Doble Grado en
Matemáticas y Estadística

TRABAJO FIN DE GRADO

REDES RECURRENTES

Jesús Pérez Guerrero

Sevilla, Junio de 2020

Tutorizado por: Rafael Pino Mejías

Índice general

Resumen

En los últimos años, el ser humano se ha ido fascinando con la idea de un "futuro" robotizado, donde las máquinas fuesen capaces de pensar como un cerebro humano. Pues bien, ese futuro se ha convertido en presente, y ya, a día de hoy, hay numerosos estudios que se dedican a intentar imitar el comportamiento del cerebro con patrones matemáticos.

Se ha replicado el comportamiento de nuestro sistema de neuronas en distintos modelos matemáticos, conocidos todos, en su conjunto, como Red de Neuronas Artificiales. Dentro de los muchos tipos de esquemas que se han desarrollado en este ámbito, con esterabajo se pondrá de manifiesto la evolución y el funcionamiento del modelo de Redes Neuronales Recurrentes (Neural Network Recurrent, RNN). que es muy utilizado en tareas como el "reconocimiento de voz", siendo esto muy utilizado en multitud de dispositivos de uso común en la actualidad.

Este trabajo consta de 4 capítulos.

En el primero de ellos nos adentramos en los términos de Inteligencia Artificial (Artificial Intelligence, IA), Aprendizaje Automático (Machine Leargnin, ML) y Aprendizaje Profundo (Deep Learning, DL).

Posteriormente se comenta el comportamiento del modelo de Red de Neuronas Artificial (RNA), imitando en su mayor medida, a la actividad neurológica de nuestro cerebro.

En el siguiente capítulo se habla del tema principal de nuestro trabajo, el modelo de RNN, explicando así, su evolución y funcionamiento.

Finalmente ejemplificaremos cómo usar un modelo de RNN en un caso de la vida cotidiana y se hará una breve mención de cómo pueden ayudarnos los modelos de redes neuronales a combatir contra el virus COVID-19.

Abstract

In recent years, human beings have been excited about a robotized future. We are excited about the idea that machines think like us. Well, that future has already become present. Today, scientists are developing mathematical models which mimic the behavior of the human brain.

Thus, scientists have developed models of artificial neural networks. We are going to focus on Neural Network Recurrent (RNN). This is a specific artificial neural network model. This is a specific artificial neural network model used for "voice recognition".

This work consists of 4 chapters. In the first one we will introduce to Deep Learning (DL) and its comparison with Artificial Intelligence (AI) and Machine Learning (ML).

The second chapter will talk about Artificial Neural Networks, making a special mention about how it tries to emulate the behavior of the brain.

The next chapter is the main theme of our work. We will explain the functioning and evolution of the recurrent neural network.

Finally, we will explain examples of how to use an RNN model in real life. We will mention some studies of how neural networks can help us with the COVID-19 virus.

Índice de figuras

Capítulo 1

Introducción al Deep Learning

1.1. ¿Qué es el Deep Learning?

En la época moderna, lo que se conoce como IA o inteligencia artificial es casi una realidad, así en cuanto a la automatización de procesos productivos de bienes y servicios, siendo ya muy popular y creíble la perspectiva de un futuro robotizado, en el que los robots realicen incluso las labores domésticas y las personas se ocupen sólo de tareas directivas, sociales y/o específicamente humanas.

Merece, por tanto, preguntarse, con esa premisa, qué es el aprendizaje profundo, el aprendizaje automático y la llamada inteligencia artificial.

1.2. Inteligencia Artificial, Aprendizaje Automático y Aprendizaje Profundo

Se intentará ahora la expresión y relación de y entre los conceptos que se vienen mencionando.

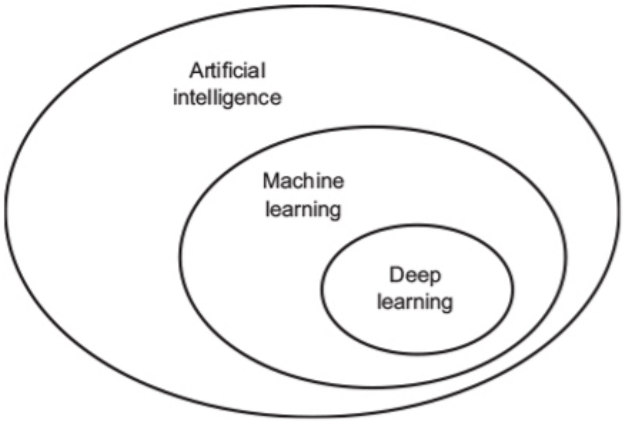


Figura 1.1: Inteligencia artificial, aprendizaje automático, y aprendizaje profundo

1.2.1. Inteligencia Artificial

El término nace allá por 1950, al formularse la pregunta de si las computadoras eran capaces de pensar por sí mismas, lo que aún se sigue planteando.

Por IA puede entenderse la automatización de tareas intelectuales humanas.

Abarca tanto el aprendizaje automático como el profundo, y otros varios campos adicionales al margen de la idea de aprendizaje.

Los primeros programas de ajedrez sólo involucraban una serie de reglas codificadas, no pudiendo clasificarse como aprendizaje automático. Se creía que la IA avanzaría hasta el nivel humano mediante la elaboración de un conjunto suficientemente grande de reglas explícitas de conocimientos. Es lo que se conoce como IA simbólica, que fue objeto de estudio entre las décadas de 1950 y 1980.

Esa técnica demostró ser adecuada para problemas lógicos bien definidos, como jugar al ajedrez, pero no así, por la dificultad de elaborar reglas explícitas, respecto de tareas más complejas, como la clasificación de imágenes, el reconocimiento de voz o la traducción de idiomas. Ante lo cual surgió, como nuevo enfoque, la idea del aprendizaje automático.

1.2.2. Aprendizaje Automático

En la Inglaterra victoriana, Charles Babbage inventó el motor analítico, basado en operaciones mecánicas, automatizaba ciertos cálculos del análisis matemático, y de ahí su denominación. Pero el concepto de computación mecánica aún no se había inventado. Lady Ada Lovelace, amiga y colaboradora de Charles, expresó en 1843 sobre su invención: «... el motor analítico no tiene pretensiones de originar nada. Puede hacer lo que sepamos como ordenar que se ejecute ... su función es ayudarnos a poner a disposición lo que ya conocemos ...». Alan Turing, otro pionero de la IA, reflexionó también, citando a Lady Ada, sobre si las computadoras (lo que entonces se entendía por ellas) podrían aprender hasta el punto de la originalidad, y concluyó en sentido afirmativo.

El aprendizaje automático es lo que inducen las siguientes preguntas: ¿podría una computadora ir más allá de lo que se le ordena realizar, aprendiendo por sí misma para llevar a cabo una tarea específica?, ¿podría sorprendernos, creando reglas de procesamiento a partir de los datos suministrados? En la programación clásica, las personas ingresan las reglas y los datos que se procesarán según las mismas, obteniéndose de ello las respuestas (véase figura 1.2). En el aprendizaje automático, los humanos ingresan datos, las consecuencias esperadas de los mismos, y de ahí se extraen las reglas, que se pueden aplicar a los datos nuevos para dar lugar a respuestas originales.

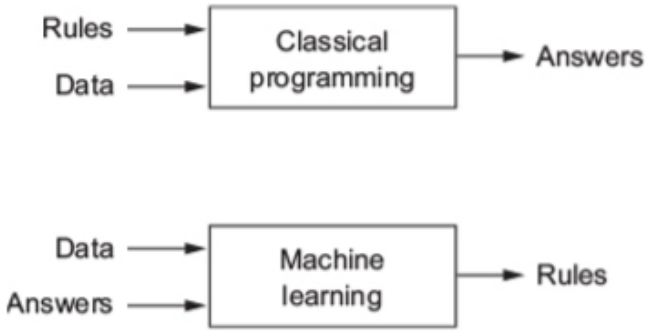


Figura 1.2: Aprendizaje automático: un nuevo paradigma de programación

Un sistema de aprendizaje automático se entrena en lugar de programarse. Se presenta con ejemplos relevantes para una tarea, hallando una estructura estadística en ellos que eventualmente permite sacar reglas para automatizar esa tarea. Ese aprendizaje, que empezó a florecer como idea en la década de 1990, se ha convertido hoy en el subcampo más popular y exitoso de IA. Está muy relacionado con las estadísticas matemáticas, pero con diferencias sensibles, así, por ejemplo, tiende a tratar con grandes y complejos conjuntos de datos, para los cuales el análisis estadístico clásico, como el bayesiano, no resulta práctico. Y es también que el aprendizaje automático, y más aún el profundo, implican poca teoría matemática, estando orientados a la ingeniería.

1.2.3. Aprendizaje Profundo

Para definir el aprendizaje profundo y distinguirlo de otros enfoques de aprendizaje automático, hay que reparar en el funcionamiento de los algoritmos destinados a hallar esas reglas de procesamiento de datos, y más en concreto en lo siguiente:

- Puntos que constituyen datos de entrada; si, por ejemplo, la tarea consiste en el reconocimiento de voz, podrían ser archivos de sonido (de personas hablando).
- Resultados de la salida esperada; en la referida tarea de reconocimiento podrían ser transcripciones realizadas de archivos sonoros.
- Forma de medir si el algoritmo realiza bien su menester; por la distancia entre su salida actual y la esperada, lo que sirve de señal de retroalimentación para ajustar el mecanismo de funcionamiento, paso de ajuste que es lo que llamamos aprendizaje.

Un modelo de aprendizaje automático transforma sus datos de entrada en salidas significativas, extrayéndolo de los ejemplos proporcionados. De modo que la cuestión básica del aprendizaje, automático y profundo, es la de la transformación significativa de los datos, tomando representaciones útiles de entrada para acercar a la salida esperada.

Los algoritmos de aprendizaje automático no son creativos en dichas transformaciones, sino que buscan en un conjunto predefinido de operaciones o posibilidades, llamado espacio de hipótesis, utilizando la guía de una señal de retroalimentación.

En base a lo anterior, ¿qué entendemos por aprendizaje profundo?

Es un subcampo específico del aprendizaje automático, una nueva toma de representaciones sobre capas sucesivas cada vez más significativas, cuya cantidad supone lo que se conoce como profundidad del modelo de datos.

El aprendizaje profundo hoy suele implicar decenas o incluso cientos de capas sucesivas de representaciones.

Cuando el aprendizaje automático se centra en una o dos capas de representaciones de datos, se le llama aprendizaje superficial.

En el aprendizaje profundo, las referidas representaciones en capas se consiguen casi siempre a través de modelos llamados redes de neuronas, con capas literalmente apiladas una sobre la otra.

El término red de neuronas alude a la neurobiología. Pero los modelos del aprendizaje profundo no son los del cerebro humano.

Sobre las representaciones de un algoritmo de aprendizaje profundo, veamos cómo una red de varias capas de profundidad (figura 1.3) transforma una imagen para la clasificación de dígitos.

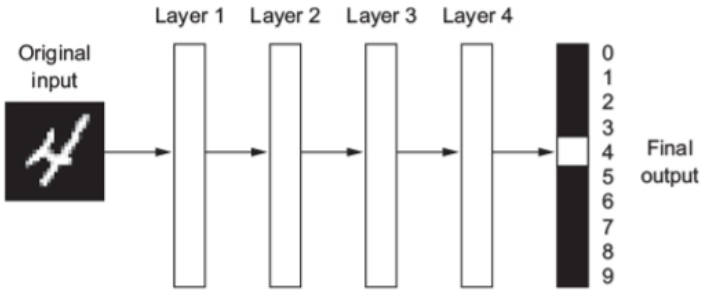


Figura 1.3: Una red de neuronas profunda para la clasificación de dígitos

En la siguiente figura (figura 1.4), la red transforma la imagen digital en representaciones progresivamente diferentes de la imagen original, cada vez con mayor información respecto al resultado final. Se puede pensar que una red profunda actúa como un proceso de destilación de información por etapas, pasando a través de filtros sucesivos para salir purificada o más útil en cuanto a alguna tarea.

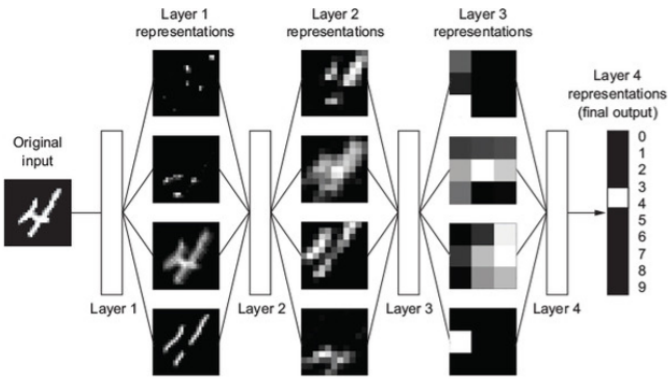


Figura 1.4: Representaciones profundas por un modelo de clasificación de dígitos

En eso consiste el aprendizaje profundo, varias etapas de filtros depurativos de la información para alcanzar la representación de datos.

Capítulo 2

Redes de Neuronas

2.1. Antecedentes generales

El cerebro humano cuenta con más de 10 billones de células neuronales, que tienen interconexiones complicadas, constituyendo una red de procesamiento de señal y memoria a gran escala.

La célula nerviosa es conocida como neurona y es considerada la unidad funcional del sistema nervioso.

A continuación, se describe gráficamente la estructura de una neurona:

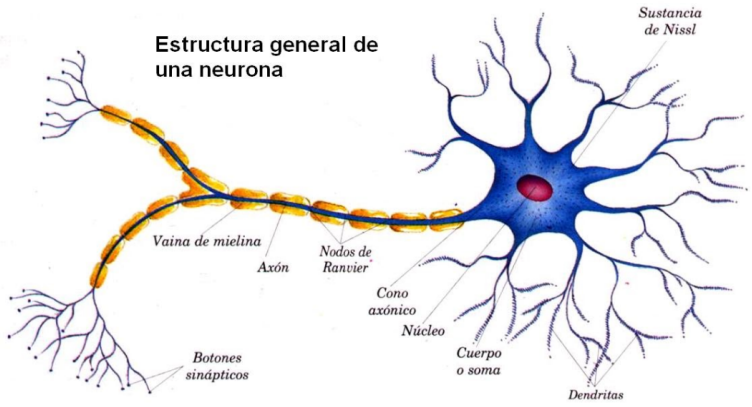


Figura 2.1: Estructura general de una neurona

Así pues, una red de neuronas es una versión simplificada del sistema nervioso de un ser vivo, diseñada para la realización de tareas. Se compone de unidades de procesamiento interconectadas, las neuronas, cada una de las cuáles recibe como entrada un conjunto de señales, discretas o continuas, para ponderarlas e integrarlas, transmitiendo el resultado a las compañeras conectadas a ella.

La conexión entre dos neuronas tiene una determinada importancia a la que se llama peso. En los pesos se halla la mayor parte del conocimiento que la red de neuronas guarda sobre una tarea en particular. Mediante un proceso de aprendizaje o entrenamiento,

esos pesos se ajustan para el objetivo perseguido. Dicho ajuste es el modo principal de aprendizaje de las redes de neuronas.

Un modelo neuronal simple es el representado en la siguiente figura:

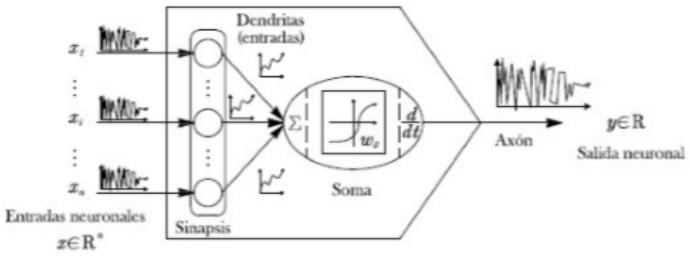


Figura 2.2: Modelo neuronal simple como un procesador con múltiples entradas (dendritas) y una sola salida (axón)

Las funciones de procesamiento de este sistema neuronal pueden ser divididas en las siguientes cuatro categorías:

1. **Dendritas**: Consisten en un árbol de fibras ramificadas actuando como puntos de entrada para el cuerpo principal de la neurona.
2. **Sinapsis**: Es un área de almacenamiento de la experiencia pasada. Proporciona la memoria a largo plazo a la experiencia acumulada pasada.
3. **Soma**: Es el cuerpo celular de la neurona. Recibe información sináptica y realiza un procesamiento más de la información. Suele llevar a cabo casi todas las funciones lógicas de la neurona.
4. **Axón**: Supone la línea de salida neuronal. La salida aparece en la forma de un potencial de acción que es transmitida a otras neuronas para un procesamiento más.

El artículo de McCulloch y Pitts (1943) viene considerándose el punto de partida en la investigación de redes de neuronas. Ambos investigadores propusieron un modelo simplificado de la actividad nerviosa real en el que cada neurona de una red de neuronas podía activarse o desactivarse en función de lo que hicieran las neuronas conectadas a ella. Dado que una neurona sólo podía estar activada o desactivada, la capacidad computacional de la red completa estaba definida en términos del conjunto de predicados lógicos que era capaz de procesar. Ya se habló entonces de redes de neuronas recurrentes, denominadas redes con ciclos, el objeto principal de este trabajo.

2.2. Red de Neuronas Artificial

Una Red de Neuronas Artificial (RNA) es un modelo matemático de procesamiento, distribuido paralelamente, en forma de grafo dirigido, simulando el comportamiento biológico de las neuronas y la estructura del cerebro. Tiene estas propiedades:

- **Representación de relaciones de entrada/salida**: Cuando se dispone de un conjunto de muestras de la relación entrada/salida a modelar se puede utilizar algún

algoritmo de aprendizaje supervisado, para optimizar los pesos de la red de neuronas componiendo adecuadamente esa relación por compleja que sea.

- **Capacidad de generalización**: Una vez entrenada la red de neuronas, se pueden presentar a la misma datos distintos a los usados durante el proceso de aprendizaje.
- **No linealidad**: Las redes de neuronas modelan generalmente procesos no lineales, aunque también pueden respecto a sistemas lineales.
- **Adaptabilidad**: Las redes de neuronas son capaces de reajustar sus pesos para adaptarse a cambios en el entorno. Resulta especialmente útil cuando el entorno que suministra los datos de entrada es no estacionario, desvariando con el tiempo algunas de sus propiedades.
- **Tolerancia ante fallos**: Una red de neuronas es tolerante en ese sentido cuando los posibles fallos de operaciones en partes de la red sólo afectan débilmente al rendimiento de la misma.
- **Procesamiento Paralelo**: Como las neuronas físicas, que trabajan en paralelo, es inherente a las redes de neuronas artificiales ese modo de funcionar.

2.3. Modelo de una Red de Neuronas Artificial

Dentro de una red de neuronas, los elementos de procesamiento se encuentran agrupados por capas, cada una de las cuales es una colección de neuronas. Según su ubicación en la red artificial, la capa recibe distintas denominaciones.

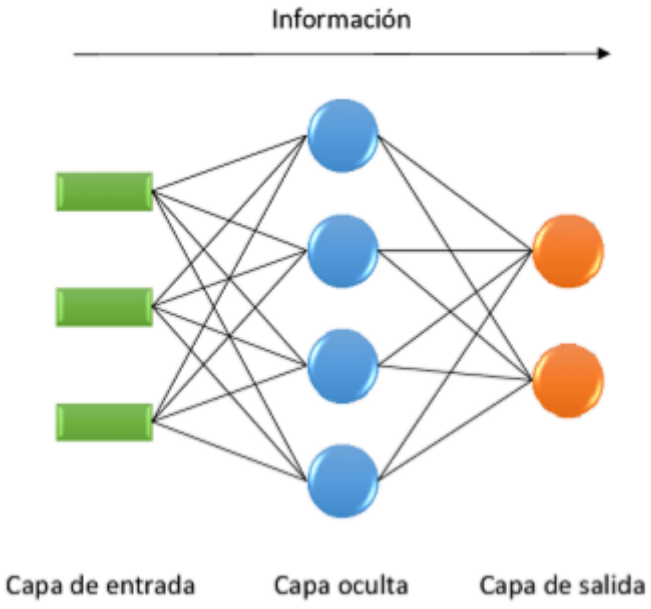


Figura 2.3: Red de Neuronas Artificial

- **Capa de entrada**: Recibe las señales de la entrada de la red.
- **Capas ocultas**: Estas capas son aquellas que no tienen contacto con el medio exterior, sus elementos pueden tener diferentes conexiones y son éstas las que determinan las diferentes topologías de la red.

- **Capa de salida**: Recibe la información de la última capa oculta y transmite la respuesta al medio externo.

En el modelo más habitual de neurona se identifican cinco elementos básicos para la j-ésima neurona de una red de tiempo discreto:

- Un conjunto de m señales de entrada, $x_i(t), i = 1, ..., m$, que suministran a la neurona los datos del entorno. Estos datos pueden ser externos a la red de neuronas, pertenecientes a la salida de otras neuronas de la red o correspondientes a la salida anterior de la propia neurona.
- Un conjunto de sinapsis, caracterizada cada una por un peso sináptico propio $w_{ij}, i = 1, ..., m$. El peso w_{ij} está asociado a la sinapsis que conecta la unidad i-ésima con la neurona j-ésima.
- Un sesgo o umbral **u** cuya presencia aumenta la capacidad de procesamiento de la neurona y que eleva o reduce la entrada a la neurona, según sea su valor positivo o negativo. Se encarga de transmitir la respuesta al medio externo.
- Un sumador o integrador **s** que suma las señales de entrada, ponderadas con sus respectivos pesos, y el sesgo: $s(x_1, ..., x_m, w_1, ..., w_m) = \sum_{i=1}^{m} w_i x_i$.
- **Función de Activación g**, que suele limitar la amplitud de la salida de la neurona. La función de activación es la que define la salida de la neurona.

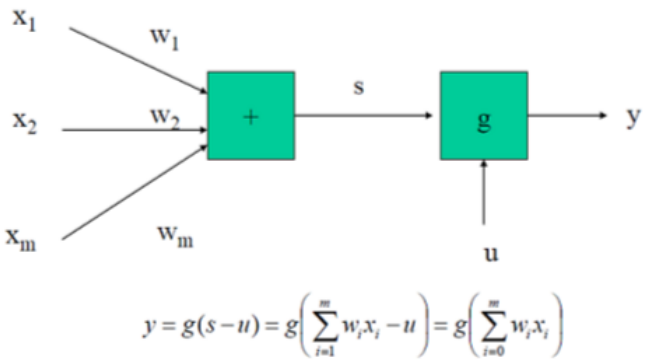


Figura 2.4: Red de Neuronas Artificial

Las funciones de activación más utilizadas habitualmente son las siguientes:

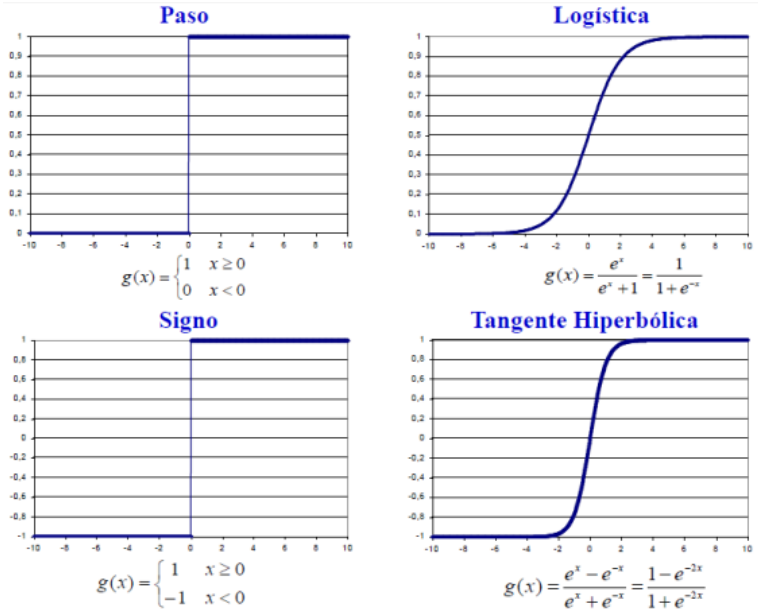


Figura 2.5: Funciones de Activación

2.4. Modelos de Aprendizaje Profundo

Los principales modelos de aprendizaje profundo son:

- **Perceptrón de una sola capa (SLP)**

  La arquitectura más simple, aquella en la que una capa de entrada se relaciona directamente con una capa de salida a través de una red ponderada (sin capas ocultas). Es un modelo denominado perceptrón de una sola capa (SLP), el tipo más elemental de redes de neuronas artificiales, bastante fácil de implementar en la práctica. Sólo es capaz de clasificar casos separables linealmente con un objetivo binario (1,0). Se muestra un ejemplo (figura 1.5):

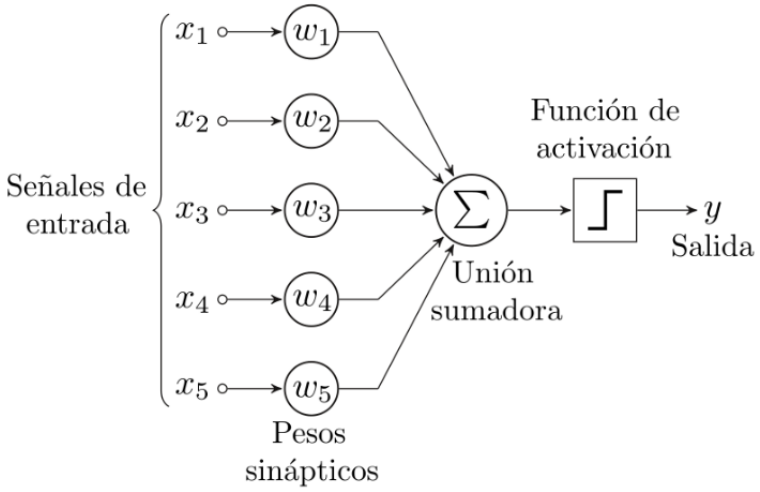


Figura 2.6: Perceptrón de una sola capa(SLP)

Consideremos una red con $x \in \mathbb{R}^m$ inputs, e $y \in \mathbb{R}^n$ outputs. Las salidas se

generan mediante una suma ponderada de los nodos de entrada, más un nodo de polarización, que luego se alimenta (o activa) a través de una función no lineal $f(\cdot)$, como se muestra a continuación (para un ejemplo de salida y_i):

$$h_i = \sum_j w_{i,j} x_j + b \qquad (2.1)$$

$$y_i = f(h_i) \qquad (2.2)$$

Expresión que podemos reescribir de forma más compacta usando el álgebra lineal:

$$H = WX \quad W \in \mathbb{R}^{(m+1)\cdot n} \qquad (2.3)$$

El diseñador de la red tiene libertad para elegir qué funciones no lineales usar, aunque hay situaciones en las que se prefiere una sobre las otras. Por ejemplo, para las tareas de clasificación, la capa de salida debe usar una función softmax, tanh o logística para vincular los valores entre 0 y 1, mientras que para una tarea de regresión, las salidas no tienen límites y se debe usar una función lineal.

- **Perceptrón multicapa (MLP)**

Un perceptrón multicapa (MLP) tiene la misma estructura que un perceptrón de una sola capa con una o más capas ocultas. Uno de los factores distintivos clave en este modelo y en el modelo de perceptrón de una sola capa es el algoritmo de retropropagación. El algoritmo de retropropagación consta de dos fases: la fase de avance donde las activaciones se propagan desde la capa de entrada a la de salida, y la fase de retroceso, donde el error entre el valor real observado y el valor nominal solicitado en la capa de salida se propaga hacia atrás en orden para modificar los pesos y los valores de sesgo.

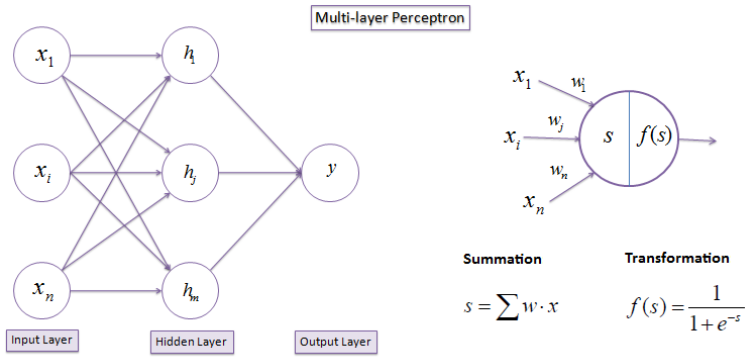


Figura 2.7: Perceptrón multicapa(MLP)

En la propagación hacia atrás se propaga los errores hacia atrás al asignarlos a cada unidad de acuerdo con la cantidad de este error del que la unidad es responsable.

1. Error en cualquier neurona de salida:

$$d_o = y \cdot (1 - y) \cdot (t - y) \qquad (2.4)$$

2. Error en cualquier neurona oculta:

$$d_i = y_i \cdot (1 - y_i) \cdot (w_i \cdot d_o) \tag{2.5}$$

3. Cambio de los pesos:

$$\Delta w = \eta \cdot d \cdot x \tag{2.6}$$

- **Redes de Neuronas Convolucionales (CNNs)**

  Las Redes de Neuronas Convolucionales (CNNs), o en inglés, Convolutional Neural Network (CNN) son modelos que se utilizan con mayor frecuencia para el procesamiento de imágenes y la visión por computadora. Están diseñados de tal manera que imitan la estructura de la corteza visual animal. Específicamente, las CNNs tienen neuronas dispuestas en tres dimensiones: ancho, alto y profundidad. Las neuronas en una capa dada solo están conectadas a una pequeña región de la capa anterior.

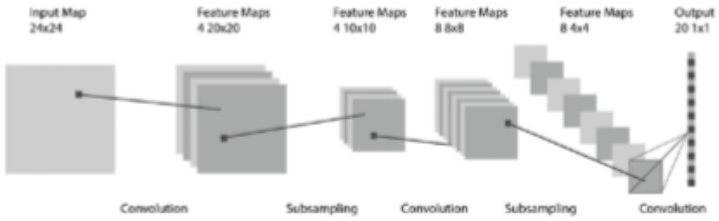


Figura 2.8: Red de Neuronas Convolucionales (CNN)

- **Redes de Neuronas Recurrentes (RNNs)**

  Las Redes de Neuronas Recurrentes (RNRs), o en inglés, Recurrent Neural Networks (RNN) son modelos de redes de neuronas artificiales (RNAs) donde las conexiones entre unidades forman un ciclo dirigido. Específicamente, un ciclo dirigido es una secuencia en la que la caminata a lo largo de los vértices y bordes está completamente determinada por el conjunto de bordes utilizados y, por lo tanto, tiene una apariencia de un orden específico. Las redes de neuronas recurrentes a menudo se usan específicamente para el reconocimiento de voz y escritura.

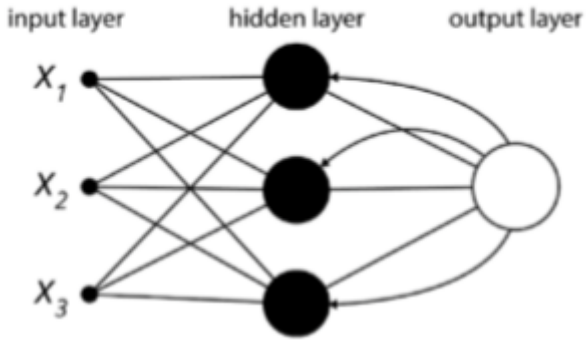


Figura 2.9: Red de Neuronas Recurrente (RNN)

- **Máquinas de Boltzmann Restringidas (RBMs)**

  Las Máquinas de Boltzmann Restringidas, o en inglés, Restricted Boltzmann Machines (RBMs) son un tipo de modelo binario de Markov que tiene una arquitectura única, de modo que hay múltiples capas de variables aleatorias ocultas y una red de unidades binarias estocásticas acopladas simétricamente. Los modelos RBM se componen de un conjunto de unidades visibles y una serie de capas de unidades ocultas. Sin embargo, no hay conexiones entre unidades de la misma capa. Los modelos RBM pueden aprender representaciones internas complejas y abstractas en tareas como el reconocimiento de objetos o de voz.

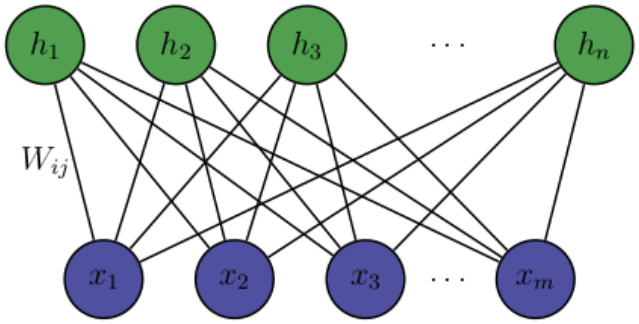


Figura 2.10: Máquina de Boltzmann Restringida (RBM)

- **Redes de Creencias Profundas (DBNs)**

  Las Redes de Creencias Profundas, o en inglés, Deep Belief Network (DBN) son similares a las RBM, excepto que la capa oculta de cada subred es la capa visible para la siguiente subred. Los modelos DBN son en general un modelo gráfico generativo compuesto por múltiples capas de variables latentes con conexiones entre las capas, pero no entre las unidades de cada capa individual.

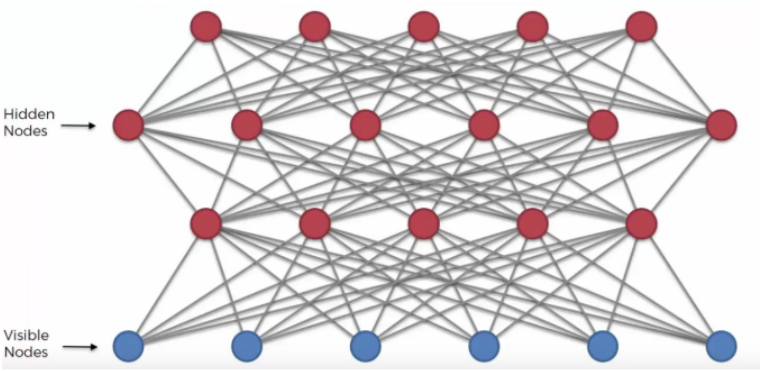


Figura 2.11: Red de Creencia Profunda (DBN)

Dicho todo esto, mostraremos a continuación, una imagen comparativo de un modelo RBM y DBN:

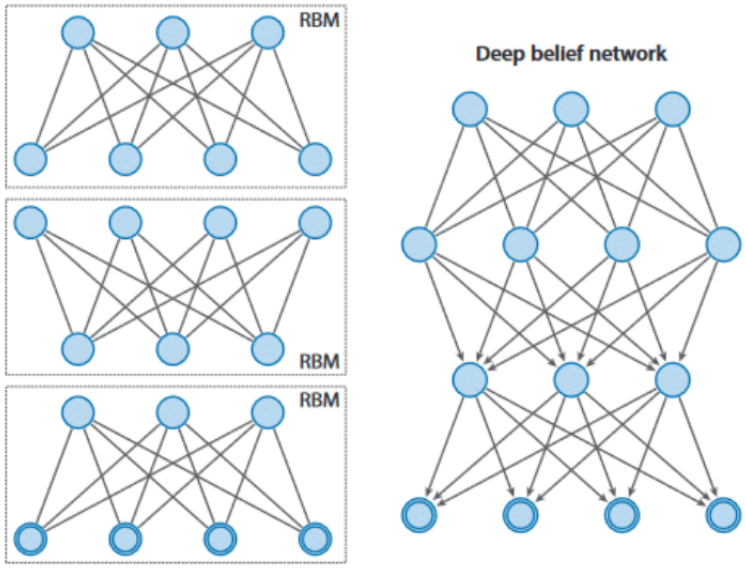


Figura 2.12: Comparativa RBM y DBN

2.5. Arquitecturas Neuronales

- **Según el número de capas**

1. *Redes de Neuronas monocapas-Perceptrón monocapa*: Se trata de la red de neuronas más sencilla ya que únicamente cuenta con una capa de neuronas que proyectan las entradas a una capa de neuronas de salida donde se realizan diferences cálculos.

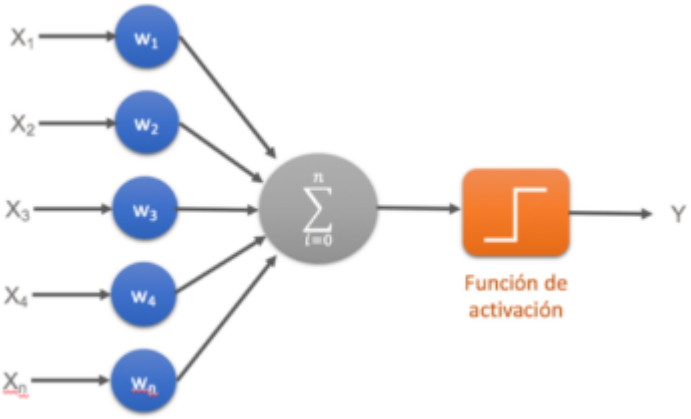


Figura 2.13: Perceptrón Simple

2. *Redes de Neuronas multicapa-Perceptrón multicapa*: Es una generalización de la anterior existiendo un conjunto de capas intermedias entre la entrada y la salida (capas ocultas). Este tipo de red puede estar total o parcialmente conectada.

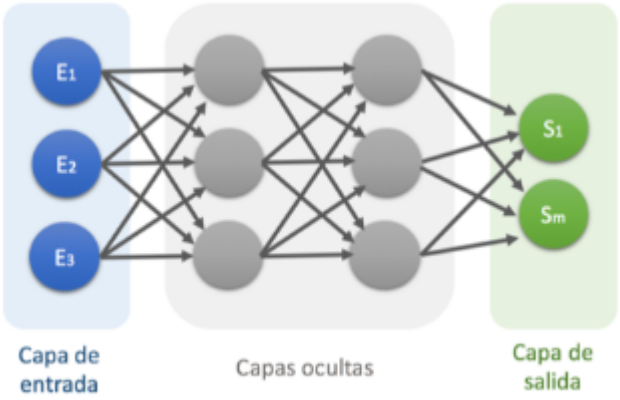


Figura 2.14: Perceptrón Multicapa

- **Según el tipo de conexiones**

  1. *Redes de Neuronas no Recurrentes*: En esta red la propagación de las señales se produce en un sentido solamente, no existiendo la posibilidad de realimentaciones. Lógicamente estas estructuras no tienen memoria.
  2. *Redes de Neuronas Recurrentes*: Esta red viene caracterizada por la existencia de lazos de realimentación. Estos lazos pueden ser entre neuronas de diferentes capas, neuronas de la misma capa o, más sencillamente, entre una misma neurona. Esta estructura recurrente la hace especialmente adecuada para estudiar la dinámica de sistemas no lineales.

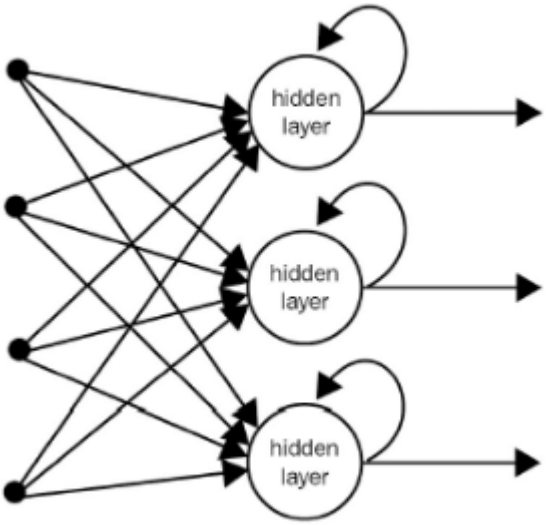


Figura 2.15: Red de Neuronas Recurrente

- **Según el grado de conexión**

  1. *Redes de Neuronas totalmente conectadas*: En este caso todas las neuronas de una capa se encuentran conectadas con las de la capa siguiente (redes no recurrentes) o con las de la anterior (redes recurrentes).
  2. *Redes parcialmente conectadas*: En este caso no se da la conexión total entre neuronas de diferentes capas.

Las estructuras neuronales se podrían conectar entre sí para dar lugar a estructuras

mayores. Es el nivel de la meso-estructura. Tal conexión se puede llevar a cabo de diferentes maneras, siendo las más usuales las de en paralelo y las jerárquicas.

2.6. Clasificación de redes según el método de aprendizaje

- **Aprendizaje supervisado**: Un supervisor realiza un entrenamiento controlado, de la respectiva entrada y la respuesta a la misma que se debe generar, modificando los pesos de las conexiones para que dicha salida a obtener sea la más aproximada a la que se desea. Cabe distinguir:

  1. *Aprendizaje por correción de error*: Se ajustan los pesos de las conexiones según la diferencia (error) entre valores esperados y obtenidos, con algoritmos como mínimo error cuadrático y programación hacia atrás.

  2. *Aprendizaje estocástico*: Realiza cambios aleatorios a los pesos, observando si la predicción mejora o no, para quedarse con los resultados positivos.

- **Aprendizaje no supervisado**: Su principal característica es que no requiere influencia externa para ajustar los pesos. Se trata de encontrar las diferencias singulares en los datos de entrada, interpretándolos según su estructura y algoritmo empleado, pudiendo ser las distintas categorías:

  1. *Aprendizaje hebbiano*: Para medir la familiaridad o extraer las características de los datos de entrada.

  2. *Aprendizaje competitivo y comparativo*: Para realizar clasificaciones de esos datos.

- **Aprendizaje por refuerzo**: Es un tipo de aprendizaje más lento, pues no se dispone de un conjunto completo de datos exactos de salida, indicándose solamente si el dato es aceptable o no, con lo cual el algoritmo ajusta los pesos en base a un mecanismo de probabilidades.

- **Aprendizaje híbrido**: Para algunas capas el aprendizaje será supervisado y para otras no.

2.7. Aplicaciones de las Redes de Neuronas

Cabe señalar las siguientes por el campo de conocimiento:

- *Medicina*: La principal proyección es hacia el diagnóstico médico, así por ejemplo:

1. *Detección de tumores cancerígenos*: Una red de neuronas entrenada localiza y clasifica en imágenes médicas la posible existencia de tumores cancerígenos.

2. *Diagnóstico de cardiopatías*: Valdría para la clasificación de electrocardiogramas.

3. *Predicción de enfermedades degenerativas cardíacas*: Serviría para modelizar el comportamiento de las arterias coronarias en los casos de pacientes de infarto, con riesgo de sufrir otro.

4. *Pronóstico del riesgo de intoxicación por digoxina*: Se trata de predecirlo, para ese fármaco (usado en problemas de corazón).

- *Procesado de la señal*: Actividades como las siguientes:

  1. *Reconocimiento de patrones en imágenes*: Se trata de una tarea sencilla para las personas, pero costosa de implementación en un sistema artificial.

  2. *Reconocimiento de voz.*

  3. *Eliminación activa de ruido*: Cuando el ruido y la señal de interés tienen los espectros frecuenciales solapados, un filtrado selectivo en frecuencia no tiene sentido. En este caso hay que intentar otras soluciones, como la cancelación activa de ruido aplicando sistemas adaptativos y redes de neuronas.

- *Economía*: En esta disciplina, habiendo de elegirse entre opciones diferentes, las redes de neuronas son útiles, frente a otros métodos, por sus características intrínsecamente no lineales. Así, cabe señalar:

  1. *Concesión de créditos*: En cuanto a decidir o no su viabilidad según la solvencia del solicitante, según determinados marcadores o datos económicos del mismo.

  2. *Tendencias a corto y medio plazo en Bolsas de Valores.*

  3. *Predicción de stocks*: Sobre el déficit o exceso de almacenaje de productos o mercaderías, necesario para el buen fin de actividades de comercio.

- *Medio Ambiente*: Para este campo, las aplicaciones de las redes de neuronas pueden ser, por ejemplo, en predicción de la erradicación solar, de los niveles tóxicos de ozono en zonas urbanas y rurales, y de variaciones globales de temperaturas.

Capítulo 3

Redes de Neuronas Recurrentes

3.1. Introducción

Las Redes de Neuronas Recurrentes (RNR), o en inglés, Recurrent Neural Networks (RNN) forman una familia de modelos apropiados para procesar datos de estructura secuencial. Son potentes por tener un estado oculto de alta dimensión con una dinámica no lineal que les permite recordar y procesar información pasada. Los vectores gradientes pueden calcularse de forma eficiente a la hora de actualizar los pesos de la red en el procesamiento de datos temporales. Pese a sus cualidades atractivas, los modelos RNN no pudieron convertirse en herramienta convencional en el aprendizaje automático por la dificultad de entrenarlas de manera efectiva, debido a una relación muy inestable entre los parámetros y la dinámica de los estados ocultos, que se manifiesta en el "problema de gradientes de desaparición/explosión" (Bengioetal, 1994). De ahí que haya sido escasa la investigación sobre el modelo RNN estándar en los últimos 20 años, habiendo sólo algunas aplicaciones exitosas que utilizan modelos RNN grandes (Robinson, 2002; Pollastri, 2002).

Vamos a comparar las redes de neuronas recurrentes con las redes "Feedforward", por lo que previamente vamos a recordar lo que era una red Feedforward.

3.2. Red de Neuronas Feedforward

Las Redes de Neuronas Feedforward, traducidas como redes de neuronas alimentadas hacia adelante, son aquellas cuyo gráfico no tiene ciclos.

Es común representarlas en capas, en cuyo caso se denominan Feedforward en capas. Las neuronas que reciben señales de excitación son las de entrada o primera capa. Las que toman su salida como la de la red pertenecen a la última capa o capa de salida. Y las que no corresponden a una ni a otra son neuronas internas de la red, pudiendo organizarse en una o más capas internas, son las ocultas.

Estas redes son actualmente las más populares, así por el método de aprendizaje, ampliamente utilizado, aunque ineficiente, de la "retropropagación", término con el que

algunos autores, de forma incorrecta, las denominan. A continuación se muestra un ejemplo gráfico:

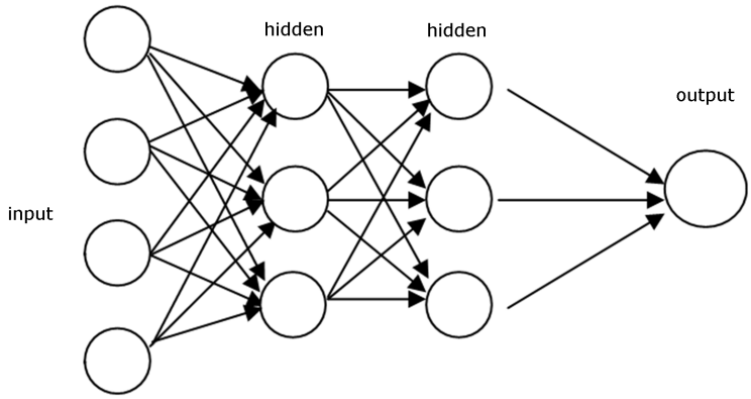


Figura 3.1: Red de Neuronas Feedforward

Tales redes son capaces de aproximar cualquier función no lineal, con una precisión dependiente del número de neuronas. Pero tienen una dinámica muy limitada, no pudiendo representar todos los sistemas, incluso usando neuronas de ecuación diferencial de primer orden o una diferencia finita.

3.3. Red de Neuronas Recurrente

Las RNNs son una familia particular de Red de Neuronas Artificiales (RNAS) caracterizadas por la existencia de conexiones de retroalimentación, esto es, ciclos o bucles dentro de la estructura de la red, mediante los cuales se pueden analizar secuencias de tiempo. De hecho, en el aspecto computacional, a menudo se realiza el despliegue de la estructura, como el que se muestra en la figura que se expone seguidamente, para obtener una versión de la red que dependa de una secuencia de entradas. Mientras que los pesos y los sesgos del bloque S son compartidos, cada salida h_t depende del procesamiento por la red de todas las entradas x_i con $i \in [0; t]$. El número de bloques en la versión desplegada depende esencialmente de la longitud de la secuencia a analizar.

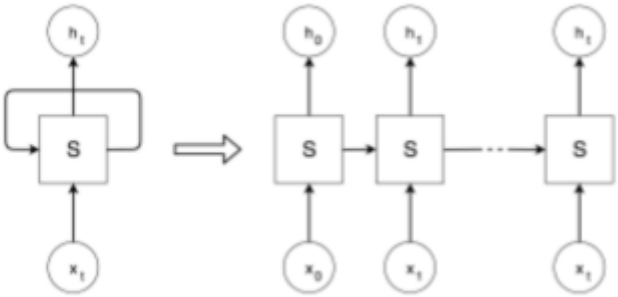


Figura 3.2: Despliegue de una red recurrente

Lo que distingue la RNN de la Feedforward es que comparte un estado (pesos y sesgo) entre los elementos de la secuencia, suponiendo lo almacenado internamente un

patrón de unión temporal de los elementos de la serie que analiza la red. Se puede mezclar parte de las entradas o de las salidas para obtener diferentes combinaciones, algunas de ellas mostradas en la figura que se muestra a continuación. Por ejemplo, cabe enlazar uno a muchos para clasificar una secuencia de datos con una sola salida o para etiquetar el conjunto de sujetos presentes de una imagen.

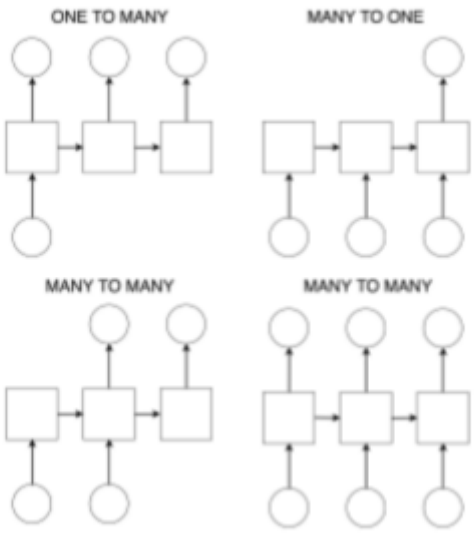


Figura 3.3: Algunas combinaciones posibles en una RNN

Así pues, resumidamente, las redes de neuronas recurrentes (RNN) son modelos para abordar problemas dentro del alcance del reconocimiento de patrones y se basan fundamentalmente en los mismos conceptos que los perceptrones multicapa (MLP) de retroalimentación. La diferencia radica en que éstos tienen por definición múltiples capas y las RNN no, pero sí, en cambio, un ciclo dirigido a través del cual las entradas se transforman en salidas. Abordaremos en este capítulo varios modelos de RNN y de aplicaciones en la vida real de dichas redes.

3.4. Modelos Neuronales Recurrentes Tradicionales

Existen redes dinámicas por naturaleza (como la de Elman, de Jordan y de Hopfield) y las de carácter estático (como de multicapa) que logran un funcionamiento dinámico realimentando sus entradas con muestras previas de salidas, comportamiento que las hace especialmente útiles para simular e identificar sistemas dinámicos no lineales.

3.4.1. Redes de Primer Orden

- **Red recurrente simple (Elman,1990)**: Las arquitecturas RNN recibieron contribuciones adicionales de Jeffrey Elman, a quién se atribuye el origen de este modelo que lleva su nombre. Esas aportaciones lo fueron sobre todo respecto de algoritmos de procesamiento de lenguaje, pero tienen también utilidad para cualquier problema en que los datos de entrada sean secuenciales o basados en series de tiempo. Se expone gráficamente a continuación la estructura básica de una red de Elman.

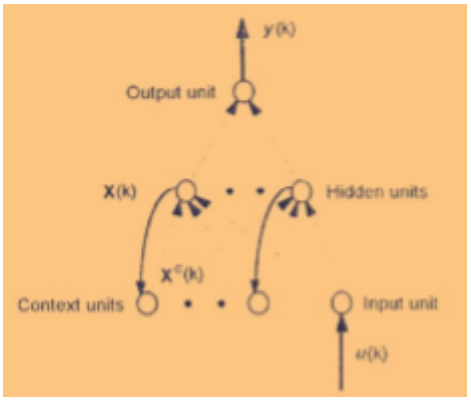
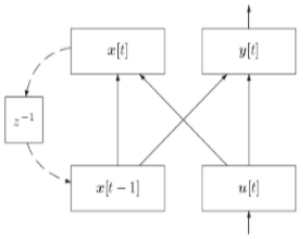


Figura 3.4: Ilustración de la red de Elman

Elman incluyó en la arquitectura una capa de unidades de contexto que se distinguen por una funcionalidad altamente preocupada por los estados internos anteriores. Lo que distingue a una red de Elman del resto es que la salida de la capa oculta también alimenta a las unidades de contexto en la capa anterior, pero los pesos que conectan una y otras tienen un valor constante de 1, lo que hace que la relación sea lineal. Después, las capas de entrada y de contexto activan simultáneamente la capa oculta, por lo que ésta también genera un valor al realizar el paso de actualización. En la sucesiva etapa, la secuencia de entrenamiento es similar, con la diferencia de que la capa con unidades de contexto adopta ahora los valores de la capa oculta de la época anterior (por ello se la ha descrito como la red "con memoria"). El entrenamiento de esta red de neuronas requiere multitud de pasos, un número que lógicamente depende de la longitud de la cadena elegida.

- **Red parcialmente Recurrente (Robinson y Fallside,1991)**: Es una red recurrente de propagación de errores o parcialmente recurrente de primer orden, con dinámica que se explica mediante el esquema de ecuaciones e ilustración siguientes:



$$y_i(t) = gy(Y_i(t)) \qquad i = 1,\ldots\ldots,ny.$$
$$Y_i(t) = \sum_{j=1}^{nx} W_{i,j}^{y,u} u_j(t) + \sum_{j=1}^{nx} W_{i,j}^{y,x} x_j(t-1) + W_i^y$$
$$x_i(t) = gx(X_i(t)) \qquad i = 1,\ldots\ldots,nx.$$
$$X_i(t) = \sum_{j=1}^{nu} W_{i,j}^{x,u} u_j(t) + \sum_{j=1}^{nx} W_{i,j}^{x,x} x_j(t-1) + W_i^x(1)$$

Figura 3.5: Esquema y dinámica de red parcialmente recurrente de primer orden

En este tipo de red, como podemos observar en el ejemplo expuesto, el estado $x(t)$ depende de las variables de entrada al sistema y del estado anterior a través de una función de retardo. Al mismo tiempo, existe una relación directa entre el estado $x(t)$ y la salida $y(t)$, lo que lo hace ser más directamente recurrente que una red de Elman, pero no totalmente debido a que la recurrencia no relaciona los estados $x(t)$ e $y(t)$.

- **Red totalmente recurrente (Williams y Zipser, 1989)**: Imaginemos que tenemos una entrada, x, que estamos ingresando en un modelo RNN, donde definimos

el estado como h, con las entradas multiplicadas por una matriz de peso, W. Hasta aquí todo funciona como en los modelos de redes neuronales descritos previamente (convolucionales, perceptrón monocapa, perceptrón multicapa ...), pero con la diferencia de la que las RNN realizan la misma tarea en las entradas a lo largo del tiempo, debido a lo cual, para calcular el estado actual de la red, procede la siguiente ecuación:

$$h_t = f(W_t x + W_R h_{t-1} + b), \quad donde \;\; f = tanh \tag{3.1}$$

$$y_t = f(W h_t + b) \tag{3.2}$$

donde $W_{t,r}$ =pesos, h_t =capa oculta, b =término independiente.

La característica clave aquí es que cuando la red de neuronas realiza tales operaciones, se despliega en múltiples estados nuevos, cada uno dependiente de los anteriores. Se realiza la misma tarea para cada entrada que se introduce,contando además con la dependencia funcional del modelo. A este procedimiento se le suele llamar memoria. Se ejemplifica a continuación:

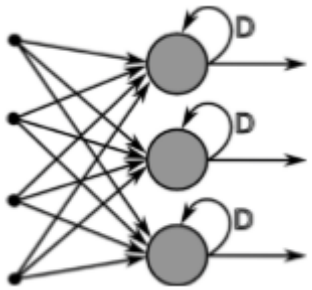


Figura 3.6: Arquitectura de una red de neuronas recurrente

Esta forma de RNN se desarrolló en la década de 1980. Al igual que otras redes de neuronas, las múltiples capas de neuronas están conectadas por pesos, y cada peso se altera a través de métodos de retropropagación, en función de una estadística de evaluación, que es la suma ponderada de las unidades de activación en un paso de tiempo dado. El error total resulta del conjunto de esas sumas ponderadas individuales en todos los pasos de tiempo. Puede haber activaciones de objetivos dirigidas por el revisor para algunas de las unidades de salida en ciertos pasos de tiempo. Por ejemplo, si la secuencia de entrada es una señal de voz que corresponde a un dígito hablado, la salida objetivo final al término de la secuencia puede ser una etiqueta clasificadora del dígito. Para cada secuencia, su error es la suma de desviaciones de las señales objetivo de las respectivas activaciones calculadas por la red. Para un conjunto de entrenamiento de muchas secuencias, el error total es la suma de todas las secuencias individuales.

3.4.2. Redes de Segundo Orden

La red recurrente (RR) simple de segundo orden (RRS2, Carrasco,1996) viene dada por:

$$y_i(t) = gy(\sum_{j=1}^{nx} W_{i,j}^{y,x} x_j(t) + W_i^y) \qquad i = 1, ..., ny \tag{3.3}$$

$$x_i(t) = gx(\sum_{j=1}^{nx}\sum_{k=1}^{nu} W_{i,j,k}^{x,x,u} x_j(t-1)u_k(t) + W_i^x) \qquad i = 1, ..., nx \tag{3.4}$$

También podemos obtener una versión de segundo orden (RPR2) de la red parcialmente recurrente, topología utilizada por *Olmin* y *Giles (1996)*, que se verifica con las siguientes ecuaciones:

$$y_i(t) = gy(\sum_{j=1}^{nx}\sum_{k=1}^{nu} W_{i,j,k}^{y,x,u} x_j(t-1)u_k(t) + W_i^y) \qquad i = 1, ..., ny \tag{3.5}$$

$$x_i(t) = gx(\sum_{j=1}^{nx}\sum_{k=1}^{nu} W_{i,j,k}^{x,x,u} x_j(t-1)u_k(t) + W_i^x) \qquad i = 1, ..., nx \tag{3.6}$$

Por otro lado, la red totalmente recurrente de segundo orden(RTR2) propuesta por *Giles (1992)* se define partiendo de:

$$y_i(t) = x_i(t) \qquad i = 1, ..., ny \tag{3.7}$$

$$x_i(t) = gx(\sum_{j=1}^{nx}\sum_{k=1}^{nu} W_{i,j,k}^{x,x,u} x_j(t-1)u_k(t) + W_i^x) \qquad i = 1, ..., nx \tag{3.8}$$

Donde se suele hacer que

$$nx \geq ny$$

.

3.4.3. Adición de una ventana temporal

Todos los modelos que hemos visto de RNN pueden ser ampliados con la incorporación a sus entradas de una memoria explícita a corto plazo, de forma que la entrada a la red consistirá en el valor actual $u(t)$ concatenado con los $p-1$ valores anteriores: $u(t-1), ..., u(t-p+1)$. A este valor p que hemos introducido se le conoce como *orden de la memoria de entrada* u *orden de entrada.*

Con esta ampliación, la red recurrente puede acceder de forma explícita al estado más reciente, y utilizar el estado para almacenar información relativa a un pasado más remoto. Así una de las ecuaciones anteriores se convierte ahora en:

$$X_i(t) = \sum_{j=1}^{nu}\sum_{k=1}^{p} W_{i,j(k)}^{x,u} u_j(t-k-1) + \sum_{j=1}^{nx} W_{i,j}^{x,x} x_j(t-1+W_i^x) \tag{3.9}$$

Donde hemos usado $W_{i,j(k)}^{x,u}$ para referirnos al peso que une $u_j(t-k-1)$ con la neurona i del estado. Las otras ecuaciones se modificarían de forma similar.

3.5. Algoritmos de Entrenamiento

Los algoritmos de entrenamiento modifican parámetros configurables de la red intentando minimizar el error cuadrático medio. Es un problema complejo para el que es necesario aplicar ciertos criterios heurísticos.

El entrenamiento de cualquier Red de Neuronas (RN) se hace mediante un proceso de aprendizaje, para el que es menester una topología de red, y en concreto:

- Número de neuronas en la capa de entrada, que depende del número de componentes del vector de entrada.
- Cantidad de capas ocultas y número de neuronas en cada una de ellas. Dichas capas deben determinarse en sus distintas configuraciones.
- Número de neuronas en la capa de salida, que depende del número de componentes del vector salida.
- Funciones de transferencia requerida en cada etapa, a elegir según las características del problema.

Con base a la topología escogida se asignan valores iniciales a cada uno de los parámetros que conforman la red.

3.5.1. Entrenamiento RNNs

A continuación, se hace expresión, comentada, de la dificultad que conlleva el entrenamiento de RNNs.

3.5.1.1. Entrenando RNNs con retropropagación a través del tiempo (BPPT)

Se considera a Sepp Hochreiter y Jurgen Schmidhuber los precursores en el desarrollo de métodos de evaluación de rendimiento de modelos para el aprendizaje profundo.

El método estándar es el denominado retropropagación (o propagación hacia atrás) a través del tiempo (BPTT). Es un modelo parecido al de la retropropagación regular, pero adaptada a las redes de neuronas recurrentes, como desarrollo del modelo a través de varios pasos del tiempo.

Por etapas, se comienza en el primer entrenamiento, en secuencias pequeñas y con aumento gradual de las mismas. Se visualiza de manera intuitiva como entrenamiento en una secuencia de longitud 1,2, a través de N, donde N es la longitud máxima posible de la secuencia. Podemos ilustrar lo dicho de forma descriptiba con la siguiente ecuación:

$$\delta_{p,j}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f^{'}(s_{pj}(t-1)) \qquad (3.10)$$

Donde t=paso de tiempo, h índice para el nodo oculto en t, j nodo oculto en el paso $t = 1$, y $\delta = errores$. En detalle, podemos ver los fenómenos con la siguiente ecuación, siendo W la matriz de pesos para la capa de salida

$$W(t+1) = W(t) + \eta s(t) e_o(t)^T \tag{3.11}$$

donde e_o representa los errores según esta expresión:

$$e_o(t) = d(t) - y(t) \tag{3.12}$$

Ahora tenemos k secuencias, a través de las cuales desplegamos la red en una red de retroalimentación regular.

Sin embargo, la capa recurrente en la red de neuronas recurrentes toma simultáneamente la entrada de la capa anterior, así como la capa sucesiva. Para compensar el cambio en los pesos que tiene lugar por la entrada simultánea al propagarse los errores, promediamos las actualizaciones que recibe cada capa.

3.5.1.2. Algoritmo BPPT

Dada una secuencia de entrada $(v_1, ..., v_T)$ (que denotamos por v_1^T), la red de neuronas recurrente calcula una secuencia de estados ocultos h_1^T y una secuencia de salidas z_1^T mediante el siguiente algoritmo:

1. **for** t **from** 1 **to** T **do**
2. $u_t \leftarrow W_{hv} v_t + W_{hh} h_{t-1} + b_h$
3. $h_t \leftarrow e(u_t)$
4. $o_t \leftarrow W_{oh} h_t + b_o$
5. $z_t \leftarrow g(o_t)$
6. **end for**

donde $e(\cdot)$ y $g(\cdot)$ son las no linealidades ocultas y de salida de la red de neuronas recurrente, y h_0 es un vector de parámetros que almacenan el primer estado oculto. La pérdida de la red neuronal recurrente suele ser una suma de pérdidas por paso de tiempo:

$$L(z, y) = \sum_{t=1}^{T} L(z_t; y_t) \tag{3.13}$$

Las derivadas de las redes de neuronas recurrentes se calculan fácilmente con el algoritmo de retropropagación a través del tiempo*(BPTT; Werbos, 1990; Rumelhart en el 1986)*:

1. **for** t **from** T **downto** t **do**
2. $do_t \leftarrow g'(o_t) \cdot dL(z_t; y_t)/dz_t$

3. $db_o \leftarrow db_o + do_t$
4. $dW_{oh} \leftarrow dW_{oh} + do_t h_t^T$
5. $dh_t \leftarrow dh_t + W_{oh}^T do_t$
6. $dz_t \leftarrow e'(z_t) \cdot dh_t$
7. $dW_{hv} \leftarrow dW_{hv} + dz_t v_t^T$
8. $db_h \leftarrow db_h + dz_t$
9. $dW_{hh} \leftarrow dW_{hh} + dz_t h_{t-1}^T$
10. $dh_{t-1} \leftarrow W_{hh}^T dz_t$
11. **end for**
12. **Return** $d\theta = [dW_{hv}, dW_{hv}, dW_{hh}, db_h, db_o, dh_o]$

3.5.1.3. La dificultad de entrenar RNNs

Aunque los gradientes de la red de neuronas recurrente son fáciles de calcular, las redes de neuronas recurrentes son particularmente difíciles de resolver, especialmente en problemas con dependencias temporales de largo alcance (Bengioetal, 1994; Martens and Sutskever, 2011; Hochreiter y Schmidhuber, 1997), debido a su naturaleza iterativa no lineal. Un pequeño cambio en un proceso iterativo puede agravarse y producir efectos muy grandes, pues se producirían muchas iteraciones tardías. Esto se conoce coloquialmente como "el efecto mariposa". El problema reside en que en una red de neuronas recurrente, la derivada de la función de pérdida en un momento puede ser exponencialmente grande con respecto a las activaciones ocultas en un momento mucho anterior. Por lo tanto, la función de pérdida es muy sensible a pequeños cambios, por lo que se vuelve efectivamente discontinua. Las redes de neuronas recurrentes también sufren el problema del gradiente de fuga, descrito primero por *Hochreiter (1991)* y posteriormente por *Bengio (1994)*. Consideremos el término:

$$\frac{\partial L(z_T; y_T)}{\partial W_{hh}} \tag{3.14}$$

que es fácil de analizar utilizando la línea 10 del algoritmo BPPT:

$$\frac{\partial L(z_T; y_T)}{\partial W_{hh}} = \sum_{t=1}^{T} dz_t h_{t-1}^T \tag{3.15}$$

donde

$$dz_t = (\prod_{\tau=t+1}^{T} W_{hh}^T e'(z_t))(W_{oh}^T g'(o_t) \frac{\partial L(z_T; y_T)}{\partial p_T}) \tag{3.16}$$

Si todos los valores propios de W_{hh} son considerablemente más pequeños que 1, entonces las contribuciones $dz_t h_{t-1}^T$ a dW_{hh} disminuirá rápidamente porque dz_t tiende a cero exponencialmente a medida que $T - t$ aumenta. El último fenómeno se conoce como el problema del gradiente de fuga, y se garantiza que se producirá en cualquier red de neuronas recurrente que pueda almacenar un bit de información de forma indefinida mientras sea capaz de aguantar un cierto nivel de ruido,condición que debería ser cumplida

por la mayoría de las redes de neuronas recurrentes que realizan cálculos interesantes. Un dz_t que se desvanece no es deseable, porque convierte el BPPT en BPPT truncado, que probablemente no sea capaz de realizar entrenamientos adecuados en las redes de neuronas recurrentes, haciendo inestable la estructura temporal a largo plazo.

La desaparición y la explosión del gradiente son problemas que hacen que sea difícil optimizar la red de neuronas recurrente en secuencias con dependencias temporales de largo alcance, y son posibles causas del abandono del estudio de la red de neuronas recurrentes por parte de los investigadores de aprendizaje automático.

3.6. Neural History Compressor

Este modelo se propuso para resolver el problema del desvanecimiento del gradiente. Cuando se entrenan redes con muchas capas, como en la que se utilizan en Deep Learning, la magnitud del gradiente va disminuyendo de forma importante al realizar la propagación hacia atrás a través de la red. De ese modo, el error inicial prácticamente no tiene efecto al llegar a las capas cercanas a la de entrada a la red.

El Neural History Compressor es una pila no supervisada de redes de neuronas. Solamente las entradas no predecibles de alguna red de neuronas artificial se convierten en entradas para la siguiente red de neuronas artificial de mayor nivel. Cada red de neuronas artificial de mayor nivel trabaja con una versión comprimida de la información de la red de neuronas artificial previa. La secuencia de entradas puede ser reconstruida a partir de la representación en los niveles superiores.

3.7. Red de Neuronas Recurrente: un modelo para la memoria

Una de las tareas consideradas estándar para un ser humano, pero muy difícil para una máquina, es la comprensión de un texto. Ante un conjunto ordenado de palabras, ¿cómo se puede enseñar a una máquina a entender (o clasificar) su significado? Es evidente que en esa tarea hay una relación más sutil entre los datos de entrada que en otros casos, pues cuando, por ejemplo, se trata de clasificar el contenido de una imagen, la máquina sí puede procesar al mismo tiempo la imagen completa. En cambio, en un texto el significado no depende ya de las palabras sino también de su contexto. Hay que relacionarlas y contextualizarlas, respetando el orden de su lectura.

Las redes de neuronas recurrentes permiten recordar datos significativos en el proceso de estudio, dependiendo de la regla de propagación y de la memoria que se emplee. Así, por ejemplo, la red Elman es muy similar a una red de neuronas oculta de alimentación de una sola capa, con añadido a esa capa oculta de un conjunto de neuronas llamadas unidades de contexto. Para cada neurona presente en la capa oculta se agrega una unidad de contexto que recibe la salida de aquella como entrada y devuelve su salida a la misma neurona oculta. Mediante esta topología, la regla de propagación para una

entrada $x(t), t = 1, ..., n$ es: primero introducimos $x(1)$ en la red, calculamos el estado de las neuronas ocultas sin considerar las unidades de contexto, y posteriormente las salidas de red se calculan a través de la tercera capa de neuronas, actualizando las unidades de contexto al mismo tiempo. En cada paso posterior $x(t)$ se introduce en la red, luego se actualizan los valores de las neuronas ocultas, $y_i(t) = \mathcal{F}(\sum_{j=1} kx_j(t)w_{x_j,y_i} + u_i(t)w_{u_i,y_i}$, con la función de activación $\mathcal{F}$ de las neuronas de la capa oculta, después de lo cual la capa de salida y las unidades de contexto se actualizan para obtener los vectores $z(t)$ y $u(t)$. Se puede observar que, de hecho, $u(t) = y(t-1)$, por lo tanto, contemplar estas unidades de estado con algunos pesos fijos es equivalente a considerar una conexión de las neuronas de la capa oculta a sí mismas, con $w_{y_i,y_i} = w_{u_i,y_i}$.

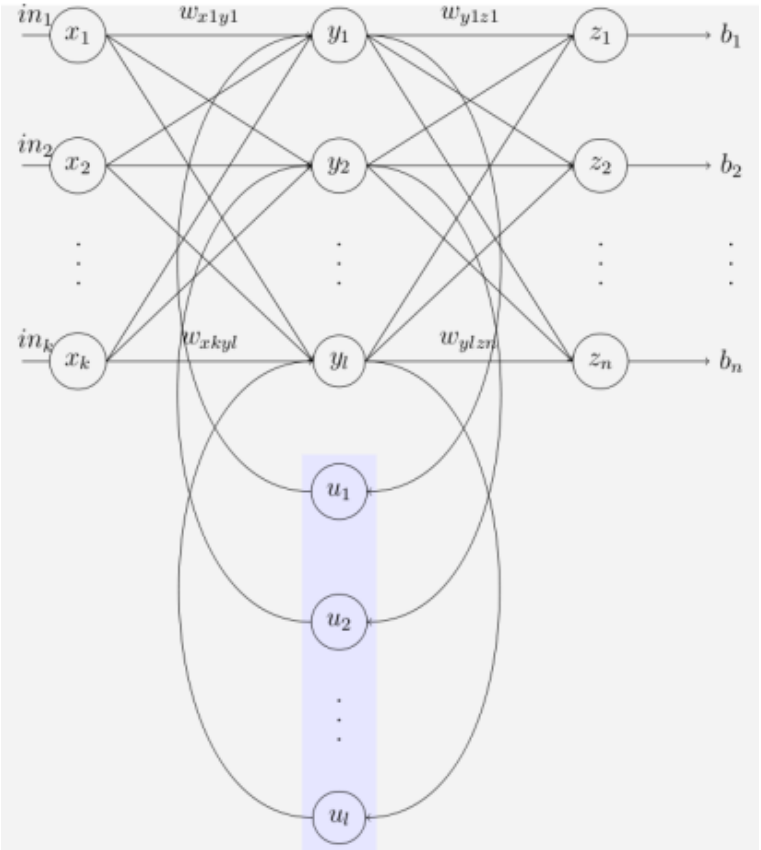


Figura 3.7: Red de Elman

Un tipo de red muy parecida a la de Elman es la de Jordan, en la que las unidades de contexto guardan los estados de las neuronas de salida en lugar de las de las neuronas ocultas. La idea subyacente es la misma: recibir una secuencia de datos como entrada, procesar una nueva secuencia de datos obtenidos como salida, y volver posteriormente a calcular los datos de las mismas neuronas. Las redes de neuronas recurrentes que se basan en este principio son múltiples, y las topologías individuales se eligen para afrontar los supuestas problemáticas. Si, por ejemplo, no basta con recordar el estado anterior de la red, pero la información procesada muchos pasos antes puede ser necesaria, se puede utilizar un modelo de redes de neuronas de memoria a largo y corto término.

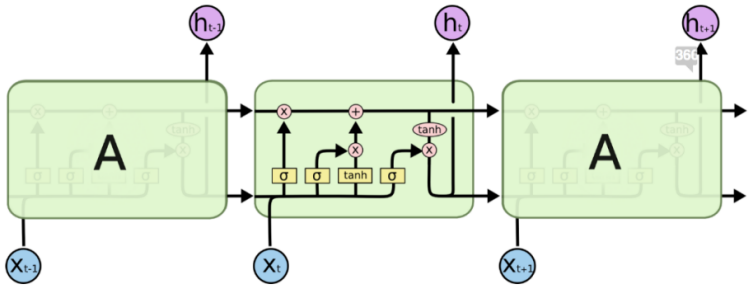


Figura 3.8: Arquitectura de una red LSTM

3.8. Long-Short Term Memory

El modelo Long-Short Term Memory, modelo de memoria a largo y corto término, o simplemente LSTM, representa una red de neuronas recurrente particular propuesta por Hochreiter y Schmidhuber para resolver el problema de la desaparición y la explosión del gradiente que compromete la eficiencia de la red de neuronas recurrente. El principio detrás del modelo LSTM es la celda de memoria, que mantiene el estado c fuera del flujo normal de la red de neuronas recurrente. De hecho, el estado tiene una conexión directa consigo mismo: el estado c_t es simplemente una suma entre el estado anterior $c_{\text{t-1}}$. Dado que la función de activación para actualizar el estado es, de hecho, la identidad, la derivada será 1 y, por lo tanto, el gradiente en la retropropagación no desaparecerá ni explotará, sino que permanecerá constante. Para el modelado de datos, la implementación de LSTM proporciona la presencia de capas llamadas compuertas (o sencillamente, puertas), que aplican una activación sigmoidea a la concatenación de entrada x_t y salida $h_{\text{t-1}}$, obteniendo un vector de salida cuyos elementos pertenecen al rango $(0; 1)$.

El uso de un sigmoide en lugar de un simple paso binario es más efectivo tanto porque es diferenciable como porque indica una cantidad de datos para recordar en lugar de proporcionar una salida dicotómica que solo indica si recordar o no recordar. Dichas puertas son las siguientes:

- **forget gate** (f): Se usa para ocultar el estado $c_{\text{t-1}}$ de modo que solo se considere parte del estado en el paso t.
- **input gate** (i): Se usa para ocultar qué información nueva se almacenará en el estado. Estas últimas son procesadas por otra capa, generalmente con tangente hiperbólica como activación, para garantizar que la suma que actualiza el estado también implique valores negativos y, por lo tanto, restas.
- **output gate** (o): Se utiliza para ocultar la salida de la red. Esto último viene dado por una simple aplicación de una tangente hiperbólica en el estado actual.

En la figura que expondremos a continuación se puede observar la estructura de la LSTM, donde las operaciones x y $+$ indican productos y sumas en cuanto a elementos, los bloques rectangulares son capas, tan_h indica una tangente hiperbólica, x la entrada, h la salida y el estado.

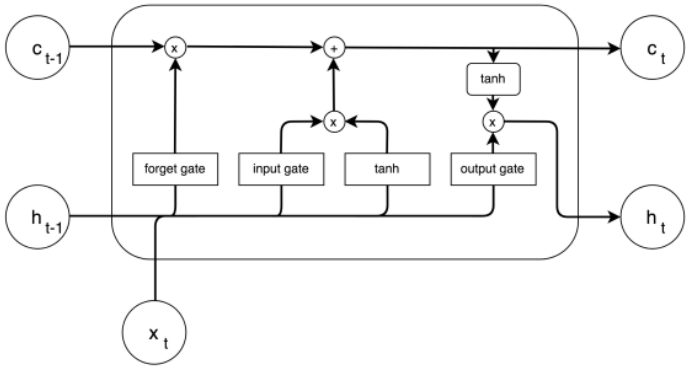


Figura 3.9: Esquema de una red LSTM

La gama de aplicaciones de LSTM es amplia y popular, utilizándose habitualmente para control de robots, predicción de series de tiempo, reconocimiento de voz y otras tareas. A diferencia de las unidades propias de otras arquitecturas de redes de neuronas recurrentes, las redes LSTM contienen bloques. Otro factor distintivo clave es poder recordar un valor dado durante períodos prolongados de tiempo y las puertas dentro del modelo que determinan varios atributos de la secuencia de entrada.

Entre las consideraciones de las puertas se encuentran la importancia de la entrada, cuándo se debe mantener la memoria o si se produce una "recolección de basura" y se eliminan los datos, así como el tiempo del valor de salida. Una implementación típica de un bloque LSTM se muestra en la siguiente figura.

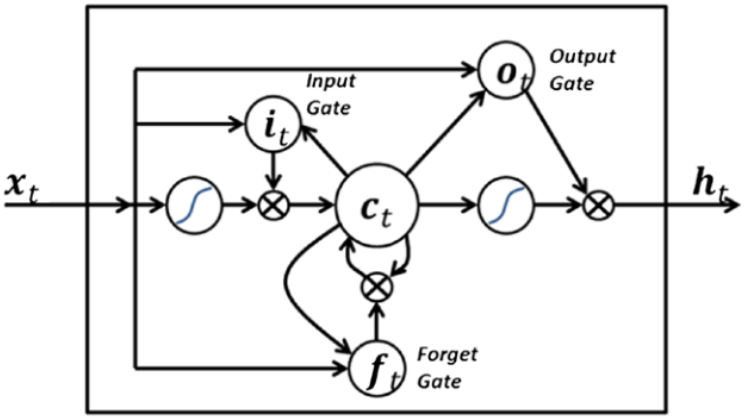


Figura 3.10: Visualización de una red LSTM

Las unidades sigmoideas en una LSTM estándar contienen la ecuación:

$$y = s(\sum_{i=1}^{N} w_i x_i) \qquad (3.17)$$

donde s es una función de aplastamiento (en muchos casos, a menudo una función logística o cualquier función de activación, como se describe en modelos anteriores). Mirando la figura de antes, la unidad sigmoidea más a la izquierda alimenta la entrada a

la memoria del bloque LSTM. A partir de este momento, las otras unidades en la figura sirven como puertas, que permiten o niegan el acceso a la memoria LSTM. La unidad titulada i, que denominamos como la puerta de entrada del diagrama, bloqueará la entrada de todos los valores en la memoria que son muy pequeños (cercanos a cero). La puerta de olvido, la unidad en la parte inferior de la figura, 'olvida' cualquier valor que estaba recordando y lo descarta de la memoria. La unidad en la esquina superior derecha de la figura es la "puerta de salida", que determina si el valor almacenado en la memoria de la LSTM debe emitirse. Ocasionalmente, observamos unidades que se denotan con los siguientes símbolos: $\prod$ o $\sum$. Las unidades que tienen el símbolo de suma se retroalimentan en el bloque LSTM para facilitar el recuerdo del mismo valor durante muchos pasos de tiempo sin disminución del valor. Por lo general, este valor también se ingresa en las tres unidades de puertas para mejorar sus procesos de toma de decisiones respecto. El producto Haramard, o producto de entrada de matrices utilizado en modelos LSTM, está dado por lo siguiente en notación de índice:

$$(A \circ B)_{i,j} = A_{i,j} x B_{i,j} \qquad (3.18)$$

3.8.1. LSTM Tradicional

De antes, tenemos las capas de una LSTM a través del cual pasan nuestros datos:

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \qquad (3.19)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \qquad (3.20)$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \qquad (3.21)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \qquad (3.22)$$

$$h_t = o_t \circ \sigma_h(c_t) \qquad (3.23)$$

donde:

$$x = vector\ \ de\ \ entrada,$$

$$h_t = vector\ \ de\ \ salida,$$

$$c_t = estado\ \ celular,$$

$$(W, U, b) = matrices\ \ de\ \ parámetros\ \ y\ \ vector,$$

$$(f_t, i_t, y\ \ o_t) = Informacion\ \ recordada,$$

$$informacion\ \ adquirida\ \ y\ \ salida, respectivamente,$$

$$\sigma_g = función\ \ sigmoidea,$$

$$\sigma_c = tangente\ \ hiperbolica\ \ original,$$

$$\sigma_h = tangente\ \ hiperbolica\ \ original.$$

3.8.2. Cómo lidiar con dependencias a largo plazo: redes LSTM

Las redes de neuronas recurrentes se pueden usar para tratar problemas relacionados con secuencias en las que el contexto temporal es importante. Sin embargo, las redes de Elman tienen una memoria bastante corta: solo guardan los valores de activación de las neuronas ocultas en el paso anterior. A veces puede ser útil, o incluso necesario, mantener cierta información durante más tiempo, para un contexto más amplio o específico en orden a un pronóstico más preciso. Un ejemplo puede ser la creación de un teclado predictivo, o instrumento que, ante determinado texto, intenta pronosticar cuál será la siguiente palabra. Así, pongamos un texto como el siguiente:

"Pasé varios años en Francia por negocios. [...]
Hablo con fluidez el ... "

Cabe inferir con facilidad que la siguiente palabra sería la del nombre del idioma, pero para saber que era francés tenía que recordarse la información que venía dada antes.

Para ese tipo de problemas de dependencia a largo plazo, existe una arquitectura específica de una red recurrente, la mencionada anteriormente, LSTM.

El elemento clave en esta estructura es el vector C_t representado por la línea horizontal en la parte superior. Retiene información incluso a largo plazo, realizando la función de una especie de "cinta transportadora". Se cambia de acuerdo con la información recibida gradualmente, pero no toda la "cinta transportadora" cambia la unidad de estado de igual manera.

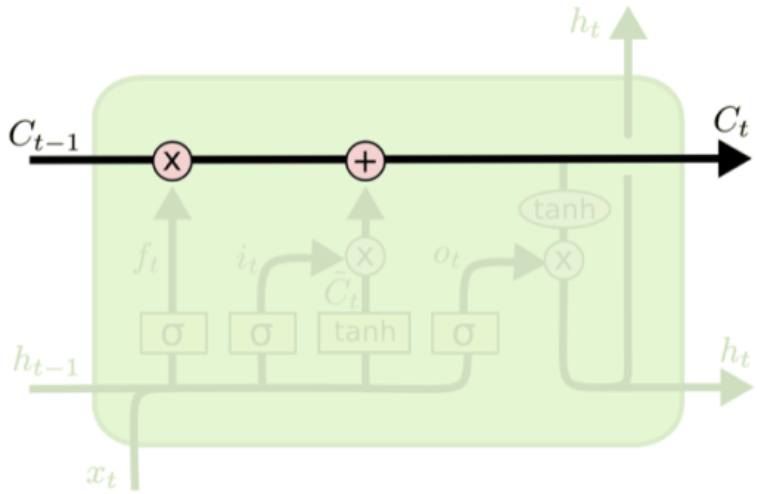


Figura 3.11: La unidad del estado

El primer paso en una iteración es la decisión de qué información debe ser olvidada por la unidad de estado. Para hacer esto, utilizamos una capa de avance con función de activación sigmoidea, que observa qué información está contenida en el vector $h_{t\text{-}1}$ y en el vector de entrada actual x_t para comprender cómo cambia el contexto, y devuelve un vector cuyos elementos están incluidos entre 0 y 1. Este vector se multiplicará elemento por elemento con el vector $C_{t\text{-}1}$ para decidir qué información debe olvidarse: si el vector

devuelto contiene números muy cercanos a 1, significa que debe recordarse la información correspondiente, si devuelve elementos muy cercanos a 0, significa que la información correspondiente debe ser olvidada.

En el ejemplo del teclado predictivo, la unidad de estado podría almacenar información sobre el género y el número del sujeto de una oración, y estos deberían olvidarse en presencia de un nuevo sujeto. Debido a su función, esta capa de la red se llama *capa de la puerta de olvido* (forget gate layer). En este punto, la información de la unidad de estado debe actualizarse.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad (3.24)$$

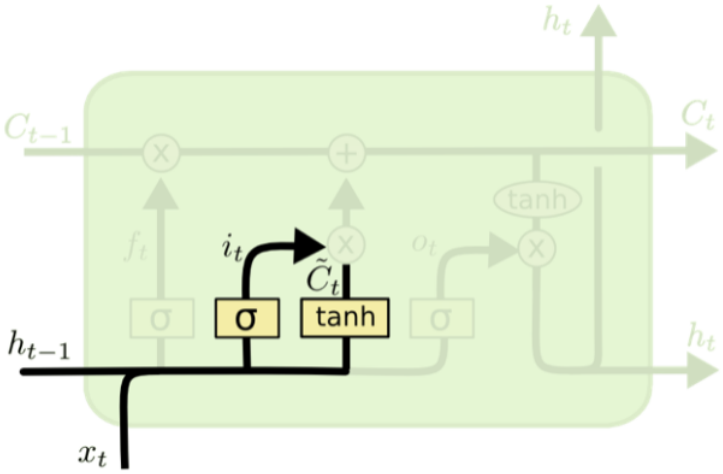


Figura 3.12: Capa de la puerta de olvido

Para este propósito, se utiliza la llamada *capa de la puerta de entrada*(input gate layer), una capa con activación sigmoidea que tendrá que modular el proceso de actualización, mientras que otra capa con activación en tan_h crea un vector de actualización candidato. Estos se multiplicarán elemento por elemento, y luego su producto se agregará a la unidad de estado anterior para generar la actual.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \qquad (3.25)$$

$$\tilde{C}_t = tan_h(W_c \cdot [h_{t-1}, x_t] + b_c) \qquad (3.26)$$

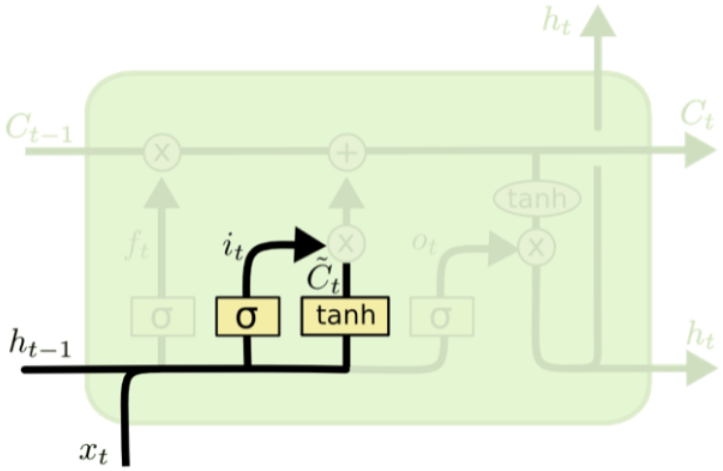


Figura 3.13: Capa de la puerta de entrada

Ahora que se ha determinado el contexto actual, es hora de generar una salida de red. Esto se basará en una versión filtrada de la unidad de estado y, por supuesto, en la entrada que acaba de recibir. En términos explícitos, mientras que al vector de entrada y al vector de salida anterior se les asigna un vector de salida o_t candidato a través de una capa adicional con activación sigmoidea, la unidad de estado se aplica a la función tan_h, y luego estos dos vectores se multiplican elemento por elemento para determinar la salida de h_t.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \qquad (3.27)$$

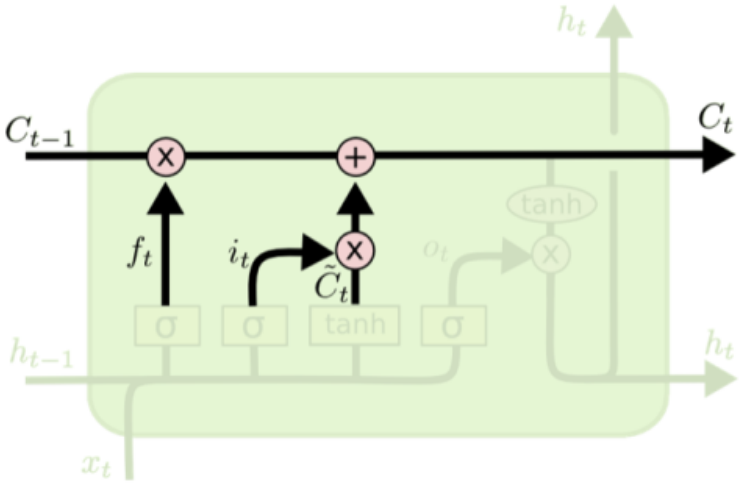


Figura 3.14: Actualización de la unidad de estado

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \qquad (3.28)$$

$$h_t = o_t \cdot tan_h(C_t) \qquad (3.29)$$

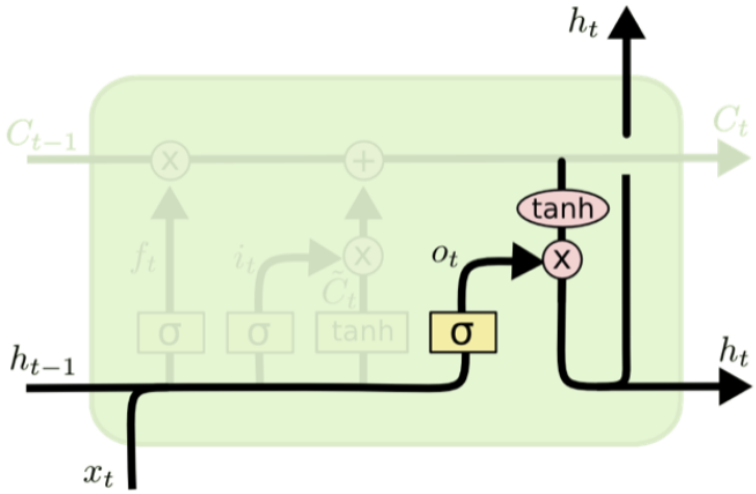


Figura 3.15: Cálculo de la salida

El operador h_t puede ser la salida de la red, pero también puede someterse a un procesamiento adicional en caso de que desee crear una red más compleja, agregando profundidad.

3.8.3. Entrenamiento LSTMs

BPPT se usa para LSTM. Los gradientes de fuga en un modelo LSTM dependen específicamente de la cantidad de errores que se produzcan. Los modelos LSTM "recuerdan" sus errores propagados hacia atrás, que luego se retroalimentan a cada peso. Por lo tanto, la retropropagación regular es efectiva para entrenar un bloque LSTM recordando valores durante períodos de tiempo muy largos.

3.9. Amortiguación estructural dentro de RNNs

El método del gradiente conjugado es un método iterativo utilizado para resolver numéricamente sistemas de ecuaciones lineales cuyas matrices son simétricas y definidas positivas.

Si estamos usando un método de gradiente conjugado y se aleja demasiado de la x original, la estimación de curvatura se vuelve inexacta y podemos observar una incapacidad para converger en el óptimo global. Sugerido por Martens y Sutskever, se recomienda la amortiguación estructural cuando se utilizan métodos de gradiente conjugado. Con este método, penalizamos grandes desviaciones de x, donde la fórmula viene dada por:

$$\tilde{f}_d(x + \Delta x) = \tilde{f}(x + \Delta x) + \lambda \|\Delta x\|^2 \qquad (3.30)$$

donde $\|\Delta x\|^2$ es la magnitud de desviación y λ sirve como parámetro de ajuste. Este parámetro es adaptable y para elegirlo se sugiere que encontremos una relación de reducción, según la siguiente ecuación:

$$\rho \equiv \frac{f(x + \Delta x) - f(x)}{\tilde{f}(x + \Delta x) - \tilde{f}(x)} \qquad (3.31)$$

3.10. Algoritmo de actualización de parámetros de ajuste

Los pesos se actualizan en cada paso de tiempo y, como tal, aumentar el valor en esta matriz puede causar cambios drásticos en la salida:

$$If(\rho > \frac{3}{4})\{$$

$$\lambda \longrightarrow \frac{2}{3}\lambda.$$

$$\}Else\ \ If(\rho < \frac{1}{4})\{$$

$$\lambda \longrightarrow \frac{3}{2}\lambda$$

$$\}Else\ \ If(\frac{1}{4} < \rho < \frac{3}{4})\{$$

$$\lambda \longrightarrow \lambda, (no\ \ update)\}$$

3.11. Redes de Neuronas Recurrentes como modelos generativos

Los modelos generativos son familias parametrizadas de distribuciones de probabilidad que extrapolan un conjunto de entrenamiento finito a una distribución en todo el espacio. Tienen muchos usos, como por ejemplo con espectrogramas para sintetizar el habla. Los modelos generativos del lenguaje natural pueden mejorar el reconocimiento del habla al decidir entre palabras que el modelo acústico no puede distinguir con precisión. Dichos modelos también pueden mejorar la traducción automática evaluando cómo de plausible son un gran número de traducciones candidatas y seleccionando la mejor. Una red de neuronas recurrente define un modelo generativo sobre secuencias si la función de pérdida satisface $L(z_t; y_t) = -logp(y_t; z_t)$ para alguna familia de distribuciones parametrizadas $p(.; z)$ y si $y_t = v_{t+1}$. Esto define la siguiente distribución sobre las secuencias v_1^T:

$$P(v_t|v_1^{t-1}) \equiv P(v_t; z_t) \tag{3.32}$$

$$P(v_1^T) = \prod_{t=1}^{T} P(v_t|v_1^{t-1}) \equiv \prod_{t=1}^{T} P(v_t; z_{t-1}) \tag{3.33}$$

donde z_0 es un vector de parámetro adicional que especifica la distribución en el primer paso de secuencia de tiempo, v_1. Esta ecuación define una distribución válida porque z_t es una función de v_1^t y es independiente de v_{t+1}^T. Las muestras de $P(v_1^T)$ se pueden obtener elaborando muestras de la distribución condicional $v_t \sim P(v_t|v_1^{t-1})$ secuencialmente, iterando de $t = 1$ a T. La función de pérdida es equivalente al promedio de probabilidad de registro negativo de las secuencias en el conjunto de entrenamiento:

$$Train_S(\theta) = E_v \sim S[-logP_\theta(v)] \tag{3.34}$$

donde en esta ecuación, P depende de forma explícita del parámetro θ.

Suponiendo que la distribución de datos D es exactamente igual a P_{θ^*} para algunos θ^* y se tiene que $\theta \to P_\theta$, es significativo discutir la velocidad a la que nuestras estimaciones de parámetros convergen a θ^* cuando aumenta el tamaño del conjunto de entrenamiento.

Si el término $logP_\theta(v)$ de (3.34) está uniformemente delimitado por θ, se sabe que el estimador de máxima verosimilitud (es decir, el parámetro que maximiza la ecuación $Train_S(\theta)$) tiene la tasa de convergencia más rápida posible entre todas las formas posibles de pasar de datos a parámetros cuando el tamaño del conjunto de entrenamiento es suficientemente grande *(Wasserman, 2004)*, lo que justifica su uso.

3.12. Aplicación de las Redes de Neuronas Recurrentes al Procesamiento de Secuencias

Se exponen a continuación algunas de las tareas relacionadas con el procesamiento de secuencias a las que se han aplicado las redes de neuronas recurrentes:

- **Predicción de series temporales**: Se trata de una aplicación muy habitual. Conociendo los valores anteriores de una o más variables, la red de neuronas debe predecir de la forma más correcta posible el valor futuro. Así, por ejemplo, se puede utilizar en series económicas (McCluskey, 1993) o en fenómenos naturales (Aussem, 1995), e incluso la finalización de melodías inacabadas.
- **Procesamiento del lenguaje humano**: Aplicación de la red de neuronas recurrente al análisis sintáctico de frases o estudio de regularidad del lenguaje (Elman, 1990, 1991).
- **Ecualización de canales digitales**: Los efectos del canal sobre la señal transmitida en comunicaciones digitales pueden hacer que ésta sea irreconocible al llegar al receptor. Por tanto, es necesario un filtro inverso que deshaga esos efectos y nos proporcione una señal similar a la original. Esta tarea de traducción de señales es conocida como ecualización y varios trabajos se han acercado a ella con redes de neuronas recurrentes (Ortiz Fuentes y Forcada, 1997; Cid-Sueiro, 1994; Kechriotis, 1994).
- **Codificación del habla**: Se trata de una técnica de compresión de una señal de voz para poder transmitirse con el menor número posible de bits por segundo. Es considerada "codificación predictiva" y consiste en enviar la diferencia entre el valor real y el valor predicho.
- **Reconocimiento del habla**: Tareas de traducción o clasificación de secuencias.
- **Inferencia gramatical**: Ante un conjunto de cadenas pertenecientes a un determinado lenguaje, se trata de estudiar la inferencia de un modelo que describa de manera correcta ese lenguaje. Las redes de neuronas recurrentes han proporcionado en este campo buenos resultados (Carrasco, 2000; Castaño, 1995; Cleeremans, 1989).

- **Control de sistemas**: Las redes de neuronas recurrentes pueden ser también entrenadas para controlar un sistema real en el que la salida siga un determinado patrón temporal (Puskorius, 1994).

Capítulo 4

Casos Prácticos

4.1. Ejemplo 1: Detección de patrones utilizando RNN

Tomemos el ejemplo de tratar de predecir datos secuenciales basados en series de tiempo. En este caso, vamos a intentar predecir la producción de leche en diferentes épocas del año. Comencemos examinando nuestros datos para comprenderlos:

```
#Borrar el espacio de trabajo
rm(list = ls())
#Cargar los paquetes necesarios.
require(rnn)
```

```
#Función que será utilizada después
#Creación de conjunto de datos
#de entrenamiento y prueba
dataset <- function(data){
  x <- y <- c()
  for (i in 1:(nrow(data)-2)){
    x <- append(x, data[i, 2])
    y <- append(y, data[i+1, 2])
  }
  #Creación de nuevo DataFrame
  output <- cbind(x,y)
  return(output[1:nrow(output)-1,])
  }
```

Al trabajar con datos de series temporales, tendremos que realizar una cantidad significativa de transformaciones de datos. Particularmente, debemos crear variables X e Y que sean ligeramente diferentes de los datos dados. A partir de la función *dataset*(), creamos una nueva variable X, que es el paso de tiempo t, a partir de la variable Y original. Hacemos una nueva variable Y que es $t + 1$ a partir de la variable Y original. Posteriormente truncamos los datos en una fila, de modo que eliminamos la observación que falta. A continuación,vamos a cargar y visualizar los datos, los cuales vamos a mostrar

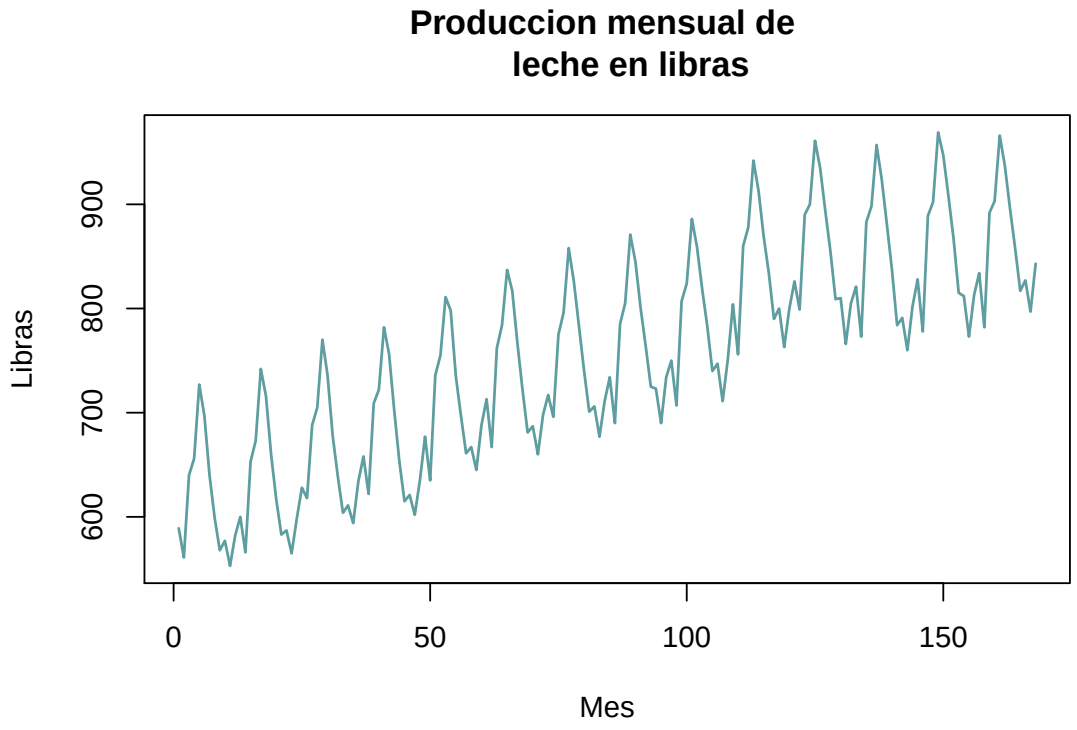
en las dos siguientes figuras:

Vamos a cargar los datos en R. Dichos datos representan la producción mensual de leche, especificando el número de libras ganadas por vaca.

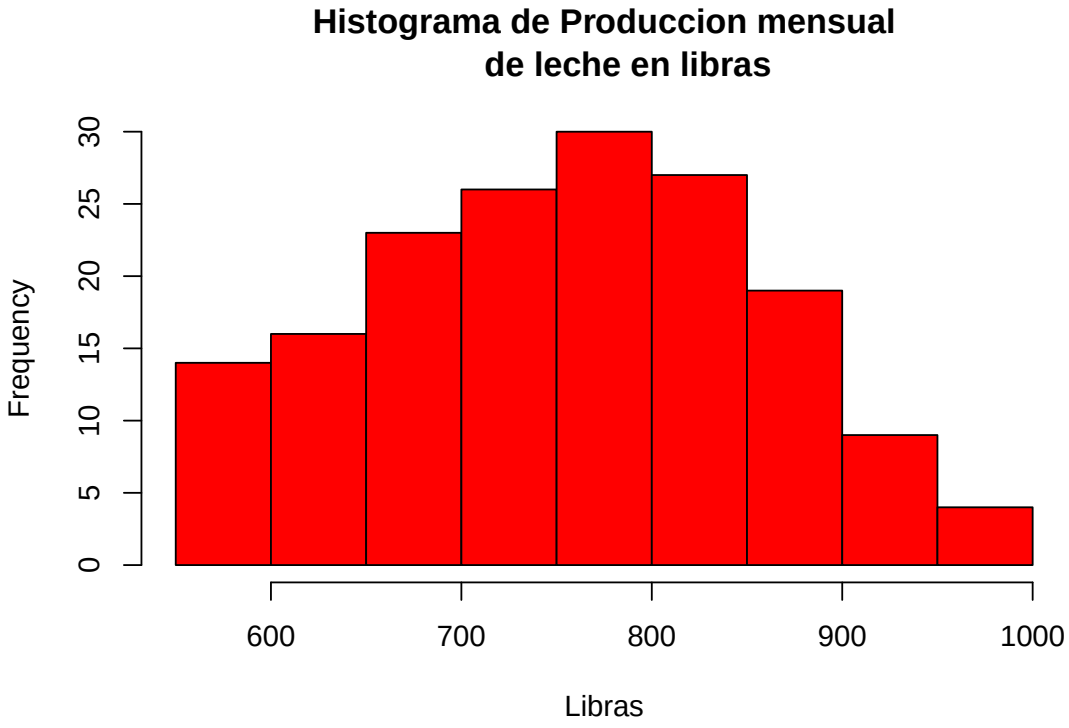
```
#Producción Mensual de Leche: Libras por vaca
data <- read.table("monthly-milk-production-pounds.csv",
                   header = TRUE, sep = ",")
```

A coninuación dibujamos la secuencia y el histograma de los datos:

```
#Dibujamos la secuencia
plot(data[,2], main = "Produccion mensual de
     leche en libras", xlab = "Mes",
     ylab = "Libras",lwd = 1.5, col = "cadetblue", type = "l")
```




```
#Dibujamos el histograma
hist(data[,2], main = "Histograma de Produccion mensual
     de leche en libras", xlab = "Libras", col = "red")
```



Se puede destacar que nuestros datos presentan una cierta asimetría hacia la izquierda.

Ahora que hemos entendido visualmente nuestros datos, avancemos y preparemos nuestros datos para ingresarlos en el modelo de RNN:

```
#Crear Test and Training Sets
newData <- dataset(data = data)
```

```
#Crear Test and Train Data
rows <- sample(1:120, 120)
trainingData <- scale(newData[rows, ])
testData <- scale(newData[-rows, ])
```

Se recomienda utilizar la escala de máximo-mínimo antes de ingresar los datos en un modelo RNN, ya que esto ayuda significativamente a reducir los errores provenientes de una red de neuronas dada. Similar a la normalización estándar, la escala de máximo-mínimo reduce significativamente el rango de su conjunto de datos de entrada, pero lo hace tranformando los valores observados al intervalo $[0, 1]$.

Tras haber realizado este paso, podemos introducir nuestros datos. Evidentemente se puede experimentar con los parámetros, no obstante, se ha entrenado la red para un buen rendimiento. A continuación se va a evaluar el rendimiento del modelo y los resultados de las pruebas, que se mostrarán en las dos siguientes figuras:

En primer lugar, lo pasamos todo a la escala de máximo-mínimo y usaremos estos datos como datos de entrenamiento:

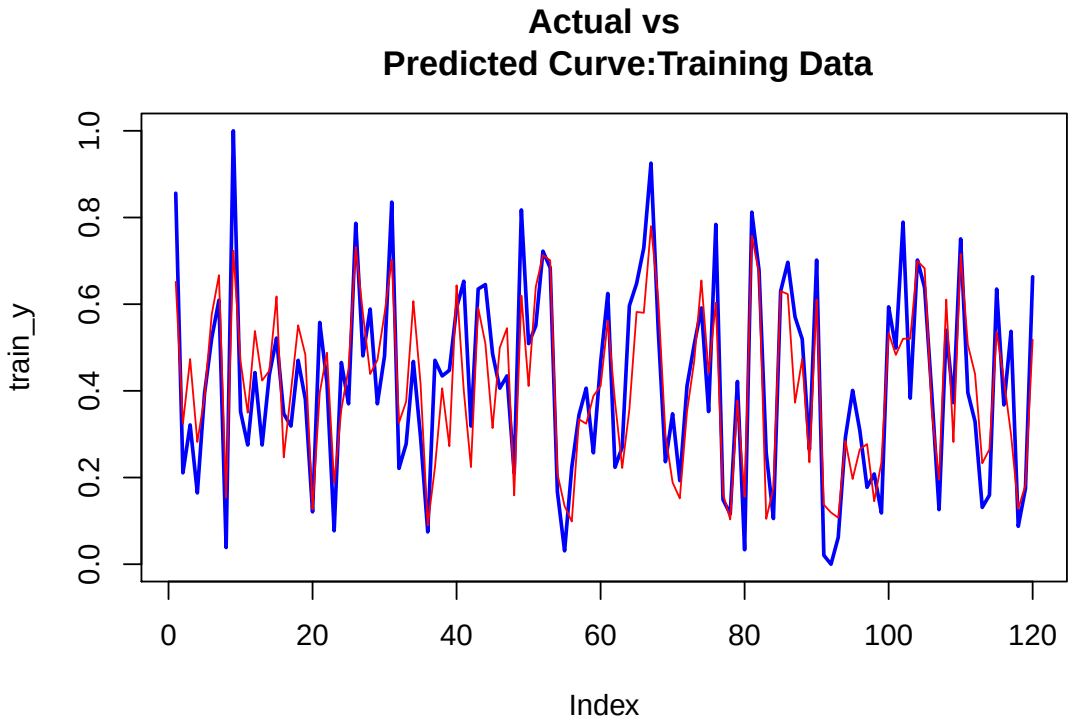
```
x <- trainingData[,1]
y <- trainingData[,2]

train_x <- (x - min(x))/(max(x)-min(x))
train_y <- (y - min(y))/(max(y)-min(y))
```

Se define el modelo de RNN:

```
library(mltools)
RNN <- trainr(Y = as.matrix(train_x),X = as.matrix(train_y),
              learningrate = 0.04, momentum = 0.1,
              network_type = "rnn", numepochs = 700, hidden_dim = 3)
```

Vamos a realizar la comparación gráfica de la curva de predicciones frente a la curva real usando los datos de entrenamiento.

```
y_h <- predictr(RNN, as.matrix(train_x))
plot(train_y, col = "blue", type = "l", main = "Actual vs
     Predicted Curve:Training Data",lwd = 2)
lines(y_h, type = "l", col = "red", lwd = 1)
```




```
mse1= mse (y_h,train_y)
cat("Train ECM: ", mse1)
```

```
## Train ECM:  0.01253554
```

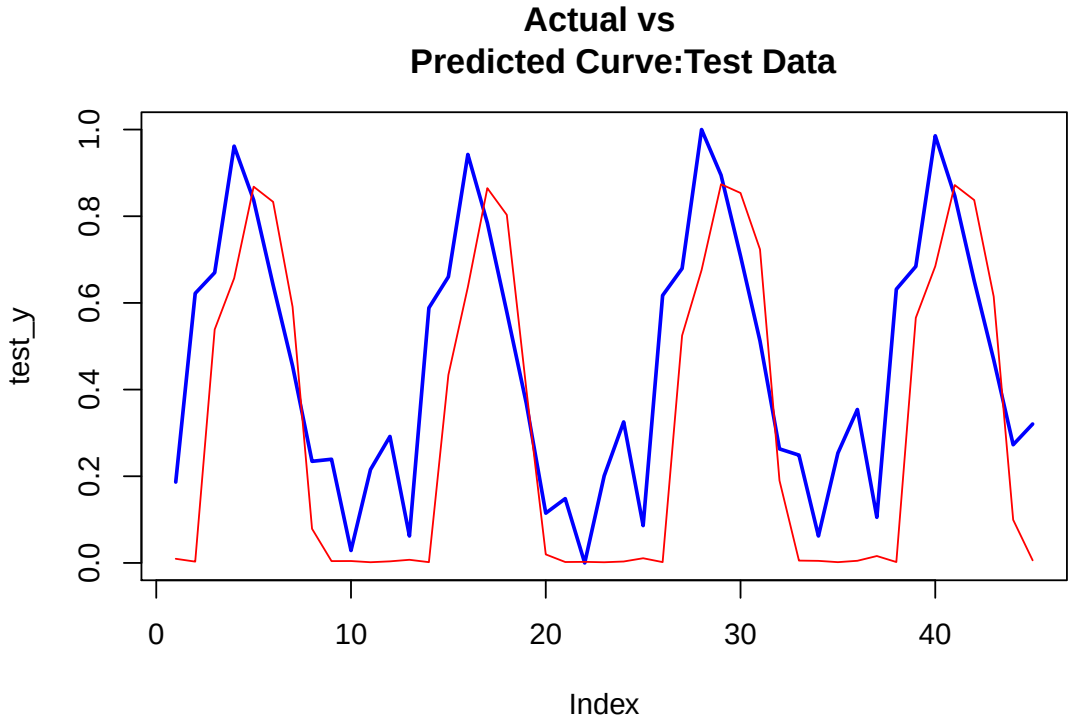
Los datos test serán:

```
testData <- scale(newData[-rows, ])
x <- testData[,1]
```

```
y <- testData[,2]
test_x <- (x - min(x))/(max(x)-min(x))
test_y <- (y - min(y))/(max(y)-min(y))
y_h2 <- predictr(RNN, as.matrix(x))
```

A continuación compararemos las gráficas de la curva de predicciones frente a la curva real usando los datos test.

```
plot(test_y, col = "blue", type = "l", main = "Actual vs
     Predicted Curve:Test Data ", lwd = 2)
lines(y_h2, type = "l", col = "red", lwd = 1)
```




```
mse2= mse(y_h2, test_y)
cat("Test ECM: ", mse2)
```

```
## Test ECM:  0.0675831
```

Respectivamente, el conjunto de entrenamiento y prueba tiene error cuadrático medio (ECM) de 0.0125355 y 0.0675831 respectivamente. Aunque el ECM para el conjunto de entrenamiento es más bajo, esto probablemente sea solo porque el conjunto de entrenamiento es significativamente más grande que el conjunto de prueba. Podemos ver cómo el rendimiento de la prueba es menos preciso que el conjunto de entrenamiento al comparar visualmente ambas.

La curva real está en azul y la curva de predicciones está en rojo. como se puede observar, la curva real en los conjuntos de entrenamiento y prueba muestra una varianza más alta que la que el modelo RNN es capaz de conseguir.

4.2. Ejemplo 2: Cómo usar RNN con estado para modelar secuencias largas de manera eficiente

Para realizar este ejemplo nos ayudaremos de la librería keras. Así pues, se comentará brevemente algunas definiciones y propiedades necesarias para la comprensión de esta librería:

- **Definición**: Una **API** es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. API es el acrónimo inglés *"Application Programming Interface"*, es decir, *"Interfaz de Programación de Aplicaciones"*. Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados.

Keras es una **API** de redes de neuronas de alto nivel desarrollada con el objetivo de permitir una experimentación rápida. Poder pasar de la idea al resultado con el menor retraso posible es clave para hacer una buena investigación. Keras tiene las siguientes características:

1. Permite que el mismo código se ejecute en la CPU o en la GPU, sin problemas.
2. Una API es fácil de usar, además, facilita la creación rápida de prototipos de modelos de aprendizaje profundo.
3. Soporte integrado para redes convolucionales (para visión por computadora), redes recurrentes (para procesamiento de secuencia) y cualquier combinación de ambas.
4. Es capaz de ejecutarse sobre múltiples back- end , incluidos TensorFlow , CNTK o Theano .

Dicho todo esto, se expondrá a continuación el ejemplo en cuestión:

Evidentemente, en primer lugar, cargamos nuestra librería keras.

```
library(keras)
```

Ya que estamos usando el modelo RNN, fijaremos tsteps como 1. Además, fijaremos el tamaño de los grupos batch_size en 25 y el número de etapas en 25 también, para evitar saturar el ordenador. Finalmente, se fijará el término lahead en 1, el cual hace referencia al número de elementos que previamente se utilizarán para hacer la predicción.

```
tsteps <- 1
batch_size <- 25
epochs <- 25
lahead <- 1
```

A continuación, se va a generar una serie temporal de coseno absoluto, donde la amplitud irá disminuyendo exponencialmente. Los argumentos que aparecerán en esta función serán los siguientes:

- **amp**: amplitud de la función del coseno.
- **period**: período de la función del coseno.
- **x0**: x inicial de la series temporales.

- **xn**: x final de la series temporales.
- **step**: paso de la discretización de las series temporales.
- **k**: tasa exponencial.

```
gen_cosine_amp <- function(amp = 100, period = 1000,
                           x0 = 0, xn = 50000, step = 1, k = 0.0001) {
  n <- (xn-x0) * step
  cos <- array(data = numeric(n), dim = c(n, 1, 1))
  for (i in 1:length(cos)) {
    idx <- x0 + i * step
    cos[[i, 1, 1]] <- amp * cos(2 * pi * idx / period)
    cos[[i, 1, 1]] <- cos[[i, 1, 1]] * exp(-k * idx)
  }
  cos
}
```

Se mostrará en pantalla los datos que se van generando y los estados de entrada:

```
cat('Generating Data...\n')
```

```
## Generating Data...
```

```
cos <- gen_cosine_amp()
cat('Input shape:', dim(cos), '\n')
```

```
## Input shape: 50000 1 1
```

La salida esperada seguirá la siguiente función:

```
expected_output <- array(data = numeric(length(cos)),
                         dim = c(length(cos), 1))
for (i in 1:(length(cos) - lahead)) {
  expected_output[[i, 1]] <- mean(cos[(i + 1):(i + lahead)])
}
```

Se mostrará en pantalla el estado de salida:

```
cat('Output shape:', dim(expected_output), '\n')
```

```
## Output shape: 50000 1
```

A continuación, vamos a crear el modelo. En primer lugar, se definirá el modelo como un modelo de keras que utilice secuencias, y por eso se utiliza el comando keras_model_sequential()

```
cat('Creating model:\n')
```

```
## Creating model:
```

```
model <- keras_model_sequential()
```

Como queremos utiliar un modelo de red recurrente específica, que es el modelo LSTM previamente estudiado en la parte teórica, se utilizará el comando layer_lstm.

```
model %>%
  layer_lstm(units = 50, input_shape = c(tsteps, 1), batch_size = batch_size,
             return_sequences = TRUE, stateful = TRUE) %>%
  layer_lstm(units = 50, return_sequences = FALSE, stateful = TRUE) %>%
  layer_dense(units = 1)
model %>% compile(loss = 'mse', optimizer = 'rmsprop')
```

Posteriormente, vamos a entrenar nuestro modelo en el número de etapas que definimos al principio.

```
cat('Training\n')
```

```
## Training
```

```
for (i in 1:epochs) {
  model %>% fit(cos, expected_output, batch_size = batch_size,
                epochs = 1, verbose = 1, shuffle = FALSE)

  model %>% reset_states()
}
```

Las predicciones son:

```
cat('Predicting\n')
```

```
## Predicting
```

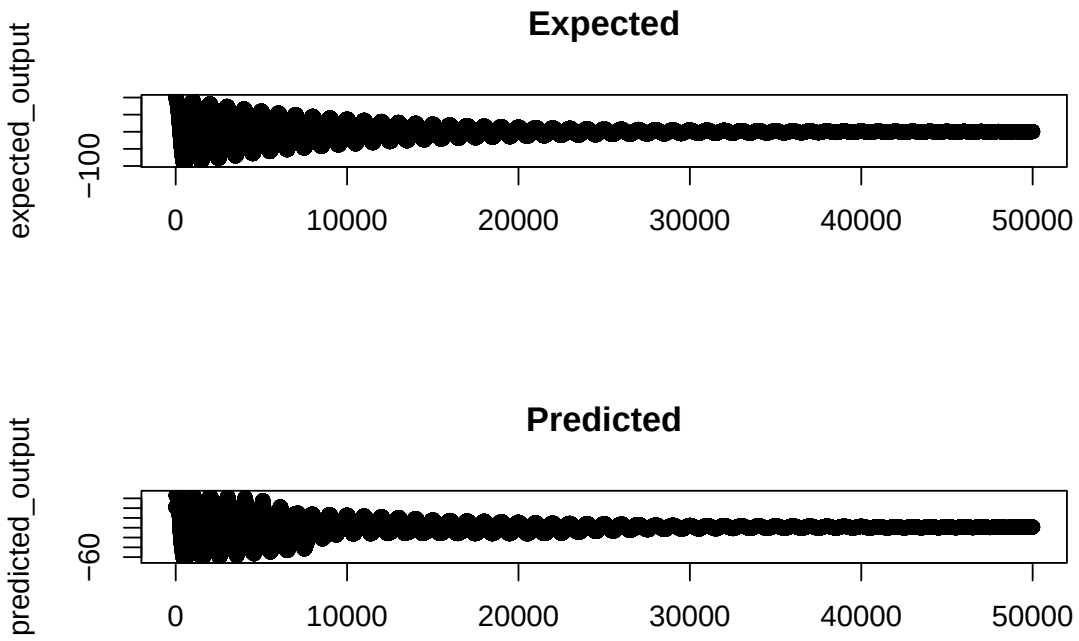
```
predicted_output <- model %>% predict(cos, batch_size = batch_size)
```

Finalmente, se graficará los resultados y se comparará ambas gráficas.

```
cat('Plotting Results\n')
```

```
## Plotting Results
```

```
op <- par(mfrow=c(2,1))
plot(expected_output, xlab = '')
title("Expected")
plot(predicted_output, xlab = '')
title("Predicted")
```




```
par(op)
```

Como se puede observar, si comparamos los valores esperados con los valores que se han obtenido en la predicción, se puede concluir que se aprecia un buen comportamiento del modelo predictivo, ya que las predicciones tienden a exhibir un comportamiento muy similar al de la serie analizada.

Apéndice A

¿Pueden ayudarnos las redes de neuronas con el virus COVID-19?

No es un sueño, es una realidad, no es algo que le ocurra al vecino, o a otros países, el mundo entero se encuentra sumergido en una crisis sanitaria global proveniente de un virus, ya más que conocido, COVID-19.

Estamos hablando de una afección vírica con una alta tasa de contagio, y que, debido a su reciente aparición y propagación, presenta muchas incógnitas.

Por el bien social, necesitamos acelerar el processo de detección de dicho virus, para poder evitar así, en la medida de lo posible, su transmisión. Así que, tras la realización de este trabajo, nos preguntamos si las redes de neuronas recurrentes, o, en su defecto, algún modelo de red de neuronas artificiales nombradas en este trabajo puede ser de ayuda para este difícil cometido.

Numerosas universidades de todo el mundo están realizando laboriosos estudios para agilizar la detección del COVID-19. Así pues, el investigador Enrique Blanco nos contó el pasado 20 de mayo en un apartado web de Telefónica llamado "Think Big/Empresas" que se está haciendo uso de toses o sibilancias respiratorias para poder hacer una clasificación. Pero, ¿cómo abordamos un problema de este tipo?. Es conocido que una de las maneras de discretizar la naturaleza u origen de los sonisdos es mediante la clasificación de imágenes. Para ello, se estan haciendo pruebas con **redes de neuronas convolucionales**, y mediente fine-tuning, o en español, "ajuste fino", que consiste en hacer un ajuste más fino y entrenar también algunas de las capas finales de la base convolucional del modelo usado para la extracción de ciertas características, de forma que, se obtendrán buenas precisiones en la tarea ahorrando tiempo de entrenamiento.

Por otro lado, según un reporte del *MIT Technology Review*, científicos de la Universidad de Waterloo y la firma de Inteligencia Artificial DarwinAI, residente en Canadá, han desarrollado un modelo de red neuronal convulacional de acceso abierto llamada **COVID-net**, que podría ayudar al diagnóstico del virus.

A día de hoy, se está entrenando dicho modelo pare reconocer imágenes en

radiografías de torax. Pese a que, por ahora, este modelo es el único que se ha puesto a disposición del público, ya es un avance de que los modelos de redes de neuronas podrían ayudarnos a acelerar la detección de este virus, y, por lo tanto, evitar su propagación.

Bibliografía

[1] Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W. and Iannone, R. 2019. *Rmarkdown: Dynamic documents for r*.

[2] Aquino, R.R. de, Carvalho Jr, M.A. de and Souza, B.A. de 2001. Redes neurais recorrentes: Despacho hidrotérmico, cálculo dos preços marginais. (2001).

[3] ASPERTI, A. RETI neurali iterative per la generazione di immagini.

[4] Chen, G. 2016. A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv:1610.02583*. (2016).

[5] Connor, J.T., Martin, R.D. and Atlas, L.E. 1994. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*. 5, 2 (1994), 240–254.

[6] Dahl, D.B., Scott, D., Roosen, C., Magnusson, A. and Swinton, J. 2019. *Xtable: Export tables to latex or html*.

[7] FERRI, M. and CILIEGI, F. TOPOLOGIE non convenzionali per reti di neuroni artificiali.

[8] Fongo, D., Chesani, F. and Cattelani, L. Previsione del declino funzionale tramite l'utilizzo di reti neurali ricorrenti.

[9] Freire, A.L., Menezes Jr, J.M.P. de and Barreto, G.A. 2009. Redes neurais recorrentes para predição recursiva de séries temporais caóticas: Um estudo comparativo. *IX CBRN. Ouro Preto*. (2009), 25–28.

[10] II, T. Beysolow 2017. *Introduction to Deep Learning Using R*. Apress.

[11] Le, Q.V. and others 2015. A tutorial on deep learning part 2: Autoencoders, convolutional neural networks and recurrent neural networks. *Google Brain*. (2015), 1–20.

[12] Luque-Calvo, P.L. 2017. *Escribir un trabajo fin de estudios con r markdown*. Disponible en http://destio.us.es/calvo.

[13] Omlin, C.W. and Giles, C.L. 1996. Extraction of rules from discrete-time recurrent neural networks. *Neural networks*. 9, 1 (1996), 41–52.

[14] Pascanu, R., Gulcehre, C., Cho, K. and Bengio, Y. 2013. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*. (2013).

[15] Pascanu, R., Mikolov, T. and Bengio, Y. 2013. On the difficulty of training recurrent neural networks. *International conference on machine learning* (2013), 1310–1318.

[16] R Core Team 2016. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing.

[17] RStudio Team 2015. *RStudio: Integrated development environment for r*. RStudio, Inc.

[18] Schuster, M. and Paliwal, K.K. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*. 45, 11 (1997), 2673–2681.

[19] Techopedia 2017. "Definition - what does business intelligence (bi) mean?". Disponible en https://www.techopedia.com/definition/345/business-intelligence-bi.

[20] Wickham, H. 2019. *Stringr: Simple, consistent wrappers for common string operations*.

[21] Wickham, H., Chang, W., Henry, L., Pedersen, T.L., Takahashi, K., Wilke, C. and Woo, K. 2019. *Ggplot2: Create elegant data visualisations using the grammar of graphics*.

[22] Wickham, H., François, R., Henry, L. and Müller, K. 2019. *Dplyr: A grammar of data manipulation*.

[23] Williams, R.J. and Zipser, D. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*. 1, 2 (1989), 270–280.

[24] Xie, Y. 2019. *Knitr: A general-purpose package for dynamic report generation in r*.

[25] Zaremba, W., Sutskever, I. and Vinyals, O. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*. (2014).