Universidad Zaragoza
1542

Trabajo Fin de Grado

Predicción de movilidad de usuarios en redes móviles mediante redes neuronales

Prediction of user mobility in mobile networks using neural networks

Autor

Fernando Simón Micheto

Director

José Ramón Gállego Martínez

Titulación del autor

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Escuela de Ingeniería y Arquitectura

2020

RESUMEN

En esta memoria se presenta una solución para predecir la movilidad de los usuarios dentro de una red celular móvil usando redes neuronales recurrentes (RNN).

La predicción de movilidad, es decir, saber cuál o cuáles van a ser las siguientes estaciones base por las que se van a desplazar los usuarios, puede mejorar la gestión de los recursos de la red en un futuro: por ejemplo, poder prever las zonas de congestión de la red debido a una concentración de usuarios en la zona que da servicio una estación base, mejorar los traspasos de llamadas entre estaciones base...

Se trata de un tema de investigación en el que se trabaja desde hace años, con diferentes aproximaciones, tales como modelos de Markov, modelos ocultos de Markov o redes neuronales. Este último campo es el que más interés despierta en la actualidad. Más concretamente, las redes neuronales recurrentes, por su capacidad para utilizar información temporal. Ya hay estudios en esa línea, sin embargo, este proyecto se va a centrar en desarrollar una solución propia para esta tecnología, añadiendo todas las mejoras posibles con el fin de mejorar la eficiencia en las predicciones.

Para poder validar esta solución, se utilizarán datos de movilidad reales cedidos por la empresa Teltronic. Estos datos serán procesados para poder entrenar la red neuronal. La mayor parte de este trabajo se centra en caracterizar el modelo adecuado para los datos. Para finalizar este proyecto y comprobar su validez, se van a comparar los resultados conseguidos por la red neuronal desarrollada en este trabajo, con los obtenidos con un modelo de Markov, que es uno de los métodos más utilizados para este tipo de estudios debido a su sencillez.

El objetivo inicial ha sido predecir a que estación base se moverá el usuario a partir de los movimientos que han realizado todos los usuarios de la red. Posteriormente, se ha incluido en esta prueba la información del usuario individual en el entrenamiento. Igualmente, a partir de la red neuronal básica desarrollada inicialmente, se ha realizado una implementación para la predicción de las dos siguientes estaciones base a las que se moverá el usuario, variando desde las dos hasta las seis siguientes. Se ha comprobado que los mejores resultados se obtienen cuando se predicen las tres o cuatro estaciones base siguientes.

Por último, una vez se han obtenido los porcentajes de acierto de todas las implementaciones anteriores, se han comparado con los obtenidos con un modelo de Markov, desarrollado también en este proyecto. Se ha comprobado que la mejora en las prestaciones obtenida con las redes neuronales es destacable, llegando a alcanzar un 83% de éxito en la predicción de la siguiente estación base, cuando con el modelo de Markov sólo se ha alcanzado un 56%.

AGRADECIMIENTOS

Quiero agradecerle a mi familia, y en especial a mis padres, todo el apoyo que me han dado ya no solo durante estos meses de trabajo, sino durante los últimos cuatro años.

También me gustaría dar las gracias a mis amigos, que siempre están ahí cuando lo he necesitado y siempre han confiado en mí. Además de ellos, a todas las personas que he conocido en estos cuatro duros años y con los que he compartido muchas horas de estudio.

Para terminar, quiero agradecer a los profesores que me han ayudado a formarme durante estos años. En especial, quiero agradecer a José Ramón todo lo que me ha ayudado en estos meses a pesar de la situación tan difícil que hemos tenido que vivir.

ÍNDICE DE CONTENIDOS

1. Introducción

Este capítulo explica los orígenes de las redes neuronales y la movilidad, la motivación y objetivos del proyecto y las herramientas que se utilizan para su desarrollo.

1.1 Antecedentes

Aunque las redes neuronales se pueden referir a un concepto muy relacionado con la actualidad, su historia se remonta a 1936. Alan Turing fue el primero en estudiar el cerebro como una forma de ver el mundo de la computación. Sin embargo, los primeros teóricos que concibieron los fundamentos de la computación neuronal fueron Warren McCulloch, un neurofisiólogo, y Walter Pitts, un matemático, quienes, en 1943, lanzaron una teoría acerca de la forma de trabajar de las neuronas [1]. Ellos modelaron una red neuronal simple mediante circuitos eléctricos. [2].

En la actualidad, las redes neuronales han tenido un importante resurgimiento. Una de las razones de este resurgimiento es que las redes neuronales son computacionalmente, costosas. Solo en los últimos años las computadoras son suficientemente rápidas para realmente operar redes neuronales de gran escala, haciendo que actualmente sean la técnica más vanguardista para muchas aplicaciones [3].

En lo que respecta a su aplicación en la predicción de movilidad de los usuarios es algo que se lleva estudiando desde los años 90 [4], si bien el problema de la predicción de movilidad ya se comenzó a estudiar en los años 50 [5]. Estos primeros estudios de movilidad aparecieron como consecuencia de la necesidad de regular el tráfico en las ciudades, entonces pudieron constatar una relación directa entre el número de viajes producidos entre dos zonas, el número de empleos en una de ellas y el número de residentes en la otra [6].

1.2 Motivación y objetivos del proyecto

La habilidad para predecir la próxima celda a la que irá un usuario o incluso la ruta que este realizará es un aspecto importante en las redes móviles de comunicaciones. Ejemplo de ello son aplicaciones orientadas a la gestión del traspaso entre celdas (*handover*), la gestión de recursos radio en la red y aplicaciones basadas en la ubicación [7]. Los principales objetivos de este trabajo son los mencionados a continuación:

- Obtener el mejor modelo de red neuronal posible en función de los datos de partida con el fin de poder predecir con mayor precisión el movimiento de los usuarios.
- Comparar los resultados obtenidos mediante la red neuronal recurrente, frente a los obtenidos usando un modelo de Markov para validar la utilidad de la solución propuesta.

1.3 Herramientas utilizadas

La herramienta principal que se usará para la predicción de movilidad es un algoritmo basado en redes neuronales recurrentes con memoria a corto y largo plazo. Las redes neuronales se inspiran en redes neuronales biológicas (el sistema nervioso central de los animales, en particular el cerebro) y son utilizadas para estimar o aproximar funciones con una gran cantidad de datos de entrada.

Para el procesado de los datos y la programación de la red neuronal se va a utilizar Google Colaboratory [8]. Google Colaboratory es un entorno gratuito de Jupyter Notebook [9] que no requiere configuración y que se ejecuta completamente en la nube. Jupyter es un entorno interactivo que permite desarrollar código Python [10] de manera dinámica. Jupyter se ejecuta en local como una aplicación cliente-servidor y posibilita tanto la ejecución de código como la escritura de texto, favoreciendo así la interactividad del entorno y que se pueda entender el código como la lectura de un documento.

Los datos serán exportados a Google Colaboratory usando el método "upload" de la biblioteca files y este devolverá un diccionario de los archivos que se han subido.

En lo que respecta al procesado de datos, se utilizará la biblioteca *pandas*. Esta biblioteca de software es una extensión de Numpy que ofrece estructuras de datos y operaciones para manipular tablas numéricas.

A continuación, en lo que respecta a la programación de la red neuronal, se utilizará la librería PyTorch [11]. PyTorch es una librería muy reciente y pese a ello, dispone de una gran cantidad de manuales y tutoriales donde encontrar ejemplos. PyTorch permite la creación de redes neuronales de manera sencilla. Al contrario que otras librerías como Tensorflow, PyTorch trabaja con grafos dinámicos en vez de estáticos. Esto significa que en tiempo de ejecución se pueden ir modificando las funciones y el cálculo del gradiente variará con ellas. Además, PyTorch dispone de soporte para su ejecución en tarjetas gráficas (GPU), utiliza internamente CUDA, una API que conecta la CPU con la GPU que ha sido desarrollado por NVIDIA.

De igual manera, se utiliza como herramienta los modelos de Markov, que se han implementado en este proyecto utilizando Google Colaboratory, al igual que la red neuronal. Los modelos de Markov son una solución sencilla y ampliamente utilizada para predicción de movilidad [12,13], y los resultados obtenidos serán comparados con los de la red neuronal. Se demostrará que los resultados usando la red neuronal serán mejores que con el modelo de Markov.

1.4 Organización de la memoria

Por último, en lo que respecta a la estructura de esta memoria, estará dividida en varios capítulos. En este primer capítulo se han tratado los objetivos y la motivación del proyecto, además de las herramientas que se han utilizado. En el segundo se explica de manera teórica el concepto de red neuronal, la red neuronal recurrente y los modelos de Markov. En el tercero se presentarán los datos que se van a utilizar, la realización del trabajo y los distintos pasos que se han seguido para conformar el modelo adecuado. En el cuarto se desarrollarán mediante implementaciones propias, las pruebas realizadas con la red neuronal y los modelos de Markov, junto con los resultados obtenidos. Y en el quinto y último se tratarán las conclusiones que se pueden sacar de este proyecto y posibles futuras líneas de trabajo.

2. Fundamentos teóricos

El objetivo de este capítulo es explicar las distintas soluciones utilizadas en el desarrollo de este proyecto.

2.1 Redes neuronales

2.1.1 Definición

Las redes neuronales son un algoritmo que se vienen utilizando desde hace unos años, como ya se ha comentado en la introducción, que fue originalmente desarrollado con el objetivo de fabricar máquinas que pudieran simular el cerebro humano. Existen numerosas formas de definir a las redes neuronales, estos son algunos ejemplos descritos de forma resumida:

> Una forma de computación inspirada en modelos biológicos.
>
> Un modelo matemático compuesto por un gran número de elementos procesales organizados en niveles.
>
> Un sistema de computación compuesto por un gran número de elementos simples, elementos de procesos muy interconectados, los cuales procesan información por medio de su estado dinámico como respuesta a entradas externas.
>
> Redes interconectadas masivamente en paralelo de elementos simples (usualmente adaptativos) y con organización jerárquica, las cuales intentan interactuar con los objetos del mundo real del mismo modo que lo hace el sistema nervioso biológico.

2.1.2 Elementos básicos de una red neuronal

Una red neuronal está formada por neuronas interconectadas y organizadas en tres capas como mínimo. Una neurona es cada uno de los nodos de la red y están repartidas por todas las capas de la red. Cada neurona tiene diferente peso (W) y un parámetro de sesgo (b) que es igual entre las neuronas de una misma capa. Los datos entran a ella por la capa de entrada, pasan por la capa oculta, la cual puede estar formada por varias capas y salen por la capa de salida, figura 2.1.

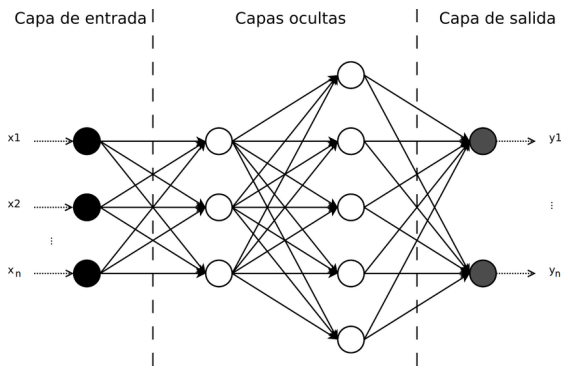

Figura 2.1. Ejemplo de una red neuronal totalmente conectada

El conjunto de datos que se utiliza estará dividido en tres tipos, el primero de ellos son los datos de entrenamiento, que es el que se usa para que la red neuronal aprenda patrones sobre los datos. El segundo es el de validación, que nos indicará como de eficiente es el modelo de red neuronal que estamos entrenando. Y el tercero y último es el de pruebas, que es el encargado de mostrar los resultados obtenidos después de haber entrenado la red.

Inicialmente, en la capa de entrada estarán todas nuestras entradas o datos de entrenamiento, estos se multiplican por los pesos que permitirán expresar su importancia.

Como ya se ha comentado, la capa oculta puede estar formada a su vez por una o varias capas. El número de capas ocultas dependerá de cuanto de sofisticado queremos que sea nuestro modelo. Sin embargo, cuanto mayor sea el número de capas ocultas, necesitaremos más recursos como el tiempo o potencia computacional.

La capa de salida contará con tantas neuronas como características se quieran encontrar.

2.1.3 Aprendizaje adaptativo y autoorganización

La capacidad de aprendizaje adaptativo es una de las características más atractivas de las redes neuronales. Esto significa que aprenden a llevar a cabo ciertas tareas mediante un entrenamiento con una porción de los datos de partida. Una red neuronal no necesita un algoritmo para resolver un problema, ya que ella puede generar su propia distribución de pesos en los enlaces mediante el aprendizaje. La labor del desarrollador es la de obtener y diseñar la arquitectura apropiada, en función de los datos de partida.

Las redes neuronales emplean su capacidad de aprendizaje adaptativo para autoorganizar la información que reciben durante el aprendizaje y/o la operación. La autoorganización consiste en la modificación de la red neuronal completa para llevar a cabo un objetivo específico. Esta autoorganización provoca la generalización; es decir, la facultad de las redes neuronales de responder apropiadamente cuando se les presentan datos o situaciones a las que no habían sido expuestas anteriormente.

Las redes neuronales tienen cierta tolerancia a los fallos, esto es debido a que tienen su información distribuida en las conexiones entre neuronas, existiendo cierto grado de redundancia en este tipo de almacenamiento [2].

2.1.4 *Forwardpropagation* y *backpropagation*

La manera en la cual las redes neuronales crean las predicciones se denomina *forwardpropagation* o propagación hacia delante. En un principio la red neuronal tiene valores de peso y sesgo aleatorios en cada neurona, como ya se ha comentado anteriormente. Los datos de entrenamiento atraviesan la red hasta llegar a la capa de salida. En esta capa, la red neuronal predice el tipo de dato al que pertenecen los datos de entrenamiento, estas predicciones las usa la función de pérdidas, explicada en el apartado 2.1.6, para medir la eficiencia de la red neuronal. Este ciclo se repite tantas veces como *epochs* o ciclos hayamos indicado. En cada ciclo se ejecuta el algoritmo de *backpropagation* o retro propagación para actualizar los valores de peso y sesgo. Además, cada iteración tiene un número de datos en el que dividimos los datos para que funcione de manera más eficiente la red, denominado *batch size*.

Generalmente se conoce el algoritmo de *backpropagation* como el encargado de optimizar la función de pérdidas para mejorar las predicciones de una red neuronal. Este algoritmo se encarga de calcular las derivadas (o gradientes) de los parámetros de peso y sesgo, para saber cómo estos parámetros afectan al resultado de la función de pérdidas. Esta definición puede ser usada para explicar el algoritmo que nos ocupa. No obstante, y para ser más precisos, la optimización de una red neuronal se divide en dos partes:

- La primera es el algoritmo de *backpropagation*, como ya se ha explicado, este algoritmo se encarga de ver como los valores de peso y de sesgo afectan al resultado de la función de pérdidas.
- La segunda parte es el algoritmo de optimización, encargado de optimizar la red neuronal y cambiar los valores de peso y de sesgo conforme pasan los ciclos (o *epochs*).

Existen diferentes algoritmos de optimización, su elección depende del tipo de problema que se esté resolviendo. Este algoritmo, llamado *optimizer*, es un parámetro de la red neuronal. Todos estos algoritmos tienen un comportamiento parecido, pero la idea general es encontrar el *global minimum* o el mínimo global de la función de pérdidas. El *optimizer* converge la función hasta llegar al punto donde la función se encuentra optimizada, esto se logra con ayuda de las derivadas que indican que camino se tiene que seguir [14].

2.1.5 Optimizador Adam

De los diferentes algoritmos de optimización existentes, en este proyecto se utilizará el algoritmo Adam [15], que, por otra parte, es el optimizador más utilizado en problemas de *machine learning*. El método es sencillo de implementar, es computacionalmente eficiente, tiene pocos requisitos de memoria, y es adecuado para problemas que trabajan con muchos datos y/o parámetros.

2.1.6 Función de pérdidas

La función de pérdidas, también conocida como función de coste, es la función que nos muestra la eficiencia o validación de la red neuronal, un resultado alto indica que la red neuronal tiene un desempeño pobre y un resultado bajo indica que la red neuronal está haciendo un buen trabajo. Esta es la función que optimizamos o minimizamos cuando realizamos el *backpropagation*. La función de pérdidas que se usará en este proyecto será la función de entropía cruzada. Esta es su ecuación para dos distribuciones de probabilidad p y q:

$$J(p,q) = -\sum_{i} log\,(q_i)\,p_i$$

2.1.7 Función de activación

La función de activación calcula el estado de actividad de una neurona; transformando la entrada global en un valor (estado) de activación, cuyo rango normalmente va de (0 a 1) o de (–1 a 1). Esto es así, porque una neurona puede estar totalmente inactiva (0 o –1) o activa (1). Existen distintos tipos de funciones de activación [16]:

- Sigmoide
- Tangente hiperbólica
- Unidad lineal rectificada (ReLu)
- Unidad lineal rectificado agujereada
- Softmax

La función de activación de la neurona se describe mediante la ecuación en el instante *t* + 1:

$$z_j\,[t+1] = g(\sum_{i=1}^{n} W_{ji}\, z_i[t] + \, W_j)$$

donde:

- z_j [t], j=1,…n, son un conjunto de n señales de entrada que se suministran a la neurona. Estos datos pueden ser externos a la red, pertenecientes a la salida de otras neuronas de la red o, correspondientes a la salida anterior de la propia neurona.
- W_{ji}, i=1,…n, es el peso asociado a la sinapsis que conecta la unidad i-ésima con la neurona j-ésima.
- W_j es un sesgo que aumenta la capacidad de procesamiento de la neurona y eleva o reduce la entrada a esta, según sea su valor positivo negativo.
- Un sumador o integrador que se encarga de sumar las señales de entrada, ponderas con sus respectivos peso y sesgo.
- Una función g de activación que suele limitar la amplitud de la salida de la neurona

El modelo de red neuronal recurrente que se utiliza en este proyecto es el conocido con el acrónimo LSTM (*long-short term memory*), que utiliza como función de activación la sigmoide, como se comentará en el apartado 2.2.1.

2.2 Redes neuronales recurrentes

Las redes neuronales recurrentes tienen caminos de retroalimentación entre todos los elementos que las conforman. Una sola neurona está entonces conectada a las neuronas posteriores en la siguiente capa, las neuronas pasadas de la capa anterior y a ella misma a través de vectores de pesos variables que sufren alteraciones en cada *epoch* con el fin de alcanzar los parámetros o metas de operación.

Para explicarlo, se adjunta la figura 2.2 donde se muestra una estructura simple de una red neuronal recurrente. Una característica importante es la inclusión de *delays* (z^{-1}) a la salida de las neuronas en las capas intermedias; es decir, las salidas parciales $S_{mn}(t + 1)$ se convierten en valores $S_{mn}(t)$, un instante de tiempo anterior, y así se retroalimenta a

todos los componentes de la red, guardando la información de los instantes de tiempo anteriores. Puede observarse cómo todos los nodos están interconectados entre sí y también con los nodos anteriores a ellos a través de conexiones directas y también con *delays* antes de cada capa, o memorias temporales. El diagrama se muestra simplificado para no incurrir en excesiva complejidad, pero cada una de las capas está representada por cierto número de neuronas [17].

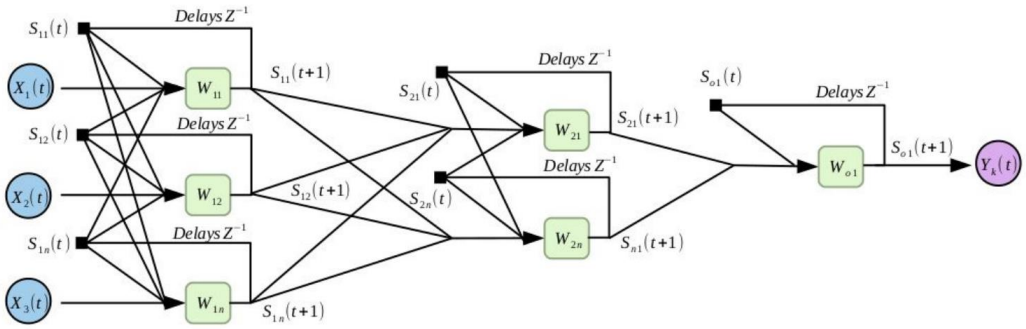

Figura 2.2. Estructura red neuronal recurrente

Las redes neuronales recurrentes son especialmente útiles en aplicaciones tales como el reconocimiento de patrones secuenciales, cambiantes en el tiempo, ya que las capacidades de predicción y mapeo de las redes neuronales recurrentes así lo permiten. Por este motivo resultan de gran interés para la predicción de movilidad tal y como se busca en este trabajo.

2.2.1 Red de memoria a corto y largo plazo (LSTM)

De entre los modelos de redes neuronales recurrentes, se va a usar el modelo LSTM [17] (*long-short term memory*). El modelo LSTM como ya hemos comentado se caracteriza porque la red dispone de memoria.

En las redes neuronales recurrentes tradicionales, durante la fase de retro propagación, la señal de gradiente puede llegar a ser multiplicada un gran número de veces por la matriz de peso asociada a las neuronas de la capa oculta. Esto puede tener un gran efecto en el proceso de aprendizaje de la red puesto que la magnitud de los pesos de la matriz se ve afectada. Si los pesos en esta matriz son pequeños (menores que 1), puede conducir a una situación conocida como gradiente evanescente, donde la señal gradiente llega a ser tan pequeña que el aprendizaje se vuelve demasiado lento o deja de funcionar por completo. Si por el contrario los pesos de la matriz son grandes (mayores que 1), puede conducir a una situación en la que la señal de gradiente es tan grande que puede causar que el aprendizaje nunca llegue a converger. A este aumento de la señal de gradiente se le conoce también como explosión de gradientes. Estas cuestiones son las que motivaron el desarrollo del modelo LSTM, que introduce una nueva estructura (figura 2.3 y 2.4) llamada celda de memoria para corregir estos problemas.

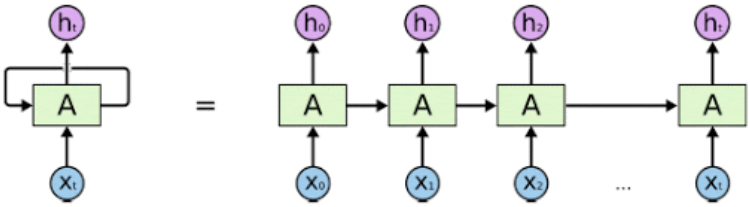

Figura 2.3. Estructura de una red LSTM

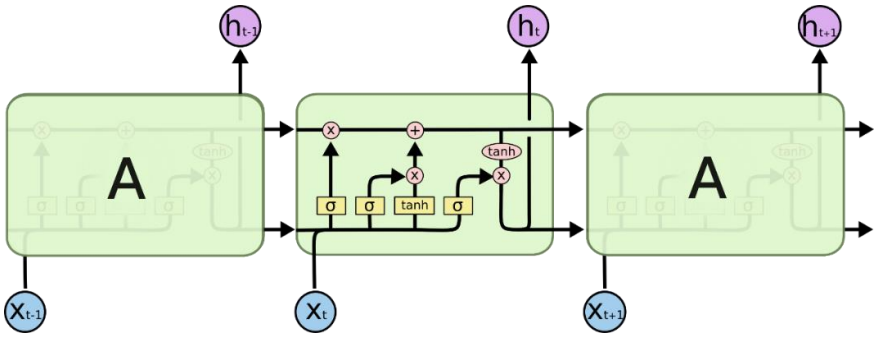

Figura 2.4. Representación de una célula LSTM

El modelo LSTM tiene la capacidad de eliminar o añadir información a la memoria. Estas operaciones están cuidadosamente reguladas por estructuras internas llamadas puertas (figura 2.5). Estas puertas, permiten añadir o eliminar información a la memoria en un momento dado del proceso de entrenamiento. Se componen de una red neuronal (normalmente con sigmoide como función de activación) de una sola capa y una operación aritmética (multiplicación o suma).

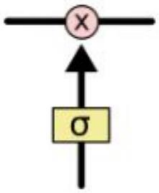

Figura 2.5. Puerta

Las salidas de la red con sigmoide serán valores entre cero y uno, lo que permite decidir la cantidad de información que la puerta debe dejar pasar. Cero significaría "no dejar pasar nada", mientras que uno sería "dejar pasar todo". La LSTM tiene tres de estas puertas, para proteger y controlar el estado de la memoria.

A continuación, se explica paso a paso el funcionamiento interno y el recorrido de la información en una red LSTM:

1. La célula LSTM decide qué información va a desechar de la memoria. Esta decisión es tomada por la puerta llamada *Forget gate*. Como se puede apreciar en la figura 2.6, h_{t-1} y xt se concatenan, y el resultado será la entrada a la pequeña red que conforma la *Forget gate*. Como se ha explicado anteriormente, el resultado de la red servirá para decidir si el estado de la memoria permanece intacto o se altera eliminando algún elemento.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

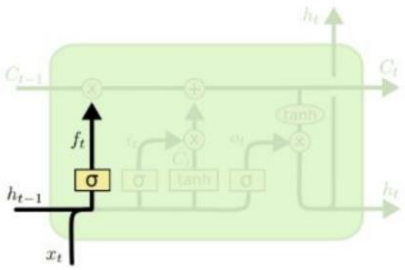

Figura 2.6. *Forget gate*

2. El siguiente paso que realiza la LSTM es decidir qué información nueva se va a almacenar en el estado de memoria. Esto se realiza en tres etapas:
    a. En primer lugar, una puerta llamada *Input Gate* (figura 2.7) decide qué valores vamos a actualizar.
    b. A continuación, una pequeña red con ayuda de la función $tanh$ crea un vector de valores de nuevos candidatos, $\hat{C}_t$, que podrían añadirse al estado.
    c. En el último paso, se combinan los dos resultados anteriores para crear una actualización de estado.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$

$$\hat{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_c)$$

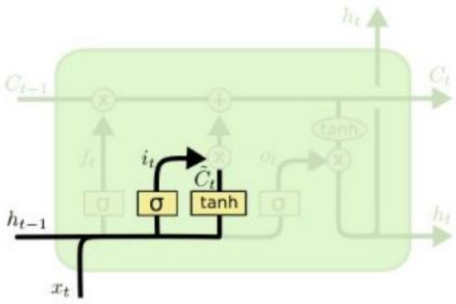

Figura 2.7. *Input gate*

3. En este momento ya podemos actualizar el estado de la memoria (figura 2.8) cambiando C_{t-1} por el nuevo estado C_t. Como en los pasos anteriores ya se ha decidido qué elementos de la memoria desechar y la información nueva a almacenar, en este paso sólo tenemos que aplicar las operaciones de cada puerta. Es decir, multiplicar el estado anterior (C_{t-1}) por f_t, descartando una cierta cantidad de información en función de f_t. A continuación, sumamos $i_t * \hat{C}_t$ a la memoria, actualizando la misma con nuevos valores que nos podrían servir en un futuro.

$$C_t = f_t * C_{t-1} + i_t * \hat{C}_t$$

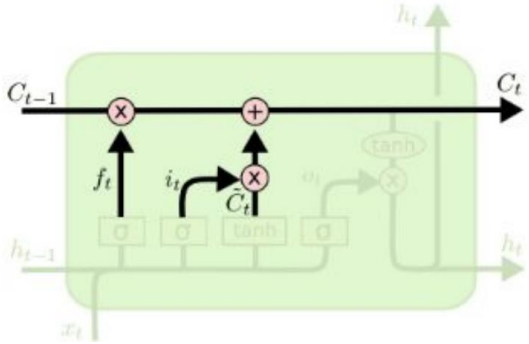

Figura 2.8. Actualización de la memoria

4. En el último paso, se tiene que decidir cuál será la salida de la célula LSTM (figura 2.9). Esta se obtendrá mediante el producto de dos elementos: el primero de ellos será la salida de la red con sigmoide, que servirá para decidir qué elementos de la memoria se combinarán; y el segundo elemento será el filtrado de datos desde la memoria por una $tanh$ (para tener valores entre -1 y 1). Finalmente, estos dos elementos se multiplicarán, dando como resultado la nueva salida de la célula (h_t).

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t * tanh(C_t)$$

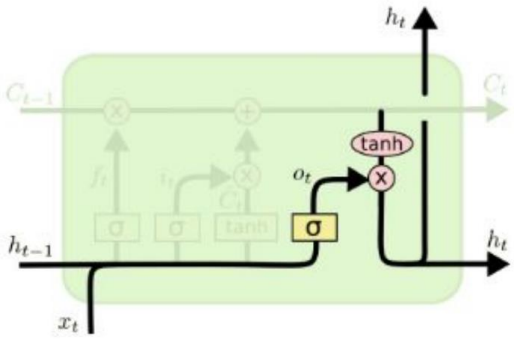

Figura 2.9. Salida

2.3 Modelos de Markov

Para comprender que son los modelos de Markov [18], primeramente, se va a explicar que es un proceso estocástico. Una sucesión de observaciones X_1, X_2… es un proceso estocástico si los valores de estas observaciones no se pueden predecir exactamente, pero sí se pueden especificar las probabilidades para los distintos valores posibles en cualquier instante de tiempo.

Donde X_1 es la variable aleatoria que define el estado inicial del proceso y X_n es la variable aleatoria que define el estado del proceso en el instante de tiempo n. Para cada posible valor del estado inicial s_1 y para cada uno de los sucesivos valores s_n de los estados X_n, donde n= 2, 3, … especificamos la probabilidad P:

$$P\,(X_{n+1} = s_{n+1}\,|X_1 = s_1, X_2 = s_2, \dots, X_n = s_n)$$

Una vez explicado, se define una cadena de Markov como un proceso estocástico en el que el estado actual X_n y los estados previos X_1,…, X_{n-1} son conocidos. Donde la probabilidad del estado futuro X_{n-1} no depende de los estados anteriores X_1,…, X_{n-1}, y solamente depende del estado actual X_n. Es decir, para n = 1, 2, … y cualquier sucesión de estados s_1,…, s_{n+1}, definimos la probabilidad P, como:

$$P\,(X_{n+1} = s_{n+1}\,|X_1 = s_1, X_2 = s_2, \dots, X_n = s_n) = P\,(X_{n+1} = s_{n+1}\,|X_n = s_n)$$

Mediante la figura 2.10 se va a explicar un ejemplo de modelo de Markov para predecir el movimiento en un escenario simplificado con 3 estaciones base. Definimos cada una de las estaciones base como un estado del modelo, y de estas saldrán tres probabilidades de transición distintas. Por ejemplo, desde la estación base 1 sale la probabilidad P12 y la P13. La primera de ellas se refiere a la probabilidad de que el usuario se traslade desde la estación base 1 a la 2 y la segunda la probabilidad de movimiento desde la primera a la tercera estación. Por tanto, si se quiere predecir el movimiento que hará un usuario desde la estación base 1, el modelo resolverá que la siguiente estación base a la que se va a desplazar el usuario es la que tenga mayor probabilidad de movimiento.

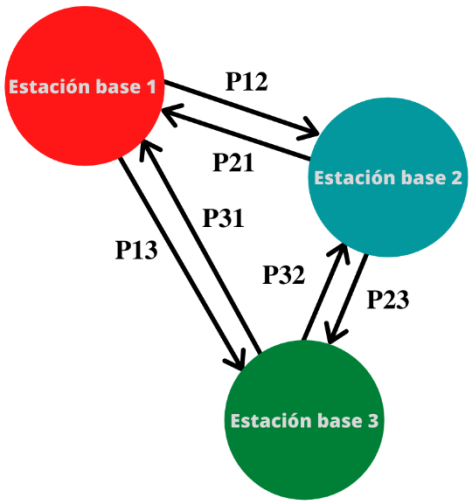

Figura 2.10. Ejemplo de modelo de Markov sin memoria

Otra aproximación que se utiliza habitualmente en la predicción de movilidad es la introducción de memoria en el modelo de Markov (similar a lo que se va a hacer con la red neuronal). Una manera sencilla de conseguir esto es definir el estado no como la estación base actual, sino como la secuencia de las últimas estaciones base que el móvil ha visitado. Un ejemplo esquemático con una memoria de dos estaciones base (actual y anterior) se muestra en la figura 2.11. En este caso, cada uno de los estados viene representado por dos números: el primero representa la estación base de la que viene el usuario, y el segundo representa la estación en la que se encuentra el usuario actualmente. Por ejemplo, en el estado 1-2, el usuario viene de la estación base 1 y está en la 2. Puede incrementarse la memoria del modelo, si bien aparecen dos problemas: el número de estados se incrementa rápidamente, con el consumo de memoria asociado, y se incrementa el número de muestras necesario para entrenarlo correctamente. En diferentes trabajos se muestra que un valor de dos estaciones base por estado [12,13] suele proporcionar las mejores prestaciones. En este trabajo se va a evaluar también el modelo de tres estaciones base por estado.

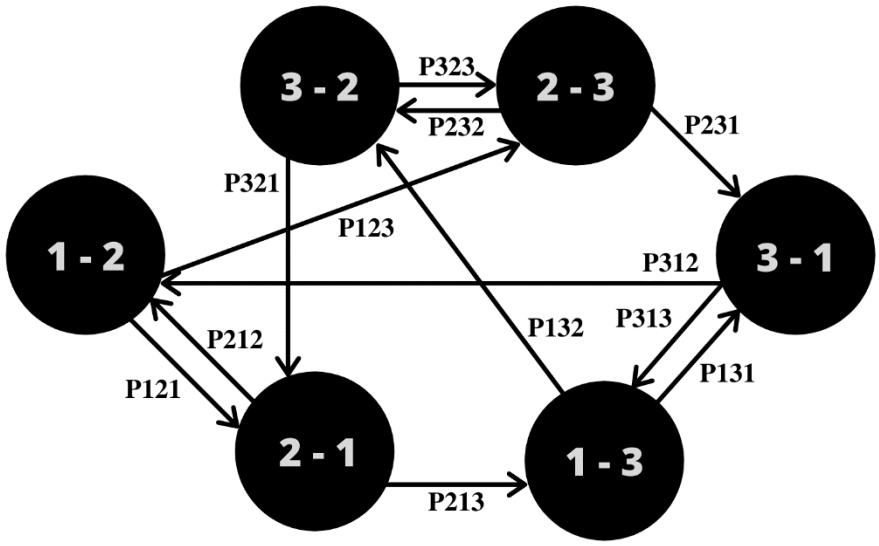

Figura 2.11. Ejemplo de modelo de Markov con memoria igual a uno para tres estaciones

Por último, y al igual que con las redes neuronales tal y como se explica en el apartado 2.1.6, se debe validar la eficiencia del modelo de Markov. Para ello, se selecciona una parte de los datos (el conjunto de validación) y se suman todas las probabilidades de transición de estado para esos datos. El modelo de Markov será mejor cuanto mayor sea dicha suma.

3. Implementación

Este capítulo se ha estructurado en los siguientes apartados para el desarrollo de la solución:

- Explicación de los datos de partida y su tratamiento.
- Descripción del algoritmo básico de la red.
- Modificaciones que se han realizado sobre ese algoritmo básico (inclusión de información de usuario en el entrenamiento y predicción de no solo la siguiente estación base si no de las siguientes estaciones).
- Desarrollo del código para los modelos de Markov.

Se añade la figura 3.1 donde se resume gráficamente el algoritmo básico de funcionamiento de la red y de los modelos de Markov:

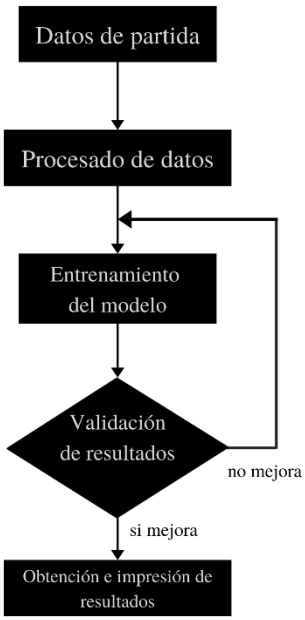

Figura 3.1. Diagrama básico de la implementación de los modelos

3.1 Datos de partida y procesado de estos

Los datos de partida están recogidos en un archivo Excel con 500.000 filas y 22 columnas, facilitados por la empresa Teltronic. Cada una de las filas se corresponde con una traza registrada dentro de la red correspondiente a algún tipo de actividad realizada por alguno de los usuarios de la misma. Se trata de una red de comunicaciones profesional.

De los datos proporcionados en las columnas, se utilizarán en este proyecto los siguientes:

- *Service type:* como su propio nombre indica, nos indica el tipo de servicio al que se corresponde la traza. Aunque aparecen distintos tipos de servicio nos centraremos en el de registro, que es el que nos indica el cambio de posición del usuario o que este ha encendido su móvil después de un "desregistro"; en concreto, cada vez que un usuario cambia de estación base se registra en la misma

y eso se recoge en la traza de registro. El "desregistro" nos indica que ese usuario no va a cambiar de estación base hasta que no vuelva a registrarse en ella.

- *Date/Time*: nos muestra el día y la hora con el siguiente formato: MM-DD-AAAA HH:MM:SS AM/PM.
- *Source:* es el número que identifica a cada uno de los usuarios de la red.
- *Source location:* es el identificador y la dirección IP de la estación base desde la que se registró esa traza.

Para poder procesar los datos, primero se debe subir el archivo y leerlo con Google Colaboratory, para ello utilizaremos las instrucciones del lenguaje Python mostradas en la figura 3.2:

Figura 3.2. Subir y leer archivo

Además, se ha añadido al final la instrucción xls.head() con la que se imprimen las 5 primeras líneas del Excel y podemos comprobar si ha sido leído correctamente.

A continuación, se va a trabajar con los datos para obtener un array en el que cada uno de los elementos se corresponda con los movimientos asociados a un usuario y ordenados por fecha y hora. Para ello se utilizará la librería *pandas*, especializada para el análisis de datos. La estructura de datos básica de pandas se denomina *dataframe*, que es una colección ordenada de columnas con nombres y tipos, es similar a una tabla de base de datos, donde una sola fila representa un único caso y las columnas representan atributos particulares.

Primero se obtienen los dos *dataframes* asociados a los registros y "desregistros", los concatenamos y ordenamos por fecha y hora (figura 3.3).

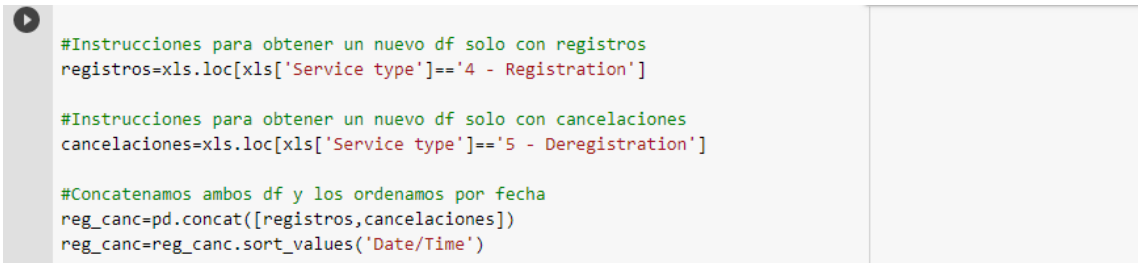

Figura 3.3. *Dataframe* registros y "desregistros"

Aunque tengamos solo los *dataframes* asociados a registros y "desregistros" seguimos teniendo los datos del resto de columnas, así que el siguiente paso es obtener el *dataframe* con las columnas que queremos del *dataframe* obtenido en el paso anterior (figura 3.4).

```
#Instrucciones para obtener la columna de cada dato necesario
columna_registro=reg_canc['Service type']
columna_hora=reg_canc['Date/Time']
columna_equipo=reg_canc['Source']
columna_localizacion=reg_canc['Source location']

#Concatenamos cada una de las columnas
matriz=pd.concat([columna_hora,columna_equipo,columna_registro,columna_localizacion],axis=1)
```

Figura 3.4. *Dataframe* matriz

Además, se eliminan las filas en las que la localización es VOIP ya que no corresponden a los datos que se deben tratar puesto que no aportan información acerca de la localización del usuario (figura 3.5).

```
#Eliminamos filas VOIP
matriz = matriz.drop(matriz[matriz['Source location']=="VOIP:"].index)
```

Figura 3.5. Eliminamos VOIP

A continuación, se obtiene el array con todos los usuarios en orden y el número de usuarios que se tiene, ya que se usarán posteriormente (figura 3.6).

```
#Obtenemos el array con todos los usuarios distintos y los ordenamos
usuarios=matriz['Source'].unique()
usuarios=np.sort(usuarios)

#Obtenemos el número de usuarios
n_usuarios=usuarios.size
print(n_usuarios)
```

Figura 3.6. Array con usuarios

Para continuar con el procesado de los datos, a partir de la matriz de la figura 3.4, se obtiene para cada uno de los usuarios, un *dataframe* en el que solo aparecerá la fecha, la hora y la localización origen. Acto seguido se crea una lista con todos los *dataframe* de todos los usuarios llamada datos (figura 3.7).

```
datos = list()
for j in range(n_usuarios):
    usuario=usuarios[j] #Sacamos el usuario del vector de usaurios
    dato=matriz.loc[matriz['Equipo origen']==usuario] #Obtenemos el df con los datos de ese usuario
    dato=dato.drop(['Tipo de servicio', 'Equipo origen'],axis=1) #Eliminamos columna tipo de servicio
                                                                 #y equipo origen
    datos.append(dato) #Añadimos a la lista
```

Figura 3.7. Lista de usuarios

3.2 Adaptación de los datos para la red

Una vez que se tiene la lista con todos los registros y “desregistros” de cada usuario se creará la matriz que será la entrada a la red neuronal. Para ello, se definirá un array de tres dimensiones, la primera dimensión se refiere a cada uno de los registros o “desregistros”, la segunda al valor de memoria que se desea y la última es el número de estaciones base posibles dentro de la red (figura 3.8).

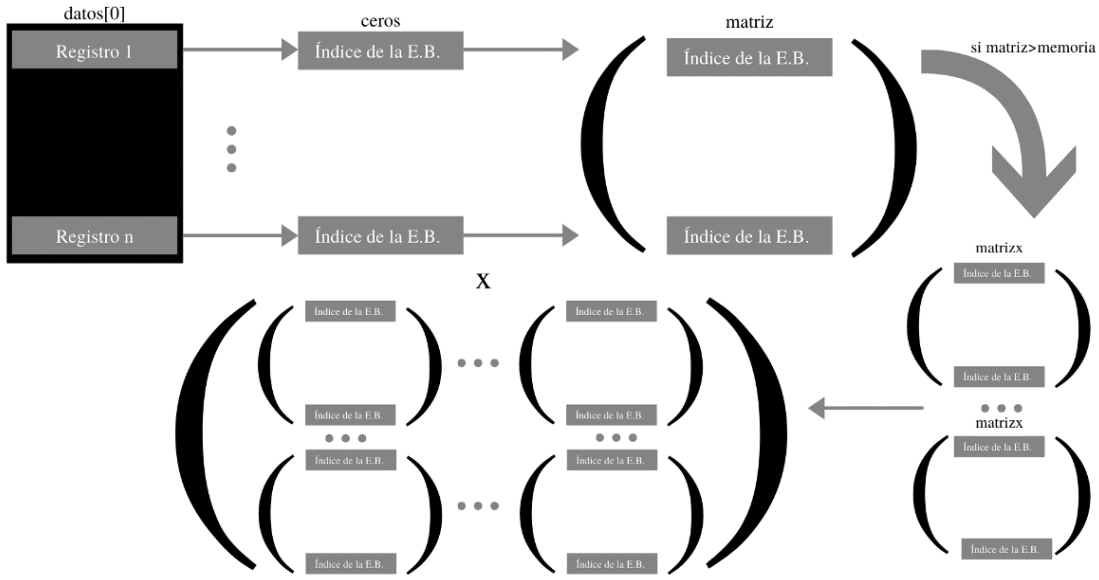

Figura 3.8. Esquema resumen de la adaptación de los datos a la red

Con memoria se hace referencia al número de registros anteriores que almacena la red en su memoria para realizar una predicción. Por tanto, lo que se hace es dividir el número total de registros en partes del tamaño de memoria.

Primero se declaran todas las variables que vamos a utilizar y obtenemos todas las estaciones base (figura 3.9).

```
import math
memoria=8
numero_partes=0
n=0
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
print(estaciones)
ceros=np.zeros(estaciones.size)
x=list()
seleccion=((memoria+1)*n)
```

Figura 3.9. Variables

A continuación, se va a ir recorriendo el *dataframe* datos creado en la figura 3.7 y se va a obtener un array con todas las estaciones base que atraviesa, por tanto, se elimina la columna de fecha y hora (figura 3.10).

```
for k in range(n_usuarios):

  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  #estacion_user=datos[k].drop(['Fecha/Hora'],axis=1)
  estacion_user=np.asarray(estacion_user)
```

Figura 3.10. Estaciones del usuario

Siguiendo dentro del bucle de la figura 3.10, se crea una matriz de ceros y unos, en la que cada fila será un registro y cada columna será la estación base asociada a ese registro. Lo que se hace es poner un uno en el índice asociado a esa estación base. Por ejemplo, si se tiene un registro asociado a la estación base con índice 23, refiriéndose a la posición que ocupa en el vector estaciones definido en la figura 3.9, se creará un array de ceros con un uno en el índice 23, y se añadirá a una matriz que quedará formada por todos los registros del usuario (figura 3.11).

```
for k in range(n_usuarios):

  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  #estacion_user=datos[k].drop(['Fecha/Hora'],axis=1)
  estacion_user=np.asarray(estacion_user)

  for l in range(estacion_user.size):
    estacion=estacion_user[l] #Estacion base a comparar
    for m in range(estaciones.size): #Comparamos esta estacion base con las del
                                     #vector en el que tenemos todas las estaciones base
      if(estaciones[m]==estacion):
        ceros=np.zeros(estaciones.size)
        ceros[m]=1
        matriz.append(ceros)
```

Figura 3.11. Matriz usuario

Por último, se tiene que comprobar si esta matriz creada en el paso anterior, figura 3.11, es mayor que la memoria que se ha definido en la figura 3.9; es decir, para que de esta forma al menos, se pueda utilizar una parte de los datos. Si es posible, se calcula el número total de partes y se va añadiendo cada una de ellas a una lista nombrada x, que será la entrada a la red neuronal, desarrollada en la figura 3.12:

```
for k in range(n_usuarios):

  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  #estacion_user=datos[k].drop(['Fecha/Hora'],axis=1)
  estacion_user=np.asarray(estacion_user)

  for l in range(estacion_user.size):
    estacion=estacion_user[l]
    for m in range(estaciones.size):
      if(estaciones[m]==estacion):
        ceros=np.zeros(estaciones.size)
        ceros[m]=1
        matriz.append(ceros)
  if(len(matriz)>memoria):
    numero_partes=math.floor(len(matriz)/(memoria+1))
    matrizx=list()
    if numero_partes==1:
      matrizx=matriz[:memoria+1]
      x.append(matrizx)
    else:
      for n in range(numero_partes):
        seleccion=(memoria+1)*n
        matrizx=matriz[seleccion:seleccion+(memoria+1)]
        x.append(matrizx)


x=np.asarray(x,dtype=np.float32)
print(x.shape)
```

Figura 3.12. Creación de x

Por haberse creado x de esta forma se estaría desaprovechando una gran parte de los datos, por lo que se incluye una variable salto con la que, en lugar de empezar la división de cada uno de los bloques (matrizx en la figura 3.12) cada m registros, siendo m el valor de la memoria, se empezará cada s (salto) registros (figura 3.8). Por ejemplo, si para un usuario se tiene un total de 13 registros, memoria=8 y salto=2, se quedaría en un total de 3 partes que empezarían en el primer, tercer y quinto registro (figura 3.13).

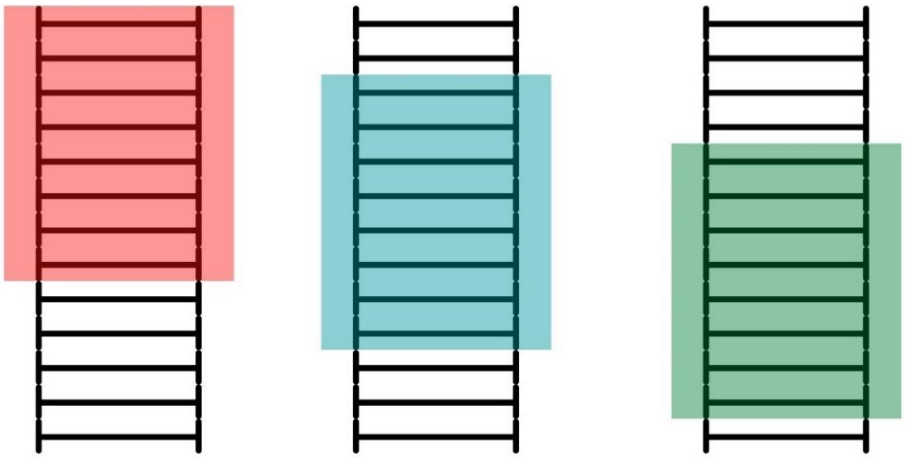

Figura 3.13. Ejemplo de datos con salto=2

El uso de esta variable será importante a la hora de definir nuestro modelo, ya que para diferentes valores de memoria se podrá obtener un número similar de muestras con las que entrenar la red de una forma más equitativa para los distintos valores de memoria. Esta variable salto puede ser elegida por el usuario, solo habría que editar el código de la figura 3.12 de la siguiente manera (figura 3.14):

```
x=list()
salto=2 #Dejamos en medio salto-1 muestras

for k in range(n_usuarios):

  inicio=0
  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  #estacion_user=datos[k].drop(['Fecha/Hora'],axis=1)
  estacion_user=np.asarray(estacion_user)

  for l in range(estacion_user.size):
    estacion=estacion_user[l] #Estacion base a comparar
    for m in range(estaciones.size): #Comparamos esta estacion base con las del
                                     #vector en el que tenemos todas las estaciones base
      if(estaciones[m]==estacion):
        ceros=np.zeros(estaciones.size)
        ceros[m]=1
        matriz.append(ceros)
  if (len(matriz)>memoria):#Si para ese usuario tenemos al menos una parte
    numero_partes=math.ceil((len(matriz)-(memoria)+1)/salto)
    matrizx=list()
    if numero_partes==1:
      matrizx=matriz[:memoria]
      x.append(matrizx)
    else:
      for n in range(numero_partes):
        matrizx=matriz[inicio:inicio+(memoria)]
        x.append(matrizx)
        inicio=inicio+(salto-1)

x=np.asarray(x,dtype=np.float32)
print(x.shape)
```

Figura 3.14. Creación de x con salto

En el caso de querer aprovechar al máximo todos los datos, lo que se va a hacer es comenzar la creación de cada bloque (matrizx) desde cada uno de los registros (figura 3.8). Con esto, volviendo al ejemplo anterior, si tenemos 13 registros y una memoria igual a 8 se dispone de un total de 6 partes (figura 3.15).

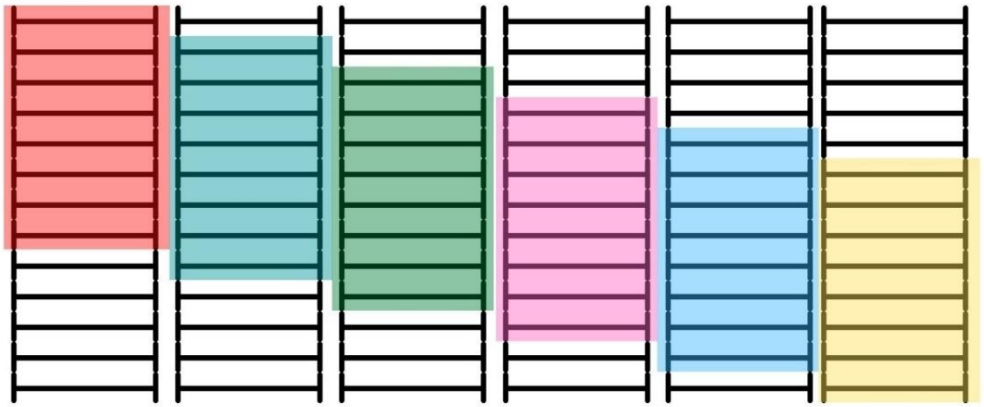

Figura 3.15. Ejemplo de datos con máximo aprovechamiento

Así es como quedaría la creación de x:

```
import math
memoria=6
numero_partes=0
n=0
inicio=0
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
print(estaciones)
ceros=np.zeros(estaciones.size)
x=list()

for k in range(n_usuarios):

  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  #estacion_user=datos[k].drop(['Fecha/Hora'],axis=1)
  estacion_user=np.asarray(estacion_user)

  for l in range(estacion_user.size):
    estacion=estacion_user[l] #Estacion base a comparar
    for m in range(estaciones.size): #Comparamos esta estacion base con las del
                                     #vector en el que tenemos todas las estaciones base
      if(estaciones[m]==estacion):
        ceros=np.zeros(estaciones.size)
        ceros[m]=1
        matriz.append(ceros)
  if (len(matriz)>memoria):
    numero_partes=len(matriz)-(memoria)+1
    matrizx=list()
    if numero_partes==1:
      matrizx=matriz[:memoria+1]
      x.append(matrizx)
    else:
      for n in range(numero_partes):
        inicio=n
        matrizx=matriz[inicio:inicio+(memoria)]
        x.append(matrizx)


x=np.asarray(x,dtype=np.float32)
print(x.shape)
```

Figura 3.16. Creación de x con todos los datos

3.3 Red neuronal

En lo que respecta al código de la red neuronal, se van a utilizar distintas librerías, pero la principal es la librería PyTorch, diseñada para realizar cálculos numéricos haciendo uso de la programación de tensores. Esta librería está adquiriendo gran relevancia en el ámbito del *machine learning*, ya que se centra en el desarrollo de aplicaciones con redes neuronales, como se va a utilizar en este proyecto.

Por tanto, lo primero que se debe hacer es importar todas las librerías que vamos a utilizar (figura 3.17):

```
import time
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
from random import randint
```

Figura 3.17. Librerías para red neuronal

Para comenzar, vamos a definir las variables N (número total de registros que tiene la entrada x) y T (número de muestras que utiliza para predecir la red de cada entrada, más la muestra que se quiere predecir). En la figura 3.18 y siguientes de este apartado, se muestra un ejemplo desarrollado en este proyecto. Por tanto, N será igual a la longitud de la primera dimensión de la entrada x y la T deberá ser igual a la memoria-1 para poder seleccionar desde el índice 0 hasta el índice memoria-1, seleccionando así para predecir un total de memoria muestras (figura 3.18).

```
N = 69609
T = memoria-1
```

Figura 3.18. Variables N y T

Después, se va a dividir la entrada en los tres tipos de muestras que comentamos en el apartado 2.1.2. Se dedicará aproximadamente el 70% de los datos al tipo de entrenamiento (x_train), el 20% al tipo de test (x_test) y el 10% restante para validación (figura 3.19). Estos porcentajes son valores estándar que se utilizan habitualmente en los problemas de *machine learning*.

```
x_test = x[55609:]
x_dev = x[48609:55609]
x_train = x[:48609]
```

Figura 3.19. División en tipos

Antes de empezar a desarrollar el código de la red, se van a declarar variables esenciales definidas en el apartado teórico, que son el número de *epochs* y el *batch size* (figura 3.20). Además, se va a definir la dimensión de entrada y la dimensión de salida de la propia red. La dimensión de entrada es igual a la tercera dimensión de la entrada x, en nuestro caso el número de estaciones base. En lo que respecta a la dimensión de salida, va a ser igual a la dimensión de entrada, en el ejemplo expuesto, porque se quiere volver a obtener un vector de ceros y unos, en el que la red nos indique la predicción que ha hecho mediante

la posición del número uno en ese vector de salida, es decir, el vector de salida contiene un uno en la estación base predicha y ceros en las restantes.

```
nb_epochs = 10
batch_size = memoria
input_dim=57
output_dim=57
```

Figura 3.20. Variable *epochs*, *batch size* y dimensiones

A continuación, ya se puede inicializar la red neuronal. Primero se define la clase torch.nn.Module, a continuación se declara el constructor del modelo en init() en el que se fijará el número de capas ocultas (2 en este caso) y el *dropout*, este valor es una probabilidad de desactivación de neuronas con el que evitamos el sobreajuste, lo que podría llevar a un mal funcionamiento de la red; se ha utilizado en todo el proyecto el valor estándar de 0,5.

Además, se inicializa el modelo de red que se quiere (LSTM en nuestro caso), la función lineal (nn.Linear de la figura 3.21), con la que se pasa de un vector con la dimensión de las capas ocultas de la LSTM al número de dimensiones de la salida, y la función de pérdidas (nn.CrossEntropyLoss).

Dentro de la misma clase Net se define la función forward, que son todas las operaciones que se van a realizar desde la entrada a la red hasta su salida. Y se devuelve x[:,-1] para que se devuelva como salida sólo la correspondiente al último instante de tiempo.

Por último, se define la función de pérdidas y se inicializa la red (figura 3.21). En la instrucción con la que inicializamos la red (model) podemos indicar la dimensión que queremos que tenga las capas ocultas.

```
class Net(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nb_layers=2, dropout=0.5):
        super(Net, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, nb_layers, batch_first=True, bidirectional=False, dropout=dropout)
        self.fc = nn.Linear( hidden_dim, output_dim)
        self.J = nn.CrossEntropyLoss()

    def forward(self, x):#Operaciones desde la entrada a la salida
        x, state = self.lstm(x)
        x = self.fc(x)
        return x[:,-1]

    def loss(self, out, y):
        return self.J(out, y.argmax(dim=1) )

model = Net(input_dim, 64, output_dim) #Parámetro central igual a la dimensión de las capas ocultas
nb_param = sum(p.numel() for p in model.parameters())
print(model)
print('# param:    %d' % nb_param)
```

Figura 3.21. Inicialización de la red

Un esquema sencillo de cómo quedaría la red neuronal sería el de la figura 3.22:

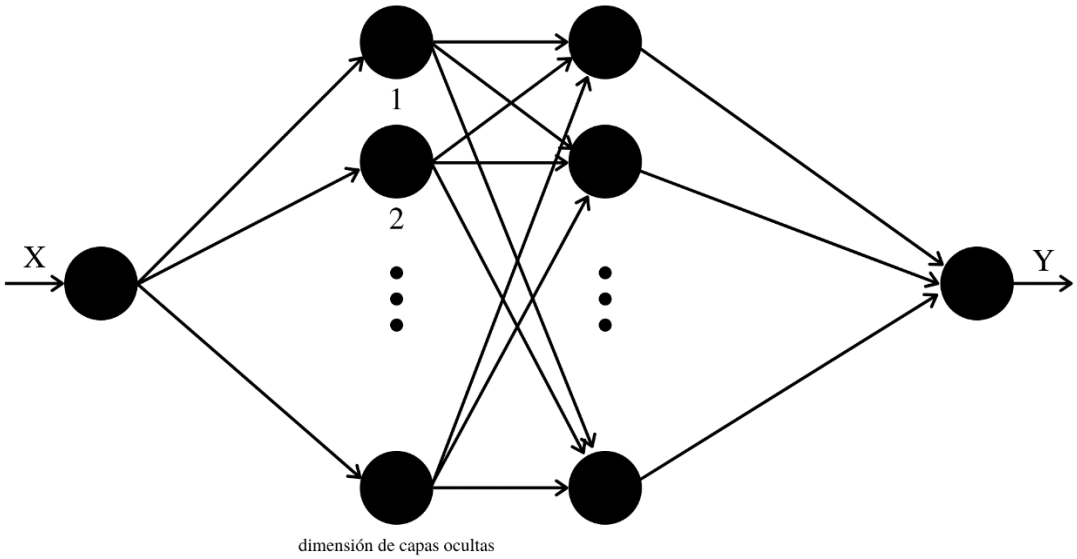

Figura 3.22. Esquema básico de red neuronal

Una vez inicializada la red se va a definir el optimizador, en el apartado 2.1.5 se mencionó que se iba a utilizar el algoritmo de Adam. Además, se incluye la función lr_scheduler.StepLR para evitar que la tasa de aprendizaje sea muy elevada y evitar de esta forma inestabilidades en la función de pérdidas (figura 3.23).

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
```

Figura 3.23. Inicialización del optimizador

En todo el proyecto, los valores de tasa de aprendizaje (*learning rate*), tamaño de paso (*step size*) y gamma son los valores típicos, según figura 3.23.

Por último, solo queda ejecutar la red neuronal desarrollando el código con las siguientes pautas.

Primero, definir dos *dataloader*, uno para los datos de entrenamiento y otro para los datos de validación. Y se inicializan el valor de la función de pérdidas del entrenamiento y el de la función de pérdidas de validación (figura 3.24).

```
trainloader = data.DataLoader(x_train, batch_size=batch_size, shuffle=True, num_workers=0)
devloader = data.DataLoader(x_dev, batch_size=1, shuffle=False, num_workers=0)
loss_train = np.zeros(nb_epochs)
loss_dev = np.zeros(nb_epochs)
```

Figura 3.23. Inicializar dataloader y valor de funciones de pérdidas

Segundo, indicar a la red que va a comenzar el entrenamiento con la función model.train() y comenzar con el bucle de *epochs* (figura 3.25). Antes de nada, se varía la tasa de aprendizaje en función de lo definido en la inicialización del optimizador. Se continúa guardando el tiempo, para poder calcular cuánto se tarda en ejecutar la *epoch* y volver a llamar a la función model.train().

Tercero, se inicializa el bucle en el que los datos de entrenamiento atraviesan la red. En primer lugar se llama a la función variable() con la que se convierte el tensor de entrada

en una variable del módulo autograd, con la que la red puede operar. Después, se declaran los valores a predecir y los valores previos a este. Se llama a la función zero_grad() del optimizador con la que ponemos a cero los gradientes para que la red pueda actualizar los parámetros correctamente; de lo contrario, el gradiente apuntaría a otra dirección que no es la prevista hacia el mínimo. Se calcula la salida de la red, el error, los gradientes (función loss.backward()) y se adaptan los pesos con los gradientes calculados. Para terminar con el entrenamiento se almacena el valor medio de pérdidas de entrenamiento y el tiempo; de esta forma, cuando se quiera imprimir la pérdida media en el entrenamiento se va a poder saber la duración de cada *epoch*.

```
model.train()
for i in range(nb_epochs):
    if i > 0:
        scheduler.step(i)

    # ---- W
    tic = time.time()
    model.train()
    for x in trainloader:
        x = Variable( x )
        y = x[:,T]
        x = x[:,:T]

        optimizer.zero_grad()
        out = model(x)
        loss = model.loss(out, y)
        loss.backward()
        optimizer.step()

        loss_train[i] += loss.item() / len(x_train) * len(x)
    toc = time.time()

    # ---- print
    print("it %d/%d, Jtr = %f, time: %.2fs" % (i, nb_epochs, loss_train[i], toc - tic))
```

Figura 3.25. Entrenamiento de la red

Cuarto, se comienza con la parte de evaluación. Mientras la red está siendo entrenada, ésta ordena aleatoriamente las muestras de entrenamiento con el fin de promediar la validación, y va adaptando los parámetros de peso y sesgo, con el fin de obtener los mejores resultados con el algoritmo de *backpropagation* explicado en el apartado 2.1.4. Para ello, al igual que en el entrenamiento, se le indica a la red con la instrucción model.eval() que comienza el bucle de evaluación con las mismas instrucciones que en el bucle de entrenamiento, a excepción de la función backward() y el *step* del optimizador. Así es como se vería todo el código de la red (figura 3.26):

```
trainloader = data.DataLoader(x_train, batch_size=batch_size, shuffle=True, num_workers=0)
devloader = data.DataLoader(x_dev, batch_size=1, shuffle=False, num_workers=0)
loss_train = np.zeros(nb_epochs)
loss_dev = np.zeros(nb_epochs)

model.train()
for i in range(nb_epochs):
    if i > 0:
        scheduler.step(i)

    # ---- W
    tic = time.time()
    model.train()
    for x in trainloader:
        x = Variable( x )
        y = x[:,T]
        x = x[:,:T]

        optimizer.zero_grad()
        out = model(x)
        loss = model.loss(out, y)
        loss.backward()
        optimizer.step()

        loss_train[i] += loss.item() / len(x_train) * len(x)
    toc = time.time()

    # ---- print
    print("it %d/%d, Jtr = %f, time: %.2fs" % (i, nb_epochs, loss_train[i], toc - tic))

    # ---- dev
    model.eval()
    for x in devloader:
        x = Variable( x )
        #x = x.cuda()

        y = x[:,T]
        x = x[:,:T]
        out = model(x)
        loss = model.loss(out, y)
        loss_dev[i] += loss.item() / len(x_dev) * len(x)

        print('    Jdev = %f, err = %f\n' % (loss_dev[i], err_dev[i]))
    print('    Jdev = %f\n' % loss_dev[i])
```

Figura 3.26. Funcionamiento red neuronal

3.3.1 Información de usuario

La red diseñada hasta el momento utiliza de manera genérica la información de todos los usuarios para predecir unos patrones de movilidad genéricos aplicables a todos ellos.

Si se quisiese realizar el entrenamiento de manera independiente para cada usuario presentaría dos problemas: por una parte, sería necesario entrenar una red neuronal distinta para cada usuario de la red, lo que incrementa la complejidad notablemente. Por otra parte, es posible que no se disponga de un número suficiente de movimientos para todos los usuarios para poder realizar un entrenamiento correcto.

En este apartado se va a incluir dentro de los datos de entrada a la red neuronal la información del usuario al que corresponde esa secuencia de estaciones base; es decir, el identificador del usuario dentro de la red. La idea es que de esa manera se siga entrenando de manera genérica la red neuronal con información de todos los usuarios, pero se incluye también información del usuario con el fin de comprobar si la red neuronal es capaz de aprender a diferenciar los patrones de movimiento de unos usuarios u otros. Volviendo al esquema de la figura 3.8, este sería ahora el esquema con la información del usuario (figura 3.27):

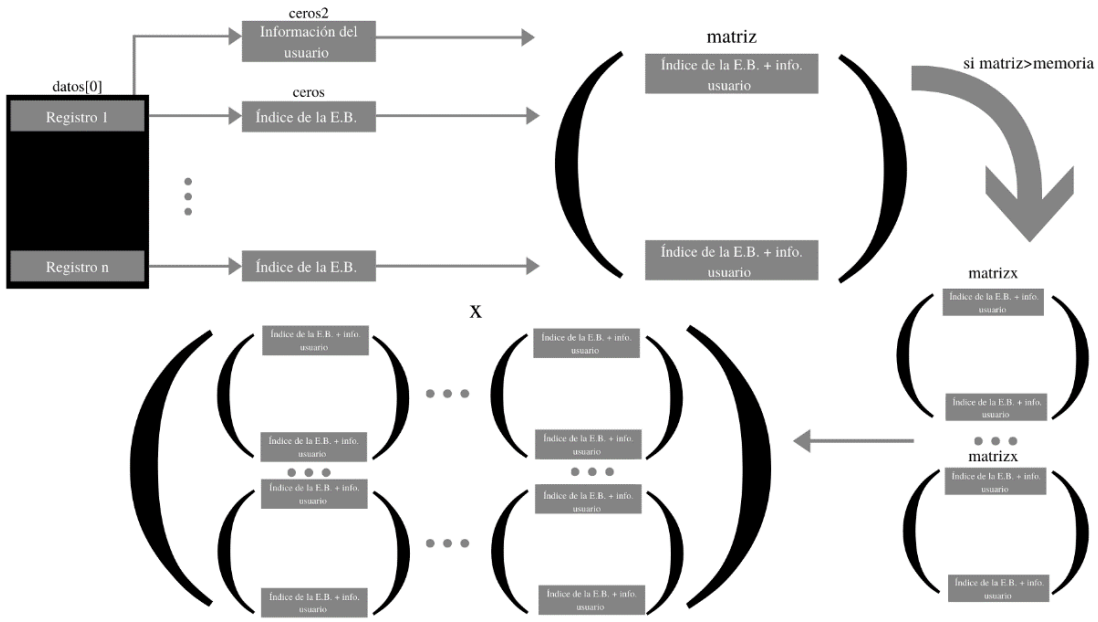

Figura 3.27. Esquema resumen de la adaptación de los datos a la red con información del usuario

Para ello se van a implementar dos posibilidades: la primera, consiste en añadir a cada fila de la matriz de entrada, un array del tamaño del número de usuarios con un uno en el índice correspondiente al usuario, denominada no codificada; y la segunda, es una idea similar a la primera, pero codificando el índice del usuario en binario, denominada codificada.

3.3.1.1 No codificada

El único cambio que se debe hacer se corresponde con el de la figura 3.12, se crea un vector de ceros de la longitud del número de usuarios y se le pone un uno en función del índice del for (figura 3.29.); además, se concatena con el vector de ceros y uno que teníamos previamente. La información será introducida según se ve en la figura 3.28:

Figura 3.28. Creación de ceros2 sin codificar

```
for k in range(n_usuarios):

  ceros2=np.zeros(n_usuarios)
  ceros2[k]=1
  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  estacion_user=np.asarray(estacion_user)

  for l in range(estacion_user.size):
    estacion=estacion_user[l]
    for m in range(estaciones.size):
      if(estaciones[m]==estacion):
        ceros=np.zeros(estaciones.size)
        ceros[m]=1
        c=np.concatenate((ceros,ceros2),axis=0)
        matriz.append(c)
  if(len(matriz)>memoria):
    numero_partes=len(matriz)-(memoria)+1
    matrizx=list()
    if numero_partes==1:
      matrizx=matriz[:memoria+1]
      x.append(matrizx)
    else:
      for n in range(numero_partes):
        inicio=n
        matrizx=matriz[inicio:inicio+(memoria)]
        x.append(matrizx)


x=np.asarray(x,dtype=np.float32)
print(x.shape)
```

Figura 3.29. Información de usuario no codificada

Aquí se debe prestar atención, porque la última dimensión de la matriz x va a variar y habrá que inicializarla de manera correcta tal y como aparece en la figura 3.20.

3.3.1.2 Codificada

Al igual que en el caso anterior, solo se modifica el código para la creación de x, pero en este caso añadiremos una nueva variable con la que saber la longitud máxima (lenbin); es decir, la longitud máxima que tendrá el índice más alto convertido en binario, para poder crear el vector de ceros y unos. En la figura 3.30 se puede ver como se creará la información del usuario de manera codificada:

Figura 3.30. Creación de ceros2 codificada

Así quedará el código (figura 3.31):

```
binmax=bin(n_usuarios)[2:]
lenbin=len(binmax)

for k in range(n_usuarios):

  binario=bin(k)[2:]
  binario=np.array(list(binario.zfill(lenbin)), dtype=int)
  matriz=list()
  estacion_user=datos[k].drop(['Date/Time'],axis=1)
  estacion_user=np.asarray(estacion_user)

  for l in range(estacion_user.size):
    estacion=estacion_user[l]
    for m in range(estaciones.size):
      if(estaciones[m]==estacion):
        ceros=np.zeros(estaciones.size)
        ceros[m]=1
        c=np.concatenate((ceros,binario),axis=0)
        matriz.append(c)
  if(len(matriz)>memoria):
    numero_partes=len(matriz)-(memoria)+1
    matrizx=list()
    if numero_partes==1:
      matrizx=matriz[:memoria+1]
      x.append(matrizx)
    else:
      for n in range(numero_partes):
        inicio=n
        matrizx=matriz[inicio:inicio+(memoria)]
        x.append(matrizx)


x=np.asarray(x,dtype=np.float32)
print(x.shape)
```

Figura 3.31. Información de usuario codificada

Igualmente hay que volver a prestar atención, porque la última dimensión de la matriz x va a variar y habrá que inicializarla de manera correcta tal y como aparece en la figura 3.20.

3.3.2 Predicción de varias estaciones

Además de poder predecir la siguiente estación, se va a implementar la posibilidad de predecir desde las dos hasta las seis siguientes estaciones con el fin de comprobar las prestaciones en los diferentes escenarios y comprobar la capacidad de la red para predecir la trayectoria del usuario y no solo la estación base siguiente. Para ello, solo se variará el código de inicialización de la red que aparece en la figura 3.16 y el de la red que aparece en la figura 3.21. Lo que se hace es inicializar tantas funciones Linear() como estaciones se quieran predecir. Asimismo, tanto en el entrenamiento como en la validación, se crean tantas salidas como estaciones se quieren predecir (figura 3.32).

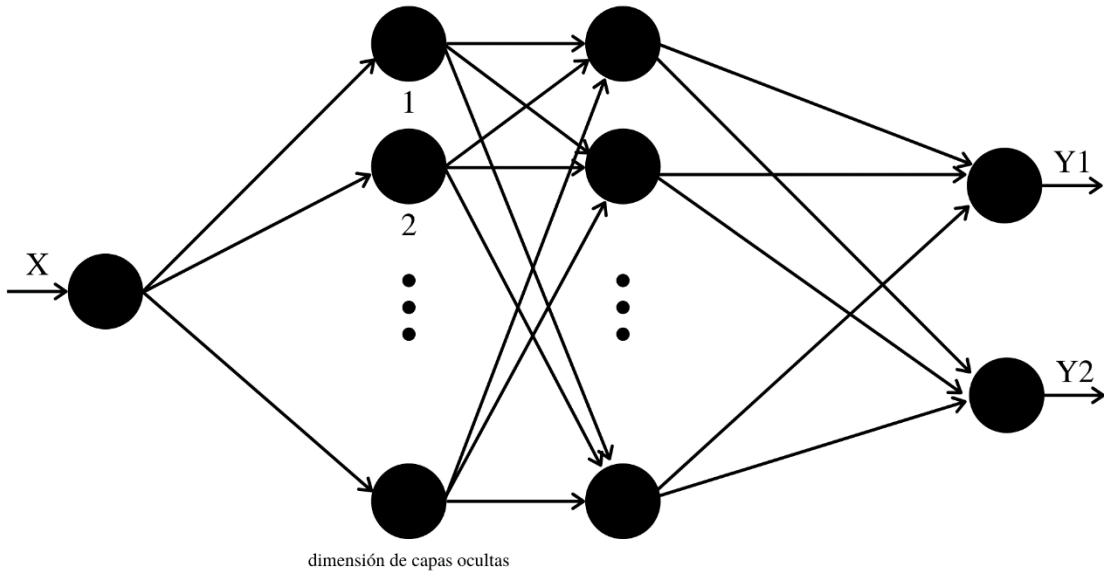

Figura 3.32. Esquema básico de red neuronal para predecir dos movimientos

En lo que respecta a la predicción para dos movimientos, el código de inicialización y de la red neuronal queda tal como se expone en las figuras 3.33 y 3.34:

```
class Net(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nb_layers=2, dropout=0.5):
        super(Net, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, nb_layers, batch_first=True, bidirectional=False, dropout=dropout)
        self.fc1 = nn.Linear( hidden_dim, output_dim)
        self.fc2 = nn.Linear( hidden_dim, output_dim)
        self.J = nn.CrossEntropyLoss()

    def forward(self, x):
        x, state = self.lstm(x)
        x1 = self.fc1(x)
        x2 = self.fc2(x)
        return x1[:,-1] ,x2[:,-1]

    def loss(self, out,y):
        return self.J(out, y.argmax(dim=1) )

model = Net(input_dim, 64, output_dim)
```

Figura 3.33. Inicialización de red para predecir dos movimientos

Cuando se añade un mayor número de estaciones a predecir, se deben añadir tantos fc_n como salidas se quieran, siendo n el número de salidas (estaciones a predecir).

```
model.train()
for i in range(nb_epochs):
    if i > 0:
        scheduler.step(i)

    # ---- W
    tic = time.time()
    model.train()
    for x in trainloader:
        x = Variable( x )
        #print(x.shape)
        #x = x.cuda()

        y1 = x[:,T-1] # value to predict
        y2 = x[:,T]
        x = x[:,:T-1]     # previous values
        #print(x.shape, y.shape)

        optimizer.zero_grad()
        out1 , out2 = model(x)
        #print(out.shape)
        loss1 = model.loss(out1,y1)
        loss2 = model.loss(out2,y2)
        loss = loss1+loss2
        loss.backward()
        optimizer.step()

        loss_train[i] += loss.item() / len(x_train) * len(x)
    toc = time.time()

    # ---- print
    print("it %d/%d, Jtr = %f, time: %.2fs" % (i, nb_epochs, loss_train[i], toc - tic))

    # ---- dev
    model.eval()
    for x in devloader:
        x = Variable( x )
        #x = x.cuda()

        y1 = x[:,T-1] # value to predict
        y2 = x[:,T]
        x = x[:,:T-1]     # previous values

        out1 , out2 = model(x)
        loss1 = model.loss(out1,y1)
        loss2 = model.loss(out2,y2)
        loss = loss1+loss2

        loss_dev[i] += loss.item() / len(x_dev) * len(x)

        print('    Jdev = %f, err = %f\n' % (loss_dev[i], err_dev[i]))
    print('    Jdev = %f\n' % loss_dev[i])
```

Figura 3.34. Funcionamiento de red para predecir dos movimientos

Cuando se añade un mayor número de estaciones a predecir, se deben añadir tantas y_n, out_n y $loss_n$ como salidas se quieran, siendo n el número de salidas (estaciones a predecir) tanto en el apartado de entrenamiento como en el de validación.

En relación con el optimizador, no habría que hacer ningún cambio.

3.4 Modelo de Markov

Para implementar el modelo de Markov se van a procesar los datos de la figura 3.6 de manera que se obtenga una matriz de n dimensiones, en función de las estaciones base anteriores que se utilicen para predecir la siguiente. Es decir, si se quiere predecir una estación base en función de la anterior estación se usará una matriz de dos dimensiones, la primera se refiere a la estación base previa y la segunda a la probabilidad que hay para que la predicción sea esa estación base.

La matriz se creará de manera muy similar a la creación de la x como en la figura 3.10, pero en este caso (figura 3.35), en vez de poner un uno se van a ir sumando todos los desplazamientos; y después se dividirán por el número total de desplazamientos que hay desde esa estación base. De este modo, tal y como se ha explicado anteriormente, se obtienen las probabilidades de transición entre estados.

```
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
markov2=np.zeros([57,57])
desplazamientos=np.zeros([57,57])
indice1=0
indice2=0
estacion1='null'
prediccion='null'

for n in range(n_usuarios):
  if (len(datos[n])>1):
    estacion_user=datos[n].drop(['Date/Time'],axis=1)
    estacion_user=np.asarray(estacion_user)
    numero_partes=len(datos[n])-1
    for l in range (numero_partes):
      estacion1=estacion_user[l]
      prediccion=estacion_user[l+1]
      for p in range(57):

        if (estaciones[p]==estacion1):
          indice1=p
        if (estaciones[p]==prediccion):
          indice2=p
      markov2[indice1][indice2]=markov2[indice1][indice2]+1
      desplazamientos[indice1]=desplazamientos[indice1]+1

markov=markov2/desplazamientos
print(markov.shape)
```

Figura 3.35. Creación de la matriz Markov

Por último, para validar la eficiencia del modelo, se suman todas las probabilidades de transición de los datos escogidos desde cada uno de sus estados.

Pero antes de calcular la validación se va a poner a cero todos los valores de la matriz para los que hemos dividido por cero, para evitar problemas al obtener el valor de validación (figura 3.36). Ya que si el valor de ese índice es cero; es decir, no ha habido ningún desplazamiento desde esa estación base, no se perderá ningún valor de probabilidad.

```
for i in range(57):
  if (desplazamientos[i][0]==0):
    for k in range(57):
      markov[i][j]=0
```

Figura 3.36. Evitar problemas al dividir por cero

Una vez solucionado el problema de dividir por cero, se puede calcular el valor de validación, para ello se recorre toda la matriz y se van sumando todas las probabilidades máximas de cada fila (figura 3.37).

```
validar=0.00
for i in range(57):
  validar=validar+np.amax(markov[i])
print(validar)
```

Figura 3.37. Cálculo para el valor de validación

Para poder predecir el movimiento a una estación base, a partir de las dos estaciones previas, el código que se usará es muy similar al anterior, con la única diferencia de que en este caso la matriz pasará a ser de tres dimensiones, las dos primeras se corresponden con las estaciones previas y la tercera con la probabilidad que hay para que la predicción sea esa estación base (figura 3.38).

```
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
markov2=np.zeros([57,57,57])
desplazamientos=np.zeros([57,57,57])
indice1=0
indice2=0
indice3=0
estacion1='null'
estacion2='null'
prediccion='null'

for n in range(n_usuarios):
  if (len(datos[n])>2):
    estacion_user=datos[n].drop(['Date/Time'],axis=1)
    estacion_user=np.asarray(estacion_user)
    numero_partes=len(datos[n])-2
    for l in range (numero_partes):
      estacion1=estacion_user[l]
      estacion2=estacion_user[l+1]
      prediccion=estacion_user[l+2]
      for p in range(57):
        if (estaciones[p]==estacion1):
          indice1=p
        if (estaciones[p]==estacion2):
          indice2=p
        if (estaciones[p]==prediccion):
          indice3=p

      markov2[indice1][indice2][indice3]=markov2[indice1][indice2][indice3]+1
      desplazamientos[indice1][indice2]=desplazamientos[indice1][indice2]+1

markov=markov2/desplazamientos
print(markov.shape)
```

Figura 3.38. Creación de la matriz Markov con dos estaciones previas

Al igual que en la implementación para una estación previa, para calcular el valor de validación ponemos a cero los valores que se han dividido por cero para evitar problemas a la hora de calcular el valor de validación (figura 3.39).

```
for i in range(57):
  for j in range(57):
    if (desplazamientos[i][j][0]==0):
      for k in range(57):
        markov[i][j][k]=0
```

Figura 3.39. Evitar problemas al dividir por cero

La única diferencia para la matriz en este desarrollo es que se tiene que recorrer las dos primeras dimensiones para calcular el valor de validación (figura 3.40).

```
validar=0.00
for i in range(57):
  for j in range(57):
    validar=validar+np.amax(markov[i][j])
print(validar)
```

Figura 3.40. Cálculo para el valor de validación para Markov con dos estaciones previas

Si se le añade una tercera estación a cada uno de los estados, se crea la matriz de Markov según la figura 3.41:

```
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
markov2=np.zeros([57,57,57,57])
desplazamientos=np.zeros([57,57,57,57])
indice1=0
indice2=0
indice3=0
indice4=0
estacion1='null'
estacion2='null'
estacion3='null'
prediccion1='null'

for n in range(n_usuarios):
  if (len(datos[n])>3):
    estacion_user=datos[n].drop(['Date/Time'],axis=1)
    estacion_user=np.asarray(estacion_user)
    numero_partes=len(datos[n])-3
    for l in range (numero_partes):
      estacion1=estacion_user[l]
      estacion2=estacion_user[l+1]
      estacion3=estacion_user[l+2]
      prediccion1=estacion_user[l+3]

      for p in range(57):
        if (estaciones[p]==estacion1):
          indice1=p
        if (estaciones[p]==estacion2):
          indice2=p
        if (estaciones[p]==estacion3):
          indice3=p
        if (estaciones[p]==prediccion1):
          indice4=p

      markov2[indice1][indice2][indice3][indice4]=markov2[indice1][indice2][indice3][indice4]+1
      desplazamientos[indice1][indice2][indice3]=desplazamientos[indice1][indice2][indice3]+1

markov=markov2/desplazamientos
print(markov.shape)
```

Figura 3.41. Creación de la matriz Markov con tres estaciones previas

Al igual que en los dos desarrollos anteriores, para calcular el valor de validación, ponemos a cero los valores que se han dividido por cero para evitar problemas a la hora de calcular el valor de validación (figura 3.42).

```
for i in range(57):
  for j in range(57):
    for l in range(57):
      if (desplazamientos[i][j][l][0]==0):
        for k in range(57):
          markov[i][j][l][k]=0
```

Figura 3.42. Evitar problemas al dividir por cero

Y, por último, para calcular el valor de validación, como se ha comentado en el capítulo 2, se va a sumar las probabilidades de transición de estados máximas para datos de validación.

4. Pruebas y resultados

En este capítulo se van a explicar todas las pruebas realizadas y los resultados obtenidos en ellas, tanto para la red neuronal como para el modelo de Markov.

4.1 Red neuronal

Lo primero que se debe hacer es encontrar los parámetros adecuados para nuestra red neuronal: dimensión de las capas ocultas, memoria, número de datos, etc…

Para empezar, se busca el valor de memoria y dimensión de las capas ocultas adecuado, para ello, inicialmente y para reducir el tiempo de entrenamiento no se ha incluido la variable de salto en los datos de entrenamiento, sino que se seleccionan en bloques del tamaño de la memoria, tal y como se hace en la figura 3.12. Realizando lo que se acaba de comentar y aplicando los porcentajes de datos que se comentaron en el apartado 3.3 para test, validación y entrenamiento, se obtienen:

Para memoria=6:

- 14258 muestras para entrenamiento
- 4000 muestras para test
- 2000 muestras para validación

Para memoria=8:

- 11172 muestras para entrenamiento
- 3000 muestras para test
- 1500 muestras para validación

Para memoria=10:

- 8959 muestras para entrenamiento
- 2500 muestras para test
- 1273 muestras para validación

Para la dimensión de las capas ocultas se prueba con los valores 32, 64 y 128. Lo que se hace es obtener los valores de la función de validación (J) para ver qué modelo es el adecuado usando el conjunto de muestras de validación, promediando los resultados de cinco pruebas distintas, en las que se varía aleatoriamente el orden de las muestras de entrenamiento, realizado cuando se declara *true* la variable shuffle de los *dataloader* de la figura 3.23. Para ello, se hace la prueba con tantos *epochs* como sean necesarios para obtener el valor mínimo de la J. Por tanto, para asegurarnos de ello se utilizarán 40 *epochs*. En primer lugar, se analizan los resultados obtenidos con memoria igual a 6 y para los tres distintos valores de dimensión de las capas ocultas que ya se han comentado.

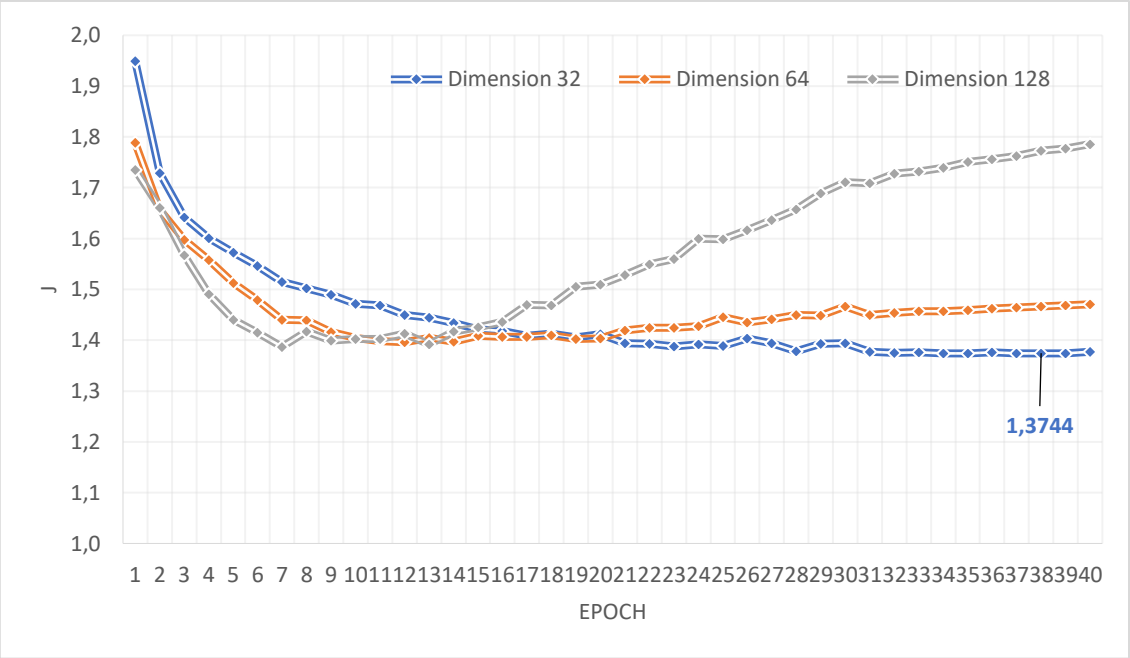

Figura 4.1. J para memoria igual a 6

Se observa en la figura 4.1 que el mejor valor de J será igual a 1,3744 con una dimensión de las capas ocultas igual a 32.

En segundo lugar, se va a analizar los valores de J al igual que en la implementación anterior, pero con memoria igual a 8.

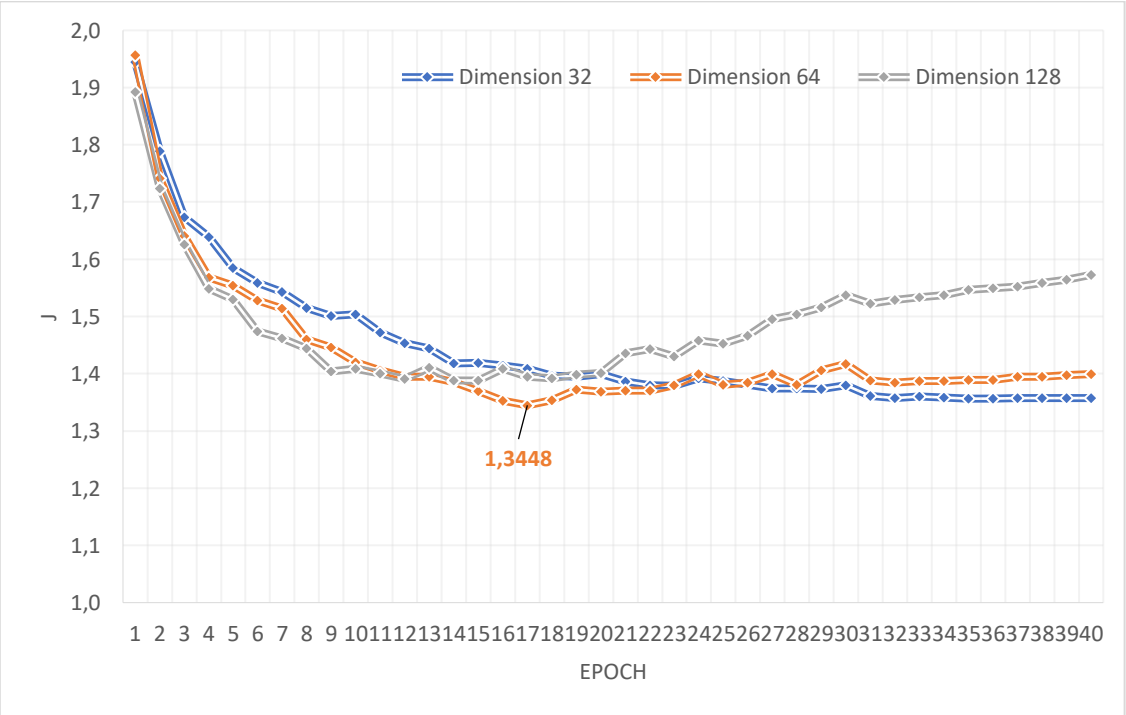

Figura 4.2. J para memoria igual a 8

En este caso (figura 4.2), el mejor valor de J es igual a 1,3448 y se obtiene con dimensión oculta igual a 64.

En último lugar, se realiza la misma prueba, pero con memoria igual a 10.

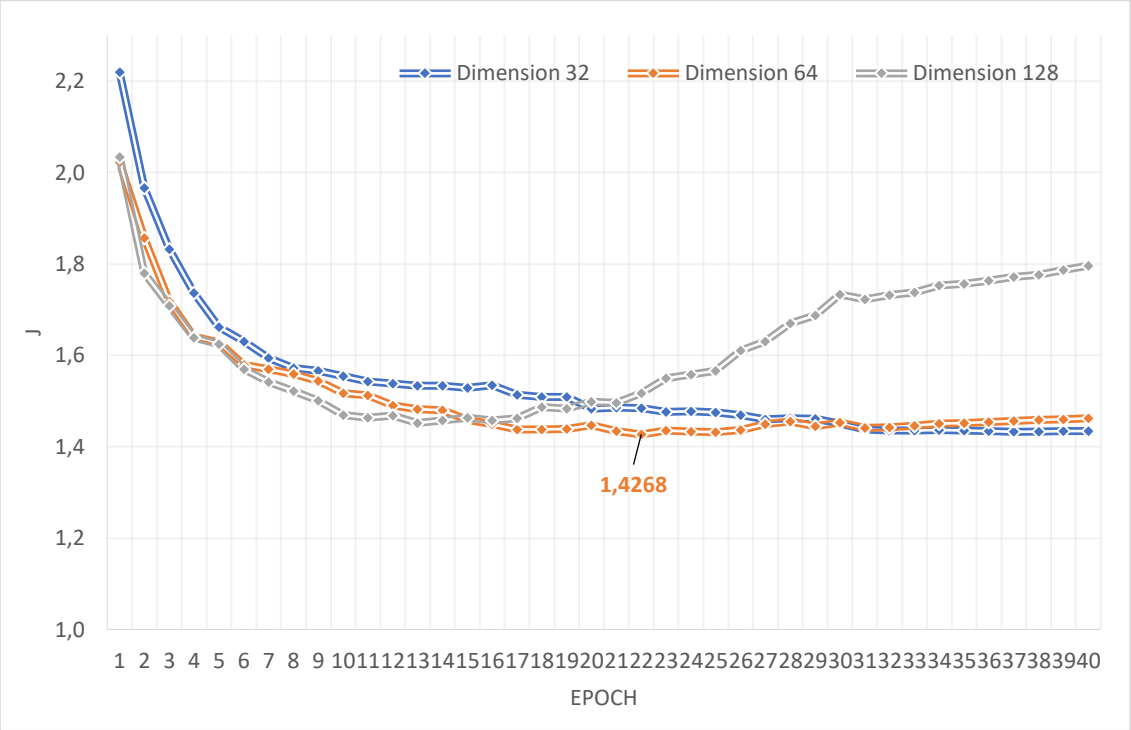

Figura 4.3. J para memoria igual a 10

En este desarrollo (figura 4.3), se obtiene una J igual a 1,4268 con dimensión oculta igual a 64. Con estos resultados, se observa que el mejor modelo de red a utilizar tiene memoria igual a 8 y un valor de dimensión de las capas ocultas igual a 64.

Con la finalidad de dar más información a la red, se va a aumentar el número de datos. Para ello se incluyó la variable salto, explicada en el apartado 3.1. Con la inclusión de esta variable, se obtiene el siguiente número de muestras:

- Salto=4:
  - Memoria=6
    - 24920 muestras para entrenar
    - 7000 muestras para test
    - 3500 muestras para validación
  - Memoria=8
    - 24500 muestras para entrenar
    - 7000 muestras para test
    - 3500 muestras para validación
  - Memoria=10
    - 24101 muestras para entrenar
    - 7000 muestras para test
    - 3500 muestras para validación

Observando estos datos, se va a poder discernir cuál de los tres valores de memoria es más adecuado, ya que tenemos un número de muestras muy similar. Estos serán los resultados que se obtendrán, pero en este caso se fijará el número de *epochs* en 20. Puesto

que, al aumentar el número de muestras, la J convergerá al mínimo antes que en los sucesos anteriores. Además, el valor de la dimensión de las capas ocultas será igual a 64.

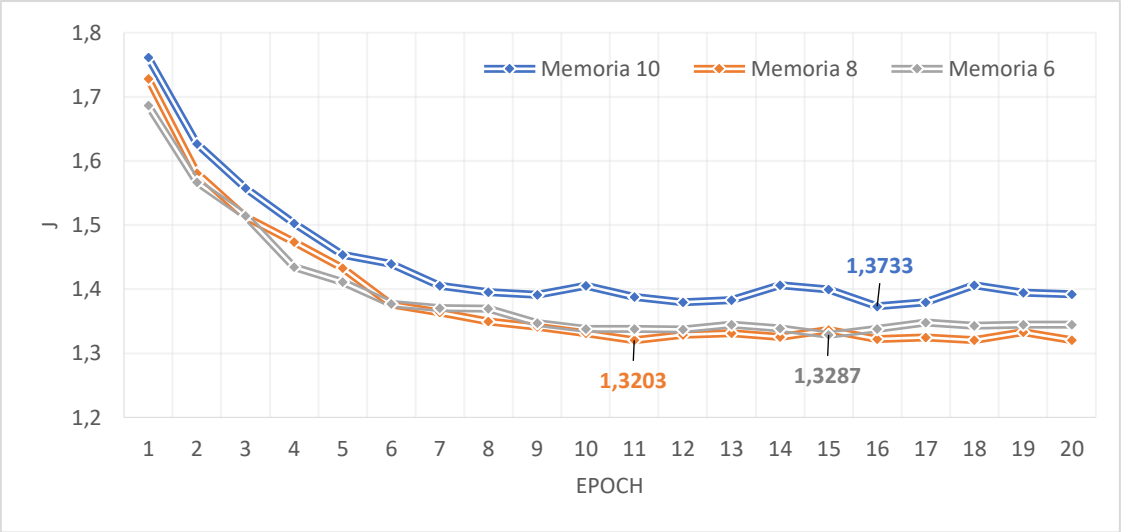

Figura 4.4. Valores de J con salto 4

Observando los resultados de la figura 4.4, el valor de J más pequeño se obtiene con memoria igual a 8 y ha bajado a 1,3203. Como se ha conseguido bajar el valor de la J, se vuelve a aumentar la variable salto a 2 y obtenemos el siguiente número de muestras:

- Salto=2:
  - Memoria=6:
    - Muestras para entrenar: 49429
    - Muestras para test: 14000
    - Muestras para validación: 7000
  - Memoria=8
    - Muestras para entrenar: 48609
    - Muestras para test: 14000
    - Muestras para validación: 7000
  - Memoria=10
    - Muestras para entrenar: 47815
    - Muestras para test: 14000
    - Muestras para validación: 7000

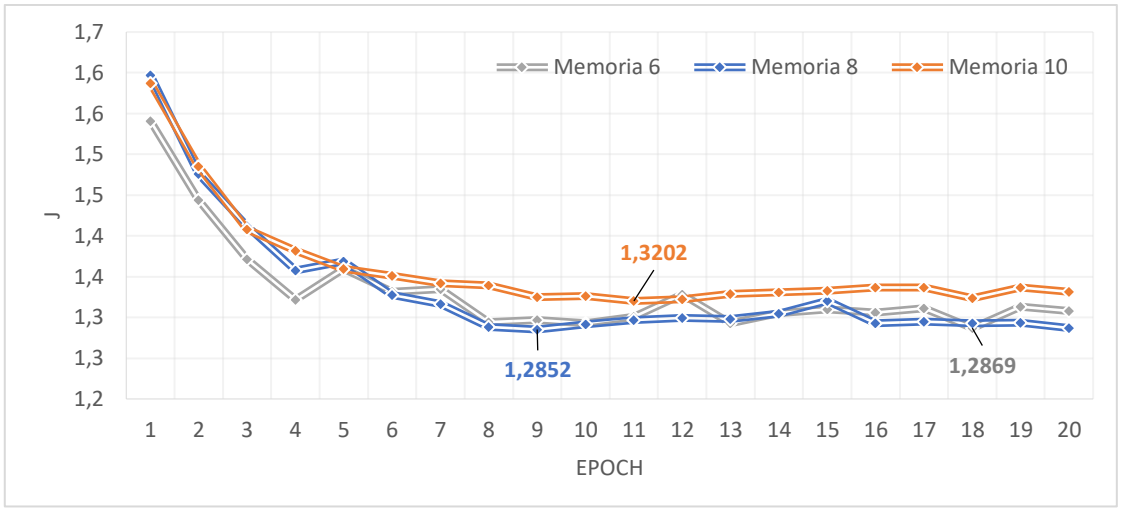

Figura 4.5. Valores de J con salto 2

En este desarrollo (figura 4.5), el valor de J sigue bajando, hasta 1,2852 y el mejor resultado se vuelve a tener cuando la memoria es igual a 8 por lo que se va a probar con todos los datos disponibles como se ha explicado en la figura 3.12 para aprovechar al máximo los datos que tenemos. Por tanto, este es el número de muestras que vamos a tener disponibles:

- Salto=1:
  - Memoria=6:
    - Muestras para entrenar: 98429
    - Muestras para test: 28000
    - Muestras para validación: 14000
  - Memoria=8
    - Muestras para entrenar: 96798
    - Muestras para test: 28000
    - Muestras para validación: 14000
  - Memoria=10
    - Muestras para entrenar: 95218
    - Muestras para test: 28000
    - Muestras para validación: 14000

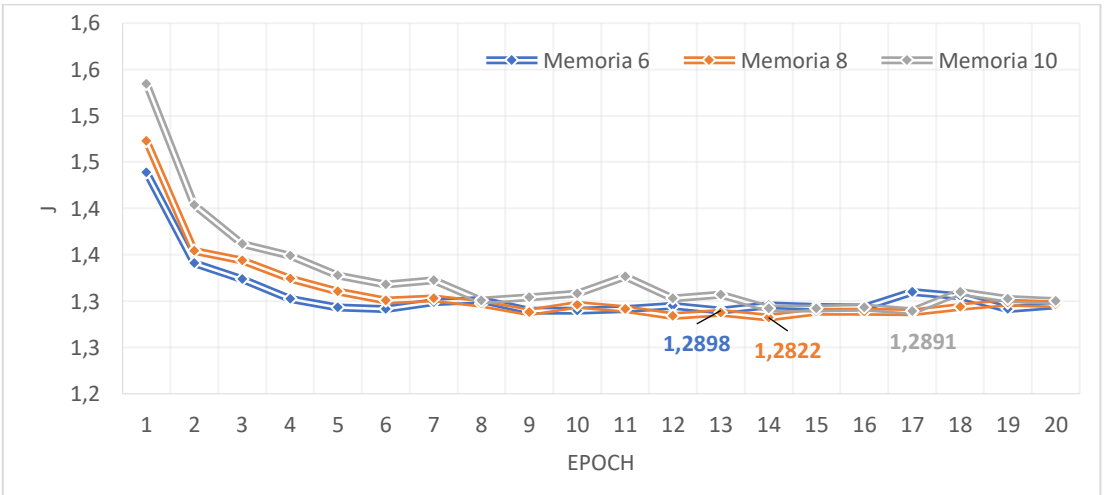

Figura 4.6. Valores de J con todos los datos

En este último ejemplo (figura 4.6), se puede observar que la J vuelve a descender, aunque mínimamente, siendo igual a 1,2822 cuando la memoria es igual a 8. Tras lo cual, se puede concluir que este desarrollo es el mejor modelo de red, ya que la red dispone de más datos para ser entrenada y se obtiene una J con el valor más bajo.

Una vez elegido este modelo se van a mostrar los resultados que se han obtenido. Para imprimir el valor que ha predicho la red se utiliza el código de la figura 4.7:

```
for i in range(4998):
    x = torch.from_numpy( x_test[i:i+1][:,:T] )
    out = model(x)
    out = out.argmax(1).item()
    print(out)
```

Figura 4.7. Imprimir salida de la red

Si se tiene una x test con una dimensión mayor a 5000, se debe ejecutar tantas veces este código como sean necesarias, puesto que el entorno no lo permite. Por ejemplo, si x test tiene una dimensión de 8000, la primera vez a ejecutarse será igual que la de la figura 4.2, y la segunda debemos cambiar al range(4998) por range(4998,8000) y así sucesivamente si fuese necesario más veces.

En lo que respecta al valor que debe predecir la red, se va a imprimir de la siguiente manera (figura 4.8):

```
for m in range(4998):
    numeroo=0
    for n in range(57):
      if (x_test[m][memoria-1][n]==1):
        numeroo=n
    print(numeroo)
```

Figura 4.8. Imprimir valor a predecir por la red

De la misma forma, si se tiene una x test con una dimensión mayor a 5000, vuelve a suceder lo mismo con el range del bucle. En el rango del segundo bucle se debe poner la dimensión de nuestra salida.

Una vez obtenidos ambos resultados se comparan y se obtiene el porcentaje de acierto obtenido. En este caso, para el modelo elegido (memoria=8, dimensión de capa oculta=64) se ha obtenido un porcentaje de acierto igual al 71% con un total de 19999 aciertos y 8001 fallos.

4.1.1 Información de usuario

Como se ha explicado en el apartado 3.2.1, se añade la información de usuario con el fin de intentar mejorar los resultados anteriores. Se sigue usando los mismos valores de memoria (8) y de dimensión de las capas ocultas (64) por lo que se dispone, como ya se ha comentado, de 96798 muestras para entrenar, 28000 para test y 14000 para validación.

4.1.1.1 No codificada

Primero se probó la red con la información sin codificar, esto supuso un aumento de la dimensión de entrada en 961, que es el número de usuarios que tratamos en la red. Se obtuvo el valor de J, igual que para los casos anteriores, sacando la media de 5 pruebas en 20 epochs, y se obtuvieron los resultados mostrados en la figura 4.9:

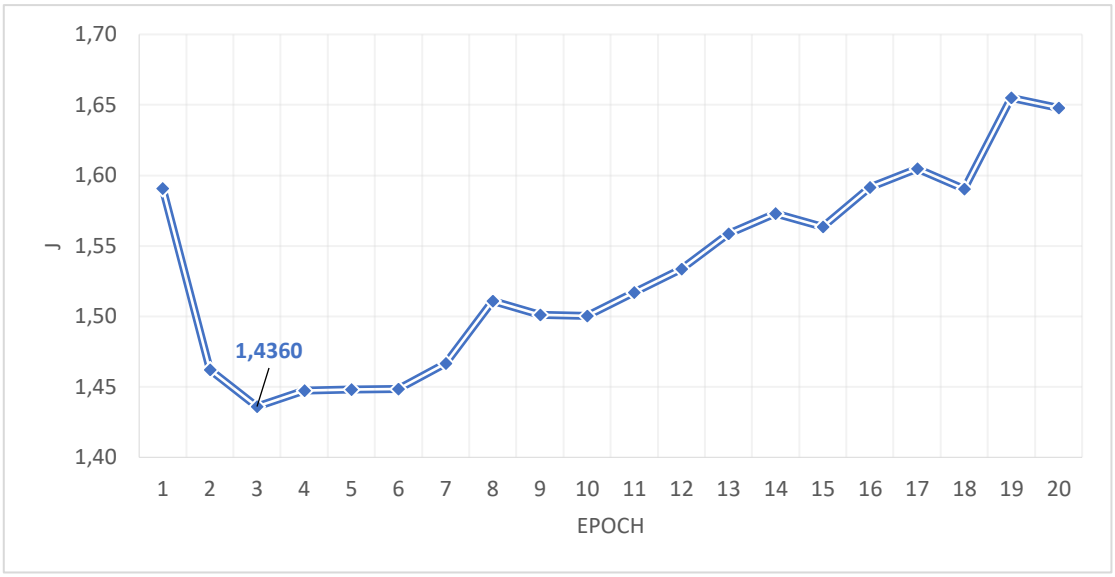

Figura 4.9. Valor de J con información de usuario no codificada

Como se puede observar en los resultados, el valor de J ha aumentado por lo que este modelo funciona peor que sin usar la información de usuario. Para comprobarlo, se imprime el resultado dado por la red y el valor que debería predecir de la misma manera que antes. Tras compararlos, se obtiene un 60% de acierto con 16689 aciertos y 11311 fallos, un acierto inferior que sin incluir la información de usuario.

4.1.1.2 Codificada

Tras esto, se pensó en la idea de incluir la información de usuario de manera codificada con el fin de reducir la dimensión de la entrada de la red. Para ello, se codificó la información en binario, tal y como se explica en el apartado 3.2.1.2. Con esto, la dimensión de entrada en lugar de aumentar en 961 solo aumenta en 10. Los valores de J que se obtienen son los siguientes:

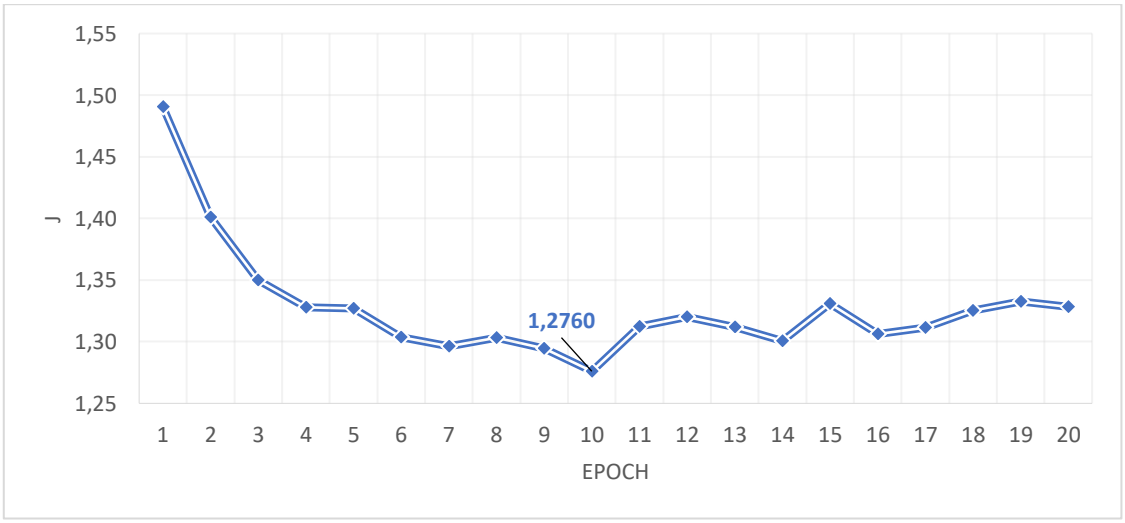

Figura 4.10. Valor de J con información de usuario codificada

Tal y como refleja la figura 4.10, el valor de J es el más pequeño obtenido hasta ahora, 1,2760. Esto indica que este es el mejor modelo de red hasta el momento. Si se comprueban los resultados tal y como se ha hecho anteriormente con las muestras de test se obtiene un 72% de acierto con 20170 aciertos y 7830 fallos.

Comparando estos dos resultados con el obtenido sin incluir la información de usuario estos serían los resultados (figura 4.11):

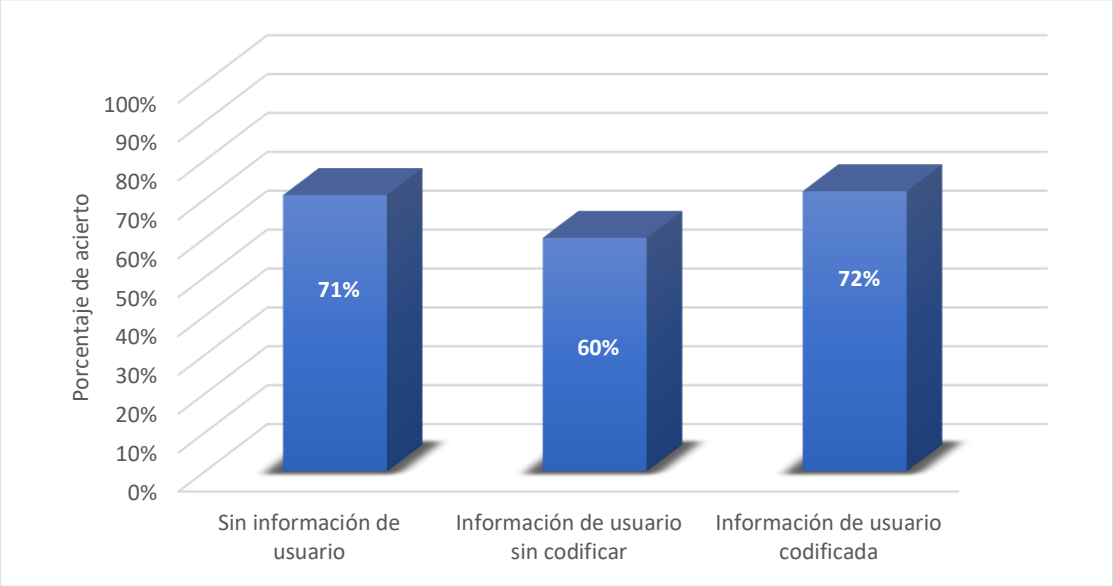

Figura 4.11. Porcentaje de acierto para distintos casos

Con esto se puede concluir que, en este escenario, añadir información propia de cada usuario de manera codificada produce una mejora en la predicción, aunque bastante limitada.

4.1.2 Predicción de varias estaciones

A continuación, se pensó en la posibilidad de predecir varias estaciones, aunque en este caso por simplicidad y viendo que la mejora ha sido limitada, no se va a utilizar información del usuario. El objetivo es ver el potencial de la red neuronal para predecir no solo la siguiente estación base, sino la trayectoria. Como se ha indicado en el apartado 3.3.2, se han realizado pruebas con predicciones desde dos hasta las seis estaciones siguientes.

En primer lugar, para cada uno de los distintos casos se obtiene el valor de J con las muestras de validación, tal y como se explica en el apartado 2.1.6. En todos los casos se usarán 10 epochs (figura 4.12):

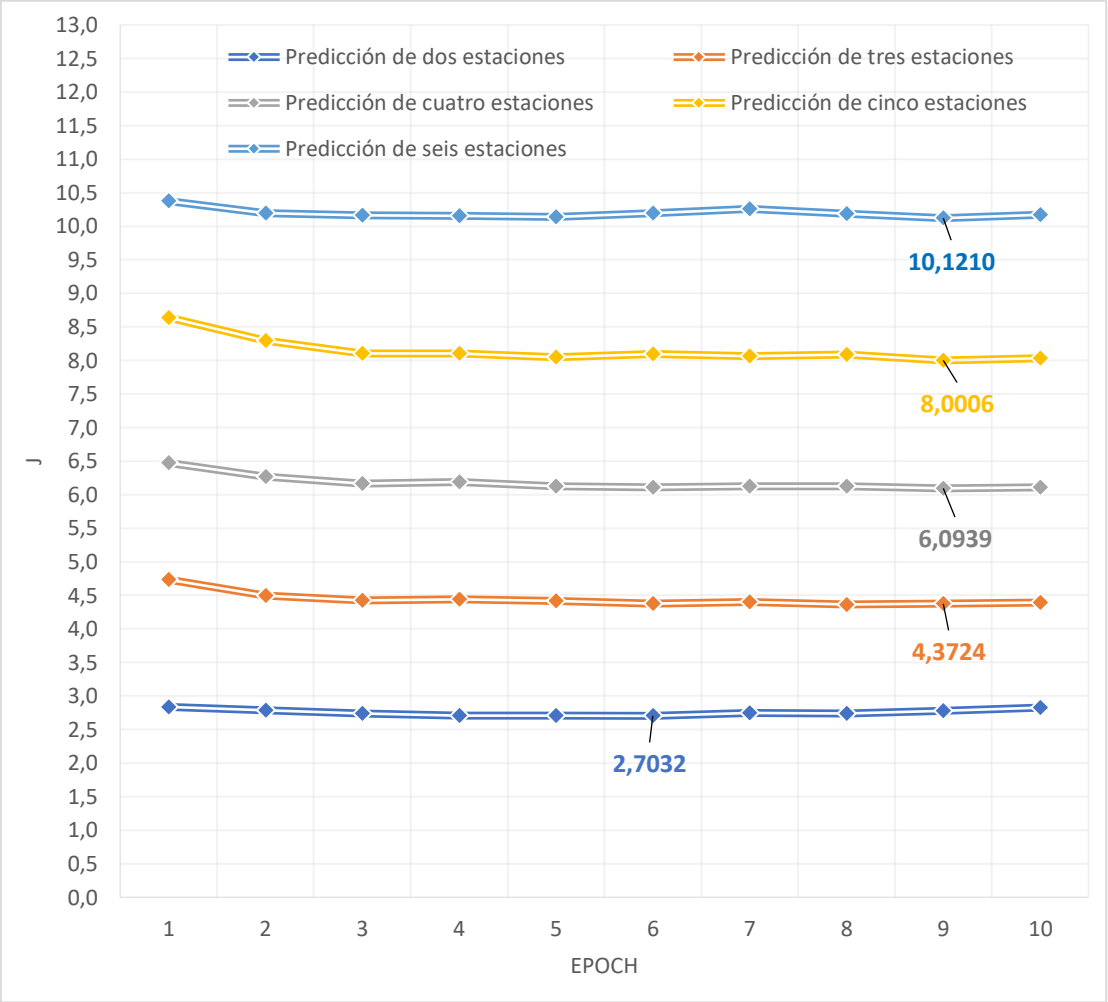

Figura 4.12. Valores de J para predecir varias estaciones

Como era de esperar, al estar sumando el valor de la función de pérdidas de ambas salidas, el valor de J es mayor a medida que aumenta el número de estaciones base. Como se observa en la figura 4.12, el valor de J mínimo se obtiene para dos estaciones base, como era de esperar, obteniendo un valor de 2,7032.

En lo que respecta a los resultados para dos estaciones base, se van a dividir en tres, los resultados de acertar solamente la primera estación, el porcentaje de acierto de la segunda estación independientemente del resultado de la primera y el porcentaje de acertar ambas. Para imprimir los resultados que después serán comparados se van a imprimir tal y como indican las figuras 4.13 y 4.14:

```
for i in range(0,4998):
    x = torch.from_numpy( x_test[i:i+1][:,:T] )
    out1 = model(x)
    out1 = out1[0].argmax(1).item()
    print(out1)
```

Figura 4.13. Imprimir valor dado por la red de la primera estación

```
for m in range(0,4998):
    numeroo=0
    for n in range(57):
      if (x_test[m][memoria-2][n]==1):
        numero=n
    print(numero)
```

Figura 4.14. Imprimir valor a predecir de la primera estación

Una vez obtenidos ambos valores, se comparan y se obtiene el porcentaje de acierto de la primera estación.

```
for i in range(0,4998):
    x = torch.from_numpy( x_test[i:i+1][:,:T] )
    out2 = model(x)
    out2 = out2[1].argmax(1).item()
    print(out2)
```

Figura 4.15. Imprimir valor dado por la red de la segunda estación

```
for m in range(0,4998):
    numero=0
    for n in range(57):
      if (x_test[m][memoria-1][n]==1):
        numero=n
    print(numero)
```

Figura 4.16. Imprimir valor a predecir de la segunda estación

Una vez obtenidos ambos valores (figuras 4.15 y 4.16), se comparan y se obtiene el porcentaje de acierto de la segunda estación. Por último, se comparan los aciertos entre ambos valores, tanto los de la primera como los de la segunda estación y obtenemos el porcentaje de acierto de ambas.

Este proceso se repetirá para el resto de los casos, teniendo en cuenta que a medida que se aumente el número de estaciones a predecir aumentará el número de resultados a comparar. Para ello se imprimen tantas salidas (out) como salidas tenga la red neuronal.

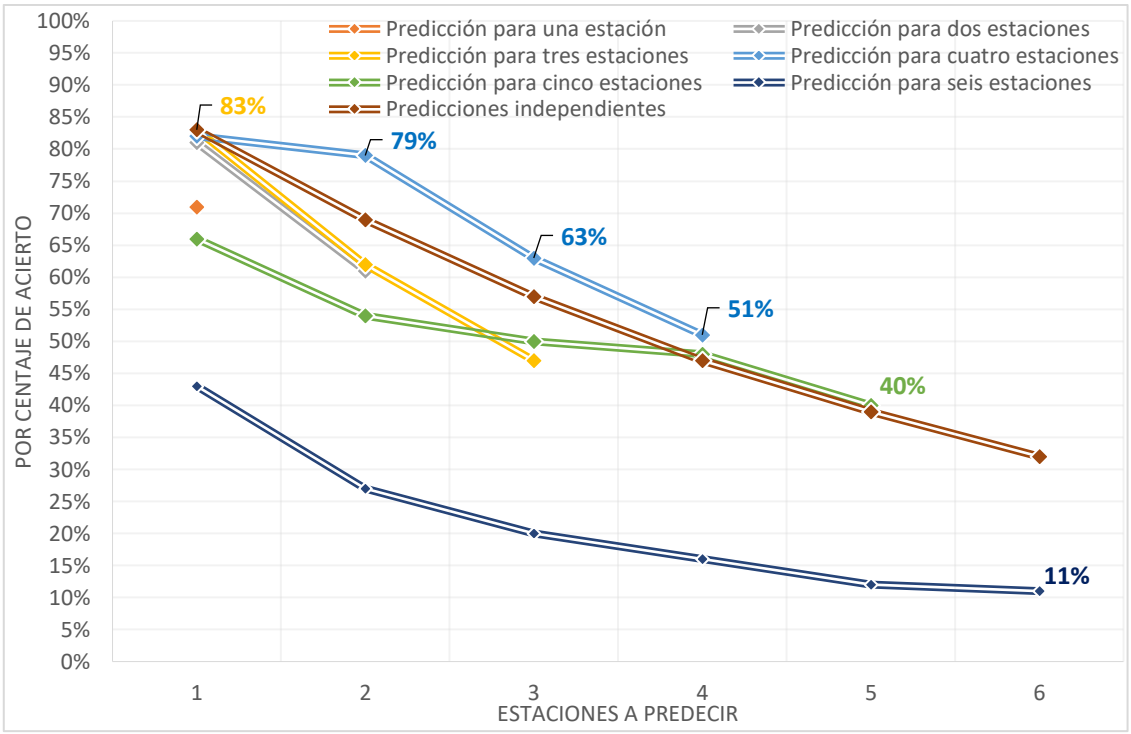

Figura 4.17. Porcentaje de acierto para predecir varias estaciones

En lo que respecta a los resultados (figura 4.17), lo primero que se observa es que existe una mejora importante en la probabilidad de acierto de la primera estación base siguiente cuando se intentan predecir las dos (82%), tres (83%) o cuatro (81%) estaciones base siguientes, frente a la que obtenemos si solo predecimos la siguiente (71%). Esta mejora puede ser debida a que la red es entrenada con datos del futuro en lo que respecta a la primera estación, dado que en el entrenamiento no solo se utiliza como salida cuál es la siguiente estación base, sino cuál es la siguiente o siguientes a la misma. Puede observarse también que la red comienza a degradar sus prestaciones cuando se intentan predecir las cinco estaciones base siguientes, lo que se confirma al intentar predecir las seis siguientes, donde se obtienen resultados muy inferiores a los casos anteriores.

En general, los mejores resultados se obtienen al predecir las cuatro siguientes estaciones base siguientes, con unos porcentajes de predecir correctamente una, dos, tres y cuatro estaciones base del 81%, 79%, 63% y 51% respectivamente.

Otra posible alternativa para predecir múltiples estaciones base consistiría en predecirlas de una en una de manera independiente, utilizando la predicción anterior como parte de la entrada a la red siguiente. Asumiendo que dichas predicciones son independientes, podríamos estimar la probabilidad de predecir correctamente n estaciones como p^n, siendo p la probabilidad de predecir correctamente la siguiente. Tomando como referencia el mejor valor posible (83%) hemos calculado teóricamente esas probabilidades, que se muestran en la gráfica como “predicciones independientes”. Puede comprobarse que la predicción de cuatro estaciones base, proporciona también mejores prestaciones que esta aproximación.

4.2 Modelos de Markov

Antes de comenzar a imprimir los resultados del modelo de Markov, tenemos que adaptar los datos para ello, así que hacemos lo siguiente (figura 4.18):

```
import math
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
print(estaciones)
ceros=np.zeros(estaciones.size)
x=list()

for k in range(n_usuarios):

  if (len(datos[k])>1):
    estacion_user=datos[k].drop(['Date/Time'],axis=1)
    estacion_user=np.asarray(estacion_user)

    for l in range(estacion_user.size-1):
      estacion1=estacion_user[l]
      prediccion=estacion_user[l+1]
      matriz=list()
      for m in range(estaciones.size):
        if (estaciones[m]==estacion1):
          ceros=np.zeros(57)
          ceros[m]=1
          matriz.append(ceros)
        if (estaciones[m]==prediccion):
          ceros2=np.zeros(57)
          ceros2[m]=1
          matriz.append(ceros2)
      x.append(matriz)

print(len(x))
```

Figura 4.18. Creación de x para modelo de Markov

Con esto se crea un array de elementos de longitud igual al número de datos disponible. Cada uno de los elementos es otro array con dos vectores, en el primero de ellos se guarda el vector de ceros y uno de la estación previa que vamos a usar para predecir la siguiente. En el segundo de los vectores guardaremos el vector de ceros y uno que el modelo debe predecir.

Una vez creada la x, utilizamos parte de los datos para test igual que se hacía para la red neuronal e imprimimos tanto la predicción que hace el modelo de Markov como la predicción que debería, tal y como indican las figuras 4.19 y 4.20:

```
x_test=x[115924:143924]
for k in range(0,4998):
    indice1=np.argmax(x_test[k][1])
    prediccion=np.argmax(markov[indice1])
    print(prediccion)
```

Figura 4.19. Imprimir predicción hecha por el modelo de Markov

```
x_test=x[115924:143924]
for k in range(0,4998):
  valor=np.argmax(x_test[k][0])
  print(valor)
```

Figura 4.20. Imprimir valor que debería predecir el modelo de Markov

Una vez se hayan impreso los valores, se comparan y se obtiene un porcentaje de acierto del 47% con 13292 aciertos y 14708 errores.

Por último, se calcula la validación, para ello se utilizarán el 10% del total de las muestras disponibles, tal y como se ha hecho con la red neuronal, y se sumarán las probabilidades de transición de estos registros, obteniendo un valor de validación igual a 7551.61; dato que posteriormente será comparado con el del resto de modelos de Markov para verificar cuál es el más adecuado.

A continuación, tal y como se ha explicado en el apartado 2.3 se van a añadir estados al modelo de Markov de manera que cada estado guarde las dos estaciones anteriores tal y como se explica en el apartado 3.4. En este caso, la creación de x queda de la siguiente manera (figura 4.21):

```
import math
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
print(estaciones)
ceros=np.zeros(estaciones.size)
x=list()

for k in range(n_usuarios):

  if (len(datos[k])>3):
    estacion_user=datos[k].drop(['Date/Time'],axis=1)
    estacion_user=np.asarray(estacion_user)

    for l in range(estacion_user.size-3):
      estacion1=estacion_user[l]
      estacion2=estacion_user[l+1]
      prediccion=estacion_user[l+2]
      prediccion2=estacion_user[l+3]
      resta=0
      matriz=list()
      for m in range(estaciones.size):
        if (estaciones[m]==estacion1):
          ceros=np.zeros(57)
          ceros[m]=1
          matriz.append(ceros)
        if (estaciones[m]==estacion2):
          ceros2=np.zeros(57)
          ceros2[m]=1
          matriz.append(ceros2)
        if (estaciones[m]==prediccion):
          ceros3=np.zeros(57)
          ceros3[m]=1
          matriz.append(ceros3)
        if (estaciones[m]==prediccion2):
          ceros4=np.zeros(57)
          ceros4[m]=1
          matriz.append(ceros4)
      x.append(matriz)

print(len(x))
```

Figura 4.21. Creación de x para modelo de Markov con dos estaciones previas

Por último, imprimimos tanto el valor dado por la red como la predicción que debe dar el modelo según las figuras 4.22 y 4.23:

```
x_test=x[114914:143014]
for k in range(0,4998):
    indice1=np.argmax(x_test[k][1])
    indice2=np.argmax(x_test[k][2])
    prediccion=np.argmax(markov[indice1][indice2])
    print(prediccion)
```

Figura 4.22. Imprimir predicción hecha por el modelo de Markov con dos estaciones

```
x_test=x[114914:143014]
for k in range(0,4998):
  valor=np.argmax(x_test[k][0])
  print(valor)
```

Figura 4.23. Imprimir valor que debería predecir el modelo de Markov con dos estaciones

Una vez obtenidos ambos resultados se comparan y se obtiene un porcentaje de acierto del 56% con 15681 aciertos y 12319 fallos.

En lo que respecta al valor de validación, se calcula igual que para el caso anterior y se obtiene un total de 9559,69; obteniendo un valor mejor que para el modelo previo, lo que valida que el porcentaje de acierto en la evaluación sea mayor.

Al obtener un mayor valor de validación, se volvió a aumentar el número de estados del modelo de manera que cada estado almacene las tres estaciones anteriores a la que se quiere predecir. Para este caso, la creación de x quedaría de la siguiente manera (figura 4.24):

```
import math
estaciones=columna_localizacion.unique()
estaciones=np.sort(estaciones)
estaciones=np.asarray(estaciones)
print(estaciones)
x=list()

for k in range(n_usuarios):

  if (len(datos[k])>3):
    estacion_user=datos[k].drop(['Date/Time'],axis=1)
    estacion_user=np.asarray(estacion_user)

    for l in range(estacion_user.size-3):
      estacion1=estacion_user[l]
      estacion2=estacion_user[l+1]
      estacion3=estacion_user[l+2]
      prediccion=estacion_user[l+3]
      matriz=list()
      for m in range(estaciones.size):
        if (estaciones[m]==estacion1):
          ceros=np.zeros(57)
          ceros[m]=1
          matriz.append(ceros)
        if (estaciones[m]==estacion2):
          ceros2=np.zeros(57)
          ceros2[m]=1
          matriz.append(ceros2)
        if (estaciones[m]==estacion3):
          ceros3=np.zeros(57)
          ceros3[m]=1
          matriz.append(ceros3)
        if (estaciones[m]==prediccion):
          ceros4=np.zeros(57)
          ceros4[m]=1
          matriz.append(ceros3)
      x.append(matriz)

print(len(x))
```

Figura 4.24. Creación de x para modelo de Markov con tres estaciones previas

Por último, al igual que en los anteriores casos imprimimos tanto el valor dado por la red como la predicción que debe dar el modelo tal y como indican las figuras 4.25 y 4.26:

```
x_test=x[114145:142145]
for k in range(0,4998):
    indice1=np.argmax(x_test[k][1])
    indice2=np.argmax(x_test[k][2])
    indice2=np.argmax(x_test[k][3])
    prediccion=np.argmax(markov[indice1][indice2][indice3])
    print(prediccion)
```

Figura 4.25. Imprimir predicción hecha por el modelo de Markov con dos estaciones

```
x_test=x[114145:142145]
for k in range(0,4998):
  valor=np.argmax(x_test[k][0])
  print(valor)
```

Figura 4.26. Imprimir valor que debería predecir el modelo de Markov con tres estaciones

Comparando los valores obtenidos en ambos, se consigue un 50% de acierto con 14077 aciertos y 13923 fallos.

Por último, se calcula el valor de validación de la misma manera que para el resto de los desarrollos y se obtiene un valor de 9327,43, concluyendo así que el mejor modelo de Markov para este escenario es aquel en el que cada uno de los estados se compone de las dos estaciones previas.

Comparando los tres resultados obtenidos con el modelo de Markov, estos serían los resultados (figura 4.27):

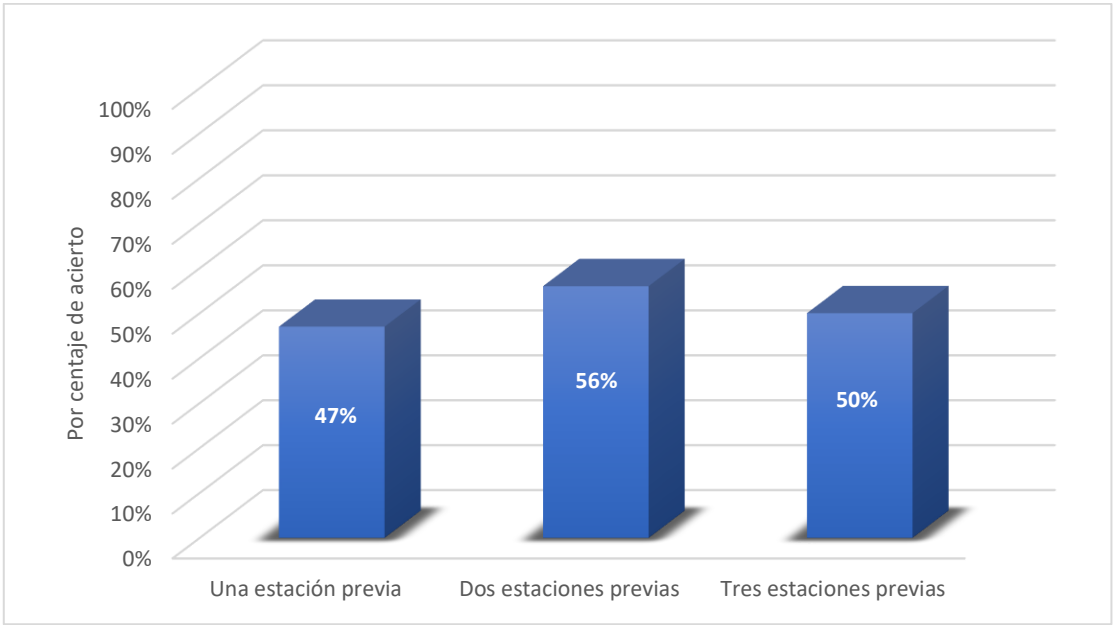

Figura 4.27. Resultados con modelo de Markov

Como puede comprobarse, el porcentaje de acierto en el mejor caso (56%) es claramente inferior al obtenido con las redes neuronales, tanto en el caso de predicción solo de la siguiente (71%) como más aún en los siguientes (81%, 83% y 82% al predecir las 2, 3 y 4 siguientes, respectivamente). Estos resultados confirman que la mejora de prestaciones obtenida mediante una red neuronal recurrente es considerable, respecto a aproximaciones más sencillas y clásicas como el modelo de Markov [19].

5. Conclusiones y futuros avances

En este capítulo se analizarán las conclusiones del proyecto y si los objetivos han sido cumplidos. Además, se comentarán futuros avances que se pueden hacer a partir de este proyecto.

5.1 Conclusiones

- Se ha comprobado el potencial de las redes neuronales para predecir el movimiento humano dentro de una red celular móvil, ya que con la red neuronal propuesta inicialmente se ha obtenido un 71% de acierto en la predicción de la siguiente estación base.
- A partir de una red neuronal básica, se puede mejorar su porcentaje de acierto, mediante la modificación de los datos de entrada de la red añadiendo información extra, o modificando el código de la red para predecir varios movimientos consecutivos de los usuarios, llegando en algunos casos a tener un 83% de acierto en la predicción de la siguiente estación base y cumpliendo así el primer objetivo que se había planteado al comienzo de este proyecto.
- Se ha comprobado que las prestaciones de un modelo de Markov son claramente inferiores a las que proporciona la red neuronal, puesto que el mejor porcentaje de acierto obtenido con este modelo ha sido del 56% frente al 71% obtenido con la red neuronal básica. Por tanto, se puede dar por cumplido el segundo de los objetivos del proyecto.

5.2 Futuros avances

Este proyecto podría ser optimizado si la empresa correspondiente pudiera facilitarnos datos con los siguientes requisitos:

- Conocimiento de la posición geográfica de las estaciones base. Si fuese posible conocer la posición de cada una de las estaciones base, la red podría aprender más rápido los posibles movimientos de los usuarios.
- Aplicación a otras redes. El mismo funcionamiento de la red neuronal usado en este proyecto podría ser usado en otra red de la misma manera que se ha hecho en esta y se podría ver su nivel de adaptabilidad.
- Utilizar nueva información. Al igual que se ha usado el identificador del usuario con el fin de mejorar los resultados de la red neuronal, se podrían usar otros datos como la fecha y la hora para datos más prolongados en el tiempo, por ejemplo, que duren varios años. De esta forma se podría predecir qué días de la semana o qué meses del año (períodos vacacionales o eventos puntuales) deberíamos usar más recursos en una determinada estación base.

Además, en esta misma línea de trabajo y con la información que se dispone actualmente, también se podría intentar predecir no sólo cuál es la siguiente estación base, sino cuánto tiempo falta para que se produzca el cambio.

BIBLIOGRAFÍA

[1] W. McCulloch, W. Pitts, "Un Cálculo Lógico de la Inminente Idea de la Actividad Nerviosa" Boletín de Matemática Biofísica 5: 115-133.

[2] F.J. Gómez Quesada, M.A. Fernández Graciani, M. T. López Bonal, M. Alonso Díaz-Mata "Aprendizaje con redes neuronales artificiales". Revista de la Facultad de Educación de Albacete, ISSN 0214-4842, ISSN-e 2171-9098, Nº. 9, págs. 169-180, 1994.

[3] A. Ng, "Curso de aprendizaje automático" de Coursera [En línea]. Disponible en: https://www.coursera.org/learn/machine-learning

[4] E. Davalo, P. Naim, Neural Networks (Computer Science Series), Ed. Scholium Intl, 1991.

[5] R. B. Mitchell, C. Rapkin, "Urban Traffic - A Function of Land Use" New York 27, Columbia University Press, 1954.

[6] P. J. Rodríguez-Rueda, I. J. Turias "Una comparativa entre redes neuronales artificiales y métodos clásicos para la predicción de la movilidad entre zonas de transporte." DYNA revista de la Facultad de Minas. Universidad Nacional de Colombia. Sede Medellín. vol. 84, nº. 200, pp. 209-216, 2017.

[7] H. Zhang, L. Dai, "Mobility Prediction: A Survey on State-of-the-Art Schemes and Future Applications" , IEEE Access, vol. 7, pp. 802-822, diciembre 2018.

[8] Página de bienvenida de Google Colaboratory [En línea] Disponible en: https://colab.research.google.com/notebooks/welcome.ipynb?hl=es#scrollTo=-Rh3-Vt9Nev9

[9] Página de bienvenida de Jupyter Network. [En línea] Disponible en: https://jupyter.org/

[10] Página de bienvenida de Python. [En línea] Disponible en: https://www.python.org/

[11] "¿Qué es Pytorch?". [En línea] Disponible en: https://cleverpy.com/que-es-pytorch-y-como-se-instala/

[12] S. Gambs, M.-O. Killijian, M. Núñez del Prado Cortez, "Next place prediction using mobilitiy Markov chains", MPM '12: Proceedings of the First Workshop on Measurement, Privacy, and MobilityApril 2012, abril, 2012.

[13] M. Canales, J. R. Gállego, Á. Hernández, A. Valdovinos, “An adaptive location management scheme for mobile broadband cellular systems”, Telecommunication Systems, vol. 52, pp. 299-315, 2013.

[14] M. Strefezza Blanco, Y. Dote, “Algoritmo de aprendizaje acelerado para backpropagation”, Simposio brasileño de automatización inteligente, São Paulo, SP, 08-10 de septiembre de 1999

[15] P. Diederik, J. Kingma, B Lei, “ADAM: A method for stochastic optimization”, 3rd International Conference on Learning Representations, ICLR 2015, mayo 2015.

[16] J. A. Pérez Ortiz, “Modelos predictivos basados en redes neuronales recurrentes de tiempo discreto”, Tesis doctoral, Universidad de Alicante, julio 2002.

[17] Besay Montesdeoca Santana, “Estudios de predicción en series temporales de datos meteorológicos utilizando redes recurrentes”, Trabajo Final de Grado, Universidad de las Palmas de Gran Canaria, julio 2016.

[18] R. Ocaña-Riola “Modelos de Markov aplicados a la investigación en ciencias de la salud”, Interciencia, vol. 34, nº 3, marzo, 2009

[19] V. Kulkarni, A. Mahalunkar, B. Garbinato, J. D. Kelleher “On the Inability of Markov Models to Capture Criticality in Human Mobility”, International Conference on Artificial Neural Networks ICANN 2019: Artificial Neural Networks and Machine Learning – ICANN 2019: Image Processing, pp 484-497.