Universidad Politécnica de Cartagena

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicaciones

Aprendizaje reforzado profundo: análisis de posibilidades y tecnología software existente

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE SISTEMAS DE TELECOMUNICACIÓN

Autor: Francisco Jesús Sánchez Rubio
Director: Pedro Sánchez Palma

Cartagena, 18 de abril de 2021

Índice general

Capítulo 1

Resumen

En la actualidad, las redes neuronales que emplean el aprendizaje reforzado encuentran su lugar dentro de una amplia variedad de aplicaciones prácticas. Algunos ejemplos de tales aplicaciones van desde casos bien conocidos como ofrecer experiencias personalizadas de servicios web, a otros más específicos la optimización de controladores de DRAM [1]. Por supuesto también se da el caso de aplicaciones que si bien en primera instancia no parecerían tan útiles, como entrenar una red para aprender a jugar a un determinado videojuego, si que ponen de manifiesto las capacidades del aprendizaje reforzado para aprender a interactuar con un entorno con un nivel considerable de complejidad [2].

Para el caso de este trabajo de fin de grado, se expondrá un ejemplo práctico donde una red neuronal de tipo DQN se entrenará por medio del aprendizaje reforzado profundo para resolver un problema de *beamforming* en el que una agrupación de $K \times M$ trata de maximizar la capacidad del canal con respecto a un emisor móvil empleando únicamente las propiedades de la señal recibida por cada antena de la agrupación.

Capítulo 2

Introducción

El Machine Learning es un campo que actualmente se estudia de forma amplia dentro del conjunto de campos de estudio que abarca la *Computer Science* que está fuertemente arraigado a los principios matemáticos de la estadística [4].

El objetivo consiste en simular una estructura similar a aquella de un cerebro. Esta estructura es conocida como una red neuronal y el objetivo es que sea capaz de aprender a partir de unos determinados datos de entrada y pueda ajustarse a sí misma en base a estos de tal forma que la red sea capaz de "entender" estos datos.

Por "entender" los datos realmente nos referimos a encontrar relaciones y patrones en estos, lo que permite a la red realizar una generalización. Esto quiere decir que la red neuronal después de ser entrenada habrá encontrado una forma de encontrar el comportamiento general que cumplen los datos introducidos durante este entrenamiento. Gracias a poder realizar esta generalización, una red neuronal puede emplearse para realizar principalmente dos funciones, dependiendo por supuesto de la estructura de esta.

Estas dos funciones son: identificar características particulares dentro del conjunto de datos introducidos y predecir una salida en función de un conjunto de parámetros de entrada, en base a lo que la red ha aprendido como un valor de salida razonable para unos valores de entrada.

Con solo estas dos funciones, el rango de posibles aplicaciones es realmente extenso. La identificación de patrones puede encontrar su uso en campos como el procesado de imágenes, identificando rasgos característicos dentro de estas (por ejemplo, rasgos faciales en una cara o los caracteres de un texto manuscrito); e incluso se puede usar como apoyo en otros sistemas de aprendizaje máquina, como reducir la dimensionalidad de un conjunto de datos para reducir la complejidad de la red neuronal que lo procesará (esto se verá más adelante). La predicción de los parámetros

de salida también es de una considerable utilidad cuando trabajamos con problemas de decisión o clasificación (por ejemplo, saber si una reseña es positiva o negativa en función del texto escrito).

En cuanto a las propias neuronas, las podemos considerar como un conjunto de tres elementos que conforman una unidad fundamental de procesado de información. Es decir, son el componente del que están hechas las redes neuronales, de ahí su nombre.

2.1. Tipos de aprendizaje

Un punto importante a tratar cuando hablamos de redes neuronales consiste en los tipos de procesos de aprendizaje que estas pueden emplear. Estos procesos se pueden agrupar en el **aprendizaje supervisado** (Supervised Learning), el **aprendizaje no supervisado** (Unsupervised Learning) y el **aprendizaje reforzado** (Reinforcement Learning) [3].

En el aprendizaje supervisado se considera que existe un componente que actúa de "profesor" para la red, quien posee de antemano conocimiento sobre el entorno en el que trabajará la red. De esta forma durante el entrenamiento este profesor podrá corregir las respuestas de salida de respuesta con las respuestas óptimas restando ambas para obtener una señal de error que se retroalimentará a la red.

Para medir las prestaciones del sistema buscaremos el MSE (*Mean Square Error*), que vendrá definido como una superficie de error multidimensional constituida por la suma de errores cuadrados producidos en las muestra de entrenamiento en función de los pesos de las neuronas, quienes actúan como los parámetros libres del sistema. De esta forma tendremos una superficie de un cierto número de dimensiones donde las coordenadas son los pesos sinápticos que configuran las neuronas de la red. Puesto que esta superficie representa el error resultante para una configuración dada, el objetivo es encontrar un punto mínimo de esta superficie. Para lograr esto se emplea un operador gradiente para obtener la superficie en términos de variaciones, e ir convergiendo de forma iterativa hacia los puntos de mayor descenso.

En el aprendizaje no supervisado no hay un elemento que vigile el proceso de aprendizaje. En vez de eso se deja una medida de calidad de la respuesta del sistema independiente de la tarea a realizar y la red se optimiza para ajustarse a esa medida. La red también debe ser capaz de codificar y representar internamente las características de las entradas, y crear automáticamente las clases que sean necesarias para clasificarlas.

Puesto que no hay un criterio objetivo para determinar la calidad de las respuestas, en la práctica se emplean distintos métodos que tratan de juzgar esta calidad. Algunos ejemplos de estos métodos incluyen: reglas de asociación que encuentren los valores conjuntos del vector de variables $X = (X_1, X_2, ..., X_p)$, análisis de clúster que puede o bien segmentar los datos en agrupaciones o *clústeres* o bien organizar de forma jerárquica dichas agrupaciones, o medios competitivos donde existen capas competitivas cuyas neuronas compiten entre ellas para responder a las entradas.

Un tipo de aprendizaje sin supervisión que además será aquel a tratar en este trabajo es el aprendizaje reforzado. Este tipo de aprendizaje se caracteriza por la interacción continua del sistema de aprendizaje con un entorno que genera una señal de refuerzo primaria. Esta señal se dirige a un "crítico" que la convierte en una señal de refuerzo heurística, y es retroalimentada de vuelta al sistema.

El objetivo consiste en minimizar una función de coste que viene definida por el coste acumulativo de las acciones tomadas por el sistema durante una determinada secuencia de pasos dentro del entorno. Mediante este método el sistema deberá ser capaz de encontrar patrones y características dentro del entorno que le permitan llegar a la secuencia de acciones óptima que obtenga los mejores resultados dentro de ese entorno.

2.2. Entrenamiento e hiperparámetros

Partiendo de todo este conjunto de definiciones expuesto, es fácil ver como el tipo de problema o entorno al que tratamos de abordar mediante una red neuronal condicionará no solo la arquitectura de esta, sino además la forma en la que la entrenamos y validamos.

Un detalle que también hay que tener en cuenta durante el entrenamiento es que si el tiempo dedicado a esta tarea es demasiado elevado en relación al número de muestras de entrenamiento disponible, la red sufrirá de sobre-ajuste a los datos de entrenamiento. Es decir, el error cometido para las muestras de entrenamiento se aproximará a cero a lo largo del tiempo porque las acaba memorizando; pero al presentarle una muestra que no esté incluida en el conjunto de entrenamiento el error cometido termina aumentando. Esto presenta un problema ya que buscamos que con la generalización la red minimice el error en la salida para entradas desconocidas, o error de test.

Por tanto existirá una duración de entrenamiento óptima para una cierta cantidad de muestras de entrenamiento disponible que minimice el error de test. Una forma

frecuente de estimar cuando la red ha alcanzado ese punto es emplear un conjunto de muestras de validación. Este conjunto consiste en una cantidad de muestras que no se incluyen en el conjunto de entrenamiento pero que se van mostrando paulatinamente a lo largo del proceso de entrenamiento. Mediante este conjunto se ponen a prueba las prestaciones de la red frente a una muestra que no ha podido memorizar previamente por lo que nos da una métrica real de la calidad de la capacidad de generalización de la red en forma de error de validación.

Al intercalar muestras de validación a lo largo del proceso de entrenamiento se obtiene una representación de la evolución error cometido por la red en función del número de épocas de entrenamiento. Como el error de validación no está sesgado por el sobreajuste de entrenamiento este error es una representación fiable del error de test que cometería la red, así que si se representara el error de validación en función del número de muestras de entrenamiento empleadas el punto de sobreajuste se vería claramente como el mínimo de la función. Con esta información es fácil encontrar en qué punto se debe finalizar el entrenamiento, porque el momento en el que la función de error deje de ser decreciente equivale al momento en el que la red se está sobre-ajustando.

El conjunto de muestras de validación surge de dividir un conjunto de muestras inicial en el subconjunto de validación y el subconjunto de entrenamiento. La proporción de muestras que se transfieren a cada subconjunto se considera un hiperparámetro. Esto es, un parámetro que no es ajustado como parte de la red neuronal, pero rigen la topología de la red y el proceso de aprendizaje y existen dos tipos: hiperparámetros de modelo e hiperparámetros de algoritmo, que ajustan respectivamente las características ya mencionadas.

La calibración de estos hiperparámetros se realiza antes de iniciar el entrenamiento de la red y hay numerosas estrategias para esta tarea. Por nombrar algunos ejemplos tenemos la optimización BlackBox, Hyperband, optimización bayesiana o *random search*.

Existen métodos adicionales que ayudan a lograr el comportamiento de generalización. Uno muy frecuente, efectivo y de fácil implementación es el *droput* probabilístico de nodos. Este método se basa en que dada una capacidad de computación ilimitada, la mejor regularización posible para una red de tamaño fijo será la media de las predicciones de todas las configuraciones posibles de parámetros donde el peso de cada configuración es su probabilidad posterior dados los datos de entrenamiento. Dicho de otro modo, hacer la media geométrica de las predicciones producidas por un número elevado de redes de distinta topología.

Naturalmente en la práctica la capacidad de computación es limitada y este acercamiento es inviable para modelos demasiado extensos o complejos, además de que se necesitaría una gran suma de muestras de entrenamiento para todas estas redes. Ahí es donde entra la técnica del *dropout*, que consiste en que exista una cierta probabilidad de ignorar nodos y sus conexiones en las distintas capas de la topología durante el entrenamiento. Esto aproxima el comportamiento obtenido al caso ideal de entrenar y promediar distintas topologías por lo que se reduce el sobre-ajuste y mejora la generalización. La probabilidad de que un nodo en particular sea ignorado depende del hiperparámetro p [6].

Otro método para abordar la generalización es la regularización de pesos por L1 o L2. Estos métodos se basan en asumir la existencia de pesos con valores muy elevados es un indicador de que puede haber sobre-ajuste, por lo que el valor absoluto de los pesos se suman a la función de coste de la red y se define un hiperparámetro λ para ajustar el efecto de la penalización. La diferencia entre L1 y L2 reside en que en L1 la penalización es lineal mientras que en L2 los pesos penalizan de forma cuadrática [7].

Un ejemplo de como funcionaría una red neuronal en un entorno práctico sería un problema de *beamforming*. El *beamforming* consiste en usar una agrupación de antenas emitiendo simultáneamente (ya sea en un contexto acústico o electromagnético) donde la señal de cada antena recibe una alimentación distinta [16]. De esta forma se construye un conjunto de lóbulos orientados a una determinada dirección. Esta técnica se suele emplear en sistemas de detección tales como el radar o el sónar, donde se pretende que el lóbulo principal de la agrupación apunte a un objetivo.

Esta tarea no es sencilla, ya que además del ruido y no sabemos de antemano de donde proviene la señal del objetivo. Es aquí donde se puede usar una red neuronal que pueda hacer una correlación entre las señales recibidas por cada antena de la agrupación y apuntar correctamente a la fuente de la señal. Más adelante se desarrollará un ejemplo detallando esta aplicación en concreto.

Capítulo 3

Objetivos

El objetivo de este trabajo consiste en realizar un análisis del estado actual de la tecnología del software llevado a la aplicación del aprendizaje reforzado profundo, centrándose en las librerías Tensorflow y Keras e realizando su implementación mediante el lenguaje de programación Python. El objetivo de esta implementación es demostrar las capacidades de esta tecnología a través de la resolución de un problema práctico perteneciente al ámbito de la ingeniería de sistemas de telecomunicaciones.

En lo referente al problema práctico, como ya se mencionó anteriormente en el resumen este se trata de un problema de *beamforming*. En esencia, cada antena viene caracterizada por su diagrama de radiación, que representa la directividad de esta antena con respecto a cada dirección posible (en coordenadas polares y a valores logarítmicos normalizados). Cuando tenemos un conjunto o agrupación de antenas el diagrama de radiación de dicha agrupación dependerá de las aportaciones de cada una de las antenas que la constituyen. Esto se debe a que las ondas emitidas por cada antena individual interferirá con el resto de ondas, y las características de esta interferencia vendrán con una fuerte dependencia con la alimentación de las antenas; que en este caso son la amplitud y fases iniciales de la corriente de alimentación.

Según la distribución de corrientes de alimentación, se pueden lograr ciertos tipos de interferecias entre las señales que dan lugar a patrones de radiación altamente directivos en direcciones concretas. Esto es algo beneficioso ya que nos inidica que la mayor parte de la potencia entregada a la antena terminará concentrada en la dirección deseada (dirección en la que, idealmente, existirá un receptor para la señal enviada), por lo que la relación de señal a ruido aumentará con respecto a patrones menos directivos y se hará un uso eficiente de la potencia disponible.

Evidentemente, mejorar la relación de señal a ruido es algo que interesa mucho en un sistema de telecomunicaciones puesto que se trata de un importante factor de

calidad que condiciona diferentes prestaciones del sistema, tales como la distancia máxima entre el emisor y el receptor o la capacidad del canal.

Asumiremos una agrupación de $K \times M$ antenas receptoras y un emisor colocado a una cierta distancia de la antenas que se desplaza aleatoriamente a lo largo del tiempo. El objetivo es entrenar una DQN que sea capaz de escoger un vector de beamforming dentro de un rango de posibles vectores de tal forma que la capacidad del canal entre el emisor y el receptor se maximice dada una determinada posición del emisor en un instante de tiempo, representando el estado del entorno.

Se asume que en cada instante de tiempo, configurado a 1 micro-segundo, el emisor envía una señal piloto en forma de una onda sinusoidal a 5 GHz con una determinada amplitud y fase. Debido a que por los efectos de la propagación cada antena individual de la agrupación receptora verá un valor de amplitud y fase distintos, este conjunto de parámetros será la forma que tendrá la red de "ver" el entorno y representará la entrada de la DQN. Adicionalmente el conjunto de valores de amplitud y fases recibidas se podrán emplear para estimar la matriz de coeficientes del canal.

Más allá de esta demostración, la intención final es permitir que el trabajo expuesto a continuación pueda servir en el futuro próximo como una introducción rápida y práctica al campo del aprendizaje reforzado profundo para quienes se interesen en él. Por esta razón en el trabajo se analizarán no solo los resultados o el proceso de implementación, si no que también se harán incisos introduciendo los conceptos teóricos que existen tras los algoritmos aplicados.

Capítulo 4

Estado del arte

En la actualidad existen numerosas herramientas a disposición del aprendizaje máquina con las cuales se permite abordar un amplio abanico de problemas.

4.1. Aprendizaje reforzado

El aprendizaje reforzado es un tipo de aprendizaje no supervisado que funciona a partir de interactuar de forma continuada con un entorno con el objetivo de minimizar un cierto índice escalar que representa la calidad de los resultados de la interacción entre la red y el entorno.

Existen dos tipos de formas de afrontar el aprendizaje reforzado [3]:

- La forma clásica, que se basa en la aplicación de recompensas y castigos durante el entrenamiento con tal de conseguir una red cuyo comportamiento se especialice en una tarea concreta.
- La forma moderna, donde se introduce el concepto de *programación dinámica* tratando de generar un comportamiento donde la red tome decisiones considerando los posibles futuros estados que se puedan producir sin haber pasado por ellos. En otras palabras, consiste en dotar a la red de la capacidad de planificar

4.2. Redes convolucionales

En función de la organización de las neuronas dentro de una red neuronal, la red puede clasificarse en distintas clases [3]:

- Las **Single-Layer Feedforward Networks** (Redes de Proalimentación monocapa): En este modelo las neuronas se agrupan en capas donde la información se transmite de forma unidireccional hacia la capa siguiente. Además en este caso solo existe una única capa de neuronas que están conectadas directamente a los nodos de entrada y sus respectivas salidas representan las salidas de la red.

- Las **Multilayer Feedforward Networks** (Redes de Proalimentación Multicapa): Modelo similar a la red monocapa, pero en este caso existe al menos una capa extra entre los nodos de entrada y las neuronas de salida. A estas capas intermedias se las conoce como capas ocultas o *hidden layers*, ya que no se pueden observar ni desde la entrada ni desde la salida. La función de dichas capas es permitir a la red extraer información estadística de órdenes superiores de sus entradas, que a efectos prácticos permite a la red aprender y modelar comportamientos más complejos. En términos de nomenclatura, a la primera capa oculta se la conoce como capa de computación y para referirse a la arquitectura se la nombra en función del número de nodos de cada capa siguiendo el patrón $m-h_1-h_2-...-h_n-q$ donde m y q representan el número de nodos en la entrada y neuronas en la salida respectivamente y cada h_i es el número de neuronas de cada capa oculta i.

- Las **Recurrent Networks (redes recurrentes)**: A diferencia de las redes de pro-alimentación, en este tipo de redes existe al menos un bucle de retroalimentación donde hayan neuronas cuyas salidas estén conectadas a las entradas de otras neuronas o nodos. Las consecuencias de esta retroalimentación afectan en gran medida al comportamiento de la red especialmente si la retroalimentación incluye elementos de retardo de unidad temporal Z^{-1}, pues permitiría comportamientos dinámicos no lineales. En otras palabras, tendríamos una red capaz de recordar información de entradas anteriores. Si bien las redes de pro-alimentación también "recuerdan" eventos anteriores, estas están limitadas a recordar en base a los datos de entrenamiento mientras que las redes recurrentes pueden recordar información de la información que hayan procesado después de haber sido entrenadas.

En el caso de este trabajo, el tipo de red que se tendrá más en cuenta son las **redes convolucionales** o **CNN** (*Convolutional Neural Network*), que son un tipo particular de redes neuronales que clasificaríamos dentro de las redes de proalimentación multicapa, y se dedican a procesar datos organizados con una topología de rejilla.

Como bien indica su nombre, el operador que caracteriza este tipo de redes es la convolución, que reemplaza a la multiplicación matricial que emplean otros tipos de redes neuronales. Una red neuronal se considera convolucional cuando al menos una de sus capas es una capa convolucional [5].

La convolución es una operación lineal entre dos funciones cuyos argumentos están en el dominio de los números reales. Se define según la ecuación 4.1 donde la función x es la entrada y a la función w se la llama *kernel*. La salida $s(t)$ también es conocida como mapa de características.

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{4.1}$$

Dentro del contexto del aprendizaje máquina, tanto la función de entrada como el *kernel* pueden ser arrays multidimensionales o tensores. Teniendo esto en cuenta, la convolución puede realizarse sobre varios ejes simultáneamente. En el caso de este trabajo, se ha hecho uso de la convolución bidimensional donde la entrada consiste en dos matrices (que en lo referido a la topología corresponden a dos rejillas de dos dimensiones) y el *kernel* también se ha tomado con dos dimensiones. Además hay que tener en cuenta que los valores de ambas funciones están discretizados, por lo que la integral de la ecuación 4.1 se reemplazará con un sumatorio. La expresión final quedaría con la expresión:

$$s(t) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-n, j-n) \tag{4.2}$$

En general las redes convolucionales tienen una conectividad esparcida dado que su *kernel* es de menos tamaño que la imagen. La motivación tras esto se debe a que el número de puntos necesarios para detectar las características que se pretenden extraer de la entrada (p.e. los bordes de los elementos de una imagen) representan solo una fracción reducida del total por lo que es mucho más eficiente limitar el número de conexiones de cada neurona.

En términos de rendimiento la multiplicación matricial tiene un coste computacional de $O(m \times n)$, pero al usar este método el coste cambia a $O(k \times n)$, donde $k < m$. En función de la diferencia entre estos parámetros las ganancias de rendimiento pueden incrementarse de forma drástica. Por supuesto va a haber una fuerte dependencia con la aplicación de la CNN, pero existen casos donde k puede estar varios órdenes de magnitud por debajo de m y aún así tener una red que presenta buenas presentaciones.

Para este trabajo la conectividad esparcida se ha empleado durante la implementación de forma implícita al usar *kernels* de tamaño menor a las matrices de entrada: 4x4 en la primera capa y 3x3 en la segunda.

Aunque existen otras prácticas en las redes convolucionales, como el *pooling*, estas no se han puesto en práctica en la red de este trabajo. De cara al futuro queda pendiente

4.3. Beamformer uniforme

Una posible aplicación práctica para las redes neuronales a la que se entrará en mayor profundidad más adelante es el *beamforming*.

La necesidad de la técnica del *beamforming* surge de la presencia de factores interferentes en la adquisición de señales transmitidas por medios no guiados. Este problema es bastante prominente en el ámbito de la entrada de señales acústicas en una agrupación de micrófonos, como el que puede encontrarse en el interior de un teléfono móvil actual. Las distintas fuentes de ruido presentes en las proximidades de la agrupación, así como los posibles fenómenos de reverberación, decrementan la calidad de la señal acústica adquirida. Por esta razón se emplean métodos de muestreo espacial basado en el empleo de múltiples receptores: se puede detectar la presencia de varias fuentes de sonido para atenuar y/o amplificar la recepción de forma selectiva.

La forma más sencilla de *beamforming* es el método de DSB (*delay-and-sum*), que consiste en aplicar retardos progresivos a las señales recibidas por cada receptor de la agrupación de tal forma que se aumente considerablemente la directividad en una dirección en concreto mientras se reduce en el resto; idealmente esta dirección será aquella de donde proviene la señal útil. La consecuencia de esto es que al sumar las señales recibidas solo las que provengan de la dirección elegida se sumarán de forma coherente. Esto causa que la señal recibida por la dirección escogida tenga una amplitud mucho mayor que las aportaciones del resto de direcciones, que se sumarán de forma interferente.

El problema con este acercamiento reside en que el *beamforming* se realiza sobre señales digitalizadas, que son discretas en el dominio temporal. Por tanto se introducen errores de cuantización en el retardo progresivo que tienen que mitigarse con interpoladores, e incluso entonces la efectividad variaría con el tipo de interpolación escogida.

Una solución posible es usar el *beamformer* convencional o de Barret, que actúa en el dominio de la frecuencia. Gracias a cuantizar la frecuencia en lugar del tiempo este último no esta limitado a pasos discretos y lo que realmente nos interesa es la diferencia de fases entre las señales recibidas. Aún así, precisamente por usar las fases de las señales, este sistema sufre de *aliasing* espacial debido a que existirán frecuencias (dadas en función de la geometría de la agrupación) que produzcan diferencias de fases nulas entre receptores. Además las prestaciones del sistema para frecuencias bajas es menor, debido a que el ancho del lóbulo principal es demasiado ancho y a que la diferencia de fase entre señales puede llegar a ser mínima.

Otra solución es recurrir al *beamformer* superdirectivo, en particular el MVDR (*Distortionless Response Beamformer*) o *beamformer* de Capon. El funcionamiento de este tipo de *beamformer* se basa en aproximar por medios estadísticos el campo de ruido presente y usar el modelo resultante para minimizar las aportaciones a la señal que vengan de direcciones distintas a la escogida. Esta solución tampoco está exenta de inconvenientes, pues sigue teniendo problemas con la anchura del lóbulo principal a bajas frecuencias.

Una solución que sí involucra el uso del aprendizaje máquina consiste en entrenar una red neuronal que tome toda la información recogida (el ruido de fondo, la propia señal útil y las posibles interferencias) por los receptores para crear un vector de *beamforming* dinámico que varíe de acuerdo a las característias de la señal. Con esto es posible filtrar las señales provenientes de un rango de direcciones, creando un patrón de *beamforming* uniforme en dicho rango mientras que se atenúa el resto del espacio [8].

Si bien este método es más afín al aprendizaje supervisado, si que ha servido de base para el desarrollo del entorno de simulación que sí se ha utilizado en el contexto del aprendizaje reforzado profundo.

4.4. Beamformer con escaneado profundo

Este *beamformer* funciona a partir de emplear una red neuronal convolucional entrenada por aprendizaje profundo reforzado para interpretar los valores de amplitud y fase recibidos por las antenas de la agrupación, generados por la señal piloto de un emisor, y a partir de ahí escoger el vector de *beamforming* que mejor se ajuste a la posición del emisor con tal de maximizar la capacidad del canal entre este y el receptor [10].

En el caso de este trabajo, se ha optado por asumir que el emisor contiene una

única antena, pero este emisor es capaz de moverse libremente por el espacio siguiendo trayectorias aleatorias. Con esto se pretende comprobar si es posible para la una red DQN predecir y escoger el vector de *beamforming* más óptimo para un emisor infiriendo su posición en un momento dado en base a las señales recibidas en este instante y la propia memoria que ha ido adquiriendo la red en base a experiencias anteriores.

Capítulo 5

Desarrollo

Para poner en práctica las técnicas expuestas dentro de un entorno práctico, la implementación ha sido realizada en el lenguaje de programación Python con la ayuda de los paquetes de librerías Tensorflow y Keras. Estas han resultado herramientas extremadamente útiles flexibles a la hora de implementar rápidamente una red neuronal funcional al incorporar tanto un amplio rango de módulos y funciones, que abstraen una porción importante del trabajo que conlleva programar una red neuronal, como el acceso a una gran variedad de *datasets* para entrenar dicha red.

Por conveniencia, el código de cada ejemplo se ha realizado sobre un cuaderno de Jupyter ya que es mucho más fácil de manejar para el *debugging*, organizar el código en pequeños bloques y facilita el compartir el código.

5.1. Beamforming basado en retardos de señales recibidas

Una forma de afrontar el beamforming está más centrada en el procesado digital de señales recibidas en lugar de ajustar los diagramas de radiación. Dicha forma está basada en el retardo de señales incidentes.

En su modelo más sencillo se asume una agrupación lineal de M receptores cuyas señales recibidas están designadas por el vector **x**, que también se puede expresar como $x_0(t), x_1(t), ..., x_{M-1}(t)$. Estos receptores vendrán equiespaciados por una distancia d. A esta agrupación llega la señal $s(t)$, que indice con un ángulo θ sobre la normal de la agrupación. Debido a esto la señal llegará primero al receptor $x_0(t)$, y alcanzará cada receptor sucesivo con un retardo fijo τ. De esta forma las señales re-

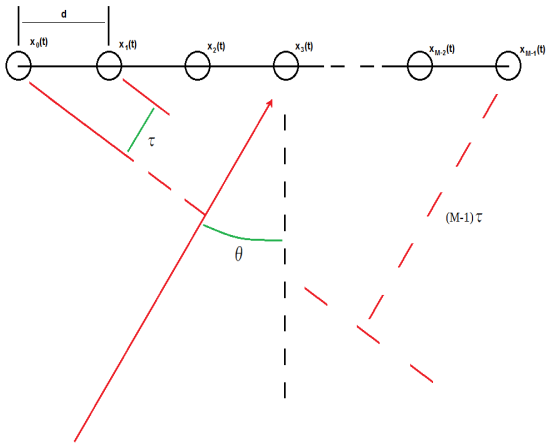

Figura 5.1: Recepción de una señal por una agrupación lineal de receptores

cibidas se expresan de forma general como $x_m(t) = s(t - m\tau)$, donde τ dependerá del ángulo de incidencia θ, la distancia d entre receptores y la velocidad de propagación v_p de la señal física: $\tau = \frac{d}{v_p} \sin\theta$.

En el modelo de *beamformer* DSB basado en el dominio temporal asumimos que cualquier señal $x_m(t)$ ya está digitalizada. Cada elemento del vector de señales $\mathbf{x}$ se introduce por un retardo dependiente de τ, el cual es progresivamente menor para las señales que mayor retardo han sufrido para llegar al sistema. Estos retardos se introducen con el objetivo de compensar los retardos producidos por el ángulo de incidencia y ponen todas las señales de entrada en fase. A continuación se aplica a la salida de los retardos un vector de pesos $\mathbf{w}$, también de longitud M. Por último se suman las señales resultantes, que al estar en fase se aplicará una suma coherente.

Como la suma coherente depende de que los retardos del sistema estén configurados para un retardo τ concreto, el cual depende del ángulo de incidencia θ de la señal de entrada, se puede ver como ajustando el valor de τ realmente estamos ajustando el ángulo para el que el *beamformer* espera recibir la señal de entrada.

Como se puede ver en la imagen, la señal de salida será la suma de las aportaciones de cada entrada. Analíticamente la salida tiene la forma de la ecuación 5.1:

$$y(t) = \sum_{m=0}^{M-1} (t - (M - m - 1)T) \tag{5.1}$$

Donde $T = \tau$ para el ángulo θ que produzca el retardo τ.

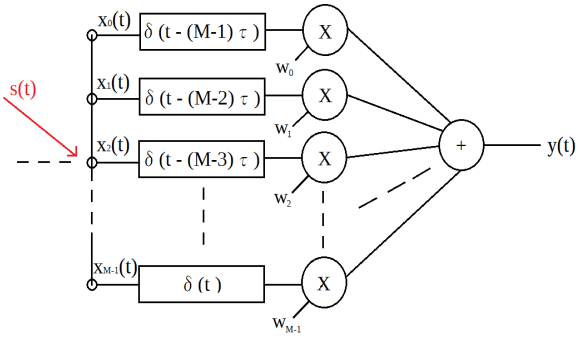

Figura 5.2: Beamformer DSB

5.2. Desarrollo teórico

Aunque en el caso de este trabajo, la forma que realmente nos concierne es la ya mencionada modificación de vectores de *beamforming* a través de la alimentación de las antenas. Pasando ya a una agrupación más compleja, se tiene un conjunto de antenas equiespaciadas distribuidas de forma horizontal: es decir, una agrupación K x M. En este tipo de agrupaciones podemos encontrar otra forma de realizar el beamforming a partir de jugar con la amplitud *a* de cada elemento de la agrupación. Si ajustamos la amplitud y fase de cada antena de tal forma que se cumpla la relación de la ecuación 5.2:

$$a_{k,m}(\phi_0, \theta_0) = e^{-j\frac{2\pi}{\lambda}}(y_k sin(\theta_0)cos(\phi_0) + z_m sin(\theta_0)sin(\phi_0)) \tag{5.2}$$

```
def make_beamformer_vectors(theta=0,phi=0):
    v = np.zeros([N_RECEIVERS_Y*N_RECEIVERS_Z],dtype=complex)

    for i in range(0,len(RECEIVER_POS_Y)):
        for j in range(0,len(RECEIVER_POS_Z)):
            v[len(RECEIVER_POS_Z)*i + j] =
                np.exp(-1j*(2*PI/WAVELENGTH))*
            (RECEIVER_POS_Y[i]*np.sin(theta)*np.cos(phi) +
                RECEIVER_POS_Z[j]*np.sin(theta)*np.sin(phi))

    return v
```

podremos conseguir un conjunto de valores que correspondería con “orientar” la

agrupación de tal forma que caiga un máximo del diagrama de radiación en el acimut ϕ_0 y elevación θ_0. Esto no quiere decir exista un único máximo en el diagrama de radiación de la agrupación, ya que existirán múltiples lóbulos secundarios de igual o menor amplitud cuyas características dependen no solo de todo el apartado de características electromagnéticas de la agrupación y la frecuencia de trabajo, sino además de sus dimensiones físicas.

Por tanto si quisieramos que una agrupación rectangular apuntase a una dirección concreta, la alimentación de sus antenas constituirá el vector de *beamforming* $\mathbf{v}(\phi_0, \theta_0) = [a_{1,1}(\phi_0, \theta_0), a_{1,2}(\phi_0, \theta_0), ..., a_{k,m}(\phi_0, \theta_0)]^T$. En esta ecuación los términos y_k y z_m corresponden con las posiciones de cada antena individual de la agrupación estando esta centrada en el origen de coordenadas y distribuida sobre el plano YZ.

Las ecuaciones de posición de las antenas se expresan como: $y_k = \frac{\lambda}{2}(k - \frac{(K+1)}{2})$ y $z_m = \frac{\lambda}{2}(m - \frac{(M+1)}{2})$, donde k y m son los índices de cada antena en los ejes Y y Z, respectivamente. Resolviendo estas ecuaciones para distintos valores de k, m muestra que la diferencia de posiciones entre una antena y otra en el mismo eje es de $\frac{\lambda}{2}$ por lo que podemos concluir con que al final tendremos lo que se conoce como una agrupación uniforme.

Sabiendo esto ya somos capaces de definir una agrupación de antenas con la que podemos construir un vector de *beamforming* de tal forma que se apunte a cualquier dirección del espacio, expresando esta dirección en términos de ángulos de acimut y elevación dentro de un sistema de coordenadas esférico.

A continuación se caracterizará la transmisión de la señal del emisor. Puesto que la distancia entre el emisor y el receptor pueden estar a cualquier distancia uno de otro no se asumirá que la onda incidente se ajusta al modelo de onda plana por lo que se considerará una onda esférica, la cual atiende al desarrollo de las ecuaciones 5.3 y 5.4 [9]:

```
def receive_signals(amp,r,vp,t0,noise):

    pulse = 2 * PI * FREQUENCY
    sig = np.zeros([len(r),TIME_RESOLUTION])
    amp_div_r = np.diag(np.divide(amp,r))
    r_div_vp = np.divide(r,vp)
    disttime_difference = np.zeros([len(r),TIME_RESOLUTION])

    for i in range(0,len(r)):
        disttime_difference[i,:] = r_div_vp[i] - (t0 + TIME_AXIS)
```

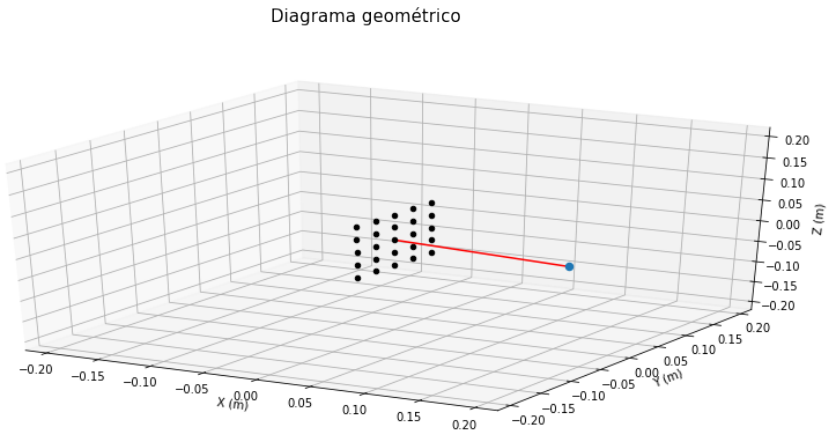

Figura 5.3: Posición de las antenas en un espacio tridimensional

```
arg = pulse * disttime_difference
sig = np.matmul(amp_div_r,np.cos(arg)) + noise

return sig,sig[:,0],(arg[:,0]%(2*PI))
```

$$f(r,t) = A(r)\cos(kr - \omega t) = \frac{A_0}{r}\cos(\frac{2\pi}{\lambda}r - \omega t) = \frac{A_0}{r}\cos(\frac{\omega}{v_p}r - \omega t) \tag{5.3}$$

$$\frac{A_0}{r}\cos(\frac{\omega}{v_p}r - \omega t) = \frac{A_0}{r}\cos(\omega(\frac{r}{v_p} - t)) \tag{5.4}$$

Siendo las constantes A_0 la amplitud inicial de la onda, ω la pulsación angular, v_p la velocidad de propagación, y con r y t como los parámetros de distancia e instante de tiempo, respectivamente.

Se puede expresar la señal recibida mediante la expresión:

$$\mathbf{r} = H\mathbf{p} + \mathbf{n} \tag{5.5}$$

Donde el vector **r** representa la señal que recibe cada antena individual, la matriz H contiene los coeficientes del canal y el vector **n** define el ruido introducido durante la transmisión, el cual a efectos de esta demostración práctica se considerará como un ruido blanco gaussiano de media y desviación típica δ de valor 1.

Hay que mencionar que los valores observados en el vector **r** únicamente contienen los valores de módulo y fase de la señal en el instante de tiempo en el que esta llega al receptor, expresados en forma de un número complejo en forma binaria.

Al conocer tanto los parámetros iniciales de la señal piloto enviada por el emisor como la señal recibida por cada antena de la agrupación podemos hacer una estimación, si bien un tanto idealizada, de la matriz de coeficientes H dividiendo cada elemento del vector **r** por la señal piloto original puesto que asumimos que solo existe una única señal piloto. Tampoco se considerará el ruido puesto que en un entorno práctico no sabríamos la aportación exacta de este a la amplitud y fase de la señal recibida en el instante en el que esta es recibida. El vector resultante de esta división será transformado en una matriz diagonal.

Sabiendo el vector de *beamforming* y la matriz de coeficientes de canal, podemos calcular la capacidad del canal entre el emisor y el receptor mediante la ecuación 5.6:

$$C = W \log_2 (1 + \frac{E_S}{N_{UE}\sigma^2}\mathbf{v}^\dagger \mathbf{H}\mathbf{H}^\dagger \mathbf{v}) \tag{5.6}$$

```
def compute_reward(v,H):
    v = np.transpose(v)
    H = np.diag(H)

    log_term = np.real((TRANSMISSION_POWER/NOISE_VARIANCE)*
        np.matmul(np.matmul(np.matmul(np.conjugate(v),H),np.conjugate(H)),v))

    if np.isnan(log_term):
        print(v)

    if log_term <= -1:
        return 0

    C = REWARD_SCALING * np.log2(1 + log_term) + 1e-12

    return C
```

Donde W es el ancho de banda, E_S la potencia de transmisión, N_{UE} el número de antenas del emisor, σ^2 representa la varianza del ruido y los vectores **v** y **H** constituyen el *beamforming* y la matriz de coeficientes, respectivamente.

```
class environment:
    # Inicializamos con las coordenadas, la velocidad, el instante de
        tiempo
    # y el espacio de acciones
    def __init__(self):
```

```
        self.coords = ENV_EMITTER_INITIAL_COORDS
        self.motion_multiplier = ENV_EMITTER_VELOCITY_FACTOR
        self.current_timestep = 0

        self.action_space = A

        self.r,self.H = make_frame(0)
        self.v = self.action_space[:,np.random.randint(N_ACTIONS)]

    # Obtemenos el estado del entorno a partir de multiplicar el
        vector
    # de beamforming con la senal recibida
    def get_state(self):

        state = np.zeros([1,N_RECEIVERS_Z,N_RECEIVERS_Y,2])

        s_pre = np.conjugate(self.v)*self.r

        amps = np.reshape(np.abs(s_pre),[N_RECEIVERS_Z,N_RECEIVERS_Y])
        phases =
            np.reshape(np.angle(s_pre),[N_RECEIVERS_Z,N_RECEIVERS_Y])

        state[0,:,:,0] = amps
        state[0,:,:,1] = phases

        return state

    # Reiniciamos el entorno y devolvemos el estado inicial
    def reset(self):
        self.coords = ENV_EMITTER_INITIAL_COORDS
        self.current_timestep = 0

        self.v = self.action_space[:,np.random.randint(N_ACTIONS)]
        self.r,self.H = make_frame(0)

        return self.get_state()

    # Damos un paso en la simulacion
    def step(self,action):
```

```
        self.v = action

        self.motion =
            2*self.motion_multiplier*(np.random.random(3)-0.5)
        self.coords += self.motion
        self.current_timestep += 1

        self.r,self.H =
            make_frame(self.current_timestep*ENV_TIMESTEP,coords=self.coords)

        new_state = self.get_state()
        reward = compute_reward(self.v,self.H)

        return new_state,reward, {}
```

5.3. Construcción de la DQN

Con toda esta información ya tenemos todo lo necesario para construir y entrenar nuestra red DQN. Lo primero será definir la estructura de la red, representada gráficamente en la figura 5.1: tendremos una entrada conformada por dos matrices de $M \times K$ conteniendo la información de módulo y fase de cada antena y estas pasan por dos capas consecutivas de convolución espacial bidimensional, cuya salida es a continuación aplanada e introducida a otras dos capas consecutivas densamente conectadas. Cabe destacar que cada capa a excepción del aplanamiento contiene una función de activación de tipo *ReLU*.

Dado que consideramos dos matrices a la entrada de similares tamaños y nos interesa hacer uso de la correlación entre la información de módulos y de fases, nos conviene emplear capas de convolución ya que su función es precisamente la de encontrar correlaciones espaciales locales [13], de acuerdo al tamaño del filtro de convolución y usar esta información para ser capaz de extraer características. La primera capa de convolución tendrá una profundidad de 32 y poseerá un filtro espacial de tamaño 4×4. En la segunda capa habrá una profundidad de 64 y usará un filtro de 3×3. Para ambos casos el paso de los filtros (o *stride*) será de 2 posiciones.

A continuación la salida de la segunda capa de convolución se aplana en un único vector para ser introducida a una capa densamente conectada de 512 neuronas. Tras

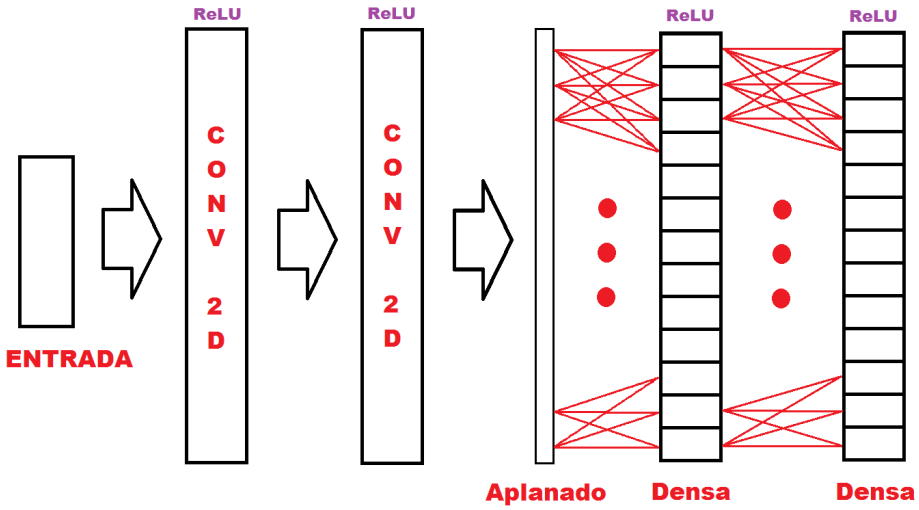

Figura 5.4: Arquitectura de la DQN

esta capa densa se sitúa la capa de salida, con la que también mantiene una conexión total. La cantidad de neuronas de esta última capa será igual al número de acciones posibles de la DQN, que en este caso será el número de vectores de *beamforming* posible.

La DQN está programada para usar el optimizador *Adam*, regulado por el hiperparámetro *learning rate*.

Una particularidad importante de la red es que es usará dos modelos de DQN similares pero separados entre sí. El objetivo de esto es que mientras un modelo de predicción se encarga de realizar las predicciones necesarias para escoger las acciones apropiadas, el otro modelo (modelo objetivo) trata de predecir qué acción queremos que realice el modelo. La razón por la que se realiza esto es debido a que esta aproximación permite que la red converja incluso cuando la complejidad del entorno se incrementa [11].

5.4. Agente DQN

En lo que se refiere al agente con el que se realizará el entrenamiento de la DQN, las interacciones del entorno vienen definidas según las ecuaciones 5.7, 5.8 y 5.9 .

$$s^{(t)} = y = v^{\dagger} r \tag{5.7}$$

$$a^{(t)} = v \tag{5.8}$$

$$d_{(a|s)}^{(t)} = C \tag{5.9}$$

Con tal de controlar el proceso de entrenamiento se asignarán manualmente los hiperparámetros: γ constituirá la penalización o descuento aplicada a las predicciones pasadas, ϵ representa la parámetro del método de decisión con el que la red regula el proceso de exploración/explotación (es decir, si probamos con acciones aleatorias para conseguir experiencia o predecimos a partir de las experiencias conocidas), y τ, que entra en juego durante el entrenamiento de los pesos donde los pesos del modelo objetivo se ajustan a un valor intermedio entre los suyos propios y los pesos del modelo de predicción en función de este valor.

La función de pérdida aplicada se define en la ecuación 5.10, donde el término $d_{(a|s)} + \gamma max_{a'}\hat{Q}(s, a')$ representa el objetivo a ajustar y $Q(s, a)$ representa la predicción de la red.

$$L = (d_{(a|s)} + \gamma max_{a'}\hat{Q}(s, a') - Q(s, a))^2 \tag{5.10}$$

El agente dispondrá de la capacidad de recordar interacciones anteriores. Esto se logra reservando una determinada cantidad de memoria donde se irán guardando los resultados de cada acción. Estos datos serán el estado del entorno, la acción realizada, la recompensa obtenida y el nuevo estado tras la interacción, en ese orden.

En el caso de este trabajo esto será una cola de 100 posiciones puesto que a medida que vamos interactuando las experiencias pasadas van cobrando cada vez menos importancia debido al factor de descuento y a largo plazo terminan siendo irrelevantes. Es por esto que aprovechamos la estructura del encolado con tamaño máximo: porque si introducimos nuevos valores cuando la cola está llena los primeros valores son descartados.

Tras realizar una acción y recordar los resultados se aplica sobre el agente el proceso de aprendizaje: se van tomando muestras de las experiencias adquiridas y se predice el siguiente valor de Q, y a continuación se actualizan los pesos de ambos modelos.

```
class DQN:
    def __init__(self,env):
        self.env = env # Asignacion del entorno
        self.memory = deque(maxlen=100) # Memoria de la red

        self.gamma = DISCOUNT_FACTOR # Penalizacion de experiencias
            pasadas
```

```
        self.epsilon = EPSILON_START # Maximo epsilon
        self.epsilon_min = EPSILON_END # Minimo epsilon
        self.epsilon_decay = EPSILON_DECAY # Ratio de decrecimiento de
            epsilon

        self.learning_rate = LEARNING_RATE # Hiperparametro del
            optimizador Adam
        self.tau = TAU # Influencia de pesos entre red
            predictora/objetivo

        self.model = self.make_network(debug=True) # Asignacion de la
            red principal
        self.target_model = self.make_network() # Red secundaria para
            la convergencia

    # Crea la arquitectura de la red neuronal mediante tf.keras
    def make_network(self,learning_rate=LEARNING_RATE,
        input_shape=(N_RECEIVERS_Z,N_RECEIVERS_Y,2),debug=False):

        model_input = tf.keras.Input(shape=input_shape)

        if debug:
            print("Entrada completada: " + str(model_input))

        layer1 = keras.layers.Conv2D(
            32,4,strides=2,padding="same",kernel_initializer=
            keras.initializers.VarianceScaling(scale=2.),
            activation='relu',use_bias=False,name="CAPA1")(model_input)

        if debug:
            print("Capa 1 completada: " + str(layer1))

        layer2 = keras.layers.Conv2D(
            64,3,strides=2,kernel_initializer=
            keras.initializers.VarianceScaling(scale=2.),
            activation='relu',use_bias=False,name="CAPA2")(layer1)

        if debug:
            print("Capa 2 completada: " + str(layer2))
```

```
    layer3 = keras.layers.Flatten()(layer2)

    if debug:
        print("Capa 3 completada: " + str(layer3))

    layer4 = keras.layers.Dense(
        512,activation='relu',name="CAPA4")(layer3)

    if debug:
        print("Capa 4 completada: " + str(layer4))

    layer5 = keras.layers.Dense(
        len(A[:,0]),activation='relu',name="CAPA5")(layer4)

    if debug:
        print("Capa 5 completada: " + str(layer5))


    model = tf.keras.Model(inputs=model_input,outputs=layer5)
    model.compile(loss="mean_squared_error",
         optimizer=keras.optimizers.Adam(lr=self.learning_rate))

    if debug:
        print("Modelo completado y compilado")

    return model

# La red elige una accion en base al estado actual
# Modelo de decision epsilon-greedy
def act(self,state):

    self.epsilon *= self.epsilon_decay
    self.epsilon = max(self.epsilon_min,self.epsilon)
    if np.random.random() < self.epsilon:
        return
            self.env.action_space[:,np.random.randint(N_ACTIONS)]

    pred = self.model.predict(state)[0]
```

```
        if np.isnan(pred[0]):
            print("NAN")
        return self.model.predict(state)[0]

    # Proceso de memorizacion de eventos pasados
    def remember(self,state,action,reward,new_state):
        self.memory.append([state,action,reward,new_state])

    # Recuerdo de experiencias anteriores
    def replay(self):

        if len(self.memory) < BATCH_SIZE:
            return

        samples = random.sample(self.memory,BATCH_SIZE)

        for sample in samples:
            state,action,reward,new_state=sample
            target = self.target_model.predict(state)

            Q_future = max(self.target_model.predict(new_state)[0])
            target[0] = np.real(reward) + Q_future * self.gamma

            self.model.fit(state,target,epochs=1,verbose=0)

    # Actualizacion de pesos
    def target_train(self):
        weights = self.model.get_weights()
        target_weights = self.target_model.get_weights()
        for i in range(len(target_weights)):
            target_weights[i] = weights[i] * self.tau +
                target_weights[i] * (1 - self.tau)
        self.target_model.set_weights(target_weights)
```

A partir de este punto, solo queda realizar inicializar el entrenamiento de la red y obtener los resultados. Se harán dos bucles anidados que controlen el episodio y el paso dentro de cada episodio, y en cada episodio se guardará la capacidad máxima alcanzada por la red y se reiniciará el entorno simulado.

```
env = environment()

    dqn_agent = DQN(env=env)
    steps = []

    for trial in range(0,N_EPISODES):
        current_state = env.reset()

        for i,step in enumerate(range(0,N_STEPS_PER_EPISODE)):

            action = dqn_agent.act(current_state)
            new_state, reward, _ = env.step(action)
            dqn_agent.remember(current_state,action,reward,new_state)
            dqn_agent.replay()
            dqn_agent.target_train()

            current_state = new_state
```

Capítulo 6

Resultados

El conjunto de receptores simulado es una agrupación de 5×5 antenas cuyo centro geométrico se sitúa en el origen de coordenadas.

Con tal de comprobar las prestaciones de la DQN, se han considerado varias posiciones iniciales para el emisor: una posición a corta distancia en el punto $r_0 = [20, 2, 3]$ (20.32 metros respecto al centro de la agrupación), otra posición a media distancia en $r_0 = [135, 10, -20]$ o 135.84 metros y una posición a larga distancia en $r_0 = [1100, 70, 130]$ o 1109.86 metros.

Adicionalmente, para cada distancia se han probado dos velocidades distintas para el emisor. En el caso de la distancia corta se tendrán una velocidad "lenta" de 0.1 metros por segundo y otra "rápida" de 5 metros por segundo. Para la distancia media se considerarán 1 m/s y 25 m/s, respectivamente. Por último para la distancia larga las velocidades serán 5 y 75 m/s.

En la figura 6.1 se muestra gráficamente la disposición del emisor y la agrupación en un espacio tridimensional, junto a la recta que une el emisor con el centro geométrico de la agrupación. A la vista de los resultados tenemos que para una matriz de coeficientes estimada y asumiendo un ancho de banda de 5 GHz, en el caso de la velocidad de 0.1 m/s la capacidad del canal se estabiliza sobre unos 0.12 THz tras 20 episodios de entrenamiento, como se puede ver en la figura 6.2.

Cuando la velocidad aumenta a 5 m/s se obtienen resultados similares tras 20 episodios de entrenamiento (figura 6.3), aunque aumenta de forma considerable la varianza entre los resultados. La razón de esto probablemente se deba a que una velocidad de 5 m/s es demasiado elevada para una distancia de 20 metros, lo que dificulta la capacidad de la DQN para converger y predecir adecuadamente el vector de *beamforming* a utilizar. Sin embargo dado que la distancia entre el emisor y el receptor es tan reducida se siguen obteniendo resultados de capacidad aceptables

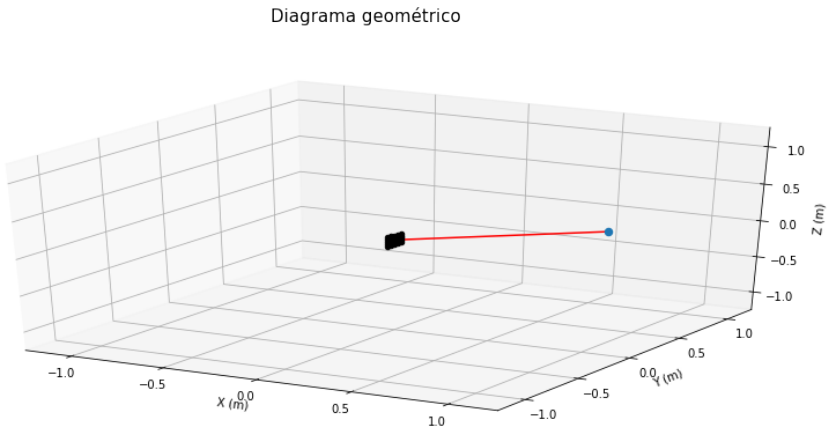

Figura 6.1: Posición inicial a corta distancia

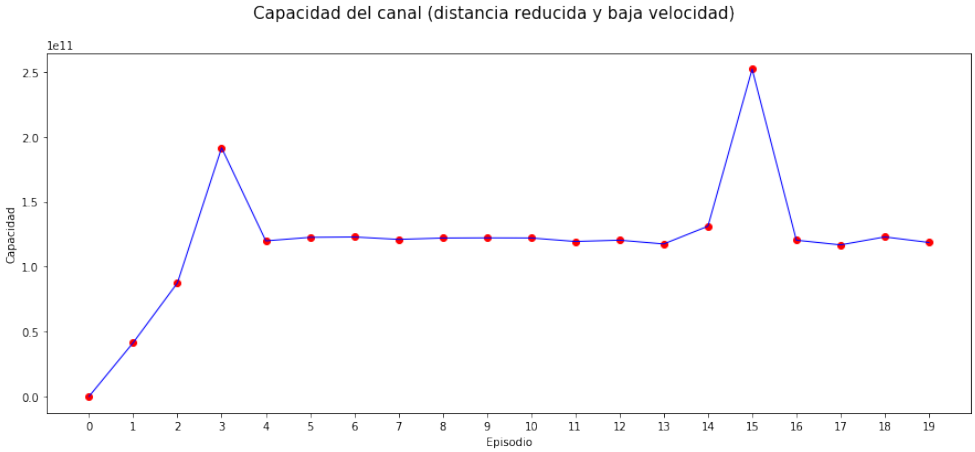

Figura 6.2: Capacidad para una velocidad de 0.1 m/s

incluso cuando se toman vectores no óptimos.

Cuando llegamos al caso de la distancia media (figura 6.4) vemos que para ambas velocidades la capacidad se reduce a medida que pasa el tiempo salvo en episodios aislados donde la capacidad aumenta considerablemente antes de volver a bajar. La razón más probable que da lugar a esta circunstancia es la baja cantidad de vectores posibles y el hecho de que el emisor se mueve aleatoriamente: es muy posible que el emisor acabe vagando a zonas donde ningún vector disponible de un buen resultado.

Esto explicaría porqué la tendencia de la curva es reducir su valor con espurios aleatorios: a una distancia de 136 metros y teniendo una separación de 5 grados entre vectores el arco que separa dos vectores consecutivos es de aproximadamente 11.87 metros y dada la alta directividad del *beamformer* si el emisor no está muy próximo a la dirección a la que apunta un vector dado las pérdidas en capacidad son

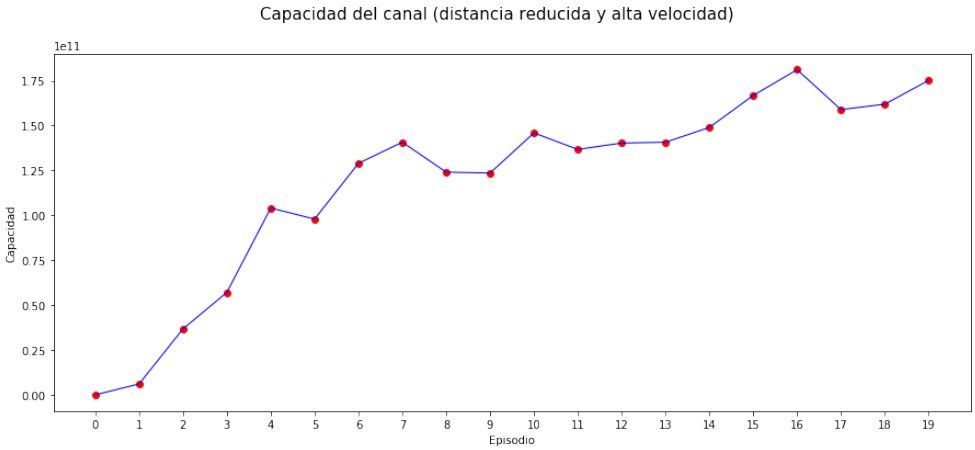

Figura 6.3: Capacidad para una velocidad de 5 m/s

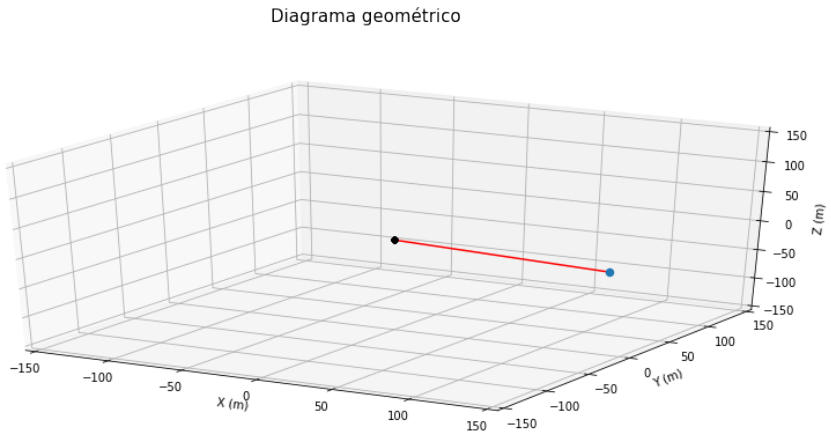

Figura 6.4: Posición inicial a media distancia

prácticamente totales, lo que se corresponde con los resultados obtenidos.

En el caso del emisor de alta velocidad, es posible que en algunos episodios el emisor haya terminado aproximándose bastante al receptor o alineándose por azar con algún vector escogido. Aún así la tendencia global es similar a la de la baja velocidad, pero más ruidosa.

Habiendo interpretado los resultados del emisor en distancias medias, es fácil ver como cuando aumentamos aún más la distancia entre el emisor y el receptor el problema de deriva del emisor se agrava más todavía. Teniendo un arco entre vectores de 96.87 metros

A la vista de estos resultados, se proponen dos soluciones para mejorar las prestaciones de la DQN en distancias medias y largas:

- La más directa y simple es aumentar el número de vectores. Naturalmente,

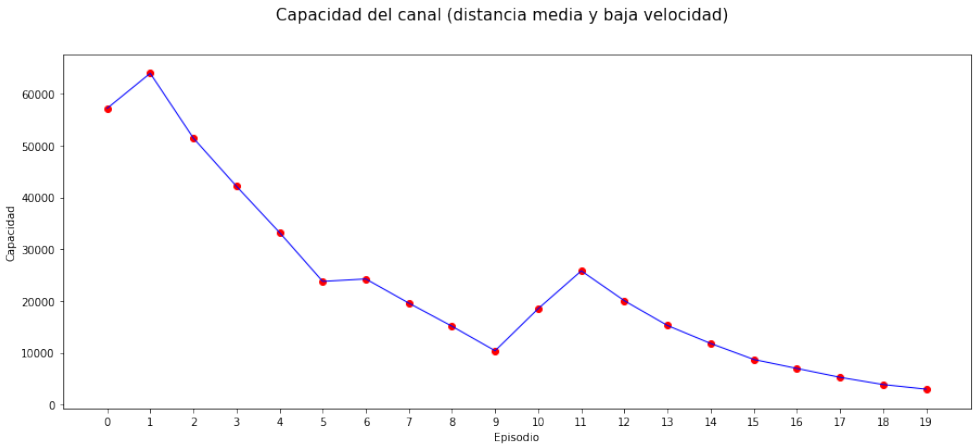

Figura 6.5: Capacidad para una velocidad de 1 m/s

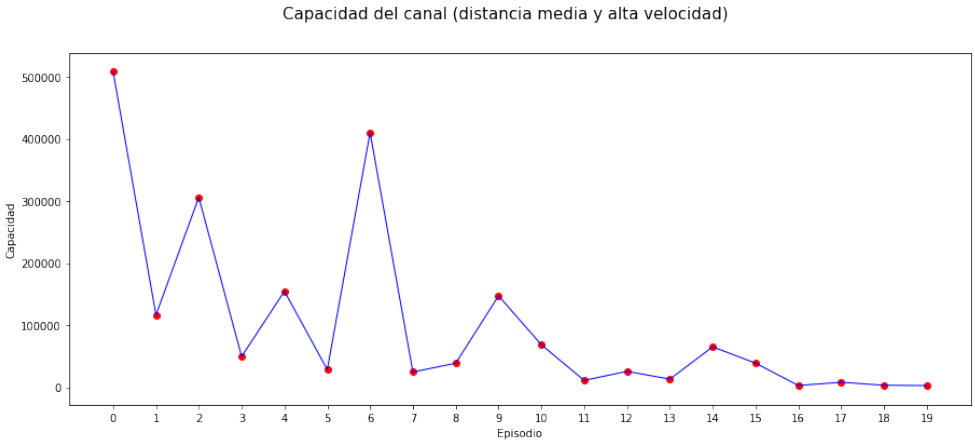

Figura 6.6: Capacidad para una velocidad de 25 m/s

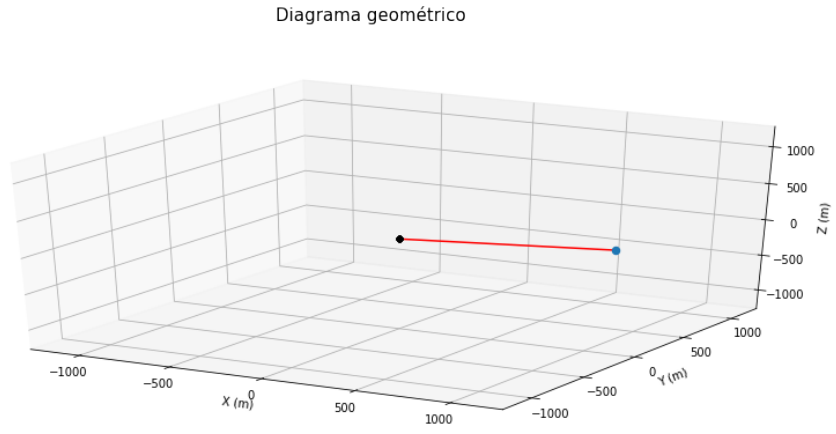

Figura 6.7: Posición inicial a larga distancia

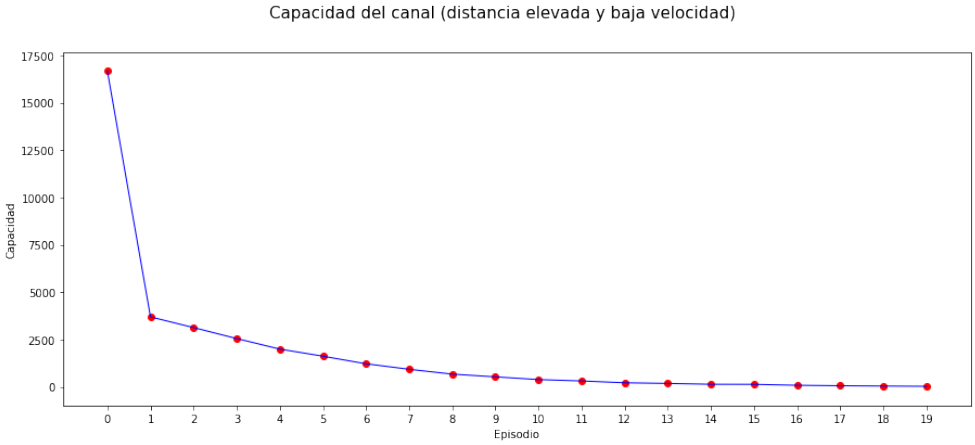

Figura 6.8: Capacidad para una velocidad de 5 m/s

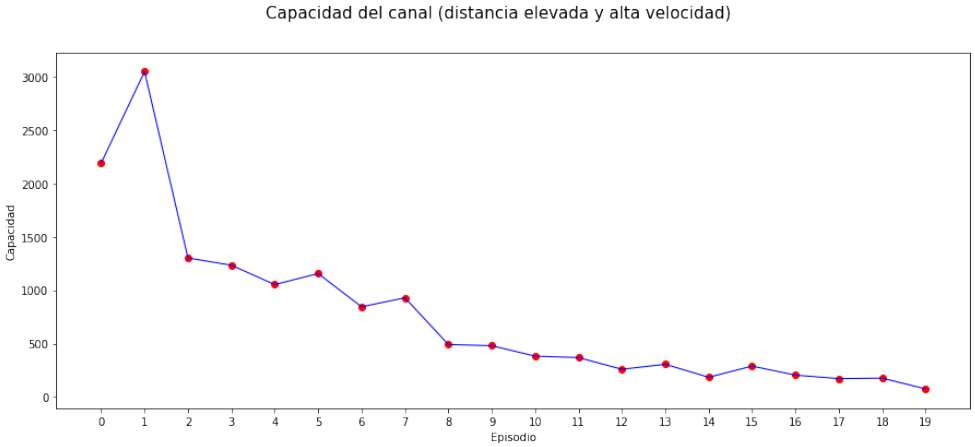

Figura 6.9: Capacidad para una velocidad de 75 m/s

esto reducirá el arco entre vectores y las posibles pérdidas por tener al emisor vagando por áreas entre vectores estarán más limitadas. El principal inconveniente de esta solución es que a mayor número de vectores, mayor será la complejidad de la última capa de la red, así como el coste computacional necesario para entrenarla.

- La otra solución es limitar el rango de ángulos en el que generamos los vectores de *beamforming*. Esto podría ser viable cuando sabemos de antemano que la agrupación va a apuntar a emisores cuyas trayectorias no se van a salir de dicho rango. Esta solución es además perfectamente compatible con la solución anterior y una combinación entre ambas mejorarían significativamente las prestaciones de la red sin aumentar excesivamente el coste computacional. La desventaja más clara es que en el momento en el que el emisor se salga las pérdidas en el canal serán muy elevadas.

Capítulo 7

Conclusiones

7.1. Consideraciones finales

En base a los resultados obtenidos, se puede ver que la aplicación del aprendizaje profundo reforzado permite encontrar soluciones a problemas de toma de decisiones en entornos de una cierta complejidad. Aún así, también queda claro que la efectividad de este tipo de aprendizaje máquina viene limitado por la calidad de las decisiones posibles dentro de un entorno concreto.

Esto se pone de manifiesto en las drásticas diferencias que existen entre los resultados cuando consideramos un emisor cercano y cuando consideramos un emisor con una distancia más o menos elevada. En el caso cercano el arco entre vectores es mucho menor al de los casos más lejanos por lo que es más fácil que el emisor se mueva cerca de estos vectores así que cuando la red escoja el vector óptimo es más probable que se consiga un buen resultado. Por el contrario en distancias largas al haber más espacio entre los vectores disponibles, incluso si la red aprende a escoger el vector más óptimo para una posición dada va a haber tal desajuste entre el ángulo del vector y el ángulo real del emisor que los resultados serán muy inferiores a lo que inicialmente se esperaría.

Si bien es importante hablar de las capacidades y ventajas que otorga el uso de las redes neuronales basadas en el aprendizaje profundo mostrando resultados "positivos", es igualmente importante mencionar aquellos casos donde se identifican sus limitaciones dando resultados "negativos".

7.2. Trabajos futuros

A modo de trabajos futuros, la primera cuestión a tratar sería mejorar la estimación de la matriz de coeficientes de canal H. Si bien el método actual funciona a efectos de demostración, está demasiado idealizado al no tener en cuenta los efectos del ruido ni posibles fenómenos tales como el *crosstalk*. Además los valores de esta matriz son una parte clave del cálculo de la recompensa de la DQN durante su entrenamiento por lo que es preciso tener una medida lo más cercana posible a la realidad para corroborar que los resultados realmente son fiables.

También sería recomendable tratar de simular un entorno más real, incluyendo obstáculos y superficies que puedan simular los efectos de la reflexión y la refracción, ampliando debidamente la simulación de la propagación de la señal. En este aspecto es esperable que las capas de convolución de la DQN sean capaces de extraer las características necesarias para tener en cuenta estos nuevos efectos, aunque también es muy posible que haga falta una mayor duración en los episodios de entrenamiento.

Otra cuestión a tratar es el número de vectores de *beamforming* disponibles. En este trabajo solo se han tomado 36 direcciones posibles el acimut, y otras 36 en elevación, dando un total de 1296 vectores disponibles ordenados en pasos de aproximadamente 5 grados en su ángulo de elevación y en acimut. Además en las direcciones de acimut solo se han tomado las direcciones entre 0 y 180 grados, es decir, solo estamos considerando vectores que estén "delante" de la agrupación. Si bien esto tendría sentido en aplicaciones donde esperamos emisores que se encuentren frente a la agrupación no quita que en otras situaciones nos interese apuntar a cualquier dirección.

Por otra parte, asumiendo que que la agrupación pueda suministrar la correspondiente distribución de amplitudes y fases, se podría aumentar el número de vectores disponibles reduciendo el paso entre ángulos. Esto permitiría a la red ajustar vectores óptimos de *beamforming* que apunten al emisor con mayor precisión.

Desgraciadamente aumentar el número de vectores disponibles también aumenta la complejidad de la red en cuanto al número de neuronas en su última capa, y por tanto su número de pesos a ajustar dentro de dicha capa. En la figura 7.1 se muestran el número de vectores requeridos para un determinado paso de ángulos (asumiendo el mismo paso para acimut y elevación).

Como se puede ver, el número de vectores necesarios para pasos pequeños se vuelve excesivamente elevado, y esto podría llegar a suponer un incremento significativo en el tiempo de entrenamiento de la DQN.

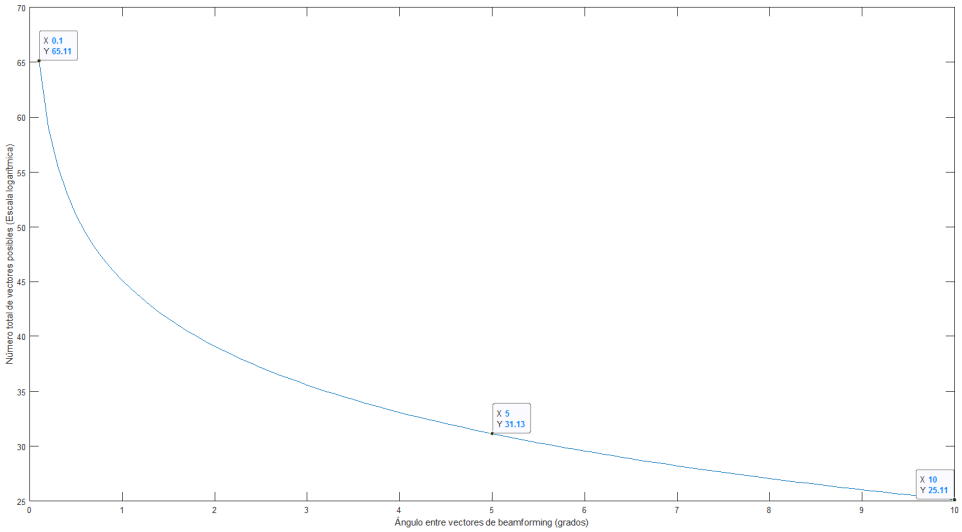

Figura 7.1: Número de vectores necesarios por ángulo de paso

Capítulo 8

Anexo

8.1. Configuración de la simulación

```
 CONSTANTES

 Utilidades
    ================================================================================
np.random.seed(np.random.randint(999999999))
PI = np.pi

 Eje temporal
    ================================================================================
TIME_RESOLUTION = 25000
TIME_LENGTH = 0.0000002
TIME_AXIS = np.linspace(0,TIME_LENGTH,TIME_RESOLUTION)

 Eje frecuencial
    ================================================================================
F_AXIS = np.fft.fftfreq(TIME_AXIS.shape[-1]) * TIME_RESOLUTION /
   TIME_LENGTH

 Parametros de senal y sistema
    ================================================================================
PROPAGATION_SPEED = 300000000 # Velocidad de propagacion.

FREQUENCY = 5000000000 # Frecuencia de la senal
WAVELENGTH = PROPAGATION_SPEED/FREQUENCY # Longitud de onda
```

```
TRANSMISSION_POWER = 1 # Potencia de transmision

N_RECEIVERS_Y = 5 # Numero de receptores en el eje Y
N_RECEIVERS_Z = 5 # Numero de receptores en el eje Z

RECEIVER_POS_Y =
    0.5*WAVELENGTH*(np.arange(1,N_RECEIVERS_Y+1)-0.5*(1+N_RECEIVERS_Y))
    # Posiciones en Y
RECEIVER_POS_Z =
    0.5*WAVELENGTH*(np.arange(1,N_RECEIVERS_Z+1)-0.5*(1+N_RECEIVERS_Z))
    # Posiciones en Y

SPACING = RECEIVER_POS_Y[-1] - RECEIVER_POS_Y[-2]

 Entorno
    ========================================================================

ENV_TIMESTEP = 1.0e-6 # Paso de tiempo en la simulacion del entorno
ENV_EMITTER_INITIAL_COORDS = (1100,70,130) # Coordenadas iniciales
    del emisor
ENV_EMITTER_VELOCITY_FACTOR = 75 # Acota la maxima velocidad
    instantanea posible

 Ruido
    ================================================================================
NOISE_AMPLITUDE = 0.00001
NOISE_VARIANCE = 1

 Vectores de beamforming
    ============================================================================
N_VECTORS_THETA = 36
N_VECTORS_PHI = 36
N_ACTIONS = N_VECTORS_PHI * N_VECTORS_THETA

THETA_RANGE = np.linspace(-90,90,N_VECTORS_THETA)
PHI_RANGE = np.linspace(0,180,N_VECTORS_PHI)
```

```
 Parametros de la DQN
    ================================================
N_EPISODES = 20
N_STEPS_PER_EPISODE = 50

EPSILON_START = 1.0
EPSILON_END = 0.001
EPSILON_DECAY = 0.999

LEARNING_RATE = 0.005
TAU = 0.125

BATCH_SIZE = 24

DISCOUNT_FACTOR = 0.975

MAX_ALLOWED_STEPS = 199

REWARD_SCALING = 1000
```

Nota: se han eliminado los caracteres que dependen de la codificación UTF-8, tales como las tildes y la letra ñ.

Bibliografía

[1] Optimización de DRAM https://www.researchgate.net/publication/4349980_Self-Optimizing_Memory_Controllers_A_Reinforcement_Learning_Approach

[2] Barto A.G., Thomas P. S., Sutton R.S. (2017) Some Recent Applications of Reinforcement Learning https://people.cs.umass.edu/~pthomas/papers/Barto2017.pdf

[3] Haykin S. (2008) Neural Networks and Learning Machines

[4] Hastie, T., Tibshirani, R., and Friedman, J. (2001). The elements of statistical learning: data mining, inference and prediction. Springer Series in Statistics. Springer Verlag.

[5] Goodfellow et al. (2016). Deep Learning

[6] Srivastava N., Hinton G., Krizhevsky A., Sutskever I., Salakhutdinov R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[7] Kasche J., Nordström F. (2020). Regularization Methods in Neural Networks https://www.diva-portal.org/smash/get/diva2:1389238/FULLTEXT01.pdf

[8] Kuno Y., Masiero B., Madhu N. (2019). A neural network approach to broadband beamforming https://pub.dega-akustik.de/ICA2019/data/articles/000879.pdf

[9] https://www.cis.rit.edu/class/simg303/Notes/Ch7-PropagationofWaves.pdf

[10] Minhow K., Woongsup L. Dong-Ho C. (2020) Deep Scanning—Beam Selection Based on Deep Reinforcement Learning in Massive MIMO Wireless Communication System https://www.mdpi.com/2079-9292/9/11/1844/pdf

[11] Mnih V., Kavukcuoglu K., Silver D. et al. (2015) Human-level control through deep reinforcement learning https://deepmind.com/research/publications/human-level-control-through-deep-reinforcement-learning

[12] Documentación de Tensorflow https://www.tensorflow.org

[13] Brunskill E. (2018) CNNs and Deep Q Learning https://web.stanford.edu/class/cs234/CS234Win2018/slides/cs234_2018_l6.pdf

[14] Sound Fields: Free versus Diffuse Field, Near versus Far Field https://community.sw.siemens.com/s/article/sound-fields-free-versus-diffuse-field-near-versus-far-field

[15] Patel Y. (2017) Reinforcement Learning w/ Keras + OpenAI: DQNs https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c

[16] Apuntes de la asignatura “Antenas y Propagación” del Grado GIST-UPCT, 2020