UNIVERSIDAD DE GRANADA

TRABAJO FIN DE MASTER
INGENIERÍA DE TELECOMUNICACIÓN

Algoritmos de Machine Learning para dispositivos LoRaWAN

Autor
Pablo Roas Domingo (alumno)

Director
Jorge Navarro Ortiz (tutor)

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, septiembre de 2021

Algoritmos de Machine Learning para dispositivos LoRaWAN

Autor
Pablo Roas Domingo (alumno)

Director
Jorge Navarro Ortiz (tutor)

Algoritmos de Machine Learning para dispositivos LoRaWAN

Pablo Roas Domingo

Palabras clave: IoT, LoRaWAN, SDN, Controlador Ryu, Machine Learning, Pytorch Forecasting.

Resumen

Gracias al IoT (*Internet of Things*) está cambiando la forma en la que la humanidad se relaciona con el mundo. Cualquier objeto físico es susceptible de ser conectado y, gracias al uso de sensores y actuadores, podemos obtener información y comunicarnos con ellos. Esto conlleva el surgimiento de nuevas tecnologías para hacer frente a los nuevos requerimientos de estos dispositivos y de sus aplicaciones, como es el caso de las redes LPWAN, por ejemplo.

Al mismo tiempo, las redes tradicionales también están evolucionando hacia paradigmas más dinámicos y ágiles, en los cuales se facilite su gestión y operabilidad por medio del uso de abstracciones. Este es el caso de SDN (*Software Defined Networking*), que se basa en la separación del plano de datos y el de control de los equipos de la red. Este concepto puede servir para tratar de manera adecuada los nuevos patrones de tráfico generados por los dispositivos IoT.

Por último, las técnicas para tratar datos, analizarlos y obtener conocimiento de ellos, también están experimentando un gran desarrollo. Si unimos la enorme cantidad de datos obtenidos gracias a los sensores de IoT y los modelos inteligentes que se basan, por ejemplo, en la experiencia, como es el caso del aprendizaje automático, las aplicaciones resultantes se pueden beneficiar en gran medida.

El presente trabajo se centra en integrar estas tres tecnologías: IoT, las redes SDNs y los algoritmos de *Machine Learning* (ML), presentando, a modo de ejemplo, una aplicación que obtiene datos del entorno por medio de sensores en un dispositivo IoT, envía los datos a través de una red definida por software y se procesan con una red neuronal para obtener predicciones temporales.

Machine Learning algorithms for LoRaWAN devices

Pablo Roas Domingo (student)

Keywords: IoT, LoRaWAN, SDN, Ryu Controller, Machine Learning, Pytorch Forecasting.

Abstract

Thanks to IoT (*Internet of Things*), the way in which humanity relates to the world is changing. Any physical object is capable of being connected, and thanks to the use of sensors and actuators, we can obtain information from the environment and interact with it. This causes new technologies to emerge in order to face the new requirements of these devices and their applications. This is the case of LPWAN networks.

At the same time, traditional networks are also changing to more dynamic and agile paradigms, in which their management and operability are easier, thanks to the use of abstractions. This is the case of SDN (*Software Defined Networking*), which is a paradigm that is based on the separation of the data plane and the control plane of the network. This concept can be used to adequately deal with new traffic patterns generated by IoT devices.

Finally, the techniques for processing, analyzing, and learning from data are also experiencing great development. If we combine the enormous amount of data obtained thanks to IoT sensors, and intelligent models that are based, for example, on experience, such as machine learning, the resulting applications can greatly benefit.

The present work focuses on integrating these three technologies; IoT, SDN and Machine Learning (ML) presenting, as an example, an application that obtains data from the environment through sensors in an IoT device, sends the collected data through a software-defined network and finally processes these data with a neural network to obtain temporal predictions.

Yo, **Pablo Roas Domingo**, alumno de la titulación MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Master en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Pablo Roas Domingo

Granada, septiembre de 2021.

D. **Jorge Navarro Ortiz**, Profesor del área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado **Algoritmos de Machine Learning para dispositivos LoRaWAN**, ha sido realizado bajo su supervisión por **Pablo Roas Domingo (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a X de mes de 2021.

El director:

Jorge Navarro Ortiz

Agradecimientos

A mi tutor Jorge, por ayudarme en la consecución de este proyecto.

Índice general:

Índice de figuras:

Índice de tablas:

Glosario de siglas:

ABP: Activation by Personalization

ACK: Acknowledge

API: Application Programming Interface

AWS: Amazon Web Services

CPU: Central Processing Unit

CSS: Chrip Spread Spectrum

DDR: Double Data Rate

DHCP: Dynamical Host Configuration Protocol

Diffserv: Differential Services

FAIR: Facebook Artificial Intelligence Research

GCP: Google Cloud Platform

GPU: Graphics Processing Unit

gRPC: Google Remote Procedure Calls

GW: Gateway

HTTP: Hyper Transfer Transport Protocol

IoT: Internet of Things

IP: Internet Protocol

ISM: Industrial Scientific and Medical

JSON: JavaScript Object Notation

LoRa: Long Range

LoRaWAN: Long Rate Wide Area Network

LPWAN: Low Power Wide Area Network

LSTM: Long Short-Term Memory

LTE-M: Long Term Evolution for Machines

M2M: Machine to Machine

MAC: Medium Access Control

ML: Machine Learning

MPLS: Multiprotocol Label Switching

MQTT: Message Queuing Telemetry Transport

N-BEATs: Neural Basis Expansion Analysis for interpretable Time Series forecasting

OTAA: Over the Air Activation

OVS: Open Virtual Switch

OVSDB: Open Virtual Switch Database Management Protocol

PC: Personal Computer

PHY: Physical

QoS: Quality of Service

RAM: Random Access Memory

ReLU: Rectified Linear Unit

REST: Representational State Transfer

RFC: Request for Comments

RJ45: Registered Jack-45

RNN: Recurrent Neural Network

SDN: Software Defined Network

SMA: SubMiniature version A

SQL: Structured Query Language

SSD: Solid State Drive

SSH: Secure SHell

SSL: Secure Sockets Layer

SVM: Support Vector Machines

TCP: Transmission Control Protocol

TFT: Temporal Fusion Transformer

TSDB: Time Series Data Base

TTN: The Things Network

VLAN: Virtual Local Area Network

WiFi: Wireless Fidelity

1. Introducción

Este capítulo pretende servir como introducción al proyecto. Para ello se incluyen las partes de contexto y motivación del proyecto, la definición de los objetivos del mismo, así como la estructura seguida en esta memoria.

1.1 Contexto

Para entender el ámbito del proyecto y las tecnologías tratadas en él, a continuación, se presentan cada una de ellas atendiendo a su utilidad. De esta manera, el trabajo en sí cobra sentido al enmarcarse en su contexto.

1.1.1 IoT

El Internet de las cosas (IoT por sus siglas en inglés) alude a una visión del mundo en el que los objetos de la vida cotidiana se conectan entre sí y con las personas por medio de Internet. Este Internet de las cosas abre la puerta a un gran número de nuevas aplicaciones que cambiarán la forma en la que las personas nos relacionamos con el mundo.

La idea básica de IoT es que virtualmente cualquier cosa física en este mundo puede convertirse también en un ordenador conectado a Internet; las denominadas cosas u objetos inteligentes.

En el Internet de las cosas, las denominadas “cosas inteligentes” se conectan gracias a la integración de pequeños chips con capacidades de comunicación, para comunicarse entre sí y con los usuarios; convirtiéndose en parte del Internet. Además, también se le añaden capacidades para interactuar con el mundo físico, como pueden ser el uso de sensores o actuadores.

Según la empresa CISCO, IoT surgió en el momento en el que la cantidad de cosas y objetos conectados a Internet superó al número de personas conectadas; por tanto, nació en algún punto entre 2008 y 2009 [1]. Además, sus previsiones son de 29.3 mil millones de dispositivos para el año 2023, con la mitad de ellos correspondientes a dispositivos de comunicaciones máquina a máquina [2].

El pequeño tamaño de los dispositivos sensores y actuadores, necesario para integrarlos en objetos del entorno, hace que sus capacidades de procesamiento, memoria y capacidades de comunicación se vean reducidas. Además, suelen tener pequeñas baterías, lo que implica que deben consumir poca energía. Estas limitaciones de recursos también son necesarias para reducir los costes de su fabricación [3].

Para aplicaciones IoT en las que se requiere un largo alcance en las comunicaciones, las tecnologías de comunicación de área local como WiFi o Bluetooth no cumplen los requerimientos de cobertura, y las tecnologías de redes celulares como LTE consumen demasiada energía. Para hacer frente a esto surge el concepto de redes LPWAN [4].

Bajo este término se engloban las denominadas redes de baja potencia de área amplia (*Low Power Wide Area Network*); en comparación con las tecnologías clásicas, persiguen el ahorro de energía usando una potencia baja con un alcance de gran distancia. Entre las tecnologías asociadas a este tipo de redes destacan:

Tecnología LPWAN		**Bandas de frecuencias**	**Alcance (zona urbana)**	**Velocidad**	**Downlink**
LoRa		ISM	5 km	50 kbps	Sí (diferentes planos)
SIGFOX		ISM	10 km	100 bps	Sí (no simétrico)
Weightless	W	Espacios en blanco de TV	5 km	10 Mbps	Sí (simétrico)
	N	Sub-GHz	5 km	100 bps	No
	P	Sub-GHz	2 km	100 kbps	Sí (simétrico)

Tabla 1.1: Comparación de tecnologías LPWAN.

En este proyecto en concreto se va a hacer uso de LoRaWAN como tecnología LPWAN para conectar un dispositivo sensor y obtener datos del entorno. LoRaWAN es una plataforma promovida por LoRa Alliance [5]. LoRaWAN define dos capas separadas. En primer lugar, la capa física (PHY), definida por la modulación LoRa. Y en segundo lugar, la capa de Control de Acceso al Medio (MAC), definida por el protocolo LoRaWAN. LoRaWAN propone una topología de estrella con pasarelas que conectan los nodos y la red central, donde los datos se almacenan y se ponen a disposición de entidades externas, que actúan como suscriptores. Los nodos se conectan a las pasarelas a través de enlaces inalámbricos de un salto utilizando la modulación LoRa (*Long Range*, del inglés largo alcance).

1.1.2 SDN

Aparte de los nuevos requerimientos en cuanto a las tecnologías para proporcionar conectividad a los dispositivos de IoT, también se experimentan cambios en el tráfico generado por ellos. Generalmente, un sistema IoT necesita de la interacción de miles de dispositivos, lo que convierte al IoT global en una red de ultra gran escala con miles de millones de dispositivos conectados. Este número tan elevado de dispositivos puede generar una cantidad enorme de eventos en la red y nuevos modelos de tráfico.

Además, IoT promueve la automatización de los procesos haciendo uso de comunicaciones M2M (*Machine to Machine*), lo que deriva en patrones de tráficos diferentes a los tradicionales, que estaban orientados a comunicaciones mayormente entre cliente y servidor.

Con un tráfico tan heterogéneo en la red, sería útil poder separar los diferentes tipos de tráfico para ofrecerles un servicio más acorde a sus requerimientos. Tradicionalmente, los conceptos que se han usado para hacer esto, son por ejemplo Diffserv o MPLS (*Multiprotocol Label Switching*), si bien, estas soluciones presentan algunas limitaciones.

Además de los requisitos asociados al IoT, las redes tradicionales también se quedan obsoletas ante nuevos paradigmas emergentes de servicios como por ejemplo la computación virtual en la nube, aplicaciones de *Big Data* o la entrega de contenido multimedia, entre otros.

Por ello, han surgido nuevas aproximaciones para satisfacer estos requerimientos, como es el caso del paradigma de las redes definidas por software [6,7]. SDN (*Software Defined Network*) se basa en proporcionar abstracción y modularidad para la gestión de la red. Para ello propone separar el plano de datos (encargado del reenvío de paquetes en los equipos de la red) y el plano de control (encargado de la gestión de la red). El plano de datos lo ejecutan los *switches*, mientras que el plano de control se ejecuta en los controladores de la red.

De este modo, SDN hace que el control de la red sea programable, centralizado y con una visión global de la topología, además de ágil y adaptable. Otra ventaja de separar plano de datos y de control, es que los dispositivos de red se simplifican, lo cual influye en el hardware subyacente de los mismos, resultando en equipos más baratos e interoperables.

La figura 1.1 muestra un esquema de la arquitectura por capas.

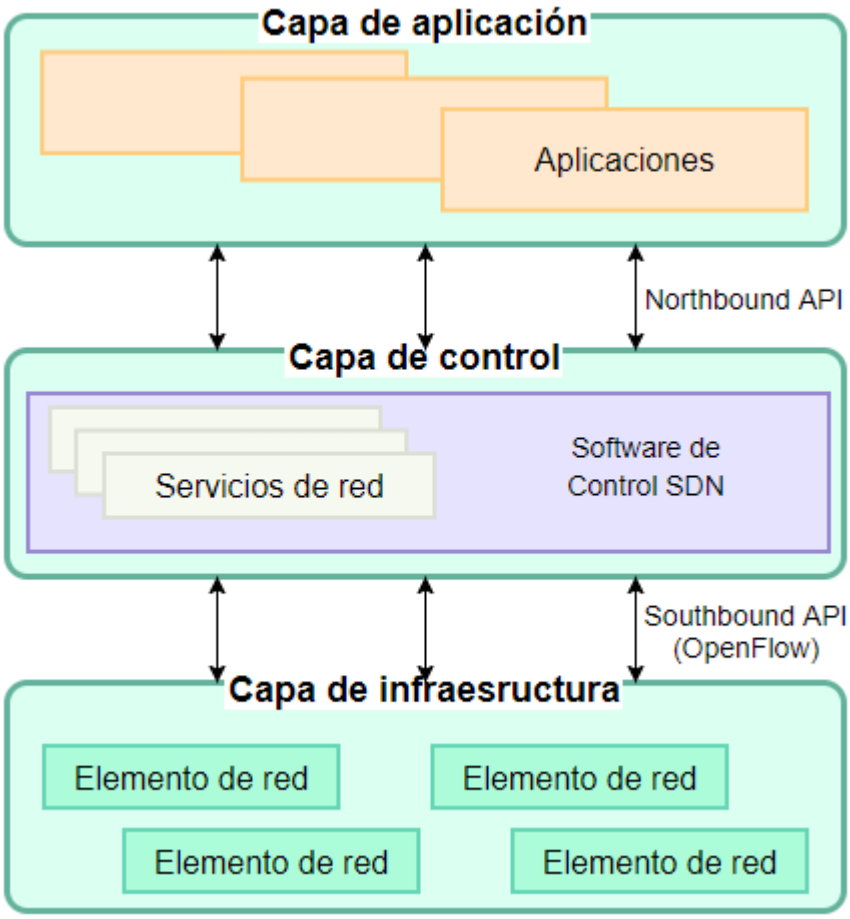


Figura 1.1: Arquitectura por capas de SDN.

El plano de control corresponde a la capa de control y de aplicación, y se encarga de todas las tareas de administración de la red, incluido el enrutamiento. Para ello toma decisiones sobre cómo y dónde se envía el tráfico, y gestiona toda la señalización de la red

para configurar correctamente los dispositivos de la misma. En SDN las decisiones de reenvío se basan en flujos, de forma que a todos los paquetes que coinciden con un criterio específico se les aplican las mismas acciones. Otras tareas que se incluyen en este plano son por ejemplo la ingeniería de tráfico o la aplicación de políticas de seguridad.

El plano de datos corresponde a la capa de infraestructura, y realiza el reenvío de tráfico en los dispositivos de red (*switches*) de acuerdo con las reglas definidas por el plano de control. Estas reglas a nivel de flujo pueden combinar acciones que tradicionalmente corresponden a diferentes tipos de dispositivos de red (por ejemplo, *routers*, *switches*, *firewalls*).

Las aplicaciones usan la API *NorthBound* para comunicarse con el plano de control, del cual se encarga el controlador, considerado como el "cerebro" de la red. En este proyecto se ha usado como controlador Ryu.

La interfaz entre el plano de control y el de datos se apoya en la API *SouthBound*, donde el controlador SDN se comunica con los equipos de la red en el plano de datos. El protocolo principal para comunicar controlador y equipos de red es OpenFlow, un protocolo estandarizado por la Open Networking Foundation (ONF).

1.1.3 Machine Learning

Podemos definir la Inteligencia Artificial como un subcampo de la Informática dedicado a la construcción de programas que exhiben aspectos del comportamiento inteligente denominados agentes [8]. Estos agentes son entidades capaces de percibir su entorno mediante sensores (entradas), procesar tales percepciones y actuar de manera racional en dicho entorno mediante efectores (salidas).

Existen muchos tipos de agentes. Algunas de las características deseables que deben tener son, por ejemplo, la autonomía, ya que deben poder actuar sin intervención humana y tener el control sobre sus propias acciones y estado interno; la racionalidad, ya que deben actuar con el fin de alcanzar sus objetivos maximizando el rendimiento; o el aprendizaje, para modificar su comportamiento en base a nuevas interacciones con el entorno.

Centrándonos en este último requerimiento, el aprendizaje es una capacidad fundamental de la inteligencia humana que nos permite adaptarnos a los cambios de nuestro entorno, desarrollar habilidades o mejorarlas y adquirir experiencia en nuevos dominios. Así surge el Aprendizaje Automático (*Machine Learning* o ML) como una rama de la Inteligencia Artificial.

El Aprendizaje Automático abarca programas que mejoran su comportamiento con la experiencia. El aprendizaje modifica el mecanismo de decisión del agente para mejorar su comportamiento y permite adquirir nuevo conocimiento. Es esencial en entornos desconocidos y normalmente se aprende a partir de datos.

Entre las acciones que realizan los programas de Aprendizaje Automático se incluyen: la selección de los datos relevantes y su transformación para tratarlos; el aprendizaje del

programa; la interpretación de resultados y evaluación de la eficacia y eficiencia del modelo aprendido. Es deseable que los modelos obtenidos sean comprensibles para el ser humano, que es usuario final de un sistema de aprendizaje.

De esta forma los programas y algoritmos de Aprendizaje Automático resultan útiles para situaciones en las que se produce una evolución constante de los datos, ya que los algoritmos de aprendizaje deben ser capaces de adaptarse a esta evolución con poco o ningún esfuerzo, reentrenando el sistema periódicamente, o actualizando el clasificador ya generado.

Uno de los puntos clave para el aprendizaje es el tipo de realimentación disponible en el proceso, según el cual podemos diferenciar las siguientes ramas dentro del Aprendizaje Automático:

- Aprendizaje supervisado: aprende una función a partir de ejemplos de sus entradas y salidas.
- Aprendizaje no supervisado: aprende o extrae conclusiones a partir de patrones de entradas para los que no se especifican los valores de sus salidas.
- Aprendizaje por refuerzo: aprende a partir del refuerzo que devuelve el entorno.

Figura 1.2: Tipos de aprendizaje automático.

En el aprendizaje supervisado, el objetivo del aprendizaje es la construcción de un modelo o clasificador que pueda servir para predecir la clase de futuros ejemplos. Dicho modelo debe construirse de manera que se maximice la efectividad del sistema, es decir, el

número de aciertos sobre los datos de entrenamiento, con la esperanza de que sea igual de efectivo sobre los futuros ejemplos.

Hay muchos algoritmos utilizados en aprendizaje supervisado como las redes neuronales, los árboles de decisión, SVM (*Support Vector Machines*), sistemas de obtención de reglas, las redes bayesianas, etc.

1.2 Motivación

Con la realización de este trabajo se pretende proporcionar un ejemplo de infraestructura que relacione y haga uso de tres tecnologías actualmente en desarrollo y crecimiento como lo son las redes LoRaWAN para IoT, las redes definidas por software (SDN) y los algoritmos de Aprendizaje Automático del campo de Inteligencia Artificial, concretamente las redes neuronales.

El resultado final, aunque esté compuesto por tres partes, se podrá entender como un todo. De esta manera, este trabajo servirá como base para futuros estudios que se realicen sobre el tema, proporcionando una manera de conectar estas tecnologías entre sí funcionalmente y facilitando que dichos estudios se centren en sus objetivos propios sin tener que atender a la forma en que se integran entre sí.

1.3 Objetivos

En esta sección se definen los objetivos que se pretenden alcanzar en el presente proyecto, de manera que se haga posible limitar el alcance del mismo, así como proporcionar una orientación sobre los aspectos necesarios de la implementación.

El objetivo principal, de acuerdo a lo recogido previamente en la motivación, se podría definir como el siguiente:

- Construir una implementación que integre las tecnologías de LoRa (IoT), SDN y *Machine Learning*, para hacer predicciones de algún parámetro medioambiental como e.g. la temperatura.

Para cumplir con dicho objetivo de manera segura y permitir una evaluación del estado de completitud del mismo; se han definido objetivos específicos que se recogen a continuación. Son los siguientes:

- Estudiar la tecnología LoRaWAN y conseguir que una mota capture datos del entorno usando sensores, y los envíe a la pasarela.
- Estudiar el concepto de SDN y conseguir que la pasarela LoRaWAN publique mensajes MQTT sobre una red SDN, permitiendo que un cliente MQTT guarde los datos en una base de datos.
- Estudiar algoritmos de *Machine Learning* para predicción y aplicar dichos algoritmos a los datos para hacer predicciones.

Es preciso mencionar que no se pretende inventar nada sino, más bien, conseguir unir y relacionar de una forma útil diferentes tecnologías.

1.4 Estructura de la memoria

En esta sección se recogen los capítulos en los que se estructura el presente documento y una breve explicación de su contenido:

1. Introducción: el primer capítulo de la memoria trata de presentar el proyecto; su contexto y motivación, así como la definición de objetivos.
2. Fundamentos teóricos: en este capítulo se trata de exponer los conceptos teóricos más importantes asociados a las tecnologías involucradas.
3. Herramientas utilizadas: contiene una breve explicación de las herramientas que se han usado en el proyecto.
4. Estado del arte: capítulo en el que se recogen soluciones similares o relacionadas con el objetivo de este proyecto.
5. Planificación, recursos y costes: este capítulo recoge la planificación temporal de las tareas llevadas a cabo, así como los recursos y gastos asociados.
6. Diseño e implementación: en este capítulo se explica en detalle cómo se ha desplegado la solución con las herramientas utilizadas.
7. Resultados: este capítulo recoge los resultados obtenidos.
8. Conclusiones y líneas futuras: último capítulo en el que se valora la realización del proyecto y se proponen los siguientes pasos a realizar o mejoras posibles.

2. Fundamentos teóricos

En este capítulo se incluye una explicación de los conceptos teóricos en los que se ha basado el proyecto. Se comienza explicando la tecnología LoRa y LoRaWAN, también se explican los protocolos MQTT (IoT) y OpenFlow (SDN) y, por último, se introducen varios conceptos de redes neuronales artificiales de los que se han hecho uso.

2.1 LoRa

LoRa [5] es una modulación inalámbrica para comunicaciones desarrollado por Semtech que se integra dentro de las tecnologías LPWAN. Utiliza una modulación *Chirp Spread Spectrum* (CSS) y mantiene el bajo consumo de energía propio de las modulaciones FSK pero con un alcance mayor. Los canales usados tienen un ancho de banda de hasta 250/500 kHz.

LoRa trabaja en bandas ISM (*Industrial, Scientific and Medical*), pero también se puede adaptar para admitir el espectro con licencia. Demodula señales hasta con una potencia de 19.5 dB por debajo del nivel de ruido, logrando así mayor alcance que los proporcionados por las tecnologías celulares. Dependiendo de la configuración de los parámetros de esta capa PHY, la velocidad de datos varia de 0.25 kbps hasta 50 kbps. La longitud máxima de la carga útil es de 242 Bytes, que, aunque quede limitada, puede satisfacer las demandas de los servicios típicos de IoT.

LoRaWAN, por su parte, incluye la capa de control de acceso al medio (MAC) y está desarrollada por la *LoRa Alliance*. LoRaWAN define también la arquitectura de red y el protocolo de comunicación entre los dispositivos que forman parte de la red.

La red LoRaWAN tiene una topología en estrella, apropiada para ahorrar energía en los dispositivos que se conectan a ella. En la figura 2.1 se muestra un esquema:

Una red LoRaWAN está compuesta por cuatro tipos de componentes:

- Nodos finales: son los dispositivos sensores o actuadores con requerimientos específicos de energía. Dentro de esta categoría, de acuerdo a las diferentes necesidades de ahorro de energía y de tasa de descarga, se definen los siguientes tipos de dispositivos:
    - Dispositivos clase A: solo pueden recibir transmisiones después de cada conexión de enlace ascendente abriendo una ventana de escucha (estrategia de transmisión iniciada por el receptor, bajo consumo de energía).
    - Dispositivos de clase B: pueden aceptar transmisiones de enlace descendente durante ventanas de enlace descendente programadas adicionales (estrategia de escucha coordinada muestreada, consumo medio de energía).
    - Dispositivos de clase C: escuchan continuamente el canal para que puedan recibir conexiones de enlace descendente en cualquier momento (estrategia de escucha continua, gran consumo de energía).

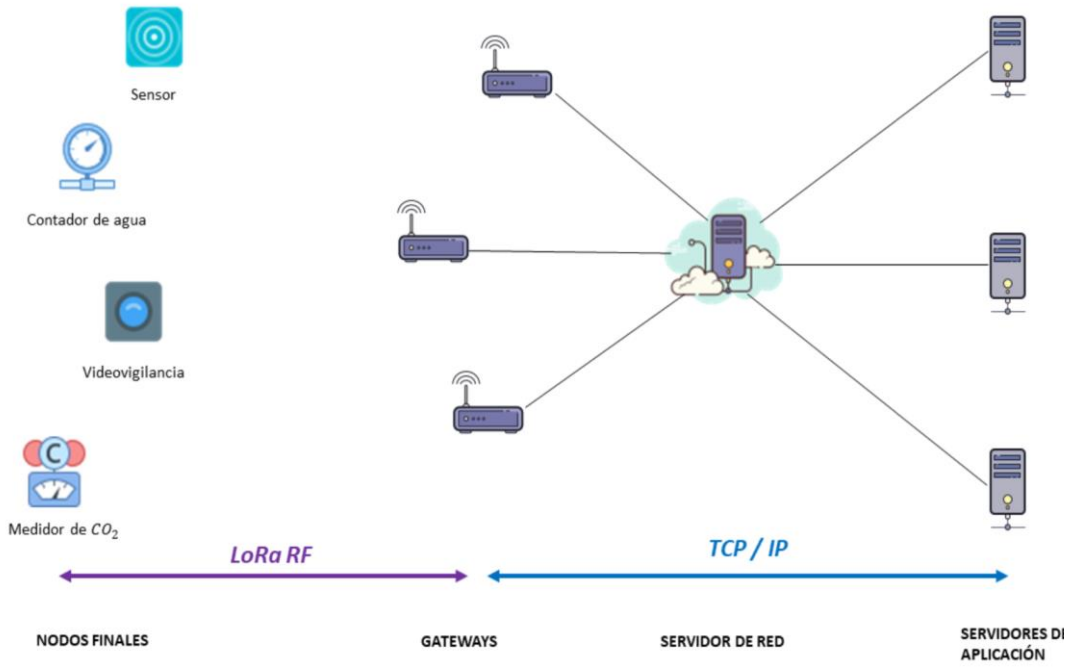


Figura 1.1: Topología de la red LoRaWAN.

- Gateways (pasarelas): son los encargados de recibir los datos procedentes de los nodos y reenviarlos hacia el servidor de red. Para ello pueden hacer uso de diferentes protocolos de comunicación, como por ejemplo una red WiFi o Ethernet.
- Servidores de red: son el cerebro de la red, los encargados de controlar y gestionar la misma, para ello realizan controles de seguridad, filtran paquetes redundantes y adaptan la tasa de datos entre otras cosas. Reciben los datos procedentes de las pasarelas y los envían a los servidores de aplicación.
- Servidores de aplicación: reciben los datos originados en los nodos finales y los envían a los clientes.

Otro aspecto importante a mencionar en este capítulo es la seguridad en LoRaWAN [9], que también está pensada para evitar un alto consumo de energía entre otras cosas. Tiene las características de integridad y confidencialidad de las comunicaciones, así como una autenticación de los dispositivos.

Para que un dispositivo se conecte a una red LoRaWAN, este debe autenticarse y, para ello, se puede hacer de dos formas:

- ABP (*Activation by Personalization*): en este método todos los parámetros a usar se encuentran previamente almacenados en el dispositivo final.
- OTAA (*Over The Air Activation*): en este método el dispositivo final conoce el DevEUI, el AppEUI y el AppKey, y a partir de ellos deriva los parámetros restantes. Para ello, envía al servidor el mensaje de *Join-Request*, al que el servidor responde con un *Join-Accept*.

Los parámetros necesarios son la clave NwSkey para la integridad de los datos y la clave AppSkey para la confidencialidad de los mismos.

2.2 MQTT

Message Queuing Telemetry Transport (MQTT [10,11]) es un protocolo de mensajería sencillo basado en publicación y suscripción. Se trata de un protocolo diseñado para dispositivos con necesidades de ancho de banda limitado y que requieran cierto nivel de fiabilidad; lo que lo hace ideal para entornos del IoT.

MQTT opera sobre TCP/IP en el puerto 1883 y en el puerto 8883 para MQTT con SSL (*Secure Socket Layer*) y, además, proporciona autenticación con la utilización de campos de usuario y contraseña.

Este protocolo asume una topología de estrella formada por los siguientes componentes:

- *Broker*: es el servidor central de la topología encargado de gestionar la comunicación con los clientes y autenticarlos.
- Clientes: pueden tener las siguientes funciones simultáneas:
    - Publicador: es el cliente que envía información al bróker en un determinado tópico.
    - Suscriptor: es el cliente que recibe información del *broker* asociada a un determinado tópico.

Los tópicos son los canales en los que los clientes publican y a los que se suscriben; es la base de la comunicación entre clientes. Los tópicos se organizan de manera jerárquica, de forma que, para un cliente, es posible suscribirse a determinados subtópicos.

Gracias al *broker* se desacopla la comunicación entre los clientes; por tanto, no es necesario que los suscriptores conozcan la dirección de los publicadores, con la del *broker* es suficiente. Además, gracias a que el *broker* es capaz de almacenar los mensajes, los suscriptores no tienen por qué estar conectados al mismo tiempo que los publicadores.

Los mensajes de MQTT tienen la estructura mostrada en la figura 2.2:

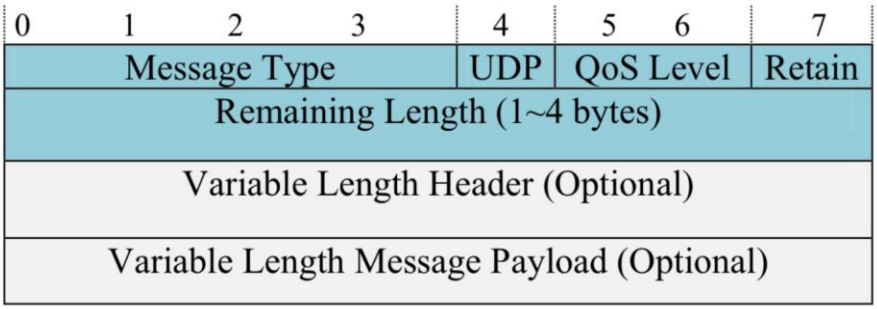


Figura 2.2: Estructura de un mensaje MQTT [12].

Los dos primeros bytes del mensaje son la cabecera fija. En este formato, el valor del campo *Message Type* (tipo de mensaje) indica una variedad de mensajes que incluyen:

Tipo de mensaje	Valor	Dirección del flujo	Descripción
RESERVED	0	*Forbidden*	Reservado
CONNECT	1	Cliente al servidor	Petición de conexión
CONNACK	2	Servidor al cliente	Confirmación de conexión
PUBLISH	3	Ambos	Publicación de un mensaje
PUBACK	4	Ambos	Confirmación de publicación
PUBREC	5	Ambos	Publicación recibida (entrega segura parte 1)
PUBREL	6	Ambos	Liberación de publicación (entrega segura parte 2)
PUBCOMP	7	Ambos	Publicación completada (entrega segura parte 3)
SUBSCRIBE	8	Cliente al servidor	Petición de suscripción
SUBACK	9	Servidor al cliente	Confirmación de suscripción
UNSUBSCRIBE	10	Cliente al servidor	Petición de cancelar suscripción
UNSUBACK	11	Servidor al cliente	Confirmación de cancelar suscripción
PINGREQ	12	Cliente al servidor	PING *request*
PINGRESP	13	Servidor al cliente	PING *response*
DISCONNECT	14	Cliente al servidor	Desconexión
RESERVED	15	*Forbidden*	Reservado

Tabla 2.1: Tipos de mensajes en MQTT.

El indicador *DUP* indica que el mensaje está duplicado y que por lo tanto el receptor puede haberlo recibido antes.

El campo *QoS Level* (nivel de *Quality of Service*) identifica tres niveles de QoS para garantizar la entrega de los mensajes de publicación:

Nivel de QoS	Valor	Descripción
Como mucho una vez	0	*Best-efford*, la recepción no está garantizada.
Al menos una vez	1	Se garantiza que ha llegado, se espera a la recepción de un mensaje de confirmación.
Exactamente una vez	2	Se garantiza la recepción con el intercambio de cuatro mensajes entre emisor y receptor.

Tabla 2.2: Tipos de calidad de servicio en MQTT.

El campo *Retain* informa al servidor que debe retener el último mensaje de publicación recibido y enviarlo a los nuevos suscriptores como primer mensaje.

El campo *Remaining Length* (longitud restante) muestra la longitud restante del mensaje, es decir, la longitud de las partes opcionales.

En la cabecera variable se incluye información como el nombre del tópico o el identificador del mensaje. Por último, en el *payload* se incluye la información a transmitir.

2.3 OpenFlow

En las redes definidas por software, como ya se ha mencionado con anterioridad, se utiliza el protocolo OpenFlow [13], que sirve para comunicar el controlador de la red con los dispositivos de conmutación, a los que denominaremos *switches* OpenFlow. De esta manera, permite que el controlador sea capaz de gestionar la red de una manera centralizada. Es un protocolo abierto que funciona sobre TCP y SSL (este último para proporcionar una conexión segura).

Este protocolo surgió en la Universidad de Stanford y está mantenido por la *Open Networking Foundation*.

Permite cambiar el estado de reenvío local de los *switches* controlando sus tablas de flujos. El formato de entradas en la tabla de flujo está estandarizado, lo que hace que los *switches* sean intercambiables. Entre las características de OpenFlow, destaca la independencia del protocolo de red, la compatibilidad con redes ya existentes y la independencia de la tecnología.

Las tablas de flujo de los *switches* se componen de entradas formadas por:

- Patrón de coincidencia: para comprobar los paquetes recibidos utiliza el puerto de entrada y la cabecera del propio paquete.
- Prioridad: para el caso que coincidan varias entradas.
- Contadores: de paquetes y bytes.
- Instrucciones: acciones a realizar.
- Temporizadores: tiempo máximo o tiempo sin recibir un paquete asociado a un flujo tras el que se elimina la entrada en la tabla.
- Cookies: valor opaco del controlador que se usa para filtrar estadísticas de flujos.

Las reglas de los patrones de coincidencia pueden ser exactas, si especifican todos los campos; o reglas con comodín, en las que al menos un campo tiene un prefijo o un “*”. Para estas últimas, puede ocurrir que coincidan varias en un mismo flujo y, en ese caso, se utilizan las prioridades.

Las acciones a realizar incluyen: el reenvío de paquetes a puertos físicos/virtuales, encolamiento en una cola particular en el puerto para QoS, descarte del paquete o modificación de un campo del paquete.

El funcionamiento del protocolo es el siguiente: cuando llega un paquete al *switch*, este se compara con las entradas de la tabla de flujo y, en el caso de que coincida, se realiza la acción que se indique. En el caso de que no coincida con ninguna, el paquete se almacena en

un buffer y la cabecera del mismo se envía al controlador que la recibe, decide una acción y la envía al *switch* para que la realice.

Al ser un protocolo de comunicación, define la estructura y los diferentes tipos de mensajes que pueden intercambiar el controlador y los equipos de conmutación. Los mensajes contienen una cabecera en la que se indica la versión del protocolo, el tipo de mensaje, la longitud y el identificador de transacción (*transaction ID*) del mensaje.

Entre los tipos de mensajes se definen: mensajes simétricos, son los mensajes *hello* y *echo*; mensajes del controlador al *switch*, para la gestión de tablas de flujo o para pedirle información sobre las capacidades y contadores del *switch*; y mensajes asíncronos, que incluyen los mensajes enviados por los *switches* al controlador sin petición previa, como pueden ser el envío de un paquete sin coincidencias o informar al controlador de errores.

Las nuevas versiones de este protocolo incluyen opciones de múltiples tablas de flujos (tablas encadenadas), grupos para gestionar varios flujos de manera conjunta, patrones extensibles de coincidencia o soporte para varios controladores (alta disponibilidad), entre otras.

2.4 Red neuronal artificial

Las personas somos capaces de aprender de la experiencia y extraer conocimiento en gran parte gracias a nuestro cerebro. Las redes neuronales artificiales son sistemas computacionales, enmarcados dentro del *Machine Learning*, que se basan en imitar el funcionamiento del cerebro, para así ser capaces de aprender [8].

El cerebro humano está formado por un gran número de neuronas que se comunican unas con otras. Cada neurona está compuesta de un cuerpo o soma que contiene el núcleo, un tallo o axón, y unas prolongaciones del cuerpo y del axón, denominadas dendritas. Las dendritas de una neurona se conectan con las de otras, en una conexión denominada sinapsis, que permite la transmisión de impulsos eléctricos.

Las redes neuronales artificiales se basan en los mismos conceptos que las redes neuronales naturales. Una red neuronal se compone de unidades llamadas neuronas. Cada neurona recibe una serie de entradas a través de interconexiones y emite una salida. Para ello, el proceso consta de dos funciones:

- Función de propagación (o excitación): consiste (normalmente) en la suma ponderada de las señales de entrada. Si el peso es positivo, la conexión es excitatoria y, si es negativo, inhibitoria. Además, como entrada se suele añadir también un *bias* (sesgo).
- Función de activación y/o transferencia: en su forma más sencilla es 1 si se supera un umbral, y 0 en otro caso. Existen otras versiones de función de activación, como una función lineal, la de la tangente hiperbólica, la sigmoidal, o el rectificador ReLU (*Rectified Linear Unit*) en *Deep Learning*.

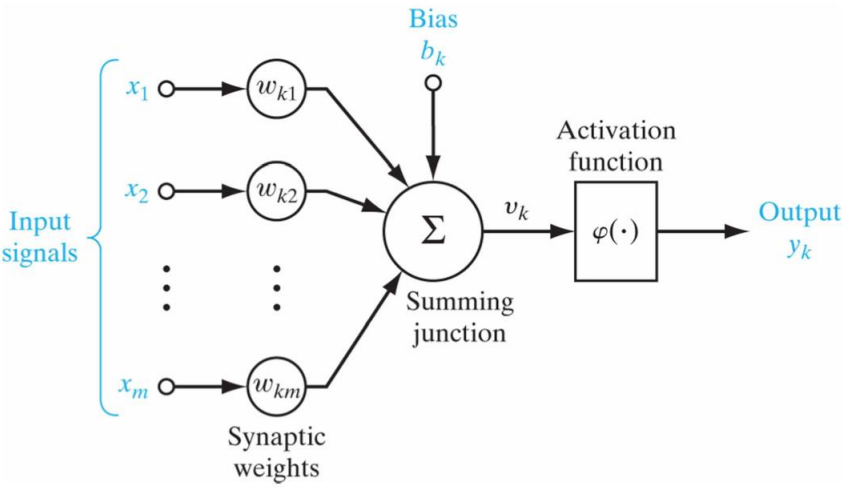


Figura 2.2: Esquema de una neurona artificial.

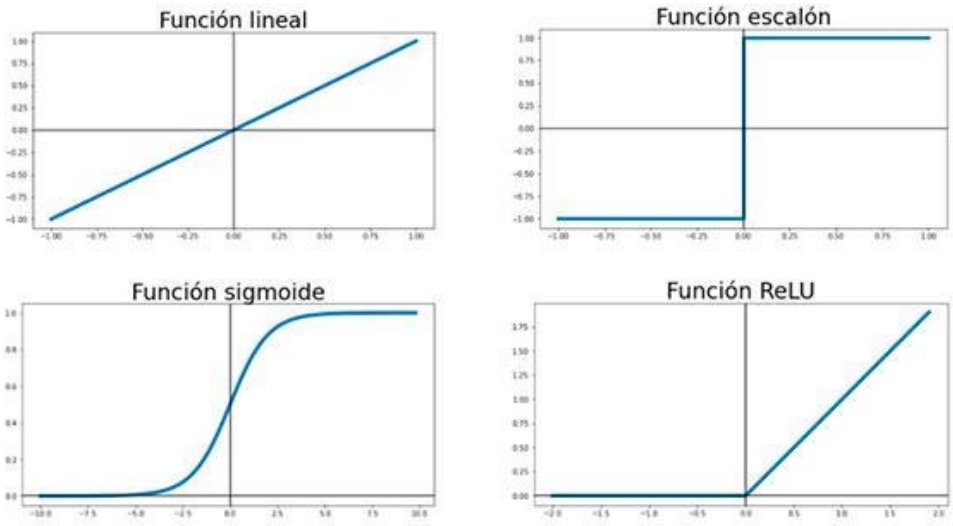


Figura 2.3: Gráficas de los tipos de funciones de activación.

Las redes neuronales se componen de la interconexión de neuronas, siendo la entrada de unas las salidas de otras. Así, una red neuronal estaría compuesta de unas entradas, capas de neuronas conectadas entre sí, y unas salidas. Ajustando adecuadamente los pesos de las entradas en cada neurona se consigue la salida deseada, es decir, el aprendizaje. En una red con varias capas de neuronas resulta complicado saber qué pesos hay que ajustar, para ello se utiliza el algoritmo de propagación hacia atrás de errores.

La propagación hacia atrás de errores (en inglés *backpropagation*) se basa en que es fácil saber el error que ocasiona cada neurona de la capa de salida. Cada una de esas neuronas está conectada con las neuronas de la capa anterior mediante unos pesos. Podemos usar esos pesos para determinar cuánto contribuyen las neuronas de la capa anterior al error. Y así, podemos obtener cuánto contribuye al error cada neurona. Sabiendo cuánto contribuye cada neurona al error, podemos intentar actualizar los pesos para reducir ese error.

Si la red neuronal hace uso de más de una capa oculta de neuronas (capa que no es la de entrada ni la de salida), se clasifica como una red neuronal de aprendizaje profundo (*Deep*

Learning). Este aprendizaje profundo trata de modelar abstracciones de alto nivel de los datos, usando para ello arquitecturas compuestas de transformación no lineales múltiples (muchas capas). Es la metodología que mejores resultados obtiene en visión por ordenador, traducciones, reconocimiento automático del habla, análisis de sentimiento o compresión del lenguaje natural. Normalmente, se hace uso de cómputo en GPUs (*Graphics Processing Unit*) para acelerar el aprendizaje de este tipo.

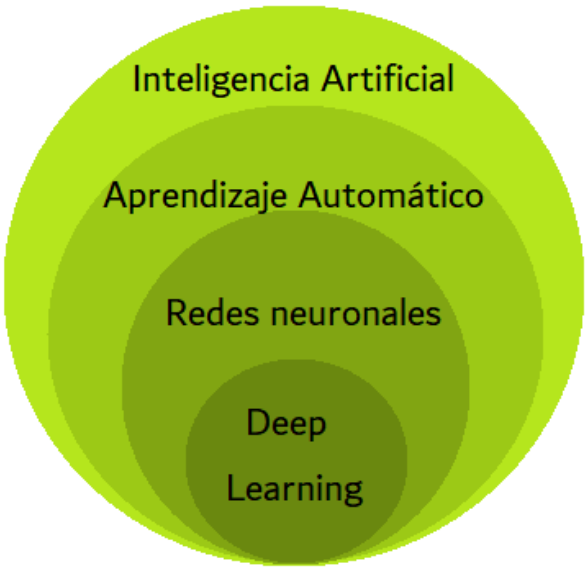


Figura 2.4: Esquema de la relación entre Inteligencia Artificial y el aprendizaje profundo.

Dentro del aprendizaje profundo, existen muchos modelos diferentes de redes neuronales, cada uno con una arquitectura y características que los hacen más adecuados para resolver ciertos problemas.

2.4.1 RNN y LSTM

En este proyecto los datos recogidos son muestras temporales de temperatura y humedad, y se pretende realizar futuros pronósticos haciendo uso de ellos. Por tanto, se puede considerar como un problema de predicción de series temporales. Para resolver este tipo de problemas utilizando aprendizaje profundo, son útiles las redes neuronales recurrentes (RNN del inglés *Recurrent Neural Network*), más concretamente las redes LSTM (*Long Short-Term Memory*).

Para dotar de memoria a una red neuronal, de manera que las entradas anteriores sigan afectando a las salidas posteriores, se puede hacer uso de redes neuronales recurrentes [14]. Estas, son redes neuronales en las cuales se forman ciclos de realimentación entre las neuronas que la forman. De esta forma, son capaces de tener cierta memoria, ya que la salida de la red en un tiempo determinado dependerá de las entradas en instantes de tiempo anteriores, lo que las hace idóneas para tratar con datos secuenciales.

Dentro de una red neuronal recurrente, se define una celda como la parte de la red que forma un ciclo, preservando cierta información en el tiempo. Si bien en este tipo de redes se

acentúa el problema de desvanecimiento de gradiente (los gradientes son valores que se utilizan para actualizar los pesos de las redes neuronales), que consiste en que, al aplicar el algoritmo de propagación de errores hacia atrás, los cambios a aplicar en los pesos se van haciendo más pequeños pudiendo tender a cero, lo que deriva en que las primeras capas de la red sean más difíciles de entrenar y por tanto que aprendan.

Para hacer frente a este problema en las redes neuronales recurrentes, surge el concepto de redes LSTM [15] que son unas extensiones de las mismas. Para ello, implementan celdas de memoria, en las cuales, en función de la importancia de la información recibida, se puede decidir si almacenarla o eliminarla. La importancia asignada a la información también se aprende en función de los pesos, de esta manera la red aprende con el tiempo qué información es más importante y cuál menos.

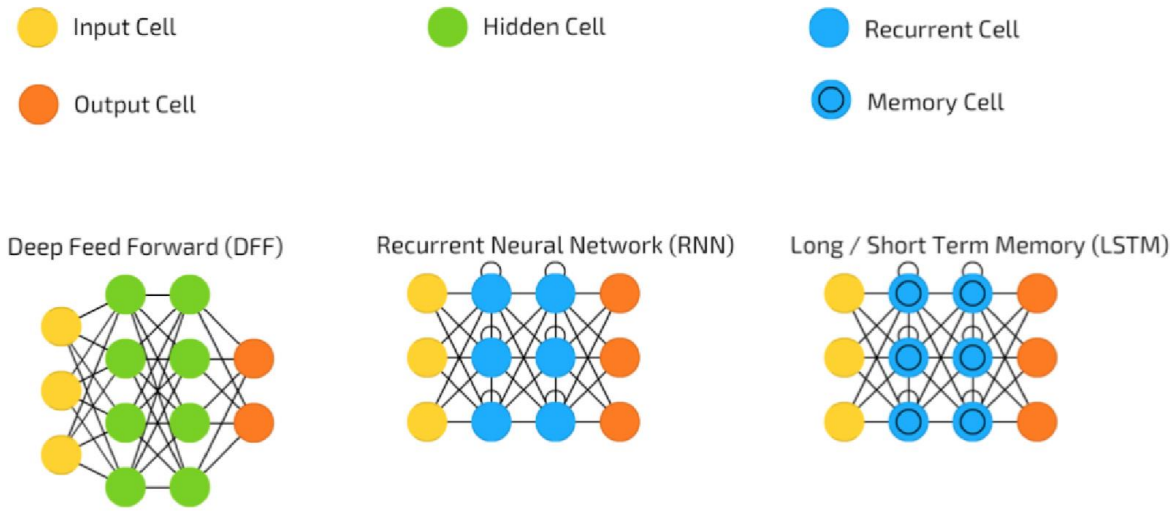


Figura 2.5: Esquemas de diferentes tipos de redes neuronales [8].

LSTM funciona incluso con retrasos prolongados entre eventos importantes y puede manejar señales que mezclan componentes de baja y alta frecuencia.

Los siguientes apartados; el 2.4.2 y el 2.4.3, recogen una breve descripción teórica de los modelos tenidos en cuenta para este proyecto. Estos se han elegido por ser arquitecturas típicas capaces de resolver problemas de predicción de series temporales con buenos resultados.

2.4.2 N-BEATs

N-BEATs [16] (*Neural Basis Expansion Analysis for interpretable Time Series forecasting*) se trata de una arquitectura neuronal profunda basada en enlaces residuales hacia atrás y hacia adelante, y en una serie de capas apiladas y completamente conectadas. Esta arquitectura proporciona una serie de propiedades; es interpretable, fácilmente aplicable a una amplia gama de dominios objetivo y rápida de entrenar.

N-BEATs surge de la idea de no utilizar ningún componente específico de series de tiempo; y utilizar solamente primitivas de aprendizaje profundo, como los bloques residuales, considerados por sí mismos suficientes para resolver problemas de pronóstico (pretende explorar el potencial de una arquitectura de *Deep Learning* pura en el pronóstico de series temporales).

La arquitectura base propuesta es simple y genérica, pero expresiva (profunda). Además, la arquitectura no debe depender de la ingeniería específica de series de tiempo o del escalado de entrada y debe ser ampliable para hacer que sus resultados sean interpretables por humanos.

El esquema es el mostrado en la figura 2.6.

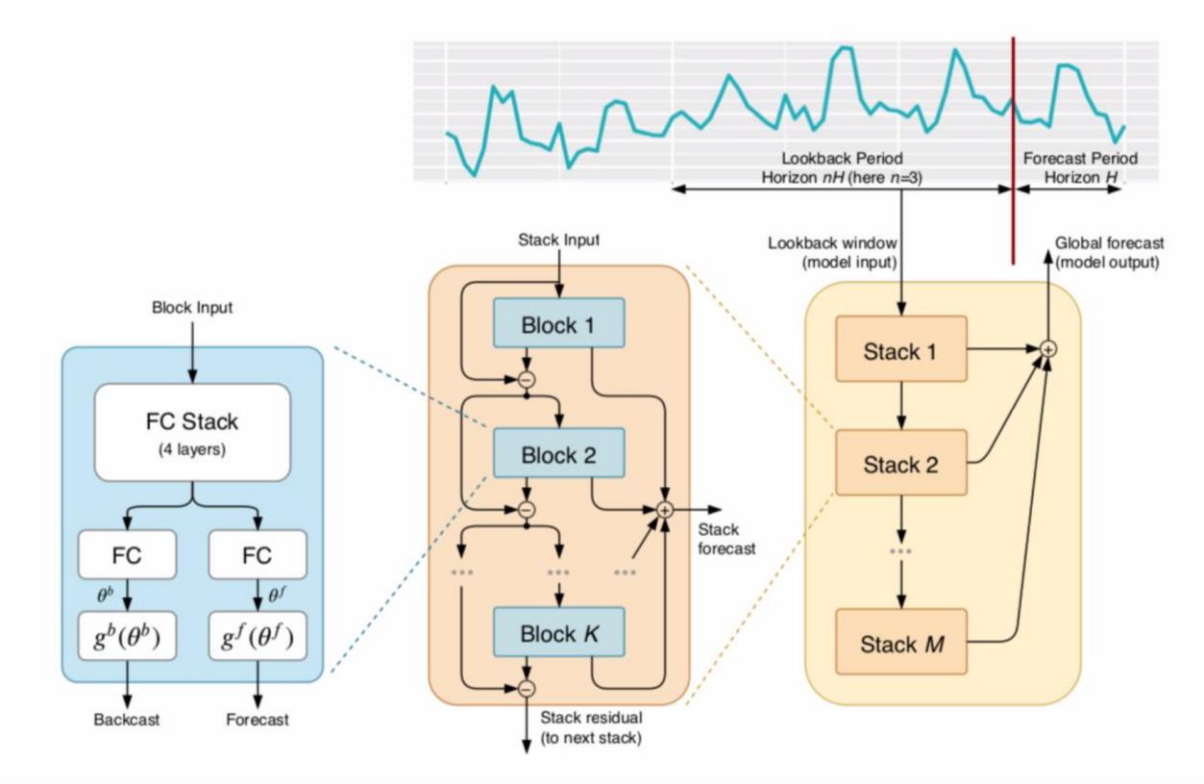


Figura 2.6: Esquemas del modelo N-BEATs [16].

2.4.3 Temporal Fusion Transformer

Es una arquitectura novedosa que combina la predicción de múltiples horizontes de alto rendimiento con conocimientos interpretables de la dinámica temporal [17]. Para aprender las relaciones temporales a diferentes escalas, TFT utiliza capas recurrentes para el procesamiento local y capas *self-attention* interpretables [18] para las dependencias a largo plazo. TFT utiliza componentes especializados para seleccionar características relevantes y una serie de capas de puerta para suprimir componentes innecesarios, lo que permite un alto rendimiento en una amplia gama de escenarios.

TFT es una arquitectura de red neuronal profunda, basada en la atención para el pronóstico de series temporales que logra un alto rendimiento al tiempo que permite nuevas formas de interpretación. Algunas de sus características son:

- Codificadores de covariables estáticos, que codifican vectores de contexto para su uso en otras partes de la red.
- Mecanismos de activación y selección de variables dependientes de la muestra para minimizar las contribuciones de entradas irrelevantes.
- Una capa de secuencia a secuencia a nivel local para procesar entradas conocidas y observadas.
- Un decodificador temporal de *self-attention* para aprender cualquier dependencia a largo plazo presente dentro del conjunto de datos.
- Intervalos de predicción a través de pronósticos cuantílicos para determinar el rango de valores probables en cada horizonte de predicción.

El uso de estos componentes especializados también facilita la interpretación; TFT ayuda a identificar variables de importancia global para el problema de predicción, patrones temporales persistentes y eventos significativos. El esquema es el mostrado en la figura 2.7.

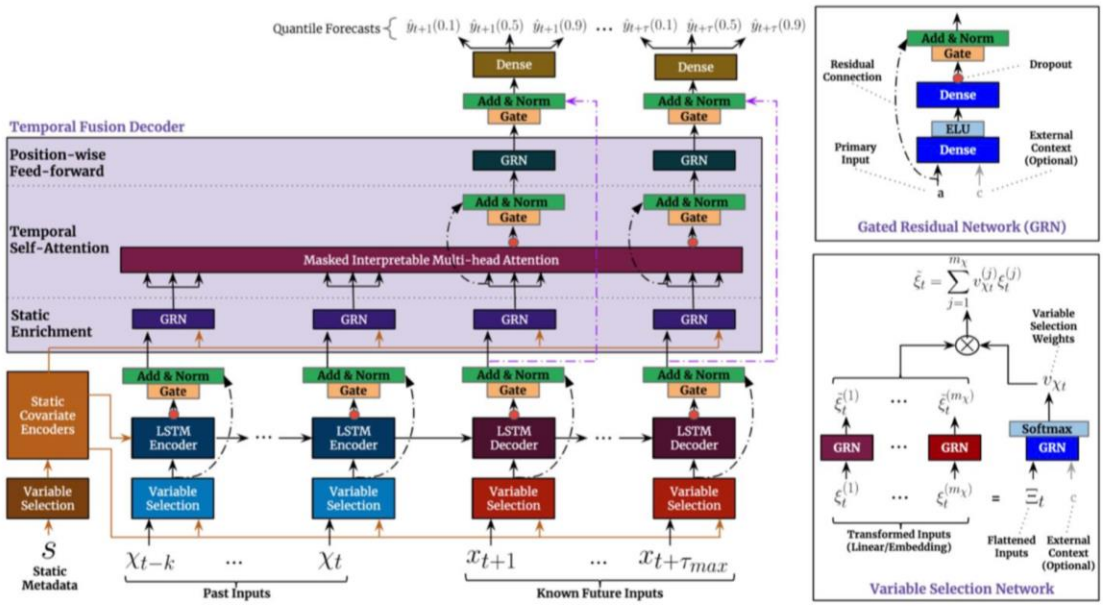


Figura 2.7: Esquemas del modelo *Temporal Fusion Transformer* [17].

La comprensión de todos estos conceptos ha sido necesaria para llevar a cabo el proyecto.

3. Herramientas

A continuación, se recogen y explican las herramientas utilizadas para llevar a cabo el desarrollo del proyecto. Nótese que se han elegido todas ellas de código abierto.

3.1 Python

Es un lenguaje de programación multiparadigma ya que permite la programación orientada a objetos, la programación funcional y la programación imperativa. Además, es un lenguaje interpretado, dinámico y de código abierto. Destaca por su sencillez, legibilidad y sintaxis exacta.

En este proyecto, todos los ejercicios de programación necesarios han sido realizados con Python 3; y también se ha tenido en cuenta a la hora de elegir las herramientas que han intervenido. La mota LoRaWAN se ha programado en MicroPython.

3.2 ChirpStack (LoRaServer)

A la hora de desplegar una red LoRaWAN, podemos usar como referencia el proyecto ChirpStack [19] (anteriormente conocido como LoRaServer), que consiste en una serie de aplicaciones y componentes de código abierto que se relacionan como se muestra en la figura 3.1.

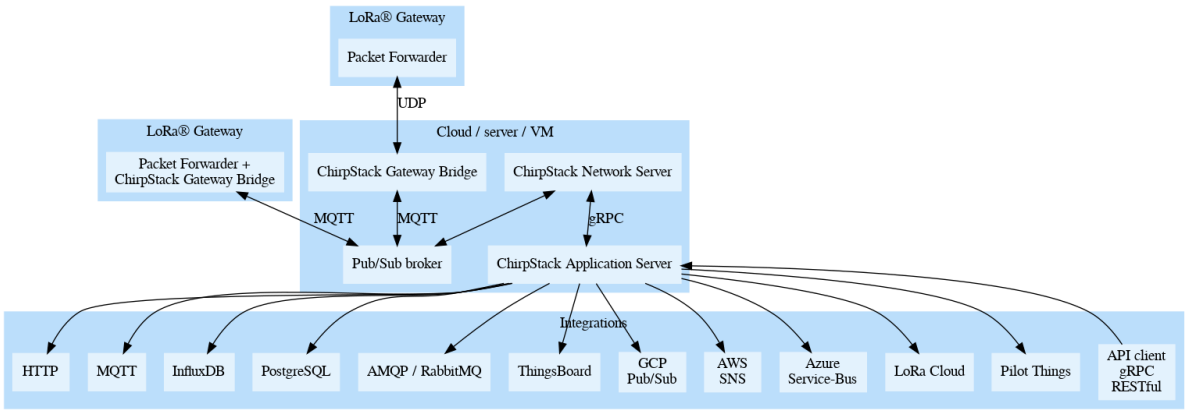


Figura 3.1: Esquema de la arquitectura ChirpStack [19].

- LoRa Gateway Bridge: es el encargado de recibir los datos del *packet forwarder* y enviarlos por MQTT al LoRa Server a través de un bróker MQTT.
- LoRa Server: es el servidor de red de LoRaWAN, se comunica con el LoRa Gateway Bridge con MQTT y con el LoRa Geo Server y LoRa App Server usando gRPC (*Google Remote Procedure Calls*). Es el encargado de gestionar el estado de la red.
- LoRa App Server: servidor de aplicación de LoRaWAN, incluye además una interfaz web para la gestión de la red y es donde se realizan las integraciones con las diferentes aplicaciones a las que se enviarán los datos.

En este proyecto, como se explicará más adelante, se han instalado todos los servidores en la misma máquina.

3.3 Mosquitto

Como se ha visto en la sección anterior, el protocolo MQTT proporciona un método ligero para realizar mensajes mediante un modelo de publicación / suscripción; esto lo hace adecuado para el Internet de las cosas, donde se utilizan sensores de baja potencia.

Eclipse Mosquitto [20] es un bróker (agente de mensajes) de código abierto que implementa las versiones 5.0, 3.1.1 y 3.1 del protocolo MQTT. Mosquitto es ligero lo cual lo hace adecuado para su uso en todos los dispositivos, desde computadoras de placa única de baja potencia hasta servidores completos.

El proyecto Mosquitto también proporciona una biblioteca escrita en C para implementar clientes MQTT, y los clientes MQTT de línea de comando mosquitto_pub y mosquitto_sub. En la red LoRaWAN, se ha usado Mosquitto como bróker de los mensajes intercambiados entre los servidores de la plataforma ChirpStack; además también se ha hecho uso del cliente suscriptor para recibir dichos mensajes y del cliente mqtt Paho para Python (desarrollado al igual que Mosquitto por Eclipse) [21].

3.4 VirtualBox

VirtualBox [22] es un potente software de virtualización de uso general de código abierto y multiplataforma. Puede ejecutarse en cualquier lugar, desde pequeños sistemas integrados o máquinas de escritorio hasta implementaciones de centros de datos e incluso entornos en la nube.

VirtualBox se está desarrollando activamente con lanzamientos frecuentes y tiene una lista cada vez mayor de características, sistemas operativos invitados compatibles y plataformas en las que se ejecuta. Oracle garantiza que el producto siempre cumpla con los criterios de calidad profesional.

Puede instalar y ejecutar tantas máquinas virtuales como desee. Los únicos límites prácticos son el espacio en disco y la memoria. En este proyecto se ha usado para crear dos máquinas virtuales, una con Ubuntu 18.04 que ejecuta Mininet y otra con Ubuntu 16.04 ejecutando InfluxDB y Grafana.

3.5 Open vSwitch

Como se ha explicado en la sección anterior, el protocolo OpenFlow sirve para comunicar el controlador con los *switches* OpenFlow para configurarlos. En este proyecto se ha usado la plataforma Open vSwitch [23] para generar dichos *switches*, y en esta sección se introduce algunos de sus conceptos teóricos.

OVS (*Open Virtual Switch*) es un conmutador software de código abierto y multicapa. Surgió en 2009 como resultado de un proyecto entre la Universidad de Berkeley y Nicira Networks. La mayoría del código está escrito en C.

Fue diseñado originalmente para su aplicación en entornos virtuales, para estar a cargo de la conectividad entre máquinas virtuales y también reenviar el tráfico entre ellas y la red física. Sin embargo, ha evolucionado y ahora se utiliza en gran variedad de entornos, por ejemplo, en la pila de control de *switches* hardware.

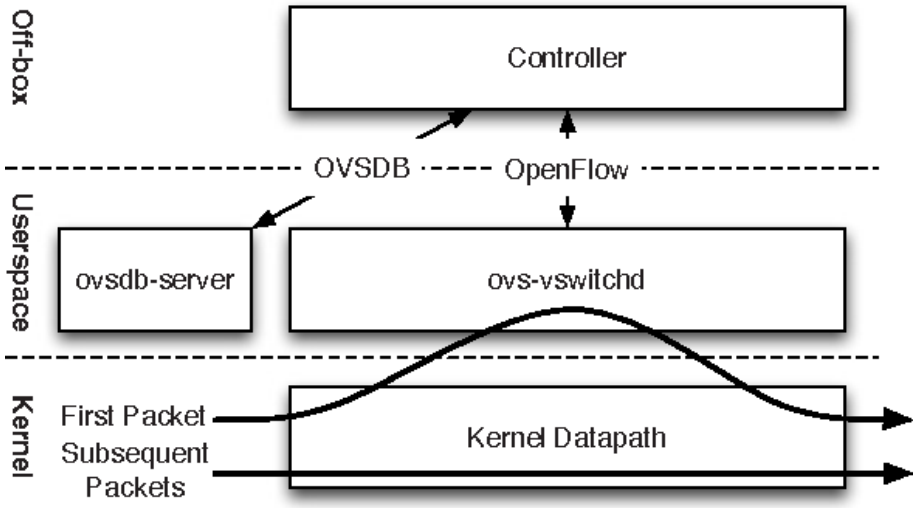


Figura 3.2: Esquema de la arquitectura de un Open vSwitch [23].

OVS expone dos interfaces bien definidas: OpenFlow para el control del comportamiento de reenvío y OVSDB para la configuración del propio OVS.

OVSDB (*Open vSwitch Database Management Protocol*) es un protocolo de código abierto diseñado para administrar implementaciones en OVS, está definido en RFC 7047. El protocolo OVSDB se basa en *JSON Remote Procedure Call*.

El *OVSDB Management Protocol* permite consultar y modificar la configuración de Open vSwitch, permitiendo por ejemplo crear, modificar y eliminar rutas de datos OpenFlow. También admite la creación, modificación y eliminación de puertos, túneles y colas, así como la configuración de políticas de QoS y la vinculación de esas políticas a las colas.

3.6 RYU

Ryu [24,25] es un controlador SDN de código abierto escrito completamente en Python. Ryu proporciona componentes software con API bien definidas que facilitan a los desarrolladores la creación de nuevas aplicaciones de gestión y control de redes. Ryu admite varios protocolos para administrar dispositivos de red, como OpenFlow (y sus diferentes versiones) u OVSDB entre otros.

La figura 3.3 muestra la arquitectura de Ryu y sus componentes principales que son:

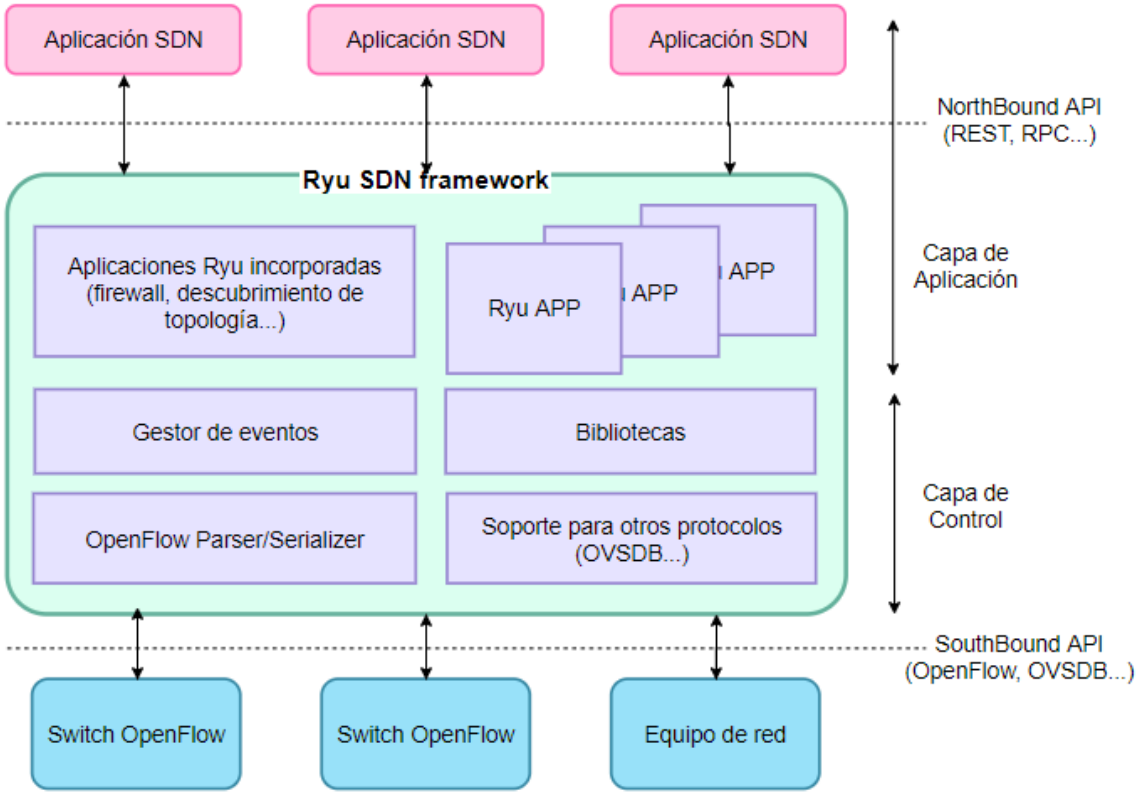


Figura 3.3: Esquema de la arquitectura de Ryu.

- Bibliotecas: Ryu tiene una colección de bibliotecas, que incluyen desde el soporte para múltiples protocolos hacia abajo (*Southbound*), hasta varias operaciones de procesamiento de paquetes de red. La biblioteca de paquetes Ryu le ayuda a analizar y construir varios paquetes para diferentes protocolos, como VLAN, MPLS, etc.
- Controlador OpenFlow: componente clave de la arquitectura Ryu que se encarga de gestionar los *switches* OpenFlow, siendo capaz de configurar flujos, gestionar eventos, etc. El controlador OpenFlow es una de las fuentes de eventos internos en la arquitectura Ryu. Incluye una biblioteca de codificadores y decodificadores del protocolo OpenFlow.
- Gestor de eventos y procesos centrales: el administrador de Ryu *(Ryu Manager*) es el ejecutable principal, encargado de escuchar en la dirección IP especificada en el puerto especificado (6633 por defecto) para que cualquier *switch* OpenFlow (hardware o virtual) pueda conectarse al controlador Ryu. El administrador de aplicaciones es el componente fundamental de todas las aplicaciones de Ryu ya que todas las aplicaciones heredan de la clase RyuApp, que, entre otras cosas, incluye la gestión de eventos, la mensajería o la gestión del estado en memoria.
- RYU en dirección norte (*Northbound*): en la capa de API, Ryu incluye un complemento Openstack Neutron. Ryu también admite una interfaz REST (*Representational State Transfer*) para sus operaciones OpenFlow.
- Aplicaciones: Ryu se distribuye con múltiples aplicaciones como simple_switch, router, isolation, firewall, topology, VLAN, etc. Las aplicaciones Ryu son entidades

de un solo subproceso, que implementan varias funcionalidades. Las aplicaciones Ryu se envían eventos asincrónos entre sí.

En el Anexo se recoge y explica el código *simple_switch_13.py*, correspondiente a la configuración para los *switches* que se ha usado en este proyecto.

3.7 Mininet

Mininet [26, 27] es un software emulador de redes de código abierto que facilita en gran medida el diseño e implementación de redes virtuales. Además de sus comandos propios para lanzar topologías y gestionar los dispositivos virtuales de la misma, tiene una API en Python con la que también es posible modificar los diferentes parámetros de la red y definir topologías y despliegues de una manera sencilla.

En Mininet los *hosts* virtualizados sólo pueden estar basados en el kernel de Linux. Si bien, estos *hosts* virtuales pueden ejecutar los programas instalados en la máquina que ejecuta el emulador. Además, Mininet incluye Open vSwitch para integrarlos en las redes que se creen; si bien el controlador OpenFlow debe ser externo a Mininet.

En este proyecto Mininet y el controlador Ryu se han instalado en una máquina virtual.

3.8 InfluxDB

Para almacenar los datos recogidos por los sensores usaremos InfluxDB [28], una base de datos de series temporales (TSDB, por sus siglas en inglés *Time Series Data Base*) de código abierto, es decir, una base de datos optimizada para datos con marcas de tiempo. Una TSDB se crea específicamente para manejar métricas y eventos o mediciones con marca de tiempo, está optimizada para medir cambios a lo largo del tiempo y tiene propiedades clave de diseño arquitectónico que la hace muy diferentes de otras bases de datos.

Algunas de las características de InfluxDB son por ejemplo el almacenamiento y la compresión de datos con marcas temporales, la gestión del ciclo de vida de los datos, el resumen de datos, la capacidad de manejar exploraciones de grandes series de tiempo de muchos registros o consultas con reconocimiento de series temporales. Se compila en un solo binario sin dependencias externas y dispone de una API HTTP (*Hyper Transfer Transport Protocol*) de consulta y escritura sencilla y de alto rendimiento.

Las marcas de tiempo en InfluxDB pueden ser de precisión en segundos, milisegundos, microsegundos o nanosegundos. Los valores pueden ser enteros de 64 bits, flotantes de 64 bits, cadenas de caracteres y valores booleanos. Las etiquetas permiten indexar series para consultas rápidas y eficientes, además, no tiene límite para la cantidad de etiquetas y campos que se pueden usar.

Utiliza el puerto 8086 para escuchar peticiones y usa un lenguaje de consulta expresivo llamado InfluxQL, similar a SQL (*Structured Query Language*), diseñado para consultar fácilmente datos agregados. Además, en las nuevas versiones ha desarrollado el lenguaje de

scripting Flux. En este proyecto también se ha hecho uso del cliente Python para comunicarse con la base de datos.

3.9 Grafana

Grafana [29] es una herramienta software de visualización de datos de código abierto. Dispone de una API HTTP y una interfaz web, por defecto utiliza el puerto 3000. Proporciona a los usuarios visualización de diferentes tipos de gráficos y alertas; resultando muy útil para el análisis de sistemas e implementaciones.

Proporciona integraciones con un gran número de herramientas, que pueden ser usadas como fuentes de datos, por ejemplo, con la base de datos InfluxDB. En dicha integración, se pueden crear gráficos y paneles para *queries* determinadas utilizando la web.

3.10 Pytorch

Pytorch [30] es la herramienta de Facebook (principalmente desarrollada por FAIR, un conjunto de laboratorios de investigación de inteligencia artificial adscritos a la empresa Facebook) para *Deep Learning*. Se trata de una librería optimizada de tensores para aprendizaje automático, la cual es de código abierto y está basada en Torch (su predecesor). Pytorch está programado en diferentes lenguajes y tiene interfaz para varios de ellos, siendo Python la que se va a usar en este trabajo.

Pytorch es una API de nivel intermedio. Es más fácil de usar y aprender que TensorFlow, pero más complejo que Keras (ambas de Google).

Entre sus características destaca el uso de CPUS y GPUS, ya que usa procesamiento de tensores que permite paralelizar los datos y acelerar los cálculos de los modelos (las redes neuronales por lo general consumen bastante tiempo). También hace uso de gráficos dinámicos, que permiten modificar el comportamiento de las redes en tiempo de ejecución, optimizando los modelos.

Algunos productos conocidos que hacen uso Pytorch son por ejemplo el piloto automático de Tesla o el lenguaje probabilístico Pyro (también usado para generar modelos de *Deep Learning*). Pytorch se encuentra en continuo desarrollo por parte de la comunidad, ejemplo de ello es el paquete Pytorch Forecasting que se ha utilizado en este trabajo.

3.10.1 Pytorch Forecasting

Este paquete [31] de Pytorch tiene como objetivo facilitar el pronóstico de series temporales con redes neuronales; tanto para la investigación como para casos de uso real. Para ello, hace uso de la biblioteca Pandas, así como de la herramienta TensorBoard.

Pandas [32], que es una extensión de NumPy; proporciona estructuras de datos para facilitar el análisis y manipulación de datos en Python. En Pandas, las tablas de datos (con filas y columnas) se denominan *Dataframes*, y son especialmente útiles para almacenar

información con una dimensión temporal, en nuestro caso, las series temporales de datos que vamos a utilizar para realizar predicciones.

TensorBoard [33], por su parte, es un kit de herramientas de visualización para *Machine Learning*, integrado en TensorFlow y diseñado por Google; aunque permite su integración con otras herramientas como es el caso de Pytorch [34]. TensorBoard permite visualizar grafos de los modelos, representaciones gráficas de diferentes métricas de los mismos y su evolución en el tiempo.

Pytorch Forecasting proporciona una API de alto nivel flexible en la cual se implementan, por ejemplo:

- La clase *TimeSeriesDataSet*, para trabajar con datos de series temporales; ya que abstrae el manejo de transformaciones de variables, valores perdidos, submuestreo aleatorio, múltiples longitudes de historial, etc. A esta clase basta con pasarle un *Dataframe* de Pandas indicando las variables relevantes.
- La clase *BaseModel*, que proporciona entrenamiento básico de modelos de series temporales junto con el registro en Tensorboard y también visualizaciones genéricas, como por ejemplo predicciones frente a valores reales o diagramas de dependencia.
- Múltiples arquitecturas de modelos conocidos de redes neuronales para la predicción de series temporales, las cuales vienen con capacidades de interpretación integradas.
- Métricas de series temporales de horizontes múltiples.

3.11 Wireshark

Wireshark [35] es un analizador de protocolos utilizado para realizar análisis de datos y protocolos, como herramienta didáctica y para solucionar problemas en redes de comunicaciones. La funcionalidad que provee es similar a la de *tcpdump*, pero añade una interfaz gráfica y muchas opciones de organización y filtrado de información. Permite ver todo el tráfico que pasa a través de una red estableciendo la configuración de la interfaz en la que se está capturando en modo promiscuo.

También permite abrir archivos de capturas para analizar la información capturada, a través de los detalles de cada paquete. Wireshark incluye un completo lenguaje para filtrar lo que queremos ver y la habilidad de mostrar el flujo reconstruido de una sesión de TCP.

Wireshark es una herramienta de software libre. En este proyecto se ha usado para comprobar el envío de mensajes MQTT sobre la red SDN.

En este capítulo se ha recogido una descripción de las herramientas que se han usado a lo largo del proyecto.

4. Estado del arte

En este capítulo se recoge un estudio del estado del arte, es decir de artículos y trabajos relacionados con el tema de nuestro proyecto. De esta manera se pretende obtener una visión actual del estado de relación de las tecnologías involucradas en el proyecto y también nos puede servir para obtener ideas. También se incluye un apartado que muestra la posibilidad de llevar a cabo una implementación similar en un entorno en la nube pública.

Son muchos los artículos que tratan el uso de SDN para facilitar el despliegue de redes IoT. La flexibilidad y dinamismo que proporcionan las redes definidas por software puede servir para gestionar aspectos que van desde la implementación de protocolos de comunicaciones del IoT y sus elementos de red, hasta hacer frente a las diferentes demandas y requerimientos en el procesamiento de los datos, acercando por ejemplo la computación en el borde.

Con el incremento en la cantidad de datos que recibimos, gracias a los sensores de IoT, se nos hace necesario nuevas formas de tratarlos. Muchos trabajos profundizan en las diferentes técnicas que se pueden usar atendiendo tanto a la procedencia de los datos como a la aplicación que se les quiera dar; incluyendo métodos de aprendizaje profundo. Este es el caso de [36] o [37].

También existen casos más concretos de trabajos que exploran el uso de algoritmos de *Machine Learning* con datos obtenidos de los sensores de IoT, incluso para tareas de predicción climática, si bien a diferencia del presente trabajo estos algoritmos no incluyen redes neuronales. Por ejemplo [38], [39] o [40].

Existen varios artículos que indagan en el uso de aprendizaje profundo para realizar predicciones de series temporales, como en [41], y más concretamente para realizar predicciones del tiempo atmosférico, como es el caso de [42] y [43]. Destaca el artículo [44], en el cual se propone una implementación para realizar predicciones de temperatura con *Deep Learning* sobre los datos obtenidos en un sistema basado en LoRa.

Existen también muchos estudios que utilizan algoritmos de *Machine Learning* para mejorar la seguridad en redes SDN, incluidas implementaciones para soportar dispositivos IoT. Utilizando aprendizaje automático por ejemplo se pueden detectar intrusiones en una red o realizar previsiones del tráfico IoT para ser capaz de gestionarlo (a diferencia de este trabajo, los algoritmos de *Machine Learning* a los que aquí se refiere no se aplican sobre los datos recogidos por los sensores).

4.1 Integración en la nube (AWS)

Si consideramos proveedores de nube pública; como es el caso de AWS (*Amazon Web Services*), Azure o GCP (*Google Cloud Platform*), todos ellos ofrecen a sus clientes servicios para albergar diferentes recursos conectados entre sí en su infraestructura. Se podría decir que estos recursos forman una red, y en la mayoría de los casos, los propios clientes pueden configurarlos y configurar la manera en la que se conectan los unos con los otros;

directamente a través de software. Por tanto, podemos admitir en cierto modo que se trata de un concepto parecido al de las SDNs.

Al ser soluciones propietarias, las opciones de configuración, los servicios y modelos difieren entre sí, pero la idea de proporcionar a los clientes redes configurables es la misma. Para ellos hacen uso de conceptos como subredes, tablas de enrutamiento, listas de control de accesos, direcciones IP elásticas o elementos como balanceadores de carga, pasarelas a Internet, instancias NAT, etc.

Si además dichos proveedores de nube ofrecen entre sus servicios unos específicos para IoT y para *Machine Learning*, podemos realizar una implementación que haga uso de todas estas tecnologías (como en el presente proyecto), utilizando recursos de la infraestructura del proveedor.

Teniendo en cuenta esto, a continuación, se recoge una breve explicación de los servicios específicos que pueden resultar útiles para implementaciones de IoT y *Machine Learning* en un entorno de redes configuradas por software como es AWS.

Amazon Web Services, es una plataforma de servicios en la nube pública [45]. Ofrece una gran cantidad de servicios que van desde servicios clásicos como puede ser el cómputo o el almacenamiento, a servicios orientados a tecnologías emergentes como la Inteligencia Artificial o lagos de datos, entre otros. AWS proporciona varios servicios para IoT, algunos de ellos incluso pensados para integrar Inteligencia Artificial. Estos productos y servicios se clasifican en las siguientes categorías:

Software de dispositivos:

- **SDK para AWS IoT**: consisten en bibliotecas de código abierto [46], disponibles para su uso en varios lenguajes, que proporcionan los métodos necesarios para conectar dispositivos hardware a la plataforma IoT de AWS. Estas bibliotecas están optimizadas para su uso en equipos con requerimientos estrictos de batería, cómputo o ancho de banda, es decir están diseñados para motas de IoT.
- **FreeRTOS**: no se trata de un servicio de AWS, es un sistema operativo en tiempo real para dispositivos embebidos, de código abierto [47]. Amazon, que es miembro de su comunidad (*partner*), ha desarrollado librerías específicas para integrar dispositivos con este sistema operativo y sus servicios de IoT que se distribuyen de manera gratuita.
- **AWS IoT Greengrass**: pensado para dispositivos de borde, que ejecuten Linux en distribuciones como Ubuntu o Raspbian [48]. Consigue que el dispositivo en el que se instale funcione como un concentrador y pasarela, pudiéndose comunicar con las motas IoT (FreeRTOS o SDK para AWS IoT) y formar grupos de dispositivos que se comuniquen entre sí y con la nube. Además, aparte de actuar reenviando mensajes, un dispositivo con Greengrass puede realizar cierto procesado de los datos que recibe, de esta manera reduce los tiempos de respuesta de algunas aplicaciones y también reduce los mensajes enviados a la nube. Otras de sus características destacables son por ejemplo el funcionamiento con conexión intermitente (puede funcionar online u offline), el uso de conectores predefinidos para ambas direcciones de conexión

(dispositivos motas y aplicaciones en la nube); los cuales simplifican el despliegue, o la seguridad de los mensajes por medio de cifrado y autenticación.

Servicios de control:

- **AWS IoT Core**: es un servicio que se ejecuta en la nube y que actúa de intermediario entre la red IoT y las diferentes aplicaciones que hacen uso de ella, ya sean servicios propios de AWS o de terceros [49]. Puede gestionar millones de dispositivos y billones de mensajes de manera segura y confiable. Además, permite a las aplicaciones que conecta, hacer un seguimiento de los dispositivos IoT, para ello almacena el estado más reciente de estos, de manera que, aunque no tengan conexión en el momento, las aplicaciones siempre los vean conectados. Además, AWS IoT Core permite seleccionar el protocolo de comunicación a usar con los dispositivos IoT y las aplicaciones; admitiendo MQTT, MQTT a través de Web Socket Secure, HTTP e incluso tiene una implementación para dispositivos LoRaWAN [50]; en la cual se elimina la necesidad de operar un servidor de red LoRaWAN.
- **AWS IoT Device Management**: es un servicio que facilita el registro, la organización, la monitorización y la administración remota de dispositivos IoT a gran escala [51]. Permite organizar los dispositivos por grupos jerárquicos o incluso gestionar políticas de los mismos o actualizaciones de firmware.
- **AWS IoT Device Defender**: permite definir configuraciones de seguridad para los dispositivos IoT, para su monitorización posterior [52]. Es capaz de integrar *Machine Learning* para detectar anomalías en el comportamiento de los dispositivos y de enviar medidas y alertas. En general muy útil para auditar la seguridad de una red IoT.

Servicios de análisis:

- **AWS IoT Analytics**: es un servicio que incluye y automatiza todos los pasos necesarios para realizar un análisis de los datos de dispositivos IoT [53]. Para ello, comienza recolectando datos, los cuales pueden ser de diferentes formatos, a continuación, realiza el preprocesado a los datos, permite filtrar, agrupar, transformar o enriquecer los datos recibidos. Una vez preprocesados, los almacena en un almacén específico para series de datos temporales para facilitar el análisis posterior. Tras esto se pueden realizar el análisis de los datos, para ello incluye diferentes opciones, como por ejemplo; por medio de consultas SQL al almacén de datos, con modelos de aprendizaje automático prediseñados para casos típicos o incorporar análisis propios (por medio de código en Junyper Notebooks o incluso con herramientas externas como Matlab).
- **AWS IoT Things Graph**: es un servicio que simplifica la interoperabilidad de componentes Iot y aplicaciones por medio de una interfaz gráfica [54]. De este modo, este servicio acelera la creación de aplicaciones y facilita su administración y monitoreo.

Estos son algunos de los servicios de AWS específicos para IoT. Como se ha visto, Amazon ofrece desde software para conectar los propios dispositivos IoT entre sí y con la nube, a servicios para la gestión de los mismos y el análisis de los datos. Aparte de estos servicios, y sin salir de la propia infraestructura de AWS, existen muchos otros servicios

totalmente integrables, que pueden resultar útiles a la hora de generar una aplicación para IoT, como disparadores de eventos o bases de datos.

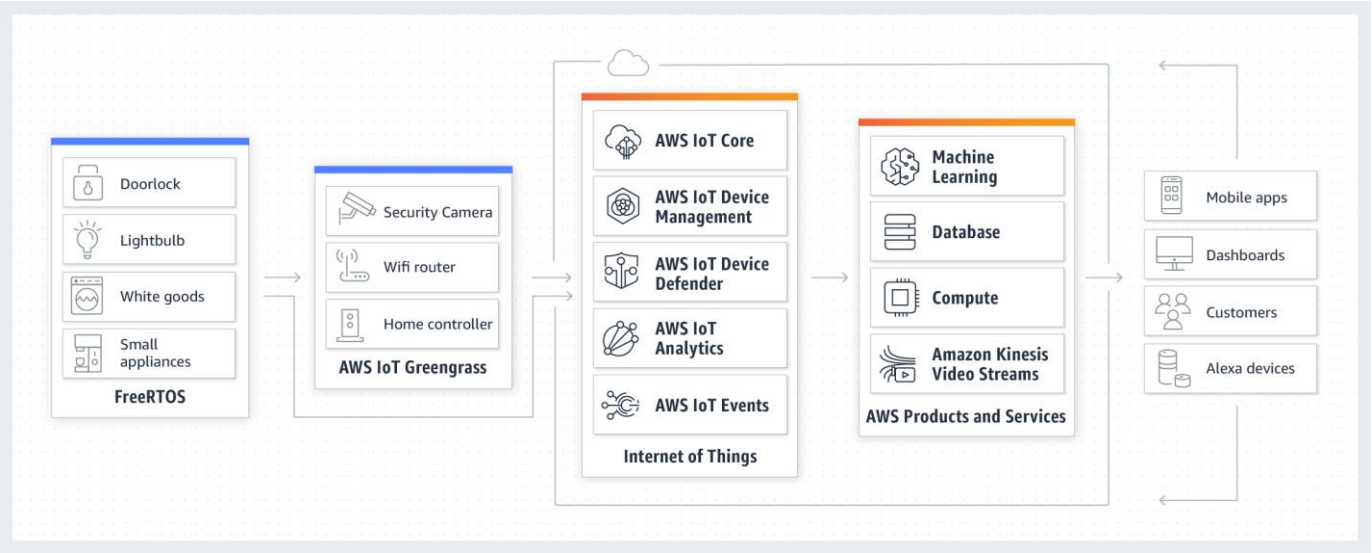


Figura 4.1: Esquema de implementación IoT en AWS [49].

Teniendo en cuenta la integración que persigue el presente trabajo, y su objetivo de obtener predicciones de temperatura, destacamos como servicios adicionales dentro de AWS los siguientes servicios orientados al estudio e implementación de modelos de *Machine Learning*:

- **Amazon Forecast**: es un servicio específico para predecir valores a partir de datos [55]. El usuario solamente debe cargar las series históricas de datos temporales, así como datos relacionados que puedan ser útiles, y Amazon Forecast directamente analiza esos datos, busca los atributos más importantes y genera modelos de predicción en base a ellos. Estos modelos también se pueden personalizar y el propio servicio se encarga de entrenarlos y optimizarlos para finalmente permitir la visualización de resultados o integración de los mismos en otras aplicaciones.
- **Amazon SageMaker**: es el servicio que integra y ofrece todas las herramientas necesarias para desarrollar aplicaciones de aprendizaje automático [56]. Incluye herramientas para el preprocesado de los datos, la creación de modelos, su entrenamiento y optimización, y la monitorización de los mismos mediante sus resultados. Permite el uso de los marcos de programación y herramientas más comunes en *Machine Learning*, entre ellos Pytorch [57].

Haciendo uso de los diferentes servicios introducidos, se puede generar una integración similar a la que se pretende llevar a cabo en este trabajo.

Con esto finaliza este capítulo en el que se han recogido diferentes estudios que hacen uso de las tecnologías implicadas en este proyecto, así como una plataforma que permite su integración haciendo uso de servicios en la nube.

5. Planificación, recursos y costes

En este capítulo se incluyen la planificación de las tareas en las que se ha descompuesto la realización del proyecto y también los recursos y costes asociados de los mismos.

5.1 Planificación de las tareas.

En esta sección se presentan las fases y tareas de las que se ha compuesto la realización de este proyecto y también se incluye una representación de la temporización de las mismas por medio de un diagrama de Gantt:

Descripción de las fases:

1. Preparación: en esta fase inicial se pretende realizar todas las tareas de estudio previo necesarias para llevar a cabo la implementación. También se incluye en esta fase el estudio del estado del arte relacionado con el presente proyecto.
2. Pruebas: esta fase se compone de las tareas destinadas a probar por separado cada una de las tecnologías que forman parte de la implementación.
3. Integración y Resultados: una vez realizadas todas las pruebas previas, en esta fase se lleva a cabo la integración de todas las partes del proyecto y se obtienen los resultados.
4. Redacción de la Memoria: por último, la fase final del proyecto, en la cual se recogen todo el proceso realizado, así como los resultados obtenidos en esta memoria.

Basándonos en estas fases y estimando una dedicación de 2 horas al día, de lunes a viernes; al comienzo del proyecto se crea un diagrama de Gantt con la planificación inicial prevista para alcanzar la consecución del mismo. Esta planificación se muestra en la figura 5.1.

En este diagrama se representan las tareas asociadas a cada fase, y como se observa, la idea llevada a cabo es separar las tareas en función de la tecnología que utilizan; IoT, SDN y *Machine Learning* (ML). Tras realizar en primer lugar todas las tareas de la fase de preparación, es decir, una vez recopilado el conocimiento necesario, se pasa a la fase de pruebas con sus respectivas tareas.

Si bien, una vez terminado el proyecto, se recoge la planificación real llevada a cabo, que es la mostrada en la figura 5.2.

Observando las diferencias entre ambas, podemos ver qué tareas requirieron menos tiempo del necesario, ya que fueron más sencillas que lo previsto o por contra qué tareas se dilataron en el tiempo por encontrarse con dificultades en su realización. Además, algunas de ellas fueron reorganizadas en el tiempo.

Se observa que la duración real en conjunto (desde el día de comienzo hasta el día final) es levemente mayor que la planeada inicialmente.

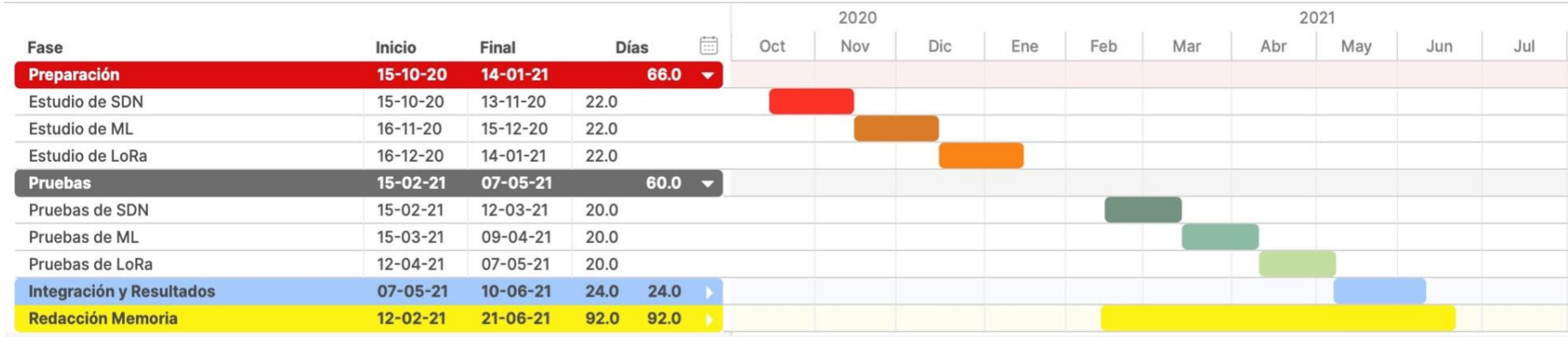


Figura 5.1: Diagrama Gantt de la planificación inicial.

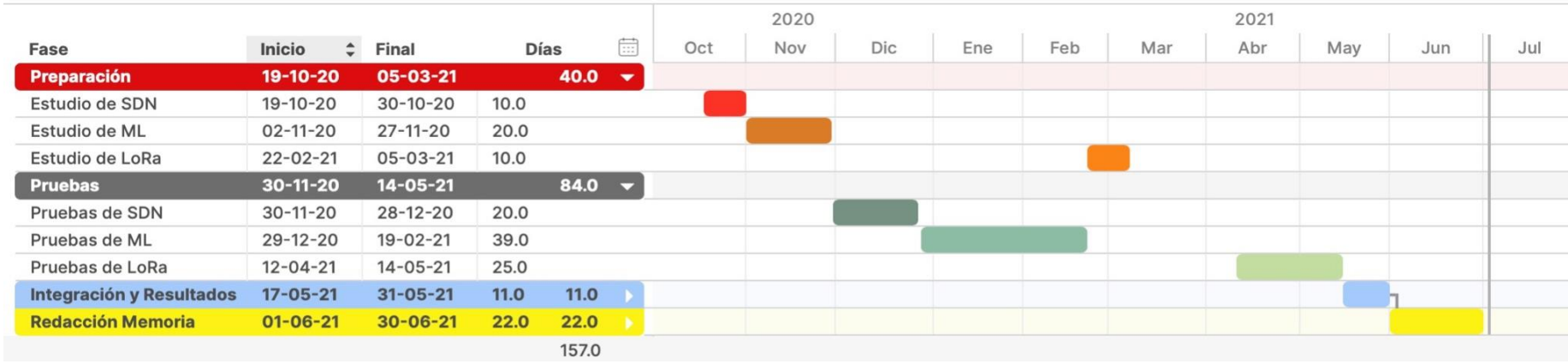


Figura 5.2: Diagrama Gantt de la planificación final.

La tabla 5.1 indica la suma de horas dedicadas a cada una de las tareas:

Tarea	Horas
Estudio SDN	20
Estudio ML	40
Estudio LoRaWAN	20
Pruebas SDN	40
Pruebas ML	78
Pruebas LoRaWAN	50
Integración y Resultados	22
Redacción de la Memoria	44
TOTAL	314

Tabla 5.1: Tareas del proyecto y horas dedicadas a cada una de ellas.

En total se han dedicado aproximadamente 314 horas a la realización del proyecto.

5.2 Recursos y Costes asociados.

Esta sección se centra en los recursos utilizados en el proyecto, así como el coste asociado a cada uno de ellos. Estos recursos se agrupan en recursos humanos, recursos software y recursos hardware.

5.2.1 Recursos humanos.

Los recursos humanos incluyen el tiempo de trabajo dedicado por cada una de las personas involucradas en el proyecto. En este caso, el proyecto ha sido realizado con la participación del alumno (autor) y su tutor.

El cálculo de las horas de trabajo dedicadas por parte del alumno asciende a 314 horas según lo recogido en la tabla 5.1; mientras que el tutor por su parte se considera que ha contribuido con un total de 20 horas, en las cuales se incluyen tutorías e indicaciones, revisión y evaluación del trabajo.

Para estimar el coste por hora se ha considerado la titulación de cada uno de ellos de acuerdo a la situación laboral en España; resultando en un precio por hora de 20€ para el alumno y 40€ para el tutor.

Todo esto se recoge en la tabla 5.2.

Persona	Coste por hora	Total de horas	Coste total
Tutor	40 €	20	800 €
Alumno	20 €	314	6280 €
TOTAL			7080 €

Tabla 5.2: Recursos humanos y costes.

El coste humano total asciende a 7080 € (800 € por el coste del tutor y 6280 € por el alumno).

5.2.2 Coste hardware.

A continuación, se recoge una breve descripción de cada uno de los recursos hardware utilizados para la realización de este proyecto; tras ello se recogen los costes asociados a los mismos en una tabla.

Como PC se ha usado un ordenador portátil MacBook Pro de principios de 2015, el cual está equipado con un procesador Intel Core i5 de 2,7 GHz, memoria RAM de 8 GB DDR3, almacenamiento SSD de 250 GB y una tarjeta gráfica Intel Iris Graphics 6100. Su precio de adquisición fue de 1500€, estimando una vida útil de 8 años (2920 días) y habiéndose utilizado un total de 157 días; el coste asociado a dicho PC es de 80 € aproximadamente.

También se ha hecho uso de una pasarela LoRaWAN, basada en una Raspberry Pi junto con un concentrador iC880A de IMST. Tiene una tensión de alimentación de 5V, soporta frecuencias entre 863-870 MHz para la modulación LoRa y tiene un puerto Ethernet con conector RJ45 con el que nos conectaremos. Su precio es de 200€. A esta pasarela le conectamos una antena CTA 868/2/DR/SM/S2 haciendo uso del interfaz SMA, para poder comunicarse sobre la banda ISM. El precio de la antena es de 6,5€; lo que hace un total para la pasarela más la antena de 206,5€.

La mota LoRa utilizada es una mota Pycom con una placa FiPy, programable en MicroPython y con soporte para 5 redes LPWAN. También incluye una placa de expansión PySense equipada con sensores de humedad, temperatura o luminosidad entre otros; y dos antenas, una antena LTE-M y otra para LoRa y Sigfox. El precio total de estos componentes es de 102,28€, al que se le suma el precio de la carcasa de 15€, en total tiene un coste de 117,28€.

Todos estos costes se recogen en la tabla 5.3:

Recursos Hardware	**Coste**
Portátil	80€
GW + antena	206,5€
Mota	117,28€
Total	403,78€

Tabla 5.3: Recursos hardware y costes.

En total los costes asociados a los recursos hardware suman un total de 403,78€.

5.2.3 Coste software.

A continuación, se recogen todos los recursos software que han sido utilizados en la implementación práctica de este proyecto:

Recursos Software:
Python 3
VirtualBox
ChirpStack

Mosquitto
Ryu
Mininet
InfluxDB
Grafana
Pytorch + Pytorch Forecasting
Wireshark

Tabla 5.4: Recursos software.

El coste de todos estos recursos es nulo ya que se ha buscado y preferido la utilización exclusiva de software gratuito de código abierto. Una explicación de todas estas herramientas se encuentra en el capítulo 3.

5.2.4 Coste total.

Para concluir este capítulo, en la tabla 5.5, se recoge la estimación del gasto total para la realización de este proyecto, calculado como la suma de todos los gastos anteriormente presentados.

Recurso	**Coste**
Humano	7080 €
Hardware	403,78 €
Software	0 €
Total	7483,78 €

Tabla 5.5: Coste total.

El total estimado para la realización del proyecto es de 7483,78€.

6. Diseño e implementación

Este capítulo se centra en explicar cómo se ha llevado a cabo la implementación del proyecto. Para ello se describen en detalle los pasos seguidos con cada una de las herramientas utilizadas, así como los comandos usados y demás, todo con el fin de facilitar una futura reproducción de esta implementación.

Para detallar las acciones realizadas, el orden a seguir en este capítulo es similar al que siguen los datos de los que hace uso la aplicación final, que surge de la integración. Es decir, se comienza explicando la parte asociada a la tecnología LoRa de IoT, donde se miden y capturan los datos del entorno, se continúa con la parte de SDN, encargada de soportar el tráfico de estos datos; y finaliza con la parte de *Machine Learning*, en la cual los datos se procesan.

El esquema de la implementación se muestra en la figura 6.1.

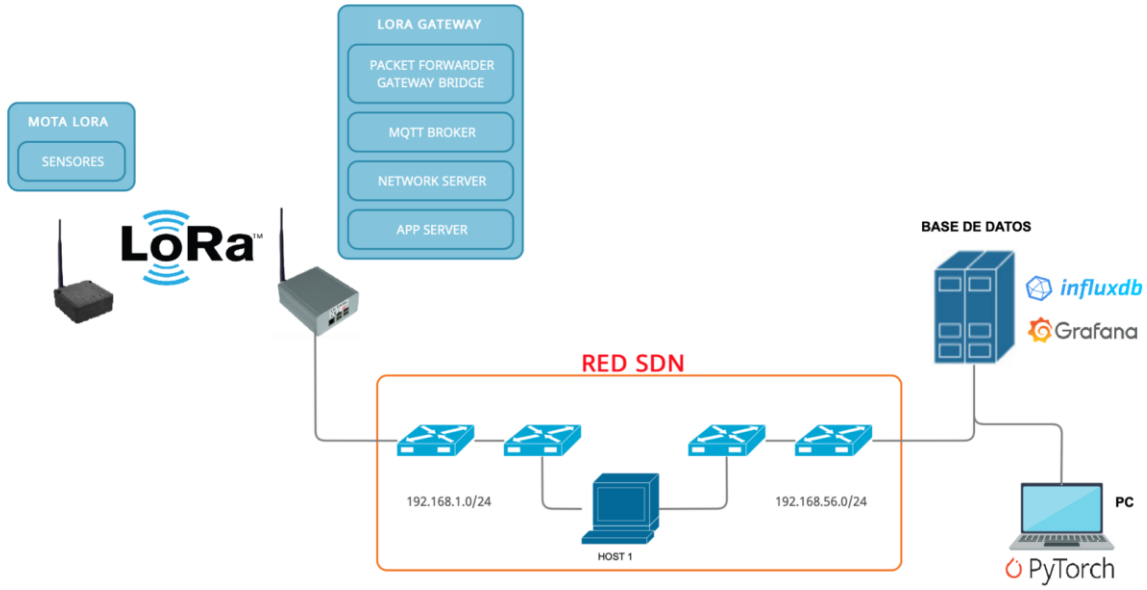


Figura 6.1: Esquema de la implementación.

6.1 Red LoRaWAN

Como se ha mencionado anteriormente, esta es la parte encargada de capturar los datos del medio. Para ello, se compone de la mota de LoRa, la cual está equipada con sensores que miden los datos del entorno; y de la pasarela de LoRa, que es la encargada de recibir dichos datos desde la mota y hacer de enlace con el resto de la red.

6.1.1 Mota LoRa

La mota utilizada es una mota Pycom FiPy, la cual se programa utilizando Micropython. Se ha usado la aplicación gratuita Atom para programarla; para ello, se crea una carpeta con

el proyecto que se quiera ejecutar en la mota y se carga en el dispositivo. El proyecto se compone de ficheros Python en mi caso.

La mota está equipada con una placa de expansión Pycom Pysense la cual contiene un sensor de temperatura y humedad incluidos en el módulo SI7006-A20, fabricado por Silicom Labs. Para hacer que dicho sensor capture datos, incluimos su biblioteca en el proyecto y tras importarla en el programa principal, seremos capaces de llamar a los métodos temperatura() y humedad() para leer los valores del sensor. En la figura 6.2 se muestra un diagrama de flujo del script ejecutado en la mota.

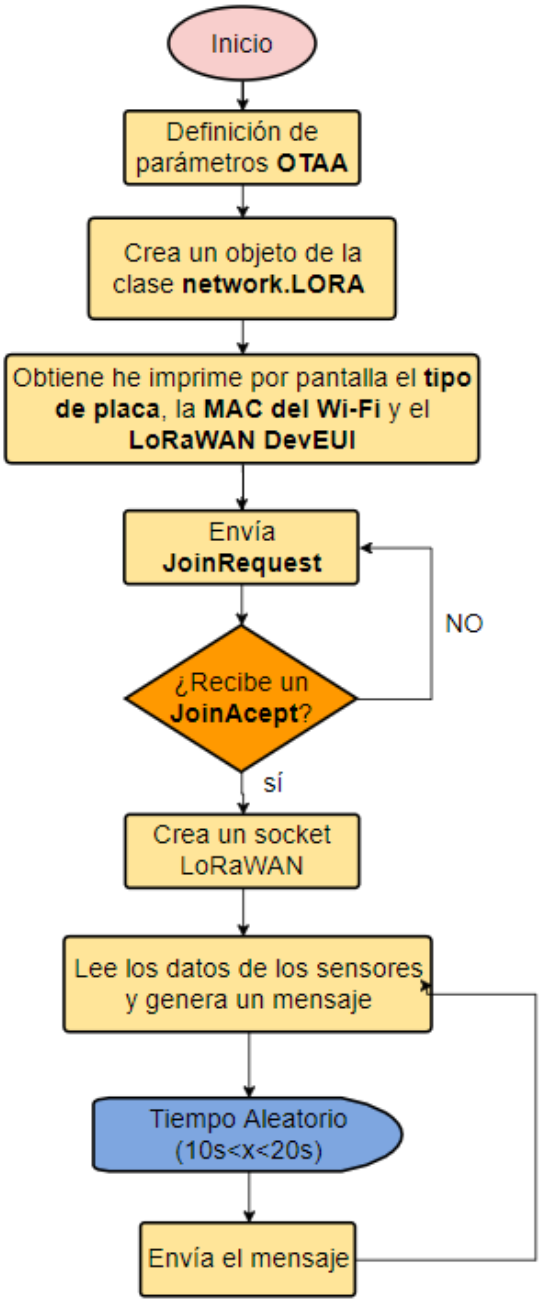


Figura 6.2: Diagrama de flujo del script ejecutado en la mota.

La mota se conecta a la pasarela, utilizando la autenticación OTAA, para ello tiene definido al comienzo del script el valor de los parámetros AppEUI y AppKey y obtiene el DevEUI a partir de la dirección MAC de su interfaz LoRa.

Se utiliza el constructor de la clase *network.LoRa* (previamente importada) para instanciar un objeto con los parámetros de configuración necesarios; por ejemplo la región Europa (*region=LoRa.EU868* para las frecuencias) o el tipo de dispositivo de clase C (*device_class=LoRa.CLASS_C*); entre otros.

Usando la clase *sys* y la clase *pycoproc*; se obtienen y muestran por pantalla diferentes datos de la mota, como la placa detectada, la placa de extensión, la dirección MAC de la interfaz Wi-Fi o el DevEUI de LoRaWAN. La imagen 6.3 muestra estos valores.

```
Connecting to /dev/tty.usbmodemPycc9bb81...
[INFO] Detected board: FiPy
Detected expansion board: PySense (HW version 5, FW version 15)
Expansion board identifier: 61458 (HW version 5, FW version 15)
[INFO] Wi-Fi MAC:      807D3AC31CB4
[INFO] LORAWAN DevEUI: 70B3D549953CF6C5
[INFO] Not joined yet...
[INFO] Not joined yet...
[INFO] Not joined yet...
[INFO] Not joined yet...
[INFO] --- Joined Sucessfully ---
```

Figura 6.3: Parámetros de la mota.

El script continúa usando el objeto *network.LoRa*, anteriormente instanciado para conectarse a la pasarela y crear un *socket*. Para ello espera a que la conexión sea exitosa antes de devolverlo y seguir con el resto del script.

Tras esto el código ejecuta un bucle infinito en el que cada cierto tiempo (intervalo aleatorio de mínimo 10 segundos), la mota lee los valores de los sensores; de temperatura y de humedad, y los envía a la pasarela con un formato JSON (JavaScript Object Notation), para que posteriormente sea más fácil su procesamiento.

```
[INFO] Waiting 16.830 s for next transmission...
[INFO] Message:
 "{"Temperatura": 25.07251,
 "Humedad": 54.83194}"
 sent at 00:00:47.489137
```

Figura 6.4: Salida del script en la mota, formato JSON del mensaje que envía.

6.1.2 Pasarela LoRaWAN

La pasarela se ha implementado en una Raspberry Pi, la cual ejecuta un Raspbian Buster (sistema operativo basado en Debian); conectada con un concentrador IMST. Nos conectamos con SSH (*Secure Shell*) para configurarla. En mi caso, para conectarme a la

pasarela, lo que he hecho es conectarla al router de mi casa (ya que mi PC no dispone de entrada Ethernet con conector RJ45, que es el utilizado por la pasarela), con DHCP (*Dynamic Host Configuration Protocol*) activado obtiene una dirección IP privada en la red WiFi de mi casa y desde mi PC me conecto por SSH a esa IP.

Para activar la configuración del dispositivo pasarela es necesario instalar y arrancar, por un lado, el servicio de *gateway* (que incluye el *packet forwarder*), y, por otro lado, los servidores LoRaWAN de la plataforma ChirpStack; compuestos por el servidor *bridge*, el de red y el de aplicación. Estos servidores se han instalado todos en la misma máquina, aunque esta no es la mejor opción en términos de seguridad, escalabilidad y eficiencia. Todos los comandos recogidos en esta sección han sido por tanto ejecutados en la propia Raspberry Pi.

La instalación del *gateway* está basada en [58], un repositorio git para una pasarela iC880a, desarrollado por *The Things Network*, concretamente por su comunidad de Zúrich. En ella se hace uso de un programa *packet forwarder* basado a su vez en el código desarrollado por Semtech; propietarios de la modulación LoRa.

TTN [59] es miembro de la LoRa Alliance, se trata de una iniciativa comunitaria abierta para establecer un entorno de IoT global usando LoRaWAN. En dicho entorno se han desarrollado herramientas de código abierto como servidores, aplicaciones y despliegues de red. Actualmente TTN ejecuta *The Things Stack Community Edition*; que es una red LoRaWAN abierta y descentralizada en la cual los usuarios pueden probar nuevas aplicaciones, dispositivos o integraciones y familiarizarse con LoRaWAN.

Tras seguir los pasos de la instalación, el fichero de configuración del *packet forwarder* se puede encontrar en '*local_conf.json*'; dónde podemos editar el gateway ID o la dirección del servidor a donde se reenvían los paquetes; como se muestra en la figura 6.5.

Figura 6.5: Fichero de configuración del *packet forwarder*.

En nuestro caso, el servidor al que se reenvían los paquetes es el LoRa-Gateway-Bridge, que se instalará en la misma máquina ("*127.0.0.1*"), el cual además escucha en el puerto 1700. Para arrancar el servicio de *packet forwarder* en la pasarela hacemos uso del comando mostrado en la figura 6.6.

```
[ttn@test-gw02:~ $ sudo systemctl start ttn-gateway
[ttn@test-gw02:~ $ sudo systemctl status ttn-gateway
● ttn-gateway.service - The Things Network Gateway
   Loaded: loaded (/lib/systemd/system/ttn-gateway.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2021-06-23 12:41:41 CEST; 26s ago
 Main PID: 831 (start.sh)
   Memory: 564.0K
   CGroup: /system.slice/ttn-gateway.service
           ├─831 /bin/bash /opt/ttn-gateway/bin/start.sh
           └─841 ./poly_pkt_fwd
```

Figura 6.6: Arranque del servicio *ttn-gateway*.

La plataforma ChirpStack (explicada en la sección 3.2) se compone de diferentes servidores, se utiliza como red privada y además hace uso de unos paquetes software externos, que es necesario instalar previamente para su correcto funcionamiento. Estos prerrequisitos incluyen la instalación de un *broker* MQTT (se ha usado Mosquitto); así como las bases de datos Redis y PostgreSQL.

Para instalar y configurar la plataforma se ha seguido la guía de inicio recogida en la web oficial [60]. Los pasos a seguir son los siguientes:

1. Instalamos todos los paquetes asociados a los prerrequisitos con el comando:

"*sudo apt install mosquitto mosquitto-clients redis-server redis-tools postgresql*".

2. Configuramos PostgreSQL; creamos la base de dato del servidor de aplicación y la base de datos del de red y sus respectivos usuarios.
3. Configuramos el repositorio de ChirpStack para poder obtener los paquetes haciendo uso del gestor de paquetes *apt*.
4. Instalamos el *ChirpStack Gateway Bridge*, y en su fichero de configuración "*/etc/chirpstack-gateway-bridge/chirpstack-gateway-bridge.toml*" indicamos como dirección del *broker* MQTT, *localhost*, ya que estamos instalando todo en la Raspberry Pi.

```
# Generic MQTT authentication.
[integration.mqtt.auth.generic]
# MQTT server (e.g. scheme://host:port where scheme is tcp, ssl or ws)
server="tcp://127.0.0.1:1883"

# Connect with the given username (optional)
username=""

# Connect with the given password (optional)
password=""
```

Figura 6.7: Fichero de configuración del *chirpstack-gateway-bridge*.

Como se observa en la figura 6.7, no se ha usado autenticación con usuario y contraseña para el intercambio de mensajes MQTT

5. Instalamos el *ChirpStack Network Server*; en su fichero de configuración "*/etc/chirpstack-network-server/chirpstack-network-server.toml*" indicamos la base de datos creada en el paso 2, así como la configuración de la banda de frecuencias.
6. Instalamos el *ChirpStack Application Server*, y en su fichero de configuración en "*/etc/chirpstack-application-server/chirpstack-application-server.toml*", debemos indicar la base de datos creada en el paso 2.

7. Tras esto, ya podemos activar los servidores ChirpStack, para ello utilizamos los comandos mostrados en las figuras 6.8, 6.9 y 6.10.

```
ttn@test-gw02:~ $ sudo systemctl start chirpstack-gateway-bridge
ttn@test-gw02:~ $ sudo systemctl status chirpstack-gateway-bridge
● chirpstack-gateway-bridge.service - ChirpStack Gateway Bridge
   Loaded: loaded (/lib/systemd/system/chirpstack-gateway-bridge.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2021-06-23 12:44:02 CEST; 1min 19s ago
     Docs: https://www.chirpstack.io/
 Main PID: 948 (chirpstack-gate)
   Memory: 1.8M
   CGroup: /system.slice/chirpstack-gateway-bridge.service
           └─948 /usr/bin/chirpstack-gateway-bridge
```

Figura 6.8: Arranque del servicio *chirpstack-gateway-bridge*.

```
ttn@test-gw02:~ $ sudo systemctl start chirpstack-network-server
ttn@test-gw02:~ $ sudo systemctl status chirpstack-network-server
● chirpstack-network-server.service - ChirpStack Network Server
   Loaded: loaded (/lib/systemd/system/chirpstack-network-server.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2021-06-21 14:03:33 CEST; 1 day 22h ago
     Docs: https://www.chirpstack.io/
 Main PID: 378 (chirpstack-netw)
   Memory: 22.3M
   CGroup: /system.slice/chirpstack-network-server.service
           └─378 /usr/bin/chirpstack-network-server
```

Figura 6.9: Arranque del servicio *chirpstack-network-server*.

```
ttn@test-gw02:~ $ sudo systemctl start chirpstack-application-server
ttn@test-gw02:~ $ sudo systemctl status chirpstack-application-server
● chirpstack-application-server.service - ChirpStack Application Server
   Loaded: loaded (/lib/systemd/system/chirpstack-application-server.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2021-06-21 14:03:34 CEST; 1 day 22h ago
     Docs: https://www.chirpstack.io/
 Main PID: 381 (chirpstack-appl)
   Memory: 29.2M
   CGroup: /system.slice/chirpstack-application-server.service
           └─381 /usr/bin/chirpstack-application-server
```

Figura 6.10: Arranque del servicio *chirpstack-application-server*.

Tras activar los servidores LoRa Server, podemos acceder vía web a la interfaz gráfica del servidor de aplicación, donde se pueden configurar diferentes parámetros de la red. Para acceder a dicha web (generada en este caso en la IP de la pasarela, en el puerto 8080) es necesario autenticarse con usuario y contraseña; una vez logueados seguimos los siguientes pasos para ajustar la red:

1. Creamos una organización, como se muestra en la figura 6.11.

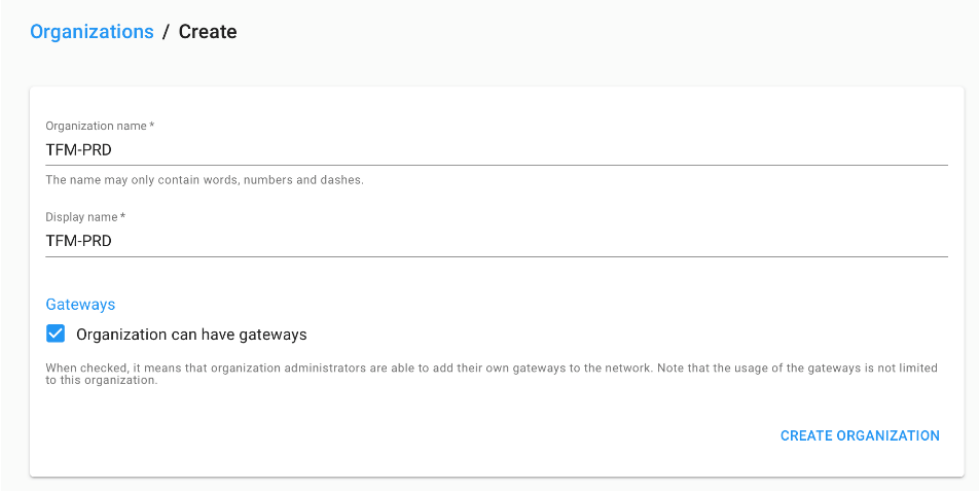


Figura 6.11: Creación de una organización en el servidor de aplicación de ChirpStack.

2. Creamos un servidor de red (indicando la dirección y el puerto del servidor de red), como se muestra en la figura 6.12.

Figura 6.12: Creación de servidor de red en el servidor de aplicación de ChirpStack.

3. Creamos un perfil de servicio (elegimos el servidor de red previamente creado), como se muestra en la figura 6.13.

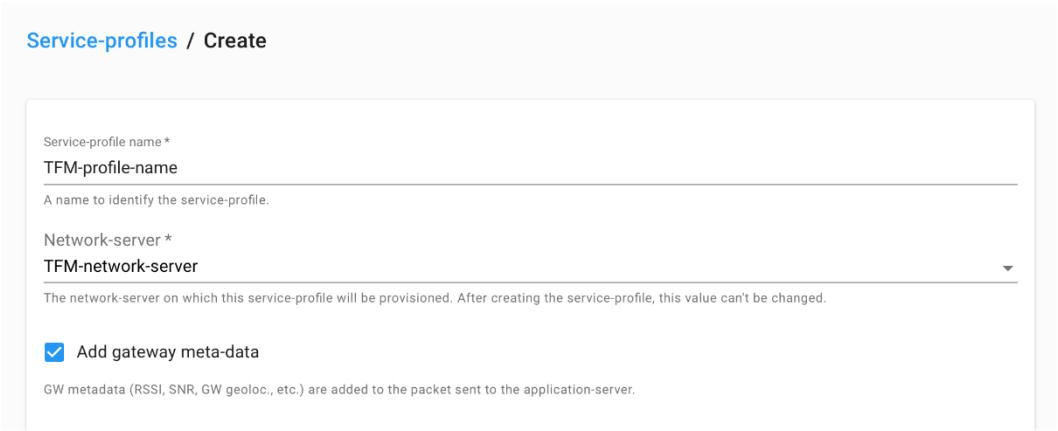


Figura 6.13: Creación de un perfil de servicio en el servidor de aplicación de ChirpStack.

4. Creamos un gateway (indicando el gateway ID), este paso se puede observar en la figura 6.14.
5. Creamos un perfil de dispositivo (indicando el servidor de red y que utilice la autenticación OTAA).
6. Creamos una aplicación (indicando el perfil de servicio creado), como se observa en la figura 6.15.
7. Dentro de la aplicación, creamos un dispositivo (indicando el DevEUI de la mota, para autenticación del nodo).
8. Desactivamos el chequeo del número de trama, ya que no es necesario porque el presente proyecto tiene un fin explicativo (de este modo facilitamos las pruebas).

Estos últimos pasos se observan en la figura 6.16.

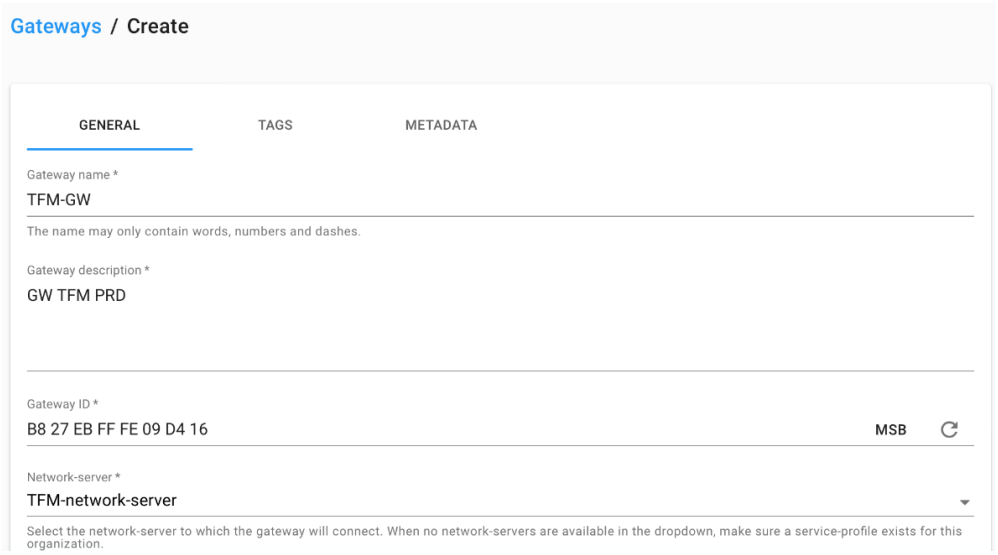


Figura 6.14: Creación de un gateway en el servidor de aplicación de ChirpStack.

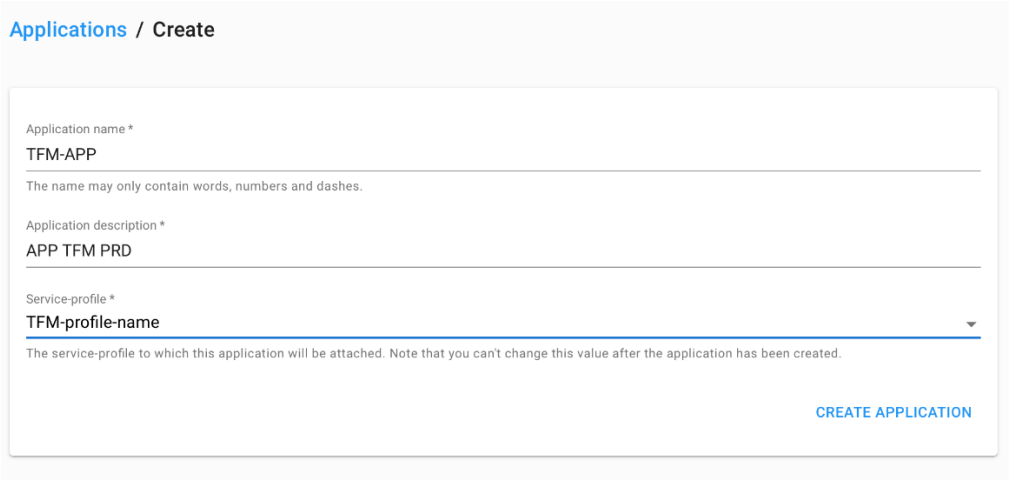


Figura 6.15: Creación de una aplicación en el servidor de aplicación de ChirpStack.

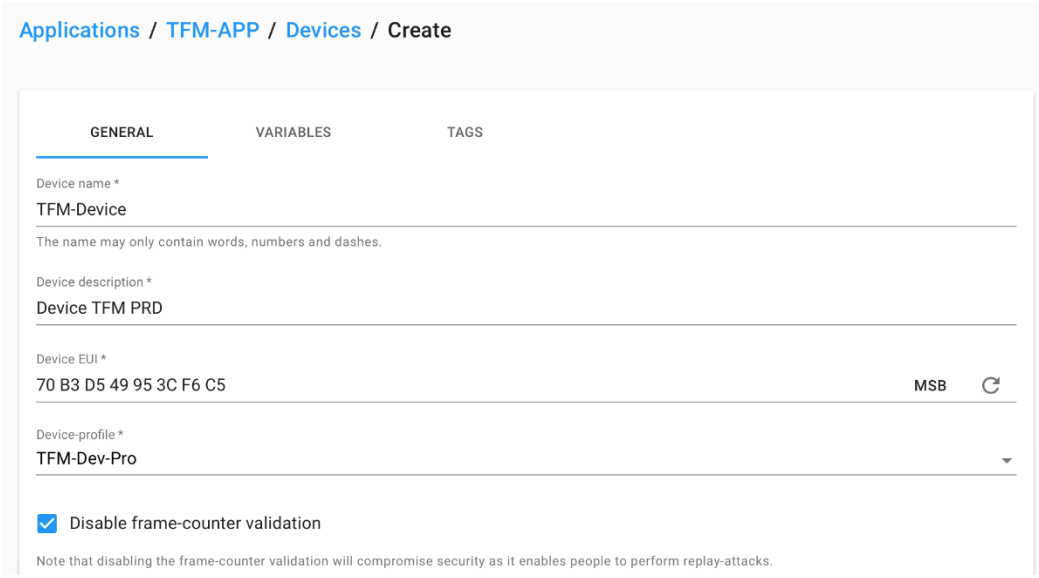


Figura 6.16: Creación de un dispositivo en el servidor de aplicación de ChirpStack.

Una vez realizados estos pasos, se comprueba que efectivamente están conectados y que la mota envía correctamente mensajes y datos a la pasarela. En la web del servidor de aplicación ChirpStack, podemos observar los datos que está enviando la mota, como se muestra en la figura 6.17.

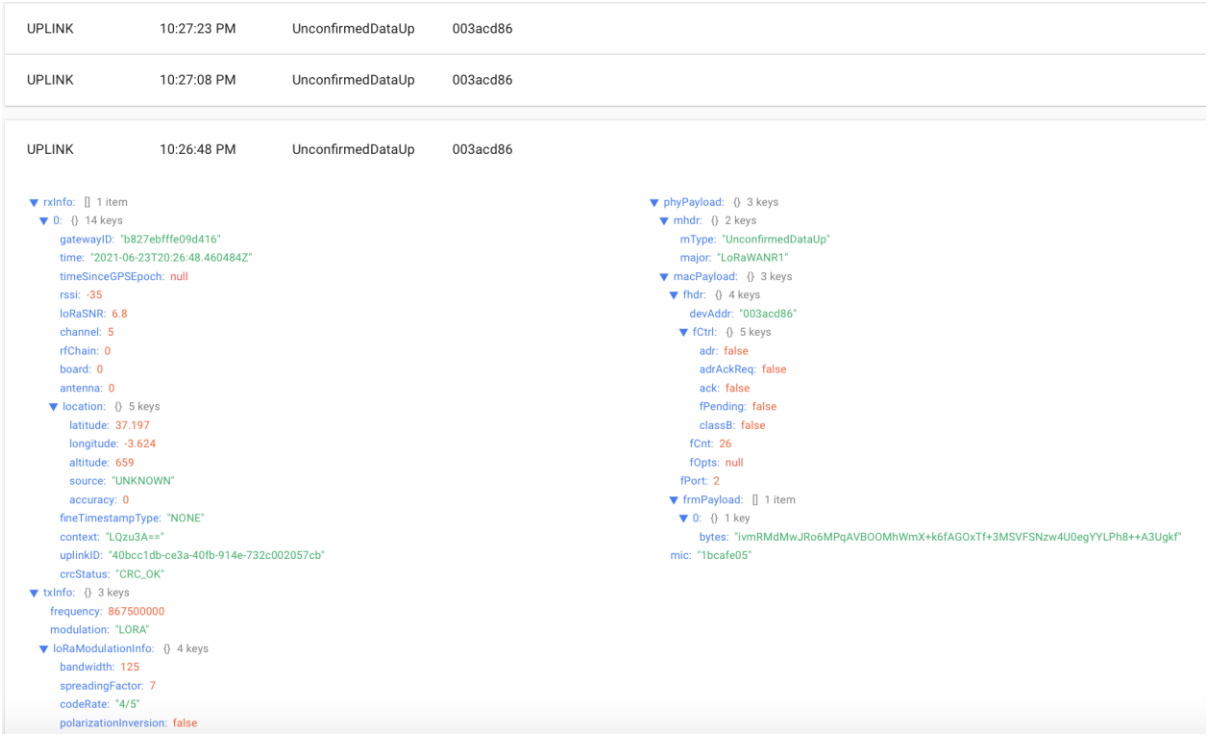


Figura 6.17: Paquetes recibidos mostrados en la web.

Los datos que envía la mota como mensajes, es decir; la información de temperatura y humedad se encuentra en el campo "*bytes*", y se codifica en base64

Tras esto sólo nos queda comprobar el funcionamiento del servicio encargado de retransmitir los datos usando MQTT. Para ello el servidor de aplicación proporciona una integración, en la cual los mensajes se publican en el siguiente tópico "*application/[ApplicationID]/device/[DevEUI]/[EventType]*". Para leer dichos mensajes usaremos Mosquitto y nos suscribimos, por ejemplo, al tópico "*application/#*" de manera que seamos capaces de visualizar todos los mensajes publicados a los subtópicos hijos.

6.2 Red SDN

La siguiente parte consiste en diseñar una red SDN a la cual se conectará la pasarela y por la cual viajarán los paquetes MQTT.

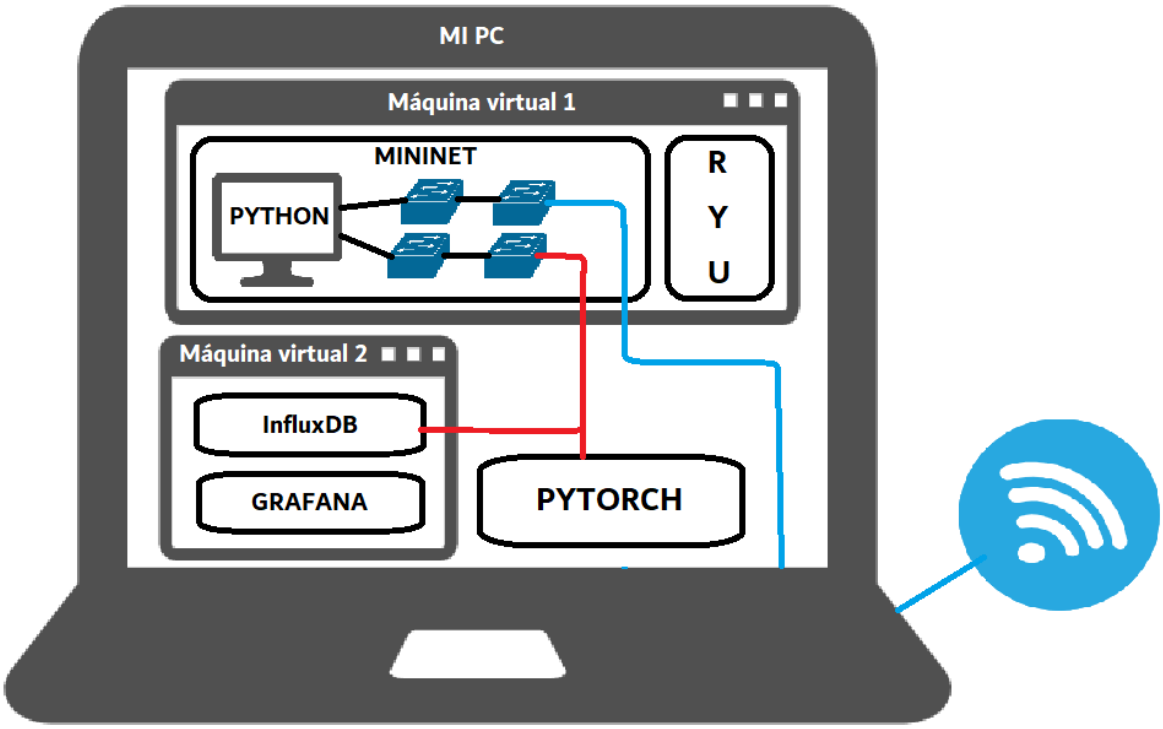


Figura 6.18: Esquema del escenario ejecutado en mi PC.

La red SDN se conecta por un lado a la pasarela (con una interfaz externa conectada a la red WiFi que da acceso a Internet) y por otro lado con la base de datos (con una interfaz externa conectada a una red sólo anfitrión, ya que la base de datos se ha instalado en una máquina virtual). En medio de la SDN hay un *host* virtual que obtiene direcciones en ambas redes y ejecuta un script de Python 3, en el cual se utiliza un cliente MQTT para suscribirse al tópico y recibir mensajes de la pasarela; se procesan y traducen los datos recibidos, y un cliente de la base de datos envía solicitudes para almacenarlos en ella.

6.2.1 Red en Mininet y conexiones externas

La figura 6.19 muestra el esquema de la topología de red que se va a utilizar en Mininet; en ella existen cuatro OpenvSwitch conectados al mismo controlador. Entre el *switch* S2 y el S3 se sitúa el *host* virtual H1.

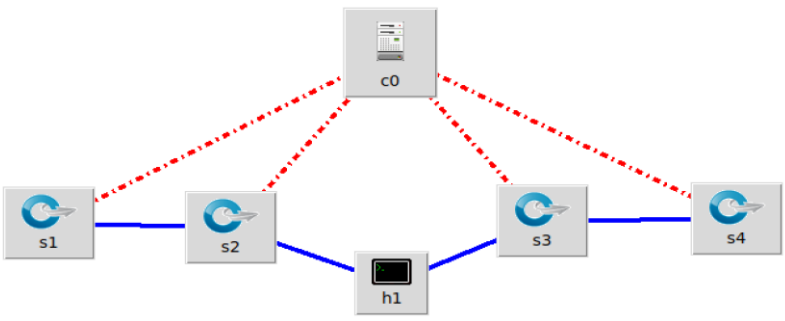


Figura 6.19: Esquema de la topología en Mininet.

Para que dicha red funcione, el primer paso a seguir es activar el controlador que gestionará los equipos de red (*switches*). En nuestro caso es el controlador Ryu, el cual, escucha en el puerto 6653, y al que le debemos indicar la aplicación a usar, es decir el nombre del fichero Python que la define; en este caso *simple_switch_13.py*.

```
jorge@jorge-sdn-vm:~/ryu$ ./bin/ryu-manager --verbose ryu/app/simple_switch_13.py
loading app ryu/app/simple_switch_13.py
loading app ryu.controller.ofp_handler
instantiating app ryu/app/simple_switch_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK SimpleSwitch13
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])}
  PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])}
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPHello
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPEchoReply
  CONSUMES EventOFPPortStatus
```

Figura 6.20: Arranque del controlador Ryu y el *simple_switch_13*.

Una vez lanzado el controlador, el siguiente paso es lanzar Mininet y cargar la topología deseada, para lo cual hacemos uso de un script. En dicho script de Python debemos añadir las interfaces externas a la red SDN que vamos a crear. Para ello indicamos el nombre de las interfaces de red de la máquina donde se esté ejecutando Mininet (máquina virtual en nuestro caso) y las asociamos cada una a un *switch*; la red WiFi al S1 y la red sólo anfitrión al S4.

Para el *host* virtual indicamos en el script que obtenga las direcciones por DHCP. Para que funcione la comunicación del *host* virtual con la red WiFi, indicamos que la dirección MAC de la interfaz conectada a esta red en el *host* virtual, sea la misma que la MAC del adaptador de la máquina virtual (esto es porque VirtualBox en su versión 6.0 no incluye virtualización anidada para procesadores Intel, y por lo tanto no trabaja bien virtualizando doblemente una interfaz física como es la interfaz WiFi de mi PC). Para el caso de la comunicación con la red sólo anfitrión, esto no es necesario. En el Anexo A.2 se incluye el código correspondiente a la topología de Mininet.

```
jorge@jorge-sdn-vm:~/mininet/examples$ sudo python topo1.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** h1 : ('dhclient h1-eth0',)
*** h1 : ('dhclient h1-eth1',)
*** Starting CLI:
mininet>
```

Figura 6.21: Lanzamiento de la topología en Mininet.

Los *switches* de la red generada con Mininet ejecutan OVS. Todos los *switches* se conectan al controlador Ryu que se ha lanzado previamente, y este los configura para que actúen de igual modo que un *learning switch*. Esto hace que los *switches* operen como un conmutador de capa 2, gestionando el controlador, para cada uno de ellos, una tabla con las direcciones MAC y las interfaces por las que están conectadas para el reenvío de tramas. Si se recibe una trama con una dirección de destino para la que no existe una entrada en la tabla de direcciones MAC, se realiza inundación, que consiste en reenviar la trama por todas las interfaces menos por la interfaz por el que se recibió (para evitar colisiones). En el Anexo A.1 se explica el código utilizado en el controlador.

Tras esto, ya estaría ejecutándose la red SDN; si bien para comprobar su correcto funcionamiento podemos realizar algunas pruebas, descritas a continuación.

En lo referente a los *switches* OVS de la topología, una vez configurados, podemos observar los flujos que tienen utilizando el comando *"sh ovs-ofctl dum-flows"*. La imagen 6.22 muestra la salida de este comando para el *switch* S2.

```
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=1201.581s, table=0, n_packets=502, n_bytes=30320, priority=1,in_port="s2-eth1",dl_src=88:03:55:61:5d:a6,dl_dst=08:00:27:42:c9:
d5 actions=output:"s2-eth2"
 cookie=0x0, duration=1201.542s, table=0, n_packets=5, n_bytes=528, priority=1,in_port="s2-eth2",dl_src=08:00:27:42:c9:d5,dl_dst=88:03:55:61:5d:a6 a
ctions=output:"s2-eth1"
 cookie=0x0, duration=768.807s, table=0, n_packets=3721, n_bytes=719804, priority=0 actions=CONTROLLER:65535
```

Figura 6.22: Flujos del OVS S2.

Los dos primeros flujos mostrados en la imagen 6.22 corresponden a la misma comunicación en las diferentes direcciones. El último flujo corresponde a la regla por defecto; con prioridad 0, para enviar paquetes al controlador si no coinciden con ningún flujo guardado.

Otra comprobación que podemos llevar a cabo es la de la configuración de las interfaces de red el *host* virtual tal y como se muestra en la figura 6.23. La conectividad es correcta ya que el *host* virtual ha obtenido las direcciones IP de sus interfaces por DHCP. La red 192.168.1.0/24 es la red externa correspondiente a la red WiFi de mi casa; con la que se conecta a la pasarela; y la red 192.168.56.0/24 es la red sólo anfitrión; con la que se conecta a la base de datos.

También podemos comprobar la conectividad externa usando ping como se muestra en la figura 6.24.

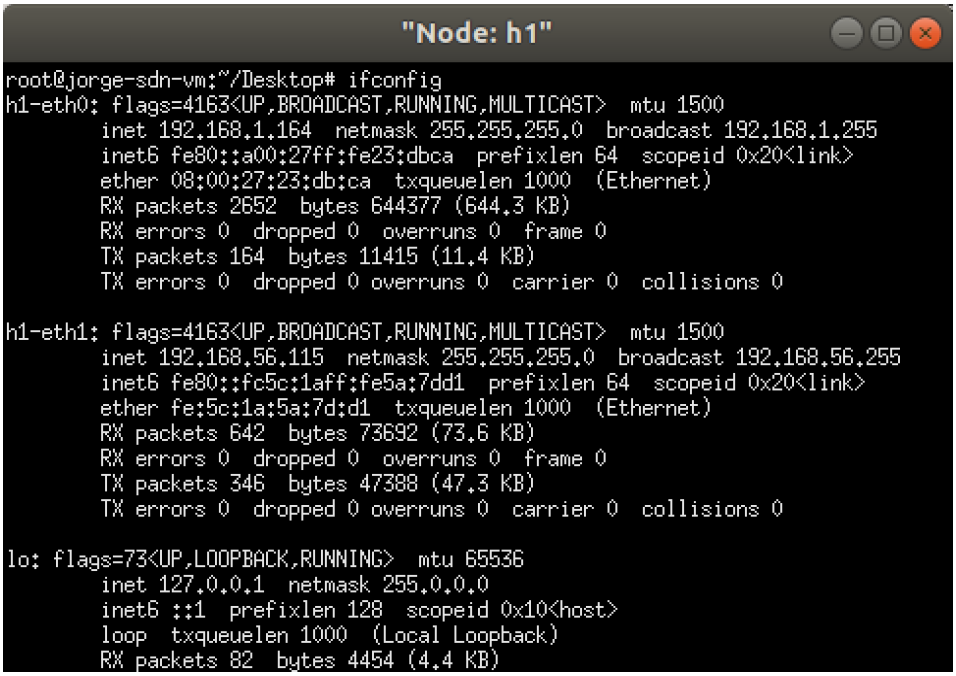


Figura 6.23: Configuración del *host* virtual.

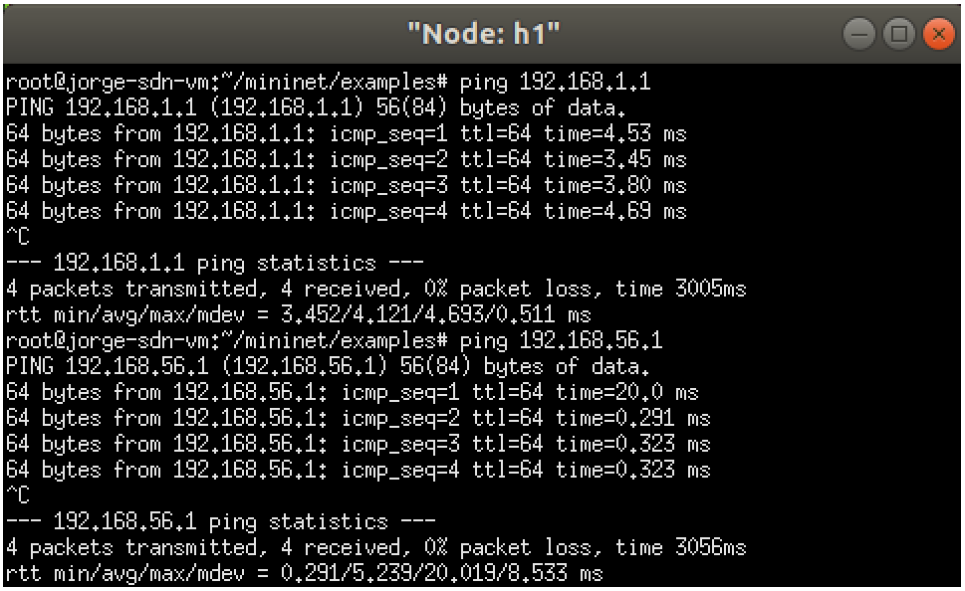


Figura 6.24: Ping desde *host* virtual a ambas redes.

6.2.2 Traducción de MQTT a la base de datos InfluxDB

El script de Python que ejecuta el *host* virtual en mitad de la red, hace uso de la librería *paho.mqtt*, que proporciona el cliente de MQTT en Python y de la librería *InfluxDBClient* para usar el cliente Python de la base de datos InfluxDB (para que estos paquetes estén disponibles en el *host* virtual es necesarios instalarlos en la máquina que ejecuta Mininet, en nuestro caso la máquina virtual). El script crea un cliente específico para nuestra base de datos y se conecta a ella, se subscribe al tópico MQTT y entra en un bucle infinito en el cual al recibir un mensaje por ese tópico lo procesa y traduce, y por último escribe los datos de

temperatura y humedad en la base de datos con el formato adecuado. En el Anexo A.3 se incluye el código del script utilizado en el *host* virtual.

Para lanzar dicho script, abrimos un terminal en el *host* virtual (desde Mininet con el comando "*xterm h1*"), y ejecutamos dicho script con permisos de superusuario. La figura 6.25 recoge este proceso y la salida del mismo.

Figura 6.25: Salida del script *puente.py* en el *host* virtual.

El script que traduce de MQTT a InfluxDB, muestra como salidas, en primer lugar, el resultado de la conexión con la base de datos ("*code 0*" en caso exitoso), tras esto cada vez que recibe un mensaje al tópico MQTT especificado, imprime por pantalla el tópico completo del mensaje recibido y el contenido del mismo; decodifica e imprime los datos (campo "*data*" codificado en base64) e indica con mensajes por pantalla que escribe dichos datos en la base de datos.

InfluxDB se ha instalado con las opciones por defecto, escuchando en el puerto 8086 y sin usuario ni contraseña para la base de datos que usaremos. Dicha base de datos se organiza de la siguiente manera; tiene dos tipos de medidas (*measurements*), la humedad y la temperatura; y una etiqueta (*tag*) "*localization*", para distinguir entre varias motas en caso necesario. En esta implementación, dado que solamente disponemos de una mota, solo existe una localización. Para cada combinación de los campos anteriores tiene los valores medidos y las marcas temporales asociadas.

Una vez lanzado el demonio con el comando *influxd*, la base de datos está corriendo. En la consola de eventos podemos comprobar que se están escribiendo los datos como se muestra en la figura 6.26.

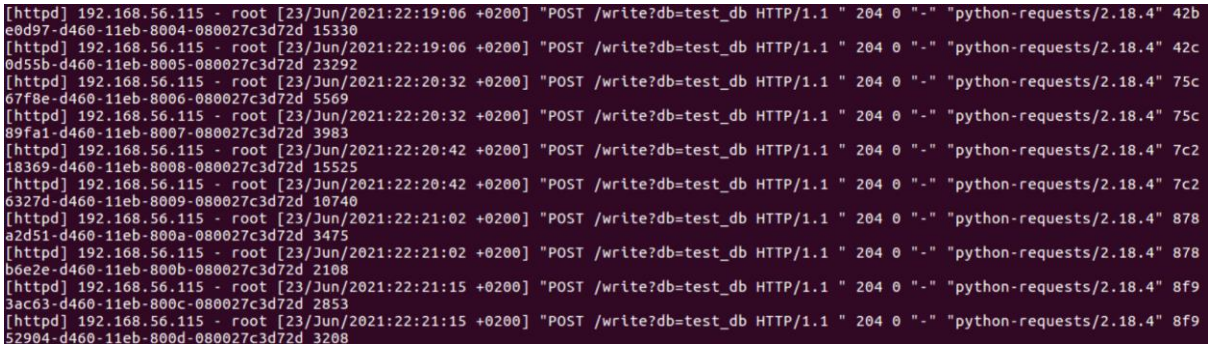


Figura 6.26: Consola de InfluxDB.

6.2.3 Visualización con Grafana

También se ha usado Grafana, concretamente la implementación que utiliza InfluxDB como fuente de datos para crear un panel de visualización con un gráfico de los mismos [61]. Para ello desde la interfaz web de Grafana, una vez autenticados (con el usuario y contraseña por defecto; *admin* para ambos campos), seleccionamos *Data Sources* y *Add New*. Nos aparece entonces un menú en el que debemos elegir *InfluxDB 0.9.x* en el tipo de fuente de datos, indicar la dirección donde se encuentra instalada InfluxDB, en nuestro caso es la misma máquina en el puerto 8086; y en los detalles de InfluxDB, indicamos el nombre de la base de datos en concreto que vamos a usar y rellenamos el usuario y contraseña de la misma (en nuestro caso no usamos contraseña por lo que usamos *root* para ambos). Una vez rellenados todos los campos podemos comprobar que funciona usando la opción *Test Connection* como se muestra en la figura 6.27.

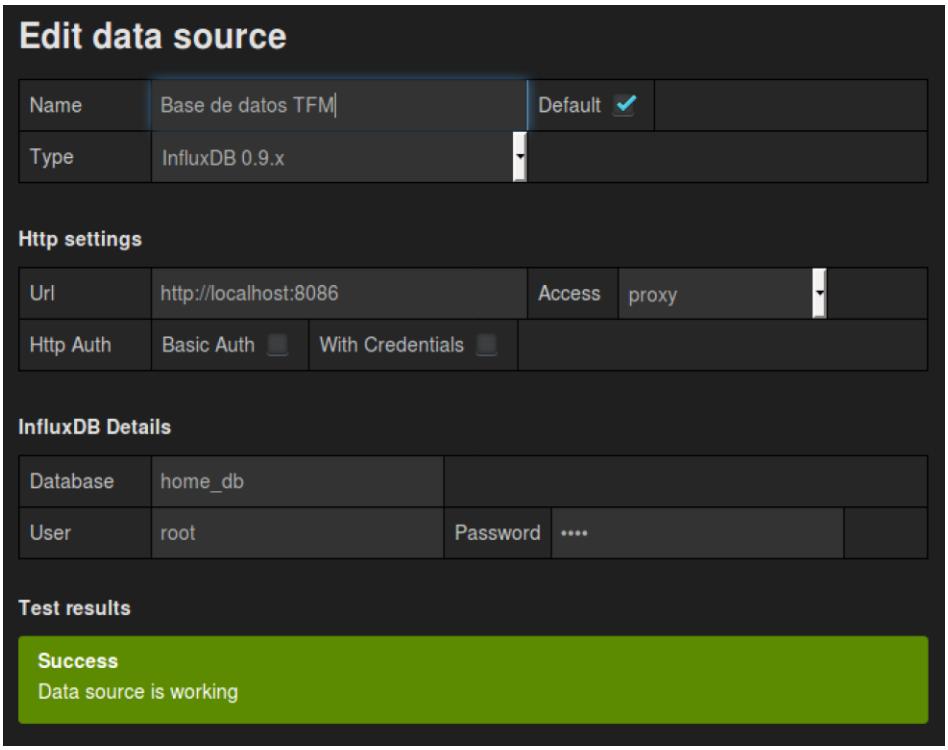


Figura 6.27: Adición de una nueva fuente de datos InfluxDB.

Una vez configurada la fuente de datos, somos capaces de crear paneles con gráficos que representen dichos datos. Para ello en el menú de creación de un nuevo panel en un *dashboard*, Grafana nos permite introducir *queries* a mostrar. Por ejemplo, en la imagen 6.28 se observa que la temperatura, capturada por la mota, aumenta desde un valor inicial de aproximadamente 25º C hasta alcanzar rápidamente valores cercanos a los 40ºC. Esto es debido al calentamiento que sufre la placa donde se encuentra el sensor en la mota.

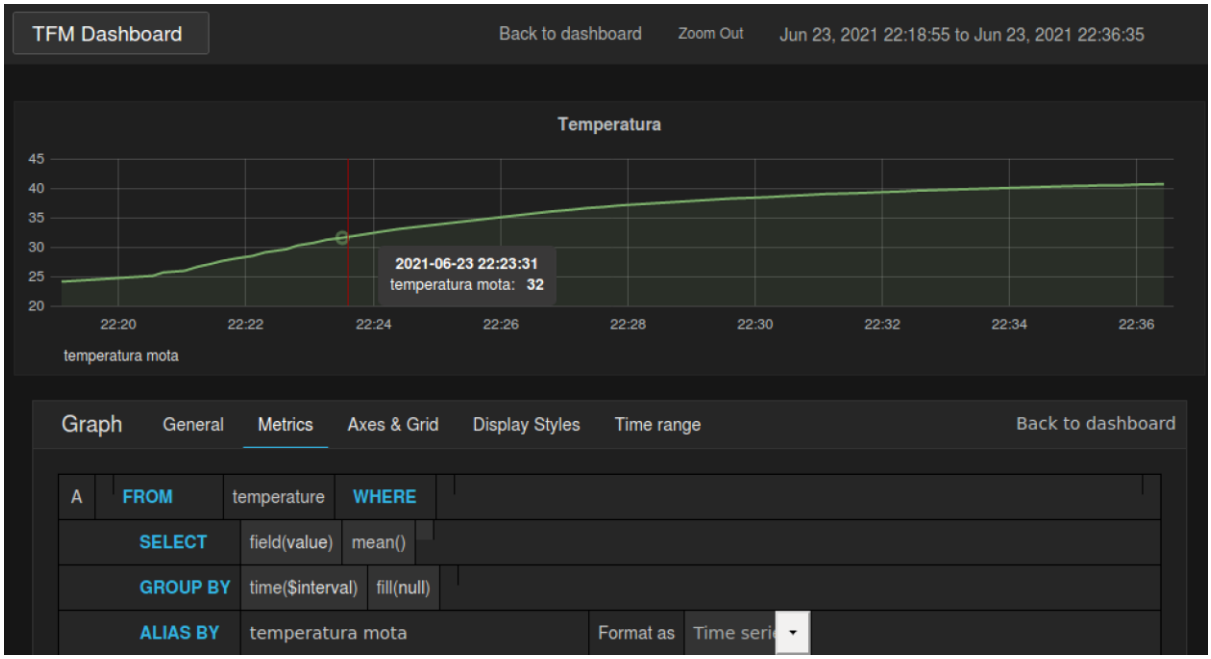


Figura 6.28: Gráfico de la temperatura capturada por la mota, en Grafana.

6.3 Procesado de datos con *Machine Learning*

Por último, para la parte de *Machine Learning*, se ha usado Pytorch Forecasting; una librería específica de Pytorch, que incluye varios modelos para predicciones de series temporales. Se ha instalado en un PC utilizando Anaconda y se ha usado con Juniper Notebooks.

Anaconda [62] es una distribución de Python, la cual viene con una serie de paquetes por defecto ya instalados y que está orientada a la computación científica. Entre sus componentes principales destaca *conda* [63], que es un gestor de paquetes y de entornos, de código abierto, gracias al cual se simplifica la instalación de software (por ejemplo, Pytorch); y también *Anaconda Navigator*, que es la interfaz gráfica de la distribución, la cual facilita su uso.

Juniper Notebooks [64] es una interfaz basada en web para programación interactiva, desarrollada por *Juniper Lab*; una organización sin ánimo de lucro que desarrolla software de código abierto. Su funcionamiento difiere del uso de scripts clásicos, en que la ejecución del código se puede separar en celdas; de modo que permite organizar un problema computacional en partes e ir avanzando a medida que los resultados de las partes anteriores son los deseados (sin tener que ejecutar todo el script completo). Se ha convertido en una interfaz de usuario popular, además permite la inserción de texto o imágenes entre los bloques de código. En *Anaconda Navigator* viene instalado por defecto.

También se ha usado el cliente de Python de InfluxDB para obtener los datos almacenados en la base de datos y escribir en ella después. Las acciones realizadas para crear y entrenar una red neuronal capaz de hacer predicciones sobre los datos, se describen a continuación.

6.3.1 Datos utilizados

En primer lugar, son necesarios datos para entrenar y probar la red neuronal. En este caso se han usado datos cedidos por parte del tutor del proyecto, por medio de un *backup* de la base de datos de origen y su posterior restauración en la base de datos del proyecto (ambas InfluxDB, para almacenarlos de la misma manera en la implementación). Estos datos proceden de sensores que recogen medidas de temperatura y humedad en dos localizaciones diferentes, una en un entorno cerrado y otra en el exterior.

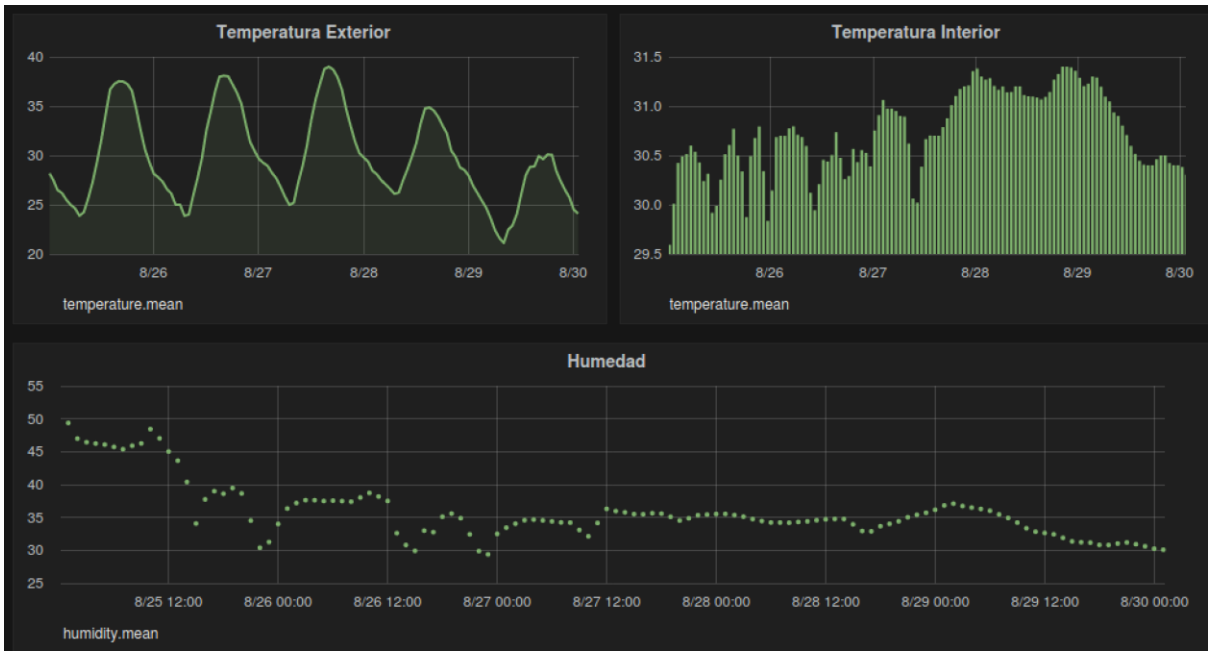


Figura 6.29: Ejemplo de panel de visualización de datos en Grafana.

En la figura 6.29 se pueden observar ejemplos de la variedad de gráficas que se pueden implementar en Grafana, las cuales se han hecho con los datos de temperatura y humedad del proyecto. Concretamente en dicha imagen se ha representado la temperatura exterior con un gráfico lineal, la temperatura interior con un histograma y por último la humedad con un gráfico de puntos.

6.3.2 Integración con Python

Una vez obtenidos los datos necesarios para el modelo; el siguiente paso es crear un cliente para la base de datos, de manera que seamos capaces de hacer solicitudes a la misma y extraerlos. Un ejemplo de *query* puede ser pedir todos los datos de temperatura medida en el interior, correspondientes al periodo entre el 15 de enero y el 15 de marzo de 2021, agrupados en intervalos de 1 hora y agregados obteniendo la media de cada intervalo:

```
SELECT MEAN(*) FROM temperature WHERE location=livingroom
AND time >= 2021-01-15T00:00:00.0Z AND time <= 2021-03-16T00:00:00.0Z
GROUP BY time(1h)
```

Una vez obtenidos los datos, los cargamos en un *Dataframe* de Pandas, y podemos añadir más columnas al *Dataframe* con el fin de ver si el modelo TFT es capaz de extraer más información (se han añadido los campos correspondientes al año, mes, día y hora, así como un índice).

	time	mean_value	Anio	Mes	Dia	Hora	time_idx
0	2021-01-15 00:00:00+00:00	12.600000	2021	01	15	00	0
1	2021-01-15 01:00:00+00:00	12.583193	2021	01	15	01	1
2	2021-01-15 02:00:00+00:00	12.572500	2021	01	15	02	2
3	2021-01-15 03:00:00+00:00	12.592500	2021	01	15	03	3
4	2021-01-15 04:00:00+00:00	12.525833	2021	01	15	04	4
...	...	...	...	...	...	...	...
1436	2021-03-15 20:00:00+00:00	17.918333	2021	03	15	20	1436
1437	2021-03-15 21:00:00+00:00	17.859167	2021	03	15	21	1437
1438	2021-03-15 22:00:00+00:00	17.676667	2021	03	15	22	1438
1439	2021-03-15 23:00:00+00:00	17.521667	2021	03	15	23	1439
1440	2021-03-16 00:00:00+00:00	NaN	2021	03	16	00	1440

Figura 6.30: Contenido del *Dataframe*.

6.3.3 Uso de Pytorch Forecasting

Una vez realizado esto comenzamos a usar Pytorch Forecasting, siguiendo los siguientes pasos:

1. Creamos un conjunto de datos de entrenamiento con *TimeSeriesDataSet* (pasándole un *Dataframe* de Pandas) e indicamos los campos de los datos.
2. Con el conjunto de datos de entrenamiento, creamos un conjunto de datos de validación con *from_dataset()*. También se puede crear un conjunto de datos de prueba o posteriormente un conjunto de datos para inferencia.
3. Creamos una instancia de un modelo usando su método *.from_dataset()*. Por ejemplo, vamos a usar *Temporal Fusion Transformer* (arquitectura explicada en la sección 2.4.3).
4. Creamos un objeto *pytorch_lightning.Trainer()*, y encontramos la tasa de aprendizaje óptima con su método *.tuner.lr_find()*.

```
   | Name                                | Type                              | Params
-------------------------------------------------------------------------------------------
0  | loss                                | QuantileLoss                      | 0
1  | logging_metrics                     | ModuleList                        | 0
2  | input_embeddings                    | MultiEmbedding                    | 217
3  | prescalers                          | ModuleDict                        | 640
4  | static_variable_selection           | VariableSelectionNetwork          | 51.7 K
5  | encoder_variable_selection          | VariableSelectionNetwork          | 34.6 K
6  | decoder_variable_selection          | VariableSelectionNetwork          | 192
7  | static_context_variable_selection   | GatedResidualNetwork              | 16.8 K
8  | static_context_initial_hidden_lstm  | GatedResidualNetwork              | 16.8 K
9  | static_context_initial_cell_lstm    | GatedResidualNetwork              | 16.8 K
10 | static_context_enrichment           | GatedResidualNetwork              | 16.8 K
11 | lstm_encoder                        | LSTM                              | 33.3 K
12 | lstm_decoder                        | LSTM                              | 33.3 K
13 | post_lstm_gate_encoder              | GatedLinearUnit                   | 8.3 K
14 | post_lstm_add_norm_encoder          | AddNorm                           | 128
15 | static_enrichment                   | GatedResidualNetwork              | 20.9 K
16 | multihead_attn                      | InterpretableMultiHeadAttention   | 10.4 K
17 | post_attn_gate_norm                 | GateAddNorm                       | 8.4 K
18 | pos_wise_ff                         | GatedResidualNetwork              | 16.8 K
19 | pre_output_gate_norm                | GateAddNorm                       | 8.4 K
20 | output_layer                        | Linear                            | 455
-------------------------------------------------------------------------------------------
294 K     Trainable params
0         Non-trainable params
294 K     Total params
Finding best initial lr: 100%|██████████| 100/100 [1:12:14<00:00, 33.86s/it]
```

Figura 6.31: Número de parámetros en el modelo.

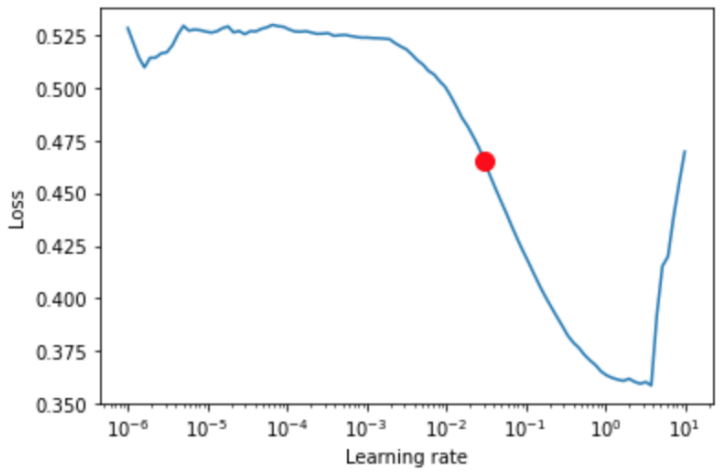


Figura 6.32: Gráfico con el *learning rate* óptimo.

5. Entrenamos el modelo usando *early stop* (parada temprana) en el conjunto de datos de entrenamiento [65]. Este método de *Machine learning* consiste en medir las pérdidas del modelo en todas las iteraciones, de manera que, si estas se estancan, se procede a la terminación del entrenamiento con el fin de evitar un sobreajuste del modelo. Tras esto usamos los registros de Tensorboard para comprobar si ha convergido con una precisión aceptable. Obtenemos la comparación de la predicción del mejor modelo obtenido y la realidad:

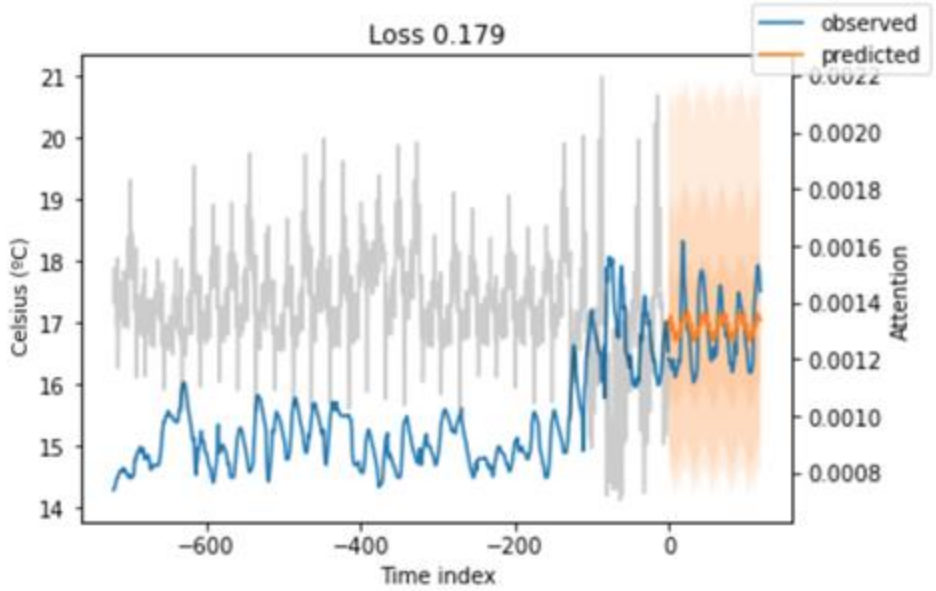


Figura 6.33: Predicción frente a los datos observados.

En la gráfica 6.33 del modelo TFT, se observa también la *Attention* (en gris), que representa la importancia o atención que ha prestado el modelo a los datos anteriores para obtener la predicción.

6. Por último, cargamos el modelo desde el punto de control y ya somos capaces de aplicarlo a los nuevos datos para hacer predicciones.

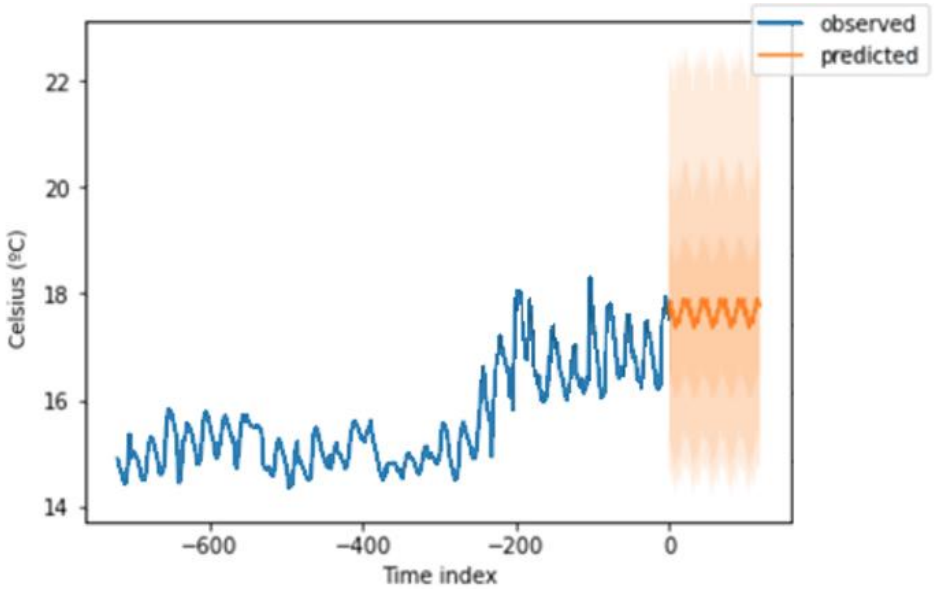


Figura 6.34: Nueva predicción.

En la sección 7.2 se recogen más resultados de predicciones obtenidas; para otros horizontes temporales y modelos.

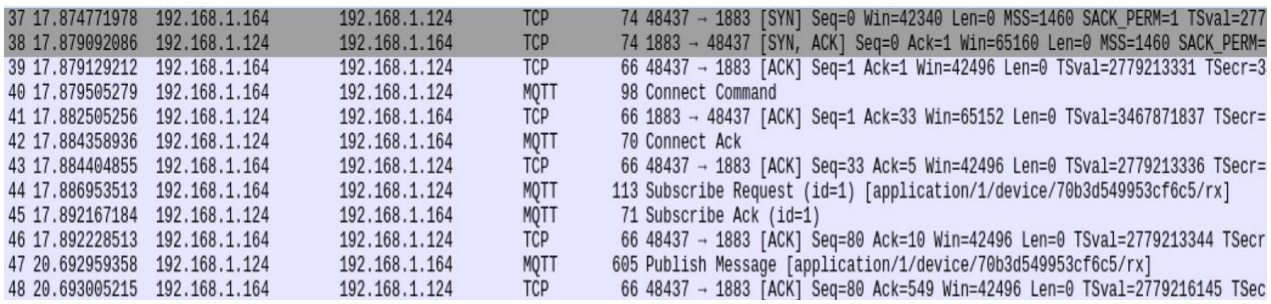
Con esto finaliza el capítulo dedicado a mostrar cómo se ha realizado la implementación de cada parte, la parte de la red IoT con la mota y la pasarela LoRaWAN; la red SDN con Mininet y el controlador Ryu; y la parte de *Machine Learning* con los modelos de Pytorch.

7. Resultados

En este capítulo se recogen algunos de los resultados obtenidos de la implementación, entre los que se incluyen la visualización del tráfico MQTT sobre SDN, los resultados de predicciones para nuevos datos y su visualización en gráficos.

7.1 Resultados de la red SDN.

Una vez, operativa la integración, podemos utilizar Wireshark para capturar los paquetes que viajan por la red SDN y comprobar que la comunicación se produce correctamente. A continuación, se muestran los resultados de realizar una captura de paquetes en una interfaz del *switch* S2, el cual se conecta por un lado al *switch* S1 con la red WiFi (red 192.168.1.0/24), y por otro con el *host* virtual.



37	17.874771978	192.168.1.164	192.168.1.124	TCP	74 48437 → 1883 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=277
38	17.879092086	192.168.1.124	192.168.1.164	TCP	74 1883 → 48437 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=
39	17.879129212	192.168.1.164	192.168.1.124	TCP	66 48437 → 1883 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=2779213331 TSecr=3
40	17.879505279	192.168.1.164	192.168.1.124	MQTT	98 Connect Command
41	17.882505256	192.168.1.124	192.168.1.164	TCP	66 1883 → 48437 [ACK] Seq=1 Ack=33 Win=65152 Len=0 TSval=3467871837 TSecr=
42	17.884358936	192.168.1.124	192.168.1.164	MQTT	70 Connect Ack
43	17.884404855	192.168.1.164	192.168.1.124	TCP	66 48437 → 1883 [ACK] Seq=33 Ack=5 Win=42496 Len=0 TSval=2779213336 TSecr=
44	17.886953513	192.168.1.164	192.168.1.124	MQTT	113 Subscribe Request (id=1) [application/1/device/70b3d549953cf6c5/rx]
45	17.892167184	192.168.1.124	192.168.1.164	MQTT	71 Subscribe Ack (id=1)
46	17.892228513	192.168.1.164	192.168.1.124	TCP	66 48437 → 1883 [ACK] Seq=80 Ack=10 Win=42496 Len=0 TSval=2779213344 TSecr
47	20.692959358	192.168.1.124	192.168.1.164	MQTT	605 Publish Message [application/1/device/70b3d549953cf6c5/rx]
48	20.693005215	192.168.1.164	192.168.1.124	TCP	66 48437 → 1883 [ACK] Seq=80 Ack=549 Win=42496 Len=0 TSval=2779216145 TSec

Figura 7.1: Captura de paquetes MQTT.

En la figura 7.1, podemos apreciar los paquetes correspondientes a la comunicación entre el *host* virtual, con IP 192.168.1.164; y la pasarela con IP 192.168.1.124. Los paquetes mostrados en dicha imagen corresponden al protocolo MQTT, en el cual el *host* hace de cliente suscriptor y la pasarela actúa como bróker.

Los primeros paquetes corresponden al establecimiento de una conexión TCP, al puerto 1883 (usado por MQTT). Tras esto se realiza la conexión MQTT, el cliente envía un mensaje *Connect* al bróker, el cual responde con un *Connect ACK*. Tras esto el cliente se suscribe al tópico *application/1/device/70b3d54993cf6c5/rx* con un mensaje *Subscribe Request*; y los últimos paquetes que se observan corresponden al envío de un mensaje por parte del bróker al suscriptor con el mensaje *Publish Message*.

También podemos capturar el tráfico en una interfaz del *switch* S3, entre el *host* virtual y la base de datos (red 192.168.56.0/24). En esta red se envían los datos a escribir en la base de datos InfluxDB, usando la API HTTP.

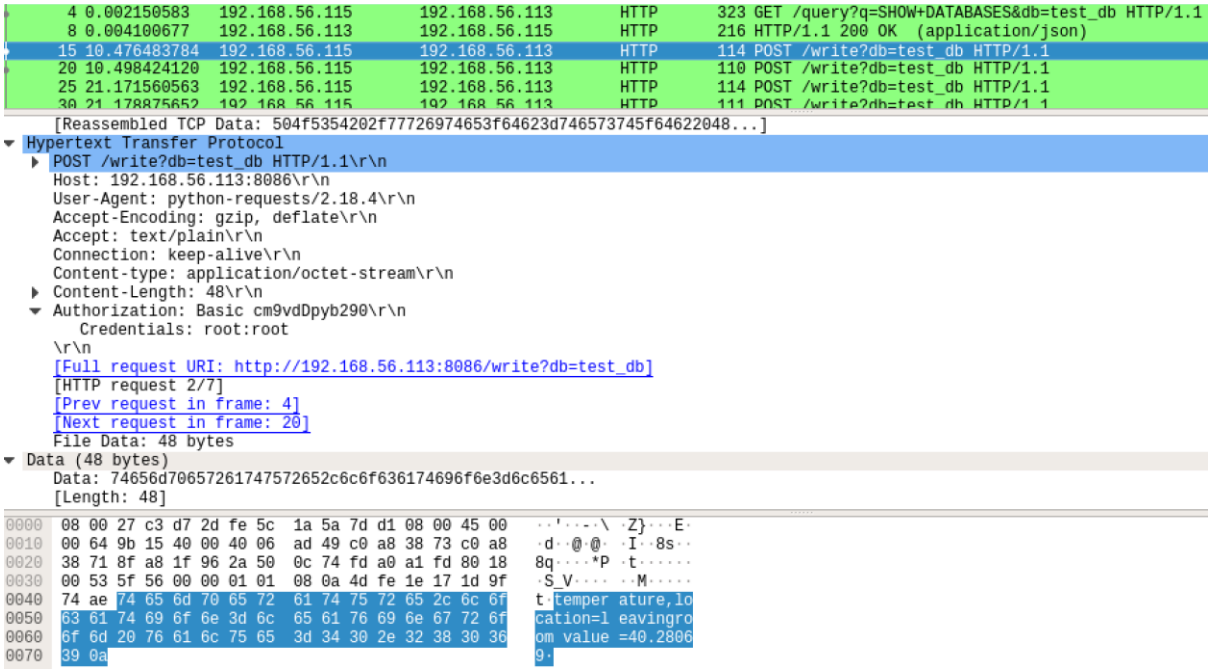


Figura 7.2: Captura de paquetes para escribir en InfluxDB a través de su API HTTP.

En la figura 7.2 podemos observar la comunicación entre el *host* virtual, con IP 192.168.56.115; y la máquina donde se encuentra corriendo InfluxDB, con IP 192.168.56.113. Los mensajes capturados corresponden a la escritura de datos en la base de datos *test_db* ejecutada por el cliente InfluxDB para Python en el script ejecutado en el *host* virtual (descrito en el Anexo A.3).

Los primeros mensajes que se observan corresponden a la conexión del cliente a la base de datos determinada, la cual se realiza por una petición GET, a la que la base de datos responde. Tras esto los mensajes capturados corresponden a peticiones POST en las cuales el cliente incluye los datos a escribir en la base de datos. En el campo de datos podemos observar dichos datos a escribir.

Con estas capturas de Wireshark, quedaría demostrado el correcto funcionamiento de la implementación, ya que se aprecia cómo los paquetes recorren los elementos de la red SDN y se comunican con el exterior.

7.2 Resultados de los algoritmos de Machine Learning

En esta subsección se recogen diferentes resultados obtenidos con los modelos implementados en Pytorch. Se han implementado dos modelos de aprendizaje automático, para comparar sus resultados; el modelo N-BEATs y el modelo *Temporal Fusion Transformer*, aplicados al problema de predecir temperatura sobre varios horizontes de predicción.

Ambos modelos han hecho uso de los datos de temperatura en el interior, y se han implementados con una configuración típica de los parámetros ajustables para cada uno de ellos. Esto es así, ya que no se pretende optimizar los modelos para el problema dado ni

estudiar su rendimiento; solamente recogerlos como un ejemplo sencillo de caso de uso. Los valores de los parámetros utilizados en la configuración de los modelos se recogen a continuación.

N-BEATs se ha usado con 1 bloque por pila (*stack*) con 4 capas de neuronas *fully connected*, con una anchura de 256 neuronas para predecir la tendencia (*trend*) y de 2048 neuronas para predecir la estacionalidad (*seasonality*). Como N-BEATs es un modelo que no admite covariables, sólo se ha tenido en cuenta la temperatura media como objetivo

TFT se ha usado con un tamaño de la red de 64, una capa de celdas LSTM, 4 *attention heads* en la capa *Temporal Self-Attention* y una salida de 7 quantiles para las predicciones. En este caso, al ser un modelo que admite covariables, se ha hecho uso también de los datos de humedad para obtener las predicciones de temperatura.

Para una predicción en la cual los valores de los datos se agrupan como la media por horas, para predecir el valor de la temperatura en los próximos cinco días, en función de la temperatura en los 30 días anteriores: el resultado con N-BEATS se muestra en la figura 7.3, y el de TFT en la figura 7.4.

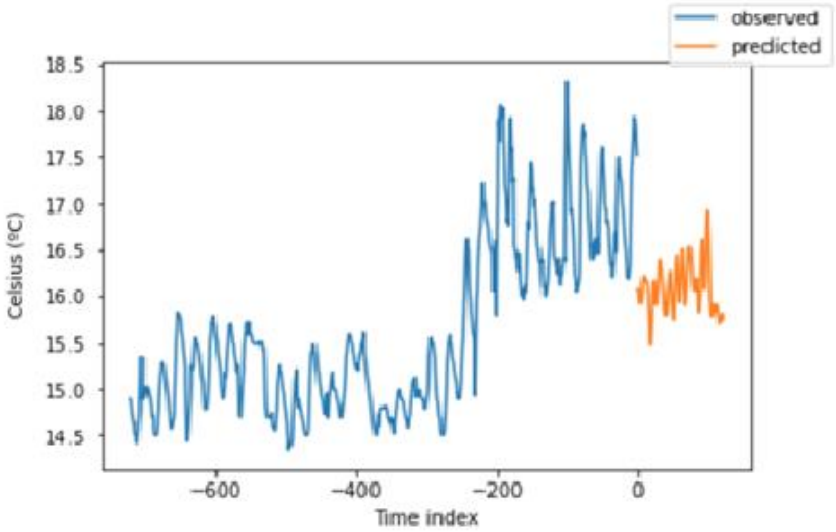


Figura 7.3: Predicción de 5 días N-BEATs.

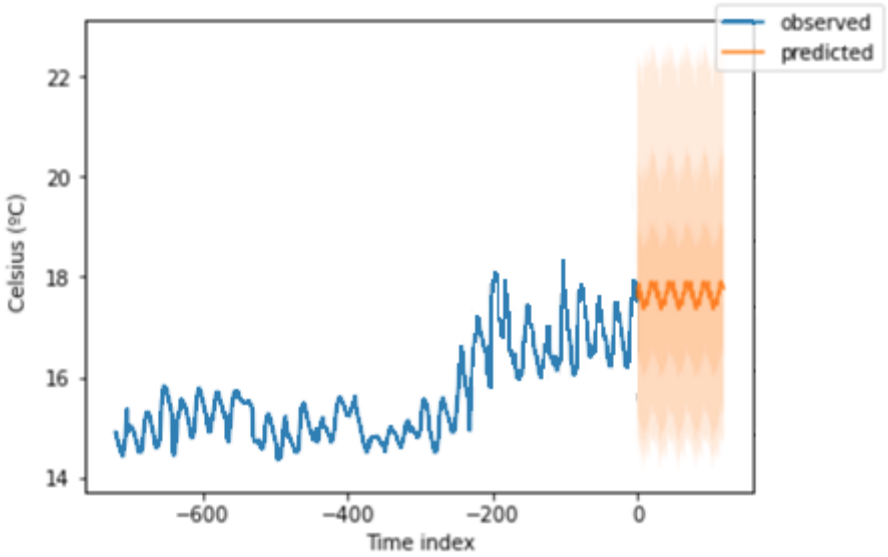


Figura 7.4: Predicción de 5 días TFT.

Si guardamos los valores obtenidos de las predicciones en la base de datos, podemos representarlos en Grafana y visualizar las predicciones conjuntamente como se muestra en la figura 7.5. En dicha imagen se ha representado el horizonte de predicción, 5 días haciendo uso de los 30 días anteriores.

Figura 7.5: Representación de predicciones de 5 días en Grafana.

Para una predicción en la cual los valores de los datos se agrupan como la media cada 20 minutos, para predecir el valor de la temperatura en el próximo día, en función de la temperatura en los 7 días anteriores: los resultados de N-BEATs se representan en la figura 7.6, y los de TFT en la figura 7.7.

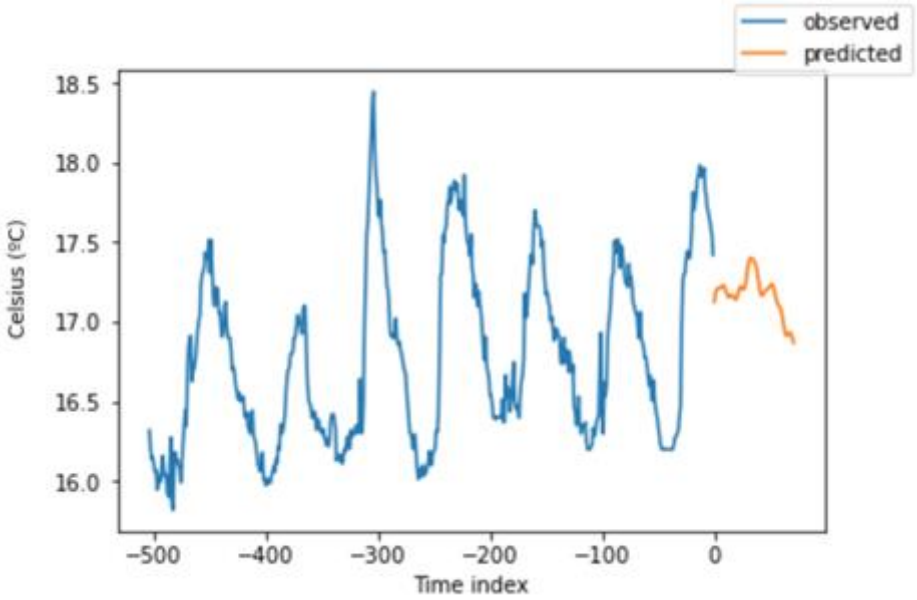


Figura 7.6: Predicción de 1 día con N-BEATs.

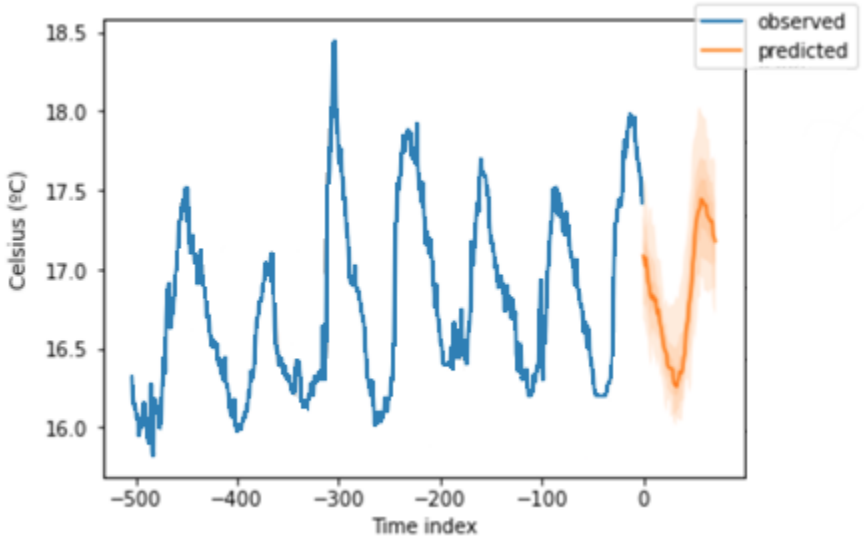


Figura 7.7: Predicción de 1 día con TFT.

La figura 7.8 muestra ambas predicciones junto con los datos reales sobre un gráfico de Grafana.

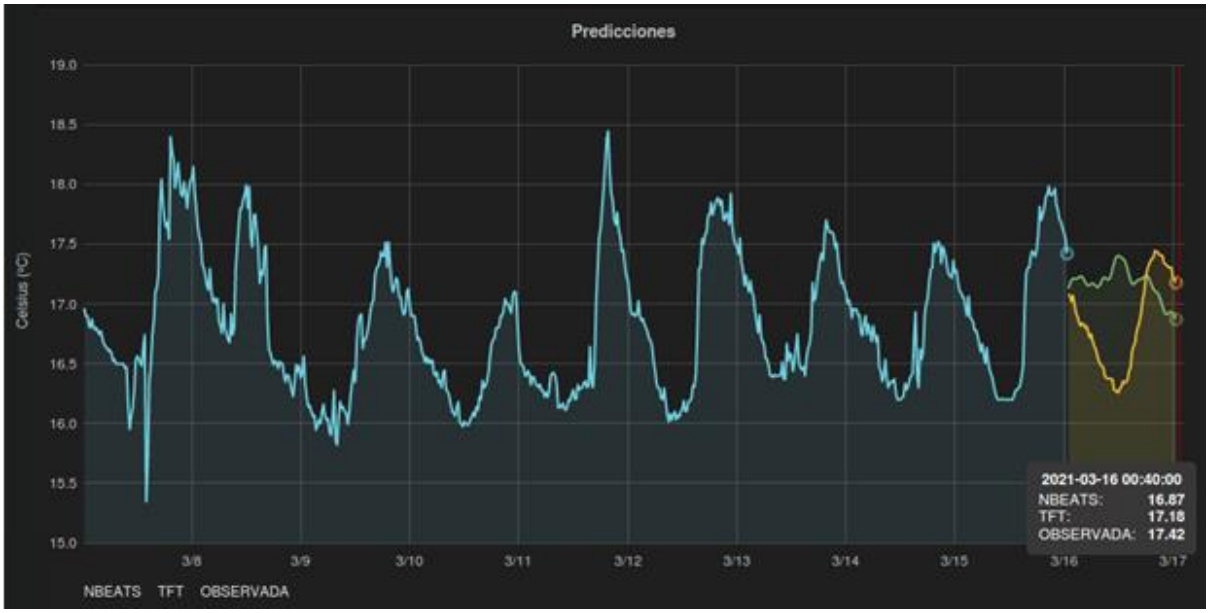


Figura 7.8: Representación de predicciones de 1 día en Grafana.

Para una predicción que agrupa los valores como la media cada 5 minutos, para predecir el valor de la temperatura en las próxima 12 horas, en función de la temperatura las 72 horas anteriores: la figura 7.9 recoge la predicción de N-BEATs, mientras que la figura 7.10 muestra lo obtenido con TFT.

Si observamos ambas predicciones junto con los valores reales medidos en un gráfico de Grafana, obtenemos lo representado en la figura 7.11.

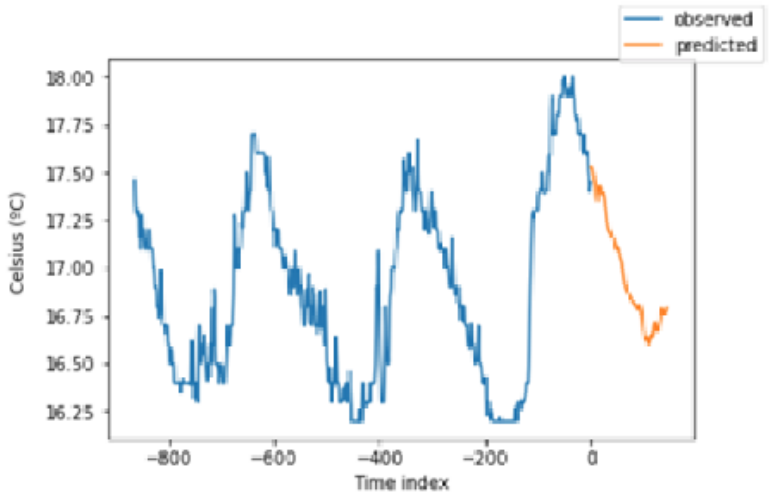


Figura 7.9: Predicción de 12 horas en N-BEATs.

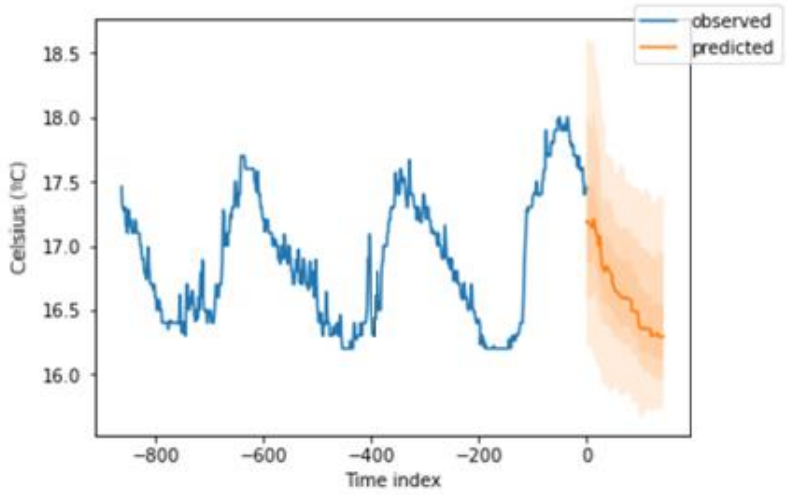


Figura 7.10: Predicción de 12 horas en TFT.

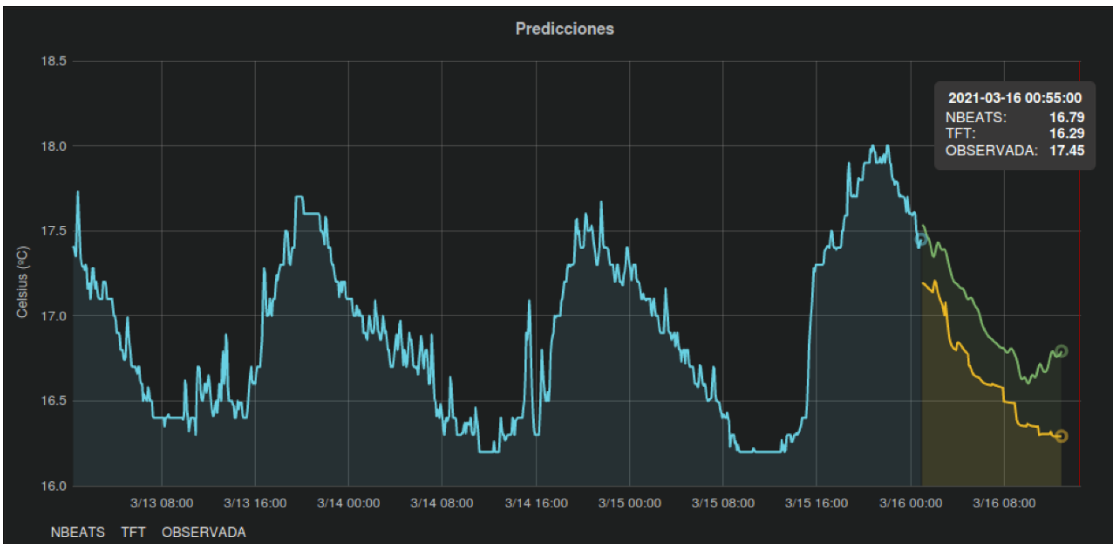


Figura 7.11: Representación de predicciones de 12 horas en Grafana.

Aparte de los resultados gráficos de las predicciones, podemos comparar las pérdidas obtenidas para cada modelo, para ver que tan bien funcionan. Estas pérdidas se han obtenido con los datos de entrenamiento, es decir realizando predicciones sobre datos conocidos y calculando la diferencia entre los valores reales medidos y los valores pronosticados y se recogen en la tabla 7.1. Mencionar que este trabajo no tiene como objetivo realizar una comparación exhaustiva de los modelos, y que esta comparación de pérdidas no tiene relevancia estadística, por lo que se incluye simplemente como ejemplo y para dar una idea de cual de los modelos usados ha funcionado mejor en este proyecto en concreto.

Horizonte de predicción			**N-BEATS**	**TFT**
Predicción de	**Basándose en**	**Muestras cada**		
5 días	30 días anteriores	60 minutos	3.75	0.179
1 día	7 días anteriores	30 minutos	4.06	0.063
12 horas	72 horas anteriores	5 minutos	8.38	0.078

Tabla 7.1: Pérdidas de los modelos.

Observando las pérdidas recogidas en la tabla 7.1, podemos observar que el modelo TFT proporciona pérdidas menores; esto se debe a que las predicciones se calculan usando cuantiles y las pérdidas también, por lo que es obvio este resultado. Respecto al modelo N-BEATs, se observa que al tratarse de un modelo que hace uso de tendencia y estacionalidad de los datos que utiliza como entrenamiento, funcionará mejor cuando el horizonte de predicción presente valores más claros para ambos parámetros. Es por eso que la eficacia del modelo mejora en un horizonte de predicción de 30 días (en el cual se aprecia claramente el cambio de temperatura cada día) a un horizonte de predicción que utiliza 72 horas.

En términos de tiempo empleado por cada modelo (en cada iteración); TFT, al ser un método que admite y aprovecha covariables, es un método más costoso de entrenar (necesita más tiempo); mientras que N-BEATs solamente admite la variable objetivo y es más sencillo de entrenar.

Volver a mencionar que estos resultados obtenidos carecen de relevancia estadística, ya que se ha usado un set de datos, que, aunque son medidas reales, es relativamente pequeño para poder analizar a fondo la eficacia exacta de los modelos.

Con esto finaliza este capítulo, en el que hemos mostrado algunos de los resultados obtenidos en la realización de la integración.

8. Conclusiones y líneas futuras

En este último capítulo de la memoria, se recogen en primer lugar las conclusiones derivadas de la realización de este proyecto y tras esto se exponen posibles mejoras de la implementación para trabajos futuros.

8.1 Conclusiones.

Con este proyecto, hemos presentado una integración práctica de tres tecnologías en auge como son IoT, las redes SDN y los algoritmos de *Machine Learning*. El crecimiento que está experimentando cada una de ellas, unido a su uso combinado como se propone en este trabajo, abre la puerta a una gran cantidad de nuevas aplicaciones.

Se han alcanzado todos los objetivos propuestos al comienzo del proyecto, en parte gracias a la definición de objetivos específicos englobados en el objetivo principal. Se ha conseguido que la mota capture datos usando sensores y los envíe a la pasarela de la red LoRaWAN. Se ha comprobado el correcto funcionamiento de la red SDN, con los *switches* OVS y el controlador Ryu, y la transmisión por ella de paquetes MQTT, así como su conexión a interfaces de red reales. Por último, se han implementado algoritmos de aprendizaje automático que hacen uso de dichos datos para predecir datos futuros.

Se ha presentado, por tanto, una manera funcional de integrar IoT, SDN y algoritmos de *Machine Learning*.

8.2 Líneas Futuras.

En esta sección se incluyen algunas indicaciones para posibles mejoras de la implementación:

Los algoritmos de aprendizaje automático, como los usados en este proyecto, se basan en datos para aprender de ellos. Partiendo de esto, si aumentamos el número de datos accesibles a dichos algoritmos, sus resultados podrían mejorar. Posibles líneas de mejora pasan por usar una mota LoRa con sensores de mejor calidad, o diferentes motas con sensores para magnitudes diferentes (por ejemplo y para el caso que hemos tratado, sensores anemómetros o sensores pluviómetros). En este caso también sería interesante recopilar información procedente de muchas motas, en lugares diferentes para ver cómo puede influir la localización.

Trabajos futuros pueden tratar de añadir e incluir más modelos de predicción diferentes para la tarea que nos ocupa, o extrapolar el aquí implementado a otras aplicaciones.

Otra mejora al presente proyecto, pasa por el uso de un entrenamiento continuo (denominado *online learning*) en el modelo de predicción usado. En lugar de entrenar el modelo cada cierto tiempo para actualizarlo, sería interesante que se fuera actualizando con cada entrada nueva que se recibe por parte de los sensores.

En cuanto a la red SDN, se podrían utilizar diferentes arquitecturas y topologías, así como diferentes configuraciones de los *switches* OpenFlow por parte del controlador. Por ejemplo, se podría implementar un flujo que modificara los paquetes para conseguir un reenvío de paquetes de una red a otra, caso en el cual el *host* virtual no sería necesario.

Bibliografía

1. D. Evans, "The internet of things: How the next evolution of the internet is changing everything", CISCO White Paper, 2011.
2. Cisco Annual Internet Report (2018–2023) White Paper
3. M. A. Razzaque, M. Milojevic-Jevric, A. Palade and S. Clarke, "Middleware for Internet of Things: A Survey,", *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70-95, Feb. 2016.
4. R. Sanchez-Iborra, M. Cano, "State of the Art in LP-WAN Solutions for Industrial IoT Services", *Sensors*, vol. 16, no. 5, p. 708, May 2016.
5. "A Technical Overview of LoRa and LoRaWAN", LoRa-Alliance: San Ramon, CA, USA, Nov. 2015. Disponible en: https://lora-alliance.org/wp-content/uploads/2020/11/what-is-lorawan.pdf (última visita el 30/06/21)
6. P. Goransson and C. Black, "Software Defined Networks: A Comprehensive Approach", Elsevier and Morgan Kauffman, 2014
7. H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey", *Comput. Networks* vol. 81, pp. 79-95, 2015.
8. Francisco Javier García Castellano, "Inteligencia Artificial para Telecomunicaciones, Diapositivas Tema 4", MUIT, UGR, 2020.
9. N. Sornin, M. Luis, T. Eirich, T. Kramp, and O.Hersent, "LoRaWAN Specification v1.0.2", LoRa Alliance, Standard Specification, Jul. 2016. Disponible en https://lora-alliance.org/wp-content/uploads/2020/11/lorawan1_0_2-20161012_1398_1.pdf (última visita el 30/06/21)
10. The HiveMQ Team, "MQTT Essentials". Disponible en https://www.hivemq.com/mqtt-essentials/ (última visita el 30/06/21)
11. Oasis, "MQTT v. 5 Specification". Disponible en https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf (última visita el 30/06/21)
12. A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications", *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347-2376, Fourthquarter 2015.
13. Open Networking Foundation, "OpenFlow Switch Specification v1.5.0", Mar. 2015, Disponible en https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (última visita el 30/06/21)
14. W. Zaremba, I. Sutskever and O. Vinyals, "Recurrent Neural Network Regularization", arXiv:1409.2329, Sep. 2014.
15. Michael Phi, "Illustrated Guide to LSTM's and GRU's: A step by step explanation". Disponible en: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 (última visita el 30/06/21)

16. B.N. Oreshkin, D. Carpov, N. Chapados and Y. Bengio, “N-BEATS: Neural basis expansion analysis for interpretable time series forecasting”, arXiv:1905.10437, 2019.
17. B. Lima, S.Ö. Arıkb, N. Loeffb and T. Pfisterb, “Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting”, arXiv:1912.09363, 2020.
18. Raimi Karim, “Illustrated: Self-Attention”. Disponible en: https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a (última visita el 30/06/21)
19. “ChirpStack open-source LoRaWAN® Network Server”. Disponible en: https://www.chirpstack.io (última visita el 30/06/21)
20. “Documentation Eclipse Mosquitto”. Disponible en: https://mosquitto.org/documentation/ (última visita el 30/06/21)
21. “Eclipse wiki for the Paho project”. Disponible en: https://wiki.eclipse.org/Paho (última visita el 30/06/21)
22. “VirtualBox User Manual”. Disponible en: https://www.virtualbox.org/manual/ch01.html (última visita el 30/06/21)
23. B. Pfaff, J. Pettit, T. Koponen, E.J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon and M. Casado, “The Design and Implementation of Open vSwitch”, *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation.* USENIX Association, pp. 117–130, 2015
24. “RYU SDN Framework, Release 1.0”, Disponible en: https://book.ryu-sdn.org/en/Ryubook.pdf (última visita el 30/06/21)
25. Sridhar Rao, “Ryu, a Rich-Featured Open Source SDN Controller Supported by NTT Labs”. Disponible en: https://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/ (última visita el 30/06/21)
26. B. Lantz, N. Handigol, B. Heller, and V. Jeyakumar, “Introduction to Mininet”, Disponible en: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet (última visita el 30/06/21)
27. “Mininet Python API Reference Manual”. Disponible en http://mininet.org/api/hierarchy.html (última visita el 30/06/21)
28. “InfluxDB 1.7 Documentation”. Disponible en: https://docs.influxdata.com/influxdb/v1.7/ (última visita el 30/06/21)
29. “Grafana Documentation”. Disponible en: https://grafana.com/docs/grafana/latest/ (última visita el 30/06/21)
30. “Pytorch Documentation”. Disponible en: https://pytorch.org/docs/stable/index.html (última visita el 30/06/21)
31. “Pytorch Forecasting Documentation”. Disponible en: https://pytorch-forecasting.readthedocs.io/en/latest/index.html (última visita el 30/06/21)

32. "Pandas Documentation". Disponible en: https://pandas.pydata.org/docs/index.html (última visita el 1/08/21)
33. "TensorFlow's visualization toolkit". Disponible en: https://www.tensorflow.org/tensorboard (última visita el 1/08/21)
34. "Visualizing models, data, and training with Tensorboard". Disponible en: https://pytorch.org/tutorials/intermediate/tensorboard_tutorial.html (última visita el 1/08/21)
35. "Wireshark User's Guide Version 3.5.0". Disponible en: https://www.wireshark.org/docs/wsug_html/ (última visita el 30/06/21)
36. M.S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi and A. P. Sheth, "Machine Learning for Internet of Things Data Analysis: A Survey", Digital Communications and Networks vol. 4, no. 3, pp. 161- 175, 2018.
37. E. Adi, A. Anwar, Z. Baig and S. Zeadally "Machine Learning and Data Analytics for the IoT", *Neural Comput. Appl*., vol. 32, no. 20, pp. 16205–16233, 2020.
38. M.K. Nallakaruppan and U. Senthil Kumaran, "IoT based Machine Learning Techniques for Climate Predictive Analysis", *International Journal of Recent Technology and Engineering,* vol. 7, pp. 171-175, 2019.
39. A. Vamseekrishna, R. Nishitha, T. Anil Kumar, K. Hanuman, and Ch. G. Supriya, "Prediction of Temperature and Humidity Using IoT and Machine Learning Algorithm", *International Conference on Intelligent and Smart Computing in Data Analytics. Advances in Intelligent Systems and Computing,* vol. 1312, 2021.
40. G. Verma, P. Mittal and S. Farheen, "Real Time Weather Prediction System Using IOT and Machine Learning", *2020 6th International Conference on Signal Processing and Communication (ICSC),* pp. 322-324, 2020.
41. John Gamboa "Deep Learning for Time-Series Analysis", arXiv:1701.01887, 2017.
42. Mohsen Hayati, and Zahra Mohebi, "Application of Artificial Neural Networks for Temperature Forecasting", *World Academy of Science, Engineering and Technology*, vol. 28, pp. 275-279, 2007
43. P. Hewage, M. Trovati, E. Pereira and A. Behera, "Deep learning-based effective fine-grained weather forecasting model", *Pattern Analysis and Applications*, pp. 1–24, 2020.
44. B.A.O. Ikram, B.A. Abdelhakim, A. Abdelali, B. Zafar, and B. Mohammed, "Deep Learning architecture for temperature forecasting in an IoT LoRa based system*", Proceedings of the 2nd International Conference on Networking, Information Systems & Security*, pp. 1-6, 2019.
45. "¿Qué es AWS?" Disponible en: https://aws.amazon.com/es/what-is-aws/?nc1=f_cc (última visita el 01/09/21)
46. "AWS IoT SDK de dispositivos" Disponible en: https://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-sdks.html (última visita el 01/09/21)

47. “FreeRTOS: Sistema operativo con funcionamiento en tiempo real para microcontroladores” Disponible en: https://aws.amazon.com/es/freertos/ (última visita el 01/09/21)
48. “AWS IoT Greengrass” Disponible en: https://aws.amazon.com/es/greengrass/?c=i&sec=srv (última visita el 01/09/21)
49. “AWS IoT Core” Disponible en: https://aws.amazon.com/es/iot-core/?c=i&sec=srv (última visita el 01/09/21)
50. “AWS IoT Core for LoRaWAN” Disponible en: https://aws.amazon.com/es/iot-core/lorawan/ (última visita el 01/09/21)
51. “AWS IoT Device Management” Disponible en: https://aws.amazon.com/es/iot-device-management/?c=i&sec=srv (última visita el 01/09/21)
52. “AWS IoT Device Defender” Disponible en: https://aws.amazon.com/es/iot-device-defender/?c=i&sec=srv (última visita el 01/09/21)
53. “AWS IoT Analytics” Disponible en: https://aws.amazon.com/es/iot-analytics/?nc=sn&loc=2&dn=6 (última visita el 01/09/21)
54. “AWS IoT Things Graph” Disponible en: https://aws.amazon.com/es/iot-things-graph/?nc=sn&loc=2&d=7 (última visita el 01/09/21)
55. “AWS IoT Forecast” Disponible en: https://aws.amazon.com/es/forecast/ (última visita el 01/09/21)
56. “AWS IoT SageMaker” Disponible en: https://aws.amazon.com/es/sagemaker/ (última visita el 01/09/21)
57. “Deep Learning with Pytorch on AWS” Disponible en: https://aws.amazon.com/es/pytorch/ (última visita el 01/09/21)
58. “Reference setup for iC880a gateways running The Things Network” Disponible en: https://github.com/ttn-zh/ic880a-gateway (última visita el 30/06/21)
59. “The Things Network Website” Disponible en: https://www.thethingsnetwork.org/ (última visita el 30/06/21)
60. “ChirpStack Quickstart Debian or Ubuntu” Disponible en https://www.chirpstack.io/project/guides/debian-ubuntu/ (última visita el 30/06/21)
61. “Grafana Documentation Data Sources InfluxDB data source”. Disponible en: https://grafana.com/docs/grafana/latest/datasources/influxdb/ (última visita el 30/06/21)
62. “Anaconda Documentation” Disponible en: https://docs.anaconda.com/anaconda/user-guide/getting-started/ (última visita el 1/08/21)
63. “Conda Documentation” Disponible en: https://conda.io/en/latest/ (última visita el 1/08/21)
64. “Juniper Notebook Documentation” Disponible en: https://jupyter-notebook.readthedocs.io/en/latest/notebook.html (última visita el 1/08/21)

65. “Early stopping” en Wikipedia Disponible en: https://en.wikipedia.org/wiki/Early_stopping (última visita el 1/08/21)
66. Gautier MECHLING, “Home sensor data monitoring with MQTT, InfluxDB and Grafana”. Disponible en https://github.com/Nilhcem/home-monitoring-grafana (última visita el 30/06/21)

Anexo A: Código

En este anexo se incluyen algunos de los códigos usados en la implementación. Se incluye el código `simple_switch13.py` del controlador SDN Ryu, así como una explicación del mismo; el código correspondiente a la topología de Mininet; el código que hace de puente entre los mensajes MQTT y la base de datos InfluxDB y el código usado para implementar los algoritmos de aprendizaje automático con Pytorch.

Todos los scripts están escritos en Python.

A.1 Script `simple_switch13`

En esta sección se recoge y explica el código de la aplicación utilizada en el controlador Ryu. Este código hace que los *switches* OpenFlow actúen como *learning switches*, es decir, como conmutadores de capa 2.

Un *learning switch* almacena una base de datos con los *hosts* a los que está conectado, y sus puertos. Los *hosts* se identifican por la dirección MAC de su tarjeta de red y los puertos se identifican simplemente por su número.

Si un *switch* recibe un paquete en cualquiera de sus puertos, buscará la dirección MAC de destino de ese paquete en su base de datos para ver si sabe a qué puerto está conectado ese *host*. Si lo descubre, reenvía ese paquete únicamente a ese puerto específico; pero si aún no tiene una entrada en su base de datos para esa dirección, inunda ese paquete por todos los puertos menos por el de entrada, y los *hosts* pueden verificar por sí mismos si el paquete estaba destinado a ellos.

Al mismo tiempo, el *switch* almacena en la base de datos la dirección MAC de origen de ese paquete y la asigna al puerto por el que recibió el paquete. De esta manera ha aprendido y no será necesario inundar en la posible respuesta al mensaje.

En el caso de las redes SDN, los equipos de la red, es decir, los *switches* OpenFlow, son muy sencillos y se encargan solamente del plano de datos (encargado del reenvío de paquetes) basado en tablas de flujos configuradas por el controlador. Para que un *switch* OpenFlow actúe como un *switch* de capa 2, el esquema de operación es el siguiente:

El *switch* se registra en el controlador. El controlador añade en el *switch* una regla para que cada vez que el *switch* reciba un paquete que no pertenezca a ningún flujo guardado le envíe el paquete al controlador. Además, el controlador genera y gestiona una tabla de MAC a puertos específica para el switch. Hecho esto, el *switch* ya estaría registrado.

Una vez registrado, cuando el *switch* recibe un paquete, comprueba en su tabla de flujos si coincide. En caso de que no coincida con ninguna entrada, se ejecutará la regla por defecto y enviará el paquete al controlador (por medio de un mensaje *PacketIN*). En el controlador se guarda la MAC de origen en la tabla específica de ese *switch* y se asocia el puerto por el que entró. Se comprueba si se tiene la MAC de destino en la tabla y se envía un *PacketOUT* al

switch indicando que inunde si no se ha encontrado dicha MAC, o el puerto en el que se encuentra conectada.

Además, el controlador agrega un flujo en el *switch* (con una prioridad más alta que el flujo por defecto) que coincida con la dirección MAC de destino, la MAC de origen y con el puerto de entrada actual de un paquete y le indique al *switch* que envíe el paquete por el puerto asociado con esa dirección de destino.

Una vez explicado el funcionamiento de un *learning switch* y cómo se puede realizar en redes SDN, a continuación, se incluye el código correspondiente a `simple_switch13.py`, que es una implementación de este tipo de *switches* en el controlador Ryu.

```
1.  # Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
2.  #
3.  # Licensed under the Apache License, Version 2.0 (the "License");
4.  # you may not use this file except in compliance with the License.
5.  # You may obtain a copy of the License at
6.  #
7.  #    http://www.apache.org/licenses/LICENSE-2.0
8.  #
9.  # Unless required by applicable law or agreed to in writing, softwr
10. # distributed under the License is distributed on an "AS IS" BASIS,
11. # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
12. # implied. See the License for the specific language governing
13. # permissions and limitations under the License.
14.
15. from ryu.base import app_manager
16. from ryu.controller import ofp_event
17. from ryu.controller.handler import CONFIG_DISPATCHER,
    MAIN_DISPATCHER
18. from ryu.controller.handler import set_ev_cls
19. from ryu.ofproto import ofproto_v1_3
20. from ryu.lib.packet import packet
21. from ryu.lib.packet import ethernet
22. from ryu.lib.packet import ether_types
23.
24.
25. class SimpleSwitch13(app_manager.RyuApp):
26.     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
27.
28.     def __init__(self, *args, **kwargs):
29.         super(SimpleSwitch13, self).__init__(*args, **kwargs)
30.         # initialize mac address table.
31.         self.mac_to_port = {}
32.
```

El script comienza importando los recursos necesarios, tras esto el resto del código consiste en la definición de la clase *SimpleSwitch13*. Como argumento de la clase, utiliza *ryu.base.app_manager.RyuApp* (importado en la primera línea). Según el manual de la API

de Ryu, la clase *app_manager* incluye la administración central de las aplicaciones de Ryu, carga aplicaciones Ryu, les proporciona contextos y enruta mensajes entre aplicaciones Ryu. Además, especifica con qué versiones del protocolo OpenFlow es compatible la aplicación, en este caso la 1.3.

Lo siguiente que hace es definir el constructor de la clase en el cual se inicializa la tabla interna de MAC a puerto.

A lo largo de este código, se hace uso del decorador *set_ev_cls*, para llamar a los diferentes métodos cada vez que ocurre un evento determinado. Este decorador utiliza dos argumentos; el primero indica un evento que hace que la función sea llamada, y el segundo indica el estado del *switch* cuando desea permitir que Ryu maneje un evento.

```
34.  @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
35.      def switch_features_handler(self, ev):
36.          datapath = ev.msg.datapath
37.          ofproto = datapath.ofproto
38.          parser = datapath.ofproto_parser
39.
40.          # install table-miss flow entry
41.          #
42.          # We specify NO BUFFER to max_len of the output action due
43.          # to OVS bug. At this moment, if we specify a lesser
44.          # number, e.g.,128, OVS will send Packet-In with invalid
45.          # buffer_id and truncated packet data. In that case, we
46.          # cannot output packets correctly.  The bug has been fixed
47.          # in OVS v2.1.0.
48.          match = parser.OFPMatch()
49.          actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
50.                                            ofproto.OFPCML_NO_BUFFER)]
51.          self.add_flow(datapath, 0, match, actions)
52.
```

Se define el método *switch_features_handler*, que se ejecuta cada vez que se agrega un *switch* al controlador y le instala la entrada de flujo por defecto en el switch, lo que permite que envíe paquetes al controlador. Para ello se utiliza el decorador que se activa cuando se produce un evento ofp_event.EventOFPSwitchFeatures durante la fase CONFIG_DISPATCHER. En esta fase de negociación, el controlador Ryu pide al *switch* sus características para configurar la conexión.

A partir del, mensaje recibido (*ev.msg*); define la variable *datapath* para referirse al switch, *ofproto* para referirse a la biblioteca con las definiciones propias de la la versión de OpenFlow usada (va a ser 1.3 ya que esta clase solo soporta esta versión, según se ha definido anteriormente), y *parser* para referenciar a la biblioteca encargada de parsear los mensajes de esta versión del protocolo.

Crea la regla de coincidencia match usando *parser.OFPMatch()* vacío, sin ningún argumento; de este modo coinciden cualquier flujo. Define actions como una lista que contiene solamente *parser.OFPActionOutput()*, que se usa para indicar el puerto de salida para un flujo; al cual pasa los argumentos *ofproto.OFPP_CONTROLLER* y

ofproto.OFPCML_NO_BUFFER. El primero indica que el flujo se debe enviar al controlador, y el segundo le indica al *switch* que no almacene paquetes de este flujo en el buffer (por un bug de OVS que se soluciona en versiones posteriores).

Por último, este método acaba llamando a *add_flow*; método que se va a explicar a continuación.

```
53.     def add_flow(self, datapath, priority, match, actions,
    buffer_id=None):
54.         ofproto = datapath.ofproto
55.         parser = datapath.ofproto_parser
56.
57.         # construct flow_mod message and send it.
58.         inst =
    [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
59.         if buffer_id:
60.             mod = parser.OFPFlowMod(datapath=datapath,
    buffer_id=buffer_id, priority=priority, match=match,
    instructions=inst)
61.         else:
62.             mod = parser.OFPFlowMod(datapath=datapath,
    priority=priority, match=match, instructions=inst)
63.         datapath.send_msg(mod)
64.
```

Se define el método *add_flow*, para facilitar la creación de nuevas entradas en la tabla de flujos. Tiene como parámetros *datapath* (*switch* en el que hay que crear el flujo), *priority* (con la prioridad para dicho flujo), *match* (para indicar las reglas de coincidencia del flujo) y *actions* (con las acciones a realizar).

Las primeras líneas de este método, inicializan referencias a las librerías del protocolo OpenFlow de manera similar al método *switch_features_handler*, tras esto crea la variable *inst* (de instrucciones) como una lista con un solo elemento *parser.OFPInstructionActions()*; el cual recibe como argumentos *ofproto.OFPIT_APPLY_ACTIONS* y *actions* (argumento del método).

Una vez hecho esto, se crea el mensaje *mod* (de modificación) usando *parser.OFPFlowMod()* e indicando como argumentos el *switch* a modificar, la prioridad, las reglas de coincidencia y las instrucciones que se acaban de crear.

Por último, se envía con el método *send_msg()*; llamado desde *datapath* (instancia del *switch* al que va destinado).

El código *simple_switch13* finaliza con la definición de un último método *packet_in_handler*, que se llama cuando Ryu recibe un mensaje OpenFlow *packet_in*.

```
65. @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
66.     def _packet_in_handler(self, ev):
```

```
67.         # If you hit this you might want to increase
68.         # the "miss_send_length" of your switch
69.         if ev.msg.msg_len < ev.msg.total_len:
70.             self.logger.debug("packet truncated: only %s of %s
    bytes", ev.msg.msg_len, ev.msg.total_len)
71.         msg = ev.msg
72.         datapath = msg.datapath
73.         ofproto = datapath.ofproto
74.         parser = datapath.ofproto_parser
75.
76.         # get the received port number from packet_in message.
77.         in_port = msg.match['in_port']
78.
79.         # analyse the received packets using the packet library.
80.         pkt = packet.Packet(msg.data)
81.         eth = pkt.get_protocols(ethernet.ethernet)[0]
82.
83.         if eth.ethertype == ether_types.ETH_TYPE_LLDP:
84.             # ignore lldp packet
85.             return
86.         dst = eth.dst
87.         src = eth.src
88.
89.         # get Datapath ID to identify OpenFlow switches.
90.         dpid = datapath.id
91.         self.mac_to_port.setdefault(dpid, {})
92.
93.         self.logger.info("packet in %s %s %s %s", dpid, src, dst,
    in_port)
94.
95.         # learn a mac address to avoid FLOOD next time.
96.         self.mac_to_port[dpid][src] = in_port
97.
98.         # if the destination mac address is already learned,
99.         # decide which port to output the packet, otherwise FLOOD.
100.        if dst in self.mac_to_port[dpid]:
101.            out_port = self.mac_to_port[dpid][dst]
102.        else:
103.            out_port = ofproto.OFPP_FLOOD
104.
105.        # construct action list.
106.        actions = [parser.OFPActionOutput(out_port)]
107.
108.        # install a flow to avoid packet_in next time
109.        if out_port != ofproto.OFPP_FLOOD:
110.            match = parser.OFPMatch(in_port=in_port, eth_dst=dst,
    eth_src=src)
111.            # verify if we have a valid buffer_id, if yes avoid to
    send both
112.            # flow_mod & packet_out
113.            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
114.                self.add_flow(datapath, 1, match, actions,
```

```
    msg.buffer_id)
115.                 return
116.             else:
117.                 self.add_flow(datapath, 1, match, actions)
118.         data = None
119.         if msg.buffer_id == ofproto.OFP_NO_BUFFER:
120.             data = msg.data
121.
122.         # construct packet_out message and send it.
123.         out = parser.OFPPacketOut(datapath=datapath,
    buffer_id=msg.buffer_id,
124.                                   in_port=in_port, actions=actions,
    data=data)
125.         datapath.send_msg(out)
```

Cuando Ryu recibe un mensaje *packet_in*, se genera un evento *ofp_event.EventOFPPacketIn*. El decorador *set_ev_cls* le dice a Ryu cuándo se debe llamar a la función asociada, que es *packet_in_handler*.

El primer argumento del decorador *set_ev_cls* es *ofp_event.EventOFPPacketIn*, y el segundo *MAIN_DISPATCHER*. *MAIN_DISPATCHER* denota el estado normal del conmutador, significa que esta función se llama solo después de que se completa la negociación.

El método comienza verificando la longitud del mensaje para asegurarse de que el *switch* recibió el mensaje completo; de lo contrario, se muestra un mensaje indicando que el paquete se truncó, lo que puede causar problemas en el enrutamiento del paquete.

Se definen variables como en los métodos anteriores, si bien se incluye también *in_port* con el puerto por el que se recibió el mensaje.

Se parsean los datos del mensaje y se crea una variable eth, la cual es una estructura de datos que contiene la dirección MAC de destino, el *ethertype* (tipo de protocolo que encapsula) y la dirección MAC de origen.

Usando el *ethertype*, ignora los paquetes del protocolo LLDP.

Imprime un mensaje indicando los campos MAC de origen, MAC de destino y el puerto de entrada; y añade a la tabla MAC de ese *switch* la dirección de origen y el puerto en el que se encuentra conectada. Así es como aprende.

Después comprueba si la dirección de destino está registrada en la tabla MAC, si es así guarda el puerto indicado en la variable *out_port*; de lo contrario rellena esta variable con la constante *ofproto.OFPP_FLOOD*, para inundar por todos los puertos.

Prepara la acción con *OFPActionOutput*, que se usa con un mensaje *packet_out* para especificar el puerto del *switch* desde el que se debe enviar el paquete.

Si la dirección de destino ya está registrada, es decir se conoce el puerto en el que está conectado, el controlador crea una entrada para la tabla de flujos del switch; de manera que el

switch envíe los próximos paquetes por dicho puerto. Para ello se define la variable *match* que contiene las reglas de coincidencia a aplicar; que son, el puerto de entrada, la dirección de destino y la dirección de origen.

Se llama entonces al método *add_flow*, definido anteriormente pasando como parámetros el switch, la prioridad (mayor que la del método *switch_features_handler*, que es la regla por defecto), las reglas del match, y la acción.

Por último, el controlador debe devolver al *switch* el paquete inicial para que lo envíe según lo deducido. Para ello se utiliza *parser.OFPPacketOut* y se construye un mensaje *out*, incluyendo los datos del mensaje inicial y la acción determinada anteriormente; y se envía al *switch* haciendo uso de su método *send_msg()*.

A.2 Script Topología Mininet

En esta sección se incluye el código utilizado para desplegar la topología en Mininet.

```
1.  #!/usr/bin/python
2.
3.  from mininet.net import Mininet
4.  from mininet.node import Controller, RemoteController, OVSController
5.  from mininet.node import CPULimitedHost, Host, Node
6.  from mininet.node import OVSKernelSwitch, UserSwitch
7.  from mininet.node import IVSSwitch
8.  from mininet.cli import CLI
9.  from mininet.log import setLogLevel, info
10. from mininet.link import TCLink, Intf
11. from subprocess import call
12.
13. def myNetwork():
14.
15.     net = Mininet( topo=None,
16.                    build=False,
17.                    ipBase='192.168.1.0/24')
18.
19.     info( '*** Adding controller\n' )
20.     c0=net.addController(name='c0',
21.                          controller=RemoteController,
22.                          protocol='tcp',
23.                          port=6633)
24.
25.     info( '*** Add switches\n')
26.     s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
27.     Intf( 'enp0s8', node=s1 )
28.     s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
29.     s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
30.     s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
31.     Intf( 'enp0s3', node=s4 )
32.
33.
34.     info( '*** Add hosts\n')
35.     h1 = net.addHost('h1', cls=Host, ip='0.0.0.0',
    mac='08:00:27:23:db:ca')
36.
37.     info( '*** Add links\n')
38.     net.addLink(s1, s2)
39.     net.addLink(h1, s2)
40.     net.addLink(h1, s3)
41.     net.addLink(s3, s4)
42.
43.     info( '*** Starting network\n')
44.     net.build()
45.     info( '*** Starting controllers\n')
```

```
46.      for controller in net.controllers:
47.          controller.start()
48.
49.      info( '*** Starting switches\n')
50.      net.get('s1').start([c0])
51.      net.get('s2').start([c0])
52.      net.get('s4').start([c0])
53.      net.get('s3').start([c0])
54.
55.      info( '*** Post configure switches and hosts\n')
56.
57.      h1.cmdPrint('dhclient '+h1.defaultIntf().name)
58.      h1.cmdPrint('dhclient h1-eth1')
59.
60.      CLI(net)
61.      net.stop()
62.
63.  if __name__ == '__main__':
64.      setLogLevel( 'info' )
65.      myNetwork()
```

La topología contiene cuatro *switches* y *host* virtual. El *switch* S1 tiene una interfaz externa, asociada a la interfaz *enp0s8* y el *switch* S4 otra asociada a *enp0s3*. El *host* virtual situado en mitad de la red obtiene las direcciones IP de sus dos interfaces por medio de DHCP.

A.3 Script Puente MQTT to InfluxDB

Esta sección contiene el código utilizado en el *host* virtual de la red SDN para recibir mensajes MQTT, procesarlos y enviar los datos a InfluxDB. Está basado en [66].

```
1.  #!/usr/bin/env python3
2.
3.  """A MQTT to InfluxDB Bridge
4.  This script receives MQTT data and saves those to InfluxDB.
5.  """
6.
7.  import json
8.  import base64
9.  import re
10. from typing import NamedTuple
11. import paho.mqtt.client as mqtt
12. from influxdb import InfluxDBClient
13.
14. INFLUXDB_ADDRESS = '192.168.56.113' # Ip de InfluxDB
15. INFLUXDB_USER = 'root'
16. INFLUXDB_PASSWORD = 'root'
17. INFLUXDB_DATABASE = 'test_db' # nombre de la base de datos
18.
19. MQTT_ADDRESS = '192.168.1.124' # Ip del bróker MQTT
20. MQTT_USER = ''    #'mqttuser'
21. MQTT_PASSWORD = ''      #'mqttpassword'
22. MQTT_TOPIC = 'application/1/device/70b3d549953cf6c5/rx'
23. MQTT_CLIENT_ID = 'MQTTInfluxDBBridge'
24.
25. influxdb_client = InfluxDBClient(INFLUXDB_ADDRESS, 8086,
    INFLUXDB_USER, INFLUXDB_PASSWORD, INFLUXDB_DATABASE)
26.
27. class SensorData(NamedTuple):
28.       location: str
29.       measurement: str
30.       value: float
31.
32. def on_connect(client, userdata, flags, rc):
33.     print('Connected with result code ' + str(rc))
34.     client.subscribe(MQTT_TOPIC)
35.
36. def on_message(client, userdata, msg):
37.     print("Mensaje recibido\n Tópico: "+msg.topic + '\n Payload: ' +
    str(msg.payload))
38.     (sensor_data1, sensor_data2) = _parse_mqtt_message(msg.topic,
    msg.payload)
39.     if sensor_data1 is not None:
40.         _send_sensor_data_to_influxdb(sensor_data1)
41.         print("Escribiendo en la base de datos: temperatura")
42.     if sensor_data2 is not None:
```

```
43.          _send_sensor_data_to_influxdb(sensor_data2)
44.          print("Escribiendo en la base de datos: humedad")
45.
46.  def _parse_mqtt_message(topic, payload):
47.      str(payload)
48.      y=json.loads(payload)
49.      x=y["data"]
50.      base64_bytes = x.encode('ascii')
51.      message_bytes = base64.b64decode(base64_bytes)
52.      message = message_bytes.decode('ascii')
53.      print("Datos decodificados: "+message)
54.
55.      mens2=json.loads(message)
56.
57.      temp=mens2["Temperatura"]
58.      humd=mens2["Humedad"]
59.      location="livingroom"
60.
61.      SensorData1=SensorData(location, "temperature", float(temp))
62.      SensorData2=SensorData(location, "humidity", float(humd))
63.
64.      return SensorData1,SensorData2
65.
66.  def _send_sensor_data_to_influxdb(sensor_data):
67.      json_body = [
68.          {
69.              'measurement': sensor_data.measurement,
70.              'tags': {
71.                  'location': sensor_data.location
72.              },
73.              'fields': {
74.                  'value': sensor_data.value
75.              }
76.          }
77.      ]
78.      influxdb_client.write_points(json_body)
79.
80.  def _init_influxdb_database():
81.      databases = influxdb_client.get_list_database()
82.      if len(list(filter(lambda x: x['name'] == INFLUXDB_DATABASE,
     databases))) == 0:
83.          influxdb_client.create_database(INFLUXDB_DATABASE)
84.      influxdb_client.switch_database(INFLUXDB_DATABASE)
85.
86.  def main():
87.      _init_influxdb_database()
88.
89.      mqtt_client = mqtt.Client(MQTT_CLIENT_ID)
90.      #mqtt_client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
91.      mqtt_client.on_connect = on_connect
92.      mqtt_client.on_message = on_message
93.      mqtt_client.connect(MQTT_ADDRESS, 1883)
```

```
94.      mqtt_client.loop_forever()
95.
96.  if __name__ == '__main__':
97.      print('MQTT to InfluxDB bridge')
98.      main()
```

En el script se define una clase *SensorData*, que contiene tres campos, el tipo de medida (temperatura o humedad), el valor numérico medido y un campo para indicar la localización del sensor (en este proyecto, este campo permanece constante "*livingroom*", pero en el caso de usar más motas/sensores, sería variable en función de algún parámetro del mensaje recibido). Estos tres campos son los que se escriben en la base de datos.

El método principal comienza conectándose a la base de datos específica (si no existe la crea), para ello hace uso del cliente para InfluxDB, tras esto crea un cliente MQTT, se suscribe al tópico y comienza a recibir y procesar mensajes. Tras esto, los envía y bloquea la hebra principal para quedarse iterando todo el rato recibiendo mensajes.

A.4 Script Pytorch

En esta sección se incluye el código utilizado para obtener los datos almacenados en InfluxDB y generar y entrenar con ellos varios algoritmos de *Machine Learning*.

Obtención de datos y determinación del horizonte de predicción

El código recogido en este apartado corresponde a la obtención de datos desde la base de datos, su procesamiento inicial y la definición del alcance de la predicción.

El código comienza importando las librerías necesarias:

```
1.  import re
2.  from typing import NamedTuple
3.  import pandas as pd
4.  from influxdb import DataFrameClient, InfluxDBClient
5.  import os
6.  import warnings
7.  warnings.filterwarnings("ignore")
8.  import copy
9.  from pathlib import Path
10. import warnings
11. import numpy as np
12. import pytorch_lightning as pl
13. from pytorch_lightning.callbacks import EarlyStopping,
    LearningRateMonitor
14. from pytorch_lightning.loggers import TensorBoardLogger
15. import torch
16. from pytorch_forecasting import Baseline, TemporalFusionTransformer,
    TimeSeriesDataSet
17. from pytorch_forecasting.data import GroupNormalizer
18. from pytorch_forecasting.metrics import SMAPE, PoissonLoss,
    QuantileLoss
19. from pytorch_forecasting.models.temporal_fusion_transformer.tuning
    import optimize_hyperparameters
20. from pytorch_forecasting.data import NaNLabelEncoder
```

Tras esto creamos el cliente de InfluxDB. también indicamos parámetro de la petición que vamos a hacer, en este caso el valor de la localización, y la fecha de inicio y de final de los datos:

```
21.     INFLUXDB_ADDRESS = '192.168.56.113'
22.     INFLUXDB_USER = 'root'
23.     INFLUXDB_PASSWORD = 'root'
24.     INFLUXDB_DATABASE = 'home_db'
25.     Localizacion="livingroom"
26.     bind_params = {'location': Localizacion,
27.                    'start': '2021-01-15T00:00:00.0Z',
28.                    'stop': '2021-03-15T23:59:59.0Z',
```

```
    29.                     }
 30.
    31.     influxdb_client = InfluxDBClient(INFLUXDB_ADDRESS, 8086,
INFLUXDB_USER, INFLUXDB_PASSWORD, INFLUXDB_DATABASE)
```

Hacemos dos queries, la primera para obtener los datos de la temperatura y la segunda para los datos de humedad. Una vez obtenidos los datos, los cargamos en Dataframes de Pandas y los unimos en uno solo (en este caso se piden la media de los datos agrupados en periodos de 1 hora):

```
    32.     query1 = 'SELECT MEAN(*) FROM temperature WHERE
location=$location AND time >= $start AND time <= $stop GROUP BY time(1h)
'
    33.     print("Querying data: " + query1)
    34.     result1 = influxdb_client.query(query1,
bind_params=bind_params)
    35.     df1 = pd.DataFrame(result1['temperature'])
    36.     df1.rename(columns={'mean_value':'temperature'},inplace=True)
    37.
    38.     query2 = 'SELECT MEAN(*) FROM humidity WHERE
location=$location AND time >= $start AND time <= $stop GROUP BY time(1h)
'
    39.     print("Querying data: " + query2)
    40.     result2 = influxdb_client.query(query2,
bind_params=bind_params)
    41.     df2 = pd.DataFrame(result2['humidity'])
    42.     df2.rename(columns={'mean_value':'humidity'},inplace=True)
    43.
    44.     df=pd.merge(df1,df2,on="time")
 45.  df
    46.
```

Aplicamos algunos cambios al Dataframe, como eliminar los valores nulos o añadir más campos de información como la Hora, o un índice numérico para facilitar el procesado posterior:

```
    47.     #df.loc[pd.isna(df.mean_value), 'mean_value'] =
df.mean_value.mean()
    48.
    49.     df=df.dropna()
    50.     df=df.reset_index(drop=True)
    51.
    52.     df['time'] = pd.to_datetime(df['time'],format='%Y-%m-%d
%H:%M:%S')
    53.
    54.     #df['Anio'] = df['time'].map(lambda x: x.strftime('%Y'))
    55.     #df['Mes'] = df['time'].map(lambda x: x.strftime('%m'))
    56.     #df['Dia'] = df['time'].map(lambda x: x.strftime('%d'))
    57.     df['Hora'] = df['time'].map(lambda x: x.strftime('%H'))
    58.     #df['Momento'] = df['time'].map(lambda x:
```

```
x.strftime('%H:%M:%S'))
    59.
    60.     df['Location'] = Localizacion
    61.
    62.     #df['mean_value'] = round(df['mean_value'],1)
    63.
    64.     df['time_idx'] = df.index+1
 65.  df
    66.
```

Indicamos la longitud de la predicción que vamos a hacer y la longitud de los datos anteriores en los que se va a basar dicha predicción.

```
    67.     max_prediction_length = 5*24
    68.     max_encoder_length = 30*24
 69.  training_cutoff = df["time_idx"].max() - max_prediction_length
    70.
```

Todo este código mostrado, es similar para los algoritmos implementados. El código a ejecutar tras esto, difiere en varios puntos.

A continuación, se diferencia entre el código para cada uno de los algoritmos:

Temporal Fusion Transformer

En esta subsección se indica el código específico para este modelo.

Creamos una instancia de TimeSeriesDataSet, indicando los diferentes campos de los datos:

```
 71.  training = TimeSeriesDataSet(
 72.      df[lambda x: x.time_idx <= training_cutoff],
 73.      time_idx="time_idx",
 74.      target="temperature",
 75.      group_ids=["Location"],
 76.      min_encoder_length=max_encoder_length // 2,
 77.      max_encoder_length=max_encoder_length,
 78.      min_prediction_length=1,
 79.      max_prediction_length=max_prediction_length,
 80.      static_categoricals=["Location"],
 81.      static_reals=[],
 82.      time_varying_known_categoricals=["Hora"],
 83.      time_varying_known_reals=[],
 84.      time_varying_unknown_categoricals=[],
 85.      time_varying_unknown_reals=[
 86.          "temperature", "humidity"
 87.      ],
```

```
88.
89.      add_relative_time_idx=False,
90.      add_target_scales=True,
91.      add_encoder_length=True,
92.  )
93.
94.  validation = TimeSeriesDataSet.from_dataset(training, df,
     predict=True, stop_randomization=True)
95.
96.  batch_size = 128  # set this between 32 to 128
97.  train_dataloader = training.to_dataloader(train=True,
     batch_size=batch_size, num_workers=0)
98.  val_dataloader = validation.to_dataloader(train=False,
     batch_size=batch_size * 10, num_workers=0)
99.
```

Opcionalmente, podemos calcular un error de base para comprobar posteriormente la eficacia del modelo:

```
100. actuals = torch.cat([y for x, y in iter(val_dataloader)])
101. baseline_predictions = Baseline().predict(val_dataloader)
102. (actuals - baseline_predictions).abs().mean().item()
103.
```

Configuramos el entrenador y el modelo:

```
104. trainer = pl.Trainer(
105.     gpus=0,
106.     gradient_clip_val=0.1,
107. )
108.
109. tft = TemporalFusionTransformer.from_dataset(
110.     training,
111.     learning_rate=0.03,
112.     hidden_size=64,
113.     attention_head_size=4,
114.     dropout=0.1,
115.     hidden_continuous_size=8,
116.     output_size=7,  # 7 quantiles
117.     loss=QuantileLoss(),
118.     log_interval=10,
119.     reduce_on_plateau_patience=4,
120. )
121. print(f"Number of parameters in network: {tft.size()/1e3:.1f}k")
122.
```

Buscamos un valor de tasa de aprendizaje (*learning rate*) adecuado:

```
res = trainer.tuner.lr_find(
    tft,
    train_dataloader=train_dataloader,
    val_dataloaders=val_dataloader,
    max_lr=10.0,
    min_lr=1e-6,
)

print(f"suggested learning rate: {res.suggestion()}")
fig = res.plot(show=True, suggest=True)
fig.show()

```

Reconfiguramos usando el valor obtenido de tasa de aprendizaje:

```
early_stop_callback = EarlyStopping(monitor="val_loss",
    min_delta=1e-4, patience=10, verbose=False, mode="min")
lr_logger = LearningRateMonitor()  # log the learning rate
logger = TensorBoardLogger("lightning_logs")  # logging results to
    a tensorboard

trainer = pl.Trainer(
    max_epochs=30,
    gpus=0,
    weights_summary="top",
    gradient_clip_val=0.1,
    limit_train_batches=30,
    callbacks=[lr_logger, early_stop_callback],
    logger=logger,
)

tft = TemporalFusionTransformer.from_dataset(
    training,
    learning_rate=0.03,
    hidden_size=64,
    attention_head_size=4,
    dropout=0.1,
    hidden_continuous_size=8,
    output_size=7,  # 7 quantiles
    loss=QuantileLoss(),
    log_interval=10,
    reduce_on_plateau_patience=4,
)
print(f"Number of parameters in network: {tft.size()/1e3:.1f}k")

```

Entrenamos la red:

```
trainer.fit(
    tft,
    train_dataloader=train_dataloader,
    val_dataloaders=val_dataloader,
)

```

Cargamos el mejor modelo obtenido y representamos lo obtenido:

```
best_model_path = trainer.checkpoint_callback.best_model_path
best_tft = TemporalFusionTransformer.load_from_checkpoint(best_model_path)

actuals = torch.cat([y for x, y in iter(val_dataloader)])
predictions = best_tft.predict(val_dataloader)
(actuals - predictions).abs().mean()

raw_predictions, x = best_tft.predict(val_dataloader, mode="raw", return_x=True)
best_tft.plot_prediction(x, raw_predictions, idx=0, add_loss_to_title=True);
```

N-BEATS

En esta subsección se incluye el código específico usado para el modelo N-BEATs:

Instanciamos un objeto de la clase TimeSeriesDataSet indicando los campos de los datos. En este caso, el modelo N-BEATs, no admite covariables, por lo que solamente se indica la variable objetivo:

```
training = TimeSeriesDataSet(
    df[lambda x: x.time_idx <= training_cutoff],
    time_idx="time_idx",
    target="temperature",
    group_ids=["Location"],
    allow_missings=True,
    time_varying_unknown_reals=["temperature"],
    max_encoder_length=max_encoder_length,
    max_prediction_length=max_prediction_length,
)
validation = TimeSeriesDataSet.from_dataset(training, df, min_prediction_idx= training_cutoff + 1)
batch_size = 128
train_dataloader = training.to_dataloader(train=True,
```

```
    batch_size=batch_size, num_workers=0)
84. val_dataloader = validation.to_dataloader(train=False,
    batch_size=batch_size, num_workers=0)
85.
```

Creamos el entrenador y el modelo; y buscamos el valor óptimo para la tasa de aprendizaje:

```
86. trainer = pl.Trainer(gpus=0, gradient_clip_val=0.1)
87. net = NBeats.from_dataset(
88.     training,
89.     learning_rate=0.03,
90.     log_interval=10,
91.     log_val_interval=1,
92.     weight_decay=1e-2,
93.     widths=[256, 2048],
94.     backcast_loss_ratio=1.0,
95. )
96. res = trainer.tuner.lr_find(net, train_dataloader=train_dataloader,
    val_dataloaders=val_dataloader, min_lr=1e-5)
97. print(f"suggested learning rate: {res.suggestion()}")
98. fig = res.plot(show=True, suggest=True)
99. fig.show()
100. net.hparams.learning_rate = res.suggestion()
101.
```

Con el valor obtenido para la tasa de aprendizaje, volvemos a configurar el modelo y lo entrenamos:

```
102. early_stop_callback = EarlyStopping(monitor="val_loss",
     min_delta=1e-4, patience=10, verbose=False, mode="min")
103. trainer = pl.Trainer(
104.     max_epochs=100,
105.     gpus=0,
106.     weights_summary="top",
107.     gradient_clip_val=0.1,
108.     callbacks=[early_stop_callback],
109.     limit_train_batches=30,
110. )
111. net = NBeats.from_dataset(
112.     training,
113.     learning_rate=0.003,
114.     log_interval=10,
115.     log_val_interval=1,
116.     weight_decay=1e-2,
117.     widths=[256, 2048],
```

```
    backcast_loss_ratio=1.0,
)
trainer.fit(
    net,
    train_dataloader=train_dataloader,
    val_dataloaders=val_dataloader,
)

```

Cargamos el mejor modelo obtenido y lo representamos:

```
best_model_path = trainer.checkpoint_callback.best_model_path
best_model = NBeats.load_from_checkpoint(best_model_path)
raw_predictions, x = best_model.predict(val_dataloader, mode="raw",
    return_x=True)
best_model.plot_prediction(x, raw_predictions, idx=0,
    add_loss_to_title=True);

```

Nuevas predicciones

Para realizar nuevas predicciones a partir de los modelos obtenidos, el código a usar es similar y es el que se incluye en esta subsección.

Creamos una nueva predicción a partir de los últimos datos obtenidos y la representamos:

```
encoder_data = df[lambda x: x.time_idx > x.time_idx.max() -
    max_encoder_length]
last_data = df[lambda x: x.time_idx == x.time_idx.max()]
decoder_data = pd.concat([last_data.assign(time=lambda x: x.time +
    pd.offsets.Hour(i)) for i in range(1, max_prediction_length +
    1)],ignore_index=True,)
decoder_data['time_idx'] = decoder_data.index
decoder_data["time_idx"] += encoder_data["time_idx"].max() + 1 -
    decoder_data["time_idx"].min()
decoder_data['Hora'] = decoder_data['time'].map(lambda x:
    x.strftime('%H'))
new_prediction_data = pd.concat([encoder_data, decoder_data],
    ignore_index=True)
new_raw_predictions, new_x = best_tft.predict(new_prediction_data,
    mode="raw", return_x=True)
best_tft.plot_prediction(new_x, new_raw_predictions, idx=0,
    show_future_observed=False);

```

Si queremos almacenar dicha predicción en la base de datos, por ejemplo, para visualizarlos junto a los datos medidos en Grafana, creamos un Dataframe con las predicciones:

```
210. npd1=new_prediction_data[["time","Location"]]
211. npd1 = npd1.tail(max_prediction_length)
212. npd1.rename(columns={'Location':'location'},inplace=True)
213.
214. idx=0
215. y_pred = new_raw_predictions["prediction"].detach().cpu()
216. y_hat = y_pred[idx, : new_x["decoder_lengths"][idx]]
217. tensor=best_tft.loss.to_prediction(y_hat.unsqueeze(0))[0]
218. px = pd.DataFrame(tensor.numpy())
219. px.columns=['value']
220. px=px.set_index(npd1.index)
221.
222. npd1=pd.concat([npd1, px], axis=1)
223. npd1.set_index('time',inplace=True)
224. npd1
225.
```

Usamos el cliente de Dataframes de la base de datos para escribirlos (en este caso las predicciones las estamos escribiendo en otra base de datos y con el nombre del modelo incluido en el nombre de la variable a guardar):

```
226. influxdb_client3 = DataFrameClient(INFLUXDB_ADDRESS, 8086,
     INFLUXDB_USER, INFLUXDB_PASSWORD, 'prediction_db')
227. influxdb_client3.write_points(npd1,'temperature_tft',
     protocol='line')
```

Con esto finaliza el código utilizado para obtener predicciones usando algoritmos de *Machine Learning* en Pytorch.