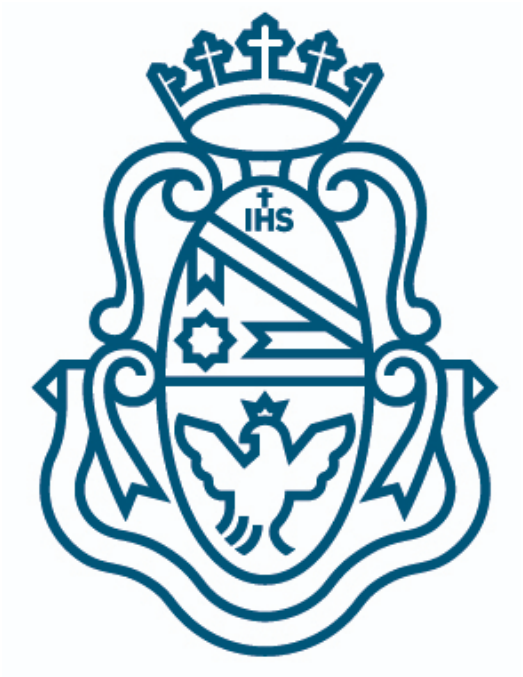
Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales



Proyecto Final Integrador

"Predicción de cantidad de defectos graves en vehículos utilitarios en planta automotriz"

Gerardo A. Collante

Supervisor
Dr. Ing. Orlando Micolini

4 de enero de 2021

Índice general

1 Motivación

El *machine learning* se ha erigido como un campo más en el mundo de las tecnologías de la información, sumado a su vertiginoso crecimiento y amparado bajo la constante mejora del *hardware* ha hecho que su popularidad se dispare.

Más allá del todo el *marketing* que envuelve a la tecnología, es innegable que los años venideros y mejoras en todos los campos serán en gran parte a la IA. Por tanto en búsqueda de mejorar profesionalmente emprendí este proyecto para a través de la práctica y la teoría obtener las herramientas necesarias para poder aspirar a un puesto como ingeniero de inteligencia artificial una vez finalizada mi etapa universitaria.

2 Objetivo

Se desea realizar un modelo de *machine learning* capaz de predecir la cantidad de defectos graves utilizando como datos de entrada los defectos anteriores (de menor gravedad usualmente) considerando una ventana de tiempo a determinar.

En funcionamiento es muy similar a lo que se conoce como *forecasting*, utilizado generalmente en la predicción del clima.

3 Marco teórico

Hagamos algunas definiciones para ponernos en contexto del campo sobre el cual este proyecto integrador será desarrollado.

3.1 Inteligencia Artificial

> La *Inteligencia Artificial (Artificial Intelligence)* se define como el estudio de los "agentes inteligentes", i.e. cualquier dispositivo que perciba su entorno y tome medidas que maximicen sus posibilidades de lograr con éxito sus objetivos.
>
> Poole et al. [1]

Esta definición nos da la idea de que la IA es un sistema reactivo, que reacciona a cambios externos y actúa en consecuencia.

3.1.1 Aprendizaje automático

> El *aprendizaje automático (Machine Learning)* es el estudio científico de algoritmos y modelos estadísticos que los sistemas informáticos utilizan para realizar una tarea específica sin utilizar instrucciones explícitas, sino que se basan en patrones e inferencia. Es visto como un subcampo de inteligencia artificial. Los algoritmos de aprendizaje automático crean un modelo matemático basado en datos de muestra, conocidos como "datos de entrenamiento", para hacer predicciones o decisiones sin ser programado explícitamente para realizar la tarea.
>
> Bishop [2]

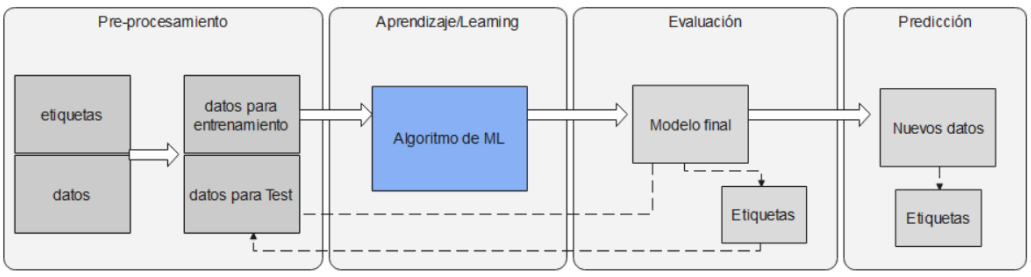


Figura 1: Diagrama de flujo de una aplicación de *Machine Learning*.

Por tanto el Aprendizaje Automático es la generación de un modelo de predicción de salida a partir de grandes cantidades de datos de entrada, realizando un tratamiento de los mismos a través de diferentes etapas bien definidas, como se pueden apreciar en la Fig. 1, las cuales iremos desarrollando en diferentes secciones.

Es importante destacar la independencia del aprendizaje automático al momento de tomar decisiones a partir de los datos proporcionados sin intervención externa, es decir que no hay una especificación de reglas que dictan cómo deben ser tomadas estas decisiones. A su vez, los modelos obtenidos a partir de los algoritmos de *Machine Learning* deben tener la capacidad de predecir a partir de nuevos datos, nunca antes procesados por el modelo, a esto se lo conoce como **generalización**.

3.2 Preprocesamiento

Este punto es vital para cualquier proyecto que utiliza algoritmos de *Machine Learning*, debido a que los datos incluidos en los conjuntos conocidos como *datasets*, no suelen presentarse en condiciones para obtener el óptimo rendimiento de los algoritmos de aprendizaje. Estos datos suelen estar desbalanceados, haya faltantes o sean demasiado ruidosos, etc.

Por lo tanto, una vez que se obtiene el *dataset* de entrada, es primordial investigar, limpiar y transformar los datos con diversas técnicas, de forma que presentemos un conjunto de datos que esté en condiciones de ser entrenado y luego el modelo resultante, al momento de ser probado con datos desconocidos, tenga un desempeño óptimo.

Para lograr este objetivo se aplican técnicas tales como normalización, reescalado, reducción de dimensionalidad, discretización, tratamiento de anomalías y *outliers*. También de ser necesario se utilizan algoritmos de Aprendizaje no supervisado.

3.3 Aprendizaje supervisado

El aprendizaje supervisado es el enfoque más utilizado y mejor entendido para el aprendizaje automático. Implica una entrada y salida para cada pieza de datos en su *dataset*.

Por ejemplo, una entrada podría ser una imagen y la salida podría ser la respuesta a "¿es esto un gato?". En el aprendizaje supervisado siempre hay una distinción entre el conjunto de entrenamiento o *training* para el cual se nos proporciona la etiqueta (o *label*), y el conjunto de test para el cual la etiqueta debe ser inferida. El algoritmo de aprendizaje debe ajustar el modelo predictivo al *dataset* de entrenamiento y usamos el conjunto de test

para evaluar la capacidad de generalización. El aprendizaje supervisado es ideal para tareas donde el modelo necesita predecir resultados.

Agregar validación, test, etc

features

Estos problemas de predicción podrían involucrar el uso de estadísticas para predecir un valor (por ejemplo, $20kg$, \$1498, $0.80cm$) o categorizar datos basados en clasificaciones dadas (por ejemplo, "gato", "verde", "feliz") [3]. El siguiente paso es profundizar en las dos categorías de aprendizaje supervisado que existen: clasificación y regresión.

3.3.1 Clasificación

En clasificación, la etiqueta es discreta, por ejemplo `Spam` y `No Spam`. En otras palabras, se proporciona una distinción clara entre las categorías.

Es más, es importante indicar que estas categorías son nominales y no ordinales. Las variables nominales y ordinales son ambas subcategorías de las variables categóricas. Las variables ordinales tienen asociado un orden, por ejemplo, las tallas de las camisetas "$XL > L > M > S$". Por el contrario, las variables nominales no implican un orden, por ejemplo, no podemos asumir (en general) "$naranja > azul > verde$" [4].

multilabel-etc

Elegir entre dos categorías se denomina **clasificación binaria**, como lo es el ejemplo de `Spam` y `No Spam`, mientras que elegir entre más de dos categorías se denomina **clasificación multiclase**.

Un ejemplo de clasificación multiclase podría ser clasificar un conjunto de imágenes de frutas, donde habrá manzanas, naranjas y peras. Es importante resaltar que si en el dataset de entrenamiento no aparece determinada categoría (por ejemplo bananas), nuestro modelo será incapaz de reconocer esa fruta.

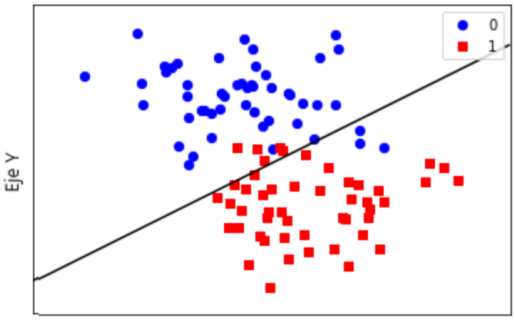


Figura 2: Ejemplo de clasificación binaria.

Para poder apreciar el concepto de clasificación binaria, en la Fig. 2 [4] se han graficado datos bidimensionales, es decir que cada dato tiene dos valores asociados de acuerdo a los ejes X e Y. El dataset cuenta con 100 muestras, las cuales están divididas en dos clases: ceros (círculos azules) y unos (cuadrados rojos).

El modelo a utilizar para la predicción será uno de los más conocidos y simples de utilizar en clasificación, **regresión logística**. Éste es un modelo lineal, lo que significa que creará una frontera de decisión que es lineal en el espacio de entrada, en 2D esto quiere decir que generará una línea recta para separar los puntos azules de los rojos.

Como puede observarse, el modelo no es 100% preciso ya que algunos puntos azules están en la categoría de los rojos y viceversa, por eso es que existen diversos modelos y debemos elegir según nuestro criterio cuál de ellos se adecúa mejor a nuestras necesidades.

Debido a la cantidad de diversos algoritmos que existen para clasificación es posible caracterizarlos en función de sus pros y contras como se observa en la Figura 3.

	Ventajas	Desventajas
Naive Bayes	• Simple de entender e implementar . • No necesita una gran cantidad de datos de entrenamiento. • Rápido.	• Supone que cada característica es independiente. • Sufre al tener características irrelevantes.
Regresion logistica	• Simple de entender e implementar. • Rara vez existe sobreajuste. • Rápido de entrenar.	• Es muy díficil lograr que se ajuste a datos no lineales. • Los valores atípicos alteran la precisión del modelo.
KNN	• Eficaz en datasets de varias clases. • Entrenamiento rápido.	• La dimensionalidad del dataset merma el rendimiento. • Lento en fase de predicción.
Arbol de decision	• Robusto a muestras con ruido. • Fácil de interpretar. • Resuelve problemas no lineales.	• Cuando hay muchas etiquetas de clase, los cálculos pueden ser complejos. • Puede sufrir sobreajuste.
Clasificador de bosque aleatorio	• No sufre sobreajuste como el árbol de decisión. • Funciona muy bien en grandes bases de datos. • Maneja automáticamente los valores faltantes.	• Consume mucho tiempo y recursos computacionales. • Díficil de interpretar. • Necesito elegir la cantidad adecuada de árboles.
Máquinas de vectores de soporte (SVC)	• Eficaz en espacios de gran dimensión. • Puede manejar soluciones no lineales. • Robusto al ruido.	• Lento para entrenarse con grandes conjuntos de datos. • Ineficaz si las clases se superponen. • Hay que elegir una buena función de kernel. • Díficil de interpretar al aplicar kernels no lineales.

Figura 3: Ventajas y desventajas de los algoritmos de clasificación.

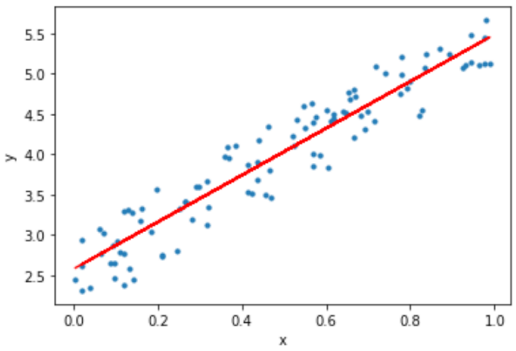


Figura 4: Ejemplo de regresión lineal.

3.3.2 Regresión

En regresión, la etiqueta es continua, es decir una salida real. Por ejemplo, en astronomía, la tarea de determinar si un objeto es una estrella, una galaxia o un cuásar es un problema de clasificación: la etiqueta viene de tres categorías distintas. Por otro lado, podríamos querer estimar la edad de un objeto basándonos en su imagen: esto sería regresión, porque la etiqueta (edad) es una cantidad continua [4].

En los problemas de regresión, tenemos como entradas las variables independientes o explicativas y las salidas o etiquetas son variables continuas. Por lo tanto, los modelos de regresión deben encontrar una relación (función lineal, polinomial, entre otras) que nos permitan predecir la salida.

Para poder apreciar el concepto de regresión lineal, en la Fig. 4 se tienen 100 muestras con sus respectivas etiquetas, entonces lo que hace el algoritmo de regresión lineal es ajustar una línea recta que minimice la distancia (en este caso distancia euclídea) entre los puntos de la muestra y dicha recta. Al obtener la recta, disponemos de los parámetros del modelo como son los coeficientes y la intersección, por ende estamos en condiciones de predecir la salida de nuevas muestras.

Así como pudimos listar las ventajas y desventajas con los distintos algoritmos de clasificación también es aplicable a los de regresión como vemos en la Fig. 5.

	Ventajas	Desventajas
Ridge	• El costo computacional no es mayor que otros algoritmos. • Permite evitar el sobreajuste.	• Se necesita una excelente selección del hiperparámetro alpha. • Incrementa el sesgo.
LASSO	• Evita el sobreajuste. • Selecciona características tendiendo que sus coeficientes sean cero.	• Las características seleccionadas tienen demasiado sesgo. • Si tenemos n datos y p características, LASSO solo seleciona como máximo n características. • El rendimiento de la predicción es peor que para Ridge Regression.
Elastic Net	• Eficaz con muestras de gran dimensión.	• Alto costo computacional en comparación con LASSO o Ridge.
KNN Regresor	• No tiene período de entrenamiento. • Fácil de interpretar. • Permite agregar datos al modelo sin inconvenientes en la precisión del algoritmo.	• La dimensionalidad del conjunto de datos influye mucho en el rendimiento. • Es sensible a datos ruidosos, valores faltantes y outliers. • En grandes datasets se vuelve muy alto el costo computacional para calcular distancias.
Regresor de árbol de decisión	• No sufre sobreajuste como el árbol de decisión. • Funciona muy bien en grandes bases de datos. • Maneja automáticamente los valores faltantes.	• Al trabajar con variables continuas, se pierde mucha información al categorizar. • Necesita variables correlacionadas. • Alto tiempo de entrenamiento. • Se puede volver demasiado complejo.
Máquinas de vectores de soporte (SVC)	• Útil cuando las clases no son linealmente separables.	• Suelen ser ineficientes al momento del entrenamiento.

Figura 5: Ventajas y desventajas de los algoritmos de regresión.

3.4 Aprendizaje no supervisado

A diferencia de lo que sucede en el aprendizaje supervisado (sección 3.3), no disponemos de una salida deseada, tampoco se disponen datos etiquetados o con estructuras definidas. Por lo que el objetivo principal del Aprendizaje no Supervisado es generar esas etiquetas a partir de la información que se extrae de datos proporcionados en el *dataset*, sin tener una referencia de salida.

Un ejemplo de aprendizaje no supervisado podría ser si nos encontramos con un texto extenso y queremos obtener una especie de resumen, de los temas o tópicos relevantes, probablemente de antemano no se sabe cuales son o su cantidad, por lo tanto nos enfrentamos a la situación de no conocer cuales serian las salidas esperadas del modelo. Otros ejemplos clásicos pueden ser agrupar fotografías similares o separación de diferentes fuentes que originan un determinado sonido.

Como se comentó anteriormente en la sección 3.2, en la etapa de preprocesamiento, es muy útil aplicar las técnicas del aprendizaje no supervisado ya que se cuentan con grandes cantidades de datos en contextos no conocidos y lo más importante, sin etiquetar. Entonces suele ser una buena práctica dar un primer paso mediante algoritmos de aprendizaje no supervisado antes de pasar los datos a un proceso de aprendizaje supervisado. Como por ejemplo cuando se realizan transformaciones de datos mediante reescalado o estandarización.

En las próximas subsecciones explicaremos brevemente cada una de las tareas que comprenden el Aprendizaje no Supervisado.

3.4.1 Detección de anomalías

Uno de los primeros pasos a realizar cuando se nos presenta un conjunto de datos, es proceder con la tarea llamada detección de anomalías (*anomaly detection*, AD), o identificación de *outliers* o datos fuera de rango.

Un *outlier* puede ser considerado como un dato atípico en un dataset. O bien un *outlier* es una observación en un dataset que parece ser inconsistente con el resto del conjunto. Johnson 1992.

Tipos de entornos en los que se produce la detección de anomalías:

- AD supervisada:
  - Las etiquetas están disponibles, tanto para casos normales como para casos anómalos.
  - En cierto modo, similar a minería de clases poco comunes o clasificación no balanceada.

- AD semi-supervisada (detección de novedades, *Novelty Detection*)
  - Durante el entrenamiento, solo tenemos datos normales.
  - El algoritmo aprende únicamente usando los datos normales.
- AD no supervisada (detección de *outliers*, *Outlier Detection*)
  - No hay etiquetas y el conjunto de entrenamiento tiene datos normales y datos anómalos.
  - Asume que los datos anómalos son poco frecuentes.
  - Algunos ejemplos típicos de detección de anomalías pueden ser, cuando se quiere detectar intrusos en tráfico de red o bien detectar acciones fraudulentas en transacciones con tarjetas de crédito.

	Ventajas	**Desventajas**
One Class SVM	• Eficiente cuando la dimensionalidad de los datos es demasiado alta.	• Es sensible ante los outliers.
Isolation Forest	• Bajo costo computacional, debido a que no usa medidas de distancia, similitud o densidad del conjunto de datos. • La complejidad crece linealmente gracias al submuestreo del dataset. • Muy útil para escalar grandes conjuntos de datos con variables irrelevantes. • Las anomalías suelen quedar en las partes altas del árbol, por lo que no es necesario construirlo completamente.	

Figura 6: Ventajas y desventajas de los algoritmos de detección de anomalías.

3.4.2 Reducción de dimensionalidad

A menudo las muestras disponibles contienen una gran variedad de características, que pueden dar como resultado un sobreajuste del modelo utilizado, por lo tanto es necesario reducir la dimensionalidad de dichos datos pero manteniendo la información relevante. Al reducir la dimensionalidad no solo se evita el sobreajuste, sino que también se obtiene una mejor visualización de los datos y se reduce el costo computacional.

Uno de los modelos más conocidos y que requieren menor costo computacional es Principal Component Analysis (PCA), pero si lo que estamos buscando es una mejor visualización de los datos y además las características no son lineales, sería recomendable usar T-SNE.

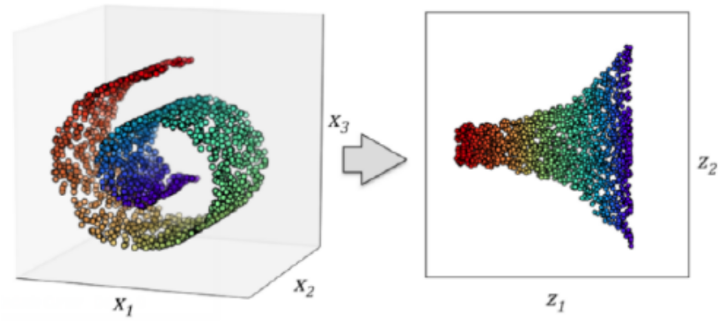


Figura 7: Reducción de dimensionalidad de 3D a 2D.

En la Fig. 7 [5] se muestra un ejemplo de cómo la reducción de dimensionalidad facilita la visualización de un dataset de alta dimensionalidad en una proyección de 1, 2 o 3 dimensiones.

Enumeramos en la Fig. 8 las ventajas y desventajas de los algoritmos disponibles para esta tarea.

	Ventajas	**Desventajas**
PCA	• Simple de implementar. • Es uno de los algoritmos más rápidos de reducción de dimensionalidad.	• No puede detectar características no lineales. • Los datos necesitan ser normalizados.
Isomap	• Puede detectar características no lineales.	• Lento en grandes cantidades de datos.
T-SNE	• Puede detectar características no lineales. • Recomendable cuando se quiere obtener una mejor visualización de un dataset con alta dimensionalidad.	• Computacionalmente costoto y lento.

Figura 8: Pro y contras de algoritmos de reducción de dimensionalidad.

3.4.3 *Clustering*

El *clustering* es una técnica que conceptualmente es simple de comprender, consiste agrupar objetos con características similares. Por lo tanto, obtenemos diferentes grupos llamados clústers, donde en cada uno de ellos están contenidos los datos que son más similares entre ellos que con los que pertenecen a otros clústers, obteniendo de esta forma una útil subdivisión del *dataset*. Como no suele tenerse conocimiento sobre los datos, es decir no están etiquetados, esta técnica pertenece al Aprendizaje no Supervisado.

El algoritmo más simple de *clustering* es K-means, el cual funciona para agrupar datos que se distribuyen en formas esféricas, si se usa la distancia euclídea. y a su vez hay que proporcionar la cantidad k de grupos en los cuales queremos distribuir el *dataset*, por ello se debe tener un conocimiento previo de cuantos clúster se espera tener. Otras alternativas pueden ser, realizar *clustering* jerárquico o *clustering* basados en densidades.

En *clustering* jerárquico, vemos como resultado un dendograma, es decir un diagrama de árbol. A partir de esto, se decide un umbral de profundidad, donde se corta el árbol y de esta forma se obtiene un agrupamientos, por lo tanto a diferencia con K-means, no necesitamos tener información para poder decidir la cantidad de grupos. En la Fig. 9 podemos observar como quedan evidenciados a través del dendograma los diferentes clusters.

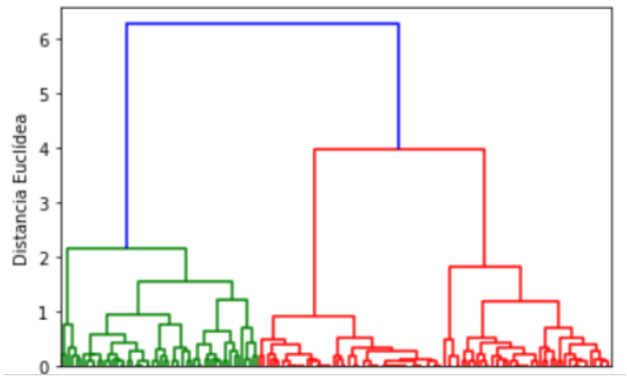


Figura 9: Dendograma generado por *clustering* jerárquico.

En cambio DBSCAN (*Density-based Spatial Clustering of Applications with Noise*), divide el *dataset* buscando las regiones densas de puntos, como podemos observar claramente en Fig. 10. Con esta técnica, tampoco especificamos el número de parámetros a priori, sino que se establecen hiperparámetros adicionales, como lo son la cantidad mínima de puntos y un radio ϵ, para lograr un óptimo funcionamiento.

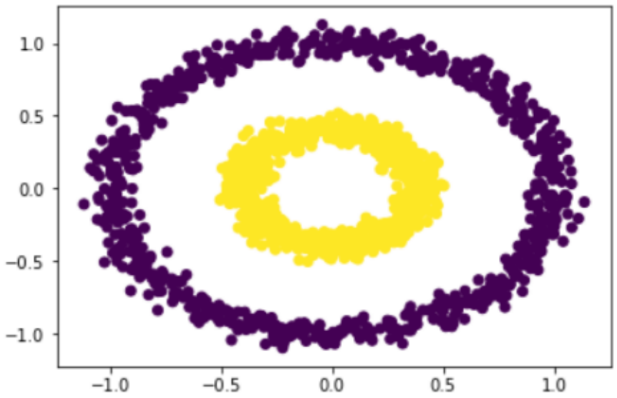


Figura 10: Clustering basado en densidades.

Enumeramos en la Fig. 8 las ventajas y desventajas de los algoritmos disponibles para esta tarea.

	Ventajas	Desventajas
K-Means	• Fácil de implementar. • Rápido.	• Hay que conocer el número de grupos y asumir que los datos están normalizados. • Utiliza la distancia euclídea, por lo que debemos estar seguros de que las variables estén en la misma escala. • Sensible al ruido.
Agglomerative Clustering	• No es necesario indicar el número de grupos a priori. • No es sensible a la elección de la métrica de distancia.	• No es muy eficiente.
Mini Batch K-Means	• Puede agrupar conjuntos de datos masivos. • Reduce el tiempo de cómputo gracias a los mini-batches.	• La calidad de los resultados podría verse reducida respecto a K-means.
Mean Shift	• No es necesario indicar el número de grupos a priori.	• No es escalable para muchos datos.
DBSCAN	• No hay que especificar el número de clusters a priori. • Puede detectar grupos con formas irregulares. • Puede detectar outliers. • Robusto al ruido.	• Sensible a datos con alta dimensión. • No funciona bien cuando los clusters son de densidad variable.

Figura 11: Pros y contras de los algoritmos de *clustering*.

3.5 Selección de algoritmo en base al *dataset*

En la Fig. 12 obtenemos una vista general sobre que hacer con nuestros datos en función a nuestro objetivo.

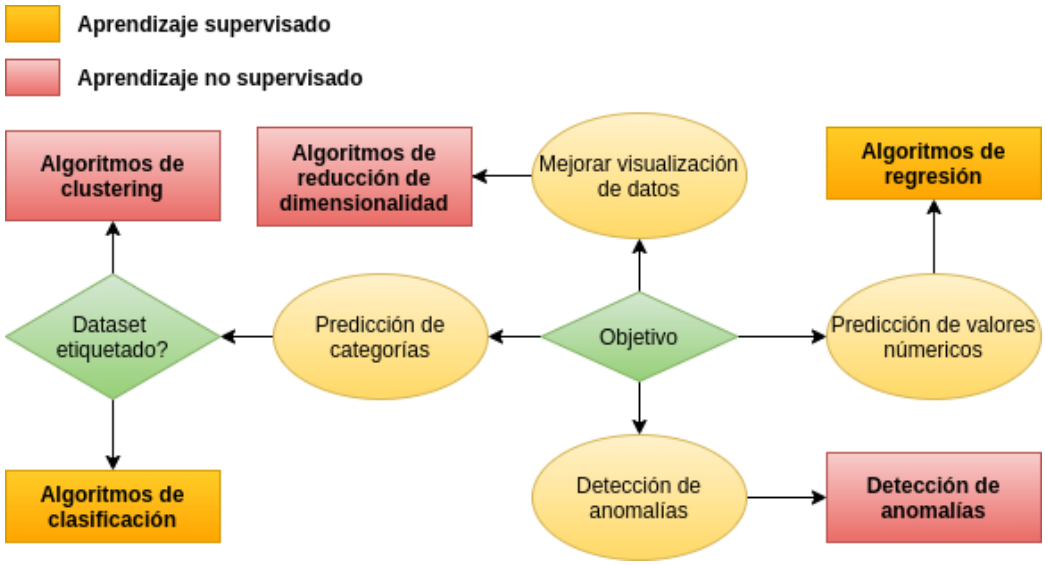


Figura 12: Diagrama general de algoritmos de aprendizaje supervisado y no supervisado.

3.5.1 Algoritmos supervisados

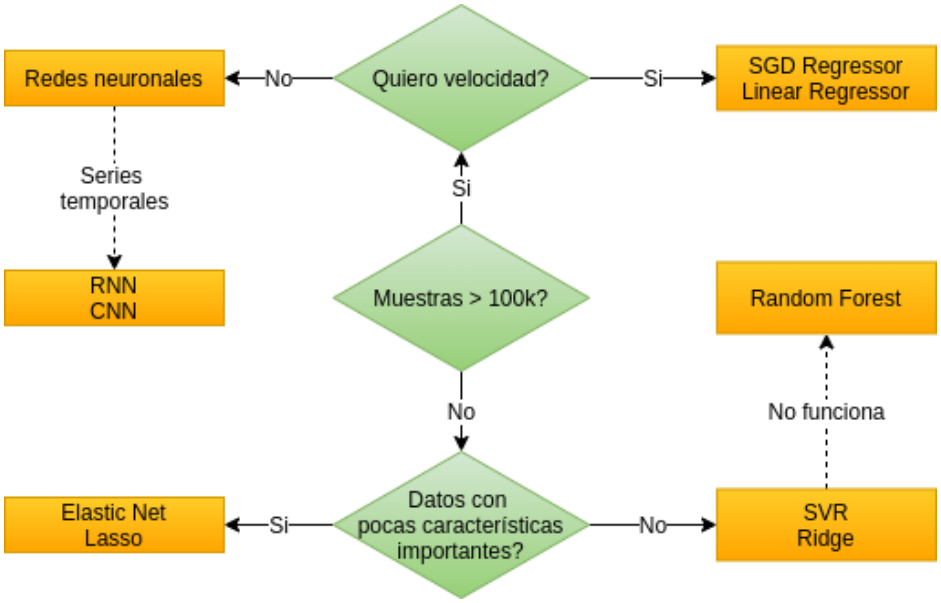


Figura 13: Diagrama general de los algoritmos de regresión.

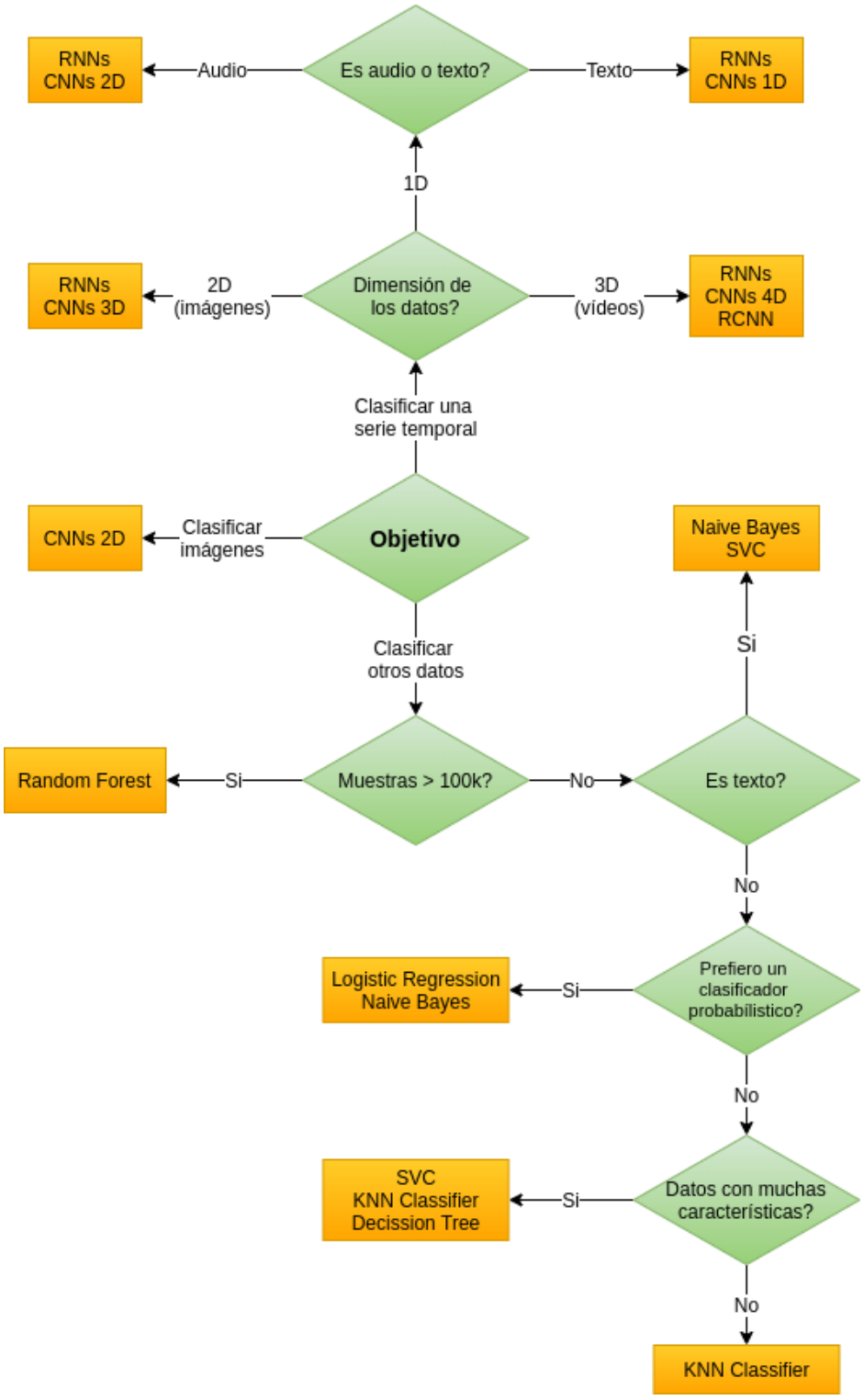


Figura 14: Diagrama general de los algoritmos de clasificación.

3.5.2 Algoritmos no supervisados

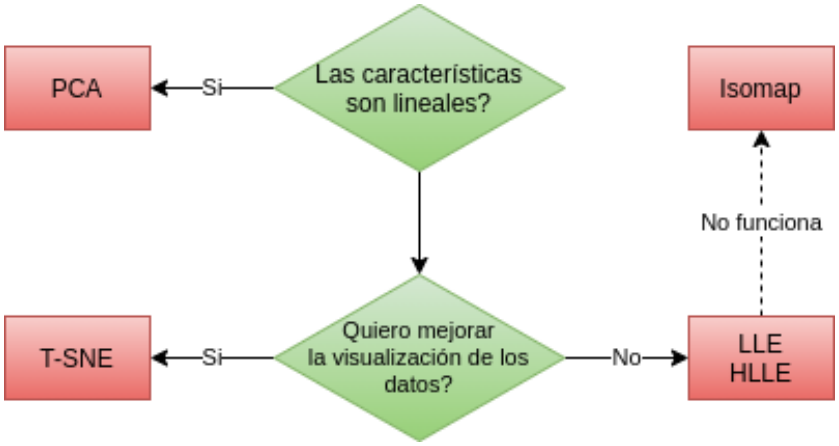


Figura 15: Diagrama general de los algoritmos de reducción de dimensión.

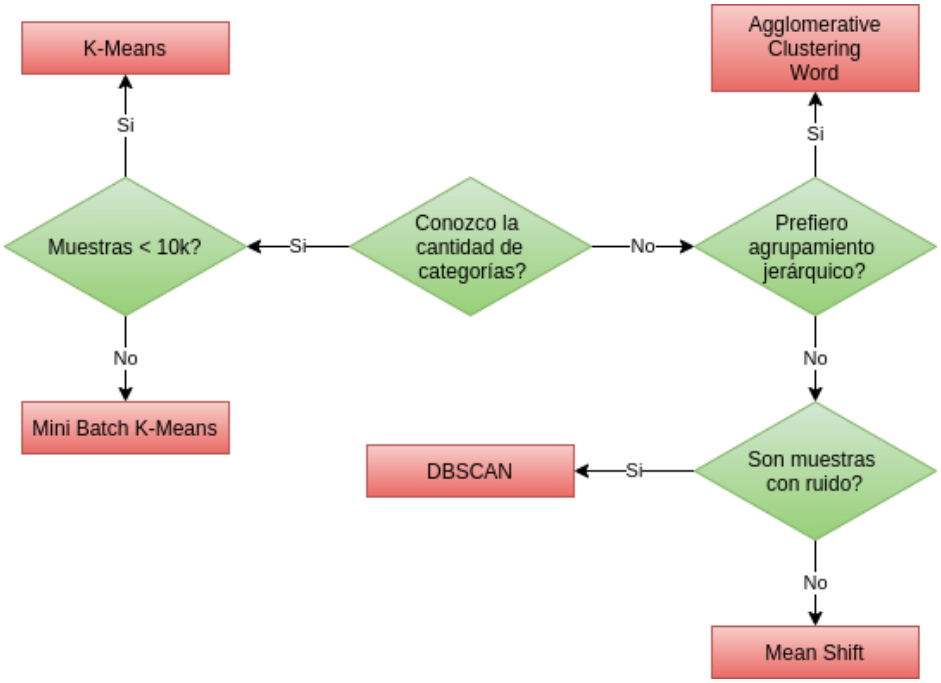


Figura 16: Diagrama general de los algoritmos de *clustering*.

Figura 17: Diagrama general de los algoritmos de detección de anomalías.

4 Redes neuronales

La palabra neuronal es la forma adjetiva de "neurona", y red denota una estructura tipo grafo; por lo tanto, una *Red Neuronal Artificial* es un sistema de computación que intenta imitar (o al menos, está inspirado en) las conexiones neuronales en nuestro sistema nervioso. Las redes neuronales artificiales también se denominan *redes neuronales* o *sistemas neuronales artificiales*. Es común abreviar la red neuronal artificial y referirse a ellas como "NN". [6]

Para que un sistema se considere un NN, debe contener una estructura de grafo dirigida y etiquetada donde cada nodo del gráfico realice un cálculo simple. Según la teoría de grafos, sabemos que un gráfico dirigido consiste en un conjunto de nodos (es decir, vértices) y un conjunto de conexiones (es decir, bordes) que unen pares de nodos.

- Las entradas ingresan a la red.
- Cada conexión lleva una señal a través de las dos capas ocultas en la red.
- Una función final calcula la etiqueta de clase de salida.

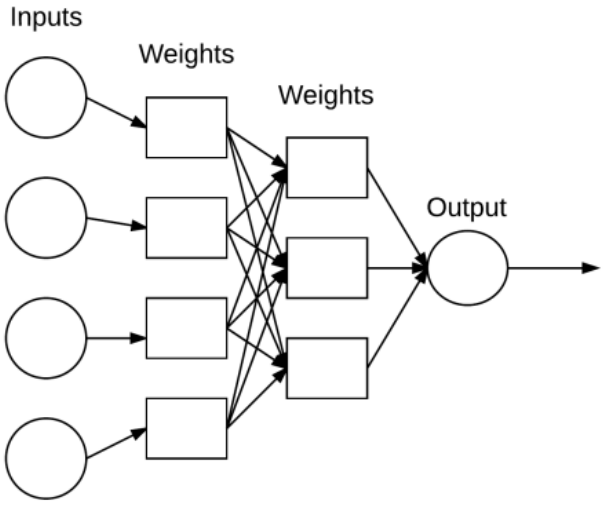


Figura 18: Arquitectura simple de red neuronal. [7]

Cada nodo realiza un cálculo simple. Cada conexión transporta una señal (es decir, la salida del cálculo) de un nodo a otro, marcada por un peso que indica el grado en que la señal se amplifica o disminuye. Algunas conexiones tienen grandes pesos positivos que amplifican la señal, lo que indica que la señal es muy importante al hacer una clasificación. Otros tienen pesos negativos, lo que disminuye la intensidad de la señal, lo que especifica que la salida del nodo es menos importante en la clasificación final. Llamamos a dicho sistema una Red Neuronal Artificial si consta de una estructura de grafo (como en la Figura 18) con pesos de conexión que se pueden modificar utilizando un algoritmo de aprendizaje.

4.1 Relación con la biología

Nuestros cerebros están compuestos por aproximadamente 10 mil millones de neuronas, cada una conectada a unas 10,000 otras neuronas. El cuerpo celular de la neurona se llama soma, donde las entradas (dendritas) y las salidas (axones) conectan el soma con otro (Figura 19).

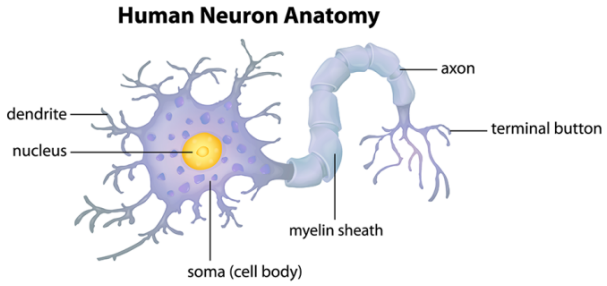


Figura 19: Estructura de una neurona biológica.

Cada neurona recibe entradas electroquímicas de otras neuronas en sus dendritas. Si estas entradas eléctricas son lo suficientemente potentes como para activar la neurona, entonces la neurona activada transmite la señal a lo largo de su axón, transmitiéndola a las dendritas de otras neuronas. Estas neuronas unidas también pueden activarse, continuando así el proceso de transmitir el mensaje. La conclusión clave aquí es que el disparo de una neurona es una operación binaria: la neurona se dispara o no se dispara. No hay diferentes "grados" de disparo. En pocas palabras, una neurona solo se disparará si la señal total recibida en el soma excede un umbral dado. Sin embargo, tenga en cuenta que los ANN simplemente se inspiran en lo que sabemos sobre el cerebro y cómo funciona. El objetivo del aprendizaje profundo no es imitar cómo funcionan nuestros cerebros, sino tomar las piezas que entendemos y permitirnos trazar paralelos similares en nuestro propio trabajo.

4.2 Modelos artificiales

Comencemos por ver un NN básico que realiza una suma ponderada simple de las entradas o *inputs* en la Figura 20. Los valores x_1, x_2 y x_3 son las *inputs* a nuestro NN y generalmente corresponden a una sola fila (es decir, punto de datos) de nuestra matriz de diseño. El valor constante 1 es nuestro sesgo o *bias* que se supone incrustado en la matriz de diseño. Podemos pensar en estas *inputs* como los vectores de características o *features* de entrada a la NN.

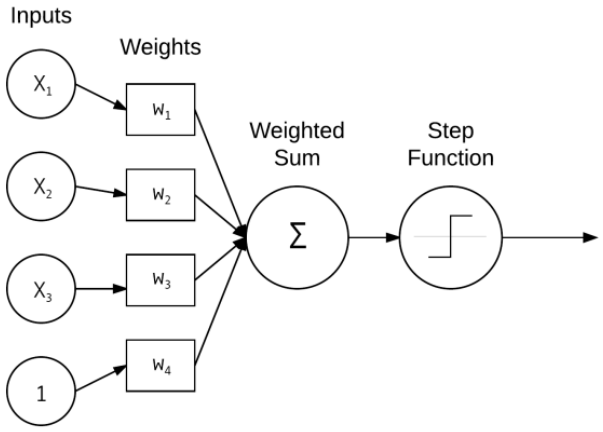


Figura 20: Simple NN.

Cada x está conectada a una neurona a través de un vector de peso W que consiste en $w_1, w_2, \ldots w_n$, lo que significa que para cada entrada x también tenemos un peso asociado w. Finalmente, el nodo de salida a la derecha toma la suma ponderada, aplica una función de activación f (utilizada para determinar si la neurona se "dispara" o no) y genera un valor. Expresando la salida matemáticamente, normalmente encontrarás las siguientes tres formas:

- $f(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$
- $f(\sum_{i=1}^{n} w_ix_i)$
- O $f(net)$, donde $net = \sum_{i=1}^{n} w_ix_i$

4.3 Funciones de activación

La función de activación más simple es la "función de paso", utilizada por el algoritmo Perceptron.

$$f(net) = \begin{cases} 1 & \text{si } net > 0 \\ 0 & \text{si } net \leq 0 \end{cases}$$

Esta es una función de umbral muy simple, sin embargo, aunque es fácil de usar e intuitiva, no es diferenciable, lo cual puede llevar a problemas cuando apliquemos el descenso por gradiente. Por ello se presentan en la Figura 21 diferentes tipos de funciones de activación con sus respectivos gráficos.

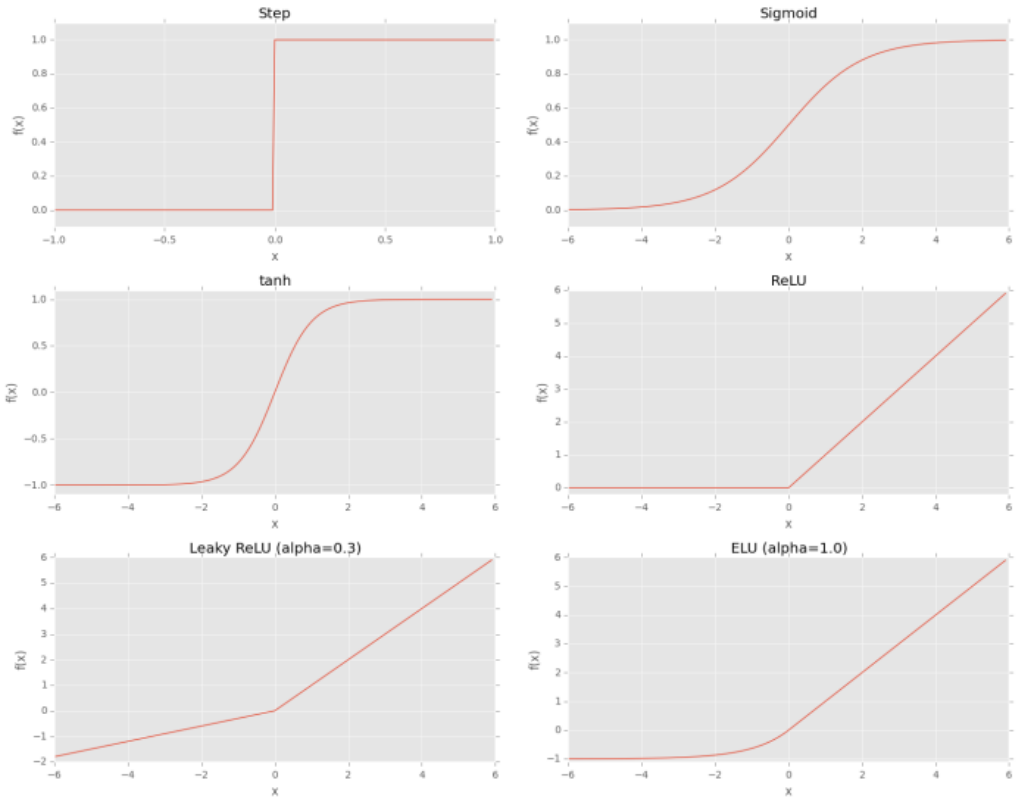


Figura 21: **Arriba-izquierda**: Función escalón. **Arriba-derecha**: Función sigmoidea. **Medio-izquierda**: Tangente hiperbólica. **Medio-derecha**: activación ReLU (función activación más usada en *Deep Learning*). **Abajo-izquierda**: Leaky ReLU, variante de ReLU que permite valores negativos. **Abajo-derecha**: ELU, otra variante de ReLU que obtiene mejor performance que Leaky ReLU.

Una de las funciones de activación más usadas en la historia de la literatura de NN es la función sigmoidea, que sigue la siguiente ecuación:

$$t = \sum_{i=1}^{n} w_i x_i \quad s(t) = \frac{1}{1 + e^{-t}} \tag{1}$$

La función sigmoidea es una mejor opción para el aprendizaje que la función de paso simple, ya que:

1. Es continua y diferenciable en todas partes.

2. Es simétrica alrededor del eje y.

3. Se acerca asintóticamente a sus valores de saturación.

La principal ventaja aquí es que la suavidad de la función sigmoidea hace que sea más fácil diseñar algoritmos de aprendizaje. Sin embargo, hay dos grandes problemas con la función sigmoidea:

1. Las salidas del sigmoide no están centradas en cero.

2. Las neuronas saturadas esencialmente eliminan el gradiente, ya que el delta del gradiente será extremadamente pequeño.

La tangente hiperbólica, o tanh (con una forma similar del sigmoide) también se usó fuertemente como una función de activación hasta fines de la década de 1990. La ecuación para tanh sigue:

$$f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2}$$

La función tanh está centrada en cero, pero los gradientes aún se eliminan cuando las neuronas se saturan. Ahora sabemos que hay mejores opciones para las funciones de activación que las funciones sigmoide y tanh. Específicamente, la Unidad Lineal Rectificada (ReLU), definida como:

$$f(x) = max(0, x) \tag{3}$$

Las ReLU también se denominan "funciones de rampa" debido a cómo se ven cuando se trazan. La función es cero para entradas negativas pero luego aumenta linealmente para positivas valores. La función ReLU no es saturable y también es extremadamente eficiente computacionalmente. Empíricamente, la función de activación ReLU tiende a superar a las funciones sigmoide y tanh en casi todas las aplicaciones. Sin embargo, surge un problema cuando tenemos un valor de cero: no se puede tomar el gradiente.

4.4 Arquitecturas de redes *feedforward*

Si bien hay muchas, muchas arquitecturas NN diferentes, la arquitectura más común es la red hacía adelante o *feedforward*, como se presenta en la Figura 22.

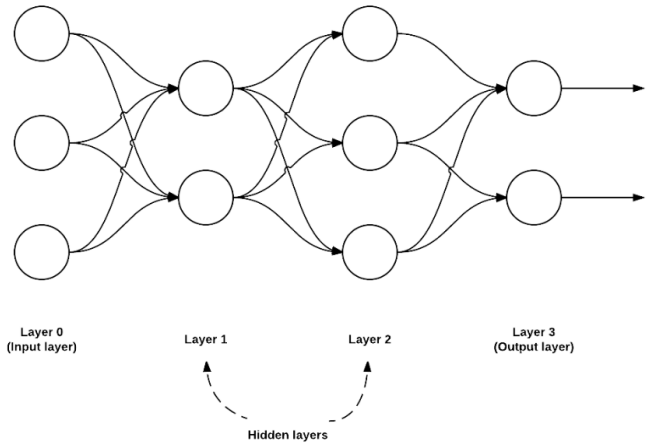


Figura 22: Un ejemplo de una red neuronal *feedforward*.

En este tipo de arquitectura, solo se permite una conexión entre los nodos de los nodos en la capa i a los nodos en la capa $i + 1$ (de ahí el término *feedforward*). No hay conexiones hacia atrás o entre capas permitidas. Cuando las redes de retroalimentación incluyen conexiones de retroalimentación (conexiones de salida que retroalimentan las entradas) se denominan redes neuronales recurrentes.

Para describir una red *feedforward*, normalmente usamos una secuencia de enteros para depositar rápida y concisamente el número de nodos en cada capa. Por ejemplo, la red en la Figura 10.5 anterior es una red de alimentación directa 3-2-3-2:

- La capa 0 contiene 3 entradas, nuestros valores x_i. Estos podrían ser intensidades de píxeles sin procesar de una imagen o un vector de características extraído de la imagen.
- Las capas 1 y 2 son capas ocultas que contienen 2 y 3 nodos, respectivamente.
- La capa 3 es la capa de salida o la capa visible: allí es donde obtenemos la clasificación de salida general de nuestra red. La capa de salida generalmente tiene tantos nodos como etiquetas de clase; un nodo para cada salida potencial.

4.5 Redes multicapa

Las redes multicapas, *i.e.* con varias capas de neuronas pueden ser modeladas matemáticamente como se muestra a continuación. Supongamos W

como la matriz de pesos y el vector b como el vector sesgo o *bias*. Consideremos:

$$z(x) = Wx + b = \sum_{i=1}^{n} w_i x_i + b \tag{4}$$

Además cabe mencionar que la multiplicación punto a punto entre dos matrices de igual dimensión es lo que se conoce como el producto Hadamard.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

Por último definimos la salida de nuestro modelo como:

$$\hat{y} = \sigma(\sum_{i=1}^{n} w_i x_i + b) \tag{6}$$

4.6 Función pérdida

El objetivo del algoritmo de descenso de gradiente es minimizar la función de costo para que nuestro modelo neuronal pueda aprender. Pero antes debemos definir que es la función costo o pérdida [8]. En el cálculo, los máximos (o mínimos) de cualquier función pueden ser descubiertos por:

1. Tomando la derivada de primer orden de la función e igualándola a 0. El punto encontrado de esta manera puede ser el punto de máximo o mínimo.

2. Sustituimos estos valores (el punto que acabamos de encontrar) en la derivada de segundo orden de la función y si el valor es positivo, *i.e.*> 0, entonces ese punto (s) representa el punto (s) de mínimos locales o máximos locales.

Necesitamos cerrar la brecha entre la salida del modelo y la salida real. Cuanto menor sea la brecha, mejor será nuestro modelo en sus predicciones y más confianza mostrará al predecir.

La **función de pérdida o costo** esencialmente modela la diferencia entre la predicción de nuestro modelo y la salida real. Idealmente, si estos dos valores están muy separados, el valor de pérdida o el valor de error deberían ser mayores. Del mismo modo, si estos dos valores están más cerca, el valor del error debería ser bajo. Una posible función de pérdida podría ser:

$$J(\Theta) = \hat{y} - y / y \in \{0, 1\} \tag{7}$$

Pero, en lugar de tomar esta función como nuestra función de pérdida, terminamos considerando la siguiente función:

$$J(\Theta) = \frac{\|\hat{y} - y\|^2}{2} \tag{8}$$

Esta función se conoce como error al cuadrado . Simplemente tomamos la diferencia entre la salida real y y la salida predicha $\hat{y}$ elevamos al cuadrado ese valor (de ahí el nombre) y lo dividimos entre 2.

Una de las principales razones para preferir el error al cuadrado en lugar del error absoluto es que el error al cuadrado es diferenciable en todas partes , mientras que el error absoluto no lo es (su derivada no está definida en 0).

Además, los beneficios de la cuadratura incluyen:

- La cuadratura siempre da un valor positivo , por lo que la suma no será cero.
- Hablamos de suma aquí porque sumaremos los valores de pérdida o error para cada imagen en nuestro conjunto de datos de entrenamiento y luego haremos un promedio para encontrar la pérdida para todo el lote de ejemplos de entrenamiento.
- La cuadratura enfatiza las diferencias más grandes, una característica que resulta ser buena y mala.

La función de error que usaremos aquí se conoce como el error cuadrático medio y la fórmula es la siguiente:

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{9}$$

Calculamos el error al cuadrado para cada *feature* en nuestro *dataset* y luego encontramos el promedio de estos valores y esto representa el error general del modelo en nuestro conjunto de entrenamiento.

Consideremos el ejemplo de una sola *feature* con solo 2 características de antes. Dos características significan que tenemos 2 valores de peso correspondientes y un valor de sesgo. En total, tenemos 3 parámetros para nuestro modelo.

$$\hat{y} = w_1 x_1 + w_2 x_2 + b \tag{10}$$

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (w_1 x_1^{(i)} + w_2 x_2^{(i)} + b - y^{(i)})^2 \tag{11}$$

Queremos encontrar valores para nuestros pesos y el sesgo que minimiza el valor de nuestra función de pérdida. Dado que esta es una ecuación de múltiples variables, eso significa que tendríamos que tratar con derivadas parciales de la función de pérdida correspondiente a cada una de nuestras variables w_1, w_2 y b.

$$\frac{\partial J}{\partial w_1}\frac{\partial J}{\partial w_2}\frac{\partial J}{\partial b} \tag{12}$$

Esto puede parecer lo suficientemente simple porque solo tenemos 3 variables diferentes. Sin embargo tenemos tantos pesos como features *i.e.* w_n pesos.

Hacer una optimización multivariante con tantas variables es computacionalmente ineficiente y no es manejable. Por lo tanto, recurrimos a alternativas y aproximaciones.

4.7 Descenso de gradiente

Es la capacidad de aprendizaje que otorga el algoritmo de descenso de gradiente lo que hace que el aprendizaje automático y los modelos de aprendizaje profundo funcionen.

El objetivo de este algoritmo es minimizar el valor de nuestra función de pérdida y queremos hacer esto de manera eficiente. Como se discutió anteriormente, la forma más rápida sería encontrar derivadas de segundo orden de la función de pérdida con respecto a los parámetros del modelo. Pero, eso es computacionalmente costoso.

La intuición básica detrás del descenso del gradiente puede ilustrarse mediante un escenario hipotético [9]: una persona está atrapada en las montañas y está tratando de bajar (es decir, tratando de encontrar los mínimos). Hay mucha niebla de tal manera que la visibilidad es extremadamente baja. Por lo tanto, el camino hacia abajo de la montaña no es visible, por lo que deben usar la información local para encontrar los mínimos.

Pueden usar el método de descenso en gradiente, que consiste en mirar la inclinación de la colina en su posición actual, luego proceder en la dirección con el descenso más empinado (es decir, cuesta abajo). Si trataban de encontrar la cima de la montaña (es decir, los máximos), entonces avanzarían en la dirección con el ascenso más empinado (es decir, cuesta arriba). Usando este método, eventualmente encontrarían su camino.

Sin embargo, suponga también que la pendiente de la colina no es inmediatamente obvia con una simple observación, sino que requiere un instrumento sofisticado para medir, que la persona tiene en ese momento.

Se necesita bastante tiempo para medir la inclinación de la colina con el instrumento, por lo tanto, deben minimizar el uso del instrumento si quieren bajar la montaña antes del atardecer. La dificultad es elegir la frecuencia con la que deben medir la inclinación de la colina para no desviarse.

En esta analogía:

- La persona representa nuestro **algoritmo de aprendizaje**, y el camino que baja por la montaña representa la **secuencia de actualizaciones de parámetros** que nuestro modelo eventualmente explorará.
- La inclinación de la colina representa la **pendiente de la superficie de error en ese punto**.
- El instrumento utilizado para medir la inclinación es la **diferenciación** (la pendiente de la superficie de error se puede calcular tomando la derivada de la función de error al cuadrado en ese punto). Esta es la aproximación que hacemos cuando aplicamos el descenso de gradiente. Realmente no sabemos el punto mínimo, pero sí sabemos la dirección que nos llevará a los mínimos (locales o globales) y damos un paso en esa dirección.
- La dirección en la que la persona elige viajar se alinea con el gradiente de la superficie de error en ese punto.
- La cantidad de tiempo que viajan antes de tomar otra medida es la **velocidad de aprendizaje del algoritmo**. Esto es esencialmente lo importante que nuestro modelo (o la persona que va cuesta abajo) decide dar cada vez.

Entonces el descenso del gradiente mide el gradiente local de la función de pérdida (costo) para un conjunto dado de parámetros (Θ) y da pasos en la dirección del gradiente descendente. Como ilustra la Figura 23, una vez que el gradiente es cero, hemos alcanzado un mínimo.

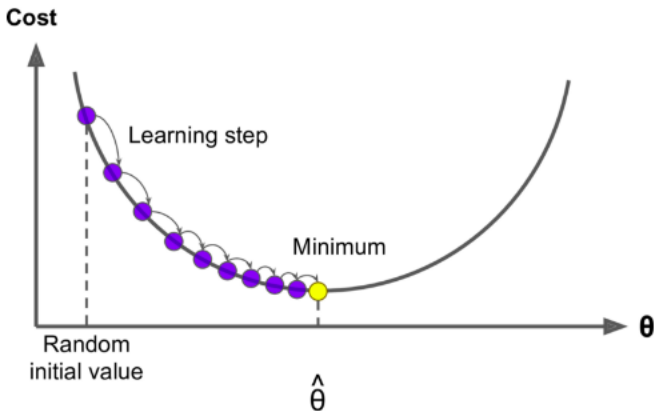


Figura 23: Representación gráfica del descenso de gradiente.

Como vemos en la Fig 24.Es importante ajustar apropiadamente el valor de la tasa de aprendizaje *(learning rate)*. Si es demasiado pequeña, entonces el algoritmo tomará muchas iteraciones (pasos) para encontrar el mínimo. Por otro lado, si es muy alta, es posible que supere el mínimo y termine más lejos que cuando comenzó.

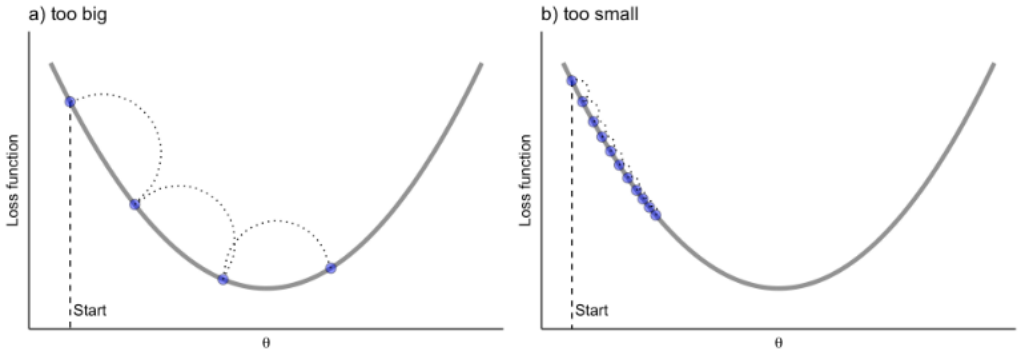


Figura 24: Ajuste de la tasa de aprendizaje.

Por tanto para actualizar la matriz de pesos y de sesgo serán utilizadas las siguientes ecuaciones.

$$W' = W - \alpha \frac{\partial J}{\partial W} \tag{13}$$

$$b' = b - \alpha \frac{\partial J}{\partial b} \tag{14}$$

El α representa la tasa de aprendizaje o *learning rate*.

4.8 Backpropagation

Ya sabemos cómo fluyen las activaciones en la dirección hacia adelante. Tomamos las *features* de entrada, las transformamos linealmente, aplicamos la activación sigmoidea en el valor resultante y finalmente tenemos nuestra activación que luego usamos para hacer una predicción [8].

Lo que veremos en esta sección es el flujo de gradientes a lo largo de la línea roja en la Figura 25 mediante un proceso conocido como retropropagación o *backpropagation*, que es esencialmente la regla de la cadena de cálculo aplicada a los gráficos computacionales.

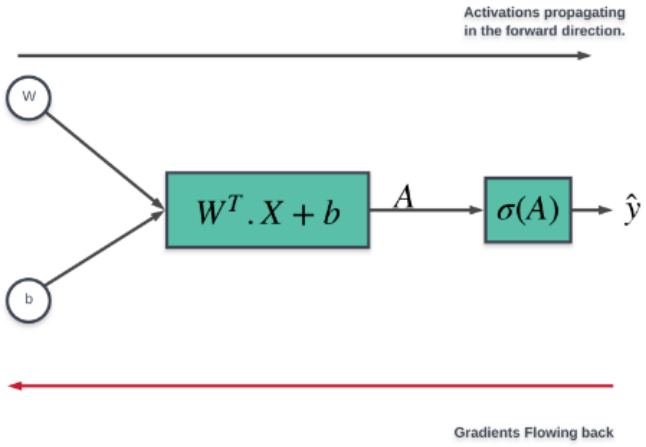


Figura 25: Las activaciones se propagan hacía adelante, pero los gradientes fluyen hacía atrás.

Digamos que queríamos encontrar la derivada parcial de la variable y con respecto a x de la Figura 26. No podemos descubrirlo directamente porque hay otras 3 variables involucradas en el gráfico computacional. Entonces, hacemos este proceso iterativamente yendo hacia atrás en el gráfico de cálculo.

Primero descubrimos la derivada parcial de la salida y con respecto a la variable C. Luego usamos la regla de la cadena de cálculo y determinamos la derivada parcial con respecto a la variable B y así sucesivamente hasta que obtengamos la derivada parcial que estamos buscando.

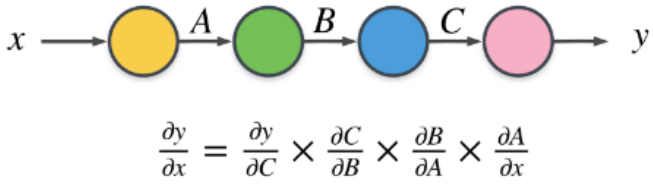


Figura 26: Representación de grafo simple.

Utilizando la función pérdida definida en la ecuación 9 y reescribiéndola en su forma vectorial.

$$J(\Theta) = \frac{1}{2}\|\hat{Y} - Y\|^2 \tag{15}$$

La derivada parcial de la función de pérdida con respecto a la activación de nuestro modelo es:

$$\frac{\partial J}{\partial \hat{Y}} = \frac{1}{2}\frac{\partial J}{\partial \hat{Y}}\|\hat{Y}-Y\|^2 = \frac{1}{2}2\hat{Y}-Y\frac{\partial}{\partial \hat{Y}}(\hat{Y}-Y) = (\hat{Y}-Y)\frac{\partial}{\partial \hat{Y}}\|\hat{Y}-Y\| = (\hat{Y}-Y) \tag{16}$$

Avancemos un paso hacia atrás y calculemos nuestra próxima derivada parcial. Esto nos llevará un paso más cerca de los gradientes reales que queremos calcular.

Este es el punto donde aplicamos la regla de la cadena que mencionamos antes. Entonces, para calcular la derivada parcial de la función de pérdida con respecto a la salida transformada lineal, es decir, la salida de nuestro modelo antes de aplicar la activación sigmoidea:

$$\frac{\partial J}{\partial D} = \frac{\partial J}{\partial \hat{Y}}\frac{\partial \hat{Y}}{\partial A} \tag{17}$$

La primera parte de esta ecuación es el valor que habíamos calculado en la ecuación 16. Lo esencial para calcular aquí es la derivada parcial de la predicción de nuestro modelo con respecto a la salida transformada linealmente.

Veamos la ecuación para la predicción de nuestro modelo, la función de activación sigmoidea.

$$\hat{Y} = \sigma(A) = \frac{1}{1+e^{-A}} \tag{18}$$

Derivada de la salida final de nuestro modelo, *i.e.* significa la derivada parcial de la función sigmoide con respecto a su entrada.

$$\frac{\partial}{\partial A}\sigma(A) = \frac{\partial}{\partial A}\frac{1}{1+e^{-A}} = \frac{\partial}{\partial A}(1+e^{-A})^{-1} \tag{19}$$

$$-1(1+e^{-A})^{-2}\frac{\partial}{\partial A}(1+e^{-A}) = -1(1+e^{-A})^{-2}(-e^{-A}) = \frac{e^{-A}}{(1+e^{-A})^2} \tag{20}$$

Continuando, podemos simplificar aún más esta ecuación.

$$\sigma(A) = \frac{1}{1+e^{-A}} \tag{21}$$

$$e^{-A} = \frac{1}{\sigma(A)} - 1 = \frac{1-\sigma(A)}{\sigma(A)} \tag{22}$$

Substituyendo este valor en la ecuación 17 obtenemos:

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \hat{Y}}\frac{e^{-A}}{(1+e^{-A})^2} \tag{23}$$

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \hat{Y}}\frac{1-\sigma(A)}{\sigma(A)}\sigma(A)\sigma(A) \tag{24}$$

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \hat{Y}}\sigma(A)(1-\sigma(A)) \tag{25}$$

Necesitamos la derivada parcial de la función de pérdida correspondiente a cada uno de los pesos. Pero como estamos recurriendo a la vectorización, podemos encontrarlo todo de una vez. Es por eso que hemos estado usando la notación mayúscula W en vez de $w_1, w_2, \ldots, w_n$.

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial A}\frac{\partial A}{\partial W} \tag{26}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial A}\frac{\partial A}{\partial b} \tag{27}$$

La derivación de los pesos queda partiendo de la ecuación 26:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial A}\frac{\partial}{\partial W}(W^T X + b) = \frac{\partial J}{\partial A}X \tag{28}$$

Y de la ecuación 27:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial A}\frac{\partial}{\partial b}((W^T X + b) = \frac{\partial J}{\partial A}1 = \frac{\partial J}{\partial A} \tag{29}$$

Se demostró desde un punto de vista matemático el concepto de *backpropagation* como se realiza la actualización de los pesos y sesgos utilizando el descenso por gradiente.

4.9 Descenso de gradiente estocástico (SGD)

Sin embargo el descenso de gradiente puede ser excepcionalmente lento en datasets muy grandes debido a que en cada iteración requiere calcular una predicción por cada punto de entrenamiento en nuestros datos de entrenamientos antes que actualizaciones nuestra matriz de pesos.

En cambio lo que se utiliza es una variante de éste, el descenso de gradiente estocástico o *Stochastic Gradient Descent (SGD)*. El SGD es una simple modificación del algoritmo de descenso de gradiente estándar que computa el gradiente y actualiza la matriz de pesos W en pequeños lotes o *batches* de datos de entrenamiento, en vez del *dataset* entero. Mientras esta modificación nos lleva a actualizaciones más " ruidosas", también nos permite tomar más pasos a lo largo del gradiente, llevando en ultima instancia a una convergencia más rápida y sin afectar negativamente a la pérdida y precisión del modelo.

En lugar de calcular nuestro gradiente en todo el conjunto de datos, en su lugar muestreamos nuestros datos, produciendo un lote. Evaluamos el gradiente en el lote y actualizamos nuestra matriz de peso W. Desde una perspectiva de implementación, también tratamos de aleatorizar nuestras muestras de entrenamiento antes de aplicar SGD ya que el algoritmo es sensible a los lotes.

En una implementación "purista" de SGD, el tamaño de su mini lote sería 1, lo que implica que muestrearíamos aleatoriamente un punto de datos del conjunto de entrenamiento, calcularíamos el gradiente y actualizamos nuestros parámetros.

Sin embargo, a menudo utilizamos mini lotes que son mayores a 1. Los tamaños de lote típicos incluyen 32, 64, 128 y 256.

A continuación enumeramos las justificaciones a esta decisión.

1. Ayudan a reducir la variación en la actualización de parámetros, lo que conduce a una convergencia más estable.

2. Las potencias de dos a menudo son deseables para los tamaños de lote, ya que permiten que las bibliotecas de optimización de álgebra lineal interna sean más eficientes.

En general, el tamaño del mini lote no es un hiperparámetro por el que debería preocuparse demasiado. Si está usando una GPU para entrenar su red neuronal, usted determina cuántos ejemplos de entrenamiento encajarán en su GPU y luego usa la potencia más cercana de dos, ya que el tamaño del lote se ajustará en la GPU. Para el entrenamiento de CPU, normalmente utiliza uno de los tamaños de lote enumerados anteriormente para asegurarse de cosechar los beneficios de las bibliotecas de optimización de álgebra lineal.

4.10 Sobreajuste y bajo-ajuste

El sobreajuste o *overfitting* y la falta de ajuste o *underfitting* [10] es muy importante para saber si el modelo predictivo está generalizando bien los datos o no. Un buen modelo debe poder generalizar bien los datos.

Figura 27: Distintas representaciones del ajuste en un mismo modelo.

En la Figura 27 **Izquierda** el modelo está sobreajustado, *i.e.* cuando funciona bien en el ejemplo de entrenamiento pero no funciona bien en datos no vistos. A menudo es el resultado de un modelo excesivamente complejo y ocurre porque el modelo está memorizando la relación entre el ejemplo de entrada (a menudo llamado X) y la variable objetivo (a menudo llamada y) o, por lo tanto, no puede generalizar bien los datos. El modelo de sobreajuste predice el objetivo en el conjunto de datos de entrenamiento con mucha precisión.

En cambio en la Figura 27 **Derecha** se dice que el modelo predictivo tiene bajo-ajuste, si funciona mal en los datos de entrenamiento. Esto sucede porque el modelo no puede capturar la relación entre el ejemplo de entrada y la variable objetivo. Podría deberse a que el modelo es demasiado simple, es decir, las características de entrada no son lo suficientemente expresivas como para describir bien la variable objetivo. El modelo con bajo-ajuste no predice los objetivos en los conjuntos de datos de entrenamiento con mucha precisión.

Un buen modelo debe ser como el de la Figura 27 **Medio** que posee una buena precisión en su conjunto de datos de entrenamiento pero a su vez también tiene una buena *performance* con datos que no haya visto.

4.11 Regularización

Para disminuir los efectos del sobreajuste se utiliza la **regularización** que después de la tasa de aprendizaje, es el parámetro más importante de su modelo que puede ajustar.

Existen varios tipos de técnicas de regularización, como la regularización L1, la regularización L2 (comúnmente llamada pérdida de peso) y Elastic Net, que se utilizan al actualizar la función de pérdida en sí, agregando un parámetro adicional para restringir la capacidad de el modelo.

La regularización nos ayuda a controlar la capacidad de nuestro modelo, asegurando que nuestros modelos sean mejores para hacer clasificaciones (correctas) en los puntos de datos en los que no fueron entrenados, lo que llamamos la capacidad de generalizar. Si no aplicamos la regularización, nuestros clasificadores pueden volverse demasiado complejos y ajustarse fácilmente a nuestros datos de entrenamiento, en cuyo caso perdemos la capacidad de generalizar a nuestros datos de prueba.

4.12 Los cuatro ingredientes de una red neuronal

Hay cuatro ingredientes principales [6] que necesita para armar su propia red neuronal y algoritmo de aprendizaje profundo: un conjunto de datos, un modelo/arquitectura, una función de pérdida y una optimización.

4.12.1 Conjunto de datos

También llamado *dataset*, es el primer ingrediente en el entrenamiento de una red neuronal: los datos en sí mismos junto con el problema que estamos tratando de resolver definen nuestros objetivos finales.

La combinación de su conjunto de datos y el problema que está tratando de resolver influye en su elección en la función de pérdida, la arquitectura de red y el método de optimización utilizado para entrenar el modelo. Por lo general, tenemos pocas opciones en nuestro conjunto de datos (a menos que esté trabajando en un proyecto de pasatiempo): se nos da un conjunto de datos con cierta expectativa sobre cuáles deberían ser los resultados de nuestro proyecto. Depende de nosotros entrenar un modelo de aprendizaje automático en el conjunto de datos para que funcione bien en la tarea dada.

4.12.2 Función de pérdida

Dado nuestro conjunto de datos y objetivo objetivo, necesitamos definir una función de pérdida que se alinee con el problema que estamos tratando de resolver.

4.12.3 Modelo/Arquitectura

La arquitectura de su red puede considerarse la primera "elección" real que tiene que hacer como ingrediente. Es probable que su conjunto de datos sea

elegido para usted (o al menos ha decidido que desea trabajar con un conjunto de datos determinado). Y si está realizando una clasificación, probablemente utilizará la entropía cruzada como su función de pérdida. Sin embargo, su arquitectura de red puede variar dramáticamente, especialmente cuando con qué método de optimización elige entrenar su red.

4.12.4 Método de optimización

El ingrediente final es definir un método de optimización. El SGD se usa con bastante frecuencia. SGD sigue siendo el caballo de batalla del aprendizaje profundo: la mayoría de las redes neuronales se entrenan a través de SGD, aunque existen otros métodos de optimización como Adam. Luego debe establecer una tasa de aprendizaje adecuada, la fuerza de regularización y el número total de épocas para las que se debe entrenar la red.

4.13 Redes Neuronales Convolucionales

Las redes neuronales convolucionales [6] (*CNNs* en Inglés) son principalmente útiles si en la entrada los datos presentados son imágenes, permite el desarrollo de modelos supervisados y no supervisados.

Podemos definir una *CNN* como una red neuronal que cambia una capa totalmente conectada (*fully-connected*) por una convolucional para al menos una de las capas de la red.

Cada capa en una *CNN* aplica un conjunto de filtros, usualmente cientos o miles de ellos y combinan los resultados, alimentando la entrada de la siguiente capa de la red. Durante el entrenamiento, una *CNN* automáticamente aprende los valores para esos filtros.

En el contexto de la clasificación de imágenes, una *CNN* puede aprender a:

- Detectar bordes a partir de datos de píxeles sin procesar en la primera capa.
- Usar esos bordes para detectar formas (*i.e. blobs*) en la segunda capa.
- Usar esas formas para detectar características de alto nivel tales como estructuras faciales, partes de un auto, etc. en las capas de más alto nivel.

La última capa en una *CNN* usa esas características de alto nivel para realizar predicciones considerando los contenidos de una imagen.

Las *CNNs* nos dan dos beneficios claves con respecto al reconocimiento de imágenes:

- **invariancia local**: nos permite clasificar una imagen que contiene un objeto particular sin importar donde aparece éste en la imagen.
- **composicionalidad**: cada filtro compone un parche local de características de nivel inferior en una representación de nivel superior, similar a cómo podemos componer un conjunto de funciones matemáticas que se basan en la salida de funciones anteriores. Esta composición permite que nuestra red aprenda características más ricas de forma más profunda.

Las convoluciones bi-dimensionales (2D) son usadas para tratar las imágenes, mientras que las convoluciones unidimensionales (1D) nos permiten analizar entradas secuenciales, obteniendo la información con dependencias temporales. Entonces al combinar estas dos técnicas, se puede apreciar cómo evolucionan en el tiempo las imágenes capturadas y así hacer predicciones a futuro.

4.13.1 Convolución 1D

Si f y g son funciones discretas [11], entonces $f * g$ es la convolución de f y g y está definida como:

$$(f * g)(x) = \sum_{u=-\infty}^{\infty} f(u)g(x-u)$$

Intuitivamente, la convolución de dos funciones representa la cantidad de superposición entre estas. La función g es la entrada y f el *kernel* o núcleo de la convolución.

Sin embargo en los algoritmos de *machine learning* lo que manejamos usualmente son vectores o arreglos de tal forma que nos resultará más provechoso analizar la convolución entre ellos.

Si la función f varía sobre un conjunto finito de valores $a = a_1, a_2, \ldots, a_n$ entonces puede ser representado como el vector $\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}$.

Si las funciones f y g son representadas como vectores $a = \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix}$ y $b = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}$, entonces $f * g$ es un vector $c = \begin{bmatrix} c_1 & c_2 & \cdots & c_{m+n-1} \end{bmatrix}$ definido de la siguiente forma:

$$c = \sum_{u} a_u b_{x-u+1}$$

donde u abarca todos los subíndices legales para a_u y b_{x-u+1}, específicamente $u = max(1, x-n+1) \ldots min(x, m)$.

Lo que puede parecer complicado en la teoría no lo es en la práctica, observemos la Fig. 28 [12]. El vector *input* también se denomina vector de características y el vector *output* mapa de características.

Lo que sucede es que si el *kernel* tiene un único valor sólo es necesario multiplicarlo por cada valor del vector *input* y guardarlo en el índice correspondiente del vector *output*.

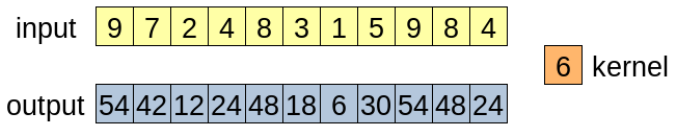


Figura 28: Vectores de convolución unidimensional con kernel simple.

En cambio si tenemos un *kernel* de dimensiones 2×1 como la Fig.29 para obtener el valor de salida i debemos usar los valores de entrada i y su vecino $i + 1$.

Para obtener el primer valor del vector de salida se realizó la operación `o[0] = i[0]k[0] + i[1]k[1] = 69`. De esta forma iteramos a lo largo de todo el vector de entrada hasta obtener todos los valores de salida. Podemos notar que el tamaño del vector de salida es menor ahora, a medida que aumentamos el tamaño del kernel disminuye el del vector de salida.

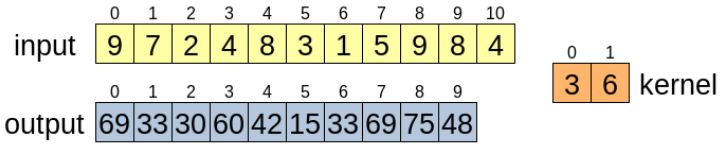


Figura 29: Vectores de convolución unidimensional con kernel doble.

Con objeto de dejar totalmente en claro el algoritmo observemos la Fig. 30. Para obtener el valor del índice 4 del vector de salida operamos `o[4] = i[3]k[0]+i[4]k[1]+i[5]k[2] = 23`.

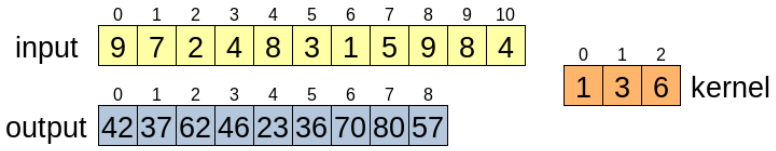


Figura 30: Vectores de convolución unidimensional con kernel triple.

Para finalizar se hará una pequeña mención al tamaño del vector de salida, que viene determinado por la siguiente formula [13]:

$$output_{size} = \frac{W - F + 2P}{S + 1}$$

donde $W = input_{size}$, $F = kernel_{size}$, P=*padding* y S=*stride*.

4.13.2 Convolución 2D

A su vez podemos extender esto a convoluciones para funciones de dos variables.

Si f y g son funciones discretas de dos variables, entonces $f * g$ es la convolución de f y g y se define:

$$(f * g)(x, y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u, v) g(x - u, y - v)$$

Podemos considerar funciones de dos variables como matrices con $A_{xy} = f(x, y)$ y obtener una definición matricial de la convolución.

Si las funciones f y g son representadas como las matrices A y B con dimensiones de $n \times m$ y $k \times i$ respectivamente, entonces $f * g$ es una matriz C de dimensiones $(n + k - 1) \times (m + i - 1)$ definida:

$$c_{xy} = \sum_{u} \sum_{v} a_{uv} b_{x-u+1, y-v+1}$$

donde u y v abarcan todos los subíndices posibles para a_{uv} y $b_{x-u+1,y-v+1}$.

Así como notamos que el algoritmo para la convolución 1D no era tan complejo como su definición formal, lo mismo sucede para la convolución 2D pero extrapolando el mecanismo a una dimensión más.

En la Fig. 31 [14] analizamos el procedimiento. Se debe centrar el kernel K sobre el primer valor a calcular, para luego realizar las respectivas multiplicaciones y luego guardarlas en la matriz de salida O, de esta manera iremos iterando de derecha a izquierda y de arriba hacia abajo toda la matriz I.

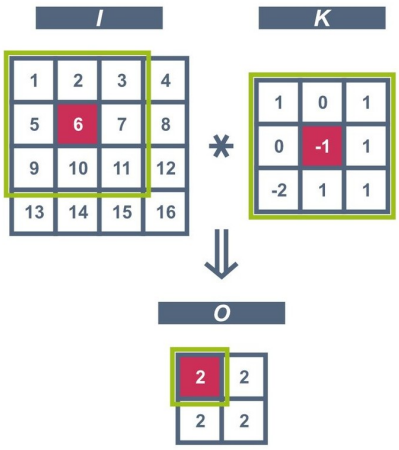


Figura 31: Vectores de convolución bidimensional con kernel 3×3.

Consideremos la Fig. 32 [15], tenemos una imagen RGB que ha sido separada por sus tres canales de color: rojo, verde y azul. Hay varios espacios de color en los que existen las imágenes: escala de grises, RGB, HSV, CMYK, etc.

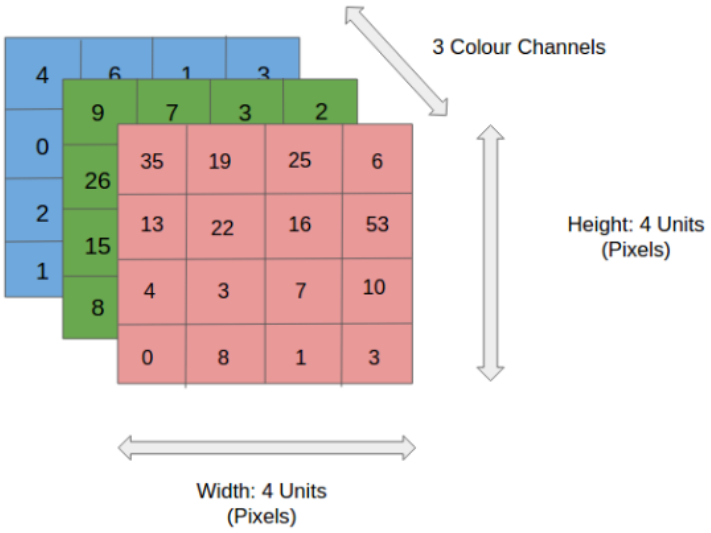


Figura 32: Imagen RGB $4 \times 4 \times 3$.

Si consideramos la totalidad de la imagen como un prisma donde la profundidad corresponde a cada canal de color, podemos ver en la Figura 33 el

movimiento que realiza el kernel (con forma de cubo) a través del volumen del prisma.

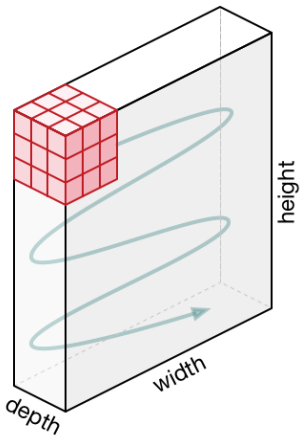


Figura 33: Movimiento del kernel.

padding stride

4.14 Redes Neuronales Recurrentes

Anteriormente hemos visto cómo las redes neuronales o convolucionales nos permiten clasificar un dato, por ejemplo una palabra, un sonido, o una imagen, pero tienen un inconveniente, y es que cuando tenemos una secuencia de datos, por ejemplo una secuencia de palabras, o una conversación, o una secuencia de imágenes, es decir un vídeo, este tipo de arquitecturas no pueden procesar ese tipo de datos.

Las Redes Neuronales Recurrentes *(RNN)* [16] resuelven este inconveniente, porque son capaces de procesar diferentes tipos de secuencias, como textos, conversaciones, vídeos, música, y además de eso no sólo clasifican los datos como lo hacen las redes neuronales o convolucionales, sino que también están en capacidad de generar nuevas secuencias.

Si a una red neuronal o convolucional se le presenta una imagen o una palabra, con el entrenamiento adecuado estas arquitecturas lograrán clasificar un sinnúmero de datos, logrando a la vez una alta precisión.

Pero, £qué sucede si en lugar de una única imagen o palabra se introduce a la red una secuencia de imágenes, es decir un vídeo, o una secuencia de palabras (una conversación)? En este caso en ninguna de estas redes será capaz de procesar los datos por dos motivos:

- Estas arquitecturas están diseñadas para que los datos de entrada y de salida siempre tengan el mismo tamaño; sin embargo, un vídeo o una conversación se caracterizan por ser un tipo de datos con un tamaño variable: una cantidad variable de "frames" en el caso del vídeo o una cantidad variable de palabras en el caso de la conversación.

- En un vídeo o en una conversación los datos están **correlacionados**, esto quiere decir que la siguiente palabra pronunciada o la siguiente imagen en la secuencia de vídeo dependerá de la palabra o imagen anterior. E incluso estas palabras e imágenes estarán relacionadas con aquellas que se presenten más adelante en la secuencia y una *NN* ó *CNN* no está en capacidad de analizar la relación entre varias palabras o imágenes de la secuencia.

Una secuencia es una serie de datos que siguen un orden específico y tienen únicamente significado cuando se analizan en conjunto y no de manera individual. Dichos datos, analizados de forma individual o en un orden diferente, carecen de significado. Es evidente que una secuencia no tiene un tamaño predefinido pues no podemos saber con antelación el número de datos.

Las *RNN* resuelven los inconvenientes expresados anteriormente, pues pueden procesar tanto a la entrada como a la salida secuencias sin importar su tamaño, y además teniendo en cuenta la correlación existente entre los diferentes elementos de esa secuencia.

Para ello este tipo de redes usan el concepto de recurrencia: para generar la salida, que también se conoce como activación, la red usa no sólo la entrada actual sino la activación generada en la iteración previa. En pocas palabras, las redes neuronales recurrentes usan un cierto tipo de memoria para generar la salida deseada.

4.14.1 Arquitecturas

Existen diversas arquitecturas disponibles para estas redes como observamos en la Figura 34 [16], donde cada rectángulo es un vector y cada fecha representa funciones. Los vectores de entrada están en rojo, los vectores de salida están en azul y los vectores verdes mantienen el estado de la *RNN*.

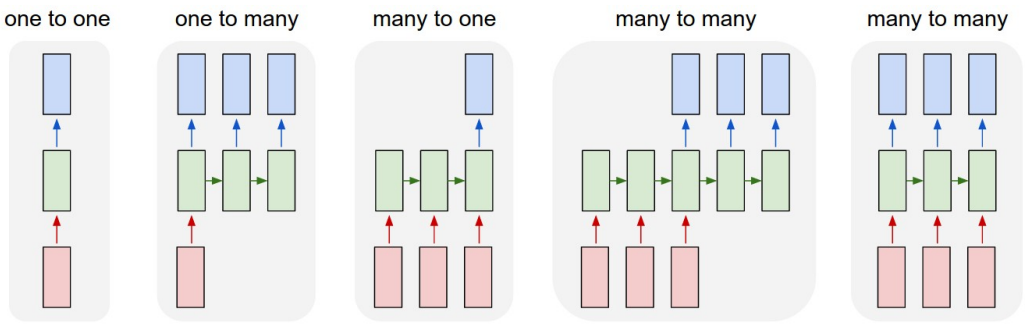


Figura 34: Tipos de arquitecturas para una *RNN*.

One-to-one Modo de procesamiento vanilla *i.e.* sin *RNN*, desde una entrada de tamaño fijo a una salida de tamaño fijo, por ejemplo la clasificación de imágenes.

One-to-many

La entrada es un único dato y la salida es una secuencia. Un ejemplo de esta arquitectura es el "*image captioning*" en donde la entrada es una y la salida es una secuencia de caracteres, un texto, que describe el contenido de la imagen.

Many-to-one

La entrada es una secuencia y la salida es por ejemplo una categoría. Un ejemplo de esto es la clasificación de sentimientos, en donde por ejemplo la entrada es un texto que contiene una crítica a una película y la salida es una categoría indicando si la película le gustó a la persona o no.

Many-to-many Tanto la entrada como a la salida se tienen secuencias. La primer figura se refiere a *RNN* utilizadas en traductores automáticos: en este caso la secuencia de salida no se genera al mismo tiempo que la secuencia de entrada pues para poder traducir por ejemplo una frase al español se requiere primero conocer la totalidad del texto en inglés. Y desde luego, en esta misma arquitectura podemos encontrar los conversores de voz a texto o texto a voz. La segunda figura se refiere a secuencias sincronizadas de entrada y salida, por ejemplo clasificación de vídeo donde deseamos etiquetar cada fotograma.

Como era de esperar, el régimen secuencial de operación es mucho más poderoso en comparación con las redes fijas que están condenadas desde el principio por un número fijo de pasos computacionales y, por lo tanto, también es más provechoso a la hora de construir sistemas más inteligentes.

Además, las *RNN* combinan el vector de entrada con su vector de estado

con una función fija (pero aprendida) para producir un nuevo vector de estado. En términos de programación, esto puede interpretarse como ejecutar un programa fijo con ciertas entradas y algunas variables internas. Visto de esta manera, los RNN esencialmente describen programas.

Si entrenar redes neuronales es optimización sobre funciones, entrenar redes recurrentes es optimización sobre programas.

4.14.2 Funcionamiento

Consideremos la Figura 35 [17], aquí podemos observar como se define de manera gráfica una unidad funcional de una *RNN* denominada A, que toma una entrada x_t y genera un valor h_t.

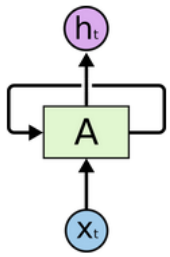


Figura 35: Unidad funcional de *RNN*.

El *loop* de A permite que la información pase de un paso de la red al siguiente.

Una red neuronal recurrente se puede considerar como múltiples copias de la misma red, cada una de las cuales pasa un mensaje a un sucesor. si desenrollamos el ciclo podemos representar la *RNN* a través del eje del tiempo, como se muestra en la Figura 36.

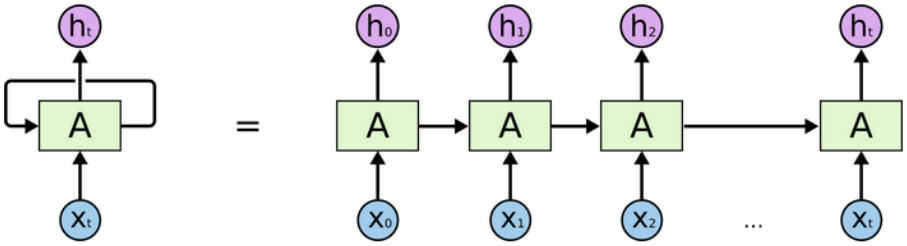


Figura 36: Una *RNN* desenrrollada.

Esta naturaleza en cadena revela que las redes neuronales recurrentes están íntimamente relacionadas con secuencias y listas.

En su esencia una *RNN* se parece demasiado a una *FFNN*, excepto que también tiene conexiones hacía atrás. La *RNN* más simple posible es la que mostramos en la figura

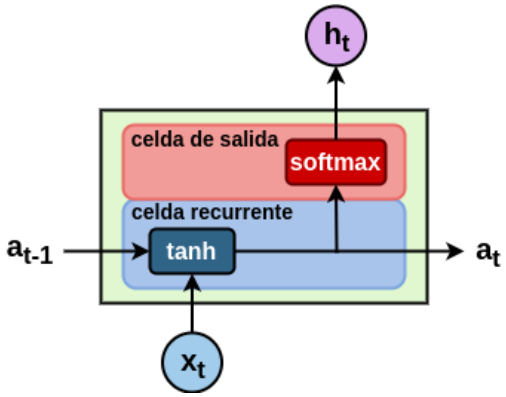


Figura 37: Unidad funcional *RNN* detallada.

En la Figura 37 se observa que en cada instante de tiempo la red tiene realmente dos entradas y dos salidas. Las entradas son el dato actual, x_t y la activación anterior, a_{t-1}, mientras que las salidas son la predicción actual, y_t, y la activación actual, a_t. Esta activación también recibe el nombre de *hidden state* o estado oculto.

Se define:

$$a_t = tanh(W_{aa}a_{t-1} + W_{ax}x_t + b_a)$$

$$h_t = softmax(W_{ya}a_t + b_y)$$

Donde:

W_{ax} : matriz de pesos multiplicando la entrada.

W_{aa} : matriz de pesos multiplicando el estado oculto.

W_{ya} : matriz de pesos que relaciona el estado oculto a la salida.

b_a : bias.

b_y : bias que relaciona el estado oculto a la salida.

Es posible entrenar una RNN con una gran cantidad de texto y le pediremos que modele la distribución de probabilidad del siguiente carácter en la secuencia dada una secuencia de caracteres anteriores. Esto nos permitirá generar texto nuevo, de a un carácter a la vez.

Como ejemplo práctico, suponga que solo tenemos un vocabulario de cuatro letras posibles `helo` y queremos entrenar a un RNN en la secuencia de entrenamiento `hello`. Esta secuencia de entrenamiento es de hecho una fuente de 4 ejemplos de entrenamiento separados:

1. La probabilidad de `e` probablemente debería estar dado el contexto de `h`.
2. `l` debería estar probablemente en el contexto de `he`.
3. `l` probablemente también debería ser dado el contexto de `hel`.
4. Y finalmente `o` debería ser probablemente dado el contexto de `hell`.

Concretamente, codificaremos cada carácter en un vector usando la codificación *1-of-k* (es decir, todo cero excepto uno en el índice del carácter en el vocabulario) y los introduciremos en el RNN uno a la vez con el función `step`. Luego observaremos una secuencia de vectores de salida de 4 dimensiones (una dimensión por carácter), que interpretamos como la confianza que el RNN asigna actualmente a cada carácter que sigue en la secuencia.

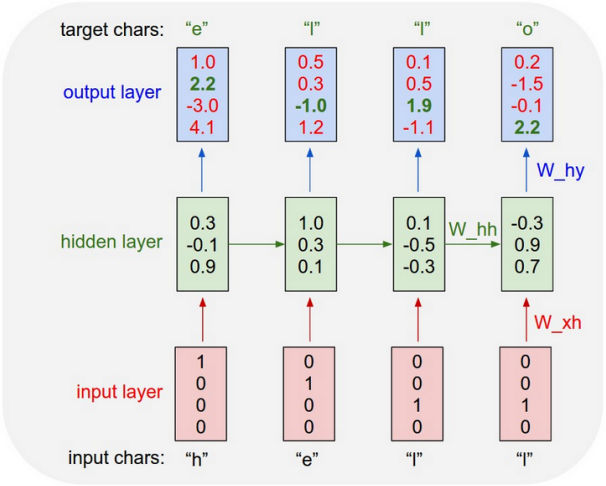


Figura 38: Unidad funcional *RNN* detallada.

En la Fig. 38 observamos un ejemplo de *RNN* con capas de entrada y salida de 4 dimensiones y una capa oculta de 3 unidades (neuronas). Las activaciones en el pase hacia adelante cuando el *RNN* recibe los caracteres `"hell"` como entrada. La capa de salida contiene los pesos que el *RNN* asigna

al siguiente carácter (el vocabulario es `"h, e, l, o"`); Queremos que los números verdes sean altos y los números rojos bajos.

Por ejemplo, vemos que en el primer paso de tiempo cuando el *RNN* vio el carácter `"h"` asignó un peso de 1.0 a la siguiente letra que era `"h"`, 2.2 a la letra `"e"`, −3.0 a `"l"` y 4.1 a "o". Dado que en nuestros datos de entrenamiento (la cadena `"hello"`) el siguiente carácter correcto es `"e"`, nos gustaría aumentar su peso (verde) y disminuir los pesos de todas las demás letras (rojo).

De manera similar, tenemos un carácter objetivo deseado en cada uno de los 4 pasos de tiempo a los que nos gustaría que la red le asignara una mayor confianza. Dado que el *RNN* consta completamente de operaciones diferenciables, podemos ejecutar el algoritmo de *back-propagaton* para averiguar en qué dirección debemos ajustar cada uno de sus pesos para aumentar los pesos de los objetivos correctos.

Luego podemos realizar una actualización de parámetros, que empuja cada peso una pequeña cantidad en esta dirección de gradiente. Si tuviéramos que alimentar las mismas entradas al RNN después de la actualización del parámetro, encontraríamos que las puntuaciones de los caracteres correctos (por ejemplo, "e" en el primer paso de tiempo) serían ligeramente más altas (por ejemplo, 2.3 en lugar de 2.2), y los pesos de los caracteres incorrectos serían ligeramente inferiores.

Luego, repetimos este proceso una y otra vez hasta que la red converge y sus predicciones son finalmente consistentes con los datos de entrenamiento en el sentido de que los caracteres correctos siempre se predicen a continuación.

4.14.3 Entrenamiento

Para entrenar una *RNN*, el truco simplemente es desenrrollarla a través del tiempo y simplemente usar *backpropagation* (estrategia que recibe el nombre de *backpropagation* a través del tiempo ($BPTT$)) como observamos en la Fig. 39 [20].

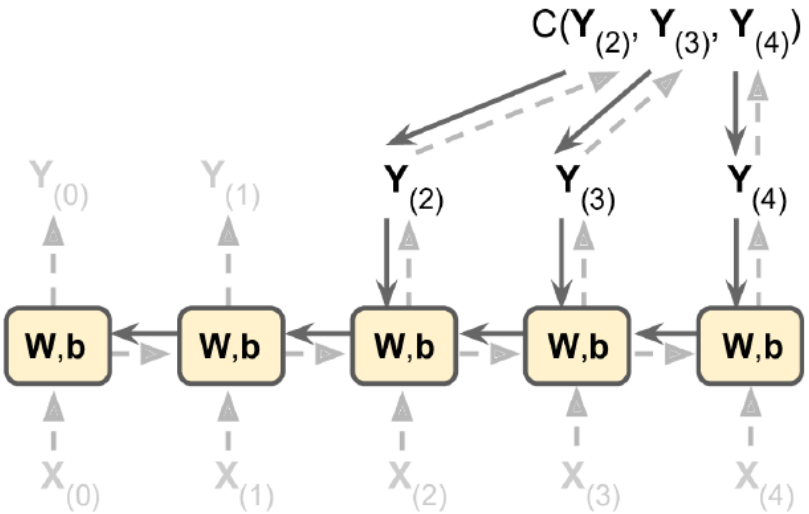


Figura 39: *Backpropagation* a través del tiempo.

Primero realizamos una pasada hacía adelante a través de la red desenrollada (representada en la figura por las flechas punteadas).

Luego la secuencia de salida es evaluada utilizando una función de costo $C(Y_{(0)}, Y_{(1)}, \ldots, Y_{(T)})$ (donde T es el paso máximo de tiempo). Notemos que la función de coste puede ignorar algunas salidas en función de lo que necesitemos como se muestra en la Fig. 39. Los gradientes de esa función de costo luego son propagados hacía atrás a través de la red desenrrollada (representada a través de las líneas sólidas).

Finalmente los parámetros del modelo son actualizadas usando los gradientes calculados por *BPTT*. Notar que los gradientes fluyen hacía atrás a través de todas las salidas utilizadas por la función de costo, no solamente a través de la salida final (notar que en el ejemplo no fluye a través de $Y_{(0)}$ e $Y_{(1)}$).

4.14.4 Desvanecimiento del gradiente

Otra forma de alimentar una *RNN* podría ser a través de las palabras individuales de una oración, dado que esto se realiza de forma secuencial, debemos proveerle de una palabra a la vez.

En el ejemplo de la Fig. 40 intentaremos predecir la intención del usuario tomando como entrada la oración `"What time is it?"`. [18].

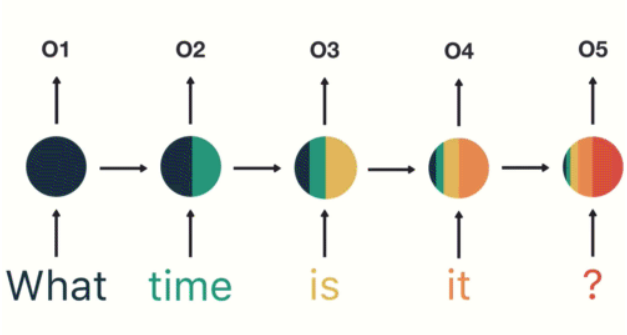


Figura 40: *RNN* siendo alimentada con las palabras de la oración.

1. Inicializa sus capas de red y el estado oculto inicial. La forma y dimensión del estado oculto dependerá de la forma y dimensión de su *RNN*.
2. Luego recorre sus entradas, pasa la palabra y el estado oculto al *RNN*.
3. El *RNN* devuelve la salida y un estado oculto modificado.
4. Continúas repitiendo hasta que te quedas sin palabras.
5. Por último, pasa la salida a la capa de *feedforward* y devuelve una predicción (Fig. 41).

Figura 41: Predicción de la *RNN*.

Pero prestemos atención a la Fig. 42. Es posible que haya notado la extraña distribución de colores en los estados ocultos. Eso es para ilustrar un problema con los *RNN* conocido como memoria a corto plazo.

Figura 42: Estado oculto final de la *RNN*.

La memoria a corto plazo es causada por el infame problema del desvanecimiento del gradiente, que también prevalece en otras arquitecturas de redes neuronales. A medida que el RNN procesa más pasos, tiene problemas para retener información de los pasos anteriores.

Como puede ver, la información de la palabra `"What"` y `"time"` es casi inexistente en el último paso. La memoria a corto plazo y el desvanecimiento del gradiente se deben a la naturaleza del algoritmo de *back-propagation*.

Al hacer *back-propagation*, cada nodo de una capa calcula su gradiente con respecto a los efectos de los gradientes, en la capa anterior. Entonces, si los ajustes a las capas anteriores son pequeños, los ajustes a la capa actual serán aún más pequeños.

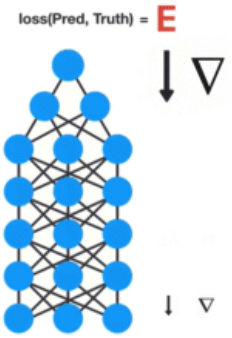


Figura 43: Desvanecimiento del gradiente desde las capas superiores a las inferiores.

Es posible pensar en cada paso de tiempo en una *RNN* como una capa y para entrenarla se usa *back-propagation* a través del tiempo. Los valores del gradiente se reducirán exponencialmente a medida que se propaga a través de cada paso de tiempo.

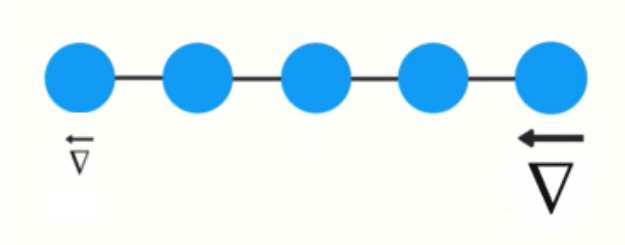


Figura 44: El gradiente se achica a medida que se propaga hacia atrás en el tiempo.

Nuevamente, el gradiente se utiliza para realizar ajustes en los pesos de las redes neuronales, lo que le permite aprender. Pequeños gradientes significan pequeños ajustes. Eso hace que las capas tempranas no aprendan.

Debido a los gradientes que desaparecen, la *RNN* no aprende las dependencias de largo alcance en los pasos de tiempo. Eso significa que existe la posibilidad de que las palabras `"What"` y `"time"` no se consideren al intentar predecir la intención del usuario. Entonces, la red tiene que hacer la mejor suposición con `"is it?"`. Eso es bastante ambiguo y sería difícil incluso para un humano. Por lo tanto, no poder aprender en pasos de tiempo anteriores hace que la red tenga una memoria a corto plazo.

Para mitigar la memoria a corto plazo, se crearon dos redes neuronales recurrentes especializadas. Las redes denominadas *Long Short-Term Memory* o *LSTM* para abreviar. El otro son Gated Recurrent Units o GRU.

4.14.5 LSTM

Los *LSTM* y *GRU* funcionan esencialmente como los *RNN*, pero son capaces de aprender las dependencias a largo plazo mediante mecanismos llamados "puertas". Estas puertas son diferentes operaciones de tensor que pueden aprender qué información agregar o quitar al estado oculto. Debido a esta capacidad, la memoria a corto plazo es un problema menor para ellos. [17]

Si consideramos la celda *LSTM* como una caja negra, aparenta ser idéntica a una *RNN* excepto que su estado se divide en dos vectores: $h_{(t)}$ y $c_{(t)}$ ("c" se mantiene por celda). Es posible pensar a $h_{(t)}$ como un estado de corto plazo y a $c_{(t)}$ como un estado de largo plazo.

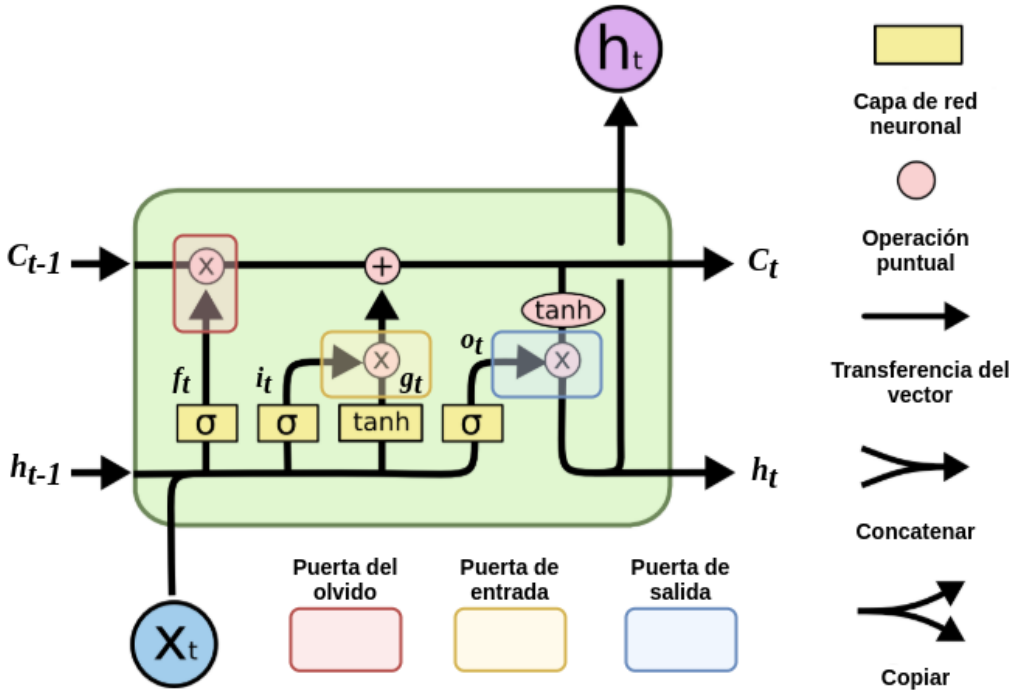


Figura 45: Celda *LSTM*.

El diagrama completo del *LSTM* lo podemos observar en la Fig.45, pero vamos a ir paso a paso analizando cada una de las partes que lo componen.

La clave de los *LSTM* es el estado de la celda, la línea horizontal que atraviesa la parte superior de la Fig. 46. El estado de la celda es como una cinta transportadora. Corre directamente a lo largo de toda la cadena, con solo algunas interacciones lineales menores. Es muy fácil que la información fluya sin cambios.

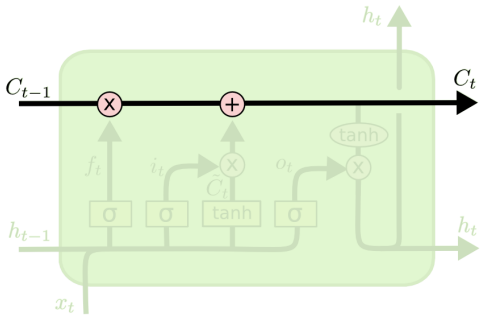


Figura 46: Celda de estado.

El *LSTM* tiene la capacidad de eliminar o agregar información al estado de la celda, regulada cuidadosamente por estructuras llamadas puertas. Las puertas son una forma de dejar pasar información opcionalmente. Están compuestos por una capa de red neuronal sigmoidea y una operación de multiplicación.

La capa sigmoidea genera números entre 0 y 1, que describen cuánto de cada componente debe dejarse pasar. Un valor de 0 significa "no dejar pasar nada", mientras que un valor de 1 significa "dejar pasar todo". Un *LSTM* tiene tres de estas puertas para proteger y controlar el estado de la celda.

El primer paso es decidir qué información vamos a eliminar del estado de la celda. Esta decisión la toma una capa sigmoidea llamada **puerta del olvido** (Fig. 47). Examina h_{t-1} y x_t, y genera un número entre 0 y 1 para cada número en el estado de celda C_{t-1}. Un 1 representa "mantener esto completamente", mientras que un 0 representa "deshacerse de esto por completo".

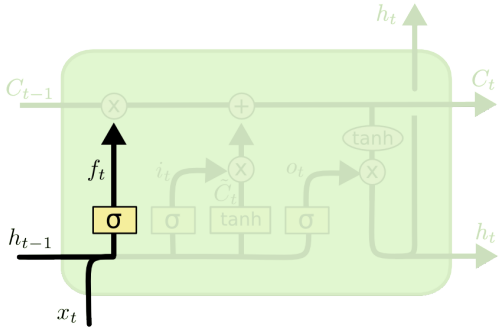


Figura 47: Puerta del olvido.

Por tanto la puerta del olvido queda representada por la siguiente ecuación:

$$f_{(t)} = \sigma(W_{xf}x_{(t)} + W_{hf}h_{(t-1)} + b_f)$$

El siguiente paso es decidir qué nueva información almacenaremos en el estado de la celda. Esto tiene dos partes.

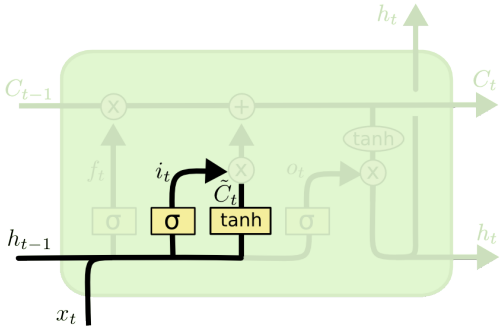


Figura 48: Puerta de entrada.

Una capa sigmoidea llamada **puerta de entrada** (Fig 48) decide qué valores actualizaremos.

$$i_{(t)} = \sigma(W_{xi}x_{(t)} + W_{hi}h_{(t-1)} + b_i)$$

A continuación, una capa *tanh* crea un vector de nuevos valores candidatos, que podrían agregarse al estado.

$$g_{(t)} = tanh(W_{xg}x_{(t)} + W_{hg}h_{(t-1)} + b_g)$$

En el siguiente paso, combinaremos estos dos para crear una actualización del estado.

Ahora es el momento de actualizar el estado de la celda anterior, $C_{(t-1)}$, al nuevo estado de la celda $C_{(t)}$. Los pasos anteriores ya decidieron qué hacer, solo tenemos que hacerlo realmente.

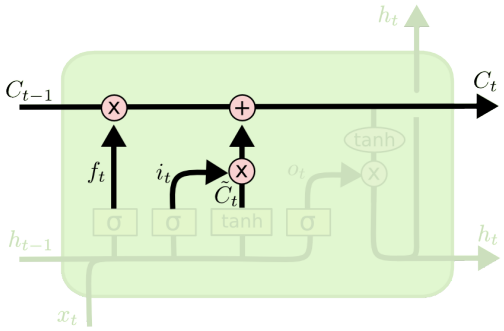


Figura 49: Actualización de la celda.

Multiplicamos el estado anterior por $f_{(t)}$, olvidando las cosas que decidimos olvidar antes. Luego le sumamos $i_{(t)} \otimes g_{(t)}$. Estos son los nuevos valores candidatos, escalados según cuánto decidimos actualizar cada valor de estado.

$$C_{(t)} = f_{(t)} \otimes C_{(t-1)} + i_{(t)} \otimes g_{(t)}$$

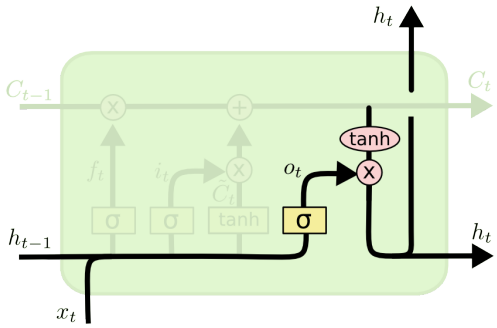


Figura 50: Puerta de salida.

Finalmente, tenemos que decidir qué vamos a producir. Esta salida se basará en el estado de nuestra celda, pero será una versión filtrada. Primero, ejecutamos una capa sigmoidea que denominaremos **puerta de salida** que decide qué partes del estado de la celda vamos a generar. Luego, colocamos el estado de la celda a través de *tanh* (para presionar los valores entre -1 y 1) y lo multiplicamos por la salida de la puerta, de modo que solo produzcamos las partes que decidimos.

$$o_{(t)} = \sigma(W_{xo}x_{(t)} + W_{ho}h_{(t-1)} + b_o)$$

$$h_{(t)} = o_{(t)} \otimes tanh(C_{(t)})$$

Pasando en limpio observando nuevamente la Fig. 45, tendremos nuestro vector de entradas $x_{(t)}$ y el estado anterior de corto plazo $h_{(t-1)}$ que alimenta 4 capas `FC`. Cada una de ellas sirve a un propósito diferente [20]:

- La capa principal es la que genera $g_{(t)}$. Tiene la función habitual de analizar las entradas actuales $x_{(t)}$ y el estado anterior (a corto plazo) $h_{(t-1)}$. En una celda básica *RNN*, no hay nada más que esta capa, y su salida va directamente hacia $y_{(t)}$ y $h_{(t)}$. Por el contrario, en una celda *LSTM*, la salida de esta capa no sale directamente, sino que se almacena parcialmente en el estado a largo plazo.
- Las otras tres capas son controladores de puerta. Dado que usan la función de activación sigmoidea, sus salidas van entre 0 y 1. Sus salidas se alimentan a operaciones de multiplicación elemento a elemento (también conocido como producto de Hadamard [19]), por lo que si generan ceros, cierran la puerta, y si generan 1 la abre. Específicamente:
  - La puerta de olvido (controlada por $f_{(t)}$) controla qué partes del estado a largo plazo $C_{(t-1)}$ deben borrarse.
  - La puerta de entrada (controlada por $i_{(t)}$) controla qué partes de $g_{(t)}$ deben agregarse al estado a largo plazo.
  - La puerta de salida (controlada por $o_{(t)}$) controla qué partes del estado a largo plazo deben leerse y generarse en este paso de tiempo (tanto en $h_{(t)}$ como en $y_{(t)}$).

En resumen, una celda LSTM puede aprender a reconocer una entrada importante (ese es el papel de la puerta de entrada), almacenarla en el estado a largo plazo, aprender a preservarla durante el tiempo que sea necesario (ese es el papel de la puerta del olvido) y aprender a extraerla siempre que sea necesario.

4.14.6 GRU

Otra variante popular es la celda *GRU* (Unidad Recurrente Cerrada, *Gated Recurrent Unit).* [17]

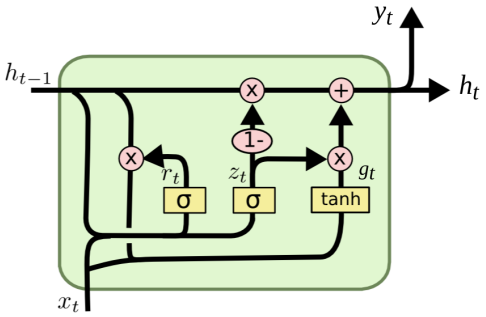


Figura 51: Celda *GRU*.

Hace algunas simplificaciones [20]:

- Ambos vectores de estado son combinados en un único vector $h_{(t)}$.
- Un controlador de puerta única $z_{(t)}$ controla tanto la puerta de olvido como la puerta de entrada. Si el controlador de puerta genera un 1, la puerta de olvido está abierta ($= 1$) y la puerta de entrada está cerrada ($1 - 1 = 0$). Si genera un 0, sucede lo contrario. En otras palabras, siempre que se deba almacenar una memoria, primero se borra la ubicación donde se almacenará.
- No hay puerta de salida; el vector de estado completo se genera en cada paso de tiempo. Sin embargo, hay un nuevo controlador de puerta $r_{(t)}$ que controla qué parte del estado anterior se mostrará en la capa principal $g_{(t)}$.

Las ecuaciones quedan de la siguiente forma:

$$
\begin{aligned}
z_{(t)} &= \sigma(W_{xz}x_{(t)} + W_{hx}h_{(t-1)} + b_z) \\
r_{(t)} &= \sigma(W_{xr}x_{(t)} + W_{hr}h_{(t-1)} + b_r) \\
g_{(t)} &= \sigma(W_{xg}x_{(t)} + W_{hg}h_{(t-1)} + b_g) \\
h_{(t)} &= z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)}
\end{aligned}
$$

5 Desarrollo

5.1 Limpieza de datos

Se tomarán datos principalmente de cuatro fuentes, las cuales debido a que difieren en su origen es necesario su normalización, tarea que se realiza a

través de la librería `Pandas` (enfocada al manejo de tablas y datos). El entorno de trabajo será `Jupyter-Notebook` que soporta `Python` como lenguaje de programación.

Referencias

[1] David I Poole, Randy G Goebel, and Alan K Mackworth. *Computational intelligence*. Oxford University Press New York, 1998.

[2] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[3] A.T. Norman and S. Bolivar. *Aprendizaje automático en acción. Un libro para el lego, guía paso a paso para los novatos*. Tektime, 2019.

[4] Pedro Antonio Gutiérrez. Github - pagutierrez/tutorial-sklearn: Tutorial sobre scikit-learn completo. `https://github.com/pagutierrez/tutorial-sklearn`, 2020. (Accessed on 12/28/2020).

[5] B. G. Trejo. *Selección de herramientas de Machine Learning aplicado a problemas de ingeniería*. Facultad de Ciencias Exactas, Físicas y Naturales, Universidad Nacional de Córdoba, 2019.

[6] A. Rosebrock. *Deep Learning for Computer Vision with Python: Starter Bundle*. PyImageSearch, 2017.

[7] D. J. Matich. *Redes neuronales: Conceptos básicos y aplicaciones*. Universidad Tecnológica Nacional, FRR, Departamento de Ing. Química, 2001.

[8] freeCodeCamp.org. Demystifying gradient descent and backpropagation via logistic regression based image..., Jul 2018. URL `https://www.freecodecamp.org/news/demystifying-gradient-descent-and-backpropagation-via-logistic-regression-ba`

[9] Wikipedia. Gradient descent - wikipedia. `https://en.wikipedia.org/wiki/Gradient_descent#An_analogy_for_understanding_gradient_descent`, 2020. (Accessed on 12/31/2020).

[10] Quora. What are the key trade-offs between overfitting and underfitting?, 2020. URL `https://www.quora.com/What-are-the-key-trade-offs-between-overfitting-and-underfitting`.

[11] Frank Keller. *Convolutions and Kernels*. School of Informatics, University of Edinburgh, Feb 2010.

[12] Cogneethi. C 4.1 | 1D Convolution | CNN | Object Detection | Machine Learning | EvODN, Aug 2019. URL `https://www.youtube.com/watch?v=yd_j_zdLDWs`. [Online; accessed 4. Jan. 2021].

[13] StackOverflow. How did they calculate the output volume for this convnet example in Caffe?, Jan 2021. URL https://stackoverflow.com/questions/32979683/how-did-they-calculate-the-output-volume-for-this-convnet-example-in-caffe. [Online; accessed 4. Jan. 2021].

[14] Louis N Andrianaivo, Roberto D'Autilia, and Valerio Palma. Architecture recognition by means of convolutional neural networks. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, 2019.

[15] Sumit Saha. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. *Medium*, Oct 2020. ISSN 3211-6453. URL https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b116

[16] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, Jun 2020. URL https://karpathy.github.io/2015/05/21/rnn-effectiveness. [Online; accessed 3. Jan. 2021].

[17] Christopher Olah. Understanding LSTM Networks, Aug 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs. [Online; accessed 4. Jan. 2021].

[20] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019. ISBN 9781492032595.

[18] Michael Phi. Illustrated Guide to Recurrent Neural Networks - Towards Data Science. *Medium*, Sep 2019. ISSN 7958-0499. URL https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9.

[19] Producto de Hadamard (matrices) - Hadamard product (matrices) - qaz.wiki, Jan 2021. URL https://es.qaz.wiki/wiki/Hadamard_product_(matrices). [Online; accessed 4. Jan. 2021].