GRADO EN INGENIERÍA DEL SOFTWARE

FRAMEWORK PARA EL DESPLIEGUE FLEXIBLE DE REDES NEURONALES DISTRIBUIDAS EN EL FOG

FRAMEWORK FOR THE FLEXIBLE DEPLOYMENT OF DISTRIBUTED NEURAL NETWORKS IN THE FOG

Realizado por

Alejandro Carnero Hijano

Tutorizado por

Daniel Garrido Márquez

Cristian Martín Fernández

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

Málaga, Septiembre de 2020

Resumen

Palabras clave: Kafka-ML, Red neuronal distribuida, Entrenamiento, Inferencia, IoT.

El objetivo general de este Trabajo Fin de Grado ha sido ampliar la funcionalidad de la aplicación Kafka-ML desarrollada por el Grupo de Investigación ERTIS, la cual gestiona y trabaja con redes neuronales. Concretamente, se ha dotado al sistema con la capacidad de trabajar con redes neuronales artificiales que son distribuidas, es decir, un conjunto de redes ordenadas que cooperan entre sí produciendo flujos de datos entre ellas.

El ciclo de uso de la aplicación consiste en: 1) el registro de redes neuronales distribuidas, pudiendo establecer todas sus características y modificarlas en cualquier momento; 2) creación de configuraciones para un conjunto de redes determinadas; 3) creación de despliegues para preparar una configuración concreta con ciertos parámetros junto con todas las redes neuronales que ésta engloba; 4) fase de entrenamiento de las redes desplegadas y obtención de las métricas del resultado; y por último, 5) la fase de inferencia en la que se obtienen los resultados de las predicciones de las redes neuronales.

El hecho de trabajar con redes neuronales que son distribuidas surge de la necesidad de obtener respuestas y resultados más rápidos en sistemas que así lo requieren, de manera que en vez de tener un dispositivo IoT que recopila información del entorno y la envía a la nube para que se procese, con todos los problemas de latencia y ancho de banda que puede haber de por medio, se pretende que haya varios dispositivos IoT dotados de inteligencia y conectados entre sí (puede haber varios niveles de dispositivos, tales como Fog y Edge), que sean capaces de procesar la información que recolectan y de esta manera obtener predicciones de forma más rápida aunque puedan ser menos precisas.

Abstract

Keywords: Kafka-ML, Distributed neural network, Training, Inference, IoT.

The main objective of this project has been to expand the functionality of the Kafka-ML application developed by the ERTIS Research Group, that manages and works with neural networks. Specifically, the system has been equipped with the ability to work with distributed artificial neural networks, which are a set of ordered networks that cooperate with each other producing data streams between them.

The application´s lifecycle consists in: 1) the registration of distributed neural networks, being able to establish all its characteristics and modify them at any time; 2) creation of configurations for a set of specific networks; 3) the creation of deployments to prepare a specific configuration with certain parameters along with all the neural networks that it encompasses; 4) a training phase of the deployed networks is executed, obtaining the result metrics; and in the end, 5) the inference phase in which the results of the neural network predictions are obtained.

The reason for working with neural networks that are distributed arises from the need to obtain faster responses and results in systems that require it, so that instead of having an IoT device that collects information from the environment and sends it to the cloud to be processed, with all the latency and bandwidth problems that may be involved, there are several IoT devices equipped with intelligence and connected to each other, (there may be multiple levels of devices, such as Fog and Edge), and these are capable of processing the information they collect, thus obtaining faster predictions, even though these may be less accurate.

Índice general

Índice de figuras

Índice de tablas

Acrónimos

TFG	Trabajo Fin de Grado
IoT	Internet of Things
API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
REST	Representational State Transfer
JSON	JavaScript Object Notation
XML	eXtensible Markup Language
HTML	HyperText Markup Language
URL	Uniform Resource Locator

Capítulo 1

Introducción

1.1. Contexto y motivación

El aprendizaje automático (del inglés, *machine learning*) es la rama de la Inteligencia Artificial que tiene como objetivo desarrollar técnicas que permitan que las máquinas aprendan. De forma más concreta, se trata de crear algoritmos capaces de generalizar comportamientos y reconocer patrones a partir de una información suministrada en forma de ejemplos. Es, por tanto, un proceso de inducción del conocimiento.

Las redes neuronales artificiales son un paradigma de aprendizaje y procesamiento automático inspirado en la forma en que funciona una red neuronal biológica. Una red neuronal artificial es un sistema de interconexión de neuronas estructurado en capas que colaboran entre sí para producir un valor de salida en función de una información de entrada, la cual se somete a diversas operaciones. Cada neurona está conectada con otras a través de unos enlaces. En estos enlaces el valor de salida de la neurona anterior es multiplicado por un valor de peso. Estos pesos en los enlaces pueden incrementar o inhibir el estado de activación de las neuronas adyacentes. Del mismo modo, a la salida de la neurona, puede existir una función limitadora o umbral, que modifica el valor resultado o impone un límite que no se debe sobrepasar antes de propagarse a otra neurona. Esta función se conoce como función de activación. Las redes neuronales son capaces de solucionar problemas con un alto nivel de complejidad, aunque previamente requieran una fase de entrenamiento. La figura 1 muestra un ejemplo de estructura de una red neuronal artificial.

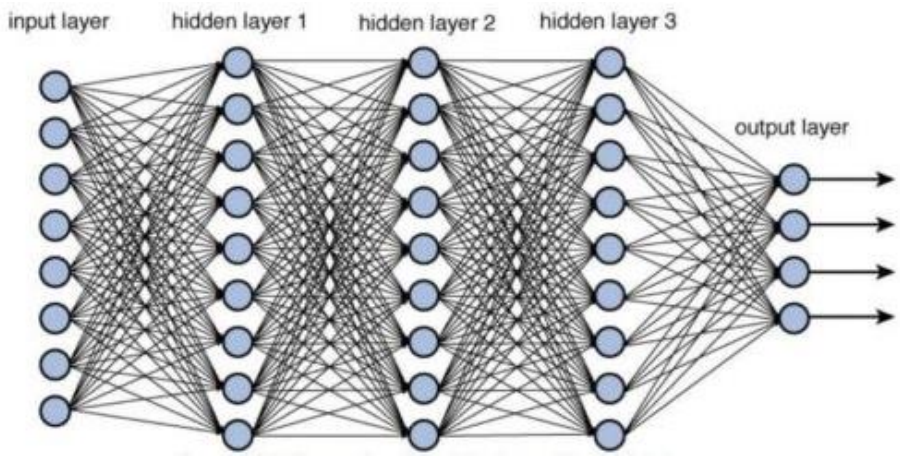

Figura 1: Ejemplo de estructura de una red neuronal artificial.

Estos sistemas aprenden y se forman a sí mismos, en lugar de ser programados de forma explícita. Para realizar este aprendizaje automático, normalmente, se intenta minimizar una función de pérdida que evalúa la red en su totalidad. Los valores de los pesos de las neuronas se van actualizando buscando reducir el valor de la función de pérdida hasta que la predicción coincida con el resultado correcto. Este proceso se realiza mediante la propagación hacia atrás, que es un método de cálculo del gradiente utilizado para entrenar redes neuronales artificiales. Una vez que se ha aplicado un patrón a la entrada de la red como estímulo, este se propaga desde la primera capa a través de las capas siguientes de la red, hasta generar una salida. La señal de salida se compara con la salida deseada y se calcula una señal de error para cada una de las salidas. Las salidas de error se propagan hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida. Sin embargo, las neuronas de la capa oculta solo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total [1]. Una vez entrenado, una red neuronal tiene el potencial de hacer una predicción precisa en todo momento.

El despliegue de redes neuronales profundas (cuentan con un gran número de capas ocultas entre las capas de entrada y salida) de forma distribuida tiende a mejorar su precisión y su tiempo de respuesta con respecto a aquellas que no lo son, sobre todo en aquellos problemas que tienen mayor complejidad y en sistema críticos los cuáles

necesitan respuestas mucho más rápidas, en donde las redes neuronales con pocas capas ocultas no obtienen una buena precisión y no hay otra alternativa que usar redes neuronales profundas con más capas de forma distribuida.

El Cloud Computing es un paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es internet. El tratamiento de redes neuronales artificiales distribuidas podría beneficiarse del uso del Fog Computing. Este paradigma de computación es una tecnología que extiende el Cloud, en la cual los datos que generan los dispositivos no son trasmitidos inmediatamente a la nube, sino que se preparan primero en centros de datos descentralizados más pequeños. El concepto abarca una red que se extiende desde el lugar donde se generan los datos hasta el destino central de esa información en la nube. Esto suele englobar al Internet de las Cosas (IoT – Internet of Things), que es la interconexión a través de internet de dispositivos informáticos integrados en objetos cotidianos, lo que les permite enviar y recibir datos. En otras palabras, IoT conecta los dispositivos a internet o a otros aparatos, para que puedan realizar nuevas funciones, como por ejemplo controlar elementos inteligentes de forma remota y recibir alertas y actualizaciones de estado. El medio idóneo para distribuir el software en este tipo de plataformas es a través de contenedores, los cuales empaquetan los recursos necesarios para el funcionamiento del sistema.

El Fog Computing puede usarse de forma complementaria al Cloud Computing, consiguiendo de esta manera una disminución de la latencia y de la sobrecarga de la red. Dentro del Fog Computing resulta muy interesante el despliegue de redes neuronales de varias capas ya que de esta forma podría obtenerse una respuesta más rápida a muchos problemas que así lo requieren.

1.2. Objetivos

El objetivo general de este proyecto es la realización de un entorno de desarrollo y despliegue apoyado en una herramienta gráfica que permita a través de contenedores y una arquitectura Fog la distribución de redes neuronales profundas de forma flexible. Para ello se definen los siguientes objetivos específicos:

1. El sistema deberá proporcionar un medio para la creación de redes neuronales distribuidas para arquitecturas de Fog Computing.

2. El sistema deberá proveer un medio para la configuración de las características de las redes neuronales distribuidas.

3. El sistema deberá poder realizar el despliegue de las redes neuronales distribuidas en una infraestructura Fog.

4. El sistema deberá proporcionar una serie de componentes y herramientas que permitan entrenar a las redes neuronales distribuidas.

5. El sistema deberá ser capaz de monitorizar y obtener los resultados del entrenamiento y de la inferencia de las redes neuronales distribuidas.

1.3. Estructura de la memoria

Este Trabajo Fin de Grado está organizado en 9 capítulos, de los cuáles el primero correspondiente a la *Introducción* ya se ha presentado. A continuación se describen los aspectos principales tratados en el resto de ellos:

- *Capítulo 2: Metodología.*
  En este capítulo se explica la metodología utilizada para el desarrollo del proyecto.

- *Capítulo 3: Tecnologías utilizadas.*
  En este capítulo se exponen las tecnologías utilizadas en el desarrollo del proyecto.

- *Capítulo 4: Análisis del sistema.*
  En este capítulo se desarrolla un análisis del proyecto, estableciendo los diferentes elementos del sistema, características y requisitos de cada uno de ellos.

- *Capítulo 5: Especificación.*

  En este capítulo se elabora una especificación del proyecto a través de los requisitos extraídos en el análisis. Se realiza el modelo de dominio y los respectivos casos de uso.

- *Capítulo 6: Diseño.*

  En este capítulo se define el diseño llevado a cabo para el sistema.

- *Capítulo 7: Implementación.*

  En este capítulo se muestra la implementación realizada para el sistema.

- *Capítulo 8: Evaluación.*

  En este capítulo se exponen las pruebas de rendimiento realizadas.

- *Capítulo 9: Conclusiones y líneas futuras.*

  En este capítulo se indican las conclusiones obtenidas al finalizar el proyecto y las líneas futuras para una posible continuación del mismo.

Capítulo 2

Metodología

En este capítulo se va a detallar la metodología empleada en el desarrollo de este TFG.

Un proyecto software engloba todo el procedimiento del desarrollo de software, desde la recogida de requisitos, pasando por las pruebas y el mantenimiento, y llevado a cabo acorde a las metodologías de ejecución, en un momento concreto en el tiempo para lograr el producto software deseado.

Una metodología de desarrollo software es un marco de trabajo que se utiliza para estructurar, planear y controlar el proceso de desarrollo en sistemas de información. Tiene que ver, por tanto, con la comunicación, la manipulación de modelos y el intercambio de información y datos entre las partes involucradas. Para ser más precisos, las metodologías de desarrollo software son enfoques de carácter estructurado y estratégico que permiten el desarrollo de programas con base a modelos de sistemas, reglas, sugerencias de diseño y guías.

En este proyecto se ha intentado seguir una metodología ágil, la cual está basada en un desarrollo iterativo e incremental. Este tipo de metodología permite adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad e inmediatez en la respuesta para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno.

2.1. Metodología ágil

Este concepto surge por la necesidad de agilizar, como su propio nombre indica, los pasos para la creación de software. El desarrollo ágil de software envuelve un enfoque para la toma de decisiones en los proyectos de software, que se refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto. Así el trabajo es realizado mediante la colaboración de equipos auto-organizados y multidisciplinarios, inmersos en un proceso compartido de toma de decisiones a corto plazo [2].

Uno de los fundamentos de estas metodologías es lo que se conoce como el ciclo de vida iterativo o incremental, también conocido como desarrollo en cascada. Este concepto consiste en el desarrollo del producto de forma progresiva, proporcionando al cliente un producto mínimo viable periódicamente y cada vez más funcional hasta llegar al producto final. Cada iteración del ciclo de vida incluye planificación, análisis de requisitos, diseño, codificación, pruebas y documentación. La figura 2 muestra el ciclo de vida de un proyecto ágil.

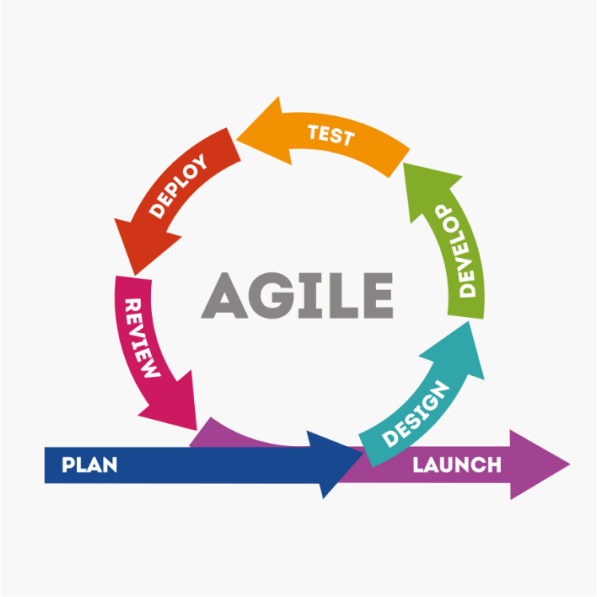

Figura 2: Ciclo de vida de un proyecto ágil.

Capítulo 3

Tecnologías utilizadas

En este capítulo se describen las tecnologías empleadas para el desarrollo del proyecto. Se mencionarán los aspectos fundamentales de cada una de ellas.

3.1. Python

Python es un lenguaje de programación interpretado de tipado dinámico cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional [3]. Posee una licencia de código abierto y está disponible para Windows, Linux y MAC. La figura 3 muestra el logotipo de Python.

Figura 3: Logotipo de Python.

Es un lenguaje sencillo y legible que atiende a un conjunto de reglas que hacen muy corta su curva de aprendizaje. Está preparado para realizar cualquier tipo de programa, desde aplicaciones Windows a servidores de red o incluso, páginas web. Es un lenguaje interpretado, lo que significa que no necesita compilar el código fuente para poder ejecutarlo, lo que ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad [4].

3.2. Angular

Angular es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles. La figura 4 muestra el logotipo de Angular.

Figura 4: Logotipo de Angular.

Angular se basa en clases tipo “Componentes”, cuyas propiedades son las usadas para hacer la unión de los datos. En dichas clases tenemos propiedades (variables) y métodos (funciones a llamar) [5].

3.3. TypeScript

TypeScript es un lenguaje de programación libre y código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos basados en clases. TypeScript puede ser usado para desarrollar aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor. Está pensado para grandes proyectos, los cuales a través de un compilador de TypeScript se traducen a código JavaScript original. La figura 5 muestra el logotipo de TypeScript.

Figura 5: Logotipo de TypeScript.

TypeScript soporta ficheros de definición que contengan información sobre los tipos de librerías JavaScript existentes, similares a los ficheros de cabeceras de C/C++ que describen la estructura de ficheros de objetos existentes. Esto permite a otros programas usar los valores definidos en los ficheros como si fueran entidades TypeScript de tipado estático [6].

3.4. TensorFlow

TensorFlow es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. TensorFlow fue originalmente desarrollado por el equipo de Google Brain para su uso interno en Google antes de ser publicado bajo la licencia de código abierto Apache 2.0 el 9 de noviembre de 2015. La arquitectura flexible de TensorFlow permite implementar el cálculo a una o más CPU o GPU en equipos de escritorio, servidores o dispositivos móviles con una sola API [7]. La figura 6 muestra el logotipo de TensorFlow.

Figura 6: Logotipo de TensorFlow.

3.5. Apache Kafka

Apache Kafka es un sistema de transmisión de datos distribuido con capacidad de escalado horizontal y tolerante a fallos. Gracias a su alto rendimiento permite transmitir datos en tiempo real utilizando el patrón de mensajería *publish/subscribe*. La figura 7 muestra el logotipo de Apache Kafka.

Figura 7: Logotipo de Apache Kafka.

Kafka fue creado por LinkedIn y actualmente es un proyecto open source mantenido por Confluent, empresa que está bajo la administración de Apache. Sus principales funcionalidades son:

- Publicar y suscribirse a flujos de datos (*streams*), actuando de forma similar a un sistema de colas de mensajes pero con un alto rendimiento obteniendo latencias muy bajas en la transmisión de mensajes.

- Permite almacenar streams que se replican para ofrecer una tolerancia a fallos.

- Facilita el procesamiento de streams en tiempo real, pudiendo transformar los datos que se almacenan en Kafka.

Kafka se usa generalmente en dos tipos de aplicaciones:

- En sistemas o aplicaciones que requieren una transmisión de streams entre ellas de manera fiable.

- En sistemas de procesamiento a tiempo real que transforman o reaccionan a los streams.

Para entender Kafka es necesario explicar los elementos que forman su estructura:

- Un *topic* es un flujo de datos sobre un tema en particular. Los topics pueden dividirse en particiones.

- Cada elemento que se almacena en un topic se denomina mensaje. Los mensajes son inmutables y son añadidos a una partición determinada en el orden en el que fueron enviados. Cada mensaje dentro de una partición tiene un identificador numérico incremental llamado *offset*.

- Un clúster de Kafka consiste en uno o más servidores denominados *Kafka brokers*. Cada bróker es identificado por un ID y contiene ciertas particiones de un topic. Permite replicar y particionar dichos topics, haciendo que Kafka sea tolerante a fallos y escalable.

- Zookeeper es un servicio imprescindible para el funcionamiento de Kafka, al cual envía notificaciones en caso de cambios. Su labor principal es gestionar los brokers de Kafka.

- Los productores permiten que una aplicación pueda publicar mensajes de un topic de Kafka de forma asíncrona.

- Los consumidores se suscriben a un topic de Kafka y pueden consumir sus mensajes para poder tratarlos en una aplicación externa [8].

La figura 8 muestra la estructura de Apache Kafka.

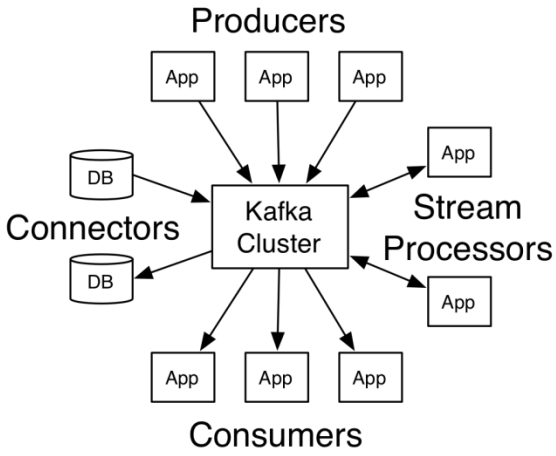

Figura 8: Estructura de Apache Kafka.

3.6. Docker

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos. Utiliza características de aislamiento de recursos del kernel de Linux para permitir que contenedores independientes se ejecuten dentro de una sola instancia de Linux, evitando la sobrecarga de iniciar y mantener máquinas virtuales. Docker es una herramienta que puede empaquetar una aplicación y sus dependencias en un contenedor virtual que se puede ejecutar en cualquier servidor Linux. Esto ayuda a permitir la flexibilidad y portabilidad en donde la aplicación se puede ejecutar [9]. La figura 9 muestra el logotipo de Docker.

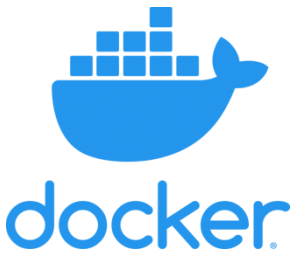

Figura 9: Logotipo de Docker.

3.7. Kubernetes

Kubernetes es un sistema de código libre para la automatización del despliegue, ajuste de escala y manejo de aplicaciones en contenedores. Kubernetes agrupa los contenedores que conforman una aplicación en unidades lógicas para una fácil administración. Elimina muchos de los procesos manuales involucrados en la implementación y escalabilidad de las aplicaciones en contenedores. Este orquestador de contenedores fue diseñado inicialmente por Google y se puede desplegar en múltiples entornos cloud. La figura 10 muestra el logotipo de Kubernetes.

Figura 10: Logotipo de Kubernetes.

3.8. Apache Zookeeper

Apache Zookeeper es un proyecto de software libre que ofrece un servicio para la coordinación de procesos distribuidos y altamente confiables que da soluciones a varios problemas de coordinación para grandes sistemas distribuidos. El servicio implementa consenso distribuido, gestión de grupos, protocolos de presencia y elección de líder. De esta forma las aplicaciones se apoyan en las primitivas expuestas por Zookeeper para resolver sus propios problemas [10]. La figura 11 muestra el logotipo de Apache Zookeeper.

Figura 11: Logotipo de Apache Zookeeper.

Capítulo 4

Análisis del sistema

El análisis de un proyecto es una de las etapas más importantes del desarrollo de software ya que hay que comprender completamente el problema a resolver y su contexto. Esto se traduce en un conjunto de requerimientos definidos por el usuario, indicando las distintas funcionalidades que desea que tenga el sistema. Es una etapa crítica en la que hay que invertir el tiempo que sea necesario y en la que se genera un documento de requerimientos [11]. En esta fase el proceso de reunión de requisitos se intensifica y se centra especialmente en el software. Dentro del proceso de análisis es fundamental que a través de la descripción de requerimientos funcionales y no funcionales, los desarrolladores comprendan completamente la naturaleza de los programas que deben construirse para desarrollar la aplicación, la función requerida, comportamiento, rendimiento e interconexión.

En este capítulo se expone una descripción global del sistema junto con sus requisitos funcionales y no funcionales.

4.1. Descripción del sistema

El sistema está formado por una serie de componentes y herramientas cuyo objetivo principal es ofrecer al usuario un entorno de desarrollo y despliegue apoyado en una aplicación web que le permita gestionar y entrenar redes neuronales distribuidas de forma flexible.

Para proporcionar toda la funcionalidad requerida el sistema cuenta con una serie de componentes:

- ***Servidor web.*** Procesa la información y realiza conexiones con los usuarios generando una respuesta.

- ***Módulo de entrenamiento.*** Encargado de llevar a cabo el entrenamiento de las redes neuronales distribuidas y de recolectar los resultados para enviárselos al servidor web.

- ***Módulo de inferencia.*** Se encarga de la fase de predicción de las redes neuronales distribuidas y de guardar los resultados para enviarlos al servidor web.

- ***Registro de control de Kafka.*** Se encarga de establecer la conexión con Kafka de la cuál recibe información de control para el entrenamiento y la inferencia y que posteriormente envía al servidor web.

4.2. Requisitos funcionales

Los requisitos funcionales de un sistema son aquellos que describen cualquier actividad que este debe realizar, en otras palabras, el comportamiento o función en particular de un software cuando se cumplen ciertas condiciones.

Por lo general, estos deben incluir funciones desempeñadas específicas, descripciones de flujos de trabajo y otros requerimientos de negocio, cumplimiento y seguridad [12].

A continuación se detallan los requisitos funcionales del sistema:

- ***RF.1.*** Los usuarios podrán crear, editar y eliminar redes neuronales distribuidas.

- ***RF.2.*** Los usuarios podrán crear, editar y eliminar configuraciones de despliegue para un conjunto de redes neuronales.

- ***RF.3.*** Los usuarios podrán desplegar una configuración concreta que incluya una serie de redes neuronales. También podrán eliminar los despliegues.

- ***RF.4.*** Los usuarios podrán enviar datos de entrenamiento a las redes neuronales desplegadas a través de un cliente Kafka.

- ***RF.5.*** Los usuarios podrán visualizar los resultados del entrenamiento de las redes neuronales. También podrán eliminar los resultados de entrenamiento.

- ***RF.6.*** Los usuarios podrán enviar datos de evaluación a las redes neuronales desplegadas para la inferencia.

- ***RF.7.*** Los usuarios podrán visualizar los resultados de la inferencia de las redes neuronales. También podrán eliminar los resultados de la inferencia.

4.3. Requisitos no funcionales

Los requisitos no funcionales o atributos de calidad son requisitos que especifican criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos. Por tanto, se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar, sino características de funcionamiento, es decir, son las restricciones o condiciones que impone el cliente al programa que necesita [13].

A continuación se detallan los requisitos no funcionales del sistema:

- ***RNF.1.*** El sistema debe visualizarse y funcionar correctamente en cualquier navegador.

- ***RNF.2.*** El servidor web permanecerá siempre activo esperando nuevas peticiones.

- ***RNF.3.*** El entorno de trabajo deberá ser accesible y amigable para los usuarios finales.

- ***RNF.4.*** El sistema deberá estar organizado para minimizar los errores de los usuarios.

- ***RNF.5.*** La aplicación web resultante será fácilmente ampliable para la incorporación de nuevos componentes.

Capítulo 5

Especificación

La fase de especificación consiste en describir detalladamente el software a ser escrito, de una forma rigurosa. Se describe el comportamiento esperado del software y su interacción con los usuarios y otros sistemas [14].

Se va a hacer uso de modelos de dominio los cuales representan un modelo conceptual de todos los temas relacionados con un problema específico. En ellos se describen las distintas entidades, sus atributos, papeles y relaciones, además de las restricciones que rigen el dominio del problema [15]. También se van a utilizar casos de uso, los cuales describen una acción o actividad. Un diagrama de casos de uso es una descripción de las actividades que podrá realizar alguien o algo para llevar a cabo algún proceso. Los diagramas de casos de uso sirven para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios y otros sistemas [16].

Este capítulo incluye un modelo de dominio, diagramas y descripción de los casos de uso con los que se expone la especificación del sistema.

5.1. Modelo de dominio

En este apartado se elabora el modelo de dominio a partir de los requisitos especificados en el Capítulo 4. Éste se muestra en la figura 12.

A continuación se detalla la finalidad de cada elemento que constituye el modelo de dominio.

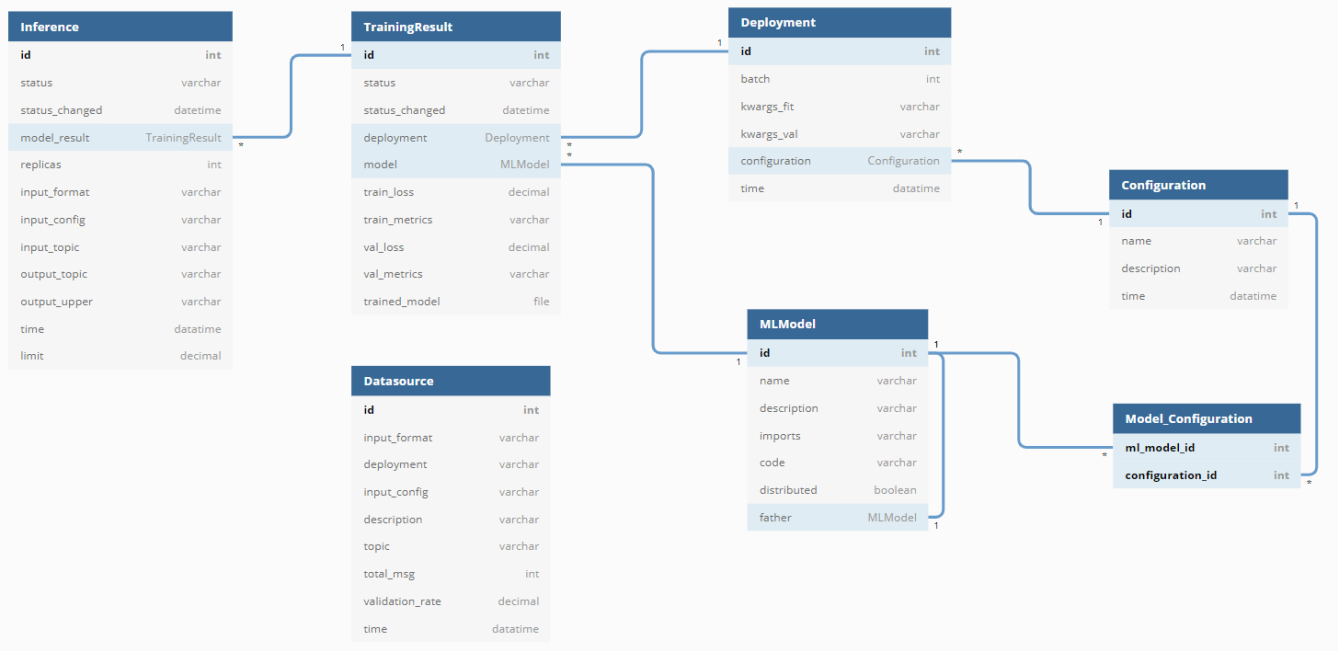

Figura 12: Modelo de dominio de Kafka-ML.

- ***MLModel.*** Representa el conjunto de redes neuronales.
- ***Configuration.*** Representa el conjunto de configuraciones.
- ***Deployment.*** Representa el conjunto de despliegues de configuraciones.
- ***TrainingResult.*** Representa el conjunto de resultados de entrenamiento.
- ***Datasource.*** Representa el conjunto de fuentes de datos.
- ***Inference.*** Representa el conjunto de inferencias.
- ***Model_Configuration.*** Representa la relación "*many-to-many*" entre MLModel y Configuration.

5.2. Diagrama de casos de uso

En este apartado se expone a través de la figura 13 el diagrama de los principales casos de uso de un usuario del sistema.

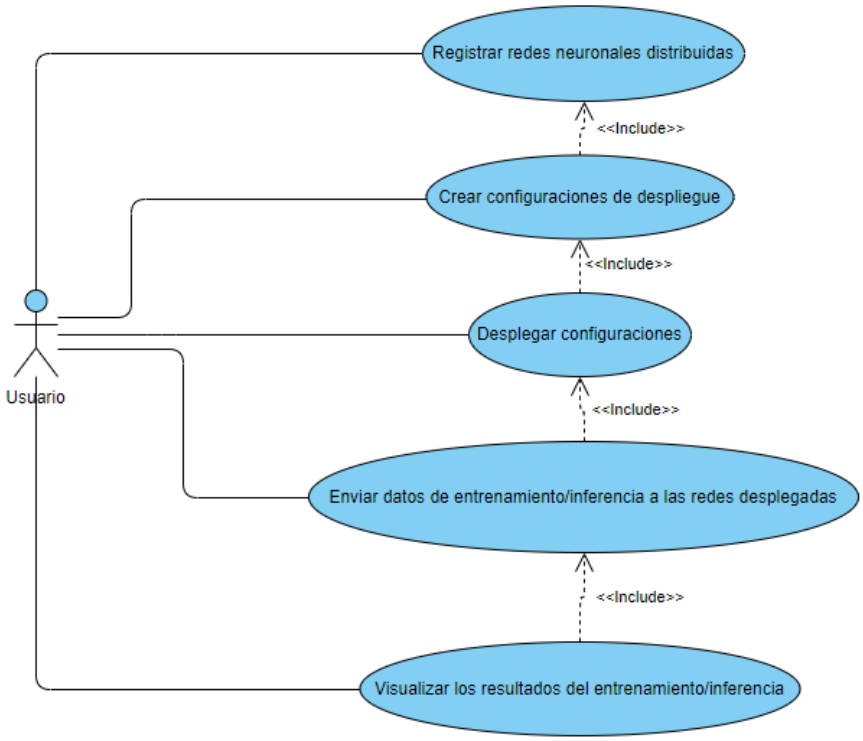

Figura 13: Diagrama de casos de uso de Kafka-ML.

5.3. Descripción de los casos de uso

En este apartado se detallan los casos de uso más relevantes del sistema. Se especifican las interacciones entre los usuarios y el sistema y el comportamiento esperado en cada caso.

Tabla 1: Descripción Caso de Uso 1.

CU.1.	**Los usuarios podrán registrar redes neuronales distribuidas.**
Contexto de uso	Cuando un usuario lo necesite, puede registrar redes neuronales distribuidas.
Ámbito	Framework web Kafka-ML.
Nivel	Usuario.
Actor principal	Cualquier usuario.
Participantes y objetivos: ▪ Usuario. Acceder al marco de trabajo para registrar redes neuronales distribuidas.	
Precondiciones	Estar arrancado el servidor web.
Garantías de éxito	El usuario accede al marco de trabajo para poder empezar a registrar redes neuronales distribuidas.
Escenario de éxito principal: 1. El usuario accede al marco de trabajo Kafka-ML. 2. El usuario registra una red neuronal distribuida. 3. La red neuronal distribuida se crea correctamente.	

Tabla 2: Descripción Caso de Uso 2.

CU.2.	**Los usuarios podrán crear configuraciones de despliegue.**
Contexto de uso	Cuando un usuario lo necesite, puede crear configuraciones de despliegue.
Ámbito	Framework web Kafka-ML.
Nivel	Usuario.
Actor principal	Cualquier usuario.
Participantes y objetivos: ▪ Usuario. Acceder al marco de trabajo para crear configuraciones de despliegue.	
Precondiciones	Estar arrancado el servidor web.
Garantías de éxito	El usuario accede al marco de trabajo para poder empezar a crear configuraciones de despliegue.
Escenario de éxito principal: 1. El usuario accede al marco de trabajo Kafka-ML. 2. El usuario crea una configuración de despliegue. 3. La configuración de despliegue se crea correctamente.	

Tabla 3: Descripción Caso de Uso 3.

CU.3.	**Los usuarios podrán desplegar configuraciones.**
Contexto de uso	Cuando un usuario lo necesite, puede desplegar configuraciones.
Ámbito	Framework web Kafka-ML.
Nivel	Usuario.
Actor principal	Cualquier usuario.
Participantes y objetivos: ▪ Usuario. Acceder al marco de trabajo para desplegar configuraciones.	
Precondiciones	1) Estar arrancado el servidor web. 2) Debe haber al menos una configuración creada.
Garantías de éxito	El usuario accede al marco de trabajo para poder empezar a desplegar configuraciones.
Escenario de éxito principal: 1. El usuario accede al marco de trabajo Kafka-ML. 2. El usuario despliega una configuración. 3. La configuración se despliega correctamente.	

Tabla 4: Descripción Caso de Uso 4.

CU.4.	**Los usuarios podrán enviar datos de entrenamiento/inferencia a las redes neuronales desplegadas.**
Contexto de uso	Cuando un usuario lo necesite, puede enviar datos de entrenamiento/inferencia a las redes neuronales desplegadas a través de un cliente Kafka.
Ámbito	Framework web Kafka-ML.
Nivel	Usuario.
Actor principal	Cualquier usuario.
Participantes y objetivos: ▪ Usuario. Acceder al marco de trabajo para enviar datos de entrenamiento/inferencia a las redes neuronales desplegadas.	
Precondiciones	Estar arrancado el servidor web.
Garantías de éxito	El usuario accede al marco de trabajo para poder empezar a enviar datos de entrenamiento/inferencia a las redes neuronales desplegadas a través de un cliente Kafka.
Escenario de éxito principal: 1. A través de un cliente Kafka el usuario envía datos de entrenamiento/inferencia a las redes neuronales de un despliegue concreto. 2. El usuario accede al marco de trabajo Kafka-ML. 3. Las redes desplegadas reciben correctamente los datos de entrenamiento/inferencia.	

Tabla 5: Descripción Caso de Uso 5.

CU.5.	**Los usuarios podrán visualizar los resultados del entrenamiento/inferencia.**
Contexto de uso	Cuando un usuario lo necesite, puede visualizar los resultados del entrenamiento/inferencia de las redes neuronales.
Ámbito	Framework web Kafka-ML.
Nivel	Usuario.
Actor principal	Cualquier usuario.
Participantes y objetivos: ▪ Usuario. Acceder al marco de trabajo para visualizar los resultados de entrenamiento/inferencia de las redes neuronales.	
Precondiciones	1) Estar arrancado el servidor web. 2) Se deben de haber enviado los datos de entrenamiento/inferencia. 3) Se ha debido realizar el entrenamiento/inferencia correctamente.
Garantías de éxito	El usuario accede al marco de trabajo para poder empezar a visualizar los resultados del entrenamiento/inferencia de las redes neuronales.
Escenario de éxito principal: 1. El usuario accede al marco de trabajo Kafka-ML. 2. El usuario accede a la ventana de resultados del entrenamiento/inferencia. 3. Los resultados del entrenamiento/inferencia se visualizan correctamente.	

Capítulo 6

Diseño

El diseño es uno de los apartados más importantes en la elaboración del TFG. Consiste en el proceso de definición de la arquitectura, componentes, interfaces y otras características del sistema, y se constituye la forma de intercomunicación entre estos.

En este capítulo se hablará de la arquitectura de diseño de la herramienta web Kafka-ML, enumerando los módulos que la componen. Cada uno de ellos se despliega en un componente de Kubernetes.

6.1. Arquitectura general

La arquitectura de Kafka-ML abarca un conjunto de componentes basados en el principio de responsabilidad única, que comprende una arquitectura de microservicios. Todos estos componentes se han desplegado en contenedores de manera que se puedan ejecutar como contenedores de Docker. Esto no sólo permite la fácil portabilidad de la arquitectura y el aislamiento entre las distintas instancias junto con un soporte de configuración rápida para distintas plataformas, sino que también permite la administración y monitorización a través de un orquestador de contenedores como es Kubernetes. Kubernetes permite la monitorización continua de contenedores y sus réplicas para asegurar que coinciden continuadamente con el estado definido para ellos, además de permitir otras características para entornos de producción como alta disponibilidad y equilibrio de carga. Kubernetes gestiona el ciclo de vida de Kafka-ML y sus componentes [17]. La figura 14 muestra la arquitectura general de la aplicación.

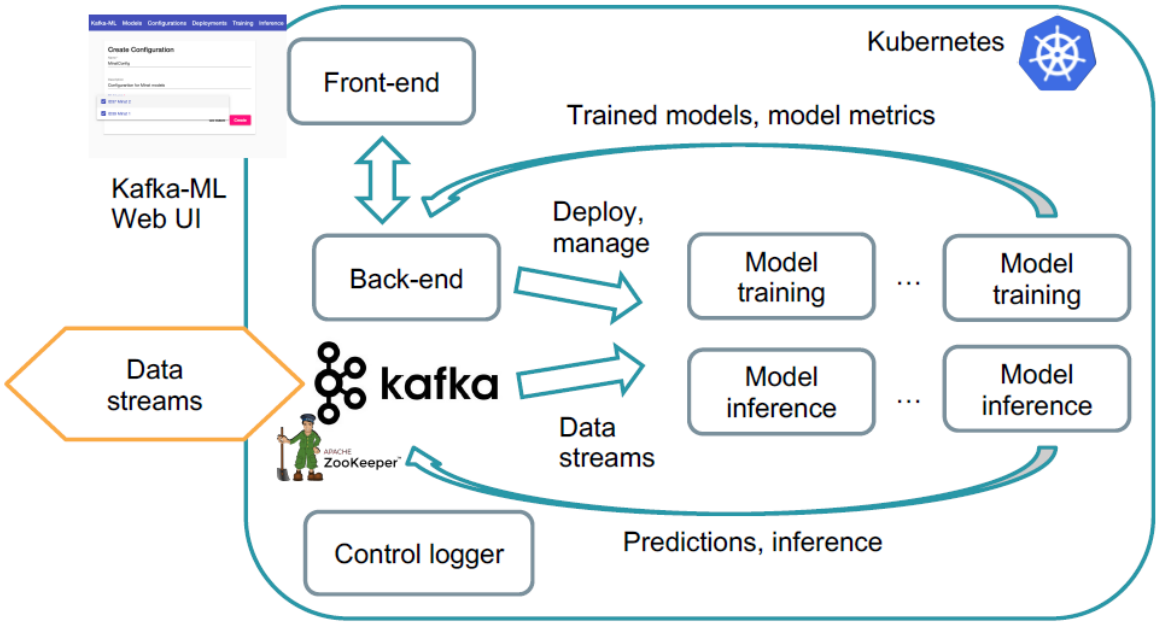

Figura 14: Arquitectura general de Kafka-ML.

6.2. Módulo backend

Este componente es la base de Kafka-ML y su papel es fundamental ya que recoge prácticamente toda la lógica y configuración necesaria para que la aplicación funcione correctamente. En él se han definido, mediante el uso del framework Django, cada una de las entidades que van a formar parte de la base de datos y las relaciones entre ellas, junto con sus serializadores, que nos ayudan a tratar con la información. En este módulo también se implementan todos los servicios que ofrece la aplicación a través de una API REST. Además en él están definidas todas las URLs de la aplicación y las pruebas unitarias de código. Está basado en Docker y Kubernetes para desplegar los servicios de entrenamiento e inferencia.

6.3. Módulo frontend

Este componente es el encargado de convertir los datos de los otros módulos en una interfaz gráfica para que el usuario pueda ver e interactuar con dicha información. En

él se definen cada una de las pantallas que forman parte de Kafka-ML. Cada una de ellas cuenta con un fichero *.html* y *.css*, que definen el diseño final que tendrán las distintas pantallas, y un fichero *.ts* y *spect.ts*, los cuáles implementan parte de la lógica necesaria para que funcionen correctamente. Este módulo también incluye una parte de la configuración de la aplicación junto con las llamadas a los servicios del backend, agrupados por cada uno de los distintos tipos de modelos existentes en la aplicación.

6.4. Módulo registro de control de Kafka

Este componente desarrollado por el Grupo de Investigación ERTIS es el encargado de recibir un mensaje de control de Kafka en el que se especifican ciertos parámetros, entre ellos está el identificador del despliegue que debe recibir los datos, el conjunto de datos para las fases de entrenamiento o inferencia y el tópico de Kafka del que se deben leer. Una vez conectado el consumidor de Kafka, se procesa cada mensaje individualmente, y si el identificador del despliegue coincide, comienza a realizar una serie de operaciones que trata los datos recibidos para posteriormente, si todo va bien, enviar esa información recibida al backend.

6.5. Módulo de entrenamiento

Este componente es el encargado de llevar a cabo el entrenamiento de las redes neuronales distribuidas. Lo primero que hace es recibir de Kafka cierta información necesaria para llevar a cabo dicho objetivo. Dentro de esa información está el grupo de identificadores de las redes neuronales implicadas en el entrenamiento, el identificador del despliegue, el conjunto de datos de entrenamiento, el tópico de Kafka del que leerlos y por último algunos parámetros que configuran el entrenamiento en sí. Después de recuperar esa información crea una red neuronal global formada por todas y cada una de las redes neuronales de la ristra distribuida para poder realizar su entrenamiento (esta es una de las formas posibles al trabajar con TensorFlow). Al final, si todo ha ido bien, recopila los resultados obtenidos para enviárselos al backend.

6.6. Módulo de inferencia

Este componente se encarga de la inferencia de las redes neuronales distribuidas una vez que están entrenadas. Primero obtiene de Kafka cierta información, la cual incluye el identificador de la red neuronal con la que se quiere predecir, los tópicos de Kafka en los que leer y escribir, el conjunto de datos para la inferencia, parámetros relacionados con la configuración del despliegue y por último el límite a partir del cual la predicción se considera correcta. El consumidor de Kafka cada vez que recibe un mensaje a inferir se lo manda a la red neuronal para que haga su predicción. Al final se recopilan los resultados obtenidos que se envían al tópico de salida si la predicción ha igualado o superado al límite, o al tópico del que lee la siguiente capa para que se sigan procesando los datos.

Capítulo 7

Implementación

Tras analizar y diseñar la solución software corresponde enfrentar la etapa de implementación de la aplicación. Consiste en la construcción definitiva donde se elaboran, adaptan y añaden los elementos previamente contemplados. El modelo diseñado en la fase anterior es la guía maestra para comenzar a ejecutar los componentes y programar las funciones que deberá cumplir el sistema. Se deben respetar las especificaciones que recomiendan los respectivos organismos al momento de elaborar o adaptar el software [18].

En este capítulo se explica la implementación llevada a cabo en el sistema, la cual se divide en los distintos módulos que forman la aplicación, exponiendo en cada uno de ellos sus principales clases. El código se ha documentado para que se pueda entender su funcionamiento.

7.1. Módulo backend

A continuación se exponen las principales clases de este módulo.

7.1.1. Models

Esta clase representa el esquema de base de datos de la aplicación Kafka-ML. En ella se especifican, con la ayuda del framework Django, cada una de las entidades del sistema y las relaciones entre ellas.

A continuación se detallan las distintas entidades.

- ***MLModel.*** Esta subclase está formada por:

  - *name*: Nombre de la red neuronal.
  - *description*: Descripción de la red neuronal.
  - *imports*: Imports necesarios para la ejecución de la red neuronal.
  - *code*: Implementación de la red neuronal.
  - *distributed*: Flag que indica si se trata de una red neuronal distribuida o no.
  - *father*: Indica la red neuronal distribuida que está por encima si es que la tiene.

  La consola 1 muestra la implementación de MLModel.

```
class MLModel(models.Model):
    """Machine learning model to be trained in the system"""

    name = models.CharField(unique=True, max_length=30)
    description = models.CharField(max_length=100, blank=True)
    imports = models.TextField(blank=True)
    code = models.TextField()
    distributed = models.BooleanField(default=False)
    father = models.OneToOneField('self', null=True, default=None, related_name='child', on_delete=models.SET_NULL)
```

Consola 1: Implementación de MLModel.

- ***Configuration.*** Esta subclase está formada por:

  - *name*: Nombre de la configuración.
  - *description*: Descripción de la configuración.
  - *ml_models*: Redes neuronales incluidas en la configuración.
  - *time*: Fecha de creación de la configuración.

  La consola 2 muestra la implementación de Configuration.

```python
class Configuration(models.Model):
    """Set of ML models to be used for training"""

    name = models.CharField(unique=True, max_length=30)
    description = models.CharField(max_length=100, blank=True)
    ml_models = models.ManyToManyField(MLModel)
    time = models.DateTimeField(default=now, editable=False)

    class Meta(object):
        ordering = ('-time', )
```

Consola 2: Implementación de Configuration.

- ***Deployment.*** Esta subclase está formada por:

  - *batch*: Tamaño de lote para el entrenamiento.
  - *kwargs_fit*: Configuración de entrenamiento.
  - *kwargs_val*: Configuración de evaluación.
  - *configuration*: Configuración asociada al despliegue.
  - *time*: Fecha de creación del despliegue.

  La consola 3 muestra la implementación de Deployment.

```python
class Deployment(models.Model):
    """Deployment of a configuration of models for training"""

    batch = models.IntegerField(default=1)
    kwargs_fit = models.CharField(max_length=100, blank=True)
    kwargs_val = models.CharField(max_length=100, blank=True)
    configuration = models.ForeignKey(Configuration, related_name='deployments', on_delete=models.CASCADE)
    time = models.DateTimeField(default=now, editable=False)

    class Meta(object):
        ordering = ('-time', )
```

Consola 3: Implementación de Deployment.

- ***TrainingResult.*** Esta subclase está formada por:

  - *status*: Estado del resultado de entrenamiento.
  - *deployment*: Despliegue asociado al resultado de entrenamiento.
  - *model*: Red neuronal asociada al resultado de entrenamiento.
  - *train_loss*: Métrica que indica la pérdida que ha tenido el entrenamiento.

- *train_metrics*: Métricas de entrenamiento (exceptuando la pérdida).
- *val_loss*: Métrica que indica la pérdida que ha tenido la evaluación.
- *val_metrics*: Métricas de evaluación (exceptuando la pérdida).
- *trained_model*: Archivo que contiene la red neuronal entrenada.

La consola 4 muestra la implementación de TrainingResult.

```
class TrainingResult(models.Model):
    """Training result information obtained once deployed a model"""

    STATUS = Choices('created', 'deployed', 'stopped', 'finished')
    """Sets its default value to the first item in the STATUS choices:"""
    status = StatusField()
    status_changed = MonitorField(monitor='status')
    deployment = models.ForeignKey(Deployment, default=None, related_name='results', on_delete=models.CASCADE)
    model = models.ForeignKey(MLModel, related_name='trained', on_delete=models.CASCADE)
    train_loss =  models.DecimalField(max_digits=15, decimal_places=10, blank=True,  null=True)
    train_metrics =  models.TextField(blank=True)
    val_loss = models.DecimalField(max_digits=15, decimal_places=10, blank=True,  null=True)
    val_metrics =  models.TextField(blank=True)
    trained_model = models.FileField(upload_to=settings.TRAINED_MODELS_DIR, blank=True)

    class Meta(object):
        ordering = ('-status_changed', )
```

Consola 4: Implementación de TrainingResult.

- ***Datasource.*** Esta subclase está formada por:

  - *input_format*: Formato de entrada.
  - *deployment*: Despliegue destino de la fuente de datos.
  - *input_config*: Configuración de entrada de los datos.
  - *description*: Descripción de la fuente de datos.
  - *topic*: Tópico de Kafka al que se envían los datos.
  - *total_msg*: Cantidad total de datos.
  - *validation_rate*: Tasa de validación.
  - *time*: Fecha en la que se envían los datos.

  La consola 5 muestra la implementación de Datasource.

```
class Datasource(models.Model):
    """Datasource used for training a deployed model"""

    INPUT_FORMAT = Choices('RAW', 'AVRO')
    """Sets its default value to the first item in the STATUS choices:"""
    input_format = StatusField(choices_name='INPUT_FORMAT')
    deployment = models.TextField()
    input_config = models.TextField(blank=True)
    description = models.TextField(blank=True)
    topic = models.TextField()
    total_msg= models.IntegerField()
    validation_rate = models.DecimalField(max_digits=7, decimal_places=6)
    time = models.DateTimeField()

    class Meta(object):
        ordering = ('-time', )
```

Consola 5: Implementación de Datasource.

- ***Inference.*** Esta subclase está formada por:

  - *status*: Estado de la inferencia.
  - *model_result*: Resultado de entrenamiento asociado a la inferencia.
  - *replicas*: Número de replicaciones de la inferencia.
  - *input_format*: Formato de entrada.
  - *input_config*: Configuración de entrada.
  - *input_topic*: Tópico de Kafka del que se leen los datos a inferir.
  - *output_topic*: Tópico de Kafka al que se envían los resultados.
  - *time*: Fecha de creación de la inferencia.
  - *limit*: Límite numérico a partir del cual un resultado se considera correcto.
  - *output_upper*: Tópico de Kafka al que se envían los datos para que se sigan procesando en la siguiente red neuronal distribuida.

  La consola 6 muestra la implementación de Inference.

```python
class Inference(models.Model):
    """Training result information obtained once deployed a model"""
    INPUT_FORMAT = Choices('RAW', 'AVRO')
    STATUS = Choices('deployed', 'stopped')
    """Sets its default value to the first item in the STATUS choices:"""

    status = StatusField()
    status_changed = MonitorField(monitor='status')
    model_result = models.ForeignKey(TrainingResult, null=True, related_name='inferences', on_delete=models.SET_NULL)
    replicas = models.IntegerField(default=1)
    input_format = StatusField(choices_name='INPUT_FORMAT')
    input_config = models.TextField(blank=True)
    input_topic =  models.TextField(blank=True)
    output_topic =  models.TextField(blank=True)
    time = models.DateTimeField(default=now, editable=False)
    limit = models.DecimalField(max_digits=15, decimal_places=10, blank=True,  null=True)
    output_upper = models.TextField(blank=True)

    class Meta(object):
        ordering = ('-time', )
```

Consola 6: Implementación de Inference.

7.1.2. Serializers

Los serializadores permiten definir al detalle cómo serán las respuestas que devolverá la API y cómo se procesa el contenido de las peticiones que llegan. Esto permite que estructuras complejas y modelos del proyecto en Django sean convertidos a estructuras nativas de Python y puedan ser transformadas fácilmente en JSON o XML.

La consola 7 muestra el serializador de la clase MLModel.

```python
class MLModelSerializer(serializers.ModelSerializer):
    father = SimpleModelSerializer(read_only=True, required=False)
    class Meta:
        model = MLModel
        fields = ['id', 'code', 'name', 'description', 'imports', 'distributed', 'father']

    def update(self, instance, validated_data):
        father = self.initial_data.get("father") if "father" in self.initial_data else None

        for (key, value) in validated_data.items():
            setattr(instance, key, value)

        if instance.distributed:
            if father:
                if father == instance.id:
                    raise serializers.ValidationError("A model can not be its own father")
                else:
                    instance.father = MLModel.objects.get(pk=father)
        else:
            instance.father = None

        instance.save()

        return instance

    def create(self, validated_data):
        father = self.initial_data.get("father") if "father" in self.initial_data else None
        model = MLModel.objects.create(**validated_data)
        if model.distributed:
            if father:
                model.father = MLModel.objects.get(pk=father)
        else:
            model.father = None

        model.save()
        return model
```

Consola 7: Serializador de MLModel.

7.1.3. Views

Esta clase contiene las funciones de Python que reciben peticiones web y se encargan de devolver respuestas web a esas peticiones. Básicamente se trata de la API REST de la aplicación que es llamada por cada uno de los servicios web que se ofrecen.

A continuación se exponen algunas funciones de esta clase.

- ***DistributedModelList.*** Esta vista se encarga de devolver la lista de redes neuronales distribuidas existentes. La consola 8 muestra su implementación.

```python
class DistributedModelList(generics.ListAPIView):
    """View to get the list of distributed models

        URL: /models/distributed
    """

    queryset = MLModel.objects.filter(distributed = True)
    serializer_class = MLModelSerializer
```

Consola 8: Implementación de DistributedModelList.

- ***ModelResultID.*** Esta vista se encarga de devolver una red neuronal a partir del ID de su resultado de entrenamiento. La consola 9 muestra su implementación.

```python
class ModelResultID(generics.RetrieveAPIView):
    """View to get a model from its result ID

        URL: /models/result/{:id_result}
    """

    def get(self, request, pk, format=None):
        if TrainingResult.objects.filter(pk=pk).exists():
            result = TrainingResult.objects.get(pk=pk)
            model = result.model
            serializer = MLModelSerializer(model, many=False)
            return HttpResponse(json.dumps(serializer.data), status=status.HTTP_200_OK)
        else:
            return HttpResponse('TrainingResult not found', status=status.HTTP_400_BAD_REQUEST)
        return HttpResponse(status=status.HTTP_400_BAD_REQUEST)
```

Consola 9: Implementación de ModelResultID.

- ***ConfigurationID.*** Esta vista se encarga de devolver la información de una configuración, de su actualización y de su eliminación. Las consolas 10 y 11 muestran su implementación.

```python
class ConfigurationID(generics.RetrieveUpdateDestroyAPIView):
    """View to get the information, update and delete a unique configuration. The configuration PK has be passed in the URL.

        URL: /configurations/{:configuration_pk}
    """
    queryset = Configuration.objects.all()
    serializer_class = ConfigurationSerializer

    def put(self, request, pk, format=None):
        """Obteins all the models from a configuration"""
        try:
            if Configuration.objects.filter(pk=pk).exists():
                data = json.loads(request.body)
                models = data['ml_models']
                all_models = []
                for m in models:
                    all_models.append(m)
                    obj = MLModel.objects.get(pk=m)
                    if hasattr(obj, 'child'):
                        son = obj.child
                        while son:
                            all_models.append(son.id)
                            if hasattr(son, 'child'):
                                son = son.child
                            else:
                                son = None
                data['ml_models'] = all_models
                configuration_obj = Configuration.objects.get(pk=pk)
                serializer = ConfigurationSerializer(configuration_obj, data=data)
                if serializer.is_valid():
                    serializer.save()
                    return HttpResponse(status=status.HTTP_200_OK)
                else:
                    return HttpResponse("Information not valid", status=status.HTTP_400_BAD_REQUEST)
            else:
                return HttpResponse(status=status.HTTP_400_BAD_REQUEST)
        except Exception as e:
            logging.error(str(e))
            return HttpResponse('Information not valid', status=status.HTTP_400_BAD_REQUEST)
```

Consola 10: Implementación de ConfigurationID 1/2.

```python
def delete(self, request, pk, format=None):
    """Deletes a configuration"""
    try:
        if Configuration.objects.filter(pk=pk).exists():
            obj = Configuration.objects.get(pk=pk)
            if Deployment.objects.filter(configuration=obj).exists():
                return HttpResponse('Configuration cannot be deleted since it is used by a deployment. Consider to delete the deployment.',
                        status=status.HTTP_400_BAD_REQUEST)
            obj.delete()
            return HttpResponse(status=status.HTTP_200_OK)
        return HttpResponse("Model does not exist", status=status.HTTP_400_BAD_REQUEST)
    except Exception as e:
        traceback.print_exc()
        return HttpResponse(str(e), status=status.HTTP_400_BAD_REQUEST)
```

Consola 11: Implementación de ConfigurationID 2/2.

- ***DeploymentList.*** Esta vista se encarga de devolver la lista de despliegues existentes y de crear un nuevo despliegue en Kubernetes. Las consolas 12, 13, 14 y 15 muestran su implementación.

```python
class DeploymentList(generics.ListCreateAPIView):
    """View to get the list of deployments and create a new deployment in Kubernetes

        URL: /deployments
    """
    queryset = Deployment.objects.all()
    serializer_class = DeploymentSerializer

    def post(self, request, format=None):
        """ Expects a JSON in the request body with the information to create a new deployment

            Args JSON:
                batch (int): Name of the model
                kwargs_fit (str): Arguments required for training the models
                configuration (int): PK of the Configuration associated with the deployment

                Example:
                {
                    "batch":"10,
                    "kwargs_fit":"epochs=5, steps_per_epoch=1000",
                    "configuration": 1
                }
            Returns:
                HTTP_201_CREATED: if the deployment has been created correctly and deployed in Kubernetes
                HTTP_400_BAD_REQUEST: if there has been any error.
        """
        try:
            data = json.loads(request.body)
            serializer = DeployDeploymentSerializer(data=data)
            if serializer.is_valid():
                deployment = serializer.save()
                try:
                    """ KUBERNETES code goes here"""
                    config.load_incluster_config() # To run inside the container
                    #config.load_kube_config() # To run externally
                    api_instance = client.BatchV1Api()

                    for result in TrainingResult.objects.filter(deployment=deployment):
                        if not result.model.distributed:
```

Consola 12: Implementación de DeploymentList 1/4.

```
    job_manifest = {
        'apiVersion': 'batch/v1',
        'kind': 'Job',
        'metadata': {
            'name': 'model-training-'+str(result.id)
        },
        'spec': {
            'ttlSecondsAfterFinished' : 10,
            'template' : {
                'spec': {
                    'containers': [{
                        'image': settings.TRAINING_MODEL_IMAGE,
                        'name': 'training',
                        'env': [{'name': 'BOOTSTRAP_SERVERS', 'value': settings.BOOTSTRAP_SERVERS},
                                {'name': 'RESULT_URL', 'value': 'http://backend:8000/results/'+str(result.id)},
                                {'name': 'RESULT_ID', 'value': str(result.id)},
                                {'name': 'CONTROL_TOPIC', 'value': settings.CONTROL_TOPIC},
                                {'name': 'DEPLOYMENT_ID', 'value': str(deployment.id)},
                                {'name': 'BATCH', 'value': str(deployment.batch)},
                                {'name': 'KWARGS_FIT', 'value': parse_kwargs_fit(deployment.kwargs_fit)},
                                {'name': 'KWARGS_VAL', 'value': parse_kwargs_fit(deployment.kwargs_val)}
                                ],
                    }],
                    'imagePullPolicy': 'Never', # TODO: Remove this when the image is in DockerHub
                    'restartPolicy': 'OnFailure'
                }
            }
        }
    }
    resp = api_instance.create_namespaced_job(body=job_manifest, namespace='default')

elif result.model.distributed and result.model.father is None:
    """Obteins all the distributed models from a deployment and creates a job for each group of them"""
    result_urls = []
    result_ids = []
    s = 'http://backend:8000/results/'
    n = ''
```

Consola 13: Implementación de DeploymentList 2/4.

```
result_urls.append(s+str(result.id))
result_ids.append(str(result.id))
n += '-'+str(result.id)
current = result
while hasattr(current.model, 'child'):
    current = TrainingResult.objects.get(model=current.model.child, deployment=deployment)
    result_urls.append(s+str(current.id))
    result_ids.append(str(current.id))
    n += '-'+str(current.id)

result_urls.reverse()
result_ids.reverse()

job_manifest = {
    'apiVersion': 'batch/v1',
    'kind': 'Job',
    'metadata': {
        'name': 'model-distributed-training'+n
    },
    'spec': {
        'ttlSecondsAfterFinished' : 10,
        'template' : {
            'spec': {
                'containers': [{
                    'image': settings.DISTRIBUTED_TRAINING_MODEL_IMAGE,
                    'name': 'training',
                    'env': [{'name': 'BOOTSTRAP_SERVERS', 'value': settings.BOOTSTRAP_SERVERS},
                            {'name': 'RESULT_URL', 'value': str(result_urls)},
                            {'name': 'RESULT_ID', 'value': str(result_ids)},
                            {'name': 'CONTROL_TOPIC', 'value': settings.CONTROL_TOPIC},
                            {'name': 'DEPLOYMENT_ID', 'value': str(deployment.id)},
                            {'name': 'BATCH', 'value': str(deployment.batch)},
                            {'name': 'KWARGS_FIT', 'value': parse_kwargs_fit(deployment.kwargs_fit)},
                            {'name': 'KWARGS_VAL', 'value': parse_kwargs_fit(deployment.kwargs_val)}
                            ],
                }],
                'imagePullPolicy': 'Never', # TODO: Remove this when the image is in DockerHub
                'restartPolicy': 'OnFailure'
            }
```

Consola 14: Implementación de DeploymentList 3/4.

```
                        }
                    }
                }
                resp = api_instance.create_namespaced_job(body=job_manifest, namespace='default')

            return HttpResponse(status=status.HTTP_201_CREATED)
        except Exception as e:
            traceback.print_exc()
            Deployment.objects.filter(pk=deployment.pk).delete()
            logging.error(str(e))
            return HttpResponse(str(e), status=status.HTTP_400_BAD_REQUEST)
    return HttpResponse(status=status.HTTP_400_BAD_REQUEST)
except Exception as e:
    logging.error(str(e))
    traceback.print_exc()
    return HttpResponse(str(e), status=status.HTTP_400_BAD_REQUEST)
```

Consola 15: Implementación de DeploymentList 4/4.

- ***InferenceResultID.*** Esta vista se encarga de devolver la información de una inferencia a partir de su resultado de entrenamiento y de desplegar una nueva. Las consolas 16 y 17 muestran su implementación.

```python
class InferenceResultID(generics.ListCreateAPIView):
    """View to get information and deploy a new inference from a training result

        URL: /results/inference/{:id_result}
    """

    def get(self, request, pk, format=None):
        """ Checks the training result exists and returns the input format and configuration if there any in other inference or
            datasource objects to facilitate the inference deployment.
        """
        try:
            if TrainingResult.objects.filter(pk=pk).exists():
                response = {
                    'input_format': '',
                    'input_config': '',
                }
                result = TrainingResult.objects.get(id=pk)
                inferences = Inference.objects.filter(model_result=result)
                if inferences.count() > 0:
                    response['input_format']=inferences[0].input_format
                    response['input_config']=inferences[0].input_config
                else:
                    model = result.model
                    datasources = Datasource.objects.filter(deployment=str(result.deployment.id))

                    if datasources.count() > 0:
                        response['input_format'] = datasources[0].input_format
                        input_config = datasources[0].input_config

                        if not hasattr(model, 'child'):
                            response['input_config'] = input_config # TODO change to input_config
                        else:
                            tensorflow_model = exec_model(model.imports, model.code, model.distributed)
                            """Loads the model to a Tensorflow model"""

                            input_shape = str(tensorflow_model.input_shape)

                            sub = re.search(', (.+?)\)', input_shape)
```

Consola 16: Implementación de InferenceResultID 1/2.

```python
                        if sub:
                            shape = sub.group(1)

                            dictionary = json.loads(input_config)

                            dictionary['data_reshape'] = shape

                            new_input_config = json.dumps(dictionary)

                            response['input_config'] = new_input_config
                        else:
                            response['input_config'] = input_config

            return HttpResponse(json.dumps(response), status=status.HTTP_200_OK)
        except Exception as e:
            traceback.print_exc()
            return HttpResponse('Result not found', status=status.HTTP_400_BAD_REQUEST)
```

Consola 17: Implementación de InferenceResultID 2/2.

7.2. Módulo frontend

En esta sección se detallará la implementación de los distintos componentes que forman el frontend. En este caso, para la creación de interfaces web son necesarios dos tipos de archivos: un archivo HTML que se encarga de desarrollar una descripción sobre los contenidos que aparecen como textos y sobre su estructura, complementando dicho texto con diversos objetos, y un archivo TypeScript que se encarga de guardar las funciones y variables globales que se ejecutarán en la página web.

A continuación se exponen algunos de los componentes que forman este módulo.

7.2.1. Model-list

HTML

Este componente es el encargado de mostrar la lista de redes neuronales registradas en el sistema. En él aparecen cada uno de los campos que forman las redes neuronales ordenadas por su identificador, junto con los botones de actualización y eliminación. La figura 15 muestra la interfaz web de Model-list.

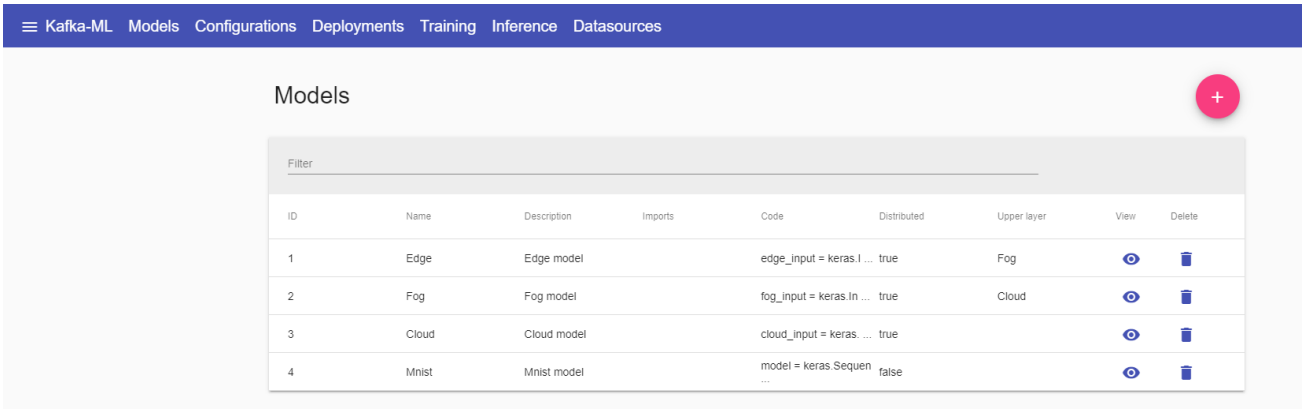

Figura 15: HTML Model-list.

TypeScript

El archivo TypeScript de Model-list está formado por una serie de funciones las cuales le permiten obtener la lista de redes neuronales existentes en el sistema, aplicar un filtro de búsqueda y eliminar una red neuronal concreta. Las consolas 18 y 19 muestran su implementación.

```
ngOnInit(): void {

  this.modelService.getModels().subscribe((data: JSON[])=>{
   this.models=data;
   this.dataSource.data=this.models;
  },(err)=>{
    this.snackbar.open('Error connecting with the server', ''), {
      duration: 3000
    }
  });
}

applyFilter(value: string) {
  value = value.trim().toLowerCase();
  this.dataSource.filter = value;
}
```

Consola 18: model-list.js 1/2.

```
delete(id: number) {
  this.modelService.deleteModel(id).subscribe(
    (data) => {},  //changed
    (err)=>{
      this.snackbar.open('Error deleting the model: '+err.error, '', {
        duration: 4000
      });
    },
    ()=>{
        this.snackbar.open('Model deleted', '', {
        duration: 3000
      });
      this.deleteRowDataTable(id);
    }
 );
}

deleteRowDataTable (id: number) {
  const itemIndex = this.dataSource.data.findIndex(obj => obj['id'] === id);
  console.log(itemIndex);
  this.dataSource.data.splice(itemIndex, 1);
  this.dataSource._updateChangeSubscription(); // <-- Refresh the datasource
}
```

Consola 19: model-list.js 2/2.

7.2.2. Model-view

HTML

Este componente es el encargado de la creación y actualización de las redes neuronales. En él aparece un formulario web en el cual se recogen todos los campos que forman una red neuronal. La figura 16 muestra la interfaz web de Model-view.

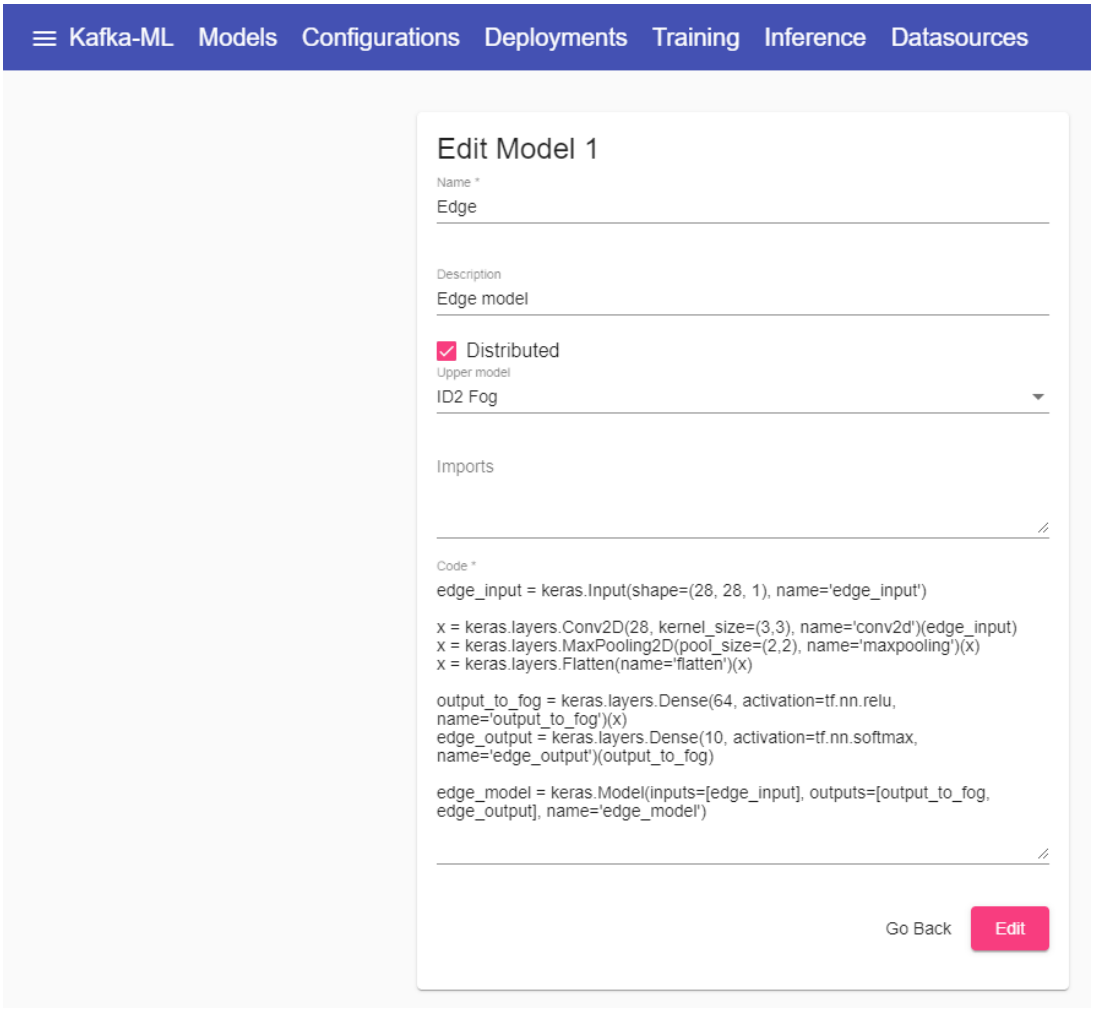

Figura 16: HTML Model-view.

TypeScript

El archivo TypeScript de Model-view está formado por una serie de funciones las cuales le permiten obtener la información de una red neuronal en el caso de que se esté editando, recuperar la lista de redes neuronales distribuidas para así asignar alguna de ellas como “padre” de la red neuronal que se esté creando o editando y enviar los nuevos datos introducidos para la creación o actualización de la red en cuestión. Las consolas 20 y 21 muestran su implementación.

```
ngOnInit(): void {
  // Get the ID in case of a edit request
  if (this.route.snapshot.paramMap.has('id')){
      this.modelId = Number(this.route.snapshot.paramMap.get('id'));
      this.create=false;
  }
  if (this.modelId!= undefined){
      this.modelService.getModel(this.modelId).subscribe(
        (data) => {
          this.model=<MLModel> data;
          this.showFather = this.model.distributed;
        },  //changed
        (err)=>{
          this.valid = false;
          this.snackbar.open('Error model not found', '', {
            duration: 3000
          });
        }
      );
  }
  this.modelService.getDistributedModels().subscribe(
    (data) => {
      this.distributedModels= data;
    },
    (err)=>{
      this.snackbar.open('Error connecting with the server', '', {
        duration: 3000
      });
    }
  );
}
```

Consola 20: model-view.js 1/2.

```
onSubmit(model: JSON) {
  if (this.modelId!= undefined){
    if (isNaN(model['father'])) {
      if (model['father'] != undefined) {
        model['father'] = model['father']['id'];
      }
    }
    this.modelService.editModel(this.modelId, model).subscribe(
      (data) => {},  //changed
      (err)=>{
        this.snackbar.open('Error updating the model: '+err.error, '', {
          duration: 3000
        });
      },
      ()=>{
        this.router.navigateByUrl('/models');
        this.snackbar.open('Model updated ', '', {
        duration: 3000
      });}
   );
  }
  else{
        this.modelService.createModel(model).subscribe(
          (data) => {},  //changed
          (err)=>{
            this.snackbar.open('Error creating the model: '+err.error, '', {
              duration: 3000
            });
          },
          ()=>{
            this.router.navigateByUrl('/models');
            this.snackbar.open('Model created ', '', {
            duration: 3000
          });}
       );
    }
}
```

Consola 21: model-view.js 2/2.

7.2.3. Inference-view

HTML

Este componente es el encargado de la creación y despliegue de una nueva inferencia asociada a un resultado de entrenamiento. En él aparece un formulario web que recoge toda la información necesaria para llevar a cabo dicho objetivo. La figura 17 muestra la interfaz web de Inference-view.

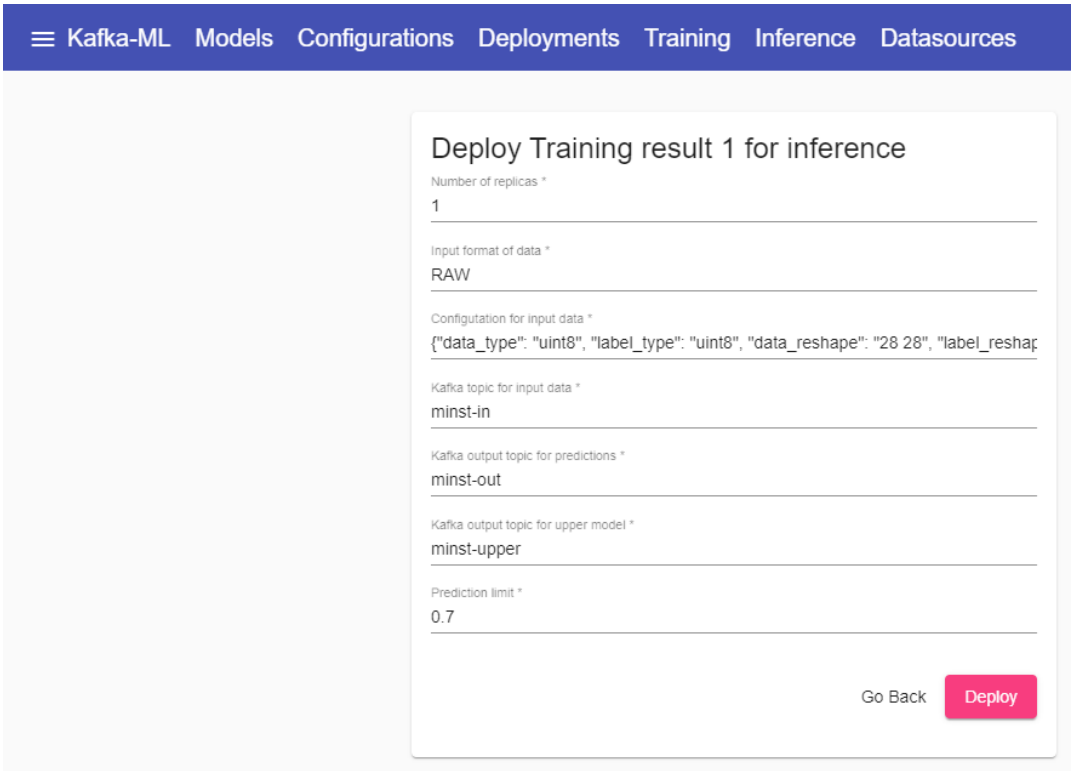

Figura 17: HTML Inference-view.

TypeScript

El archivo TypeScript de Inference-view está formado por una serie de funciones las cuales le permiten saber si se trata de una inferencia asociada a una red neuronal distribuida, recuperar el formato y configuración de entrada si existe otra inferencia creada previamente relacionada con la misma red neuronal, o en su defecto, recuperar esa información de la fuente de datos recibida para el entrenamiento previo, y por último realizar el despliegue de la inferencia. Las figuras 22 y 23 muestran su implementación.

```
ngOnInit(): void {
  if (this.route.snapshot.paramMap.has('id')) {
    this.resultID = Number(this.route.snapshot.paramMap.get('id'));
    this.resultService.getInferenceInfo(this.resultID).subscribe((data: JSON[]) => {
      if (data['input_format']!=''){
        this.inference.input_format = data['input_format'];
        this.inference.input_config = data['input_config'];
        this.snackbar.open('Input format and configuration found from another dataset/inference', '', {
          duration: 3000
        });
      }
      this.valid = true;
    },
    (err) => {
      this.snackbar.open('The training result does not exist', '', {
        duration: 3000
      });
    });
  }
  this.modelService.getModelResultID(this.resultID).subscribe(
    (data) => {
      this.model=<MLModel> data;
      this.distributed = this.model.distributed;
    },  //changed
    (err)=>{
      this.valid = false;
      this.snackbar.open('Error model not found', '', {
        duration: 3000
      });
    }
  );
}
```

Consola 22: inference-view.js 1/2.

```
deployInference(inference: Inference){
  inference.model_result = this.resultID;
  this.resultService.deployInference(this.resultID, inference).subscribe((data: JSON[]) => {
      this.snackbar.open('Model deployed for inference', '', {
        duration: 3000
      });
      this.router.navigateByUrl('/inferences');
    },
    (err) => {
      this.snackbar.open('Error deploying the model for inference', '', {
        duration: 3000
      });

  });
}
```

Consola 23: inference-view.js 2/2.

7.2.4. Servicios

El módulo frontend también contiene una clase por cada entidad en la que se definen los servicios que se encargan de llamar a las funciones implementadas en las vistas del backend expuestas anteriormente. Para ello se define una URL que conecta estos servicios con un grupo concreto de funciones de las vistas, que se diferencian entre sí por los distintos métodos que implementan cada una.

Las consolas 24 y 25 muestran los servicios definidos para la clase MLModel.

```
export class ModelService {

  baseUrl = environment.baseUrl;

  constructor(private httpClient: HttpClient) { }

    url = this.baseUrl + '/models/';

    getModels(){
      return this.httpClient.get<JSON[]>(this.url);
    }

    getDistributedModels(){
      const url = `${this.url}${'distributed'}`
      return this.httpClient.get<JSON[]>(url);
    }

    getFatherModels(){
      const url = `${this.url}${'fathers'}`
      return this.httpClient.get<JSON[]>(url);
    }

    createModel(data: JSON){
      return this.httpClient.post<JSON>(this.url, data)
    }

    getModel(id: number){
      const url = `${this.url}${id}`
      return this.httpClient.get(url);
    }
```

Consola 24: model.service.ts 1/2.

```
getModelResultID(id: number) {
  const url = `${this.url}${'result/'}${id}`
  return this.httpClient.get(url);
}

deleteModel(id: number){
  const url = `${this.url}${id}`
  return this.httpClient.delete<JSON>(url);
}

editModel(id: number, data: JSON){
  const url = `${this.url}${id}`
  return this.httpClient.put<JSON>(url, data);
}
```

Consola 25: model.service.ts 2/2.

7.3. Módulo registro de control de Kafka

Este componente desarrollado por el Grupo de Investigación ERTIS es el encargado de recibir un mensaje de control de Kafka en el que se define el identificador del despliegue que debe recibir los datos, la configuración de entrada y el conjunto total de datos para el entrenamiento o inferencia. Si todo va bien, este módulo envía esa información recibida al backend. La consola 26 muestra su implementación.

```
for msg in consumer:
    """Gets a new message from Kafka control topic"""
    logging.info("Message received in control topic")
    logging.info(msg)
    try:
      deployment_id = int.from_bytes(msg.key, byteorder='big')
      """Whether the deployment ID received matches the received in this task, then it is a datasource for this task."""

      data = json.loads(msg.value)
      """ Data received from Kafka control topic. Data is a JSON with this format:
          dic={
              'topic': ..,
              'input_format': ..,
              'input_config' : ..,
              'validation_rate' : ..,
              'total_msg': ..
              'description': ..,
          }
      """

      retry = 0
      data['deployment'] = str(deployment_id)
      data['input_config'] = json.dumps(data['input_config'])
      data['time'] = datetime.datetime.utcfromtimestamp(msg.timestamp/1000.0).strftime("%Y-%m-%dT%H:%M:%S%Z")
      ok = False
      logging.info("Sending datasource to backend: [%s]", data)
      while not ok and retry < RETRIES:
        try:
          request = Request(url, json.dumps(data).encode(), headers={'Content-type': 'application/json'})
          with urlopen(request) as resp:
            res = resp.read()
            ok = True
            resp.close()
          logging.info("Datasource sent to backend!!")

          consumer.commit()
          """commit the offset to Kafka after sending the data to the backend"""
```

Consola 26: logger.py

7.4. Módulo de entrenamiento

Este módulo, el cual se despliega como un *job* en Kubernetes, es el encargado de realizar el entrenamiento de las redes neuronales distribuidas. Lo primero que hace es recuperar ciertas variables de entorno que son necesarias para llevar a cabo dicho objetivo. Después se conecta con Kafka a través de un consumidor y un productor los cuáles reciben el conjunto de datos de entrenamiento y envían los resultados respectivamente. Por último, este componente lleva a cabo el propio entrenamiento de las redes neuronales distribuidas y recopila los resultados para enviárselos al backend. Para ello lo que hace es crear una red neuronal global formada por todo el conjunto de redes distribuidas implicadas, y ésta es la red que se entrena, entrenando a su vez a todas las redes distribuidas que la forman. Las consolas 27, 28, 29, 30, 31, 32 y 33 muestran su implementación.

```python
def load_environment_vars():
    """Loads the environment information received from dockers
    boostrap_servers, result_url, result_update_url, control_topic, deployment_id, batch, kwargs_fit
    Returns:
        boostrap_servers (str): list of boostrap server for the Kafka connection
        result_url (str): URL for downloading the pre model
        result_id (str): Result ID of the model
        control_topic(str): Control topic
        deployment_id (int): deployment ID of the application
        batch (int): Batch size used for training
        kwargs_fit (:obj:json): JSON with the arguments used for training
        kwargs_val (:obj:json): JSON with the arguments used for validation
    """

    bootstrap_servers = os.environ.get('BOOTSTRAP_SERVERS')
    result_url = eval(os.environ.get('RESULT_URL'))
    result_id = eval(os.environ.get('RESULT_ID'))
    N = len(result_id)
    control_topic = os.environ.get('CONTROL_TOPIC')
    deployment_id = int(os.environ.get('DEPLOYMENT_ID'))
    batch = int(os.environ.get('BATCH'))
    kwargs_fit = json.loads(os.environ.get('KWARGS_FIT').replace("'", '"'))
    kwargs_val = json.loads(os.environ.get('KWARGS_VAL').replace("'", '"'))

    return (bootstrap_servers, result_url, result_id, control_topic, deployment_id, batch, kwargs_fit, kwargs_val, N)
```

Consola 27: load_environment_vars()

```python
def get_train_data(boostrap_servers, kafka_topic, group, batch, decoder):
    """Obtains the data and labels for training from Kafka

    Args:
        boostrap_servers (str): list of boostrap servers for the connection with Kafka
        kafka_topic (str): Kafka topic   out_type_x, out_type_y, reshape_x, reshape_y) (raw): input data
        batch (int): batch size for training
        decoder(class): decoder to decode the data

    Returns:
        train_kafka: training data and labels from Kafka
    """

    logging.info("Starts receiving training data from Kafka servers [%s] with topics [%s]", boostrap_servers,  kafka_topic)
    train_data = kafka_io.KafkaDataset([kafka_topic], servers=boostrap_servers, group=group, eof=True, message_key=True).map(lambda x, y: decoder.decode(x, y)).batch(batch)

    return train_data
```

Consola 28: get_train_data(args**)

```python
PRE_MODEL_PATHS = []
'''Paths of the received pre-models'''
for i, url in enumerate(result_url, start=1):
    path='pre_model_{}.h5'.format(i)
    PRE_MODEL_PATHS.append(path)

    download_model(url, path, RETRIES, SLEEP_BETWEEN_REQUESTS)
    """Downloads the model from the URL received and saves in the filesystem"""

tensorflow_models = []
for path in PRE_MODEL_PATHS:
    tensorflow_models.append(load_model(path))
    """Loads the model from the filesystem to a Tensorflow model"""
```

Consola 29: Recuperación de las redes neuronales distribuidas.

```
"""TENSORFLOW code goes here"""
outputs = []
img_input = tensorflow_models[0].input
outputs.append(tensorflow_models[0](img_input))
for index in range(1, N):
    next_input = outputs[index-1]
    outputs.append(tensorflow_models[index](next_input[0]))
    """Obteins all the outputs from each distributed submodel"""

predictions = []
for index in range(0, N-1):
    s = outputs[index]
    predictions.append(s[1])
predictions.append(outputs[-1])
"""Obteins all the prediction outputs from each distributed submodel"""

model = keras.Model(inputs=[img_input], outputs=predictions, name='model')
"""Creates a global model consisting of all distributed submodels"""

weights = {}
for m in tensorflow_models:
    weights[m.name] = 'sparse_categorical_crossentropy'
    """Sets the format of true labels"""

learning_rates = []
for index in range (0, N):
    learning_rates.append(0.001)
    """Sets the value 0.001 as the learning rate from each distributed model"""

model.compile(optimizer='adam', loss=weights, metrics=['accuracy'], loss_weights=learning_rates)
"""Compiles the global model"""

model_trained = model.fit(train_dataset, **kwargs_fit)
"""Trains the model"""

logging.info("Model trained history: %s", str(model_trained.history))
```

Consola 30: Formación de la red neuronal global y su entrenamiento.

```
train_losses = []
for m in tensorflow_models:
    s = m.name
    s += '_loss'
    train_losses.append(model_trained.history[s][-1])
    """Get last value of losses"""

logging.info("Models trainned! Losses: %s", str(train_losses))

if validation_size > 0:
    logging.info("Models ready to evaluation with configuration %s", str(kwargs_val))
    evaluation = model.evaluate(validation_dataset, **kwargs_val)
    """Validates the models"""
    logging.info("Model trained evaluation: %s", str(evaluation))
    logging.info("Models evaluated!")

retry = 0
finished = False

dictionaries = []
metrics = []
for m in tensorflow_models:
    metrics_dic = {}
    train_metrics = ""
    for key in model_trained.history.keys():
        if not 'loss' in key and m.name in key:
            metrics_dic[key] = model_trained.history[key][-1]
            train_metrics += key+": "+str(round(model_trained.history[key][-1],10))+"\n"
            """Get all metrics except the loss"""
    dictionaries.append(metrics_dic)
    metrics.append(train_metrics)
```

Consola 31: Evaluación y recopilación de los resultados 1/2.

```
while not finished and retry < RETRIES:
    try:
        TRAINED_MODEL_PATHS = []
        for i in range (1, N+1):
            path = 'trained_model_{}.h5'.format(i)
            TRAINED_MODEL_PATHS.append(path)

        for m, p in zip(tensorflow_models, TRAINED_MODEL_PATHS):
            m.save(p)
            """Saves the trained models in the filesystem"""

        files = []
        for p in TRAINED_MODEL_PATHS:
            files_dic = {'trained_model': open(p, 'rb')}
            files.append(files_dic)

        results_list = []
        for loss, metrics in zip(train_losses, metrics):
            results = {
                'train_loss': round(loss,10),
                'train_metrics': metrics,
            }
            results_list.append(results)

        if validation_size > 0:
            """if validation has been defined"""
            for i, r in enumerate(results_list, start=1):
                r['val_loss'] = float(evaluation[i])
                r['val_metrics'] = ''
                dic = dictionaries[i-1]
                j = i + N
                for key in dic:
                    r['val_metrics'] += key+": "+str(round(evaluation[j],10))+"\n"
                    j += N
```

Consola 32: Recopilación de resultados 2/2.

```
    responses = []
    for (result, url, f) in zip(results_list, result_url, files):
        data = {'data' : json.dumps(result)}
        logging.info("Sending result data to backend")
        r = requests.post(url, files=f, data=data)
        responses.append(r.status_code)
        """Sends the training results to the backend"""

    if responses[0] == 200 and len(set(responses)) == 1:
        finished = True
        datasource_received = True
        logging.info("Results data sent correctly to backend!!")
    else:
        time.sleep(SLEEP_BETWEEN_REQUESTS)
        retry+=1
except Exception as e:
    traceback.print_exc()
    retry+=1
    logging.error("Error sending the results to the backend [%s].", str(e))
    time.sleep(SLEEP_BETWEEN_REQUESTS)
consumer.close(autocommit=True)
```

Consola 33: Envío de resultados al backend.

7.5. Módulo de inferencia

Este módulo es el encargado de realizar la inferencia de las redes neuronales distribuidas. Lo primero que hace es recibir el conjunto de datos a inferir a través de un consumidor de Kafka, después realiza la predicción y por último, a través de un productor de Kafka envía los resultados al backend. Dependiendo del resultado de la predicción que se compara con un límite, los resultados se envían a un tópico o a otro. Si la predicción supera o iguala al límite, la predicción se considera correcta y los resultados se envían al tópico de salida y eso significa que no hace falta pasar a la siguiente red distribuida; y si la predicción es menor que dicho límite, se deben enviar los datos al tópico del que esté leyendo la siguiente red neuronal distribuida para que así vuelva a realizar una nueva predicción. La consola 34 muestra su implementación.

```python
for msg in consumer:
  try:
    logging.info("Message received for prediction")

    input_decoded = decoder.decode(msg.value)
    """Decodes the message received"""

    prediction_to_upper, prediction_output = model.predict(np.array([input_decoded]))
    """Predicts the data received"""

    prediction_value = prediction_output.tolist()[0]
    """Gets the prediction value"""

    logging.info("Prediction done: %s", str(prediction_value))

    response = {
      'values': prediction_value
    }
    """Creates the object response"""

    response_to_kafka = json.dumps(response).encode()
    """Encodes the object response"""

    if max(prediction_value) < limit:
      producer.send(output_upper, json.dumps(prediction_to_upper.tolist()).encode())
    else:
      producer.send(output_topic, response_to_kafka)
    """Sends the message to Kafka"""

    producer.flush()
    """Flush the message to be sent now"""

    logging.info("Prediction sent to Kafka")

    consumer.commit()
    """Commit the consumer offset after processing the message"""

  except Exception as e:
    traceback.print_exc()
    logging.error("Error with the received data [%s]. Waiting for new a new prediction.", str(e))
```

Consola 34: distributed_inference.py

Capítulo 8

Evaluación

Las pruebas de software son parte esencial del proceso de desarrollo de software. Esta parte del proceso tiene la función de detectar los errores de software lo antes posible. Concretamente las pruebas de software se pueden definir como una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas registrándose los resultados obtenidos. Seguidamente se realiza un proceso de evaluación en el que los resultados obtenidos se comparan con los resultados esperados para localizar fallos en el software. Estos fallos conducen a un proceso de depuración en el que es necesario identificar la falta asociada con cada fallo y corregirla, pudiendo dar lugar a una nueva prueba. Como resultado final se puede obtener una determinada predicción de fiabilidad o un cierto nivel de confianza en el software probado. El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, únicamente puede demostrar que existen defectos en el software.

8.1. Pruebas

Las pruebas realizadas tratan de medir el tiempo que toma la aplicación Kafka-ML a la hora de llevar a cabo las tareas de entrenamiento de redes neuronales distribuidas, junto con el tiempo que tarda en recibir y preparar el conjunto de datos de entrenamiento y el posterior envío de los resultados, y por otro lado el tiempo total de la fase de inferencia para redes distribuidas, que incluye el intervalo desde que se reciben los datos a inferir hasta que se envían cada una de las predicciones.

Para ello se ejecuta el ciclo de funcionamiento de la aplicación y se van recopilando todos los tiempos mencionados. Para cada una de las pruebas se han realizado tres

intentos con distintas redes neuronales, por lo que el tiempo que se muestra es el tiempo medio calculado entre los tres casos. Los resultados se exponen a continuación.

Prueba	**Tiempo (segundos)**
Recepción de los datos de entrenamiento	58.9147
Entrenamiento distribuido	34.5872
Envío de resultados al backend	0.2051
Inferencia distribuida	9.3505

Como se puede apreciar en la tabla, los tiempos obtenidos no son muy grandes para toda la cantidad de operaciones, comunicaciones y tratamiento de datos que hay de por medio. Sólo en el primer caso, a la hora de recibir los datos de entrenamiento, se alarga un poco más que en los demás, pero esto es debido a que justo antes de entrenar se debe realizar el despliegue del componente en cuestión y que éste se quede esperando un tiempo a que el usuario ejecute el cliente que envía el conjunto de datos, y después emplear algo más de tiempo en su recepción y en su tratamiento posterior necesario para que la red neuronal pueda trabajar con él.

Por otro lado destacar también que el tiempo empleado en el entrenamiento distribuido depende mucho de la configuración que elijamos para el despliegue, en la cual podemos decidir el número de épocas a entrenar y el número de pasos dentro de cada una, lo que hace que se pueda alargar o reducir considerablemente el tiempo final.

Capítulo 9

Conclusiones y líneas futuras

El resultado final de este Trabajo Fin de Grado ha sido el desarrollo de un entorno de trabajo y despliegue apoyado en una herramienta gráfica que permite a través de contenedores y una arquitectura Fog la distribución de redes neuronales profundas de forma flexible. El marco de desarrollo se compone de una serie de componentes los cuales permiten gestionar y entrenar redes neuronales distribuidas, y posteriormente realizar predicciones con ellas.

La gestión de la infraestructura de Fog Computing consta de dos partes muy bien diferenciadas: el plano de control, usado para la administración de la red de nodos, monitorización y actualizaciones; y el plano de datos, utilizado para gestionar el flujo de datos que va hacia y desde la nube. Las características que ofrece Fog Computing como la reducción de latencia y el ancho de banda son utilizadas en este trabajo y por las redes neuronales distribuidas para la optimización en su ejecución de forma distribuida.

En el futuro se pretende utilizar este sistema para realizar el despliegue real de redes neuronales distribuidas una vez entrenadas en distintas máquinas. Con esto se busca dotar de inteligencia a dispositivos IoT para que trabajen conjuntamente enviando y recibiendo datos entre sí, en el ámbito que se desee, con la idea de que sean capaces no sólo de captar información del entorno que les rodea sino también de procesarla y poder generar respuestas a esos estímulos de manera más rápida antes de enviar esa información a la nube, aunque puedan ser resultados menos precisos.

Además, otra línea de investigación futura podría consistir en la posibilidad de diseñar y generar distintos modelos para dispositivos IoT a través de la herramienta web Kafka-ML.

Apéndice A

Manual de usuario del framework Kafka-ML

A continuación, se va a exponer el funcionamiento del framework web Kafka-ML, el cuál permite trabajar con modelos de redes neuronales desde su registro en la herramienta, pasando por la fase de entrenamiento y por último la fase de inferencia de dichas redes.

La estructura mínima son las redes neuronales. Estas redes se pueden conectar entre sí, en el caso de que sean distribuidas, y podemos crear configuraciones que engloben a un conjunto de ellas de manera que se pueda realizar su posterior despliegue. Una vez en este punto el usuario puede enviar datos de entrenamiento a estos despliegues para realizar tanto la fase de entrenamiento como la fase de predicción.

A.1. Inicio de la aplicación web

Lo primero que tenemos que hacer para arrancar el servidor es abrir Docker y configurar la aplicación tal y como se explica en el fichero *README.md*, el cual se encuentra en la carpeta raíz del proyecto. En resumen, tendremos que crear un registro local para ir subiendo las imágenes de los distintos componentes y por último desplegarlos.

Una vez arrancada la aplicación web a través de la línea de comandos de nuestro ordenador, observamos en la figura 18 como el terminal nos indica que podemos acceder a ella mediante la siguiente dirección: *http://localhost:4200/*

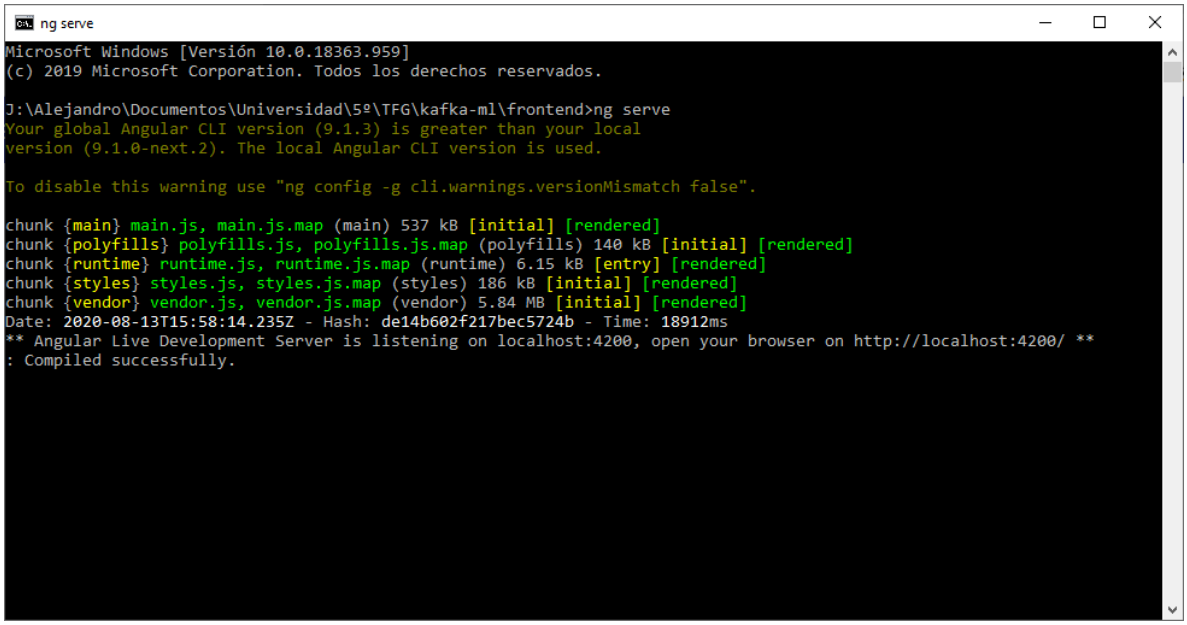

Figura 18: Información consola Kafka-ML.

Al acceder con un navegador a la dirección indicada, podemos ver en la figura 19 como se muestra la interfaz de la aplicación. Ésta se divide en las siguientes secciones: Models, Configurations, Deployments, Training, Inference y Datasources.

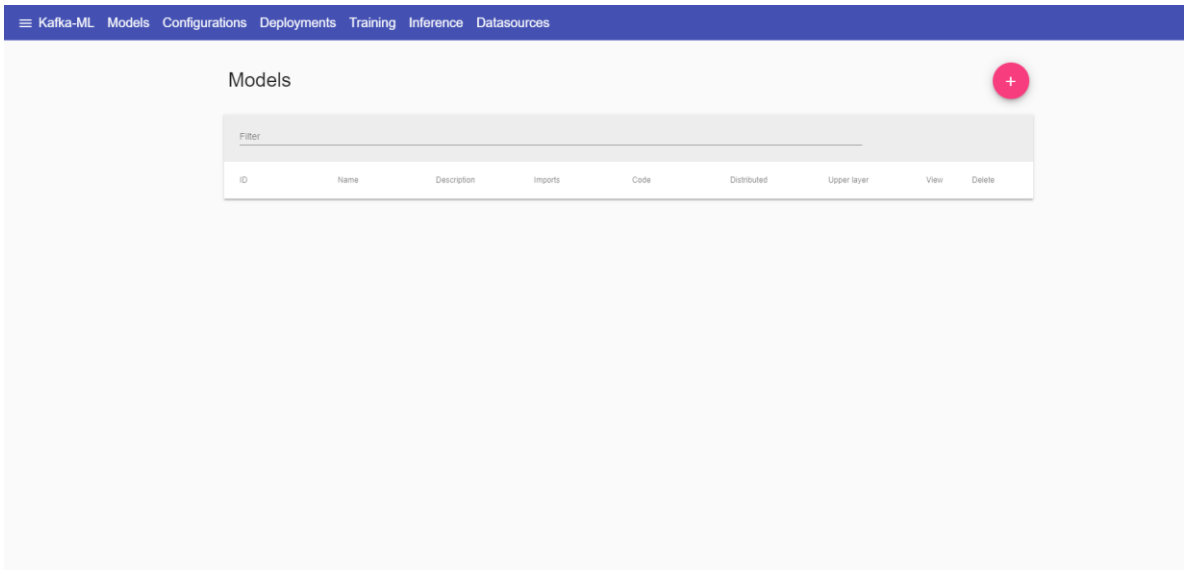

Figura 19: Página principal.

A.2. Registro de redes neuronales

Una vez conocidas las partes principales de la interfaz procederemos a realizar un flujo completo de uso de la herramienta.

Para registrar una red neuronal lo primero que tenemos que hacer es pulsar el botón de *"Add a model"* de la página principal que nos llevará al formulario de creación de modelos. Éste nos pedirá un nombre, una descripción, el código de la red neuronal, los imports necesarios para ejecutarla y por último un checkbox para indicar si el modelo va a ser distribuido o no junto con una lista desplegable con todas las redes existentes para, en el caso de que sea una red distribuida, establecer quién es su capa "padre", es decir, aquella que se encuentra por encima. La figura 20 muestra un ejemplo de creación de una red neuronal distribuida.

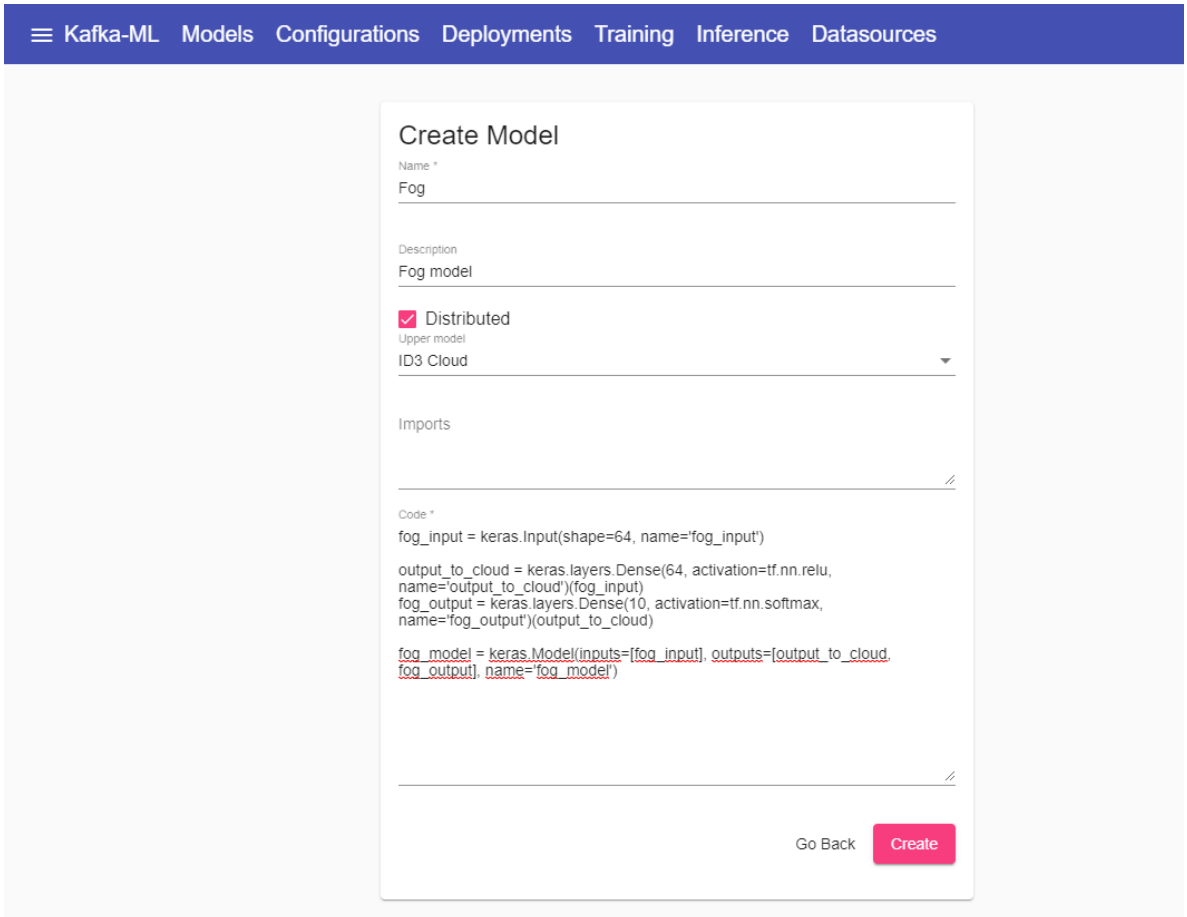

Figura 20: Creación de una red neuronal.

Cuando se pulsa el botón de *"Create"* nos redirige al listado principal de modelos y ahí podemos observar que se ha creado correctamente.

A.3. Creación de configuraciones

Para crear una nueva configuración nos desplazamos a la sección de configuraciones y pulsamos el botón de *"Add a configuration"* que nos llevará al formulario de creación. El único propósito de las configuraciones es agrupar un conjunto de modelos para su posterior despliegue. El formulario nos pedirá el nombre, descripción y el grupo de redes neuronales que la formen. La figura 21 muestra un ejemplo de creación de una configuración.

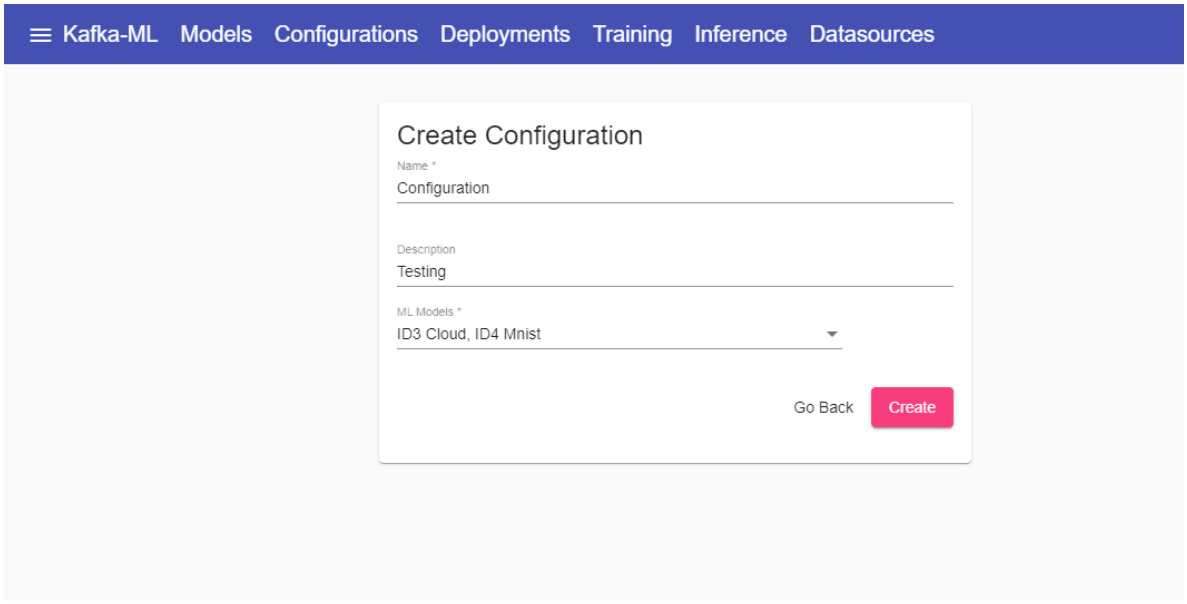

Figura 21: Creación de una configuración.

Una vez introducidos los datos pulsamos el botón de *"Create"* que nos redirigirá a la página con el listado de configuraciones existentes, donde comprobaremos que se ha creado correctamente.

A.4. Creación de despliegues

Una vez creada una configuración ya se puede realizar su despliegue. Para ello nos vamos a la sección de configuraciones y pulsamos sobre los tres pequeños puntos de la configuración que queramos y pulsamos el botón de *"Deploy"*, que nos lleva al formulario de creación de despliegues. La figura 22 muestra un ejemplo.

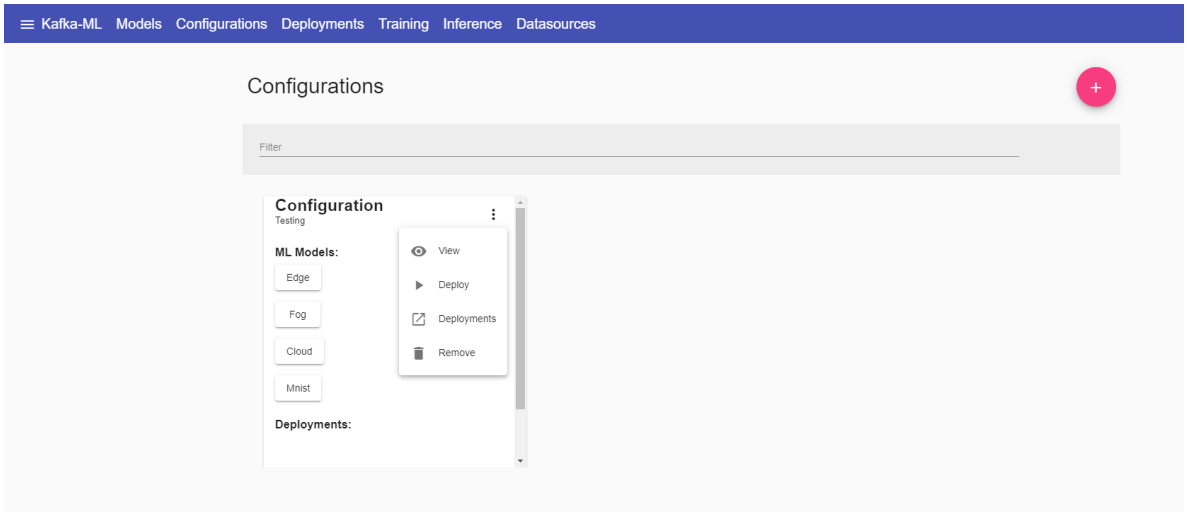

Figura 22: Despliegue de una configuración.

El formulario nos pedirá el tamaño de lote y la configuración para el entrenamiento y la configuración para la evaluación. Una vez introducidos los datos pulsamos el botón de *"Deploy"* que nos llevará a la página con el listado de despliegues existentes, donde podremos comprobar que se ha creado correctamente. La figura 23 muestra un ejemplo de creación de un despliegue.

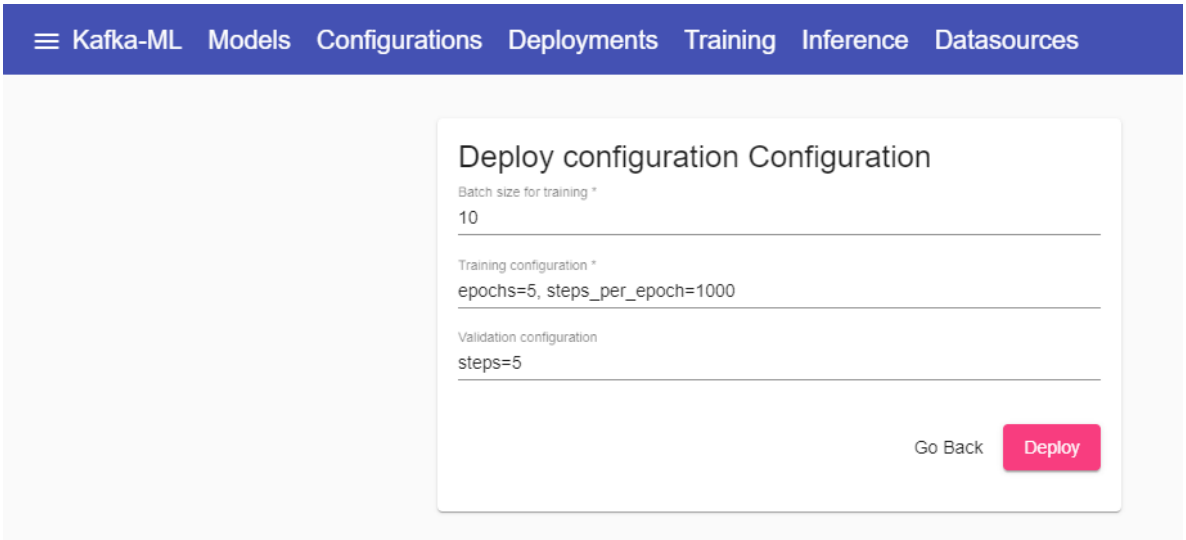

Figura 23: Creación de un despliegue.

A.5. Fase de entrenamiento

Una vez creado un despliegue ya podemos ver desplegados los distintos modelos que lo forman. Desde la pantalla de despliegues pulsamos sobre los tres pequeños puntos del despliegue que queramos para ver sus resultados. La figura 24 muestra un ejemplo.

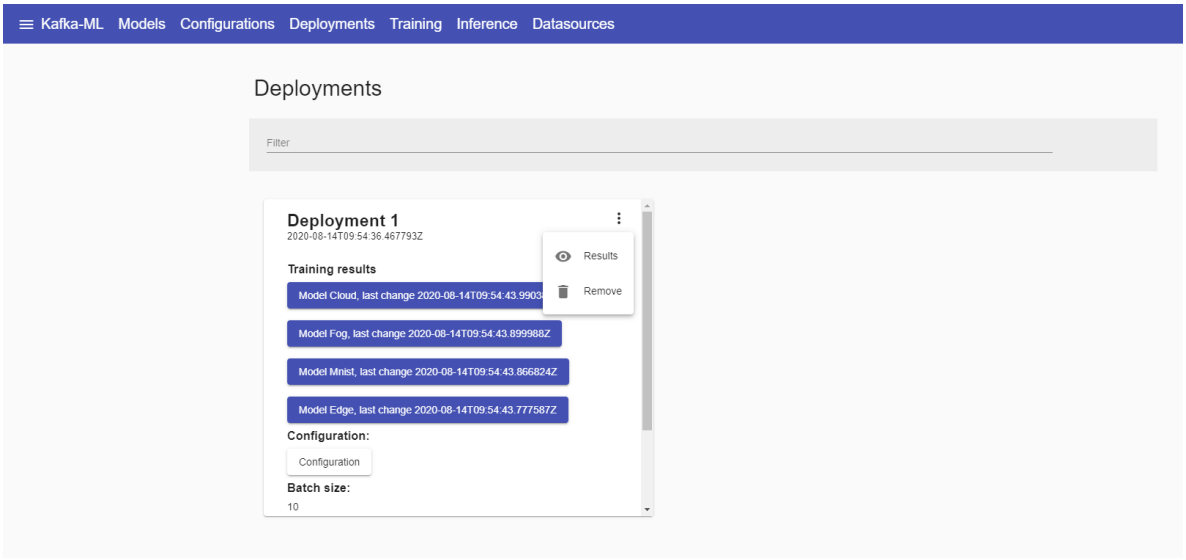

Figura 24: Listado de despliegues existentes.

La figura 25 muestra los modelos desplegados, los cuales se quedan esperando datos de entrada para comenzar la fase de entrenamiento.

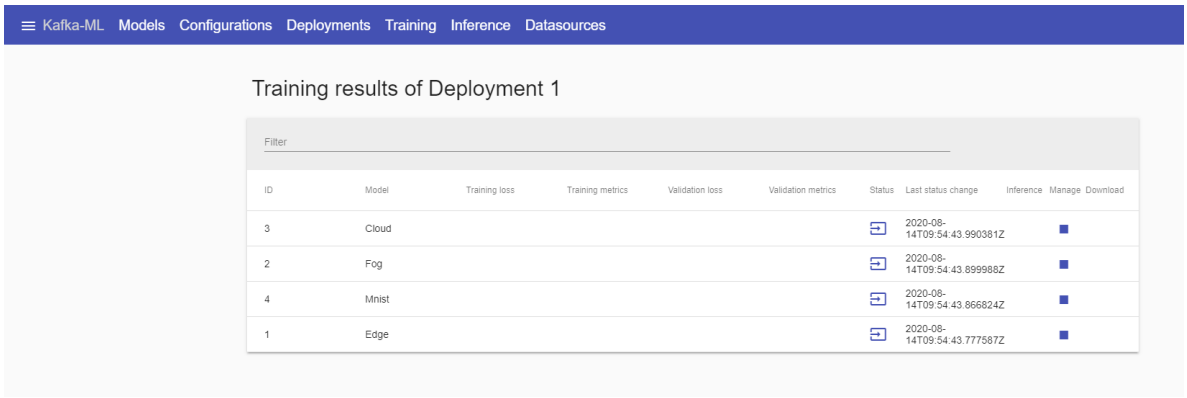

Figura 25: Lista de modelos desplegados.

En este punto el usuario debe enviar los datos de entrenamiento a los modelos desplegados. Para ello debe ejecutar un cliente Kafka que se encargue de dicha tarea, especificando una serie de parámetros, tal y como se muestra en la figura 26.

mnist_dataset_training_example.py

examples > MINST_RAW_format > mnist_dataset_training_example.py > ...

```python
import sys
sys.path.append(sys.path[0] + "/../..")
"""To allow importing datasources"""

from datasources.raw_sink import  RawSink
import tensorflow as tf
import logging

logging.basicConfig(level=logging.INFO)

mnist = RawSink(boostrap_servers='127.0.0.1:9094', topic='automl', deployment_id=1,
                description='Mnist dataset', validation_rate=0.1,
                data_type='uint8', label_type='uint8', data_reshape='28 28')

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print("train: ", (x_train.shape, y_train.shape))

for (x, y) in zip(x_train, y_train):
  mnist.send(data=x.tobytes(), label=y.tobytes())

for (x, y) in zip(x_test, y_test):
  mnist.send(data=x.tobytes(), label=y.tobytes())

mnist.close()

```

Figura 26: Cliente Kafka encargado de enviar los datos de entrenamiento.

Una vez ejecutado el cliente Kafka los modelos reciben los datos de entrada y comienzan a entrenarse. Una vez finalizado los resultados se muestran en el listado de los modelos desplegados. La figura 27 muestra un ejemplo.

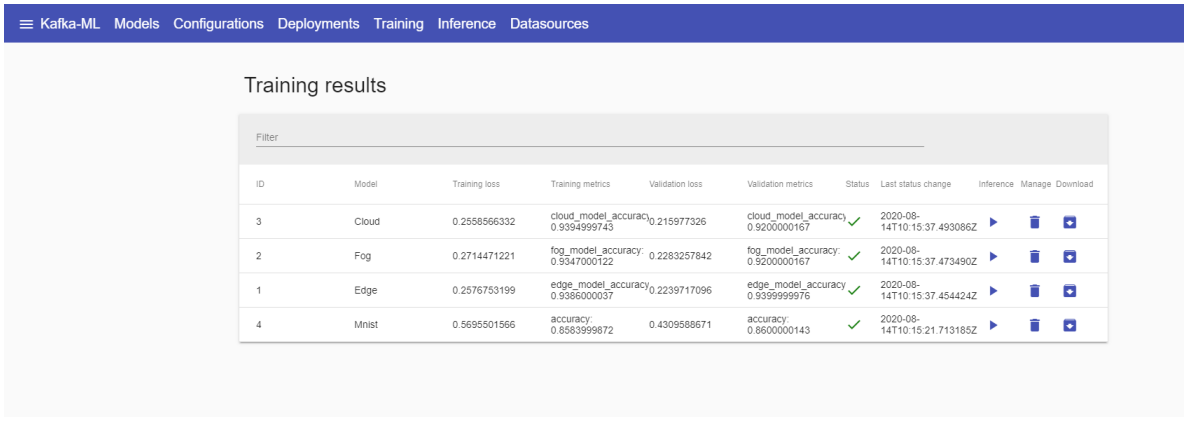

Figura 27: Resultados de entrenamiento.

En la sección de Datasources se pueden ver las fuentes de datos recibidas, y desde ahí, volver a enviárselas a otro despliegue.

A.6. Fase de inferencia

Una vez entrenados los modelos podemos pasar a realizar predicciones con ellos. Para ello desde la ventana con los resultados del entrenamiento pulsamos el botón de *"Deploy it for Inference"* del modelo con el que queramos inferir, que nos llevará al formulario de creación de inferencias. El formulario nos pedirá el número de réplicas de la inferencia, formatos y configuraciones de entrada (estos campos se rellenan solos si la aplicación encuentra inferencias anteriores del mismo modelo o si no las recupera de la configuración de la fuente de datos), el tópico de entrada para leer los datos y los de salida para escribir los resultados (si no se trata de una red distribuida tiene sólo un tópico de salida) y por último un límite numérico para, a partir del cual, considerar una predicción correcta. La figura 28 muestra un ejemplo de despliegue de un resultado para inferir.

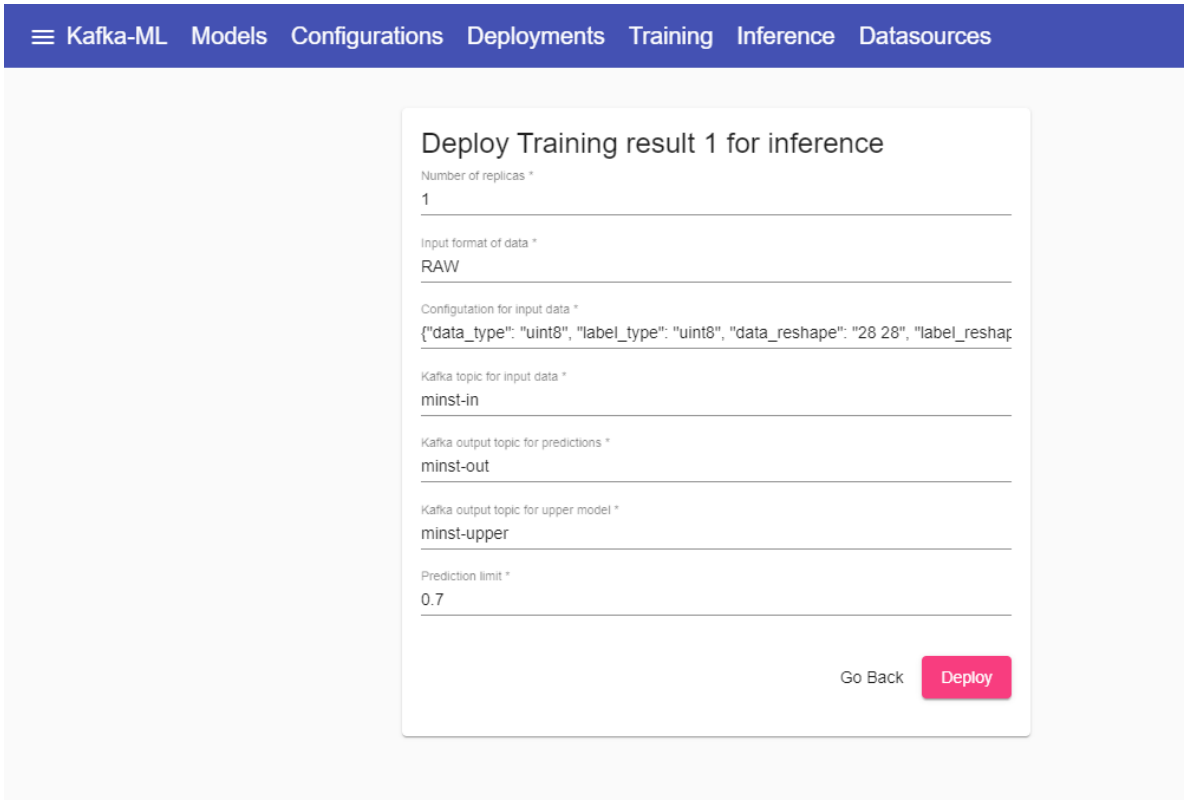

Figura 28: Despliegue de un resultado para inferir.

Una vez pulsamos el botón de *"Deploy"* la aplicación nos redirige al listado de inferencias existentes donde podemos comprobar que se ha creado correctamente. Los resultados desplegados para inferir se encuentran ahora esperando los datos a evaluar, los cuales deben ser enviados por el usuario a través de otro cliente Kafka. La figura 29 muestra un ejemplo.

mnist_dataset_inference_example.py

examples > MINST_RAW_format > mnist_dataset_inference_example.py > ...

```python
import tensorflow as tf
import logging
from kafka import KafkaProducer, KafkaConsumer

logging.basicConfig(level=logging.INFO)

INPUT_TOPIC = 'minst-in'
OUTPUT_TOPIC = 'minst-out'
BOOTSTRAP_SERVERS= '127.0.0.1:9094'
ITEMS_TO_PREDICT = 10

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print("Datasize minst: ", x_test.shape)

producer = KafkaProducer(bootstrap_servers=BOOTSTRAP_SERVERS)
"""Creates a producer to send the values to predict"""
for i in range (0, ITEMS_TO_PREDICT):
  producer.send(INPUT_TOPIC, x_test[i].tobytes())
  """Sends the value to predict to Kafka"""
producer.flush()
producer.close()

output_consumer = KafkaConsumer(OUTPUT_TOPIC, bootstrap_servers=BOOTSTRAP_SERVERS, group_id="output_group")
"""Creates an output consumer to receive the predictions"""

print('\n')

print('Output consumer: ')
for msg in output_consumer:
  print (msg.value.decode())
```

Figura 29: Cliente Kafka encargado de enviar los datos a inferir.

Para visualizar los resultados de la inferencia el usuario debe leer de los tópicos de salida que especificó anteriormente, en el caso de que estemos trabajando con una red neuronal distribuida. Existen dos posibilidades, o que el modelo haya escrito en el tópico de salida, lo cual significa que los resultados superan o igualan el límite, o que haya escrito en el tópico de la siguiente red neuronal para que se sigan procesando los datos. La figura 30 muestra un ejemplo en el que se pueden ver los resultados de las predicciones.

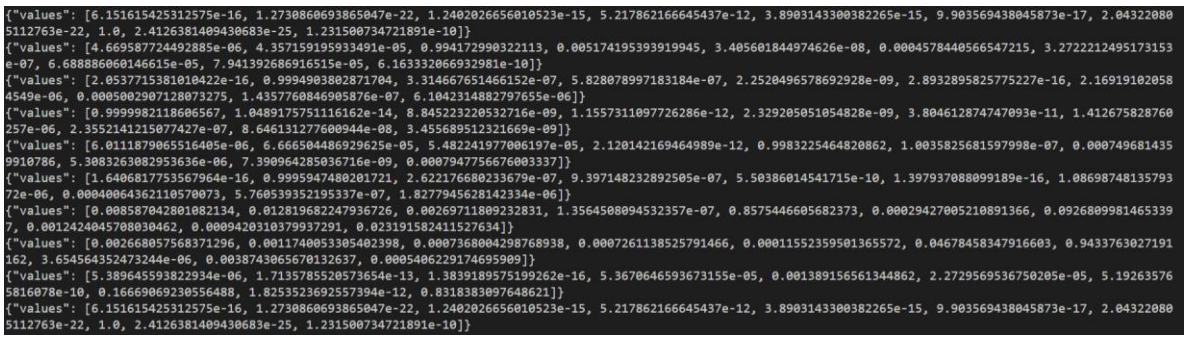

Figura 30: Predicciones de una inferencia.

Bibliografía

[1] «Red neuronal artificial», *Wikipedia, la enciclopedia libre*. jun. 19, 2020, Accedido: jun. 23, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Red_neuronal_artificial&oldid=127062057.

[2] «Desarrollo ágil de software», *Wikipedia, la enciclopedia libre*. ene. 13, 2020, Accedido: jun. 23, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Desarrollo_%C3%A1gil_de_software&oldid=122731968.

[3] «Python», *Wikipedia, la enciclopedia libre*. may 16, 2020, Accedido: jun. 29, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Python&oldid=126115328.

[4] «Qué es Python». https://desarrolloweb.com/articulos/1325.php (accedido jun. 23, 2020).

[5] «Angular (framework)», *Wikipedia, la enciclopedia libre*. may 25, 2020, Accedido: jun. 23, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Angular_(framework)&oldid=126367011.

[6] «TypeScript», *Wikipedia, la enciclopedia libre*. may 09, 2020, Accedido: jun. 23, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=TypeScript&oldid=125913536.

[7] «TensorFlow», *Wikipedia, la enciclopedia libre*. abr. 10, 2020, Accedido: jun. 23, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=TensorFlow&oldid=125078370.

[8] «Comunicando microservicios con Apache Kafka». https://www.paradigmadigital.com/dev/comunicacion-microservicios-apache-kafka/ (accedido jun. 25, 2020).

[9] «Docker (software)», *Wikipedia, la enciclopedia libre*. jun. 02, 2020, Accedido: jun. 25, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Docker_(software)&oldid=126606072.

[10] «Apache ZooKeeper», *Wikipedia, la enciclopedia libre*. abr. 03, 2020, Accedido: jun. 30, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Apache_ZooKeeper&oldid=124814751.

[11] «Conoce las 6 fases de un desarrollo de software a medida - Neosystems». http://www.neosystems.es/noticias/conoce-las-6-Fases-de-un-desarrollo-de-software-a-medida (accedido jun. 26, 2020).

[12] P. por pmoinformatica.com, «Requerimientos funcionales: Ejemplos». http://www.pmoinformatica.com/2017/02/requerimientos-funcionales-ejemplos.html (accedido jul. 01, 2020).

[13] «Requisito no funcional», *Wikipedia, la enciclopedia libre*. dic. 11, 2019, Accedido: jul. 01, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Requisito_no_funcional&oldid=121948326.

[14] Xavi, «Las cinco etapas de ingeniería del software», *Proyectos de Guerrilla*, feb. 10, 2013. http://proyectosguerrilla.com/blog/2013/02/las-cinco-etapas-en-la-ingenieria-del-software/ (accedido jul. 01, 2020).

[15] «Modelo de dominio», *Wikipedia, la enciclopedia libre*. ene. 02, 2020, Accedido: jul. 01, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Modelo_de_dominio&oldid=122442460.

[16] «Caso de uso», *Wikipedia, la enciclopedia libre*. mar. 24, 2020, Accedido: jul. 01, 2020. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Caso_de_uso&oldid=124533502.

[17] C. Martín, P. Langendoerfer, P. S. Zarrin, M. Díaz, y B. Rubio, «Kafka-ML: connecting the data stream with ML/AI frameworks», *ArXiv200604105 Cs*, jul. 2020, Accedido: ago. 25, 2020. [En línea]. Disponible en: http://arxiv.org/abs/2006.04105.

[18] S. User, «Implementación de software», *Applicatta*. https://www.applicatta.cl/index.php/soluciones/metodologia-applicatta/implementacion-de-software (accedido jul. 27, 2020).

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA