
Universidad de Granada
*Departamento de Estadística e Investigación Operativa*
MÁSTER UNIVERSITARIO EN ESTADÍSTICA APLICADA

---

# Series Temporales Avanzadas: Aplicación de Redes Neuronales para el Pronóstico de Series de Tiempo

Trabajo de Fin de Máster:
**Ricardo Alonzo Fernández Salguero**

Director:
**Dr. Francisco Javier Alonso Morales**

Septiembre 2021

**Resumen**

Las redes neuronales son muy útiles para predecir valores futuros en lo que respecta a series de tiempo, ejemplificando un modelo tradicional como es el modelo ARIMA, con una red neuronal simple, una red neuronal recurrente simple y una red neuronal de memoria a largo plazo y corto plazo, se observa diferencias en cuanto la capacidad predictiva de estos modelos, sin embargo se recalca que tanto los modelos tradicionales como las redes neuronales tienen ventajas y desventajas, los modelos tradicionales son muy útiles en problemas más sencillos o con pocos datos, mientras que las redes neuronales son ventajosas en contextos de grandes volúmenes de datos. Se presenta una exposición teórica de estos modelos y una aplicación práctica con Keras y Tensorflow.

# Índice general

# Índice de figuras

# Introducción 1

Durante el siglo XX y el siglo XXI se reunieron muchos esfuerzos para predecir el precio o el rendimiento futuro de un activo en un mercado financiero, debido a que es un objetivo necesario para reducir el riesgo en el proceso de toma de decisiones al determinar con precisión el movimiento futuro de un activo. (Batres-Estrada, 2015)

La previsión de series de tiempo financieras tradicionales a menudo se basa únicamente en modelos económicos/financieros. Sin embargo, recientemente, está evolucionando rápidamente para aplicar de manera integral varios campos a la predicción del mercado financiero, y la predicción del mercado financiero se ha convertido en un campo prometedor de investigación financiera.

El mercado financiero es un sistema que se considera ruidoso, de propiedades estocásticas, por lo que interfiere con el desempeño de las previsiones. (Sezer, 2020; Liu, 2020; Navon, 2017)

Los métodos tradiciones provocan problemas como una calibración excesiva o un retraso de convergencia en la aplicación. Y es que, el análisis y la previsión de datos financieros siempre ha sido una tarea abrumadora en la industria financiera, y los métodos de previsión analítica existentes también tienen desafíos específicos. Por ejemplo, los enfoques paramétricos tradicionales pueden no ser adecuados para analizar datos financieros complejos, en grandes volúmenes y en presencia de mucho ruido estocástico.(Pedrycz, 2020)

Los modelos tradicionales de características no lineales aún no han podido modelar con precisión estos datos de los sistemas financieros cuya complejidad es tan elevada.

El rendimiento de los métodos tradicionales de aprendizaje automático se

basa en gran medida en el diseño funcional y de arquitectura del modelo. La inteligencia artificial es un área de gran éxito en la investigación de resolución de problemas como es el caso del procesamiento del lenguaje natural, en la clasificación de imágenes y, a su vez diversas tareas de series de tiempo.

Detrás de este proceso se encuentra una de las evoluciones de un marco de aprendizaje automático que es conocido como aprendizaje profundo, o también llamado Deep Learning.

En el Deep Learning la estructura básica es una red neuronal multicapa con una potente potencia informática y disponibilidad que permite e incluso se potencia en entornos de Big Data. Es decir, son modelos más potentes en grandes volúmenes de datos, que pueden ser variados (estructurados como no estructurados), que según la capacidad computacional, la arquitectura de la red y los algoritmos de optimización usados puede ser utilizado incluso con datos que son procesados en streaming.

Las técnicas de aprendizaje profundo desarrolladas en una variedad de marcos, incluidos codificadores automáticos apilables, redes profundas, redes neuronales complejas profundas, brindan nuevos conocimientos sobre la prospectiva del mercado financiero.

El aprendizaje profundo procesa información como el procesamiento de datos en el cerebro (o en parte se inspira de ello). A través de nuevos algoritmos se mejora las redes neuronales existentes, e incluso existen técnicas como la transferencia de aprendizaje donde de una serie de capas ocultas preentrenadas se agregan más con nuevos datos para mejorar las generalizaciones.

El aplicar Deep learning a las finanzas no solo alivia algunos de los problemas analíticos y de previsión, sino que también conduce a un cambio de paradigma en el análisis financiero empírico, que hoy en día, crece a pasos exponenciales.

Con el advenimiento de la era inteligente y la creciente demanda de pronósticos de series de tiempo financieras, el aprendizaje profundo se ha convertido en una aplicación de futuro e incluso de presente exitoso en el sector financiero.

El aprendizaje profundo se utiliza ampliamente para la previsión del mercado financiero, como es en:

- La previsión de volatilidad,
- La previsión de precios y
- La previsión de tendencias

La investigación en el aprendizaje automático y a su vez en el aprendizaje profundo se concentra en:

- Diseñar nuevas arquitecturas
- Ajustar varias estructuras de tiempo, de múltiples variables.
- Ejecutar e integrar soluciones diversas
- Con el propósito genera de producir mejores resultados predictivos.

Los investigadores evalúan el rendimiento predictivo del aprendizaje profundo en contextos diferentes de series de tiempo, concluyen que aunque existan muchos contextos donde el Deep Learning posee ventajas, no es así en todos los contexto, el método de aprendizaje profundo puede mejorar la precisión de la predicción cuando se poseen muchos datos, pero los modelos tradicionales pueden ser más eficientes para resolver problemas más sencillos, o que por su naturaleza requieran de soluciones analíticas.

Este trabajo pretende mostrar el uso de redes neuronales aplicadas a series de tiempo, sin desmerecer que en diferentes contextos los modelos tradicionales siguen siendo alternativas útiles, menos complicadas que las redes neuronales y de mayor capacidad explicativa en el sentido mecanísmico, debido a que muchas redes neuronales son cajas negras.

# Capítulo 1: Las redes neuronales 2

Redes neuronales artificiales (RNA), son una rama de la inteligencia artificial, que se remonta a la década de 1940. Fueron McCulloch y Pitts los que desarrollaron los primeros modelos de redes neuronales.(Kelleher, 2019)

Ha habido un interés generalizado en las redes neuronales artificiales, tanto por parte de los investigadores como en varios campos de aplicación y, se han introducido redes más complejas, con mejores y más optimizados algoritmos de entrenamiento, a la vez que el hardware mejoraba.(Pedrycz, 2020)

El problema fundamental resuelto por las RNA es la adquisición inductiva del aprendizaje, a través de ejemplos.

La capacidad de aprender de los datos y generalizar, es decir, de imitar la capacidad de aprender de la experiencia humana, ayuda a las RNA a automatizar el proceso de aprendizaje de varias reglas de aplicación.

Las RNA son uno de los modelos de pronóstico más precisos y ampliamente utilizados, es útil para pronósticos en investigaciones de índole de sociales, económicos, técnicos, monetarios, de mercado de valores, etc. Las redes neuronales artificiales son atractivas e interesantes para tareas de predicción. (Pedrycz, 2020) Debido a que:

• A diferencia de los métodos tradicionales basados en modelos, las redes neuronales son una adaptación única de la base de datos, por lo que hay pocos modelos a priori para estudiar problemas.

• Las redes neuronales artificiales se pueden generalizar conociendo los datos (muestra) proporcionados, las RNA pueden inferir correctamente la parte invisible del conjunto, incluso si contiene información de ruido en los datos de

muestra.

• Las RNA son una aproximación funcional universal. Se ha demostrado que la red puede aproximar todas las funciones continuas con la precisión requerida.

• Las redes neuronales artificiales no son lineales, por sus funciones de activación, por lo que encontrarán relaciones no lineales en el conjunto de entrenamiento.

Los enfoques tradicionales para el pronóstico de series de tiempo, como lo es, por ejemplo la metodología de Box Jenkins y ARIMA, asumen que la serie de tiempo bajo investigación se corresponde mediante un proceso lineal. Esto frente a las RNA en general que no lo suponen.

Habrá que notar entonces, que no son apropiadas las RNA si el mecanismo que se pretende investigar no es lineal.

Las RNA, se utilizan ampliamente para funciones de estimación y predicción. Una de las ventajas más importantes del modelado de RNA sobre otros tipos de modelos no lineales es que las RNA son un método de aproximación universal que puede aproximar con precisión una amplia gama de funciones para a su vez, una amplia gama de datos. (Kelleher, 2019)

De lo anterior se desprende que:

• El proceso de modelado no requiere predecir la forma del modelo.

• El modelo de red está determinado en gran medida por la naturaleza de los datos.

El modelo de las RNA se caracteriza por una red de tres capas de unidades de procesamiento simples conectadas por sus enlaces. La relación entre la salida $(y_t)$ y las entradas $(y_{t-1}, \ldots, y_{i-p})$ tiene la siguiente representación matemática:

$$y_t = w_0 + \sum_{j-1}^{q} w_j \cdot g\left(w_{0,j} + \sum_{i-1}^{p} w_{i,j} \cdot y_{t-i}\right) + \varepsilon_t$$

dónde, $w_{i,j}(i = 0, 1, 2, \ldots, p, j = 1, 2, \ldots, q)$ y $w_j(j = 0, 1, 2, \ldots, q)$ son parámetros del modelo que a menudo se denominan pesos de conexión; $p$ es el número de nodos de entrada; y $q$ es el número de nodos ocultos.

Si se utiliza tan solo las entradas y las salidas de una red neuronal la función del modelo es equivalente a una regresión lineal, es decir, necesitará de funciones de activación no lineales y unidades o capas ocultas para diferenciarse.(Kelleher,

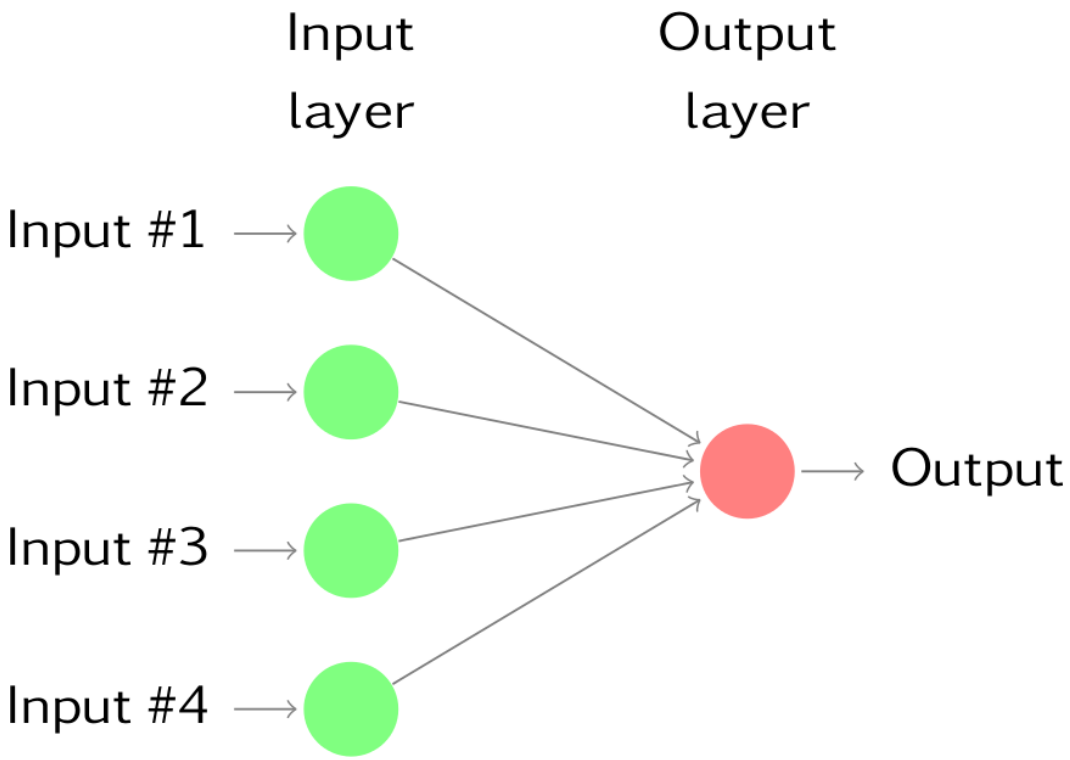


FIGURA 2.1: Ilustración extraída de (Hyndman and Athanasopoulos, 2018) como ejemplo de una red neuronal simple equivalente a una regresión lineal.

2019) Por ejemplo la Figura 2.1. muestra una red neuronal simple que al no poseer capas ocultas es equivalente a una regresión lineal.

La Figura 2.2. ilustra una red neuronal pequeña, que poseerá una capa oculta y en ella neuronas ocultas, que permitirán distribuir los pesos de los parámetros a la vez que utilizan funciones de activación no lineales.

El tipo de función de activación se muestra en la situación de la neurona dentro de la red. En la mayoría de los casos, las neuronas de nivel de entrada no tienen función de activación porque su función es transmitir las entradas a la capa oculta.(Kelleher, 2019)

La función de activación más utilizada para el nivel de salida es la función lineal, ya que la función de activación no lineal puede insertar una distorsión en la salida anterior. Las funciones logísticas e hiperbólicas se utilizan a menudo como una función de transferencia de nivel oculto. (Pedrycz, 2020)

También se pueden usar otras funciones de activación, como lineal y cuadrado.

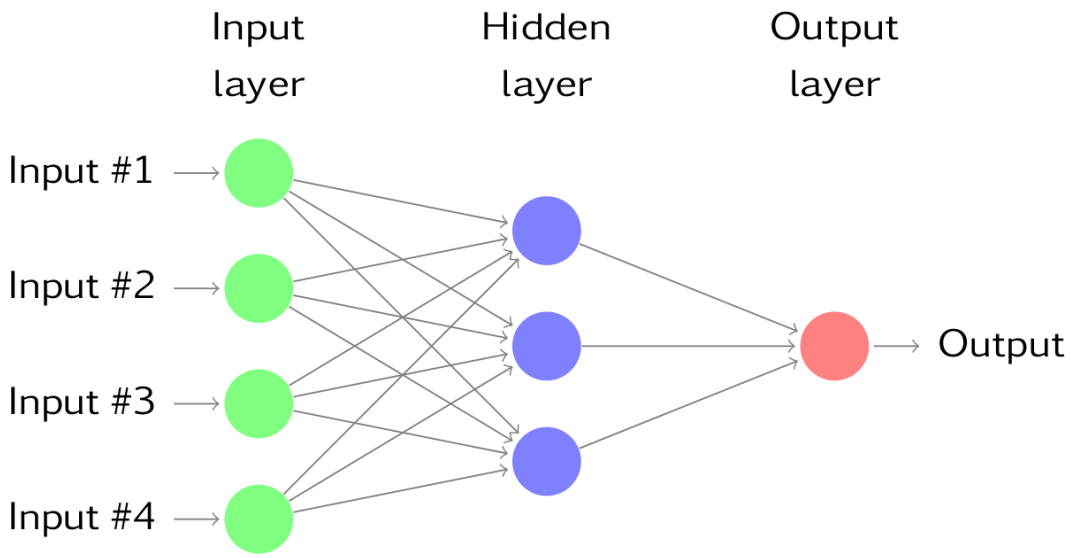


Figura 2.2: Ilustración extraída de (Hyndman and Athanasopoulos, 2018) como ejemplo de una red neuronal con cuatro entradas y una capa oculta con tres neuronas ocultas.

$$\text{Sig}(x) = \frac{1}{1 + \exp(-z)}$$

$$\text{Tanh}(x) = \frac{1 - \text{cop}(-2x)}{1 + \text{cop}(-2x)}.$$

Por lo tanto, el modelo RNA, de hecho, realiza un mapeo funcional no lineal de las observaciones pasadas al valor futuro $y_t$, es decir:

$$y_t = f\left(y_{t-1}, \ldots, y_{t-p}, w\right) + \varepsilon_t$$

donde, $w$ es un vector de todos los parámetros y $f(\cdot)$ es una función determinada por la estructura de la red y los pesos de conexión. Por tanto, la red neuronal es equivalente a un modelo autorregresivo no lineal .

La red simple es sorprendentemente poderosa porque puede aproximarse a cualquier función del número de nodos ocultos cuando $q$ es lo suficientemente grande (Pedrycz, 2020) .

Las estructuras de red simples con una pequeña cantidad de nodos realmente ocultos a menudo funcionan con predicciones fuera de la muestra. Este podría ser un efecto de supervisión común al modelar redes neuronales.

Hay varias formas de encontrar la mejor arquitectura para RNA, pero estos métodos suelen ser muy complejos y difíciles de implementar, incluso tienen alto costo computacional. (Kelleher, 2019)

No existen métodos que puedan garantizar la mejor solución a todos los problemas de predicción del mundo real. Hasta la fecha, no existe una forma fácil y clara de determinar hiperparámetros.

Como es común, un procedimiento para determinar mejores modelos dados sus hiperparámetros es probar varias redes con diferentes números de entrada y unidades o capas ocultas, así, se pueda estimar su error general y luego seleccionar la red con la menor cantidad de errores.(Kelleher, 2019)

Una vez que se especifica la forma funcional y arquitectura de la red, la red está lista para comenzar el proceso de estimación de parámetros. Los parámetros se estiman para minimizar la función de costo de la red neuronal.

La función de costo es una medida de precisión estándar dónde comúnmente se escoge al error cuadrático medio tanto del set de entrenamiento como del set de prueba (Pedrycz, 2020) .

$$E = \frac{1}{N}\sum_{n=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{n=1}^{N}\left(y_i - \left(w_0 + \sum_{j=1}^{Q} w_j g\left(w_{0j} + \sum_{i=1}^{P} w_{i,j} y_{t-4}\right)\right)\right)^2$$

donde, N es el número de términos de error.

Esta mìnimización se realiza con algunos algoritmos de optimización no lineales eficientes distintos del algoritmo básico de entrenamiento de retropropagación o backpropagation en el que los parámetros de la red neuronal, $w_{i,j}$ se cambian por una cantidad $\Delta w_{i,j}$, según la siguiente fórmula:

$$\Delta w_{\delta,j} = -\eta \frac{\partial E}{\partial w_{ij}}$$

donde, el parámetro $\eta$ es la tasa de aprendizaje $y \frac{\partial E}{\partial u_j j}$ es la derivada parcial de la función F con respecto al peso $w_{ij}$. Esta derivada se calcula comúnmente en dos pasos. En el paso hacia adelante, un vector de entrada del conjunto de entrenamiento se aplica a las unidades de entrada de la red y se propaga a través de la red, capa por capa, produciendo la salida final.

Durante el paso hacia atrás, la salida de la red se compara con la salida deseada y el error resultante luego se propaga hacia atrás a través de la red,

ajustando los pesos en cada proceso. Para acelerar el proceso de aprendizaje, evitando la inestabilidad del algoritmo se toma siguiente regla:

$$\Delta w_{i,j}(t+1) = -\eta \frac{\partial E}{\partial r_{ij}} + \delta \Delta w_{i,j}(t)$$

El término de impulso también puede ser útil para evitar que el proceso de aprendizaje quede atrapado en mínimos locales lejanos al mínimo global, y generalmente se elige en el intervalo [0; 1].

Finalmente, el modelo estimado se evalúa utilizando una muestra de reserva separada que no está expuesta al proceso de capacitación o entranamiento (Pedrycz, 2020).

# Capítulo 2: Los modelos ARIMA 3

Los modelos de media móvil integrada autorregresiva (ARIMA) han dominado muchas áreas de la predicción de series de tiempo, en finanzas, retail, clima, etc (Pedrycz, 2020; Dingli, 2017).

En un modelo ARIMA, se supone que el valor futuro de una variable es una función lineal de varias observaciones pasadas y errores aleatorios. Es decir, el proceso subyacente que genera la serie temporal con la media tiene la forma:

$$\phi(B)\nabla^d\left(y_t - \mu\right) = \theta(B)a_i$$

dónde, $y_t$ y $a_t$ son el valor real y el error aleatorio en el período de tiempo t, respectivamente; $\phi(B) = 1 - \sum_{i=1}^{p} \varphi_i B^i, \theta(B) = 1 - \sum_{j-1}^{q} \theta_j B^j$ son polinomios en $B$ de grado p y q $\phi_i(i = 1, 2, \ldots, p)_y \theta_j(j = 1, 2, \ldots, q)$ son parámetros del modelo, $\nabla = (1 - B)$, B es el operador de desplazamiento hacia atrás, p y q son números enteros y a menudo se hace referencia como órdenes de modelo, d es un número entero $y_1$z a menudo referido como orden de diferenciación.

Los errores aleatorios $a_t$ se consideran independientes y se distribuyen de manera similar con una varianza constante $\sigma^2$ de media 0.

El método de Box y Jenkins consta de tres pasos iterativos:

- Definición del modelo,
- Estimación de parámetros y
- Validación diagnóstica.

La idea básica del reconocimiento de patrones es que si el proceso ARIMA

produce una serie de tiempo, debe tener los atributos teóricos de autocorrelación (Karakoyun, 2018).

Al hacer coincidir un modelo de autocorrelación experimental con un modelo teórico, a menudo identificamos múltiples modelos potenciales en una serie de tiempo determinada. Box y Jenkins sugieren usar la función de autocorrelación (ACF) y la función de autocorrelación parcial (PACF) de los datos de la muestra como herramientas básicas para determinar el orden del modelo ARIMA.

Se han propuesto varios métodos de selección de orden diferentes basados en enfoques de la teoría de la información, como los criterios de validez, los criterios de información de Akaike (AIC) y la longitud mínima de descripción (MDL por sus siglas en inglés).(Karakoyun, 2018)

En el paso de identificación, a menudo se requiere la transformación de datos para hacer estacionaria la serie de tiempo.

La estacionariedad es una condición necesaria en la construcción de un modelo ARIMA utilizado para la previsión.

Una serie de tiempo estacionaria se caracteriza por características estadísticas como la media y la estructura de autocorrelación que son constantes en el tiempo.

Cuando la serie de tiempo observada presenta tendencia y heterocedasticidad, la diferenciación y la transformación de potencia se aplican a los datos para eliminar la tendencia y estabilizar la varianza antes de que se pueda ajustar un modelo ARIMA. (Karakoyun, 2018)

Una vez que se identifica un modelo, la estimación de los parámetros del modelo es sencilla. Los parámetros se estiman de manera que se minimice una medida general de errores.

La minimización también se realiza por una función de coste. Esto se puede lograr mediante un procedimiento de optimización no lineal. El último paso en la construcción de modelos es la verificación de diagnóstico de la adecuación del modelo. Esto es básicamente para verificar si las suposiciones del modelo sobre los errores, se cumplen.

Se pueden usar varias estadísticas de diagnóstico y gráficos de los residuos para examinar la bondad del ajuste del modelo tentativamente considerado a los datos históricos.(Karakoyun, 2018) Si el modelo no es adecuado, se debe identificar un nuevo modelo tentativo, que nuevamente será seguido por los pasos de estimación de parámetros y verificación del modelo.

La información de diagnóstico puede ayudar a sugerir modelos alternativos. Este proceso de construcción de modelos de tres pasos generalmente se

repite varias veces hasta que finalmente se selecciona un modelo satisfactorio.(Karakoyun, 2018)

El modelo final seleccionado se puede entonces utilizar a continuación con fines de predicción.

# Capítulo 3: Descenso de gradiente 4

El descenso de gradiente es un algoritmo para optimizar y para encontrar un mínimo local de función diferencial.

La idea es tomar un paso repetido en la dirección opuesta del gradiente (o degradado aproximado) de la función en el punto estimado, ya que es la dirección más fuerte del descenso. Por otro lado, un paso hacia el gradiente conduce a un máximo local de esta función.(Goodfellow, 2016)

El descenso de gradiente solo requiere el gradiente de la función, luego funciona con una función diferencial. Si puede calcular o enfocar, se puede utilizar para encontrar un mínimo global o un máximo global de una función.

Cuando se encuentra un mínimo global, el algoritmo más simple para el flujo de gradiente es la disminución más fuerte. (Hua, 2019)

La orden es de comenzar con un punto (la referencia de optimización también hace referencia a una elección heurística), luego se crea un paso de módulo en el que la función optimiza y actualiza el punto.

Este algoritmo converge en la mayoría de las iteraciones. Se tiene en cuenta que el orden iterativo de los puntos generados por el flujo de gradiente no se convierte o converge necesariamente al mínimo; puede converger un mínimo local. (Goodfellow, 2016)

Para la convergencia global, necesitamos información sobre la función de degradado en el sentido de "disminución suficiente". El descenso del gradiente tiene una diferencia importante entre los métodos estocásticos y deterministas. En métodos deterministas, se optimiza la función para una prueba individual en cada paso. (Goodfellow, 2016)

En los métodos estocásticos, se evalúa la función en un área de puntos aleatorios para encontrar un valor funcional y seleccionar el mejor valor para la actualización. (Hua, 2019)

Hay muchos métodos diferentes para encontrar un gradiente estocástico. Una función es diferenciable, si, y solo si el gradiente es continuo y, por lo tanto, elimina la cantidad de funciones que pueden disminuir reduciendo el gradiente, incluidas estas funciones para las cuales el gradiente es continuo.

El conjunto de funciones diferenciales es un subconjunto de las funciones reales para las cuales todos los puntos de discontinuidad están abiertos.

Ya que no está garantizado que una búsqueda de descenso desvanecida encuentra un mínimo local o incluso un máximo local de una función diferenciable. Para mejorar el desempeño de la aceptación de los gradientes en una función compleja, se busca reducir la regularización de la función y/o implementar las funciones de precarga. Como se indicó anteriormente, el objetivo del algoritmo es actualizar los parámetros en la dirección del gradiente de la función y luego se usa en principio para cada problema de optimización de parámetros. (Hua, 2019)

El esquema de descenso de gradiente básica hereda todas las ventajas (referidas a demandas computacionales) y todas las desventajas (tasa de convergencia lenta) de la familia algorítmica de descenso de gradiente.(Goodfellow, 2016)

Para acelerar la tasa de convergencia, se invirtió un gran esfuerzo de investigación a fines de la década de 1980 y principios de la de 1990 y se propuso una serie de variantes del esquema de retropropagación de descenso de gradiente básico . (Goodfellow, 2016)

Descenso de gradiente con un término de momento. Una forma de mejorar la tasa de convergencia, sin dejar de estar dentro de la lógica del descenso de gradiente, se emplea el término de momento y el término de corrección ahora se modifica como:

$$\Delta\theta_j^r(\text{new}) = a\Delta\theta_j^r(\text{old}) - \mu \sum_{n=1}^{N} \delta_{nj}^r \boldsymbol{y}_n^{r-1}$$

donde $a$ es el factor de impulso.

En otras palabras, el algoritmo tiene en cuenta la corrección utilizada en el paso de iteración anterior, así como los cálculos de gradiente actuales. (Hua, 2019) Su efecto es aumentar el tamaño del paso, en regiones donde la función de costo presenta una curvatura baja. Suponiendo que el gradiente es aproximadamente constante durante, digamos 1, iteraciones sucesivas, se puede

mostrar que usando el término de momento las actualizaciones son equivalentes a:

$$\Delta\theta_j^r(I) \approx -\frac{\mu}{1-a}\boldsymbol{g}$$

donde $g$ es el valor del gradiente sobre los pasos de iteración sucesivos.

Los valores típicos de $a$ están en el rango de 0,1 a 0,8 (Goodfellow, 2016). Se ha informado de que el uso de un término de impulso puede acelerar la tasa de convergencia hasta un factor de dos. (Goodfellow, 2016)

El Tamaño de paso dependiente de la iteración se produce una variante heuristica de las versiones anteriores de retropropagación si se deja que el tamaño de paso varie a medida que avanzan las iteraciones. Una regla es cambiar su valor según si la función de costo en el paso de iteración actual es mayor o menor en comparación con el anterior.

Sea $J^{(i)}$el valor de costo calculado en la iteración actual. Entonces, si $J^{(i)} < J^{(i-1)}$, la tasa de aprendizaje aumenta en un factor de $r_i$.

Si, por el contrario, el nuevo valor es mayor que el anterior en un factor mayor que $c$, entonces la tasa de aprendizaje se reduce en un factor de $d$. De lo contrario, se mantiene el mismo valor. Los valores típicos de los parámetros involucrados son $r_i = 1{,}05$, $r_d = 0{,}7$ y $c = 1{,}04$. Para los pasos de iteración donde el valor del costo aumenta, es aconsejable establecer el factor de impulso igual a cero.

Otra posibilidad es no realizar la actualización siempre que aumente el costo. Tales técnicas, también conocidas como impulso adaptativos , son más apropiadas para el procesamiento por lotes, porque para las versiones en línea los valores del costo tienden a oscilar de una iteración a otra. (Hua, 2019)

Usar un tamaño de paso diferente para cada peso : es beneficioso para mejorar la tasa de convergencia, emplear un tamaño de paso diferente para cada peso individual; esto le da libertad al algoritmo para explotar mejor la dependencia de la función de costo en cada dirección en el espacio de parámetros.(Goodfellow, 2016)

Se sugiere aumentar la tasa de aprendizaje, asociada con un peso, si el valor de gradiente respectivo tiene el mismo signo para dos iteraciones sucesivas.(Goodfellow, 2016) Por el contrario, la tasa de aprendizaje disminuye si el signo cambia, porque esto es indicativo de una posible oscilación.

Para nuestros propósitos, en el contexto de este algoritmo, una red neuronal artificial es una colección de células de computación (neuronas artificiales) inter-

conectadas a través de enlaces ponderados (sinapsis con diferentes fortalezas). (Hua, 2019)

Las celdas realizan cálculos simples utilizando información disponible localmente o de celdas topológicamente adyacentes a través de enlaces ponderados. Hay una gran variedad de RNAs que difieren en su topología, el comportamiento de las neuronas, de los mecanismos de actualización de peso, cantidad de supervisión o de realimentación requiere, etc. Pero con al menos con redes con 3 capas de células: de entrada, "oculta" y de salida, son unidireccionales o feedforward conexiones. (Hua, 2019)

En una red de múltiples perceptrones (MLP), por ejemplo dada una entrada d -dimensional $\mathbf{x} = [x_1, x_2, \ldots, x_d]^\top$, la j - ésima salida de la red, $y_i$ está dada por:

$y_i(\mathrm{x}) = f\left(\sum_{j=1}^{M} w_{ij} z_j(\mathrm{x}) + \theta_i\right)$

donde $z_j$ es la salida de la $j$ - ésima unidad oculta: $z_j = g\left(\sum_{k=1}^{d} v_{jk} x_k + \theta_j\right)$

En lo anterior, $V_{jk}$ denota el peso de la conexión entre la $j$ - ésima entrada y la k- ésima unidad oculta, y $w_{ij}$ el peso entre la $j$ - ésima unidad oculta y la i- ésima salida. La "función de activación.º función de transferencia $g(\cdot)$ tiene forma de So sigmoidea : no lineal, monotónicamente creciente y acotada. Lo tipicoLa elección es el mapa logistico (a veces llamado sigmoide): $g(a) = \frac{1}{1+e^{-\alpha}}$, , que está acotado entre 0 y 1; o la tangente hiperbólica $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-i}}$, delimitado entre $-1$ y 1. La función de transferencia de salida $f(\cdot)$ puede ser lineal o sigmoidal según sea necesario.

El MLP realiza un mapa estático entre las entradas xy las correspondientes salidas $y(\mathbf{x})$. Este mapa depende de los parámetros u, wy $\theta$ (pesos). Estos pesos son entrenados o ajustan en base a muestras de entrenamiento $\{\mathbf{x}(n), \mathbf{t}(n)]$, donde $\mathbf{t}(n)$ es el vector deseado objetivo para el $n^\circ$ vector de entrada, $x(n)$. Este ajuste se basa en el error $\mathrm{t(n)} - \mathrm{y(x(n))}$, normalmente utilizando un descenso de gradiente estocástico en el valor cuadrático medio de este error, o mediante métodos de segundo orden.

Otra red de feedforward popular para realizar mapas estáticos es la red de función de base radial (RBFN). Una función de base radial (RBF), $\phi$, es aquella cuya salida es simétrica alrededor de un centro asociado,$\mu_0$. Es decir, $\phi_c(\mathbf{x}) = \phi\left(\|x - \mu_c\|\right)$, donde $\|\cdot\|$ es una norma de distancia. Por ejemplo, seleccionando la norma euclidiana $y$ dejando$\phi$(r)=$\mathrm{e}^{-r^2}/\sigma^2$, uno ve que la función gaussiana es un RBF. (Hua, 2019) Hay que tener en cuenta que las funciones gaussianas también se caracterizan por un parámetro de ancho o escala, $\sigma, y$ esto también es cierto para muchas otras clases RBF populares.

Un conjunto de RBF puede servir como base para representar una amplia clase de funciones que se pueden expresar como combinaciones lineales de los RBF elegidos:

$$y_i(\mathrm{x}) = \sum_{j=1}^{M} w_{ij}\phi\left(\left\|\mathbf{x} - \boldsymbol{\mu}_j\right\|\right)$$

Una red de función de base radial no es más que una realización de este ecuación como una red feedforward con tres capas: las entradas, la capa oculta y el (los) nodo (s) de salida. Cada unidad oculta representa una única función de base radial, con la posición central y el ancho asociados. Estas unidades ocultas a veces se denominan centroides o núcleos. Cada unidad de salida realiza una suma ponderada de las unidades ocultas, utilizando $w_j s$ como pesos.

De lo anterior, se puede ver que diseñar un RBFN implica seleccionar el tipo de funciones base, con anchos asociados, $\sigma$, el número de funciones, M , las ubicaciones centrales $\mu\ j$ y los pesos $W_j$ . Normalmente se utilizan funciones gaussianas u otras funciones en forma de campana con soporte compacto. La elección de M está relacionada con la complejidad del mapa deseado.(Goodfellow, 2016) Dado que se han seleccionado el número de funciones base y su tipo, entrenar una RBFN implica determinar los valores de tres conjuntos de parámetros: los centros, los anchos y los pesos, con el fin de minimizar una función de coste adecuada. En general, este es un problema de optimización no convexo.

Se puede realizar un descenso de gradiente estocástico en una función de costo de error cuadrático medio para actualizar iterativamente los tres conjuntos de parámetros, una vez por presentación de muestra de entrenamiento.(Goodfellow, 2016)

Esto puede ser adecuado para entornos no estacionarios o configuraciones en línea. Pero para los mapas estáticos, los RBFN con funciones de base localizadas ofrecen una alternativa muy atractiva, a saber, que en la práctica, la estimación de parámetros se puede desacoplar en un procedimiento de dos etapas, determinando $\mu_j S_y \sigma_j S$, y para los centros $y$ anchos obtenidos se determina los pesos a las unidades de salida.

Ambos subproblemas permiten soluciones en modo por lotes muy eficientes. En la primera etapa, solo los valores de entrada $\{x(n)\}se$ utilizan para determinar los centros $\mu_j y$ los anchos $\sigma_j$ de las funciones base. Por lo tanto, el aprendizaje no está supervisado e incluso puede utilizar datos sin etiquetar. Una vez que se fijan los parámetros de la función base, se puede emplear el entrenamiento supervisado (es decir, el entrenamiento usando información de

destino) para determinar los pesos de la segunda capa. Ahora describimos brevemente una tercera red, una que tiene una topología plana. Cada celda está conectada a todas las celdas, incluida ella misma, y también es capaz de recibir señales de entrada directas. Una de las redes çompletamente recurrentes"más simples es el modelo binario de Hopfield. (Hua, 2019)

En una red de n celdas, la k ésima celda recibe una entrada constante $\mathrm{t_k}$, y actualiza su estado usando:

$$x_k(t+1) = \operatorname{sgn}\left(\sum_{j=1}^{n} w_{kj} x_j(t) + I_k\right)$$

donde "sgn"denota la función signum, igual a 1 si $x$ no es negativo, $y$ $-1$ en caso contrario. La matriz de pesos debe ser simétrica y fija, es decir, no hay adaptación de pesos.

Hay que determinar $\mu_j S_y \sigma_j S, y$ y para los centros $y$ anchos obtenidos en este paso se determina los pesos a las unidades de salida.

Ambos subproblemas permiten soluciones en modo por lotes muy eficientes. En la primera etapa, solo los valores de entrada $\{x(n)\}se$ utilizan para determinar los centros $\mu_j y$ los anchos $\sigma_j$ de las funciones base. Por lo tanto, el aprendizaje no está supervisado e incluso puede utilizar datos sin etiquetar. Una vez que se fijan los parámetros de la función base, se puede emplear el entrenamiento supervisado (es decir, el entrenamiento usando información de destino) para determinar los pesos de la segunda capa. (Hua, 2019)

Partiendo de un estado inicial, dado por los valores de todas las celdas en $t = 0$, las celdas actualizan las celdas actualizan su estado de forma asincrónica hasta que se alcanza un estado final donde los lados izquierdo y derecho son iguales para cada una de las celdas. Se puede demostrar que ese estado final está garantizado mediante la construcción de una función de energía o costo:

$$E = -\frac{1}{2} \sum_{i} \sum_{j} w_{l_j} x_l x_j - \sum_{l} I_l x_l$$

Al mostrar que $E$ está acotado desde abajo $y$, además, que $E$ se reduce en una cantidad finita cada vez que una celda cambia su valor en una actualización, se concluye que las actualizaciones deben terminar en un tiempo finito.

# Capítulo 4: Retropropagación o backpropagation 5

El algoritmo de retropropagación se usa comúnmente para entrenar redes neuronales artificiales.

El entrenamiento generalmente se realiza mediante la actualización iterativa de ponderaciones, generalmente empleando el gradiente negativo de una función de error cuadrático medio. (Hua, 2019)

La señal de error es el producto de la diferencia entre los valores de salida deseados y reales y la pendiente de una función de activación sigmoidea. Esta señal de error luego se propaga hacia atrás a las capas inferiores.(Goodfellow, 2016)

Se utilizan dos parámetros, llamados tasa de aprendizaje y factor de momento, para controlar el ajuste de peso a lo largo de la dirección de descenso más pronunciada y para amortiguar las oscilaciones. (Hua, 2019)

El algoritmo de la retropropagación o backpropagation para redes multicapa es un procedimiento de descenso de gradiente que se utiliza para minimizar una función objetivo de mínimos cuadrados (función de error).

De un lote de pares de muestras de entrenamiento: $(I_1, I_1), \ldots, (I_n, I_n)$ donde $I_5, 1 \leqslant s \in n$, se han representar la s-ésima entrada en el lote, $yT_s, 1 \in s$ deseada correspondiente (objetivo). Para neuronas de capas ocultas arbitrarias, la función objetivo de mínimos cuadrados en el espacio de peso de las redes es:

$$\mathrm{E} = 1\mathrm{nZ_M} \sum s = \ln \left[\mathrm{T_5} - \mathrm{O_5}^{\mathrm{M}}\right]^{\mathrm{T}} \left[\mathrm{T_s} - \mathrm{O_s^M}\right]$$

donde $O_s M_{\text{es}}$ *el* vector de salida de una red de capas $M$ con $I_s$ como entrada $yZ_M$ es el número de neuronas de salida.

Sea W un vector formado por todos los pesos de la red n $\nabla F(W(k))el$ gradiente de $\underline{F}$ en $W = W(k)$, siendo $k = 1, 2, 3, \ldots,$ Nel número de iteración del vector de peso. E. algoritmo de BP de dos términos con un término de impulso viene dado por:

$$\Delta \mathrm{W(k)} = \alpha(-\nabla \mathrm{E(W(k))}) + \beta \Delta \mathrm{W(k-1)}$$

donde $\alpha_Y \beta$ son LR $yMF_1$ respectivamente, $y\Delta W(k) = W(k+1) - W(k)$

Los cálculos de retroalimentación de la red con $I_s$ presentados a la capa de entrada están dados por:

$$0_{s,1}{}^{\mathrm{m}} = \mathrm{f}\left(\left[\mathrm{W_i^m(k+1)}\right]^{\mathrm{T}} \mathrm{O_s}^{\mathrm{m-1}}\right)$$

dónde $0_{5,i}^{\mathrm{m}}, 1 \leqslant \mathrm{i} \leqslant \mathrm{Z_m}$, denota la i-ésima salida de la capa m,

$$1 \leqslant \ \mathrm{m} \leqslant \mathrm{M}_i f(\cdot)$$

Es la función de activación $W_i{}^m(k+1)$ es un subvector de $W(k+1)$, que consta de todos los pesos de las neuronas de la capa $m-1$ ao $s, i^m; yO_{s,i}^{m-1}$ es un vector formado por todas las salidas de la capa m $-1$ (incluida una salida unitaria, que se usa como referencia para sesgar la siguiente capa) y está dada por:

$$\mathrm{O_{s,i}^{m-1}} = \left[1\mathrm{O_{s,1}^{m-1}} \ldots \mathrm{O}_{s,Z_n}\ \mathrm{m}-1\right]^{\mathrm{T}} \text{ para } > 1, [1\mathrm{I_s T}]^{\mathrm{T}} \text{ para m} = 1$$

El algoritmo de backpropagation se modifica agregando un término adicional para aumentar la velocidad de aprendizaje. Este término es proporcional $ae(W(k))$ que representa la diferencia entre la salida y el objetivo en cada iteración. (Hua, 2019) Para el aprendizaje por lotes:

$$\mathrm{e(W(k))} = [\mathrm{e}_s \mathrm{e}_s \ldots \mathrm{e_s}]^{\mathrm{T}}$$

donde el vector $e$ es de dimensión apropiada $y_s = \sum_{s=1}^{n} [I_s - O_s M]$ para el aprendizaje por lotes. [1]

Por tanto, el algoritmo modificado es:

$\Delta \mathrm{W(k)} = \alpha(-\nabla \mathrm{E(W(k))}) + \beta \Delta \mathrm{W(k-1)} + \gamma \mathrm{e(W(k))}$ donde $y$ es denotará la función.

Se observa que el algoritmo tiene tres términos, uno proporcional a la derivada de $F(W(k))$, otro proporcional al valor anterior del cambio incremental de los pesos y un tercer término proporcional a $e(W(k))$.

Como función de activación de la red neuronal profunda, la función ReLU tiene un rendimiento excelente y una estructura simple. (Goodfellow, 2016)

Esta función ayuda a la red neuronal profunda a realizar una activación escasa. En el proceso de uso, después de la inicialización, el peso puede hacer que aproximadamente la mitad de la salida de las unidades ocultas sea igual a 0.(Goodfellow, 2016)

La activación escasa no solo está más en línea con el mecanismo de la actividad del cerebro humano, sino que tiene una buena ventaja a nivel matemático.. Para una entrada dada, el subconjunto activado de neuronas es único y el cálculo del subconjunto es lineal, lo que hace que la función ReLU sea una función de activación sin la desaparición del gradiente. La aplicación de la función sigmoidea y la función tanh en redes neuronales profundas está limitada debido a la desaparición del gradiente. (Hua, 2019)

Las expresiones de la función ReLU y sus derivadas son:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

$$\mathrm{Re}\,LU'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Existen diferentes tipos de funciones de activación (Goodfellow, 2016) de las cuales las básicas son:

Función de escalón o umbral:

$$\varphi(v) = \begin{cases} 1 \text{ if } v \geqslant 0 \\ 0 \text{ if } v < 0 \end{cases}$$

Función lineal por partes:

$$\varphi(v) = \begin{cases} 1 \text{ if } v \geqslant 1 \\ v \text{ if } 0 < v < 1 \\ 0 \text{ if } v \leqslant 0 \end{cases}$$

Función sigmoidea - función logística:

$$\varphi(v) = \frac{1}{1 + e^{-\alpha t}}$$

Función sigmoidea - tangente hiperbólica:

$$\varphi(v) = \tanh(v)$$

# Capítulo 5: RNN y LSTM 6

Una red neuronal recurrente (RNN) es una clase de redes neuronales artificiales donde las conexiones entre nodos forman un gráfico dirigido a lo largo de una secuencia temporal. Esto le permite exhibir un comportamiento dinámico temporal.

Las RNN pueden usar su estado interno (memoria) para procesar secuencias de entradas de longitud variable. Esto los hace aplicables a tareas como el reconocimiento de escritura conectado no segmentado o el reconocimiento de voz.(Sezer, 2020)

Redes neuronales recurrentes se dividen en dos amplias clases de redes con una estructura general similar, donde una es impulso finito y la otra es impulso infinito. Aquí notaremos también que estas clases de redes exhiben un comportamiento dinámico temporal.(Kelleher, 2019)

Una red recurrente de impulso finito es un gráfico acíclico dirigido que se puede desenrollar y reemplazar con una red neuronal estrictamente feedforward, mientras que una red recurrente de impulso infinito es un gráfico cíclico dirigido que no se puede desenrollar.(Sezer, 2020)

Tanto las redes recurrentes de impulso finito como las de impulso infinito pueden tener estados almacenados adicionales, y el almacenamiento puede estar bajo el control directo de la red neuronal.

El almacenamiento también se puede reemplazar por otra red, si eso incorpora retrasos de tiempo o tiene bucles de retroalimentación. Estos estados controlados se denominan estado de compuerta o memoria de compuerta. Esto también se llama Red neuronal de retroalimentación. (Sezer, 2020)

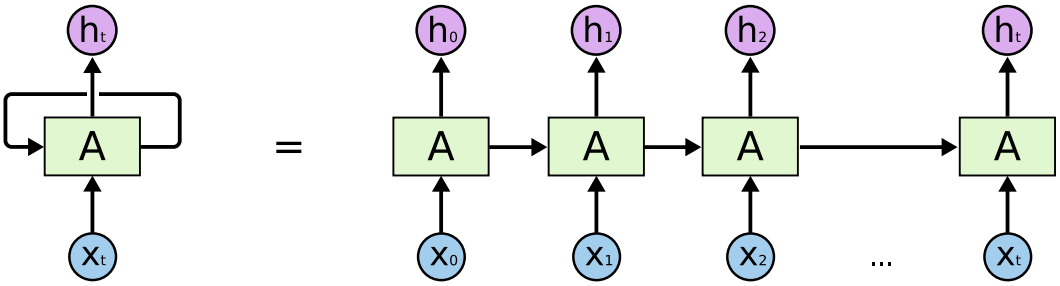


Figura 6.1: Ilustración extraída de (Gulli, 2017) como ejemplo de una RNN

Las redes neuronales completamente recurrentes conectan las salidas de todas las neuronas a las entradas de todas las neuronas. Esta es la topología de red neuronal más general porque todas las demás topologías se pueden representar estableciendo algunos pesos de conexión en cero para simular la falta de conexiones entre esas neuronas. (Goodfellow, 2016)

Las topologías prácticas de redes neuronales se organizan con frecuencia en capas, que al tener diferentes pasos en el tiempo de la misma red neuronal lo vuelven completamente recurrente.

La memoria a largo plazo y corto plazo o Long short-term memory (LSTM) en inglés es una arquitectura de red neuronal recurrente artificial utilizada en el campo del aprendizaje profundo de modelos secuenciales definidos por variables continuas, muy útil para la sincronización. (Hua, 2019)

A diferencia de la red neuronal de retroalimentación estándar, las LSTM tienen una conexión de retroalimentación. Pueden procesar no solo un único punto de datos (como una imagen), sino todo el flujo de datos (como audio, video o cualquier serie).(Kelleher, 2019)

Los LSTM se pueden aplicar a tareas como el reconocimiento de escritura a mano sin particiones vinculadas, el reconocimiento de voz, la detección de anomalías en el tráfico de la red o el sistema de detección de intrusiones, la clasificación de spam, el reconocimiento de géneros musicales y el análisis de sentimientos. Además, se han utilizado en el campo del procesamiento del lenguaje natural para el modelado de árboles de dependencia, expresiones de oraciones, modelado del lenguaje y respuesta a preguntas. (Sezer, 2020)

Una unidad LSTM típica consta de celdas, entradas, salidas y puertas de olvido. La celda almacena valores para períodos de tiempo arbitrarios y estas tres puertas regulan el flujo de información dentro y fuera de la celda.

LSTM es una extensión del modelo RNN clásico ampliamente utilizado en problemas de aprendizaje automático e inteligencia artificial. Los modelos de memoria más flexibles y precisos se consideran más dinámicos y han tenido

éxito en muchas aplicaciones comerciales y financieras. (Sezer, 2020)

La idea principal de LSTM es extender el modelo RNN tradicional con bloques de memoria adicionales o estados de celda.

El modelo aprende a mantener su estado de "memoria celular.[a] lo largo del tiempo, lo que permite que el modelo actualice su estado interno. Esto se puede hacer agregando una o más puertas nuevas que le permitan agregar o eliminar información en la "memoria de la celda.[en] cada paso de tiempo.(Hua, 2019)

Si bien los modelos basados en LSTM se consideran actualizados en la literatura de aprendizaje automático supervisado, los modelos de redes neuronales recurrentes (RNN) en general forman entre ellos una dependencia persistente de datos en el modelado.

Las LSTM se utilizan para hacer predicciones basadas en datos de series de tiempo porque puede haber uno o más retrasos desconocidos entre eventos de series de tiempo importantes.

Las LSTM fueron desarrolladas para resolver el problema de la "gradación explosiva" y el desvanecimiento que puede ocurrir durante el entrenamiento tradicional RNN, producto de una alta dependencia entre las neuronas, razón por lo que se plantea el olvida de parte de la información.(Hua, 2019)

Bien, para formalizar las LSTM se consideran dentro de redes neuronales especiales que alteran las memorias que utilizan, desechando información innecesaria para guardar información relevante.

Sean matrices $W_q$ y $U_q$ que contienen, respectivamente, los pesos de las conexiones de entrada y se tomará que $c_t \in \mathbb{R}^h$ será una celda que contiene $h$ celdas de unidad.(Hua, 2019)

Para el caso LSTM con una puerta para olvidar las ecuaciones para el pase directo de una unidad LSTM con una puerta de olvido los valores iniciales son $c_0 = 0$ y $h_0 = 0$. Teniendo las siguientes variables:

$x_t \in \mathbb{R}^d$ : vector de entrada a la unidad.

$L\ f_t \in \mathbb{R}^h$. que será la orden de olvidar el vector de activación de la puerta

$i_t \in \mathbb{R}^h$ : entrada/actualización del vector de activación de la puerta

$\mathbb{R}^h$ : vector de activación de la puerta de salida

$h_t \in \mathbb{R}^h$ : vector de estado también conocido como vector de salida de la unidad LSTM

$\tilde{c}_t \in \mathbb{R}^h$ : vector de entrada de celda

$\mathbb{R}^h$ : vector de estado de celda

$W \in \mathbb{R}^{h \times d}, U \in \mathbb{R}^{h \times h}$ y $b \in \mathbb{R}^h$ : matrices de peso $y$ parámetros de vector de sesgo que deben aprenderse durante el entrenamiento

donde los superíndices $d$ y $h$ se refieren al número de funciones de entrada y al número de unidades ocultas, respectivamente.

Donde se pueden denotar las siguientes funciones de activación a escoger:

$\sigma_g$ : función sigmoidea

$\sigma_c$ : función tangente hiperbólica.

$\sigma_h$. función de tangente hiperbólica o, LSTM de mirilla $\sigma_h(x) = x$

Ahora bien las conexiones de mirilla LSTM

Las conexiones de mirilla permiten que las puertas accedan al "carrusel" de error constante (CEC), cuya activación es el estado de la celda. $h_{t-1}$ usando lo siguiente:

$$f_t = \sigma_g \left( W_f x_t + U_f c_{t-1} + b_f \right)$$

$$i_t = \sigma_g \left( W_i x_t + U_i c_{t-1} + b_i \right)$$

$$o_t = \sigma_g \left( W_o x_t + U_o c_{i-1} + b_o \right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_e \left( W_c x_t + b_c \right)$$

$$h_t = o_t \circ \sigma_h \left( c_t \right)$$

Ahora bien también pueden tener una "mirilla convolucional" LSTM.

El * denota el operador de convolución. Por lo que se tendría:

$$f_t = \sigma_g \left( W_f * x_t + U_f * h_{t-1} + V_f \circ c_{t-1} + b_f \right)$$

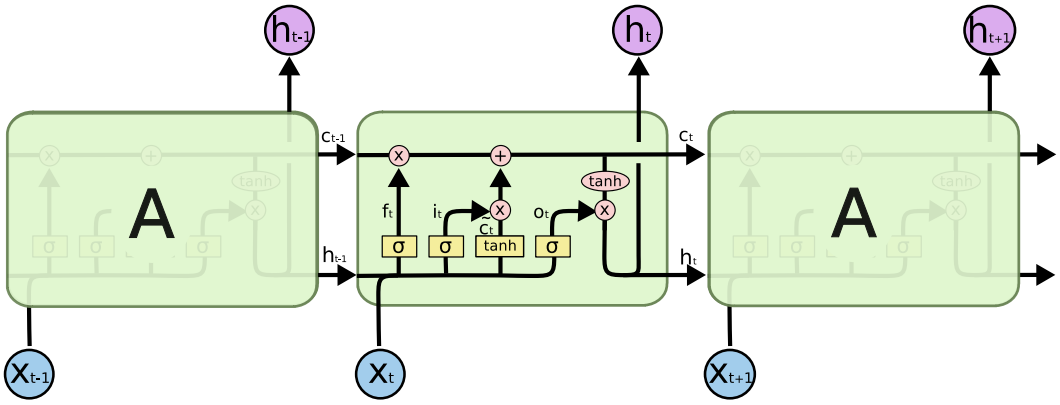


FIGURA 6.2: Ilustración extraída de (Gulli, 2017) como ejemplo de una celda LSTM

$$i_t = \sigma_g \left( W_i * x_t + U_i * h_{t-1} + V_i \circ c_{t-1} + b_i \right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c \left( W_c * x_t + U_c * h_{t-1} + b_c \right)$$

$$o_t = \sigma_g \left( W_o * x_t + U_o * h_{t-1} + V_o \circ c_t + b_o \right)$$

$$h_t = o_t \circ \sigma_h \left( c_t \right)$$

Ahora bien, resumiendo esto, en su forma simple puede ser presentada de la siguiente manera:

Los bloques de la célula de la red neuronal LSTM. (Hua, 2019) En el momento t, la unidad de memoria interna registra toda la información histórica hasta el momento actual y está controlada por tres "puertas":

1. Puerta de entrada: puerta de entrada $g_k^{(t)}$ (para paso de tiempo $t_y$ celular $i_1$ de manera similar en otros símbolos) controla la nueva información introducida en la unidad de memoria interna, que se puede denotar como:

$$g_i^{(t)} = \sigma \left( b_i^q + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i.j}^g h_j^{(t-1)} \right)$$

donde la función de activación $\sigma$ es la función sigmoidea, $x^{(t)}$ es el vector de entrada actual $h^{(t)}$ es el vector de capa oculta actual, que contiene la salida

de todas las celdas LSTM $y^g, U^g$ y $W^g$ indican el sesgo, los pesos de entrada y los pesos recurrentes para la puerta de entrada, respectivamente.

2. Puerta de olvido: Donde se olvida de la unidad de puerta $f_i^{(t)}$ controla la unidad de memoria interna sobre cuánta información en los momentos anteriores necesita ser guardada.(Goodfellow, 2016)

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

donde la función de activación $\sigma$ es la función sigmoidea con un valor entre 0 y 1 $b^f, U^f, W^f$ son el sesgo, las ponderaciones de entrada y las ponderaciones recurrentes para la puerta de olvido, respectivamente.

Cuando $f_i^{(t)} = 1$; se abre la puerta de olvido y el estado de la celda en el momento anterior se introduce en la celda.

De lo contrario, cuando $f_i^{(t)} = 0$; se cierra la puerta de olvido y se descarta el estado de la celda en el momento anterior.

La unidad de estado interno $S_i^{(t)}$ de la celda LST'M que tiene un peso de bucle propio condicional $f_i^{(t)}$ se actualiza, que se formula como:

$$S_i^{(t)} = f_i^{(t)} S_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

dónde $b_i, U_i, W_i$ denotan sesgo, ponderaciones de entrada y ponderaciones recurrentes en la celda LST'M, respectivamente.

Denotando que la primera parte es la información del estado de la celda controlada por la puerta de olvido en el último momento, $y$ la última parte es la información de entrada controlada por la puerta de entrada.(Goodfellow, 2016)

3. Puerta de salida: La puerta de salida $O_i^{(t)}$ controla la unidad de memoria interna, es decir produce y reproduce la información requerida, que puede ser dada por:

$$h_i^{(t)} = \tanh\left(S_i^{(t)}\right) O_i^{(t)}$$

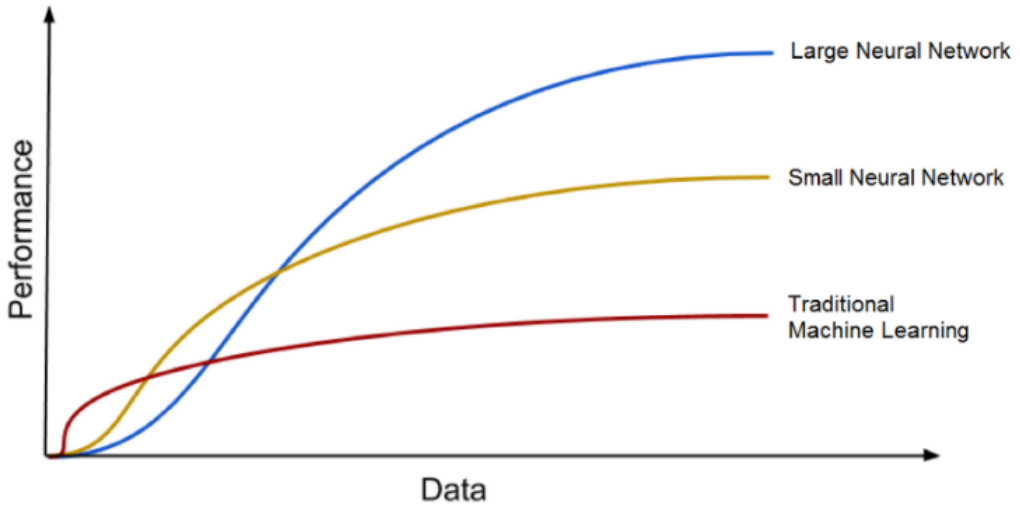


FIGURA 6.3: Ilustración extraída de (Gulli, 2017) como ejemplo del rendimiento de modelos

$$O_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(i)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

donde $b^\circ, U^o, W^o$ denotan sesgo, pesos de entrada y pesos recurrentes parala puerta salida, respectivamente.

La salida de la unidad LSTM actúa como un estado de capa oculta en el RNN. Por el momento, no existe una regla general para especificar el número de capas ocultas y retrasos lo que juega un papel vital que afecta el rendimiento general del entrenamiento y las pruebas en la arquitectura de la red LSTM. Sin embargo se puede probar varias arquitecturas con un set de datos seleccionado para buscar mejores arquitecturas.(Goodfellow, 2016)

Los modelos tradicionales de machine learning y de series temporales tienen mayor perfomance resolviendo problemas con pocos datos, mientras que las redes neuronales, tienen mayor performance en contextos de muchos datos, mientras más grande sea la arquitectura de la red neuronal también obtiene mayor performance en cuanto se agregan más datos. (Goodfellow, 2016)

# Capítulo 6: Tensorflow y Keras 7

El aprendizaje automático con Tensorflow y Keras, específicamente para aprendizaje profundo ha convertido al Deep Learning en una actividad mucho más sencilla.

Debido a que el aprendizaje profundo se ha convertido en uno de los temas más candentes de la actualidad, con éxitos históricos como nuevos algoritmos, hardware de procesamiento masivo paralelo y precisión superior a la humana en el reconocimiento de objetos de imagen.(Gamboa, 2017; Busseti, 2012; Shen, 2020; Torres, 2018)

Los investigadores de Google han estado a la vanguardia de la informática en los últimos años al lanzar el marco de aprendizaje profundo desarrollado internamente que es TensorFlow, que además es un software de código abierto.

Google ha proporcionado una plataforma estable para aplicaciones e investigación de aprendizaje profundo. Las capacidades de investigación de TensorFlow son fascinantes y brindan al menos a los científicos y profesionales un sistema más estandarizado que tiene el potencial de mejorar la usabilidad y reproducibilidad de las técnicas de aprendizaje profundo.(Kelleher, 2019)

Un modelo de aprendizaje profundo (generalmente una red neuronal multicapa) consta de varias capas computacionales que procesan los datos de forma jerárquica. Cada nivel toma entradas y produce salidas. Generalmente se calcula como una función no lineal de una combinación lineal ponderada de valores de entrada. (Kelleher, 2019)

La salida de una capa se convierte en la entrada de la siguiente capa de procesamiento, creando una arquitectura profunda. En cada nivel sucesivo, los datos se representan de una manera cada vez más abstracta. (Gamboa, 2017;

Shen, 2020)

Estos modelos intensivos pueden aprender abstracciones de datos muy complejas para capturar las estructuras complejas de conjuntos de datos muy grandes.

Implementar un modelo de red neuronal profunda complejo y exitoso no es una tarea fácil y, a menudo, se limita a los profesionales del aprendizaje profundo o cualquier persona que se interese en el Deep Learning. Afortunadamente, gracias a la estructura modular de la red y las herramientas de inferencia estándar, hay varios marcos de software disponibles para acelerar el diseño y entrenamiento de redes neuronales profundas. (Torres, 2018)

TensorFlow es la última herramienta agregada a este kit de herramientas. Ofrece una variedad de mejoras, incluida la reproducción de gráficos mejorada y el tiempo de compilación.

Los marcos de aprendizaje profundo más utilizados en la actualidad son Torch, Theano y Caffe, que son especialmente adecuados para redes neuronales complejas.

Muchas startups y grupos de investigadores han publicado herramientas de aprendizaje profundo como Neon (un marco basado en Python), Deeplearning J (motor Java de Skymind) y H2O3 (Skymind en aprendizaje automático basado en Java). Un conjunto de herramientas desarrollado por H2O con interfaces como Python, R y Scala. (Shukla, 2018)

TensorFlow, como Theano, usa un modelo de programación declarativo. Esto permite a los investigadores centrarse en la definición simbólica de lo que se necesita calcular o modelar, en lugar del método exacto o el orden específico en el que se realizan estos cálculos, como en la programación imperativa.(Gulli, 2017)

El modelo se puede optimizar para la estabilidad y el rendimiento digitales, y las partes individuales se pueden convertir para ser ejecutadas por la CPU, GPU o bien las TPU. (Shukla, 2018)

Quizás lo más importante es que la representación simbólica permite la discriminación automática. Esta es una forma conveniente de optimizar múltiples funciones. Estas pueden ser redes neuronales u otras funciones comúnmente utilizadas para representar matemáticamente diferentes problemas basados en datos.

El marco de procesamiento administra automáticamente la representación abstracta del modelo. Esto hace que TensorFlow y Theano sean particularmente adecuados para desarrollar nuevos modelos mediante la optimización basada

en gradientes.

No solo las arquitecturas de redes neuronales profundas, sino que también entran en esta categoría otros tipos de modelos los que se pueden modelar en Tensorflow.

Y Tensorflow puede tener ventajas frente a otros marcos de aprendizaje profundo, por ejempla la mayor desventaja de Theano es que lleva mucho tiempo compilar el modelo y es más lento por ejemplo en la tokenización. TensorFlow mejora drásticamente este cuello de botella. (Gulli, 2017)

Keras es una interfaz de programa de aplicación de red neuronal de código abierto de alto nivel escrita en Python.

Keras está desarrollado originalmente por François Chollet, ahora ingeniero de Google. (Hua, 2019)

Keras se puede ejecutar en varias bibliotecas de aprendizaje automático como TensorFlow (desarrollada por Google), CNTK (desarrollada por Microsoft) y Theano (desarrollada por la Universidad de Montreal).

Keras tiene como objetivo proporcionar a los usuarios una creación de prototipos rápida y sencilla mediante la facilidad de uso, la modularidad y la extensibilidad. (Gulli, 2017)

Por ejemplo en 2018, Keras ocupó el segundo lugar después de TensorFlow en evaluaciones de eficacia basadas en el uso, el interés y la popularidad. Con Keras se puede configurar fácilmente una red neuronal con la estructura deseada. Todas las funciones estándar de pérdida también están disponibles. La propagación hacia atrás es automática internamente. (Hua, 2019)

Keras también proporciona todos los algoritmos de optimización estándar, incluidos SGD, RMSProp y Adam.(Gulli, 2017)

En el cuadro 7.1 en la página 36 se hace referencia a las funciones que se usa en Keras para modelar redes neuronales, crear, entrenar, evaluar, persistir y generar predicciones.

Entre las funciones está $keras_model()quereunemodelosdiversosdeMachineLearning, entreello$

fit() será usado para iniciar el entrenamiento, evaluate() evaluará en términos de error el modelo de redes neuronales.

Cuadro 7.1: Funciones de Keras

| | |
|---|---|
| keras_model() | Modelo Keras |
| keras_model_sequential() | Modelo de Keras compuesto por una pila lineal de capas |
| compile() | Configurar un modelo de Keras para entrenamiento |
| fit() | Entrena un modelo de Keras |
| evaluate() | Evaluar un modelo de Keras |
| predict() | Método de predicción para modelos de Keras |
| summary() | Imprime un resumen de un modelo |
| save_model_hdf5() | Guardar / cargar modelos usando archivos HDF5 |
| get_layer() | Recupera una capa según su nombre (único) o índice. |
| pop_layer() | Eliminar la última capa de un modelo |
| save_model_weights_hdf5() | Guarde / cargue pesos de modelo |
| load_model_weights_hdf5() | usando archivos HDF5 |
| get_weights() set_weights() | Ponderaciones de capa / modelo como matrices R |
| get_config() from_config() | Configuración de capa / modelo |
| model_to_json() | Configuración del modelo |
| model_from_json() | como JSON |
| model_to_yaml() | Configuración del modelo |
| model_from_yaml() | como YAML |

# Capítulo 7: Aplicación de redes neuronales 8

Para estos ejemplos de aplicación se usarán algunos paquetes disponibles para R, que son forecast, Keras y Tensorflow, aunque también es recomendable utilizar H2O por sus nuevas incorporaciones.

En un entorno donde se tiene instalado Python se puede trabajar con Tensorflow y Keras en R, así por ejemplo se dispondrá de un cuaderno de Google Collaboratory para mostrar cómo trabajar con R, de forma que cualquier lector pueda seguir lo que se menciona en este capítulo tanto si lo quiere hacer en su propio entorno como si lo desea hacer en el entorno del cuaderno dispuesto.

El siguiente es el enlace al cuaderno:

https://n9.cl/pd0jt

Desde este enlace el lector podrá ingresar al cuaderno, en cada sesión deberá secuencialmente correr cada celda para instalar paquetes.

Los pasos según la metodología convencional para elaborar una red neuronal (Kelleher, 2019) son:

• Decida la duración de su serie de tiempo;

• La duración de una ventana de tiempo del modelo; • Extraiga todos sus datos en un conjunto de datos de series de tiempo;

• Decida cómo desea preprocesar sus datos (si existe tal preprocesamiento) para obtener un conjunto de datos para su modelo y

• Desarrolle un modelo que sea capaz de analizar sus datos de series de tiempo. Considerar la longitud de los datos y longitud de la ventana que se

usará, es probable que un modelo tradicional sea más que una red neuronal en contextos de pocos datos.

• Definir la función de activación de cada nodo. Esta función asigna un valor real (por ejemplo, 1 para un estado activado y 0 en caso contrario) a un nodo en cada paso de tiempo.

• Con el modelo se debe predecir el valor del estado.

• El modelo puede ampliarse mediante un bucle automático para predecir el futuro de un estado determinado

Para instalar los paquetes Keras y Tensorflow se puede utilizar repositorios de rstudio como sigue:

```
devtools::install_github("rstudio/tensorflow")
devtools::install_github("rstudio/keras")
install.packages('reticulate')
```

A su vez se instaló reticulate por el entorno en relación a Python.

Para confirmar que Tensorflow se ha instalado correctamente se puede utilizar el siguiente código:

```
library(tensorflow)
tf$constant("Hellow Tensorflow")
```

La salida correcta es: "tf.Tensor(b'Hellow Tensorflow', shape=(), dtype=string)"

Se necesitan además otros paquetes para trabajar, entre ellos están quantmod, forecast y tidyverse.

El primer ejemplo que se puede mostrar es la comparación entre modelos ARIMA y una red neuronal simple.

A partir de quantmod se pueden extraer datos financieros que se descargan de Yahoo Finance, por ejemplo, con el siguiente código:

```
#Obtener datos de:
#S&P 500 (EE. UU.)(GSPC)
```

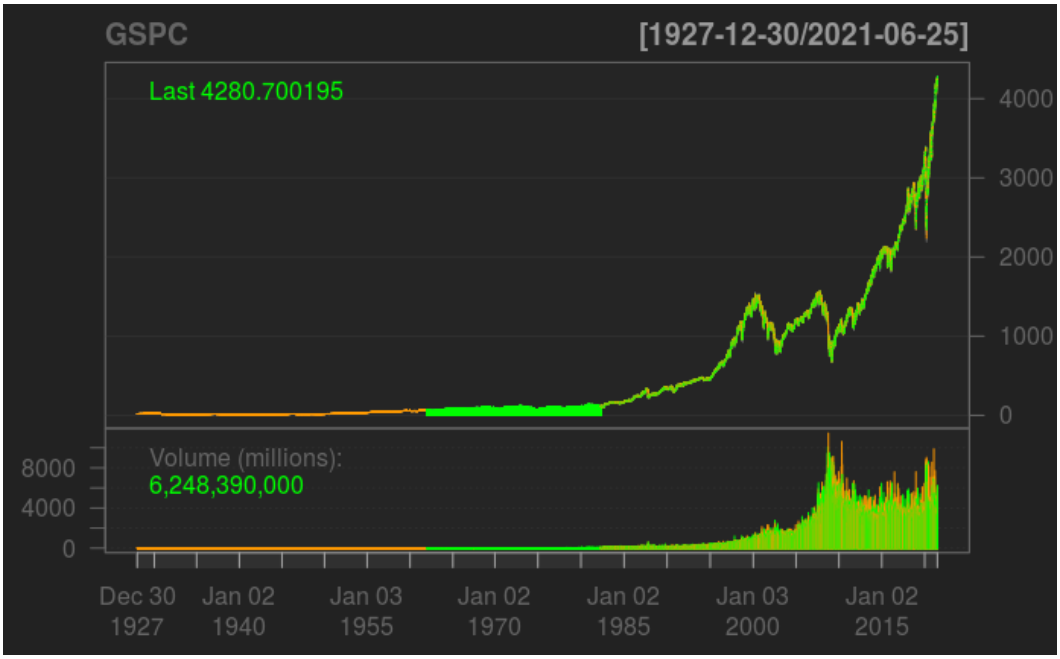


FIGURA 8.1: Datos de volumen y precio de SP500 descargados a partir de quantmod

```
#Dow Jones (EE. UU.)(DJI)
#NASDAQ (EE. UU.)(IXIC)
#Nikkei 225 (Japón)(N225)
#Índice de componentes de Shenzen (China)(399001.SZ)
#Índice S&P/NZX 50 bruto (Nueva Zelanda)(NZ50)
#Índice ponderado TSEC (Taiwán)(TWII)

tickers <- c("^GSPC", "^DJI" ,"^IXIC" ,"^N225" ,"399001.SZ", "^NZ50",
             "^TWII")
getSymbols(tickers, src = "yahoo", from = "1927-01-01",
           periodicity= "daily")

chartSeries(GSPC)
chartSeries(DJI)
chartSeries(IXIC)
```

Con chartSeries se graficas las siguientes series de tiempo:

Para SP500:

Con lo cual se puede evidenciar los datos históricos de SP500, pero se escoge en todas las series solo los valores ajustados.

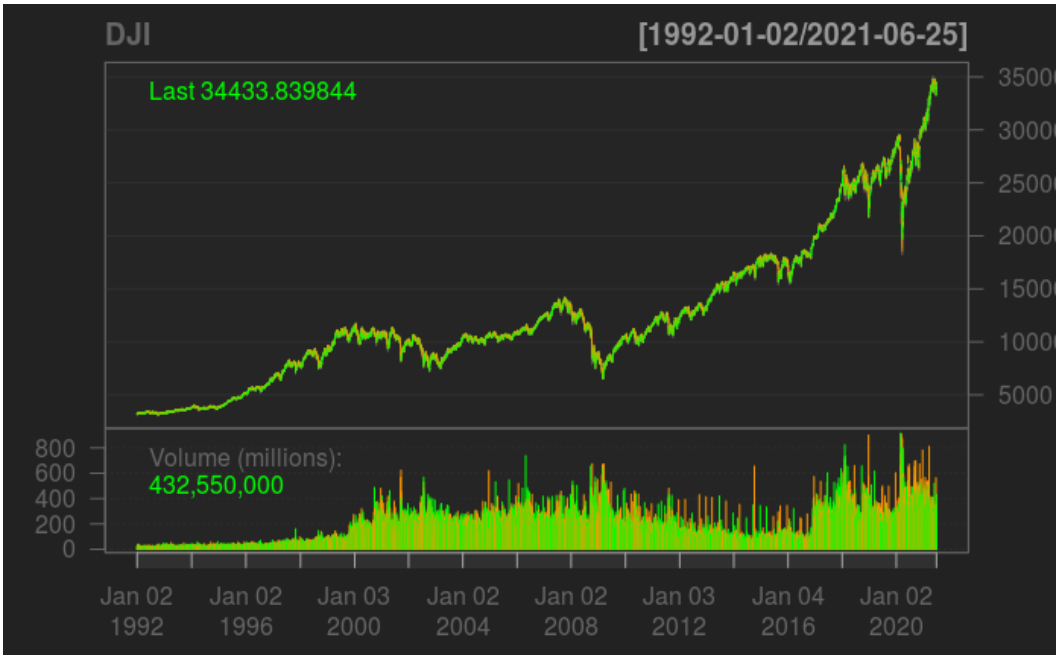


Figura 8.2: Datos de volumen y precio de Dow Jones descargados a partir de quantmod

Se grafica además Dow Jones.

Para observar sus tendencias, desde todos los datos del repositorio de Yahoo Finance.

Por último se puede observar datos de NASDAQ una bolsa tecnológica.

El paquete forecast tiene nnetar que es una función con una red neuronal que es fácilmente comparable a los modelos ARIMA, pero que atraparían las relaciones no lineales de las series temporales.

```
ADJSP500=GSPC$GSPC.Adjusted
ADJDJI=DJI$DJI.Adjusted
ADJSIXIC=IXIC$IXIC.Adjusted
modeloIXIC<-auto.arima(ADJSIXIC)
modeloIXICN<-nnetar(ADJSIXIC)
modeloDJI<-auto.arima(ADJDJI)
modeloDJIN<-nnetar(ADJDJI)
modeloSP500<-auto.arima(ADJSP500)
modeloSP500N<-nnetar(ADJSP500)
```

Una vez creados todos los modelos se pueden hacer predicciones.

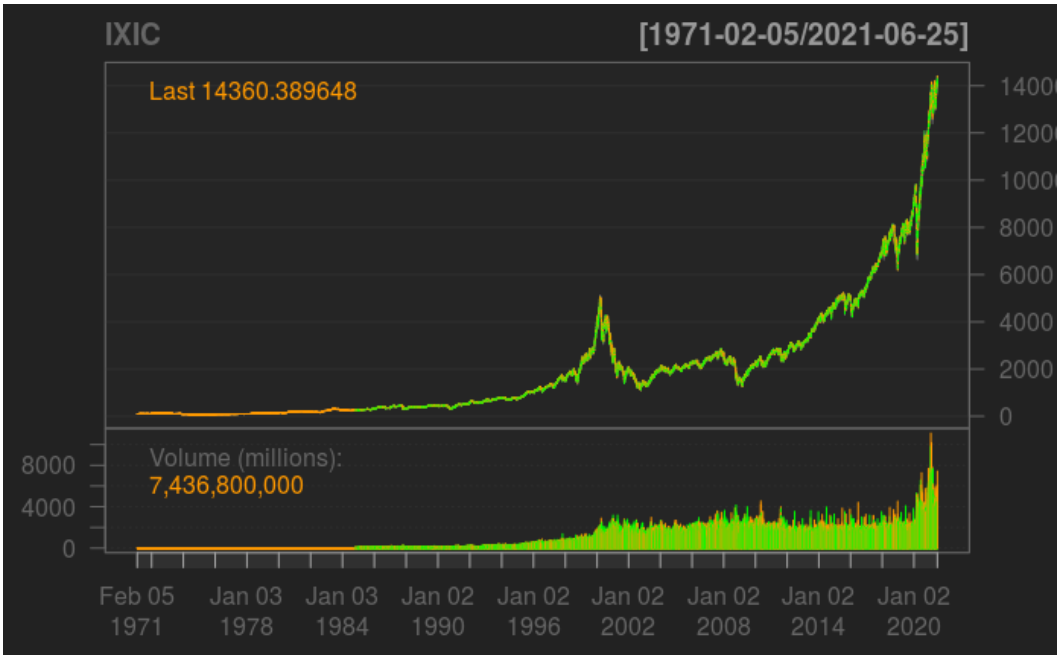


FIGURA 8.3: Datos de volumen y precio de NASDAQ descargados a partir de quantmod

Por ejemplo por rangos de tiempo de 100 días

```
plot(forecast(modeloIXIC, h=100))
plot(forecast(modeloIXICN, h=100))
plot(forecast(modeloDJI, h=100))
plot(forecast(modeloDJIN, h=100))
plot(forecast(modeloSP500, h=100))
plot(forecast(modeloSP500N, h=100))
```

Aquí mostramos algunas gráficas de estos modelos.

Por ejemplo las correspondientes a NASDAQ en cuanto a su modelo ARIMA y el modelo de la red neuronal simple.

Este corresponde al modelo ARIMA, se observa una tendencia positiva para los próximos 100 días.

Este corresponde al modelo NNAR, se observa una tendencia constante para los próximos 100 días.

Ambos modelos difieren, por lo que en diferentes contextos, unos son más pesimistas que otros, sin embargo NNAR reconoce algunos patrones no lineales, por lo que puede tener una ventaja.

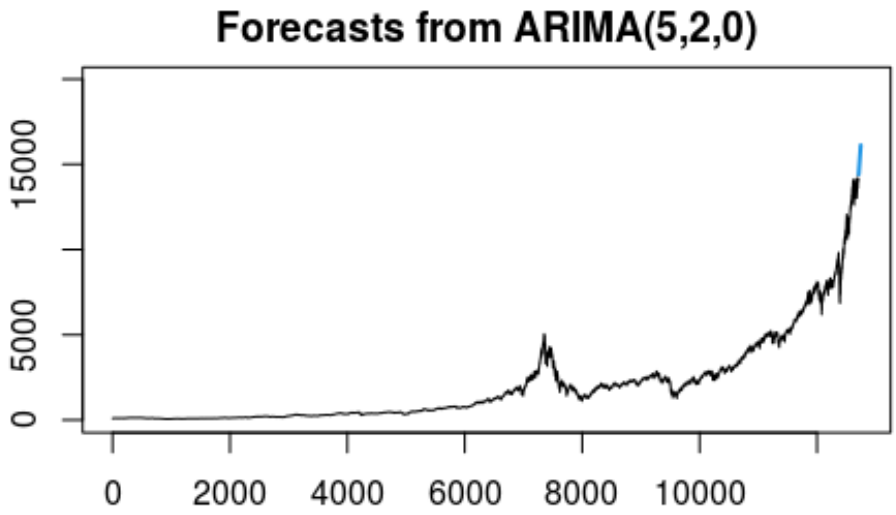


Figura 8.4: Predicción a 100 días con ARIMA de NASDAQ

Otro ejemplo que se puede hacer es con las RNN y LSTM

Para el ejemplo usando Keras, se plantea los mismos datos creados artificialmente:

```
library(keras)
N = 400
step = 2
set.seed(123)
n = seq(1:N)
a = n/10+4*sin(n/10)+sample(-1:6,N,replace=T)+rnorm(N)
a = c(a,replicate(step,tail(a,1)))
```

De este modo tendremos datos que usar para las redes neuronales.

```
x = NULL
y = NULL
for(i in 1:N)
{
 s = i-1+step
 x = rbind(x,a[i:s])
 y = rbind(y,a[s+1])
```

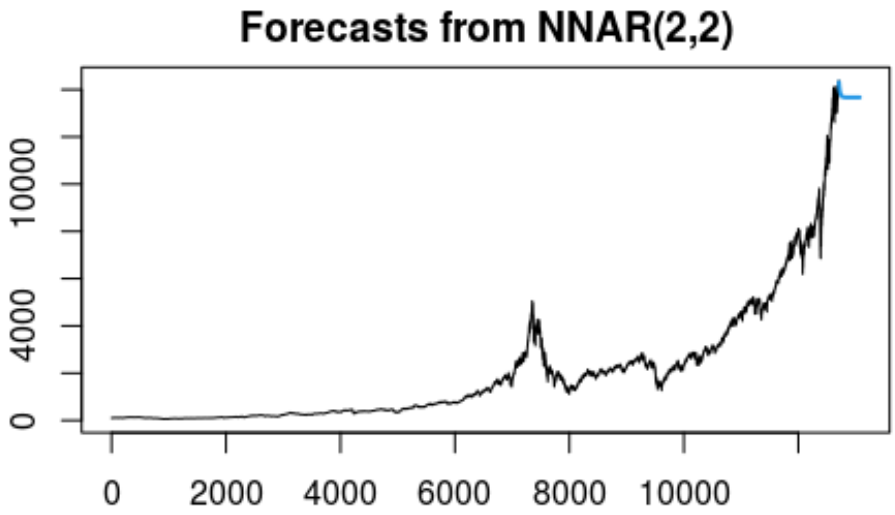


FIGURA 8.5: Predicción a 100 días con NNAR de NASDAQ

```
}
X = array(x, dim=c(N,step,1))
```

Lo anterior es una pequeña estructuración de los datos, que es útil para que la red neuronal pueda procesarlos, serán entonces necesarios entradas como salidas.

Primero se crea un modelo RNN simple, con sus neuronas correspondientes, utilizando en este caso la función de activición ReLU.

La gran ventaja que tiene Keras es que crear modelos es muy sencillo, como se observa a continuación:

```
#Creación del modelo
model = keras_model_sequential() %>%
  layer_simple_rnn(units=128, input_shape=c(step, 1), activation="relu") %>%
  layer_dense(units=64, activation = "relu") %>%
  layer_dense(units=32) %>%
  layer_dense(units=1, activation = "linear")

#Compilación del modelo usando mse y el optimizador de adam
model %>% compile(loss = 'mse',
                  optimizer = 'adam',
```

Cuadro 8.1: Resumen del modelo de RNN

Model: "sequential"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| simple_rnn (SimpleRNN) | (None, 128) | 16640 |
| dense_2 (Dense) | (None, 64) | 8256 |
| dense_1 (Dense) | (None, 32) | 2080 |
| dense (Dense) | (None, 1) | 33 |

Total params: 27,009
Trainable params: 27,009
Non-trainable params: 0

| loss | mean_absolute_error |
|---|---|
| 9.478787 | 2.536582 |

```
                  metrics = list("mean_absolute_error")
                   )
#Resumen del modelo
model %>% summary()

model %>% fit(X,y, epochs=50, batch_size=32, shuffle = FALSE, verbose=0)
y_pred  =  model %>% predict(X)
#Evaluación
scores  =  model %>% evaluate(X, y, verbose = 0)
print(scores)

#Graficando
x_axes = seq(1:length(y_pred))
plot(x_axes, y, type="l", col="red", lwd=2)
lines(x_axes, y_pred, col="blue",lwd=2)
legend("topleft", legend=c("y-original", "y-predicted"),
        col=c("red", "blue"), lty=1,cex=0.8)
```

Tiene un error medio absoluto de 2.53

Por último crearemos un modelo LSTM que solo es necesario modificar un poco al modelo con la función layerlstm y quedaría de la siguiente manera:

Cuadro 8.2: Resumen del modelo de LSTM

Model: "sequential_3"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| lstm (LSTM) | (None, 128) | 66560 |
| dense_11 (Dense) | (None, 64) | 8256 |
| dense_10 (Dense) | (None, 32) | 2080 |
| dense_9 (Dense) | (None, 1) | 33 |

Total params: 76,929
Trainable params: 76,929
Non-trainable params: 0

| loss | mean_absolute_error |
|---|---|
| 9.426416 | 2.550603 |

Tiene un error medio absoluto de 2.550785

Esto muestra la alta adecuación entre los modelos de redes neuronales y los datos.

Observando los cuadros de resumen por ejemplo se uso el mismo número de neuronas ocultas tanto en RNN y LSTM, con 3 capas ocultas para ambos modelos, evidentemente por la arquitectura de las LSTM se tendrán mayor número de parámetros en estas redes neuronales.

En este caso al ser $2.53 < 2.55$ con esta arquitectura las redes neuronales LSTM tienen mayor error medio absoluto que las RNN.

Observando tanto el gráfico de las RNN como de las LSTM se observa que las redes neuronales buscan adecuarse de forma ajustada a los datos.

Sin embargo siempre los investigadores deberán evitar los problemas del sobreajuste, estas redes neuronales son especialmente susceptibles a ello, por lo que siempre se debe testear una buena generalización.

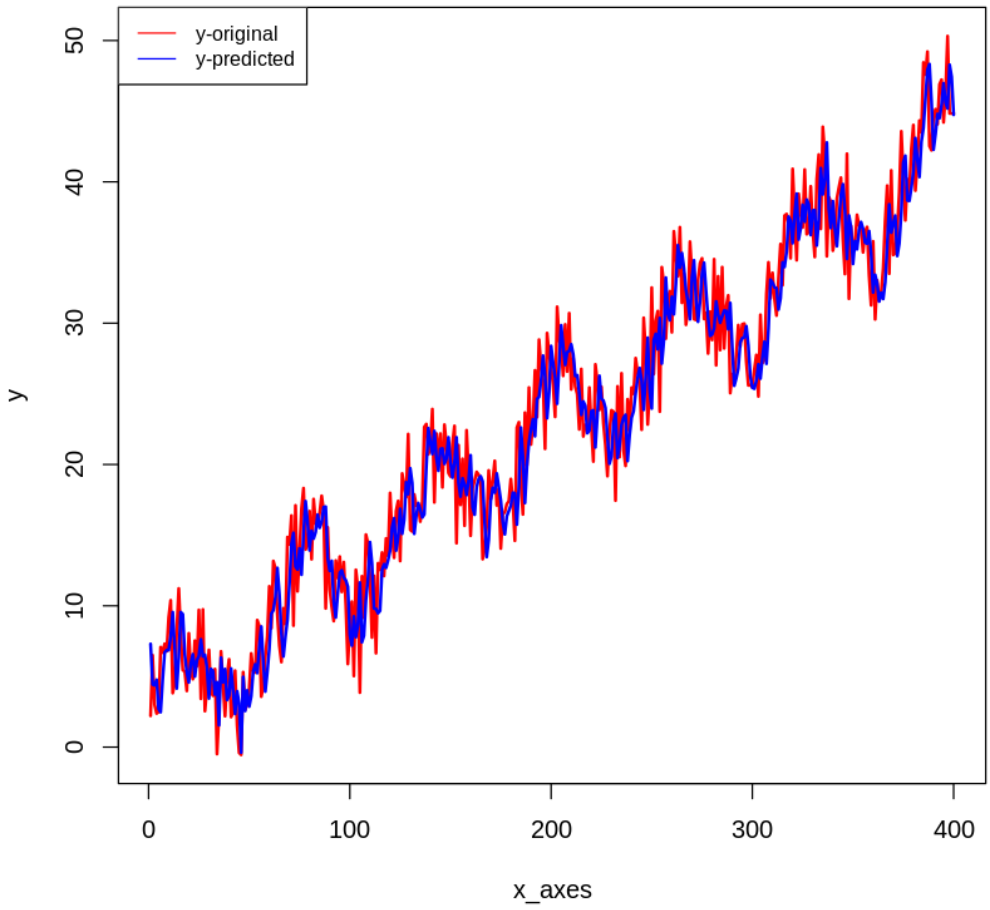


Figura 8.6: Predicción de RNN, datos originales versus predichos

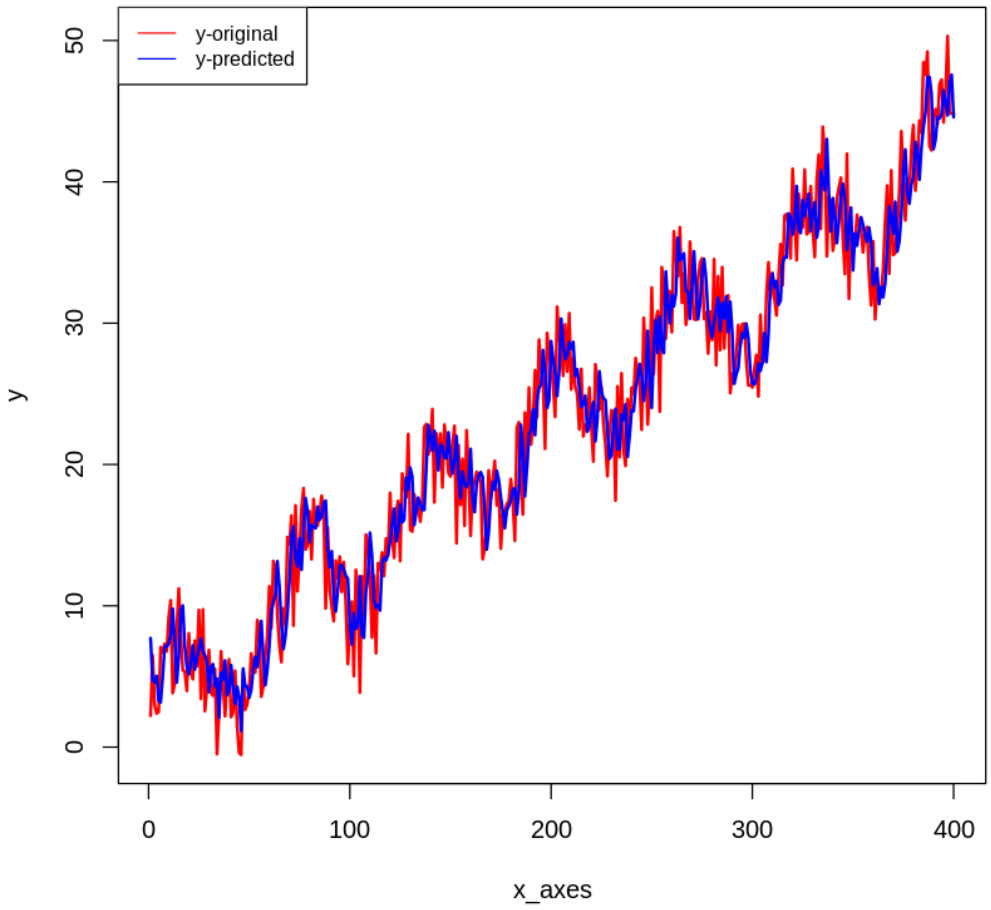


FIGURA 8.7: Predicción de LSTM, datos originales versus predichos

# Anexo: Código utilizado 9

```
#Instalción Keras y Tensorflow

    devtools::install_github("rstudio/tensorflow")
    devtools::install_github("rstudio/keras")
    install.packages('reticulate')
```

Librerías útiles y utilizadas

```
    # Core Tidyverse
    library(tidyverse)
    library(glue)
    library(forcats)
    # Series temporales
    library(timetk)
    library(tidyquant)
    library(tibbletime)
    # Visualización
    library(cowplot)
    # Preprocesado
    library(recipes)
    # Muestreo
    library(rsample)
    library(yardstick)
    # Modelado
    library(keras)
    library(quantmod)
```

```
library(forecast)
library(neuralnet)
```

DATOS para modelar

```
#Obtener datos de:
#S&P 500 (EE. UU.)(GSPC)
#Dow Jones (EE. UU.)(DJI)
#NASDAQ (EE. UU.)(IXIC)
#Nikkei 225 (Japón)(N225)
#Índice de componentes de Shenzen (China)(399001.SZ)
#Índice S&P/NZX 50 bruto (Nueva Zelanda)(NZ50)
#Índice ponderado TSEC (Taiwán)(TWII)

tickers <- c("^GSPC", "^DJI" ,"^IXIC" ,"^N225" ,"399001.SZ", "^NZ50",
             "^TWII")
getSymbols(tickers, src = "yahoo", from = "1927-01-01",
           periodicity= "daily")

chartSeries(GSPC)
chartSeries(DJI)
chartSeries(IXIC)
```

Modelos ARIMA y NNAR

```
ADJSP500=GSPC$GSPC.Adjusted
ADJDJI=DJI$DJI.Adjusted
ADJSIXIC=IXIC$IXIC.Adjusted
modeloIXIC<-auto.arima(ADJSIXIC)
modeloIXICN<-nnetar(ADJSIXIC)
modeloDJI<-auto.arima(ADJDJI)
modeloDJIN<-nnetar(ADJDJI)
modeloSP500<-auto.arima(ADJSP500)
modeloSP500N<-nnetar(ADJSP500)
```

Ejemplo RNN

```
library(keras)
N = 400
step = 2
set.seed(123)
```

```
    n = seq(1:N)
    a = n/10+4*sin(n/10)+sample(-1:6,N,replace=T)+rnorm(N)
    a = c(a,replicate(step,tail(a,1)))
```

DATOS y manipulación de datos

```
    x = NULL
    y = NULL
    for(i in 1:N)
    {
     s = i-1+step
     x = rbind(x,a[i:s])
     y = rbind(y,a[s+1])
    }
    X = array(x, dim=c(N,step,1))

#Creación del modelo
model = keras_model_sequential() %>%
  layer_simple_rnn(units=128, input_shape=c(step, 1), activation="relu") %>%
  layer_dense(units=64, activation = "relu") %>%
  layer_dense(units=32) %>%
  layer_dense(units=1, activation = "linear")

#Compilación del modelo usando mse y el optimizador de adam
model %>% compile(loss = 'mse',
                  optimizer = 'adam',
                  metrics = list("mean_absolute_error")
                   )
#Resumen del modelo
model %>% summary()

model %>% fit(X,y, epochs=50, batch_size=32, shuffle = FALSE, verbose=0)
y_pred  =  model %>% predict(X)
#Evaluación
scores  =  model %>% evaluate(X, y, verbose = 0)
print(scores)

#Graficando
x_axes = seq(1:length(y_pred))
plot(x_axes, y, type="l", col="red", lwd=2)
lines(x_axes, y_pred, col="blue",lwd=2)
legend("topleft", legend=c("y-original", "y-predicted"),
        col=c("red", "blue"), lty=1,cex=0.8)
```

```
#Creación del modelo
model = keras_model_sequential() %>%
  layer_lstm(units=128, input_shape=c(step, 1), activation="relu") %>%
  layer_dense(units=64, activation = "relu") %>%
  layer_dense(units=32) %>%
  layer_dense(units=1, activation = "linear")

#Compilación del modelo usando mse y el optimizador de adam
model %>% compile(loss = 'mse',
                  optimizer = 'adam',
                  metrics = list("mean_absolute_error")
                   )
#Resumen del modelo
model %>% summary()

model %>% fit(X,y, epochs=50, batch_size=32, shuffle = FALSE, verbose=0)
y_pred  =  model %>% predict(X)
#Evaluación
scores  =  model %>% evaluate(X, y, verbose = 0)
print(scores)

#Graficando
x_axes = seq(1:length(y_pred))
plot(x_axes, y, type="l", col="red", lwd=2)
lines(x_axes, y_pred, col="blue",lwd=2)
legend("topleft", legend=c("y-original", "y-predicted"),
        col=c("red", "blue"), lty=1,cex=0.8)
```

# Bibliografía


Bilberto Batres-Estrada. Deep learning for multivariate financial time series, 2015.

Ian Wong Scott Busseti, Enzo Osb. Deep learning for time series modeling. *Technical report, Stanford University*, pages 1–5, 2012.

Karl Sant Dingli, Alexiei Fournier. Financial time series forecasting–a deep learning approach. *International Journal of Machine Learning Computing*, 7(5):118–122, 2017.

John Cristian Borges Gamboa. Deep learning for time-series analysis. *arXiv preprint arXiv:1701.01887*, 2017.

Yoshua Courville Aaron Bengio Yoshua Goodfellow, Ian Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

Sujit Gulli, Antonio Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

Zhifeng Li Rongpeng Chen Xianfu Liu Zhiming Zhang Honggang Hua, Yuxiu Zhao. Deep learning with long short-term memory for time series prediction. *IEEE Communications Magazine*, 57(6):114–119, 2019.

Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

AO Karakoyun, ES Cibikdiken. Comparison of arima time series model and lstm deep learning algorithm for bitcoin price forecasting. In *The 13th Multidisciplinary Academic Conference in Prague*, volume 2018, pages 171–180, 2018.

John D Kelleher. *Deep learning*. MIT press, 2019.

Zhihao Liu, Hui Long. An improved deep learning model for predicting stock market price time series. *Digital Signal Processing*, 102:102741, 2020.

Yosi Navon, Ariel Keller. Financial time series prediction using deep learning. *arXiv preprint arXiv:1711.04174*, 2017.

Shyi-Ming Pedrycz, Witold Chen. *Deep Learning: Algorithms and Applications*. Springer, 2020.

Mehmet Ugur Ozbayoglu Ahmet Murat Sezer, Omer Berat Gudelek. Financial time series forecasting with deep learning: A systematic literature review: 2005–2019. *Applied Soft Computing*, 90:106181, 2020.

Yuanming Lu Jiawei Xu Jun Xiao Gang Shen, Zhipeng Zhang. A novel time series forecasting model with deep learning. *Neurocomputing*, 396:302–313, 2020.

Kenneth Shukla, Nishant Fricklas. *Machine learning with TensorFlow*. Manning Greenwich, 2018.

Antonio Troncoso A Martínez-Álvarez Francisco Torres, José F Galicia. A scalable approach based on deep learning for big data time series forecasting. *Integrated Computer-Aided Engineering*, 25(4):335–348, 2018.