Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Orquestación de contenedores en aplicaciones con componente IA

Autor: D. Álvaro Barchín Rubio

Tutor: Dr. D. Antonio Moratilla Ocaña

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Orquestación de contenedores en aplicaciones con componente IA

Autor: D. Álvaro Barchín Rubio

Tutor: Dr. D. Antonio Moratilla Ocaña

Tribunal:

Presidente:

Vocal 1º:

Vocal 2º:

Fecha de depósito: 1 de enero de 2021

A nuestros alumnos pasados, presentes y futuros...

"Empieza haciendo lo necesario, luego haz lo posible y de pronto empezarás a hacer lo imposible."
Francisco de Asís

Resumen

La inteligencia artificial se encuentra presente actualmente en múltiples ámbitos de nuestra vida cotidiana. El trabajo en esta área no deja de incrementarse y, a su vez, la búsqueda de nuevas herramientas que faciliten la vida de los profesionales de este sector. El uso de contenedores permite escalar una infraestructura de forma sencilla. Para ello, es necesario contar con una herramienta de orquestación con un interfaz simple e intuitiva que se encargue de administrar estos contendedores agrupados en clústeres. En este trabajo se explorarán herramientas ya existentes para utilizar contenedores en flujos de IA, recopilando diferentes tipos de algoritmos y realizando experimentos con varios de ellos. También se estudiarán conceptos de DevOps que puedan ser aplicados en este caso. El resultado final será un marco de trabajo para la instalación del entorno y la creación de flujos de trabajo de IA en una plataforma basada en contenedores.

Palabras clave: Orquestación, contenedor, inteligencia artifical, machine learning, Docker, Kubernetes, Kubeflow, DevOps

Abstract

Artificial intelligence is now around us in multiple aspects of our everyday life. The research done in this field keeps growing at an unprecedented rate and, to the same extent, the search of new tools that help professionals in their jobs. The use of containers brings simple autoscaling to the table. For this reason, it is essential to rely on an orquestration service that provides an intuitive and simple interface for managing container clusters. We aim to explore already available tools for creating AI pipelines on a container infraestructure in order to carry out experiments with different types of algorithms. In this context, some DevOps pratices will be explored and applied to these pipelines. The resulting product of this process will become a framework for the environment configuration and AI pipeline creation in a container based platform.

Keywords: Orchestration, container, artificial intelligence, machine learning, Docker, Kubernetes, Kubeflow, DevOps

Índice general

Índice de figuras

Índice de algoritmos

Capítulo 1

Introducción

1.1 Presentación

A lo largo de los últimos años, se ha incrementado el uso de contenedores en el ámbito del desarrollo de software y su adopción no para de crecer [24]. Este dato no sorprende dado que el uso de contenedores proporciona mejoras en seguridad, resiliencia de las aplicaciones y, principalmente, en escalabilidad. Una aplicación basada en microservicios y respaldada por contenedores es la clave para proporcionar un servicio escalable. Este modelo de arquitectura ya es usado hoy en día por empresas grandes con millones de usuarios como Google, Spotify o Airbnb [25].

Los contenedores son una tecnología de virtualización ligera que pueden ser usados para ejecutar desde un microservicio a una aplicación convencional. Un contenedor empaqueta todos los ejecutables, el código binario, las bibliotecas y los archivos de configuración necesarios. En comparación con modelos de virtualización tradicionales, los contenedores son más ligeros, portables y requieren una menor cantidad de recursos para su ejecución. También es posible ejecutar múltiples contenedores agrupados en uno o varios clústeres, gestionados por un orquestador de contenedores como Kubernetes.

La orquestación de contenedores ha supuesto un cambio de paradigma desde el lanzamiento de Kubernetes en 2014 por parte de Google [26]. Google fue la primera empresa en comprender que la gestión de contenedores a gran escala puede llegar a ser un reto desafiante. Por este motivo, destinaron una gran cantidad de recursos a este proyecto (internamente conocido como *Borg* en un principio) mientras Docker seguía utilizándose para la virtualización de contenedores de forma local. Kubernetes fue diseñado acorde a las nuevas necesidades de arquitectura de microservicios de Google. Docker también lanzó Docker Swarm como alternativa pero su enfoque era diferente y, conforme las empresas empezaron a adoptar Kubernetes, se hizo más palpable esa diferencia.

La inteligencia artificial es una disciplina en alza actualmente, especialmente en aplicaciones como el machine learning. En este campo de estudio es importante disponer de entornos de trabajo estables y evitar preocuparse de la arquitectura donde se ejecuta el modelo desarrollado. Partiendo de esta premisa y, con el aumento de popularidad de Kubernetes, surgió Kubeflow. Esta herramienta permite aprovechar las características de los contenedores como la escalabilidad y el rendimiento, abstrayendo completamente al científico de datos de la arquitectura subyacente. Además, Kubeflow proporciona diversos componentes que facilitan la creación de un flujo de IA. Este proceso de creación de pipelines será tratado en este proyecto.

1.2 Objetivos

El campo de la IA es muy extenso y no existe una clasificación clara de las diversas categorías debido a que la línea que separa los distintos tipos es difusa. Se pretende realizar una recopilación de diferentes familias de algoritmos representativos, estudiar las características de cada uno de ellos y escoger varios ejemplos para su posterior ejecución en un entorno de contenedores. El entorno elegido para esta ejecución se trata de Kubeflow.

El uso de Kubeflow permite abstraernos de la propia orquestación de contenedores y centrar la atención en las necesidades de cada tipo de algoritmo. Se realizarán varios casos prácticos para ejemplificar las necesidades de diferentes algoritmos de IA, partiendo de un caso simple y culminando es un ejemplo más complejo. En este caso final se aplicaran a su vez conceptos de CI/CD para modelar un ejemplo lo más parecido a la realidad. El conjunto de todo el desarrollo conformará un marco de trabajo para la creación de pipelines en Kubeflow, desde la instalación del entorno hasta optimizaciones para casos complejos.

Capítulo 2

Estado del arte

Antes de abordar la parte práctica del proyecto, es necesario partir sobre una base teórica acerca de la inteligencia artificial. Se comenzará definiendo de qué trata la inteligencia. Dado que este campo es complejo y cuenta con numerosos matices, se realizará una exposición de algunos de los grupos más interesantes de algoritmos siguiendo un punto de vista más práctico, agrupando los distintos algoritmos en base a sus características. Se intentará encontrar, por lo tanto, un equilibrio entre exhaustividad y exposición clara, dado que para el desarrollo de este proyecto tampoco es necesario contar con una comprensión perfecta de los conceptos teóricos subyacentes. Es más interesante para nuestro caso de estudio analizar las características de cada tipo de algoritmo y plantear qué diferencias y similitudes existen entre ellos de cara a la implementación de su despliegue mediante contenedores.

Una vez se ha realizado el trabajo de estudio, se examinarán las diferentes soluciones que existen hoy en día para el despliegue de modelos de inteligencia artificial. Se realizará una exposición de los diferentes enfoques, analizando cuáles son las características intrínsecas de cada planteamiento. Finalmente, se profundizará en el despliegue mediante contenedores, poniendo en contexto esta solución con implementaciones más tradicionales

2.1 Inteligencia Artificial

La inteligencia artificial es una disciplina que ha ganado popularidad en los últimos años, pero ¿cómo es posible concretar este concepto? Una posible definición podría ser la propuesta por J. McCarthy: *"La inteligencia artificial es el uso de la ciencia y la ingeniería con el objetivo de crear máquinas inteligentes, especialmente programas de ordenador inteligentes. Está relacionada, a su vez, con el uso de ordenadores para comprender la inteligencia humana, pero la IA no debe limitarse a métodos que son biológicamente observables"* [27].

Esta perspectiva se distancia de definiciones más clásicas como las ofrecidas por Russell y Norvig (2010, Capítulo 1) [28], donde se definen cuatro subgrupos que contienen distintos enfoques. Una primera aproximación describe la inteligencia artificial como la aproximación de una máquina a las características de un ser humano. Dentro de estas características, son dos las que se desea conseguir: actuar como un humano y pensar como un humano. Por lo tanto, se puede observar que, aunque estas dos categorías contengan un componente común, se enfocan en distintas partes de ese mismo concepto. Actuar como un humano engloba la realización de acciones típicas de un ser humano para las que se presupone que es necesaria inteligencia para su realización. Un ejemplo son acciones que requieran coordinación como practicar un deporte o pintar. Sin embargo, pensar como un humano, se asocia a actividades como la

toma de decisiones, la resolución de problemas complejos o el aprendizaje. También aparecen conceptos más cercanos al ámbito de la filosofía como la consciencia y la personalidad, pero estos conceptos se alejan del ámbito de este proyecto.

La otra aproximación que se ha definido trata el concepto de la racionalidad. El fin de este planteamiento es dejar a un lado la imitación de comportamientos humanos y conseguir que una máquina pueda tanto actuar como pensar de forma racional. Es necesario especificar, por lo tanto, qué significa ser racional. Entendemos como agente racional aquel que es capaz de tomar la mejor acción posible y pensar de forma lógica. Un agente que es capaz de pensar de forma racional es, por lo tanto, capaz de obtener conclusiones adecuadas valiéndose únicamente de la lógica, siendo este campo de estudio una de las fundaciones principales de las que se parte para poder obtener un agente racional. Dadas las premisas adecuadas, un agente racional es capaz de obtener una conclusión correcta, si esta existe, pero el principal problema radica en la dificultad de trasladar el conocimiento impreciso del mundo a una representación precisa. Actuar de forma racional se basa en conceptos similares para realizar inferencias y tomar la mejor acción posible basándose en la información sobre el mundo disponible. No obstante, obtener una racionalidad perfecta es imposible en escenarios complejos debido al alto coste computacional.

Tras comprender un poco mejor el concepto de inteligencia artificial, para el desarrollo de este proyecto se va a optar por un enfoque más práctico dado que el objetivo final es agrupar los distintos algoritmos en base a sus características de implementación. Se ha decidido, por lo tanto, analizar los tipos de algoritmos existentes que implementan una solución a un problema, dejando a un lado las definiciones de los problemas para los que estos algoritmos han sido planteados. La base matemática sobre la que se apoyan las definiciones de estos problemas queda fuera del alcance de este proyecto. Existen múltiples clasificaciones de algoritmos de IA dependiendo de los criterios elegidos y de las preferencias del autor. En este proyecto, no se realizará otra clasificación más. En su lugar, se realizará una exposición para contar con una muestra representativa de los diferentes campos. Los grupos que serán analizados son los siguientes:

- Búsquedas
- Algoritmos bioinspirados
- Algoritmos evolutivos
- Machine learning
- Deep learning

A lo largo de los siguientes apartados se analizarán las características de cada grupo y se seleccionarán algunos ejemplos de cada categoría para ser explicados en más detalle. Estos ejemplos se rescatarán más adelante para ser implementados mediante el uso de contenedores. Una diferencia fácilmente observable entre los grupos seleccionados se trata de la necesidad de algunos algoritmos de un proceso de entrenamiento antes de poder ser usados para realizar predicciones, mientras que otros algoritmos tan solo requieren los datos del problema a resolver y esperar hasta que se encuentre la solución. Otras diferencias que se pueden apreciar son, tanto la cantidad de datos que necesitan ser suministrados en cada caso como la posibilidad de aplicar paralelización para acelerar los cálculos del algoritmo. Todo esto se analizará más en detalle en los siguientes apartados.

2.1.1 Búsquedas

Las búsquedas son una de las áreas más importantes y antiguas de la inteligencia artificial. Este tipo de problemas está estrechamente relacionado con agentes racionales o agentes creados para la resolución de

problemas que son usados para encontrar la solución de un problema propuesto [28]. Para su correcto funcionamiento, es necesario poder representar escenarios del mundo real como un espacio de estados con acciones que sirven como transiciones entre los distintos estados. En este espacio de estados uno o varios de estos estados son iniciales o de partida y el objetivo es alcanzar mediante transformaciones o cambios de estado uno o varios estados meta. Estos estados meta se identifican mediante el uso de un test o diversos criterios. Es necesario también utilizar operadores de cambio de estado o funciones sucesor para poder calcular a qué nuevos estados se puede llegar desde el estado actual y el coste de estos en algunos tipos de problemas. La solución obtenida tras la resolución de una búsqueda se trata, por lo tanto, de una secuencia de acciones necesarias para obtener el estado final. En casos en los que se utilizan representaciones más complejas y estructuradas se conoce a estos agentes como agentes de planificación. Para comparar la eficacia de diferentes métodos de búsqueda se suelen valorar las siguientes características:

- **Completitud:** dicho de un método que siempre es capaz de encontrar la solución si esta existe.
- **Optimalidad:** cuando se encuentra una solución, esta es la más cercana al punto inicial, ya sea por contar con el menor número de pasos o con el menor coste acumulado cuando este no es constante.
- **Complejidad espacial o en memoria:** estimación del tamaño de las listas que deben mantenerse cargadas en memoria, en función del factor de ramificación y la profundidad de las soluciones encontradas.
- **Complejidad temporal o en operaciones:** estimación de la cantidad de estados o nodos que serán generados y explorados. Se expresa también en función del factor de ramificación y la profundidad.

Existen dos tipos de búsquedas dependiendo de si es posible contar con un criterio para agilizar el proceso, las búsquedas informadas y las no informadas.

Las búsquedas no informadas son una clase de algoritmos de búsqueda de propósito general. Este tipo de algoritmos son métodos de fuerza bruta que operan explorando sistemáticamente todas las posibles opciones del problema. Este enfoque es necesario dado que en este caso, no se dispone de información adicional sobre el problema que pueda acelerar la resolución. La búsquedas no informadas son, por lo tanto, el tipo más básico de búsquedas. Algunos ejemplos de algoritmos de búsqueda no informada son la búsqueda en anchura, profundidad, profundidad limitada, profundidad iterativa o coste uniforme. En la Fig. 2.1 aparece una comparación de la eficiencia de estos algoritmos utilizando los criterios mencionados anteriormente. En esta comparación aparecen las siguientes variables:

- p $\rightarrow$ profundidad
- $\pi \rightarrow$ ramificación o nº de nodos sucesores
- l $\rightarrow$ límite establecido para la búsqueda contenidos...

Por otra parte, las búsquedas informadas cuentan con información sobre el espacio de estados del problema. Este conjunto de conocimiento puede incluir aspectos como una estimación de la distancia a la que se encuentra el estado final, coste de cada ruta o cómo llegar al nodo final. Esto es posible gracias al uso de las heurísticas. Una heurística es una función usada en las búsquedas informadas para seleccionar el camino más prometedor. Utiliza como entrada la posición actual de la búsqueda y devuelve una estimación de la distancia a la que se encuentra el objetivo. Esta función puede que no devuelva siempre la mejor solución pero nos asegura encontrar una buena solución en una espacio de tiempo

Método	Gestión abiertos	Completo	Óptimo	C. Espacial	C. Temporal
Anchura	Cola	Si	Si (*1)	$O(\pi^{p+1})$	$O(\pi^{p+1})$
Profundidad	Pila	No (*2)	No	$O(\pi \cdot p)$	$O(\pi^{p+1})$
Prof. limit.	Pila	No	No	$O(\pi \cdot l)$	$O(\pi^{l+1})$
Prof. iterat.	Pila	Si	Si	$O(\pi \cdot p)$	$O(\pi^{p+2})$
Coste uniforme	Menor gasto	Si	Si		$O(\pi^{p+1})$

Figura 2.1: Eficiencia de diferentes algoritmos de búsqueda [1]

razonable. La elección de una buena función heurística determinará, por lo tanto, el rendimiento del algoritmo de búsqueda. Para que una heurística sea admisible, es decir, que el coste resultante no sea mayor que el real es necesario que se cumpla la siguiente inecuación:

$$h(n) \leq h^*(n)$$

Donde $h(n)$ es la heurística escogida y $h^*(n)$ sería la heurística ideal que devolvería siempre el coste óptimo en cada nodo explorado, obteniendo el mejor camino posible. El valor devuelto por la heurística admisible debe ser, por tanto, igual o menor que el real. De lo contrario se explorarían caminos que no nos acercan a la solución y no se obtendría ninguna ventaja con respecto a una búsqueda no informada. Algunos ejemplos de búsqueda informada son el algoritmo hill climbing o búsqueda en escalada, la búsqueda primero el mejor o el algoritmo A^*.

Para conseguir obtener una buena función heurística es necesario contar con información detallada de ese caso en concreto y analizar en profundidad el modelo del problema. Por este motivo los métodos heurísticos están estrechamente relacionados con el problema que resuelven. Los algoritmos denominados como metaheurísticas se concibieron para intentar sortear esta desventaja.

El término metaheurística se obtiene añadiendo el prefijo **"meta"**, que significa **"más allá"** a la palabra heurística. Las metaheurísticas son, por lo tanto, estrategias para diseñar algoritmos heurísticos más generales y que pueden ser aplicados a más tipos de problemas a diferencia de las heurísticas tradicionales [29]. El objetivo, por tanto, es aprovechar las ventajas de los métodos heurísticos en un ámbito más general para resolver un mayor número de problemas.

Las metaheurísticas, al ser estrategias para diseñar algoritmos heurísticos más generales pueden aplicarse a diferentes tipos de procedimientos como métodos de relajación, procesos constructivos, búsquedas por entornos y procedimientos evolutivos. En este apartado el tipo más apropiado para tratar son las metaheurísticas de búsqueda. Dentro de este subconjunto, hay un algoritmo que puede resultar interesante, la búsqueda tabú. Este algoritmo, utiliza una memoria para guardar información sobre el recorrido realizado y evitar concentrar la búsqueda en la misma zona del espacio de estados. La búsqueda tabú se analizará en más detalle a continuación. Además, se rescatará más adelante como ejemplo de búsqueda en la parte práctica del proyecto.

2.1.1.1 Búsqueda tabú

La búsqueda tabú es un procedimiento metaheurístico de búsqueda local utilizado para resolver problemas de optimización. En este tipo de algoritmo, se parte de una solución al problema que va mejorándose progresivamente. El método termina cuando no es posible encontrar una solución mejor que la anterior. Para ello, este algoritmo utiliza la memoria para almacenar una lista de los pasos anteriores [2]. De esta forma, se evita caer en soluciones ya exploradas, penalizando los pasos que llevarían al mismo punto

y evitando óptimos locales. Es esta característica la que da sentido al nombre del algoritmo, ya que la palabra **tabú**[1] indica la prohibición de realizar alguna acción.

En este algoritmo se utilizan dos memorias, la memoria a corto y a largo plazo. La memoria a corto plazo sirve como lista tabú para almacenar las soluciones recientes y evitar repetir soluciones. Por otra parte, la memoria a largo plazo permite admitir algunas soluciones incluso si estas se encuentran en la lista tabú siguiendo ciertos criterios. Este proceso es conocido como aspiración. Para crear la lista tabú es necesario considerar algunos aspectos como:

- Qué elementos guardar en la lista, ya sea soluciones completas o algunos atributos.
- Tamaño de la lista.
- En el caso de memoria de atributos, qué atributos deben ser seleccionados.
- Establecer el criterio de aspiración.

Algoritmo de la BÚSQUEDA TABÚ SIMPLE

Generar solución inicial $\mathbf{x_0}$
k := 1.
x= x_0. (x es la solución actual)

MIENTRAS la condición de finalización no se encuentre
HACER:
Identificar *N*(x). (Vecindario de x)
Identificar *T*(x,k). (Lista Tabú)
Identificar A(s,k). (Conjunto de Aspirantes)
Determinar $N^*(x,k) = \{N(x) - T(x,k)\} \cup A(x,k)$. (Vecindario reducido)
Escoger la mejor $x \in N^*(x,k)$
"Guardar" x si mejora la mejor solución conocida $x_k := x$.
Actualizar la lista tabú
k := k+1.
FIN MIENTRAS

Figura 2.2: Pseudocódigo de la búsqueda tabú [2]

En la Fig. 2.2 se muestra cómo sería la aplicación de este algoritmo. La búsqueda tabú opera de forma idéntica al resto de algoritmos de búsqueda. Partiendo de una solución, se definen las soluciones adyacentes que pueden ser alcanzadas desde la solución actual, se evalúan y se escoge la mejor opción disponible. No obstante, en la búsqueda tabú no se considera todo el espacio de soluciones adyacentes ya que la lista tabú limita qué opciones pueden ser escogidas. La solución inicial desde la que se parte puede ser elegida mediante algún tipo de heurística o puede ser aleatoria. Tras esto se identifican las soluciones que pueden ser alcanzadas desde la solución actual. Estas soluciones son filtradas antes de elegir la mejor opción disponible. El conjunto de soluciones final está compuesto por las soluciones disponibles, descartando aquellas que se encuentren en la lista tabú y añadiendo las soluciones que cumplan los criterios de aspiración elegidos. Finalmente, se elige la mejor opción y se realiza una nueva iteración del bucle. Este proceso continúa hasta que se alcanza la condición de parada que ha sido escogida para el problema. La eficiencia de este algoritmo dependerá de varias decisiones de diseño como:

[1] Siguiendo la definición del diccionario de la RAE en https://dle.rae.es/tabú

- **Lista tabú** - lista donde se almacenan las soluciones anteriores que no podrán ser elegidas durante varias iteraciones del algoritmo. Es necesario elegir aspectos como el tamaño de la lista y el número de iteraciones en las que las soluciones seguirán estando bloqueadas.
- **Conjunto de aspirantes** - Conjunto de soluciones que pueden ser elegidas aunque se encuentren en la lista tabú si se cumplen ciertas condiciones. La elección de qué criterio utilizar para determinar qué soluciones se encuentran en este grupo tendrá un gran impacto en el rendimiento del algoritmo. Para elegir este criterio es necesario, por lo tanto, contar con información experta sobre el problema a resolver
- **Criterio de parada** - Para evitar un bucle infinito es necesario seleccionar una condición para la finalización del algoritmo. La opción más simple es establecer un número máximo de iteraciones. Otra opción es utilizar el conocimiento del problema para establecer algún tipo de condición que debe cumplirse. Un ejemplo de esta condición puede ser un umbral de optimalidad a partir del cual se acepta la solución actual.

2.1.2 Algoritmos bioinspirados

Existe una gran cantidad de problemas de optimización que no pueden ser resueltos de forma sencilla utilizando técnicas tradicionales. Estos problemas pueden incluso llegar a ser inviables debido al tiempo necesario para su resolución. En estos casos, un enfoque diferente puede simplificar el proceso de resolución. Una opción es la computación bioinspirada.

Los algoritmos bioinspirados toman como referencia procesos naturales o sociales para diseñar métodos heurísticos que se comportan como analogías de los procesos originales. Se utilizan modelos de fenómenos existentes en la naturaleza, aplicando de esta forma una metáfora biológica para la resolución de problemas. El resultado son algoritmos no determinísticos y adaptativos (es decir, que utilizan el entorno como realimentación para realizar cambios en el modelo y sus parámetros) que frecuentemente presentan de forma implícita un sistema multiagente (con estructura paralela) donde múltiples agentes inteligentes interactúan entre ellos [5]. Algunos ejemplos de algoritmos bioinspirados son los siguientes:

- Algoritmos evolutivos
- Algoritmos inmunológicos
- Algoritmos de enjambres
- Redes neuronales

Los **algoritmos evolutivos** (o **computación evolutiva**), al igual que el resto de los algoritmos bioinspirados, utilizan un modelo natural como inspiración. En este caso se toma como referencia el proceso evolutivo de los seres vivos. Dependiendo de los métodos usados para la generación de soluciones, siendo este el equivalente a las diferentes generaciones de individuos en la naturaleza, existen varios tipos de algoritmos evolutivos. Se profundizará en esta categoría en el siguiente apartado.

Los **algoritmos inmunológicos** son una familia de sistemas computacionales inspirados por la respuesta inmune de los vertebrados. Estos algoritmos suelen ser modelados tomando como base las características de aprendizaje y memoria del sistema inmunológico para su eso en la resolución de problemas. Un ejemplo de aplicación de este tipo de algoritmos es la optimización del diseño de una red de distribución de agua para conseguir el coste mínimo [30].

Los **algoritmos de enjambres**, también conocidos como **Inteligencia de enjambre**, suelen ser usados para resolver problemas de optimización. El objetivo de este tipo de algoritmos () es modelar sistemas descentralizados compuestos por agentes artificiales simples con capacidad de auto organización y un comportamiento inteligente colectivo [31]. Estos sistemas están compuestos por agentes simples que interactúan de forma local entre ellos y con su entorno. La inspiración de este campo proviene de sistemas biológicos. Estos agentes siguen reglas muy sencillas pero, pese a que no existe una estructura de control centralizada que dirija al enjambre, las interacciones entre todos los agentes provocan la aparición de un sistema inteligente emergente, que es desconocido para los agentes individuales. Algunos ejemplos de inteligencia de enjambre en la naturaleza, sin carácter exhaustivo, son las colonias de hormigas, las colonias de abejas, los bancos de peces, el alineamiento de las aves en vuelo, los rebaños de animales e, incluso, el crecimiento de bacterias. Estos ejemplos naturales se toman como referencia para la creación de diversos algoritmos de inteligencia de enjambres como:

- Optimización de colonia de hormigas
- Algoritmo de colonia de abejas
- Optimización de enjambre de partículas
- Algoritmo de murciélago
- Búsqueda Cuckoo

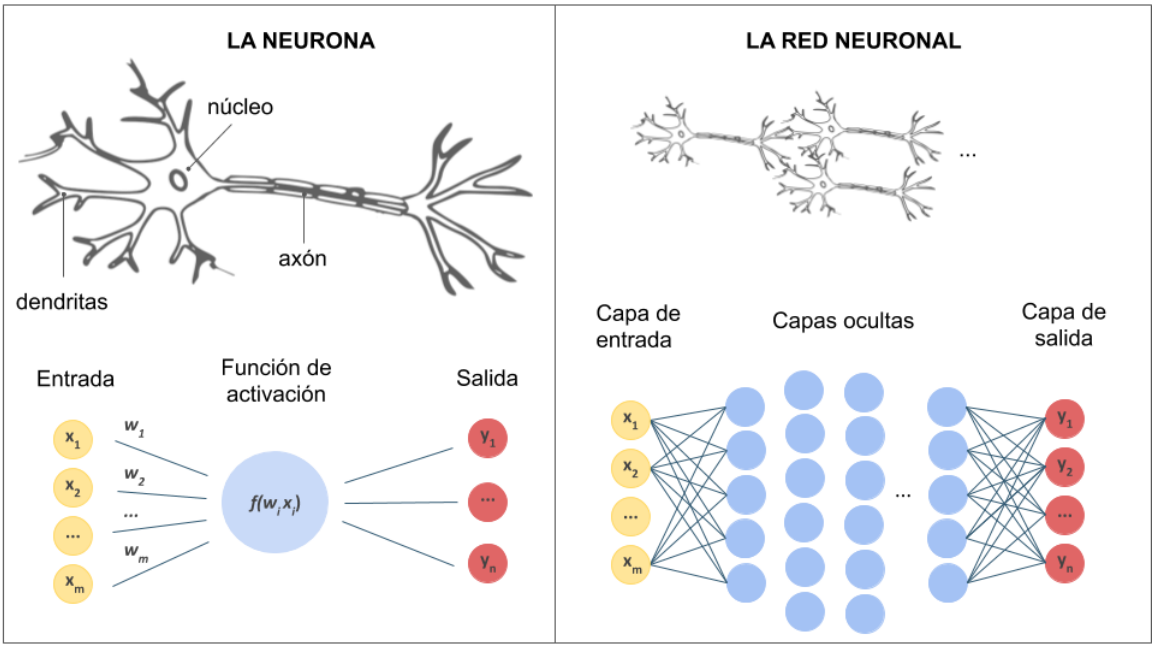

Figura 2.3: Comparación entre neuronas reales y artificiales [3]

Las **redes neuronales** recrean un modelo simplificado del funcionamiento del sistema nervioso (ver Fig. 2.3). Para ello, al igual que en un cerebro natural, se utiliza la neurona como unidad básica. Las conexiones entre las neuronas artificiales consiguen dar forma a un sistema inteligente. Las redes neuronales se pueden clasificar como parte de los algoritmos bioinspirados por estar basadas en una estructura biológica. Sin embargo, también pertenece al ámbito del machine learning por ser necesaria una fase de entrenamiento para establecer correctamente las conexiones entre neuronas. Por este motivo, se analizarán más en detalle en un apartado posterior.

2.1.3 Algoritmos evolutivos

El término de algoritmos evolutivos describe sistemas de resolución de problemas de búsqueda u optimización que emplean modelos computacionales basados en la evolución de seres vivos, diferenciándose del resto de algoritmos bioinspirados. Los algoritmos evolutivos aparecieron como respuesta a problemas difíciles o altamente irresolubles por métodos tradicionales. Estos problemas están caracterizados por una alta dimensionalidad, multimodalidad, fuerte no linealidad, no diferenciabilidad y presencia de ruido. También se incluye en este grupo a los problemas que tratan funciones dependientes del tiempo.

Las estrategias evolutivas para la resolución de problemas tuvieron un gran desarrollo en torno al año 1960 de la mano de John H. Holland. Durante esta época, Holland propuso incorporar los mecanismos naturales de selección y superviviencia a la resolución de problemas de IA [5]. Unos años más tarde, agrupó toda su investigación en su libro: *"Adaptation in Natural and Artificial Systems"* [32]. Esta simulación del proceso de selección de las especies dio lugar a un método de optimización estocástico que fue denominado como algoritmos evolutivos. Aun así, su investigación quedó relevada al ámbito académico dado que en esa época todavía no se disponía de los recursos computacionales para llevarlo a la práctica. Desde la concepción de este nuevo enfoque, surgieron nuevas vías de investigación que culminaron en varios subtipos de algoritmos evolutivos. Estos subtipos se analizarán más adelante.

Con la aparición de computadores más potentes y accesibles desde mediados de los 80 se ha podido aplicar esta clase de algoritmos a problemas que antes parecían imposibles y su desarrollo ha sido constante. Hoy en día, los algoritmos evolutivos han cobrado una gran importancia debido a su aplicación para la resolución de problemas de optimización, sobretodo en el ámbito de la ingeniería. Algunos ejemplos de aplicaciones en el mundo real son: diseño de circuitos, cálculo de estrategias de mercado, reconocimiento de patrones, acústica, ingeniería aeroespacial, astronomía y astrofísica, química, juegos, programación y secuenciación de operaciones, contabilidad lineal de procesos, programación de rutas e interpolación de superficies [5].

El elemento principal de los algoritmos evolutivos es la población de individuos, que representa las soluciones candidatas del problema. Este conjunto es sometido a diversas transformaciones y, después se aplica un proceso de selección para favorecer las mejores soluciones. Se denomina generación a cada ciclo de transformación y selección. Tras sucesivas generaciones, el mejor individuo de esa generación debería encontrarse cerca de la solución óptima del problema. De esta forma, los algoritmos evolutivos combinan la búsqueda aleatoria (que se realiza durante la fase de transformación) con la búsqueda dirigida que surge en el proceso de selección. Los algoritmos evolutivos, por lo tanto, cuentan con los siguientes componentes:

- **Población de individuos**, o representación de las diferentes soluciones.
- **Procedimiento de transformación**, utilizado para la creación de nuevos individuos partiendo de los anteriores.
- **Método de selección**, que se basa en la capacidad de los individuos para la resolución del problema.

Cada tipo de algoritmo evolutivo está definido por los métodos usados para la generación de soluciones. Partiendo de un conjunto inicial de soluciones, se emplean diversos operadores de búsqueda para refinar la solución final. Este refinamiento de las soluciones puede incluir técnicas más clásicas (como el algoritmo Hill Climbing) complementadas con procesos biológicos de exploración como la población de soluciones u operadores genéticos. A continuación se muestran los diferentes tipos, que consisten en modificaciones de la estructura general de los algoritmos genéticos:

- Estrategias de evolución
- Programación evolutiva
- Algoritmos genéticos
- Programación genética

Las **estrategias de evolución** (o **estrategias evolutivas**) fueron desarrolladas por Ingo Rechenberg y Hans-Paul Schwefel a mediados de la década de los 60 orientadas a la optimización continua de parámetros en problemas de ingeniería [33]. Las estrategias evolutivas utilizan vectores de números reales para codificar las distintas soluciones de los problemas. Para la creación de la siguiente generación se sirve de la recombinación o cruce (crossover aritmético) y mutación. Por otra parte, para el proceso de selección (tanto determinística como probabilística) se descartan las peores soluciones de la población, eliminando a los individuos que no superan la aptitud promedio.

La **programación evolutiva** fue concebida por Lawrence J. Fogel en 1960 mientras trabajaba en la Fundación Nacional de Ciencias (NSF) de Estados Unidos [34]. Fogel diseñó esta nueva técnica en un intento de crear inteligencia artificial aplicando sus conocimientos en el campo de la biotecnología. Más tarde, otros investigadores como G. H. Burgin y J. W. Atmar, entre otros, realizarían aportaciones al esquema original. Para representar el problema, la programación evolutiva utiliza números reales agrupados en cualquier estructura de datos y, al igual que las estrategias evolutivas, se basa en los procesos de selección y mutación. La principal diferencia con respecto a las estrategias evolutivas radica en la ausencia del proceso de recombinación.

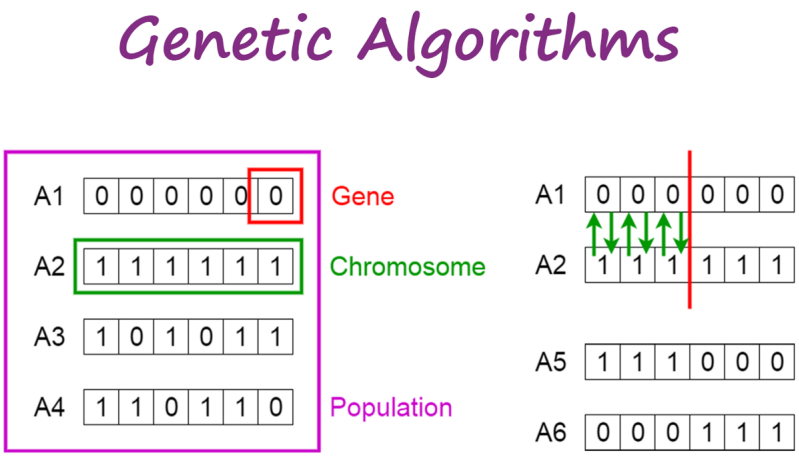

Figura 2.4: Representación de la población en algoritmos genéticos [4]

Los **algoritmos genéticos** se basan en el proceso genético de los seres vivos, con soluciones análogas a los cromosomas. Durante las sucesivas generaciones, los individuos de la población se reproducen y sus descendientes contienen la combinación genética de sus padres. Los algoritmos genéticos trabajan con cadenas binarias como representación de los individuos (o soluciones del problema) de la población (ver Fig. 2.4). Este espacio de soluciones se explora aplicando diversas trasformaciones. Estas transformaciones son idénticas a las que suceden en los seres vivos: cruce, inversión y mutación. Para el proceso de

selección se emplea el mecanismo de la ruleta (que a veces puede contar con elitismo). Los algoritmos genéticos componen el paradigma más completo de computación evolutiva, ya que sintetizan todas las ideas fundamentales de este enfoque. En consecuencia, cuentan con la mayor base teórica entre los algoritmos evolutivos. Estos algoritmos tienen una gran adaptabilidad, ya que es posible incorporar fácilmente nuevos cambios que surjan en el ámbito de la computación evolutiva. Además, también permiten la hibridación con otros paradigmas y enfoques, incluso si forman parte de otras áreas de investigación Por este motivo, los algoritmos genéticos son el tipo de algoritmos evolutivos más popular en este momento.

La **programación genética** fue introducida por John R. Koza [35]. En este algoritmo, los individuos de la población son representados como autómatas de longitud variable, definidos empleando Expresiones-S de LISP que representan árboles binarios. Como operadores de variación se utilizan los métodos de crossover y modificación, junto a procesos de selección.

Como final de esta sección se muestra a continuación una tabla que resume las características de todos los tipos de algoritmos evolutivos analizados:

Algoritmo	Representación del problema	Operadores de variación	Métodos de selección
Algoritmos genéticos (Goldberg)	Cadena binaria	Mutación y crossover	Selección de rueda de ruleta (a veces con elitismo)
Estrategias de Evolución (Rechenberg/Schwefel)	Vector de reales + desviaciones estándar	Mutación gaussiana y crossover aritmético (diferentes tipos)	Diferentes tipos de selección: lambda,mu; lambda+mu...
Programación evolutiva (Fogel)	Números reales	Mutación	Diversos tipos de selección
Programación genética (Koza)	Expresiones-S de LISP representadas habitualmente como árboles	Crossover, algo de mutación	Diversos tipos de selección

Figura 2.5: Resumen de algoritmos evolutivos [5]

2.1.3.1 Recocido simulado

Existen mútiples problemas en áreas como biología, física, matemáticas e ingeniería que exigen localizar el óptimo global de funciones multidimensionales. El recocido simulado es un algoritmo metaheurístico que puede ser usado para resolver esta clase de problemas de optimización global.

A diferencia de otros algoritmos evolutivos, el recocido simulado no se inspira en un proceso biológico. En su lugar, utiliza como base el proceso metalúrgico usado para modificar las propiedades de un metal. El recocido físico es *"un tratamiento térmico mediante el que se calienta el metal hasta una temperatura determinada durante un período también definido, para después enfriarlo poco a poco"* [36]. Si este proceso de enfriamiento es lo suficientemente lento, se consigue que los átomos del metal formen una estructura estable, sin imperfecciones. Sin embargo, si el proceso se realiza demasiado rápido, se obtiene una estructura con imperfecciones. Existe, por tanto, una analogía entre el recocido físico y el simulado [37]. La solución del problema equivale a la estructura atómica del metal. De la misma forma, el valor de la función que debe ser optimizada se corresponde con la medida de la energía en el sistema físico, utilizando la temperatura como elemento de control. Por último, un estado físico imperfecto se asocia con un óptimo local, mientras que el estado estable sería el óptimo global buscado.

El recocido simulado comienza con un estado inicial S. Este estado puede elegirse de forma aleatoria o siguiendo algún tipo de criterio. Partiendo del estado inicial, se genera el estado vecino S'. Si la evaluación o la energía del estado S' es menor que S, se toma S' como el nuevo estado. En el caso contrario, la solución puede ser peor que la actual, por lo que se elige esta nueva solución dependiendo de una probabilidad determinada (siendo este). Esta probabilidad depende tanto de las diferencias en las evaluaciones como de la temperatura del sistema T. Para realizar este cálculo se utiliza la siguiente ecuación [23]:

$$P(\Delta f, T) = e^{(\Delta f / T)}$$

donde:

P es la probabilidad de aceptación del nuevo estado.

Δf es la diferencia de las evaluaciones de la función para cada estado.

T es la temperatura del sistema.

Esta aleatoriedad del recocido simulado es el motivo por el que se considera un algoritmo estocástico y, por tanto, introduciendo entradas iguales el resultado final puede ser distinto. Al comienzo del algoritmo, con un valor alto de T, es común que se acepten peores soluciones. Sin embargo, según disminuye el valor de la temperatura se va reduciendo la probabilidad de aceptación de estas soluciones y, cuando este valor se acerca a 0, llega un punto en el que solo se aceptan soluciones que mejoran a la anterior. Se ha demostrado que este proceso converge a la solución óptima si T decrece con la lentitud suficiente [38]. Es posible utilizar diversas funciones para controlar la reducción de la temperatura. No obstante, la función más usada es [23]:

$$T_{k+1} = T_k \, \alpha$$

donde:

T_{k+1} es el nuevo valor de la temperatura, o T .

T_k es el valor anterior de T.

α es una constante que debe estar comprendida dentro del intervalo $[0,8 - 0,99]$.

Por tanto, se puede observar que la elección de diversos parámetros como la constante α, el estado inicial, la generación de los estados vecinos, el número de iteraciones o el criterio de parada puede influir de forma considerable en el rendimiento final del algoritmo. El pseudocódigo resultante del recocido simulado se muestra en la Fig. 2.1.

Data: Sea $f(s)$ el coste de la solución s y sea $G(s)$ su entorno.
Seleccionar una solución inicial S;
Seleccionar una temperatura inicial $T_i > 0$;
Seleccionar una función de reducción de la temperatura α;
Seleccionar un número de iteraciones N;
Seleccionar un *criterio* de parada;
Result: La mejor solución encontrada.
begin
  $i = 0$;
  while *criterio de parada* $\neq$ *true* **do**
    while $i < N$ **do**
      Seleccionar aleatoriamente una solución $S' \in G(s)$;
      Sea $\Delta f = f(S') - f(S)$;
      if $\Delta f < 0$ **then**
        $S_{optimo} = S'$;
      else
        Generar aleatoriamente $t \in L(0, 1)$;
        if $t < e^{(\Delta f/T)}$ **then**
          $S_{optimo} = S'$;
      i += 1;
    $T_{i+1} = T_i * \alpha$;

Algoritmo 2.1: Pseudocódigo del recocido simulado [23]

2.1.4 Machine learning

El machine learning o aprendizaje autónomo engloba una disciplina que tiene como objetivo crear programas que son capaces de aprender a resolver un problema por su cuenta, sin ser programados para realizar esa tarea en concreto. Esta disciplina de inteligencia artificial ha ganado popularidad en los últimos años gracias al incremento de la capacidad de cómputo de los ordenadores y al uso de nuevos algoritmos de entrenamiento más eficientes. También ha sido muy importante la disponibilidad de una gran cantidad de datos de entrenamiento gracias a la expansión de Internet. La principal diferencia entre esta familia de algoritmos y el resto radica en que es necesario un entrenamiento previo antes de poder aprovechar el modelo para realizar predicciones. Esta tarea se puede paralelizar fácilmente y, por tanto, se consigue una gran reducción del tiempo necesario para completar el entrenamiento al ejecutar el algoritmo utilizando GPUs.

El machine learning es una parte muy importante del campo de la ciencia de datos [39]. Mediante el uso de modelos estadísticos, los algoritmos son entrenados para realizar clasificaciones y predicciones. Esto permite recabar información clave en proyectos de data mining (proceso de extracción de información de gran cantidad de datos), que puede ser usada más tarde para facilitar la toma de decisiones en empresas e incrementar sus métricas de crecimiento. El machine learning, por tanto, no solo se limita al ámbito académico, ya que es usado hoy en día en diversas aplicaciones como el reconocimiento del habla, visión artificial, sistemas de recomendación o el intercambio automático de acciones en bolsa.

Dentro de esta categoría, existen tres subgrupos principales dependiendo del tipo de datos usados para el entrenamiento y la forma de analizar si el algoritmo está progresando de forma adecuada:

- Aprendizaje supervisado

- Aprendizaje no supervisado
- Aprendizaje por refuerzo

El **aprendizaje supervisado** utiliza un conjunto de datos etiquetados que son suministrados durante la etapa de entrenamiento. El objetivo final es conseguir que el algoritmo sea capaz de generalizar y clasificar adecuadamente otros datos distintos a los suministrados durante el entrenamiento, asignando las etiquetas correctas. Durante el proceso de entrenamiento, se ajustan progresivamente los diferentes parámetros internos para obtener una salida lo más parecida a la deseada. Además, el entrenamiento debe realizarse para evitar en la medida de lo posible tanto el **overfitting** (o sobreajuste) como el **underfitting** (o desajuste), como se muestra en la Fig. 2.6. El overfitting se produce cuando el modelo se ha entrenado demasiado y no es capaz de generalizar lo suficiente para nuevos datos. Es decir, ha memorizado los datos de entrenamiento. Por otra parte, el underfitting sucede al realizarse un entrenamiento insuficiente, de tal forma que el modelo aún no ha aprendido lo suficiente para devolver un resultado aceptable. Estos dos casos también aparecen en otros tipos de machine learning.

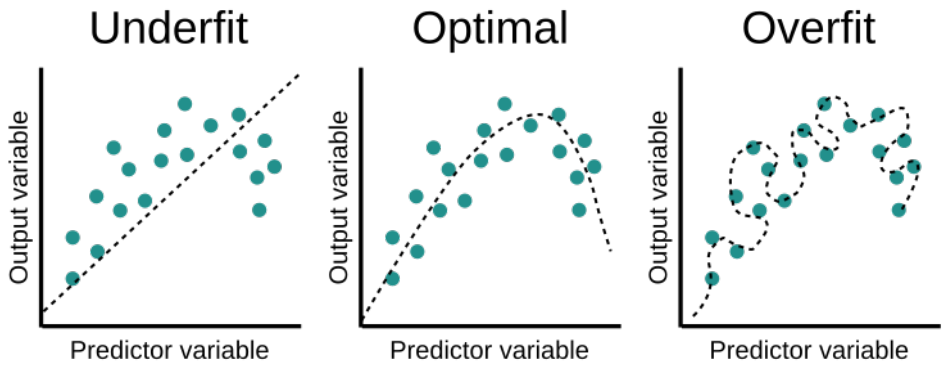

Figura 2.6: Resultados de entrenamiento [6]

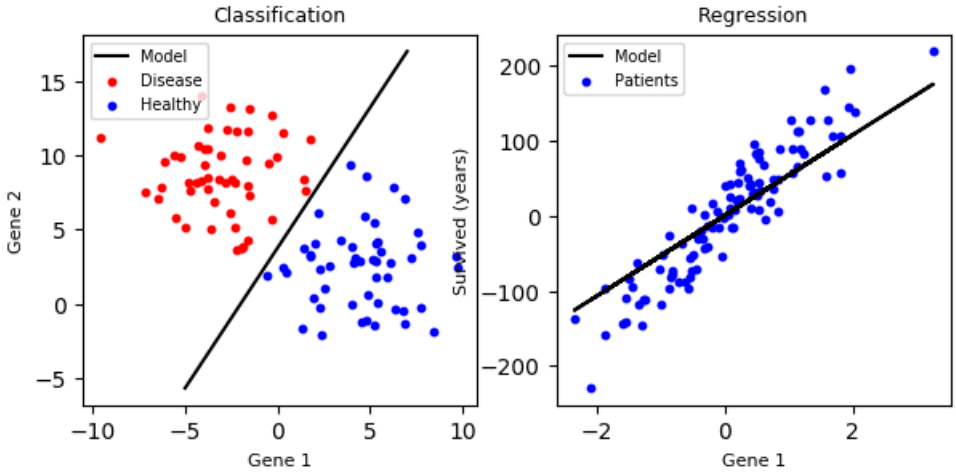

Figura 2.7: Clasificación y regresión [7]

A su vez, dentro del aprendizaje supervisado existen dos subcategorías dependiendo del tipo de variable objetivo o tipo de problema a resolver [40]: clasificación y regresión. En los problemas de clasificación la variable obtenida como resultado es de tipo categórico, mientras que en regresión es de tipo numérico. En la Fig. 2.6, se muestran ambos casos. Las redes neuronales son un ejemplo importante de aprendi-

zaje supervisado. Se analizarán más en detalle en el siguiente apartado. Otros ejemplos de algoritmos supervisados son [40]:

- Árboles de decisión
- Clasificación de Naïve Bayes
- Regresión por mínimos cuadrados
- Regresión Logística
- Support Vector Machines (SVM)
- Métodos "Ensemble" (Conjuntos de clasificadores)

En el caso del **aprendizaje no supervisado** los datos que se suministran al algoritmo no tienen ninguna clasificación realizada de antemano. Por lo tanto, se debe conseguir que el algoritmo encuentre patrones de forma autónoma, pudiendo incluso arrojar conclusiones que son difíciles de apreciar para los humanos. Estos algoritmos tienen, en consecuencia, un carácter exploratorio. Esta capacidad de encontrar similitudes y diferencias en un conjunto de datos, convierten al aprendizaje no supervisado en la opción idónea para el análisis de datos, la segmentación de clientes y el reconocimiento tanto de imágenes como de patrones. También suelen ser usados para reducir la dimensionalidad de algunos conjuntos de datos. Los tipos de algoritmos más usados en el aprendizaje no supervisado son [40]:

- Algoritmos de clustering (agrupamiento) como k-means (o k-medias)
- Análisis de componentes principales
- Descomposición en valores singulares
- Análisis de componentes principales

Las redes neuronales también pueden formar parte de esta categoría [39] dependiendo del tipo de datos suministrados.

El **aprendizaje por refuerzo**, sin embargo, trata de aplicar conceptos de psicología recompensando al algoritmo cuando realiza las acciones deseadas. Es similar al aprendizaje supervisado, pero no se proporcionan datos de muestra. El modelo aprende utilizando un proceso de prueba y error. Cuando se suceden las salidas favorables, este comportamiento se verá reforzado y, progresivamente, el algoritmo acabará adaptándose al modelo deseado.

2.1.4.1 Redes neuronales

Las redes neuronales son un modelo computacional inspirado en el funcionamiento de un cerebro humano. Como se ha mencionado antes, toma conceptos de machine learning y de algoritmos bioinspirados. El carácter bioinspirado se ve reflejado por la estructura biológica que toma como referencia. Por otra parte, se incluye como método de machine learning, ya que, es necesaria una fase de entrenamiento para ajustar su estructura interna.

La estructura de una red neuronal está compuesta por múltiples nodos. Estos nodos se denominan neuronas artificiales y son la unidad básica de una red neuronal. Las neuronas se conectan entre sí formando varias capas dando lugar a una red neuronal (ver Fig. 2.8).

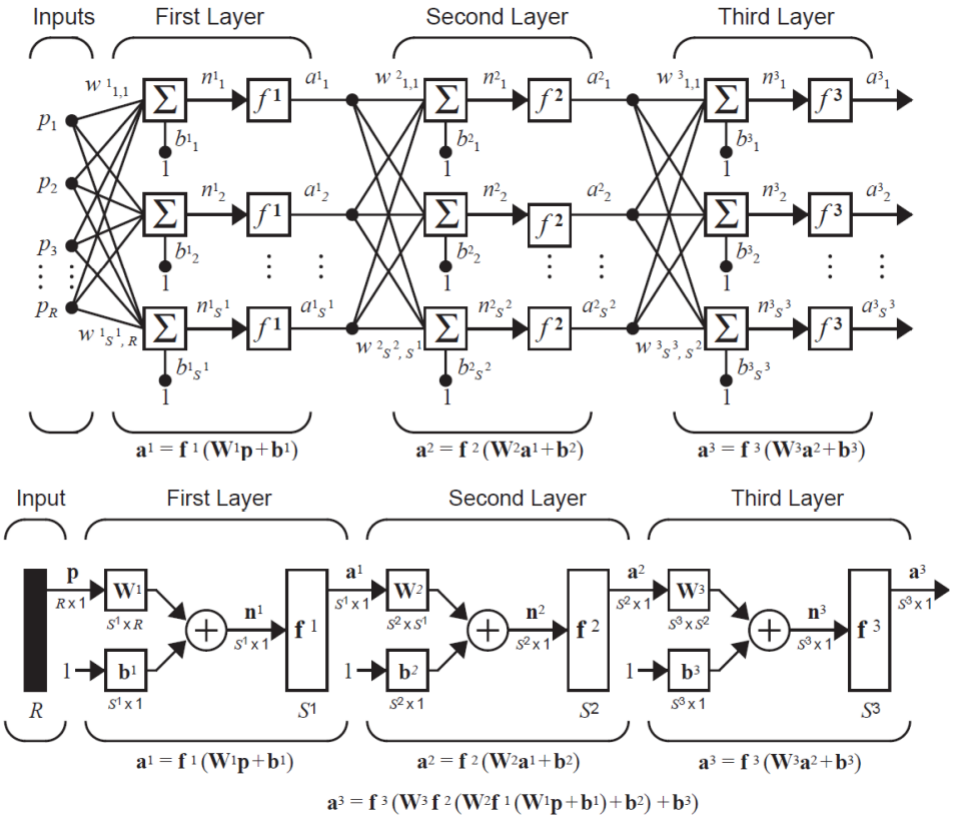

Figura 2.8: Red neuronal de múltiples capas [8]

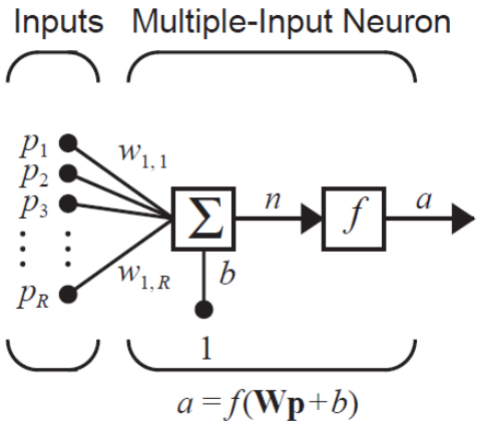

Figura 2.9: Neurona artificial con múltiples entradas [8]

Una neurona artificial se trata de una función que recibe varios valores como entrada y tras realizar diversas transformaciones devuelve un valor de salida (ver Fig. 2.9). Cada una de las entradas es multiplicada por su peso correspondiente. Después se realiza la suma de estos valores junto a un parámetro conocido como ***bias*** (o sesgo en español) y se aplica una función de activación que varía dependiendo del tipo de neurona usada. Por ejemplo, en el **perceptrón** [8], que es uno de los tipos de redes neuronales más simples, se utiliza la siguiente función escalón (también conocida como función escalón de Heaviside) en las diversas neuronas:

$$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$$

Otros ejemplos usados frecuentemente son funciones lineales o sigmoidales, entre otras. Una vez se ha decidido la estructura de red neuronal más adecuada para el problema que debe ser resuelto, es necesario realizar el entrenamiento de la red. El entrenamiento es un proceso que consiste en suministrar datos de prueba a la red neuronal y comparar su salida con la salida ideal. En función del resultado obtenido se adaptan los pesos de todas las neuronas. A lo largo de diversas iteraciones este proceso consigue aproximar el rendimiento de la red neuronal a la función objetivo. No obstante, durante el proceso de entrenamiento es imprescindible seleccionar adecuadamente parámetros como el nº de parámetros de entrada y el límite de ciclos para evitar los casos de **overfitting** y **underfitting** que fueron comentados anteriormente (ver Fig. 2.6).

Para adaptar los valores internos de la red neuronal se pueden utilizar diferentes algoritmos. El método más simple y más conocido es el algoritmo de descenso por el gradiente. Otros ejemplos son el método de Newton, el gradiente conjugado, el método Cuasi-Newton y el algoritmo de Levenberg-Marquardt [9]. En la Fig. 2.10 se muestra una comparación de los diferentes métodos analizando su velocidad y su coste de memoria.

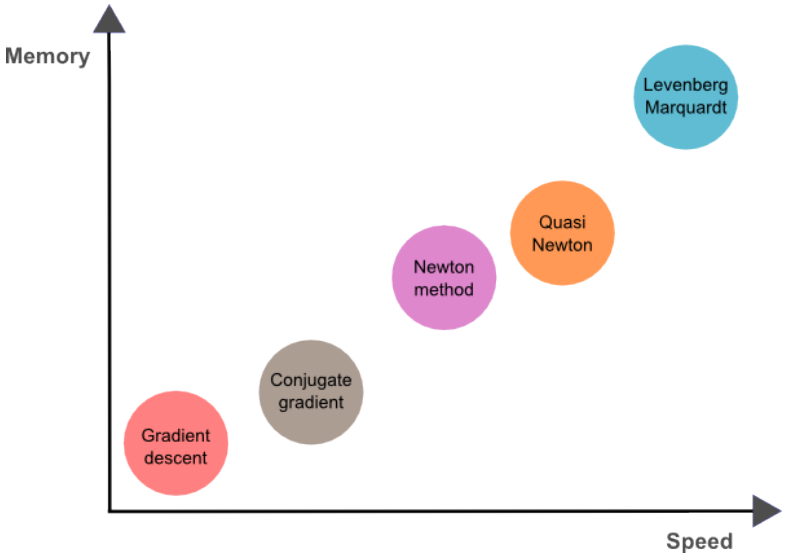

Figura 2.10: Métodos de entrenamiento de redes neuronales [9]

2.1.4.2 Backpropagation

Para el entrenamiento de las redes neuronales multicapa es necesario un enfoque diferente. El algoritmo de backpropagation (propagación hacia atrás) es utilizado en este caso. Este algoritmo se introdujo en torno a la década de 1970 [41] pero no cobró especial importancia hasta que D. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams publicaron un artículo en 1986 [42] mostrando la utilidad del nuevo procedimiento. En este artículo se detallan varios ejemplos de redes neuronales donde el algoritmo de backpropagation obtiene un tiempo de entrenamiento mucho menor que el resto de algoritmos de aprendizaje. Esto hace posible utilizar las redes neuronales para la resolución de problemas que antes eran inviables. Por este motivo, el algoritmo de backpropagation es uno de los más usado hoy en día para el entrenamiento de redes neuronales.

El algoritmo de backpropagation se basa en la derivada parcial del coste C con respecto a cada uno de los pesos w y los sesgos b de la red [10]:

$$\frac{\partial C}{\partial w}, \frac{\partial C}{\partial b}$$

Esta expresión explica cómo varía el coste cuando se alteran tanto los pesos como los sesgos de la red. De esta forma el algoritmo de backpropagation nos proporciona información específica sobre qué pesos y sesgos deben ser modificados. Es decir, calcula qué influencia tiene cada neurona individual sobre el resultado final y modifica sus parámetros en consecuencia. El objetivo de este algoritmo, por tanto, es calcular las derivadas parciales para todas las neuronas de la red.

Antes de continuar con la aplicación del algoritmo de backpropagation es necesario puntualizar varios apartados. En primer lugar, es necesario especificar la notación que será usada en el resto de la explicación. Para los pesos, se usará w^l_{jk} para hacer referencia al peso de la conexión desde la neurona k en la capa $(l-1)$ a la neurona j en la capa l. Por ejemplo, en el diagrama que aparece a continuación se muestra el peso de una conexión que parte desde la cuarta neurona de la segunda capa a la segunda neurona de la tercera capa de la red:

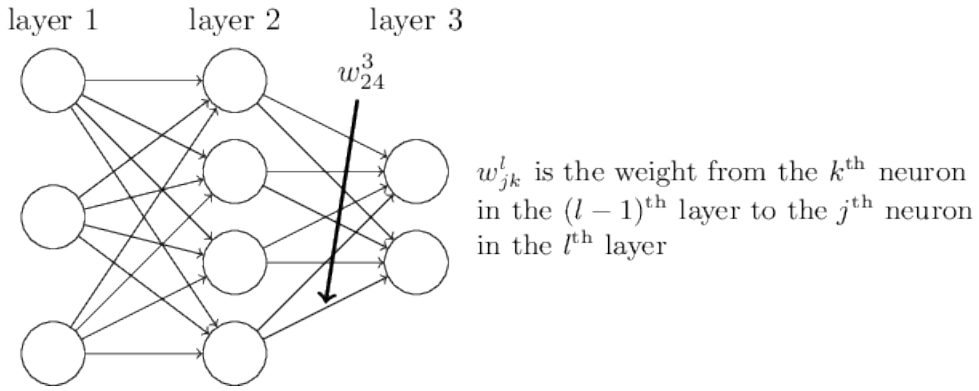

Figura 2.11: Notación de pesos [10]

Para los sesgos se procederá de forma similar, siendo b^l_j el sesgo de la neurona j en la capa l. Las salidas se representan de forma idéntica, a^l_j es la activación de la neurona j en la capa l. A continuación se muestra un ejemplo de la notación:

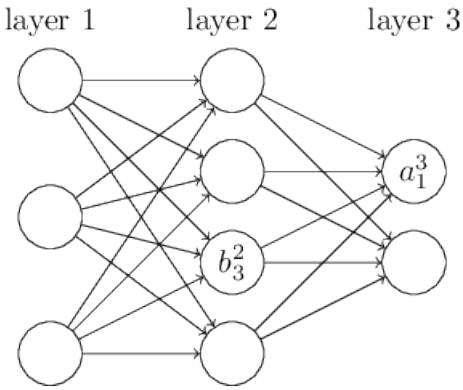

Figura 2.12: Notación de sesgo y activación [10]

Por lo tanto, la activación de una neurona depende de la capa anterior de la red de la siguiente forma:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_j^{l-1} + b_j^l\right)$$

Esta ecuación, a su vez, puede reescribirse de forma más clara aplicando el concepto de función vectorizada. Utilizando esta notación, expresamos de forma más compacta que se aplica una función a cada elemento de un vector. Por ejemplo, si contamos con la función $f(x) = x^2$, la forma vectorizada de f, donde se eleva al cuadrado cada elemento del vector, sería:

$$f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix}$$

De esta forma, la función de activación mencionada anteriormente quedaría de la siguiente forma:

$$a_j^l = \sigma(w^l a^{l-1} + b^l)$$

Esta notación permite centrar la atención en cómo las activaciones de una capa de neuronas están condicionadas por las salidas de la capa anterior, simplemente se aplica la matriz de pesos a las salidas anteriores, se añade el vector de sesgo y, finalmente, se aplica la función de activación σ. De esta forma, se trabaja con un menor número de índices y se obtiene una mejor vista global del algoritmo.

Por otra parte, es necesario definir una función de coste para calcular el error de la red neuronal en cada iteración. La función de coste que se utilizará en este ejemplo es la función del coste cuadrático, siendo en este caso [10]:

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

donde n es el número total de ejemplos de entrenamiento, L es el número de capas de la red y a^L es el vector de activaciones de salida de la red.

Además, en lugar de utilizar la multiplicación de matrices típica, se usará el producto de Hadamard, dado que existen librería de operaciones matriciales que contienen implementaciones rápidas de esta operación. No obstante, el algoritmo se puede encontrar también usando la multiplicación de matrices estándar. El producto de Hadamard no es más que una multiplicación elemento a elemento de dos matrices. Por ejemplo, la notación $s \odot t$ expresa el producto de Hadamard de ambos vectores. Este proceso se llevaría a cabo de la siguiente forma:

$$\begin{bmatrix}5\\2\end{bmatrix} \odot \begin{bmatrix}1\\3\end{bmatrix} = \begin{bmatrix}5*1\\2*3\end{bmatrix} = \begin{bmatrix}5\\6\end{bmatrix}$$

Contando con todos elementos, la aprte restante sería calcular el error de todas las capas de la red neuronal, comenzando por la última de ellas. El error δ_j^l de la neurona j en la capa l queda definido por la expresión:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

donde z representa un cambio en el peso de la neurona j en la capa l. Durante la ejecución del algoritmo de backpropagation se calcula el error y el gradiente de la función de coste en cada iteración, utilizando cuatro ecuaciones principales que describen cómo calcular estos valores para cada capa de la red (ver Fig.

2.13). Para una explicación detallada de cómo se obtienen estas ecuaciones y más información sobre el algoritmo de backpropagation es recomendable consultar al capítulo 2 del libro online escrito por Michael Nielsen [10], ya que se ha tomado como fuente de información para esta sección.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Figura 2.13: Ecuaciones fundamentales del algoritmo de backpropagation [10]

Finalmente, en cada iteración del algoritmo de backpropagation se operaría de la siguiente forma:

1. **Entrada** x**:** Se sestablece la activación correspondiente a^1 para la capa de entrada.
2. **Transmisión:** Para cada $l = 2, 3, ..., L$ se calcula $z^l = w^l a^{l-1} + b^l$ y $a^l = \sigma(z^l)$.
3. **Error de salida** δ^L**:** Cálculo del vector $\delta^L = \Delta_a C \odot \sigma'(z^l)$.
4. **Propagación hacia atrás del error:** Para cada $l = L-1, L-2, ..., 2$ calcular $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Salida:** El gradiente de la función de coste se obtiene mediante $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ y $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Este proceso se repite hasta que el error de la red neuronal cae por debajo de un umbral definido, aunque también se pueden definir otros criterios de parada. Por otra parte, aunque el algoritmo de backpropagation cuenta con múltiples ventajas, también es necesario tener en cuenta sus debilidades. Dependiendo del problema, se puede dar el caso en el que la convergencia es lenta para una tasa de aprendizaje fija. Para mejorar el rendimiento del algoritmo en estos casos, se pueden utilizar varias modificaciones heurísticas [8]. Una opción es añadir un momento inicial a la tasa de aprendizaje. De esta forma, se suele acelerar la convergencia cuando la trayectoria del gradiente se mueve en la misma dirección. Otra aproximación utilizar una tasa de aprendizaje variable que es ajustada durante el entrenamiento. Si en un instante de tiempo dado se incrementa el error de la red en un valor que supera el intervalo definido, la tasa de aprendizaje actual es multiplicada por un factor $0 < \rho < 1$ y, si se está usando un coeficiente de momento, se establece a 0.

2.1.5 Deep learning

El deep learning, o aprendizaje profundo, es un subtipo de redes neuronales multicapa con múltiples capas intermedias de neuronas (ver Fig. 2.14). La base del deep learning es el uso de redes neuronales profundas para obtener múltiples niveles de abstracción. De esta forma, se reduce progresivamente la complejidad del problema y cada neurona solo debe encargarse de una tarea simple. Por ejemplo, si la tarea que

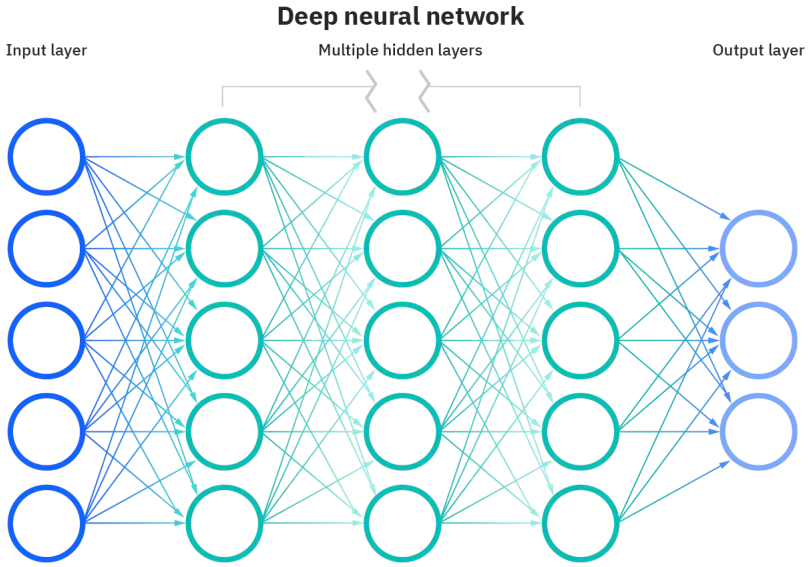

Figura 2.14: Red neuronal profunda [11]

debe ser abordada es el reconocimiento visual de patrones [10], las neuronas de la primera capa podrían ser capaces de reconocer aristas. La segunda capa de neuronas sería capaz de identificar formas como triángulos o rectángulos formados a partir de esas aristas. Más tarde, una tercera capa estaría capacitada para reconocer figuras más complejas y así sucesivamente. El uso de múltiples capas de abstracción brinda a los algoritmos de deep learning la capacidad de resolver problemas de reconocimiento de patrones más complejos con respecto a las redes neuronales tradicionales.

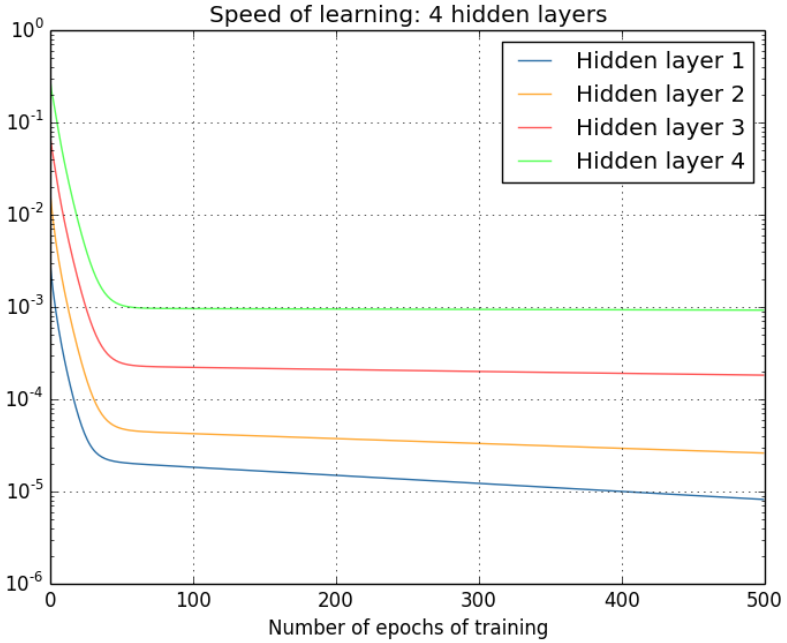

Figura 2.15: Tasa de aprendizaje de las capas ocultas [10]

Sin embargo, ¿cómo se entrenan estas redes neuronales? Si se utiliza el algoritmo de descenso por el gradiente mediante backpropagation se observa que el rendimiento de la red profunda no es mucho mejor,

e incluso puede ser peor, que el de una red neuronal tradicional. Analizando más en detalle la situación, se aprecia que cada capa de la red profunda aprende a un velocidad distinta (ver Fig. 2.15). Esta situación es conocida como el problema de desvanecimiento de gradiente [12]. Esta situación sucede especialmente cuando la función de activación usada por las neuronas es una función sigmoidal. La función sigmoidal comprime un gran espacio de entrada en un pequeño intervalo de salida comprendido entre 0 y 1. Por lo tanto, un cambio grande en la entrada se traducirá en un pequeño cambio de salida, obteniendo un valor pequeño de la derivada (en la Fig. 2.16 se muestra de forma gráfica esta situación). Si este proceso se repite durante varias capas de neuronas, el valor de la derivada irá reduciéndose y las neuronas de las últimas capas aprenderán a un ritmo muy lento.

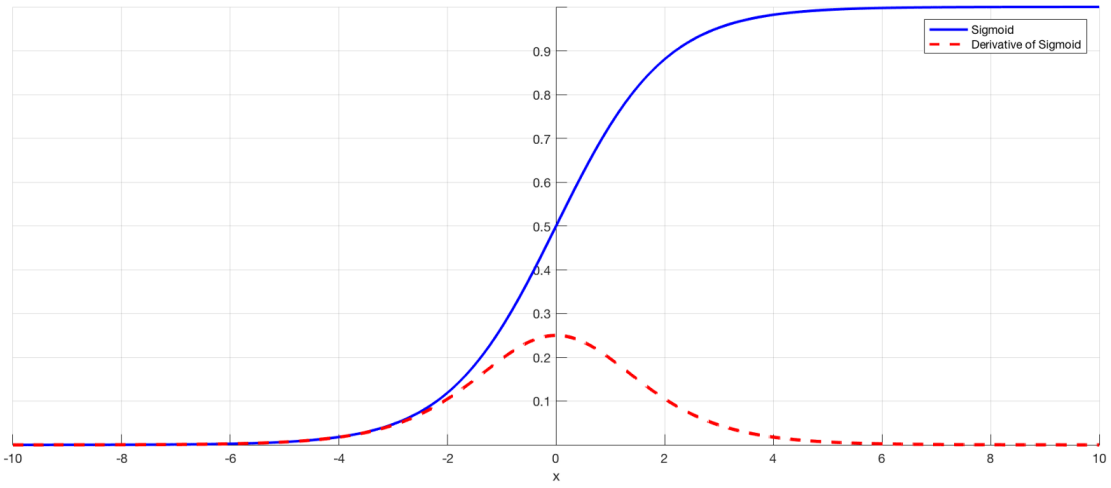

Figura 2.16: Gráfica de la función sigmoidal y su derivada [12]

Normalmente, las capas adyacentes de una red neuronal están conectadas por completo entre ellas. Es decir, cada neurona está conectada a todas las neuronas de las capas adyacentes. Sin embargo, para la clasificación de imágenes resulta extraño optar por esta estructura dado que no se está teniendo en cuenta la posición de los píxeles, se está tratando a todos por igual. Aprovechando esta característica, es posible diseñar una estructura más adaptada y que además pueda ser entrenada con mayor facilidad. Este tipo de redes neuronales son conocidas como redes neuronales convolucionales, o CNN, y son muy usadas actualmente en el campo de la visión artificial. Entre sus aplicaciones prácticas se encuentran tareas como el reconocimiento de imágenes y vídeos, la clasificación, o el procesamiento del lenguaje natural.

La estructura de una red convolucional está inspirada en la corteza visual del cerebro humano [13]. La base de este algoritmo son las neuronas individuales solo responden a una parte limitada del campo de visión, conocido como campo receptivo. Estas estructuras se solapan para conseguir cubrir por completo el campo visual. De forma análoga, en la red convolucional este campo receptivo se conoce como filtro o kernel y se aplica en la capa de convolución. Durante las sucesivas capas de la red, las dimensiones de la matriz original se reducen, extrayendo las características esenciales. Finalmente, cuando la imagen de entrada ha sido procesada, se puede introducir como entrada de una red neuronal tradional que actúa como capa final. Esta red neuronal sí puede ser entrenada mediante el algoritmo de backpropagation. Se pueden observar estas etapas de la red convolucional en el ejemplo de la Fig. 2.17.

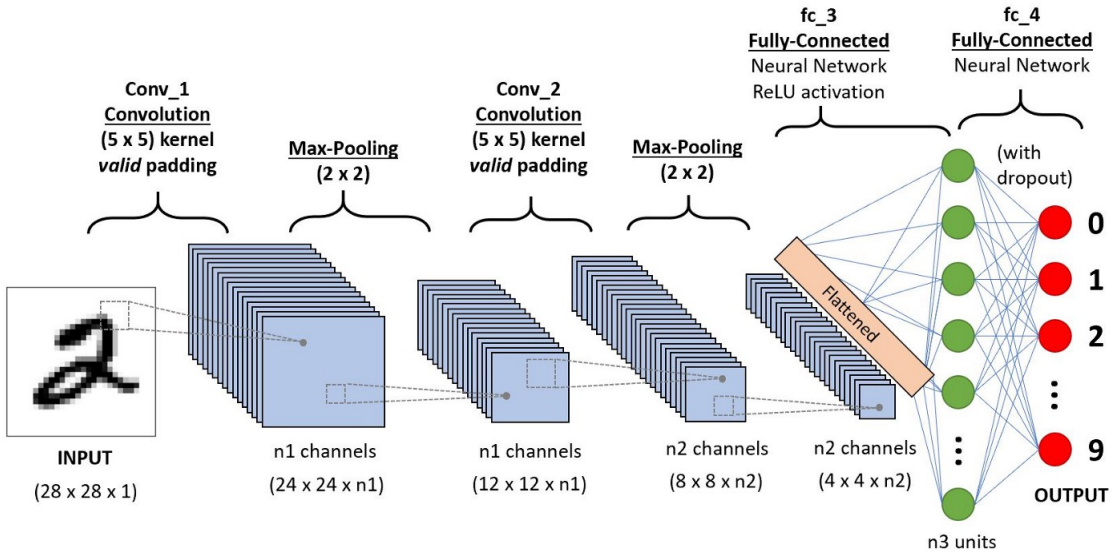

Figura 2.17: Red neuronal convolucional [13]

2.1.5.1 Word2vec

El deep learning es ampliamente utilizado para el procesamiento de imágenes y audio dado que se trabaja con datos multidimensionales que sintetizan una gran cantidad de información. Esta información se representa mediante el uso de vectores que contienen parámetros como la intensidad de los canales RGB en los diferentes píxeles cuando se trata de una imagen o coeficientes espectrales en el caso de una grabación de audio [43].

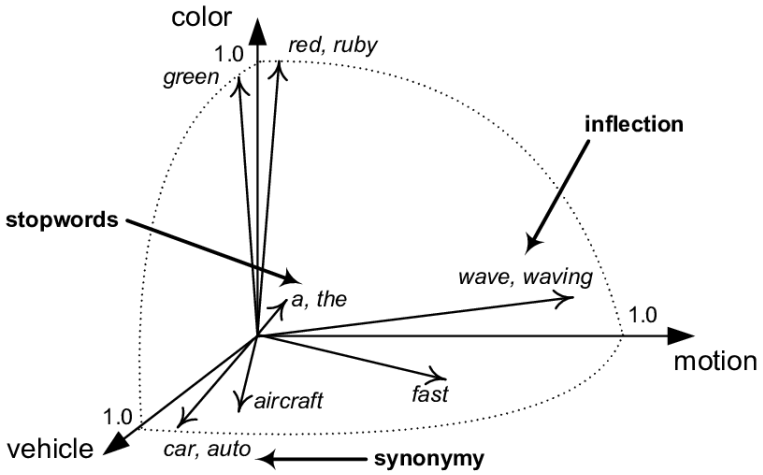

Figura 2.18: Visualización un modelo de espacio vectorial basado en temas [14]

Sin embargo, en el campo del procesamiento de lenguaje natural, o NLP (Natural Language Processing), es común utilizar una representación de símbolos individuales y discretos para representar las diferentes palabras. Por ejemplo se podría representar la palabra “cobra” con el id *99* y “anaconda” como *115*. Esta codificación arbitraria no refleja correctamente las relaciones que existen entre palabras (en este caso que ambas palabras representan a especies de serpientes, por ejemplo) y ocasiona una dispersión muy grande de los datos que dificulta enormemente el entrenamiento de algoritmos de deep learning. Una representación alternativa más interesante son los modelos de espacio vectorial, o VSM (Vector Space

Models). De esta forma, se codifican las palabras como vectores en un espacio multidimensional donde las palabras relacionadas se encuentran en zonas cercanas del espacio. Es decir, cuanto menor sea el ángulo entre los dos vectores, mayor será la relación de las dos palabras. Por ejemplo, palabras como "rojo" y "verde" aparecen próximas en el espacio vectorial (ver Fig. 2.18).

Word2vec es una de las técnicas más populares para el aprendizaje de este tipo de representaciones, ya que es un método eficiente computacionalmente. Este algoritmo fue desarrollado por Tomas Mikolov y otros trabajadores de Google en 2013 [44]. Existen dos modelos de arquitecturas que permiten aplicar este método: **Continuous Bag of Words** (**CBOW**) o el modelo **Skip Gram**.

CBOW toma como entradas el contexto de cada palabra y trata de predecir una palabra adecuada a ese contexto. La entrada de la red neuronal es, por tanto, un vector *one-hot* (cadena de bits donde solo uno de ellos puede estar activado) de tamaño V. La capa oculta contiene N neuronas, mientras que la salida tiene el mismo tamaño V que la entrada. A estos datos finales se les aplica una función softmax antes de ser enviados como salida de la red. En la Fig. 2.19 se puede observar esta estructura, donde [15]:

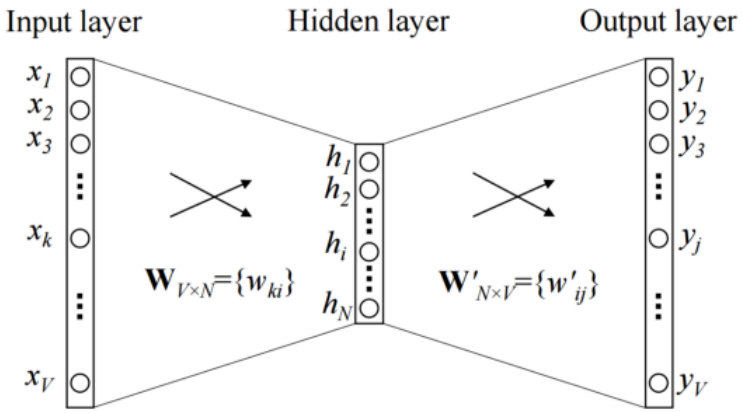

Figura 2.19: Modelo CBOW simple con una única palabra en el contexto [15]

W_{vn} es la matriz de pesos de tamaño $V \times N$ que conecta la entrada de la red con la capa oculta.

W'_{nv} es la matriz de pesos de tamaño $N \times V$ que realiza la misma función pero conectando las salidas de la capa oculta con la capa final.

Las neuronas de la capa oculta no cuentan con función de activación como en otros tipos de redes neuronales. Simplemente transmiten la suma ponderada de las entradas a la siguiente capa de neuronas. Partiendo de este modelo básico, se puede diseñar un modelo más complejo que utiliza múltiples palabras de contexto (ver Fig, 2.20). En este caso se emplean C palabras de contexto y, cuando se utiliza la matriz W_{vn} para calcular las entradas de la capa oculta, se considera la media de de todas las palabras de contexto. Mediante el uso este procedimiento es posible obtener la representación de vectores de palabras que se comentó anteriormente. No obstante, es posible realizar el proceso inverso. Partiendo de las palabras objetivo se puede predecir su contexto, obteniendo también de esta forma la representación vectorial. De esta forma opera el modelo **Skip Gram**.

La arquitectura del modelo **Skip Gram** es prácticamente igual a la estructura de CBOW pero invertida. En este caso, las palabras objetivo son introducidas en la red neuronal como entrada. El resultado son C distribuciones de probabilidad. Es decir, para cada palabra de contexto se obtienen a su vez C distribuciones de probabilidad de V elementos, cuyo número coincide con la cantidad de palabras.

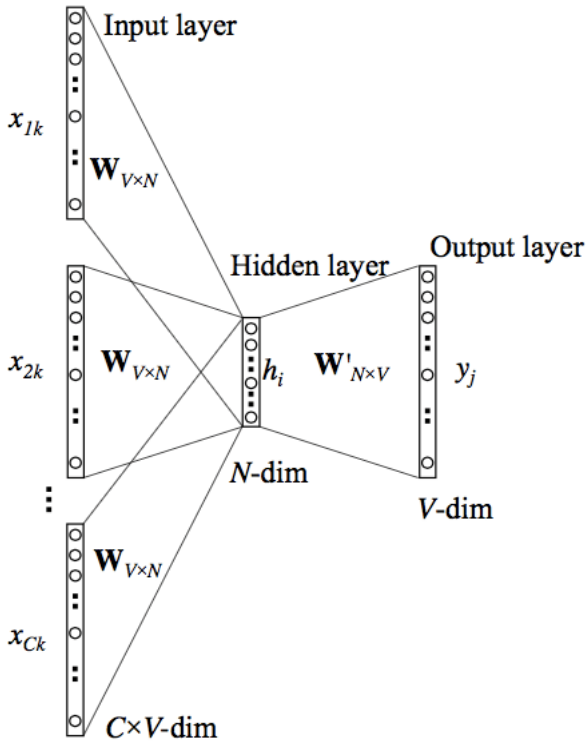

Figura 2.20: Modelo CBOW más complejo [15]

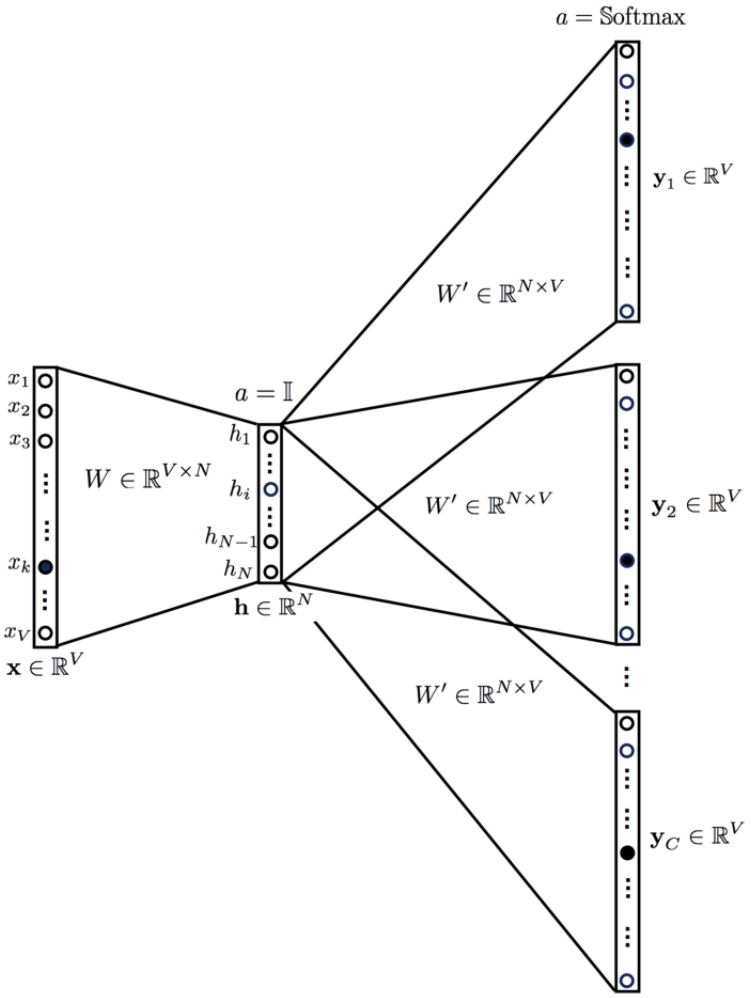

Figura 2.21: Modelo Skip Gram [15]

En ambos métodos se utiliza el algoritmo de backpropagation para entrenar la red neuronal. La base matemática detrás de cada uno de los métodos de word2vec se puede consultar en el artículo original de Mikolov [44]. Otra lectura recomendada es el siguiente artículo de Xin Rong [45].

Tras analizar ambos modelos puede surgir la duda de qué método tiene un mejor desempeño. En este caso los dos cuentan con ventajas y desventajas. Skip Gram funciona bien con un número reducido de datos y se ha observado que puede representar correctamente palabras raras [44]. Por otra parte, CBOW es un procedimiento más rápido y contiene mejores representaciones para palabras más comunes. Por lo tanto, ambos métodos pueden ser útiles en diferentes contextos.

Sin embargo, la representación de word2vec no está exenta de problemas. El problema principal de este método es que proporciona una única representación de cada palabra, ignorando su contexto [16]. De esta forma, palabras polisémicas como "cubo", que puede hacer referencia tanto a *un recipiente esférico con asa* como a *un cuerpo geométrico*, contarán con una representación compuesta por la media de ambos significados.

2.1.5.2 Transformers

Para solventar el problema del contexto, la opción más lógica es utilizar múltiples representaciones para una palabra. Esta es la idea principal detrás de las **context word embeddings**, o representaciones de palabras contextuales. Este concepto es implementado mediante el uso de un modelo de redes neuronales recurrentes (RNN) que es usado para predecir la siguiente palabra de una oración, tomando una palabra como entrada. Este tipo de redes neuronales son usadas debido a su capacidad de almacenar información sobre estados anteriores en sus capas ocultas (ver Fig. 2.22). Tras introducir la palabra actual, se concatena el estado oculto con la representación típica de word2vec para mantener tanto la información sobre la palabra actual como su contexto anterior.

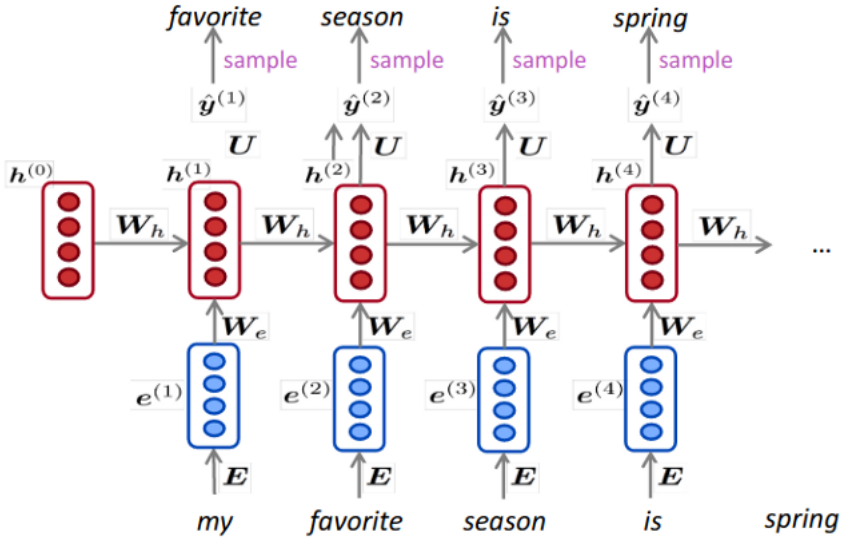

Figura 2.22: RNN donde la capa oculta (en rojo) almacena información de la palabra anterior de la oración [16]

Esta aproximación ha resultado exitosa para obtener una mejor representación contextual en problemas de procesamiento del lenguaje natural (NLP). No obstante, este tipo de modelos, debido a su naturaleza recurrente y secuencial, no pueden ser paralelizados de forma sencilla y, por tanto, su entrenamiento es lento. Por este motivo, en 2017 Aswani y sus compañeros desarrollaron una alternativa basada en una unidad funcional conocida como *Transformer* [16].

Figura 2.23: Ejemplo de atención en una palabra, donde un color más oscuro representa un mayor nivel de atención [17]

La característica principal de un Transformer es el mecanismo de **atención**. La atención captura las relaciones entre palabras de una oración de forma similar al proceso de convolución (ver Fig. 2.23). Al igual que sucede en la convolución, se puden utilizar múltiples cabeza simultáneas para calcular, en cada caso, dónde se debe centrar la atención y la relación que esta representa (ver Fig. 2.24). Donde difiere de la convolución es al capturar la similitud entre palabras en el espacio en el que las matrices de pesos para las diferentes cabezas proyectan sus representaciones. Para capturar relaciones más distantes es posible agrupar varios bloques de transformers.

En el núcleo de un Transformer se encuentra un conjunto de capas de **codificadores** y **decodificadores**. Ambos contienen una capa de entrada para adaptar la entrada que reciben (embedding layer) y una capa para generar la salida final. Todos los codificadores presentes en el modelo son idénticos entre ellos, al igual que sucede con los decodificadores. El codificador está compuesto por una capa de atención propia y una capa prealimentada (Feed Forward). Por otra parte, el decodificador contiene los mismos elementos que un codificador además de una segunda capa de atención. Además cada codificador y decodificador utiliza sus propios pesos. La Fig. 2.25 muestra la estructura final de un Transformer que es obtenida tras un proceso de entrenamiento. Esta es la arquitectura estándar pero también existen otras variaciones de transformers. En algunos casos pueden no existir los decodificadores y solo se trabaja con codificadores.

El funcionamiento al completo de la estructura del Transformer es realmente complejo y excede el alcance de este proyecto. Para una explicación en detalle de este proceso es recomendable consultar la guía creada por Ketan Doshi, disponible de forma online [17].

Esta arquitectura destaca en el manejo de texto secuencial. Toma una cadena de palabras como entrada y produce otra cadena distinta como salida. Por este motivo, los transformers son ampliamente utilizados en aplicaciones de procesamiento de lenguaje natural como la traducción de texto y la creación de chatbots. Un ejemplo actual que está ganando popularidad debido a su buen desempeño es el modelo de lenguaje **GPT-3** desarrollado por la empresa OpenAI en 2020 [46]. Este algoritmo es capaz de autocompletar un texto que es introducido como entrada de la forma más coherente posible. Otro ejemplo más antiguo es el algoritmo **BERT** creado por Google en 2018 [47].

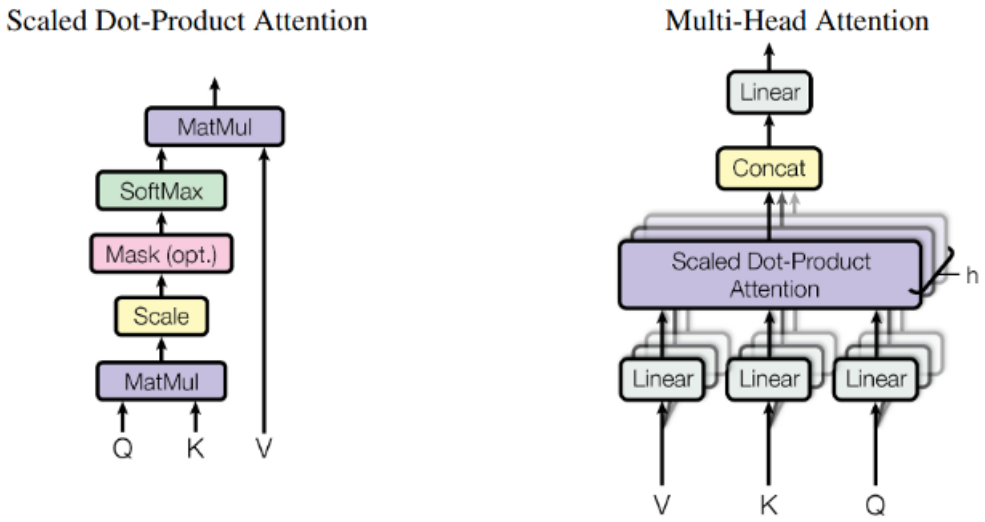

Figura 2.24: Mecanismo de atención [16]

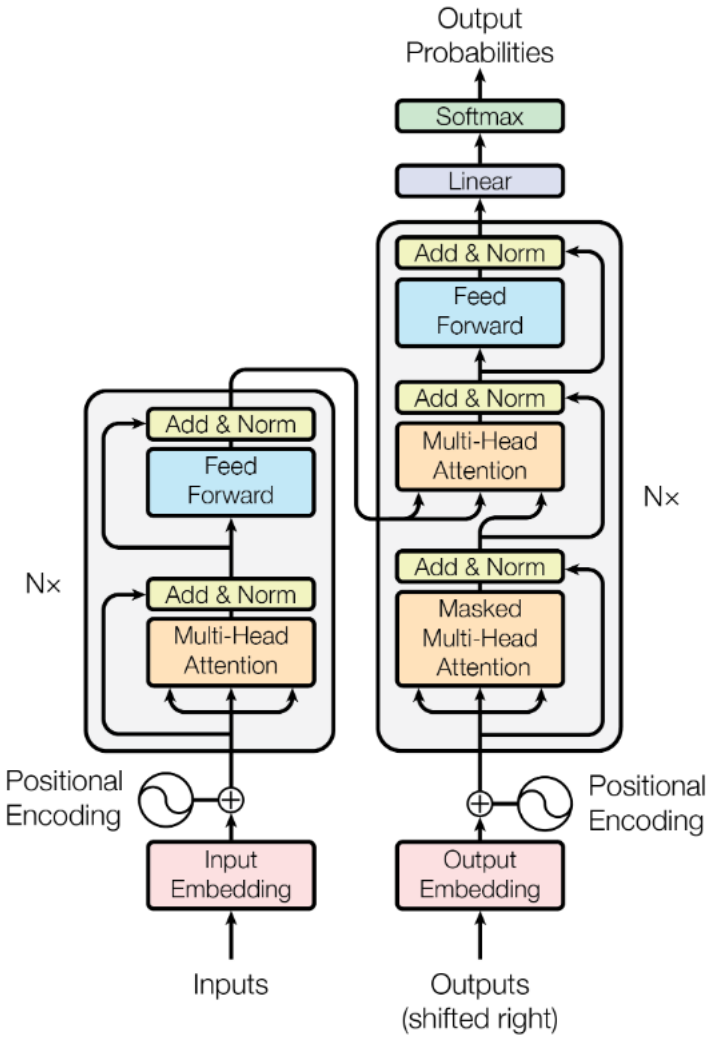

Figura 2.25: Estructura del Transformer [16]

2.2 Métodos de Despliegue

Tras examinar los distintos tipos de algoritmos usados en el campo de la inteligencia artificial, el siguiente paso lógico es analizar qué tipos de despliegue se pueden realizar. El objetivo es comprobar qué características nos ofrece cada solución y cuáles serían las ventajas de decantarse por una opción u otra. Al estudiar el panorama actual podemos destacar las siguientes soluciones:

- Servicios en la nube
- Despliegue local
- Virtualización tradicional
- Containerización

En los siguientes apartados se estudiará caso por caso cada propuesta y se valorarán tanto sus ventajas como sus desventajas desde el contexto de este proyecto. Primero se realizará una exposición de la solución para luego analizar sus características. El objetivo final es contar con una visión general de los distintos métodos de despliegue.

2.2.1 Servicios en la nube

Hoy en día los servicios en la nube ofrecidos por diferentes proveedores se encuentran al alza y cada vez son más usados por distintas empresas. Algunos de los proveedores más conocidos son empresas muy exitosas como Amazon, Google o Microsoft. Dentro de los servicios en la nube, debemos diferenciar entre los 5 grandes grupos dependiendo de qué parte del sistema administra el propio cliente: **IaaS** (*Infrastructure as a Service)*, **CaaS** (*Containers as a Service*), **PaaS** (*Platform as a Service*), **FaaS** (*Function as a Service)* y **SaaS** (*Software as a Service*).

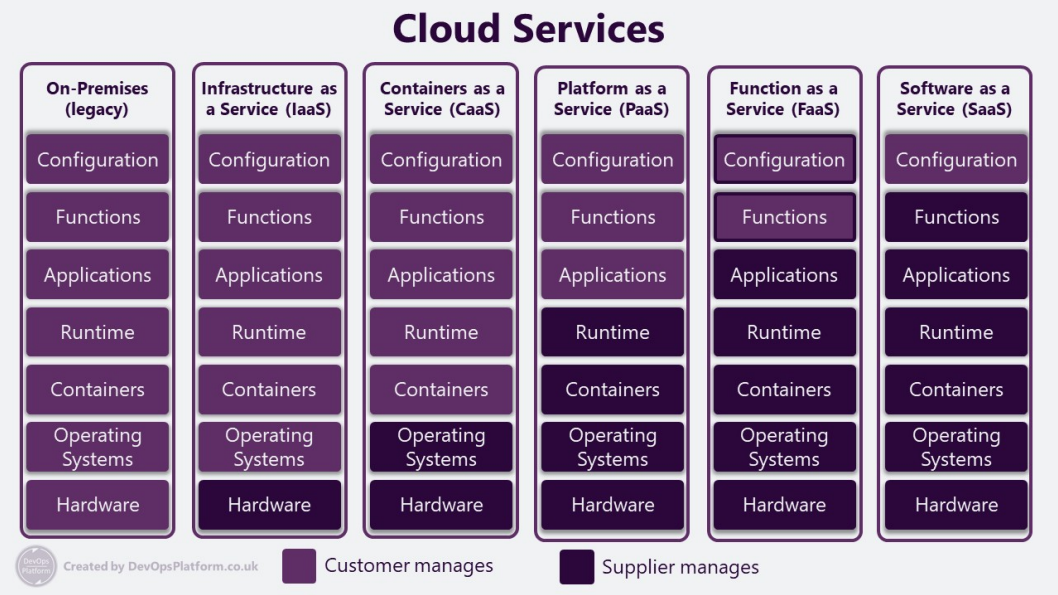

Figura 2.26: Comparación de los diferentes modelos de servicios web [18]

- **IaaS** o **Infraestructura como Servicio**, como su propio nombre indica, se refiere a un tipo de servicio donde el usuario administra toda la infraestructura. Dentro de este esquema se incluyen

tanto las propias aplicaciones y sus datos, como el entorno de ejecución y el sistema operativo. El proveedor proporciona, por lo tanto, la administración de la virtualización, los servidores y la gestión de la red junto con las comunicaciones. El servicio AWS EC2 de Amazon es un ejemplo de este modelo de negocio.

Dado que la configuración de una infraestructura requiere una gran cantidad de tiempo, se originó la metodología conocida como **IaC** o **Infraestructura como Código** [48]. De esta forma, es posible crear y gestionar una infraestructura utilizando código. En vez de realizar una configuración manual, se definen archivos de configuración con las especificaciones de la arquitectura, facilitando la portabilidad del sistema. También es posible dividir el sistema en elementos modulares que se combinarán más tarde con la automatización. Este paradigma permite al desarrollador olvidarse de gestionar manualmente sistemas operativos, entornos de red y almacenamiento cada vez que se aborda el desarrollo de un nuevo proyecto. A la hora de aplicar la IaC, es posible seguir 2 enfoques diferentes: iterativo y declarativo. En el enfoque iterativo es necesario indicar qué comandos específicos son necesarios para obtener la configuración final. Por otra parte, en el declarativo se definen tanto el estado como las propiedades del sistema deseado y la herramienta de IaC se encarga de automatizar el proceso. Ambos enfoques se pueden usar en la mayoría de IaC pero hay algunas herramientas que están más enfocadas a uno de los dos. Algunos ejemplos de este tipo de servicio son Terraform, AWS CloudFormation y Red Hat Ansible Automation Platform.

- **CaaS** o **Contenedores como Servicio**, proporciona una capa extra de abstracción donde es posible administrar contenedores que se ejecutan por encima de un sistema operativo al que el usuario final no tiene acceso. De esta forma se puede contar con todas las ventajas del uso de contenedores pero con una mayor facilidad de uso, ya que el distribuidor del servicio nos proporciona las herramientas de administración. Además cuenta con un precio más atractivo que el servicio anterior por requerir menos. En el caso de Microsoft, la solución ofrece se conoce como Azure Container Instances (ACI).

- **PaaS** o **Plataforma como Servicio**, engloba un servicio en el que el usuario únicamente tiene que mantener su aplicación y los datos, mientras que el resto de los componentes son administrados por el proveedor del servicio. Un ejemplo de esta categoría es Google App Engine.

- **FaaS** o **Función como Servicio**, permite definir métodos que pueden ser ejecutados en respuesta a un evento determinado como la interacción de un usuario con un elemento de una página web. Este modelo brinda una gran flexibilidad y escalabilidad, siendo una solución interesante para implementar una arquitectura de microservicios. AWS Lambda es un ejemplo de este servicio ofrecido por Amazon.

- **SaaS** o **Software como Servicio**, se trata de una aplicación web accesible desde cualquier dispositivo. Algunos ejemplos de este modelo de servicio son las aplicaciones web ofrecidas por Google como Google Drive, Google Docs o Gmail.

La ventaja más comúnmente asociada con los desarrollos en la nube suele ser el coste con respecto a arquitecturas tradicionales. Esto es cierto, en parte, dado que la inversión inicial es considerablemente menor que el coste del hardware necesario y, en algunos casos, sería incluso inviable. Otras ventajas importantes ofrecidas por los servicios en la nube son la escalabilidad, el rendimiento y la ausencia de mantenimiento dado que las máquinas físicas son administradas por el proveedor del servicio.

Sin embargo, desde otro punto de vista este tipo de modelo de despliegue tiene como desventaja la disponibilidad de los recursos. Dependiendo del servicio elegido, es posible que el proveedor tenga

problemas con los equipos o que haya momentos en los que no es posible contar con recursos disponibles para ejecutar nuestro software. Además, es necesario valorar los posibles problemas de seguridad, dado que toda la comunicación con los servidores se realiza a través de internet. También podría darse el caso de que los centros de datos de nuestro proveedor sufran algún tipo de ciberataque que acabe repercutiendo en la usabilidad del servicio.

2.2.2 Despliegue local

El despliegue local es un método de despliegue más tradicional comparado con los servicios en la nube. En este tipo de despliegue el modelo se ejecuta directamente en una única máquina física o en un clúster compuesto por varios nodos físicos. Esta solución es la más usada en escenarios de prototipado y en el mundo de la enseñanza, dado que prácticamente cualquier ordenador convencional se puede usar para este fin. En estos casos el rendimiento no es una prioridad, ya que es más importante entender el funcionamiento y probar cambios de forma local.

Una gran ventaja que aporta esta arquitectura es la flexibilidad de poder construir un sistema a medida. La única limitación sería el presupuesto y el espacio donde alojar todos los componentes del sistema. Dependiendo del tipo de algoritmo ejecutado, es posible elegir la configuración de hardware óptima para su ejecución. Por ejemplo, puede ser que un modelo se beneficie la ejecución multihilo, por lo que sería recomendable utilizar un procesador con el máximo número de núcleos que sea posible. Además, al no ser un recurso compartido en la nube, no sería necesario preocuparse del tiempo de uso asignado a la ejecución del código.

Por otra parte, es necesario analizar el aspecto de la seguridad desde diferentes ángulos. Al encontrarse todos los recursos físicos en el mismo sitio, es más fácil cumplir con la seguridad deseada dado que hay menos partes involucradas y la interconexión de los diferentes equipos se realiza a través de una red local. Sin embargo, en la mayoría de los casos el sistema necesita una conexión a internet si el objetivo final es atender peticiones externas, por lo que seguiría existiendo una posible fuente de vulnerabilidades. Además, al ser un sistema centralizado, si se produce un ciberataque, será más difícil recuperar la normalidad y es posible que exista pérdida de datos.

Otra desventaja clara de este modelo de despliegue es el mantenimiento. Al ser una infraestructura administrada por el usuario en todos los niveles, es necesario, tanto realizar un mantenimiento regular del hardware, como comprobar que todo el software esté actualizado a la última versión para evitar posibles vulnerabilidades. Además, es posible que al actualizar existan problemas de compatibilidades. En este caso sería necesario valorar si es preferible mantener la versión actual, con el riesgo de seguridad que esto conlleva, o actualizar y arreglar los problemas de compatibilidades, produciendo un aumento de la carga de trabajo. Además, es posible que el entorno de desarrollo difiera del entorno de producción, por lo que podrían aparecer nuevos problemas que deberán ser subsanados. Tras analizar todos estos aspectos, es fácil observar que el coste del proyecto puede dispararse rápidamente tanto por el coste del hardware necesario, como por el trabajo extra que debe invertirse en mantenimiento. Además, si en algún momento es necesario escalar el sistema o se necesita un mayor rendimiento, el coste también aumentaría considerablemente.

2.2.3 Virtualización tradicional

Partiendo de la base de un despliegue local, es posible seguir un enfoque diferente. Otra posible opción es configurar máquinas virtuales sobre el host en vez de limitarse al sistema operativo ejecutado por el

equipo. Sin embargo, antes de estudiar qué características aporta esta tecnología, es necesario comprender el concepto de máquina virtual y la tecnología en la que se apoya para su funcionamiento.

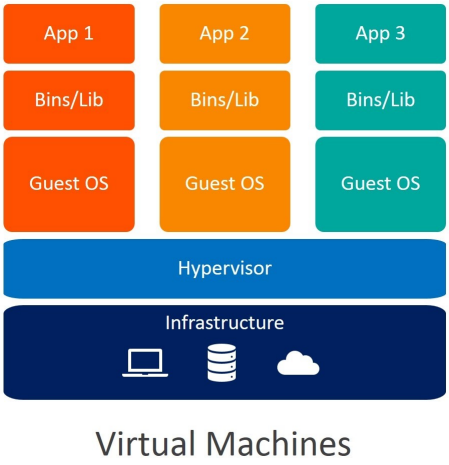

Figura 2.27: Arquitectura de una máquina virtual [19]

¿Pero qué es una máquina virtual? Según la empresa RedHat [49], *"Una máquina virtual (VM) es un entorno que funciona como un sistema informático virtual con su propia CPU, memoria, interfaz de red y almacenamiento, pero el cual se crea en un sistema de hardware físico, ya sea on-premise o no (Fig 2.27). El sistema de software se llama hipervisor, y se encarga de separar los recursos de la máquina del sistema de hardware e implementarlos adecuadamente para que la VM pueda utilizarlos"*. El hipervisor o monitor de máquina virtual (VMM) permite tanto crear como ejecutar y gestionar máquinas virtuales. Existen 2 tipos de hipervisor dependiendo del nivel al que ejecuta este software, Fig. 2.28.

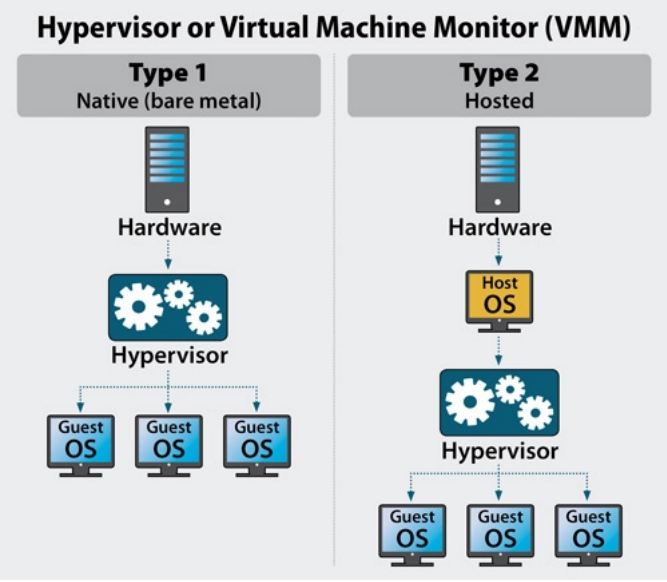

Figura 2.28: Tipos de hipervisores [20]

El **tipo 1**, o hipervisor nativo, se ejecuta directamente sobre el hardware, reemplazando al sistema operativo del equipo o "host" y gestionando los sistemas operativos de las máquinas virtuales "guest". Este tipo de hipervisor consume menos recursos y, por lo tanto, proporciona un mayor rendimiento. Además, una máquina virtual puede acceder de forma directa a dispositivos del equipo como periféricos, tarjetas gráficas, memoria externa, etc. Esta característica se conoce como "device passthrough" (Fig.

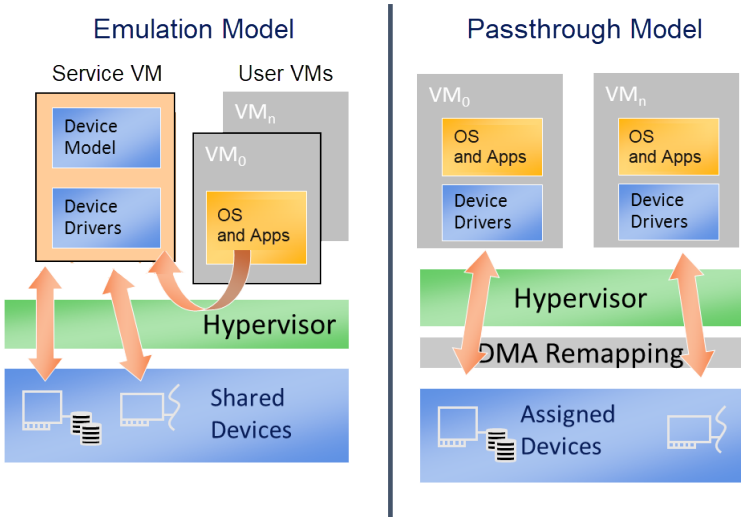

Figura 2.29: Device passthrough [21]

2.29) y permite utilizar hardware sin apenas pérdida de rendimiento con respecto a la ejecución nativa. No obstante, los dispositivos que han sido asignados a una máquina virtual no pueden ser compartidos por otras máquinas. Este tipo de hipervisor es el más usado en servidores y centros de datos empresariales. Algunos ejemplos son KVM y Microsoft Hyper-V. KVM se fusionó con el kernel de Linux en 2007, por lo que en distribuciones modernas se puede configurar sin necesidad de instalar software adicional.

Por otra parte, el hipervisor de **tipo 2** o alojado se ejecuta como una aplicación más del sistema operativo. Los recursos disponibles son asignados desde el host y más tarde se ejecutan en el hardware real. Este tipo de hipervisor es más utilizado por usuarios finales y desarrolladores dada su facilidad de uso. Algunas de sus aplicaciones más comunes pueden ser probar otros sistemas operativos, utilizar software que solo está disponible en otros sistemas o probar el comportamiento de una aplicación en diferentes entornos. Oracle VirtualBox y VMware Workstation son ejemplos de este tipo de hipervisor.

Una vez se ha definido qué es una máquina virtual, es posible examinar qué ventajas aporta este modelo de despliegue. En primer lugar, esta tecnología permite ejecutar diferentes sistemas operativos en la misma máquina física, lo cual puede ser útil para algunas aplicaciones. Además, es posible compartir una imagen de un sistema virtualizado, por lo que el entorno que se ha usado durante el desarrollo puede ser idéntico al entorno de despliegue, evitando así problemas de compatibilidades. Otra ventaja clara de este tipo de arquitectura es una mayor seguridad del sistema dado que todo el software se ejecuta en un entorno aislado del sistema operativo del anfitrión y del hardware.

No obstante, este modelo de despliegue no está exento de inconvenientes. La mayor desventaja de los sistemas virtualizados es un menor rendimiento con respecto a software corriendo de forma nativa. Comparado con la ejecución típica, en un sistema virtualizado el hardware debe soportar la carga añadida del hipervisor encargado de administrar las máquinas virtuales junto con las propias máquinas virtuales instanciadas en este momento. Además, en la máquinas virtuales se encuentran las dependencias necesarias y la propia aplicación que se está ejecutando. Lógicamente, todo esto se traduce en una carga de trabajo mayor para el ordenador. Por otra parte, aunque no hay diferencia entre el entorno de desarrollo y el entorno de producción, el mantenimiento y el coste general del sistema aumentan. Esto es lógico, dado que es necesario invertir más horas de trabajo en la configuración de las máquinas virtuales y en comprobar que todo el sistema funcione correctamente, siendo necesaria la experiencia en entornos virtualizados.

Por último, es necesario recalcar que estas características pueden variar dependiendo del tipo de hipervisor utilizado. Además, también influye el entorno donde se ejecuten, dado que es posible tanto configurar máquinas virtuales en una máquina local como utilizar algún tipo de servicio en la nube.

2.2.4 Containerización

Las máquinas virtuales no son la única tecnología de virtualización existente actualmente. Otra opción interesante que está cobrando popularidad en los últimos años son los contenedores, pero ¿qué es un contenedor? La containerización es una tecnología de virtualización a nivel de sistema operativo, a diferencia de una máquina virtual que implementa virtualización de hardware [50]. Un único contenedor puede ser usado para ejecutar desde un pequeño microservicio o proceso a una aplicación más compleja. Dentro de este contenedor se encuentran todos los ejecutables, binarios, librerías y archivos de configuración necesarios para su ejecución. De forma similar al modelo IaC visto en la sección 2.2.1, un contenedor se crea a partir de una imagen en la que se definen todos los componentes necesarios. Esta característica proporciona una gran portabilidad. En la siguiente figura se compara la estructura de los contenedores con la virtualización tradicional:

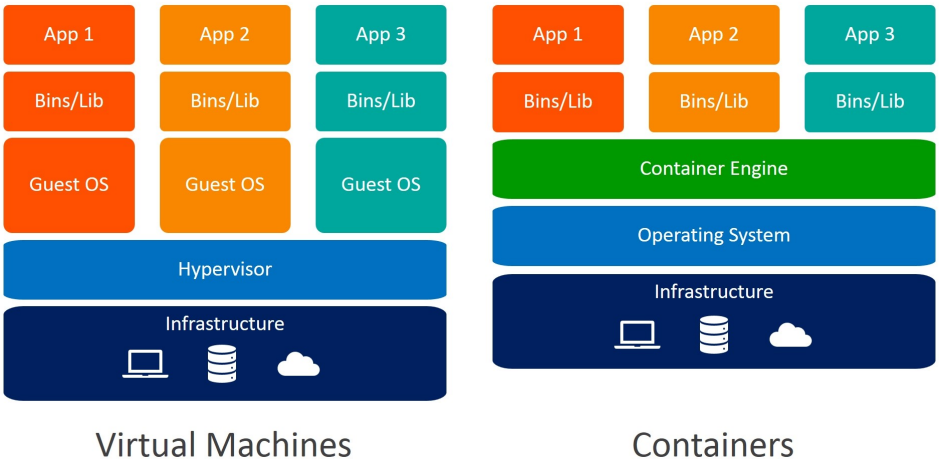

Figura 2.30: Representación de contenedores y máquinas virtuales [19]

Un contenedor, por lo tanto, no necesita virtualizar un sistema operativo completo para cada instancia como sucede en una máquina virtual. Esto es posible dado que todos los contenedores que se encuentran en ejecución comparten el mismo kernel de Linux, siendo el kernel la parte del sistema operativo que permite al software acceder de forma segura al hardware del dispositivo. Si el sistema operativo anfitrión no se trata de GNU/Linux, es posible ejecutar una máquina virtual ligera en segundo plano para suplir esta carencia, sin perjudicar de forma significativa el rendimiento de los contenedores.

No obstante, un contenedor no serviría de nada si no existiera una forma de ejecutarlo y administrarlo. Para ello, es necesario utilizar un software conocido como *runtime* que se encarga de llamar a los componentes de bajo nivel necesarios para la ejecución de contenedores [51]. El runtime utiliza las funciones de namespaces y cgroups proporcionadas por el kernel de Linux para realizar esta tarea. Los namespaces se utilizan para aislar procesos mientras que los cgroups sirven para asignar una cantidad limitada de recursos a estos procesos, por lo que ambas utilidades son necesarias para conseguir ejecutar un contenedor [52]. Existen varios runtimes con diferentes niveles de complejidad como LXC o containerd, siendo este último un runtime de alto nivel que a su vez realiza llamadas al runtime de bajo nivel runC. Por encima del runtime se encuentran otras herramientas de mayor nivel como Docker. Este software es

un conjunto de herramientas de código abierto que añade más funcionalidad a containerd para facilitar la administración de contenedores como se muestra en la Fig. 2.31. Containerd se encontraba integrado antiguamente dentro del proyecto de Docker pero se separó para hacer la plataforma más modular. Pese a que existen otras opciones, para el desarrollo de este proyecto se ha decidido enfocarse en Docker por gozar de una amplia popularidad actualmente y contar, por lo tanto, con una gran cantidad de documentación.

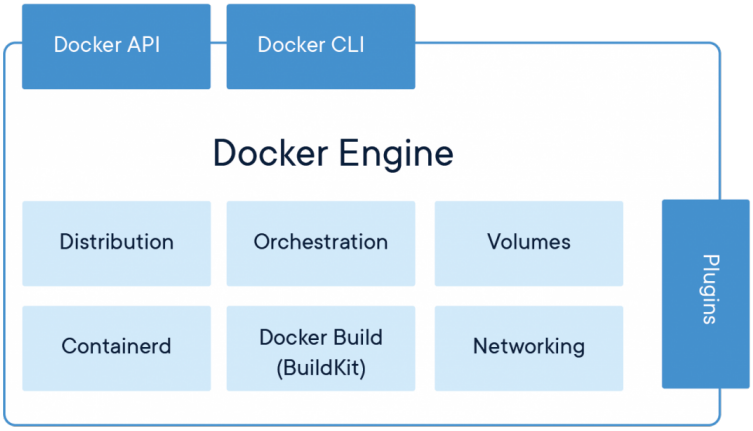

Figura 2.31: Componentes de Docker [22]

Aun así, en despliegues de aplicaciones más grandes es posible que se utilice una gran cantidad de contenedores alojados en uno o varios clústeres. En este caso, dada la mayor complejidad de control y monitorización de los clústeres es posible utilizar un orquestador como Kubernetes. Este software se puede utilizar para organizar el trabajo de múltiples contenedores, siendo capaz de administrar recursos como la capacidad de cómputo, las redes y el almacenamiento. Otra forma de entender cómo funciona Kubernetes es analizarlo como una plataforma de microservicios [53]. En este tipo de arquitectura, las diversas tareas del sistema están repartidas en módulos y cada uno de estos elementos se encarga de realizar una única función. Los orquestadores como Kubernetes permiten automatizar y escalar flujos de trabajo basados en contenedores para ser usados en producción.

Tras comprender mejor esta tecnología, se analizarán qué características aporta como modelo de despliegue. Para empezar, comparado una máquina virtual tradicional, los contenedores aportan un mayor rendimiento comparable al de una ejecución nativa. Además, dado que el software se ejecuta en un entorno aislado de la máquina local como sucedía en una máquina virtual, se obtiene una mayor seguridad comparado con un despliegue local. Otra ventaja compartida con una máquina virtual es el mismo entorno de desarrollo y despliegue. Esta tecnología nos permite crear imágenes que incluyen todas las dependencias usadas por la aplicación desarrollada y pueden ser compartidas entre diferentes máquinas. Este modelo de despliegue también permite configurar un clúster compuesto por múltiples nodos físicos y desplegar contenedores de forma homogénea. De esta forma, resulta sencillo escalar una aplicación dado que es posible configurar un balanceador de carga de forma automática.

Aún así, es necesario tener en cuenta también las desventajas de este sistema. Antes de poder utilizar contenedores en un proyecto es necesario planificar cómo se realizará el despliegue y cómo se administrarán los contenedores una vez desplegados. Si simplemente se utilizarán un par de contenedores y la comunicación entre ellos es simple o inexistente con usar las herramientas integradas de serie en Docker puede ser suficiente. Aun así, para proyectos más grandes y complejos es conveniente usar soluciones como Kubernetes dado que permite realizar un control a alto nivel y es más fácil configurar un clúster de contenedores que colaboran entre ellos.

También es necesario adaptar la aplicación desarrollada para aprovechar este modelo de ejecución y crear las imágenes necesarias incluyendo todas las dependencias del proyecto. Junto a la configuración inicial, también se incrementa la complejidad de monitorizar la aplicación durante su despliegue junto con el mantenimiento del sistema. Por lo tanto, el coste y la mano de obra necesaria para desarrollar el proyecto crece en la misma medida.

Junto a las características comentadas anteriormente es importante tener en cuenta que, dependiendo de si los contenedores se despliegan sobre una máquina local o sobre un servicio en la nube, existirán las mismas ventajas y desventajas que en estos modelos de despliegue expuestos en secciones anteriores.

2.3 Conclusiones

Tras analizar diferentes soluciones de despliegue podemos observar que cada opción puede resultar atractiva para distintos tipos de proyectos. La elección de un modelo u otro dependerá de aspectos como la escalabilidad o aquellos que sean más prioritarios en cada caso. Para el desarrollo de este proyecto se ha decidido utilizar contenedores con el objetivo de demostrar que es una opción viable para el despliegue de algoritmos de inteligencia artificial. La ventaja principal de este modelo de despliegue radica en la facilidad de compartir una imagen que encapsula la aplicación y sus dependencias. De esta forma se puede compartir fácilmente el entorno de ejecución entre diferentes máquinas y es una característica que se puede aprovechar especialmente al migrar del entorno de desarrollo al entorno de despliegue. Esta propiedad junto a la facilidad de escalado del sistema, resultan muy interesantes.

Capítulo 3

Desarrollo del proyecto

3.1 Introducción

Tras conocer mejor el estado del arte de algunos tipos de algoritmos de IA y habiendo realizado una exposición de diferentes métodos de despliegue, es posible continuar con la siguiente fase del proyecto. En esta sección se realizará un ejemplo de pipeline de IA utilizando la herramienta Kubeflow. Antes de llegar a ese punto, se llevará a cabo la instalación del entorno de trabajo junto con la creación de varios experimentos para familiarizarse con el sistema. Además, se explorarán casos con diferentes algoritmos de IA para destacar distintos casos de uso.

El objetivo principal de este proceso es confeccionar una hoja de ruta para desplegar aplicaciones basadas en IA utilizando Kubeflow. Para ello, el trabajo de dividirá en múltiples fases. En primer lugar, se realizará la instalación de todo el software necesario y la configuración del entorno de desarrollo. Más adelante, se realizarán varios experimentos con distintos tipos de algoritmos de IA. La finalidad de esta fase es tanto familiarizarse con el entorno de Kubeflow como analizar las necesidades de cada tipo de algoritmo. Finalmente, se explorará la configuración de un sistema de CI/CD para el montaje del pipeline final. El desglose de las diferentes fases, por tanto, sería el siguiente:

- Instalación y configuración del entorno
- Aprendizaje de Kubeflow
  - Explorando los pipelines
  - Algoritmos de IA
- CI/CD
- Arquitectura final

Tras realizar un análisis de las diferentes técnicas de virtualización del sector y una vez vistas las diferentes familias de algoritmos de IA, es posible comenzar la parte práctica del proyecto. Para el desarrollo de esta sección se ha decidido, por tanto, utilizar Kubeflow como base para la orquestación de contenedores en los diferentes pipelines. Existen múltiples formas de instalación de Kubeflow, en este caso, y tras probar con diferentes opciones, se ha decidido utilizar MiniKF.

3.2 Instalación y configuración del entorno

Kubeflow es un conjunto de herramientas de software basado en Kubernetes y desarrollado para simplificar flujos de trabajo en machine learning. Otros objetivos de Kubeflow son la portabilidad y escalabilidad. Entre estas herramientas se encuentran utilidades para crear y manejar cuadernos de Jupyter, el servicio de modelos de Tensorflow o Kubeflow Pipelines [54]. Esta última funcionalidad, será ampliamente utilizada durante el desarrollo de este proyecto. Kubeflow Pipelines se compone de:

- Una interfaz de usuario para gestionar y monitorizar experimentos, trabajos y ejecuciones.
- Un motor para planificar flujos de ML de múltiples pasos.
- Un SDK para definir y manipular tanto pipelines como componentes.
- Cuadernos para interactuar con el sistema utilizando el SDK.

Por otra parte, los objetivos de esta herramienta son:

- Orquestación de extremo a extremo, habilitando y simplificando la orquestación de flujos de ML.
- Facilidad de experimentación, permitiendo la prueba de diferentes técnicas y la gestión de los distintos experimentos de forma sencilla.
- Reutilización de componentes y pipelines para recrear soluciones sin necesidad de reconstruir el flujo entero continuamente.
- Cuadernos para interactuar con el sistema utilizando el SDK

Estas características resultan muy útiles para el proyecto y serán exploradas en más detalle en sucesivas secciones.

Existen dos opciones principales para la instalación del entorno de Kubeflow: el despliegue local y el uso de un servicio en la nube. En una instalación local es necesario instalar Kubernetes junto con la herramienta kubectl para gestionar los diversos clusters de contenedores y añadir más tarde el módulo de Kubeflow. No obstante, en vez de instalar la distribución estándar de Kubernetes también sería posible utilizar Minikube para simplificar el proceso. Minikube es una herramienta de código abierto que ejecuta un único cluster de Kubernetes de forma local con un bajo consumo de recursos. Para ello, puede utilizar diversas técnicas de virtualización como contenedores o hipervisores de máquinas virtuales (Docker, KVM, VirtualBox, etc) [55]. En resumen, Minikube empaqueta y configura una máquina virtual de Linux que cuenta con Docker y todos los componentes de Kubernetes. Además, esta opción seguiría teniendo cierta complejidad dado que es necesario añadir Kubeflow posteriormente y pueden surgir múltiples problemas de configuración. Además, este tipo de despliegue local no recibe soporte oficial actualmente [56] por lo que sería más difícil solucionar los posibles errores. Por este motivo, esta opción no es la más recomendable. Ampliando lo visto en Minikube, se encuentra la herramienta MiniKF creada por la empresa Arrikto. MiniKF es una herramienta que empaqueta Minikube junto a Kubeflow en una única VM gestionada utilizando Vagrant, siendo este último un software utilizado para la creación y la gestión de máquinas virtuales. De esta forma, se elimina la necesidad de instalación de Kubeflow sobre Kubernetes y se simplifica todo el proceso. La principal desventaja de una instalación local es el gran coste computacional y de memoria para una correcta ejecución. En el caso de MiniKF, se recomiendan 12GB de RAM, una CPU de, al menos, 2 núcleos y 50GB de almacenamiento [57]. Otra opción igualmente válida es el uso de un servicio en la nube como Google Kubernetes Engine (GKE) o Amazon Elastic Kubernetes Service

(EKS). Esta opción sí cuenta con soporte oficial y existen múltiples artículos que detallan su uso. No obstante, para este proyecto se decidió no optar por este camino por ser necesaria una suscripción de pago a la plataforma y querer contar con un mayor control sobre el sistema desarrollado. Por tanto, la opción final ha sido MiniKF.

La instalación de MiniKF es sencilla comparada con el resto de opciones mencionadas anteriormente. Para ello se ha seguido la guía presente en la documentación oficial de Kubeflow [57]. En primer lugar y antes de comenzar con la instalación, es necesario instalar tanto Vagrant como Virtualbox. En el equipo usado para el desarrollo de este proyecto se encuentra instalada la distribución Pop!_OS, basada en Ubuntu. Por lo tanto, en todas las guías se seguirá el apartado dedicado a Ubuntu o Debian. Para instalar Vagrant es necesario añadir su repositorio e instalar el software utilizando el gestor de paquetes de Debian mediante los siguientes comandos:

```
$ curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
$ sudo apt-add-repository "deb [arch=amd64] \
  https://apt.releases.hashicorp.com $(lsb_release -cs) main"
$ sudo apt-get update && sudo apt-get install vagrant
```

En el caso de Virtualbox es posible descargar el paquete .deb desde su página oficial. Una vez ambos requisitos se encuentran presentes es posible instalar y ejecutar MiniKF con las instrucciones:

```
$ vagrant init arrikto/minikf
$ vagrant up
```

Tras esto, MiniKF tardará algunos minutos en arrancar. Cuando finalice el proceso, utilizando el navegador web es posible acceder a la url ***http://10.10.10.10*** y seguir las instrucciones que aparecen en la página para iniciar Kubeflow 3.1. Este proceso se demorará más tiempo que el arranque de MiniKF dado que se deben arrancar todos los contenedores de los servicios de Kubeflow y puede llegar a tardar hasta 30 minutos dependiendo de la potencia del equipo.

Una vez se finalice el despliegue de Kubeflow se mostrarán en pantalla las credenciales necesarias para acceder a la interfaz web 3.2. Tras ingresar las credenciales correctas se accederá a la página principal de Kubeflow 3.3. Tras terminar todo el trabajo, es posible detener MiniKF antes de apagar el equipo mediante la orden:

```
$ vagrant halt
```

3.3 Aprendizaje de Kubeflow

Una vez la instalación de Kubeflow se encuentra funcionando correctamente, es posible pasar a la siguiente fase. Antes del montaje del pipeline final, es necesario estar familiarizado con el software. Kubeflow puede ser usado, tanto utilizando la interfaz gráfica que es accesible navegando a la página web de inicio, como utilizando la API disponible para Python. En esta sección se comenzará empleando la interfaz gráfica para asimilar gradualmente los conceptos básicos de Kubeflow como pipelines, experimentos, buckets, instantáneas, etc. Más tarde, durante la fase de automatización, se hará uso de la API para la publicación de las pipelines actualizadas sin intervención humana. Este proceso será subdividido en dos fases. En la primera fase, se explorarán las capacidades de Kubeflow progresivamente para familiarizarse con su interfaz y sus diferentes componentes. Más tarde, se ejecutarán varios algoritmos de IA aprovechando la utilidad de pipelines de Kubeflow y se analizarán las necesidades de cada uno de ellos.

Figura 3.1: Pantalla de bienvenida MiniKF

3.3.1 Explorando Kubeflow

Antes de realizar casos más avanzados es necesario conocer las bases de Kubeflow. Para ello, se ha tomado como base el codelab de Google llamado **"From Notebook to Kubeflow Pipelines with HP Tuning: A Data Science Journey"** [58]. En este curso se muestra cómo construir un pipeline de ciencia de datos con ajuste automático de hiperparámetros utilizando la interfaz gráfica de Kubeflow. Las utilidades que se usarán para ello son: Kubeflow Pipelines, Rok, Kale y Katib.

Kubeflow Pipelines (KFP) [54] es una plataforma diseñada para construir y diseñar flujos de trabajo de ML basados en contenedores. Es la base de Kubeflow.

Rok [59] es una capa de almacenamiento y gestión de datos utilizada para compartir recursos entre diferentes aplicaciones. El principal uso de Rok es la creación de instantáneas de una aplicación junto con sus datos y su distribución de forma eficiente. Este intercambio de datos puede ocurrir entre nodos del mismo clúster o entre dispositivos externos. Esta utilidad ha sido desarrollada por la empresa Arrikto.

Kale [54] se trata de una extensión de JupyterLab que simplifica a los científicos de datos la orquestación de flujos de trabajo de ML. Utilizando la interfaz de Kale es posible anotar celdas en un cuaderno de Jupyter para definir pasos del pipeline, ajuste de hiperparámetros, uso de la GPU y seguimiento de

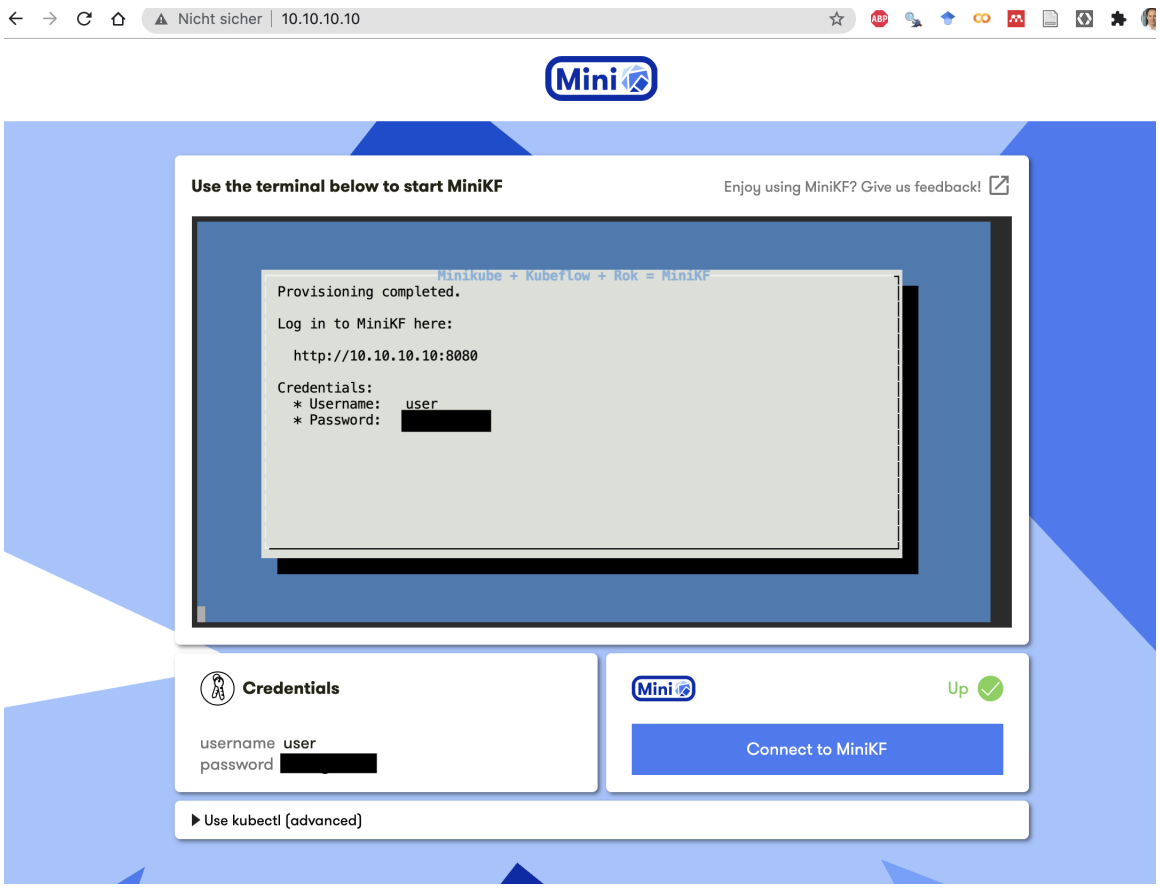

Figura 3.2: Credenciales de Kubeflow

métricas. Con la pulsación de un botón permite crear automáticamente componentes del pipeline adaptados al lenguaje de Kubeflow Pipelines, resolver dependecias, inyectar datos en cada paso, desplegar el pipeline final y servir el mejor modelo.

Katib [60] es una herramienta de aprendizaje automático (AutoML) que permite el ajuste automático de hiperparámetros. Los hiperparámetros son variables que controlan proceso de entrenamiento del modelo. Algunos ejemplos son: la tasa de aprendizaje, el número de capas de una red neuronal, el número de nodos de cada capa, etc. Estos valores no son *aprendidos*, es decir, el proceso de entrenamiento no ajusta estas variables. El ajuste de hiperparámetros es, por tanto, el proceso de optimización de sus valores para maximizar la precisión predictiva del modelo. Katib elimina la necesidad de ejecutar múltiples entrenamientos y ajustar manualmente los hiperparámetros hasta encontrar los valores óptimos. Además, Katib soporta la detención temprana y la búsqueda de arquitectura neuronal (NAS). Esta herramienta es independiente de los frameworks de ML utilizados y es capaz de ajustar parámetros de aplicaciones escritas en cualquier lenguaje de programación. Entre los algoritmos de ajuste soportados se encuentran algunos como la optimización Bayesiana o la búsqueda aleatoria.

El primer paso será crear un servidor de JupyterLab. **Jupyter** es un entorno de desarrollo que permite combinar código, material audiovisual y texto en documentos denominados cuadernos. Jupyter soporta múltiples lenguajes de programación y es una herramienta muy utilizada en el mundo científico para intercambiar experimentos, ya que permite reproducir los resultados fácilmente. En el ámbito del machine learning es usado para probar y compartir modelos rápidamente. Para crear un servidor de Jupyter en Kubeflow, es necesario acceder a la sección ***Notebooks*** del menú izquierdo.

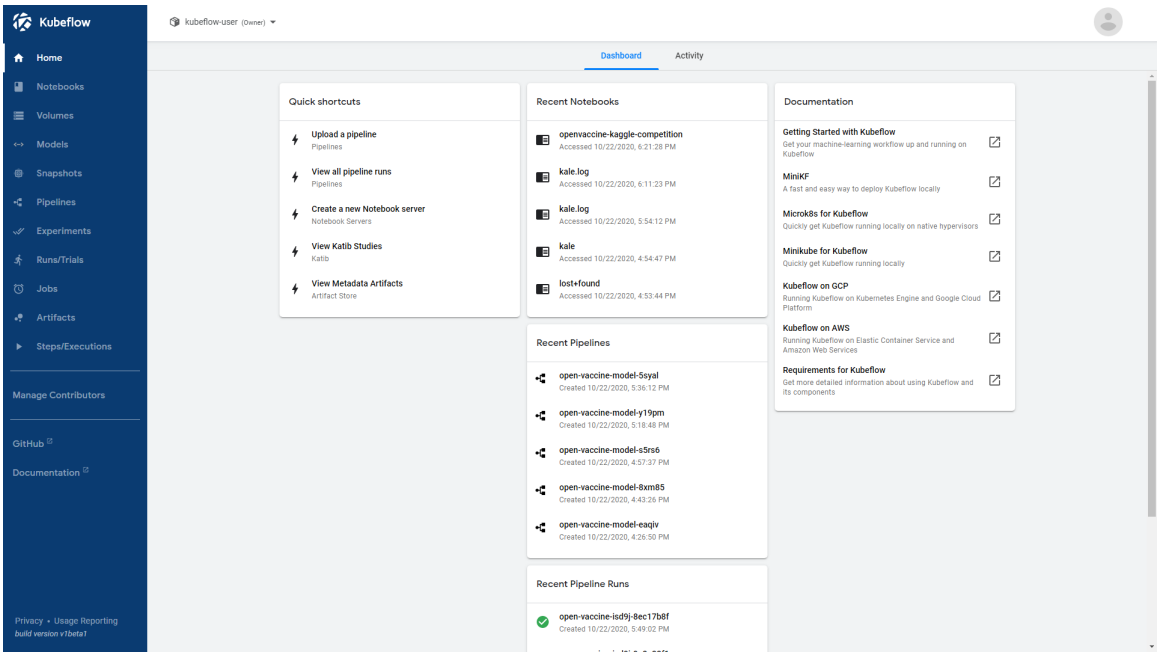

Figura 3.3: Interfaz de Kubeflow

Una vez aquí, seleccionamos la opción de "New Server". Tras esto, aparecerá un asistente para configurar los ajustes del servidor de Jupyter. Aquí especificaremos el nombre del servidor y el namespace al que pertenecerá. En este caso se usará el namespace por defecto. También es posible indicar qué imagen de Docker se quiere usar para el servidor. Esta imagen puede ser tanto una de las proporcionadas por Kubeflow como una imagen personalizada que se encuentre disponible en Docker Hub. En este caso se usará la imagen:

```
gcr.io/arrikto/jupyter-kale:f20978e
```

Es posible modificar los recursos que serán asignados al servidor, tanto núcleos de procesador como la cantidad de RAM disponible e incluso habitar la GPU si fuera necesario. Para el almacenamiento se podrá especificar la cantidad deseada y también se permite crear un volumen virtual que será montado junto a este. En este caso, añadiremos un volumen llamado *data* de 5GB para almacenar el dataset del ejemplo. Tras terminar toda la configuración necesaria, se lanzará el servidor.

Una vez el servidor se encuentre en ejecución, se podrá acceder a la instancia de JupyterLab. JupyterLab es...

Una vez tengamos acceso al cuaderno el primer paso será descargar los datos necesarios para el desarrollo del ejemplo. Para ello, se abrirá una terminal y se descargarán los datos utilizando la siguiente instrucción:

```
git clone https://github.com/kubeflow-kale/kale
```

De esta forma, se clonará el repositorio de git que contiene todos los elementos necesarios. El cuaderno de este ejemplo se encuentra en la ruta *data/kale/examples/dog-breed-classification/* y tiene el nombre *dog-breed.ipynb*. Al abrir el cuaderno podemos explorar las diferentes celdas de las que está compuesto. La primera celda con la que nos encontramos contiene todas las dependencias necesarias. Normalmente, estas dependencias se incluyen en la imagen de Docker que usaremos, sin embargo, gracias al uso de Rok y Kale, cualquier librería que se instale durante el desarrollo se incluirá en el pipeline final. Esto es posible mediante el uso de instantáneas de Rok que son más tarde montadas en los pasos del pipeline por Kale. Todo este proceso es trasparente al usuario final.

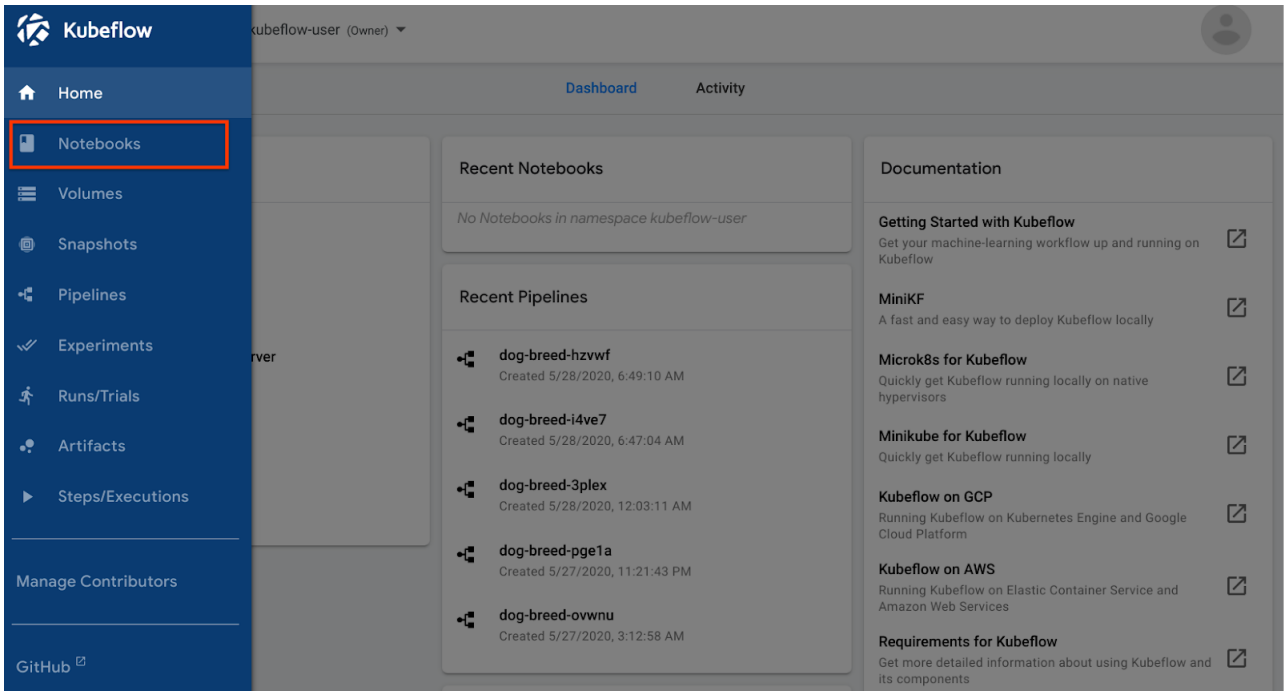

Figura 3.4: Notebooks

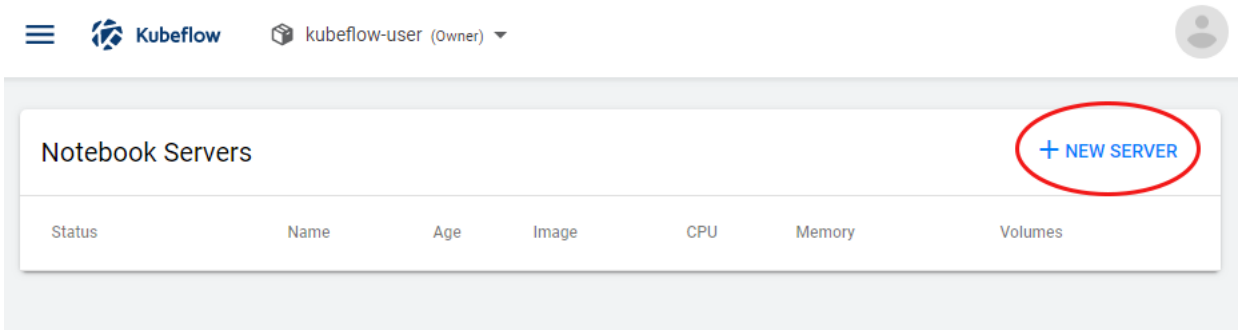

Figura 3.5: Notebooks

El siguiente paso es convertir el cuaderno en un pipeline. Para ello, usaremos la herramienta Kale. Como ya fue comentado anteriormente, Kale nos proporciona la capacidad de generar un pipeline partiendo de un cuaderno de Jupyter utilizando diferentes anotaciones. Echando un ojo al cuaderno podemos comprobar los diferentes tipos de celdas que se han usado en este caso. Algunas de ellas, como la celda de *imports*, son estándar mientras que otras celdas son personalizadas. Para añadir una nueva anotación es necesario definir un nombre para el paso y especificar qué pasos anteriores son necesarios para la ejecución del actual. Kale se encargará de conectar las entradas y salidas de los dos pasos. Una vez todos los pasos del pipeline se encuentran definidos es posible la compilación del pipeline. Para ellos, basta con seleccionar la opción de Kale en el menú izquierdo y pulsar la opción de *Compile Pipeline*. Tras esto, se creará el pipeline y se ejecutará un experimento. Podemos observar el progreso pulsando el enlace que aparece. Una vez acabe la ejecución, es posible comprobar el resultado final y depurar si se ha producido algún error.

Tras conocer el proceso de creación de un pipeline partiendo de un cuaderno de Jupyter hay otro aspecto que no ha sido tratado todavía. Muchos algoritmos de IA contienen ciertos valores utilizados durante el proceso de entrenamiento denominados hiperparámetros, tal y como se comentó anteriormente. Estos valores no pueden ser deducidos de los datos y, por tanto, deben ser especificados por el científico

de datos. En lugar de optimizar estos parámetros mediante prueba y error, la herramienta Katib puede simplificar esta tarea. Para realizar esta prueba usaremos otro cuaderno de Jupyter llamado *dog-breed-katib.ipynb* y localizado en la misma ruta que el anterior. Para que Katib pueda ajustar el valor de los hiperparámetros es necesario añadir dos nuevas anotaciones al cuaderno. La primera anotación necesaria es *pipeline-parameters* y es usada para definir qué variables se tomarán como hiperparámetros. La segunda se utilizará para especificar qué valor se quiere maximizar o minimizar y se denomina *pipeline-metrics*. Una vez se han definido ambas celdas es posible iniciar la tarea de optimización activando la opción correspondiente en el menú de Kale. Este menú se encuentra en la parte izquierda de la pantalla y la opción para activar Katib aparece como *"HP tuning with Katib"*. Al marcar esta opción aparecerá un botón con el texto *SET UP KATIB JOB* y, al presionarlo, aparecerá una interfaz para definir las características de la tarea. En este apartado se pueden configurar aspectos como el tipo de las variables tomadas como hiperparámetros, el rango de valores que podrán contener o el tamaño de los incrementos. Respecto a la búsqueda, es posible especificar el algoritmo de búsqueda utilizado, si es un trabajo de minimización o maximización, el valor objetivo, número de ejecuciones paralelas, número máximo de intentos y el número máximo de intentos fallidos. En este caso se dejará todo como aparece por defecto. Al pulsar el botón que aparece en la parte inferior, se compilará el pipeline y se iniciará el experimento de Katib. Pulsando en el enlace que aparece, se podrá observar el progreso de los diferentes intentos del experimento.

3.3.2 Algoritmos de IA

Una vez se conocen los fundamentos de Kubeflow, el siguiente punto tratará la creación de varios pipelines de IA. Basándonos en los casos vistos anteriormente en la sección 2.1, se tomarán algunos ejemplos representativos para ilustrar las diferencias a la hora de crear pipelines y en el flujo de vida de la aplicación final. A grandes rasgos, existen dos grupos principales dependiendo de si una vez publicado el pipeline se necesita un mantenimiento posterior o no. En la primera categoría se encuentran algoritmos como las búsquedas y algunos bioinspirados o evolutivos como el recocido simulado. En el segundo grupo se encuentran los algoritmos restantes de las categorías mencionadas anteriormente junto a algoritmos de ML y deep learning como las redes neuronales.

En el primer caso, se trata de algoritmos simples que reciben datos de entrada y devuelven una salida. Una vez han sido desplegados no requieren intervención alguna mientras no cambie el objetivo del proyecto. Sin embargo, en el segundo grupo los algoritmos requieren un paso extra comúnmente llamado entrenamiento para adaptar la estructura al problema que se quiere resolver. Una vez el algoritmo ha sido entrenado y el modelo ha sido desplegado, puede darse el caso en que la precisión de este decaiga por debajo de un umbral determinado. En esta situación sería necesario un reentrenamiento. Este aspecto no debe ser ignorado a la hora de diseñar la arquitectura de la aplicación. También puede darse el caso en que se quiera adaptar el algoritmo a otro problema y sea igualmente necesario otro entrenamiento. Teniendo en cuenta estas diferencias, los dos grupos quedarían de la siguiente forma:

- Algoritmos simples
- Algoritmos con entrenamiento

Vistas las diferencias entre ambos grupos, también comparten algunas características. En ambos casos puede ser necesario preprocesar los datos de entrada, normalizando los datos y/o eliminando valores atípicos. Además, es necesario diseñar una interfaz para poder enviar datos de entrada al servicio y recuperar su respuesta.

Como algoritmo simple se tomará como base el ejemplo de búsqueda tabú publicado por Taylan Kabbani en la página Medium [61]. En este caso, se utiliza la búsqueda tabú para resolver un problema

de tardanza total ponderada. En este problema, contamos con una máquina que solo puede trabajar en una tarea simultáneamente y existen un número determinado de tareas $i \in N$ que deben ser procesadas sin interrupciones. Cada tarea requiere un tiempo de procesamiento P_i , tiene una fecha límite d_i y una tardanza T_i que se calcula al acabar la tarea como $T_i = max\{C_i - d_i, 0\}$, siendo C_i el tiempo en que se ha completado la tarea. De esta forma, si una tarea se ha completado antes de su fecha límite $C_i \leq d_i$, la tardanza será igual a 0. El objetivo final es ordenar todas las tareas de forma que se minimice la tardanza ponderada total de todo el proceso.

Una vez separado el código original en celdas y usando las anotaciones de Kale el cuaderno quedaría de la siguiente forma:

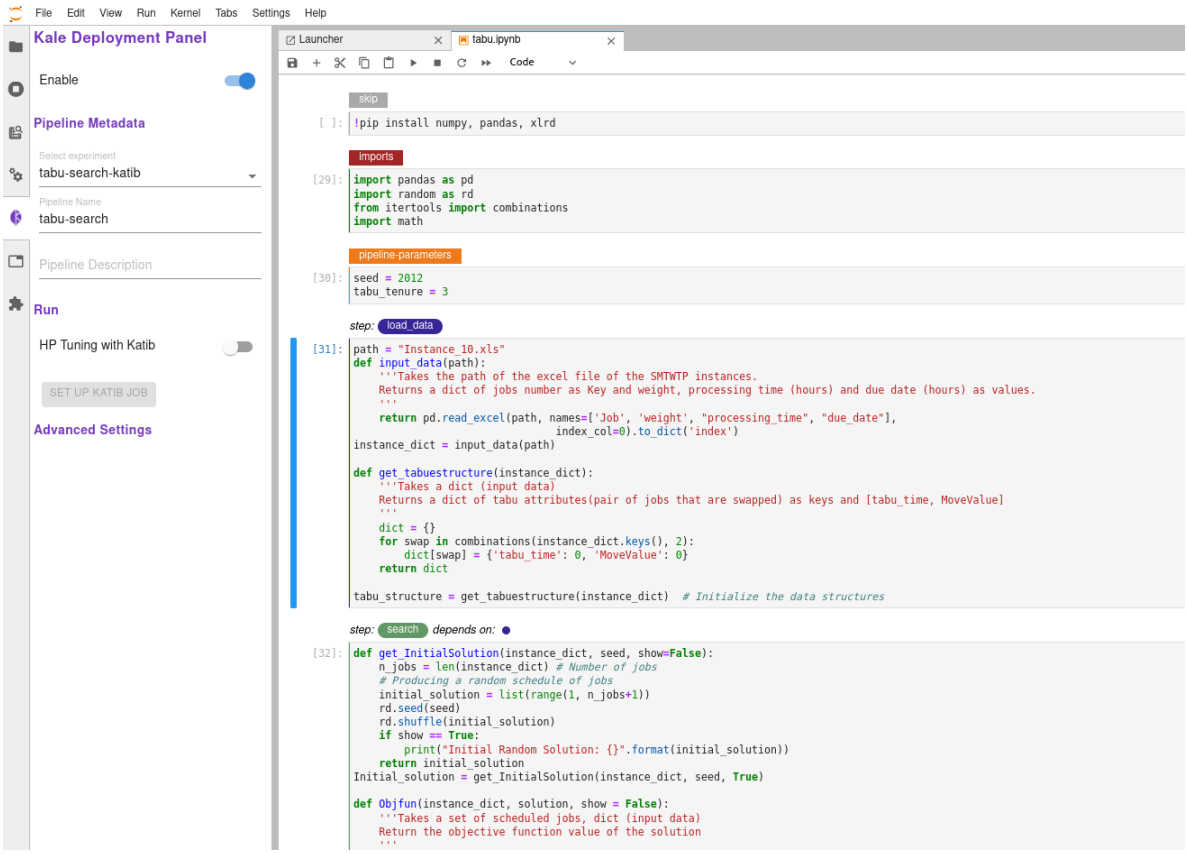

Figura 3.6: Cuaderno de Jupyter de la búsqueda tabú

En este caso se han definido 5 celdas diferentes. Las 3 primeras celdas están destinadas a preparar el entorno de ejecución. Primero se importan las librerías de Python necesarias, después se definen los parámetros del pipeline y finalmente se realiza la lectura de datos de una hoja de cálculos. La ejecución como tal del algoritmo se realiza en el paso llamado *search*, que depende a su vez del paso anterior. Finalmente, se registran las métricas del pipeline. En este ejemplo se ha tomado el mejor valor objetivo como métrica, siendo este el valor que la búsqueda tabú intenta maximizar. Una vez se compila el pipeline y se ejecuta, los pasos resultantes son los siguientes:

Una vez se ha ejecutado el pipeline es posible depurar cada paso comprobando las secciones de *Logs* y *Input/Output* (ver Fig. 3.8).

Una vez visto un caso simple, para la siguiente categoría se utilizará una red neuronal con backpropagation como ejemplo de algoritmo que requiere entrenamiento. Como base, se ha tomado la implementación de backpropagation planteada por Jason Brownlee en la página Machine Learning Mastery [62]. En este ejemplo, se toma un dataset de semillas muy utilizado en ML como datos de entrenamiento de la red

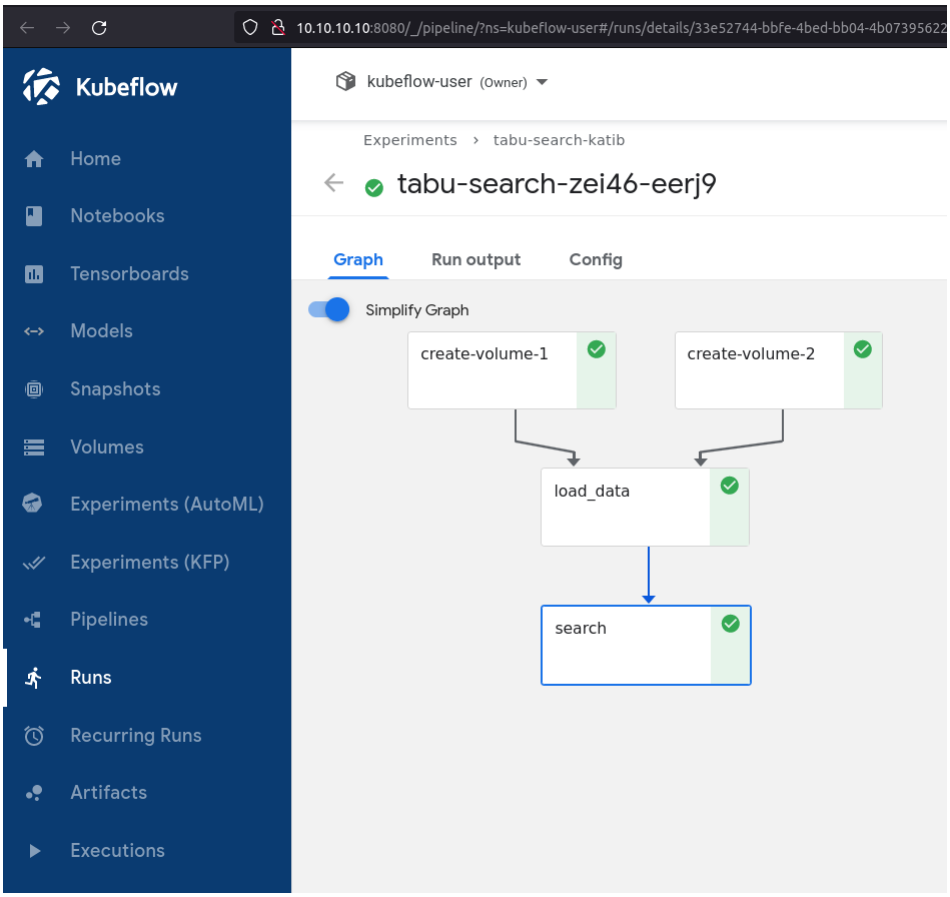

Figura 3.7: Pasos del pipeline de la búsqueda tabú

neuronal [63]. Cada línea de este conjunto de datos representa las características geométricas de un grano que puede pertenecer a 3 variedades diferentes de trigo. Las características registradas son las siguientes (siendo cada una de ellas una variable real y continua):

- Área, A
- Perímetro, P
- Compacidad, $C = \frac{4 * \pi * A}{P^2}$
- Longitud del grano
- Anchura del grano
- Coeficiente de asimetría
- Longitud del surco del grano

La última columna contiene un valor entero comprendido entre 1 y 3 que representa la variedad a la que pertenece ese grano. Por tanto, este conjunto de datos es un conjunto etiquetado. El objetivo de esta red neuronal es ser capaz de predecir la variedad a la que pertenece un grano tomando como entrada sus características. Para ello, este conjunto de datos inicial se divide en varios subconjuntos con el objetivo de realizar una validación cruzada. Además, cada subconjunto de datos se divide de nuevo en 2 partes, una se usará para el entrenamiento de la red y la restante para la validación posterior.

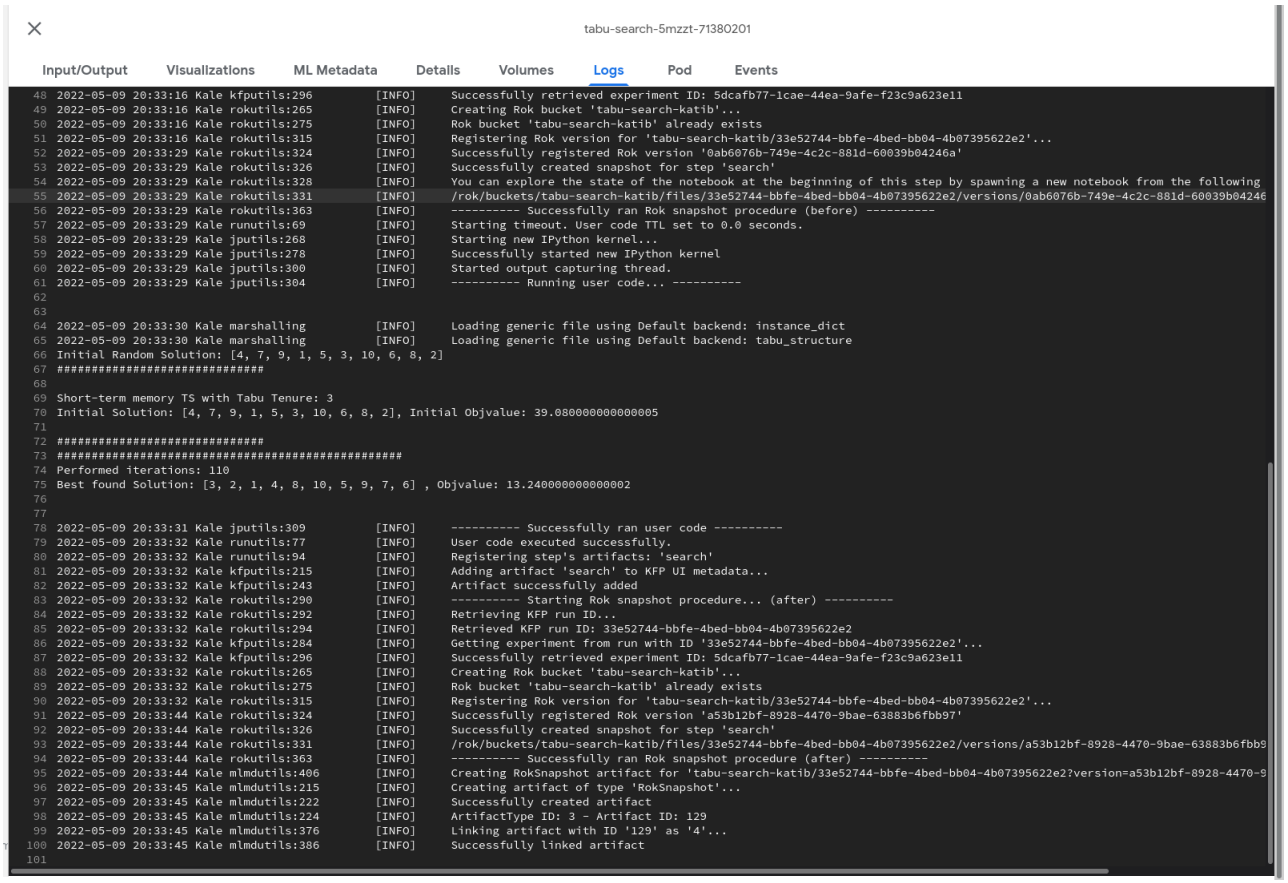

Figura 3.8: Logs del paso *search* de la búsqueda tabú

Partiendo del código inicial, se ha adaptado su estructura para aprovechar las ventajas de Kubeflow, etiquetando las celdas del cuaderno. Las 3 primeras celdas son idénticas al ejemplo de la búsqueda tabú (*imports*, textitpipeline-parameters y *load_data*). Sin embargo, para la propia ejecución se han dividido las 5 iteraciones de la validación cruzada en celdas independientes para aprovechar la paralelización. Esto es posible dado que las diferentes celdas no tienen dependencia entre sí. En este punto, también sería posible subdividir cada uno de los 5 pasos de ejecución (*fold_n*) en un paso de entrenamiento y un paso de ejecución. Finalmente, se registran los resultados de la validación de los 5 subconjuntos y se realiza la media de todos ellos, siendo estas las métricas del pipeline. En la Fig. 3.9 se pueden observar los pasos resultantes tras la compilación del cuaderno.

Tras realizar ambos ejemplos, podemos observar que en ambos casos se realizan los pasos de preprocesado de datos y el registro de resultados al final. No obstante, como fue mencionado anteriormente, mientras que en la búsqueda tabú simplemente se realiza la ejecución, en la red neuronal es necesaria la fase de entrenamiento antes de poder utilizar el algoritmo para realizar predicciones. En este último ejemplo podemos comprobar una de las ventajas de Kubeflow como es la capacidad de aplicar paralelismo a los pipelines. Si hay pasos que no dependen unos de otros, es posible realizar estos pasos de forma paralela, reduciendo el tiempo utilizado para la ejecución del pipeline.

3.4 CI/CD

En el caso de un reentrenamiento completo es necesario volver a desplegar y ejecutar el pipeline de nuevo. Esta situación puede darse en múltiples ocasiones dependiendo del problema tratado. Para simplificar

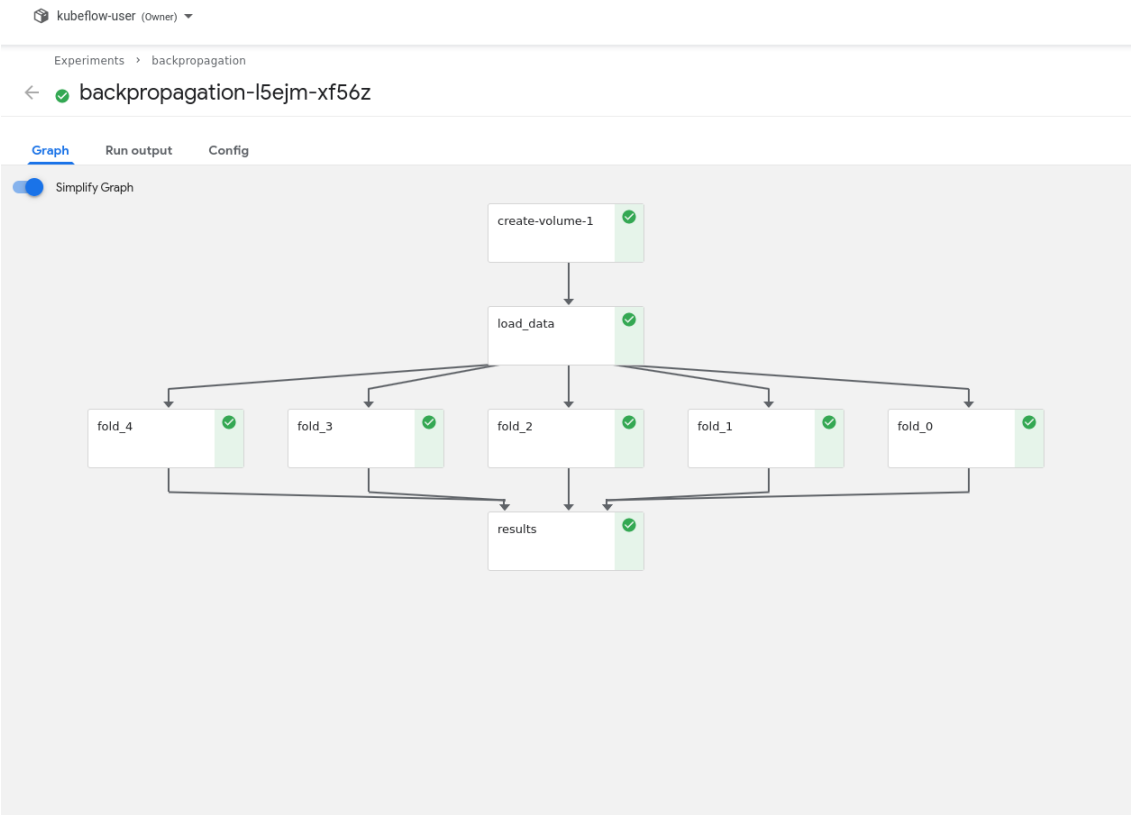

Figura 3.9: Pasos del pipeline de la red neuronal con backpropagation

este proceso, es posible aplicar los conceptos de **CI/CD**. La integración, implementación y distribución continuas, o CI/CD, es un conjunto de prácticas enfocado a distribuir aplicaciones a los clientes finales frecuentemente mediante el uso de la automatización en las diferentes etapas del desarrollo de software. El principal objetivo de CI/CD es solucionar los problemas que pueden surgir la integración del código nuevo para los equipos de desarrollo y operaciones. Dentro de CI/CD la parte de integración hace referencia al proceso de automatización creado para los desarrolladores.

Durante la fase de **CI**, se diseñan, prueban y combinan nuevos cambios en el código de forma continua utilizando un repositorio compartido. Este proceso minimiza los conflictos derivados de la división de tareas durante el desarrollo de una aplicación.

La parte de **CD** hace referencia a la distribución y/o implementación continuas. Ambos conceptos están relacionados y suelen utilizarse indistintamente. No obstante, aunque ambos términos hacen referencia a la automatización de las etapas posteriores al desarrollo, a veces se utilizan por separado para indicar hasta qué punto llega la automatización. El paso de distribución continua engloba la aplicación de pruebas automáticas para detectar errores cuando los desarrolladores publican algún cambio. Además, si el resultado de estas pruebas es exitoso, se realiza la subida de los cambios a un repositorio o registro de contenedores. Finalmente, el equipo de operaciones se encarga de desplegar la nueva versión del software en el entorno de producción real. En la implementación continua, este último paso se automatiza para que los cambios implementados por los desarrolladores se actualicen de forma autónoma en producción, reduciendo la carga de trabajo del equipo de operaciones.

Las tres fases que han sido comentadas pueden estar presentes a la vez en un proyecto o, en su defecto, solo aplicar las fases de integración y distribución. Incluso, se puede dar el caso en que un equipo comience inicialmente con la fase de integración y progresivamente vaya añadiendo el resto de fases. Como se puede

observar, es imprescindible ser cuidadoso con la nomenclatura de CI/CD ya que, de lo contrario, puede crear confusión.

Para este proyecto, la parte que más beneficio puede aportar son los conceptos de CD, en concreto el apartado de implementación continua. De esta forma, si se requiere un reentrenamiento en algún momento o se modifica alguna de las partes del flujo, el pipeline de Kubeflow puede ser desplegado y ejecutado automáticamente. Para este caso se barajaron diferentes herramientas de CI/CD. Los candidatos analizados fueron los siguientes:

- Jenkins
- GitHub Actions
- GitLab CI/CD

Jenkins [64] es un servidor de automatización open source escrito en Java ue permite a sus usuarios la capacidad de compilar, probar y despegar software de forma confiable. Jenkins cuenta con un número limitado de características por defecto pero existe una gran cantidad de complementos que pueden ser instalados para ampliar su funcionalidad. Esto brinda una gran flexibilidad para adaptar Jenkins a cada caso de uso. No obstante, es precisamente esta característica el motivo por el que se ha decidido optar por otra solución en este proyecto. Esta gran variabilidad, dificulta el aprendizaje de la herramienta y para este caso concreto sería suficiente con un subconjunto más limitado de características. Además, sería necesaria la instalación de Jenkins en algún equipo con disponibilidad completa, siendo este un requisito que no es viable.

GitHub Actions [65] es una funcionalidad ofrecida por GitHub que permite automatizar flujos de trabajo de software aplicando CI/CD. Esta herramienta permite compilar, probar y desplegar aplicaciones directamente desde el entorno de GitHub. La característica principal de GitHub Actions es la capacidad de lanzar acciones al registrar eventos como un despliegue a un repositorio, la creación de incidencias o un nuevo lanzamiento. Las tareas de un flujo de trabajo se pueden definir fácilmente utilizando un archivo yaml con sintaxis propia y se ejecutarán mediante el uso de contenedores. Este servicio no tiene una curva de aprendizaje muy elevada y es accesible en todo momento mediante los servidores de GitHub pero existe un problema. En repositorios privados el tiempo de uso del servicio para las cuentas básicas está limitado a 2000 minutos al mes y solo se puede lanzar una única instancia. Para ampliar estos límites es necesario el pago de una suscripción mensual y pagar un extra por el uso de sucesivas instancias.

GitLab CI/CD [66] es el servicio equivalente a GitHub Actions ofrecido por la empresa GitLab. Al igual que en el caso anterior, permite la creación de flujos de trabajo personalizados que se activan con diferentes eventos del repositorio. Para la definición de las tareas también se utilizan archivos yaml con una sintaxis muy similar. Sin embargo, en este caso no es necesario un pago adicional para acceder a todas las características del servicio. Además, dado que ya se estaba usando GitLab para almacenar los archivos usados en este proyecto, no es necesaria la migración de un servicio a otro. Por estos motivos, finalmente se ha decidido utilizar GitLab CI/CD para la arquitectura final del proyecto.

3.5 Arquitectura final

Tras analizar qué herramienta de CI/CD utilizar y contando con el conocimiento necesario de Kubeflow, el último paso es combinar ambas partes para crear la arquitectura final. Se partirá de un repositorio de GitLab que contiene todo el código necesario para el pipeline de Kubeflow y un script escrito en Python. Este script utilizará la API de Kubeflow Pipelines para publicar el pipeline y ejecutarlo en Kubeflow. En

este repositorio, se utilizará GitLab CI/CD para definir un flujo que se activará si el archivo del pipeline sufre alguna modificación. Cuando se active este flujo, se enviará el nuevo archivo de pipeline junto al script mencionado anteriormente al equipo que cuenta con una instalación de Kubeflow. Este envío de datos se realizará mediante scp. Finalmente, se utilizará ssh para conectar con el equipo local y ejecutar el script que subirá el pipeline a la interfaz de Kubeflow (ver Fig. 3.10).

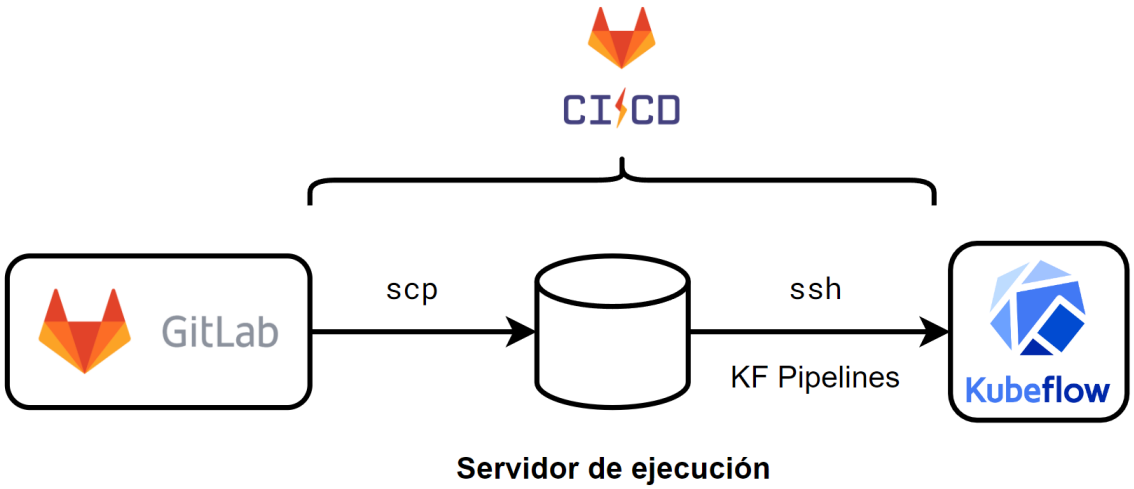

Figura 3.10: Esquema de la arquitectura

Para acceder a la herramienta de GitLab CI/CD es necesario acceder al repositorio y pulsar el icono correspondiente en la parte izquierda de la página. Este icono tiene forma de çohete". Al posar el ratón encima de esta opción aparecen varias opciones adicionales. Al seleccionar la opción de *"Editor"* aparecerá un editor web para modificar el archivo yaml que definirá el comportamiento del pipeline (figura 3.11).

En este archivo se definirán los pasos del pipeline de CI/CD utilizando una sintaxis propia. Como referencia para este paso se ha tomado tanto la documentación oficial [66] como un artículo de Karol Filipczuk [67] explicando el uso de ssh en GitLab CI/CD. El siguiente código resultante es el siguiente:

```
stages:
  - deploy
variables:
  YAML_NAME: open-vaccine-model.kale.yaml
  SCRIPT_NAME: upload_pipeline.py
  ALPINE_VERSION: 3.15.2
deploy:
  image: alpine:$ALPINE_VERSION
  stage: deploy
  before_script:
    - apk update
    - 'which ssh-agent || ( apk update && apk add openssh-client )'
    - mkdir -p ~/.ssh
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | ssh-add -
    - ssh-keyscan -H $SERVER_IP >> ~/.ssh/known_hosts
  rules:
    # if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
    # if: '$CI_PIPELINE_SOURCE == "push"'
    - changes:
      # pipeline.yaml
      - $YAML_NAME
  script:
      # ssh commands here
```

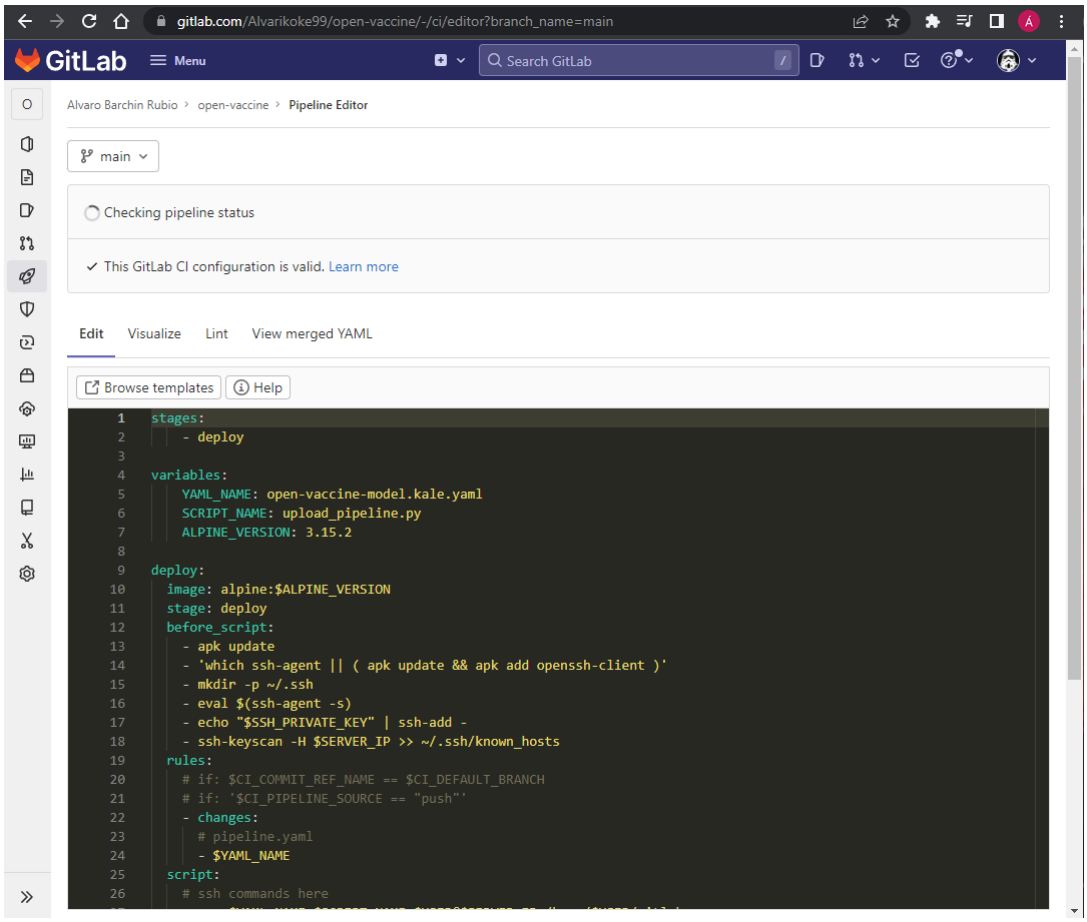

Figura 3.11: Editor de GitLab CI/CD

```
25        - scp $YAML_NAME $SCRIPT_NAME $USER@$SERVER_IP:/home/$USER/gitlab
26        # ssh $USER@$SERVER_IP "cd /home/$USER/gitlab && touch hello_there"
27        - ssh $USER@$SERVER_IP "export KUBECONFIG=/home/$USER/minikf-kubeconfig
28            && cd /home/$USER/gitlab
29            && python3 $SCRIPT_NAME"
```

En este archivo, primeramente se definen varias constantes que se usarán a lo largo del script. Entre ellas se encuentran el nombre del archivo en el que se define el pipeline de Kubeflow, el nombre del script que se utilizará para lanzar este pipeline, y la versión de **Alpine** que se utilizará. Alpine es una distribución de Linux muy ligera que se utilizará en el contenedor o *runner* que ejecutará todos los comandos definidos en este archivo. Este flujo solo tendrá un paso, *deploy*. En este paso, primero se definen los comandos que serán ejecutados para preparar el entorno en el apartado *before_script*. Aquí nos aseguraremos de que el servicio de ssh se encuentra funcionando y se añadirá la clave privada del equipo local para poder realizar la conexión. Esta clave privada junto con la ip y el usuario del equipo local se encuentran almacenadas como variables protegidas de GitLab. Para definir estas variables es necesario acceder a los ajustes del repositorio, a la sección de CI/CD y finalmente al apartado de variables.

Después, se definen las reglas que se utilizarán para lanzar la ejecución del flujo. En este caso cuando se modifique el archivo del pipeline de Kubeflow. Finalmente, se definen las órdenes que serán ejecutadas. En este caso, se copiará tanto el archivo del pipeline de Kubeflow como el script de despliegue utilizando el protocolo scp y se ejecutará este script mediante una conexión por ssh.

Una vez tengamos el trabajo configurado, cada vez que se ejecute el flujo aparecerá un nuevo registro en la categoría *Jobs* de GitLab CI/CD (figura 3.12) y se podrán obtener más detalles de la ejecución.

Tras la ejecución de este flujo el pipeline de Kubeflow se encontrará subido en la plataforma y se podrá trabajar con él como se ha visto en secciones anteriores.

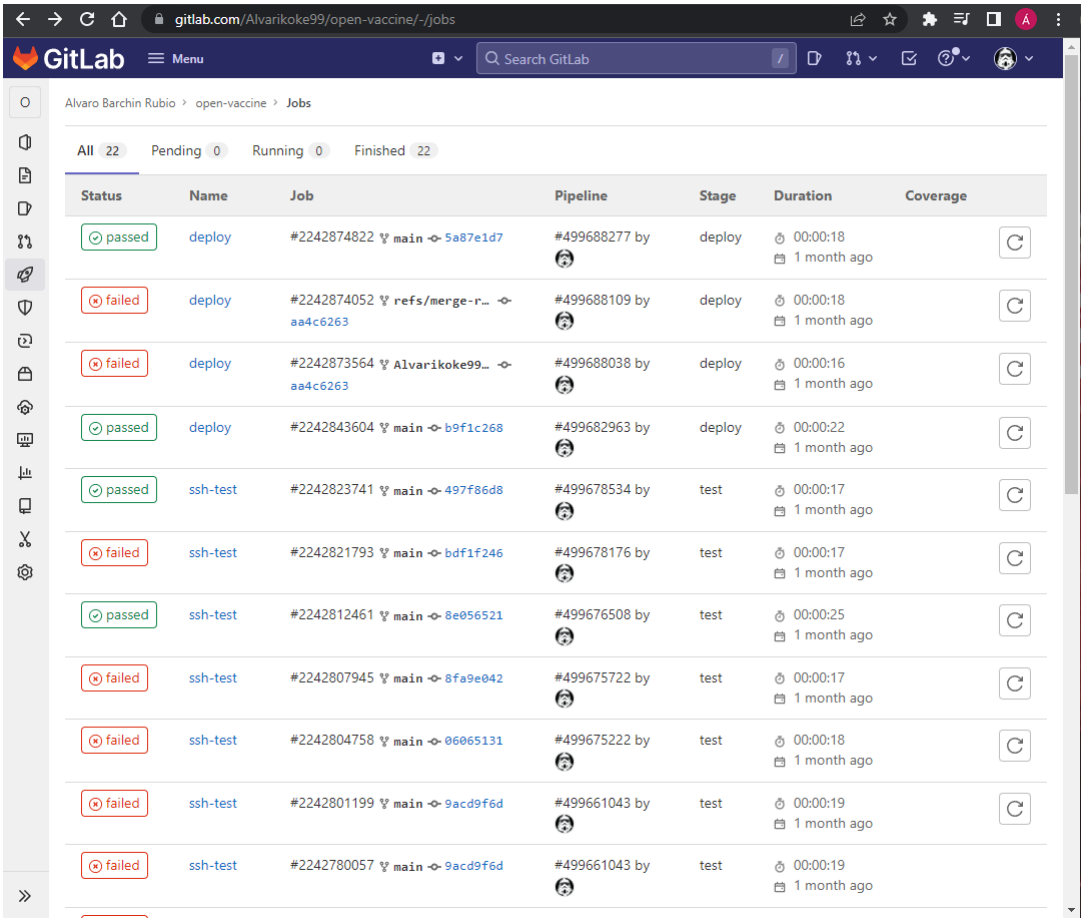

Figura 3.12: Trabajos ejecutados de GitLab CI/CD

El script usado para la publicación del pipeline está escrito en Python y aprovecha el SDK de Kubeflow Pipelines visto en la sección 3.3. Como base para este script se ha tomado un ejemplo publicado por Kubeflow en GitHub [68]. En este script (ver Apéndice A.1) se utiliza la librería de kfp para conectar con el servicio de Kubeflow, publicar el pipeline usando un archivo en el que ha sido definido y lanzar un experimento. El pipeline se encuentra definido en un archivo yaml obtenido mediante la compilación de un cuaderno de Jupyter usando Kale (ver Apéndice A.2).

Como base para este pipeline se ha utilizado el ejemplo de un tutorial de Arrikto [69]. Este ejemplo es una solución al reto de **OpenVaccine**, una competición organizada por Kaggle en 2020 y relacionada con la necesidad de dar el siguiente paso para la producción en masa de la vacuna del COVID-19 [70]. En este pipeline se crea un modelo de Tensorflow capaz de predecir tasas de degradación probables para cada base de una molécula de RNA. Tensorflow es una librería de Python utilizada para tareas de ML. En este caso, se utilizará para definir y entrenar una red neuronal recurrente o **RNN**. Por otra parte, en este pipeline también se emplea **KF Serving** para poder desplegar el modelo entrenado. KF Serving es un componente de Kubeflow que permite servir modelos ya entrenados en producción. Una vez el modelo se encuentra servido es posible enviar peticiones HTTP y recibir la respuesta en formato JSON. La mayor ventaja de KF Serving es la facilidad de despliegue partiendo de un pipeline de Kubeflow y la elasticidad automática que proporciona. Este balanceo de carga se realiza automáticamente mediante el uso de contenedores. Para servir el modelo únicamente hay que importar la librería necesaria y crear un servidor de la siguiente forma:

```
from kale.common.serveutils import serve
kfserver = serve(model, preprocessing_fn=process_features,
        preprocessing_assets={'tokenizer': tokenizer})
```

3.6 Problemas encontrados

Durante el transcurso de esto proyecto no todo el proceso ha sido desarrollado sin dificultades. Inicialmente, hubo cierta incertidumbre a la hora de elegir el entorno y el método de instalación de Kubeflow. Primero se realizaron pruebas con **WSL** (Windows Subsystem for Linux) para poder desarrollar con mayor comodidad desde Windows. Sin embargo, dado que hay ciertas características de virtualización que no se encuentran disponibles todavía en este sistema y algunos errores se descartó la idea. Al final se decidió utilizar **Pos!_OS**, una distribución de Linux derivada de Ubuntu, instalada localmente para simplificar el proceso. Una vez configurado el entorno, el siguiente paso era la propia instalación de Kubeflow. En este punto, inicialmente se intentó instalar Kubernetes con **microk8s** y más tarde añadir el complemento de Kubeflow, pero hubo varios problemas con las versiones de los diferentes componentes. Por ejemplo, la versión más reciente de microk8s no soportaba todavía el plugin de Kubeflow. Finalmente se optó por **MiniKF** para la instalación de Kubeflow.

A la hora de integrar los componentes de **CI/CD** en el proyecto se encontró cierta dificultad debido a la gran cantidad de información disponible y los múltiples contextos en los que aparecen estos conceptos. El ámbito de DevOps es muy extenso y goza de una gran popularidad hoy en día, por lo que fue necesario dedicar tiempo a filtrar la información necesaria para decidir qué herramienta utilizar en este caso y cómo implementar la funcionalidad deseada.

También se encontraron dificultades durante la publicación de un pipeline en Kubeflow de forma automática. Inicialmente, se intentó realizar el proceso mediante las utilidades de Kubeflow disponibles en la propia terminal. No obstante, se encontraron diversos problemas y se decidió cambiar a la librería de Python de Kubeflow Pipelines. En este caso, se alargó el desarrollo de esta funcionalidad debido a la escasa información disponible.

3.7 Conclusiones

Tras el desarrollo de este proyecto, se ha tenido una primera toma de contacto con Kubeflow y la creación de pipelines. Se ha aprendido cómo configurar todo el entorno y qué pasos deben seguirse desde la planificación de un pipeline hasta su concepción. Además, se ha trabajado con herramientas como Kale y Katib para simplificar el proceso de optimización de un pipeline, ajustando hiperparámetros de forma automática. Por otra parte, se ha profundizado en el mundo del DevOps y se ha trabajado con diferentes herramientas de CI/CD. De esta forma, se han analizado los pasos necesarios para adaptar un pipeline a la metodología de CI/CD. Por último, se ha aprendido cómo servir un modelo de IA creado en un pipeline aprovechando la flexibilidad que proporcionan los contenedores.

Capítulo 4

Presupuesto

En esta sección se ha realizado un desglose de los costes derivados del desarrollo del proyecto. Se han tenido en cuenta todos los costes de materiales, personal y herramientas tecnológicas.

RECURSOS MATERIALES

Concepto	Descripción	Coste
Ordenador	Ryzen 7 5800x, RTX 3080, 32GB RAM, SSD 1TB	2.000,00 €
Internet	600 MB/s simétricos	80,00 €
Electricidad	Consumo del PC: 850 W	100,00 €
Imprevistos	Contratiempos que puedan surgir durante el desarrollo	100,00 €
Total		**2.280,00 €**

RECURSOS HUMANOS

Concepto	Descripción	Coste hora	Nº horas	Coste
Ingeniero informático	Desarrollador del proyecto	35,00 €	240	8.400,00 €
Total				**8.400,00 €**

RECURSOS TECNOLÓGICOS

Concepto	Descripción	Coste
Formación en Kubernetes	Aprendizaje de Kubernetes (uso y conceptos)	1.000,00 €
Formación en Kubeflow	Aprendizaje sobre Kubeflow y sus servicios	1.500,00 €
Formación en DevOps	Aprendizaje de DevOps orientado a CI/CD	1.250,00 €
Software Vagrant	Herramienta de gestión de VM	0,00 €
Software Kubernetes	Herramienta de orquestación de contenedores	0,00 €
Software Kubeflow	Herramienta de orquestación de flujos de IA	0,00 €
GitLab CI/CD	Utilidad de CI/CD de GitLab	0,00 €*
Total		**3.750,00 €**

* Tendrá un coste de **19,00€** al mes a partir del 22-06-2022 si se superan los 400 min/mes

Nota: Las herramientas que se han usado de forma gratuita pueden necesitar el pago de una licencia si se utilizan en un entorno de producción.

COSTE TOTAL

Concepto	**Descripción**	**Coste**
Recursos materiales	Elementos y materiales	2.280,00 €
Recursos tecnológicos	Software y formaciones	3.750,00 €
Recursos humanos	Fuerza de trabajo	8.400,00 €
Total		**14.430,00 €**

Capítulo 5

Conclusiones y líneas futuras

5.1 Conclusiones

El incremento en el uso de contenedores ha creado un cambio de paradigma a la hora de diseñar y desplegar aplicaciones. Este cambio también ha influido en disciplinas como la inteligencia artificial y el machine learning. En este proyecto hemos visto la utilidad que puede aportar aprovechar las ventajas de una infraestructura basada en contenedores como la portabilidad del desarrollo, el rendimiento, la seguridad y, sobretodo, la facilidad de escalado horizontal. Estas ventajas son palpables en todas las fases del desarrollo, desde el prototipado hasta el despligue y la monitorización. Gran parte de este resultado se consigue gracias a los servicios proporcionados por Kubeflow.

A lo largo de este proyecto se ha trabajado explorando las capacidades de esta herramienta y confeccionando una hoja de ruta de su utilización. Kubeflow proporciona una interfaz intuitiva que permite a los científicos de datos trabajar de forma transparente en un entorno basado en contenedores sin ser conscientes de la arquitectura a bajo nivel. Además, Kubeflow ofrece diversos servicios como Kale y Katib que, a su vez, pueden ser utilizados para automatizar tareas repetitivas que forman parte de los flujos de IA como el ajuste de hiperparámetros. Todas estas características hacen de Kubeflow un entorno muy atractivo para los desarrollos del área de la IA. No obstante, no hay que olvidar que este tipo de arquitectura también implica un periodo de adaptación por parte de los usuarios.

Por otra parte, en este trabajo se ha realizado una primera aproximación a los diferentes tipos de algoritmos de inteligencia artificial con el objetivo de plantear varios casos de uso de la herramienta Kubeflow. Este acercamiento podría haber ahondado más en el campo de la IA, pero esto queda fuera del alcance de este proyecto.

Finalmente, se han explorado aspectos de DevOps como el CI/CD que pueden entrar en juego en algunos flujos de IA más complejos, como aquellos que pueden requerir un reentrenamiento. Los contenedores comenzaron su adopción en este ámbito y es un aspecto que todavía sigue estando presente. A partir de este punto, todo parece indicar que los servicios basados en contenedores continuarán aumentando su porcentaje de uso, tanto en el campo de la IA como en otros sectores.

5.2 Líneas futuras

Durante el desarrollo de este proyecto se ha trabajado con diferentes algoritmos de IA, adaptando su ejecución al entorno de Kubeflow. Esta herramienta resulta de gran ayuda para optimizar flujos de

trabajo y aprovechar la escalabilidad de los contenedores. En este caso se ha optado por una instalación local para tener un mayor control sobre el entorno de desarrollo. No obstante, dado que los servicios en la nube cada vez son más utilizados y este modelo de negocio está cobrando fuerza como la opción preferida por las empresas, sería interesante trasladar el trabajo realizado en este proyecto a este entorno. La principal diferencia que se encontraría en este caso sería la autenticación del servicio elegido a la hora de conectar con la herramienta de GitLab CI/CD.

Por otra parte, el pipeline desarrollado en Kubeflow, una vez servido, actúa como un servicio que recibe un entrada de datos y devuelve una respuesta en formato JSON. El siguiente paso, si se quisiera construir un producto completo, sería diseñar una interfaz para poder hacer uso del servicio. Esta interfaz podría cobrar forma en una aplicación web o, incluso, en una aplicación para dispositivos móviles. De esta forma, el resultado final sería un desrrollo completo que simularía la arquitectura de un producto comercial.

Finalmente, los diferentes algoritmos utilizados en el desarrollo no han aprovechado el uso de la GPU dado que para el objetivo de este proyecto no era algo imprescindible. Varios de estos algoritmos, como las redes neuronales, se benefician enormemente de la capacidad de paralelización aportada por una tarjeta gráfica. Esto algoritmos deberían ser adaptados para aprovechar esta característica y Kubeflow tendría que ser configurado para este nuevo modelo de ejecución. Este desarrollo se distanciaría del trabajo realizado en este proyecto, enfocándose en la optimización de algoritmos a bajo nivel y el estudio de arquitecturas de computación masivamente paralelas.

Bibliografía

[1] L. Atilano González Sotos, "Inteligencia Artificial, Apuntes: Técnicas de Búsqueda," Universidad de Alcalá, Departamento de Ciencias de la Computación, 2019.

[2] A. C. Cañari Riojas, "Búsqueda Tabú : conceptos, algoritmo y aplicación al problema de las N-reinas ," Ph.D. dissertation, Facultad de Ciencias Matemáticas, EAP. de Investigación Operativa, 2005. [Online]. Available: https://sisbib.unmsm.edu.pe/bibvirtual/monografias/basic/riojas_ca/contenido.htm

[3] A. Barragán Montero, "¿Qué es una red neuronal artificial?" 09 2020. [Online]. Available: https://cebebelgica.es/es_ES/blog/10/que-es-una-red-neuronal-artificial.html

[4] V. Mallawaarachchi, "Introduction to Genetic Algorithms - Including Example Code," 03 2020. [Online]. Available: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[5] A. Mateos, "Algoritmos Evolutivos y Algoritmos Genéticos," Tech. Rep., 2004. [Online]. Available: http://www.it.uc3m.es/~jvillena/irc/practicas/estudios/aeag

[6] Edpresso Team, "Overfitting and underfitting." [Online]. Available: https://www.educative.io/edpresso/overfitting-and-underfitting

[7] petercour, "Machine Learning Classification vs Regression," 07 2019. [Online]. Available: https://dev.to/petercour/machine-learning-classification-vs-regression-1gn

[8] L. M. Bergasa, "Sistemas de Control Inteligente, Apuntes: Chapter 3. Introduction to the Neural Networks," Universidad de Alcalá, Departamento de Electrónica, 2020.

[9] F. S. Caparrini, "Entrenamiento de Redes Neuronales: mejorando el Gradiente Descendente - Fernando Sancho Caparrini," 12 2020. [Online]. Available: http://www.cs.us.es/%7Efsancho/?e=165

[10] M. Nielsen, "Neural networks and deep learning," 12 2019. [Online]. Available: http://neuralnetworksanddeeplearning.com

[11] "Neural Networks," 08 2021. [Online]. Available: https://www.ibm.com/uk-en/cloud/learn/neural-networks

[12] C.-F. Wang, "The Vanishing Gradient Problem - Towards Data Science," 01 2019. [Online]. Available: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

[13] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way," 12 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[14] A. Polyvyanyy and D. Kuropka, *A Quantitative Evalution of the Enhanced Topic-based Vector Space Model*, 08 2009.

[15] D. Karani, "Introduction to Word Embedding and Word2Vec - Towards Data Science," 09 2018. [Online]. Available: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

[16] A. Lopardo, "Word2Vec to Transformers - Towards Data Science," 01 2020. [Online]. Available: https://towardsdatascience.com/word2vec-to-transformers-caf5a3daa08a

[17] K. Doshi, "Transformers Explained Visually (Part 1): Overview of Functionality," 12 2020. [Online]. Available: https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452

[18] DevOps Platform, "Cloud Services - IaaS, CaaS, PaaS, FaaS, SaaS | DevOps Platform," 05 2020. [Online]. Available: https://www.devopsplatform.co.uk/blogs/cloud-services

[19] Weaveworks, "A Practical Guide to Choosing between Docker Containers and VMs," 01 2020. [Online]. Available: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms

[20] C. Taylor, "What is a Hypervisor Server?" 10 2021. [Online]. Available: https://www.serverwatch.com/virtualization/hypervisor-server/

[21] "Device Passthrough - Project ACRN™ 2.7-unstable documentation." [Online]. Available: https://projectacrn.github.io/latest/developer-guides/hld/hv-dev-passthrough.html

[22] Docker, "Container Runtime." [Online]. Available: https://www.docker.com/products/container-runtime

[23] C. Millán Páramo, O. Begambre Carrillo, and E. Millán Romero, "Propuesta y validación de un algoritmo simulated annealing modificado para la solución de problemas de optimización," *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería*, vol. 30, no. 4, pp. 264–270, 2014.

[24] Paniagua, Álvaro. El uso de contenedores en España va viento en popa. [Online]. Available: https://empresas.blogthinkbig.com/uso-contenedores-espana-va-viento-en-popa/

[25] N. Matherson, "How 8 Giant Companies Use Kubernetes & 60 Others That Use It," 10 2021. [Online]. Available: https://www.containiq.com/post/companies-using-kubernetes

[26] S. Dillenburg. A brief history of container virtualization - An Idea (by Ingenious Piece). [Online]. Available: https://medium.com/an-idea/a-brief-history-of-container-virtualization-57fc96c02924

[27] J. McCarthy, "What is AI?" 11 2007. [Online]. Available: http://www-formal.stanford.edu/jmc/whatisai.pdf

[28] S. Russell and P. Norvig, *Artificial Intelligence*, 3rd ed. Prentice Hall, 2010.

[29] J. Brito Santana, C. Campos Rodríguez, F. C. García López, M. García Torres, B. Melián Batista, J. A. Moreno Pérez, and J. M. Moreno Vega, "Metaheurísticas: una revisión actualizada," Grupo de Computación Inteligente Universidad de La Laguna, 2004. [Online]. Available: https://jamoreno.webs.ull.es/www/papers/paper37.pdf

[30] C.-W. Chu, M.-D. Lin, G.-F. Liu, and Y.-H. Sung, "Application of immune algorithms on solving minimum-cost problem of water distribution network," *Mathematical and Computer Modelling*, vol. 48, no. 11-12, pp. 1888–1900, 2008.

[31] M. N. Ab Wahab, S. Nefti-Meziani, and A. Atyabi, "A Comprehensive Review of Swarm Optimization Algorithms," *PLOS ONE*, vol. 10, no. 5, p. e0122827, 2015.

[32] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Amsterdam, Países Bajos: Amsterdam University Press, 1992.

[33] S.-H. Eduardo and S.-A. Giselle, "Estrategias evolutivas para la minimización del makespan en una máquina con tiempos de preparación dependientes de la secuencia," *Ingeniería, Investigación y Tecnología*, vol. 15, no. 1, pp. 1–10, 2014.

[34] G. B. Fogel, D. Fogel, and L. Fogel, "Evolutionary programming," *Scholarpedia*, vol. 6, no. 4, p. 1818, 2011, revision #137293.

[35] J. R. Koza and R. Poli, "Genetic Programming," *Search Methodologies*, pp. 127–164, 2005.

[36] "Tratamientos térmicos: el recocido de metal | Alsimet," 08 2021. [Online]. Available: http://alsimet.es/es/noticias/tratamientos-termicos-el-recocido-de-metal

[37] J. H. Wilches Visbal and A. M. Da Costa, "Algoritmo de recocido simulado generalizado para Matlab," *Ingeniería y Ciencia*, vol. 15, no. 30, pp. 117–140, 2019.

[38] K. Dowsland and B. Díaz, "Diseño de heuristica y fundamentos del simulated annealing," *Revista Iberoamericana de Inteligencia Artificial*, vol. 19, pp. 93–102, 2003.

[39] IBM Cloud Education. Machine learning. [Online]. Available: https://www.ibm.com/cloud/learn/machine-learning

[40] P. R. D. L. Recuero, "Machine learning: conoce qué es y las diferencias entre sus tipos," 12 2021. [Online]. Available: https://empresas.blogthinkbig.com/que-algoritmo-elegir-en-ml-aprendizaje/

[41] E. Blanco, "¿Cómo funciona el algoritmo Backpropagation en una Red Neuronal? - Think Big Empresas," 10 2019. [Online]. Available: https://empresas.blogthinkbig.com/como-funciona-el-algoritmo-backpropagation-en-una-red-neuronal/

[42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[43] G. Ruiz de Villa, "Introducción a Word2vec (skip gram model) - Gonzalo Ruiz de Villa," 05 2018. [Online]. Available: https://gruizdevilla.medium.com/introducci%C3%B3n-a-word2vec-skip-gram-model-4800f72c871f

[44] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[45] X. Rong, "word2vec Parameter Learning Explained," 2014. [Online]. Available: https://arxiv.org/abs/1411.2738

[46] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[47] R. Horev, "BERT Explained: State of the art language model for NLP," 11 2018. [Online]. Available: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

[48] RedHat, "¿Qué es la infraestructura como código (IaC)?" [Online]. Available: https://www.redhat.com/es/topics/automation/what-is-infrastructure-as-code-iac

[49] ——, "What is a virtual machine (VM)?" 09 2019. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine

[50] "What are Containers? – Benefits and Use Cases | NetApp." [Online]. Available: https://www.netapp.com/devops-solutions/what-are-containers/

[51] E. Baker, "A Comprehensive Container Runtime Comparison," 06 2020. [Online]. Available: https://www.capitalone.com/tech/cloud/container-runtime/

[52] T. Mauro, "What Are Namespaces and cgroups, and How Do They Work?" 09 2021. [Online]. Available: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/

[53] Kubernetes, "¿Qué es Kubernetes?" 06 2020. [Online]. Available: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[54] Kubeflow. Kubeflow pipelines. [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/introduction/

[55] Kubernetes. minikube start. [Online]. Available: https://minikube.sigs.k8s.io/docs/start/

[56] Installing kubeflow. [Online]. Available: https://www.kubeflow.org/docs/started/installing-kubeflow/

[57] MiniKF on laptop/desktop. [Online]. Available: https://v1-4-branch.kubeflow.org/docs/distributions/minikf/minikf-vagrant/

[58] Google. From Notebook to Kubeflow Pipelines with HP Tuning: A Data Science Journey. [Online]. Available: https://codelabs.developers.google.com/codelabs/cloud-kubeflow-minikf-kale-katib/#0

[59] Arrikto. Rok cloud-native data management platform | arrikto. [Online]. Available: https://www.arrikto.com/rok-data-management-platform/

[60] Introduction to katib. [Online]. Available: https://www.kubeflow.org/docs/components/katib/overview/

[61] T. Kabbani. Tabu Search | Python | Np-hard | Metaheuristics | Heuristics | mathematical optimization | Scheduling problem | The Startup. [Online]. Available: https://medium.com/swlh/tabu-search-in-python-3199c44d44f1

[62] J. Brownlee. How to Code a Neural Network with Backpropagation In Python (from scratch). [Online]. Available: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

[63] University of California, Irvine, "UCI Machine Learning Repository: seeds Data Set," 09 2012. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/seeds

[64] Jenkins. [Online]. Available: https://www.jenkins.io/

[65] GitHub. Features - GitHub Actions. [Online]. Available: https://github.com/features/actions?utm_source=google&utm_medium=ppc&utm_campaign=2022q3-adv-WW-Google_Search-eg_brand&scid=7013o000002CdxYAAS&gclid=Cj0KCQjwpcOTBhCZARIsAEAYLuVkFBJ0Jo8U8Sj15Zp4HRjAeNPAvbrLGJHMPwMq83DmFUqYrKiF39EaAmJwcB

[66] GitLab CI/CD | GitLab. [Online]. Available: https://docs.gitlab.com/ee/ci/

[67] K. Filipczuk. GitLab CI/CD Pipeline: Run Script via SSH to remote server. [Online]. Available: https://karol-filipczuk.medium.com/gitlab-ci-cd-pipeline-run-script-via-ssh-to-remote-server-9594f326bc2f

[68] Kubeflow. Kubeflow pipelines. [Online]. Available: https://github.com/kubeflow/pipelines/blob/master/tools/benchmarks/run_service_api.ipynb

[69] Kubeflow. GitHub - kubeflow-kale/kale: Kubeflow's superfood for Data Scientists. [Online]. Available: https://github.com/kubeflow-kale/kale

[70] Kaggle. OpenVaccine: COVID-19 mRNA Vaccine Degradation Prediction | Kaggle. [Online]. Available: https://www.kaggle.com/c/stanford-covid-vaccine/overview

[71] M. Murray, "Hosting Applications in the Cloud: "as-a-service" explained," 08 2016. [Online]. Available: https://www.turnkeytec.com/hosting-applications-in-the-cloud-iaas-paas-and-saas-explained/

[72] Cassotis Consulting, "Blog Cassotis I ¿Qué es la optimización?" 10 2020. [Online]. Available: https://www.cassotis.com/insights/que-es-la-optimizacion

[73] RedHat, "¿Qué es Docker?" [Online]. Available: https://www.redhat.com/es/topics/containers/what-is-docker

[74] RedHat, "¿Qué es un hipervisor?" [Online]. Available: https://www.redhat.com/es/topics/virtualization/what-is-a-hypervisor

[75] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE transactions on cloud computing*, vol. 7, no. 3, pp. 677–692, 2019.

[76] T. Donohue, "The differences between Docker, containerd, CRI-O and runc," 08 2021. [Online]. Available: https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

[77] J. Kirenz. Kubeflow Installation Tutorial. [Online]. Available: https://kirenz.github.io/codelabs/codelabs/kubeflow-install/#0

[78] ¿Qué son la integración/distribución continuas (CI/CD)? [Online]. Available: https://www.redhat.com/es/topics/devops/what-is-ci-cd

[79] Kale, "kale/open-vaccine.ipynb at master · kubeflow-kale/kale." [Online]. Available: https://github.com/kubeflow-kale/kale/blob/master/examples/openvaccine-kaggle-competition/open-vaccine.ipynb

Apéndice A

Código fuente

A.1 Script de despliegue del pipeline

Basado en un ejemplo publicado en el repositorio de GitHub oficial de Kubeflow [68].

```
import kfp
import random
import string
import kfp_server_api
import time
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

pipeline_file_url = './open-vaccine-model.kale.yaml'
run_status_polling_interval_sec = 60

def random_suffix() -> string:
return ''.join(random.choices(string.ascii_lowercase + string.digits, k=10))

def run_finished(run_status: string) -> bool:
return run_status in {'Succeeded', 'Failed', 'Error', 'Skipped', 'Terminated'}

def run_succeeded(run_status: string) -> bool:
return run_status in {'Succeeded'}


'''client = kfp.Client()
client.upload_pipeline('./open-vaccine-model.kale.yaml', pipeline_name='test')'''


# Create a pipeline and we'll use its default version to create runs.
client = kfp.Client()
pipeline = client.upload_pipeline(pipeline_file_url, pipeline_name='open-vaccine-'
+ random_suffix())

default_version_id = pipeline.default_version.id

# Create an experiment.
experiment_name = 'experiment-' + random_suffix()
experiment = client.experiments.create_experiment(body={'name' : experiment_name})
experiment_id = experiment.id
```

```
# Measure create run latency. Note this time is the roundrip latency of
# CreateRun method. The actual run is not finished when client side
# gets the CreateRun response. Run duration will be measured below
# when run is actually finished.
resource_references = []
key = kfp_server_api.models.ApiResourceKey(id=experiment_id,
type=kfp_server_api.models.ApiResourceType.EXPERIMENT)
reference = kfp_server_api.models.ApiResourceReference(key=key,
relationship=kfp_server_api.models.ApiRelationship.OWNER)
resource_references.append(reference)

key = kfp_server_api.models.ApiResourceKey(id=default_version_id,
type=kfp_server_api.models.ApiResourceType.PIPELINE_VERSION)
reference = kfp_server_api.models.ApiResourceReference(key=key,
relationship=kfp_server_api.models.ApiRelationship.CREATOR)
resource_references.append(reference)

start = time.perf_counter()
run_name = 'run-' + random_suffix()
run = client.runs.create_run(body={'name':run_name,
        'resource_references': resource_references})
dur = time.perf_counter() - start

# Wait for the runs to finish.
# TODO(jingzhang36): We can add a timeout for this polling.
# For now we rely on the timeout of runs in KFP.
while True:
if run_finished(run.run.status):
break
else:
time.sleep(run_status_polling_interval_sec)

# When all runs are finished, measure run durations and get run latencies.
get_run_latencies = []
succeeded_run_durations = []
run_results = []

start = time.perf_counter()
dur = time.perf_counter() - start
get_run_latencies.append(dur)
if run_succeeded(run.run.status):
run_results.append('succeeded')
succeeded_run_durations.append((run.run.finished_at -
run.run.created_at).total_seconds())
else:
run_results.append('not_succeeded')

# Measure delete run latency.
delete_run_latencies = []
start = time.perf_counter()
dur = time.perf_counter() - start
delete_run_latencies.append(dur)

# Cleanup
'''client.pipelines.delete_pipeline(pipeline.id)
client.experiments.delete_experiment(experiment.id)'''
```

A.2 Archivo del pipeline open-vaccine

Obtenido tras la compilación del cuaderno publicado en el repositorio de GitHub oficial de Kale [79].

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: open-vaccine-model-
  annotations: {pipelines.kubeflow.org/kfp_sdk_version: 1.5.0,
    pipelines.kubeflow.org/pipeline_compilation_time: '2022-03-22T22:48:46.380068',
    pipelines.kubeflow.org/pipeline_spec: '{"description": "ML model for the open
      vaccine Kaggle competition", "inputs": [{"default": 0.001, "name": "LR", "optional":
      true, "type": "float"}, {"default": 10, "name": "EPOCHS", "optional": true,
      "type": "int"}, {"default": 64, "name": "BATCH_SIZE", "optional": true, "type":
      "int"}, {"default": 100, "name": "EMBED_DIM", "optional": true, "type": "int"},
      {"default": 128, "name": "HIDDEN_DIM", "optional": true, "type": "int"}, {"default":
      0.5, "name": "DROPOUT", "optional": true, "type": "float"}, {"default": 0.3,
      "name": "SP_DROPOUT", "optional": true, "type": "float"}, {"default": 107, "name":
      "TRAIN_SEQUENCE_LENGTH", "optional": true, "type": "int"},
      {"default": "http://rok.rok.svc.cluster.local/swift/v1/
        kubeflow-user/notebooks/
        open-vaccine-0_workspace-open-vaccine-o5ogqbnw6?version=5722d948-b7ac-49f5-8201-7629191ee05f",
      "name": "rok_workspace_open_vaccine_o5ogqbnw6_url", "optional": true, "type":
      "str"}], "name": "open-vaccine-model"}'}
  labels: {pipelines.kubeflow.org/kfp_sdk_version: 1.5.0}
spec:
  entrypoint: open-vaccine-model
  templates:
  - name: create-volume-1
    resource:
      action: create
      setOwnerReference: true
      manifest: |
        apiVersion: v1
        kind: PersistentVolumeClaim
        metadata:
          annotations:
            rok/origin: '{{inputs.parameters.rok_workspace_open_vaccine_o5ogqbnw6_url}}'
          name: '{{workflow.name}}-workspace-open-vaccine-o5ogqbnw6'
        spec:
          accessModes:
          - ReadWriteMany
          resources:
            requests:
              storage: 5Gi
          storageClassName: rok
    inputs:
      parameters:
      - {name: rok_workspace_open_vaccine_o5ogqbnw6_url}
    outputs:
      parameters:
      - name: create-volume-1-manifest
        valueFrom: {jsonPath: '{}'}
      - name: create-volume-1-name
        valueFrom: {jsonPath: '{.metadata.name}'}
      - name: create-volume-1-size
        valueFrom: {jsonPath: '{.status.capacity.storage}'}
  - name: final-auto-snapshot
    container:
      args: []
      command: [python3, -u, -m, kale, final-auto-snapshot, --standalone]
```

```
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/arrikto/jupyter-kale-py36@sha256:
      5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan, name: create-volume-1}
    workingDir: /home/jovyan/open-vaccine
  inputs:
    parameters:
    - {name: create-volume-1-name}
  outputs:
    artifacts:
    - {name: mlpipeline-ui-metadata, path: /tmp/mlpipeline-ui-metadata.json}
  metadata:
    annotations: {kubeflow-kale.org/dependent-templates:
                  '["create-volume-1",
                   "model-evaluation",
                   "serving"]',
      kubeflow-kale.org/volume-name-parameters: '["create-volume-1-name"]',
      pipelines.kubeflow.org/component_spec:
      '{"description": "", "implementation":
       {"container": {"args": [], "command": ["python3", "-u", "-m", "kale",
       "final-auto-snapshot","--standalone"], "env": {"PYTHONUNBUFFERED": "1"},
       "image": "gcr.io/arrikto/jupyter-kale-py36@sha256:
       5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44"}},
        "inputs": [], "name": "final-auto-snapshot"}',
      pipelines.kubeflow.org/component_ref: '{"name":
        "open-vaccine-model"}'}
    labels:
      access-ml-pipeline: "true"
      access-rok: "true"
      pipelines.kubeflow.org/metadata_written: "true"
  volumes:
  - name: create-volume-1
    persistentVolumeClaim: {claimName: '{{inputs.parameters.create-volume-1-name}}'}
- name: load-data
  container:
    args: []
    command: [python3, -u, -m, kale, /home/jovyan/open-vaccine/open-vaccine.ipynb,
      --step, load_data]
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/arrikto/jupyter-kale-py36@sha256:
      5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan, name: create-volume-1}
    workingDir: /home/jovyan/open-vaccine
  inputs:
    parameters:
    - {name: create-volume-1-name}
  outputs:
    artifacts:
    - {name: mlpipeline-ui-metadata, path: /tmp/mlpipeline-ui-metadata.json}
    - {name: load_data, path: /tmp/load_data.html}
  metadata:
    annotations: {kubeflow-kale.org/dependent-templates: '["create-volume-1"]',
      kubeflow-kale.org/volume-name-parameters: '["create-volume-1-name"]',
      pipelines.kubeflow.org/component_spec: '{"description":
        "", "implementation": {"container": {"args": [], "command": ["python3",
```

```
        "-u", "-m", "kale", "/home/jovyan/open-vaccine/open-vaccine.ipynb", "--step",
        "load_data"], "env": {"PYTHONUNBUFFERED": "1"}, "image":
        "gcr.io/arrikto/jupyter-kale-py36@sha256:
        5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44"}},
        "inputs": [], "name": "load_data"}', pipelines.kubeflow.org/component_ref:
        '{"name":
        "open-vaccine-model"}'}
    labels:
      access-ml-pipeline: "true"
      access-rok: "true"
      pipelines.kubeflow.org/metadata_written: "true"
  volumes:
  - name: create-volume-1
    persistentVolumeClaim: {claimName: '{{inputs.parameters.create-volume-1-name}}'}
- name: model-evaluation
  container:
    args: [--param, HIDDEN_DIM, '{{inputs.parameters.HIDDEN_DIM}}', --param, DROPOUT,
      '{{inputs.parameters.DROPOUT}}', --param, SP_DROPOUT,
      '{{inputs.parameters.SP_DROPOUT}}',
      --param, EMBED_DIM, '{{inputs.parameters.EMBED_DIM}}', --param,
      TRAIN_SEQUENCE_LENGTH,
      '{{inputs.parameters.TRAIN_SEQUENCE_LENGTH}}']
    command: [python3, -u, -m, kale, /home/jovyan/open-vaccine/open-vaccine.ipynb,
      --step, model_evaluation]
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/arrikto/jupyter-kale-py36@sha256:
    5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan, name: create-volume-1}
    workingDir: /home/jovyan/open-vaccine
  inputs:
    parameters:
    - {name: DROPOUT}
    - {name: EMBED_DIM}
    - {name: HIDDEN_DIM}
    - {name: SP_DROPOUT}
    - {name: TRAIN_SEQUENCE_LENGTH}
    - {name: create-volume-1-name}
  outputs:
    artifacts:
    - {name: mlpipeline-ui-metadata, path: /tmp/mlpipeline-ui-metadata.json}
    - {name: model_evaluation, path: /tmp/model_evaluation.html}
  metadata:
    annotations: {kubeflow-kale.org/dependent-templates: '["create-volume-1",
    "model-training"]',
      kubeflow-kale.org/volume-name-parameters: '["create-volume-1-name"]',
      pipelines.kubeflow.org/component_spec: '{"description":
        "", "implementation": {"container": {"args": ["--param", "HIDDEN_DIM",
        {"inputValue":"HIDDEN_DIM"}, "--param", "DROPOUT", {"inputValue": "DROPOUT"},
        "--param", "SP_DROPOUT", {"inputValue": "SP_DROPOUT"}, "--param", "EMBED_DIM",
        {"inputValue":"EMBED_DIM"}, "--param", "TRAIN_SEQUENCE_LENGTH",
        {"inputValue": "TRAIN_SEQUENCE_LENGTH"}], "command": ["python3", "-u",
        "-m", "kale", "/home/jovyan/open-vaccine/open-vaccine.ipynb",
        "--step", "model_evaluation"], "env": {"PYTHONUNBUFFERED": "1"}, "image":
        "gcr.io/arrikto/jupyter-kale-py36@sha256:
        5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44"}},
        "inputs": [{"default": 128, "name": "HIDDEN_DIM", "type": "int"}, {"default":
        0.5, "name": "DROPOUT", "type": "float"}, {"default": 0.3, "name": "SP_DROPOUT",
        "type": "float"}, {"default": 100, "name": "EMBED_DIM", "type": "int"},
```

```
        {"default": 107, "name": "TRAIN_SEQUENCE_LENGTH", "type": "int"}], "name":
        "model_evaluation"}', pipelines.kubeflow.org/component_ref: '{"name":
        "open-vaccine-model"}',
      pipelines.kubeflow.org/arguments.parameters: '{"DROPOUT":
      "{{inputs.parameters.DROPOUT}}",
        "EMBED_DIM": "{{inputs.parameters.EMBED_DIM}}", "HIDDEN_DIM":
        "{{inputs.parameters.HIDDEN_DIM}}",
        "SP_DROPOUT": "{{inputs.parameters.SP_DROPOUT}}", "TRAIN_SEQUENCE_LENGTH":
        "{{inputs.parameters.TRAIN_SEQUENCE_LENGTH}}"}'}
    labels:
      access-ml-pipeline: "true"
      access-rok: "true"
      pipelines.kubeflow.org/metadata_written: "true"
  volumes:
  - name: create-volume-1
    persistentVolumeClaim: {claimName: '{{inputs.parameters.create-volume-1-name}}'}
- name: model-training
  container:
    args: [--param, HIDDEN_DIM, '{{inputs.parameters.HIDDEN_DIM}}', --param, EPOCHS,
      '{{inputs.parameters.EPOCHS}}', --param, DROPOUT, '{{inputs.parameters.DROPOUT}}',
      --param, LR, '{{inputs.parameters.LR}}',
      --param, SP_DROPOUT, '{{inputs.parameters.SP_DROPOUT}}',
      --param, EMBED_DIM, '{{inputs.parameters.EMBED_DIM}}', --param, BATCH_SIZE,
      '{{inputs.parameters.BATCH_SIZE}}', --param, TRAIN_SEQUENCE_LENGTH,
      '{{inputs.parameters.TRAIN_SEQUENCE_LENGTH}}']
    command: [python3, -u, -m, kale, /home/jovyan/open-vaccine/open-vaccine.ipynb,
      --step, model_training]
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/arrikto/jupyter-kale-py36@sha256:
    5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan, name: create-volume-1}
    workingDir: /home/jovyan/open-vaccine
  inputs:
    parameters:
    - {name: BATCH_SIZE}
    - {name: DROPOUT}
    - {name: EMBED_DIM}
    - {name: EPOCHS}
    - {name: HIDDEN_DIM}
    - {name: LR}
    - {name: SP_DROPOUT}
    - {name: TRAIN_SEQUENCE_LENGTH}
    - {name: create-volume-1-name}
  outputs:
    artifacts:
    - {name: mlpipeline-ui-metadata, path: /tmp/mlpipeline-ui-metadata.json}
    - {name: mlpipeline-metrics, path: /tmp/mlpipeline-metrics.json}
    - {name: model_training, path: /tmp/model_training.html}
  metadata:
    annotations: {kubeflow-kale.org/dependent-templates: '["create-volume-1",
    "preprocess-data"]',
      kubeflow-kale.org/volume-name-parameters: '["create-volume-1-name"]',
      pipelines.kubeflow.org/component_spec: '{"description":
        "", "implementation": {"container": {"args": ["--param", "HIDDEN_DIM",
        {"inputValue":
        "HIDDEN_DIM"}, "--param", "EPOCHS", {"inputValue": "EPOCHS"}, "--param",
        "DROPOUT", {"inputValue": "DROPOUT"}, "--param", "LR", {"inputValue": "LR"},
        "--param", "SP_DROPOUT", {"inputValue": "SP_DROPOUT"}, "--param", "EMBED_DIM",
```

```
        {"inputValue": "EMBED_DIM"}, "--param", "BATCH_SIZE",
        {"inputValue": "BATCH_SIZE"},
        "--param", "TRAIN_SEQUENCE_LENGTH",
        {"inputValue": "TRAIN_SEQUENCE_LENGTH"}],
        "command": ["python3", "-u", "-m", "kale",
        "/home/jovyan/open-vaccine/open-vaccine.ipynb",
        "--step", "model_training"], "env": {"PYTHONUNBUFFERED": "1"}, "image":
        "gcr.io/arrikto/jupyter-kale-py36@sha256:
        5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44"}},
        "inputs": [{"default": 128, "name": "HIDDEN_DIM", "type": "int"}, {"default":
        10, "name": "EPOCHS", "type": "int"}, {"default": 0.5, "name": "DROPOUT",
        "type": "float"}, {"default": 0.001, "name": "LR", "type": "float"}, {"default":
        0.3, "name": "SP_DROPOUT", "type": "float"}, {"default": 100, "name": "EMBED_DIM",
        "type": "int"}, {"default": 64, "name": "BATCH_SIZE", "type": "int"}, {"default":
        107, "name": "TRAIN_SEQUENCE_LENGTH", "type": "int"}], "name": "model_training"}',
      pipelines.kubeflow.org/component_ref: '{"name": "open-vaccine-model"}',
      pipelines.kubeflow.org/arguments.parameters: '{"BATCH_SIZE":
        "{{inputs.parameters.BATCH_SIZE}}", "DROPOUT": "{{inputs.parameters.DROPOUT}}",
        "EMBED_DIM": "{{inputs.parameters.EMBED_DIM}}",
        "EPOCHS": "{{inputs.parameters.EPOCHS}}",
        "HIDDEN_DIM": "{{inputs.parameters.HIDDEN_DIM}}", "LR": "{{inputs.parameters.LR}}",
        "SP_DROPOUT": "{{inputs.parameters.SP_DROPOUT}}", "TRAIN_SEQUENCE_LENGTH":
        "{{inputs.parameters.TRAIN_SEQUENCE_LENGTH}}"}'}
    labels:
      access-ml-pipeline: "true"
      access-rok: "true"
      pipelines.kubeflow.org/metadata_written: "true"
  volumes:
  - name: create-volume-1
    persistentVolumeClaim: {claimName: '{{inputs.parameters.create-volume-1-name}}'}
- name: open-vaccine-model
  inputs:
    parameters:
    - {name: BATCH_SIZE}
    - {name: DROPOUT}
    - {name: EMBED_DIM}
    - {name: EPOCHS}
    - {name: HIDDEN_DIM}
    - {name: LR}
    - {name: SP_DROPOUT}
    - {name: TRAIN_SEQUENCE_LENGTH}
    - {name: rok_workspace_open_vaccine_o5ogqbnw6_url}
  dag:
    tasks:
    - name: create-volume-1
      template: create-volume-1
      arguments:
        parameters:
        - {name: rok_workspace_open_vaccine_o5ogqbnw6_url,
          value: '{{inputs.parameters.rok_workspace_open_vaccine_o5ogqbnw6_url}}'}
    - name: final-auto-snapshot
      template: final-auto-snapshot
      dependencies: [create-volume-1, model-evaluation, serving]
      arguments:
        parameters:
        - {name: create-volume-1-name,
          value: '{{tasks.create-volume-1.outputs.parameters.create-volume-1-name}}'}
    - name: load-data
      template: load-data
      dependencies: [create-volume-1]
      arguments:
```

```
        parameters:
        - {name: create-volume-1-name,
          value: '{{tasks.create-volume-1.outputs.parameters.create-volume-1-name}}'}
    - name: model-evaluation
      template: model-evaluation
      dependencies: [create-volume-1, model-training]
      arguments:
        parameters:
        - {name: DROPOUT, value: '{{inputs.parameters.DROPOUT}}'}
        - {name: EMBED_DIM, value: '{{inputs.parameters.EMBED_DIM}}'}
        - {name: HIDDEN_DIM, value: '{{inputs.parameters.HIDDEN_DIM}}'}
        - {name: SP_DROPOUT, value: '{{inputs.parameters.SP_DROPOUT}}'}
        - {name: TRAIN_SEQUENCE_LENGTH,
          value: '{{inputs.parameters.TRAIN_SEQUENCE_LENGTH}}'}
        - {name: create-volume-1-name,
          value: '{{tasks.create-volume-1.outputs.parameters.create-volume-1-name}}'}
    - name: model-training
      template: model-training
      dependencies: [create-volume-1, preprocess-data]
      arguments:
        parameters:
        - {name: BATCH_SIZE, value: '{{inputs.parameters.BATCH_SIZE}}'}
        - {name: DROPOUT, value: '{{inputs.parameters.DROPOUT}}'}
        - {name: EMBED_DIM, value: '{{inputs.parameters.EMBED_DIM}}'}
        - {name: EPOCHS, value: '{{inputs.parameters.EPOCHS}}'}
        - {name: HIDDEN_DIM, value: '{{inputs.parameters.HIDDEN_DIM}}'}
        - {name: LR, value: '{{inputs.parameters.LR}}'}
        - {name: SP_DROPOUT, value: '{{inputs.parameters.SP_DROPOUT}}'}
        - {name: TRAIN_SEQUENCE_LENGTH,
          value: '{{inputs.parameters.TRAIN_SEQUENCE_LENGTH}}'}
        - {name: create-volume-1-name,
          value: '{{tasks.create-volume-1.outputs.parameters.create-volume-1-name}}'}
    - name: preprocess-data
      template: preprocess-data
      dependencies: [create-volume-1, load-data]
      arguments:
        parameters:
        - {name: create-volume-1-name,
          value: '{{tasks.create-volume-1.outputs.parameters.create-volume-1-name}}'}
    - name: serving
      template: serving
      dependencies: [create-volume-1, model-training]
      arguments:
        parameters:
        - {name: create-volume-1-name,
          value: '{{tasks.create-volume-1.outputs.parameters.create-volume-1-name}}'}
- name: preprocess-data
  container:
    args: []
    command: [python3, -u, -m, kale, /home/jovyan/open-vaccine/open-vaccine.ipynb,
      --step, preprocess_data]
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/arrikto/jupyter-kale-py36@sha256:
    5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan, name: create-volume-1}
    workingDir: /home/jovyan/open-vaccine
  inputs:
    parameters:
```

```
    - {name: create-volume-1-name}
  outputs:
    artifacts:
    - {name: mlpipeline-ui-metadata, path: /tmp/mlpipeline-ui-metadata.json}
    - {name: preprocess_data, path: /tmp/preprocess_data.html}
  metadata:
    annotations: {kubeflow-kale.org/dependent-templates: '["create-volume-1",
    "load-data"]',
      kubeflow-kale.org/volume-name-parameters: '["create-volume-1-name"]',
      pipelines.kubeflow.org/component_spec: '{"description":
        "", "implementation": {"container": {"args": [], "command": ["python3",
        "-u", "-m", "kale", "/home/jovyan/open-vaccine/open-vaccine.ipynb", "--step",
        "preprocess_data"], "env": {"PYTHONUNBUFFERED": "1"},
        "image": "gcr.io/arrikto/jupyter-kale-py36@sha256:
        5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44"}},
        "inputs": [], "name": "preprocess_data"}',
        pipelines.kubeflow.org/component_ref: '{"name":
        "open-vaccine-model"}'}
    labels:
      access-ml-pipeline: "true"
      access-rok: "true"
      pipelines.kubeflow.org/metadata_written: "true"
  volumes:
  - name: create-volume-1
    persistentVolumeClaim: {claimName: '{{inputs.parameters.create-volume-1-name}}'}
- name: serving
  container:
    args: []
    command: [python3, -u, -m, kale, /home/jovyan/open-vaccine/open-vaccine.ipynb,
      --step, serving]
    env:
    - {name: PYTHONUNBUFFERED, value: '1'}
    image: gcr.io/arrikto/jupyter-kale-py36@sha256:
    5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44
    securityContext: {runAsUser: 0}
    volumeMounts:
    - {mountPath: /home/jovyan, name: create-volume-1}
    workingDir: /home/jovyan/open-vaccine
  inputs:
    parameters:
    - {name: create-volume-1-name}
  outputs:
    artifacts:
    - {name: mlpipeline-ui-metadata, path: /tmp/mlpipeline-ui-metadata.json}
    - {name: serving, path: /tmp/serving.html}
  metadata:
    annotations: {kubeflow-kale.org/dependent-templates: '["create-volume-1",
    "model-training"]',
      kubeflow-kale.org/volume-name-parameters: '["create-volume-1-name"]',
      pipelines.kubeflow.org/component_spec: '{"description":
        "", "implementation": {"container": {"args": [], "command": ["python3",
        "-u", "-m", "kale", "/home/jovyan/open-vaccine/open-vaccine.ipynb", "--step",
        "serving"], "env": {"PYTHONUNBUFFERED": "1"},
        "image": "gcr.io/arrikto/jupyter-kale-py36@sha256:
        5c30d30c0459b0d7597293900be0897d5595a819f5a8311765cd928f87835d44"}},
        "inputs": [], "name": "serving"}',
        pipelines.kubeflow.org/component_ref: '{"name":
        "open-vaccine-model"}'}
    labels:
      access-ml-pipeline: "true"
      access-rok: "true"
```

```
      pipelines.kubeflow.org/metadata_written: "true"
  volumes:
  - name: create-volume-1
    persistentVolumeClaim: {claimName: '{{inputs.parameters.create-volume-1-name}}'}
arguments:
  parameters:
  - {name: LR, value: 0.001}
  - {name: EPOCHS, value: 10}
  - {name: BATCH_SIZE, value: 64}
  - {name: EMBED_DIM, value: 100}
  - {name: HIDDEN_DIM, value: 128}
  - {name: DROPOUT, value: 0.5}
  - {name: SP_DROPOUT, value: 0.3}
  - {name: TRAIN_SEQUENCE_LENGTH, value: 107}
  - {name: rok_workspace_open_vaccine_o5ogqbnw6_url,
  value: 'http://rok.rok.svc.cluster.local/swift/v1/kubeflow-user/
  notebooks/open-vaccine-0_workspace-open-vaccine-o5ogqbnw6?
  version=5722d948-b7ac-49f5-8201-7629191ee05f'}
serviceAccountName: pipeline-runner
```

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá