Grado en Estadística

TRABAJO FIN DE ESTUDIOS

Redes recurrentes

Fernando Rodríguez Fenoy

Sevilla, Julio de 2023

Tutorizado por: María Dolores Cubiles de la Vega

Índice general

Resumen

Las redes neuronales recurrentes, aunque fueron concebidas en el siglo pasado, han ganado mucho protagonismo en los últimos años gracias a los avances tecnológicos que se han producido. Su habilidad para modelar secuencias encontrando patrones que se repiten dentro de éstas, ha hecho que este tipo de red neuronal se aplique en una gran diversidad de tareas que todos conocemos y usamos, como puede ser traducir un texto o pronosticar el clima de una ciudad.

El objetivo de este trabajo ha sido introducir estas redes recurrentes. En primer lugar repasando a qué campo de la ciencia pertenecen, después explicando el funcionamiento de una red neuronal, para finalmente profundizar en nuestro tema del trabajo. Para ello, primero se ha indagado en la teoría que lleva detrás y después hemos terminado realizando una aplicación práctica utilizando el lenguaje de programación R junto con la librería Keras, que consistía en realizar una predicción de la temperatura del aire en una ciudad para catorce días, con ayuda de un histórico de datos acerca de la presión atmosférica, la humedad o la temperatura, entre otras más variables.

Los resultados de esta práctica nos llevaron a concluir que cuando tenemos datos secuenciales, es decir, que siguen un orden específico y que por tanto tiene sentido analizarlos de forma conjunta, el aplicar redes recurrentes puede ser una gran opción para capturar patrones que se repitan en el tiempo y que los datos sean tratados teniendo en cuenta el contexto en el que se presentan.

Abstract

Recurrent neural networks, although they were conceived in the last century, they have gained much prominence in recent years thanks to the technological advances that have taken place. Their ability to model sequences by finding repeating patterns within them has led to the application of this type of neural network in a wide range of tasks that we all know and we use, such as translating a text or forecasting the weather in a city.

The aim of this work has been to introduce these recurrent networks. First of all, by reviewing the field of science to which they belong, then explaining how a neural network works, and finally going deeper into the subject of our work. To do this, we first explored the theory behind it and then we ended up carrying out a practical application using the R programming language together with the Keras library, which consisted of making a prediction of the air temperature in a city for fourteen days, with the help of a history of data on atmospheric pressure, humidity and temperature, among other variables.

The results of this practice led us to conclude that when we have sequential data, that is, data that follow a specific order and therefore it makes sense to analyse them together, the application of recurrent networks can be a great option to capture patterns that repeat over time and that the data are treated taking into account the context in which they are presented.

Índice de figuras

Capítulo 1

Introducción

En estos tiempos que corren, se ha normalizado que haya coches que se conducen solos, dispositivos móviles que se desbloquean con sólo mostrar nuestra cara, asistentes de chat virtuales que parecen saberlo todo,... pero claro, ¿qué hay detrás de todos estos procesos que hasta hace no muchos años parecían indispensables? El Deep Learning o Aprendizaje profundo.

Puede que éste no sea un término demasiado popular, pero si hablamos de Inteligencia Artificial, éste se encuentra en los últimos años en boca de todos. Y esto no es casualidad, ya que muchos de los avances tecnológicos más recientes están llegando de la mano de esta disciplina.

Cabe mencionar que existe un campo intermedio entre la inteligencia artificial y el aprendizaje profundo, que es el Machine Learning (ML) o Aprendizaje Automático, del que también hablaremos.

Para dar una idea de cómo se relacionan entre sí estos tres campos mencionados, vendrá muy bien esta representación:

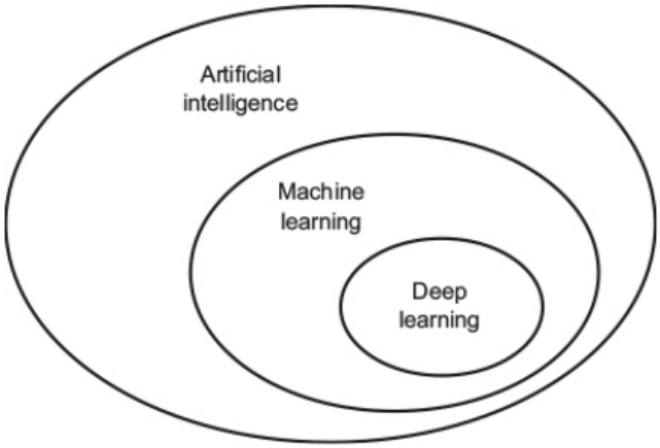


Figura 1.1: Inteligencia artificial, Aprendizaje automático y Aprendizaje profundo

Un tema que va a tomar bastante importancia a lo largo de todo el trabajo es el aprendizaje, ya que es una fase crucial en todo modelo del que acabemos hablando.

Para el aprendizaje de cualquier modelo, sea DL o ML, necesitamos:

- Datos de entrada
- Salidas respectivas esperadas
- Forma de medir el rendimiento del modelo: se usará para ajustar el funcionamiento de éste, y es lo más importante ya que lo que llamamos aprendizaje ocurrirá en base a esta medida

Decimos que un modelo ha aprendido a realizar una tarea cuando después de haber observado unos datos de entrada con sus salidas respectivas esperadas, es capaz de resolverla introduciendo nuevos datos en la entrada.

Este aprendizaje se consigue buscando transformaciones en los datos de entrada de forma que nos quedemos con las que mejor nos ayuden a resolver la tarea fácilmente.

Por tanto, el objetivo siempre va a ser buscar las mejores transformaciones para los datos de forma que éstos nos ayuden a resolver la tarea más fácilmente.

Hasta ahora, se ha hablado del aprendizaje desde el punto de vista del Machine Learning. Si nos vamos a Deep Learning, lo que lo hace especial es que su aprendizaje se realiza de una forma más avanzada ya que aprende a partir de sucesivas capas de representación cada vez más significativas. Es decir, cada capa de representación va diferenciando mejor los datos de entrada hasta que consigue realizar la tarea correctamente.

1.1. Inteligencia Artificial

La Inteligencia Artificial (IA) es una disciplina que tiene como objetivo hacer que los ordenadores piensen y resuelvan problemas por sí mismos como hacemos los humanos, y que de esa forma por tanto presenten una "inteligencia" que les permita tomar buenas decisiones.

Cabe destacar que se trata de un campo enorme, y que por consiguiente abarca muchos más temas aparte de los que se van a tratar en este trabajo, que no están ligados a ningún tipo de aprendizaje como puede ser el Procesamiento del Lenguaje Natural.

La IA no es algo novedoso que haya sido descubierto recientemente, sino que sus inicios se remontan a los años 50 del siglo pasado. Aunque desde un primer momento tuvo mucha repercusión, nada puede compararse a la popularidad y el desarrollo que ha obtenido estos últimos veinte años. Esto se debe a los grandes avances tecnológicos que se han dado en los ordenadores, que hacen posible el procesamiento y almacenamiento de grandes volúmenes de datos. Sumado al interés por las empresas de incorporar IA en sus negocios, y las mejoras en cuanto a la infraestructura de red global, todo ha hecho que este campo sea conocido por todos a día de hoy.

1.2. Machine Learning

El Machine Learning (ML, Aprendizaje Automático) es un campo que se encuentra dentro de la IA que se centra en desarrollar modelos estadísticos a partir de unos datos

de entrenamiento. Se les llama modelos de aprendizaje porque éstos resultan del entrenamiento de un algoritmo de aprendizaje automático.

Surge de la siguiente pregunta: "¿puede un ordenador sorprendernos?". Esta pregunta proviene de que antes del ML, el paradigma de programación consistía en que una persona programase a mano lo que quisiera que el ordenador hiciera con unos datos de entrada (IA simbólica), pero con el Machine Learning llegaba un nuevo paradigma: que el ordenador aprenda a realizar una tarea con sólo mirar los datos y sin ayuda de ninguna persona. Es decir, los humanos introducen unos datos con sus salidas esperadas, y a raíz de esto detecta patrones que le permiten realizar esa tarea con datos distintos.

Esta disciplina puede considerarse una fusión entre la Estadística y la Ciencia Computacional.

La Estadística la usa a la hora de buscar los "patrones" en los datos que permitirán al modelo realizar las predicciones, de forma que esos patrones no son más que una estructura estadística. La ciencia computacional aporta al Machine Learning los diferentes lenguajes de programación, las bases de datos y por supuesto los algoritmos de aprendizaje y técnicas de optimización.

1.3. Deep Learning

El Deep Learning (DL, Aprendizaje Profundo) es una rama del ML que se centra en construir algoritmos que expliquen y aprendan, a diferencia del resto de técnicas de ML, con un alto y bajo nivel de abstracción de los datos.

El "deep" hace alusión a que se trata de técnicas que hacen uso de redes neuronales con muchas capas ocultas. Se les llama ocultas porque no necesariamente se conocen las entradas y salidas de las neuronas que forman las capas intermedias.

El objetivo del DL es que sus algoritmos necesiten menos instrucciones que los de ML, es decir, menos intervención de los humanos, de forma que sean más autónomos.

A destacar su aplicación en problemas de aprendizaje no supervisado, entre otros.

Capítulo 2

Redes de neuronas

Una red de neuronas artificial (RNA) es un modelo computacional basado en una representación simplificada del funcionamiento del sistema nervioso humano.

Una neurona artificial (nodo) es la unidad básica de procesamiento, y su apilamiento forman las capas de las redes neuronales, que luego se pasarán a explicar.

Durante los últimos años, las redes neuronales han tenido un crecimiento exponencial en cuanto a su utilidad, pero muchas de las técnicas que deslumbran ahora, como pueden ser el algoritmo LSTM, fueron desarrolladas en el pasado.

Aunque la teoría se tenía, entre el año 1990 y el 2000 el problema consistía en que la calidad de los datos y el hardware del momento hacían imposible que se produjesen más avances, y por ello se produjo un estancamiento. No debe olvidarse que este modelo forma parte del Machine Learning, y el desarrollo de éste es directamente dependiente del de la ingeniería.

Las redes neuronales se utilizan a día de hoy para realizar una multitud de tareas distintas, entre ellas:

- Para problemas en los que no sea necesaria una interpretación de los resultados, sólo su obtención.
- Problemas en los que el tiempo de entrenamiento del modelo no sea excesivamente limitado, ya que éste suele ser duradero debido a la gran cantidad de datos que se demandan para alcanzar un correcto desempeño.
- Tareas en las que tengamos datos no estructurados como pueden ser las de reconocimiento visual o reconocimiento de voz en los que se trata con imágenes y vídeos.

2.1. Comparación con las redes de neuronas biológicas

Aunque las redes neuronales son un tema de actualidad, las primeras ideas surgieron en los años 50. Pero debido a la falta de herramientas para entrenar estas redes, no fue hasta los 80, con el redescubrimiento del algoritmo de retropropagación (backpropagation en inglés) y su aplicación en estas redes, cuando empezó a despegar esta idea.

Explicamos de forma simple cómo funciona una neurona biológica.

Nuestras neuronas se dividen en:

- Dendritas: se encargan de captar los impulsos nerviosos que reciben de otras neuronas.
- Soma: lugar en el que se procesan estos impulsos para luego ser transmitidos al axón.
- Axón: emite el impulso nervioso hasta otras neuronas próximas.

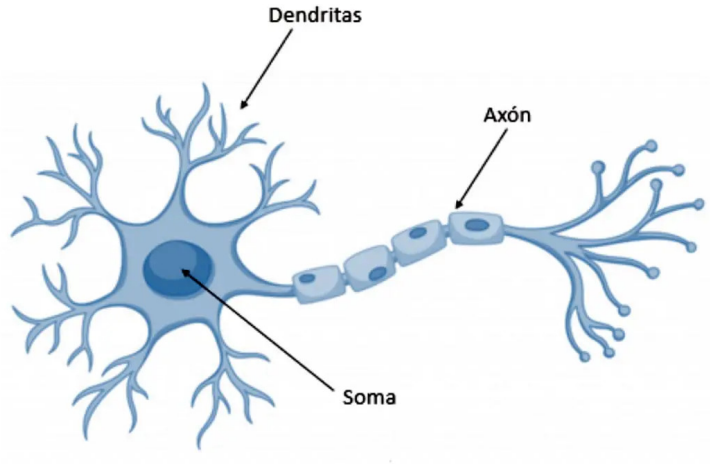


Figura 2.1: Representación de una neurona biológica

Podríamos hacer entonces las siguientes comparaciones:

- Los datos de entrada que se le suministran a una neurona artificial podríamos entenderlos como los impulsos nerviosos que recibe la neurona biológica.
- La función del soma, que es procesar estos "datos" para luego transmitirlos si se sobrepasa un determinado umbral, podría verse como la función de activación que recibe los datos y luego transmite una salida.
- El axón se puede comparar con el momento en el que una neurona artificial devuelve unos datos y los transmite a otra neurona.

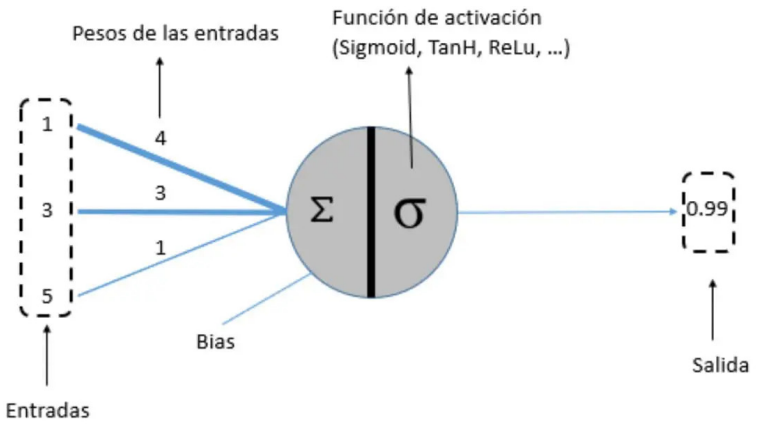


Figura 2.2: Representación de una neurona artificial

2.2. Componentes de una RNA

- **Capas**

Antes de definir lo que es una capa, es preciso saber lo que es un tensor. Un tensor es una generalización de vectores y matrices a un número arbitrario de dimensiones, y es donde se almacenan los datos. La dimensión de un tensor indica el número de índices necesarios para especificar todos sus componentes.

Una capa es un módulo de procesamiento de datos que obtiene de entrada uno o más tensores y devuelve uno o más tensores. Con otras palabras, es una función que transforma todos los datos que pasan por ella, por lo que podría decirse que actúa como un filtro.

Estas transformaciones están parametrizadas por unos pesos que determinan los cambios que se le aplican a los datos, por tanto, en este contexto el aprendizaje podría verse como la búsqueda de la mejor combinación de pesos que hace más fácil la realización de la tarea.

Se trata de la estructura de datos más importante dentro de una red neuronal, ya que las neuronas se encuentran organizadas en distintas capas que a su vez están unidas entre sí.

Según el lugar en el que se localicen en la red, hay tres tipos:

- Capa de entrada o “input layer”
- Capa oculta o “hidden layer”
- Capa de salida o “output layer”

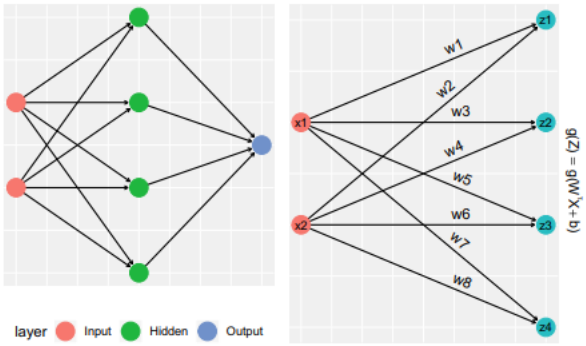


Figura 2.3: Representación de las distintas capas de una red neuronal

- **Red**

La topología de la red es un aspecto muy diferencial ya que configura la forma en que los distintos nodos o neuronas están conectados. La más simple es la que tiene una entrada y una salida, pero las hay también con dos o más ramificaciones separadas que acaban combinándose en algún momento posterior.

- **Función de pérdida**

La función de pérdida, también conocida como "función objetivo" o "loss function", se encarga de medir la diferencia entre las predicciones del modelo y los valores reales. Esto ayuda a obtener un feedback acerca del rendimiento del modelo.

El objetivo de cualquier algoritmo siempre va a ser minimizar esta función, porque cuanto menor es el resultado, más eficiente es la red.

La red tendrá tantas funciones de pérdida como salidas, por tanto se calculará una media entre todas ellas que se usará posteriormente en el proceso del gradiente descendente (se explicará en los siguientes apartados). Aunque el algoritmo de descenso de gradiente es el más conocido, cabe resaltar que podrían usarse otros procesos en lugar de éste que nos permitiesen encontrar el mínimo para la función de pérdida.

La elección de la función de pérdida es muy importante en el modelo, ya que ésta debe ir acorde a la tarea que queremos que realice el modelo. Si esto no se cumple, los resultados no serán satisfactorios.

- **Optimizador**

Para minimizar la función de pérdida necesitamos de un algoritmo de optimización, que es el que se encarga de actualizar los pesos de la red conforme a los resultados que se obtengan en la función de pérdida. Implementa alguna variante del algoritmo del Gradiente Descendente.

Algoritmo del Gradiente Descendente

Éste es esencial, puesto que es la base de todos los algoritmos de entrenamiento de modelos de ML (redes neuronales, convolucionales, recurrentes,...).

Permite encontrar de forma automática el mínimo de una función. Para ello usa el gradiente, que no es más que la derivada de dicha función. A partir de esas derivadas, se va moviendo en dirección contraria al gradiente multiplicado por una tasa de aprendizaje. De esta forma, en cada iteración va acercándose al mínimo global de la función.

2.3. Funcionamiento general de una red de neuronas

Dentro de nuestra red de neuronas, éstas se encuentran organizadas por capas. Cada neurona está unida con otras a través de conexiones, que se representan en la figura con flechas y tienen una dirección establecida. Esto estaría modelando la forma en que el axón de una neurona se conecta con las dendritas de otra.

Para cada conexión, por ejemplo entre una neurona "a" y otra "b", hay un peso asociado que puede representarse como w_{ab}. Este peso puede tomar valores tanto negativos como positivos.

Cuando la neurona "a" envía una salida hasta la neurona "b", esta señal se manda a través de la conexión entre ambas de forma que esta salida acaba siendo la entrada para la neurona "b".

Respecto a cómo se genera esa salida, ésta depende de las entradas que recibe y sus respectivos pesos.

Resumiendo, la red recibe unas entradas del exterior y después de sucesivas trasformaciones procede a devolver otras salidas en relación con la tarea que se esté abordando.

Por último, si nos centramos en la fase de aprendizaje de la red, ésta consiste en encontrar los pesos asociados a las conexiones entre neuronas de forma que la red acabe actuando de forma similar a como lo hizo para un conjunto de entrenamiento anteriormente dado.

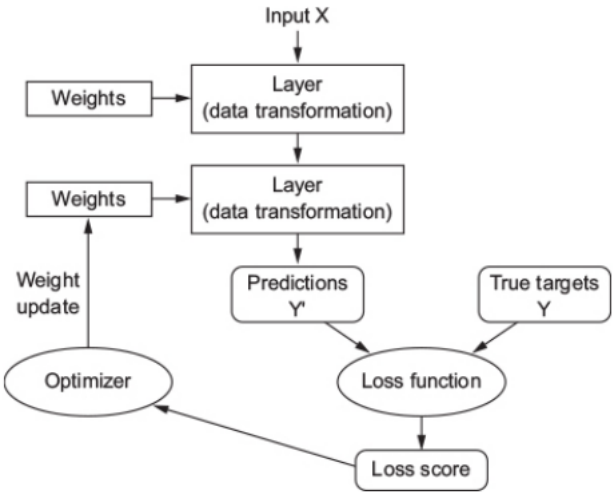


Figura 2.4: Funcionamiento de una red de neuronas

2.4. Modelo del perceptrón simple (SLP) y modelo del perceptrón multicapa (MLP)

2.4.1. SLP

El modelo del perceptrón unicapa, creado por Frank Rosenblatt en 1959, es la representación más simple de lo que es una red neuronal.

Podemos apreciar sus componentes en la siguiente representación:

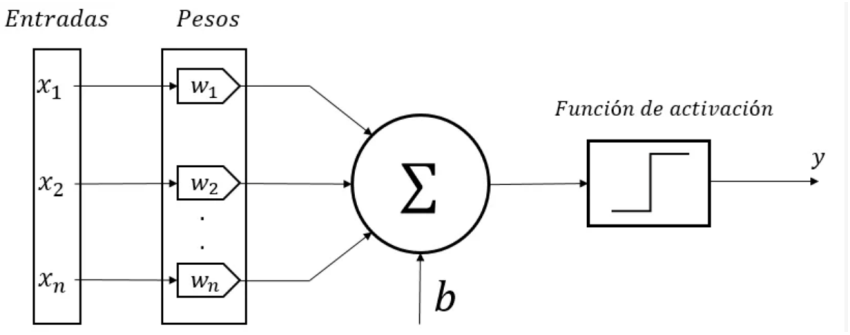


Figura 2.5: Representación del perceptrón simple

Este modelo no dispone de ninguna capa oculta, únicamente tiene una capa de entrada que se conecta directamente con la de salida. Esta conexión se realiza una vez que los valores de entrada son multiplicados por los pesos creando así una suma acumulada a la que se le aplica una función de activación para convertirlo en un dato de salida (output). Si éste sobrepasa un umbral, nos quedamos con ese output.

donde:

$$y = \begin{cases} 1, & \text{si } \sum w_i x_i \geq \text{umbral} \\ 0, & \text{en caso contrario} \end{cases}$$

2.4.2. MLP

Podemos considerar al modelo del perceptrón multicapa como una evolución del anterior. En cuanto a estructura, se diferencia en que cuenta con capas intermedias también llamadas capas ocultas. Éstas se disponen entre la capa de entrada y la de salida, haciendo que la red se convierta en una red neuronal con propagación hacia delante.

Como ventaja a destacar de este perceptrón, es que permite representar funciones no lineales.

Una de las diferencias más importantes respecto al SLP es la forma en la que son entrenados, implementando el algoritmo de retropropagación.

Este algoritmo podemos dividirlo en dos pasos:

1. Propagación: se calcula el resultado de salida de la red desde los valores de entrada hacia delante.

2. Aprendizaje: los erros obtenidos a la salida del perceptrón se van propagando hacia atrás (backpropagation) con el objetivo de modificar los pesos de las conexiones para que el valor estimado de la red se parezca cada vez más al real. Esta aproximación se realiza haciendo uso del algoritmo del gradiente descendente.

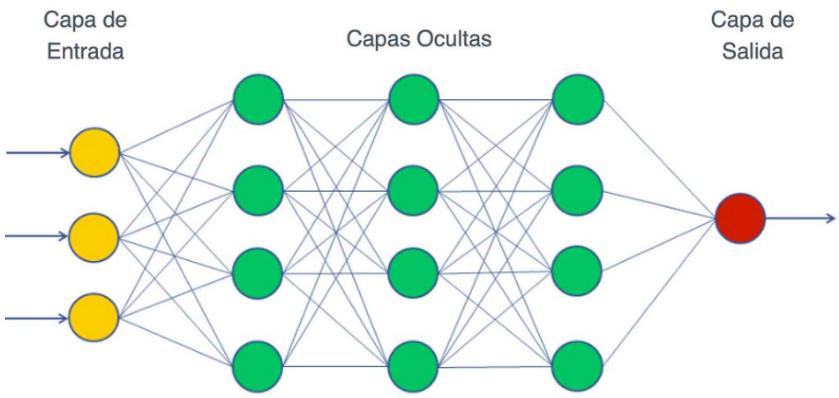


Figura 2.6: Representación del perceptrón multicapa

2.5. Tipos de arquitecturas de redes neuronales

Los dos tipos de arquitecturas que se van a ver en este trabajo son las redes feed-forward y las redes recurrentes.

- **Redes neuronales alimentadas hacia delante (FFNNs)**

También conocidas como redes neuronales feed-forward, son redes cuyas neuronas o nodos están organizados en capas sucesivas, de forma que si ordenamos estas capas de izquierda a derecha, únicamente hay conexiones entre neuronas de capas sucesivas, en sentido izquierda-derecha. La primera capa es siempre la de entrada, y la última de salida, además de contar con capas ocultas situadas en mitad de estas dos.

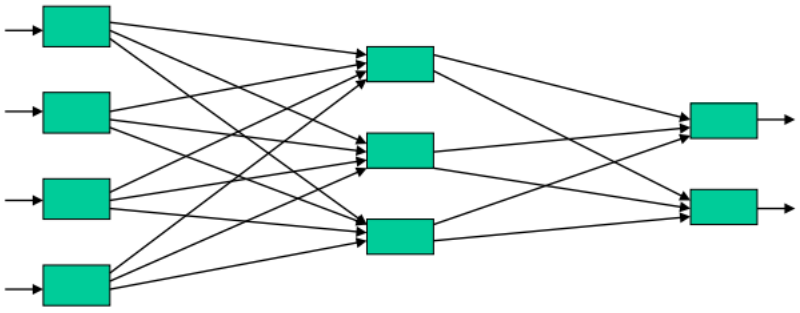


Figura 2.7: Representación de una red feed-forward

Estas redes van aplicando transformaciones a los datos de entrada de forma que cada capa va creando una nueva representación de estos datos. Se usan tanto para problemas de aprendizaje supervisado como no supervisado.

- **Redes neuronales recurrentes (RNNs)**

Las redes recurrentes se usan en tareas donde los datos vienen en forma de secuencia, como pueden ser series de tiempo o texto (palabras como secuencias de letras, y textos como secuencia de palabras).

La característica principal de este tipo de redes es que se dice que tienen "memoria". Se indagará más acerca de este tipo de arquitectura en el siguiente capítulo.

2.6. Evolución de las redes neuronales

El "deep" del Deep Learning viene referido al número de capas intermedias que tiene el modelo. Por tanto, podemos considerar que un modelo es de DL cuando tiene múltiples capas ocultas. Se les llama ocultas porque no necesariamente se conocen las entradas y las salidas de estas neuronas que forman las capas intermedias.

Cada una de las capas sucesivas van actuando como un filtro para los datos de entrada, haciendo que vaya siendo más fácil de realizar la tarea propuesta.

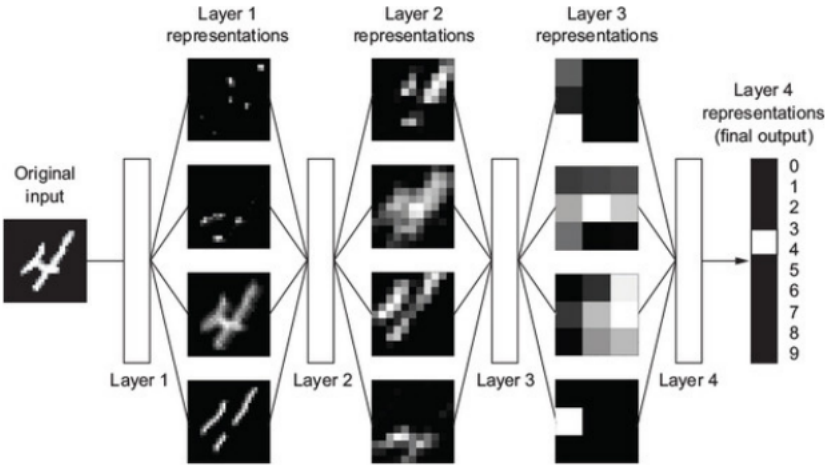


Figura 2.8: Representaciones por capas para un modelo de clasificación de dígitos

Una de las principales ventajas que tiene utilizar modelos de redes de neuronas profundas es que es capaz de realizar automáticamente la fase más difícil del flujo de trabajo de cualquier tarea de Machine Learning, que es el proceso de feature engineering o ingeniería de características. Éste consiste en la transformación y selección de variables que se introducirán en el modelo, y tiene siempre una gran repercusión en el rendimiento final de éste.

También destacan por su buen rendimiento en problemas en los que a priori no se tienen datos acerca de las salidas esperadas, los cuales se llaman problemas de aprendizaje no supervisado.

Por tanto, las tres características que hacen tan útil a un modelo del DL son:

- La forma de procesar los datos de entrada capa por capa hasta poder realizar la tarea más fácilmente.

- El aprendizaje conjunto al que se someten todas las capas, de forma que el cambio en una capa influya en todas las demás.
- Capacidad para realizar tareas con menor ayuda humana, comparado con el ML.

2.7. Ventajas e inconvenientes

Si nos ponemos a analizar los pros y contras que tiene el uso de redes neuronales, tendríamos lo siguiente:

Como ventajas:

- No se necesitan técnicas especializadas para la ingeniería de características, lo que simplifica mucho el proceso. Esta fase es completada automáticamente con operaciones de tensores.
- Posibilidad de procesar conjuntos de datos de cualquier tamaño, debido a que en el proceso de aprendizaje de un modelo de red neuronal éste se entrena dividiendo los datos en pequeños grupos y procesándolos por lotes.
- Capacidad de re-entrenamiento sin tener que empezar desde cero, lo que posibilita que pueda estar en continuo aprendizaje constante.

Y como inconvenientes:

- Alto volumen de tiempo requerido para el aprendizaje si el tamaño del conjunto de datos utilizado es grande, es decir, costoso computacionalmente.
- Necesitan un gran volumen de datos para el correcto entrenamiento del modelo

Capítulo 3

Redes neuronales recurrentes

3.1. Introducción

Las redes neuronales recurrentes, en inglés Recurrent Neural Networks (RNN), o también llamadas Modelos de Secuencia, fueron creadas en la década de 1980. El problema que había es que eran redes difíciles de entrenar para la tecnología del momento, por lo que no fue hasta hace unos pocos años con los avances tecnológicos cuando empezaron a ganar importancia en cuanto a utilidad y popularidad.

Éstas son el tipo de red de neuronas idóneas cuando se quiere tratar con datos secuenciales o datos de series temporales. Esta característica es importante a la hora del análisis de los datos ya que implica que éstos están correlacionados, y por tanto no deben ser tratados de forma independiente como se haría en cualquier otra red feed-forward.

Por ejemplo, podemos considerar que un texto es una secuencia de palabras, y una palabra es una secuencia de letras. De la misma forma, un vídeo puede verse como una secuencia de imágenes. El poder manejar este tipo de datos es lo que hace tan útiles a las redes recurrentes.

Mientras leemos este trabajo, nuestro cerebro está leyendo y a su vez concatenando cada una de las palabras que lo forman, de manera que nos permite comprender la idea principal del texto. Palabra por palabra sería imposible llegar a sacar unas conclusiones de lo leído, ya que necesitamos de un contexto que es obtenido únicamente leyendo todo en conjunto. Las redes recurrentes buscan imitar este comportamiento.

Entonces, la gran utilidad que tienen estas redes, y que las diferencia de las demás redes de neuronas, es que son capaces de analizar los datos de entrada de forma conjunta. Es decir, generando un contexto que facilite obtener información acerca de ellos que sería imposible si se analizasen uno por uno.

Las RNN son una de las redes de neuronas más usadas en la actualidad, y son utilizadas a día de hoy en tareas como la traducción de lenguaje, procesamiento del lenguaje natural (NLP) o predicción de series temporales, ya que permiten tratar la dimensión de “tiempo”.

3.2. Redes recurrentes vs redes feed-forward

La topología es una de las características más diferenciales cuando hablamos de redes neuronales. Tanto es que la mayoría de redes podríamos clasificarlas en uno de estos grupos:

- Redes recurrentes
- Redes feed-forward

Como ya hemos visto, las redes feed-forward se caracterizan porque en ellas la información circula de forma acíclica, de izquierda a derecha, desde la capa de entrada hasta la de salida. Es decir, las capas de neuronas se encuentran enlazadas de forma unidireccional.

Esto hace que carezca de sentido usarlas con datos secuenciales, porque procesarían de forma independiente cada dato sin tener en consideración la relación existente entre los datos de entrada, generando outputs independientes entre sí.

Poniendo un ejemplo, si queremos predecir hacia qué dirección va a dirigirse una pelota que está en el aire, si únicamente tenemos en cuenta una foto de ella, la respuesta será completamente aleatoria. Sin embargo, si contamos con varios fotogramas ordenados y separados en distintos instantes, será sencillo predecir la dirección a la que se dirigirá la pelota. La única forma de tener en consideración esta serie de fotogramas para realizar la predicción es usando una red neuronal recurrente.

Otra diferencia importante entre estos dos tipos de redes es que las feed-forward sólo pueden procesar un dato a la vez, como por ejemplo una imagen, pero en cambio, son incapaces de procesar un vídeo, que podríamos verlo como una secuencia de imágenes. Esta tarea sí que puede ser resuelta por una red recurrente.

Además, las redes recurrentes se distinguen por su "memoria", ya que obtienen información de entradas anteriores para influir en la entrada y salidas actuales.

3.3. Funcionamiento

Para el procesamiento de datos secuenciales, las redes recurrentes tienen un bucle interno que les permite ir iterando elemento por elemento de la secuencia, a la misma vez que va manteniendo un "estado oculto", o hidden state, que contiene información de los elementos de la secuencia ya procesados anteriormente. Este estado oculto es lo que construye la memoria de la red.

El estado oculto se reinicia cada vez que se termine el procesamiento de una secuencia, haciendo que no exista relación entre éstas. De esta forma, se consigue tomar una secuencia como un único dato de entrada, con la peculiaridad de que ese dato será procesado elemento a elemento mediante el bucle ya mencionado.

Estos modelos son llamados recurrentes porque el bucle hace que cada elemento de la secuencia sea procesado de la misma forma, y gracias a eso es posible procesar un texto sin tener que introducirlo de una sola vez en la capa de entrada, como se haría en una red feed-forward.

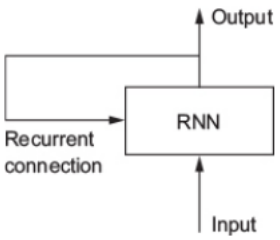


Figura 3.1: Esquema básico de funcionamiento de una RNN

Cada elemento de la secuencia lleva asociada su posición en ésta, que llamaremos timestep (instante de tiempo) y representaremos con la letra "t".

Las RNN son redes profundas que presentan una capa oculta y reciben un dato de entrada por cada timestep, con la característica de que todas estas capas usan los mismos pesos. Esto hace que el funcionamiento de estas redes pueda representarse como un bucle.

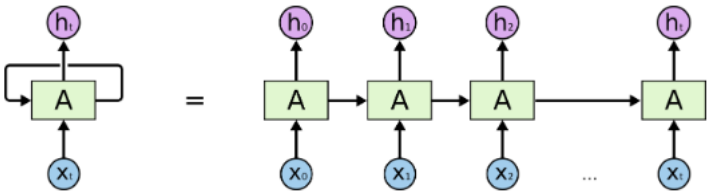


Figura 3.2: Despliegue de una red recurrente

3.4. Arquitecturas de RNN

Según el número de entradas y salidas que tenga una red recurrente, podemos diferenciar los siguientes tipos:

- **One to one**

Para esta configuración, una única entrada devuelve una única salida para cada instante de tiempo (timestep). Esto quiere decir que cada elemento asociado a un timestep se procesa de manera independiente, perdiendo así la capacidad de creación de memoria, y por tanto no aprovechando la característica diferencial de las RNN. De esta forma, la red funciona como una red feed-forward cualquiera.

Un caso práctico en el que se usa esta configuración es para la clasificación de imágenes.

- **One to many**

Se toma como entrada un único dato que acaba devolviendo varios datos de salida en forma de secuencia. A partir de un dato de entrada la red devuelve una salida para el primer instante de tiempo, y utilizando esta salida como entrada vuelve a generar otra nueva salida, repitiéndose este proceso hasta que se haya generado la secuencia.

Un ejemplo puede ser la generación de música. Como entrada puede tomarse "triste", y la red irá generando a partir de esa palabra una melodía acorde a lo solicitado.

- **Many to one**

Se tiene de entrada una serie de datos secuenciales que tras ser procesados devuelven un único dato como salida.

Esta configuración se usa para la clasificación de sentimientos. Como entrada se le introduce un texto como puede ser una crítica a una película, y devuelve como salida si el comentario ha sido positivo o negativo.

- **Many to many**

Se tomará como entrada una secuencia de datos que acabará devolviendo otra secuencia. Un ejemplo puede ser la traducción de idiomas.

Existe una casuística (dibujo de la derecha) que consiste en que la red devuelva un dato de salida para cada elemento de la secuencia de entrada, formando así como salida otra secuencia. Un ejemplo es la clasificación de vídeo, donde se quiere clasificar cada frame de éste.

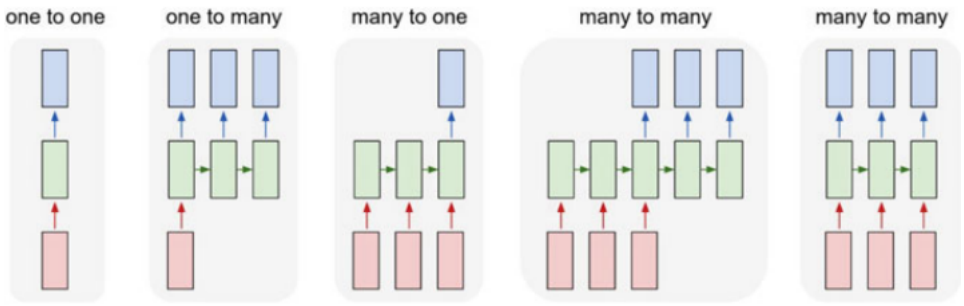


Figura 3.3: Tipos de arquitecturas de RNN

3.5. Red de Elman

Jeffrey Elman realizó importantes aportaciones a las arquitecturas de redes recurrentes al desarrollar este modelo que lleva su nombre, y que es considerado el primer modelo exitoso de red recurrente entrenado con backpropagation.

En un primer momento, su utilidad iba dirigida a tareas de procesamiento de lenguaje, pero éstas también son perfectamente aprovechables en cualquier problema en el que tengamos datos en forma de secuencia como entrada. En la siguiente imagen podemos ver la estructura básica de una red de Elman.

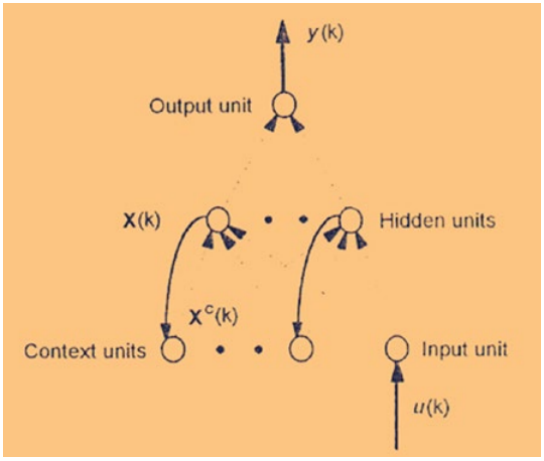


Figura 3.4: Estructura básica de una red de Elman

En la arquitectura propuesta por Elman, se incluyó una capa adicional de unidades de contexto que se encarga de almacenar la información de los estados anteriores de la capa oculta. Una de las características más importantes de esta red es que la salida de la capa oculta se retroalimenta hacia las unidades de contexto de la capa anterior, pero los pesos que conectan las unidades de contexto y la capa oculta tienen un valor constante de 1, por lo que la relación es lineal y no se producen modificaciones.

Después de que la capa oculta tomen entradas provenientes de la capa de entrada y de la capa de unidades de contexto, ésta genera una salida que se propaga hasta la capa de salida.

Durante la siguiente iteración, la secuencia de entrenamiento es similar, excepto porque ahora la capa de unidades de contexto adopta los valores de la capa oculta de la primera etapa.

Como es de esperar, para entrenar esta red se necesitará una cantidad de pasos en relación con la longitud de la secuencia.

3.6. Red totalmente recurrente

Imaginemos que estamos proporcionándole una entrada x a un modelo RNN, en el que definimos a su estado oculto como h, con las entradas siendo multiplicadas por una matriz de pesos que denotaremos como W.

A diferencia de las redes neuronales que hemos visto hasta ahora, las redes recurrentes realizan siempre la misma tarea en sus entradas a lo largo del tiempo. Por ello, para calcular el estado actual de una RNN se deriva la siguiente ecuación:

$$h_t = f(W_t x_t + W_R h_{t-1} + b),$$
$$y_t = f(W h_t + b)$$

donde:

t = instante de tiempo
h_t = estado oculto
x_t = vector de entrada
b = sesgo
W_t = matriz de pesos que conecta x_t con h_t
W_R = matriz de pesos que conecta h_{t-1} con h_t
W = matriz de pesos que conecta h_t con y_t
f = función de activación (tanh, ReLU)

Los coeficientes W y b para estas dos ecuaciones son obtenidos mediante el entrenamiento, ya explicado anteriormente.

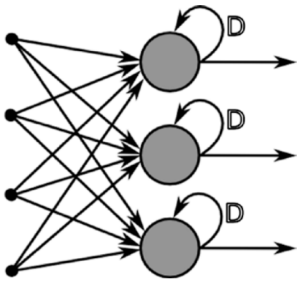


Figura 3.5: Arquitectura de una red recurrente

Si observamos ambas ecuaciones, se comprende de dónde vienen los conceptos de recurrencia y memoria asociados a este tipo de redes. La salida y_t depende del estado oculto actual h_t, pero a su vez este estado depende no sólo de la entrada actual x_t, sino también del valor del estado oculto anterior h_{t-1}. Ésta es la memoria de la red.

3.7. Entrenamiento del modelo (BPTT)

El hecho de que este tipo de redes trate con datos secuenciales en los que cada elemento tiene asignado un instante de tiempo (timestep), hace que no sea posible utilizar el algoritmo de *backpropagation,* como se haría en una red neuronal estándar, para entrenar la red. En su lugar, Sepp Hochreiter y Jurgen Schmidhuber desarrollaron una extensión del algoritmo llamado *backpropagation through time* (BPTT), o propagación hacia atrás en el tiempo.

Los principios del BPTT son los mismos que los de la retropropagación tradicional, donde el modelo se entrena a sí mismo calculando los errores desde la capa de salida hasta la capa de entrada. El BPTT se diferencia del enfoque tradicional en que éste suma los errores en cada paso temporal, mientras que las redes feed-forward no necesitan sumar los errores porque no comparten parámetros en cada capa.

A lo largo de este proceso, aparecen dos problemas causados por el tamaño del gradiente denominados gradientes explosivos y gradientes desvanecientes. Se profundizará en ellos en el próximo apartado.

Para cada iteración durante el entrenamiento del modelo, se comenzará entrenando secuencias cortas y se irá aumentando gradualmente la longitud de éstas. Puede verse de la siguiente forma: se van entrenando secuencias de longitud 1, 2, ... hasta N, siendo N la longitud máxima de la secuencia.

Podemos mostrar este proceso de una forma matemática:

$$\delta_{p,j}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(s_{pj}(t-1))$$

donde t es el instante de tiempo, h es el índice para el nodo oculto en t, j es el nodo oculto en el paso $t = 1$, y δ son los errores.

En detalle, podemos ver los fenómenos de la manera siguiente: definiendo W como la matriz de pesos para la capa de salida

$$W(t+1) = W(t) + \eta s(t) e_0(t)^T$$

donde e_0 representa los errores según la expresión siguiente:

$$e_0(t) = d(t) - y(t)$$

Ahora tenemos k secuencias, a través de las cuales desenrollamos la red hasta convertirla en una red feed-forward cualquiera.

Sin embargo, la capa recurrente en un modelo RNN toma simultáneamente la entrada de la capa anterior, además de la capa sucesiva. Debido al cambio en los pesos que se produce a causa de la entrada simultánea al realizarse la propagación de los errores, se promedian las actualizaciones que recibe cada capa.

3.8. Problema del algoritmo de entrenamiento de las redes

Como se comentó en el apartado anterior, se pueden plantear los siguientes problemas.

3.8.1. Desvanecimiento del gradiente

Con desvanecimiento de gradientes nos referimos a cuando, mientras se está produciendo el entrenamiento del modelo mediante el algoritmo de backpropagation, los gradientes en las primeras capas de la red pasan a ser muy pequeños. Esto ocurre con cualquier función de activación, ya sea la tangente hiperbólica como la sigmoide.

El problema que esto acarrea es que el aprendizaje del modelo en estas capas puede ralentizarse e incluso detenerse, haciendo que no sea posible reconocer patrones que se den a largo plazo.

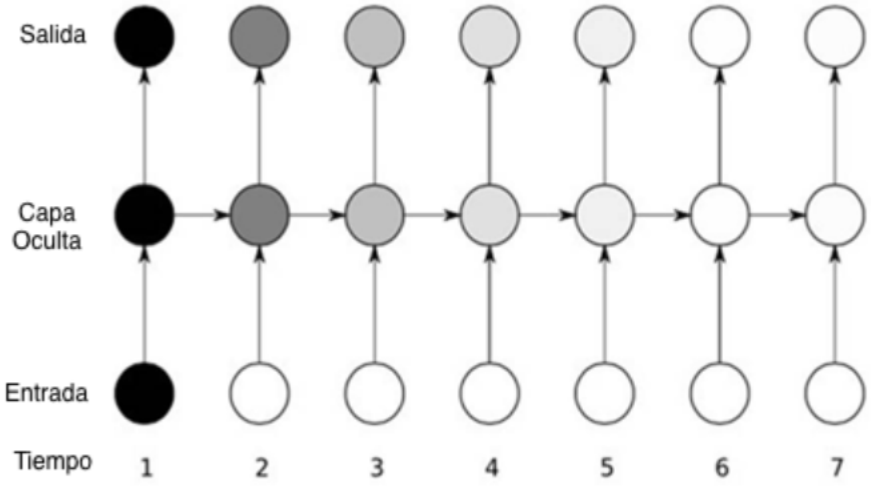


Figura 3.6: Ilustración del desvanecimiento del gradiente

Por otro lado, también puede darse el fenómeno contrario, es decir, que durante la etapa de backpropagation los gradientes comiencen a ser demasiado grandes, desestabilizando así al modelo.

Cuantos más timesteps tengan los datos, más posibilidades habrán de que se produzcan estos dos problemas.

Por esta razón, en la práctica no suelen utilizarse redes recurrentes como las que hemos visto hasta ahora. En lugar de ello, se hacen uso de un modelo más sofisticado como son las LSTMs, en los que nos centraremos más adelante.

3.8.2. Problema de las dependencias a largo plazo

La ventaja que tienen las RNN frente a las demás redes de neuronas es que éstas pueden preservar información del pasado para después conectarla con la tarea actual. De esta forma, si a un modelo de lenguaje le proporcionamos como entrada la frase "José fue a pelarse a la peluquería", el modelo será capaz de predecir sin problema alguno que la última palabra es peluquería. Por tanto, vemos que si no hay demasiada distancia entre el instante de tiempo en el que se encuentra la información relevante y el instante en el que hacemos la predicción, el modelo actuará correctamente.

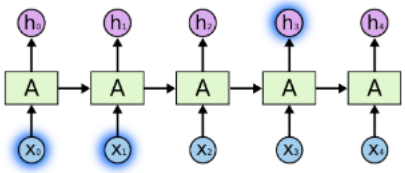


Figura 3.7: Dependencia a corto plazo

Las dificultades llegan cuando se le presenta al modelo un texto en el que la distancia entre esa información importante y el momento a predecir es grande. Rescatando la frase anterior, imaginemos que pertenece a un texto más largo en el que, al principio de éste,

se habló de que el tío de José es peluquero y acostumbra a pelar a sus familiares. La información más reciente que le llega al modelo es que el sujeto va a pelarse, y por tanto lo más probable es que acuda a una peluquería, pero si vamos hacia atrás en el texto obtendremos un contexto que dice que su tío es peluquero, y que por tanto es más probable que vaya a su casa a pelarse.

Para que el modelo establezca ese contexto, debe retroceder una gran cantidad de instantes en el tiempo, lo que puede hacer que éste no sea capaz de conectar la información y por consiguiente no hacer una predicción correcta.

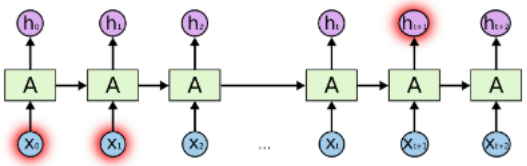


Figura 3.8: Dependencia a largo plazo

Para solucionar estos dos problemas, se hace uso de las LSTMs, que, resumidamente, son un tipo de redes recurrentes que tienen la habilidad de poder conectar información que se encuentra a larga distancia, y poder así reconocer dependencias a largo plazo.

Otra alternativa puede ser utilizar una capa GRU (Gated Recurrent Unit) que, al igual que la LSTM, fue desarrollada para abordar los problemas antes mencionados, pero con la diferencia de que lo consigue con un coste computacional menor, lo cual puede beneficiarnos a la hora de la práctica.

3.9. Long Short-Term Memory (LSTM)

El modelo Long Short-Term Memory (LSTM), o modelo de memoria a largo y corto plazo en español, es un tipo de red recurrente desarrollado por Hochreiter y Schmidhuber a finales de la década de 1990, con el objetivo de solucionar los problemas de desvanecimiento de gradiente y de dependencias a largo plazo.

Una red LSTM es capaz de "recordar" información relevante en la secuencia y de preservarlo durante varios instantes de tiempo. Con esto consigue tener una memoria tanto a corto, como ya hacen las RNN básicas, como a largo plazo. El añadir una capa LSTM a una red neuronal podría asemejarse a incorporar una unidad de memoria que pueda recordar el contexto desde los primeros instantes de tiempo de la secuencia de entrada.

Podemos comparar el funcionamiento de este modelo con una cinta transportadora que se encuentra paralela al procesamiento de la secuencia. Los datos de la secuencia pueden ir añadiéndose a la cinta en cualquier momento, ser transportados a instantes de tiempo posteriores o incluso salir de la cinta. Realmente lo que hace es ir recogiendo la información relevante de la secuencia, evitando así que ésta pueda ir perdiéndose durante el procesamiento como vimos con el problema de desvanecimiento del gradiente.

Como ya hemos visto, todas las redes recurrentes tienen una forma de cadena en la que se van repitiendo módulos de redes neuronales. Mientras que para rnn básicas este

módulo que se repite es bastante básico con una sola capa tangente hiperbólica, si nos vamos a las LSTMs la cosa se complica. En lugar de tener una capa, tienen cuatro capas interactivas para las cuales explicaremos su función en el siguiente apartado.

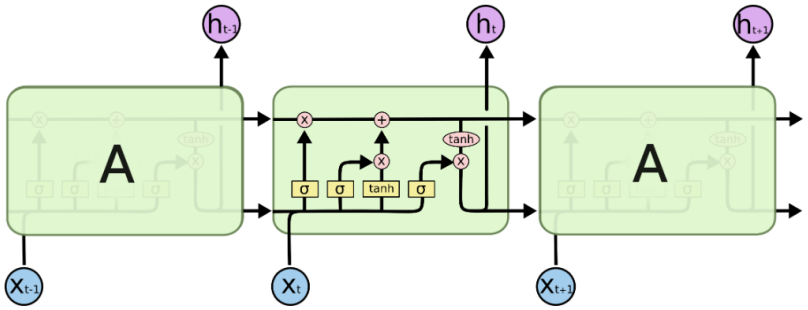


Figura 3.9: Módulo repetitivo de una red LSTM

- **Funcionamiento de una LSTM tradicional**

Forget gate

El primer paso en una LSTM es decidir qué información vamos a descartar de la celda de estado. Esta decisión se toma haciendo uso de una capa sigmoide, que recibe de entrada h_{t-1} y x_t, y devuelve como salida un número entre 0 y 1 para cada número en la celda de estado C_{t-1}. Un 1 significaría que debemos mantenerlo sin duda, mientras que un 0 nos diría que hay que descartarlo.

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

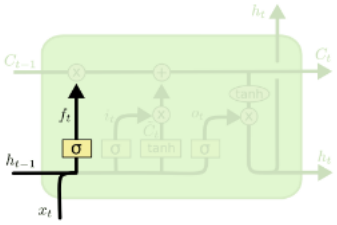


Figura 3.10: Representación de la forget gate

Input gate

El segundo paso consiste en decidir qué nueva información vamos a guardar en la celda de estado, y puede dividirse en dos partes:

- Primero, una capa sigmoide que llamaremos “input gate” decide qué valores vamos a actualizar.
- Luego, otra capa tangente hiperbólica se encarga de crear un vector de nuevos valores candidatos, $\tilde{C}_t$, que podrían ser añadidos a la celda de estado.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C)$$

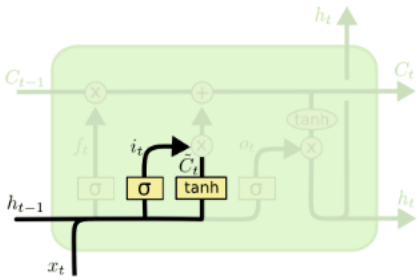


Figura 3.11: Representación de la input gate

Para actualizar la antigua celda de estado, C_{t-1} , a C_t , las decisiones ya han sido tomadas en los dos pasos anteriores, por lo que veamos de qué forma se realiza la actualización.

Para olvidar lo que decidimos en la forget gate, multiplicamos la celda de estado antigua por f_t , y añadimos también $i_t * \tilde{C}_t$. Con esto último lo que hacemos es añadir los nuevos candidatos escalados por la cantidad que hemos decidido actualizar cada valor.

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

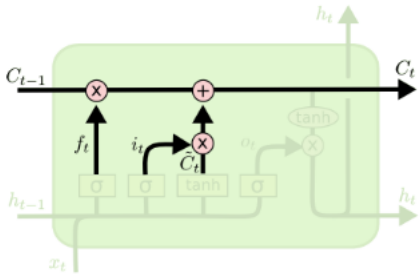


Figura 3.12: Proceso de actualización de la celda de estado

Output gate

Para finalizar, hay que decidir lo que se va a devolver como salida. Éste output se obtendrá de una versión filtrada de la celda de estado.

- Se empieza aplicando una capa sigmoide que decide qué partes de la celda de estado se van a devolver.
- Después, se le aplica a estos valores una capa tangente hiperbólica (que establecerá los valores en el intervalo (-1,1)) y se multiplicarán por la capa sigmoide anterior, de forma que la salida estará comprendida por las partes que ésta decida.

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

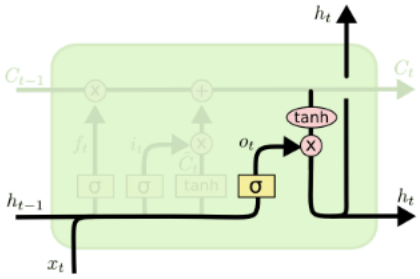


Figura 3.13: Representación de la output gate

donde:

x_t = vector de entrada
h_t = vector de salida
c_t = celda de estado
(W, b) = matriz de parámetros y vector
(f_t, i_t, o_t) = información recordada, información adquirida y salida, respectivamente.
σ = función sigmoide
tanh = función tangente hiperbólica

- **Entrenamiento**

Para entrenar las LSTMs puede usar el método de backpropagation through time que vimos anteriormente, aunque realmente podríamos entrenarlo al igual que cualquier red recurrente básica mediante el tradicional backpropagation.

Esto es posible por las siguientes razones:

- Las LSTMs ya solucionan el problema del desvanecimiento de gradiente,
- y además podría decirse que recuerdan los errores de la backpropagation, con los que retroalimentan a cada uno de los pesos.

Por estos motivos, podemos retomar el método tradicional y seguirá siendo de utilidad para hacer que el bloque LSTM recuerde valores durante un largo periodo de tiempo.

3.10. GRU (Gated Recurrent Unit)

Una variante muy utilizada de las LSTM son las Gated Recurrent Unit (GRU), desarrolladas por Chung et al. en 2014, y que también resuelven el problema del desvanecimiento de gradiente. Se le considera una variante de la LSTM porque comparten un diseño muy parecido y, en muchos casos, hasta resultados bastante positivos.

Entre los cambios más significativos entre estos dos modelos encontramos:

- Mientras que las LSTM tienen 3 compuertas, las GRU se reducen a únicamente dos. Esto es debido a que combina la input y la forget gate para crear una única "update gate".
- Fusiona la celda de estado, C_t, con el estado oculto, h_t.

Tras esas modificaciones queda un modelo más simple que un LSTM estándar, y lo que es más interesante: menos costoso computacionalmente. Estas ventajas están provocando que las GRU estén ganando mucha popularidad en estos últimos años.

- **Funcionamiento de una GRU**

Primero vamos a ver una representación del esquema de funcionamiento de una GRU, para después pasar a explicar los pasos que realiza.

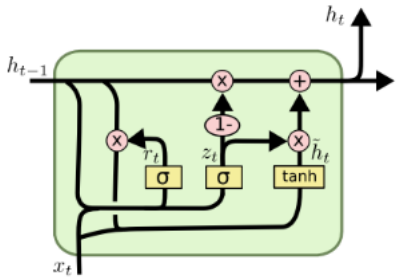


Figura 3.14: Gated Recurrent Unit

Update gate

En este primer paso lo que se decide es cuánta información del pasado se va a conservar y cuánta información nueva se va a agregar a la unidad recurrente en el paso de tiempo actual. Esta decisión se toma haciendo uso de una capa sigmoide que recibe de entrada el estado oculto de la unidad anterior, h_{t-1}, y la entrada actual, x_t, y que devuelve como salida un número entre 0 y 1 que determinará cuánta información debe seguir transmitiéndose.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

Reset gate

El segundo paso consiste en decidir cuánta información del pasado se va a descartar. Esta decisión se toma también haciendo uso de una capa sigmoide que actúa igual que la de la compuerta anterior, con la diferencia de que utiliza otra matriz de pesos.

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

Contenido actual de la memoria

Llegado a este punto, se introduce un "nuevo estado oculto", $\tilde{h_t}$, que va ser usado por la reset gate para almacenar la información relevante del pasado. Todo a través de una capa tangente hiperbólica que va a crear un vector de nuevos valores candidatos a estado oculto en el paso de tiempo actual, que son calculados de la siguiente forma:

$$\tilde{h_t} = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

Estado oculto final

En el último paso, se calcula el estado oculto final que almacenará la información relevante de la unidad actual para después transmitirla a la siguiente.

Para ello retomamos la update gate. La función de ésta es determinar con qué información debe quedarse la red de entre el contenido actual, o estado actual, de la memoria, $\tilde{h_t}$, y la información proveniente de los pasos previos, h_{t-1}. Este cálculo se realiza de la siguiente forma:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t}$$

3.11. Aplicaciones

Algunos ejemplos de tareas para las que se utilizan redes recurrentes son las siguientes:

- Reconocimiento de voz: a partir de las ondas de sonido producidas por las voces, son capaces de generar palabras.
- Clasificación de sentimientos: reciben por ejemplo una crítica a una película como secuencia de entrada y devuelven si la crítica ha sido negativa o positiva.
- Traducción de idiomas
- Predicciones meteorológicas: utilizando datos históricos, pueden detectar dependencias temporales y modelizar relaciones entre las variables meteorológicas como pueden ser la temperatura y la humedad. A partir de eso, son capaces de aprender patrones climáticos y hacer predicciones futuras.

Capítulo 4

Práctica

La aplicación de redes de neuronas profundas se ha conseguido gracias al desarrollo de frameworks de Machine Learning como Tensorflow, que concretamente para Deep Learning es la librería más usada, acompañado de librerías como Keras que tiene como objetivo hacer más accesible el uso de Tensorflow. Después de su lanzamiento en 2015, Keras se ha convertido en una herramienta indispensable para la implementación de redes neuronales, usada en numerosas empresas start-ups, universidades e investigaciones.

4.1. Keras

Keras es un framework de ML que proporciona una sistema fácil y eficiente para definir y entrenar cualquier tipo de modelo de red neuronal, en concreto lo usaremos para trabajar con redes recurrentes.

Keras es una API de redes de neuronas de alto nivel desarrollada con el objetivo de facilitar una rápida experimentación

Tensorflow es una librería matemática de bajo nivel creada para la construcción de arquitecturas de aprendizaje profundo.

Tiene las siguientes características:

- Ejecución versátil: permite que sea posible ejecutar el código tanto en CPU como GPU.
- Simplicidad: tienen una API intuitiva que permite usar modelos de redes neuronales de forma sencilla y sin tener amplios conocimientos en programación.
- Soporte para distintos tipos de redes: capaz de tratar con redes convolucionales, recurrentes y cualquier combinación entre éstas.
- Flexibilidad: es apropiado para construir desde los modelos de redes más simples hasta los más avanzados.
- Potencia: aporta tal rendimiento que es usado en organizaciones como la NASA.

El flujo de trabajo al usar Keras es el siguiente:

1. Definimos el modelo: lo hacemos usando la función `keras_model_sequential()`, que genera un modelo keras compuesto por un conjunto apilado de capas lineales.

2. Compilamos: utilizando la función `compile()` configuramos el modelo para el entrenamiento, pudiendo definir optimizador, métrica para la función de pérdida, ... entre otros.

3. Ajustamos el modelo: usaremos `fit()` para entrenar el modelo durante un número de iteraciones.

4. Evaluar al modelo: utilizaremos `evaluate()`.

5. Realizar predicciones: por último se realizan predicciones con `predict()`.

Tras dar esta pequeña introducción a la librería que se va a utilizar, procedemos con la práctica.

4.2. Aplicación práctica: predicción de la temperatura del aire en Jena

Para la parte práctica de este trabajo, vamos a utilizar un conjunto de datos de series temporales registrados por el Instituto Max Planck de Biogeoquímica. Éste contiene información acerca del clima en la ciudad de Jena, Alemania.

Contamos con 14 variables cuantitativas (como la temperatura, la presión atmosférica, la humedad, entre otros) que han sido observadas cada 10 minutos durante varios años. Concretamente, nuestro dataset cubre desde el 1 de junio de 2009 hasta el 31 de diciembre de 2016.

El objetivo de esta práctica va a ser construir un modelo que tome de entrada un histórico de datos, y que a partir de ellos pueda predecir la temperatura del aire para varios instantes futuros.

Para ello, vamos a hacer uso de distintos tipos de redes neuronales recurrentes que iremos mejorando progresivamente para ver de qué forma se comporta el modelo en cuanto a rendimiento. Consideramos que estos datos son ideales para aplicar redes recurrentes, ya que de esa forma en las predicciones se tendrá en cuenta la dimensión "tiempo", que tan importante es en una serie temporal.

La metodología que vamos a usar es la siguiente: vamos a dividir los datos en conjunto de entrenamiento, validación y test. Los de entrenamiento y validación nos servirán para entrenar y medir el rendimiento del modelo, respectivamente, y luego pasaremos a realizar predicciones sobre el conjunto test las cuales enfrentaremos con los datos reales en una representación gráfica, para que así veamos al modelo en acción.

4.2.1. Librerías

Para realizar el estudio, vamos a necesitar de las siguientes librerías de R:

- Tidyverse: conjunto de paquetes en R diseñado para ciencia de datos que permite, entre otras más funcionalidades, la lectura, manipulación y representación de datos.
- Tensorflow: paquete que da acceso a la API de Tensorflow desde R.
- Keras: este paquete permite el uso fácil de Keras y Tensorflow en R.

```
library(tidyverse)
library(tensorflow)
library(keras)
```

4.2.2. Lectura y exploración de datos

Para la lectura del dataset usamos la función `read_csv` del paquete readr que, además de leer, proporciona la dimensión del conjunto de datos y el tipo de dato para cada variable.

```
raw_data = readr::read_csv('datos/jena_climate_2009_2016.csv')
```

Echamos un pequeño vistazo al dataframe generado:

```
glimpse(raw_data)
```

```
## Rows: 420,551
## Columns: 15
## $ ‘Date Time‘       <chr> "01.01.2009 00:10:00", "01.01.2009 00:20:00", "01.01~
## $ ‘p (mbar)‘        <dbl> 996.52, 996.57, 996.53, 996.51, 996.51, 996.50, 996.~
## $ ‘T (degC)‘        <dbl> -8.02, -8.41, -8.51, -8.31, -8.27, -8.05, -7.62, -7.~
## $ ‘Tpot (K)‘        <dbl> 265.40, 265.01, 264.91, 265.12, 265.15, 265.38, 265.~
## $ ‘Tdew (degC)‘     <dbl> -8.90, -9.28, -9.31, -9.07, -9.04, -8.78, -8.30, -8.~
## $ ‘rh (%)‘          <dbl> 93.3, 93.4, 93.9, 94.2, 94.1, 94.4, 94.8, 94.4, 93.8~
## $ ‘VPmax (mbar)‘    <dbl> 3.33, 3.23, 3.21, 3.26, 3.27, 3.33, 3.44, 3.44, 3.36~
## $ ‘VPact (mbar)‘    <dbl> 3.11, 3.02, 3.01, 3.07, 3.08, 3.14, 3.26, 3.25, 3.15~
## $ ‘VPdef (mbar)‘    <dbl> 0.22, 0.21, 0.20, 0.19, 0.19, 0.19, 0.18, 0.19, 0.21~
## $ ‘sh (g/kg)‘       <dbl> 1.94, 1.89, 1.88, 1.92, 1.92, 1.96, 2.04, 2.03, 1.97~
## $ ‘H2OC (mmol/mol)‘ <dbl> 3.12, 3.03, 3.02, 3.08, 3.09, 3.15, 3.27, 3.26, 3.16~
## $ ‘rho (g/m**3)‘    <dbl> 1307.75, 1309.80, 1310.24, 1309.19, 1309.00, 1307.86~
## $ ‘wv (m/s)‘        <dbl> 1.03, 0.72, 0.19, 0.34, 0.32, 0.21, 0.18, 0.19, 0.28~
## $ ‘max. wv (m/s)‘   <dbl> 1.75, 1.50, 0.63, 0.50, 0.63, 0.63, 0.63, 0.50, 0.75~
## $ ‘wd (deg)‘        <dbl> 152.3, 136.1, 171.6, 198.0, 214.3, 192.7, 166.5, 118~
```

Confirmamos que, sin contar la variable que nos indica el instante de tiempo, el resto son numéricas.

Antes de empezar a manipular los datos, es conveniente realizar algunas representaciones de pequeños fragmentos de la serie temporal para ver qué aspecto tiene, y para ello utilizaremos la librería ggplot que viene incorporada dentro de tidyverse. Éstas van a ir

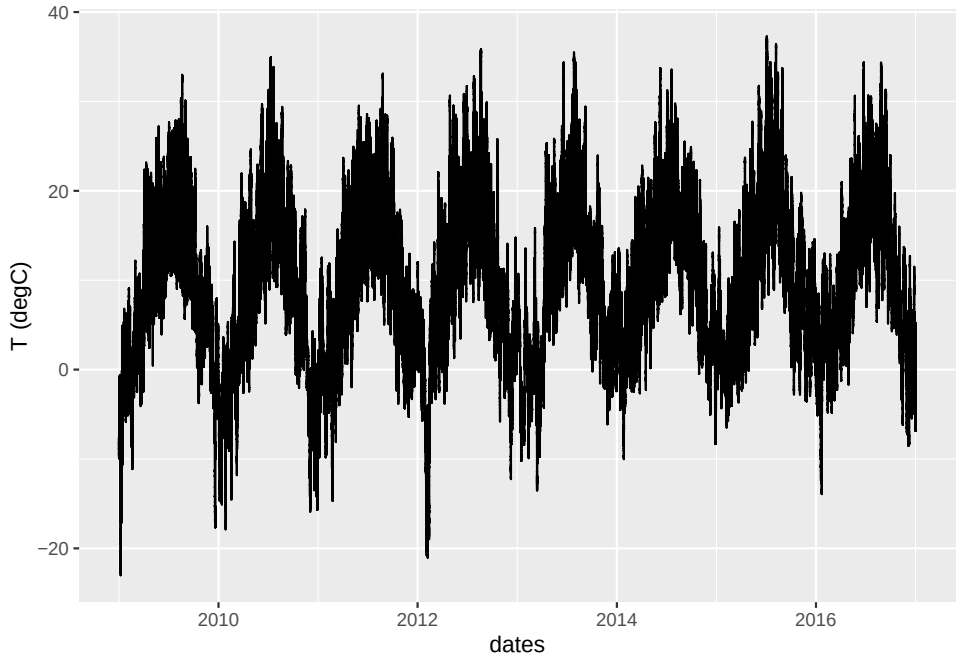
enfocadas a estudiar el comportamiento de la variable a predecir, "T (degC)", que es la temperatura en grados centígrados, a lo largo del tiempo.

Tal y como viene el dataset por defecto, el programa no es capaz de reconocer que los datos son series temporales. Por ello, vamos a crear una variable que contenga la serie temporal correspondiente, para luego pasar a representarla:

```
raw_data <- raw_data %>%
  mutate(dates = dmy_hms(str_replace_all(raw_data$`Date Time`, "[.]", "-")))
```

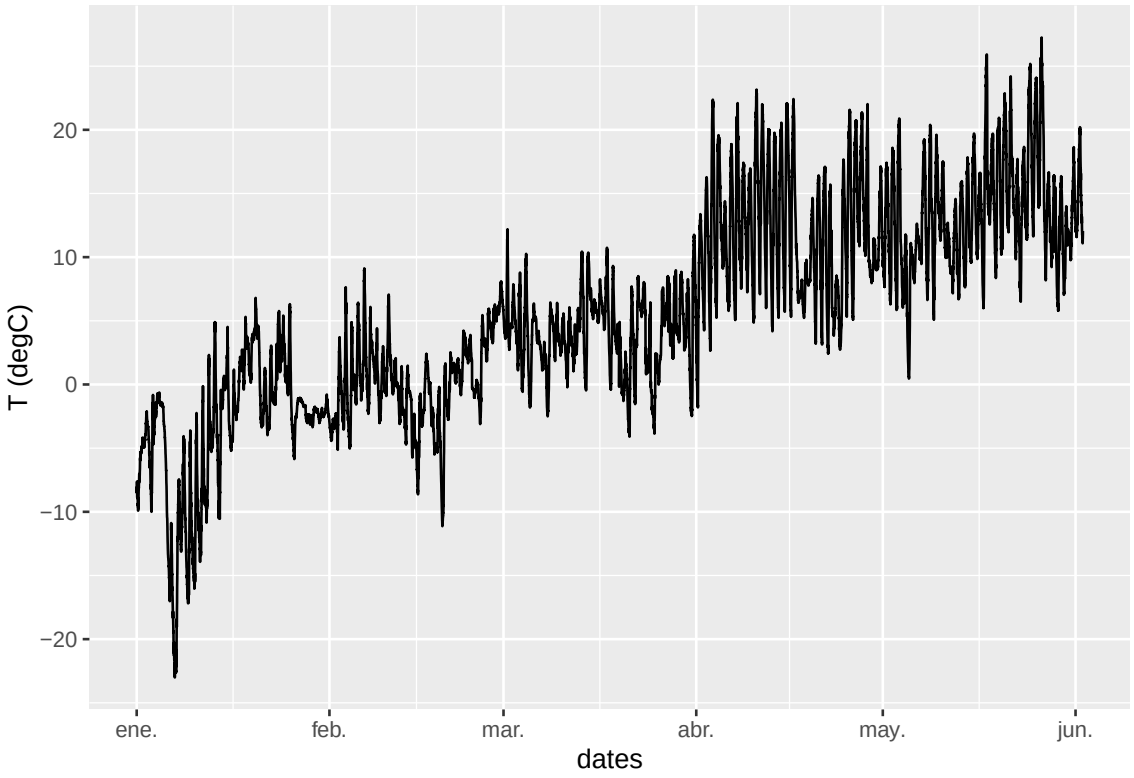
Empecemos representando la serie temporal al completo:

```
ggplot(raw_data, aes(x = dates, y = `T (degC)`)) + geom_line()
```



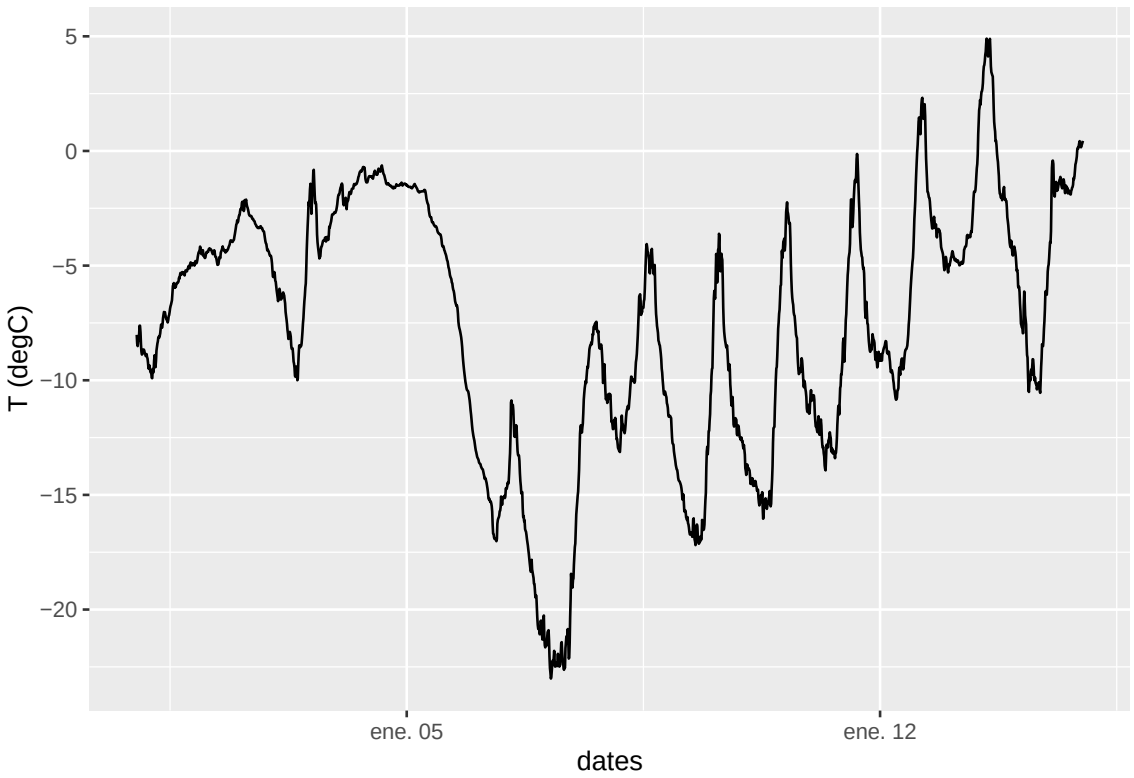
Vemos claramente que existe una clara periodicidad anual de la temperatura, ya que los datos muestran variaciones que se repiten en un ciclo de un año.

Observemos ahora los 5 primeros meses, con el objetivo de ver la dificultad de hacer predicciones de mes a mes:



Aunque se observan también algunos patrones repetitivos cada mes, no parece que haya tanta relación entre dos meses consecutivos como sí pasa con los años.

Para terminar con este breve análisis exploratorio, veamos la temperatura durante las dos primeras semanas de estudio. Cuando pasemos al modelo, los datos que usemos como test tendrán este rango de instantes de tiempo.



4.2.3. Preprocesamiento

Algo indispensable a la hora de trabajar con redes de neuronas es la transformación de los datos, de forma que éstas puedan trabajar con ellos. En este caso, todas las variables que tenemos son numéricas, y por tanto no va a ser necesario realizar ninguna vectorización. El problema que tenemos es que cada campo está en una escala distinta, por ejemplo la temperatura y la presión atmosférica, y eso puede causarnos problemas en el estudio.

Para solucionarlo vamos a normalizar cada variable de manera independiente, de forma que pasen todas a tomar valores dentro de una escala similar. Eso lo haremos calculando la media de cada serie temporal y dividiéndola por la desviación estándar. Como vamos a usar de entrenamiento los primeros 200.000 instantes de tiempo, sacaremos la media y desviación típica para la normalización sólo a partir de esos datos.

Antes de eso, debemos convertir nuestro conjunto de datos en una matriz de puntos flotantes para que puedan ser tratados por una red neuronal. Vamos a descartar las dos variables que corresponden al tiempo ya que no nos serán de utilidad.

```
# Convertimos el dataframe en una matriz de puntos flotantes
data <- data.matrix(raw_data[, -c(1, 16)])

# Seleccionamos el conjunto de entrenamiento:
train_data = data[1:200000,]

# Normalizamos los datos:
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
data <- scale(data, center = mean, scale = std)
```

El paso siguiente es definir una función generadora que tome de entrada la matriz de datos que acabamos de crear, y devuelva lotes de datos de su pasado reciente junto con el valor real de la temperatura en el futuro. La utilidad que tienen este tipo de funciones es que nos permiten tratar con una cantidad de datos muy grande sin tenerlos que cargar al completo en memoria, lo que podría ser un problema para el ordenador.

A una función generadora se la llama repetidamente para obtener secuencias de valores. Normalmente, los generadores necesitan mantener un estado interno, es decir, una memoria que les indique los valores que ha devuelto previamente. Para ello, suelen ser construidos llamando a otra función que devuelve la función generadora.

Cuando estas funciones son introducidas en un entrenamiento usando Keras, éstas deben poder devolver valores de forma infinita, y para controlar el número de llamadas que se le harán lo controlamos con los siguientes parámetros:

- `epochs` => número de iteraciones para entrenar el modelo
- `steps_per_epoch` => número de pasos de entrenamiento que se realizarán en cada iteración. En cada paso de entrenamiento se procesarán lotes de datos con un tamaño determinado (batch_size).

Vamos a utilizar la siguiente función para generar las secuencias:

```
generator <- function(data,
                      lookback,
                      delay,
                      min_index,
                      max_index,
                      shuffle = FALSE,
                      batch_size = 128,
                      step = 6) {

  if (is.null(max_index))
    max_index <- nrow(data) - delay - 1

  i <- min_index + lookback

  function() {
    if (shuffle) {
      rows <-
        sample(c((min_index + lookback):max_index), size = batch_size)
    } else {
      if (i + batch_size >= max_index)
        i <<- min_index + lookback
      rows <- c(i:min(i + batch_size - 1, max_index))
      i <<- i + length(rows)
    }

    samples <- array(0, dim = c(length(rows),
                                lookback / step,
                                dim(data)[[-1]]))
    targets <- array(0, dim = c(length(rows)))

    for (j in 1:length(rows)) {
      indices <- seq(rows[[j]] - lookback, rows[[j]] - 1,
                     length.out = dim(samples)[[2]])
      samples[j, , ] <- data[indices, ]
      targets[[j]] <- data[rows[[j]] + delay, 2] # temperatura (col 2)
    }

    list(samples, targets)
  }
}
```

Este generador devolverá una lista (samples, targets), donde samples será el lote de datos de entrada que tomará el modelo, y targets contendrá los correspondientes valores reales de temperatura para cada timestep del lote. A continuación se definen los argumentos que toma la función y que aún no se han explicado:

- `lookback` => número de instantes de tiempo anteriores que serán considerados para realizar la predicción

- `delay` => periodo en instantes de tiempo entre el último dato de entrada y la posterior predicción.
- `min_index` y `max_index` => índices de nuestro conjunto de datos que delimitarán los timesteps que debe tomar el generador.
- `shuffle` => booleana para indicar si queremos que los datos no vengan ordenados cronológicamente
- `batch_size` => tamaño de los lotes (batches) de muestras que se utilizarán en el entrenamiento.
- `steps` => periodo en timesteps en el que queremos que vengan representados los datos. Al establecerlo en 6, pasaremos a tener un registro por hora en lugar de cada 10 minutos.

Definida la función generadora, vamos a hacer uso de ella para instanciar de momento un generador para entrenamiento y otro para validación. Cada generador obtendrá los datos de periodos de tiempo distintos: el de entrenamiento usará los primeros 200.000 instantes de tiempo, y el de validación los 100.000 siguientes.

Antes de ello, vamos a explicar los valores que hemos dado a algunos parámetros:

- `lookback = 1440` : para hacer la predicción de la temperatura para un instante determinado, se tendrán en cuenta los 1440 instantes previos, que equivalen a 10 días.
- `steps = 6` : al establecerlo en 6, pasaremos a tener un registro por hora en lugar de cada 10 minutos. Esto tiene más sentido ya que no habrá demasiada variación en la temperatura en un intervalo de 10 minutos.
- `delay = 144` : después de mirar los 10 días anteriores, se realizará la predicción para 1 día (144 timesteps) después del último instante de tiempo.

```
lookback = 1440  # (10 dias)
step = 6         # un registro por hora
delay = 144      # predicciones para dentro de un día
batch_size = 128 # tamaño de cada lote

# Entrenamiento
train_gen <- generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 1,
  max_index = 200000,
  shuffle = FALSE,
  step = step,
  batch_size = batch_size)
```

```
train_gen_data <- train_gen()

# Validación
val_gen = generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 200001,
  max_index = 300000,
  step = step,
  batch_size = batch_size)

val_steps = (300000 - 200001 - lookback) / batch_size
```

El valor "val_steps" nos indica el número de veces que debe llamarse a la función generadora de validación para que ésta se vea al completo.

El generador para el conjunto test será instanciado cuando pasemos a explicar las predicciones del modelo.

4.2.4. Modelización

Llegada esta fase, vamos a experimentar con distintos modelos hasta llegar a uno que me produzca los mejores resultados. Para cada modelo vamos a mostrar lo siguiente:

- Construcción
- Entrenamiento
- Representación para un mes sacado del conjunto test en el que se enfrenten los valores reales frente a las predicciones realizadas por el modelo.
- Evaluación del modelo para ese conjunto test, basándome en el error absoluto medio (MAE).

Feed-forward

Aunque esta práctica esté dirigida a las redes recurrentes, consideramos que puede ser interesante ver el desempeño de un modelo de red neuronal más básico con el objetivo de después compararlo con los distintos modelos que vayamos implementando. Hemos optado por empezar implementando una red feed-forward.

```
model =  keras_model_sequential() %>%
  layer_flatten(input_shape = c(lookback / step, dim(data)[-1])) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
```

```
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history = model %>% fit(
  train_gen,
  steps_per_epoch = 500,
  epochs = 20,
  validation_data = val_gen,
  validation_steps = val_steps,
  callbacks = callback_early_stopping(monitor = "val_loss",
                                      patience = 2,
                                      mode = "auto",
                                      restore_best_weights = TRUE)
)
```

El objeto "history" incluye los parámetros que se han usado para entrenar al modelo, además de las métricas que han sido monitorizadas (la MAE en este caso). Éste también nos permite visualizar las métricas del entrenamiento y la validación iteración por iteración.

Pasamos a visualizar las curvas de pérdida tanto para el conjunto de entrenamiento como para el de validación:

```
plot(history)
```

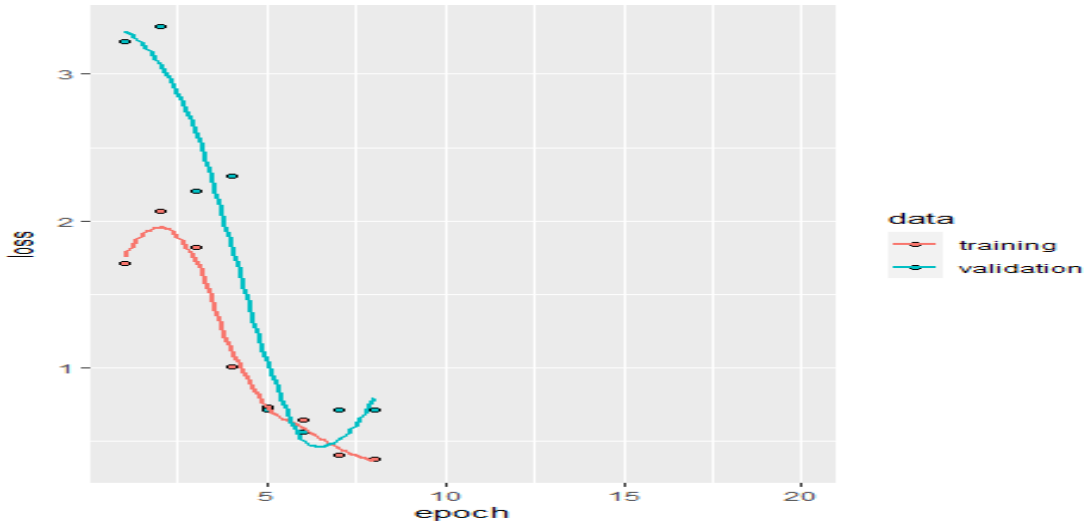


Figura 4.1: Curvas de pérdida usando modelo con red feed-forward

Cabe mencionar que, aunque estaba configurado que el entrenamiento se realizara durante 20 iteraciones, éste ha parado en la octava etapa debido al argumento `callbacks`. Éste consigue que el entrenamiento se pare cuando las métricas para el conjunto de validación no están mejorando durante un número de etapas seguidas, y lo más interesante es

que se queda con los pesos de la iteración con la función de pérdida más baja (por ser nuestra métrica la MAE). Usándolo, ahorraremos bastante tiempo durante el entrenamiento de los modelos.

Para ver al modelo en acción, vamos a representar dos semanas del conjunto test con sus temperaturas reales frente a las predicciones realizadas por el modelo para esos mismos datos. Para ello, primero se instancia el generador para el conjunto test y se le hace una llamada para obtener los datos que servirán de entrada para nuestro modelo, que será un array de dimensiones (2016 x 240 x 14) donde:

- 2016 son el número de instantes a predecir,
- 240 son los lookbacks (1440) divididos entre los steps (6) para tener las series temporales con registros de hora en hora,
- y 14 son las variables que intervienen el modelo, incluyéndose aquí la variable a predecir y únicamente desechando las variables tiempo.

Una vez que se hayan generado estos datos de entrada, finalmente se realizan las predicciones.

Después de obtenerlas, las enfrentaremos en un gráfico donde vendrán representadas en rojo contra los valores reales en azul.

```
## 1º Definimos generador para el conjunto test:

  batch_size_plot <- 2016 # 14 días
  lookback_plot <- lookback
  step_plot <- 6

  test_gen <- generator(
    data,
    lookback = lookback_plot,
    delay = 0,
    min_index = 300001,
    max_index = 315250,
    shuffle = FALSE,
    step = step_plot,
    batch_size = batch_size_plot
  )

  test_gen_data <- test_gen()

  V1 = seq(1, length(test_gen_data[[2]]))

  plot_data <-
    as.data.frame(cbind(V1, test_gen_data[[2]]))

  inputdata <- test_gen_data[[1]]
  dim(inputdata) <- c(batch_size_plot, lookback/step, 14)
```

```
## 2º Se realizan las predicciones para ese conjunto de datos:

  pred_out <- model %>%
    predict(inputdata)


### Representamos en un gráfico:

  plot_data <-
    cbind(plot_data, pred_out)

 p <-
    ggplot(plot_data, aes(x = V1, y = V2)) +
      geom_point(colour = "blue", size = 0.1,alpha=0.4)
  p <-
    p + geom_point(aes(x = V1, y = pred_out), colour = "red",
                   size = 0.1 ,alpha=0.4)

  p
```

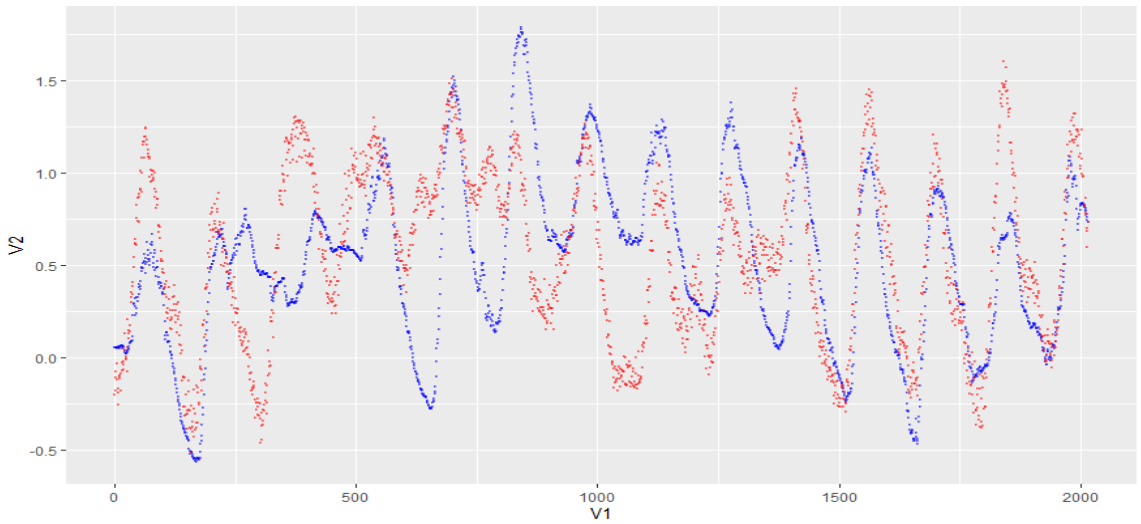


Figura 4.2: Predicción usando modelo feed-forward (conjunto test)

Calculemos ahora la métrica MAE:

```
mean(abs(plot_data$pred_out - plot_data$V2))
```

Se ha obtenido un error absoluto medio de 0.36. Como antes hemos normalizado los datos, no podemos interpretar este dato directamente. Para ello:

```
0.36 * std[[2]]
```

Se traduce a un error absoluto medio de 3.19ºC, el cual es bastante grande. Trataremos de mejorar este error de partida a lo largo de la práctica.

GRU

El hecho de que los resultados del anterior modelo de red neuronal no sean del todo buenos, no significa que este problema no pueda abarcarse usando Machine Learning. De hecho, el bajo rendimiento podemos achacárselo a la falta de consideración por el modelo anterior de que nuestros datos son secuenciales. Si queremos tener en cuenta la dimensión "tiempo", debemos usar redes recurrentes.

Podríamos empezar aplicando una capa simple de red recurrente, pero ya vimos en teoría los problemas que tenía en la práctica que tanto la limitaban. Otra opción sería utilizar una LSTM, pero ya se vio anteriormente que en términos de eficiencia es preferible usar una GRU, por lo tanto comenzaremos implementando una capa de este último.

Para construir el siguiente modelo de red recurrente hemos hecho uso de la función `keras_model_sequential()`, que se encarga de crear un modelo secuencial vacío en el que se irán apilando las distintas capas . Está formada por una capa GRU seguida de varias otras densas.

Para compilar el modelo usamos `compile()`, indicando que nuestro optimizador será el RMSprop, que en resumidas cuentas es una variante del algoritmo de descenso del gradiente, y que la función de pérdida será el error absoluto medio (MAE, por sus siglas en inglés).

Una vez construido el modelo, para su entrenamiento utilizamos la función `fit()`, haciendo uso del generador de datos de entrenamiento y el de validación generados anteriormente. Debido a que se han explicado previamente los argumentos de esta función, el único que no hemos visto es `use_multiprocessing = T` , que beneficiará al entrenamiento cuando tengamos generadores de datos como entrada del modelo.

```
# 1) Construcción del modelo:
model <- keras_model_sequential() %>%
  layer_gru(units = 64, input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "relu")


# 2) Compilamos:
model %>% compile(optimizer = optimizer_rmsprop(),
                  loss = "mae")


# 3) Entrenamiento:
history <- model %>% fit(
  train_gen,
  batch_size = 120,
```

```
  steps_per_epoch = 500,
  epochs = 20,
  use_multiprocessing = T,
  validation_data = val_gen,
  validation_steps = val_steps,
  callbacks = callback_early_stopping(monitor = "val_loss",
                                      patience = 3,
                                      mode = "auto",
                                      restore_best_weights = TRUE)

)
```

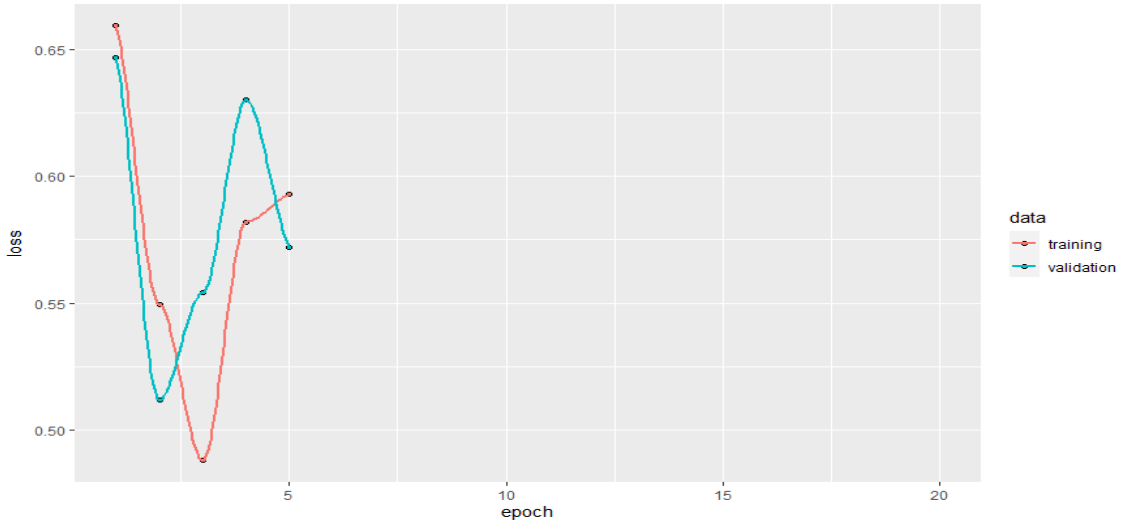


Figura 4.3: Curvas de pérdida usando modelo con una capa GRU

A partir de la tercera etapa, podemos observar que nuestro modelo comienza a sobreajustarse, es decir, está ajustándose demasiado a los datos de entrenamiento y eso provoca que al probar a predecir un nuevo conjunto de datos no obtenga buenos resultados.

Veamos las predicciones que realiza este modelo para 14 días del conjunto test:

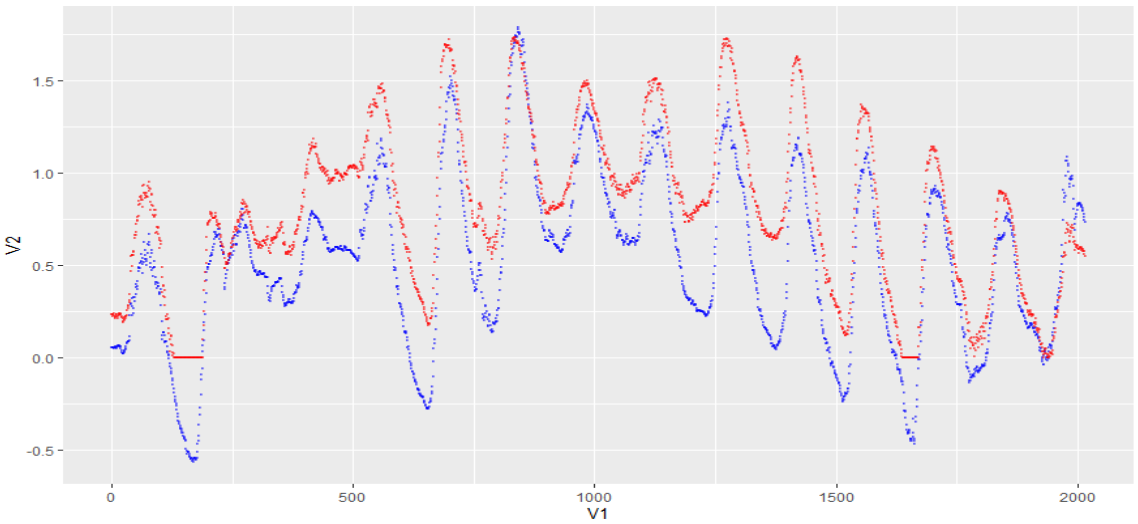


Figura 4.4: Predicción usando modelo GRU (conjunto test)

A simple vista parecen haber mejorado, pero para comprobarlo hemos calculado el error absoluto medio que comete, y éste es de 0.29, que podríamos traducirlo a 2.57 ºC. Por tanto, ha habido una mejora importante respecto al anterior modelo.

Como hemos visto antes, el modelo estaba sufriendo un problema muy frecuente en Machine Learning: el overfitting o sobreajuste. Para intentar solucionarlo, vamos a usar la técnica del dropout.

GRU usando dropout

El dropout es una técnica en la que se seleccionan neuronas aleatoriamente para que sean ignoradas durante el entrenamiento, por tanto, no participan en la propagación hacia delante ni hacia atrás.

Los pesos de las neuronas se ajustan a características específicas, lo que podríamos decir que les da cierta especialización. Las neuronas vecinas llegan a depender de esa especialización, lo que a mayor escala puede dar lugar a un modelo frágil y demasiado especializado para los datos de entrenamiento.

Entonces, al eliminar esas neuronas de forma aleatoria, hacemos que el modelo tenga que adaptarse a esos cambios y de esa forma se vuelva menos sensible a los pesos específicos de las neuronas. Esto a su vez nos lleva a conseguir una red con mayor capacidad de generalización y por tanto menos propensa al sobreajuste.

En el caso de las redes recurrentes, esta técnica sufre algunas variaciones, ya que las neuronas seleccionadas aleatoriamente deben ser ignoradas en todos los timesteps. No podemos ignorarlas en unas sí y en otras no, con tal de preservar el buen entrenamiento del modelo. Por eso, toda capa recurrente en Keras tiene dos parámetros para controlar esta técnica: `dropout` , decimal que especifica el ratio de dropout para las unidades de entrada de la capa, y `recurrent_dropout`, que especifica el ratio de dropout de las unidades recurrentes.

```
model <- keras_model_sequential() %>%
  layer_gru(units = 64, dropout = 0.2, recurrent_dropout = 0.2,
            input_shape = list(NULL, dim(data)[[-1]])) %>%
```

```
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "relu")

model %>% compile(optimizer = optimizer_rmsprop(),
                  loss = "mae")

summary(model)

history <- model %>% fit(
  train_gen,
  batch_size = 120,
  steps_per_epoch = 500,
  epochs = 20,
  use_multiprocessing = T,
  validation_data = val_gen,
  validation_steps = val_steps,
  callbacks = callback_early_stopping(monitor = "val_loss",
                                      patience = 4,
                                      mode = "auto",
                                      restore_best_weights = TRUE)

)
```

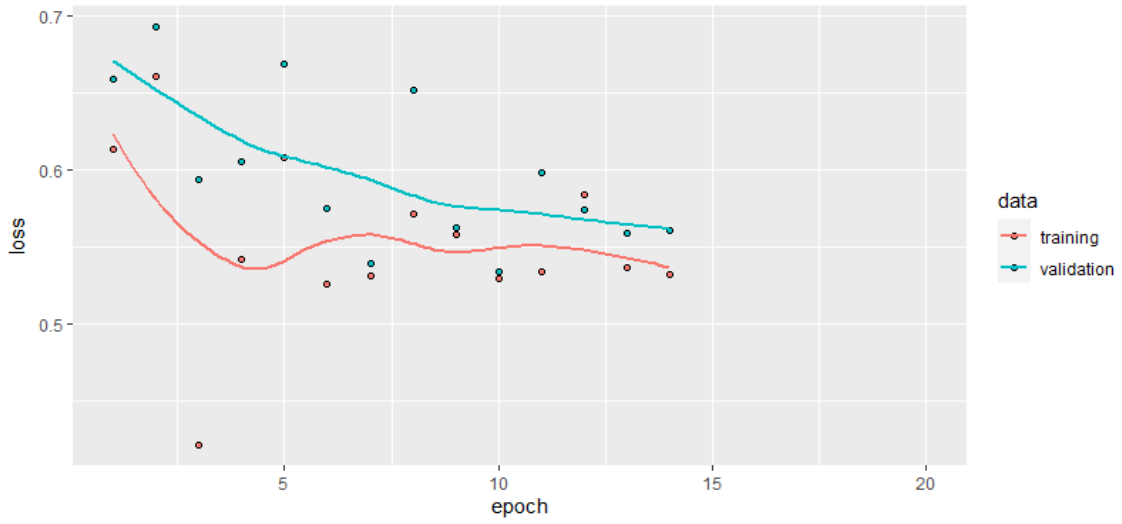


Figura 4.5: Curvas de pérdida usando modelo con una capa GRU usando dropout

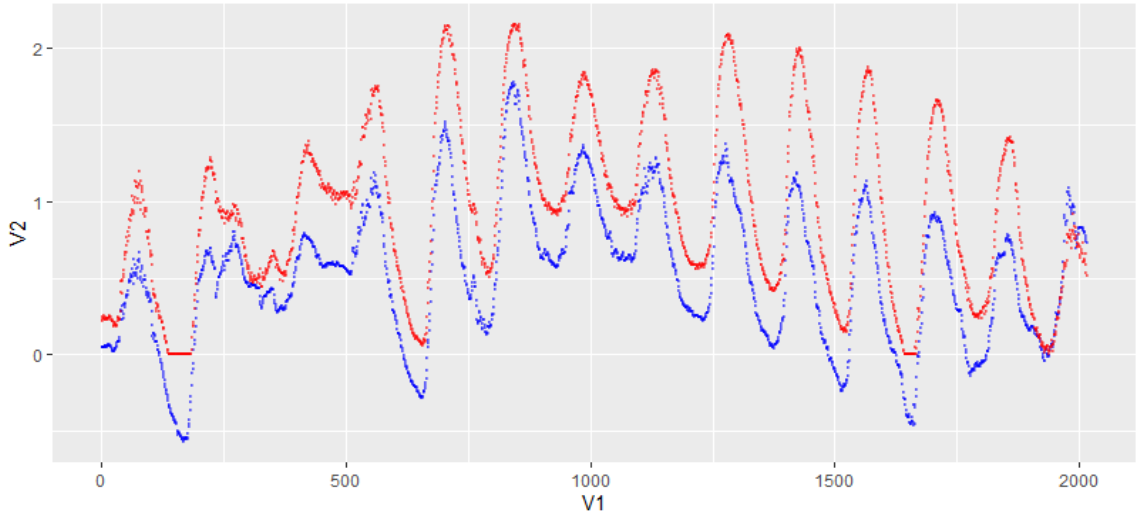


Figura 4.6: Predicción usando modelo GRU con dropout (conjunto test)

Aunque parece que hemos solucionado el problema del overfitting, obtenemos un MAE de 0.43, que se traduce a 3.81 ºC. Así que después de aplicar dropout, no hemos conseguido mejorar al modelo anterior. Procedemos a probar con otras técnicas.

Apilamiento de capas recurrentes

Llegado a este punto, puede ser interesante probar a aumentar la capacidad de nuestra red. Para hacerlo, pasaremos a usar más capas. El apilamiento de capas recurrentes es una buena forma de mejorar el rendimiento de una RNN.

Para realizar este apilamiento en Keras, todas las capas intermedias deberán devolver toda la secuencia de salidas en lugar de la salida para el último instante de tiempo. Esto lo hacemos indicando `return_sequences = TRUE` .

Para construir este modelo hemos apilado dos capas recurrentes GRU con 32 y 64 neuronas, respectivamente., acompañadas de una capa densa.

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32,
            dropout = 0.2,
            recurrent_dropout = 0.5,
            return_sequences = TRUE,
            input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_gru(units = 64, activation = "relu",
            dropout = 0.2,
            recurrent_dropout = 0.5) %>%
  layer_dense(units = 1)


model %>% compile(optimizer = optimizer_rmsprop(),
                  loss = "mae")

summary(model)
```

```
history <- model %>% fit(
  train_gen,
  batch_size = 120,
  steps_per_epoch = 500,
  epochs = 20,
  use_multiprocessing = T,
  validation_data = val_gen,
  validation_steps = val_steps,
  callbacks = callback_early_stopping(monitor = "val_loss",
                                      patience = 4,
                                      mode = "auto",
                                      restore_best_weights = TRUE)

)
```

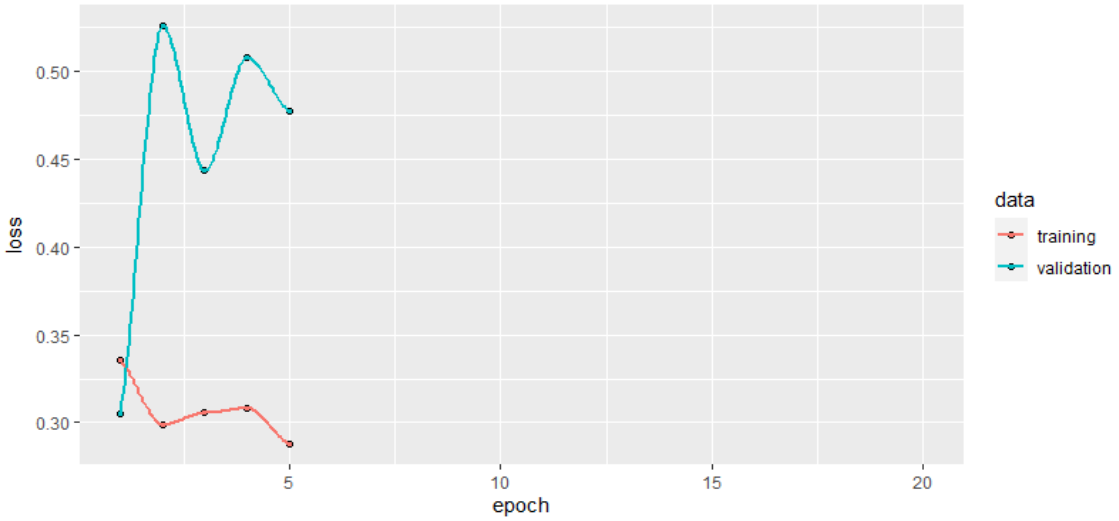


Figura 4.7: Curvas de pérdida usando modelo con dos capas GRU apiladas

Antes de empezar el modelo a sobreajustarse, en la primera etapa ha conseguido un error medio absoluto de 0.3 aproximadamente. Éste es el mejor resultado que hemos obtenido hasta ahora.

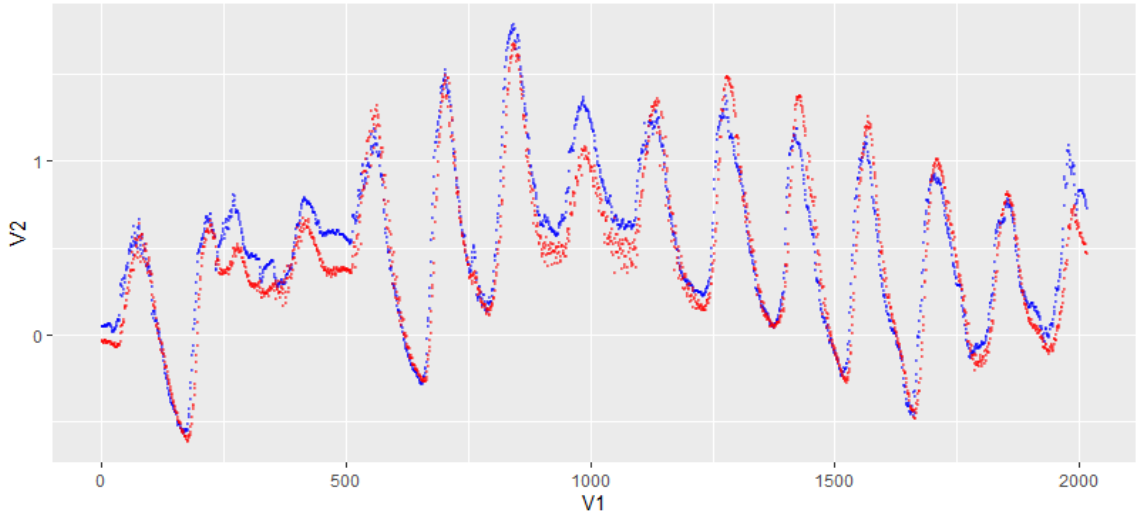


Figura 4.8: Predicción usando modelo con dos capas GRU apiladas (conjunto test)

Si comparamos con los anteriores modelos, las predicciones se ajustan mucho mejor a los valores reales.

Como MAE para la predicción de esos 14 días del conjunto test, hemos obtenido 0.13, que traducido sería un error absoluto medio de 1.15 ºC. Se podrían probar otras configuraciones, pero vamos a considerar este resultado como aceptable.

Una vez dada por terminada la práctica, se concluye que el utilizar redes recurrentes para modelizar series de tiempo nos proporciona grandes resultados frente a los que obtendríamos con otras arquitecturas.

Capítulo 5

Conclusiones

En este trabajo se ha realizado una introducción teórica a las redes neuronales recurrentes, y posteriormente se ha probado su aplicación en una serie temporal.

Dentro de la teoría, se ha comparado a estas redes con las redes feed-forward con el objetivo de diferenciar ambas arquitecturas y los campos en los que cada una destaca, como son los datos secuenciales para las redes recurrentes. Luego se ha pasado a explicar el funcionamiento de las RNN, en cuanto a su estructura y su entrenamiento. Finalmente se han planteado los problemas que tienen este tipo de redes y después cómo solventarlos.

En cuanto a la palicación práctica, para tener un punto de partida o una meta que superar, hicimos predicciones usando un modelo de red feed-forward, sabiendo de antemano que los resultados iban a ser muy pobres. Al pasar a usar un modelo de red recurrente, en concreto una red que usaba una capa GRU, los resultados mejoraron bastante, demostrando así todo el rendimiento que nos aporta usar este tipo de redes cuando tratamos con series temporales. A partir de esa primera red recurrente básica, implementamos la técnica dropout para intentar resolver el famoso problema del overfitting, aunque no conseguimos mejorar el modelo. No abandonamos la idea de seguir aplicando dropout, pero este caso implementando una técnica que suele funcionar muy bien para redes recurrentes, que es el apilamiento de capas. Fusionando estas dos técnicas llegamos al modelo con el mejor rendimiento de todos.

Después de realizar los pasos descritos y de experimentar con distintos valores de parámetros, quedó demostrado que el usar una red recurrente es beneficioso a la hora de hacer pronósticos de series temporales. Además, descubrimos que el combinar la técnica del dropout con el apilamiento de capas recurrentes mejora bastante el modelo. Aunque existen muchas más posibilidades para mejorar el modelo, consideramos que se han conseguido resultados lo suficientemente buenos y sin llegar a complejidades excesivas en cuanto a conocimiento sobre computación ni utilizar un equipo extremadamente potente.

Tanto la parte teória como la práctica dejan abiertas muchas líneas futuras de trabajo, como por ejemplo estudiar el uso combinado de redes convolucionales y redes recurrentes, o realizar una aplicación práctica que no trate series temporales, como puede ser una enfocada a la traducción del lenguaje.

Bibliografía

[1] Abadi, Martín et al. (2015). «TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems».
http://download.tensorflow.org/paper/whitepaper2015.pdf.

[2] Allaire, JJ y Chollet, François (2023). *keras: R Interface to 'Keras'*.
https://tensorflow.rstudio.com/. R package version 2.11.1.

[3] Beysolow II, Taweh (2017). *Introduction to Deep Learning Using R*. Apress.

[4] Chollet, François y Allaire, J. J. (2018). *Deep Learning with R*. Manning.

[5] Ghatak, Abhijit (2019). *Deep Learning with R*. Springer.

[6] Kostadinov, Simeon (2017). «Understanding GRU Networks». Disponible en https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

[7] Luque Calvo, Pedro L. (2017). *Escribir un Trabajo Fin de Estudios con R Markdown*.
http://destio.us.es/calvo.

[8] Olah, Christopher (2015). «Understanding LSTM Networks». Disponible en http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[9] RStudio Team (2020). *RStudio: Integrated Development Environment for R*.
http://www.rstudio.com/.

[10] Sotaquirá, Miguel (2019). «Redes Neuronales Recurrentes: Explicación Detallada». Disponible en https://www.codificandobits.com/blog/redes-neuronales-recurrentes-explicacion-detallada/.

[11] Torres, Jordi (2019). «Redes Neuronales Recurrentes». Disponible en https://torres.ai/redes-neuronales-recurrentes/.