Trabajo Fin de Grado

Ingeniería de las Tecnologías de Telecomunicación

Detección MIMO con Deep Learning

Autor: Óscar González Fresno

Tutor: Juan José Murillo Fuentes

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Ingeniería de las Tecnologías de Telecomunicación

Detección MIMO con Deep Learning

Autor:
Óscar González Fresno

Tutor:
Juan José Murillo Fuentes
Catedrático de Universidad

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2022

Trabajo Fin de Grado: Detección MIMO con Deep Learning

Autor: Óscar González Fresno

Tutor: Juan José Murillo Fuentes

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2022

El Secretario del Tribunal

Agradecimientos

La finalización de este trabajo supone el cierre de una etapa muy bonita de mi vida y de la cual estoy orgulloso. Me gustaría agradecer primeramente a mi tutor, Juan José Murillo Fuentes, por su ayuda, consejos y conocimientos aportados para la realización de este proyecto, y al resto de docentes que me han acompañado durante estos años.

También a los compañeros más cercanos que he conocido durante el grado y de los cuales algunos se han convertido en grandes amigos.

A mis amigos de siempre, por su apoyo y su forma de ser.

Y, por último, a mi familia, por brindarme la oportunidad de formarme académicamente y por sus muestras de afecto y orgullo.

Óscar González Fresno

Gijón, 2022

Resumen

En los últimos años, la sociedad ha ido avanzando hacia un mundo cada vez más conectado. Esto se debe gracias al avance de la tecnología y a nuevos y mejores métodos de comunicación, sin los cuales sería imposible compartir la cantidad de información diaria que se transmite cada día.

Este proyecto tiene como objetivo el estudio mediante simulación en Python de una comunicación digital de varias antenas en cada extremo, cuyo proceso de detección se basa en el método de Máxima Verosimilitud. Este método ha sido a su vez comparado con otros más sencillos para estudiar su comportamiento frente a diferentes canales de comunicación. Se han comparado los resultados obtenidos tanto con Numpy como con PyTorch.

Por último, como la detección mediante el método de Máxima Verosimilitud puede resultar extremadamente compleja a nivel computacional, se han aplicado técnicas de aprendizaje profundo para reducir la carga computacional sin afectar al resultado, utilizando PyTorch en Google Colaboratory.

Abstract

In recent years, society has been moving towards an increasingly connected world. This is due to the advancement of technology and new and better communication methods, making possible to share the amount of daily information that is transmitted every day.

This project aims to study by means of simulation using Python a digital communication with several antennas at each end, whose detection process is based on the Maximum Likelihood method. This method has also been compared with other simpler ones to study its behavior against different communication channels. The obtained results have been compared with both Numpy and PyTorch.

Finally, as Maximum Likelihood detection method can be computationally complex, Deep Learning techniques have been applied to reduce the computational load minimizing the modification of the result, using PyTorch in Google Colaboratory.

Índice

Índice de Tablas

Índice de Figuras

Acrónimos

AWGN	Additive White Gaussian Noise
BER	Bit error rate
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
LMMSE	Linear Minimum Mean Squared Error
MIMO	Multiple-Input Multiple-Output
ML	Maximum Likelihood
PAM	Pulse Amplitude Modulation
SER	Simbol error rate
SISO	Single-Input Single-Output
SNR	Signal-to-noise ratio
ZF	Zero Forcing

Notación

$argmin$	Argumento del mínimo
$\mathbf{y}$	Vector
$\mathbf{H}$	Matriz
$\mathbf{H^{-1}}$	Matriz inversa
$\mathbf{H^{T}}$	Matriz traspuesta
$\in$	Perteneciente a
$\|\cdot\|^2$	Norma de orden 2
$Cond(\cdot)$	Número de condición de una matriz
λ	Escalar
σ^2	Varianza

1 INTRODUCCIÓN

1.1 Motivación

La elección del tema propuesto me ha resultado de gran interés puesto que siempre me he sentido atraido por el mundo de la inteligencia artificial, pero hasta ahora nunca se me había dado la oportunidad de relaccionarla con algo de lo visto en la carrera. Por eso, he visto una gran oportunidad de cumplir mi objetivo con este proyecto. Además, el empleo de técnicas MIMO en las comunicaciones también me resultó muy interesante cuando me lo menciaron en alguna materia, pero de nuevo, nunca me he podido acercar de forma práctica a ello. La combinación de estos dos motivos han hecho que la elección del tema para este trabajo fuese una tarea fácil.

1.2 Datos y herramientas empleados

El material de las prácticas de laboratorio de la asignatura *Comunicaciones Digitales* de tercer curso se ha empleado como base sobre la que desarrollar el trabajo [1]. La programación se ha hecho sobre Python, utilizando una distribución denominada Anaconda, que permite la instalación de los diferentes elementos que constituyen un entorno de desarrollo de aplicaciones en Python [2]. De las diferentes aplicaciones intaladas por defecto, se ha elegido Spyder como IDE [3], ya que es comúnmente usada para programación científica en Python y guarda muchas similitudes con MATLAB.

Posterioremente, y con el objetivo de emplear técnicas de Deep Learning, se ha utilizado PyTorch, una plataforma de aprendizaje automático para Python [4].

En todo momento ha sido necesario el empleo de librerías que contubiesen los métodos adecuados para la correcta programación de la simulación. Es por ello que se han empleado las librerías *numpy* y *matplotlib* [5] [6]. Todo el código mostrado a lo largo del trabajo se puede encontrar en un repositorio de GitHub [7].

1.3 Metodología

Para una mayor facilidad de desarrollo y seguimiento del trabajo, se ha optado por dividirlo en tres partes:

1. Simulación de sistema MIMO en Python
2. Simulación de sistema MIMO en Python con PyTorch
3. Diseño y simulación de detección MIMO basada en aprendizaje profundo

En la primera parte se ha definido el modelo de comunicación empleado a partir del material provisto [1]. Se ha creado un sistema de comunicación digital que utiliza una modulación 4-PAM y varias antenas transmisoras y receptoras, de mismo número en cada extremo. En la fase de detección se comparan tres métodos distintos, el Zero Forcing (ZL), el Linear Minimum Mean Squared Error (LMMSE) y el Maximum Likelihood (ML).

La segunda parte consiste únicamente en trasladar el modelo usado en el IDE Spyder a la plataforma PyTorch, haciendo los cambios oportunos para que el funcionamiento del programa sea el mismo.

Por último, en la tercera parte se pretende diseñar el modelo adecuado de aprendizaje profundo para utilizar el método de detección ML y ver las ventajas e inconvenientes que supone el uso de Deep Learning.

2 SIMULACIÓN DE SISTEMA MIMO EN PYTHON

El primer paso de esta parte ha consistido en modificar el Código 7.6 presente en el material de prácticas [1] para definir primeramente un sistema SISO con modulación 4-PAM. Sobre esta modificación se ha pasado luego a un modelo con varias antenas de transmisión y recepción, es decir, un sistema MIMO.

2.1 Creación del Sistema

2.1.1 Fundamento teórico

La multiplexación espacial es ampliamente utilizada debido a que aumenta la eficiencia espectral de un sistema de comunicación inalámbrica. En la **Figura 2.1** se representa el diagrama de un sistema MIMO genérico.

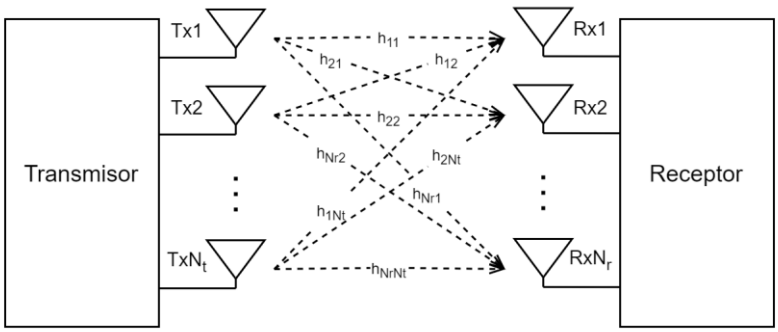


Figura 2.1: Esquema de un sistema MIMO

El uso de varias antenas tanto en transmisión como en recepción permite lo siguiente:

- Mandar más información cuando es un único usuario el que transmite por todas las antenas. Si estas están suficientemente espaciadas, es decir, si los canales son estadísticamente independientes, se puede mandar N veces lo que se mandaría en un sistema SISO, siendo *N* el número de antenas transmisoras y receptoras ($N = N_r = N_t$).
- Compartir el acceso al medio de varios usuarios. Se puede usar de forma más efectiva el canal para enviar más información a través de él.
- Mandar la misma información por varios canales para obtener diversidad y aumentar la robustez frente a posibles desvanecimientos en alguno de los canales.

A cada antena de recepción le llega la información transmitida por cada una de las antenas multiplicada además por un coeficiente que guarda relación con el canal de transmisión. A esto se le añade el ruido presente en el medio. En este caso se ha optado por utilizar un modelo conocido como *canal de ruido auditivo*, en el que el efecto del canal se representa por una señal aleatoria que se añade a la señal transmitida y es estadísticamente independiente de la misma [8]. Por sus siglas en inglés también se le conoce como Additive White Gaussian Noise (AWGN).

De esta manera, un sistema MIMO puede expresarse mediante la siguiente ecuación,

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n} \tag{2.1}$$

La información de las diferentes antenas transmisoras está contenida en el vector **x**, de longitud N_t , mientras que **y** representa el vector de información de la etapa receptora, de longitud N_r. La caracterización del canal viene expresada en la matriz **H**, también conocida como *matriz del canal.* Su tamaño es de $N_r \times N_t$. El ruido viene definido en el vector **n**.

A continuación, se mostrará la programación del sistema dividida en varios bloques de código con el fin de facilitar la explicación y la lectura, pero se podrá encontrar el programa completo en el archivo *MIMO_numpy.py* [7].

2.1.2 Inicialización

Para llevar a cabo la programación, el primer paso fue definir las características principales del sistema. Se asumirá de ahora en adelante un igual número de antenas transmisoras y receptoras. En transmisión, la información por cada antena será distinta e independiente del resto y, en la etapa de recepción, se tendrán disponibles todas las señales recibidas por las antenas. Para una mayor simplicidad, se optará por realizar una simulación directamente de los símbolos junto con el ruido, sin la necesidad de generar pulsos que muestreen esos símbolos.

Por simplicidad y velocidad de procesamiento, se ha simulado un sistema 2x2 con modulación 4-PAM.

```
import numpy as np
import matplotlib.pyplot as plt
from labcomdig import transmisorpamV2, Qfunct, detectaSBF, simbolobit
from functions import de2Nary

#%% INITIALIZATION

Nb = 1e6                    #Number of bits on transmission
Eb = 9                      #Average energy of the bit
M = 4                       #4-PAM modulation
n = 2                       #Antenna array NxN
SNRmin = 0                  #Minimum SNR [dB]
SNRmax = 24                 #Maximun SNR [dB]
SNRstep = 1                 #Step for SNR array[dB]
k = np.log2(M)              #Number of bits on each symbol
Ns = int(np.floor(Nb/k))    #Number of symbols (multiple of k)
Nb = int(Ns*k)              #Number of bits on transmission (multiple of
k)
Es = k*Eb                   #Symbol energy
verbose = 1                 #Progress indicator

#Data generation as one sequence
#We are supposing a multi-user transmission

bn = np.random.randint(2,size=Nb*n)
```

Código 2.1: Definición de los parámetros de entrada.

En el **Código 2.1** se muestran los parámetros que definen la comunicación, como son el número de bits que transmite cada antena, la energía media del bit, el número de símbolos que tiene la modulación, el número de antenas y el valor máximo de SNR que estará presente en la comunicación. También se hace un ajuste del número de bits y de símbolos para que estos sean múltiplos del número de bits presentes en cada símbolo.

La información transmitida se genera con una llamada a una función que genera un vector de números aleatorios. Para el caso de estudio, los números irán comprendidos entre el 0 y el 2 (no incluido), lo cual representa un bit. La longitud viene dada por el número de bits multiplicado por el número de antenas. De esta manera, a nivel de simulación se obtiene un único vector que contiene toda la información a transmitir.

El principio del programa es necesario importar las librerías necesarias que hagan funcionar correctamente el programa. A parte de las librerías de acceso libre *numpy* y *matplotlib*, se deben importar funciones de los archivos *labcomdig* [1] y *functions* [7]. Ambos archivos recogen diferentes funciones que serán usadas a lo largo del programa.

2.1.3 Transmisor

El diseño de un transmisor PAM se realiza mediante la construcción en Python de una función denominada *transmisorpamV2*. Se trata de un ligero cambio en la función *transmisorpam* [1], en el que se modifica la forma de la variable que conforma el pulso y se comprueba que el producto escalar consigo mismo no sea nulo. El **Código 2.2** muestra la creación de un transmisor PAM.

```
#%% M-PAM TRANSMITTER

[Xn,Bn,An,phi,alphabet] = transmisorpamV2(bn,Eb,M,np.ones(1),1)

#Data reshape. Each row represents an antenna
Xn = Xn.reshape([n,Ns])
An = An.reshape([n,Ns])
Bn = Bn.reshape([n,Nb])
```

Código 2.2: Diseño del transmisor

Se introducen como entrada la secuencia de bits, el valor de energía promedio de bit, el número de símbolos, el pulso de las señales PAM y el número de muestras por símbolo. Como para este trabajo es indiferente el tipo de pulso, se ha decidido eliminarlo y simular a nivel de símbolo. Es decir, se tiene una única muestra por símbolo.

A la salida de la función se obtiene el tren de pulsos discreto generado, la secuencia de bits múltiplo de $\log_2 M$, la secuencia de puntos de la constelación, el pulso básico discreto de energía unidad utilizado en la transmisión y un vector en el que se recogen los niveles de amplitud asociados a cada uno de los M diferentes símbolos transmitidos.

Dado que a la entrada se ha introducido un único vector de información correspondiente a los bits que transmiten cada usuario, a la salida se obtienen también únicamente vectores. Por ello, para su uso posterior, se han modificado las dimensiones de los vectores *Xn, Bn* y *An* para obtener matrices en las que a cada fila le corresponda una antena.

2.1.4 Canal

A continuación, se debe realizar la caracterización del canal. Para ello, basta con definir un *matriz del canal* que ponderará las señales transmitidas por las antenas de la primera etapa. En el **Código 2.3** se muestra el proceso previamente definido.

```
#%% CHANNEL

#Various matrix should be used for comparing the differences between
algorithms

H = np.eye(n)           #Channel matrix
#H = np.array([[1, 0.45],[0.25, 1]])
#H = np.array([[1, 0.20],[1, 0.25]])

Hinv = np.linalg.inv(H) #Inverse channel matrix

SNRdb = np.arange(SNRmin,SNRmax,SNRstep) #SNR values [dB]
```

Código 2.3: Creación del canal

Es interesante el modelado de varias **H** para estudiar cómo afectan los distintos tipos de matrices al proceso de recepción. Según el método de detección que se use, es necesario el uso de una *matriz del canal inversa*, $\mathbf{H}^{-1}$.

A parte de las matrices del canal, se ha definido también un vector con distintos valores de SNR, tomando como valor máximo la variable *SNRmax* y con salto cada dos unidades. Este vector se usará posteriormente para crear un bucle *for* en el que se calculen la tasa de error de símbolo (SER) y la tasa de error de bit (BER) para cada valor de SNR.

2.1.5 Transmisión

Una vez definido el canal, se procede a transmitir los datos. Se crean las variables en las que se guardará el VER para cada método, se crea una matriz que recoge las posibles combinaciones transmitidas (se usará en la detección con ML) y se entra en un bucle que itera con cada valor de SNR del array. Dentro del bucle, a la señal creada en la etapa de transmisión se le añade el ruido correspondiente.

```
#%% TRANSMISION AND DETECTION

berZF= np.empty([n,0])
berLMMSE= np.empty([n,0])
berML= np.empty([n,0])

#Possible alphabet combinations at reception
x = alphabet[de2Nary(np.arange(M**n),n,len(alphabet))]

if verbose: print('Simulating SNR = ', end = '')
for ii in range(len(SNRdb)):
    if verbose: print('{}, '.format(SNRdb[ii]), end = '')

    SNR = 10**(SNRdb[ii]/10)          #Signal to Noise Ratio [n.u.]
    varzn = Eb/(2*SNR)                          #Noise variance
    Wn = np.sqrt(varzn)*np.random.randn(*Xn.shape)  #AWGN

    Rn = H@Xn + Wn  #Channel output]
```

Código 2.4: Transmisión

2.1.6 Detección

La recuperación de los datos transmitidos se hace en el proceso de detección. Para este trabajo se ha planteado el estudio de dos métodos de detección:

2.1.6.1 Cancelador (Zero Forcing, ZF)

Con ZF se pretende resolver la siguiente ecuación [9]:

$$\tilde{\mathbf{x}}(\mathbf{y}) = \text{argmin}_{\mathbf{x}} \|\mathbf{y} - \mathbf{H}\mathbf{x}\|^2 \tag{2.2}$$

Para una matriz cuadrada invertible, el problema se reduce a

$$\tilde{\mathbf{x}}(\mathbf{y}) = \mathbf{H}^{-1}\mathbf{y} = \mathbf{x} + \mathbf{H}^{-1}\mathbf{n} \tag{2.3}$$

Se trata de un proceso sencillo pero cuya validez depende mucho de como sea la *matriz del canal*, pues según cómo sea su inversa, el ruido podría verse muy amplificado y generar una gran cantidad de errores en la recepción.

```
    # ZERO FORCING (ZF)
    Z = Hinv@Rn

    Andetected = detectaSBF(Z.flatten(),alphabet).reshape([n,Ns])
#Detected symbols
    Bndetected =
simbolobit(Andetected.flatten(),alphabet).reshape([n,Nb])#Detected bits
    errorsBit = Nb-np.sum(Bndetected==Bn,axis=1) # N° Errors per bit
    berZF = np.hstack([berZF,(errorsBit/Nb).reshape([n,1])])
```

Código 2.5: Detección con método ZF

El **Código 2.5** muestra el proceso de detección con el algoritmo ZF para distintos valores de SNR. En un primer lugar, se crean dos vectores vacíos en los que se guardará el resultado de las tasas de error calculadas. A

continuación, se inicia un bucle que recorrerá los distintos valores de la relación señal a ruido, mostrando por pantalla la interacción en la que se esté en cada momento.

Dentro del bucle, primero se define el ruido del canal. Por simplicidad, y como se explica en el apartado 2.1.1, se asume un canal AWGN. En este modelo se supone que el efecto del canal se representa por una señal aleatoria que se añade a la señal transmitida y cuyas muestras son estadísticamente independientes e idénticamente distribuidas, según una normal de media cero. De esta manera, se tiene que la señal en recepción es `Rn = H@Xn + Wn` ,(2.1).

El proceso de detección consiste en multiplicar la señal recibida por la matriz del canal inversa, `Z = Hinv@Rn` , (2.3). Una vez obtenida esta señal, se procede a evaluar los errores cometidos. Para ello, se llama a la función *detectaSBF,* que devuelve la secuencia de puntos de la constelación detectados. Posteriormente, se hace una comparación con los puntos de constelación transmitidos y se calcula la tasa de error de símbolo.

La función *simbolobit* devuelve los bits correspondientes a cada punto de la constelación. De esta manera, es posible saber los bits detectados a partir de los puntos de la constelación detectados. Si se hace la comparación con los bits realmente transmitidos, es posible obtener la tasa de error de bit.

Durante la escritura de este código, se ha probado en un primer momento a realizar el proceso de detección de manera independiente por cada antena y utilizando alguna función diferente. En el **Código 2.6** se puede ver un ejemplo de ello. Haciendo varias simulaciones, resultó no ser un algoritmo bien diseñado, pues el tiempo de ejecución del código era demasiado lento. Por este motivo, se optó por una optimización de este y se obtuvo el **Código 2.5**.

```
for j in range(n):
        # Salida del Detector
        Andetectado = detectaSBF(Z[j], alfabeto) #Símbolos detectados
        erroresSimbolo = Ns-sum(Andetectado==An[j]) #N° Errores de
símbolo
        serSimulada[j,:,ii] = erroresSimbolo/Ns
        Bndetectado = simbolobit(Andetectado, alfabeto) #Bits detectados
        erroresBit = Nb-sum(Bndetectado==Bn[j]) # N° Errores de bit
        berSimulada[j,:,ii] = erroresBit/Nb
```

Código 2.6: Proceso de detección descartado

El objetivo estuvo en todo momento en eliminar el bucle *for* correspondiente a cada antena. Para ello, se optó nuevamente por unificar las señales de información de cada usuario como si fuese una única señal de un único usuario. Esto se debe a que las funciones *detectaSBF* y *simbolobit* están diseñadas para trabajar con vectores y no con matrices. Una vez unificadas las señales e introducidas como parámetros de entrada a estas funciones, se procede a modificar la forma de su salida para obtener nuevamente matrices cuyas filas corresponden a cada antena.

Por otro lado, se observó también un cambio significativo en la eficiencia del algoritmo al modificar la función *sum,* nativa de Python, a la función *np.sum,* propia de la librería *numpy*. El motivo de la mejora viene dado porque la función *sum* sólo opera con los tipos del lenguaje de Python, y los *narray*, que son los tipos con los que se están trabajando en el programa, no lo son. Entonces, para poder hacer uso de la función, se hace una conversión de un tipo a otro, lo cual lleva mucho tiempo cuando se está tratando con un vector de mucha longitud.

Por último, para almacenar los resultados obtenidos para cada valor de SNR, se optó por crear un vector vacío al que se le concatena el resultado de cada iteración.

2.1.6.2 Estimación Lineal de Error Cuadrático Medio (Linear Minimum Mean Squared Error, LMMSE)

El detector ZF puede causar amplificación del ruido cuando la matriz **H** es casi no invertible, ya que su inversa tendría unos coeficientes muy grandes que aumentarían la aportación del ruido a la señal recibida. Esto se puede estudiar también con el *número de condición* de una matriz, que mide mide cuánto se modifica el valor de salida si se realiza un gran cambio en el valor de entrada. Si el número de condición es cercano a la unidad se dice que la matriz está *bien condicionada*, y si es mayor que uno se dice *mal condicionada* [10].

Sea **A** una matriz de tamaño $m \times n$, se le llama *número de condición* a $Cond(\mathbf{A})$ tal que

$$Cond(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\| \tag{2.4}$$

Para reducir la sensibilidad de los receptores lineales respecto a la condición de la matriz, se puede añadir un término de regularización, como se muestra en la ecuación (2.5) [9],

$$\tilde{\mathbf{x}}(\mathbf{y}) = \text{argmin}_{\mathbf{x}}\|\mathbf{y} - \mathbf{H}\mathbf{x}\|^2 + \lambda\|\mathbf{x}\|^2 \tag{2.5}$$

donde λ corresponde a la varianza del ruido dividida por la energía del símbolo para modulaciones multinivel, σ^2/E_s [11]. La solución viene dada por

$$\tilde{\mathbf{x}}(\mathbf{y}) = \left(\mathbf{H}^{\mathrm{T}}\mathbf{H} + \lambda\mathbf{I}\right)^{-1}\mathbf{H}^{\mathrm{T}}\mathbf{y} \tag{2.6}$$

```
    # LINEAR MINIMUM MEAN SQUARED ERROR (LMMSE)
    Z = np.linalg.inv(H.T@H + varzn/Es*np.eye(n))@H.T@Rn

    Andetected = detectaSBF(Z.flatten(),alphabet).reshape([n,Ns])
#Detected symbols
    Bndetected =
simbolobit(Andetected.flatten(),alphabet).reshape([n,Nb]) #Detected bits
    errorsBit = Nb-np.sum(Bndetected==Bn,axis=1) # N° Errors per bit
    berLMMSE = np.hstack([berLMMSE,(errorsBit/Nb).reshape([n,1])])
```

Código 2.7: Detección con método LMMSE

El **Código 2.7** muestra el algoritmo empleado para la detección con el método lineal de mínimos cuadrados. La mayor parte del código se asimila al anterior método, siendo diferente únicamente la línea en la que se realiza la detección, en la que se escribe la ecuación (2.6).

2.1.6.3 Método de Máxima Verosimilitud (Maximum Likelihood, ML)

El método de Máxima Verosimilitud trata de resolver el siguiente problema [9]:

$$\tilde{\mathbf{x}}(\mathbf{y}) = \text{argmin}_{\mathrm{x}\in\chi^{M_t}}\|\mathbf{y} - \mathbf{H}\mathbf{x}\|^2 \tag{2.7}$$

, donde x ∈ χ^{M_t} hace referencia a una combinación de todo el conjunto posible de combinaciones transmitidas.

Se trata de un detector óptimo desde el punto de vista de minimizar la probabilidad de error, asumiendo que todas las *x* son equiprobables. Su funcionamiento consiste en evaluar las posibles combinaciones de *x* a la entrada y quedarse con aquella en la que el resultado de la ecuación (2.7) sea menor. El hecho de tener que tratar con todas las combinaciones posibles hace que la complejidad del algoritmo incremente exponencialmente según el número de antenas y número de puntos de la constelación haya. Por ejemplo, para un sistema 2x2 que utilice una 2-PAM se tendría:

	Salidas Tx	Combinaciones Rx
Antena 1	0,1	00,01,10,11
Antena 2	0,1	

Tabla 2.1: Combinaciones para un sistema 2x2

Se tiene entonces que el número de combinaciones posibles es igual a M^n. Para el ejemplo previo, el número total de combinaciones es $2^2 = 4$. Si la modulación fuese una 4-PAM, se tendrían $4^2 = 16$ combinaciones. Para un sistema 4x4 con una 64-PAM el número de combinaciones ascendería a la cifra de $64^4 = 16\,777\,216$. Por este motivo, se pretende aplicar herramientas de aprendizaje profundo que faciliten el cálculo de los bits detectados.

El algoritmo del método ML viene descrito en el **Código 2.8**, que guarda bastante parecido en algunas partes con el **Código 2.5**.

```
    # MAXIMUM LIKELIHOOD (ML)
    hx = np.expand_dims(H@x,axis=2) #New axis
    Z = x[:,np.argmin(np.linalg.norm(Rn.reshape(n,1,Ns)-
hx,axis=0),axis=0)]

    Andetected = detectaSBF(Z.flatten(),alphabet).reshape([n,Ns])
#Detected symbols
    Bndetected = simbolobit(Andetected.flatten(),
alphabet).reshape([n,Nb]) #Detected bits
    errorsBit = Nb-np.sum(Bndetected==Bn,axis=1) # N° Errors per bit
    berML = np.hstack([berML,(errorsBit/Nb).reshape([n,1])])
```

Código 2.8: Detección con método ML

Se empieza creando dos vectores vacíos en los que se guardará el resultado de las tasas de error. A continuación, se crea la matriz que contiene los valores de las posibles combinaciones en recepción. En lugar de bits, se tienen los posibles puntos de la constelación detectados. Posteriormente, se crea la señal de recepción de la misma manera que vista anteriormente.

Para la detección, primero se crea la matriz en la que irán almacenados los valores de la señal detectada. A continuación, se hace una multiplicación punto por punto de la matriz `H` con la matriz `x` que alberga todas las posibles combinaciones a la entrada de la antena receptora. Al resultado se le añade una tercera dimensión y se almacena en una nueva variable. Por último, se calcula la ecuación (2.7), modificando antes las dimensiones de la variable `Rn` para que la operación no de errores. Se obtiene como resultado los índices en los que se encuentran las combinaciones con mayor verosimilitud. La señal detectada es por tanto la evaluación de estos índices en la matriz de posibles combinaciones.

Finalmente, se calculan las tasas de error de bit y de símbolo de la misma manera que para el método ZF.

En el proceso de escritura de este código, se tuvo como primera propuesta el algoritmo descrito en el **Código 2.9**, en el que se utilizaba un bucle para recorrer una a una cada columna y hallar el índice correspondiente de la mejor combinación.

```
    Z = np.empty([n,0])
    for column in Rn.T:
        y = np.array([column]).T #Used for operating with the column
        ind = np.argmin(np.linalg.norm(y-H@x,axis=0)) #Index of the
minimum
        Z = np.hstack([Z,x[:,ind].reshape(n,1)])
```

Código 2.9: Proceso de detección descartado

Este, pese a ser un algoritmo más claro de entender, conlleva mayor tiempo de ejecución que el mostrado en el **Código 2.8** cuando se tiene un número pequeño de combinaciones. Sin embargo, se ha observado que a partir de 1024 combinaciones es mejor este último código, obteniendo un menor tiempo de ejecución y consumiendo menos recursos. De todas formas, como el objetivo de este proyecto no ha sido hacer un sistema MIMO masivo ni utilizar muchos puntos en la modulación, se ha optado por utilizar la primera de las soluciones presentadas.

Sería interesante realizar una modificación del **Código 2.8** de manera que disminuyese el uso que se hace de la memoria del ordenador, permitiendo así ejecutar el programa más rápidamente y salvando recursos.

2.1.7 BER

En una siguiente parte del programa se calculan los errores teóricos y simulados, atendiendo a las ecuaciones (2.8) y (2.9) [1].

$$P_M = \frac{2(M-1)}{M} Q\left(\sqrt{\frac{6\log_2 M}{M^2-1}\frac{E_{bav}}{N_0}}\right) \tag{2.8}$$

$$P_b = \frac{P_M}{\log_2 M} \tag{2.9}$$

El algoritmo se muestra en el **Código 2.10**. Las tasas de error simuladas se calculan haciendo una media ponderada de los errores correspondientes a cada antena. El error teórico hace referencia al error comedido en un canal SISO AWGN.

```
#%% BER

SNRdb2 = np.arange(SNRmax, step=0.01)    #SNR array for theoretical
calculus
SNR2 = 10**(SNRdb2/10)                   #SNR in narutal units

print('\n\nCalculating theoretical symbol error rate')
serTheo = (2*(M-1)/M)*Qfunct(np.sqrt((6*np.log2(M)/(M**2-1))*SNR2))
berTheo = serTheo/np.log2(M)

print('Calculating simulated bit error rate')
berZFav    = sum(berZF)/n
berLMMSEav = sum(berLMMSE)/n
berMLav    = sum(berML)/n
```

Código 2.10: Cálculo del BER

2.1.8 Representación de los resultados

La última parte del programa consiste en la visualización de la gráfica correspondiente a la tasa de error de bit, comparando los resultados teóricos con los obtenidos en la simulación. El proceso se muestra en el **Código 2.11**.

```
#%% RESULTS

print('\nShowing results')
font = {'weight' : 'normal',
        'size'   : 20}
plt.rc('font', **font)

# BER per antenna
plt.figure(1,figsize=(10,7))
legend = ('ZF-1','ZF-2','LMMSE-1','LMMSE-2','ML-1','ML-2','Theor. SISO')
plt.semilogy(SNRdb,berZF.T,'*',SNRdb,berLMMSE.T,'v',SNRdb,berML.T,'x',
SNRdb2,berTheo,'-k')
plt.axis([SNRmin,SNRmax,10**-7,1])
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('BER per antenna')

plt.show()

# Averaged BER
plt.figure(2,figsize=(10,7))
legend = ('ZF','LMMSE','ML','Theoretical SISO')
plt.semilogy(SNRdb,berZFav,'*r',SNRdb,berLMMSEav,'vg',SNRdb,berMLav,'xb',
SNRdb2,berTheo,'-k')
plt.axis([SNRmin,SNRmax,10**-7,1])
```

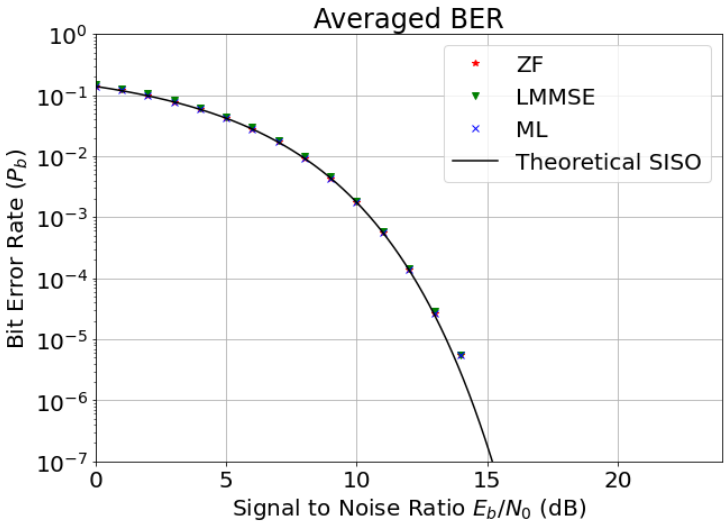
```
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('Averaged BER')

plt.show()
```

Código 2.11: Representación de los resultados

2.2 Resultados

En este apartado se muestran algunos ejemplos estudiados en la simulación del programa para distintos tipos de matrices del canal. Se pretende comparar las diferencias existentes entre el Zero Forcing, el LMMSE y el método de Máxima Verosimilitud.

2.2.1 Matriz identidad



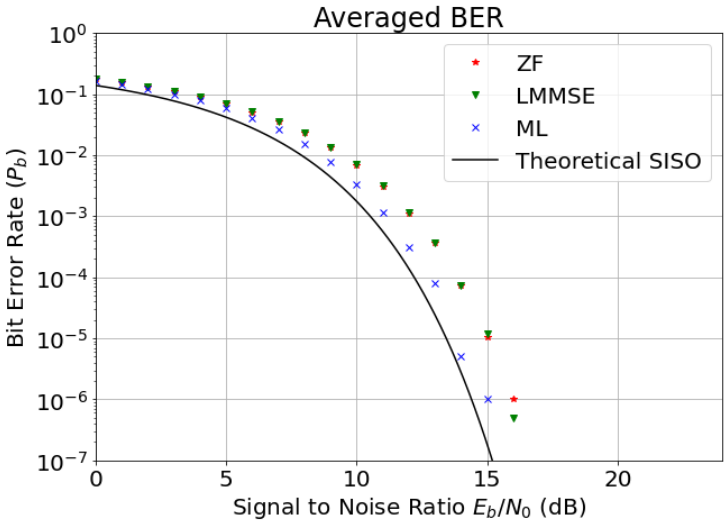
$$\mathbf{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figura 2.2: BER para Matriz 1 Matriz 1

La **Figura 2.2** muestra uno de los casos más sencillos, en los que **H** está compuesta únicamente por unos en su diagonal. El error promedio para todos los métodos es el mismo, pero este dista de ser un ejemplo cercano a la realidad, ya que a cada antena únicamente le llega la señal de una sola antena más la componente de ruido correspondiente. Los resultados se corresponden con el caso teórico de un canal Single-Input Single-Output AWGN.

2.2.2 Matriz identidad + coefecientes

En la **Figura 2.3** se puede observar que los errores son ligeramente diferentes al haber cambiado los coeficientes de fuera de la diagonal, como muestra la Matriz 2. Ahora, cada antena receptora recibe información tanto de la primera antena como de la segunda, junto con una componente de ruido. Se puede apreciar que existe una ligera mejora en el algoritmo ML, pero no es significativa para valores bajos de SNR.



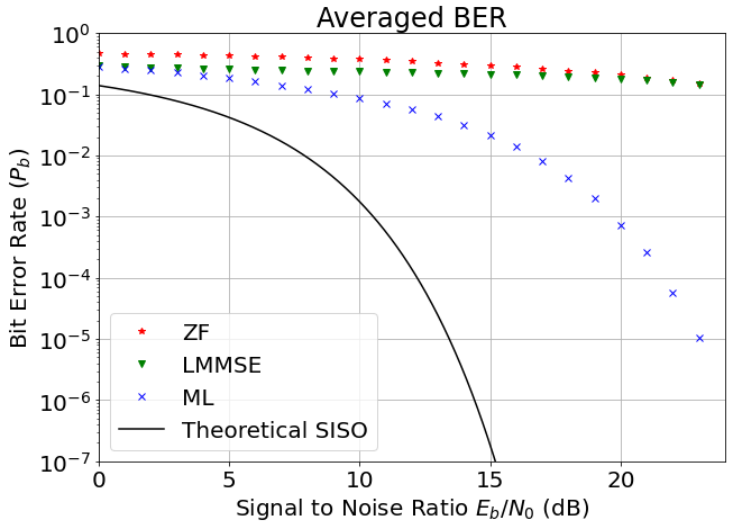
$$\mathbf{H} = \begin{bmatrix} 1 & 0{,}45 \\ 0{,}25 & 1 \end{bmatrix}$$

Figura 2.3: BER para Matriz 2 Matriz 2

2.2.3 Matriz casi no invertible

En el último caso se tiene como *matriz del canal* a la Matriz 3. Se puede observar que sus filas son casi idénticas, lo que la hace estar cerca de ser no invertible. Cabe recordar que cuando una matriz es no invertible su determinante es cero. Para este caso, el determinante de **H** es 0,049, lo cual está bastante próximo. También se obtiene la misma conclusión calculando el número de condición de la matriz, (2.4), que para este caso daría un valor de 42.026, que al ser muy lejano de la unidad se dice que está *mal condicionada* [10].

Si se aplica el método de detección ZF, que hace uso de la inversa de **H**, (2.3), lo que se tiene es una gran amplificación del ruido, pues los coeficientes de $\mathbf{H}^{-1}$son bastante elevados. La **Figura 2.4** muestra la comparativa entre los dos métodos de detección para este último ejemplo.



$$\mathbf{H} = \begin{bmatrix} 1 & 0{,}20 \\ 1 & 0{,}25 \end{bmatrix}$$

Figura 2.4: BER para Matriz 3 Matriz 3

En esta situación, se tiene en cualquier caso un error que dista mucho de la curva teórica. Por una parte, el método LMMSE consigue mejorar los resultados del ZF cuando el aporte del ruido es muy notable. Sin embargo, a medida que este desciende, la tasa de error de bit se va asimilando más a la obtenida por el forzado a cero.

El método de Máxima Verosimilitud es el que presenta mejores prestaciones que el resto, siendo el más cercano a la curva teórica. Por ejemplo, si se quisiera tener en recepción una tasa de error de un 1%, con ML se debería

tener una relación señal a ruido de aproximadamente 16 dB. Sin embargo, para conseguir lo mismo con el método ZF o LMMSE, habría que aumentar la ganancia en la etapa de transmisión para conseguir una SNR de 36 dB aproximadamente.

Cabe destacar que las gráficas mostradas hacen referencia al promedio del error cometido en el conjunto de antenas, y no al BER que se produce en cada una, como muestra la **Figura 2.5**. Mediante este desglose del error se puede apreciar que el método LMMSE funciona bastante bien en una de las antenas, pero como la otra tiene un mal comportamiento, la tasa de error binaria promedio resulta bastante similar a la del ZF. El método ML también muestra diferencia en ambas antenas, siendo una incluso peor que una de las antenas del LMMSE, pero a la hora de realizar el promedio no se ve gravemente afectado.

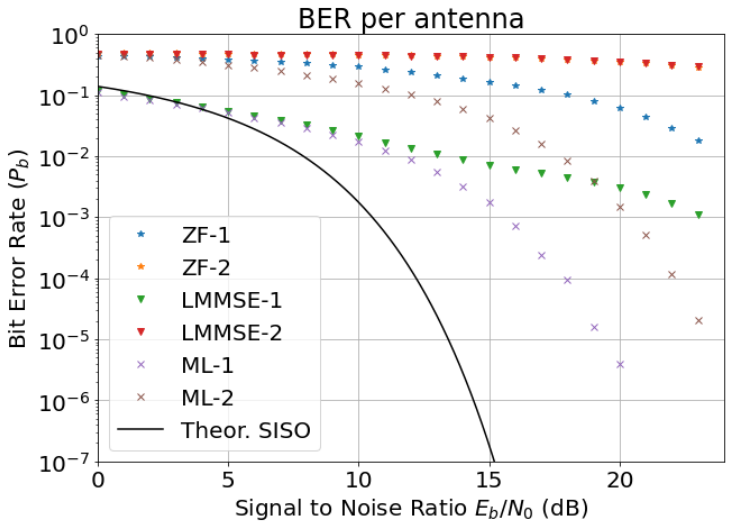


Figura 2.5: BER de cada antena

2.2.4 Matrices de coeficientes con distribución normal

Hasta ahora, se ha representado al canal de comunicaciones con matrices que, pese a aportar interesantes resultados, no tienen por qué ajustarse a un modelo real de comunicación. Por este motivo, se ha decido continuar el estudio evaluando unas nuevas matrices del canal cuyos coeficientes se generan aleatoriamente y siguen una distribución normal. En el **Código 2.12** se muestra el programa que genera estas nuevas matrices y calcula la tasa de error de bit para los tres métodos. Corresponde con el archivo *Comparison.py* [7].

```
import numpy as np
import matplotlib.pyplot as plt
from labcomdig import transmisorpamV2, Qfunct, detectaSBF, simbolobit
from functions import de2Nary

#%% INITIALIZATION

Nb = 1e5                        #Number of bits on transmission
Eb = 9                          #Average energy of the bit
M = 4                           #4-PAM modulation
n = 2                           #Antenna array NxN
SNRmin = 10                     #Minimum SNR [dB]
SNRmax = 36                     #Maximun SNR [dB]
SNRstep = 1                     #Step for SNR array[dB]
k = np.log2(M)                  #Number of bits on each symbol
Ns = int(np.floor(Nb/k))        #Number of symbols (multiple of k)
Nb = int(Ns*k)                  #Number of bits on transmission (multiple of
k)
Es = k*Eb                       #Symbol energy
```

```
verbose = 1                 #Progress indicator
Nh = 100                    #Number of Random Channels to average

#%% M-PAM TRANSMITTER

#Data generation as one sequence
#We are supposing a multi-user transmission
bn = np.random.randint(2,size=Nb*n)
[Xn,Bn,An,phi,alphabet] = transmisorpamV2(bn,Eb,M,np.ones(1),1)

#Data reshape. Each row represents an antenna
Xn = Xn.reshape([n,Ns])
An = An.reshape([n,Ns])
Bn = Bn.reshape([n,Nb])

#%% CHANNEL

SNRdb = np.arange(SNRmin,SNRmax,SNRstep) #SNR values [dB]

#Possible alphabet combinations at reception (needed in ML)
x = alphabet[de2Nary(np.arange(M**n),n,len(alphabet))]

BERav = np.empty([3,len(SNRdb),Nh])

berZFh= np.empty([n,len(SNRdb),Nh])
berLMMSEh= np.empty([n,len(SNRdb),Nh])
berMLh= np.empty([n,len(SNRdb),Nh])

normalization = True #This will avoid the Rayleigh fading behaviour
#Normalization is performed rowwise: rows of H have unit norm.

#%% TRANSMISION AND DETECTION
for jj in range(Nh):
    H = np.random.randn(n,n)
    if normalization:
        H=H/np.linalg.norm(H,axis=0)
        H=H.T
    Hinv = np.linalg.inv(H) #Inverse channel matrix

    berZF = np.empty([n,0])
    berML = np.empty([n,0])
    berLMMSE = np.empty([n,0])

    if verbose: print('\nSimulating Channel # = ', jj)
    if verbose: print('Simulating SNR = ', end = '')
    for ii in range(len(SNRdb)):
        if verbose: print('{}, '.format(SNRdb[ii]), end = '')
        SNR = 10**(SNRdb[ii]/10)          #Signal to Noise Ratio [n.u.]
        varzn = Eb/(2*SNR)                                  #Noise variance
        Wn = np.sqrt(varzn)*np.random.randn(*Xn.shape)   #AWGN

        Rn = H@Xn + Wn  #Channel output

        # ZF
        Hinv = np.linalg.inv(H)
        Z = Hinv@Rn

        Andetected = detectaSBF(Z.flatten(),alphabet).reshape([n,Ns])
#Detected symbols
        Bndetected =
simbolobit(Andetected.flatten(),alphabet).reshape([n,Nb]) #Detected bits
        errorsBit = Nb-np.sum(Bndetected==Bn,axis=1) # N° Errors per bit
        berZF = np.hstack([berZF,(errorsBit/Nb).reshape([n,1])])
```

```
        # LMMSE
        Z = np.linalg.inv(H.T@H + varzn/Es*np.eye(n))@H.T@Rn

        Andetected = detectaSBF(Z.flatten(),alphabet).reshape([n,Ns])
#Detected symbols
        Bndetected =
simbolobit(Andetected.flatten(),alphabet).reshape([n,Nb]) #Detected bits
        errorsBit = Nb-np.sum(Bndetected==Bn,axis=1) # N° Errors per bit
        berLMMSE = np.hstack([berLMMSE,(errorsBit/Nb).reshape([n,1])])

        # ML
        hx = np.expand_dims(H@x,axis=2) #New axis
        Z = x[:,np.argmin(np.linalg.norm(Rn.reshape(n,1,Ns)-
hx,axis=0),axis=0)]

        Andetected = detectaSBF(Z.flatten(),alphabet).reshape([n,Ns])
#Detected symbols
        Bndetected =
simbolobit(Andetected.flatten(),alphabet).reshape([n,Nb]) #Detected bits
        errorsBit = Nb-np.sum(Bndetected==Bn,axis=1) # N° Errors per bit
        berML = np.hstack([berML,(errorsBit/Nb).reshape([n,1])])

    print('\nCalculating simulated bit error rate')
    berZFh[:,:,jj] = berZF
    berLMMSEh[:,:,jj] = berLMMSE
    berMLh[:,:,jj] = berML

berZFhav    = np.sum(berZFh,axis=2)/Nh
berLMMSEhav = np.sum(berLMMSEh,axis=2)/Nh
berMLhav    = np.sum(berMLh,axis=2)/Nh

#%% BER

SNRdb2 = np.arange(SNRmax, step=0.01)    #SNR array for theoretical
calculus
SNR2 = 10**(SNRdb2/10)                   #SNR in narutal units

print('\n\nCalculating theoretical bit error rate')
serTheo = (2*(M-1)/M)*Qfunct(np.sqrt((6*np.log2(M)/(M**2-1))*SNR2))
berTheo = serTheo/np.log2(M)

#%% RESULTS

print('\nShowing results')
legend = ('ZF','LMMSE','ML','Theoretical SISO')
font = {'weight' : 'normal',
        'size'   : 20}
plt.rc('font', **font)

# BER per antenna
plt.figure(1,figsize=(10,7))
legend = ('ZF-1','ZF-2','LMMSE-1','LMMSE-2','ML-1','ML-2','Theor. SISO')
plt.semilogy(SNRdb,berZFhav.T,'*',SNRdb,berLMMSEhav.T,'v',
SNRdb,berMLhav.T,'x',SNRdb2,berTheo,'-k')
plt.axis([SNRmin,SNRmax,10**-7,1])
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('BER per antenna')

plt.show()
```

```
# Averaged BER
plt.figure(2,figsize=(10,7))
legend = ('ZF','LMMSE','ML','Theoretical SISO')
plt.semilogy(SNRdb,sum(berZFhav)/n,'*r',SNRdb,sum(berLMMSEhav)/n,'+g',
SNRdb,sum(berMLhav)/n,'xb',SNRdb2,berTheo,'-k')
plt.axis([SNRmin,SNRmax,10**-7,1])
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('Averaged BER')

plt.show()
```

Código 2.12: Sistema MIMO para varios canales

A continuación, se muestra una parte de los resultados obtenidos. La **Figura 2.6** muestra el error promedio que se comete en la simulación de los 100 canales diferentes creados.

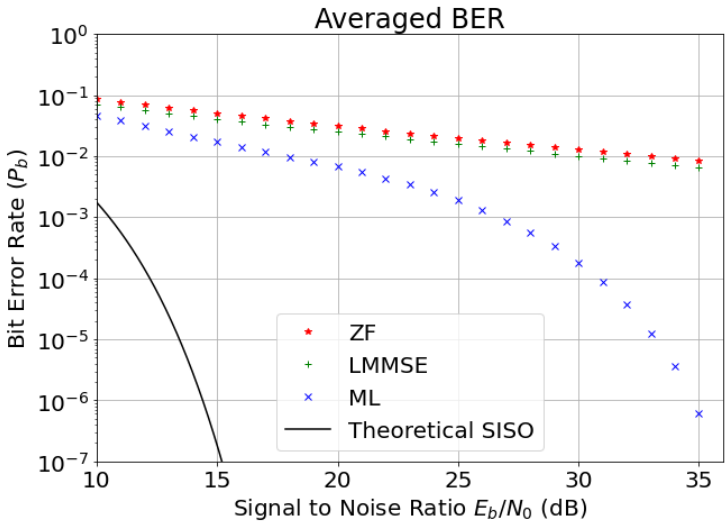


Figura 2.6: BER promedio para los 100 canales

Se puede observar que el método de máxima verosimilitud funciona correctamente, pues en la mayoría de los casos permite detectar la señal con la misma probabilidad de error que los otros métodos con una menor relación señal a ruido.

Sin embargo, existe el problema relacionado con su carga computacional, pues tener que calcular por cada nivel de amplitud recibido su valor esperado puede ser demasiado complejo, en especial cuando se tiene número elevado de combinaciones posibles. En los dos primeros ejemplos vistos, se han usado tramas de 10 kbits, 10 valores de SNR, dos antenas a cada lado y una modulación 4-PAM. Un fichero que sólo ejecute el algoritmo del método de detección ML tarda 0,046 segundos para este caso, con 16 combinaciones[1]. Si se cambia el escenario a un sistema 7x7, se tienen $4^7 = 16384$ combinaciones, lo que conlleva un tiempo de ejecución de aproximadamente 3 minutos, haciendo a su vez un consumo excesivo se los recursos del ordenador. Para un

[1] CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz; RAM: 8 GB

sistema que utilice 8 antenas en transmisión y recepción no se podría ejecutar el programa, pues las 65536 combinaciones requieren más espacio en memoria del que dispone el equipo de trabajo.

3 SIMULACIÓN DE SISTEMA MIMO CON PYTORCH

La segunda parte de este proyecto pretende pasar el sistema creado con anterioridad a un nuevo formato basado en el uso de tensores. Para ello, ha sido necesario el uso de *PyTorch*, de forma que se pudiera hacer este cambio manteniendo el uso del mismo lenguaje y entorno de programación [4]. Se ha optado esta vez por ejecutar el código en Google Colaboratory, para poder hacer uso de una GPU de manera gratuita [12].

3.1 Introducción

Uno de los conceptos más frecuentes en Machine Learning es el de *tensor*, pues es la estructura de datos principal de una red neuronal. La generalización matemática de un tensor se puede entender con mayor precisión con la siguiente tabla.

Índices	Estructura de datos	Matemáticas
0	Número	Escalar
1	Array	Vector
2	2d array	Matriz
n	nd array	nd tensor

Tabla 3.1: Interpretación de un tensor según el número de índices

Para el uso de tensores en el programa, es necesario importar la librería *torch* de PyTorch. La clase *torch.tensor* permite trabajar cómodamente con tensores tanto en la CPU como en la GPU, estando está última mejor diseñada para trabajar con ellos [13]. En la siguiente sección se muestra una nueva versión del sistema anterior visto, usando ahora tensores [14].

3.2 Creación del Sistema

En el **Código 3.1** se muestra un sistema de comunicaciones MIMO empleando tensores. Se han comparado nuevamente los tres métodos de detección para comprobar si existe alguna mejora en alguno de ellos al usar tensores. Este programa se encuentra en el archivo *MIMO_pytorch.ipynb* y puede ser ejecutado en Google Colab, añadiendo una aceleración por hardware y subiendo los archivos *labcomdig.py, functions.py* y *network.py.* Si se prefiere hacer una ejecución de forma local también se puede optar por correr el archivo *mimo_pytorch.py* [7].

```
"""MIMO_pytorch.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1P1ZiShtYZfPwNXDe9gXfCpzSfc_snxdp
"""
```

```
import torch
import matplotlib.pyplot as plt
from functions import *

"""# INITIALIZATION"""

Nb = 1e6                          #Number of bits on transmission
Eb = 9                            #Average energy of the bit
M = 4                             #4-PAM modulation
n = 2                             #Antenna array NxN
SNRmin = 0                        #Minimum SNR [dB]
SNRmax = 24                       #Maximun SNR [dB]
SNRstep = 1                       #Step for SNR array[dB]
k = torch.log2(torch.tensor(M)) #Number of bits on each symbol
Ns = int(torch.floor(Nb/k))       #Number of symbols (multiple of k)
Nb = int(Ns*k)                    #Number of bits on transmission
(multiple of k)
Es = k*Eb                         #Symbol energy
verbose = 1                       #Progress indicator

#Data generation as one sequence
#We are supposing a multi-user transmission
bn = torch.randint(0,2,(1,Nb*n))

"""# M-PAM TRANSMITTER"""

[Xn,Bn,An,phi,alphabet] = t_transmisorpamV2(bn,Eb,M,torch.ones(1),1)

#Data reshape. Each row represents an antenna
Xn = Xn.reshape(n,Ns)
An = An.reshape(n,Ns)
Bn = Bn.reshape(n,Nb)

"""# CHANNEL"""

H = torch.eye(n)           #Channel matrix
#H = torch.tensor(((1, 0.45),(0.25, 1)))
#H = torch.tensor(((1, 0.2),(1, 0.25)))

Hinv = torch.linalg.inv(H)

SNRdb = torch.arange(SNRmin,SNRmax,SNRstep) #SNR values

"""# TRANSMISION AND DETECTION"""

berZF= torch.empty(n,0)
berLMMSE = torch.empty(n,0)
berML= torch.empty(n,0)

#Possible alphabet combinations at reception
x = alphabet[t_de2Nary(torch.arange(M**n),n,len(alphabet)) ]

if verbose: print('Simulating SNR = ', end = '')
for ii in range(len(SNRdb)):
    if verbose: print('{}, '.format(SNRdb[ii]), end = '')

    SNR = 10**(SNRdb[ii]/10)          #Signal to Noise Ratio [n.u.]
    varzn = Eb/(2*SNR)                              #Noise variance
    Wn = torch.sqrt(varzn)*torch.randn(*Xn.shape)  #AWGN

    Rn = H@Xn + Wn  #Channel output

    # ZERO FORCING (ZF)
```

```
    Z = Hinv@Rn

    Andetected = t_detectaSBF(Z.flatten(),alphabet).reshape(n,Ns)
#Detected symbols
    Bndetected =
t_simbolobit(Andetected.flatten(),alphabet).reshape(n,Nb) #Detected bits
    errorsBit = Nb-torch.sum(Bndetected==Bn,axis=1) # N° Errors per bit
    berZF = torch.hstack((berZF,(errorsBit/Nb).reshape(n,1)))

    # LINEAR MINIMUM MEAN SQUARED ERROR (LMMSE)
    Z = torch.linalg.inv(H.T@H + varzn/Es*torch.eye(n))@H.T@Rn

    Andetected = t_detectaSBF(Z.flatten(),alphabet).reshape(n,Ns)
#Detected symbols
    Bndetected =
t_simbolobit(Andetected.flatten(),alphabet).reshape(n,Nb) #Detected bits
    errorsBit = Nb-torch.sum(Bndetected==Bn,axis=1) # N° Errors per bit
    berLMMSE = torch.hstack((berLMMSE,(errorsBit/Nb).reshape(n,1)))

    # MAXIMUM LIKELIHOOD (ML)
    hx = torch.unsqueeze(H@x,2) #H@x with shape (n,combinations,1)
    Z = x[:,torch.argmin(torch.linalg.norm(Rn.reshape(n,1,Ns)-
hx,dim=0),axis=0)]

    Andetected = t_detectaSBF(Z.flatten(),alphabet).reshape(n,Ns)
#Detected symbols
    Bndetected =
t_simbolobit(Andetected.flatten(),alphabet).reshape(n,Nb) #Detected bits
    errorsBit = Nb-torch.sum(Bndetected==Bn,axis=1) # N° Errors per bit
    berML = torch.hstack((berML,(errorsBit/Nb).reshape(n,1)))

"""# BER"""

SNRdb2 = torch.arange(0,SNRmax,step=0.01)   #SNR array for theoretical
calculus
SNR2 = 10**(SNRdb2/10)                    #SNR in natural units

print('\n\nCalculating theoretical bit error rate')
serTheo = (2*(M-
1)/M)*t_Qfunct(torch.sqrt((6*torch.log2(torch.tensor(M))/(M**2-
1))*SNR2))
berTheo = serTheo/torch.log2(torch.tensor(M))

print('Calculating simulated bit error rate')
berZFav    = sum(berZF)/n
berLMMSEav = sum(berLMMSE)/n
berMLav    = sum(berML)/n

"""# RESULTS"""

print('\nShowing results')
font = {'weight' : 'normal',
        'size'   : 20}
plt.rc('font', **font)

# BER per antenna
plt.figure(1,figsize=(10,7))
legend = ('ZF-1','ZF-2','LMMSE-1','LMMSE-2','ML-1','ML-2','Theor. SISO')
plt.semilogy(SNRdb,berZF.T,'*',SNRdb,berLMMSE.T,'v',SNRdb,berML.T,'x',
SNRdb2,berTheo,'-k')
plt.axis([SNRmin,SNRmax,10**-7,1])
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
```

```
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('BER per antenna')

plt.show()

# Averaged BER
plt.figure(2,figsize=(10,7))
legend = ('ZF','LMMSE','ML','Theoretical SISO')
plt.semilogy(SNRdb,berZFav,'*r',SNRdb,berLMMSEav,'vg',SNRdb,berMLav,
'xb',
SNRdb2,berTheo,'-k')
plt.axis([SNRmin,SNRmax,10**-7,1])
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('Averaged BER')

plt.show()
```

Código 3.1: Sistema MIMO en PyTorch

El cambio ha consistido principalmente en modificar la librería de los métodos, pasando de *numpy* a *torch.* Respecto a las funciones importadas de *functions.py* y *labcomdig.py*, se han reescrito algunas de ellas para que trabajen con tensores, como se muestra en la **Tabla 3.2**. La estructura del programa se mantiene casi intacta respecto a los ejemplos anteriores.

Función en Numpy	Función en PyTorch
Qfunct	*t_Qfunct*
gray2de	*t_gray2de*
de2gray	*t_de2gray*
simbolobit	*t_simbolobit*
detectaSBF	*t_detectaSBF*
transmisorpamV2	*t_transmisorpamV2*
de2Nary	*t_de2Nary*

Tabla 3.2: Lista de funciones modificadas

3.3 Resultados

En cuanto al cálculo de la tasa de error binario (BER), no se esperaba ninguna diferencia entra usar una librería u otra, pues el resultado debía de ser el mismo. Como muestra la **Figura 3.1**, se obtienen las mismas gráficas al usar las matrices del canal vistas anteriormente.

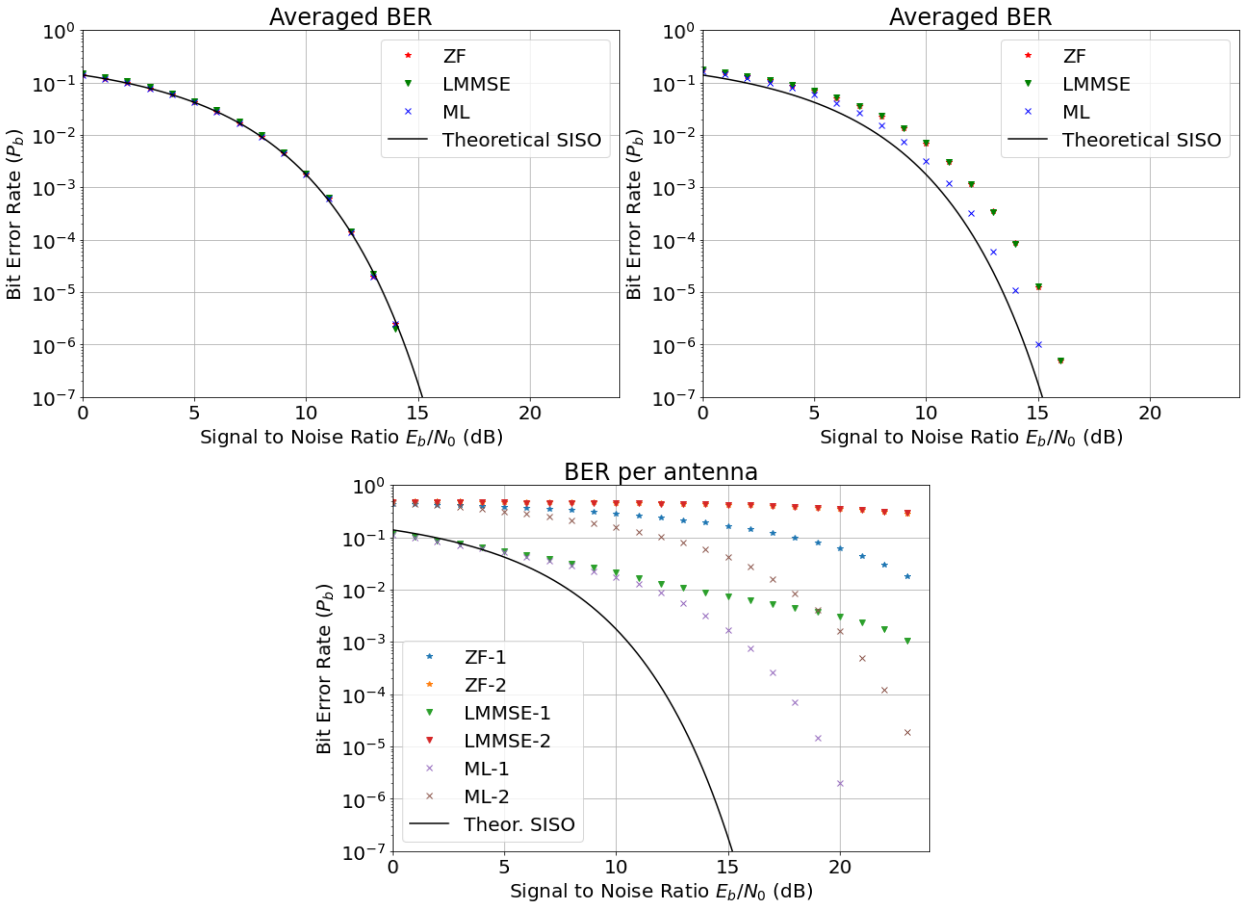


Figura 3.1: BER para las matrices 1, 2 y 3

Sin embargo, si se esperaba encontrar una diferencia en lo que se refiera a tiempos de ejecución. Este nuevo código, ejecutado en Google Colaboratory y aprovechando el uso de una GPU, permite reducir el tiempo de ejecución de los método Cancelador y LMMSE. Se ha probado a aumentar el número de antenas para tener unas matrices **H** y $\mathbf{H}^{-1}$ más grandes y así evaluar una posible mejora en la inversión de matrices. Por ejemplo, al usar 8 o 10 antenas, se ha visto que el tiempo de ejecución del programa se reduce a la mitad respecto a su versión en *numpy*, mostrando una mejora significativa.

Sin embargo, esta misma prueba no se ha podido testar con el método de Máxima Verosimilitud, pues el hecho de usar tantas antenas requería de más memoria de la disponible. Aun así, con un número de antenas pequeño, se ha podido observar que este método funciona ligeramente peor con tensores. Haciendo diversas pruebas y observando detenidamente el código, se ha visto que el problema está en la función *torch.linalg.norm.* Se desconoce el motivo por el que esta función sea más lenta en esta librería. Una posible opción es que esté peor optimizada para el uso de tensores.

A la vista de los problemas de memoria y tiempos de ejecución que surgen del método ML mediante un algoritmo tradicional, resulta interesante entrenar a una red neuronal para que aprenda a realizar este proceso de detección y después utilizar el modelo entrenado para evaluar los símbolos detectados. De esta manera, una vez que la red esté entrenada, el proceso de detección sería prácticamente inmediato.

4 DISEÑO Y SIMULACIÓN DE DETECCIÓN MIMO BASADA EN APRENDIZAJE PROFUNDO

4.1 Introducción

El aprendizaje automático (Machine Learning) es rama del campo de la Inteligencia Artificial que permite dotar a las máquinas de capacidad de aprendizaje. Dentro de esta rama, existen diferentes técnicas que se usan según la aplicación que se necesite. Una de las más conocidas es la de las Redes Neuronales (Neural Networks).

Una Red Neuronal está compuesta, como su propio nombre indica, por un conjunto de "neuronas" que procesan información de forma jerarquizada y están interconectadas entre sí. Esta unidad de procesamiento recibe el nombre de *perceptrón* y puede interpretarse de la siguiente manera:

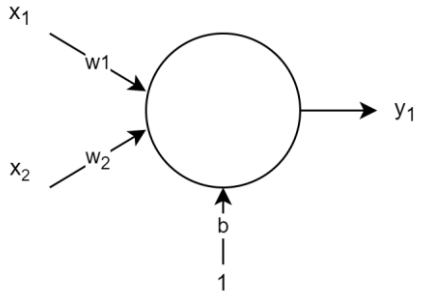


Figura 4.1: Esquema de un perceptrón

Dados unos datos de entrada x_1 y x_2, a la salida se tiene la suma de estos múltiplicados por unos coeficientes conocidos como *pesos (w_i)* más un término independiente llamado *bias (b).*

$$y_1 = w_1 \cdot x_1 + w_2 \cdot x_2 + b \tag{4.1}$$

A la salida de cada perceptrón se pasa el resultado por una *función de activación*, que distorsionará el valor de salida añadiendo deformaciones no lineales. De esta forma, se podrá encadenar de forma efectiva la computación de varias neuronas.

Cuando se agrupan varios perceptrones lo que se tiene es un Multilayer Perceptron (MLP), o lo que es lo mismo, una Red Neuronal. Un MLP está compuesto por varias capas, siendo una capa un conjunto de perceptrones que pertenecen al mismo nivel jerárquico. Así, se tiene una capa de entrada por la que se introducen los datos de partida; al menos una capa oculta, que procesará esos datos de entrada para pasarlos; y una capa de salida, en la que se devuelven los datos ya tratados. En la **Figura 4.2** se muestra un ejemplo de una Red Neuronal.

Al principio, la red es ingenua y desconoce como transformar la información de entrada correctamente. Por ello, es necesario entrenar a la red con ejemplos de datos reales y, en función del error cometido, ajustar los parámetros de la red para ir acercándonos más al resultado esperado.

A medida que aumentan la complejidad de los algoritmos y el número de capas, a este tipo aprendizaje automático se le conoce como *aprendizaje profundo* (Deep Learning).

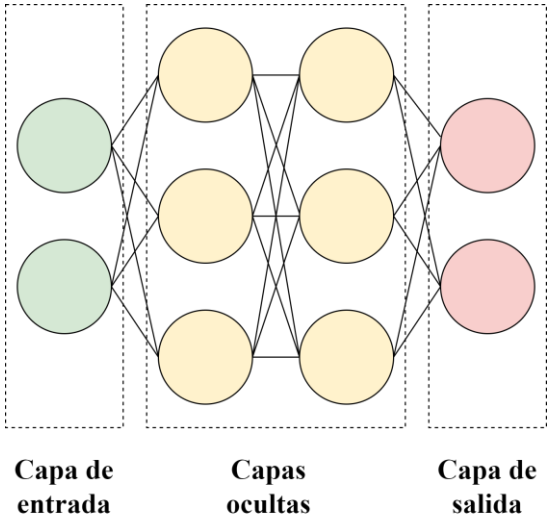


Figura 4.2: Esquema de una Red Neuronal sencilla

4.2 Creación del Sistema

El conjunto de códigos presentados a continuación pertenece al archivo *MIMO_pytorch.ipynb* y está pensado para su ejecución en la nube a través de Google Colab, siendo recomiendo el uso de aceleración por hardware. Es necesario que se suban a la sesión los archivos *labcomdig.py, functions.py* y *network.py.* Si se prefiere hacer una ejecución de forma local también se puede optar por correr el archivo *mimo_nn.py* [7].

4.2.1 Inicialización

Para llevar a cabo la programación, el primer paso fue definir las características principales del sistema. En el **Código 4.1** se muestra una definición de variables muy similar a la vista hasta ahora en los ejemplos anteriores. Se importan primero las librerías necesarias y los archivos creados que contienen funciones y clases útiles para la programación. Después de define el número de bits usados para entrenar a la red y el usado para comprobar el funcionamiento del modelo hallado. Se define un sistema 2x2 con modulación 4-PAM sobre un canal con una relación señal a ruido máxima de 20 dB.

Posteriormente, se calcula el número de bits por símbolo y se llama a una función que devuelve el número de bits y de símbolos usados para entrenar y para el test. Esta función modifica el valor inicial de bits de entrenamiento para retornar uno múltiplo del número de combinaciones de símbolos posibles, de forma que la red se entrenará con una señal en la que todas las combinaciones tienen la misma probabilidad.

Se establece el valor de la energía del símbolo en 1 J para conseguir que el alfabeto de la señal transmitida tenga unos valores comprendidos en el intervalo (-1,1) aproximadamente.

```
"""MIMO_nn.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/16NGJu7ligCI2gNNsttWhujvsZJFkH6C_
"""

import torch
from torch import nn, optim
import matplotlib.pyplot as plt

from functions import *
from network import *
```

```
"""# INITIALIZATION"""

Nb_train = 1e3
Nb_test  = 1e4
M = 4
n = 2
SNRmin = 0                          #Minimum SNR [dB]
SNRmax = 24                         #Maximun SNR [dB]
SNRstep = 1                         #Step for SNR array[dB]
k = int(torch.log2(torch.tensor(M)))      #Number of bits on each symbol
[Nb_train,Ns_train,Nb_test,Ns_test] = correction(Nb_train,Nb_test,k,M,n)
Es = 1                      #Symbol energy
Eb = Es/k

# channel matrix
#H = torch.eye(n)
#H = torch.tensor(((1, 0.45),(0.25, 1)))
H = torch.tensor(((1, 0.2),(1, 0.25)))

Hinv=torch.linalg.inv(H)

batch_size = Nb_train//4 #16*(M**n)
valid_size = 0.2   # Percentage of training set to use as validation
verbose = 1
```

Código 4.1: Definición de los parámetros de entrada

Por último, se crea la matriz del canal, se establece el tamaño del *batch* para iterar con el dataset y se determina el porcentaje de datos que serán usados para validación dentro del conjunto de datos de entrenamiento. Los valores escogidos son los que han dado mejores resultados tras varias pruebas realizadas.

4.2.2 La red

El siguiente ha sido el de crear la Red Neuronal. Para ello, se ha hecho una clase en la que se definen variables y métodos propios de la red, como se muestra en el **Código 4.2**.

Las clases creadas para este proyecto se encuentran definidas dentro del archivo *network.py*.

```
class Net(nn.Module):
    def __init__(self,M,n):
        super().__init__()
        self.fc1 = nn.Linear(n, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, M**n)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(x.shape[0], -1)

        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))

        x = F.log_softmax(self.fc3(x), dim=1)

        return x

    def restart(self):
        self.fc1.bias.data.fill_(0)
        self.fc1.weight.data.normal_(std=0.01)
        self.fc2.bias.data.fill_(0)
        self.fc2.weight.data.normal_(std=0.01)
        self.fc3.bias.data.fill_(0)
```

```
        self.fc3.weight.data.normal_(std=0.01)
```

Código 4.2: Definición de la Red Neuronal

Se ha optado por usar una red con 3 capas ocultas, pues al principio, para un sistema con pocas antenas y pocos puntos de modulación, no es necesario más. La capa de entrada de la red tiene un tamaño igual al número de antenas, y por cada entrada se pasa el símbolo que le llega a cada antena receptora. A la salida se tienen M^n valores, que hacen referencia a la combinación de símbolos que se ha introducido a la entrada. El tamaño de las capas ocultas ha sido definido a base de ir probando con distintas configuraciones.

La función de activación escogida para modificar los valores de cada capa ha sida la *tangente hiperbólica*, que tiene la siguiente forma:

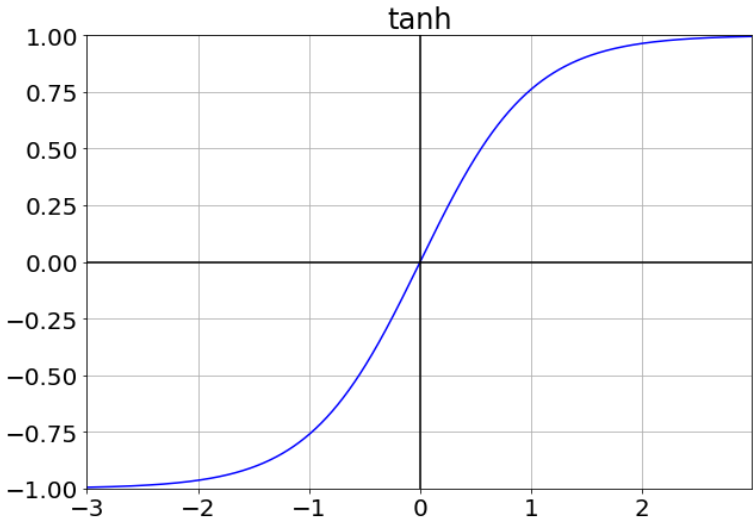


Figura 4.3: Función *tanh*

Tras hacer muchas pruebas con varias funciones, como pueden ser las funciones *sigmoide* o *ReLu*, se observó que, tanto en resultado como en rapidez, la función *tangente hiperbólica* era que la que mejor funcionaba en el modelo.

Por último, se pasan los valores de salida por la función *softmax* para calcular la probabilidad de cada combinación. Esta función mapea cada entrada a un valor entre 0 y 1 y normaliza los valores para darle una distribución de probabilidad adecuada donde todas las probabilidades suman uno.

$$\sigma(x_i) = \frac{e^{x_i}}{\Sigma_k^K e^{x_k}} \tag{4.2}$$

En el **Código 4.3** se inicializa la red y se definen la función de pérdidas y el optimizador. La función de pérdidas (también llamada *coste*) escogida ha sido la *Negative Log Likelihood Loss*, la cual es muy útil para problemas de clasificación. Para actualizar el peso de los gradientes una vez calculado el error, el optimizador escogido ha sido el *Rprop*, que implementa el algoritmo *Resilient Backpropagation.* Este ha mostrado ser el mejor optimizador en cuanto a resultado y velocidad de procesamiento, pues otros como el *SGD* o el *Adam* ofrecían un peor rendimiento y malos resultados.

```
"""# NETWORK"""
# Initialize network
model = Net(M,n)

# Loss functions
criterion = nn.NLLLoss()

# Optimizer
optimizer = optim.Rprop(model.parameters(), lr=0.001)
```

Código 4.3: Creación de la Red Neuronal

4.2.3 Dataset

Para que el modelo creado por una Red Neuronal funcione correctamente, es muy importante que previamente se haga una correcta selección de los datos que se quieren medir. Existen bases al alcance de cualquier usuario que ofrecen una buena recopilación de datos que pueden ser usados para la creación de una Red Neuronal. Por ejemplo, la base de datos MNIST almacena dígitos escritos a mano usados comúnmente para entrenar varios sistemas de procesamiento de imágenes [15].

Dado que este proyecto requería unos datos más específicos, no se ha podido encontrar una base de datos que cumpliese con las características necesarias. Por este motivo, ha sido necesario crear manualmente la información que la red necesita como entrada. Se ha optado por crear una clase propia para facilitar el tratamiento de los datos y, para una mejor lectura del código, se han construido dos funciones para todo el proceso de generar la información. En el **Código 4.4** se muestra la creación de esta clase.

```
class SymbolsDataset(Dataset):

    def __init__(self, Rn, Cn, Bn):
        self.Rn = Rn    # Symbols
        self.Cn = Cn    # Combinations
        self.Bn = Bn    # Bits

    def __len__(self):
        return len(self.Rn.T)

    def __getitem__(self, idx):
        if is_tensor(idx):
            idx = idx.tolist()

        symbol = self.Rn[:,idx]
        comb = self.Cn[idx]

        k = int(len(self.Bn.T) / len(self.Rn.T))
        bits = self.Bn[:,k*idx:k*idx+k]

        return [symbol,comb,bits]

    def split_data(self, split):
        Nb = len(self.Bn.T)
        Ns = len(self.Rn.T)
        k = int(Nb/Ns)

        train_Rn, valid_Rn = torch.split(self.Rn, [Ns-
split,split],dim=1)
        train_Cn, valid_Cn = torch.split(self.Cn, [Ns-split,split])
        train_Bn, valid_Bn = torch.split(self.Bn, [Nb-
k*split,k*split],dim=1)

        return [SymbolsDataset(train_Rn,train_Cn,train_Bn),
SymbolsDataset(valid_Rn,valid_Cn,valid_Bn)]
```

Código 4.4: Clase usada para tratar el dataset

La clase creada hereda de la clase *Dataset*, una clase perteneciente a *PyTorch* que permite un manejo más cómo de los datasets [16]. El conjunto de datos creados está formado por:

- Una matriz Rn que guarda los valores de los símbolos que llegan a recepción (muestra).
- Una matriz Bn en la que se almacenan los bits transmitidos (etiqueta).
- Una matriz Cn que hace referencia a la combinación del símbolo transmitido (etiqueta).

A la red se le introduce como entrada cada columna de Rn y se compara el resultado de la salida con la combinación correcta almacenada en Cn. A esta forma de entrenar a la red se la conoce como *aprendizaje supervisado*, pues la red puede evaluar si el resultado que da es correcto o no gracias a que la respuesta está contenida en el dataset.

Se ha creado un método que retorna la longitud de la cadena de símbolos; otro que devuelve un item de la clase, retornando los símbolos recibidos en cada antena, el número de la combinación de símbolos transmitida y los bits correspondientes a esa combinación; y un último que divide el conjunto de datos para separarlos en datos de entrenamiento y datos de validación.

```
def generate_data(H,Nb,Eb,M,n,SNRdb,train):
    k = int(torch.log2(torch.tensor(M)))      #Number of bits on each
symbol
    Ns = int(torch.floor(torch.tensor(Nb/k))) #Number of symbols
(multiple of k)

    alphabet = torch.sqrt(3*Eb*torch.log2(torch.tensor(M))/(M**2-
1))*(2*(torch.arange(M))-M+1)

    # possible combinations
    ind = t_de2Nary(torch.arange(M**n),n,M)
    x = alphabet[ind]

    if train:  # data for training. same number of symbols transmitted

        Xn = x.tile(int(Ns/x.size(1)))
        Xn = Xn[:,torch.randperm(Xn.size(1))] # Shuffle
        Bn = t_simbolobit(Xn.flatten(),alphabet) #Detected bits

        #Data reshape. Each row represents an antenna
        Bn = Bn.reshape(n,Nb)

        #Index of the combinations
        Cn = getidx(x,Xn,n)

    else:     # data for testing

        # fixed seed for same results always
        torch.manual_seed(1)

        bn = torch.randint(0,2,(1,Nb*n)) #Bits to transmit
        [Xn,Bn,An,phi,alphabet] =
t_transmisorpamV2(bn,Eb,M,torch.ones(1),1)

        #Data reshape. Each row represents an antenna
        Xn = Xn.reshape(n,Ns)
        Bn = Bn.reshape(n,Nb)
        An = An.reshape(n,Ns)

        #Index of the combinations
        Cn = getidx(x,An,n)

    SNR = 10**(SNRdb/10)    #Signal to Noise Ratio [n.u.]
    varzn = Eb/(2*SNR)          #Noise variance
    Wn = torch.sqrt(varzn)*torch.randn(*Xn.shape)  #AWGN

    Rn = H@Xn + Wn

    return [Rn,Cn,Bn.long(),x,alphabet]
```

Código 4.5: Función *generate_data*

La información que se transmite por el canal se crea en la función descrita en el **Código 4.5**. Dependiendo de si los datos se van a usar para entrenar a la red o no, se hace una distinción en el proceso. Para el primer caso, se

crea una matriz con todas las combinaciones posibles de símbolos a transmitir y se repite las veces necesarias, hasta completar el número de símbolos transmitidos introducido como parámetro de la función. Luego la matriz se mezcla aleatoriamente y se crean las matrices de bits y de combinaciones. Cuando los datos van a ser utilizados para testar la red, se genera una secuencia aleatoria de bits y posteriormente se obtienen los símbolos transmitidos y su número de combinación.

Finalmente, se simula el paso de la señal generada por la antena de transmisión por el canal de comunicación, nuevamente un canal AWGN.

```
def
create_datasets(H,Nb_train,Nb_test,Eb,M,n,SNRdb,batch_size,valid_size):
    # Data generation for training
    [Rn,Cn,Bn,x,alphabet] =
generate_data(H,Nb_train,Eb,M,n,SNRdb,train=True)
    trainset = SymbolsDataset(Rn,Cn,Bn)

    # Data generation for testing
    [Rn,Cn,Bn,x,alphabet] =
generate_data(H,Nb_test,Eb,M,n,SNRdb,train=False)
    testset = SymbolsDataset(Rn,Cn,Bn)

    # Indices used for validation
    num_train = len(trainset)
    split = int(np.floor(valid_size * num_train))

    # Split data
    train_split, valid_split = SymbolsDataset.split_data(trainset,split)

    # Load training data in batches
    trainloader = torch.utils.data.DataLoader(train_split,
                                              batch_size=batch_size,
                                              num_workers=0)

    # Load validation data in batches
    validloader = torch.utils.data.DataLoader(valid_split,
                                              batch_size=batch_size,
                                              num_workers=0)

    # load test data in batches
    testloader = torch.utils.data.DataLoader(testset,
                                             batch_size=batch_size,
                                             num_workers=0)

    return [trainloader,validloader,testloader,x,alphabet]
```

Código 4.6: Función *create_datasets*

Para crear los datasets necesarios se hace uso de la función definida en el **Código 4.6**. Se genera primero la información transmitida, haciendo distinción entre los datos de entrenamiento y los de prueba o test. Una vez creados, se inicializa un objeto de la clase *SymbolsDataset*. A continuación, se separa parte de los datos de entramiento para usarlos como datos de validación. Por últmo, se hace uso de la herramienta *Dataloader* de *PyTorch* para cargar el dataset y poder iterar sobre él.

4.2.4 Entrenamiento y evaluación del modelo

Una vez definidos los parámetros iniciales, la red y la estructura de los datos a utilizar, se procede a entrenar la red. Nuevamente, dado que existen distintos valores de SNR, es necesario crear el dataset y entrenar a la red para cada uno de ellos. El proceso se describe en el **Código 4.7**.

```
"""# TRAINING AND TEST"""
```

```
n_epochs = 150
patience = 20

berTrain = torch.empty(n,0)
berValid = torch.empty(n,0)
berTest  = torch.empty(n,0)

berZF    = torch.empty(n,0)
berLMMSE = torch.empty(n,0)
berML    = torch.empty(n,0)

# Train the network for each value of SNR
SNRdb = torch.arange(0,SNRmax,1)
if verbose: print('Simulating SNR = ', end = '')
for ii in range(len(SNRdb)):
    if verbose: print('{}, '.format(SNRdb[ii]), end = '')

    SNR = 10**(SNRdb[ii]/10)      #Signal to Noise Ratio [n.u.]
    varzn = Eb/(2*SNR)                          #Noise variance

    # data generation
    [trainloader,validloader,testloader,x,alphabet] =
create_datasets(H,Nb_train,

Nb_test,Eb,

M,n,

SNRdb[ii],

batch_size,

valid_size)
    # restart weigths and bias
    model.restart()

    # train the model
    model, avg_train_losses, avg_valid_losses = train_model(model,
                                                            trainloader,
                                                            validloader,
                                                            optimizer,
                                                            criterion,
                                                            patience,
                                                            n_epochs)

    # visualize the loss as the network trained
    #printloss(avg_train_losses,avg_valid_losses,SNRdb[ii],save=False)

    # Calculate BER for training, validation and test data
    errorTrain, errorValid, errorTest =
eval_model(model,trainloader,validloader,testloader,x,alphabet,k)

    # BER of every SNR
    berTrain = torch.hstack((berTrain,errorTrain))
    berValid = torch.hstack((berValid,errorValid))
    berTest = torch.hstack((berTest,errorTest))

    # Comparison with other detection methods
    Rn = torch.hstack((trainloader.dataset.Rn, validloader.dataset.Rn))
    Bn = torch.hstack((trainloader.dataset.Bn, validloader.dataset.Bn))

    errorZF =
detect(Rn,Bn,H,alphabet,Nb_train,Ns_train,n,'ZF',Hinv=Hinv)
```

```
        errorLMMSE =
    detect(Rn,Bn,H,alphabet,Nb_train,Ns_train,n,'LMMSE',Es=Es,varzn=varzn)
        errorML = detect(Rn,Bn,H,alphabet,Nb_train,Ns_train,n,'ML',x=x)

        berZF = torch.hstack((berZF,errorZF))
        berLMMSE = torch.hstack((berLMMSE,errorLMMSE))
        berML = torch.hstack((berML,errorML))
```

Código 4.7: Entrenamiento y test del modelo

Primeramente, se definen el número de *epoch* y la *patience* y se crean variables vacías en las que se guardaran las tasas de error binario. Un *epoch* hace referencia al número de veces que el conjunto total de datos es usado para entrenar la red. En este caso, se ha establecido en 150 el número máximo de veces que se puede pasar el dataset por la red. Sin embargo, no es necesario llegar a ese máximo para conseguir un modelo válido. Gracias a separar parte de los datos de entrenamiento para usarlos como validación, se puede obtener una gráfica que muestre el error cometido en cada *epoch* para ambos conjuntos de datos.

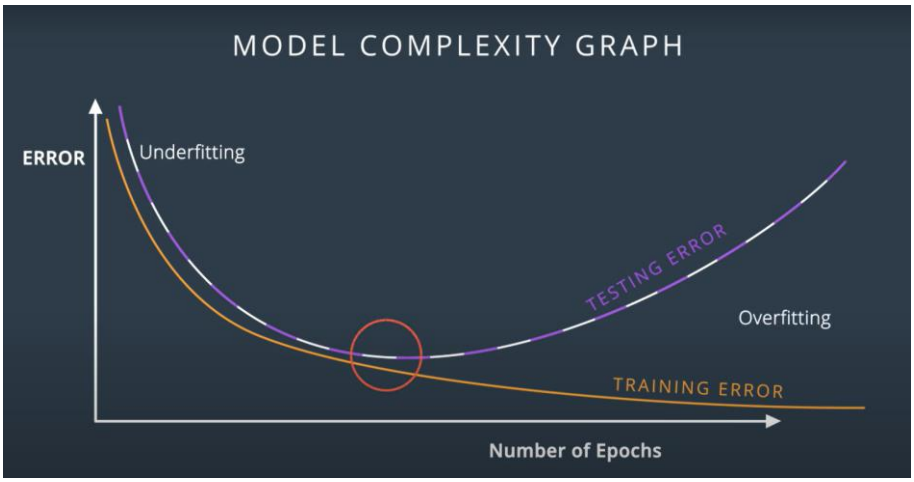


Figura 4.4: Early Stopping [14]

Para los primeros *epoch*, el error cometido será alto en ambos casos, pues apenas se ha entrenado a la red (*underfitting)*. A medida que se entrena la red pasándole varias veces el dataset, el error desciende progresivamente. Sin embargo, si se continua con ese proceso, el error para los datos de entrenamiento seguirá descendiendo, mientras que para los datos de validación, que son distintos, aumentará. En este caso, se dice que se está sobreentrenando a la red (*overfitting)*. Existe por tanto un punto intermedio en el que el modelo tiene un error aceptable para ambos conjuntos, como se muestra en la **Figura 4.4**.

Para obtener el número de *epoch* óptimo se utiliza la técnica del *Early Stopping*. Esta consiste en detectar un mínimo en el error de validación, guardar el modelo actual y, si dentro de x *epoch* el error no disminuye, detener el entrenamiento y quedarse con el modelo guardado. En el **Código 4.7** la variable *patience* define cuántos *epoch* más se realizan tras alcanzar el mínimo.

Una vez definidas las variables anteriores, se crea un tensor que almacena todos los valores de SNR a evaluar y se entre en un bucle *for* que entrena y prueba la red para cada valor del tensor. Dentro del bucle, el primer paso es el de crear el dataset mediante la función *create_datasets*. Posteriormente, se resetean los pesos guardados en el modelo anterior de la red y se hace el entrenamiento. A continuación, se evalua el modelo y se guarda el BER de los datos de entrenamiento, de validación y de test.

Para comprobar la validez del modelo creado, se calcula la tasa de error binario cometida por los tres métodos de detección conocidos mediante un algoritmo tradicional, como se muestra en las secciones anteriores.

4.2.5 BER

En el **Código 4.8** se calcula el BER promedio de las simulaciones, tanto del modelo como de los métodos

tradicionales, y el BER teórico (2.8) y (2.9) [1].

```
"""# BER"""

SNRdb2 = torch.arange(0,SNRmax,step=0.01)   #SNR array for theoretical
calculus
SNR2 = 10**(SNRdb2/10)                      #SNR in natural units

serTheo = (2*(M-
1)/M)*t_Qfunct(torch.sqrt((6*torch.log2(torch.tensor(M))/(M**2-
1))*SNR2))
berTheo = serTheo/torch.log2(torch.tensor(M))

print('\nCalculating simulated bit error rate')
berTrainAv = sum(berTrain)/n
berValidAv = sum(berValid)/n
berTestAv = sum(berTest)/n

berZFav = sum(berZF)/n
berLMMSEav = sum(berLMMSE)/n
berMLav = sum(berML)/n
```

Código 4.8: Cálculo del BER

4.2.6 Representación de los resultados

La última parte del programa consiste en la visualización de la gráfica correspondiente a la tasa de error de bit, comparando los resultados teóricos con los obtenidos en la simulación. El proceso se muestra en el **Código 4.9**.

```
"""# RESULTS"""

# Train and Validation Loss
print('\nShowing results')
legend = ('Train','Validation','Test','ZF','LMMSE','ML','Theoretical')

font = {'weight' : 'normal',
        'size'   : 20}

plt.rc('font', **font)

plt.figure(1,figsize=(10,7))
plt.semilogy(SNRdb,berTrainAv,'*b',SNRdb,berValidAv,'.r',
SNRdb,berTestAv,'+g')
plt.semilogy(SNRdb,berZFav,'^c',SNRdb,berLMMSEav,'<m',
SNRdb,berMLav,'>y',SNRdb2,berTheo,'-k')
plt.axis([0,SNRmax,10**-7,1])
plt.grid()
plt.xlabel('Signal to Noise Ratio $E_b/N_0$ (dB)');
plt.ylabel('Bit Error Rate ($P_b$)');
plt.legend(legend)
plt.title('BER with {} bits for training. Batch =
{}'.format(Nb_train,batch_size))

plt.show()
```

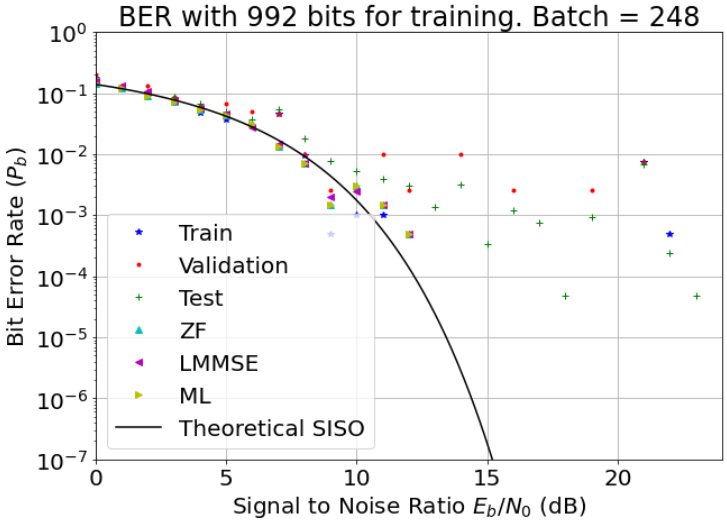
Código 4.9: Representación de los resultados

4.3 Resultados

En este apartado se muestran algunos ejemplos estudiados en la simulación del programa para distintos tipos de matrices del canal. Se pretende comparar las diferencias existentes entre el Zero Forcing, el LMMSE, el método de Máxima Verosimilitud y el modelo creado por la Red Neuronal.

4.3.1 Matriz identidad

En la **Figura 4.5** se puede observar la tasa de error binario usando como *matriz del canal* una matriz identidad. El resultado para los distintos métodos es muy similar para valores de SNR menores de 10 dB. A partir de ahí, los resultados del modelo empiezan a distanciarse del error cometido con los otros métodos, siendo difícil de intuir la tendencia que sigue.



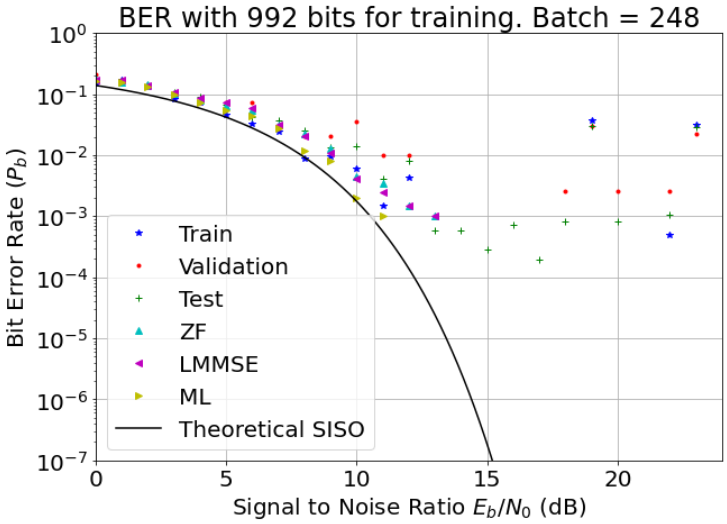
$$\mathbf{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figura 4.5: BER para Matriz 1 Matriz 1

4.3.2 Matriz identidad + coeficientes

El uso de una Matriz 2 genera una BER como la mostrada en la **Figura 4.6**. Nuevamente, se observan unas tasas de error muy similares en las distintas formas de detección para una SNR de hasta 10 dB aproximadamente. medida que se sube, el error en el modelo es peor que en los otros métodos, aunque esta vez muestra una tendencia descendente más clara.

Se puede apreciar que el método de Máxima Verosimilitud muestra resultados ligeramente superiores a los otros dos métodos de detección. Sin embardo, su modelado con mediante el uso de una Red Neuronal no consigue exactamente los mismos resultados en todos los casos, en especial para valores de SNR superiores a 10 dB.



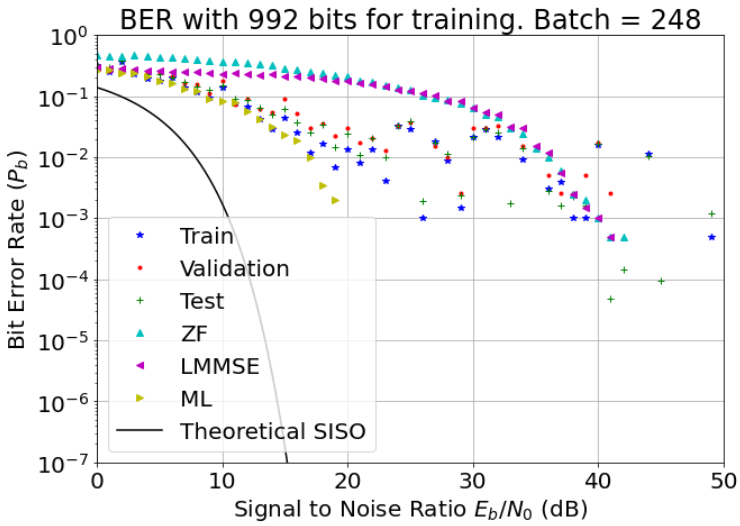
$$\mathbf{H} = \begin{bmatrix} 1 & 0{,}45 \\ 0{,}25 & 1 \end{bmatrix}$$

Figura 4.6: BER para Matriz 2 Matriz 2

4.3.3 Matriz casi no invertible

La **Figura 4.7** muestra la tasa de error de bit que se produce al tener a una matriz *mal condicionada* o casi no invertible como *matriz del canal*, como es la Matriz 3. En este caso, y como se ha visto en ejemplos anteriores, los métodos de detección ZF y LMMSE consiguen peores resultados que el método ML para un mismo valor de SNR.

Por lo que respecta al modelo de la Red Neuronal, se tiene, como hasta ahora, un resultado que sigue la traza de BER del método ML tradicional para valores bajos de SNR, pero que, a partir de 17 dB aproximadamente, el modelo no sigue una línea clara.



$$\mathbf{H} = \begin{bmatrix} 1 & 0{,}20 \\ 1 & 0{,}25 \end{bmatrix}$$

Figura 4.7: BER para Matriz 3 Matriz 3

4.4 Modificaciones

Una vez visto el sistema básico de dos antenas a cada extremo y con una modulación 4-PAM, queda por experimentar con los cambios que se producen al introducir más antenas o añadir más puntos a la modulación y ver si la estructura de la red actual es válida para estos cambios.

4.4.1 Variación del número de antenas

Primeramente, se ha probado a añadir más antenas al sistema. Se ha hecho una simulación con un sistema 4x4 y una 4-PAM como modulación. Dado que al aumentar el número de antenas aumenta exponencialmente el número de combinaciones posibles, la red propuesta anteriormente no presenta buenas características para este sistema. En la **Tabla 4.1** se muestra el cambio propuesto en el número de cada capa para esta simulación.

Capa de Entrada	Capa Oculta 1	Capa Oculta 2	Capa de Salida
n = 2	64	32	$M^n = 16$
n = 4	128	256	$M^n = 256$

Tabla 4.1: Modificación de la Red Neuronal

Ha sido necesario aumentar el número de neuronas en las dos capas ocultas debido a que el número de salidas había crecido considerablemente. De varias configuraciones probadas, la opción elegida resultó ser la más adecuada.

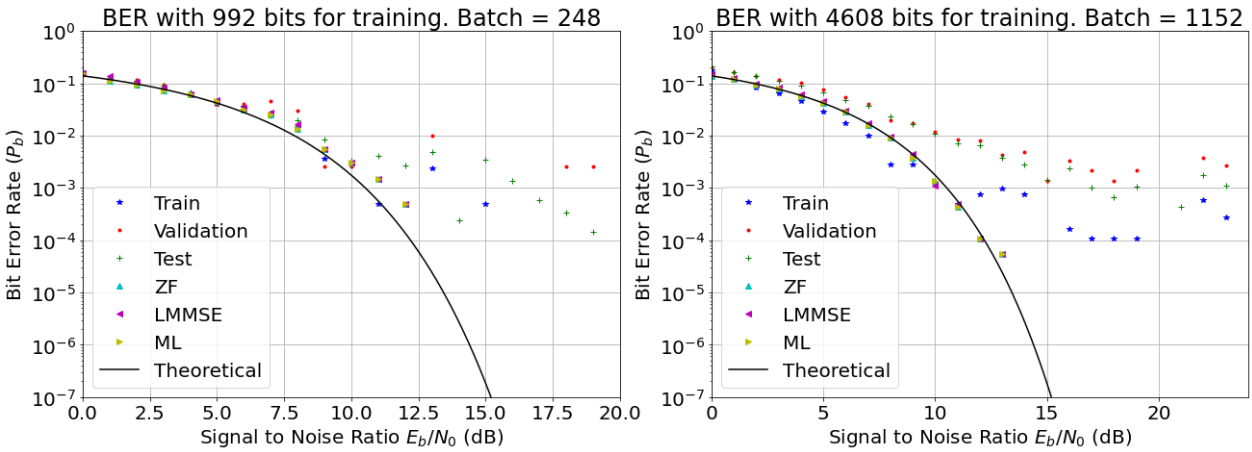


Figura 4.8: Comparación de BER para Matriz 1

En la **Figura 4.8** se muestra el resultado para el modelo antiguo (izq.) y para el nuevo (der.). Ha sido necesario aumentar el número de bits de entrenamiento y el tamaño del batch, pues la anterior configuración no era válida, ya que con el anterior tamaño no entraba todo el conjunto de posibles combinaciones. Se observa un mismo comportamiento del modelo también en este caso, obteniendo mejores resultados para valores de SNR no muy altos. Para valores más altos de SNR, pese a obtenerse una mayor tasa de error, es lo suficientemente baja para establecer una comunicación aceptable.

Se ha intentado realizar otro sistema 6x6, pero no se ha encontrado la configuración adecuada. El aumento del número de antenas parece aumentar la complejidad de la red, pues modifica gravemente el número de salidas necesarias. En este caso, se tendrían 4096 salidas, que equivale el número de combinaciones posibles que se pueden dar en la etapa transmisora. Para que el resultado de esas salidas sea el adecuado, las capas ocultas deben tener un número de neuronas acorde a esa cantidad de salidas. No se puede obtener un buen resultado si, por ejemplo, se pasa de una última capa oculta de 256 neuronas a una capa de salida con 4096. Queda entonces este sistema como propuesta para una investigación futura.

4.4.2 Variación de los puntos de modulación

El siguiente cambio a realizar ha sido el del número de puntos de la modulación. Se ha querido observar que sucede al tomar un valor de *M* más grande. Nuevamente, esta modificación afecta a la estructura de la red, por lo que ha sido necesario realizar ajustes para obtener unos resultados adecuados. Los cambios realizados de muestran en la **Tabla 4.2**.

Capa de Entrada	Capa Oculta 1	Capa Oculta 2	Capa de Salida
n = 2	64	32	$M^n = 4^2 = 16$
n = 2	64	128	$M^n = 8^2 = 64$
n = 2	128	256	$M^n = 16^2 = 256$

Tabla 4.2: Modificación de la Red Neuronal

Primeramente, se aumentó a 8 el número de puntos de la modulación. Fue necesario también incrementar el número de bits de entramiento y el tamaño del batch. El resultado se muestra en la **Figura 4.9**. Se puede observar que una modificación en el número de puntos de la constelación también afecta a la curva teórica del error para un sistema SISO, haciendo que, para una misma probabilidad de error, sea necesario un mayor valor de SNR para modulaciones con más puntos.

La gráfica obtenida muestra un comportamiento muy similar a los anteriormente vistos, con peores resultados que los algoritmos tradicionales para valores más altos de SNR, pero que aun así aseguran un funcionamiento del sistema aceptable.

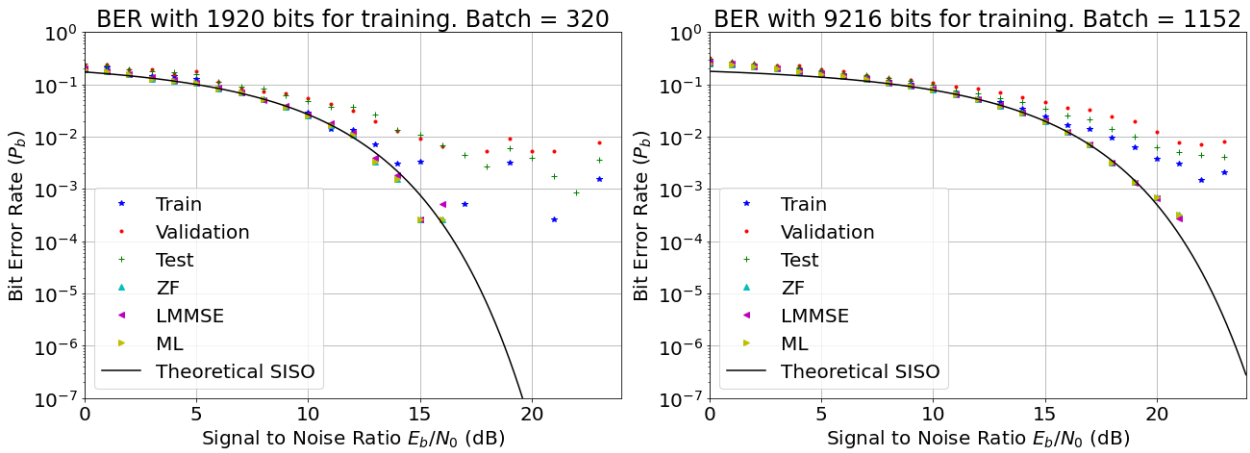


Figura 4.9: BER para 8-PAM (izq.) y 16-PAM (der.)

A continuación, se pasó a una modulación 16-PAM, siendo necesario otra vez aumentar el número de bits de entrada hasta casi 10 kbits y el tamaño del batch a 1 kbit aproximadamente. Se puede ver un resultado muy similar entre el ofrecido por la red neuronal y el generado por los otros tres métodos, por lo que se podría decir que se ha obtenido un buen modelado.

De los resultados obtenidos se puede deducir que el aumento del número de puntos de la modulación incrementa, como en la modificación anterior del número de antenas, la complejidad de la red neuronal y requiere de una mayor número de bits para su entrenamiento. Queda también como propuesta para una futura investigación del proyecto la creación de redes que trabajen con modulaciones aún más grandes.

5 CONCLUSIONES

Tras todas la pruebas realizadas durante este proyecto y los resultados mostrados en las secciones previas, se pueden extraer varias conclusiones de interés y proponer posibles líneas de futura investigación.

5.1 Conclusiones

Primeramente, en lo que respecta a los tres métodos de detección estudiados, queda claro el mejor funcionamiento del método de Máxima Verosimilitud frente al Cancelador y el LMMSE, a costa de una mayor carga computacional.

En referencia a el modelado de este método mediante una Red Neuronal, se ha observado que la gran cantidad de parámetros que conforman tanto a la red (número de capas, número de neuronas por capa, función de activación, función de coste, optimizador, *learning rate*, *dropout*, etc.) como al dataset (bits de entramiento, bits de test, porcentaje de validación, tamaño del batch, etc.) hacen que la elección de un modelo adecuado sea muy compleja, permitiendo además que existan varias configuraciones óptimas posibles.

Se ha visto que los resultados obtenidos mediante el uso técnicas de Deep Learning se ajustan notablemente a los resultados de los algoritmos tradicionales, si bien es cierto que, para señales con poco ruido, la eficacia de la Red Neuronal es mucho menor.

Por último, si se quieren usar sistemas que empleen más antenas y/o utilicen modulaciones más grandes, será necesario un aumento de la complejidad de la red neuronal y más datos de entrenamiento.

5.2 Líneas de futura investigación

De acuerdo con el apartado 2.1.6.3, queda como propuesta la mejora del algoritmo de detección para que no consuma una cantidad inmensa de recursos del ordenador cuando se trabaje con un número alto de posibles combinaciones de símbolos.

Siguiendo por este línea, se podría estudiar por qué el algoritmo de detección para el método de Máxima Verosimilitud está peor implementado en *PyTorch* que en *numpy* o proponer otra alternativa.

Como punto final, sería interesante la creación de redes más complejas que abarquen sistemas de comunicaciones con más antenas y puntos de modulación, cuya simulación mediante un algoritmo tradicional fuese inviable.

Referencias

[1] F. J. Payán Somet, J. J. Murillo Fuentes y J. C. Aradillas Jaramillo, "Manual de Laboratorio de Comunicaciones Digitales con Python", 2020 ed., Departamento de Teoría de la Señal y Comunicaciones. Escuela Técnica Superior de Ingeniería. Universidad de Sevilla.

[2] «Anaconda,» [En línea]. Available: https://www.anaconda.com/.

[3] «Spyder IDE,» [En línea]. Available: https://www.spyder-ide.org/.

[4] «PyTorch,» [En línea]. Available: https://pytorch.org/.

[5] «NumPy,» [En línea]. Available: https://numpy.org/.

[6] «Matplotlib,» [En línea]. Available: https://matplotlib.org/.

[7] Ó. González Frenso, «Github,» Febrero 2022. [En línea]. Available: https://github.com/osgofre/MIMO-DL.

[8] «Additive White Gaussian Noise,» Wikipedia, 25 Octubre 2021. [En línea]. Available: https://en.wikipedia.org/wiki/Additive_white_Gaussian_noise. [Último acceso: 11 Enero 2022].

[9] A. Goldsmith, «MIMO detection algorithms,» de *Wireless Communications*, Stanford University, 2017.

[10] «Número de condición,» Wikipedia, 12 Enero 2022. [En línea]. Available: https://es.wikipedia.org/wiki/N%C3%BAmero_de_condici%C3%B3n. [Último acceso: 12 Enero 2022].

[11] J. J. M. Fuentes, «Unit 3: Notes on the LMMSE in MIMO,» de *Procesado Digital para Sistemas de Comunicaciones y Audiovisuales*, Departamento de Teoría de la Señal y Comunicaciones. Escuela Técnica Superior de Ingeniería. Universidad de Sevilla, 2021.

[12] «Google Colaboratory,» Google, LLC, [En línea]. Available: https://colab.research.google.com/notebooks/welcome.ipynb?hl=es.

[13] R. Raina, A. Madhavan y A. Y. Ng, Large-scale Deep Unsupervised Learning using Graphics Processors, Standford: ACM Press the 26th Annual International Conference - Montreal, Quebec, Canada (2009.06.14-2009.06.18), 2009.

[14] Udacity, «Intro to Deep Learning with PyTorch,» Facebook, 2021. [En línea]. Available:

https://www.udacity.com/course/deep-learning-pytorch--ud188.

[15] Y. LeCun, C. Cortes y C. J.C. Burges, «The MNIST Database,» [En línea]. Available: http://yann.lecun.com/exdb/mnist/.

[16] «Datasets & Dataloaders,» PyTorch, [En línea]. Available: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.

Índice de Códigos

Funciones propias

Se muestran a continuación todas las funciones creadas en el archivo *functions.py* [7].

```
'''
Conjunto de funciones desarrolladas para ser utilizadas en los programas
principales
'''
import numpy as np
import torch
import matplotlib.pyplot as plt

from labcomdig import gray2de
from network import *

'''
NUMPY
'''
def transmisorpamV2(Bn,Eb,M,p,L):
    """
    [Xn,Bn,An,phi,alfabetopam] = transmisorpamV2(Bn,Eb,M,p,L)

    Entradas:
     Bn = Secuencia de dí-gitos binarios
     Eb = Energía media por bit transmitida en Julios
     M  = Número de síímbolos del código PAM
     p  = Pulso paso de baja o paso de banda
     L  = Número de puntos a utilizar en la representación de un sí-
mbolo

    Devuelve:
     Xn = la señal de información (discreta)
     Bn = La secuencia de dí-gitos binarios realmente transmitidos
     An = La secuencia de niveles de amplitud transmitidos
     phi = Pulso básico real normalizado (energí-a unidad)
     alfabetopam = Los niveles de amplitud asociados a cada sí-mbolo
    """
    # Comprobación de parámetros de entrada
    p=np.reshape(p,[np.size(p)])  # <=   #p=p.squeeze()
    if len(Bn)<1 or Eb<=0 or M<2 or np.dot(p,p)==0 or L<1:
        raise Exception('Error: revise los parámetros de entrada')
    # Se obtienen en primer lugar los niveles asociado a cada sí-mbolo
¿Cuántos bits hay en cada sí-mbolo?
    k = int(np.ceil(np.log2(M)))
    M = 2**(k) # Se Ajusta M a una potencia de dos
```

```
    # El alfabeto [Ver la ecuación (4.21)]
    alfabetopam = np.sqrt(3*Eb*np.log2(M)/(M**2-1))*(2*(np.arange(M))-
M+1)

    # Si Bn no tiene una longitud múltiplo de k, se completa con ceros
    Nb = len(Bn)  # Número de bits a transmitir, actualizado
    Bn = Bn.squeeze().astype(int) #Comprobación de int y dimensiones
    Bn = np.r_[Bn,np.zeros(int(k*np.ceil(Nb/k)-Nb)).astype(int)] #
    Nb = len(Bn)  # Número de bits a transmitir tras la corrección
    Ns = Nb//k # Número de sí-mbolos a transmitir

    # La secuencia generada
    if M>2:
        An = alfabetopam[gray2de(np.reshape(Bn,[Ns,k]))]
    else:
        An = alfabetopam[Bn]

    # Comprobación de las longitudes y otros datos del pulso
suministrado para
    # hacer que el número de muestras del mismo sea efectivamente L
    Ls = len(p)
    if Ls<L:
        p = np.r_[p, np.zeros(L-Ls)]
    elif Ls>L:
        print('La duración del pulso se trunca a {}
muestras'.format(str(L)))
        p = p[:L] #Debe modificarse si se quiere un pulso de más de L
muestras
    # Se normaliza la energí-a del pulso para obtener la base del
sistema
    phi = p / np.sqrt(p@p)

    # Obtención del tren de pulsos, Xn = np.kron(An,phi) ó
    Xn = np.reshape(np.reshape(An,[Ns,1])*phi,[Ns*L,]) #Debe modificarse
si se quiere un pulso de más de L muestras
    return [Xn,Bn,An,phi,alfabetopam]

def de2Nary(d,n,N):
    """
    b = de2Nary(d,n,N)
    Convierte un número decimal, d, en un vector binario, b, de longitud
n
    con base N
    """
    c = np.zeros([len(d),int(n)])
    for i in range(int(n)): d, c[:,i] = np.divmod(d,N)
    c = np.fliplr(c);
    return c.astype(int).T


'''
TORCH
'''


def t_Qfunct(x):
    """
     y = Qfunct(x) evalúa la función Q en x.
    Donde y = 1/sqrt(2*pi) * integral desde x hasta inf de exp(-t^2/2)
dt
    """
    y=(1/2)*torch.special.erfc(x/(torch.tensor(2)**.5))
    return y
```

```
def t_gray2de(b):
    """
    Convierte cada columna de la matriz formada por dígitos binarios b
en un vector
        fila de los valores decimales correspondientes, aplicando
codificación de Gray.
    """
    if not torch.is_tensor(b):
        raise Exception('Error: la entrada no es un tensor')
    b = b.long() #Aseguro que sea tipo int
    c = torch.zeros_like(b); c[:,0] = b[:,0]
    for i in range(1,b.size(dim=1)):
        c[:,i] = torch.logical_xor(c[:,i-1], b[:,i])
    c = torch.fliplr(c) # Convierte los bits menos significativos en los
más significativos
    #Comprueba un caso especial.
    [n,m] = c.size()
    if torch.min(torch.tensor((m,n))) < 1:
        d = []
        return
    d = c @ 2**torch.arange(m)
    return d

def t_de2gray(d,n):
    """
    b = de2gray(d,n)
    Convierte un número decimal, d, en un vector binario, b, de longitud
n
    """
    c = torch.zeros(len(d),int(n))
    for i in range(int(n)):
        c[:,i] = torch.fmod(d,2) #resto
        d = torch.div(d,2,rounding_mode='floor') #cociente
    c = torch.fliplr(c); b = torch.zeros_like(c); b[:,0] = c[:,0]; aux =
b[:,0]
    for i in range(1,int(n)):
        b[:,i] = torch.logical_xor(aux, c[:,i])
        aux = torch.logical_xor(b[:,i], aux)
    return torch.reshape(b,[-1]).long()

def t_simbolobit(An,alfabeto):
    """
    Bn = simbolobit(An, alfabeto)
    An = secuencia de sí-mbolos pertenecientes al alfabeto
    alfabeto = tabla con los sí-mbolos utilizados en la transmisión
    Bn = una secuencia de bit, considerando que los sí-mbolos se habí-an
    generado siguiendo una codificación de Gray
    """

    k = torch.log2(torch.tensor(len(alfabeto))) # bits por sí-mbolo

    if k>1:
        distancia = abs(alfabeto[0]-alfabeto[1])
        indices   = torch.round((An-alfabeto[0])/distancia)
        #Bn        = torch.reshape(t_de2gray(indices,k),int(k*len(An)))
        Bn = t_de2gray(indices,k)
    else:
        Bn = ((An/max(alfabeto))+1)/2

    return Bn

def t_detectaSBF(r,alfabeto):
```

```
    """
    An = detectaSBF(r,alfabeto)
    r = secuencia a la entrada del detector, con estimaciones de s_i
    alfabeto = tabla con los niveles de amplitud/símbolos

    Genera:
    An = una secuencia de símbolos pertenecientes al alfabeto de acuerdo
con
    una regla de distancia euclidiana mínima (mínima distancia)
    """

    # Obtiene el índice respecto al alfabeto
    r_repeated = torch.repeat_interleave(r.reshape(r.size(dim=0),1),
alfabeto.size(dim=0),1)
    ind = torch.argmin(torch.abs(r_repeated - alfabeto), 1)

    # Genera la secuencia de niveles detectados
    An = alfabeto[ind]

    return An

def t_transmisorpamV2(Bn,Eb,M,p,L):
    """
    [Xn,Bn,An,phi,alfabetopam] = transmisorpamV2(Bn,Eb,M,p,L)

    Entradas:
     Bn = Secuencia de dí-gitos binarios
     Eb = Energía media por bit transmitida en Julios
     M  = Número de síímbolos del código PAM
     p  = Pulso paso de baja o paso de banda
     L  = Número de puntos a utilizar en la representación de un
sí-mbolo

    Devuelve:
     Xn = la señal de información (discreta)
     Bn = La secuencia de dí-gitos binarios realmente transmitidos
     An = La secuencia de niveles de amplitud transmitidos
       phi = Pulso básico real normalizado (energí-a unidad)
     alfabetopam = Los niveles de amplitud asociados a cada sí-mbolo
    """
    # Paso a tensores de los parámetros de entrada
    Eb = torch.tensor(Eb)
    M = torch.tensor(M)
    L = torch.tensor(L)

    # Comprobación de parámetros de entrada
    p=torch.reshape(p,p.size())  # <=   #p=p.squeeze()
    if Bn.size(dim=1)<1 or Eb<=0 or M<2 or torch.dot(p,p)==0 or L<1:
        raise Exception('Error: revise los parámetros de entrada')
    # Se obtienen en primer lugar los niveles asociado a cada sí-mbolo
¿Cuántos bits hay en cada sí-mbolo?
    k = torch.ceil(torch.log2(M))
    M = 2**(k) # Se Ajusta M a una potencia de dos

    # El alfabeto [Ver la ecuación (4.21)]
    alfabetopam = torch.sqrt(3*Eb*torch.log2(M)/(M**2-
1))*(2*(torch.arange(M))-M+1)

    # Si Bn no tiene una longitud múltiplo de k, se completa con ceros
    Nb = torch.tensor(Bn.size(dim=1))  # Número de bits a transmitir,
actualizado
    Bn = Bn.squeeze().long() #Comprobación de int y dimensiones
```

```
    Bn = torch.hstack((Bn,torch.zeros((k*torch.ceil(Nb/k)-Nb).long())))
#
    Nb = torch.tensor(Bn.size(dim=0))  # Número de bits a transmitir
tras la corrección
    Ns = torch.div(Nb,k,rounding_mode='trunc') # Número de sí-mbolos a
transmitir

    # La secuencia generada
    if M>2:
        An = alfabetopam[t_gray2de(torch.reshape(Bn,(int(Ns),int(k))))]
    else:
        An = alfabetopam[Bn.long()]

    # Comprobación de las longitudes y otros datos del pulso
suministrado para
    # hacer que el número de muestras del mismo sea efectivamente L
    Ls = p.size(dim=0)
    if Ls<L:
        p = torch.hstack(p,torch.zeros(L-Ls))
    elif Ls>L:
        print('La duración del pulso se trunca a {}
muestras'.format(str(L)))
        p = p[:L] #Debe modificarse si se quiere un pulso de más de L
muestras
    # Se normaliza la energí-a del pulso para obtener la base del
sistema
    phi = p / torch.sqrt(p@p)

    # Obtención del tren de pulsos, Xn = np.kron(An,phi) ó
    a = torch.reshape(An,(int(Ns),1))*phi
    Xn = torch.reshape(a,(int(Ns)*L,)) #Debe modificarse si se quiere un
pulso de más de L muestras
    return [Xn,Bn.long(),An,phi,alfabetopam]

def t_de2Nary(d,n,N):
    """
    b = de2Nary(d,n,N)
    Convierte un número decimal, d, en un vector binario, b, de longitud
n
    con base N
    """
    c = torch.zeros(len(d),int(n))
    for i in range(int(n)):
        c[:,i] = torch.fmod(d,N) #resto
        d = torch.div(d,N,rounding_mode='floor') #cociente
    c = torch.fliplr(c);
    return c.long().T

'''
NEURAL NETWORK
'''

def correction(Nb_train,Nb_test,k,M,n):
    """
    [Nb_train,Ns_train,Nb_test,Ns_test] =
correction(Nb_train,Nb_test,k,M,n)

    Cambia el valor de Nb_train y Nb_test si es necesario para
    que sea múltiplo de k y M**n
    """

    Ns_train = int(torch.floor(torch.tensor(Nb_train/(k*M**n)))) #Number
of symbols (multiple of k and M**n)
```

```
    Nb_train = int(Ns_train*(k*M**n)) #Number of bits on transmission
    Ns_train = int(torch.floor(torch.tensor(Nb_train/k))) #Number of
symbols (multiple of k)

    Ns_test = int(torch.floor(torch.tensor(Nb_test/k))) #Number of
symbols (multiple of k)
    Nb_test = int(Ns_test*k)          #Number of bits on transmission

    return [Nb_train,Ns_train,Nb_test,Ns_test]

def
create_datasets(H,Nb_train,Nb_test,Eb,M,n,SNRdb,batch_size,valid_size):
    """
    [trainloader,validloader,testloader,x,alphabet] =
create_datasets(H,Nb_train,Nb_test,Eb,M,n,SNRdb,batch_size,valid_size)

    Devuelve los conjuntos de datos listos para itererar, además del
conjunto
    de combinaciones posibles y el alfabeto de la constelación
    """
    # Data generation for training
    [Rn,Cn,Bn,x,alphabet] =
generate_data(H,Nb_train,Eb,M,n,SNRdb,train=True)
    trainset = SymbolsDataset(Rn,Cn,Bn)

    # Data generation for testing
    [Rn,Cn,Bn,x,alphabet] =
generate_data(H,Nb_test,Eb,M,n,SNRdb,train=False)
    testset = SymbolsDataset(Rn,Cn,Bn)

    # Indixes used for validation
    num_train = len(trainset)
    split = int(np.floor(valid_size * num_train))

    # Split data
    train_split, valid_split = SymbolsDataset.split_data(trainset,split)

    # Load training data in batches
    trainloader = torch.utils.data.DataLoader(train_split,
                                              batch_size=batch_size,
                                              num_workers=0)

    # Load validation data in batches
    validloader = torch.utils.data.DataLoader(valid_split,
                                              batch_size=batch_size,
                                              num_workers=0)

    # Load test data in batches
    testloader = torch.utils.data.DataLoader(testset,
                                             batch_size=batch_size,
                                             num_workers=0)

    return [trainloader,validloader,testloader,x,alphabet]

def generate_data(H,Nb,Eb,M,n,SNRdb,train):
    """
    [Xn/Rn,Bn,Cn,x,alphabet] = generate_data(H,Nb,Eb,M,n,SNRdb,test)

    Entradas:
     H     = Matriz del canal
     Nb    = Número de bits a transmitir
     Eb    = Energía media por bit transmitida en Julios
     M     = Número de símbolos del código PAM
```

```
      n     = Número de antenas del sistema MIMO nxn
      SNRdb = Signal to Noise Ratio
      test  = Booleano para determinar si son datos para entrenar o
testar

    Devuelve:
      Rn = La señal de información (discreta) recibida
      Bn = La secuencia de dí-gitos binarios realmente transmitidos
      Cn = La combinación correspondiente a cada valor de An
      x  = La matriz con las posibles combinaciones
      alphabet = Los niveles de amplitud asociados a cada sí-mbolo
    """

    k = int(torch.log2(torch.tensor(M)))      #Number of bits on each
symbol
    Ns = int(torch.floor(torch.tensor(Nb/k))) #Number of symbols
(multiple of k)

    alphabet = torch.sqrt(3*Eb*torch.log2(torch.tensor(M))/(M**2-
1))*(2*(torch.arange(M))-M+1)

    # possible combinations
    ind = t_de2Nary(torch.arange(M**n),n,M)
    x = alphabet[ind]

    if train:  # data for training. same number of symbols transmitted

        Xn = x.tile(int(Ns/x.size(1)))
        Xn = Xn[:,torch.randperm(Xn.size(1))] # Shuffle
        Bn = t_simbolobit(Xn.flatten(),alphabet) #Detected bits

        #Data reshape. Each row represents an antenna
        Bn = Bn.reshape(n,Nb)

        #Index of the combinations
        Cn = getidx(x,Xn,n)

    else:     # data for testing

        # fixed seed for same results always
        torch.manual_seed(1)

        bn = torch.randint(0,2,(1,Nb*n)) #Bits to transmit
        [Xn,Bn,An,phi,alphabet] =
t_transmisorpamV2(bn,Eb,M,torch.ones(1),1)

        #Data reshape. Each row represents an antenna
        Xn = Xn.reshape(n,Ns)
        Bn = Bn.reshape(n,Nb)
        An = An.reshape(n,Ns)

        #Index of the combinations
        Cn = getidx(x,An,n)

    SNR = 10**(SNRdb/10)   #Signal to Noise Ratio [n.u.]
    varzn = Eb/(2*SNR)          #Noise variance
    #if varzn <= 0.0025: varzn = torch.tensor(0.0025)
    Wn = torch.sqrt(varzn)*torch.randn(*Xn.shape)  #AWGN

    Rn = H@Xn + Wn

    return [Rn,Cn,Bn.long(),x,alphabet]
```

```
def getidx(x,An,n):
    """
    idx = getidx(x,An,n)

    Función que retorna el número de combinación correspondiente a cada
    valor de An

    x  = La matriz con las posibles combinaciones
    An = La secuencia de niveles de amplitud transmitidos
    n  = El número de antenas
    """

    idx = torch.empty(0)
    for col in An.T:
        i = (sum(x==col.view(n,1)) == n).nonzero(as_tuple=True)[0]
        idx = torch.hstack((idx,i))
    return idx.long()

def train_model(model, trainloader, validloader, optimizer, criterion,
patience, n_epochs):
    """
    [model, avg_train_losses, avg_valid_losses] =
create_datasets(H,Nb_train,Nb_test,Eb,M,n,SNRdb,batch_size,valid_size):

    Función que entrena a la red y retorna el modelo y los errores
cometidos

    """
    min_valid_loss = np.inf
    es_counter = 0

    # loss per batch
    train_losses = []
    valid_losses = []

    # loss per epoch
    avg_train_losses = []
    avg_valid_losses = []

    for epoch in range(1, n_epochs + 1):

        # Train the model
        model.train()
        for symbols, combs, bits in trainloader:
            # clear the gradients
            optimizer.zero_grad()
            # forward pass
            output = model(symbols)
            # calculate the loss
            loss = criterion(output, combs)
            # backward  pass
            loss.backward()
            # perform a single optimization step
            optimizer.step()
            # record training loss
            train_losses.append(loss.item())

        # Validate the model
        model.eval()
        for symbols, combs, bits in validloader:
            # forward pass
            output = model(symbols)
            # calculate the loss
```

```
            loss = criterion(output, combs)
            # record valid loss
            valid_losses.append(loss.item())

        # calculate average loss over an epoch
        train_loss = np.average(train_losses)
        valid_loss = np.average(valid_losses)
        avg_train_losses.append(train_loss)
        avg_valid_losses.append(valid_loss)

        if min_valid_loss > valid_loss:
            min_valid_loss = valid_loss
            # Saving State Dict
            torch.save(model.state_dict(), 'saved_model.pth')
        else:
            es_counter += 1
            if es_counter == patience: break

        # load saved model
        model.load_state_dict(torch.load('saved_model.pth'))

        # clear lists to track next epoch
        train_losses = []
        valid_losses = []

    return [model, avg_train_losses, avg_valid_losses]

def printloss(avg_train_losses,avg_valid_losses,SNR,save):
    """
    Función para mostrar por consola las pérdidas de entramiento y
validación
    en un epoch
    """

    fig = plt.figure(1,figsize=(10,8))
    plt.plot(range(1,len(avg_train_losses)+1),avg_train_losses,
label='Training Loss')
    plt.plot(range(1,len(avg_valid_losses)+1),
avg_valid_losses,label='Validation Loss')

    # find position of lowest validation loss
    minposs = avg_valid_losses.index(min(avg_valid_losses))+1
    plt.axvline(minposs, linestyle='--', color='r',label='Early Stopping
Checkpoint')

    plt.xlabel('epochs')
    plt.ylabel('loss')
    plt.ylim(0, 2) # VARIABLE scale
    plt.xlim(0, len(avg_valid_losses)+1) # consistent scale
    plt.grid(True)
    plt.legend()
    plt.title('SNR = {}dB'.format(SNR))
    plt.tight_layout()
    plt.show()
    if save: fig.savefig('loss_plot.png', bbox_inches='tight')
    pass

def eval_model(model, trainloader, validloader, testloader, x, alphabet,
k):
    """
    [berTrain, berValid, berTest] = eval_model(model, trainloader,
validloader, testloader, x, alphabet, k)
```

```
    Función que evalua el modelo y devuelve las distintas tasas de error
binarias
    """

    # BER for saved model with train data
    model.eval()

    berTrain = 0
    for symbols, combs, bits in trainloader:
        # forward pass
        output = model(symbols)
        berTrain += ber(output,x,alphabet,bits,k)
    berTrain /= len(trainloader)

    berValid = 0
    for symbols, combs, bits in validloader:
        # forward pass
        output = model(symbols)
        berValid += ber(output,x,alphabet,bits,k)
    berValid /= len(validloader)

    berTest = 0
    for symbols, combs, bits in testloader:
        # forward pass
        output = model(symbols)
        berTest += ber(output,x,alphabet,bits,k)
    berTest /= len(testloader)

    return [berTrain, berValid, berTest]

def ber(output,x,alphabet,bits,k):
    """
    ber = ber(output,x,alphabet,bits,k)
    Calcula la tasa de error de bit de una señal transmitida

    Entradas:
     output   = Salida de la Red Neuronal (bacth_size x n_combinations)
     x        = La matriz con las posibles combinaciones
     alphabet = Los niveles de amplitud asociados a cada sí-mbolo
     bits     = El conjunto de Bn transmitidos
     k        = Los bits que hay por símbolo
    """

    Nb = k*output.size(0) # Número de bits transmitidos (k*Ns)
    n = x.size(0)         # Núermo de antenas

    val, idx = torch.max(output,1) # El máximo de cada salida

    # Es necesario modificar la variable 'bits' de manera que estén
    # en la forma nxNb, correspondiendo cada fila a la transmisión de
una antena
    Bn = torch.empty(0,Nb)
    for i in range(int(n)): Bn =
torch.vstack((Bn,bits[:,i,:].flatten()))


    Andetected = x[:,idx]     # Detected symbols
    Bndetected =
t_simbolobit(Andetected.flatten(),alphabet).reshape(n,Nb) #Detected bits
    errorsBit = Nb-torch.sum(Bndetected==Bn,axis=1)

    return (errorsBit/Nb).reshape(n,1)
```

```
def detect(Rn,Bn,H,alphabet,Nb,Ns,n,method,**kwargs):
    """
    ber = detect(Rn,Bn,H,alphabet,Nb,Ns,n,method,**kwargs)

    Calcula la BER de los métodos ZF, LMMSE y ML
    """
    if method == 'ZF':
        Hinv = kwargs.get('Hinv',None)

        Z = Hinv@Rn

    elif method == 'LMMSE':
        Es = kwargs.get('Es',None)
        varzn = kwargs.get('varzn',None)

        Z = torch.linalg.inv(H.T@H + varzn/Es*torch.eye(n))@H.T@Rn

    elif method == 'ML':
        x = kwargs.get('x',None)

        hx = torch.unsqueeze(H@x,2) #H@x with shape (n,combinations,1)
        Z = x[:,torch.argmin(torch.linalg.norm(Rn.reshape(n,1,Ns)-
hx,dim=0),axis=0)]

    Andetected = t_detectaSBF(Z.flatten(),alphabet).reshape(n,Ns)
#Detected symbols
    Bndetected =
t_simbolobit(Andetected.flatten(),alphabet).reshape(n,Nb) #Detected bits
    errorsBit = Nb-torch.sum(Bndetected==Bn,axis=1) # N° Errors per bit


    return (errorsBit/Nb).reshape(n,1)
```