UNIVERSIDAD SANTO TOMAS
PRIMER CLAUSTRO UNIVERSITARIO DE COLOMBIA

Comunicaciones en Estadística
Septiembre 2022, Vol. 15, No. 2, pp. 16–38

Pronóstico del COVID-19 en Colombia utilizando Redes Neuronales Recurrentes con celdas de gran memoria de corto plazo y unidades recurrentes cerradas

Forecasting of COVID-19 in Colombia using recurrent neural networks with long short term memory and gated recurrent units

Yeison Armando Buitrago López[a]
yeabuitragol@correo.udistrital.edu.co

Luis Alejandro Másmela Caita[b]
lmasmela@udistrital.edu.co

Resumen

El 6 de marzo del 2020, el primer caso de COVID-19 fue reportado en Colombia, este virus, declarado como una emergencia de salud pública de importancia internacional ha afectado diferentes sectores. Existe un auge en cuanto al número de estudios que buscan hacer pronósticos en diversos aspectos que tienen que ver con este virus. El presente trabajo muestra los aspectos teóricos de las redes neuronales recurrentes y se utilizan para crear una predicción de 60 días sobre los casos acumulados, fallecidos acumulados y recuperados acumulados disponibles desde el 6 de marzo del 2020 hasta el 6 de marzo del 2022. Redes neuronales con celdas GRU y LSTM junto con las clásicas RNN fueron utilizadas para hacer estos pronósticos.

***Palabras clave*:** Redes Neuronales Recurrentes, COVID-19, RNN, GRU, LSTM.

Abstract

On march 6 of 2020, the first case of COVID-19 was reported in Colombia. This virus, declared a public health emergency of international importance, has affected different sectors. There is a boom in the number of studies that make forecasts in various aspects that have to do with this virus. The present work shows the theoretical aspects of recurrent neuronal networks and his use to create a 60-day forecast on cumulative cases, cumulative deaths and cumulative recovered, available from march 6 2020 to march 6 2022. Neural networks with GRU and LSTM cells along with the classic RNN were used to make these forecasts.

***Keywords*:** Recurrent Neural Networks, COVID-19, RNN, GRU, LSTM.

[a]Egresado Universidad Distrital
[b]Cordinador y docente de matemáticas Universidad Distrital.

1. Introducción

Los coronavirus (CoV) son virus que surgen periódicamente en diferentes áreas del mundo y que son causa de Infección Respiratoria Aguda (IRA), se presenta como cuadro gripal, que puede llegar a ser leve, moderado o grave. El nuevo coronavirus (COVID-19) ha sido catalogado por la Organización Mundial de la Salud como una emergencia en salud pública de importancia internacional (ESPII). Se han identificado casos en todos los continentes, el 6 de marzo del 2020 se confirmó el primer caso en Colombia.

En Colombia se han tomado diferentes medidas para controlar la propagación del virus, tales como cuarentenas, testeos, uso obligatorio del tapabocas, distanciamiento social, entre otras. En un principio, el virus se propago de manera rápida afectando la salud de las personas y la economía del país. Una de las mayores preocupaciones de las autoridades y personas interesadas es tener un estimado del número de casos diarios, recuperados y fallecidos utilizando los datos disponibles para generar un pronóstico. Estos pronósticos, pueden ayudar al gobierno o al sistema de salud a tomar medidas de antemano para un número previsto de casos.

Actualmente el aprendizaje automático y el aprendizaje profundo son dos técnicas que han ganado mucho terreno debido a su capacidad de trabajar con cantidades grandes de datos. Estas técnicas son muy eficaces encontrando la relación entre los datos sin haberla definido antes, como menciona Swapnarek (rekja Hanumanthu, 2020, pág 11). Además, estas técnicas son capaces de encontrar tendencias en las series de tiempo dado un periodo de tiempo determinado como menciona Swapnarek (rekja Hanumanthu, 2020, pág 18). Varios investigadores han usado el aprendizaje automático y el aprendizaje profundo para hacer un pronóstico a corto plazo de la pandemia del COVID-19. Ismail et al et al. (2020) realizó un análisis comparativo y pronóstico del COVID-19 en diferentes países europeos, utilizando modelos como LSTM (*Long Short Term Memory*), ARIMA (*Autoregressive Integrated Moving Average*) y NARNN (*Autoregression Neural Network*). En este estudio el modelo LSTM fue el más preciso con las predicciones realizando un pronóstico de 14 días. Rahele Kafieh et al et al. (2021b), realizaron un pronóstico del COVID-19 en Irán utilizando MLP (*Multilayer perceptrón*), bosques aleatorios y diferentes versiones de LSTM; encontraron que el mejor rendimiento lo obtuvo una versión modificada de LSTM, llamada M-LSTM.

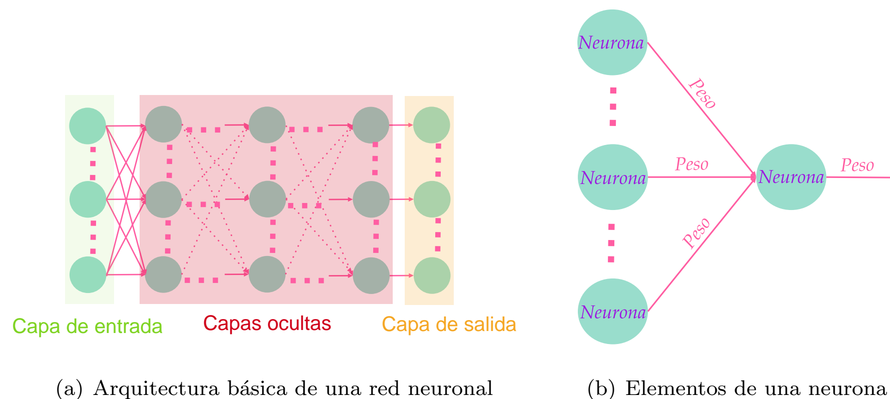
En este trabajo realizamos un estudio del artículo *Forecasting of COVID-19 using deep layer Recurrent Neural Network (RNNs) with Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) cells.* et al. (2021a), en él se exploran diferentes arquitecturas del aprendizaje profundo con el objetivo de lograr mejores pronósticos. Nosotros implementamos estas metodologías al conjunto de datos de Colombia en el mismo periodo de tiempo y presentamos los resultados. En la primera Sección 2, introduciremos conceptos claves para la construcción de redes neuronales, tales como el algoritmo de optimización del descenso del gradiente y funciones de activación. Luego, en esta misma sección, veremos cómo se construye un perceptrón 2.6 y cómo a través de éste logramos crear una red más complicada llamada MLP 2.7. Después, introducimos las redes neuronales recurrentes 3, las cuales son especiales para tratar con sucesiones, o series de tiempo, pero presenta problemas con el algoritmo de aprendizaje retropropagación a través del tiempo, de lo que hablaremos en la Sección 3.4. En la Sección 3.5 introducimos las RNN (Recurrent Neural Network) con celdas LSTM (Long Short Term Memory) y GRU (Gated Recurrent Unit), que ayudan a evitar los problemas que se presentan en las RNN Computer Science (2022). Y por último, en la Sección 4 mostraremos resultados de simulación, que es un pronóstico de 60 días de los casos acumulados, recuperados

acumulados y fallecidos acumulados de Colombia utilizando RNN simple, RNN con celdas LSTM y RNN con celdas GRU, utilizando los datos de la página de datos abierto de Colombia Nacional (2022).

2. Preliminares

2.1. Breve Descripción

Las redes neuronales son un modelo del aprendizaje profundo inspirado en la estructura de las redes neuronales del cerebro. La arquitectura de una red neuronal se divide en una capa de entrada, las capas ocultas y una capa de salida, como se muestra en la Figura 1(a). En cada una de estas capas encontramos un número determinado de neuronas, nodos o unidades, cuyas conexiones se llaman pesos, como en la Figura 1(b) a través de estas conexiones se realizan cálculos complejos.



(a) Arquitectura básica de una red neuronal

(b) Elementos de una neurona

Figura 1: Breve descripción de la arquitectura de una red neuronal

2.2. Funciones de activación

Una función de activación es una función que esta en los nodos, neuronas o unidades de las redes neuronales, estas están presentes para ayudar en el aprendizaje de patrones complejos que se encuentran en los datos. Una función de activación toma el valor de salida de la celda anterior y le da cierta forma para ser llevada a la celda siguiente.

Las funciones de activación en las redes neuronales son de diferentes tipos y:

- Ayudan a mantener los valores de la salida en cierto rango, por ejemplo: si utilizamos un función sigmoidea, nuestro valores de salida siempre estarán entre 0 y 1. Es importante mantener los valores de las redes neuronales en cierto rango para no tener problemas computacionales, debido a la extensión de parámetros a entrenar.
- Agregan no-linealidad a la red neuronal.

La Figura 2, muestra algunas de las funciones de activación más utilizadas en las redes neuronales.

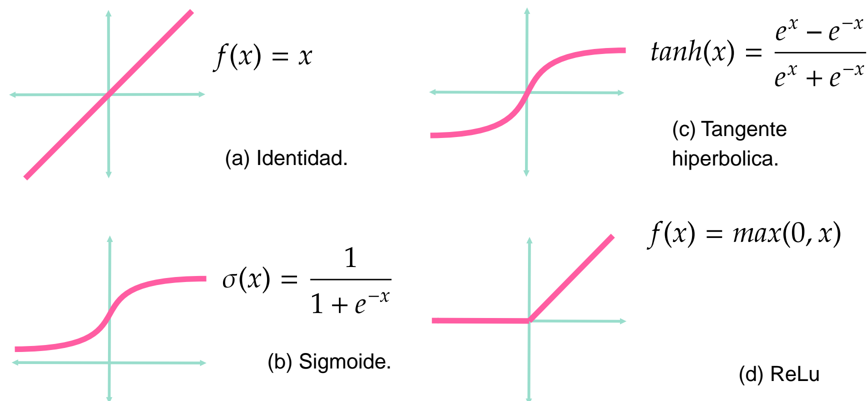


Figura 2: Algunos ejemplos de funciones de activación más utilizadas

2.3. Función de costo

- **Función de pérdida**: Es una función definida en un punto o en un dato, la predicción y el valor actual. Por ejemplo, el error cuadrático:

$$\left(y^{(i)} - \hat{y}^{(i)}\right)^2,$$

donde $y^{(i)}$ es valor actual y $\hat{y}^{(i)}$ es la predicción para el i-ésimo ejemplo.

- **Función de costo**: Esta función es más general. Es la suma de las funciones de pérdida a lo largo del conjunto de entrenamiento[1] de nuestra red neuronal. El error cuadrático medio (MSE), por ejemplo, dado por:

$$\text{MSE} = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2,$$

donde $y^{(i)}$ es valor actual, $\hat{y}^{(i)}$ es la predicción el i-ésimo ejemplo, y m es el número de ejemplos.

- **Función RMSE**: En el contexto de las redes neuronales, utilizaremos la raíz de la función MSE, así:

$$\text{RMSE} = \sqrt{\frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}, \tag{1}$$

y aquellos modelos con un menor valor de RMSE, diremos que tienen mejor desempeño.

[1]El conjunto de entrenamiento es un conjunto de datos de ejemplos utilizados durante el proceso de aprendizaje de la red neuronal y se utiliza para ajustar los parámetros (pesos).

2.4. Regularización

Uno de los diferentes problemas que se presentan al momento de trabajar con diferente modelos de aprendizaje automático y aprendizaje profundo es el sobreajuste. El sobreajuste ocurre cuando nuestra red neuronal aprende muy bien los detalles del conjunto de entrenamiento. Esto afecta el rendimiento del modelo al momento de trabajar con datos nuevos. La regularización es una técnica que afecta el aprendizaje de una red neuronal de tal manera que el modelo generalice de mejor forma.

2.4.1. Regularización L2

Es uno de los tipos de regularización más usados, este actualiza la función de costo agregando otro término, conocido como término de regularización o penalización, así,

$$\text{Función de costo} = \sum \text{Función de perdida} + \frac{\lambda}{2m} \sum ||w||^2.$$

Al optimizar, los valores en los pesos decrecen, por lo tanto las redes neuronales con pesos más pequeños nos producen modelos más simples. Se introduce un parámetro λ. Este hiperparámetro es optimizado para mejores resultados. La regularización $L2$ es también conocida como disminución de pesos o *weigth decay.*

2.5. Descenso del gradiente

El descenso del gradiente en un algoritmo de optimización que utilizaremos para entrenar nuestra red neuronal. Sea $J(w, b)$ una función de costo, donde, w son los pesos de la red neuronal, b el sesgo de la misma, y α la taza de aprendizaje. La tasa de aprendizaje define la longitud de los pasos que daremos para llegar al mínimo de la función. Deseamos encontrar un mínimo de esta función, entonces a medida que calculamos la función de costo, actualizamos los pesos de la siguiente manera:

$$w_{t+1} := w_t - \alpha \frac{\partial J(w_t, b_t)}{\partial w_t},$$

$$b_{t+1} := b_t - \alpha \frac{\partial J(w_t, b_t)}{\partial b_t}.$$

- **Descenso del gradiente por lotes**: Utilizamos todos los datos disponibles para calcular el gradiente.

- **Descenso del gradiente estocástico**: Utilizamos solo un dato aleatorio para encontrar el gradiente.

- **Descenso del gradiente por minilotes**: Utilizamos una muestra aleatoria de menor tamaño que el número total de datos para encontrar el gradiente.

2.5.1. Modificaciones del descenso del gradiente.

A continuación se presentan diferentes modificaciones al algoritmo de descenso del gradiente. Estas modificaciones son requeridas para optimizar este algoritmo.

- **Momentum**: Utilizamos el promedio exponencial ponderado de movimiento, este nos ayuda a movernos en la dirección adecuada de manera mucho más rápida y las posibles oscilaciones son mas atenuadas. Ambas características nos ayudan a que la convergencia ocurra en menor tiempo.

$$V_{t+1} := \beta V_t + (1-\beta)\frac{\partial J(w_t,b_t)}{\partial w_t}, \qquad w_{t+1} := w_t - \alpha V_{t+1}.$$

$$V_{t+1} := \beta V_t + (1-\beta)\frac{\partial J(w_t,b_t)}{\partial b_t}, \qquad b_{t+1} := b_t - \alpha V_{t+1}.$$

  Donde $V_0 = 0$, V_t es el promedio en movimiento exponencial del gradiente, β es la constante de momentum y α es la taza de aprendizaje.

- **RMSprop (Root Mean Square-Propagation)**: En este método, la tasa de aprendizaje se adapta para cada parámetro.

$$S_{t+1} := \beta S_t + (1-\beta)\left(\frac{\partial J(w_t,b_t)}{\partial w_t}\right)^2, \qquad w_{t+1} := w_t - \alpha\left(\frac{1}{\sqrt{S_{t+1}}+\epsilon}\right)\left(\frac{\partial J(w_t,b_t)}{\partial w_t}\right).$$

$$S_{t+1} := \beta S_t + (1-\beta)\left(\frac{\partial J(w_t,b_t)}{\partial b_t}\right)^2, \qquad b_{t+1} := b_t - \alpha\left(\frac{1}{\sqrt{S_{t+1}}+\epsilon}\right)\left(\frac{\partial J(w_t,b_t)}{\partial b_t}\right).$$

  Donde $S_0 = 0$, S_t es el promedio en movimiento exponencial del cuadrado de los gradientes, β es la constante de momentum y α la taza de aprendizaje.

- **Adam (Adaptive Moment Estimation)**: Es una combinación de RMSProp y Momentum. Por un lado tenemos el promedio exponencial ponderado del movimiento de los gradientes al cuadrado, y por otro lado tenemos el promedio exponencial ponderado del movimiento de los pasos anteriores. Adam, por lo general es más rápido que los otros métodos.

$$V_{t+1} := \beta V_{t+1} + (1-\beta)\frac{\partial J(w_t,b_t)}{\partial w_t} \qquad V_{t+1} := \beta V_t + (1-\beta)\frac{\partial J(w_t,b_t)}{\partial b}$$

$$S_{t+1} := \beta S_t + (1-\beta)\left(\frac{\partial J(w_t,b_t)}{\partial w_t}\right)^2, \qquad S_{t+1} := \beta S_t + (1-\beta)\left(\frac{\partial J(w_t,b_t)}{\partial b_t}\right)^2.$$

$$w_{t+1} := w_t - \alpha\frac{V_{t+1}}{\sqrt{S_{t+1}+\epsilon}}, \qquad b_{t+1} := b_t + \alpha\frac{V_{t+1}}{\sqrt{S_{t+1}+\epsilon}}.$$

2.5.2. Tasa de aprendizaje con decaimiento

La tasa de decaimiento es una técnica para entrenar redes neuronales. Se comienza con una gran tasa de aprendizaje y luego se decae varias veces. Este proceso ayuda a la optimización como a la generalización. Una tasa de aprendizaje inicialmente grande acelera el entrenamiento de la red neuronal, luego, el decaimiento ayuda a la red neuronal a converger a una solución.

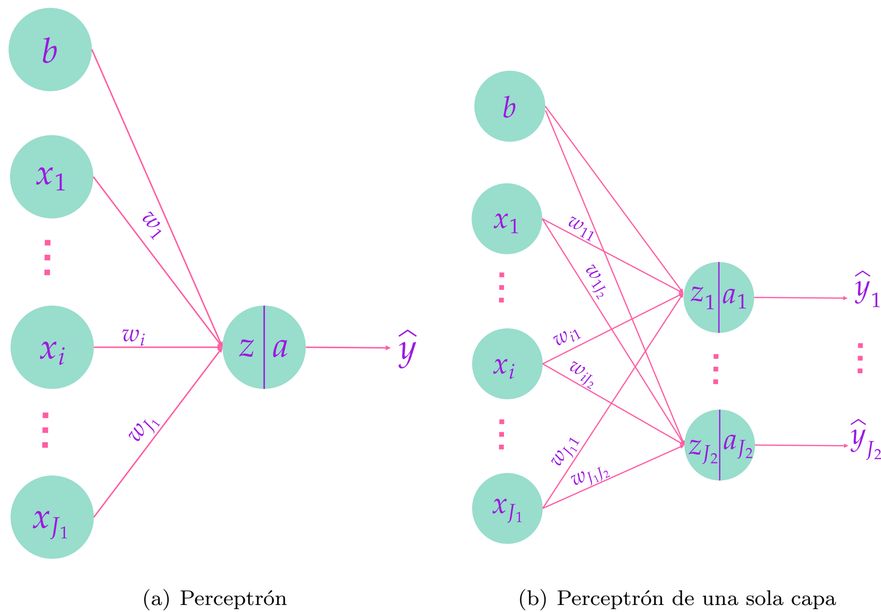
2.6. Perceptrón

2.6.1. Perceptrón de una sola neurona

El perceptrón, es la primera y más simple red neuronal. Se usan los perceptrones de una sola neurona para la clasificaciones de patrones linealmente separables. La arquitectura del perceptrón se muestra en la Figura 3(a), donde x_i para $i = 1, \cdots J_1$ son las neuronas de la capa de entrada, w_i son los pesos de la red neurona, b es un sesgo, z corresponde la ecuación (2), a corresponde a la ecuación (3) donde $\phi(\cdot)$ es una función de activación.

$$z = \sum_{i=1}^{J_1} w_i x_i + b = w^T x + b. \tag{2}$$

$$a = \phi(z) \tag{3}$$



(a) Perceptrón (b) Perceptrón de una sola capa

Figura 3: Perceptrones

2.6.2. Perceptrón de una sola capa

Cuando más neuronas son agregadas en la capa oculta junto a la función de activación $\phi(\cdot)$, tenemos una perceptrón de una sola capa como se muestra en la Figura 3(b). Este perceptrón de una sola capa puede ser usado para clasificar un vector de entrada $\mathbf{x}$ en más clases. Para un perceptrón con J_1 neuronas en la capa de entrada y J_2 neuronas en la capa de salida tenemos que:

$$z = W^T x + b, \tag{4}$$

$$\hat{y} = a = \phi(z), \tag{5}$$

donde el vector de entrada será $x = (x_1, \cdots, x_{J_1})$, z esta dado por $z = (z_1, \cdots, z_{J_2})$ y el vector de salida es $\hat{y} = (\hat{y}_1, \cdots, \hat{y}_{J_2})$ donde $\hat{y}_i = a_i$ con $i = 1, ..., J_2$ y además $a = (a_1, \cdots, a_{J_2}) = (\phi(z_1), \cdots, \phi(z_{J_2})) = \phi(z)$ donde ϕ es la función de activación.

2.7. Redes neuronales multicapa

Las redes neuronales multicapa (MLP) son redes feedfoward con una o más capas de neuronas entre la capa de entrada y la capa de salida. Las neuronas de salida representan un hiper-plano en el espacio de las características de entrada. La arquitectura de una red neuronal multicapa se presenta en la Figura 4. Suponga que esta red neuronal tiene L capas, cada una con J_l neuronas con $l = 1, \cdots, L$. Los pesos entre la capa $(l-1)$ y la capa l son denotados por $W^{[l-1]}$; el sesgo, la salida, y la función de activación en la i-ésima neurona en la l-ésima capa son, respectivamente: $b_i^{[l]}, a_i^{[l]}$ y $\phi_i^{[l]}$

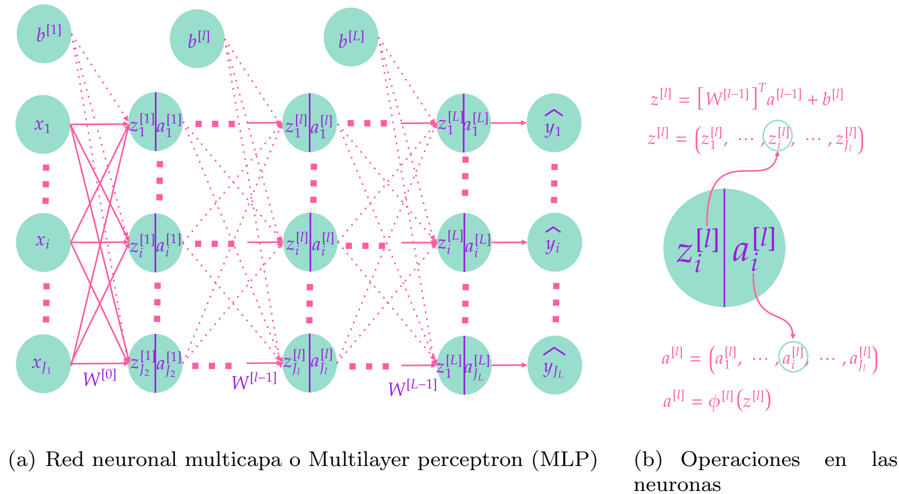
Las redes neuronales multicapa pueden ser utilizadas para clasificación de patrones linealmente no separables y para aproximar funciones. De la Figura **??**, observamos las siguientes relaciones. Para $l = 2, \cdots, L$ y el i-ésimo ejemplo del conjunto de datos de entrenamiento:

$$\hat{y}^{(i)} = a^{[L](i)}, \quad a^{[1](i)} = x^{(i)},$$

$$z^{[l](i)} = \left[W^{[l-1]}\right]^T a^{[l-1](i)} + b^{[l]},$$

$$a^{[l](i)} = \phi^{[l]}(z^{[l](i)}),$$

donde $z^{[l](i)} = \left(z_1^{[l](i)}, \cdots, z_{J_l}^{[l](i)}\right)$, $W^{[l]}$ es la matriz de pesos con dimensiones $(J_{m-1} \times J_m)$, los valores de a serán $a^{[l]}(i) = \left(a_1^{[l](i)}, \cdots, a_{J_{J_{l-1}}}^{[l](i)}\right)^T$, y el sesgo viene dado por $b^{[l]} = \left(b_1^{[l]}, \cdots, b_{J_l}^{[l]}\right)^T$ y $\phi^{[l]}(\cdot)$ aplica $\phi^{[l](i)}(\cdot)$ a la i-ésima componente del vector



(a) Red neuronal multicapa o Multilayer perceptron (MLP) (b) Operaciones en las neuronas

Figura 4: Red neuronal MLP

Las redes neuronales MLP son aproximadores universales, esto se debe a funciones de activación que quitan las linealidades utilizadas en las neuronas. Este tipo de redes neuronales son muy eficientes al momento de aproximar funciones en espacios de altas dimensiones. Ver (Ke-Lin Du, 2019, pag 98)

2.7.1. Retropropagación

La retropropagación, es una regla de aprendizaje para el desarrollo de aprendizaje supervisado. No solo es utilizado para entrenar redes neuronales feedfoward tales como las MLP, sino también son utilizadas para el entrenamiento de las redes neuronales recurrentes (RNN).

El algoritmo de la retropropagación, propaga el error entre el valor actual y el resultado de la red neuronal. Después de proporcionar un patrón de entrada, la salida de la red se compara con un patrón objetivo dado y se calcula el error de cada neurona en la salida. Esta señal de error se propaga hacia atrás y, por lo tanto, se establece un sistema de control de bucle cerrado. Los pesos pueden ser ajustados mediante el algoritmo de descenso del gradiente.

Para implementar el algoritmo de la retropropagación, es necesario tener una función continua, monótona creciente y diferenciable. La función sigmoidea o logística y la tangente hiperbólica son utilizadas usualmente.

La función objetivo a optimizar, será MSE, entre la salida de la red neuronal y el valor actual para todos los ejemplos del conjunto de entrenamiento.

$$J = \frac{1}{m}\sum_{i=1}^{m} l = \frac{1}{2m}\sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2,$$

donde m es el número de ejemplos en el conjunto de entrenamiento.

$$l = \frac{1}{2}||\hat{y}^{(i)} - y^{(i)}||^2 = \frac{1}{2}e^{T(i)}e^{(i)},$$

$$e^{(i)} = \hat{y}^{(i)} - y^{(i)},$$

aquí el j-ésimo elemento de $e^{(i)}$ será $e_j^{(i)}$ y este será $e_j^{(i)} = \hat{y}_j^{(i)} - y_j^{(i)}$. Note que el factor $\frac{1}{2}$ es usado convenientemente al momento de calcular las derivadas. Todo $W^{[l-1]}, b^{[l]}$ con $l = 2, ..., L$ pueden ser representado mediante matrices $W = [w_{ij}]$, la función de costo J puede ser minimizada utilizando el algoritmo de descenso del gradiente.

Aplicando la regla de la cadena

$$\frac{\partial l}{\partial w_{uv}^{[l]}} = \frac{\partial l}{\partial z_v^{[l+1](i)}} \frac{\partial z_v^{[l+1](i)}}{\partial w_{uv}^{[l]}}. \tag{6}$$

El segundo factor se obtiene

$$\begin{aligned} \frac{\partial z^{[l+1](i)}}{w_{uv}^{[l]}} &= \frac{\partial}{\partial w_{uv}^{[l]}} \left(\sum_{\omega=1}^{J_l} w_{\omega v}^{[l]} a_{\omega}^{[l](i)} + b_v^{[l+1]} \right) \\ &= a_u^{[l](i)}. \end{aligned} \tag{7}$$

El primer factor de (6), puede ser derivado usando la regla de la cadena

$$\begin{aligned} \frac{\partial l}{\partial z_v^{[l+1](i)}} &= \frac{\partial l}{\partial a_v^{[l+1](i)}} \frac{\partial a_v^{[l+1](i)}}{\partial z_v^{[l+1](i)}} \\ &= \frac{\partial l}{\partial a_v^{[l+1](i)}} \phi'^{[l+1]}_v \left(z_v^{[l+1](i)} \right), \end{aligned} \tag{8}$$

Para resolver el factor de (8), necesitamos considerar dos situaciones para las neuronas de salida: $(l = L - 1)$ y para las neuronas ocultas $(l = 1, ..., L - 2)$:

$$\frac{\partial l}{\partial a_v^{[l+1](i)}} = e_v^{(i)}, \qquad l = L - 1, \tag{9}$$

$$\begin{aligned} \frac{\partial l}{\partial a_v^{[l+1](i)}} &= \sum_{\omega=1}^{J_{m+2}} \left(\frac{\partial l}{\partial z_\omega^{[l+2](i)}} \frac{\partial z_\omega^{[l+2](i)}}{\partial a_v^{[l+1](i)}} \right), \\ &= \sum_{\omega=1}^{J_{m+2}} \frac{\partial l}{\partial z_\omega^{[l+2](i)}} w_{v\omega}^{[l+1]}, \qquad l = 1, \cdots, L - 2. \end{aligned} \tag{10}$$

Definimos la función delta de la siguiente manera

$$\delta_v^{[l](i)} = -\frac{\partial l}{\partial z_v^{[l](i)}}, \qquad l = 2, \cdots, L. \tag{11}$$

Sustituyendo (6),(10),y (11) en (8), finalmente obtenemos las neuronas de salidas ($l = L-1$) y las neuronas ocultas ($l = 1, .., L-2$) así:

$$\delta^{[L](i)} = -e_v^{(i)} \dot{\phi}_v^{[L]} \left(z_v^{[L](i)} \right), \qquad l = L-1, . \tag{12}$$

$$\delta_v^{[l+1](i)} = \dot{\phi}^{[l+1]} \left(z^{[l+1]_v} \right) \sum_{\omega=1}^{J_{l+2}} \delta_\omega^{[l+2](i)} w_{v\omega}^{[l+1]}, \qquad l = 1, \cdots, L-2. \tag{13}$$

Las ecuaciones (12) y (13) nos dan un método recursivo para resolver $\delta_v^{[l+2](i)}$ para toda la red neuronal. De esta forma, W puede ser ajustado por

$$\frac{\partial l}{\partial w_{uv}^{[l]}} = -\delta^{[l+1](i)} a_u^{[l](i)}. \tag{14}$$

3. Redes Neuronales Recurrentes

3.1. Modelado secuencial

Los modelos de aprendizaje automático o profundo, cuya entrada o salida es una sucesión[2], son conocidos como modelos secuenciales. Textos, audio, vídeo, series de tiempo y otros tipos de sucesiones son ejemplos de datos secuenciales.

3.2. Redes neuronales recurrentes

Las Redes Neuronales Multicapa (MLP) son estáticas, y por tanto son incapaces de procesar series de tiempo o sucesiones, es decir no son capaces de capturar dependencias los datos. Es acá donde aparecen las Redes Neuronales Recurrentes (RNN), que son una clase de red neuronal donde las conexiones logran capturar estas dependencias a lo largo de una sucesión. Se ha demostrado que las redes neuronales recurrentes tienen la llamada propiedad de aproximación universal Zimmermann (2007), es decir, son capaces de aproximar arbitrariamente sistemas dinámicos no lineales, mediante la realización de funciones complejas de la sucesión de entrada a la sucesión de salida. Sin embargo, la arquitectura particular de una RNN determina cómo la información fluye entre las diferentes neuronas y su diseño correcto es crucial para la realización de un sistema de aprendizaje robusto MIT (2022a).

[2]Una secuencia de números u otros objetos (letras, palabras..etc) relacionados entre sí

3.2.1. Arquitectura

Las capas de una RNN puede ser divididas en *capas de entrada*, *capas ocultas*, y *capa de salida*. Mientras que la capa de salida esta caracterizada por conexiones feedfoward, las capas ocultas tienen conexiones recurrentes o ciclos. La sucesión de entrada tiene longitud T_x y la sucesión de salida tiene longitud T_y.Utilizaremos los parentisis $\langle\cdot\rangle$ para decir que $x^{\langle t\rangle}$ es el dato en el tiempo t o la posición t, donde $t = 1, \cdots T_x$. Esta nueva red neuronal cuenta con diferentes pesos, respecto a la MLP; introducimos los pesos $W_{aa}, W_{ax}, W_{ya}, b_a$ y b_y con las ecuaciones que la definen son:

$$a^{\langle t\rangle} = \phi_1 \left(W_{aa} a^{\langle t-1\rangle} + W_{ax} x^{\langle t\rangle} + b_a \right). \tag{15}$$

$$\hat{y}^{\langle t\rangle} = \phi_2 \left(a^{\langle t\rangle} W_{ya} + b_y \right). \tag{16}$$

donde, $a^{\langle 0\rangle} = 0$, (15) nos muestra cómo las entradas de la red y las capas ocultas se relacionan y en (16) vemos como la capa de salida en un tiempo t se relaciona con la entrada en un tiempo t y la memoria pasada $a^{\langle t-1\rangle}$ a través de (15). Existen diferentes tipo de arquitecturas, que dependen del problema y por lo tanto dependen de las longitudes de T_x y T_y. Algunos ejemplos de arquitecturas son:

- **Uno para uno**: Una red neuronal tradicional puede ser vista como una RNN, y su arquitectura se llama uno para uno. Acá tenemos que $T_x = T_y = 1$. Ver Figura 5(a)

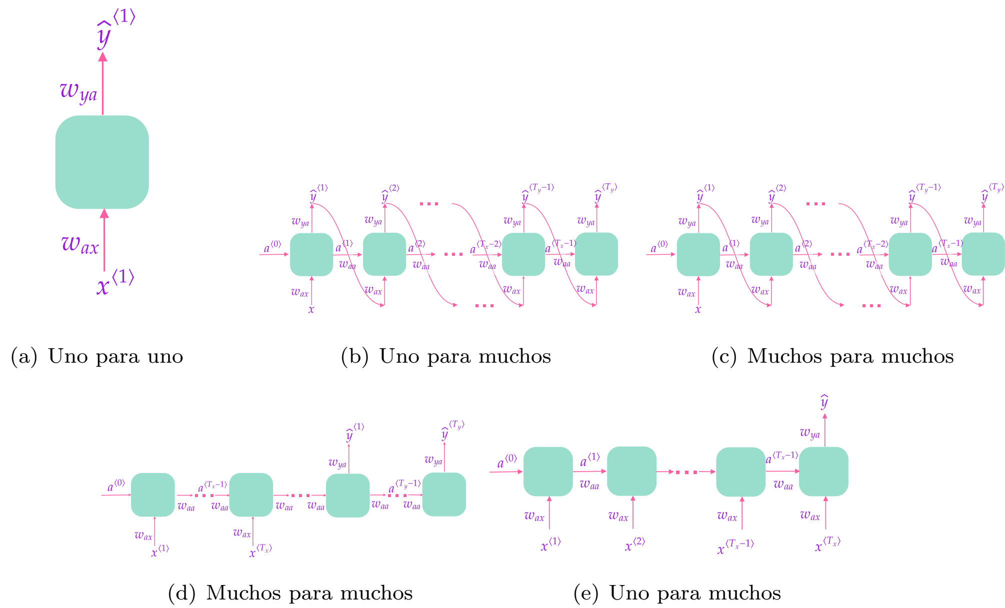
- **Uno para muchos**: Este tipo de arquitectura es útil para, por ejemplo la generación de musica. Acá tenemos que $T_x = 1$ y $T_y > 1$. Ver Figura 5(b)

- **Muchos para muchos**:

  - Este tipo de arquitectura es utilizada, por ejemplo, para reconocimiento de la entidad nombrada, que busca localizar y clasificar en categorías predefinidas, como personas, organizaciones, expresiones de tiempo y cantidades. Acá $T_x = T_y$. Ver Figura 5(c)

  - En esta arquitectura tenemos que $T_x \neq T_y$, esta es muy útil para máquinas de traducción. Ver Figura 5(d)

- **Muchos para uno:** Este tipo de arquitectura es utilizada, por ejemplo, para la clasificación de sentimientos. Acá tenemos $T_x > 1$ y $T_y = 1$. Ver Figura 5(e)



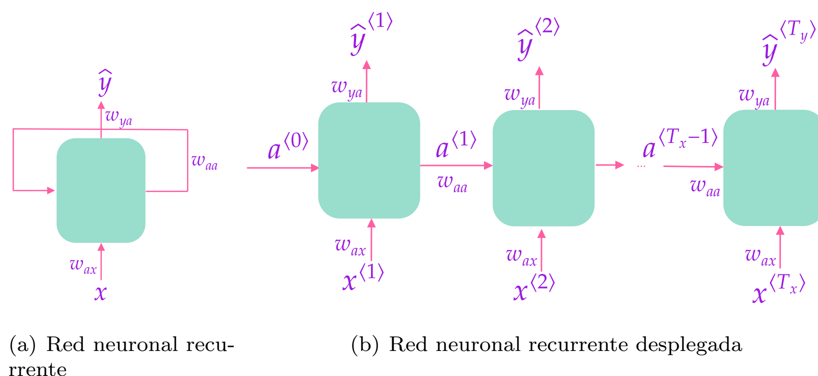
(a) Uno para uno (b) Uno para muchos (c) Muchos para muchos

(d) Muchos para muchos (e) Uno para muchos

Figura 5: Arquitecturas redes neuronales recurrentes

3.3. Retropropagación a través del tiempo

El algoritmo de aprendizaje de una RNN es una técnica de aprendizaje muy parecida a la retropropagación vista en capítulos anteriores aunque algunas precauciones se deben tomar en cuenta al momento de hacer esto con una RNN cuyo grafo correspondiente es uno con ciclos, vease la Figura 6(a). De hecho, para poder encontrar la relación entre los parámetros y la función de perdida, presentamos la red neuronal como una equivalente sin ciclos, este proceso se llama desplegado y consiste en replicar el estado oculto de la red neuronal por cada intervalo de tiempo, obteniendo una red neuronal multicapa particular, observe la Figura (6(b). La diferencia clave de una RNN desplegada con una MLP estándar es que las matrices de peso están restringidas para asumir los mismos valores en todas las réplicas de las capas, ya que representan la aplicación recursiva de la misma operación. A través de esta transformación, la red puede ser entrenada con algoritmos estándar de aprendizaje, que originalmente se utilizaron el las redes multicapas, este proceso se llama *retropropagación a través del tiempo*, y es una de las técnicas más útiles para el entrenamiento de redes neuronales recurrentes.



(a) Red neuronal recu-rrente

(b) Red neuronal recurrente desplegada

Figura 6: Red neuronal recurrente desplegada

En la retropropagación vista en la Sección anterior, encontramos el gradiente de la función de pérdida respecto a los pesos de toda la red para su posterior actualización a través del algoritmo del descenso del gradiente. En la retropropagación a través del tiempo, necesitamos encontrar el gradiente de la función de pérdida respecto a los pesos $W_{aa}, W_{ax}, W_{ya}, b_a$ y b_y. Debido a la estructura esencialmente diferente de una RNN y una MLP, debemos hacer diferentes consideraciones respecto al valor de t para encontrar estos gradientes.

La función a optimizar será el MSE, entre la salida y el valor actual,

$$J = \frac{1}{m}\sum_{t=1}^{T_y} l^{\langle t \rangle} = \frac{1}{2m}\sum_{t=1}^{T_y} ||\hat{y}^{\langle t \rangle} - y^{\langle t \rangle}||^2.$$

El error de la función de perdida se calcula respecto a cada una de las neuronas en la capa de salida, estaremos calculado el gradiente $\frac{\partial l^{\langle \gamma \rangle}}{\partial \theta^{\langle \gamma \rangle}}$, con $\theta = \{W_{aa}, W_{ax}, W_{ya}, b_a, b_y\}$ y $\gamma = 1, \cdots, t, \cdots, T_y$, entonces sin pérdida de generalidad consideremos $\frac{\partial l}{\partial \theta^{\langle \gamma \rangle}} = \frac{\partial l^{\langle \gamma \rangle}}{\partial \theta^{\langle \gamma \rangle}}$ y $T_x = T_y$. Primero consideremos $\frac{\partial l}{\partial a^{\langle T_x \rangle}}$, éste es el error de la ultima unidad de tiempo, entonces no tiene que considerar errores, así: anteriores.

$$\frac{\partial l}{\partial a_i^{\langle T_x \rangle}} = \sum_j \frac{\partial l}{\partial o_j^{\langle T_x \rangle}} \frac{\partial o_j^{\langle T_x \rangle}}{\partial a^{\langle T_x \rangle}} = \sum_j \frac{\partial l}{\partial o_j^{\langle T_x \rangle}} w_{ya},$$

por lo tanto

$$\frac{\partial l}{\partial a^{\langle T_x \rangle}} = W_{ya}^T \frac{\partial l}{\partial o^{\langle T_x \rangle}}.$$

Consideremos ahora el caso en el que t es menor a la longitud de la sucesión de entrada.

$$\frac{\partial l}{\partial a^{\langle t \rangle}}, \qquad t < T_x,$$

$$\frac{\partial l}{\partial a^{\langle t\rangle}}=\sum_{k}\frac{\partial l^{\langle k\rangle}}{\partial a^{\langle t\rangle}}=\sum_{k\geq t+1}\frac{\partial l^{\langle k\rangle}}{\partial a^{\langle t\rangle}}+\frac{\partial l^{\langle k\rangle}}{\partial a^{\langle t\rangle}},$$

por lo tanto,

$$\frac{\partial l^{\langle t\rangle}}{\partial a^{\langle t\rangle}}=W_{ya}^{T}\frac{\partial l^{\langle t\rangle}}{\partial o^{\langle t\rangle}}.$$

$$\begin{aligned}\sum_{k\geq t+1}\frac{\partial l^{\langle k\rangle}}{\partial a_i^{\langle t\rangle}}&=\sum_{k\geq t+1}\sum_{j}\frac{\partial l^{\langle k\rangle}}{\partial h_j^{\langle t+1\rangle}}\frac{\partial h_j^{\langle t+1\rangle}}{\partial a_i^{\langle t\rangle}}\\&=\sum_{k\geq t+1}\sum_{j}\frac{\partial l^{\langle k\rangle}}{\partial a_j^{\langle t+1\rangle}}\frac{\partial a_j^{\langle t+1\rangle}}{\partial h_j^{\langle t+1\rangle}}\frac{\partial h_j^{\langle t+1\rangle}}{\partial a_i^{\langle t\rangle}}=\sum_{k\geq t+1}\sum_{j}\frac{\partial l^{\langle k\rangle}}{\partial a_j^{\langle t+1\rangle}}\phi'\left(h_j^{\langle t+1\rangle}\right)w_{ij}\\&=\sum_{j}\phi'\left(h_j^{\langle t+1\rangle}\right)w_{ij}^{T}\frac{\sum_{k\geq t+1}l^{\langle k\rangle}}{\partial a_j^{\langle t+1\rangle}}=\sum_{j}\phi'\left(h_j^{\langle t+1\rangle}\right)w_{ij}^{T}\frac{\partial l}{\partial a_j^{\langle t+1\rangle}},\end{aligned}$$

por lo tanto, tenemos,

$$\sum_{k}\frac{\partial l^{\langle k\rangle}}{\partial a^{\langle t\rangle}}=\sum_{k\geq t+1}\frac{\partial l^{\langle k\rangle}}{\partial a^{\langle t\rangle}}+\frac{l^{\langle t\rangle}}{\partial a^{\langle t\rangle}}=W^{T}\phi'\left(h_{\langle t+1\rangle}\right)\frac{\partial l}{\partial a^{\langle t+1\rangle}}+W_{ya}^{T}\frac{\partial l^{\langle t\rangle}}{\partial o^{\langle t\rangle}}.$$

Encontremos el gradiente de la función de pérdida respecto al peso W_{aa}, es decir $\dfrac{\partial l}{\partial W_{aa}}$.

$$\frac{\partial l}{\partial (w_{aa})_{ij}^{\langle t\rangle}}=\frac{\partial l}{\partial h_i^{\langle t\rangle}}\frac{\partial h_i^{\langle t\rangle}}{\partial (w_{aa})_{ij}^{\langle t\rangle}}=\frac{\partial l}{\partial a_i^{\langle t\rangle}}\frac{\partial a_i^{\langle t\rangle}}{\partial h_i^{\langle t\rangle}}\frac{\partial h_i^{\langle t\rangle}}{\partial (w_{aa})_{ij}^{\langle t\rangle}}=\frac{\partial l}{\partial a_i^{\langle t\rangle}}\phi'\left(h_i^{\langle y\rangle}\right)a_j^{\langle t-1\rangle},$$

por lo tanto,

$$\frac{\partial l}{\partial W_{aa}}=\sum_{t}\frac{\partial l}{\partial W_{aa}^{\langle t\rangle}}=\sum_{t}\frac{\partial l}{\partial a^{\langle t\rangle}}\phi'\left(h^{\langle t\rangle}\right)\left(a^{\langle t\rangle}\right)^{T}.$$

Encontremos el gradiente de la función de pérdida respecto el peso W_{ax}, es decir $\dfrac{\partial l}{\partial W_{ax}}$,

$$\frac{\partial l}{\partial (w_{ax})_{ij}^{\langle t\rangle}}=\frac{\partial l}{\partial h_i^{\langle t\rangle}}\frac{\partial h_i^{\langle t\rangle}}{\partial (w_{ax})_{ij}^{\langle t\rangle}}=\frac{\partial L}{\partial a_i^{\langle t\rangle}}\frac{\partial a_i^{\langle t\rangle}}{\partial}\frac{\partial h_i^{\langle t\rangle}}{\partial (w_{ax})_{ij}^{\langle t\rangle}}=\frac{\partial l}{\partial a_i^{\langle t\rangle}}\phi'\left(h_i^{\langle t\rangle}\right)x_j^{\langle t\rangle},$$

por lo tanto,

$$\frac{\partial l}{\partial W_{ax}}=\sum_{t}\frac{\partial l}{\partial W_{ax}^{\langle t\rangle}}=\sum_{t}\frac{\partial l}{\partial a^{\langle t\rangle}}\phi'\left(h^{\langle t\rangle}\right)\left(x^{\langle t\rangle}\right)^{T}.$$

Encontremos el gradiente de la función de pérdida respecto al peso W_{ya}, es decir $\frac{\partial l}{\partial W_{ya}}$,

$$\frac{\partial l}{\partial (w_{ya})_{ij}^{\langle t\rangle}} = \frac{\partial l}{\partial o_i^{\langle t\rangle}} \frac{\partial o_i^{\langle t\rangle}}{\partial (w_{ay})_{ij}^{\langle t\rangle}} = \frac{\partial L}{\partial o_i^{\langle t\rangle}} a_j^{\langle t\rangle},$$

por lo tanto,

$$\frac{\partial l}{\partial W_{ya}} = \sum_t \frac{\partial L}{\partial W_{ya}^{\langle t\rangle}} = \sum_t \frac{\partial l}{\partial o^{\langle t\rangle}} \left(a^{\langle T\rangle}\right)^T,$$

Siguiendo un procedimiento similar podemos encontrar $\frac{\partial l}{\partial b_a}$ y $\frac{\partial l}{\partial b_y}$.

Así pues hemos encontrado $\frac{\partial l}{\partial W_{aa}}$, $\frac{\partial l}{\partial W_{ax}}$, $\frac{\partial l}{\partial W_{ya}}$,$\frac{\partial l}{\partial b_a}$ y $\frac{\partial l}{\partial b_y}$ y por lo tanto podemos realizar el proceso de descenso del gradiente para realizar sus correspondientes actualizaciones Tamura (2020).

3.4. El gradiente que desaparece o explota.

3.4.1. Gradiente que desaparece

A medida que el algoritmo de la retropropagación a través del tiempo se lleva a cabo en una RNN, los gradientes pueden volverse cada vez más pequeños, debido a que muchos de estos valores son menores que 1, tanto que se aproximan a cero lo cual afecta el algoritmo en el sentido de que la actualización de los pesos no se altera y por lo tanto no podemos encontrar un mínimo de la función. Esto se conoce como el gradiente que desaparece (*vanishing gradient*) (Filippo Maria Bianchi, 2017, pag 17)

3.4.2. Gradiente que explota

También, a medida que el algoritmo de retropropagación a través del tiempo se desarrolla, los gradientes pueden volverse demasiado grandes, debido a que muchos de estos valores son mayores a 1, de tal manera que las actualizaciones de los pesos se vea afectadas y el algoritmo de descenso del gradiente diverge. Esto se conoce como el gradiente que explota (*exploding gradients*)

Para evitar estos problemas en el algoritmo de aprendizaje de una RNN, aparecen dos arquitecturas, LSTM y GRU, cuya idea principal es usar puertas para agregar o eliminar información selectivamente en cada unidad recurrente (Filippo Maria Bianchi, 2017, pag 17).

3.5. Redes de gran memoria de corto plazo o Long Short-Memory

La arquitectura *Long Short-Term memory* (LSTM) es de las arquitecturas más usadas debido a su funcionamiento superior en el modelado preciso de dependencias de datos tanto a corto como a largo plazo. LSTM intenta resolver el problema del gradiente que se desvanece no imponiendo

ningún sesgo hacia las observaciones recientes, pero mantiene un error constante que fluye hacia atrás en el tiempo. Una celda LSTM está compuesta por cinco componentes no-lineales diferentes, que interactúan entre ellas de una forma particular. El estado interno de la celda es modificado solo a través de interacciones lineales. Esto permite que la información se propague de manera suave a través del tiempo, con la consecuencia de mejoramiento en la capacidad de memoria de la celda. Las celdas de LSTM protegen y controlan la información de las celdas a través de tres puertas las cuales son implementadas por una sigmoidea y multiplicación uno a uno. Para controlar el comportamiento de cada puerta, un conjunto de parámetros son entrenados con el descenso del gradiente, para así poder resolver una tarea objetivo.

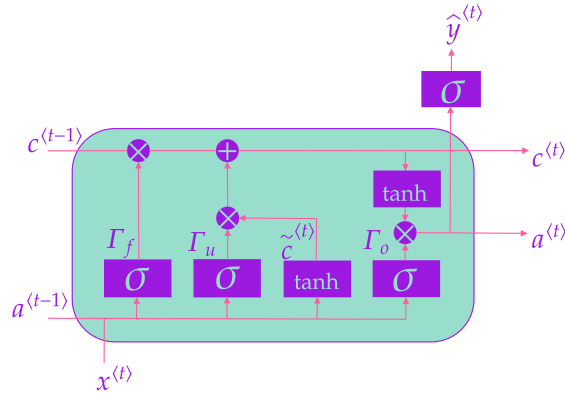


Figura 7: Celda LSTM

$$
\begin{aligned}
\tilde{c}^{\langle t\rangle} &= \tanh\left(W_{ca}\cdot a^{\langle t-1\rangle} + W_{cx}\cdot x^{\langle t\rangle} + b_c\right)\\
\Gamma_u &= \sigma\left(W_{ua}\cdot a^{\langle t-1\rangle} + W_{ux}\cdot x^{\langle t\rangle} + b_u\right)\\
\Gamma_f &= \sigma\left(W_{fa}\cdot a^{\langle t-1\rangle} + W_{fx}\cdot x^{\langle t\rangle} + b_f\right)\\
\Gamma_o &= \sigma\left(W_{oa}\cdot a^{\langle t-1\rangle} + W_{ox}\cdot x^{\langle t\rangle} + b_o\right)\\
c^{\langle t\rangle} &= \Gamma_u\cdot \tilde{c}^{\langle t\rangle} + \Gamma_f\cdot c^{\langle t\rangle}\\
a^{\langle t\rangle} &= \Gamma_o\cdot c^{\langle t\rangle}
\end{aligned}
$$

El vector de entrada en el tiempo t se nota $x^{\langle t\rangle}$, mientras que W_{cx}, W_{ux}, W_{fx} y W_{ox} son matrices de pesos rectangulares, que son aplicadas a las entradas de la RNN. W_{ca}, W_{ua}, W_{fa} y W_{oa} son matrices cuadradas que definen los pesos de las conexiones recurrentes, mientras que b_c, b_u, b_f y b_o son vectores de sesgo, la función σ es la función de activación sigmoidea, y tanh es la función de activación tangente hiperbólica. Finalmente $(*)$ es la multiplicación Hadamard entre dos matrices.

Cada puerta tiene una función en específico. La puerta de olvido o *forget gate* Γ_f decide que información puede ser descartada del estado de celda anterior $c^{\langle t-1\rangle}$. La puerta de actualización o

update gate Γ_u opera el estado anterior $c^{\langle t-1\rangle}$, después de ser modificado por Γ_f, y decide que tanto el nuevo estado $c^{\langle t\rangle}$ debe ser actualizado con un candidato $\tilde{c}^{\langle t\rangle}$. La puerta de salida o *output gate* selecciona la parte del estado $c^{\langle t\rangle}$ que se devolverá como salida. Cada puerta depende de la entrada actual y de la salida de la celda previa $a^{\langle t\rangle}$. En la figura 7 vemos como se ve una celda LSTM y sus respectivas ecuaciones (Filippo Maria Bianchi, 2017, pag 25).

3.6. Unidades recurrentes cerradas (GRU)

Gated Recurrent Unit (GRU) es otra arquitectura muy conocida, que captura dependencias en diferentes escalas de tiempo, en GRU las puertas de entrada y de olvido se combinan en una sola puerta de actualización la cual controla que tanto cada unidad oculta puede recordar u olvidar. El estado interno en una celda GRU siempre está expuesto en la salida, debido a la falta de un mecanismo de control, como la puerta de salida en una celda LSTM.

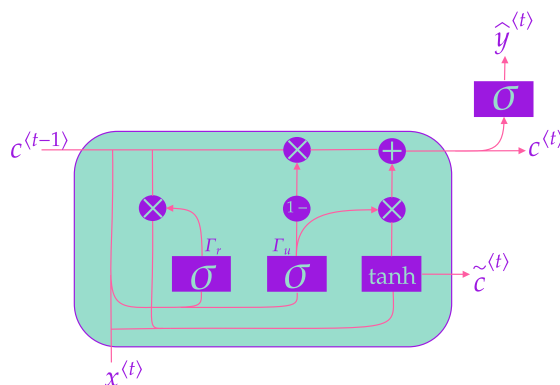


Figura 8: Celda GRU

$$
\begin{aligned}
\tilde{c}^{\langle t\rangle} &= \tanh\left(W_{cc}\cdot c^{\langle t-1\rangle} + W_{cx}\cdot x^{\langle t\rangle} + b_c\right)\\
\Gamma_u &= \sigma\left(W_{uc}\cdot c^{\langle t-1\rangle} + W_{ux}\cdot x^{\langle t\rangle} + b_u\right)\\
\Gamma_r &= \sigma\left(W_{rc}\cdot c^{\langle t-1\rangle} + W_{rx}\cdot x^{\langle t\rangle} + b_r\right)\\
c^{\langle t\rangle} &= \Gamma_u\cdot\tilde{c}^{\langle t\rangle} + (1-\Gamma_u)\cdot c^{\langle t-1\rangle}
\end{aligned}
$$

GRU también hace uso de puertas. La primera es la puerta de actualización o *update gate* Γ_u, la cual controla qué tanto se debe actualizar el contenido actual de la celda con el nuevo estado candidato $\tilde{c}^{\langle t\rangle}$. La segunda es la puerta de reinicio o *reset gate* que si tiene un valor cercano a 0, puede reiniciar efectivamente la memoria de la celda y hacer que la unidad actúe como si la siguiente entrada fuera la primera de la sucesión. La celda GRU y sus ecuaciones se muestran en la Figura 8.

En este caso, la función tanh es la función de activación tangente hiperbólica y σ es la función de activación sigmoidea. En una celda GRU, el número de parámetros es mayor que en una RNN simple, pero menor que en una celda LSTM (Filippo Maria Bianchi, 2017, pag 27).

4. Resultados simulación

Se hizo un estudio de los resultados del artículo et al. (2021a) utilizando los datos de Brasil, aunque se obtuvieron curvas parecidas a las dadas en él , debido a que no se dice explícitamente la semilla[3] que se utilizó, y por la cantidad de posibles condiciones iniciales, los resultados no son exactamente iguales.

En el artículo emplean la paquetería PyTorch, de python para la construcción de redes neuronales, y las simulaciones fueron realizadas en Google COLAB. Para encontrar la mejor RNN, utilizaron diferentes hiperparametros, tales como:

- **Capas ocultas:** $1, 2, 3, 4$
- **Número de neuronas en las capas ocultas:** $10, 100, 200, 300$
- **Taza de aprendizaje:** 0.1, 0.01, 0.001, 0.0001, 0.00001

Para optimizar la red neuronal usaron Adam, como función de perdida se utilizó la MSE y RMSE. Se normalizaron los datos con la función MinMaxScaler. Se dividió el conjunto de datos entre entrenamiento y testeo. El conjunto de testeo se conformó de los últimos 14 datos. El conjunto de entrenamiento es dividido en diferentes secuencias, llamadas ventanas, cada uno de longitud 30. Cada secuencia alimenta la red neuronal, por ejemplo, se alimenta la red con la información del primer día y del día 30, y se predice el día 31, el paso siguiente será alimentar la red con los datos del segundo día, hasta el día 31 y se predice el día 32. La simulación se realizó con 5000 epochs [4]. Este proceso se ejecutó por diferentes países, tales como Estados Unidos, Brasil, India, Rusia, Sur África, México, Perú, Chile, Reino Unido e Irán.

En el estudio realizado al caso de Brasil, el conjunto de datos va desde el $2020 - 03 - 06$ hasta el $2020 - 08 - 4$ y se crea un pronóstico de 60 días a partir de esta última fecha. Se construyen redes LSTM y GRU, mientras que las RNN simples no son tomadas en cuenta. Los datos son extraídos de Jonh Hopkins Coronavirus Resource Center Johns Hopkins (2022).

4.1. Metodología Simulación.

Los datos fueron extraídos de la página de datos abiertos de Colombia Nacional (2022). El código fue escrito con el paquete Tensorflow de Python y las simulaciones fueron realizadas en equipos propios. Se optimizó el número de neuronas en cada capa junto con la taza de aprendizaje, se utilizó un optimizador Bayesiano para esto. El rango de número de neuronas por cada capa va desde 32 hasta 512 con saltos 32 y las tazas de aprendizaje fueron $0.01, 0.001$ y 0.0001 y aplicando

[3] Números aleatorios con los cuales empieza el modelo de redes neuronales
[4] El número de veces que nuestro conjunto de datos pasa por la red neuronal.

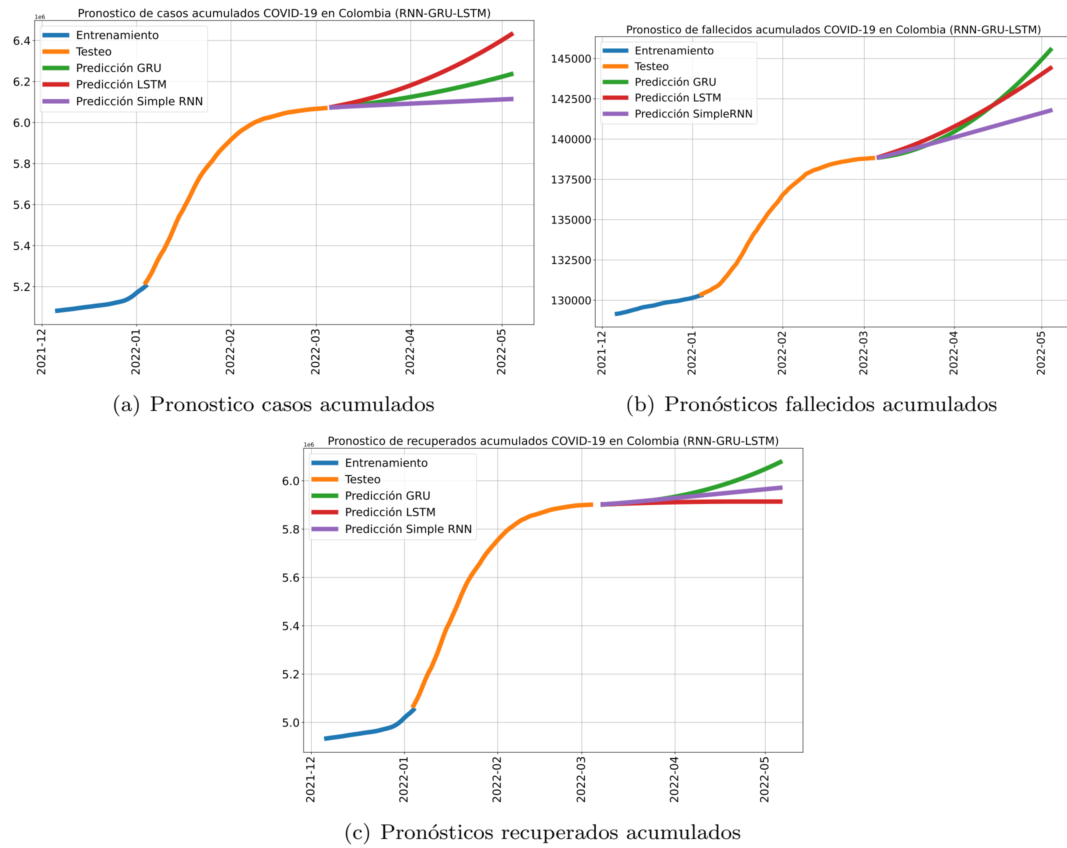
un decaimiento en la respectiva taza a partir del quinto epoch. El optimizador Adam es usado para actualizar de manera iterativa los pesos de las redes junto con MSE Y RMSE. Los datos fueron estandarizados empleando (17)

$$x_n = \frac{x - x_{min}}{x_{max} - x_{min}}, \tag{17}$$

donde x, x_{min} y x_{max} representan el dato original, el mínimo y el máximo de la serie de tiempo, respectivamente. El conjunto de datos se divide en dos subconjuntos, datos de entrenamiento (período comprendido entre las fechas $2020-03-06$ y $2022-01-06$), datos de testeo (entre $2022-01-07$ y $2022-03-07$). El último conjunto a través del cual queremos observar el comportamiento de las 60 predicciones estará comprendido entre $2022 - 03 - 08$ y $2022 - 05 - 06$. Los datos de entrenamiento y testeo fueron divididos en diferentes secuencias llamadas ventanas, cada una de longitud 60 (Brownlee, 2018, pág 24). Estas secuencias son alimentadas a la red para hacer el pronóstico, por ejemplo los dato del 1 al 60 se utilizan para predecir el día 61, los datos del 2 al 61 se utilizan para predecir el día 62. La simulación utilizó 100 epochs y el mejor epoch fue utilizada para realizar el pronóstico final.

4.2. Resultados y discusión.

En las siguientes gráficas se muestran los resultados obtenidos Buitrago (2022), mirando los últimos 30 días del conjunto de entrenamiento, el conjunto de testeo y el pronóstico realizado.



(a) Pronostico casos acumulados

(b) Pronósticos fallecidos acumulados

(c) Pronósticos recuperados acumulados

Figura 9: Resultados de la simulación

En cada una de las gráficas observamos tres curvas asociadas a la predicción dada por cada modelo. En la Figura 9(a), el modelo relacionado con la Simple RNN, indica que para el $2022 - 05 - 04$, el número de casos acumulados será de $6.114741e + 06$, mientras que el modelo de LSTM con $6.430160e + 06$ y por ultimo el modelo GRU con $6.236141e + 06$. En la Figura 9(b), tenemos la gráfica asociada a los fallecidos acumulados en Colombia, acá el modelo de Simple RNN indica que para el $2022 - 05 - 04$, el número de fallecidos acumulados será de 141775.420527, mientras que el modelo de LSTM indica 144408.791514 fallecidos acumulados, y GRU indica 145538.986489. En la Figura 9(c), referentes a los recuperados acumulados, tenemos que el modelo de Simple RNN indica que para el $2022 - 05 - 04$, el número de recuperados acumulados será de $5.971182e + 06$, mientras que el modelo LSTM indica que habrán $5.914043e + 06$ recuperados acumulados y GRU $6.077595e + 06$ recuperados acumulados.

Casos acumulados						
Modelo	**Epochs**	**Número de capas**	**Número de neuronas**	**Taza de aprendizaje**	**MSE**	**RMSE**
RNN	10	2	416	0.0001	25448.21	159.52
LSTM	8	2	50	0.01	384597.61	620.15
GRU	16	2	32	0.01	340606.64	583.61
Recuperados acumulados						
RNN	8	2	224	0.0001	427586.48	653.90
LSTM	21	2	50	0.0001	24523.67	156.60
GRU	16	2	96	0.001	31204.44	176.64
Fallecidos acumulados						
RNN	7	1	348	0.001	16578.64	128.75
LTSM	100	2	50	0.001	968.82	31.12
GRU	8	2	32	0.001	5780.41	76.02

Tabla 1: Mejores hiperparametros.

En la Tabla 1, se presentan los hiperparametros con los cuales cada modelo de RNN da los mejores resultados en terminos del menor RMSE, estos fueron optenidos utilizando un optimizador Bayesiano de la paqueria Keras de python.

5. Conclusiones

En el presente estudio se hizo un pronóstico de la pandemia de COVID-19 en Colombia usando RNN simple, RNN con celdas LSTM y RNN con celdas GRU.

Para los casos acumulados en Colombia, el modelo con menor RMSE fue la RNN simple, aunque el número de neuronas en sus capas ocultas fue muchísimo mayor con respecto a los otros dos modelos, que presentan también, muy buenos resultados.

Para los recuperados acumulados en Colombia, el mejor modelo fue RNN con celdas LSTM, este además presenta el menor número de neuronas en sus capas ocultas, lo cual es una ventaja computacional.

Para los fallecidos acumulados en Colombia, el mejor modelo es la RNN con celdas LSTM.

En general, observamos que las RNN simple, utilizan más neuronas para realizar el pronóstico, y esto puede representar un costo computacional mayor.

Recibido:
Aceptado:

Referencias

J. Brownlee. *Deep Learning for Time Series Forecasting.* Machine Learning Mastery, fourth edition, 2018.

Y. Buitrago. Pronóstico del COVID-19 en Colombia. `https://github.com/YeisonABL/Pronostico_Colombia_COVID19`, 2022. [En línea; acceso 6/marzo/2022].

U. o. T. Computer Science. Lecture 15: Exploding and Vanishing Gradients . `https://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf`, 2022. [En línea; acceso Enero/2022].

I. et al. Comparative analysis and forecasting of covid-19 cases in various european countries with arima, narnn and lstm approaches. *ELSEVIER*, 2020.

K. A. et al. Forecasting of covid-19 using deep layer recurrent neural networks (rnns) with gated recurrent units (grus) and long short-term memory (lstm) cell. *ELSEVIER*, mar 2021a.

R. K. et al. Covid-19 in iran: Forecasting pandemic using deep learning. *Hindawi*, 2021b.

M. C. K. A. R. R. J. Filippo Maria Bianchi, Enrico Maiorino. *Recurrent Neural Networks for Short-Term Load Forecasting.* Springer, first edition, 2017.

U. . M. Johns Hopkins. Coronavirus Resource Center . `https://coronavirus.jhu.edu/`, 2022. [En línea; acceso Enero/2022].

M. S. Ke-Lin Du. *Neural Networks and Statistical Learning.* Springer, 2019.

MIT. MIT 6.S191: Recurrent Neural Networks and Transformers. `https://youtu.be/QvkQ1B3FBqA`, 2022a. [En línea; acceso Enero/2022].

MIT. Mit 6.s191: Recurrent neural networks and transformers, mar 2022b. URL `https://www.youtube.com/watch?v=QvkQ1B3FBqA\&t=2101s\&ab_channel=AlexanderAmini`. Youtube.

G. Nacional. Datos Abiertos de Colombia. `https://www.datos.gov.co/`, 2022. [En línea; acceso 6/marzo/2022].

S. rekja Hanumanthu. Role of intelligent computing in covid-19 prognosis: A state-of-the-art review. *Pre-proof*, 2020.

Y. Tamura. Simple RNN: the first foothold for understanding LSTM. `https://data-science-blog.com/blog/2020/06/17/simple-rnn-the-first-foothold-for-understanding-lstm/`, 2020. [En línea; acceso 6/marzo/2022].

A. M. S. H.-G. Zimmermann. Recurrent neural networks are universal approximators. *International Journal of Neural Systems*, 17, 2007.