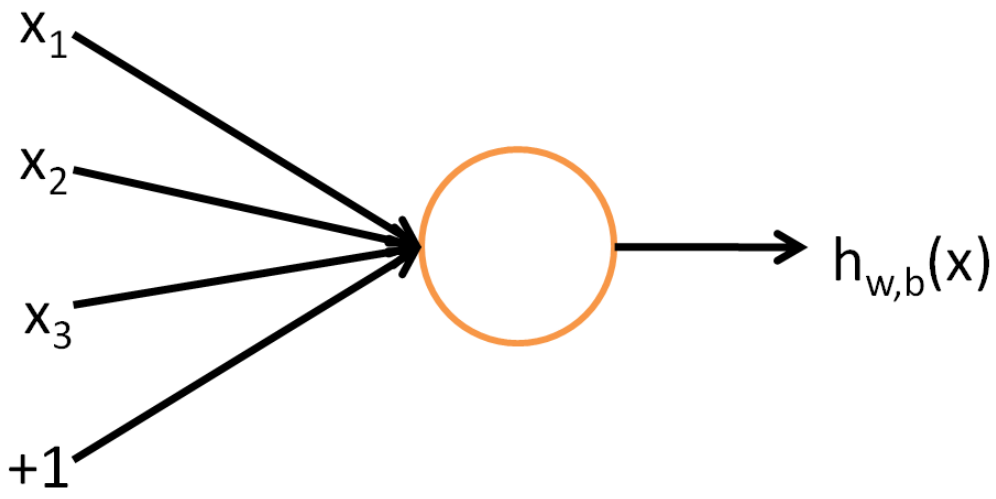
Por la razón anterior, la mayoría de software y plataformas orientadas a *MLOps* ofrecen herramientas para la gestión y control de experimentos, así como control sobre el entorno software y/o hardware. Además, las herramientas de *MLOps* están orientadas en su mayoría a la ejecución de trabajos en la 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. Redes neuronales

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 una diagrama es la siguiente:



{fig:neuron}

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 $h_{W,b}(x) = f\left(W^T x\right) = f\left(\sum_{i=1}^{3} W_i x_i + b\right)$, donde $f : \mathbb{R} \mapsto \mathbb{R}$ es la llamada **función de activación**. Entre las función de activación más comunes se encuentran: sigmoide, tanh, RELU, LeakyRELU y Swish.

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 **??**).

En la figura, los círculos representan una neurona, y aquellos con etiqueta $+1$ son las **unidades de sesgo**. 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 sesgo 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 $a_i^{(1)} = x_i$. Gracias a la notación vectorial, podemos definir el vector de activaciones de una capa como:

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

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

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

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

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

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

3.5.1. Algoritmo de propagación hacia atrás

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

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

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

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

El primer termino de $J(W, b)$ es la media de los cuadrados de los residuos (errores). El segundo termino 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 muestreados 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:

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

El parámetro α corresponde al ratio de aprendizaje. El algoritmo de propagación hacia atrás [43] 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.

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

El motivo por el que ambas ecuaciones difieren, es que la regularización no se aplica al sesgo. 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:

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

3. Para cada capa $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ 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:

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

3.6. Autoencoders

Autoencoders [44] 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 INSERTAR FIGURA.

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 la mayoría de casos. Debido a que solamente podemos reconstruir los datos de entrada de manera aproximada. Debido a dichas restricciones, el modelo es forzado a priorizar partes de información que deben ser copiadas y encontrando así patrones útiles en los datos.

Tradicionalmente, este tipo de arquitecturas se han utilizado para reducción de dimensionalidad o aprendizaje de características [45]. La reducción de la dimensionalidad es posible debido que la capa interna (*código*) contienen 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 [46]. 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.

3.6.1. Autoencoders según la dimensión del código

Según el tamaño del código existen dos categorías de autoencoders. Cuando el código tiene un tamaño menor que los datos de entrada, ase le conocen como autoencoders **undercomplete**. Si por el contrario, el código es mayor que los datos de entrada, 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:

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

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

decoder sea lineal (aquel cuya función de activación para todas sus neuronas sea $f(x) = x$), este aprende a generar un subespacio equivalente al de 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. 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.

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.

3.6.2.1. Autoencoders dispersos

Un autoencoder disperso [42], [46] es simplemente un autoencoder cuya función de coste contiene una penalización por dispersión. La nueva función de coste es la siguiente:

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

Donde $\Omega(\boldsymbol{h})$ 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:

$$\Omega(\boldsymbol{h}) = \beta \sum_{j=1}^{s_2} \mathrm{KL}\left(\rho \| \hat{\rho}_j\right)$$

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:

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

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 $\mathrm{KL}\left(\rho \| \hat{\rho}_j\right) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$ 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.

3.6.2.2. 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:

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

Donde $\tilde{\boldsymbol{x}}$ 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.

3.6.3. Autoencoders variacionales

Los autoencoders variacionales [47] 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:

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

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

3.6.3.1. Truco de la reparametrización

El termino 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**) [48].

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

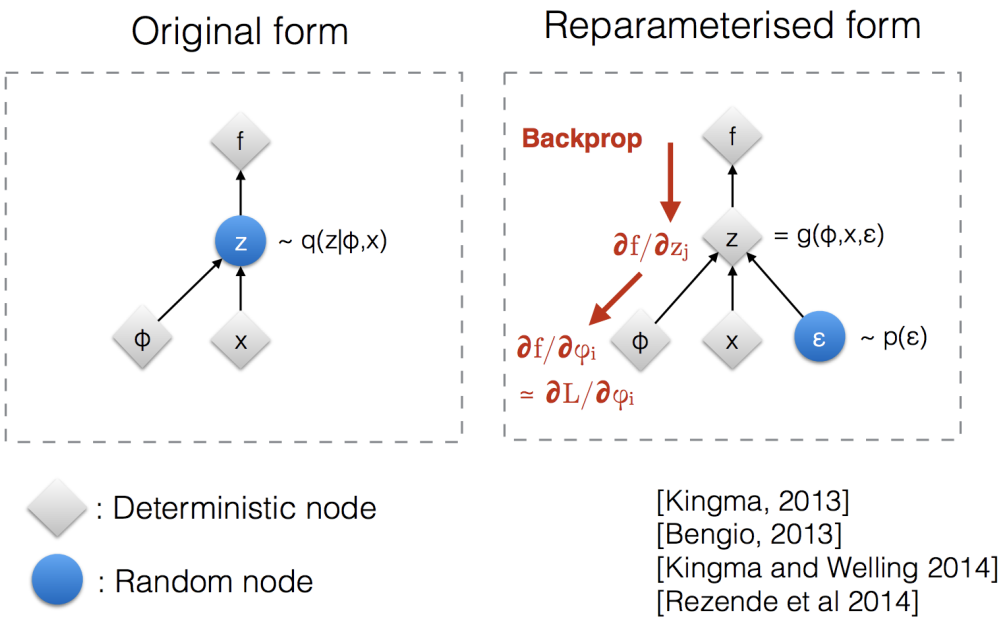


Figura 3.5: Ilustración de como el truco de la reparametrización hace el proceso de muestreo de $\mathbf{z}$ entrenable. Dispositiva 12 en el workshop de Kingma para NIPS 2015

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

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

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 $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$.

3.6.4. 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**) [46] . 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 si, 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 un 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 corresponda con la transpuesta del primero:

$$\theta_d = \theta_e^T$$

Esta técnica mejor el rendimiento en el entrenamiento, ya que se entrenan menos parámetros, pero además, sirve como método de regularización [49].

3.6.5. 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 informa-

ción, 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 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. Si se entrena un autoencoder para cada clase (con ejemplos exclusivos de esa clase), el error de reconstrucción de cada autoencoder se puede 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 entrenan 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.

3.7. 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.

3.7.1. 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* [50] 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 [51] 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*.

3.7.2. 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 módilos principales: Tracking, Projects, Models.

  - *Tracking*: La API de Tracking permite registrar experimentos, parámetros, métricas, artefactos, y otros metadatos.
  - *Projects*: El module 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 [52]k 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* [53]), 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* [54] es una herramienta enfocada también al ciclo de vida completo de un proyecto de ML. La plataforma utiliza Kubernetes [55] 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* [56] 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* [57] 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 ciencia de datos. A este conjunto de herramientas se le conoce como *Google AI Platform* [58], 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* [59]. 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 maquinas 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 [60] y desplegarlos en diferentes entornos objetivos ofertados por Azure, incluido instancias con FPGA [61].

- **Neptune**: *Neptune* [62] 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 visualizarlo, 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.

3.7.3. Análisis de rayos gammas

Capítulo 4

Planificación del trabajo

- Planificación optimista
- Planificación real

Capítulo 5

Presupuesto

- Comparitiva 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 utilizas, cabe destacar:

- Docker [63]: 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 [64]: Opt una 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 [65].

- MLFlow [3]: Ver sección de *Fundamentos*.
- Ray [66]: 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) [67] 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* [68]. 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

6.3. Tracking de experimentos

6.4. Hiperparametrización y entrenamiento distribuido

6.5. Sistema de notificaciones y callbacks

6.6. Interfaz Web

6.7. Futuro desarrollo

Capítulo 7

Experimentos

En este último capitulo 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 [69], 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 [70]. 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 observado la cascada de partículas que produce.

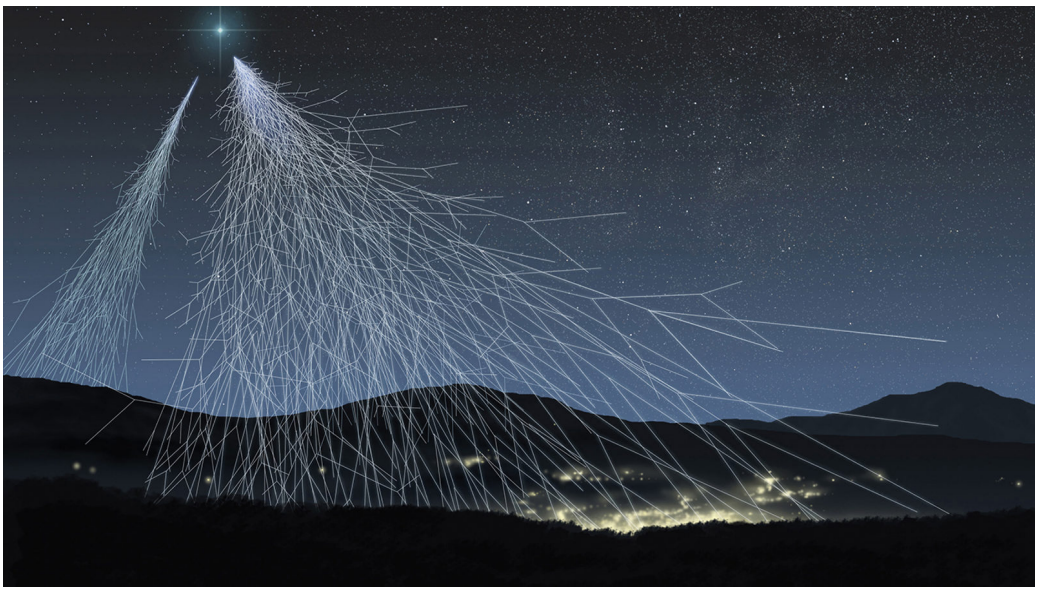


Figura 7.1: Cascada atmosférica extensa. (Observatorio Pierre Auger)

Una *cascada atmosférica extensa* [71] 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*.

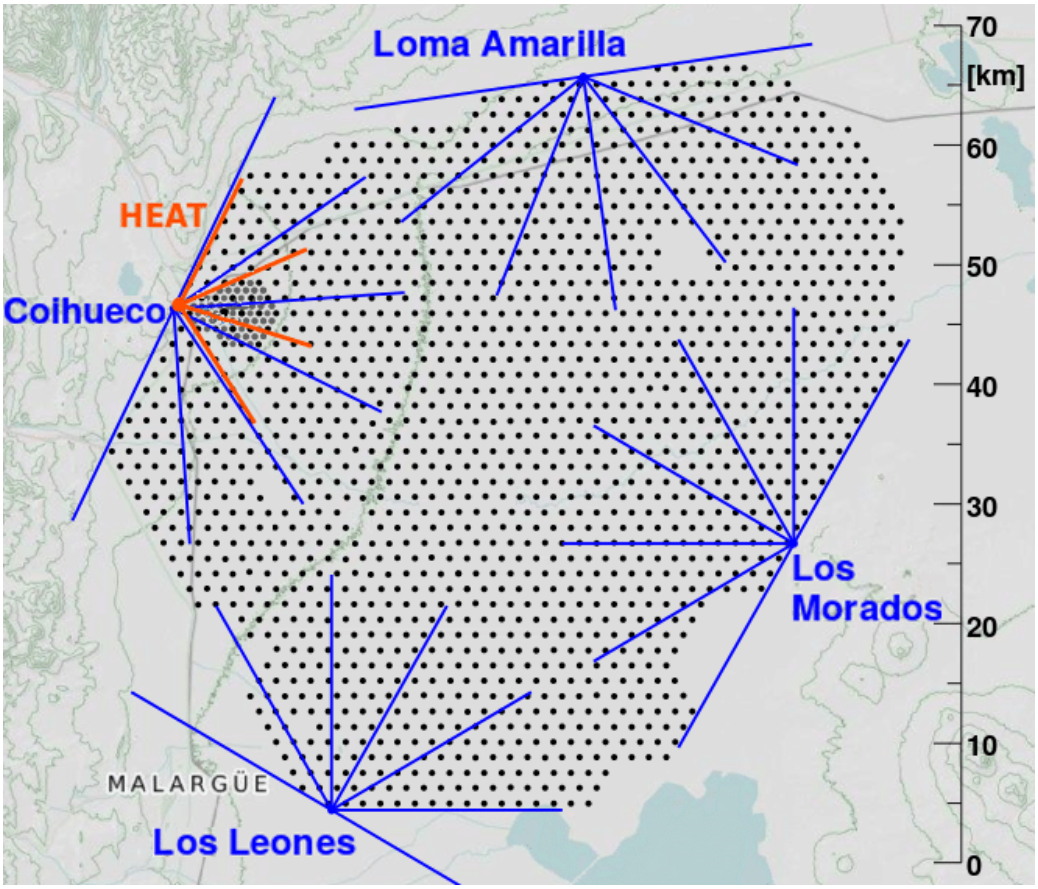


Figura 7.2: Mapa del observatorio de Pierre Auger. Cada punto negro representa un detector WCD

El Observatorio de Pierre Auger [15] 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 PMTs distribuidos simétricamente, los cuales se encargan de medir la radiación Cherenkov. La señal de estos PMTs 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 PMTs, 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 [72] 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 [73] o EPOS-LHC [74]. La señales de los WCDs producidas por las partículas se generan utilizando el software *offline* de Auger [75]. Finalmente, los datos de las simulaciones se almacenan en formato ROOT [76] 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 PMTs en un mismo detector, pueden existir relaciones entre las señales de cada fotomultiplicador. Para poder atajar este problema con ML/DL es necesario encontrar un representación del problema tal que nos permita utilizar las señales de muónicas capturadas por los PMTs para clasificar entre primarios. Una representación utilizada en trabajos previos, consiste en integrar la señal muónica, obteniendo un único valor real para cada PMT. Disponiendo así de un vector con N valores reales, tantos como PMTs haya en el WCD.

La representación que se propone en este trabajo consiste en utilizar la señal de los PMTs de manera independiente, es decir, modelar un clasificador a nivel de PMT en lugar de WCDs. De esta forma 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 PMTs 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 PMT, a nivel de WCD, o a nivel de estación. En nuestro caso lo vamos a analizar a nivel de PMT.

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, la traza

puede ser caracterizada completamente mediante un conjunto más pequeño de variables que se describen a continuación:

Variables extraídas directamente de las simulaciones

- Energía Monte Carlo $\cdot \mathrm{E} \cdot$: La energía total (en EeV, Exaelectron Voltios) del rayo cósmico primario (transformada con *log10*)
- Angulo de Zenit Monte Carlo Θ: Angulo en grados entre el zenit 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.
- Area sobre el punto máximo de la señal: Suma de todas las señales en cada traza dividida por el máximo valor en cada traza.

7.2. Procedimiento

Para el desarrollo de los diferentes experimentos se utilizado el framework 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 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. Los 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.

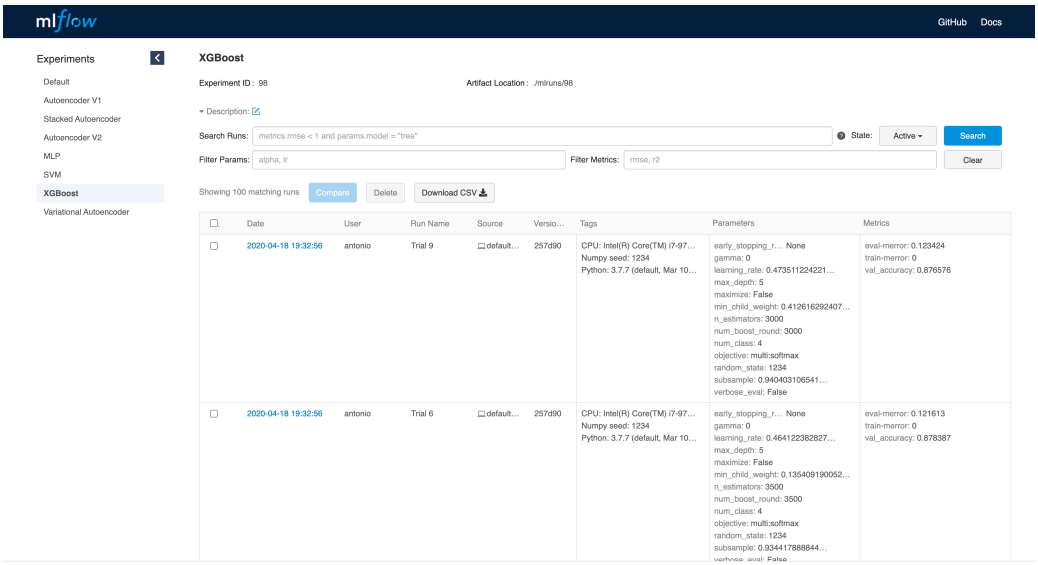


Figura 7.3: Todos los experimentos ejecutados con parámetros, métricas, artefactos, y otros metadatos, se almacenan en un servidor de MLFlow en local

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 un *DataLoader* (ver Manual). Además, las semillas para la partición, procesado y entrenamiento de modelos se establece y queda almacenada 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. Por otra parte, los parámetros, métricas y artefactos de cada experimento están almacenados en el servidor de *MLFlow* (ver figura 7.3), permitiendo visualizar y comparar entre las 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 es la siguiente:

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

7.3. Modelos considerados

7.3.1. Deep Learning

7.3.1.1. Autoencoders

7.3.2. Machine Learning tradicional

7.3.2.1. SVM

7.3.2.2. Xgboost

7.3.2.3. Autoencoder Simple

7.3.2.4. Autoencoder Apilado

7.3.2.5. Autoencoder Variacional

7.4. Resultados

Capítulo 8

Anexo: Manual de Usuario

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] "Artificial intelligence faces reproducibility crisis | Science." https://science.sciencemag.org/content/359/6377/725.summary (accessed Jun. 21, 2020).

[11] C. Collberg, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, "Measuring Reproducibility in Computer Systems Research," p. 37.

[12] 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.

[13] 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.

[14] 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.

[15] "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.

[16] P. Warden, *Big Data Glossary*. O'Reilly Media, Inc., 2011.

[17] 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.

[18] F. Provost and R. Kohavi, "Glossary of terms," *Journal of Machine Learning*, vol. 30, nos. 2-3, pp. 271–274, 1998.

[19] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature News*, vol. 533, no. 7604, p. 452, May 2016, doi: 10.1038/533452a.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] "Edge.org." https://www.edge.org/response-detail/25340 (accessed Jun. 21, 2020).

[25] 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.

[26] 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).

[27] R. D. Peng, "Reproducible Research in Computational Science," *Science*, vol. 334, no. 6060, pp. 1226–1227, Dec. 2011, doi: 10.1126/science.1213847.

[28] "STANDARD DATA SCIENCE TASKS," in *Data Science*, The MIT Press, 2018.

[29] "MLOps: Continuous delivery and automation pipelines in machine learning," *Google Cloud*. https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelin (accessed Jun. 21, 2020).

[30] 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.

[31] 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].

[32] D. Sculley *et al.*, "Machine Learning: The High Interest Credit Card of Technical Debt," 2014.

[33] 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.

[34] 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.

[35] 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.

[36] M. Kearns and Y. Nevmyvaka, "Machine learning for market microstructure and high frequency trading," *High Frequency Trading: New Realities for Traders, Markets, and Regulators*, 2013.

[37] 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.

[38] 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.

[39] S. Schelter *et al.*, "On challenges in machine learning model management." *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 5–15, 2018.

[40] 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.

[41] J. Collins, "Delivering on the Vision of MLOps," p. 33.

[42] A. Ng and others, "Sparse autoencoder," *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[43] 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.

[44] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proceedings of ICML workshop on unsupervised and transfer learning*, 2012, pp. 37–49.

[45] 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.

[46] Y. Bengio, *Deep Learning*. Cambridge, Massachusetts: MIT Press, 2017.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] *IDSIA/sacred*. IDSIA, 2020.

[52] "Comet – Build better models faster!" https://www.comet.ml/site/ (accessed Jun. 22, 2020).

[53] 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).

[54] "Polyaxon - machine learning at scale," *Polyaxon*. https://polyaxon.com/ (accessed Jun. 22, 2020).

[55] "Production-Grade Container Orchestration," *Kubernetes*. https://kubernetes.io/ (accessed Jun. 22, 2020).

[56] "Kubeflow," *Kubeflow*. https://www.kubeflow.org/ (accessed Jun. 22, 2020).

[57] "Amazon SageMaker," *Amazon Web Services, Inc.* https://aws.amazon.com/es/sagemaker/ (accessed Jun. 22, 2020).

[58] "AI Platform," *Google Cloud*. https://cloud.google.com/ai-platform (accessed Jun. 22, 2020).

[59] "Azure Machine Learning | Microsoft Azure." https://azure.microsoft.com/es-es/services/machine-learning/ (accessed Jun. 22, 2020).

[60] *onnx/onnx*. Open Neural Network Exchange, 2020.

[61] 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.

[62] "Data science collaboration hub." *neptune.ai*. https://neptune.ai/ (accessed Jun. 22, 2020).

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

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] T. Stanev, "Overview," in *High Energy Cosmic Rays*, Springer Science & Business Media, 2010.

[70] 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.

[71] T. Stanev, *High Energy Cosmic Rays*. Springer Science & Business Media, 2010.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.