Redes Neuronales LSTM y Aplicaciones en Datos de Secuencia

Arantza Ivonne Pineda Sandoval, Ana Luisa Masetto Herrera, Alexis Solis Cancino

2018-12-09

1 Introducción – El Problema de Procesamiento Secuencial

Los problemas de aprendizaje profundo (*deep learning*) para procesamiento secuencial, por ejemplo procesamiento de texto (entendidos como secuencias de palabras o caracteres) y series de tiempo han adquirido relevancia desde hace mucho tiempo ya que son considerados por muchos unos de los problemas más complejos a ser resueltos. Éstos involucran temas de una amplia gama de industrias y ramas de la ciencia. Sin embargo, gracias al veloz desarrollo en el área de aprendizaje de máquina, se ha descubierto que para una gran parte de estos problemas, las redes recurrentes tipo Memoria Corto-largo plazo son las más efectivas. Las razones son múltiples, en especial, la propiedad que tienen de recordar patrones de manera selectiva a lo largo del tiempo es aquello que atrae nuestro interés para este proyecto.

En este proyecto se presentará el método LSTM[1] (Memoria Corto-Largo plazo), se detallan las implicaciones teóricas del método así como las estrategias adecuadas para procesar y validar los datos y se ejemplifica lo anterior con dos aplicaciones desarrolladas en el lenguaje de `R`. La primera aplicación permite realizar clasificación de sentimiento en reseñas de películas con la base llamada *IMDB* (del paquete `keras`); se comparan diferentes redes LSTM y se extraen conclusiones generales respecto al desempeño. La segunda aplicación permite realizar predicciones para series de tiempo; incorpora un análisis exploratorio de los datos para identificar patrones de comportamiento y autocorrelaciones así como estrategias de validación en series de tiempo para analizar la base de datos: *Daily Minimum Temperatures in Melbourne, Australia, 1981-1990*[2]. Con los resultados anteriores, se realizan las predicciones futuras en el horizonte de un año y se evalúa el desempeño de la red. De este modo se concluirá el proyecto explicando la utilidad de este tipo de redes, ventajas y limitaciones de acuerdo a los resultados obtenidos en ambas aplicaciones.

[1] por las siglas en inglés: Long-Short-Term Memory

[2] (Hyndman, R. sf)

Es necesario definir en primer lugar los elementos de entrada que serán utilizados. Para la primer aplicación de las redes LSTM, el enfoque se encuentra en el tema de clasificación de sentimiento en textos (positivo o negativo) que ejemplifica una de la formas más extendidas para datos sequenciales. Aunque este tipo de modelos (redes recurrentes y su

especificación con redes LSTM) aprenden del lenguaje, no lo entienden de la misma manera que un humano, sino que más bien permiten estructurar el lenguaje de tipo escrito e identificar patrones.

Para la segunda aplicación, se ha elegido el uso de series de tiempo como entrada a las redes. Las series de tiempo o temporales son una forma de agrupar los datos de manera cronológica. A lo largo de la historia han permitido realizar análisis de la evolución en el comportamiento de distintos procesos con base en los registros de datos históricos para su posterior interpretación. Éstas permiten capturar elementos como patrones y correlaciones con el fin de entender y predecir el comportamiento de estos procesos. Han capturado la atención de científicos y expertos en distintas áreas de conocimiento porque otorgan la posibilidad de generar predicciones a futuro de maneras más certeras. Uno de los retos principales para su implementación en este proyecto será su validación, sin embargo, a través de la validación tipo *backtesting* se prueba una buena técnica.

Debido a que el tema aquí presentado es una forma alternativa de las redes neuronales tradicionales, es necesario recordar que en aprendizaje supervisado, el desempeño de las redes tradicionales está basado en la creación de estructuras interconectadas llamadas neuronas (unidades de procesamiento) que reciben un conjunto de ciertas entradas, las ponderan e integran para finalmente transmitir el resultado a otras neuronas conectadas a ella. Para las redes neuronales recurrentes y sus especificaciones, la arquitectura de las redes tendrá ciertos cambios que se detallarán en las siguientes secciones.

2 Redes Neuronales Recurrentes (*RNN – Recurrent Neural Networks*)

En aprendizaje de máquina se han estudiando previamente las redes neuronales, sin embargo, al tratar datos secuenciales, puede resultar más apropiado utilizar redes que permitan incorporar memoria para capturar los diferentes estados en el tiempo.

Hay básicamente 2 maneras de poner memoria en una red neuronal: con retrasos (redes con retrasos) y con recurrencia (redes recurrentes). El enfoque de este proyecto será en las redes neuronales recurrentes (como introducción al modelo LSTM), ya que éstas son una herramienta muy apropiada para modelar series de tiempo.

Una manera sencilla de entender estas redes es con el siguiente ejemplo. De un momento a otro, nuestro cerebro funciona como una función: acepta

entradas de nuestros sentidos (externos) y nuestros pensamientos (internos) y produce resultados en forma de acciones (externos) y nuevos pensamientos (internos). Por ejemplo, si estamos en un bosque y de pronto vemos un oso, primero ocurre la acción de verlo y luego pensamos "oso". Podemos modelar este comportamiento con una red neuronal *feedforward*: podemos enseñar a una red neuronal *feedforward* a pensar "oso" cuando se muestra una imagen de un oso.

Aunque nuestro ejemplo del "oso" puede parecer burdo, creémos que ejemplifica bien el concepto básico.

Pero nuestro cerebro no es una función de una única ejecución. Se ejecuta repetidamente a través del tiempo. Por ejemplo, vemos un oso, luego pensamos "oso", luego pensamos "correr". Es importante destacar que la misma función que transforma la imagen de un oso en el pensamiento "oso" también transforma el pensamiento "oso" en el pensamiento "correr" . Es una función recurrente, que podemos modelar con una *red neuronal recurrente* (RNR).

Las redes recurrentes[3] son redes neuronales que aplican la misma operación a cada elemento de una secuencia de datos de entrada (texto, voz, video, etc); y cuya salida depende tanto de los datos de entrada presentes como de los pasados incorporando un estado variante en el tiempo. Se trata de redes con una arquitectura que implementa una cierta memoria, es decir, permiten capturar un sentido temporal.

[3] Desarrolladas por John Hopfiels en 1982

Se trata de una composición de redes neuronales *feedforward* idénticas, una para cada momento, o estado del tiempo, a la que nos referiremos como "célula RNR". Estas células operan sobre su propia salida (*output*), lo que les permite ser compuestas. También pueden operar con entrada externa (*external input*) y producir salida externa (*external output*).

La siguiente grafica muestra el funcionamiento de una red recurrente simple:

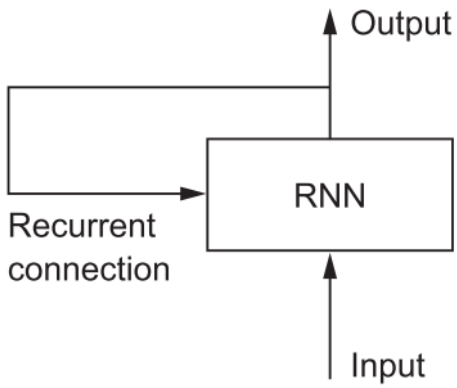


Figure 1: Una red recurrente: una red con un 'loop'.

Con lo anterior, si ahora se quiere entender el funcionamiento de una sola célula RNN, se presenta el diagrama siguiente:

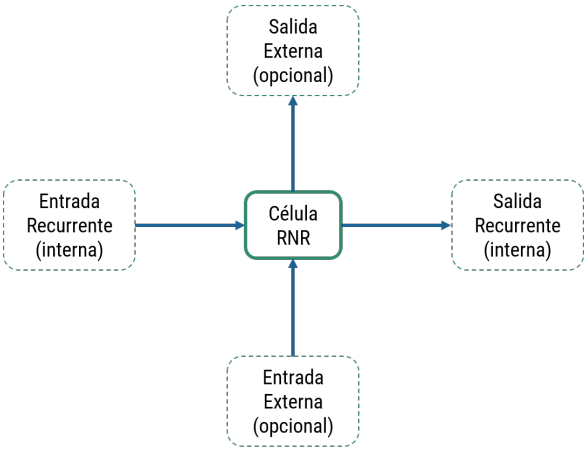


Figure 2: Celda de una red neuronal recurrente – acepta como inputs el estado anterior y la entrada actual y da como output un nuevo estado y una salida actual

Aquí hay un diagrama de tres células de RNR compuestas:

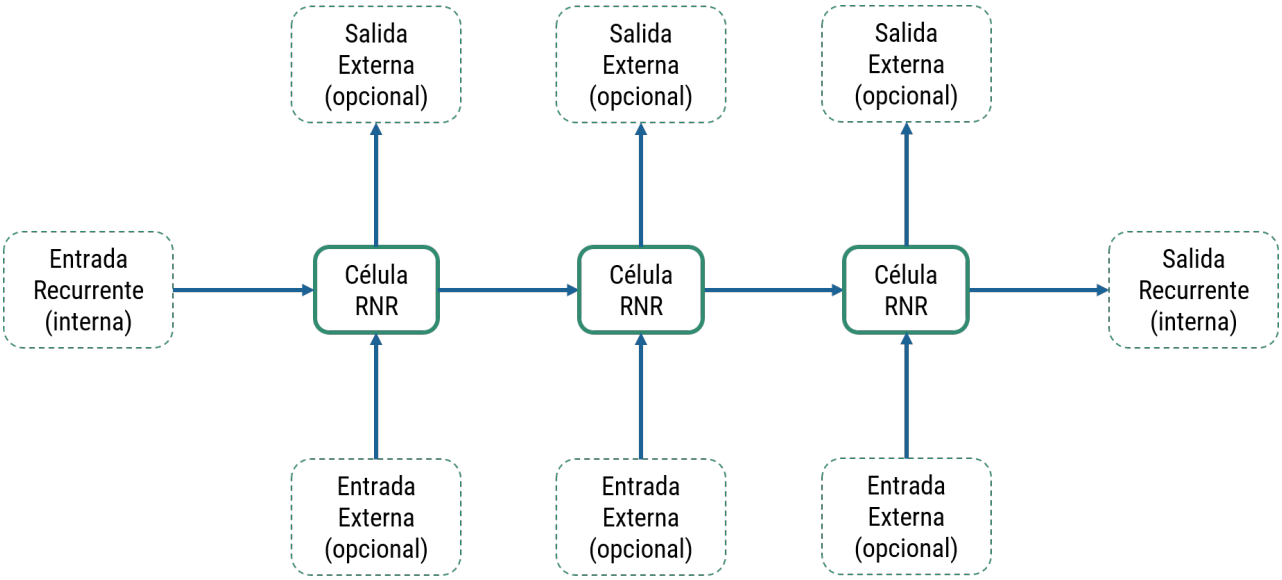


Figure 3: El procesamiento de la información de una red neuronal recurrente puede 'desdoblarse' como se muestra en esta imagen.

Se puede pensar en las salidas recurrentes como un "estado" que se pasa al siguiente período de tiempo. Por lo tanto, una celda de RNR acepta un estado anterior y una entrada actual (opcional) y produce un estado actual y una salida actual (opcional). Ellas procesan los datos uno a la vez, manteniendo en sus capas ocultas el vector de estado que contiene información acerca de la historia

de todas las entradas pasadas. Algunas neuronas reciben como entrada la salida de una de las capas e inyectan su salida en una de las capas de un nivel anterior a ella.

La manera en que funcionan, es que las RNRs aceptan como entrada un vector $\vec{x}$, tienen internamente un vector de estados $\vec{h}$, y combinando $\vec{x}$ y $\vec{h}$ en una función (fija pero aprendida), producen como salida un vector $\vec{y}$. De este modo, la salida esta influenciada no sólo por la entrada ($\vec{x}$), sino por toda la historia de las entradas que hubo en el pasado ($\vec{h}$).

A continuación se muestra el funcionamiento de la célula de RNR matemáticamente:

$$\begin{pmatrix} s_t \\ o_t \end{pmatrix} = f \begin{pmatrix} s_{t-1} \\ x_t \end{pmatrix}$$

donde:

- s_t y s_{t-1} son los estados actual y anterior, respectivamente.
- o_t es la salida actual (podría ser vacía).
- x_t es la entrada actual (podría ser vacía).
- f es una función de recurrencia.

Ahora, retomando el ejemplo del cerebro, pensemos que nuestro cerebro funciona en un mismo lugar: es decir, la actividad neuronal actual reemplaza a la actividad neuronal pasada. También podemos ver que las RNR operan de esta forma: como las celdas de RNR son idénticas, todas pueden verse como el mismo objeto, con un "estado" de la celda que se sobrescribe en cada paso del tiempo. Esto puede expresarse gráficamente como:

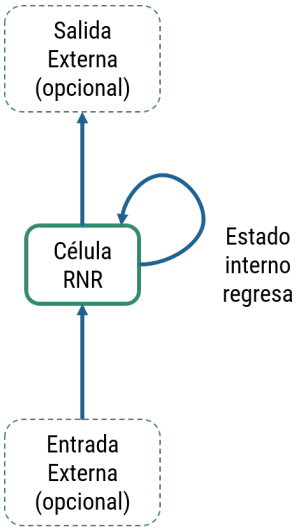


Figure 4: Representación 'corta' de una celda de red neuronal recurrente

2.1 Capacidades de las RNR; elección del período o segmento de tiempo

La red neuronal recurrente descrita anteriormente es muy general. En teoría, puede hacer cualquier cosa: si le damos a la red neuronal dentro de cada celda al menos una capa oculta, cada celda se convierte en un aproximador de cierta función. Esto significa que una celda RNN puede emular cualquier función, de la que se deduce que una RNR podría en teoría, emular perfectamente cualquier relación entre ciertas variables.

Usando la analogía del cerebro, todo lo que tenemos que hacer para ver cómo podemos usar un RNR para manejar cierta tarea es preguntar cómo un humano manejaría la misma tarea. Consideremos por ejemplo, la traducción del inglés al español. Un humano lee una oración en inglés ("the dog licked the rug"), se detiene y luego escribe la traducción en español ("el perro lamió el tapete"). Para emular este comportamiento con una RNR, la única elección que debemos tomar

(además del diseño de la celda RNR en sí, que por ahora consideramos como una “caja negra”) es decidir cuál será el segmento de tiempo, lo que determina la forma de las entradas y salidas; o dicho de otra forma, determina cómo la RNR “interactúa con el mundo externo”.

Una opción es configurar el intervalo de tiempo de acuerdo con el contenido. Es decir, podríamos usar la oración completa como un paso de tiempo, en cuyo caso nuestra RNN es solo una red de avance:

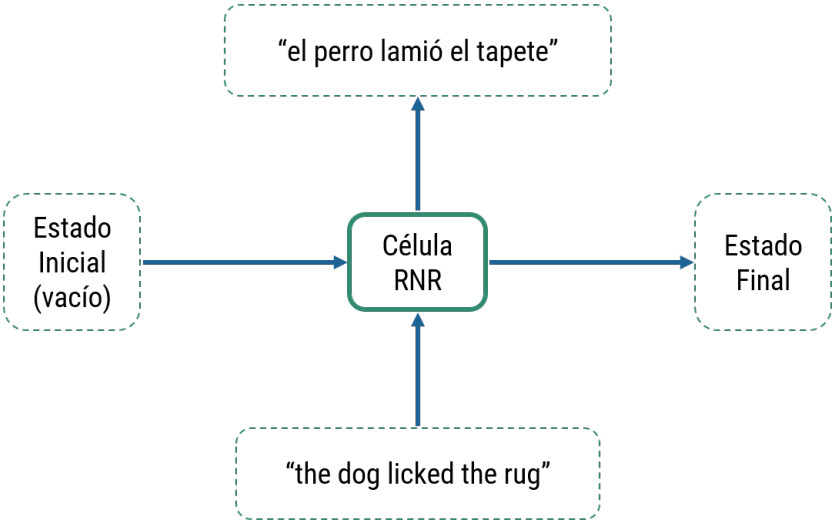


Figure 5: Una red neuronal recurrente cuyo input es toda una oración.

El estado final no importa cuando se traduce una sola oración. Sin embargo, podría importar si la oración formaba parte de un párrafo que se estaba traduciendo, ya que contendría información sobre las oraciones anteriores. Cabe notar que el estado inicial está indicado como vacío, pero al evaluar secuencias individuales, puede ser útil considerar el estado inicial como una variable. Puede ser que la mejor representación para el estado inicial en no sea el vacío.

Alternativamente, podríamos decir que cada palabra es un período de tiempo. A continuación se ilustra cómo podría verse una RNR traduciendo “the dog licked” con base en un período de tiempo igual a una palabra:

Después del primer paso de tiempo, el estado de la RNR contiene una representación interna de “the”; después del segundo paso, el estado cambia a “the dog”; después del tercer paso, el estado cambia a “the dog licked”. La red no produce ninguna salida en los primeros tres pasos de tiempo. Comienza a producir salidas cuando recibe una entrada en blanco, momento en el cual sabe

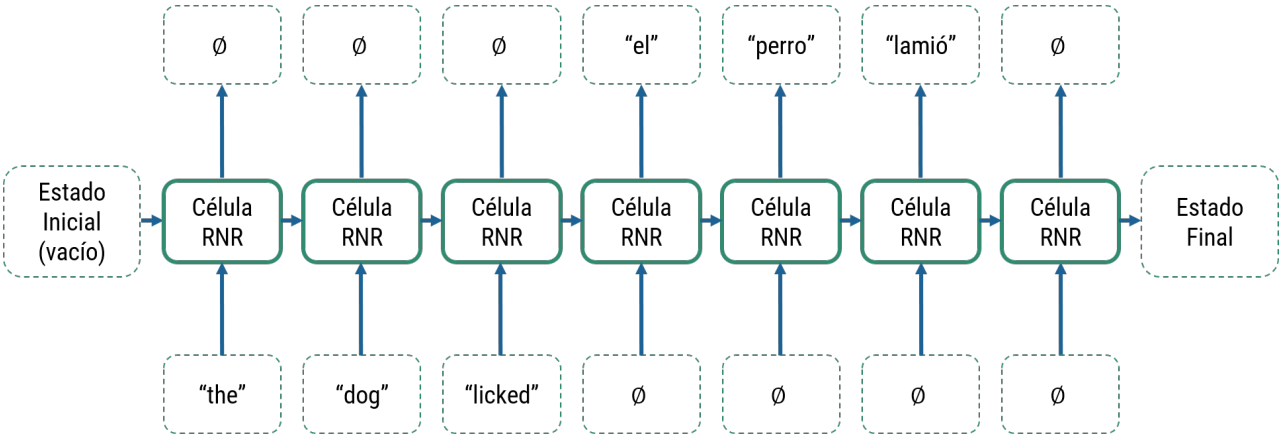


Figure 6: RNR cuyo input en cada paso de tiempo es igual a una palabra.

que las entradas han terminado. Cuando termina de producir salidas, produce una salida en blanco para indicar que ha finalizado.

La determinación de cada período de tiempo no tiene que estar basada en el contenido (en este caso, en las palabras); podría estar basada en una unidad de tiempo real. Por ejemplo, podríamos considerar que el paso del tiempo es de un segundo y aplicar una velocidad de lectura de 5 caracteres por segundo[4].

[4] Las entradas para los primeros tres pasos de tiempo entonces serían: i) `the d`; ii) `og li`; y iii) `cked`.

2.2 La RNR básica

Ahora que hemos cubierto el panorama general, analicemos a detalle la célula de una RNR. La célula RNR más básica es una red neuronal de una sola capa, cuya salida (*output*) se utiliza tanto como salida actual (externa), y también como estado actual de la célula:

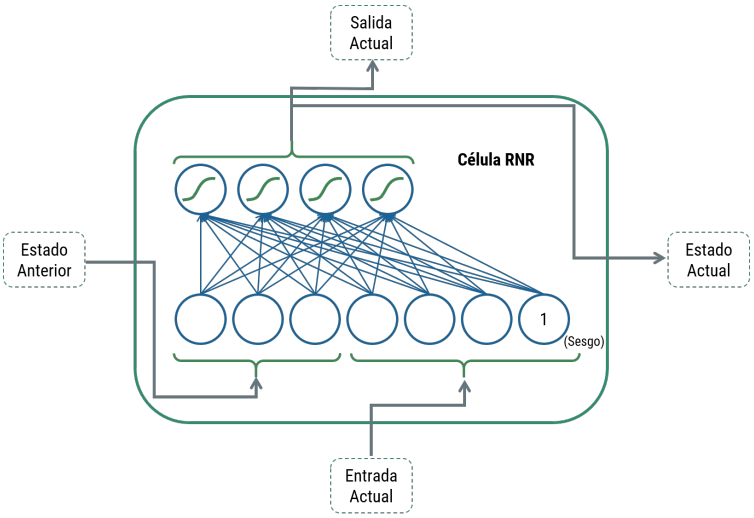


Figure 7: Red Neuronal Recurrente básica (‘plain vanilla’).

Observemos que el vector de estado anterior tiene el mismo tamaño que el vector de estado actual. Esto es una característica primordial para la composición de las células RNR. Aquí está la descripción algebraica de la célula RNR básica:

$$s_t = \phi(W\, s_{t-1} \,+\, U\, x_t \,+\, b)$$

donde:

- ϕ es la función de activación (e.g. tanh, ReLU),
- $s_t \in \mathbb{R}^n$ es el estado actual de la célula (y también la salida actual),
- $s_{t-1} \in \mathbb{R}^n$ es el estado anterior,
- $x_t \in \mathbb{R}^m$ es la entrada actual,
- $W \in \mathbb{R}^{n \times n}$, $U \in \mathbb{R}^{m \times n}$ son los pesos,
- $b \in \mathbb{R}^n$ son los sesgos,
- n y m son los tamaños de estado y de entrada.

Incluso esta célula RNN básica es bastante poderosa. Aunque una sola celda no cumple con los criterios para la aproximación universal de cualquier función, se sabe que una serie de celdas RNR básicas compuestas puede implementar cualquier algoritmo (Siegelmann y Sontag (1992)[5].

[5] http://binds.cs.umass.edu/papers/1995_Siegelmann_JComSysSci.pdf

Esto es bueno, en teoría, pero hay un problema en la práctica: entrenar RNRs con un algoritmo de propagación hacia atrás (*backpropagation*) resulta bastante difícil, incluso más que entrenar redes neuronales muy avanzadas. Esta dificultad se debe a los problemas de: i) transformación de información[6] ; y ii) desvanecimiento o crecimiento de la sensibilidad del modelo[7]. Ambos problemas son causado por la aplicación repetida de la misma función no lineal.

[6] en inglés es denominado: *information morphing*.

[7] denominado *explosión* o *desvanecimiento* del gradiente.

Es importante destacar que las redes recurrentes normales, van olvidando en su estado la historia más lejana (problema del vanishing gradient). Existe un tipo de redes específico, las redes de Long-Short Term Memory (LSTM) que a diferencia de las anteriores, sí pueden recordar en su estado, historias lejanas. Justamente estas redes son el enfoque principal del proyecto que se detallará en los siguientes apartados.

2.3 Transformación de información y el Problema de Degradación

Si la información se transforma constantemente, es difícil explotar la información pasada correctamente cuando la necesitamos. Con cada paso de

tiempo, la función no lineal crea distintas representaciones de los datos. La mejor representación de la información puede haber ocurrido en algún momento en el pasado. Además de aprender a explotar la información actual (si existiera en su forma original y utilizable), también debemos aprender a descifrar el estado original a partir del estado actual, si es posible. Esto lleva a un aprendizaje difícil y consecuentemente, a malos resultados. A esto se le denomina el *problema de degradación*.

Es muy fácil demostrar que la transformación de información ocurre en una RNR básica. De hecho, supongamos que es posible que una célula RNR mantenga su estado anterior completamente en ausencia de entradas externas. Entonces $F(x) = \phi(W s_{t-1} + b)$ es la función de identidad con respecto a s_{t-1}. Pero la función de identidad es lineal y $F(x)$ no es lineal por construcción, lo cuál implica una contradicción. Por lo tanto, una célula RNR inevitablemente transforma la información de un estado en cierto tiempo al siguiente. Incluso la tarea trivial de generar $s_t = x_t$ es imposible para una RNR simple. Esta es la razón principal por la que tenemos el problema llamado *problema de degradación* (ver He et al., 2015)[8].

[8] https://arxiv.org/abs/1512.03385.

2.4 Desvanecimiento o Crecimiento de la Sensibilidad de un RNR

El problema principal que enfrentan las redes neuronales recurrentes es que el estado de la red cambia constantemente y, encima de eso, puede que el estado de la red sea extremadamente sensible (o bien, el otro extremo: completamente insensible) a cambios pasados: los efectos pasados aumentan exponencialmente (crecimiento de sensibilidad), o bien, se degradan (desvanecimiento de sensibilidad) con cada paso en la red. Estos problemas también están presentes en la redes neuronales de avance (*feedforward*): cuando se agregan cada vez más capas en estas redes, se vuelven no-entrenables.

2.5 Gradientes que Explotan o se Desvanecen

El punto anterior (sensibilidad) es una manera fácil de explicar el problema de entrenar una RNR con el método de propagación hacia atrás (*backpropagation*). Pero la propagación hacia atrás es un algoritmo basado en gradientes, y la "sensibilidad" que se desvanece o explota es sólo otra forma de decir gradientes que desaparecen y explotan (este de hecho es el término "formal" para describir este fenómeno). Si los gradientes explotan, no podemos entrenar a nuestra red. Por otro lado, si desaparecen, es difícil que la red "aprenda" las dependencias a largo plazo, ya que la propagación hacia atrás será demasiado sensible a los cambios de estado recientes. Esto dificulta el entrenamiento, como bien lo explican Bengio et al. (1994)[9].

[9] http://ai.dinfo.unifi.it/paolo/ps/tnn-94-gradient.pdf

Si nuestro gradiente explota, la propagación hacia atrás no funcionará porque obtendremos valores de `NaN` para el gradiente en las capas iniciales. Una solución fácil para esto es limitar el gradiente a un valor máximo, según lo propuesto por (Mikolov, 2012)[10] y reafirmado en (Pascanu et al., 2013)[11]. Esto funciona para prevenir los valores de `NaN` y permite que el entrenamiento continúe.

[10] http://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf
[11] http://proceedings.mlr.press/v28/pascanu13.pdf

Los gradientes que se desvanecen son más difíciles de resolver en las RNR simples. El enfoque sugerido en Pascanu et al. (2013)[12] es introducir un término de regularización que impone un flujo constante de errores hacia atrás. Esta es una solución fácil que parece funcionar para los pocos experimentos en los que se probó en dicha investigación. Desafortunadamente, es difícil encontrar una justificación de por qué esto debería funcionar todo el tiempo, porque estamos imponiendo una opinión sobre la forma en que los gradientes deben fluir en el modelo. Esta opinión puede ser correcta para algunas tareas, en cuyo caso nuestra imposición ayudará a lograr mejores resultados. Sin embargo, puede ser que para algunas tareas deseamos que los gradientes desaparezcan por completo, y para otras, puede ser que queramos que crezcan. En estos casos, el regularizador restaría valor al rendimiento del modelo y no parece haber ninguna justificación para decir que una situación es más común que la otra. Los LSTM evitan este problema por completo.

[12] http://proceedings.mlr.press/v28/pascanu13.pdf

3 Redes LSTM

Muy similar un mensaje cambiando a través de un juego de "teléfono descompuesto", la información se va perdiendo en las redes neuronales recurrentes con el paso del tiempo. Y por tanto, un pequeño cambio en el mensaje inicial puede tener un gran impacto sobre el mensaje final (o bien, podría no tener impacto alguno). ¿Cómo podemos proteger la integridad de los mensajes? Este es el principio fundamental de los LSTM: para garantizar la integridad de nuestros mensajes, los escribimos. Esto fue propuesto por Hocreiter y Schmidhuber (1997)[13] cuando introdujeron el concepto de redes LSTM en 1997. Estos investigadores se preguntaron: "¿cómo podemos lograr un flujo constante de errores a través de una sola unidad con una sola conexión a sí misma (es decir, ¿cómo podemos construir una red neuronal recurrente cuyo gradiente no crezca o disminuya)?"

[13] http://isle.illinois.edu/sst/meetings/2015/hochreiter-lstm.pdf

La respuesta, sencillamente, es evitar la transformación de información: los cambios en el estado de una red LSTM se escriben explícitamente, mediante una suma o resta explícita, de modo que cada elemento del estado permanezca constante sin interferencia externa: "la activación de la unidad tiene que permanecer constante ... esto se asegurará mediante el uso de la función de identidad". La mejor manera de imaginarlo es pensando en una cinta

transportadora funcionando en paralelo a la secuencia que se está procesando sobre la red. La información de la secuencia puede saltar en la cinta transportadora en cualquier punto, ser transportado a un paso de tiempo posterior y saltar intacta, cuando se necesite. Esto es esencialmente lo que hace LSTM: guarda información para más tarde, evitando así que las señales más antiguas se desvanezcan gradualmente durante el procesamiento. De manera matemática, esto se traduce en que los cambios del estado de la red deben de ser incrementales, esto es:

$$s_{t+1} = s_t + \Delta s_{t+1}$$

Ahora bien, Hocreiter y Schmidhuber se dieron cuenta que "escribir explícitamente los cambios de los estados" ya se había intentado previamente pero no funcionaba tan bien como se esperaba. Esto se debe a que al principio del entrenamiento de la red, comenzamos con inicializaciones aleatorias y nuestra red está realizando algunas "escrituras"" bastante aleatorias. Este es la dificultad principal que enfrentan las redes LSTM: "escritura de la información" de manera no coordinadas, particularmente al principio del entrenamiento cuando éstas son completamente aleatorias, crean un estado caótico que se traduce en malos resultados de entrenamiento.

Hochreiter y Schmidhuber reconocieron este problema, dividiéndolo en varios subproblemas, que denominaron "conflicto de pesos de entrada", "conflicto de pesos de salida", el "problema de abuso" y "deriva interna del estado". La arquitectura LSTM fue cuidadosamente diseñada para superar estos problemas, comenzando con la idea de selectividad.

3.1 Selectividad para Control y Coordinación de la Escritura

De acuerdo con la literatura anterior sobre LSTM, la clave para superar el desafío fundamental de las LSTM y mantener nuestro gradiente bajo control es ser selectivo en tres cosas: lo que escribimos, lo que leemos (porque necesitamos leer algo para saber qué escribir), y lo que olvidamos (porque la información obsoleta es una distracción y debe ser olvidada).

Parte de la razón por la que nuestro estado puede volverse tan caótico es que la RNR "escribe" en cada paso de tiempo. Esto es como si escribiéramos absolutamente todo lo que se dice en una clase: incluiríamos en nuestras notas información que no es relevante. Hochreiter y Schmidhuber describen esto como el "*conflicto de pesos de entrada*" (*input weight conflict*): Si la red está escribiendo los cambios de estado en cada paso de tiempo, recopilará una gran cantidad de información inútil, haciendo inutilizable su estado original. Por lo tanto, la RNR debe aprender a usar algunas de sus capas para cancelar algunos cambios de estado y "proteger" el estado original, lo que resulta en un

aprendizaje difícil. Podemos describir esta característica como “escritura selectiva”.

La segunda razón por la que nuestro estado puede volverse caótico es el reverso de la situación anterior: por cada escritura que se hace, la RNR lee desde todos los elementos del estado. Hochreiter y Schmidhuber describen esto como “*conflicto de peso de salida*” (*output weight conflict*): si todas las demás unidades leen unidades irrelevantes en cada paso de tiempo, producen un gran flujo de información irrelevante. Por lo tanto, la RNR debe aprender a usar algunas de sus unidades para cancelar la información irrelevante, lo que resulta en un aprendizaje difícil. Podemos describir esta característica como “lectura selectiva”.

La tercera forma de selectividad se relaciona con la forma en que desechamos la información que ya no es necesaria. Esta intuición no se introdujo en el documento LSTM original, lo que llevó al modelo LSTM original a tener problemas con tareas simples que involucraban secuencias largas. Fue introducido por Gers et al. (2000)[14]. Según esta publicación, en algunos casos, el estado del modelo LSTM original crecería indefinidamente, lo que eventualmente causaría que la red dejara de funcionar. En otras palabras, la red LSTM original sufría por una sobrecarga de información. En otras palabras, con el fin de dejar espacio para la nueva información, debemos olvidar selectivamente la información vieja menos relevante. Para que las RNR puedan hacer esto, necesitan un mecanismo para el “olvido selectivo”.

[14] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.5709&rep=rep1&type=pdf

3.2 Compuertas como Mecanismo de Selectividad

La lectura, escritura y el olvido selectivo implican acciones separadas de lectura, escritura y olvido para cada elemento del estado de una red LSTM. Matemáticamente, estas acciones se representan con vectores de lectura, escritura y olvido, con valores entre 0 y 1 que especifican el porcentaje de lectura, escritura y olvido que se hace para cada elemento de estado. Tengamos en cuenta que, si bien puede ser más natural pensar en leer, escribir y olvidar como decisiones binarias, necesitamos que nuestras decisiones se implementen a través de una función diferenciable. La función logística es una elección natural ya que es diferenciable y produce valores continuos entre 0 y 1.

Llamamos a estos vectores “compuertas” de lectura, escritura y olvido, y podemos calcularlos utilizando la función más simple que tenemos, como hicimos con la RNR simple: una red neuronal de una sola capa. Nuestras tres puertas en el paso de tiempo t se denotan: i_t la puerta de entrada (para escribir); k_t, la puerta de salida (para lectura); y f_t, la puerta de olvido (para recordar). Matemáticamente, estas compuertas se definen como:

$$i_t = \phi(W_i\, s_{t-1} + U_i\, x_{t-1} + b_i)$$

$$k_t = \phi(W_k\, s_{t-1} + U_k\, x_{t-1} + b_k)$$

$$f_t = \phi(W_f\, s_{t-1} + U_f\, x_{t-1} + b_f)$$

, donde:
x_t : representa la entrada (*input*) a la celda en el tiempo t.
$W_i, W_f, W_k, U_i, U_f, U_k$: son matrices de pesos.
b_i, b_f, b_k : son vectores que contienen los sesgos.
$\phi(\cdot)$ es la función de activación (e.g. la función logística).

3.3 Cintas Transportadoras

En resumen, las redes LSTM son un tipo de red neuronal recurrente, pero las LSTM además agregan una forma de llevar la información a través de muchos pasos de tiempo. Imaginemos que es como una cinta transportadora paralela a la secuencia de datos que se está procesando. La información de la secuencia puede saltar a la cinta transportadora en cualquier momento, ser transportada a un paso de tiempo posterior y saltar, intacta, cuando se necesite. Básicamente, esto es lo que hace LSTM: guarda la información para más tarde, evitando así que las señales más antiguas se desvanezcan gradualmente durante el procesamiento.

3.4 Derivando la Red LSTM

Para entender esto en detalle, comencemos desde la celda RNR simple. Como se tendrán muchas matrices con pesos, indexemos las matrices W y U en la celda de RNR con la letra o: (W_o y U_o) para denotar que son relacionadas con la salida (*output*).

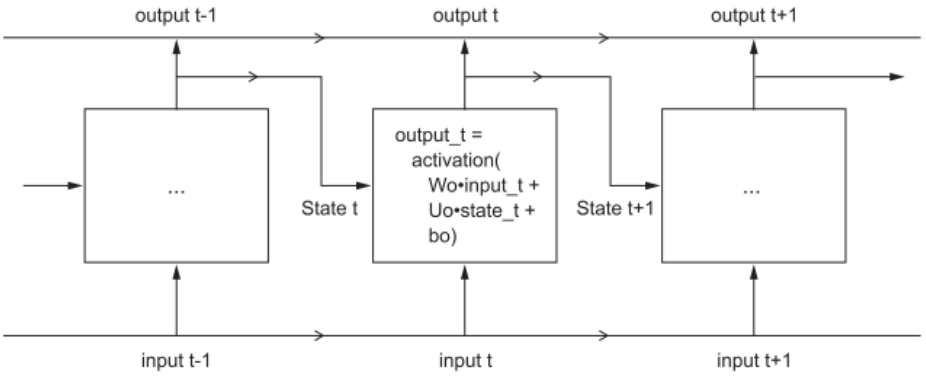


Figure 8: El punto de partida de una capa LSTM: una RNR simple.

Agreguemos a esta imagen un flujo de datos adicional que transporta información a través de los pasos de tiempo. Llamemos a sus valores en diferentes pasos de tiempo C_t, donde C significa *carry*. Esta información tendrá el siguiente impacto en la celda: se combinará con la conexión de entrada y la conexión recurrente (a través de una transformación densa: un producto punto con una matriz de pesos seguida de una suma de sesgo y la aplicación de una función de activación), y afectará el estado que se envía al siguiente paso de tiempo (a través de una función de activación y una operación de multiplicación). Conceptualmente, el flujo de datos que se transporta es una forma de modular la siguiente salida y el siguiente estado (ver la siguiente figura):

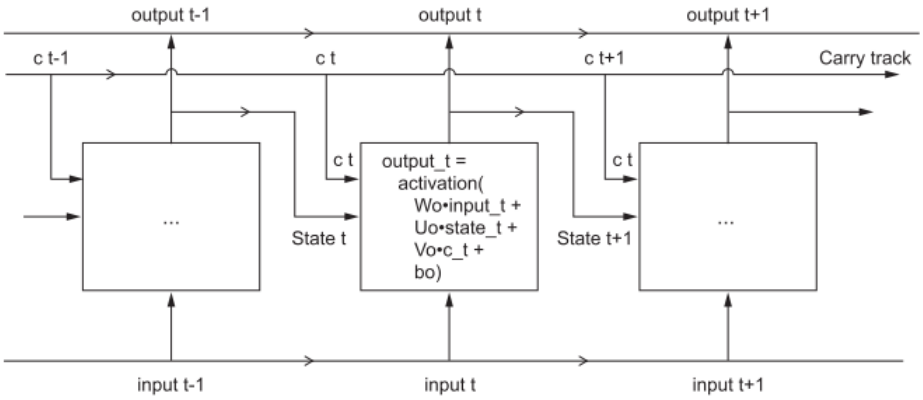


Figure 9: Para pasar de una RNR a una LSTM: añadimos una 'cinta transportadora'.

Ahora va la parte sutil: la forma en que se calcula el siguiente valor del flujo de datos transportado (i.e. el siguiente *carry*). Se trata de tres transformaciones distintas. Los tres tienen la forma de una simple celda RNR; esto es:

$$y = \phi(W_i\, s_{t-1} + U_i\, x_{t-1} + b_i)$$

pero cada transformación tiene su propia matriz de vectores, que se indexaran con las letras i, f y k. Con lo cuál tenemos:

$$i_t = \phi(W_i\, s_{t-1} + U_i\, x_{t-1} + b_i)$$

$$k_t = \phi(W_k\, s_{t-1} + U_k\, x_{t-1} + b_k)$$

$$f_t = \phi(W_f\, s_{t-1} + U_f\, x_{t-1} + b_f)$$

y además, la salida en cada paso se calcula como:

$$o_t = \phi(W_i\, s_{t-1} + U_i\, x_{t-1} + b_i)$$

Y por último, el siguiente *carry* se obtiene combinando i_t, k_t y f_t de la siguiente forma:

$$c_{t+1} = i_t \cdot k_t \cdot c_t \cdot f_t$$

Si añadimos este cálculo a la figura anterior, obtenemos lo siguiente:

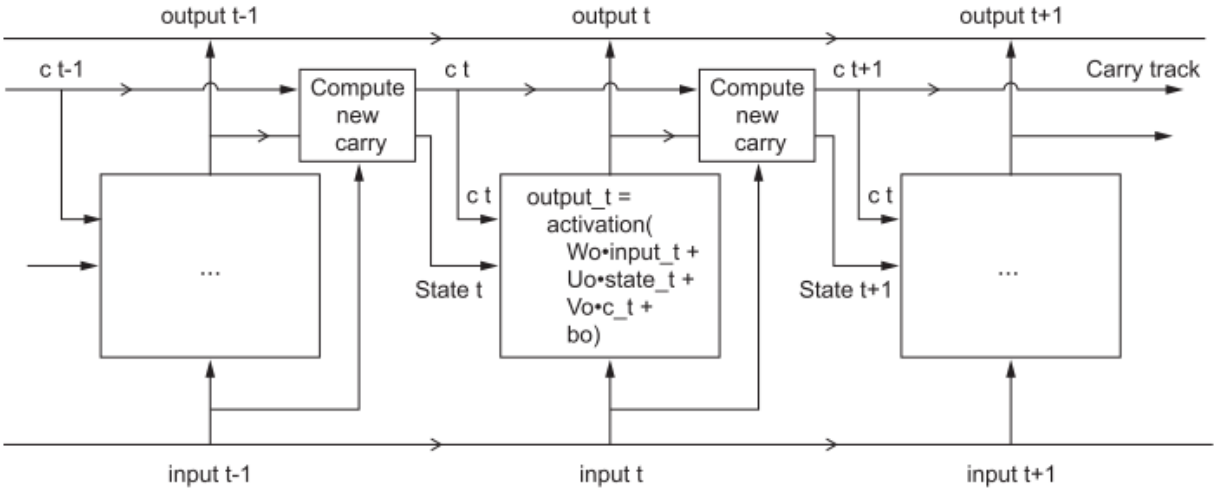


Figure 10: Anatomía de una LSTM.

4 Aplicación 1: Clasificación de Texto

La primera aplicación que se expone en este trabajo es la clasificación de sentimiento o *sentiment analysis* con un texto derivado de reseñas de películas. Lo que se busca es determinar la actitud de un usuario respecto a las películas cuyas reseñas se encuentran en la base de datos. Los juicios o evaluaciones utilizadas para comunicar la parte emocional funcionarán como indicadores de actitudes positivas o negativas. La base de datos es extraída del paquete `keras` y se llama IMDB que proviene de una base de datos en línea inaugurada en 1990 y que almacena información relacionada con películas.

Con la base de datos se realiza una minería de opinión o análisis de sentimiento el cual busca identificar y extraer información subjetiva del recurso, en este caso, extrae información del texto en forma de reseñas. Se trata de una clasificación de manera automática en función de la connotación positiva o negativa del lenguaje ocupado en el texto, aunque cabe resaltar que los análisis están basados en relaciones estadísticas y de asociación más no en un análisis lingüistico.

La siguiente sección desarrolla el código para la segunda aplicación de clasificación de texto, contiene 2 modelos; el primero es un modelo sencillo donde la red contiene una sola capa layer_lstm() con dropout como

regularización, y el segundo modelo utiliza, además de una capa LSTM, redes convolucionales, lo cual sugiere un resultado mejor con respecto al primero, sin embargo, eso sólo se sabra una vez que se evaluen los modelos y se comparen los resultados en la última parte de esta sección.

4.1 Procesamiento de textos: Word Embedding

Para la aplicación de clasificación de texto con redes LSTM es necesario infresar las reseñas al modelo, sin embargo, las redes neuronales, así como muchos modelos de *deep-learning* no son capaces de procesar textos de manera "cruda" (*raw text*), ya que en general es necesario transformarlos a unidades numéricas (*numeric tensors*) ; este proceso es llamado Vectorización de textos (*Vectorizing*). De manera general su funcionamiento es a través de la separación del texto en unidades llamadas *tokens* y su asociación con vectores numéricos. Los vectores numéricos son finalmente ingresados a las redes neuronales profundas.

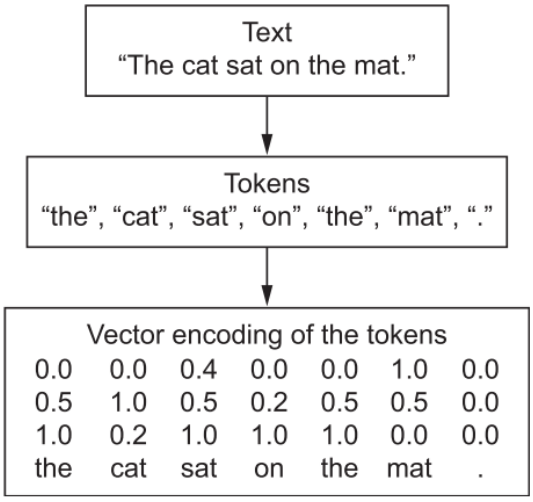


Figure 11: Transformación de texto a vectores de tokens.

En la imagen anterior se observa el proceso de Vectorización que comienza con la separación del texto en crudo en tokens y posteriormente su conversión a vectores. Aunque existen múltiples maneras de asociación de vectores con *tokens* en este proyecto se utilizará la técnica conocida como *token embedding*; en especial su uso para palabras conocido como *word embedding*.

La ventaja principal de este método es que permite el almacenamiento de los datos en pocas dimensiones comparado con métodos alternativos. Se trata de vectores densos de bajas dimensiones; típicamente 256-dimensional, 512-dimensional o 1,024-dimensional vs. vectores 20,000-dimensional o más en otros métodos alternativos. Además, *word embedding* es una técnica que aprende de los datos.

Su funcionamiento es similar al lenguaje humano pero traducido a un espacio geométrico en el cual los vectores están conformados por palabras con significados o usos similares y la distancia geométrica entre vectoress representa una distancia semántica entre las palabras asociadas. A pesar de que su funcionamiento parece simple, los problemas aparecen mayoritariamente en la interpretación humana de campos semánticos de palabras, diferencia de lenguajes, culturas y contextos que llevan a diferencias en cuanto a asociación de palabras. Por esta razón, resulta razonable aprender este tipo de "espacios incrustados" (*embedding spaces*) con cada nueva tarea. El paquete `keras` hace este proceso fácilmente utilizando una capa adicional en la arquitectura llamada *layer_embedding*.

La configuración de esta capa requiere indicar dos parámetros: i) el número de tokens posibles y ii) la dimensionalidad.

5 Desarrollo de código, ajuste y clasificación (base de datos: IMDB)

5.1 Modelo 1: Modelo LSTM regularizado

El propósito de este modelo es crear una estructura básica de una red LSTM para clasificar las reseñas de las películas de la base de datos, y agregarles un porcentaje de dropout para hacer la red más robusta.

5.2 Librerías

```
library(zeallot)
library(keras)  # Librería para arquitectura de redes
```

5.3 Parámetros

```
max_features <- 10000  # No. de palabras a considerar como features
maxlen <- 500  # Corta reseñas después de 500 palabras
batch_size <- 32
```

5.4 Cargar y Preparar los Datos

```
imdb <- dataset_imdb(num_words = max_features)

c(c(input_train, y_train), c(input_test, y_test)) %<-%
```

```
    imdb  # Carga los datos como una lista de enteros
cat(length(input_train), "train sequences\n")
```

```
## 25000 train sequences
```

```
cat(length(input_test), "test sequences")
```

```
## 25000 test sequences
```

Se determina la forma de los inputs del modelo tanto de entrenamiento como de prueba:

```
cat("Pad sequences (samples x time)\n")
```

```
## Pad sequences (samples x time)
```

```
input_train <- pad_sequences(input_train, maxlen = maxlen)
input_test <- pad_sequences(input_test, maxlen = maxlen)
cat("input_train shape:", dim(input_train), "\n")
```

```
## input_train shape: 25000 500
```

```
cat("input_test shape:", dim(input_test), "\n")
```

```
## input_test shape: 25000 500
```

5.5 Construcción del modelo LSTM utilizando Keras

En el chunk que sigue a este párrafo, se muestra la estructura que va a tener la red LSTM, ,seguido por chunk con el código que compila el modelo, dando como resultado una impresión de pantalla que muestra que en total se tiene 328,353 parámetros a calcular.

La estructura del modelo es justamente donde va a radicar la diferencia de ambos modelos.

Para construir una red LSTM con Keras en R lo que se debe de hacer es:

1. Utilizar el comando `keras_model_sequential()` y asignarlo a un objeto llamado “modelo”
2. Asignar las capas al modelo por medio del comando `layer_lstm()`. Este comando dentro de la paquetería de keras puede recibir un gran número de parámetros, sin embargo, en este caso, se selecciona el número de celdas de memoria y los porcentajes de dropout.

```
model <- keras_model_sequential()
model %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  #especificar el número máximo de entradas y de salidas
  layer_lstm(units = 32, dropout = 0.2,
             recurrent_dropout = 0.2) %>% # cambio
  layer_dense(units = 1, activation = 'sigmoid')
```

Compilar el modelo:

```
model %>% compile(loss = "binary_crossentropy",
    optimizer = "adam", metrics = c("accuracy"))
model
```

```
## Model
## _____________________________________________
## Layer (type)        Output Shape      Param #
## =============================================
## embedding_1 (Embedd (None, None, 32)  320000
## _____________________________________________
## lstm_1 (LSTM)       (None, 32)        8320
## _____________________________________________
## dense_1 (Dense)     (None, 1)         33
## =============================================
## Total params: 328,353
## Trainable params: 328,353
## Non-trainable params: 0
## _____________________________________________
```

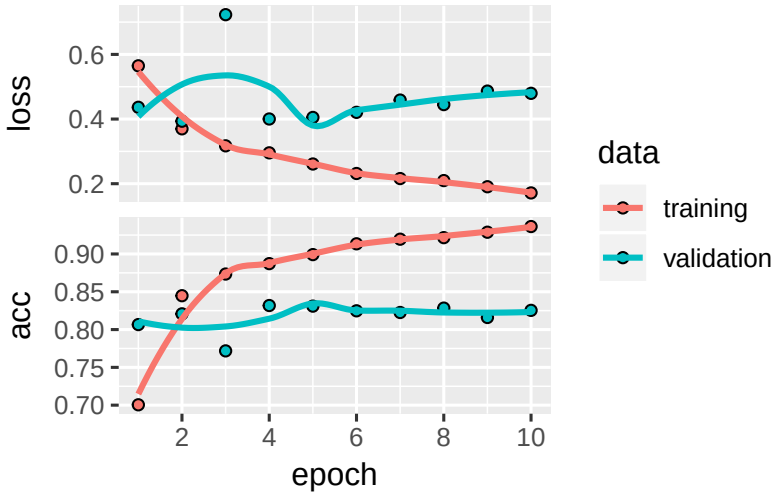
Correr el modelo y ajustar:

Dada la complejidad y estructura del modelo, en especial por el tamaño del batch, esta parte puede tardar un poco en compilar.

```
history <- model %>% fit(input_train, y_train,
    epochs = 10, batch_size = 128, validation_split = 0.2)
```

Al ajustar el modelo, se puede observar las siguiente gráfica donde se aprecia como se van moviendo los niveles de pérdida y de accuracy con respecto a los conjuntos de entranamiento y validación.

```
plot(history)
```



Opcional: Salvar el modelo

Cuando se corren este tipo de modelos, es muy probable que el tiempo que tarda el modelo en ajustar sea bastante, es por ello que se salvan los resultados del modelo para no tener la necesidad de correrlo otra vez.

```
# model %>% save_model_hdf5('imdb_LSTM.h5')
# modelo <- load_model_hdf5('imdb_LSTM.h5')
```

Evaluar el modelo entrenado con conjunto de prueba:

```
scores <- model %>% evaluate(input_test, y_test,
    batch_size = batch_size)
```

5.6 Resultados

```
cat("Test score:", scores[[1]])
```

```
## Test score: 0.4709888
```

```
cat("Test accuracy", scores[[2]])
```

```
## Test accuracy 0.82812
```

La interpretación de estos valores se complementa al tener otros modelos como punto de comparación, es por eso que antes de sacar una conclusión relacionada con los resultados (Scores) de este modelo, primero se van a correr otro. Por ahora, lo único que se va a mencionar es que el modelo 1 tiene una presición de: 0.81908, indicando que del conjunto de prueba, el modelo fue ccapaz de clasificar el 81.9% de ellas correctamente.

5.7 Modelo 2: Modelo LSTM con Redes Convolucionales

El propósito de este modelo, al igual que el anterior, es establecer una estructura para una red LSTM, con la diferencia de que en este modelo se ajusta la estructura de tal manera que se utilizan redes lstm convolucionales.

5.8 Librerías

La librerias fueron cargadas como el modelo anterior, es por eso que en este punto ya no se carga.

5.9 Parámetros

Para este modelo, se necesita establecer dos tipos de parámetros, los primeros se mantienen como en los dos modelos anteriores y los parámetros relacionados con las capas de redes convolucionales son los siguientes:

```
kernel_size = 5
filters = 64
pool_size = 4
embedding_size = 128
lstm_output_size = 70
```

5.10 Cargar y preparar los Datos

De la misma manera que el modelo anterior, la forma en la que se cargan y preparan los datos es la misma.

3.1. Determinar la forma de los inputs del modelo tanto de entrenamiento como de prueba

Para este inciso, se toma la información estructurada del modelo 1.

5.11 Construcción del modelo LSTM utilizando Keras

Estructura del modelo

Como se mencionó con anterioridad, es en esta parte donde los modelos se ajustan. Los cambios que se hicieron en relación con el modelo anterior con el fin de añadirle robustez e intentar que la clasificación sea mejor son los siguientes:

El porcentaje de dropout se cambio del inciso anterior a este (antes 0.2, ahora 0.25) Se agrega una capa de red convolucional.

Cabe mencionar que la razón por la cuál se selecciona esta combinación de redes, convolucionales y lstm, es que redes convolucionales ayudan a que el modelo entrene más rápido gracias a su funcionamiento con filtros y LSTM permite guardar memoria (Ahamed, 2018).

```
model <- keras_model_sequential()
model %>%
  layer_embedding(max_features, embedding_size,
                  input_length = maxlen) %>%
  layer_dropout(0.25) %>%
  layer_conv_1d(
    filters,
    kernel_size,
    padding = "valid",
    activation = "relu",
    strides = 1
  ) %>%
  layer_max_pooling_1d(pool_size) %>%
  layer_lstm(units=64) %>% # se hizo un cambio
  layer_dense(1) %>%
  layer_activation("sigmoid")
```

Compilar el modelo:

```
model %>% compile(loss = "binary_crossentropy",
    optimizer = "adam", metrics = "accuracy")
```

Correr el modelo y ajustar:

```
model %>% fit(input_train, y_train, batch_size = 128,
    epochs = 10, validation_split = 0.2)
```

Opcional: Guardar el modelo

```
# model %>%
# save_model_hdf5('imdb_LSTM_CNN.h5') modelo
# <- load_model_hdf5('imdb_LSTM_CNN.h5')
```

Evaluar el modelo entrenado con conjunto de prueba

```
scores <- model %>% evaluate(input_test, y_test,
    batch_size = batch_size)
```

5.12 Resultados

```
cat("Test score:", scores[[1]])
```

```
## Test score: 0.6092503
```

```
cat("Test accuracy", scores[[2]])
```

```
## Test accuracy 0.8646
```

Los resultados que arrogan este modelo son los anteriores, de estos se considera el segundo para poder comparar con el modelo anterior en la última parte de esta sección. Mientras, se puede decir que este modelo tuvo una presición de 0.8586, lo cual quiere decir que la red logró clasificar correctamente el 85.86% de las reseñs dentro del conjunto de prueba.

5.13 Contraste de Resultados (Modelo 1 vs. Modelo 2)

Para culminar con esta sección y con la primera aplicación de las redes LSTM, es necesario llegar a una conclusión relacionada con los modelos anteriores.

El primer modelo resultó en: *Precisión* $= 0.8191$ El segundo modelo resultó en: *Precisión* $= 0.8586$.

Lo que se puede observar con estos valores es que el segundo modelo fue el mejor con un valor de precisión mayor que el primero. Pero lo más importante a destacar con respecto a esta sección es que, en ambos casos, las redes LSTM lograron un valor de precisión mayor al 80%, mostrando que usar este tipo de redes puede generar un buen resultado en problemas de clasificación donde la dimensión de los datos es bastante amplia y donde se requiere de tener un contenedor que guarde información relevante y la maneje propiamente dependiendo el caso.

6 Aplicación 2: Predicción de Series de Tiempo

La segunda aplicación aquí presentada será otro tipo de procesamiento de datos secuenciales; ya no serán relacionados con el lenguaje, sino que ahora se trata de un problema de series de tiempo. Se utiliza una base de datos con temperaturas mínimas diarias en la ciudad de Melbourne en Australia (*Daily Minimum Temperatures in Melbourne, Australia*[15]) y debido a ciertas complejidades que se añaden al tratar con este tipo de datos temprales, primero se explicarán dos conceptos importantes: i) Validación con Backtesting y ii) Evaluación de pronósticos.

[15] obtenida de la página web del gobierno de Australia

6.1 Backtesting: Validación Cruzada para Series de Tiempo

Backtesting es un término usado en el modelado de predicción y se refiere a probar un modelo predictivo sobre datos históricos. *backtesting* es un tipo de validación cruzada aplicada a períodos de tiempo anteriores.

En el análisis financiero, el *backtesting* busca estimar el desempeño de una estrategia o modelo si dicha estrategia o modelo se hubiera implementado

durante algún período pasado. Esto requiere simular las condiciones pasadas con suficiente detalle, lo que hace que una limitación de las pruebas de *backtesting* sea la necesidad de datos históricos detallados. Además, el *backtesting*, al igual que otros modelos, está limitado por un posible sobreajuste. Es decir, a menudo es posible encontrar una estrategia que hubiera funcionado bien en el pasado, pero que no funcionará bien en el futuro. A pesar de estas limitaciones, el *backtesting* proporciona una buena guía para validar un modelo de predicción de series de tiempo.

¿Qué significa *backtesting*? Según Kerkhof y Melenberg (J. Kerkhof and B. Melenberg. *Backtesting for risk-based regulatory capital*. Journal of Banking & Finance, 28, 2004.), se trata de una verificación de diagnóstico final de un modelo de predicción utilizando datos históricos, es decir, un conjunto de procedimientos estadísticos destinados a inspeccionar si las realizaciones pasadas de las variables a predecir, observadas ex post, cumplen con lo que dicta el modelo. Generalmente, el *backtesting* denota alguno de los siguientes dos conceptos:

1. En finanzas, se refiere a la evaluación del desempeño teórico de una estrategia de inversión en un periodo pasado;
2. En análisis de series de tiempo, se refiere a la evaluación de los modelos de predicción, por medio de datos históricos, con base en alguna medida o regla matemática que refleje la exactitud o desviación de los datos vs. las estimaciones.

Debe quedar claro que nuestra discusión se centrará en el segundo concepto, que consiste en considerar las predicciones de las variables aleatorias en cuestión *ex-ante*, y evaluarlas frente a las realizaciones verificadas *ex-post*. Por lo general, la elección de la metodología de *backtesting* depende del tipo de predicción deseada. Como se afirma en Emmer et al. (2013)[16], existen métodos de *backtesting* para:

1. Estimación puntual, e.g. el valor de una variable aleatoria (Y). Usualmente, la estimación está definida en términos de un valor esperado condicional en toda la información disponible en el momento en el que se realiza la predicción ($\mathcal{F}$); es decir:

$$E\left[Y_{t+k} \,|\mathcal{F}(Y_s)\right] \qquad \text{, con } s \leq t$$

2. Predicciones de intervalo – delinean un intervalo en el cual el valor de pronóstico tiene un cierto rango, con una cierta probabilidad predeterminada p.

Nosotros nos enfocaremos en el primer método.

[16] S. Emmer, M. Kratz, and D. Tasche. *What is the best risk measure in practice? A comparison of standard measures*. ESSEC working paper, (2013)

6.2 Evaluación de pronósticos

De acuerdo con Tsay (Tsay, R. S. *Multivariate Time Series Analysis: With R and financial applications*. Hoboken, NJ: Wiley. 2014), para evaluar la calidad de un pronóstico se utilizan estadísticas que tratan de medir cuánto se aleja $\hat{Y}_n(h)$ de su valor observado Y_{n+h} , $h = 1, 2, \ldots, H$. Si se tienen N datos disponibles, para evaluar los pronósticos se acostumbra estimar un modelo a partir de n datos, $n \leq N$, generar $H = N - n + 1$ pronósticos y compararlos frente a los valores reales. Al comparar pronósticos de distintos modelos, *debe elegirse el modelo para el cuál estas estadísticas sean mínimas*:

Raíz del Error Cuadrático Medio

RMSE por las siglas en inglés: *Root Mean Square Error*

$$RMSE = \sqrt{\frac{1}{H}\sum_{h=1}^{H} e_n^2(h)}$$

Error Absoluto Medio

MAE por las siglas en inglés: *Mean Absolute Error*

$$MAE = \frac{1}{H}\sum_{h=1}^{H} |e_n(h)|$$

Error Absoluto Medio Porcentual

MAPE por las siglas en inglés: *Mean Absolute Percentual Error*

$$MAPE = \frac{1}{H}\sum_{h=1}^{H} \left| \frac{e_n(h)}{Y_{n+h}} \times 100 \right|$$

Por simplicidad, nosotros utilizaremos la raíz del error cuadrático medio (RMSE) como medida de bondad de ajuste durante el *backtesting*. Es decir, en resumen, estamos equipados con un modelo, con parámetros ya calibrados con los datos de entrenamiento, el cual utilizaremos para hacer predicciones en función de la información disponible hasta cierto período determinado ($\mathcal{F}$). Deseamos tener un fundamento racional para decidir si ese modelo hace predicciones aceptables sobre el futuro, y queremos decidir esto con base en el desempeño del modelo al hacer las predicciones del pasado. Este desempeño lo mediremos con la estadística Raíz del Error Cuadrático Medio (RMSE).

7 Desarrollo de código y Predicciones (base de datos: Daily Minimum Temperatures in Melbourne, Australia, 1981-1990)

7.1 Librerías

A continuación, se indican todas las librerías que se utilizan a lo largo del código. Cabe mencionar, que al igual que en Redes Neuronales No Recurrentes, para poder construir una estructura acorde al modelo LSTM es necesario

instalar nuevamente el paquete `keras` con el código `keras::install.keras()`.

```
library(tint)
library(gridExtra)
# invalidate cache when the package version changes
knitr::opts_chunk$set(tidy = FALSE,
                      cache.extra = packageVersion('tint'))
options(htmltools.dir.version = FALSE)
library(tidyverse)
library(lubridate)
library(glue)
library(forcats)
library(timetk)
library(tidyquant)
library(tibbletime)
library(xts)
library(tsbox)
library(astsa)
library(cowplot)
library(recipes)
library(readxl)
library(rsample)
library(yardstick)
library(keras) # Librería para arquitectura de redes
```

Con el siguiente código se definen las caracteristicas del diseño de las gráficas del documento.

```
theme_set(theme_classic(base_size = 13))
options(
  ggplot2.continuous.colour = "viridis",
  ggplot2.continuous.fill = "viridis"
)
```

7.2 Datos

La base de datos utilizada se denomina *Daily Minimum Temperatures in Melbourne, Australia* obtenida de la página web del gobierno de Australia – Bureau of Meteorology.

Se debe de leer los datos y asignarlos a un objeto llamado “australia” para a continuación convertir el conjunto de datos a un formato de series de tiempo que permita realizar el análisis posterior.

```r
# Leer archivo e imprimir algunos datos
australia <- read_excel("australia.xlsx")
head(australia)
```

```
## # A tibble: 6 x 2
##   date                value
##   <dttm>              <dbl>
## 1 1981-01-01 00:00:00  20.7
## 2 1981-01-02 00:00:00  17.9
## 3 1981-01-03 00:00:00  18.8
## 4 1981-01-04 00:00:00  14.6
## 5 1981-01-05 00:00:00  15.8
## 6 1981-01-06 00:00:00  15.8
```

```r
# Determinar el formato del objeto
class(australia)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```r
#Dar formato de serie de tiempo y observar que efectivamente
# el formato (class()) cambia
australia_ts <- australia %>%
  mutate(index = as_date(date)) %>%
  select("index", "value") %>%
  as_tbl_time(index = index)

head(australia_ts)
```

```
## # A time tibble: 6 x 2
## # Index: index
##   index      value
##   <date>     <dbl>
## 1 1981-01-01  20.7
## 2 1981-01-02  17.9
## 3 1981-01-03  18.8
## 4 1981-01-04  14.6
## 5 1981-01-05  15.8
## 6 1981-01-06  15.8
```

```r
class(australia_ts)
```

```
## [1] "tbl_time"   "tbl_df"     "tbl"
## [4] "data.frame"
```

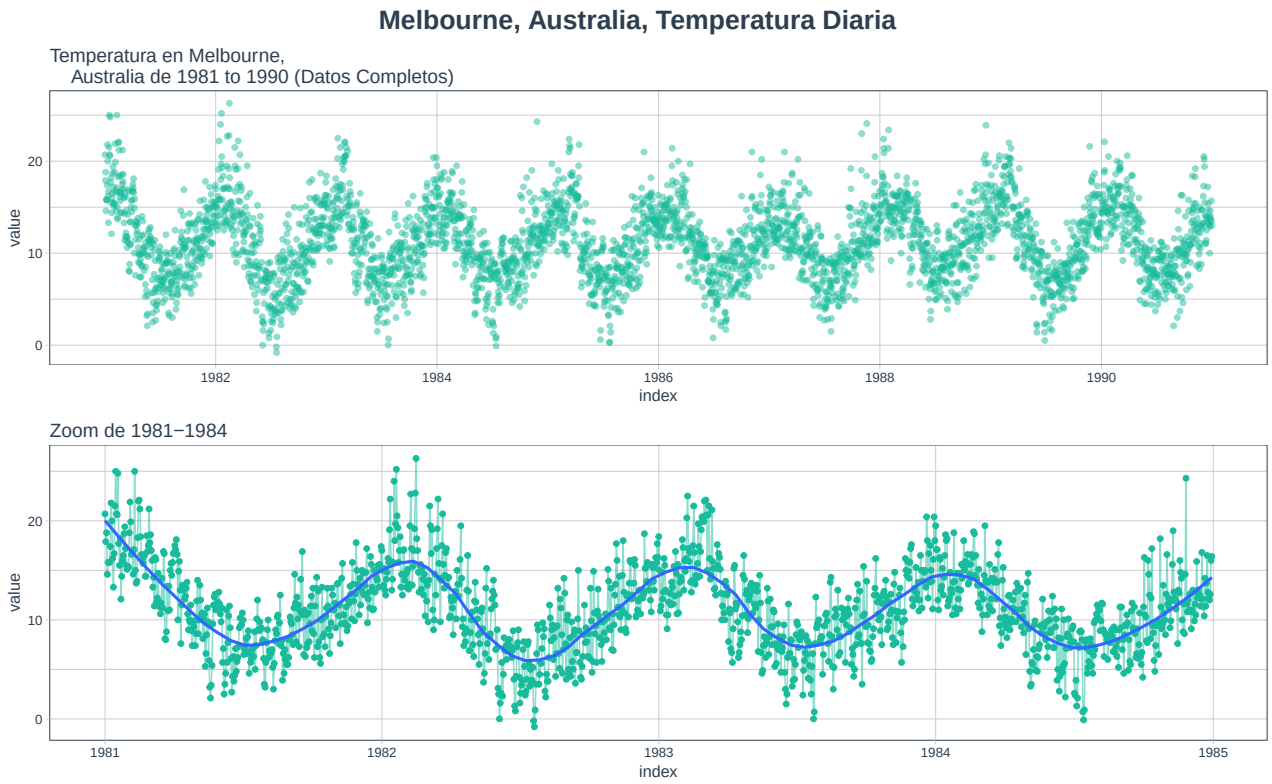
7.3 Análisis Exploratorio de la Base de Datos

Esta serie de tiempo cubre los años 1981 - 1990 (9 años), por lo tanto, a continuación se visualiza la serie de tiempo completa y posteriormente, a la grafica se le aplica un zoom de 1981 - 1984 para visualizar mejor los ciclos presentes en la serie de tiempo.

```
# Gráfica con todos los datos observados
grafica_aus_1 <- australia_ts %>%
  ggplot(aes(index, value)) +
  geom_point(color = palette_light()[[3]], alpha = 0.5) +
  theme_tq() +
  labs(
    title = "Temperatura en Melbourne,
    Australia de 1981 to 1990 (Datos Completos)"
  )

# Gráfica con zoom para visualizar ciclos

grafica_aus_2 <- australia_ts %>%
  filter_time('1981-01-01' ~ '1984-12-31') %>%
  ggplot(aes(index, value)) +
  geom_line(color = palette_light()[[3]], alpha = 0.5) +
  geom_point(color = palette_light()[[3]]) +
  geom_smooth(method = "loess", span = 0.2, se = FALSE) +
  theme_tq() +
  labs(
    title = "Zoom de 1981-1984"
  )

# Juntar las dos gráficas en una sola imagen
p_title <- ggdraw() +
  draw_label("Melbourne, Australia, Temperatura Diaria",
             size = 18, fontface = "bold",
             colour = palette_light()[[1]])

cowplot::plot_grid(p_title, grafica_aus_1,
                   grafica_aus_2, ncol = 1,
                   rel_heights = c(0.1, 1, 1))
```

Lo que se puede observar con la imagen anterior, es lo siguiente:

En la primera gráfica, se logra apreciar un patrón a lo largo de los 9 años, pero para estar seguros de ello, es necesario, aplicar zoom a un rango más pequeño y dibujar una línea suave que permita apreciar de mejor manera el



comportamiento de los datos y visualizar el ciclo. Se observa que sí existen ciclos de aproximadamente un año, con una amplitud relativamente invariante.

7.4 Análisis de Autocorrelación

Dada la información obtenida previamente, se puede pensar que un buen modelo a emplear que permita capturar los ciclos como memoria pasada es la red LSTM, sin embargo, para poder hacer uso de éste hay que inspeccionar la autocorrelación presente en los datos. Este valor debe estar cerca (o preferentemente mayor) de 0.5 indicando una autocorrelación importante.

El código siguiente, muestra la función para calcular la autocorrelación y seguido, se presenta un factor muy importante a considerar, que es el punto de corte de los ciclos que servirá como tamaño de *batch* para hacer la predicción a futuro.

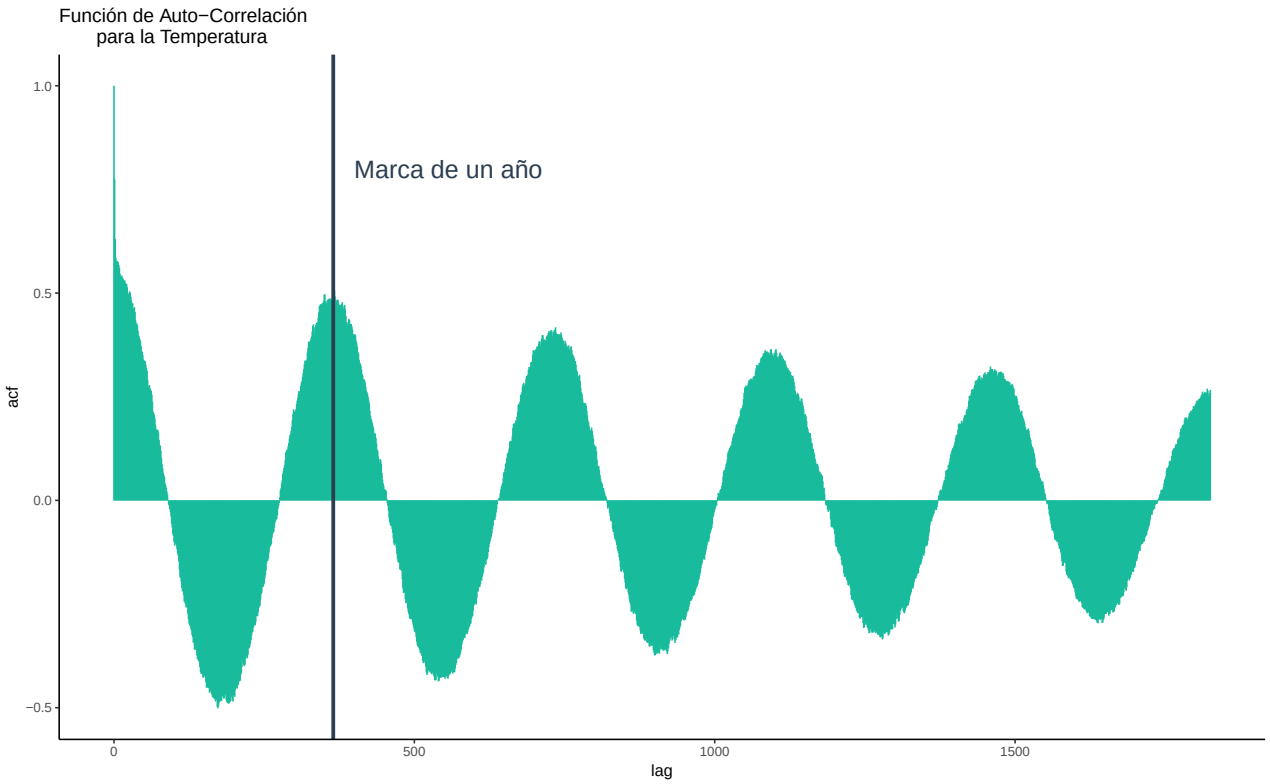
Primero, se genera la Función de Autocorrelación (ACF, por sus siglas en inglés) la cuál calcula la correlación lineal entre la variable de predicción con versiones rezagadas de sí misma (*lags*). Estos valores de correlación lineal se denominan autocorrelación. La función fue personalizada para que al aplicar la función `stats::acf`, ésta regrese una tabla con los valores, en lugar de una gráfica.

```
# Función de Autocorrelación
funcion_ac <- function(data, value, lags = 0:20) {

  value_expr <- enquo(value)
  acf_values <- data %>%
    pull(value) %>%
    acf(lag.max = tail(lags, 1), plot = FALSE) %>%
    .$acf %>%
    .[,,1]

  ret <- tibble(acf = acf_values) %>%
    rowid_to_column(var = "lag") %>%
    mutate(lag = lag - 1) %>%
    filter(lag %in% lags)

  return(ret)
}

max_lag <- 365 * 5

# Observamos las correlaciones
australia_ts %>%
  funcion_ac(value, lags = 0:max_lag)
```

```
## # A tibble: 1,826 x 2
##      lag   acf
##    <dbl> <dbl>
##  1     0 1.000
##  2     1 0.774
##  3     2 0.630
##  4     3 0.585
##  5     4 0.578
##  6     5 0.577
##  7     6 0.575
##  8     7 0.574
##  9     8 0.568
## 10     9 0.561
## # ... with 1,816 more rows
```

```
australia_ts %>%
  funcion_ac(value, lags = 0:max_lag) %>%
  ggplot(aes(lag, acf)) +
  geom_segment(aes(xend = lag, yend = 0),
               color = palette_light()[[3]]) +
```

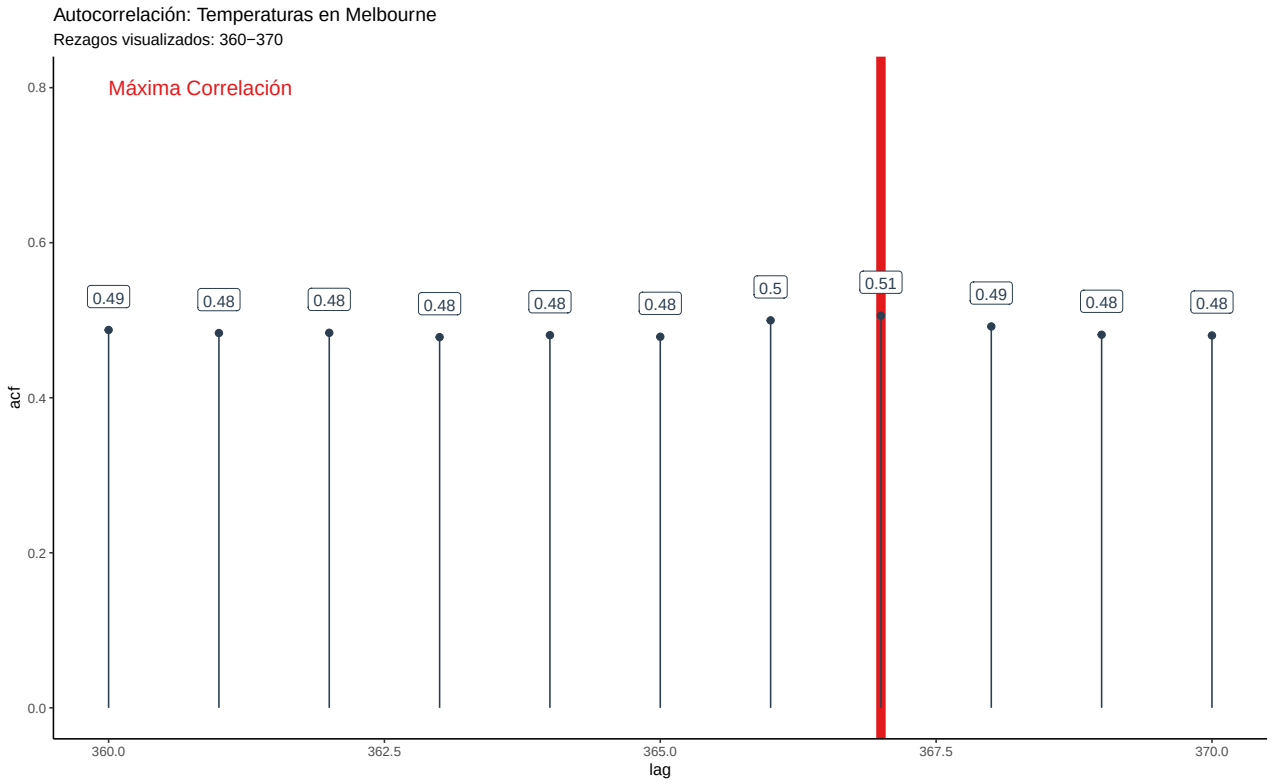
```
  geom_vline(xintercept = 365, size = 1.2,
             color = palette_light()[[1]]) +
  annotate("text", label = "Marca de un año",
           x = 400, y = 0.8,
           color = palette_light()[[1]],
           size = 6, hjust = 0) +
  theme_classic() +
  labs(title = "Función de Auto-Correlación
       para la Temperatura")
```



Se puede observar en la gráfica anterior que existen valores cercanos al 0.5 entre los rezagos 360 - 370, por lo tanto, podemos utilizar una red neuronal LSTM para aprovechar dicha autocorrelación significativa y hacer predicciones. Para estar seguros de tener el valor más alto de autocorrelación, se desarrolló el siguiente código:

```
#Gráfica con Correlación Máxima
australia_ts %>%
  funcion_ac(value, lags = 360:370) %>%
  ggplot(aes(lag, acf)) +
  geom_vline(xintercept = 367, size = 3, color = palette_light()[[2]]) +
```

```
  geom_segment(aes(xend = lag, yend = 0), color = palette_light()[[1]]) +
  geom_point(color = palette_light()[[1]], size = 2) +
  geom_label(aes(label = acf %>% round(2)), vjust = -1,
             color = palette_light()[[1]]) +
  annotate("text", label = "Máxima Correlación", x = 360, y = 0.8,
           color = palette_light()[[2]], size = 5, hjust = 0) +
  theme_classic() +
  labs(title = "Autocorrelación: Temperaturas en Melbourne",
       subtitle = "Rezagos visualizados: 360-370")
```




```
# Obtención numérica del rezago óptimo
optimal_lag_setting <- australia_ts %>%
  funcion_ac(value, lags = 360:370) %>%
  filter(acf == max(acf)) %>%
  pull(lag)
optimal_lag_setting
```

```
## [1] 367
```

Visualmente, parece que el mejor rezago ocurre al año (365 días), sin

embargo, el cálculo anterior muestran que el rezago óptimo es poco más de un año (i.e. día 367).

Como se vio durante la clase, para evaluar un modelo es necesario dividir la base de datos en entrenamiento y prueba, sin embargo, cuando se tienen las suficientes entradas, se recomienda dividir los datos en tres conjuntos: Entrenamiento, Validación y Prueba. Pero, existen casos donde no se quiere apartar una muestra de validación, es por ello que se puede utilizar el criterio de "validación cruzada" que es un método computacional que permite producir una estimación interna (usando sólo muestra) del error de predicción (González, 2018) por medio de una partición al azar con tamaños similares de la muestra de entrenamiento.

Sin embargo, cuando se trata con series de tiempo este proceso es un poco diferente, podemos crear un plan de muestro de validación cruzada utilizando ventanas de periodos de tiempo que se van desplazando y esto es el equivalente a la selección de submuestras. Actualmente, existe un paquete en R que permite implementar el método de backtesting de manera sencilla, dicho paquete se llama: `rsample` y se utiliza a continuación.

7.5 Datos de Entrenamiento y Prueba – Periodos de Desplazamiento

El siguiente código se encarga de dividir los datos en entrenamiento y prueba, lo interesante de *backtesting* es que al definir la estrategia se requiere de un valor llamado *periodo de salto* (*skip span*) que se va a encargar de ir recorriendo la muestra de datos de entrenamiento el tiempo para hacer la evaluación a lo largo de toda la serie de tiempo.

Dentro del paquete de `rsample`, existe una función llamada `rolling_origin()` que crea automáticamente las muestras adecuadas para la validación de series de tiempo. Lo difícil en este punto fue calcular los periodos de entrenamiento y prueba, ya que necesitaban ser periodos que cubrieran uniformemente los años y con ello distribuir equitativamente los datos en 6 conjuntos (*slices*) que cubrieran los 9 años de historia.

```
# Estrategia de separación
periodos_entrenamiento <- 365 * 3
periodos_prueba  <- 365 * 1
skip_span     <- 365 * 1

rolling_origin_resamples <- rolling_origin(
  australia_ts,
  initial    = periodos_entrenamiento,
  assess     = periodos_prueba,
  cumulative = FALSE,
```

```
  skip       = skip_span
)

rolling_origin_resamples
```

```
## # Rolling origin forecast resampling
## # A tibble: 6 x 2
##   splits             id
##   <list>             <chr>
## 1 <split [1.1K/365]> Slice1
## 2 <split [1.1K/365]> Slice2
## 3 <split [1.1K/365]> Slice3
## 4 <split [1.1K/365]> Slice4
## 5 <split [1.1K/365]> Slice5
## 6 <split [1.1K/365]> Slice6
```

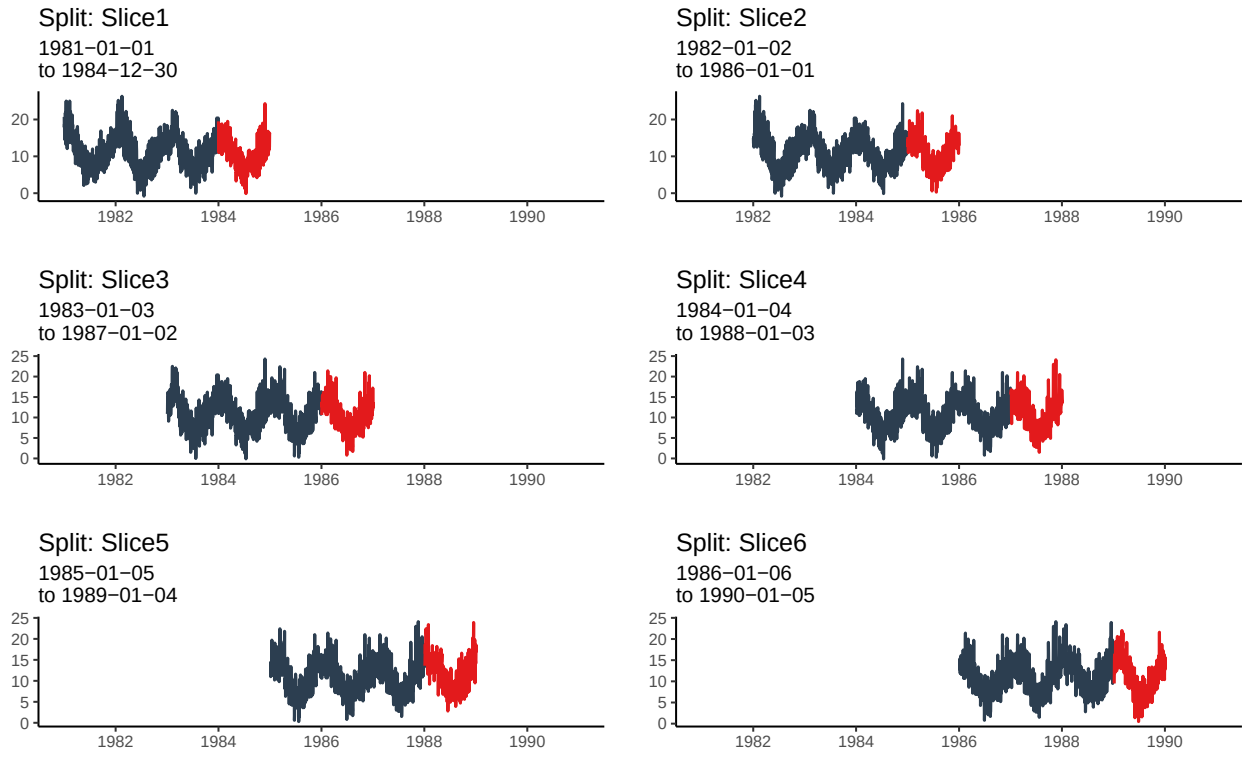
7.6 Visualización de la Estrategia de Backtesting

En esta sección se puede observar la partición en muestras de entrenamiento y prueba a lo largo de los 9 años. Primero se hace una función para graficar individualmente cada partición y posteriormente se grafican de forma conjunta utilizando `grid.arrange()`, cabe mencionar que las muestras de entrenamiento se distinguen por el color negro, mientras que las muestras de prueba de distinguen por el color rojo.

```
#Función para hacer la gráfica por partición

plot_split <- function(split, expand_y_axis = TRUE,
                               alpha = 1, size = 1,
                       base_size = 14) {

  entrenamiento_tbl <- training(split) %>%
    add_column(key = "training")

  prueba_tbl  <- testing(split) %>%
    add_column(key = "testing")

  data_manipulated <- bind_rows(entrenamiento_tbl,
                                prueba_tbl) %>%
    as_tbl_time(index = index) %>%
    mutate(key = fct_relevel(key, "training", "testing"))

  entrenamiento_tiempo_resumen <- entrenamiento_tbl %>%
    tk_index() %>%
```

```
    tk_get_timeseries_summary()

  prueba_tiempo_resumen <- prueba_tbl %>%
    tk_index() %>%
    tk_get_timeseries_summary()

  #Diseño de gráficas
  g <- data_manipulated %>%
    ggplot(aes(x = index, y = value, color = key)) +
    geom_line(size = size, alpha = alpha) +
    theme_classic(base_size = base_size) +
    scale_color_tq() +
    labs(
      title    = glue("Split: {split$id}"),
      subtitle = glue("{entrenamiento_tiempo_resumen$start}
                      to {prueba_tiempo_resumen$end}"),
      y = "", x = ""
    ) +
    theme(legend.position = "none")

  if (expand_y_axis) {

    australia_ts_time_summary <- australia_ts %>%
      tk_index() %>%
      tk_get_timeseries_summary()

    g <- g +
      scale_x_date(limits = c(australia_ts_time_summary$start,
                              australia_ts_time_summary$end))
  }

  return(g)
}

particion_1 <- rolling_origin_resamples$splits[[1]] %>%
    plot_split(expand_y_axis = TRUE)

particion_2 <- rolling_origin_resamples$splits[[2]] %>%
    plot_split(expand_y_axis = TRUE)

particion_3 <- rolling_origin_resamples$splits[[3]] %>%
    plot_split(expand_y_axis = TRUE)
```

```
particion_4 <- rolling_origin_resamples$splits[[4]] %>%
    plot_split(expand_y_axis = TRUE)

particion_5 <- rolling_origin_resamples$splits[[5]] %>%
    plot_split(expand_y_axis = TRUE)

particion_6 <- rolling_origin_resamples$splits[[6]] %>%
    plot_split(expand_y_axis = TRUE)


todas <- grid.arrange(particion_1,particion_2,
                      particion_3, particion_4,
                      particion_5, particion_6,
                      ncol=2)
```




```
todas
```

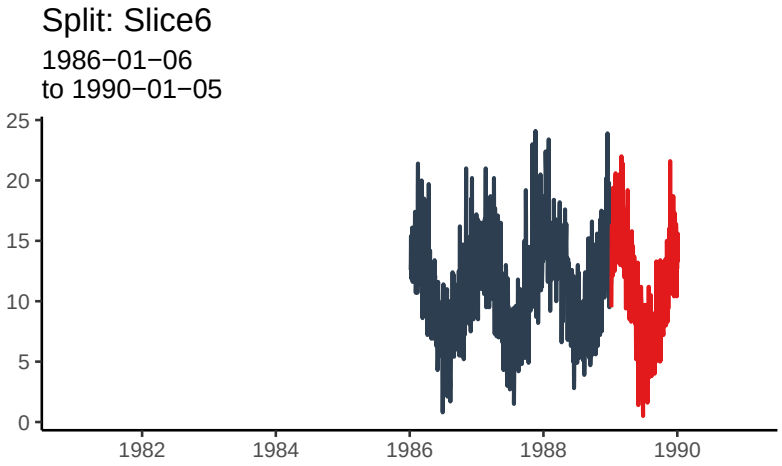
```
## TableGrob (3 x 2) "arrange": 6 grobs
##   z     cells    name           grob
## 1 1 (1-1,1-1) arrange gtable[layout]
```

```
## 2 2 (1-1,2-2) arrange gtable[layout]
## 3 3 (2-2,1-1) arrange gtable[layout]
## 4 4 (2-2,2-2) arrange gtable[layout]
## 5 5 (3-3,1-1) arrange gtable[layout]
## 6 6 (3-3,2-2) arrange gtable[layout]
```

7.7 Red Neuronal Long-Short Term Memory (LSTM)

A continuación, se desarrolla el modelo LSTM con `keras`. Para entender el funcionamiento de las redes LSTM, se realiza primero un ejemplo sencillo. Se selecciona el *split* número 6 y se grafica:

```
split    <- rolling_origin_resamples$splits[[6]]
split_id <- rolling_origin_resamples$id[[6]]
particion_6
```



Se dividen los datos del split 6 en entrenamiento y prueba para que se vea como la gráfica anterior y se colocan en una tabla tipo tibble de series de tiempo:

```
#Separación de datos
entrenamiento_df <- training(split)
prueba_df <- testing(split)

#Convertir a formato de tiempo
df <- bind_rows(
  entrenamiento_df %>% add_column(key = "training"),
  prueba_df %>% add_column(key = "testing")
  ) %>%
  as_tbl_time(index = index)
df
```

```
## # A time tibble: 1,460 x 3
## # Index: index
##    index      value key
##    <date>     <dbl> <chr>
##  1 1986-01-06  12.6 training
##  2 1986-01-07  13.1 training
##  3 1986-01-08  15.4 training
##  4 1986-01-09  11.9 training
##  5 1986-01-10  13.8 training
##  6 1986-01-11  14.4 training
##  7 1986-01-12  15.2 training
##  8 1986-01-13  12.5 training
##  9 1986-01-14  12.2 training
## 10 1986-01-15  16.1 training
## # ... with 1,450 more rows
```

El modelo LSTM requiere que los datos estén centrados y escalados, por esta razón se utiliza el paquete `recipes`, para hacer estas transformaciones con la función `bake()`.

```
rec_obj <- recipe(value ~ ., df) %>%
  step_center(value) %>%
  step_scale(value) %>%
  prep()

datos_transformados <- bake(rec_obj, df) #Transformar los datos de acuerdo a la receta anterior

head(datos_transformados)
```

```
## # A tibble: 6 x 3
##   index      value key
##   <date>     <dbl> <fct>
## 1 1986-01-06 0.354 training
## 2 1986-01-07 0.484 training
## 3 1986-01-08 1.08  training
## 4 1986-01-09 0.173 training
## 5 1986-01-10 0.665 training
## 6 1986-01-11 0.820 training
```

A pesar de que LSTM requiere que los datos estén centrados y escalados, es de suma importancia guardar los valores con los que se realizaron estas transformaciones (media y desviación estándar) para poder regresar a la escala original después de hacer la predicción.

```
media_historica <- rec_obj$steps[[1]]$means["value"]
escala_historica  <- rec_obj$steps[[2]]$sds["value"]

c("Media" = media_historica, "Escalamiento" = escala_historica)
```

```
##        Media.value Escalamiento.value
##          11.228699           3.869253
```

7.8 Inputs de la Red LSTM

Para poder determinar propiamente los valores inciales de una red LSTM hay diversas reglas, sin embargo, los puntos más importantes a considerar son:

-Tamaño de entranamiento (M) y prueba(N): El cociente de entrenamiento entre prueba tiene que ser un entero positivo En este caso:

$$\frac{M}{N} \in \mathbb{Z}^+$$

En este caso, tenemos:

$$\frac{1095}{365} = 3$$

El tamaño del batch (B): Este valor es determinado por un número entero que pueda dividir tanto al conjunto de datos de entrenamiento como al de prueba:

$$\frac{M}{B} = \frac{1095}{73} = 15$$

$$\frac{N}{B} = \frac{365}{73} = 5$$

```
lag_setting  <- 365 # = nrow(prueba_df)
batch_size   <- 73
train_length <- 365*3
tsteps       <- 1
epochs       <- 300
```

7.9 Preparación de los datos para ser leídos propiamente por `keras`.

```
lag_entrenamiento_tbl <- datos_transformados %>%
  mutate(value_lag = lag(value, n = lag_setting)) %>%
  filter(!is.na(value_lag)) %>%
  filter(key == "training") %>%
  tail(train_length)

x_train_vec <- lag_entrenamiento_tbl$value_lag
```

```
x_train_arr <- array(data = x_train_vec, dim = c(length(x_train_vec), 1, 1))

y_train_vec <- lag_entrenamiento_tbl$value
y_train_arr <- array(data = y_train_vec, dim = c(length(y_train_vec), 1))


lag_prueba_tbl <- datos_transformados %>%
  mutate(
    value_lag = lag(value, n = lag_setting)
  ) %>%
  filter(!is.na(value_lag)) %>%
  filter(key == "testing")

x_test_vec <- lag_prueba_tbl$value_lag
x_test_arr <- array(data = x_test_vec, dim = c(length(x_test_vec), 1, 1))

y_test_vec <- lag_prueba_tbl$value
y_test_arr <- array(data = y_test_vec, dim = c(length(y_test_vec), 1))
```

7.10 Modelo LSTM con Keras

Al igual que un modelo de Redes Neuronales convencionales, se deben definir las capas y parámetros correspondientes con el comando `layer_lstm()`.

```
model <- keras_model_sequential()

model %>%
  layer_lstm(units            = 50,
             input_shape      = c(tsteps, 1),
             batch_size       = batch_size,
             return_sequences = TRUE,
             stateful         = TRUE) %>%
  layer_lstm(units            = 50,
             return_sequences = FALSE,
             stateful         = TRUE) %>%
  layer_dense(units = 1)

model %>%
  compile(loss = 'mae', optimizer = 'adam')

model
```

```
## Model
## ___________________________________________
```

```
## Layer (type)        Output Shape      Param #
## =================================================
## lstm_3 (LSTM)       (73, 1, 50)       10400
## _________________________________________________
## lstm_4 (LSTM)       (73, 50)          20200
## _________________________________________________
## dense_3 (Dense)     (73, 1)           51
## =================================================
## Total params: 30,651
## Trainable params: 30,651
## Non-trainable params: 0
## _________________________________________________
```

Se observa que el modelo cuenta con 2 capas LSTM y una capa final densa. Al compilar el modelo se observa que en total hay 30,651 parámetros y la función de pérdida es el error absoluto promedio (MAE, *Mean Average Error*) que se espera optimizar por medio del optimizador Adam.

7.11 Ajuste del modelo LSTM

ara hacer el ajuste se utilizan 300 épocas para entrenar los parámetros. Es en este punto donde el modelo LSTM resuelve el problema del desvanecimiento del gradiente como se explicó con anterioridad.

```
for (i in 1:epochs) {
  model %>% fit(x          = x_train_arr,
                y          = y_train_arr,
                batch_size = batch_size,
                epochs     = 1,
                verbose    = 1,
                shuffle    = FALSE)

  model %>% reset_states()
  cat("Epoch: ", i)

}
```

```
## Epoch:  1Epoch:  2Epoch:  3Epoch:  4Epoch:  5Epoch:  6Epoch:  7Epoch:  8Epoch:  9Epoch:  10Epoch:  1
```

7.12 Predicciones

En los siguientes pedazos de código, se realizan las predicciones para el conjunto de prueba de este ejemplo sencillo, utilizando la función `predict()`. Posteriormente se retoman los valores de escalamiento y centrado para

regresar los datos a sus dimensiones originales y se unen los datos originales con los predichos para compararlos.

```
#Se hace la predicción
pred_out <- model %>%
  predict(x_test_arr, batch_size = batch_size) %>%
  .[,1]
head(pred_out)
```

```
## [1] 0.4025530 0.3928197 0.3696899 0.3997523
## [5] 0.3877857 0.3748012
```

```
#Se regresan a escala orginal los datos
pred_tbl <- tibble(
  index   = lag_prueba_tbl$index,
  value   = pred_out * escala_historica + media_historica
)
head(pred_tbl)
```

```
## # A tibble: 6 x 2
##   index      value
##   <date>     <dbl>
## 1 1989-01-06  12.8
## 2 1989-01-07  12.7
## 3 1989-01-08  12.7
## 4 1989-01-09  12.8
## 5 1989-01-10  12.7
## 6 1989-01-11  12.7
```

```
# Combinar información actual con predicciones en una tibble
tbl_1 <- entrenamiento_df %>%
  add_column(key = "actual")

tbl_2 <- prueba_df %>%
  add_column(key = "actual")

tbl_3 <- pred_tbl %>%
  add_column(key = "predict")

time_bind_rows <- function(data_1, data_2, index) {
  index_expr <- enquo(index)
  bind_rows(data_1, data_2) %>%
    as_tbl_time(index = !! index_expr)
}
```

```
ret <- list(tbl_1, tbl_2, tbl_3) %>%
  reduce(time_bind_rows, index = index) %>%
  arrange(key, index) %>%
  mutate(key = as_factor(key))

ret
```

```
## # A time tibble: 1,825 x 3
## # Index: index
##    index      value key
##    <date>     <dbl> <fct>
##  1 1986-01-06  12.6 actual
##  2 1986-01-07  13.1 actual
##  3 1986-01-08  15.4 actual
##  4 1986-01-09  11.9 actual
##  5 1986-01-10  13.8 actual
##  6 1986-01-11  14.4 actual
##  7 1986-01-12  15.2 actual
##  8 1986-01-13  12.5 actual
##  9 1986-01-14  12.2 actual
## 10 1986-01-15  16.1 actual
## # ... with 1,815 more rows
```

7.13 Cálculo del RMSE (*Root-Mean-Square Error*)

A continuación, se hace el cálculo del RMSE con una función que utiliza el comando de `rmse()` del paquete yardstick, sin embargo, los datos están en el formato *long* (útil para las gráficas) y esto no debe de ser así. Para transformar los datos y realizar el cálculo del RMSE, se creó la función `calc_rmse()`.

```
calc_rmse <- function(prediction_tbl) {

  rmse_calculation <- function(data) {
    data %>%
      spread(key = key, value = value) %>%
      select(-index) %>%
      filter(!is.na(predict)) %>%
      rename(
        truth    = actual,
        estimate = predict
      ) %>%
      rmse(truth, estimate)
  }
```

```
  safe_rmse <- possibly(rmse_calculation, otherwise = NA)

  as.numeric(safe_rmse(prediction_tbl)[,3])

}

calc_rmse(ret)
```

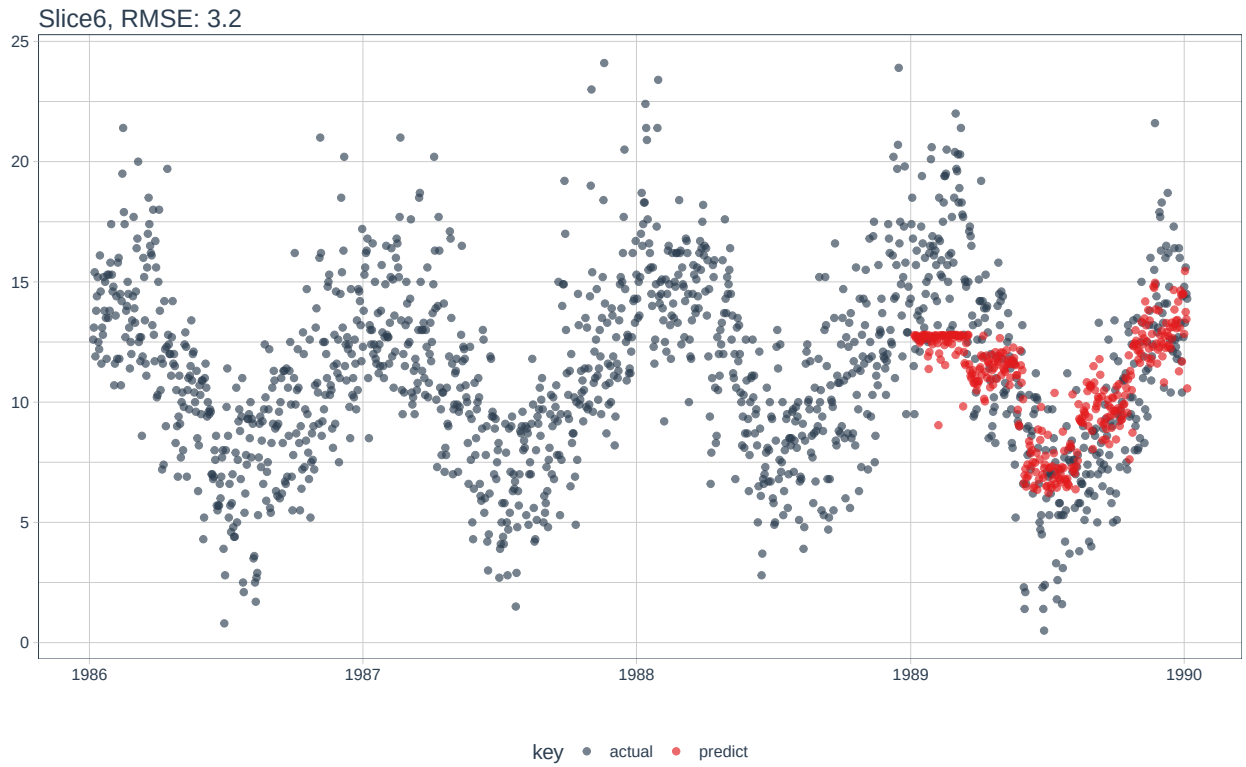
```
## [1] 3.228733
```

Por el momento, el RMSE no dice nada, sin embargo, durante la validación con backtesting, este será útil para determinar un error de predicción esperado.

7.14 Visualización de la predicción del ejemplo sencillo

```
pred_grafica_completa <- function(data, id, alpha = 1, size = 2,
                                  base_size = 14) {

  rmse_val <- calc_rmse(data)

  g <- data %>%
    ggplot(aes(index, value, color = key)) +
    geom_point(alpha = alpha, size = size) +
    theme_tq(base_size = base_size) +
    scale_color_tq() +
    theme(legend.position = "none") +
    labs(
      title = glue("{id}, RMSE: {round(rmse_val, digits = 1)}"),
      x = "", y = ""
    )

  return(g)
}

ret %>%
  pred_grafica_completa(id = split_id, alpha = 0.65) +
  theme(legend.position = "bottom")
```

Se puede observar en esta gráfica que el modelo LSTM realizó predicciones muy cercanas a las originales y que además logra capturar de manera adecuada la estacionalidad de la serie de tiempo. Con esto, finaliza el ejemplo sencillo que ilustra la funcionalidad de las redes LSTM. A continuación, se procede a la implementación del modelo con todas las particiones y además se utiliza el método de validación *backtesting*.



7.15 Modelo LSTM y Backtesting para todos los datos (1:6 splits)

Una vez que se explicó el modelo sencillo, se escala el modelo a las 6 particiones con la siguiente función:

```
# Se siguen los mismos pasos que en el ejemplo sencillo

predecir_lstm_completo <- function(split, epochs = 300, ...) {

  prediccion_lstm <- function(split, epochs, ...) {

    # Separación de datos
    entrenamiento_df <- training(split)
    prueba_df <- testing(split)

    # Convertir a formato de tiempo
    df <- bind_rows(
      entrenamiento_df %>% add_column(key = "training"),
      prueba_df %>% add_column(key = "testing")
    ) %>%
      as_tbl_time(index = index)
```

```r
# Centrar y Escalar Datos
rec_obj <- recipe(value ~ ., df) %>%
  step_center(value) %>%
  step_scale(value) %>%
  prep()

datos_transformados <- bake(rec_obj, df)

media_historica <- rec_obj$steps[[1]]$means["value"]
escala_historica  <- rec_obj$steps[[2]]$sds["value"]

# Inputs de la red LSTM
lag_setting  <- 365 # = nrow(prueba_df)
batch_size   <- 73
train_length <- 365*3
tsteps       <- 1
epochs       <- 300

# Preparación de los datos para ser leídos propiamente por Keras
lag_entrenamiento_tbl <- datos_transformados %>%
  mutate(value_lag = lag(value, n = lag_setting)) %>%
  filter(!is.na(value_lag)) %>%
  filter(key == "training") %>%
  tail(train_length)

x_train_vec <- lag_entrenamiento_tbl$value_lag
x_train_arr <- array(data = x_train_vec, dim = c(length(x_train_vec), 1, 1))

y_train_vec <- lag_entrenamiento_tbl$value
y_train_arr <- array(data = y_train_vec, dim = c(length(y_train_vec), 1))

lag_prueba_tbl <- datos_transformados %>%
  mutate(
    value_lag = lag(value, n = lag_setting)
  ) %>%
  filter(!is.na(value_lag)) %>%
  filter(key == "testing")

x_test_vec <- lag_prueba_tbl$value_lag
x_test_arr <- array(data = x_test_vec, dim = c(length(x_test_vec), 1, 1))

y_test_vec <- lag_prueba_tbl$value
y_test_arr <- array(data = y_test_vec, dim = c(length(y_test_vec), 1))
```

```
# Modelo LSTM con Keras para 6 particiones
model <- keras_model_sequential()

model %>%
  layer_lstm(units            = 50,
             input_shape      = c(tsteps, 1),
             batch_size       = batch_size,
             return_sequences = TRUE,
             stateful         = TRUE) %>%
  layer_lstm(units            = 50,
             return_sequences = FALSE,
             stateful         = TRUE) %>%
  layer_dense(units = 1)

model %>%
  compile(loss = 'mae', optimizer = 'adam')

model


# Ajuste del modelo LSTM
for (i in 1:epochs) {
  model %>% fit(x          = x_train_arr,
                y          = y_train_arr,
                batch_size = batch_size,
                epochs     = 1,
                verbose    = 1,
                shuffle    = FALSE)

  model %>% reset_states()
  cat("Epoch: ", i)

}

# Predicciones
pred_out <- model %>%
  predict(x_test_arr, batch_size = batch_size) %>%
  .[,1]

# Se regresan a escala original los datos
pred_tbl <- tibble(
  index   = lag_prueba_tbl$index,
```

```
      value   = (pred_out * escala_historica + media_historica)
    )

    # Combinar infromación actual con predicciones en una tibble de serie de tiempo
    tbl_1 <- entrenamiento_df %>%
      add_column(key = "actual")

    tbl_2 <- prueba_df %>%
      add_column(key = "actual")

    tbl_3 <- pred_tbl %>%
      add_column(key = "predict")

    time_bind_rows <- function(data_1, data_2, index) {
      index_expr <- enquo(index)
      bind_rows(data_1, data_2) %>%
        as_tbl_time(index = !! index_expr)
    }

    ret <- list(tbl_1, tbl_2, tbl_3) %>%
      reduce(time_bind_rows, index = index) %>%
      arrange(key, index) %>%
      mutate(key = as_factor(key))

    return(ret)

  }

  safe_lstm <- possibly(prediccion_lstm, otherwise = NA)

  safe_lstm(split, epochs, ...)

}
```

Se manda llamar la función anterior (`predecir_lstm_completo()`) y con 10 epocas se predicen los valores en formato largo para cada partición.

```
predecir_lstm_completo(split, epochs = 10)
```

```
## Epoch:  1Epoch:  2Epoch:  3Epoch:  4Epoch:  5Epoch:  6Epoch:  7Epoch:  8Epoch:  9Epoch:  10Epoch:  1
```

```
## # A time tibble: 1,825 x 3
## # Index: index
##    index      value key
##    <date>     <dbl> <fct>
```

```
##  1 1986-01-06  12.6 actual
##  2 1986-01-07  13.1 actual
##  3 1986-01-08  15.4 actual
##  4 1986-01-09  11.9 actual
##  5 1986-01-10  13.8 actual
##  6 1986-01-11  14.4 actual
##  7 1986-01-12  15.2 actual
##  8 1986-01-13  12.5 actual
##  9 1986-01-14  12.2 actual
## 10 1986-01-15  16.1 actual
## # ... with 1,815 more rows
```

7.16 Generación de predicciones para las particiones

Con esta función se almacenan las predicciones en una columna que contiene elementos tipo lista para cada uno de las particiones.

```
predicciones_tbl_completo <- rolling_origin_resamples %>%
  mutate(predict = map(splits, predecir_lstm_completo, epochs = 300))
```

```
## Epoch:  1Epoch:  2Epoch:  3Epoch:  4Epoch:  5Epoch:  6Epoch:  7Epoch:  8Epoch:  9Epoch:  10Epoch:  1
```

```
predicciones_tbl_completo
```

```
## # Rolling origin forecast resampling
## # A tibble: 6 x 3
##   splits            id    predict
## * <list>            <chr> <list>
## 1 <split [1.1K/365~ Slic~ <tibble [1,825 x ~
## 2 <split [1.1K/365~ Slic~ <tibble [1,825 x ~
## 3 <split [1.1K/365~ Slic~ <tibble [1,825 x ~
## 4 <split [1.1K/365~ Slic~ <tibble [1,825 x ~
## 5 <split [1.1K/365~ Slic~ <tibble [1,825 x ~
## 6 <split [1.1K/365~ Slic~ <tibble [1,825 x ~
```

Ahora, se genera una tabla con los RMSE de las particiones.

```
rmse_tbl_completo <- predicciones_tbl_completo %>%
  mutate(rmse = map_dbl(predict, calc_rmse)) %>%
  select(id, rmse)

rmse_tbl_completo
```

```
## # Rolling origin forecast resampling
## # A tibble: 6 x 2
```

```
##   id      rmse
## * <chr>  <dbl>
## 1 Slice1  3.24
## 2 Slice2  3.20
## 3 Slice3  2.92
## 4 Slice4  3.26
## 5 Slice5  3.45
## 6 Slice6  3.23
```

Se hace el promedio de los RMSE de las 6 particiones para tener un indicador del desempeño de este modelo, en caso de que se quisiera comparar con otros modelos.

```
rmse_tbl_completo %>%
  summarize(
    mean_rmse = mean(rmse),
    sd_rmse   = sd(rmse)
  )
```

```
## # Rolling origin forecast resampling
## # A tibble: 1 x 2
##   mean_rmse sd_rmse
##       <dbl>   <dbl>
## 1      3.22   0.171
```

7.17 Gráficas de predicción utilizando la estrategia de Backtesting

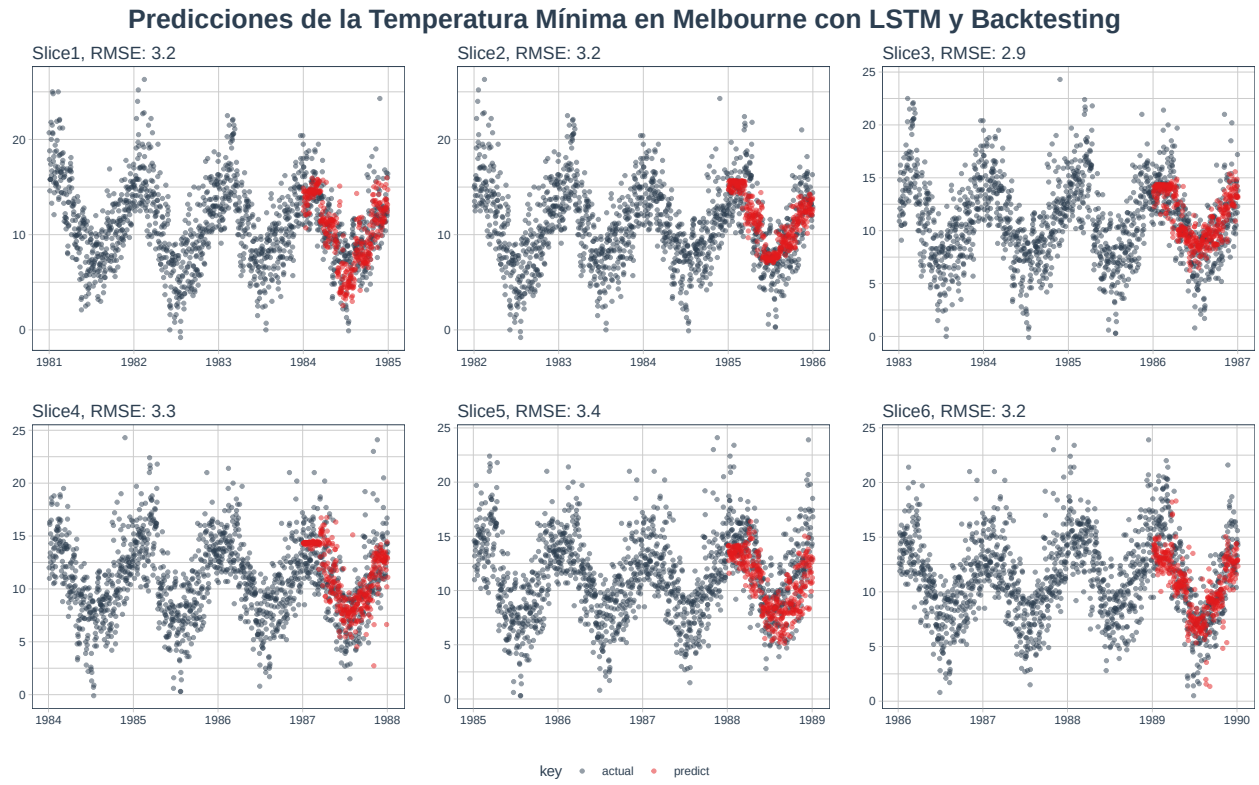
Se crea una función que grafica en una sola imagen las predicciones para cada partición del conjunto de datos.

```
grafica_predicciones_completo <- function(sampling_tbl, predictions_col,
                              ncol = 3, alpha = 1, size = 2, base_size = 14,
                              title = "Backtested Predictions") {

  predicciones_col_expr <- enquo(predictions_col)

  tbl_plots_completo <- sampling_tbl %>%
    mutate(gg_plots = map2(!! predicciones_col_expr, id,
                           .f        = pred_grafica_completa,
                           alpha     = alpha,
                           size      = size,
                           base_size = base_size))

  plot_list <- tbl_plots_completo$gg_plots
```

```
  p_temp <- plot_list[[1]] + theme(legend.position = "bottom")
  legend <- get_legend(p_temp)

  p_body  <- plot_grid(plotlist = plot_list, ncol = ncol)

  p_title <- ggdraw() +
    draw_label(title, size = 18, fontface = "bold", colour = palette_light()[[1]])

  g <- plot_grid(p_title, p_body, legend, ncol = 1, rel_heights = c(0.05, 1, 0.05))

  return(g)
}
```

Se manda llamar la función anterior para hacer la imagen con las 6 gráficas y sus respectivas predicciones.

```
predicciones_tbl_completo %>%
  grafica_predicciones_completo(predictions_col = predict, alpha = 0.5, size = 1, base_size = 10,
                  title = "Predicciones de la Temperatura Mínima en Melbourne con LSTM y Backtesting")
```



La imagen anterior muestra que para cada partición se realizaron

predicciones muy parecidas a los casos reales, por esta razón se utilizara el modelo entrenado para predecir un año a futuro.

7.18 Predicciones a Futuro

Dados los resultados anteriores, que aparentan ser muy buenos, se predice un año adicional (1991) con el modelo previamente entrenado.

```
predecir_lstm_completo_future <- function(data, epochs = 300, ...) {

  prediccion_lstm <- function(data, epochs, ...) {

    df <- data

    #Centrar y escalar datos
    rec_obj <- recipe(value ~ ., df) %>%
      step_center(value) %>%
      step_scale(value) %>%
      prep()

    datos_transformados <- bake(rec_obj, df)

    media_historica <- rec_obj$steps[[1]]$means["value"]
    escala_historica  <- rec_obj$steps[[2]]$sds["value"]

    #Inputs de la red LSTM
    lag_setting  <- 365 # = nrow(prueba_df)
    batch_size   <- 73
    train_length <- 365*3
    tsteps       <- 1
    epochs       <- 300

    #Preparación de los datos para ser leídos propiamente por keras
    lag_entrenamiento_tbl <- datos_transformados %>%
      mutate(value_lag = lag(value, n = lag_setting)) %>%
      filter(!is.na(value_lag)) %>%
      tail(train_length)

    x_train_vec <- lag_entrenamiento_tbl$value_lag
    x_train_arr <- array(data = x_train_vec, dim = c(length(x_train_vec), 1, 1))

    y_train_vec <- lag_entrenamiento_tbl$value
    y_train_arr <- array(data = y_train_vec, dim = c(length(y_train_vec), 1))
```

```
x_test_vec <- y_train_vec %>% tail(lag_setting)
x_test_arr <- array(data = x_test_vec, dim = c(length(x_test_vec), 1, 1))

#Modelo LSTM con Keras
model <- keras_model_sequential()

model %>%
  layer_lstm(units            = 50,
             input_shape      = c(tsteps, 1),
             batch_size       = batch_size,
             return_sequences = TRUE,
             stateful         = TRUE) %>%
  layer_lstm(units            = 50,
             return_sequences = FALSE,
             stateful         = TRUE) %>%
  layer_dense(units = 1)

model %>%
  compile(loss = 'mae', optimizer = 'adam')

#Ajuste del modelo LSTM
for (i in 1:epochs) {
  model %>% fit(x          = x_train_arr,
                y          = y_train_arr,
                batch_size = batch_size,
                epochs     = 1,
                verbose    = 1,
                shuffle    = FALSE)

  model %>% reset_states()
  cat("Epoch: ", i)

}

#Predicciones
pred_out <- model %>%
  predict(x_test_arr, batch_size = batch_size) %>%
  .[,1]

#Año futuro
idx <- data %>%
  tk_index() %>%
  tk_make_future_timeseries(n_future = lag_setting)
```

```
    #Regresar a escala original
    pred_tbl <- tibble(
      index   = idx,
      value   = (pred_out * escala_historica + media_historica)
    )

    #Combinar datos originales con predichos
    tbl_1 <- df %>%
      add_column(key = "actual")

    tbl_3 <- pred_tbl %>%
      add_column(key = "predict")

    time_bind_rows <- function(data_1, data_2, index) {
      index_expr <- enquo(index)
      bind_rows(data_1, data_2) %>%
        as_tbl_time(index = !! index_expr)
    }

    ret <- list(tbl_1, tbl_3) %>%
      reduce(time_bind_rows, index = index) %>%
      arrange(key, index) %>%
      mutate(key = as_factor(key))

    return(ret)

  }

  safe_lstm <- possibly(prediccion_lstm, otherwise = NA)

  safe_lstm(data, epochs, ...)

}
```

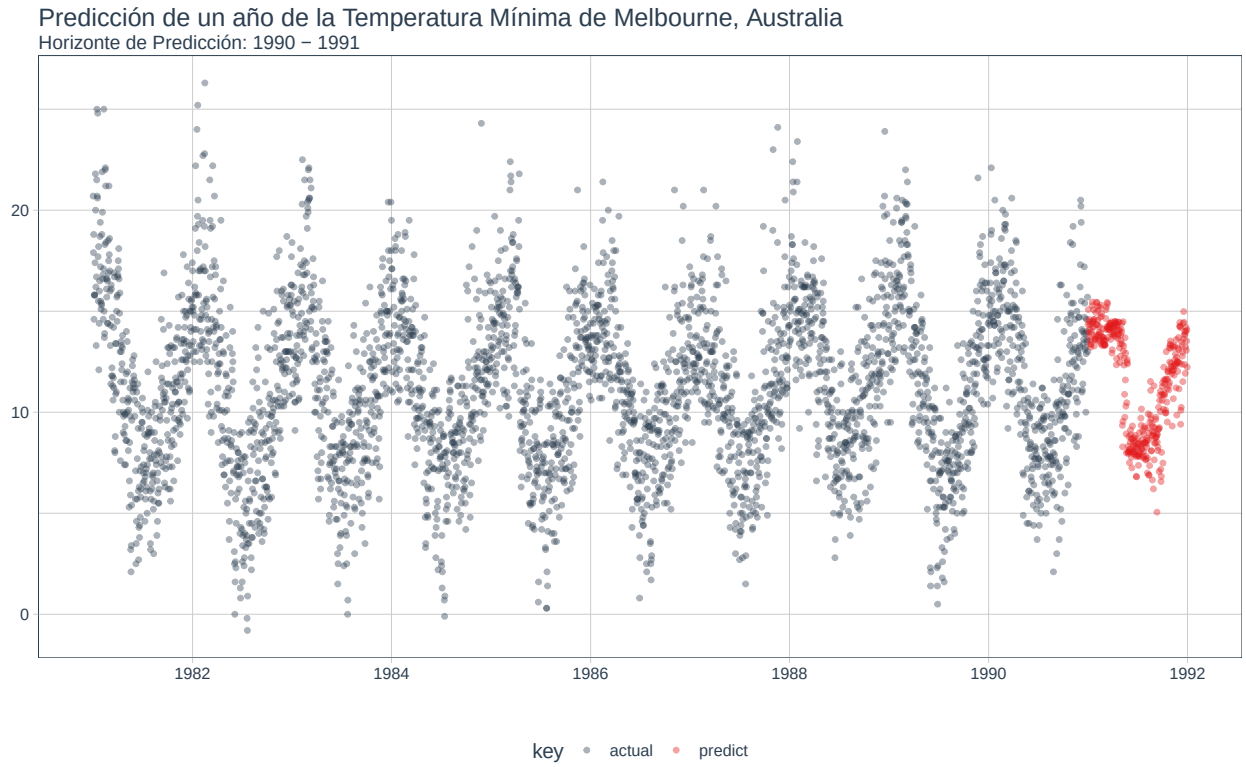
Se manda llamar la función previa para realizar la predicción del año 1991.

```
future_australia_ts_tbl <- predecir_lstm_completo_future(australia_ts, epochs = 300)
```

```
## Epoch:  1Epoch:  2Epoch:  3Epoch:  4Epoch:  5Epoch:  6Epoch:  7Epoch:  8Epoch:  9Epoch:  10Epoch:  1
```

Para finalizar, se hace la gráfica con la función pred_grafica_completa y se obtiene lo siguiente:

```
future_australia_ts_tbl %>%
  filter_time('1981-01-01' ~ "end") %>%
  pred_grafica_completa(id = NULL, alpha = 0.4, size = 1.5) +
  theme(legend.position = "bottom") +
  ggtitle("Predicción de un año de la Temperatura Mínima de Melbourne, Australia", subtitle = "Horizonte
```



Se puede observar en esta imagen una predicción que mantiene el mismo patrón cíclico que las predicciones anuales anteriores y los datos orginales. Por esta razón, se comprueba que el modelo LSTM es bueno recordando horizontes temporales pasados, en este caso, mediciones de temperaturas mínimas a lo largo de 9 años, esto nos da una buena señal de que el modelo puede ser aplicado a otras bases de datos donde la correlación temporal sea alta.

8 Conclusiones

Las redes neuronales LSTM son un tipo especial de red neuronal recurrente que son capaces de procesar datos secuenciales y almacenar ciertos datos en una "memoria" de largo plazo. En este documento se exploraron dos aplicaciones: clasificación binaria en datos secuenciales (texto) y predicción para series de

tiempo. Este tipo de redes son las adecuadas para secuencias largas porque no sufren del problema de sensibilidad de gradiente (que sí tienen otro tipo de redes, como las recurrentes, o las *feedforward*). En análisis de texto (y clasificación de sentimiento), la principal complejidad radica en pre-procesar los datos: i) existen diferentes métodos para "tokenizar" las palabras; y ii) es difícil la conversión de texto a vectores numéricos (en este caso se solucionó con "*word embeddings*"). En análisis de series de tiempo, la principal complejidad de aplicar este modelo para predicción es que es fácil sobreajustar los datos y, además, es difícil hacer validación cruzada. En este documento *backtesting* demostró ser un buen método de validación para comprobar que la red LSTM podía ajustarse bien a los datos. En específico, que la red capturaba bien la estacionalidad de los datos y en cuanto a las predicciones, la red replicaba los patrones cíclicos observados en la serie. Al final, vimos que, para esta serie en específico, el error de predicción, medido por RMSE, fue bastante homogéneo y por tanto el modelo no sobreajusta. Para clasificación de sentimiento, en general estas redes tienen una mayor precisión que la regresión logística u otras redes neuronales (que podrían volverse no-entrenables para dicha aplicación).