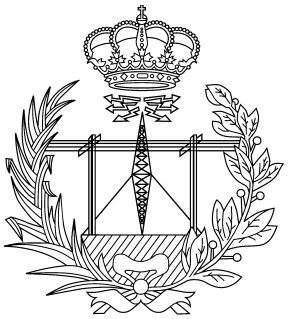
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Trabajo Fin de Grado

Uso de Inteligencia Artificial para solucionar problemas de *Network Embedding* en redes móviles con virtualización

(Use of Artificial Inteligence to solve problems of Network Embedding in mobile networks with virtualization)

Para acceder al Título de

Graduado en
Ingeniería de Tecnologías de Telecomunicación

Autor: Julio Luis Medina Samamé

Septiembre- 2023

UNIVERSIDAD DE CANTABRIA

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Julio Luis Medina Samamé
Director del TFG: Luis Francisco Díez Fernández, Ramón Agüero Calvo
Título: "Uso de Inteligencia Artificial para solucionar problemas de *Network Embedding* en redes móviles con virtualización"
Title: "Use of Artificial Inteligence to solve problems of Network Embedding in mobile networks with virtualization"

Presentado a examen el día: 21 de septiembre de 2023

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del tribunal:
Presidente (Apellidos, Nombre): Zamanillo Sainz de la Maza, José María
Secretario (Apellidos, Nombre): Vielva Martínez, Luis Antonio
Vocal (Apellidos, Nombre): García Gutiérrez, Alberto Eloy

Este Tribunal ha resulto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFG (solo si es distinto del Secretario)
V^{o} B^{o} del Subdirector	Trabajo Fin de Grado N^{o} (a asignar por Secretaría)

Agradecimientos

Agradezco a mis padres y mi hermana por el apoyo y comprensión ofrecidos en todo momento a lo largo del Grado, y a mi amigo Carlos por acompañarme durante todos estos años.

Igualmente, agradezco a mis directores Luis Francisco y Ramón por la paciencia que han tenido conmigo desde el principio.

Finalmente, agradezco también a todos aquellos profesores que durante el Grado no solo han hecho una gran labor docente sino que también han tenido paciencia y han hecho del Grado una experiencia maravillosa.

Resumen

En el contexto de las redes celulares de quinta generación uno de los principales cambios a nivel de arquitectura es la capacidad de virtualizar los elementos de acceso. En este sentido, las estaciones base 5G se desagregan en entidades físicas y virtuales, *Central Units* y *Distributed Units*, que implementan el procesado de banda base. Éstas deben ser virtualizadas sobre la red sustrato o red física, y para ello se precisan sistemas adaptativos que determinen cuál es la mejor forma de hacer la asignación en función las preferencias e intereses del operador de red. Este trabajo propone una arquitectura basada en *Machine Learning*, concretamente en Aprendizaje Reforzado, con la que se puedan entrenar y probar redes neuronales con el objetivo de solucionar el problema de *network embedding* mencionado.

Palabras clave: Redes Móviles; Quinta Generación; 5G; Inteligencia Artificial; Machine Learning; Aprendizaje Reforzado; Redes Neuronales; Network Embedding; Virtualización de Redes;

Abstract

In the context of fifth generation of cellular networks one of the key architectural shifts comes from the virtualization of access network elements. In this sense, 5G base stations are disaggregated into physical and virtual entities, Central Units and Distributed Units, which perform the baseband processing. In turn, these entities must be embedded on the substrate network or physical network, and for this a system that determines the best way to make the assignment, based on the operator requirements and preferences, is needed. This work proposes an architecture based on Machine Learning, specifically on Reinforcement Learning, with which Neural Networks can be trained and tested with the aim of solving the network embedding problem mentioned.

Keywords: Mobile Networks; Fifth Generation; 5G; Artificial Intelligence; Machine Learning; Reinforcement Learning; Neural Networks; Network Embedding; Network Virtualization

Índice general

Índice de figuras

Índice de tablas

Índice de acrónimos

BBU	*Base Band Unit*.
BS	*Base Station*.
C-RAN	*Cloud Radio Access Network*.
CU	*Central Unit*.
DL	*Dense Layer*.
DU	*Distributed Unit*.
FS	*Functional Split*.
GCN	*Graph Convolutional Network*.
IA	Inteligencia Artificial.
ML	*Machine Learning*.
NFV	*Network Function Virtualization*.
NN	*Neural Network*.
RL	*Reinforcement Learning*.
RRH	*Remote Radio Head*.
RU	*Radio Unit*.
SDN	*Software Defined Networking*.
SN	*Substrate Network*.
VN	*Virtual Network*.
VNE	*Virtual Network Embedding*.
vRAN	*Virtual Radio Access Network*.

Capítulo 1

Introducción

La aparición y desarrollo de nuevos servicios impone requisitos cada vez más exigentes a las redes de comunicaciones. Recientemente, la aparición de la tecnología 5G ha supuesto una transformación profunda de los diferentes segmentos de red, abarcando desde el interfaz radio hasta el núcleo de la red. Además del aumento de capacidad de comunicación, los nuevos servicios requieren reconfigurar la red en función de sus características.

Uno de los principales avances desde el punto de vista de arquitectura y reconfiguración de la red proviene de la capacidad de virtualizar y gestionar funciones de red mediante técnicas de *Software Defined Networking* (SDN) y *Network Function Virtualization* (NFV). Esto permite que funciones tradicionalmente estáticas o con una capacidad de re-configuración limitada puedan ser actualizadas sin mucho esfuerzo, al tratarse de servicios virtualizados. Además, los entornos de virtualización permiten ubicar estos servicios allí donde sea más conveniente.

Los esquemas de SDN y NFV se han venido utilizando en la red de transporte para gestionar la ubicación y configuración des dispositivos como *switches* o *firewalls*. Más recientemente se ha llevado esta idea a la red de acceso para sistemas celulares, dando lugar a la llamada *Virtual Radio Access Network* (vRAN). De este modo, funcionalidades de las estaciones base se virtualizan y se ubican en puntos distantes a las antenas, cuya ubicación dependerá de varios factores.

Inicialmente, se propusieron soluciones de centralización total, en lo que se llamó *Cloud Radio Access Network* (C-RAN) [1, 2]. De este modo, la estación base se separa en dos entidades: una primera que incluye las antenas y las funcionalidades de radio-frecuencia, llamada *Remote Radio Head* (RRH), y una entidad virtualizada que implementa las funciones de procesado de banda base o *Base Band Unit* (BBU). Este esquema permitiría un gran nivel de centralización de las funciones de las estaciones base, lo que a su vez facilitaría una mayor coordinación entre estaciones base. Sin embargo estas arquitecturas requieren una gran capacidad de la red *fronthaul*, que comunica las RRH y BBU [3], lo que supone grandes inversiones para el despliegue de fibra óptica. Para solventar este problema se propuso centralizar únicamente algunas de las funcionalidades de las BBU para reducir la capacidad requerida, de modo que las funciones concretas que se centralizan vienen dadas por la llamada división funcional o *functional split* [4, 5]. En este nuevo esquema la estación base se divide en 3 entidades: *Radio Unit* (RU), DU y CU. La RU es una entidad física que reemplaza a la RRH y contiene las antenas y funciones de radio-frecuencia. Mediante la red *fronthaul* la RU se conecta a la DU, que se trata de una entidad virtual que implementa

los las capas bajas de la pila de protocolos de la estación base. Finalmente, la DU se conecta con la CU, que implementa el resto de funcionalidades. De este modo, la configuración de *functional split* establece las funcionalidades ubicadas en DU y CU.

La selección de la división funcional estará condicionada por la calidad de servicio requerida, por la capacidad de comunicaciones entre la DU y CU, y por las capacidades de cómputo disponibles para estas entidades virtualizadas. Esto implica que los operadores de red deberán decidir el emplazamiento de las unidades de forma eficiente, y modificar estas ubicaciones en función de los servicios desplegados, dando lugar a problemas de optimización de *Virtual Network Embedding* (VNE) que son generalmente combinatorios [6] y, por lo tanto, difíciles de resolver. Además, la comparación de diferentes decisiones de ubicación responderá a las políticas de los operadores (incluyendo perspectivas de calidad de servicio, gestión de infraestructura, energéticas, etc.), lo que puede ser complejo de modelar de forma matemática.

1.1. Objetivos

En este contexto, en este trabajo se plantea, desarrolla y evalúa una arquitectura basada en Inteligencia Artificial (IA) y *Reinforcement Learning* (RL) para resolver el problema de ubicación de las entidades virtualizadas CU y DU, basándose en otras soluciones propuestas en diferentes publicaciones para problemas más generales o entornos similares. La solución desarrollada se plantea como un posible punto de partida para el desarrollo de una solución más completa al problema de virtualización de redes planteado. Por ello se busca definir una arquitectura clara, y documentar el código generado de forma que sea entendible para poder adaptado a condiciones de red concretas.

Con todo ello, en este trabajo se definen los siguientes objetivos:

1. Evaluar la posibilidad de utilizar entornos IA basados en redes neuronales para resolver el problema específico de VNE, identificando el tipo de algoritmo o solución que se adapte al problema.

2. Presentar un diseño para resolver este problema, que pueda adaptarse a requisitos o políticas variables, haciendo uso de RL.

3. Generar, a partir del diseño, un código capaz entrenar una red neuronal para que tome decisiones de *embedding* de una *Virtual Network* (VN) sobre una *Substrate Network* (SN). Estos términos se detallarán en el Capítulo 2.

1.2. Estructura del documento

A continuación se describe la estructura del documento.

En el Capítulo 2 se exponen los conceptos teóricos y tecnológicos de los que se hace uso en el trabajo, incluyendo tanto conceptos de arquitecturas de redes 5G como de soluciones basadas en IA.

Posteriormente, en el Capítulo 3 se presenta el diseño funcional de la solución de optimización implementada, en el que se indican tanto los bloques que forman parte de la solución como la relación entre ellos.

A partir de este diseño, en el Capítulo 4 se describe la implementación software de la solución desarrollada en lenguaje de programación Python, indicando sus aspectos más relevantes y las dependencias con paquetes software utilizados, y se presenta la validación de dicha implementación sobre escenarios concretos.

Finalmente, en el el Capítulo 5 se resume el trabajo realizado en función de los objetivos inicialmente planteados y se indican posibles líneas futuras de investigación, que pueden realizarse a partir de este trabajo.

Capítulo 2

Marco teórico

En este capítulo se exponen los conceptos teóricos necesarios para entender el problema que se está tratando de solucionar, así como la solución que se propone.

Para ello se presentan dos apartados relacionados con el problema real, uno explicando la topología de la red física de la que se dispone y otro profundizando en el concepto de virtualización de redes y su aplicación al problema real. Posteriormente, en otras dos secciones, se explican los conceptos básicos de ML utilizados en el diseño de la solución software desarrollada.

Con este capítulo se busca no sólo mostrar los conocimientos teóricos y las fuentes que se han empleado para afrontar este trabajo, sino también hacer de éste un contenido accesible a toda persona interesada en el tema, aportando explicaciones concisas y facilitando bibliografía que profundice más y de forma rigurosa en aquellos aspectos sobre los que pueda resultar conveniente o interesante, para entender o incluso mejorar este trabajo.

2.1. Topología de red

Como se ha mencionado en la introducción, este trabajo se sitúa en el contexto de las redes para telefonía inalámbrica de quinta generación o 5G. Estas redes son la evolución de las tecnologías anteriores, 3G y 4G, y, por tanto, conservan y mejoran muchas de sus características. Desde una perspectiva de arquitectura, existen dos ideas clave: la centralización y la separación de las funciones de red.

Por contradictorio que parezca, estas dos ideas conviven en 5G, ya que hereda la centralización de las funciones de red sobre la estación base o *Base Station* (BS), y, a su vez, se propone una división funcional en la BS. Estas dos ideas se implementan mediante la NFV o virtualización de funciones de red y *Functional Split* (FS), o separación funcional.

La NFV consiste en la utilización de infraestructura y *hardware* de propósito general para virtualizar todas las funciones de red implementadas en software. Esto permite la centralización de las mismas, lo cual facilita el despliegue, el mantenimiento y la re-configuración de esa parte de la infraestructura.

Mientras que el concepto de NFV es genérico a cualquier segmento de red, el de FS se refiere a la red de acceso inalámbrica. Concretamente, la FS atañe a la estrategia que se utiliza a la hora de realizar la virtualización, separando las funciones de las BS en diferentes componentes. Como se mencionó anteriormente, la separación se realiza en 3 elementos (RU, DU y CU), pero únicamente dos de ellos son

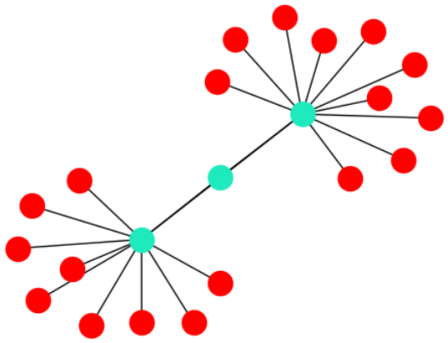


Figura 2.1: Representación de la red física. Los nodos azules y rojos corresponden a ubicaciones potenciales de CUs y DUs respectivamente.
Elaboración propia

virtuales. Estas dos entidades virtuales, CU y DU, son la evolución de lo que en tecnologías anteriores se conocía como "unidad de procesado banda base" o BBU, que implementaba todas las funcionalidades de las estaciones base para poder enviar solamente los datos a transmitir a la "unidad remota de radio" o RRH, que era la antena en sí, encargándose por tanto de funciones básicas de radio-frecuencia. Mediante la FS, una parte de las funciones de la BS se puede acercar a los emplazamientos remotos próximos a las antenas, mientras que otra parte se puede mantener centralizada a una gran distancia.

En resumen, la red física cuenta con nodos de procesamiento centralizados y otros remotos (cercanos a las antenas), sobre los cuales se pueden virtualizar CUs y DUs, respectivamente. Esta asignación se debe realizar por parejas, es decir, que cada DU requerirá de una CU, pero las CUs conviene virtualizarlas de la forma más centralizada posible sobre los nodos físicos correspondientes, mientras que las DUs deben estar próximas a la unidad de radiofrecuencia vinculada en un nodo remoto. En la Figura 2.1 se muestra una representación de la red física mencionada, con dos tipos de nodos separados por colores.

Teniendo en cuenta que la topología de la red 5G y los elementos que la componen son muy complejos, y que esta explicación sólo pretende esbozar los conceptos más relevantes respecto a este trabajo, se proponen las siguientes referencias para complementar la información ya aportada: [7], [8].

2.2. Virtualización de redes

En la sección anterior se ha hablado repetidamente de "virtualización de funciones de red" y "virtualización de nodos", pero este concepto de "virtualización" o "*embedding*" (a pesar de que el término no significa exactamente lo mismo se utiliza en este contexto para referirse a la virtualización) no se ha explicado de manera detallada hasta ahora. Virtualizar un elemento o función de una red significa utilizar un *hardware* no específico para realizar todas las operaciones, cálculos y cómputo de ese elemento o función, generalmente aprovechando sólo una fracción de la capacidad computacional de dicho *hardware*. Esto significa que sobre ese mismo elemento físico se pueden instanciar distintos elementos virtuales de forma independiente, y cada uno hará uso de una parte de los recursos de tiempo, cómputo, almacenamiento y comunicaciones disponibles.

En el caso de las redes 5G explicadas en la sección anterior, en uno de los nodo azules en la Figura 2.1, se pueden desplegar múltiples CUs virtualizadas, de forma que diferentes DUs se comunican con el

mismo nodo físico para realizar funciones de red de forma centralizada.

Al mismo tiempo, esta virtualización se puede realizar a distintos niveles. Si se imagina el sistema a virtualizar como una cadena de procesos a seguir, al virtualizar sólo una parte del sistema se estaría cortando dicha cadena por un punto determinado, de forma que una de las partes se lleva a cabo dentro de, por ejemplo, un ordenador central con una potencia de cálculo muy alta, y la otra parte se realiza en el punto de despliegue con un *hardware* específico diseñado a medida, y que al requerir menos potencia de cálculo se ve muy simplificado.

Este ejemplo sirve para ilustrar que un sistema puede ser virtualizado total o parcialmente, y que dependiendo de la aplicación el balance puede variar de forma considerable. En el caso específico de las redes 5G, ambas partes del sistema se encuentran virtualizadas, pero en dispositivos *hardware* distintos con localizaciones diferentes, lo cual hace que el punto donde "cortar la cadena" de procesos pueda fijarse a diversos niveles.

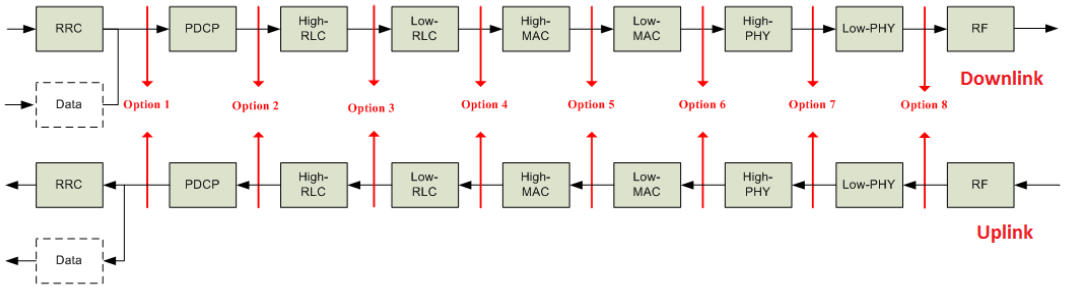


Figura 2.2: Posibles separaciones funcionales entre CU y DU. [9]

Estos niveles son configuraciones bien definidas en 5G [9], y la FS explicada previamente no se realiza de cualquier manera. Estas opciones de separación funcional se muestran en la Figura 2.2. Como se puede observar, los puntos de corte se encuentran en la separación de los protocolos de comunicaciones que forman parte de la BS, así como en algunos puntos internos a estos protocolos. Cada uno de estos puntos de corte o FS presenta ventajas e inconvenientes, tal como se explica en [10], y su idoneidad dependerá de las preferencias y necesidades del operador de red.

Si bien las posibles configuraciones están bien definidas, no se especifican los requisitos de capacidad de cómputo, así que en este trabajo se han utilizado las mismas estimaciones de estos valores que las utilizadas en [11, Tabla 4.3].

2.3. *Machine Learning* y Aprendizaje Reforzado

ML es un término que se refiere a todas aquellas soluciones a problemas reales que hacen uso de estrategias matemáticas diversas para conseguir que un programa sea capaz de aprender una función o un algoritmo desconocido. El ML resuelve problemas para los cuales no es posible diseñar una solución, bien sea por la complejidad de esta misma o por la del problema en sí. Es por esto que se contemplan programas y sistemas capaces de realizar tareas extremadamente complejas, y que tienen en cuenta muchas variables diferentes a la vez, lo que lo sitúa dentro de lo que se conoce como **Inteligencia Artificial**.

Un sistema de ML requiere de un método de aprendizaje que le permita extraer información abstracta

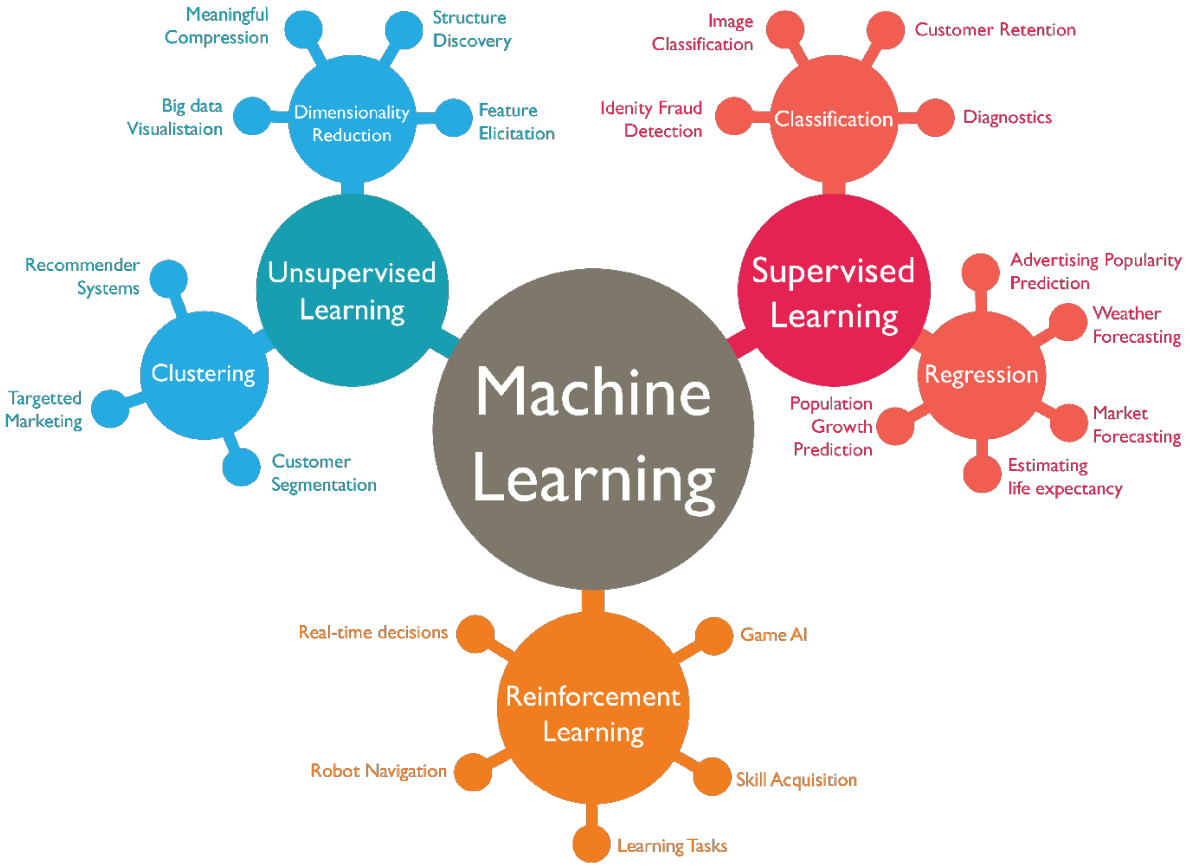


Figura 2.3: Paradigmas de ML y algunas de sus aplicaciones. [12]

a través de la práctica o del ensayo y error. Este aprendizaje se puede realizar de diversas maneras, existiendo distintos algoritmos, divididos en tres categorías principales:

- Aprendizaje supervisado: requiere de un conjunto de datos previamente etiquetados correctamente. Se compararán los resultados obtenidos con los resultados esperados para aprender las características de cada etiqueta y poder luego etiquetar nuevos datos no vistos previamente.
- Aprendizaje no supervisado: requiere de un conjunto de datos sin etiquetar, y se aprende a agrupar dichos datos según relaciones o patrones previamente desconocidos, de forma que se generen *clusters* con los que se puedan clasificar datos no vistos previamente.
- Aprendizaje reforzado o RL: difiere bastante de los otros dos paradigmas de aprendizaje, ya que en vez de contar con un conjunto de datos para el aprendizaje, busca maximizar una función de recompensa obtenida al hacer a un agente tomar acciones sobre un entorno y modificar su estado.

En la Figura 2.3 se pueden observar estos tres paradigmas con algunas de sus aplicaciones más relevantes.

En este trabajo se emplea un algoritmo de aprendizaje reforzado, ya que encaja perfectamente con el problema a resolver. A continuación se ofrece una explicación sobre los conceptos principales necesarios para entender este tipo de aprendizaje, especialmente enfocada a presentar la terminología y elementos clave que conforman el núcleo del *software* desarrollado. En el libro [13] se puede encontrar

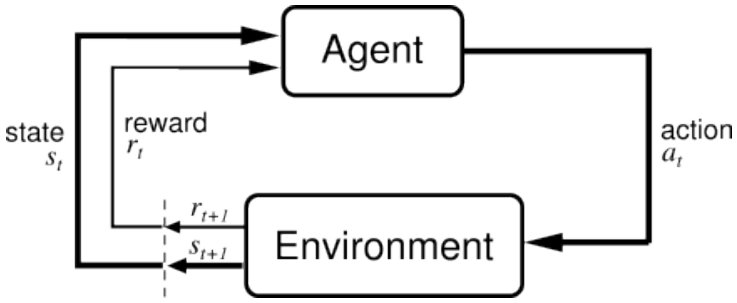


Figura 2.4: Esquema básico del sistema agente-entorno [13]

más información al respecto, y en el artículo [14] se explican también los conceptos básicos, incluyendo los que se incluyen posteriormente en el documento.

En el contexto del aprendizaje reforzado se trabaja con un sistema agente-entorno, siendo estos dos elementos que interactúan entre sí de forma iterativa hasta alcanzar una condición de parada. Un esquema de este sistema se muestra en la Figura 2.4.

El agente es el elemento que se encarga de, en función del estado actual del entorno, tomar una acción determinada (o generar una distribución de probabilidad de las posibles acciones a tomar) siguiendo una política o *policy*. El entorno es el elemento que se encarga de llevar a cabo las acciones indicadas por el agente, actualizando su estado, y calculando una recompensa o *reward* correspondiente. El nuevo estado se utiliza para realizar una nueva iteración, y el *reward* se utilizará para ajustar la *policy*. El ciclo se detiene cuando el entorno detecte que se cumple la condición de parada, específica para cada situación. Cabe indicar que en el caso planteado en este trabajo el *reward* estaría sujeto a las preferencias y necesidades del operador de red, de lo que se deriva la necesidad de entornos de aprendizaje de la *policy* a partir de diferentes *rewards*.

La *policy* definida por el agente, la función que establece qué acción tomar en función del estado de la red, será en este caso una red neuronal que se entrenará con un algoritmo determinado. Hay otros tipos de *policies*, pero se salen del marco de este trabajo.

Existen diversos algoritmos para realizar el entrenamiento de la red neuronal, y constantemente se investigan y proponen mejoras y versiones más eficientes considerando diversos criterios. Al ser un área tan amplia y en constante evolución, se ha optado por utilizar un algoritmo muy probado y utilizado, a pesar de no ser el más reciente o eficiente: REINFORCE [12]. Para conocer mejor la clasificación de los algoritmos de aprendizaje utilizados en RL y sus diferencias, se puede consultar el artículo [15].

REINFORCE es un algoritmo con diferentes variantes, pero se va a utilizar su versión más básica. No se va a profundizar mucho en el funcionamiento del algoritmo, sino que solamente se comentarán los aspectos prácticos de mayor relevancia. Una explicación más detallada sobre REINFORCE se puede encontrar en el libro [13] y el artículo [12].

En resumen, se busca entrenar un agente (una red neuronal) para que aprenda a realizar acciones (virtualizar nodos) sobre un entorno (la red sustrato). La decisión de qué acción tomar (qué nodos físicos asignar a qué nodos virtuales) produce un cambio en el estado del entorno y, para cada nuevo estado, se calcula un valor de recompensa mediante una función que toma en cuenta unos parámetros predefinidos (preferencias del operador). Con este valor de recompensa se ejecutará un algoritmo de aprendizaje

(REINFORCE) en el agente.

2.4. Redes neuronales

Así como hay diversos paradigmas de aprendizaje dentro de los cuales hay a su vez diversos algoritmos, los "elementos que aprenden" también cuentan con una gran variedad. Pero definitivamente uno de los más populares son las **redes neuronales**, que están cobrando cada vez más relevancia, no sólo entre las comunidades investigadoras de innumerables campos de conocimiento, sino también en el conocimiento popular.

No se va a profundizar mucho en las redes neuronales, ya que hay una enorme cantidad de información que contar al respecto. Solamente se van a exponer los conceptos aplicados a este trabajo, pero en el libro [16] hay mucha más información sobre todo tipo de ramas de ML, incluyendo las redes neuronales.

El concepto básico de una red neuronal o NN es relativamente sencillo. La idea inicial consiste en un conjunto de "neuronas" que realizan una combinación lineal de una serie de entradas, multiplicadas por una serie de pesos. Esto produce una única salida, la cual se pasa por una función de activación para generar la salida final de la neurona, y poder transmitirla a las siguientes. La Figura 2.5 muestra el funcionamiento básico de una neurona clásica o "perceptrón".

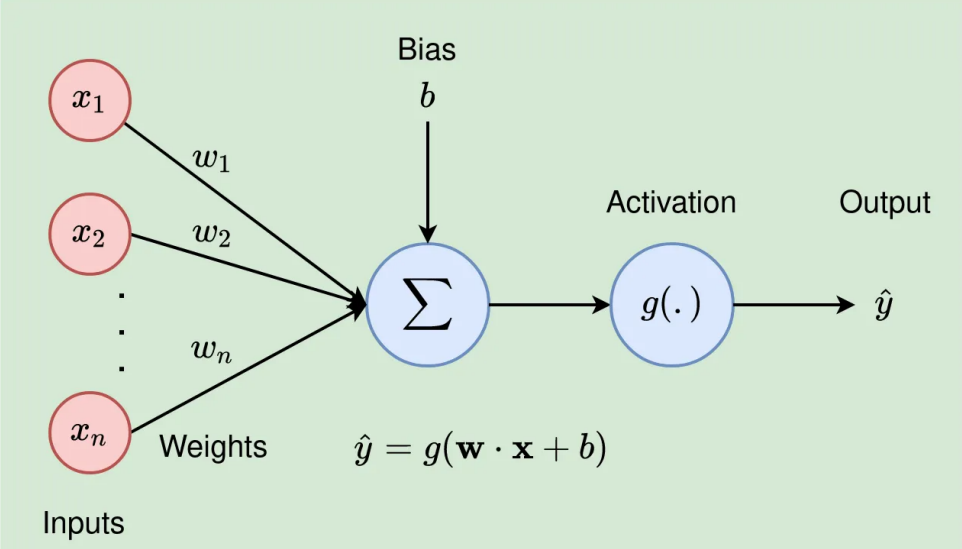


Figura 2.5: Neurona de una NN [17]

Evidentemente esta descripción es únicamente el comienzo, pero la idea principal es la de procesar y combinar un conjunto de entradas para producir un resultado. En la práctica no se suele trabajar con neuronas individuales, sino con operaciones matriciales que se encargan de realizar las multiplicaciones y sumas en bloque. Es decir, el nivel de abstracción se sitúa en la capa de la NN y no en la neurona.

Estas capas, además, pueden realizar diferentes operaciones, no necesariamente multiplicaciones y sumas lineales. Este es el caso de las capas de una *Graph Convolutional Network* (GCN), un tipo de red que realiza una convolución sobre las características de los nodos de un grafo. El artículo [18] ofrece una muy buena introducción a la teoría de grafos y a las GCN, y el artículo [19] no sólo incluye una explicación amplia de las GCNs sino también enumera un conjunto de herramientas web interactivas, que

pueden utilizarse para ayudar a entenderlas.

Un grafo cuenta con nodos y enlaces entre los nodos. Podemos considerar, por ejemplo, que los nodos tienen una característica (como su capacidad de cómputo disponible), y que los enlaces son bidireccionales (grafo no dirigido). Con este escenario, una convolución de grafo consiste en difundir la característica de cada nodo hacia sus nodos vecinos, influyendo en la de estos.

Tras la convolución, la característica de cada nodo será igual a su propia característica inicial más todas las de sus vecinos multiplicadas por una serie de pesos, que determinan la influencia de cada vecino. Son precisamente estos pesos los que la red neuronal aprende a ajustar. En la Figura 2.6 se muestra esta operación de convolución sobre dos nodos distintos de un grafo. Se ve cómo la propiedad de un nodo se propaga por el grafo capa tras capa. La operación completa implicaría realizar esa propagación sobre todos los nodos del grafo simultáneamente.

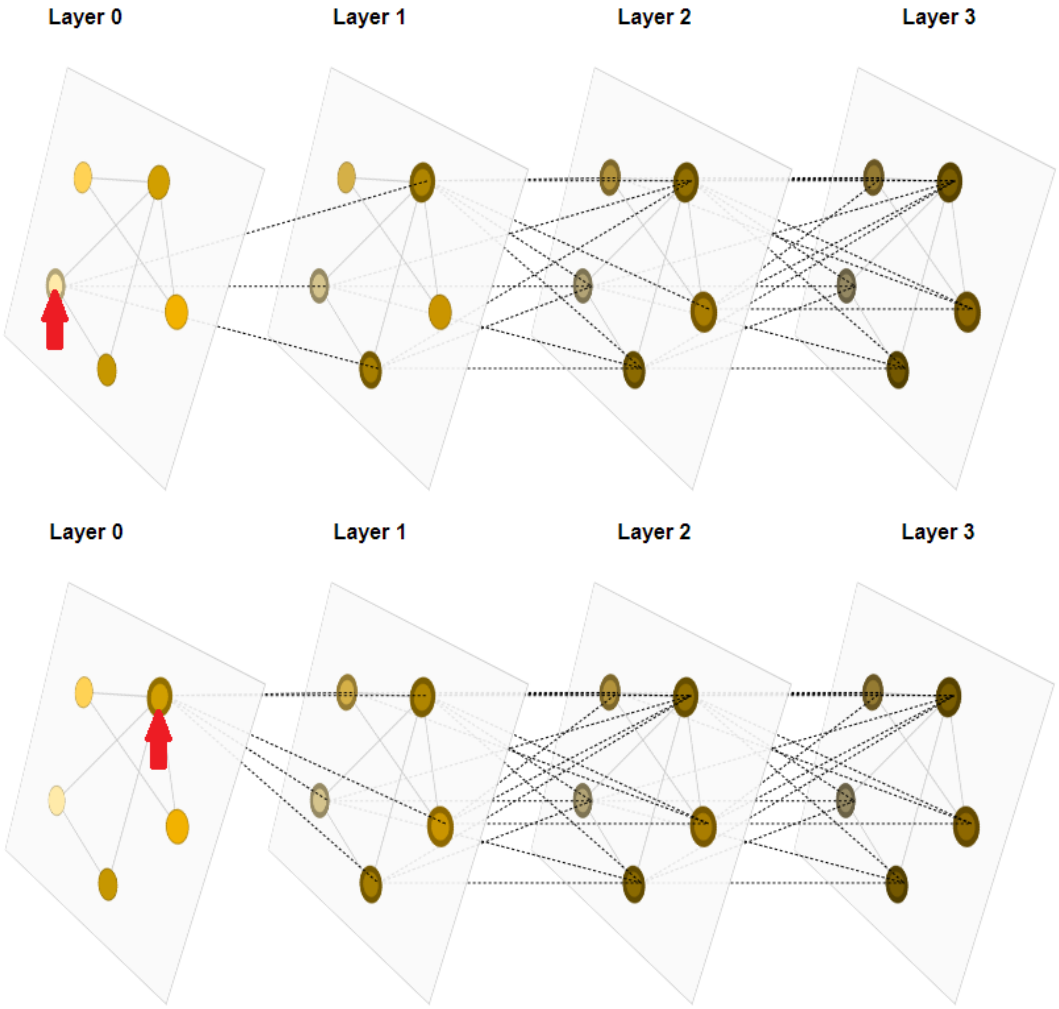


Figura 2.6: Visualización de la operación de convolución [18]

De forma algo más rigurosa, podemos expresar la operación realizada por la GCN con la Ecuación 2.1:

$$Y = [(D^{-0,5} \cdot \hat{A} \cdot D^{-0,5}) *_{\cdot} W] \cdot X \tag{2.1}$$

Los símbolos utilizados en esta ecuación se encuentran descritos en la Tabla 2.1. No se va a

profundizar en el significado de cada uno de ellos, ya que requeriría de una explicación sobre teoría y tratamiento de grafos algo más extensa, pero en los artículos mencionados se puede encontrar en más detalle el desarrollo matemático que lleva a obtener esa ecuación.

Tabla 2.1: Elementos de la Ecuación 2.1

Símbolo	**Definición**
$\hat{A}$	matriz de adyacencia + matriz identidad ($A + I$)
A	matriz de adyacencia del grafo
D	matriz diagonal de grados de los nodos
W	matriz de pesos
X	vector o vectores columna de las características de los nodos a convolucionar
Y	vector o vectores columna de las características de los nodos convolucionadas

Este tipo de redes neuronales se utilizan en conjunto con otros tipos de capas, como las capas clásicas explicadas anteriormente, las cuales se conocen como *Dense Layer* (DL) o capas densamente conectadas, para realizar predicciones de distintos tipos sobre los nodos, los enlaces o las características de nodos y enlaces del grafo.

Para concluir la explicación sobre redes neuronales, se incluyen la ecuación y la tabla de símbolos correspondientes a la operación que realizan las DL.

$$Y = X \cdot K + B \tag{2.2}$$

Tabla 2.2: Elementos de la Ecuación 2.2

Símbolo	**Definición**
B	*bias* o matriz de pesos que se suman a las entradas
K	*kernel* o matriz de pesos que multiplican a las entradas
X	vector fila de entrada
Y	vector fila de salida

Capítulo 3

Diseño

En este capítulo se presenta el diseño de la solución implementada, dividido en tres secciones: funcionalidad, estructura y función de *reward*.

En la primera sección se explica la funcionalidad que se asume para el planteamiento del problema de *embedding* que se trata de solucionar, y las consideraciones que se han tenido en cuenta para cada uno de los elementos de red. En la segunda sección se presenta la arquitectura general de la solución implementada, indicando sus componentes y la funcionalidad de los mismos, así como la relación entre ellos. Los detalles de la implementación *software* se presentarán en el Capítulo 4. Posteriormente, en la tercera sección se detallará la función de *reward* implementada que modela la preferencia del gestor de la red y que es utilizada para el entrenamiento del sistema.

3.1. Funcionalidad

Como se indica en los objetivos del trabajo, la funcionalidad del *software* generado debe ser la de entrenar una red neuronal para realizar la tarea de *embedding* de una red virtual sobre una red sustrato. Esto es difícil de entender sin primero conocer más en profundidad el problema específico a resolver.

En el Capítulo 2 se ha explicado el concepto de virtualización y su relación con las redes 5G. El problema que se plantea ahora es en qué nodos físicos realizar la virtualización de CUs y DUs. Para ello se han definido unos parámetros de calidad sencillos, pero que permiten diseñar posteriormente una función de *reward* adecuada. De forma genérica, el área sobre el que se trabaja se divide en zonas, y estas en sub-zonas. De este modo, cuando se despliegan estaciones base virtuales, se busca que den cobertura en una zona/sub-zona determinada, donde se despliega un servicio. Asimismo, las entidades virtuales (CU y DU) tienen unos requisitos de cómputo, y los nodos físicos tienen capacidades. Con todo, se tienen en cuenta las siguientes consideraciones:

- Ningún nodo virtual podrá ser desplegado sobre un nodo físico cuya capacidad disponible sea inferior a la demanda del nodo virtual.
- Todos los nodos virtuales deben desplegarse sobre nodos físicos del tipo adecuado. Si un nodo físico puede albergar CUs, ninguna DU podrá ser virtualizada en él y viceversa.
- Las CUs deben estar virtualizadas, preferentemente, sobre nodos físico de su misma zona geográfica.

- Las DUs deben estar virtualizadas, preferentemente, sobre nodos físicos de su misma sub-zona o, alternativamente, sobre nodos de su misma zona. Nunca deberán ser virtualizadas sobre nodos de una zona diferente, ya que se daría cobertura lejos de donde se busca desplegar el servicio.
- Las CUs deben estar virtualizadas de la forma más centralizada posible, es decir, repartirse lo menos posible entre los nodos físicos.

La formulación matemática de estas consideraciones se detallará en la Sección 3.3, donde se explica la implementación de la función de *reward*. La idea de este trabajo es intentar que estas decisiones de qué nodos físicos asignar a qué nodos virtuales queden en manos de un sistema de IA, más concretamente en una NN.

Es importante también entender que se ha decidido trabajar solamente con las características de los nodos, sin considerar costes o retardos en los enlaces. Se asume, por tanto, que los enlaces cuentan con una capacidad suficiente para soportar todo el tráfico que vaya a circular por ellos.

Siguiendo la lógica agente-entorno presentada en el Capítulo 2, se ha definido que la red neuronal debe ser capaz de tomar una petición de virtualización y el estado de red física para producir una acción que el entorno evaluará. A continuación se detallan los formatos en los que se deben encontrar estos tres datos (petición, estado y acción).

La petición consistirá en una pareja CU-DU definidas por los requisitos computacionales de cada elemento, y la zona y sub-zona geográficas a las que la estación base debe dar servicio.

El estado de la red física se representa mediante una matriz con una fila por nodo y cinco columnas que contienen los siguientes datos:

1. Capacidad disponible actualmente en el nodo
2. Número de nodos ya virtualizados dentro del nodo físico
3. Zona geográfica a la que pertenece el nodo
4. Subzona geográfica a la que pertenece el nodo
5. Tipo de nodo (CU o DU) que puede albergar el nodo físico

La decisión tomada, es decir, la salida de la red neuronal, consistirá en dos números que indicarán los índices de los nodos físicos asignados a la CU y a la DU respectivamente.

Con estos tres conjuntos de datos, el *software* debe implementar el modelo agente-entorno de RL para permitir que una NN entrene, de forma que aprenda a realizar la mejor asignación de nodos físicos a una petición de virtualización cualquiera sobre la red en un estado cualquiera.

3.2. Estructura

El sistema está encapsulado dentro de un elemento llamado *Training and Inference Manager* o *TIManager* para abreviar. El sistema dentro del TIManager está estructurado en distintos componentes que funcionan en conjunto, y van más allá de solamente un agente y un entorno, ya que para que estos dos

elementos interactúen entre sí se requieren diversos procesos adicionales realizados por otros bloques del sistema. En la Figura 3.1 se presenta un esquema completo del sistema, a partir del cual se explicarán los bloques mostrados.

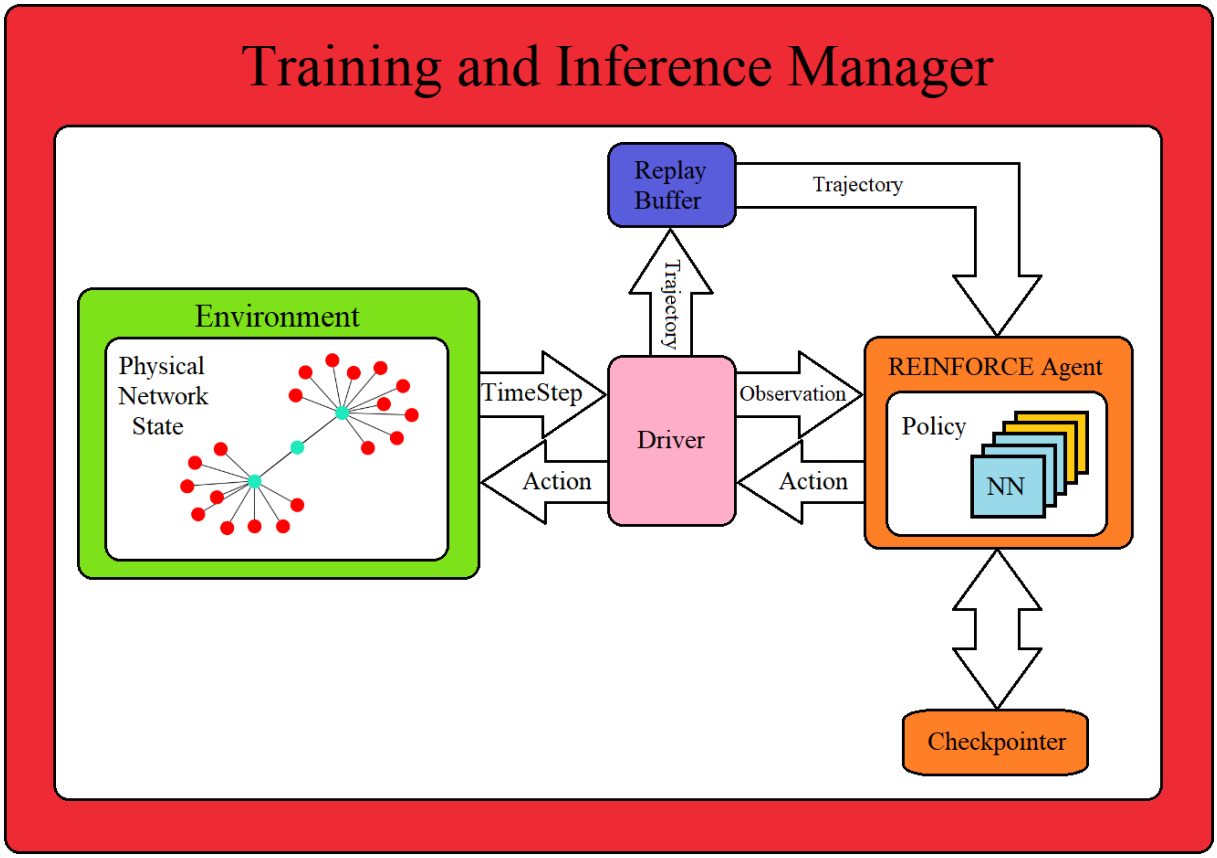


Figura 3.1: Esquema completo del *software* desarrollado
Elaboración propia

3.2.1. Training and Inference Manager (TIManager)

Este elemento es el que encapsula al sistema en su totalidad, y con él se controla el entrenamiento de la red neuronal, la realización de inferencias con la misma y la exportación del progreso de entrenamiento. Además de instanciar todos los elementos, también ofrece un fácil acceso a sus funciones más relevantes, permitiendo invocar las siguientes funcionalidades:

- Instanciar los elementos con parámetros ajustables por el usuario, entre los que se encuentra la posibilidad de importar un *checkpoint* del agente, y restaurarlo desde un directorio.
- Realizar un bucle de entrenamiento del agente con un número de iteraciones definido por el usuario.
- Exportar en ficheros un *checkpoint* del entrenamiento del agente.
- Realizar una inferencia sobre un conjunto de peticiones de *embedding* cualquiera y guardar el resultado en un fichero.
- Generar una representación gráfica de la red física con el *embedding* obtenido (número de nodos virtualizados en cada nodo físico).

3.2.2. Environment

Este elemento hace todas las funciones necesarias para implementar un entorno adecuado para RL, constituyendo uno de los dos componentes clave del aprendizaje reforzado. Además permite acceder al estado actual de la red física y a los resultados del *embedding*, pudiendo exportar y representar estos últimos. Este componente realiza las siguientes funciones:

- Importar y almacenar las características de los nodos de la red física en su estado actual.
- Importar y almacenar las peticiones de *embedding* de la red virtual, así como incluirlas en las observaciones.
- Almacenar las decisiones de *embedding* tomadas y posibilitar su exportación a un fichero.
- Generar una representación gráfica de la red física con el número actual de nodos virtualizados en cada nodo físico.
- Ofrecer una observación adecuada tras cada acción tomada (incluida en un objeto TimeStep) o bajo demanda.
- Ofrecer un objeto TimeStep adecuado tras cada acción tomada.
- Recibir acciones de asignación de nodos físicos para las peticiones de *embedding* extraídas de la red virtual, ofrecer un valor de *reward* para dichas acciones y llevarlas a cabo si son viables.

En el Capítulo 4 se detalla la implementación de las observaciones, peticiones y el concepto de "TimeStep".

3.2.3. REINFORCE Agent

Este elemento constituye el segundo componente clave del RL, ya que implementa el agente que decide qué acción tomar en base a una observación realizada por el entorno y siguiendo una *policy* basada en una red neuronal. Concretamente este agente realiza el entrenamiento de la red contenida en él mediante el algoritmo REINFORCE. El agente realiza las siguientes funciones:

- Contener la red neuronal que sirve como *policy*.
- Recibir observaciones generadas por el entorno para ofrecer una acción adecuada calculada mediante la *policy*.
- Entrenar la red neuronal mediante REINFORCE en base a la experiencia recolectada en forma de trayectoria o Trajectory.

Aunque pueda parecer que el agente realiza pocas funciones, permite realizar el entrenamiento de la red como una caja negra en la que entran trayectorias y la red se ajusta "automáticamente". Esto es gracias a que el agente implementa todos los algoritmos necesarios para que esto sea posible.

3.2.4. Neural Network

La red neuronal es el elemento alrededor del cual gira casi todo el sistema, ya que el propósito de casi todos los elementos es permitir y facilitar el entrenamiento y el uso de esta red neuronal para realizar la tarea de *embedding*. Esta red sigue una estructura secuencial, es decir, que las capas se sitúan una a continuación de otra y los datos de entrada atraviesan todas las capas en orden.

Para la implementación de la red neuronal se han utilizado dos tipos de capas. Las primeras consisten en una GCN que se encarga de difundir dos de las características de los nodos de la red física. Tal y como se mencionó en la Sección 3.1, los nodos de la red física se definen por cinco valores: capacidad, nodos virtualizados, zona, subzona y tipo. De estas cinco características, solamente la capacidad actualmente disponible y el número de nodos actualmente virtualizados son sometidos a la convolución de esta GCN. El resto de datos de la observación atraviesan sin cambio alguno estas capas.

Tras ellas se encuentra una serie de capas densamente conectadas (Dense), que condensan todos los valores contenidos en la observación (con dos columnas convolucionadas por la GCN) en dos únicos valores enteros, que representan la acción escogida.

El número de capas de la GCN, el grado de la operación de convolución de cada capa, y el número de capas Dense así como el número de neuronas de cada una, son parámetros totalmente configurables por el usuario del TIManager. En su conjunto, la red solamente realiza una función, que es la de aplicar las operaciones de cada capa a los datos de entrada, que pasan de capa en capa y producen unos datos de salida.

3.2.5. Checkpointer

El Checkpointer es un elemento conceptualmente muy simple, que se encarga de exportar o importar el agente con la red neuronal entrenada, de forma que tras un periodo de entrenamiento se pueda guardar el progreso para restaurarlo en cualquier otro momento, aunque se detenga la ejecución del sistema. Este elemento realiza solamente estas dos funciones:

- Exportar bajo demanda el agente y su *policy* en su estado actual en ficheros.
- Importar bajo demanda el agente y su *policy* desde ficheros.

3.2.6. Driver

Este elemento se encarga de gestionar la interacción agente-entorno para la generación de un "episodio" completo. Esto significa que recoge los TimeSteps ofrecidos por el Environment, le pasa las observaciones al Agent y recoge la acción generada, se la pasa al Environment y repite el ciclo hasta que el entorno indique que se han agotado las peticiones de virtualización. Todos los intercambios desde el estado inicial hasta la finalización del *embedding* conforman un episodio. Durante todo este proceso el Driver se puede comunicar, mediante un observador u *observer*, con un *buffer* que almacena todos estos intercambios de datos. Las funciones que realiza el Driver son:

- Recolectar TimeSteps provenientes del Environment.
- Recolectar acciones provenientes del Agent.

- Solicitar al Environment la toma de una acción determinada.
- Solicitar al Agent la generación de acciones en función de observaciones determinadas.
- Llamar a los *observers* indicados cada vez que se realice una interacción entre el agente y el entorno.

3.2.7. Replay Buffer

Este elemento se corresponde, como su nombre indica, con un búfer. Es decir, almacena información para acceder a ella posteriormente. En este caso almacena todas las acciones que se le pasan al Environment y los TimeSteps que este devuelve, todo ello a través del Driver que gestiona esta interacción y notifica al Buffer de estos eventos. Estos intercambios de acciones y TimeSteps se van acumulando en una Trajectory que posteriormente, al finalizar el episodio, será enviada al agente para que realice el entrenamiento de la red neuronal. Las funciones que realiza el Replay Buffer son:

- Recibir, formatear y almacenar la trayectoria que genera el Driver.
- Ofrecer la experiencia recolectada en la trayectoria.

3.3. Función de *reward*

Como se ha mencionado en la sección anterior, una de las funciones del Environment es calcular el *reward* correspondiente a cada acción tomada, y como se ha mencionado en la Sección 3.1 este *reward* toma en cuenta unos criterios de calidad determinados.

Aunque no se haya profundizado mucho todavía en ellas, las funciones de *reward* son una pieza fundamental de los sistemas de RL, ya que una función capaz de evaluar las decisiones tomadas y el estado de la red de forma numérica es vital para poder indicarle al agente si sus decisiones son más o menos acertadas.

En este trabajo se propone una función de *reward* relativamente sencilla, de forma que permita comprobar el funcionamiento del sistema. Esta función se podría extender para tener en cuenta requisitos e indicadores de calidad específicos y ajustados a las necesidades del operador y de los servicios desplegados. En concreto, la función implementada tiene en cuenta los criterios mencionados en la Sección 3.1:

- Capacidad disponible y requerida de nodos físicos y virtuales respectivamente
- Tipo de nodos físicos y virtuales
- Zona geográfica de las CUs
- Zona y subzona geográficas de las DUs
- Grado de centralización de las CUs

De estos criterios se extraen unos requisitos que, de no cumplirse, harían que la función diera un resultado negativo. En caso de que los requisitos se cumplan, se evaluarían tres parámetros de calidad, que definen una salida positiva de la función.

Las variables de entrada que maneja la función de *reward* son dos: la acción tomada y el estado de la red. Ya se han descrito estas variables en la Sección 3.1, pero en este apartado se definirán matemáticamente. Se presentan también las expresiones matemáticas que definen esta función de *reward*. Todos los símbolos necesarios para entenderlas se encuentran en la Tabla 3.1.

La acción se define como la dupla (i_0, i_1) donde $i_0, i_1 \in [0, |N|]$, es decir, que ambos son enteros entre 0 y el número de nodos de la red física. De esta manera, el cero se corresponde con un valor no válido, que sirve para filtrar las salidas negativas de la red neuronal, que son convertidas a cero junto con los resultados menores que 1. El resto de valores se consideran válidos según este primer filtrado. De esta manera, se genera un requisito adicional que no se extrae de los criterios de calidad iniciales, sino de la definición de la propia acción.

Las observaciones del estado de la red física que realiza el entorno no son más que una matriz cuyas filas son los nodos. La definición de un nodo es la siguiente: $n_i = [c_i, v_i, Z_i, z_i, t_i]$. Las características a su vez se definen como:

- Capacidad: $c_i \in \mathbb{Q} \geq 0$
- #Nodos virtualizados: $v_i \in \mathbb{N}$
- Zona y subzona geográficas: $Z_i, z_i \in \mathbb{Z}$
- Tipo de nodo: $t_i = \begin{cases} 0 & \text{si } n_i \in N_{du} \\ 1 & \text{si } n_i \in N_{cu} \end{cases}$

El *reward* se calcula a partir de seis variables intermedias $x_0, x_1, x_2, x_3, x_4, x_5$. Cada una de estas variables tiene en cuenta uno de los criterios ya explicados.

- x_0 (3.1) comprueba si los valores de la acción son válidos.
- x_1 (3.2) comprueba si las capacidades de los nodos físicos asignados son suficientes.
- x_2 (3.3) comprueba si los tipos de los nodos físicos asignados son correctos.
- x_3 (3.4) evalúa la zona del nodo físico asignado a la CU.
- x_4 (3.5) evalúa la zona y la sub-zona del nodo físico asignado a la DU.
- x_5 (3.6) evalúa el grado de centralización de las CUs.

Una vez obtenidas las seis variables, se combinan para obtener el resultado final de la función. Esto se realiza en dos pasos. El primero es comprobar si alguno de los requisitos no se está cumpliendo, para lo cual hay que detectar si alguna de las variables intermedias es negativa. Una vez comprobado, se hace una ponderación de las tres variables cuyos valores son mayores o iguales a cero. La expresión final se muestra en la ecuación (3.7).

$$x_0 = \begin{cases} -1 & \text{si } i_0 = 0 \text{ and } i_1 = 0 \\ -0{,}666 & \text{si } i_0 = 0 \text{ xor } i_1 = 0 \\ 0 & \text{en otro caso} \end{cases} \tag{3.1}$$

Tabla 3.1: Símbolos utilizados en la función de *reward*

Símbolo	**Definición**
	Conjuntos
N	Conjunto de todos los nodos físicos
N_{cu}	Subconjunto de los nodos físicos capaces de albergar CUs
N_{du}	Subconjunto de los nodos físicos capaces de albergar DUs
P	Conjunto de todos los nodos físicos
	Variables
n_i	Nodo físico con índice i
i_0, i_1	Índices de los nodos físicos asignados a la CU y la DU respectivamente
c_i	Capacidad del nodo físico n_i
c_{cu}, c_{du}	Capacidades requeridas por la CU y la DU respectivamente
t_i	Tipo del nodo físico n_i
v_i	Número de nodos virtualizados en el nodo físico n_i
Z_i	Zona geográfica del nodo físico n_i
Z_{cu}, Z_{du}	Zonas geográficas de la CU y la DU respectivamente
z_i	Subzona geográfica del nodo físico n_i
z_{du}	Subzona geográfica de la DU
t_i	Tipo de nodo del nodo n_i
$\omega_1, \omega_2, \omega_3$	Pesos utilizados en el cálculo del *reward*

$$x_1 = \begin{cases} -0,5 & \text{si } c_{i0} < c_{cu} \text{ or } n_{i1} < c_{du} \\ 0 & \text{en otro caso} \end{cases} \tag{3.2}$$

$$x_2 = \begin{cases} -0,5 & \text{si } n_{i0} \notin N_{cu} \text{ or } n_{i1} \notin N_{du} \\ 0 & \text{en otro caso} \end{cases} \tag{3.3}$$

$$x_3 = \begin{cases} 1 & \text{si } Z_{i0} = Z_{cu} \\ 0 & \text{en otro caso} \end{cases} \tag{3.4}$$

$$x_4 = \begin{cases} 1 & \text{si } Z_{i1} = Z_{du} \text{ and } z_{i1} = z_{du} \\ 0,5 & \text{si } Z_{i1} = Z_{du} \text{ and } z_{i1} \neq z_{du} \\ -0,333 & \text{en otro caso} \end{cases} \tag{3.5}$$

$$x_5 = \frac{\sqrt{\sum_{n_i \in N_{cu}} v_i^2}}{\sum_{n_i \in N_{cu}} v_i} \tag{3.6}$$

$$reward = \begin{cases} min(x_0, x_1, x_2, x_4) & \text{si } min(x_0, x_1, x_2, x_4) < 0 \\ \omega_1 \cdot x_3 + \omega_2 \cdot x_4 + \omega_3 \cdot x_5 & \text{en otro caso} \end{cases} \tag{3.7}$$

donde $\omega_1, \omega_2, \omega_3$ son pesos para dar más prioridad a algunos de los factores, de tal forma que $\sum_i \omega_i = 1$.

Con todo la función de *reward* de cada asignación está acotada entre -1 y 1.

Capítulo 4

Implementación y validación

En este capítulo se detalla cómo se traduce el diseño propuesto en el Capítulo 3 a código Python, utilizando distintas librerías conocidas en el ámbito del ML, como TensorFlow y NumPy.

Este capítulo está orientado a entender en mayor profundidad el código desarrollado, y por tanto presenta un contenido más técnico en el área del desarrollo de software. Por no sobrecargar esta memoria con contenidos teóricos, no se van a explicar aspectos como el funcionamiento de la programación orientada a objetos o el uso de tensores, ya que sería muy extenso. Por este motivo, para entender este capítulo sería interesante disponer de conocimientos previos de programación y conceptos como declaración o instanciación de una clase, herencia, *callable objects*, forma de un tensor, dimensiones externas e internas de un tensor, etc.

Las librerías de código externas requeridas son las siguientes:

- TensorFlow
- TFAgents
- NumPy
- StellarGraph
- NetworkX

En la Sección 4.1 se describe la implementación desarrollada, así como su relación con las librerías adoptadas. En las Sección 4.2 se muestra cómo deben ser los archivos que contienen los datos de entrada que serán importados. Tras esta descripción, en la Sección 4.3 se explica cómo ejecutar el sistema para utilizarlo y en la Sección 4.4 se presentan los resultados de dicha ejecución.

4.1. Estructura del código

La estructura del código es análoga al esquema mostrado en la Figura 3.1, ya que cada bloque del esquema se implementa con una clase específica, resultando en el diagrama de clases de la Figura 4.1. En dicha Figura, todas las clases implementadas en este trabajo aparecen solamente con su nombre, mientras que aquellas clases extraídas de librerías contienen el nombre de la librería correspondiente antes del

nombre de la clase. Como se puede observar, la mayoría de clases forman parte de alguna librería de TensorFlow, o heredan de una clase que sí lo hace. En este sentido, la documentación de TensorFlow es el mejor apoyo para entender el funcionamiento de las clases y módulos utilizados.

En la Figura 4.1b se han dibujado, sobre el diagrama de clases, líneas de colores que relacionan dicho diagrama con la Figura 3.1 del Capítulo 3, de forma que se pueda observar visualmente qué bloques del sistema corresponden a qué clases. A continuación se comentan las clases utilizadas y se detallan las funciones más importantes de aquellas que se han desarrollado en este trabajo. También se indican los anexos donde se puede encontrar el código generado de cada clase.

4.1.1. GNCLayer y DenseLayer - Anexos F y G

Ambas clases heredan de la clase base Layer, ofrecida por la librería Keras de TensorFlow. Esto implica que deben implementar las siguientes funciones:

- `build(input_shape)`: Esta función le indica al objeto la forma que tendrán los datos de entrada, de manera que se puedan generar los pesos de esa capa. En el caso de GCNLayer, la inicialización se hace al instanciar el objeto, y por tanto esta función no tiene ningún efecto. Los objetos DenseLayer se pueden instanciar sin conocer previamente la forma de la entrada, así que esta función sí que genera los pesos de la red.

- `call(input) -> Tensor`: Esta función permite hacer llamadas al objeto para que aplique la operación de la capa neuronal correspondiente sobre unos datos de entrada, cuya forma debe haberse definido de antemano, bien sea al instanciar el objeto o mediante la función `build(input_shape)`. Esta función retorna el resultado de aplicar a las entradas la operación de la capa neuronal correspondiente.

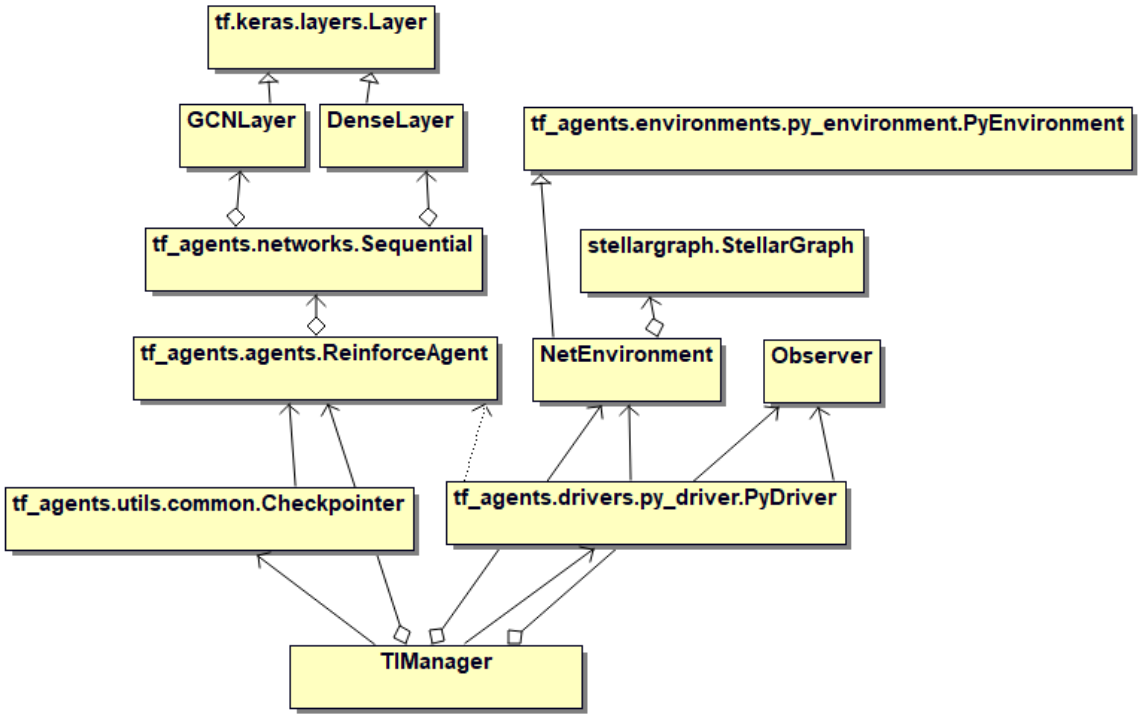
Generalmente los datos de entrada de los objetos Layer de Keras están pensados para tener un tamaño fijo que no puede ser cambiado una vez definido, y esto aplica para todas las dimensiones de la matriz de entrada. Sin embargo, diversas funciones y clases de la librería utilizan distinto número de dimensiones externas de tamaño 1 para encapsular los datos y, por tanto, se han tenido que adaptar las dos clases desarrolladas para poder gestionar datos cuyo número de dimensiones exteriores varía entre uno y dos. El tamaño de los datos propiamente dichos sigue siendo constante, porque de lo contrario no coincidirían con las formas de los pesos.

Los objetos instanciados de estas clases serán agrupados mediante la clase Sequential de TensorFlow, que representa la red neuronal en su conjunto, con todas las capas situadas una detrás de otra en orden.

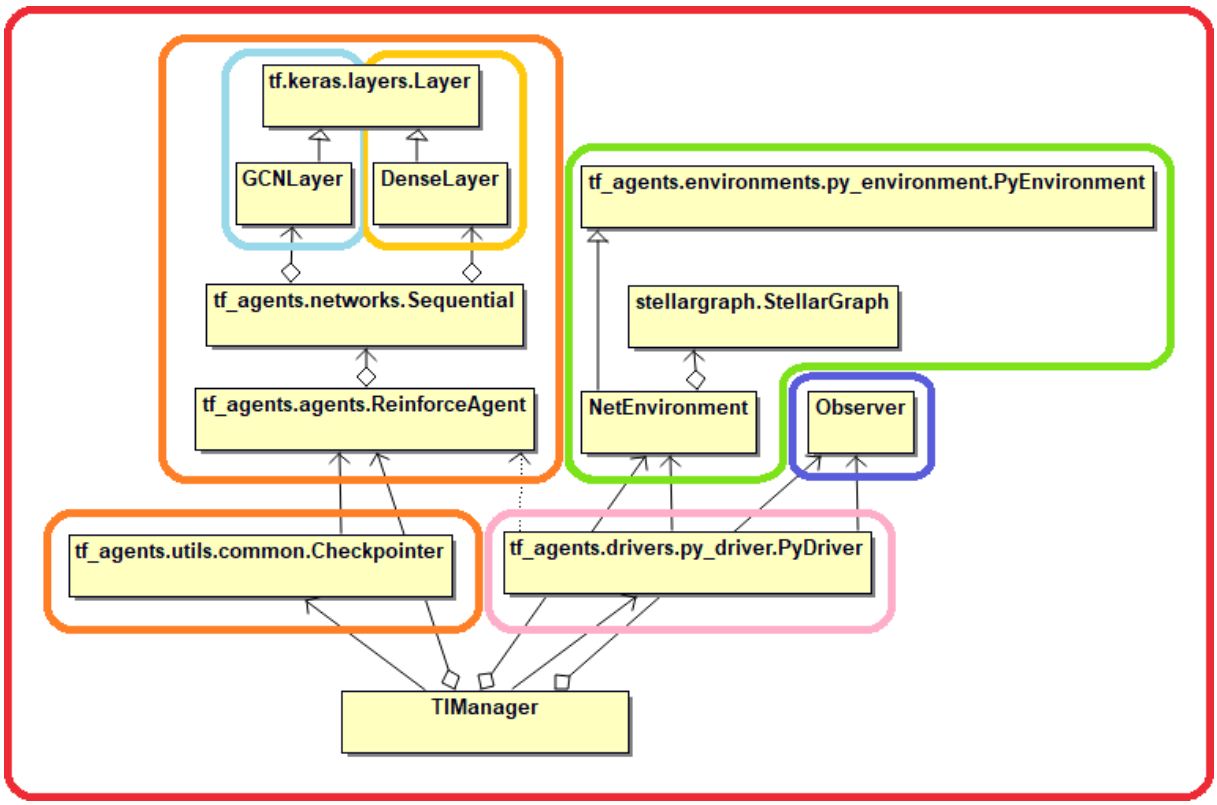
4.1.2. NetEnvironment - Anexo F

Esta clase implementa un Environment de TensorFlow Agents, más concretamente un PyEnvironment. Esto implica que debe implementar, al menos, las siguientes funciones:

- `_reset() -> TimeStep`: Esta función reinicializa las variables del entorno y reimporta las redes física y virtual, para luego devolver un TimeStep que representa el estado inicial del entorno.



(a) Diagrama de clases del sistema implementado



(b) Diagrama de clases delineado con los colores usados en la Figura 3.1

Figura 4.1: Diagramas de clases.
Elaboración propia

- `_step(action) -> TimeStep`: Esta función recibe una acción y ofrece un *reward* y una nueva observación empaquetados en un TimeStep. En este caso, la función comprueba primero si el *reward*

obtenido es negativo antes de actualizar el estado de la red. En caso de que la acción a tomar sea válida, recalcula el *reward* con el estado actualizado para añadirlo al TimeStep.

- `action_spec() -> TensorSpec`: Esta función es el mecanismo de comunicarle al resto de elementos del sistema qué forma tiene que tener un tensor que represente una acción.
- `observation_spec() -> TensorSpec`: Esta función es el mecanismo de comunicarle al resto de elementos del sistema qué forma tiene que tener un tensor que represente una observación del estado de la red.

Además de estas funciones, la clase NetEnvironment cuenta con una función `_reward()`, que se utiliza en cada llamada a `_step()` para calcular el *reward*. Esta función es muy importante, ya que **en caso de querer cambiar los criterios de calidad e implementar una nueva función de recompensa, esta es la función que se debe modificar o sustituir**, bien sea directamente en el código o mediante herencia.

Los objetos de clase TimeStep no son más que contenedores que agrupan, principalmente, el *reward* obtenido tras cada acción, una observación del entorno y un número que indica si el paso dado es el comienzo, el intermedio o el final del episodio.

La observación del entorno es un tensor que será la entrada de la red neuronal, y que combina la matriz del estado de la red física con la pareja CU-DU que componen la petición correspondiente. Específicamente, esta petición se inserta como dos filas adicionales al comienzo de la matriz de estado.

4.1.3. Observer - Anexo H

Esta clase no hereda de ninguna otra, aunque sí está pensada para reemplazar una clase específica. La intención original era utilizar la librería Reverb y las clases de TensorFlow pensadas para combinar ambas librerías, pero no se consiguió que funcionase el replay buffer adecuadamente, así que se decidió desarrollar una clase propia que realizase solamente la función deseada. Se desconoce el impacto en escalabilidad y eficiencia de esta decisión, y si bien el utilizar el replay buffer de Reverb podría suponer una mejora, en términos de complejidad resulta mucho más sencilla la clase que se ha desarrollado, ya que cumple con la función estrictamente necesaria de almacenar los datos del episodio en una trayectoria.

Esto se realiza implementando una función `call(Trajectory)` que el Driver ejecutará con cada TimeStep que obtenga del entorno.

4.1.4. TIManager - Anexo D

Esta clase sirve como *wrapper* de todo el sistema, y permite acceder a las funciones necesarias para ofrecer las funcionalidades deseadas, definidas en en Capítulo 3. En la Sección 4.3 se muestran de forma práctica las funciones necesarias para utilizar esta clase.

4.1.5. ReinforceAgent

Esta clase implementa un agente que facilita el entrenamiento de una red neuronal. Viene ya desarrollado en TensorFlow Agents, y se utiliza como una "caja negra" a la cual se le ofrecen las variables

y objetos necesarios para instanciarla, para posteriormente utilizarla tal y como indican la documentación y los tutoriales oficiales de la librería.

4.1.6. PyDriver

Esta clase de TensorFlow Agents posibilita la comunicación entre la red neuronal y el entorno, permitiendo que ambos intercambien acciones y TimeSteps. Por cada TimeStep recibido, el objeto PyDriver llama a una lista de observadores u *observers* que se le indican en la instanciación, los cuales deben implementar el método `call(Trajectory)`. En el bucle de entrenamiento se usa un objeto PyDriver junto con un objeto Observer para almacenar la trayectoria recogida durante el episodio.

4.1.7. Checkpointer

Esta clase ofrecida también por TensorFlow se utiliza tal cual indica la documentación para exportar o importar el agente y su *policy* (esto es, la red neuronal).

4.1.8. StellarGraph

Esta clase se utiliza solamente a modo de herramienta para contener la información de la red física, importando desde un fichero los datos una sola vez y guardándolos en un objeto de esta clase para poder actualizarlos cuando haga falta. Existen otras librerías capaces de realizar esta función, pero esta es simple, fácil de usar y permite convertir el grafo a un objeto de la librería NetworkX, la cual facilita la representación gráfica de la red física y los parámetros deseados.

4.2. Ficheros de entrada

Se ha mencionado ya que los datos de la red física y la red virtual se importan desde ficheros, pero no se ha precisado el formato que deben tener estos ficheros. Deben ser archivos de extensión json, de texto plano que representan una estructura de datos determinada. Python permite importar estos ficheros como un objeto de forma muy sencilla, y solamente debe procesarse dicho objeto para convertirlo en uno de clase StellarGraph.

El formato de los datos contenidos en los ficheros json no es complejo, por lo que solamente se mostrará, a través de un ejemplo, para la red sustrato y otro para la red virtual en los Anexos A y B, respectivamente.

En el contexto de este trabajo se ha desarrollado un programa capaz de generar estos datos de forma automática, teniendo en cuenta las posibles configuraciones de las parejas de nodos virtuales, que permite además ajustar el tamaño de la red entre cuatro posibles valores para poder probar el sistema, variando el número de nodos físicos y virtuales. El código de este programa no se incluye en los anexos, pero se recomienda usar una estrategia similar para automatizar la generación de los datos.

4.3. Utilización del código

Para utilizar el código se muestra un ejemplo simple en el Anexo C. En esta sección se usa dicho código para explicar la utilización del sistema. Se divide en cuatro partes, que se describen a continuación.

4.3.1. Inicialización de los parámetros

Este fragmento de código asigna valores a las variables que contienen los diferentes parámetros configurables. A continuación se explica el significado de cada variable.

Las variables `path` son las que indican en qué ficheros se hallan los datos de la red sustrato y la red virtual. Los parámetros `w` son los que configuran los pesos de los tres componentes de la función de *reward*.

Por otra parte, los parámetros de la convolución indican cuántas capas de GCN tendrá la NN y de qué grado será la operación de convolución de cada capa.

La variable `fc_layers` es una lista de *strings* que permite configurar cuántas capas Dense y de qué tamaño tendrá la NN. Estos *strings* deben ser expresiones matemáticas interpretables con la función `eval()` de Python, pero se permite el uso de una variable `n` que representa el número de nodos de la red sustrato, de forma que se puede definir el número de unidades de las capas Dense en función de `n`. Es importante tener en cuenta que, de forma automática, se agregará una capa Dense al final, con dos unidades, de modo que esta variable solo permite añadir capas intermedias entre la GCN y la capa final.

La tasa de aprendizaje `learning_rate` y el `momentum` son parámetros relacionados con el proceso de aprendizaje de la red. Ambos sirven para la instanciación del agente.

El número de iteraciones es el que define cuántos ciclos de entrenamiento se llevarán a cabo. La variable `restore` indica si se desea restaurar un checkpoint previamente exportado desde un directorio, el cual se indica mediante otra variable `path`.

Por último, la variable `debug` permite activar o desactivar una serie de mensajes mostrados en la consola, que permiten observar cómo fluyen los datos a través de las distintas partes del sistema. Si aún así resultan insuficientes, a lo largo del código se encuentran comentadas algunas líneas con más mensajes que, si fuera necesario, se pueden des-comentar, haciendo así que se muestren cuando la variable `debug` esté activada.

4.3.2. Instanciación del TIManager

Una vez definidos los parámetros, se crea el objeto TIManager, que generará automáticamente la red neuronal, el agente, el entorno y el Observer. El objeto Driver sólo se genera cuando es necesario, es decir, en el bucle de entrenamiento o en la inferencia.

Todos los parámetros son opcionales y tienen valores por defecto, pero estos no son necesariamente óptimos ni están ajustados para un caso particular. Los nombres de los directorios se corresponden con los ficheros generados por el programa de generación de datos mencionado en la Sección 4.2, pero se puede utilizar cualquier directorio deseado.

4.3.3. Bucle de entrenamiento y guardado del progreso

Este fragmento consiste solamente en dos instrucciones. Una que inicia el bucle de entrenamiento y otra que genera el checkpoint para guardar el progreso del entrenamiento. El argumento de esta segunda función indica el nombre de la carpeta en la que se desea guardar el checkpoint, la cual se situará dentro de una carpeta llamada `policies/`. Dentro de la carpeta con el nombre indicado se generará a su vez

una carpeta por cada checkpoint que se genere, de forma que se pueda usar el mismo argumento para la función `save()` múltiples veces, sin tener que sobrescribir los checkpoints generados.

4.3.4. Realización de inferencia

Con esta instrucción se pueden realizar inferencias para la red virtual deseada, bien sea la indicada a la hora de instanciar el TIManager u otra distinta, cuya ubicación se indica mediante el argumento `vnet_path`. La red sustrato no se puede modificar.

También se puede indicar si se quiere exportar el resultado de la inferencia y el directorio en el que se guardaría, lo cual generará un fichero de texto plano en el lugar indicado.

Por último, se puede elegir si mostrar o no una representación gráfica de la red sustrato, similar a la de la Figura 2.1, con la diferencia de que se incluye en cada nodo el número de nodos virtualizados.

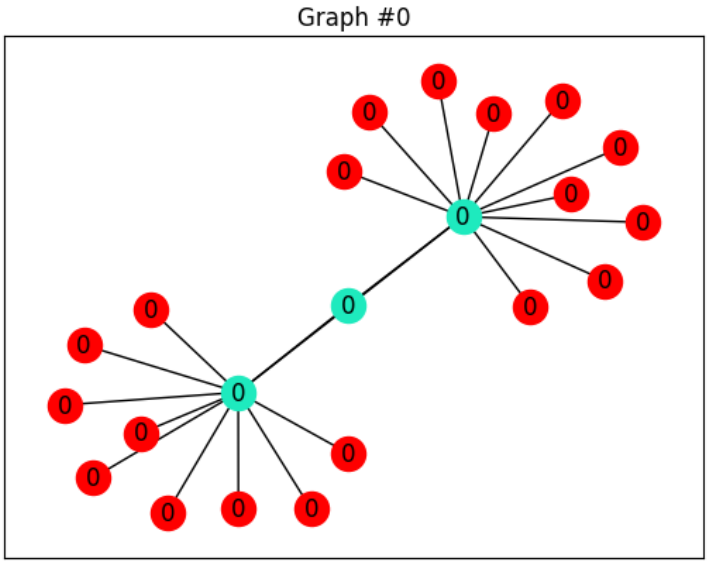
4.4. Resultados de la ejecución

Tras la ejecución de un código similar al explicado en la Sección 4.3 saldrá la información mostrada en la Sub-figura 4.2a por pantalla. Además, se genera una ventana con la imagen de la Sub-figura 4.2b.

Se observa que ninguna de las peticiones de virtualización se ha llevado a cabo, ya que todas las decisiones tomadas han sido descartadas. Esto resulta evidente en una red entrenada con tan solo tres iteraciones, pero lamentablemente aún con un numero mucho mayor de iteraciones, no se ha conseguido obtener un resultado positivo para ningún tamaño de la red. En el siguiente capítulo se propondrán sugerencias de mejora que podrían contribuir a mejorar el funcionamiento del entorno implementado.

```
------------------------------------------------------
[VirtualizationManager] Starting iteration 1...
[NetEnvironment]: Virtualization finished. Successfull petitions: 0 of 26 petitions.
Loss: 70.63376
------------------------------------------------------
[VirtualizationManager] Starting iteration 2...
[NetEnvironment]: Virtualization finished. Successfull petitions: 0 of 26 petitions.
Loss: 70.616104
------------------------------------------------------
[VirtualizationManager] Starting iteration 3...
[NetEnvironment]: Virtualization finished. Successfull petitions: 0 of 26 petitions.
Loss: 70.59314
[VirtualizationManager]: Policy saved in /madridMed in policies folder
[NetEnvironment]: Virtualization finished. Successfull petitions: 0 of 26 petitions.
```

(a) Salida de texto



(b) Imagen generada

Figura 4.2: Salida del código ejecutado.
Elaboración propia

Capítulo 5

Conclusiones

En este capítulo se resume el trabajo realizado y se analiza el nivel de completitud de los objetivos fijados en el Capítulo 1. Asimismo, se propondrán extensiones del entorno implementado para ampliar y mejorar su funcionalidad.

5.1. Evaluación de objetivos

En primer lugar, la investigación realizada muestra que diversos autores [20-25] están de acuerdo en que sistemas basados en redes neuronales, GCNs y RL pueden resultar soluciones adecuadas para afrontar problemas de *network embedding*, y al aplicar este tipo de soluciones al problema específico de este trabajo se ha encontrado que hay una excelente adecuación de la arquitectura al problema, siendo el modelo agente-entorno excelente para representar cómo la toma de decisiones de virtualización van afectando al estado de la red, condicionando por tanto las decisiones siguientes.
Se considera este objetivo alcanzado.

En segundo lugar, el diseño propuesto en el Capítulo 3, que queda condensado en la Figura 3.1, cumple con el objetivo de adaptarse a distintos criterios de calidad o requisitos. No sólo permite ajustar el peso de cada criterio, sino que simplemente sustituyendo una única función (*reward*) permite la definición de una política totalmente distinta. Esto es una consecuencia del RL, en el que el *reward* es una función separada de los datos de entrada, a diferencia de otras estrategias de aprendizaje.
Se considera este objetivo alcanzado.

En tercer lugar, se ha generado un código que cumple con todas las especificaciones, las funcionalidades requeridas y que permite hacer los ajustes necesarios para adecuarse más aún a las necesidades de un operador específico.
Se considera este objetivo alcanzado.

5.2. Propuestas de mejora

A pesar de haber alcanzado los objetivos de este trabajo, los resultados distan de ser perfectos, y por tanto hay muchos puntos sobre los que se puede seguir mejorando e investigando para lograr que la red neuronal entrenada por el sistema consiga ser una solución óptima al problema de *embedding*.

A continuación se exponen tres aspectos que se sugiere considerar en caso de que se desee mejorar el sistema.

- Se considera que es conveniente replantear los valores de salida de la red neuronal, ya que las NN se comportan mejor al trabajar con valores pequeños y centrados en cero, lo cual no se cumple para la salida de la red neuronal diseñada. Una posible solución a esto sería replantear la representación de la acción a tomar.
- La *policy* del agente es determinista, lo que significa que consiste en una sola acción con probabilidad 1. Los sistemas de RL se comportan mejor con *policies* que, en vez de ser deterministas, generan una distribución de probabilidad de las posibles acciones a tomar. Este cambio en el diseño es complejo, ya que el número de acciones posibles aumenta enormemente con el número de nodos y, por tanto, requeriría reconsiderar no solo la red neuronal, sino también el entorno.
- La librería TensorFlow es muy completa y extensa, pero en muchos casos la documentación resulta insuficiente para entender el funcionamiento de muchas de sus clases y, por tanto, es complicado solucionar problemas de compatibilidad que surgen entre algunas de ellas. Por tanto, se sugiere la posibilidad de probar otras librerías de ML, como PyTorch.

Bibliografía

[1] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger y L. Dittmann. "Cloud RAN for Mobile Networks—A Technology Overview". En: *IEEE Communications Surveys Tutorials* 17.1 (2015), págs. 405-426. DOI: 10.1109/COMST.2014.2355255.

[2] J. Wu, Z. Zhang, Y. Hong e Y. Wen. "Cloud radio access network (C-RAN): a primer". En: *IEEE Network* 29.1 (2015), págs. 35-41. DOI: 10.1109/MNET.2015.7018201.

[3] G. O. Pérez, J. A. Hernández y D. Larrabeiti. "Fronthaul network modeling and dimensioning meeting ultra-low latency requirements for 5G". En: *IEEE/OSA Journal of Optical Communications and Networking* 10.6 (2018), págs. 573-581. DOI: 10.1364/JOCN.10.000573.

[4] C. I, Y. Yuan, J. Huang, S. Ma, C. Cui y R. Duan. "Rethink fronthaul for soft RAN". En: *IEEE Communications Magazine* 53.9 (2015), págs. 82-88. DOI: 10.1109/MCOM.2015.7263350.

[5] L. M. P. Larsen, A. Checko y H. L. Christiansen. "A Survey of the Functional Splits Proposed for 5G Mobile Crosshaul Networks". En: *IEEE Communications Surveys Tutorials* 21.1 (2019), págs. 146-172. DOI: 10.1109/COMST.2018.2868805.

[6] A. Garcia-Saavedra, J. X. Salvat, X. Li y X. Costa-Perez. "WizHaul: On the Centralization Degree of Cloud RAN Next Generation Fronthaul". En: *IEEE Transactions on Mobile Computing* 17.10 (2018), págs. 2452-2466.

[7] 3GPP. *5G System Overview*. 2022. URL: https://www.3gpp.org/technologies/5g-system-overview.

[8] C. A. Hervella Baturone. *Análisis del comportamiento de controladores vRAN en redes 5G*. Secciones 2.1 y 2.2. 2020. URL: http://hdl.handle.net/10902/19018.

[9] C. C. Erazo-Agredo, M. Garza-Fabre, R. A. Calvo, L. Diez, J. Serrat y J. Rubio-Loyola. "Joint Route Selection and Split Level Management for 5G C-RAN". En: *IEEE Transactions on Network and Service Management* 18.4 (2021), págs. 4616-4638. DOI: 10.1109/TNSM.2021.3091543. URL: https://ieeexplore.ieee.org/document/9462353.

[10] 3. G. P. P. (3GPP). *Study on new radio access technology: Radio access architecture and interfaces*. TR. Ver. 14.0.0. 2017.

[11] D. Martínez Rica. *Análisis de rendimiento de redes vRAN en escenarios con limitación de recursos de virtualización*. Tabla 4.3. 2021.

[12] D. Karunakaran. *REINFORCE — a policy-gradient based reinforcement Learning algorithm*. 2020. URL: https://medium.com/intro-to-artificial-intelligence/reinforce-a-policy-gradient-based-reinforcement-learning-algorithm-84bde440c816.

[13] R. S. Sutton y A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[14] D. Karunakaran. *Key concepts in Reinforcement Learning*. 2020. URL: https://medium.com/intro-to-artificial-intelligence/key-concepts-in-reinforcement-learning-2af715dfbfa.

[15] S. AI. *Reinforcement Learning algorithms — an intuitive overview*. 2019. URL: https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc.

[16] E. Alpaydin. *Introduction to Machine Learning*. Third. MIT Press, 2014. URL: https://www.cmpe.boun.edu.tr/~ethem/i2ml3e/index.html.

[17] K. E. Koech. *The Basics of Neural Networks (Neural Network Series) — Part 1*. 2022. URL: https://towardsdatascience.com/the-basics-of-neural-networks-neural-network-series-part-1-4419e343b2b.

[18] B. Sanchez-Lengeling, E. Reif, A. Pearce y A. B. Wiltschko. "A Gentle Introduction to Graph Neural Networks". En: *Distill* (2021). URL: https://distill.pub/2021/gnn-intro.

[19] A. Daigavane, B. Ravindran y G. Aggarwal. "Understanding Convolutions on Graphs". En: *Distill* (2021). URL: https://distill.pub/2021/understanding-gnns.

[20] N. Chen, P. Zhang, N. Kumar, C.-H. Hsu, L. Abualigah y H. Zhu. "Spectral graph theory-based virtual network embedding for vehicular fog computing: A deep reinforcement learning architecture". En: *Knowledge-Based Systems* 257 (2022), pág. 109931. DOI: https://doi.org/10.1016/j.knosys.2022.109931. URL: https://www.sciencedirect.com/science/article/pii/S0950705122010243.

[21] A. Rkhami, T. A. Quang Pham, Y. Hadjadj-Aoul, A. Outtagarts y G. Rubino. "On the Use of Graph Neural Networks for Virtual Network Embedding". En: *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. 2020, págs. 1-6. DOI: 10.1109/ISNCC49221.2020.9297270. URL: https://ieeexplore.ieee.org/document/9297270.

[22] S. Ma, H. Yao, T. Mai, J. Yang, W. He, K. Xue y M. Guizani. "Graph Convolutional Network Aided Virtual Network Embedding for Internet of Thing". En: *IEEE Transactions on Network Science and Engineering* 10.1 (2023), págs. 265-274. DOI: 10.1109/TNSE.2022.3207205. URL: https://ieeexplore.ieee.org/document/9894092.

[23] Z. Yan, J. Ge, Y. Wu, L. Li y T. Li. "Automatic Virtual Network Embedding: A Deep Reinforcement Learning Approach With Graph Convolutional Networks". En: *IEEE Journal on Selected Areas in Communications* 38.6 (2020), págs. 1040-1057. DOI: 10.1109/JSAC.2020.2986662. URL: https://ieeexplore.ieee.org/document/9060910.

[24] P. Zhang, C. Wang, N. Kumar, W. Zhang y L. Liu. “Dynamic Virtual Network Embedding Algorithm Based on Graph Convolution Neural Network and Reinforcement Learning”. En: *IEEE Internet of Things Journal* 9.12 (2022), págs. 9389-9398. DOI: 10.1109/JIOT.2021.3095094. URL: https://ieeexplore.ieee.org/document/9475485.

[25] Z. Yan, J. Ge, Y. Wu, H. Zheng, L. Li y T. Li. “Automatic Virtual Network Embedding Based on Deep Reinforcement Learning”. En: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, págs. 625-631. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00095. URL: https://ieeexplore.ieee.org/document/8855447.

Anexos

Anexo A

JSON de la red sustrato

```
{
    "nodes": [
        {
            "id": "a",
            "capacity": 100,
            "vnodes": 0,
            "zone": 1,
            "subzone": 1,
            "type": "C"
        },
        {
            "id": "b",
            "capacity": 100,
            "vnodes": 0,
            "zone": 1,
            "subzone": 2,
            "type": "C"
        },
        {
            "id": "c",
            "capacity": 90,
            "vnodes": 0,
            "zone": 2,
            "subzone": 1,
            "type": "D"
        },
        {
            "id": "d",
            "capacity": 150,
            "vnodes": 0,
            "zone": 2,
            "subzone": 1,
            "type": "D"
        },
        {
            "id": "e",
            "capacity": 100,
```

```
            "vnodes": 0,
            "zone": 2,
            "subzone": 2,
            "type": "D"
        }
    ],
    "edges": [
        {
            "node-ids": [
                "a",
                "b"
            ]
        },
        {
            "node-ids": [
                "a",
                "c"
            ]
        },
        {
            "node-ids": [
                "a",
                "d"
            ]
        },
        {
            "node-ids": [
                "b",
                "c"
            ]
        },
        {
            "node-ids": [
                "b",
                "e"
            ]
        },
        {
            "node-ids": [
                "d",
                "e"
            ]
        }
    ]
}
```

Anexo B

JSON de la red virtual

```
[
    {
        "c_node": {
            "capacity": 11,
            "zone": 1,
            "subzone": 1
        },
        "d_node": {
            "capacity": 12,
            "zone": 1,
            "subzone": 1
        }
    },
    {
        "c_node": {
            "capacity": 21,
            "zone": 1,
            "subzone": 2
        },
        "d_node": {
            "capacity": 22,
            "zone": 2,
            "subzone": 1
        }
    },
    {
        "c_node": {
            "capacity": 31,
            "zone": 2,
            "subzone": 2
        },
        "d_node": {
            "capacity": 32,
            "zone": 2,
            "subzone": 2
        }
    }
```

```
38 ]
```

Anexo C

Ejemplo de uso

```
import TIManager

#Inicializacion de los parametros
size = "Med"
snet_path = f"scenarios/madrid{size}.json"
vnet_path = f"scenarios/madridRequest{size}.json"
w1 = 0.3
w2 = 0.4
w3 = 0.3
convolution_layers = 3
layer_convolution_degree = 1
fc_layers = ["(n+2)*5"]
learning_rate = 0.01
momentum = 0.3
learning_iterations = 100
restore = False
checkpoint_path = f"policies/madrid{size}/0"
debug = False

#Instanciacion del TIManager
vm = TIManager.TIManager(snet_path=snet_path,
                         vnet_path=vnet_path,
                         checkpoint_path=checkpoint_path,
                         w1=w1,
                         w2=w2,
                         w3=w3,
                         convolution_layers=convolution_layers,
                         layer_convolution_degree=layer_convolution_degree,
                         fc_layers=fc_layers,
                         learning_rate=learning_rate,
                         momentum=momentum,
                         restore_checkpoint=restore,
                         debug=debug)

#Bucle de entrenamiento y guardado del progreso
vm.train(learning_iterations)
vm.save(f"madrid{size}")
```

```
38
39 #Realizacion de inferencia
40 vm.inference(save = False,
41              #vnet_path=,
42              #embed_path=,
43              render=True)
```

Anexo D

Clase TIManager

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import warnings
from scipy.sparse import SparseEfficiencyWarning
warnings.filterwarnings("ignore", category=SparseEfficiencyWarning)

import NetEnvironment
import GCNLayer
import DenseLayer
import Observer

import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
from tf_agents.networks import Sequential
from tf_agents.agents import ReinforceAgent
from tf_agents.drivers import py_driver
from tf_agents.utils.common import Checkpointer

class TIManager:
    def __init__(self,
                 snet_path: str = "snet.json",
                 vnet_path: str = "vnet.json",
                 embed_path: str = "embeddings/",
                 checkpoint_path: str = "policies/",
                 w1: float = 0.3,
                 w2: float = 0.4,
                 w3: float = 0.3,
                 convolution_layers: int = 3,
                 layer_convolution_degree: int = 1,
                 fc_layers: list = [],
                 learning_rate: float = 1e-2,
                 momentum: float = 0.1,
                 restore_checkpoint: bool = False,
                 debug: bool = False):

        self.env = NetEnvironment.NetEnvironment(snet_path=snet_path,
                                                 vnet_path=vnet_path,
```

```
                                                embed_path=embed_path,
                                                w1=w1,
                                                w2=w2,
                                                w3=w3,
                                                debug=debug)
        self._Gs = self.env.graph()
        self._n = self._Gs.number_of_nodes()
        self.debug = debug

        self._npetitions = len(self.env.vnet)

        self._actor_network = Sequential(
            layers=self._get_layers(convolution_layers,
                                    layer_convolution_degree,
                                    fc_layers),
            input_spec=self.env.observation_spec(),
            name="ActorNetwork")

        self._agent = ReinforceAgent(
            time_step_spec=self.env.time_step_spec(),
            action_spec=self.env.action_spec(),
            actor_network=self._actor_network,
            optimizer=tf.keras.optimizers.SGD(learning_rate=learning_rate,
                                              momentum=momentum),
            use_advantage_loss = False,
            name="Agent")

        self._observer = Observer.Observer(debug=self.debug)

        if restore_checkpoint:
            self._saver = Checkpointer(ckpt_dir=checkpoint_path,
                                       max_to_keep=1,
                                       agent=self._agent,
                                       policy=self._agent.policy)
            self._saver.initialize_or_restore()

    def train(self, iterations: int = 10):
        for i in range(iterations):
            print("----------------------------------------------------------")
            print(f"[VirtualizationManager] Starting iteration {i+1}...")
            initial_time_step = self.env.reset()
            collect_driver = py_driver.PyDriver(
                env=self.env,
                policy=self._agent.collect_policy,
                observers=[self._observer],
                max_steps=self._npetitions,
                max_episodes=1,
                )

            if self.debug: print("[VirtualizationManager] Collecting trajectory...")
            collect_driver.run(initial_time_step)

```

```
90            if self.debug: print("[VirtualizationManager] Trajectory collected")
91            trajectory = self._observer.get_trajectory()
92            if self.debug: print("Trajectory:", trajectory)
93            trajectory = trajectory.replace(action=tf.expand_dims(trajectory.action,
     axis=0),
94                                            discount=tf.expand_dims(trajectory.
    discount,axis=0),
95                                            next_step_type=tf.expand_dims(trajectory
    .next_step_type,axis=0),
96                                            observation=tf.expand_dims(trajectory.
    observation,axis=0),
97                                            reward=tf.expand_dims(trajectory.reward,
    axis=0),
98                                            step_type=tf.expand_dims(trajectory.
    step_type,axis=0))
99
100           if self.debug: print("[VirtualizationManager] Training agent...")
101           train_loss = self._agent.train(experience=trajectory).loss
102           print("Loss:", train_loss.numpy())
103
104           self._observer._reset()
105
106   def inference(self,
107                 save: bool = False,
108                 render: bool = False,
109                 vnet_path: str = None,
110                 embed_path: str = None):
111       self.env.change_path(vnet_path = vnet_path,
112                            embed_path = embed_path,
113                            auto_reset = False)
114       initial_time_step = self.env.reset()
115       collect_driver = py_driver.PyDriver(
116           env=self.env,
117           policy=self._agent.policy,
118           observers=[],
119           max_steps=self._npetitions,
120           max_episodes=1,
121           )
122       collect_driver.run(initial_time_step)
123
124       if save:
125           self.env.store_embedding(render=render)
126       elif render:
127           self.env.render()
128
129       return self.env.get_embedding()
130
131   def save(self, export_name: str = "policy"):
132       try:
133           os.mkdir("policies/"+export_name)
134       except FileExistsError:
135           pass
```

```
136         index = len(os.listdir("policies/"+export_name+"/"))
137         saver = Checkpointer(ckpt_dir="policies/"+export_name+"/"+str(index)+"/",
138                              max_to_keep=1,
139                              agent=self._agent,
140                              policy=self._agent.policy)
141         saver.save(global_step=0)
142         print(f"[VirtualizationManager]: Policy saved in /{export_name} in policies
    folder")
143 
144 
145     def _get_layers(self,
146                     convolution_layers: int = 3,
147                     layer_convolution_degree: int = 1,
148                     fc_layers: list = []):
149         """Creates the layers of a neural network.
150 
151         Creates the neural network layers based on the number of convolution layers
    and the degree of each convolution.
152 
153         Args:
154             Gs (StellarGraph): Object containing an undirected graph.
155             convolution_layers (int, optional): Number of GCN layers. Defaults to 3.
156             layer_convolution_degree (int, optional): Degree of the convolution of
    each GCN layer. Defaults to 1.
157             bias (bool, optional): Indicates if the dense layers will have a bias or
     not. Defaults to True.
158 
159         Returns:
160             list: List containing the layers of the neural network.
161         """
162         layers_list = []
163         for i in range(convolution_layers):
164             if i == convolution_layers-1:
165                 layers_list.append(GCNLayer.GCNLayer(self._n,
166                                                      self._Gs,
167                                                      layer_conv_degree=
    layer_convolution_degree,
168                                                      next_is_dense=True,
169                                                      debug=self.debug))
170             else:
171                 layers_list.append(GCNLayer.GCNLayer(self._n,
172                                                      self._Gs,
173                                                      layer_conv_degree=
    layer_convolution_degree,
174                                                      debug=self.debug))
175 
176         n = self._n
177         for x in fc_layers:
178             layers_list.append(DenseLayer.DenseLayer(eval(x), debug=self.debug))
179         layers_list.append(DenseLayer.DenseLayer(2, int_out=True, int_max=self._n,
    debug=self.debug))
180         return layers_list
```

Anexo E

Clase NetEnvironment

```
from stellargraph import StellarGraph

import numpy as np

from tf_agents.environments import py_environment
from tf_agents.specs import array_spec
from tf_agents.trajectories import TimeStep, StepType

class NetEnvironment(py_environment.PyEnvironment):

    def __init__(self,
                 snet_path: str = "snet.json",
                 vnet_path: str = "vnet.json",
                 embed_path: str = "embeddings/",
                 w1: float = 0.3,
                 w2: float = 0.4,
                 w3: float = 0.3,
                 debug: bool = False):
        self._snet_path = snet_path
        self._vnet_path = vnet_path
        self._embed_path = embed_path
        self._w1 = w1
        self._w2 = w2
        self._w3 = w3
        self.debug = debug
        self._load_graph()
        self.n = self.Gs.number_of_nodes()
        self._load_petitions()
        self._petition_index = 0
        self._embedding = []

        self._action_spec = array_spec.BoundedArraySpec(
            shape=(2,), dtype=np.int32, minimum=0, maximum=self.n, name="action")
        self._observation_spec = array_spec.BoundedArraySpec(
            shape=(self.n+2,5), dtype=np.float32, minimum=0, name='observation')
        self._state = self.Gs

```

```
    @property
    def batch_size(self) -> int:
        return None

    def graph(self) -> StellarGraph:
        return self.Gs

    def action_spec(self) -> array_spec.BoundedArraySpec:
        return self._action_spec

    def observation_spec(self) -> array_spec.BoundedArraySpec:
        return self._observation_spec

    def change_path(self,
                    snet_path: str = None,
                    vnet_path: str = None,
                    embed_path: str = None,
                    auto_reset: bool = True):
        if snet_path is not None:
            self._snet_path = snet_path
        if vnet_path is not None:
            self._vnet_path = vnet_path
        if embed_path is not None:
            self._embed_path = embed_path

        if auto_reset:
            self._reset()
        return

    def set_debug(self, debug: bool = False):
        self.debug = debug
        return

    def set_weights(self, w1: float = None, w2: float = None, w3: float = None):
        if w1 is not None:
            self._w1 = w1
        if w2 is not None:
            self._w2 = w2
        if w3 is not None:
            self._w3 = w3
        return

    def _reset(self) -> TimeStep:
        self._load_graph()
        self._state = self.Gs
        self.n = self.Gs.number_of_nodes()
        self._load_petitions()
        self._petition_index = 0
        self._embedding = []
        if self.debug: print("[NetEnvironment] Environment reseted")
        ret = TimeStep(step_type = StepType.FIRST,
                       reward = np.asarray(0., dtype=np.float32),
```

```
                        discount = np.asarray(1., dtype=np.float32),
                        observation = self.observation(step=0,log=True))
        #if self.debug: print("[NetEnvironment] TimeStep returned:", ret)
        return ret

    def _step(self, action) -> TimeStep:
        #if self.debug: print("[NetEnvironment] Step requested:", action)
        [node_c, node_d] = self.petition(1)
        [embed_c, embed_d] = action

        reward = self._reward(node_c, node_d, embed_c, embed_d)
        if reward < 0:
            self._embedding.append((int(embed_c), int(embed_d), False))
        else:
            self._embedding.append((int(embed_c), int(embed_d), True))
            self._state.node_features()[embed_c-1][0] -= node_c["capacity"]
            self._state.node_features()[embed_d-1][0] -= node_d["capacity"]
            self._state.node_features()[embed_c-1][1] += 1
            self._state.node_features()[embed_d-1][1] += 1

            reward = self._reward(node_c, node_d, embed_c, embed_d)

        #Return new observation
        observation = self.observation(log=True, step=0)
        if observation is None:
            print("[NetEnvironment]: Virtualization finished.",
                  "Successfull petitions:", len([petition for petition in self.
    _embedding if petition[2]==True]),
                  "of", len(self._embedding), "petitions.")
            ret = TimeStep(step_type = StepType.LAST,
                        reward = np.asarray(reward, dtype=np.float32),
                        discount = np.asarray(1., dtype=np.float32),
                        observation = self.observation(step=0,no_petition=True))
            #if self.debug: print("[NetEnvironment] TimeStep returned:", ret)
            return ret
        else:
            ret = TimeStep(step_type = StepType.MID,
                        reward = np.asarray(reward, dtype=np.float32),
                        discount = np.asarray(1., dtype=np.float32),
                        observation = observation)
            #if self.debug: print("[NetEnvironment] TimeStep returned:", ret)
            return ret

    def observation(self, state: StellarGraph = None,
                    step: int = 1,
                    log: bool = False,
                    no_petition: bool = False) -> np.ndarray:
        if state is None:
            state = self._state
        petition = self.petition(step)
        if not no_petition:
            ret = np.array(np.concatenate((petition, np.array([
```

```
                [node[1], node[2], node[3], node[4], 0 if node[0]=="C" else 1] for node in state.node_features()]
            ))), dtype=np.float32) if petition is not None else None #, ndmin=3)
        else:
            ret = np.array(np.concatenate((np.array([[0,0,0,0,0],[0,0,0,0,0]]),
                                           np.array([[node[1], node[2], node[3], node[4], 0 if node[0]=="C" else 1] for node in state.node_features()])
                                           )), dtype=np.float32)#, ndmin=3)
        #if log and self.debug: print("[NetEnvironment] Observation made:", ret)
        return ret

    def petition(self, step: int = 0) -> np.ndarray or None:
        if self._petition_index < len(self.vnet):
            petition = self.vnet[self._petition_index]
            self._petition_index += step
            ret = np.array(
                [[petition["c_node"]["capacity"], 0, petition["c_node"]["zone"], petition["c_node"]["subzone"], 0],
                [petition["d_node"]["capacity"], 0, petition["d_node"]["zone"], petition["d_node"]["subzone"], 1]])
            #if self.debug: print("[NetEnvironment] Petition read from queue:", ret)
            return ret
        else:
            self._petition_index += step
            return None

    def _load_petitions(self):
        import json
        with open(self._vnet_path, 'r') as json_file:
            data = json.load(json_file)
            vnet = [{
                "c_node": {
                    "capacity": petition["c_node"]["capacity"],
                    "zone": petition["c_node"]["zone"],
                    "subzone": petition["c_node"]["subzone"]},
                "d_node": {
                    "capacity": petition["d_node"]["capacity"],
                    "zone": petition["d_node"]["zone"],
                    "subzone": petition["d_node"]["subzone"]}
                } for petition in data]

        self.vnet = vnet
        if self.debug: print("[NetEnvironment] Petitions loaded:", len(vnet))
        return

    def _load_graph(self):
        import json
        import pandas as pd
        with open(self._snet_path, 'r') as json_file:
            data = json.load(json_file)
            nodes = pd.DataFrame({
                "capacity": [node["capacity"] for node in data["nodes"]],
```

```
                "vnodes": [node["vnodes"] for node in data["nodes"]],
                "zone": [node["zone"] for node in data["nodes"]],
                "subzone": [node["subzone"] for node in data["nodes"]],
                "type": [0 if node["type"]=="C" else 1 for node in data["nodes"]]},
                index=[node["id"] for node in data["nodes"]]
                )
            edges = pd.DataFrame({
                "source": [edge["node-ids"][0] for edge in data["edges"]],
                "target": [edge["node-ids"][1] for edge in data["edges"]]
                })
            Gs = StellarGraph(nodes, edges)

        self.Gs = Gs
        if self.debug: print("[NetEnvironment] Graph loaded:", Gs.info())
        return

    def render(self, block=False, graph_number=0):
        import networkx as nx
        import matplotlib.pyplot as plt
        if self.debug: print(self.Gs.info())
        Gx = nx.Graph(self.Gs.to_networkx(feature_attr="features"))
        pos = nx.spring_layout(Gx,seed=666)
        nx.draw_networkx_nodes(Gx,pos,nodelist=[n[0] for n in Gx.nodes.data() if n
    [1]['features'][4]==0],node_color="#1eeabe")
        nx.draw_networkx_nodes(Gx,pos,nodelist=[n[0] for n in Gx.nodes.data() if n
    [1]['features'][4]==1],node_color="#fe0000")
        nx.draw_networkx_edges(Gx,pos,width=1,alpha=1)
        nx.draw_networkx_labels(Gx,pos,{n[0]: int(n[1]['features'][1]) for n in Gx.
    nodes.data()})
        plt.title("Graph #"+str(graph_number))
        plt.show(block=block)
        return

    def _reward(self, req_c, req_d, sel_c: int, sel_d: int) -> int:
        #Check if selected nodes are valid and viable
        if sel_c == 0:
            if sel_d == 0:
                return -1
            else:
                sel_d = self._state.node_features()[sel_d-1]
                if req_d[0] > sel_d[0] or sel_d[4] != 1:
                    return -0.666
                else:
                    return -0.5
        if sel_d == 0:
            return -0.666

        sel_d = self._state.node_features()[sel_d-1]
        sel_c = self._state.node_features()[sel_c-1]

        #Check [0]capacities and [4]types of selected nodes
        if req_c[0] > sel_c[0] or req_d[0] > sel_d[0] or \
```

```
            sel_c[4] != 0 or sel_d[4] != 1:
                return -0.5

        #Check [2]zone of selected C node
        if req_c[2] == sel_c[2]:
            c_zone = 1

        #Check [2]zone and [3]subzone of selected D node
        if req_d[2] == sel_d[2]:
            d_zone = 0.5
            if req_c[3] == sel_c[3]:
                d_zone = 1
        else:
            return -0.333

        #Calculate centralization degree of currently virtualized C nodes
        c_nodes = [node for node in self._state.node_features() if node[4]==0]
        DoC = np.sqrt(np.sum([node[1] for node in c_nodes]**2))/np.sum([node[1] for
    node in c_nodes])

        return self._w1*c_zone + self._w2*d_zone + self._w3*DoC

    def store_embedding(self, render: bool = False):
        from fnmatch import filter
        from os import listdir
        index = len(filter(listdir(self._embed_path), 'embedding-*.txt'))
        with open(self._embed_path + "embedding-" + str(index) + ".txt", 'w') as f:
            print("C | D | Possible", file=f)
            for node_c,node_d,result in self._embedding:
                print(node_c, "|", node_d, "|", result, file=f)
        if render:
            self.render(block=False, graph_number=index)
        return

    def get_embedding(self):
        return self._embedding
```

Anexo F

Clase GCNLayer

```
from stellargraph import StellarGraph
from scipy.sparse import csr_matrix, identity
import tensorflow as tf

class GCNLayer(tf.keras.layers.Layer):
    def __init__(self,
                 node_number: int,
                 Gs: StellarGraph,
                 F: csr_matrix = None,
                 layer_conv_degree: int = 1,
                 next_is_dense: bool = False,
                 debug: bool = False,
                 **kwargs):
        self._config = {
            "node_number": node_number,
            "Gs": Gs,
            "F": F,
            "layer_conv_degree": layer_conv_degree,
            "next_is_dense": next_is_dense,
            "debug": debug,
        }

        super(GCNLayer, self).__init__(dtype=tf.float32, **kwargs)
        self.node_number = node_number
        self.units = (node_number+2)*5
        self._batch_size = None
        self._sequence_size = None
        self._next_is_dense = next_is_dense
        self.debug = debug
        self.input_spec = tf.keras.layers.InputSpec(min_ndim=2,
                                                    max_ndim=4,
                                                    dtype=tf.float32)

        if F is None:
            # Adjacency matrix (A) + Identity matrix (I)
            A = csr_matrix(Gs.to_adjacency_matrix())
            A = A + csr_matrix(identity(node_number))
```

```
            # Degree diagonal matrix (D) ** -0.5
            D = csr_matrix((node_number,node_number))
            D.setdiag(list(Gs.node_degrees().values()))
            D = csr_matrix(D.power(-0.5))
            self._config["F"] = D*A*D
            self.F = tf.Variable(name="function", initial_value=csr_matrix((D*A*D)**
    layer_conv_degree).toarray(), dtype=tf.float32 , trainable=False)
        else:
            self.F = tf.Variable(name="function", initial_value=(F**
    layer_conv_degree).toarray(), dtype=tf.float32, trainable=False)

        self.W1 = self.add_weight("weights1",
                                  shape=(self.node_number,self.node_number),
                                  dtype=tf.float32,
                                  initializer=tf.keras.initializers.RandomNormal(
                                              mean=0.0, stddev=0.5, seed=None),
                                  regularizer='l1',
                                  trainable=True)
        self.W2 = self.add_weight("weights2",
                                  shape=(self.node_number,self.node_number),
                                  dtype=tf.float32,
                                  initializer=tf.keras.initializers.RandomNormal(
                                              mean=0.0, stddev=0.5, seed=None),
                                  regularizer='l1',
                                  trainable=True)

    def build(self, input_shape):
        return

    def set_shape(self, input_shape):
        if self.debug: print("[GCNLayer]: Setting input shape to", input_shape)
        if len(input_shape) == 4:
            self._batch_size = input_shape[0]
            self._sequence_size = input_shape[1]
        elif len(input_shape) == 3:
            self._batch_size = input_shape[0]
            self._sequence_size = 1
        elif len(input_shape) == 2:
            self._batch_size = 1
            self._sequence_size = 1
        else:
            raise ValueError("Input shape must be 2D (data), 3D (batch,data) or 4D (
    batch,sequence,data).")
        return

    def call(self, inputs):
        if len(inputs.shape) == 2:
            if not hasattr(inputs, "numpy"):
                if self._next_is_dense:
                    return tf.squeeze(tf.keras.Input([self.units]),axis=0)
                return inputs
            inputs = tf.expand_dims(tf.expand_dims(inputs, axis=0), axis=0)
```

```
            dimension = 2
        elif len(inputs.shape) == 3:
            if not hasattr(inputs, "numpy"):
                if self._next_is_dense:
                    return tf.keras.Input([self.units], batch_size=inputs.shape[0])
                return inputs
            inputs = tf.expand_dims(inputs, axis=0)
            dimension = 3
        elif len(inputs.shape) == 4:
            if not hasattr(inputs, "numpy"):
                if self._next_is_dense:
                    return tf.keras.Input([inputs.shape[1],self.units], batch_size=
inputs.shape[0])
                return inputs
            dimension = 4
        else:
            raise ValueError("Input shape must be 2D (data), 3D (batch,data) or 4D (
batch,sequence,data).")

        self.set_shape(inputs.shape)

        outputs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True, name="
outputs")
        output = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True, name="
output")

        for i in range(self._batch_size):
            sequence = inputs[i]
            for j in range(self._sequence_size):
                input = sequence[j]
                #if self.debug: print("Input:",input)

                capacity = tf.gather(input, [0], axis=1)[2:]
                capacity = tf.matmul(tf.math.multiply(self.F,self.W1),capacity)
                capacity = tf.concat([tf.gather(input, [0], axis=1)[0:2],capacity],
axis=0)
                #if self.debug: print("Capacity:",capacity)

                vnodes = tf.gather(input, [1], axis=1)[2:]
                vnodes = tf.matmul(tf.math.multiply(self.F,self.W2),vnodes)
                vnodes = tf.concat([tf.gather(input, [1], axis=1)[0:2],vnodes], axis
=0)
                #if self.debug: print("Vnodes:",vnodes)

                out = tf.gather(input, [2,3,4], axis=1)
                out = tf.concat([capacity, vnodes, out], axis=1)
                #if self.debug: print("Out:",out)
                output = output.write(j, out)
                #if self.debug: print("Output_temp:",output)

            output = output.stack()
            outputs = outputs.write(i, output)
```

```
        outputs = outputs.stack()

        if dimension == 2:
            outputs = tf.squeeze(tf.squeeze(outputs, axis=0), axis=0)
            if self._next_is_dense:
                outputs = tf.reshape(outputs, [-1])
        elif dimension == 3:
            outputs = tf.squeeze(outputs, axis=0)
            if self._next_is_dense:
                outputs = tf.reshape(outputs, [self._batch_size, -1])
        elif dimension == 4 and self._next_is_dense:
            outputs = tf.reshape(outputs, [self._batch_size, self._sequence_size,
    -1])

        if self.debug: print("[GCNLayer]: Output shape:",outputs.shape)
        return outputs

    def get_config(self):
        return self._config
```

Anexo G

Clase DenseLayer

```
import tensorflow as tf
import numpy as np

class DenseLayer(tf.keras.layers.Layer):

    def __init__(self, units:int, dtype=tf.float32, int_out:bool=False, int_max:int
    =0, debug:bool=False):
        self._config = {
            "units": units,
            "dtype": dtype,
            "int_out": int_out,
            "int_max": int_max,
            "debug": debug
        }
        super(DenseLayer, self).__init__(dtype=dtype)
        self.units = units
        self._int_out = int_out
        self._int_max = int_max
        self.debug = debug
        self.input_spec = tf.keras.layers.InputSpec(min_ndim=1,
                                                    max_ndim=3,
                                                    dtype=tf.float32)

    def build(self, input_shape):
        self.w = self.add_weight("kernel",
                                 shape=(input_shape[-1],self.units),
                                 initializer=tf.keras.initializers.RandomNormal(
                                             mean=0.0, stddev=0.5, seed=None),
                                 regularizer='l1',
                                 trainable=True)
        self.b = self.add_weight("bias",
                                 shape=(self.units,),
                                 initializer=tf.keras.initializers.RandomNormal(
                                             mean=0.0, stddev=0.5, seed=None),
                                 regularizer='l1',
                                 trainable=True)
        return
```

```
37
38      def call(self, inputs):
39          if self.debug: print("[SimpleDense]: Input shape:",inputs.shape)
40          #if self.debug: print("[SimpleDense]: Input:",inputs.numpy())
41
42          if len(inputs.shape) == 1:
43              if not hasattr(inputs, "numpy"):
44                  return tf.squeeze(tf.keras.Input([self.units]),axis=0)
45              ret = tf.squeeze(tf.matmul(tf.expand_dims(inputs,axis=0), self.w) + self
    .b, axis=0)
46          elif len(inputs.shape) == 2:
47              if not hasattr(inputs, "numpy"):
48                  return tf.keras.Input([self.units], batch_size=inputs.shape[0])
49              ret = tf.stack([tf.squeeze(tf.matmul(tf.expand_dims(input,axis=0), self.
    w) + self.b, axis=0) for input in inputs])
50          elif len(inputs.shape) == 3:
51              if not hasattr(inputs, "numpy"):
52                  return tf.keras.Input([inputs.shape[1],self.units], batch_size=
    inputs.shape[0])
53              ret = tf.stack([tf.stack([tf.squeeze(tf.matmul(tf.expand_dims(input,axis
    =0), self.w) + self.b, axis=0) for input in sequence]) for sequence in inputs])
54          else:
55              raise ValueError("Input shape must be 1D (data), 2D (batch,data) or 3D (
    batch,sequence,data).")
56
57          if self._int_out:
58              #print("[SimpleDense]: Output pre-int:",ret.numpy())
59              if self._int_max == 0:
60                  print("[SimpleDense]: WARNING: int_max set to 0 when int_out is True
    .")
61              ret = tf.clip_by_value(ret, clip_value_min=0, clip_value_max=self.
    _int_max)
62              ret = tf.cast(ret, np.int32)
63          if self.debug: print("[SimpleDense]: Output shape:",ret.shape)
64          #if self.debug: print("[SimpleDense]: Output:",ret.numpy())
65          return ret
```

Anexo H

Clase Observer

```
import tensorflow as tf

class Observer:
    def __init__(self, debug: bool = False):
        self.debug = debug
        self._reset()
    def _reset(self):
        self._trajectory = None
    def __call__(self, trajectory):
        if self.debug: print("[Observer]: Called with following input:", trajectory)
        if self._trajectory is None:
            self._trajectory = trajectory.replace(
                action=tf.expand_dims(trajectory.action, axis=0),
                discount=tf.expand_dims(trajectory.discount,axis=0),
                next_step_type=tf.expand_dims(trajectory.next_step_type,axis=0),
                observation=tf.expand_dims(trajectory.observation,axis=0),
                reward=tf.expand_dims(trajectory.reward,axis=0),
                step_type=tf.expand_dims(trajectory.step_type,axis=0)
                )
        else:
            trajectory = trajectory.replace(
                action=tf.expand_dims(trajectory.action, axis=0),
                discount=tf.expand_dims(trajectory.discount,axis=0),
                next_step_type=tf.expand_dims(trajectory.next_step_type,axis=0),
                observation=tf.expand_dims(trajectory.observation,axis=0),
                reward=tf.expand_dims(trajectory.reward,axis=0),
                step_type=tf.expand_dims(trajectory.step_type,axis=0)
                )
            self._trajectory = self._trajectory.replace(
                action=tf.concat([self._trajectory.action, trajectory.action], axis
    =0),
                discount=tf.concat([self._trajectory.discount, trajectory.discount],
     axis=0),
                next_step_type=tf.concat([self._trajectory.next_step_type,
    trajectory.next_step_type], axis=0),
                observation=tf.concat([self._trajectory.observation, trajectory.
    observation], axis=0),
```

```
34                 policy_info=tf.concat([self._trajectory.policy_info, trajectory.
   policy_info], axis=0),
35                 reward=tf.concat([self._trajectory.reward, trajectory.reward], axis
   =0),
36                 step_type=tf.concat([self._trajectory.step_type, trajectory.
   step_type], axis=0)
37                 )
38 
39             if self.debug: print("[Observer]: Current trajectory:", self._trajectory
   )
40     def get_trajectory(self):
41         return self._trajectory
```