

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

29 de marzo de 2021

Índice general

1 Motivación	4
2 Objetivo	5
3 Clasificación de modelos de inteligencia artificial	6
3.1 Inteligencia Artificial	6
3.1.1 Aprendizaje automático	6
3.2 Preprocesamiento	7
3.3 Aprendizaje supervisado	8
3.3.1 Clasificación	8
3.3.2 Regresión	10
3.4 Aprendizaje no supervisado	12
3.4.1 Detección de anomalías	12
3.4.2 Reducción de dimensionalidad	13
3.4.3 <i>Clustering</i>	14
3.5 Selección de algoritmo en base al <i>dataset</i>	16
3.5.1 Algoritmos supervisados	18
3.5.2 Algoritmos no supervisados	20
4 Redes neuronales	21
4.1 Relación con la biología	22
4.2 Modelos artificiales	23
4.3 Funciones de activación	24
4.4 Arquitecturas de redes <i>feedforward</i>	26
4.5 Redes multicapa	27
4.6 Función pérdida	28
4.7 Descenso de gradiente	30
4.8 Backpropagation	33
4.9 Descenso de gradiente estocástico (SGD)	36
4.10 Sobreajuste y bajo-ajuste	37
4.11 Regularización	37
4.12 Los cuatro ingredientes de una red neuronal	38
4.12.1 Conjunto de datos	38
4.12.2 Función de pérdida	38
4.12.3 Modelo/Arquitectura	38
4.12.4 Método de optimización	39

5 Redes neuronales convolucionales	39
5.1 Convolución 1D	40
5.2 Convolución 2D	42
5.2.1 <i>Padding</i>	44
5.2.2 <i>Stride</i>	45
5.3 Tipos de capas	45
5.3.1 Convolución	46
5.3.2 Activación	48
5.3.3 <i>Fully-connected</i>	49
5.3.4 <i>Pooling</i>	49
5.3.5 <i>Batch Normalization</i>	50
5.3.6 <i>Dropout</i>	51
5.4 <i>WaveNet</i> y capas convolucionales causales dilatadas	52
6 Redes neuronales recurrentes	55
6.1 Arquitecturas	56
6.2 Funcionamiento	58
6.3 Entrenamiento	61
6.4 Desvanecimiento del gradiente	62
6.5 Tipos de <i>RNNs</i>	65
6.5.1 LSTM	65
6.5.2 GRU	70
6.6 Secuencias de entradas y salida	71
7 Series temporales	73
7.1 Clasificación del modelo	73
7.1.1 Tipo de variables de entrada	73
7.1.2 Objetivo	74
7.1.3 Estructura	74
7.1.4 Cantidad de variables utilizadas como características .	75
7.1.5 Horizonte de pronóstico	75
7.1.6 Estático <i>vs</i> Dinámico	75
7.1.7 Uniformidad en el tiempo	76
7.2 Modelos disponibles	76
7.2.1 <i>Prophet</i>	76
7.2.2 <i>GluonTS</i>	77
7.2.3 <i>sktime</i>	77
7.2.4 Conclusión	77

8 Desarrollo	78
8.1 Breve introducción	78
8.1.1 Organización de la usina	78
8.1.2 Nomenclatura de defectos	78
8.1.3 Tipo	79
8.1.4 Puntos de captaje	79
8.2 Procesamiento de datos	80
8.2.1 Recolección de datos	80
8.2.2 Limpieza de datos	81
8.2.3 Preprocesamiento de datos	81
8.2.4 Análisis y visualización de datos	81
8.2.5 Imputación de datos	89
8.2.6 Formateo de datos	97
9 Modelos	105
9.1 Métricas	105
9.1.1 MAPE	105
9.1.2 MASE	106
9.2 <i>Framework</i>	107
9.2.1 Callbacks	107
9.3 Arquitecturas	108
9.3.1 Hiperparámetros de capas	108
9.3.2 Tipos	110
9.4 Entrenamiento	112

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 Clasificación de modelos de inteligencia artificial

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.

90000.

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]

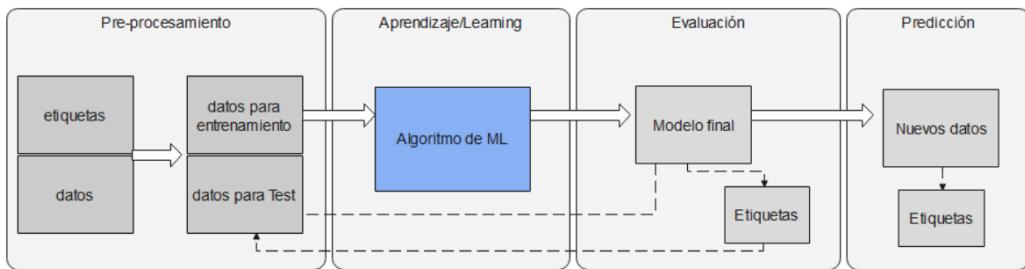


Figura 1: Diagrama de flujo de una aplicación de *Machine Learning*.

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 multiclasa**.

Un ejemplo de clasificación multiclasa 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.

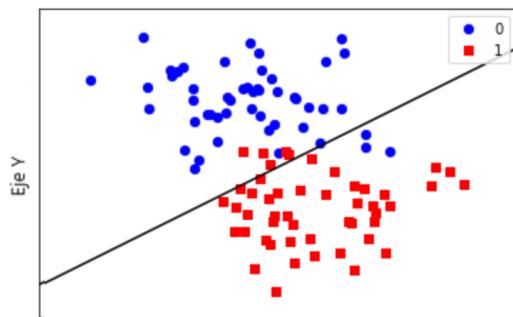


Figura 2: Ejemplo de clasificación binaria.

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	<ul style="list-style-type: none"> • Simple de entender e implementar . • No necesita una gran cantidad de datos de entrenamiento. • Rápido. 	<ul style="list-style-type: none"> • Supone que cada característica es independiente. • Sufre al tener características irrelevantes.
Regresion logistica	<ul style="list-style-type: none"> • Simple de entender e implementar. • Rara vez existe sobreajuste. • Rápido de entrenar. 	<ul style="list-style-type: none"> • Es muy difícil lograr que se ajuste a datos no lineales. • Los valores atípicos alteran la precisión del modelo.
KNN	<ul style="list-style-type: none"> • Eficaz en datasets de varias clases. • Entrenamiento rápido. 	<ul style="list-style-type: none"> • La dimensionalidad del dataset merma el rendimiento. • Lento en fase de predicción.
Arbol de decision	<ul style="list-style-type: none"> • Robusto a muestras con ruido. • Fácil de interpretar. • Resuelve problemas no lineales. 	<ul style="list-style-type: none"> • Cuando hay muchas etiquetas de clase, los cálculos pueden ser complejos. • Puede sufrir sobreajuste.
Clasificador de bosque aleatorio	<ul style="list-style-type: none"> • No sufre sobreajuste como el árbol de decisión. • Funciona muy bien en grandes bases de datos. • Maneja automáticamente los valores faltantes. 	<ul style="list-style-type: none"> • Consumo mucho tiempo y recursos computacionales. • Difícil de interpretar. • Necesito elegir la cantidad adecuada de árboles.
Máquinas de vectores de soporte (SVC)	<ul style="list-style-type: none"> • Eficaz en espacios de gran dimensión. • Puede manejar soluciones no lineales. • Robusto al ruido. 	<ul style="list-style-type: none"> • Lento para entrenarse con grandes conjuntos de datos. • Ineficaz si las clases se superponen. • Hay que elegir una buena función de kernel. • Difícil de interpretar al aplicar kernels no lineales.

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

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

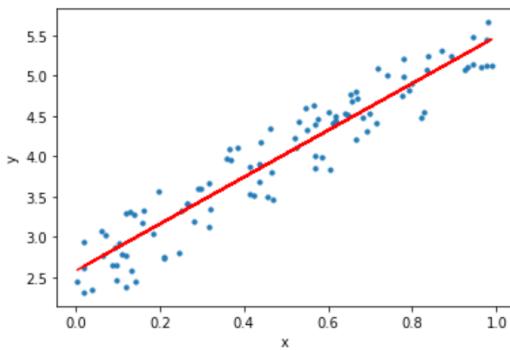


Figura 4: Ejemplo de regresión lineal.

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	<ul style="list-style-type: none"> • El costo computacional no es mayor que otros algoritmos. • Permite evitar el sobreajuste. 	<ul style="list-style-type: none"> • Se necesita una excelente selección del hiperparámetro alpha. • Incrementa el sesgo.
LASSO	<ul style="list-style-type: none"> • Evita el sobreajuste. • Selecciona características tendiendo que sus coeficientes sean cero. 	<ul style="list-style-type: none"> • Las características seleccionadas tienen demasiado sesgo. • Si tenemos n datos y p características, LASSO solo selecciona como máximo n características. • El rendimiento de la predicción es peor que para Ridge Regression.
Elastic Net	<ul style="list-style-type: none"> • Eficaz con muestras de gran dimensión. 	<ul style="list-style-type: none"> • Alto costo computacional en comparación con LASSO o Ridge.
KNN Regresor	<ul style="list-style-type: none"> • No tiene período de entrenamiento. • Fácil de interpretar. • Permite agregar datos al modelo sin inconvenientes en la precisión del algoritmo. 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • No sufre sobreajuste como el árbol de decisión. • Funciona muy bien en grandes bases de datos. • Maneja automáticamente los valores faltantes. 	<ul style="list-style-type: none"> • 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)	<ul style="list-style-type: none"> • Útil cuando las clases no son linealmente separables. 	<ul style="list-style-type: none"> • 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 serían 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	<ul style="list-style-type: none"> • Eficiente cuando la dimensionalidad de los datos es demasiado alta. 	<ul style="list-style-type: none"> • Es sensible ante los outliers.
Isolation Forest	<ul style="list-style-type: none"> • 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.

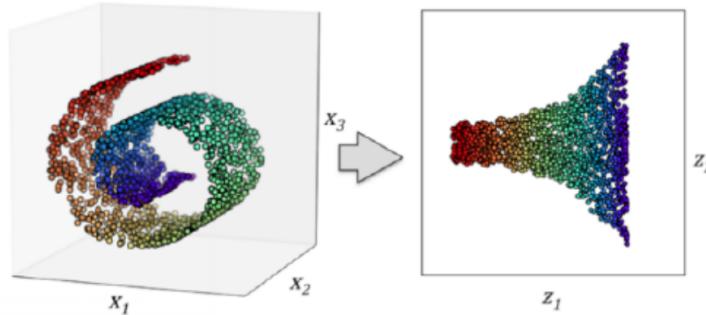


Figura 7: Reducción de dimensionalidad de 3D a 2D.

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	<ul style="list-style-type: none"> • Simple de implementar. • Es uno de los algoritmos más rápidos de reducción de dimensionalidad. 	<ul style="list-style-type: none"> • No puede detectar características no lineales. • Los datos necesitan ser normalizados.
Isomap	<ul style="list-style-type: none"> • Puede detectar características no lineales. 	<ul style="list-style-type: none"> • Lento en grandes cantidades de datos.
T-SNE	<ul style="list-style-type: none"> • Puede detectar características no lineales. • Recomendable cuando se quiere obtener una mejor visualización de un dataset con alta dimensionalidad. 	<ul style="list-style-type: none"> • Computacionalmente costoso 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 agrupamiento, 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.

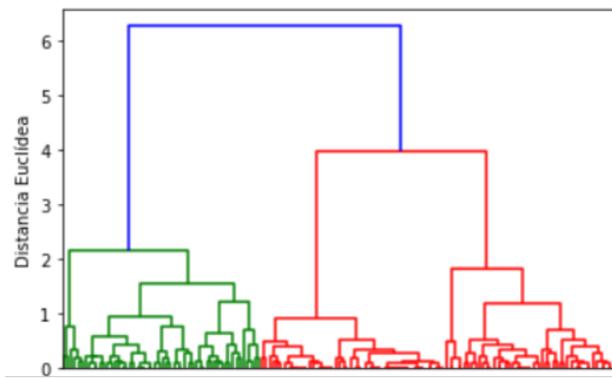


Figura 9: Dendograma generado por *clustering* jerárquico.

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.

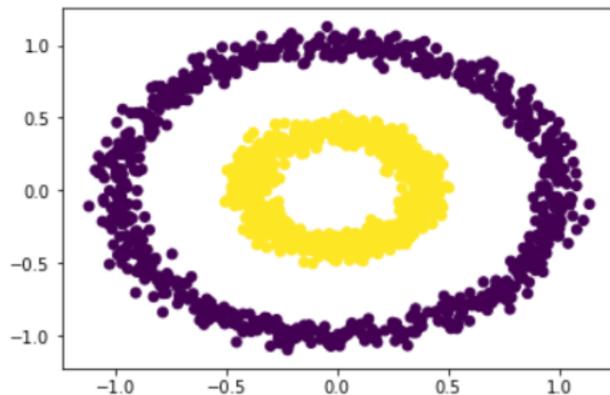


Figura 10: Clustering basado en densidades.

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

	Ventajas	Desventajas
K-Means	<ul style="list-style-type: none"> • Fácil de implementar. • Rápido. 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • No es necesario indicar el número de grupos a priori. • No es sensible a la elección de la métrica de distancia. 	<ul style="list-style-type: none"> • No es muy eficiente.
Mini Batch K-Means	<ul style="list-style-type: none"> • Puede agrupar conjuntos de datos masivos. • Reduce el tiempo de cómputo gracias a los mini-batches. 	<ul style="list-style-type: none"> • La calidad de los resultados podría verse reducida respecto a K-means.
Mean Shift	<ul style="list-style-type: none"> • No es necesario indicar el número de grupos a priori. 	<ul style="list-style-type: none"> • No es escalable para muchos datos.
DBSCAN	<ul style="list-style-type: none"> • No hay que especificar el número de clusters a priori. • Puede detectar grupos con formas irregulares. • Puede detectar outliers. • Robusto al ruido. 	<ul style="list-style-type: none"> • 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 qué hacer con nuestros datos en función a nuestro objetivo.

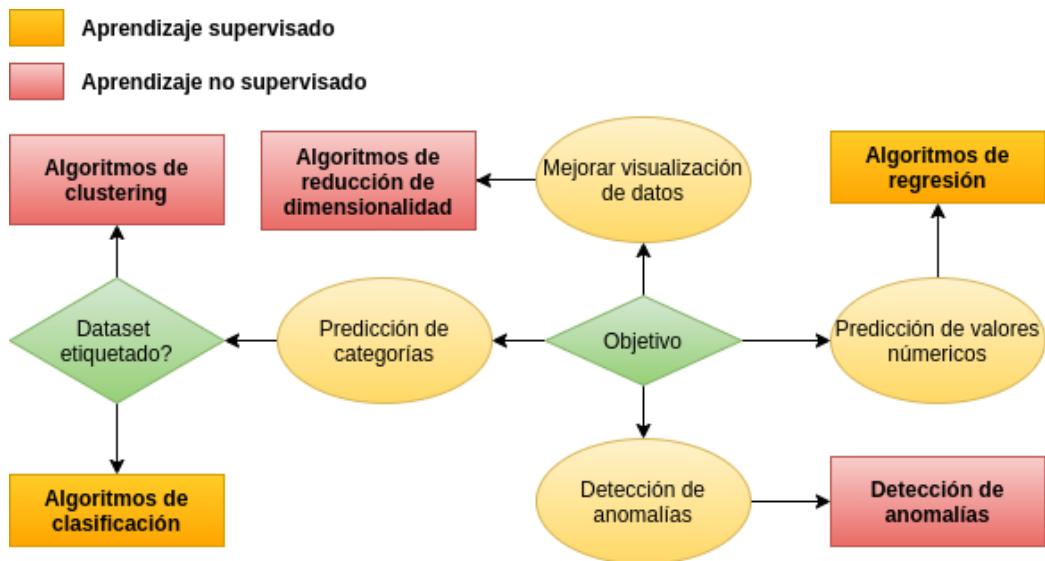


Figura 12: Diagrama general de algoritmos de aprendizaje supervisado y no supervisado.

3.5.1 Algoritmos supervisados

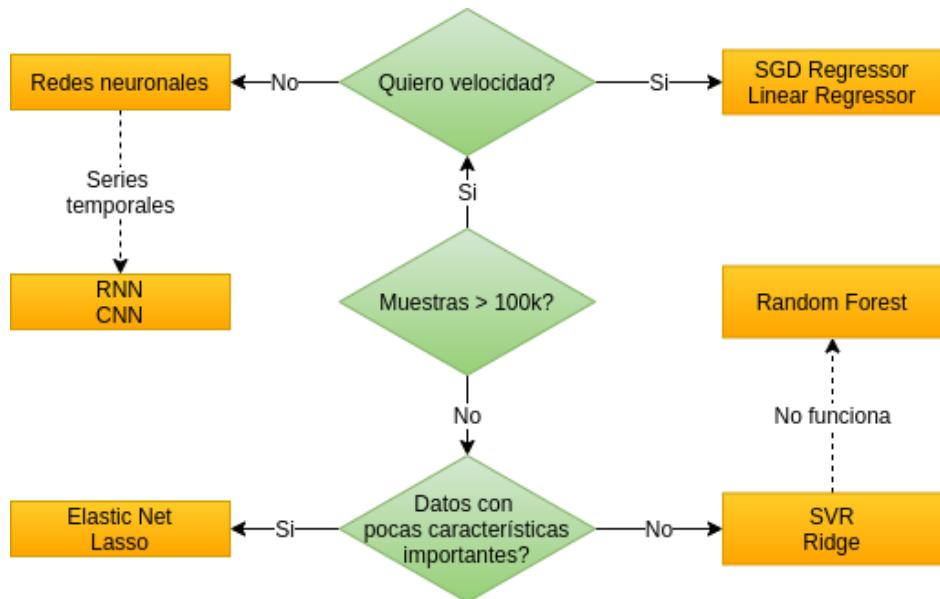


Figura 13: Diagrama general de los algoritmos de regresión.

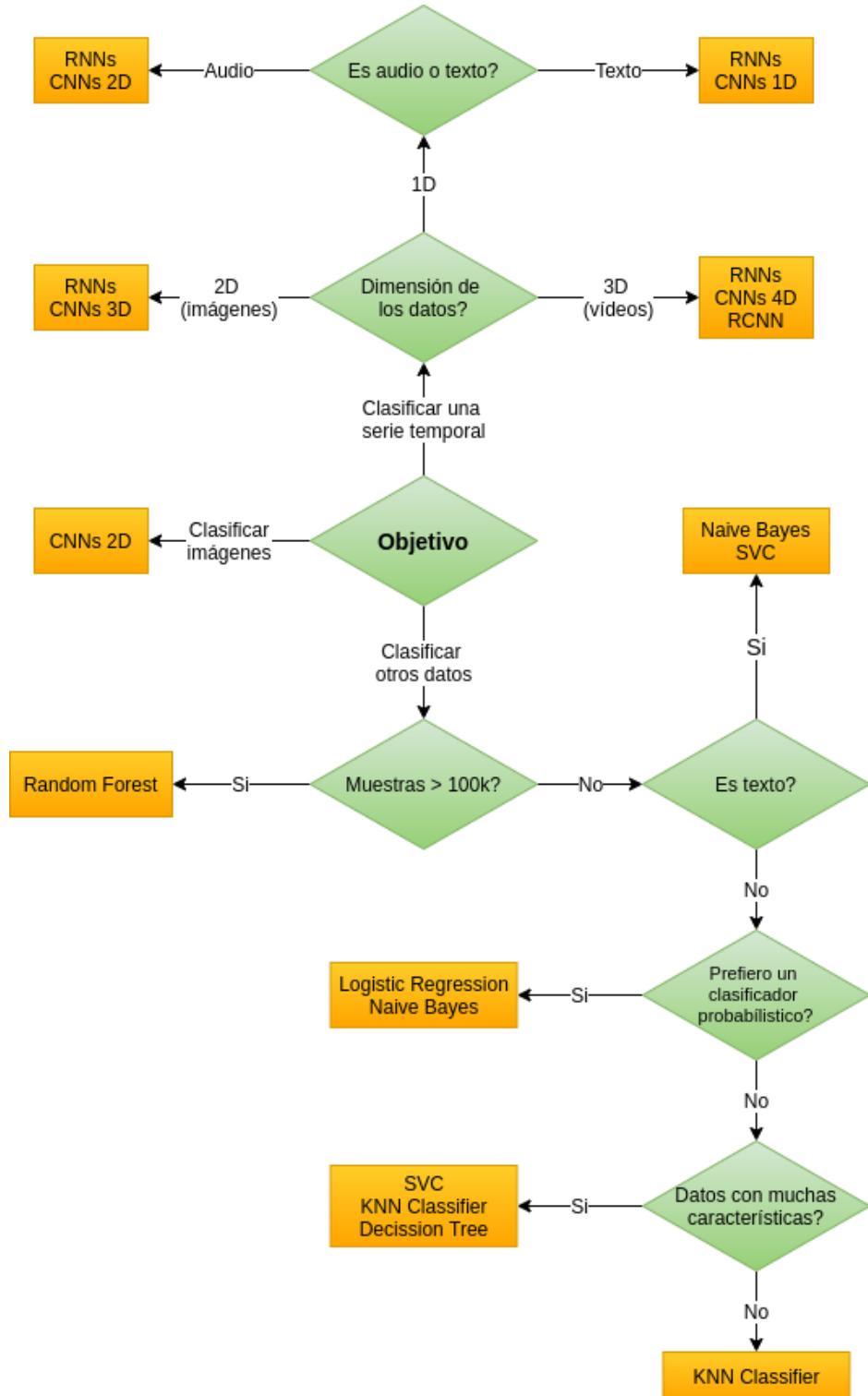


Figura 14: Diagrama general de los algoritmos de clasificación.

3.5.2 Algoritmos no supervisados

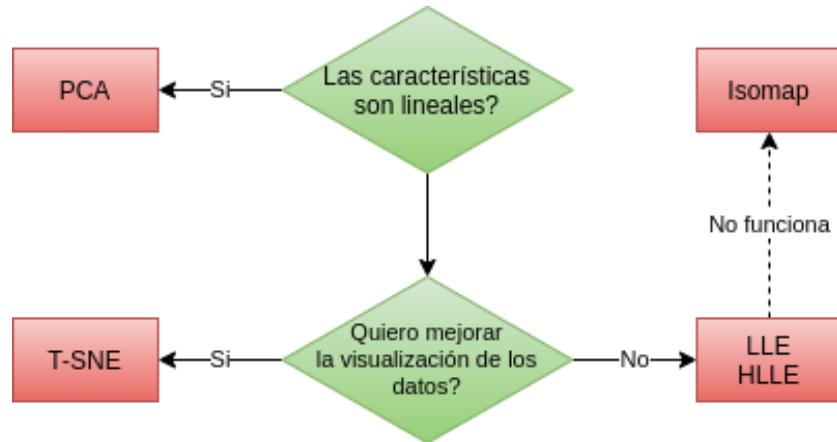


Figura 15: Diagrama general de los algoritmos de reducción de dimensión.

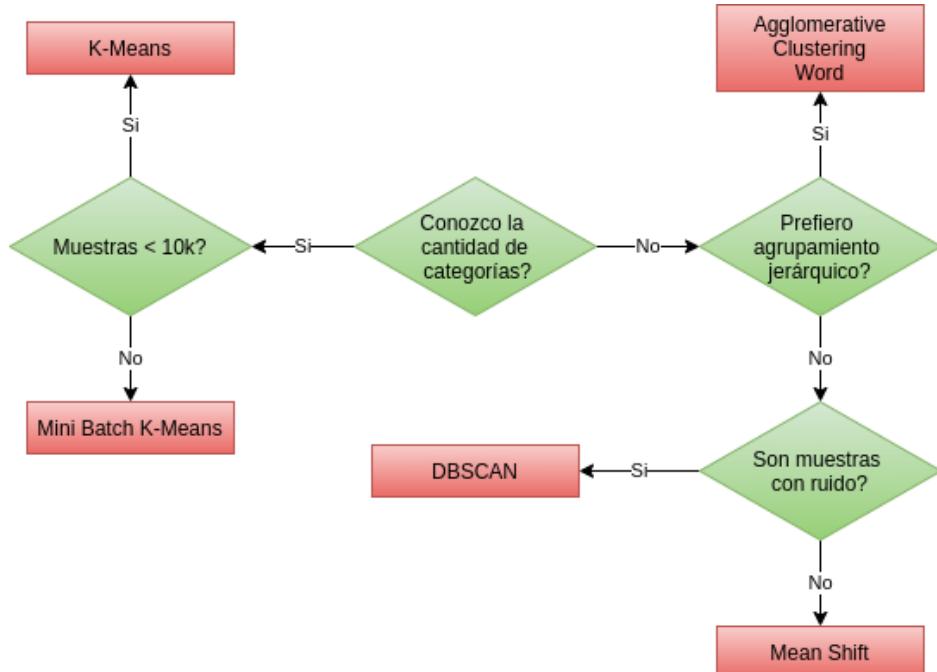


Figura 16: Diagrama general de los algoritmos de *clustering*.



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 abbreviar 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.

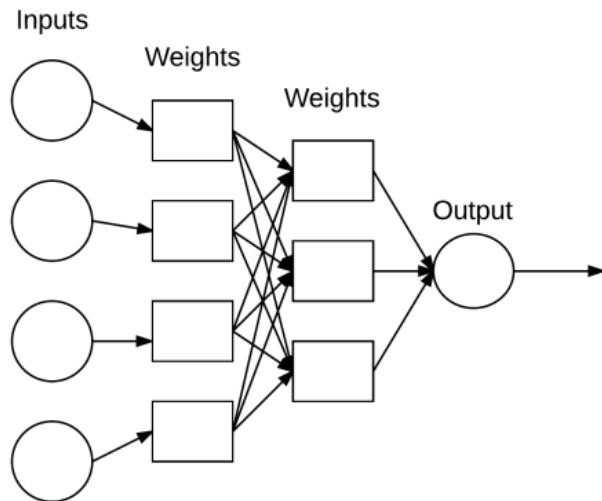


Figura 18: Arquitectura simple de red neuronal. [7]

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).

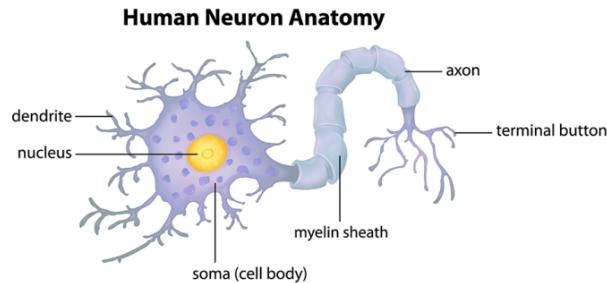


Figura 19: Estructura de una neurona biológica.

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.

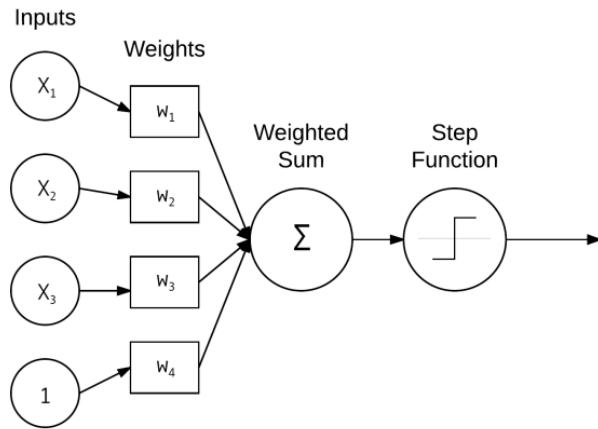


Figura 20: Simple NN.

Cada x está conectada a una neurona a través de un vector de peso W que consiste en w_1, w_2, \dots, 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:

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

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.

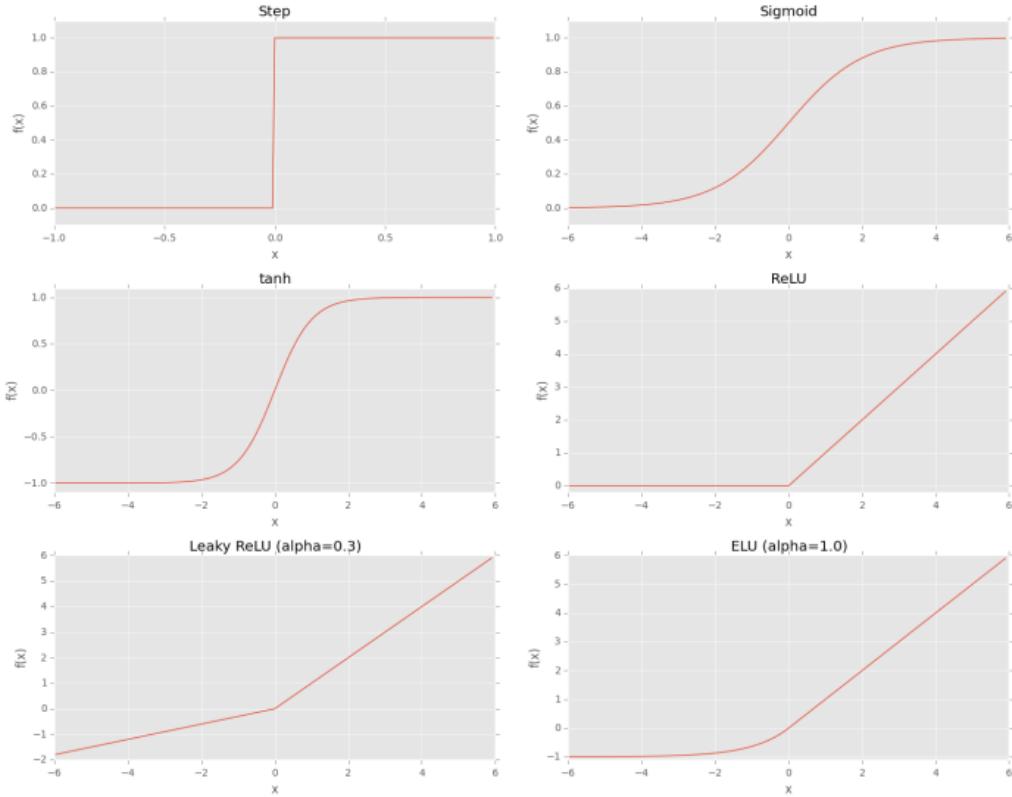


Figura 21: **Arriba-izquierda:** Función escalón. **Arriba-derecha:** Función sigmoidea. **Medio-izquierdo:** Tangente hiperbólica. **Medio-derecho:** activación ReLU (función activación más usada en *Deep Learning*). **Abajo-izquierdo:** Leaky ReLU, variante de ReLU que permite valores negativos. **Abajo-derecha:**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

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

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:

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

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:

$$f(x) = \max(0, x) \quad (3)$$

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 hacia adelante o *feedforward*, como se presenta en la Figura 22.

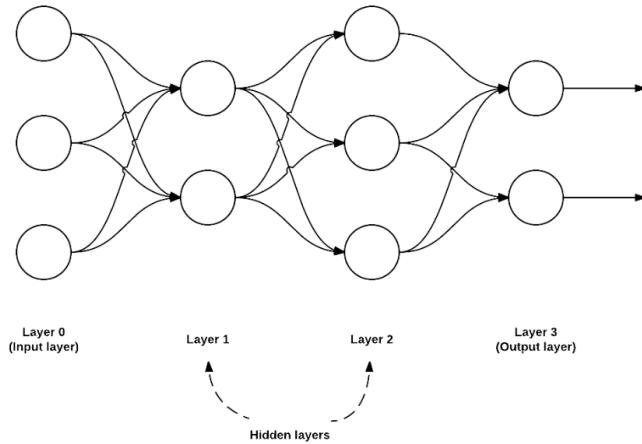


Figura 22: Un ejemplo de una red neuronal *feedforward*.

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:

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

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

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

Por último definimos la salida de nuestro modelo como:

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

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:

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

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

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

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:

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

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.

$$\hat{y} = w_1 x_1 + w_2 x_2 + b \quad (10)$$

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

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 .

$$\frac{\partial J}{\partial w_1} \frac{\partial J}{\partial w_2} \frac{\partial J}{\partial b} \quad (12)$$

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.

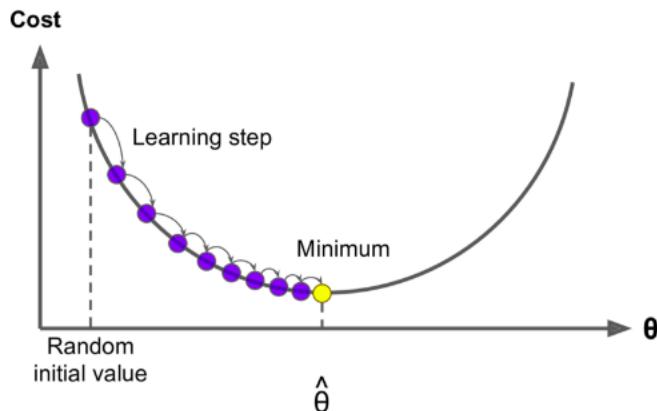


Figura 23: Representación gráfica del descenso de gradiente.

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.

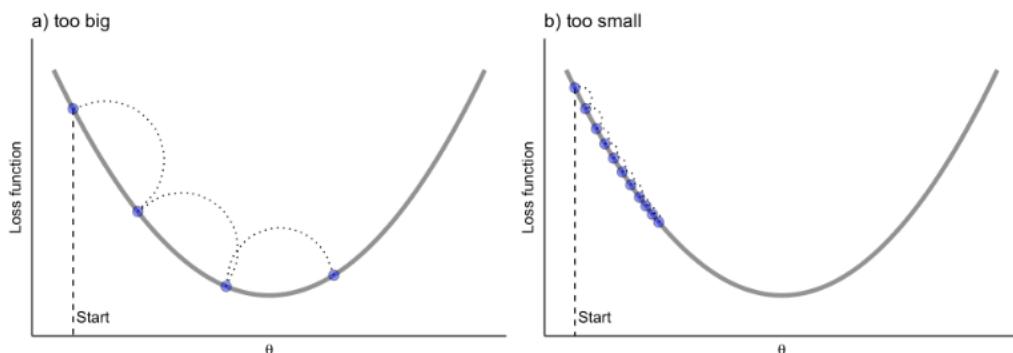


Figura 24: Ajuste de la tasa de aprendizaje.

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

$$W' = W - \alpha \frac{\partial J}{\partial W} \quad (13)$$

$$b' = b - \alpha \frac{\partial J}{\partial b} \quad (14)$$

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.

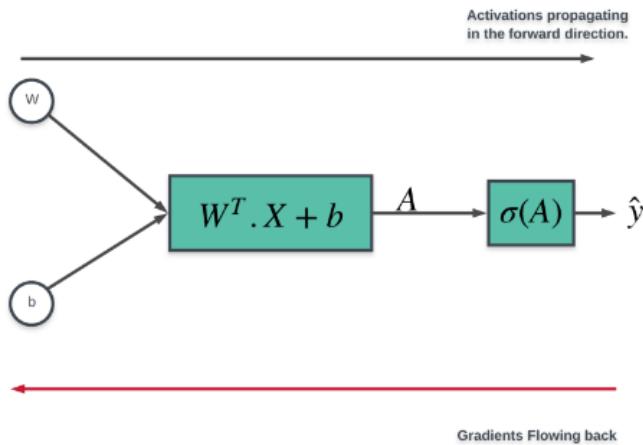
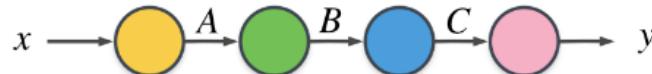


Figura 25: Las activaciones se propagan hacia adelante, pero los gradientes fluyen hacia atrás.

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.



$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial C} \times \frac{\partial C}{\partial B} \times \frac{\partial B}{\partial A} \times \frac{\partial A}{\partial x}$$

Figura 26: Representación de grafo simple.

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

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

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

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

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:

$$\frac{\partial J}{\partial D} = \frac{\partial J}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial A} \quad (17)$$

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.

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

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

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

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

Continuando, podemos simplificar aún más esta ecuación.

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

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

Substituyendo este valor en la ecuación 17 obtenemos:

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

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

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

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, \dots, w_n .

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial A} \frac{\partial A}{\partial W} \quad (26)$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial A} \frac{\partial A}{\partial b} \quad (27)$$

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

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

Y de la ecuación 27:

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

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 última 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 entran 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.

5 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.

5.1 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 $a = a_1, a_2, \dots, a_n$ entonces puede ser representado como el vector $\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}$.

Si las funciones f y g son representadas como vectores $a = \begin{bmatrix} a_1 & a_2 & \dots & a_m \end{bmatrix}$ y $b = \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}$, entonces $f * g$ es un vector $c = \begin{bmatrix} c_1 & c_2 & \dots & c_{m+n-1} \end{bmatrix}$ 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 $u = \max(1, x - n + 1) \dots \min(x, m)$.

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 guardarla en el índice correspondiente del vector *output*.

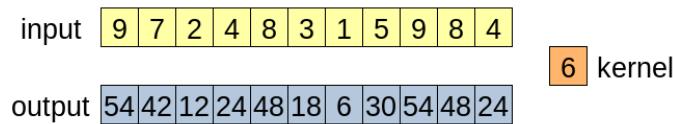


Figura 28: Vectores de convolución unidimensional con kernel simple.

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.

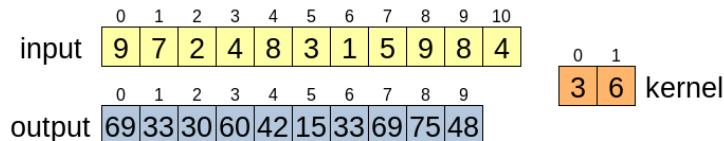


Figura 29: Vectores de convolución unidimensional con kernel doble.

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 $o[4] = i[3]k[0]+i[4]k[1]+i[5]k[2] = 23$.

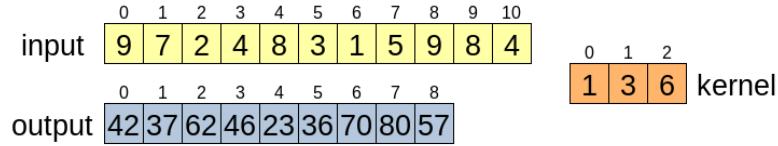


Figura 30: Vectores de convolución unidimensional con kernel triple.

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

$$\text{output}_{\text{size}} = \frac{W - F + 2P}{S + 1}$$

donde $W = \text{input}_{\text{size}}$, $F = \text{kernel}_{\text{size}}$, $P = \text{padding}$ y $S = \text{stride}$.

5.2 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 .

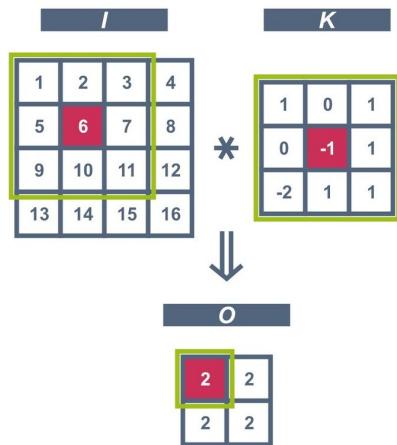


Figura 31: Vectores de convolución bidimensional con kernel 3×3 .

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.

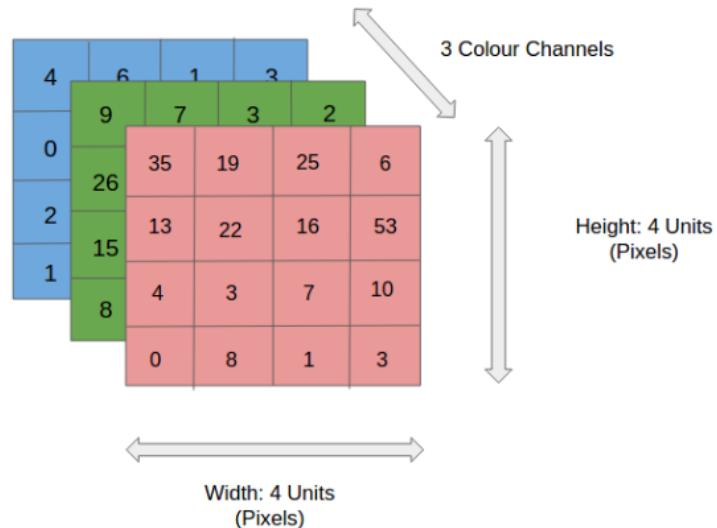


Figura 32: Imagen RGB $4 \times 4 \times 3$.

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.

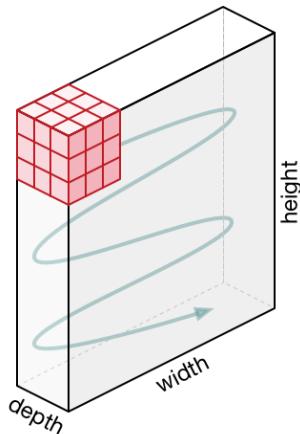


Figura 33: Movimiento del *kernel*.

5.2.1 *Padding*

Un problema a abordar al aplicar la convolución es que tendemos a perder píxeles en el perímetro de nuestra imagen (o vector). Dado que normalmente usamos núcleos pequeños, para cualquier convolución dada, es posible que solo perdamos unos pocos píxeles, pero esto puede sumarse a medida que aplicamos muchas capas convolucionales sucesivas. Una solución sencilla a este problema es agregar píxeles adicionales de relleno (*padding*) alrededor del límite de nuestra imagen de entrada, aumentando así el tamaño efectivo de la imagen. Normalmente, establecemos los valores de los píxeles adicionales en cero. [16]

En la figura 34, rellenamos una entrada de 3×3 , aumentando su tamaño a 5×5 . La salida correspondiente aumenta entonces a una matriz de 4×4 .

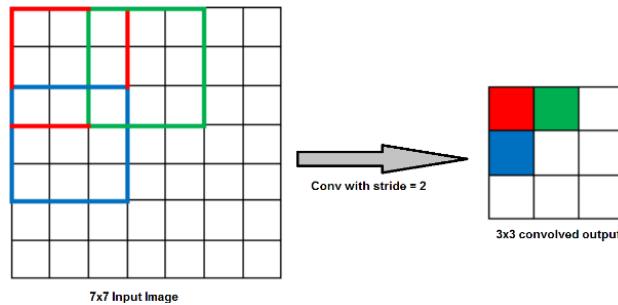
Input	Kernel	Output
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 25 & 10 \\ 21 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix}$

Figura 34: Relleno o *padding*.

5.2.2 *Stride*

Al realizar la convolución, comenzamos con la ventana en la esquina superior izquierda del tensor de entrada y luego la deslizamos sobre todas las ubicaciones, tanto hacia abajo como hacia la derecha. Usualmente deslizamos un elemento a la vez. Sin embargo, a veces, ya sea por eficiencia computacional o porque deseamos reducir la resolución, movemos nuestra ventana más de un elemento a la vez, omitiendo las ubicaciones intermedias.

Nos referimos al número de filas y columnas atravesadas por diapositiva como la zancada o *stride*. Hasta ahora, hemos utilizado *stride* de 1, tanto para altura como para ancho. A veces, es posible que deseemos usar un paso más grande, como en la figura 35 [15].

Figura 35: Convolución con *stride* de 2.

5.3 Tipos de capas

Existen varias tipos de capas usadas [6] para construir *CNNs* pero las más comunes incluyen:

- Convolucional (CONV)

- Activación (ACT)
- *Pooling* (POOL)
- *Fully-connected* (FC)
- *Dropout* (DO)

Apilando estas capas de una manera específica producimos una *CNN*. De estos tipos de capas, CONV y FC (y en menor medida, BN) son las únicas capas que contienen parámetros que se aprenden durante el proceso de entrenamiento.

Las capas ACT y DO no se consideran verdaderas capas en sí mismas, pero a menudo se incluyen en los diagramas de red para que la arquitectura sea explícitamente clara.

Las capas (POOL), de igual importancia que CONV y FC, también se incluyen en los diagramas de red, ya que tienen un impacto sustancial en las dimensiones espaciales de un imagen mientras se mueve a través de una *CNN*.

5.3.1 Convolución

La capa convolucional (CONV) es el componente básico de una red neuronal convolucional. Los parámetros de la capa CONV consisten en un conjunto de K *kernels* entrenables, donde cada uno tiene un ancho y un alto, y casi siempre son cuadrados.

Una capa o filtro puede tener varios *kernels* que al convolucionar con el vector de entrada (que podría ser una imagen) producen mapas de características (*features map*).

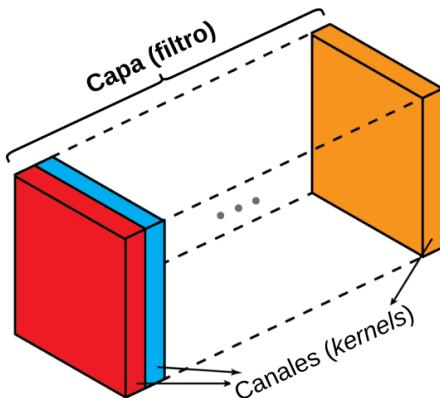


Figura 36: Una capa está compuesta de una colección de *kernels*.

La diferencia entre filtro y *kernel* es un poco complicada. A veces, se usan indistintamente, lo que podría crear confusiones. Esencialmente, estos dos términos tienen una sutil diferencia. Un "*kernel*" se refiere a una matriz 2D de pesos. El término "filtro" se refiere a estructuras 3D de varios *kernels* apilados juntos. Para un filtro 2D, el filtro es igual que el kernel. Pero para un filtro 3D y la mayoría de las convoluciones en el aprendizaje profundo, un filtro es una colección de *kernels* (fig. 36). Cada *kernel* es único, enfatizando diferentes aspectos del canal de entrada.

El funcionamiento de la capa convolucional se resume en pensar en cada uno de los K *kernels* deslizándose a través del vector de entrada, computando un producto de Hadamard, sumando cada uno de sus valores y luego almacenando el valor generado en un mapa 2D de activación. En la figura 37 se puede observar una visualización de la secuencia.



Figura 37: **Izquierda:** en cada capa convolucional de una *CNN*, hay K *kernels* distintos. **Medio:** Cada uno de los *kernels* es convolucionado con el vector de entrada. **Derecha:** Cada *kernel* produce una salida 2D, llamada mapa de activación o *features*.

Luego de aplicar los K filtros al vector de entrada, ahora tenemos $K \times 2D$ mapas de activación. Luego apilamos nuestro K mapas de activación a través de la dimensión de profundidad de nuestra matriz para formar el volumen final de salida (figura 38).

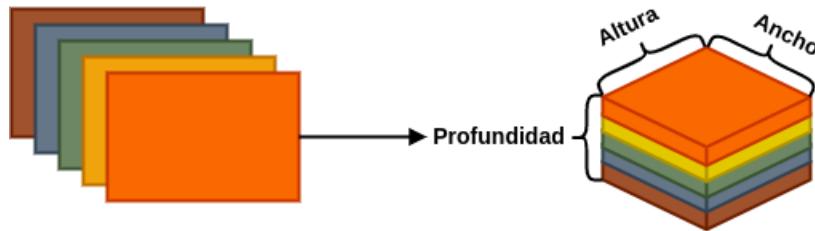


Figura 38: Luego de obtener los K mapas de activación, son apilados juntos para formar el volumen de entrada a la siguiente capa en la red.

Cada entrada en el volumen de salida es, por tanto, una salida de una neurona que "mira" sólo una pequeña región de la entrada. De esta manera, la red "aprende" los filtros que se activan cuando ven un tipo específico de característica en una ubicación espacial determinada en el volumen de entrada.

En las capas inferiores de la red, los filtros pueden activarse cuando ven regiones con forma de borde o de esquina. Luego, en las capas más profundas de la red, los filtros pueden activarse en presencia de características de alto nivel, como partes de la cara, la pata de un perro, el capó de un automóvil, etc.

5.3.2 Activación

Luego de cada capa CONV en una CNN, normalmente aplicamos una función no lineal, como ReLU, ELU, etc (se ejemplifica en la figura 39). Las capas ACT no son técnicamente "capas" (debido al hecho de que no se aprenden parámetros/pesos dentro de una capa de activación) y, a veces, se omiten en los diagramas de arquitectura de red, ya que se supone que una activación sigue inmediatamente a una convolución.

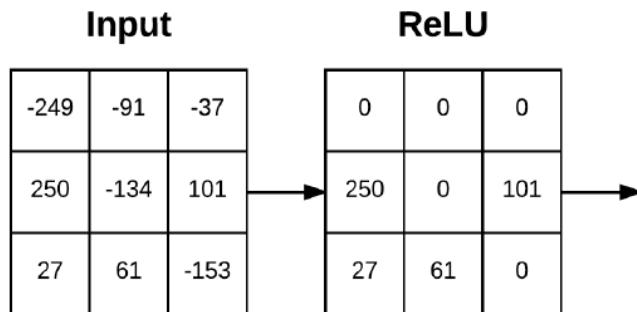


Figura 39: Un ejemplo de un volumen de entrada desplazándose a través de una ACT ReLU.

5.3.3 *Fully-connected*

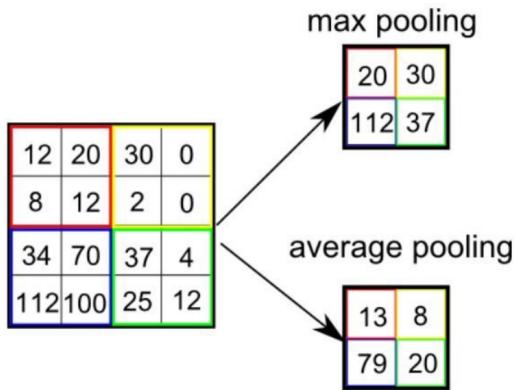
Las neuronas en las capas FC están totalmente conectadas a todas las activaciones de la capa anterior, como en una red neuronal *feed-forward*. Siempre se ubican al final de la red.

5.3.4 *Pooling*

Similar a la capa convolucional, la capa de POOL es responsable de reducir el tamaño espacial de la entidad convolucionada. Esto es para disminuir la potencia computacional requerida para procesar los datos a través de la reducción de dimensionalidad. Además, es útil para extraer características dominantes que son invariantes rotacionales y posicionales, manteniendo así el proceso de entrenamiento efectivo del modelo.

Existen dos tipos:

- *Max pooling* (MPOOL): devuelve el valor máximo de la parte de la imagen cubierta por el *kernel* (figura 41 arriba).
- *Average pooling* (APOLL): devuelve el promedio de todos los valores de la parte de la imagen cubierta por el *kernel* (figura 41 abajo).

Figura 40: Tipos de *pooling*.

Cabe destacar que MAXPOOL también actúa como supresor de ruido ya que descarta las activaciones ruidosas por completo además de la reducción de dimensionalidad. Por otro lado, el AVGPOOL simplemente realiza la reducción de dimensionalidad como un mecanismo de supresión de ruido. Por lo tanto, podemos decir que MAXPOOL funciona mucho mejor que AVGPOOL.

5.3.5 *Batch Normalization*

Por lo general, para entrenar una red neuronal, realizamos un preprocesamiento de los datos de entrada, por ejemplo, normalizar todos los datos para que se parezcan a una distribución normal (es decir, media cero y una varianza unitaria). Algunas razones para realizar esto puede ser prevenir la saturación temprana de funciones de activación no lineales como la función sigmoidea, asegurar que todos los datos de entrada estén en el mismo rango de valores, etc. [17]

Pero el problema aparece en las capas intermedias porque la distribución de las activaciones cambia constantemente durante el entrenamiento. Esto ralentiza el proceso de entrenamiento porque cada capa debe aprender a adaptarse a una nueva distribución en cada paso del entrenamiento. Este problema se conoce como **cambio de covariables interno**.

Podemos utilizar la normalización por lotes o *Batch Normalization* (BN) como un método para normalizar las entradas de cada capa para así forzarlo a tener aproximadamente la misma distribución en cada paso de entrenamiento, con el fin de combatir el problema expresado anteriormente.

Durante el tiempo de entrenamiento, una capa de normalización por lotes se obtiene el promedio y la varianza del lote:

$$\mu_\beta = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_\beta^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_\beta)^2$$

Normalizamos las entradas de la capa usando las estadísticas del lote calculados previamente:

$$\bar{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}$$

Establecemos $1e-7 \leq \epsilon \leq 0$ para evitar sacar la raíz cuadrada de cero. Aplicar esta ecuación implica que las activaciones que salen de una capa BN tendrán una media y una varianza unitaria aproximadamente cero (es decir, centrada en cero). Reemplazamos el mini-lote μ_β y σ_β con promedios de μ_β y σ_β calculados durante el proceso de entrenamiento. Esto asegura que podemos pasar vectores a través de nuestra red y aún así obtener predicciones precisas sin ser sesgados por μ_β y σ_β del mini-lote final pasado a través de la red en el momento del entrenamiento.

La BN también tiene el beneficio adicional de ayudar a "estabilizar" el entrenamiento, lo que permite una mayor variedad de tasas de aprendizaje y fortalezas de regularización. Esto no alivia la necesidad de ajustar estos parámetros, por supuesto, pero le facilitará la vida al hacer que la tasa de aprendizaje y la regularización sean menos volátiles y más fáciles de ajustar. También tenderá a notar pérdidas finales más bajas y una curva de pérdida más estable en sus redes.

5.3.6 Dropout

El *dropout* (DO) es en realidad una forma de regularización que tiene como objetivo ayudar a prevenir el sobreajuste aumentando la precisión de las pruebas, quizás a expensas de la precisión del entrenamiento. [6]

La razón está en reducir el sobreajuste alterando de forma explícita la arquitectura de la red en tiempo de entrenamiento. La desconexión aleatoria de las conexiones garantiza que ningún nodo de la red sea responsable de la activación cuando se le presenta un patrón determinado. El DO garantiza que haya múltiples nodos redundantes que se activarán cuando se les presenten entradas similares (lo que a su vez ayuda a que nuestro modelo a generalizar).

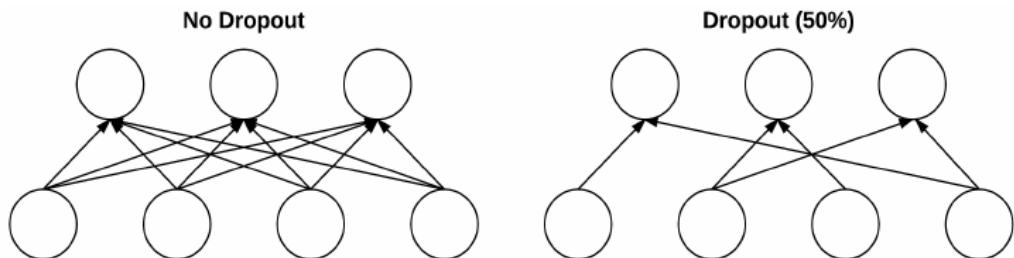


Figura 41: **Izquierda:** Dos capas FC sin D0. **Derecha:** Las mismas dos capas luego de realizar *dropout* sobre la mitad de las conexiones.

5.4 *WaveNet* y capas convolucionales causales dilatadas

WaveNet es una red neuronal profunda para generar audio muestra a muestra. La arquitectura de este modelo permite aprovechar las eficiencias de las capas de convolución al mismo tiempo que alivia el desafío de aprender las dependencias a largo plazo en una gran cantidad de pasos de tiempo (más de 1000) [18].

En el núcleo de *WaveNet* se encuentra la **capa de convolución causal dilatada** (figura 42), que le permite tratar adecuadamente el orden temporal y manejar las dependencias a largo plazo sin una explosión en la complejidad del modelo. [19]

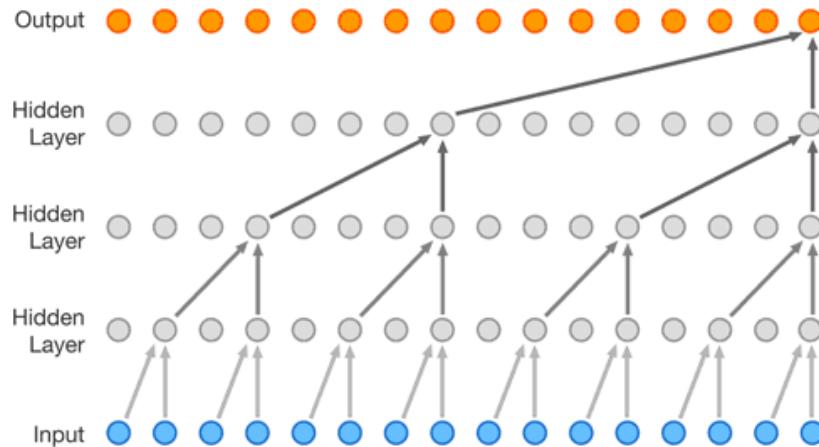


Figura 42: Paso de la información a través de la capa de convolución causal dilatada.

En una capa de convolución unidimensional tradicional, deslizamos un filtro de pesos a través de una serie de entrada, aplicándolo secuencialmente a las regiones (generalmente superpuestas) de la serie. Pero cuando utilizamos el historial de una serie temporal para predecir su futuro, debemos tener cuidado. A medida que formamos capas que eventualmente conectan los pasos de entrada a las salidas, debemos asegurarnos de que las entradas no influyan en los pasos de salida que los siguen a tiempo. De lo contrario, estaríamos usando el futuro para predecir el pasado, lo que sería hacer trampa.

Para asegurarnos de no hacer trampa de esta manera, ajustamos nuestro diseño de convolución para prohibir explícitamente que el futuro influya en el pasado. En otras palabras, solo permitimos que las entradas se conecten a salidas de pasos de tiempo futuros en una estructura **causal**, como se muestra a continuación en una visualización del documento WaveNet. En la práctica, esta estructura 1D causal es fácil de implementar *shifteando* las salidas convolucionales tradicionales en varios pasos de tiempo.

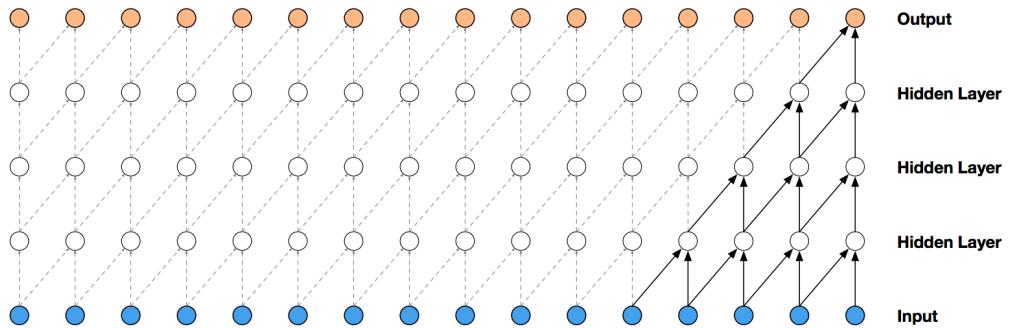


Figura 43: Visualización de una pila de capas causales convoluciones.

Las convoluciones causales proporcionan la herramienta adecuada para manejar el flujo temporal, pero necesitamos una modificación adicional para manejar adecuadamente las dependencias a largo plazo. En la figura 43 de convolución causal simple, puede ver que solo los 5 pasos de tiempo más recientes pueden influir en la salida resaltada. De hecho, necesitaríamos una capa adicional por paso de tiempo para llegar más atrás en la serie (para usar la terminología adecuada, para aumentar el **campo receptivo de la salida**). Con una serie de tiempo que tiene una gran cantidad de pasos, el uso de convoluciones causales simples para aprender de toda la historia rápidamente haría un modelo demasiado complejo computacional y estadísticamente.

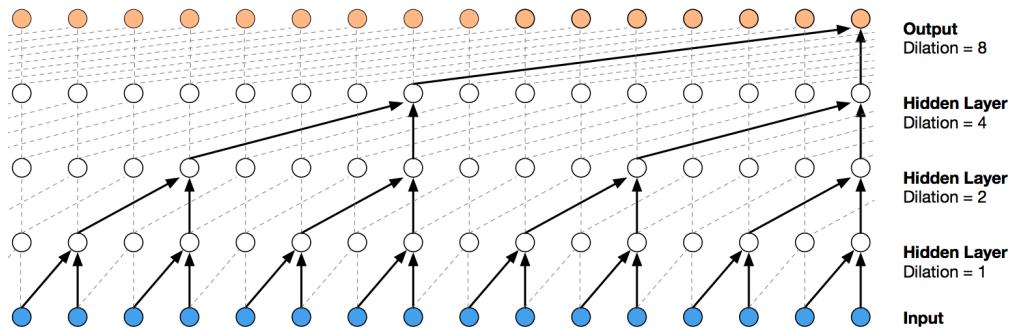


Figura 44: Visualización de una pila de capas causales convoluciones dilatadas.

En lugar de cometer ese error, *WaveNet* utiliza **convoluciones dilatadas**, que permiten que el campo receptivo aumente exponencialmente en

función de la profundidad de la capa de convolución.

En una capa de convolución dilatada, los filtros no se aplican a las entradas de una manera secuencial simple, sino que omiten una entrada de tasa de dilatación constante entre cada una de las entradas que procesan, como en el diagrama *WaveNet* a continuación. Al aumentar la tasa de dilatación multiplicativamente en cada capa (por ejemplo, 1, 2, 4, 8,), podemos lograr la relación exponencial entre la profundidad de la capa y el tamaño del campo receptivo que deseamos.

En la figura 44, puede ver cómo ahora solo necesitamos 4 capas para conectar los 16 valores de la serie de entrada a la salida resaltada (digamos, el valor de paso de tiempo 17). Por extensión, cuando se trabaja con una serie de tiempo diaria, se puede capturar más de un año de historia con solo 9 capas de convolución dilatadas de esta forma.

6 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*) [20] 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.

6.1 Arquitecturas

Existen diversas arquitecturas disponibles para estas redes como observamos en la Figura 45 [20], 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*.

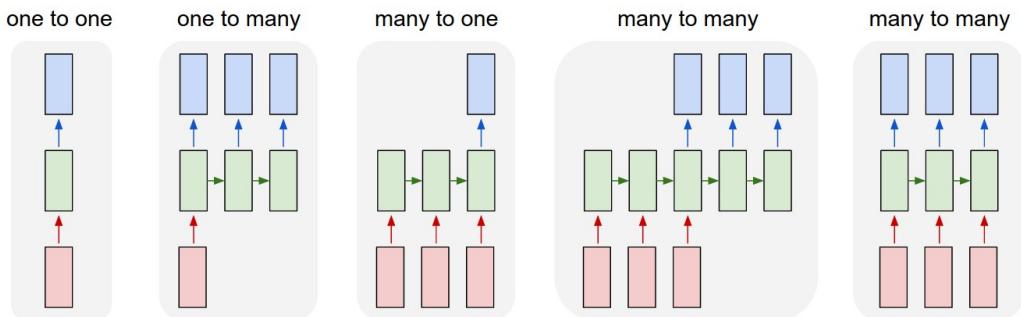


Figura 45: Tipos de arquitecturas para una *RNN*.

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.

6.2 Funcionamiento

Consideremos la Figura 46 [21], 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 .

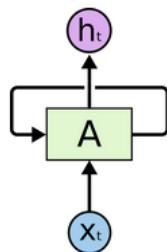


Figura 46: Unidad funcional de *RNN*.

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 47.

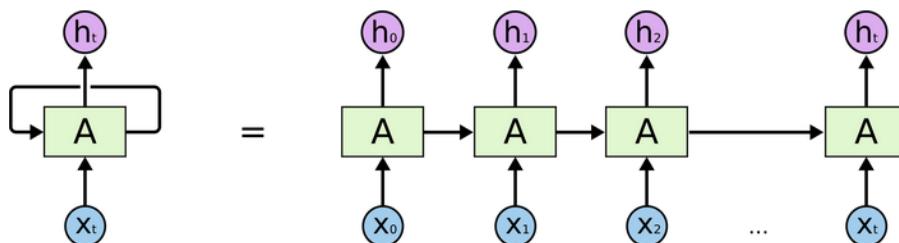


Figura 47: Una *RNN* desenrollada.

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 hacia atrás. La *RNN* más simple posible es la que mostramos en la figura

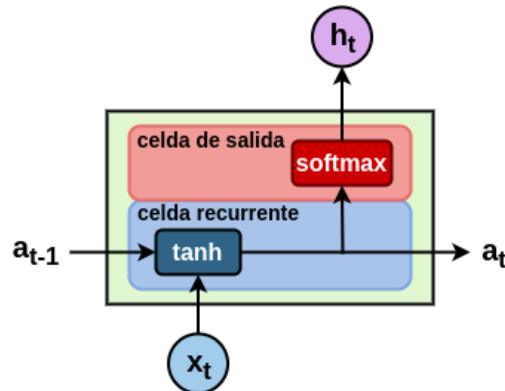


Figura 48: Unidad funcional *RNN* detallada.

En la Figura 48 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:

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

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

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 pedimos 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.

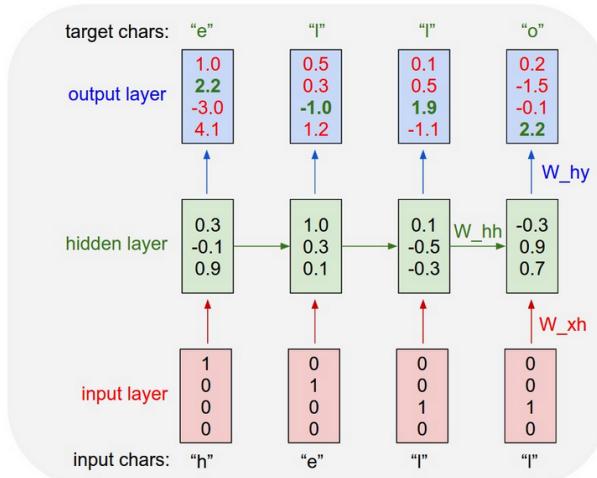


Figura 49: Unidad funcional *RNN* detallada.

En la Fig. 49 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-propagation* 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.

6.3 Entrenamiento

Para entrenar una *RNN*, el truco simplemente es desenrollarla 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. 50 [22].

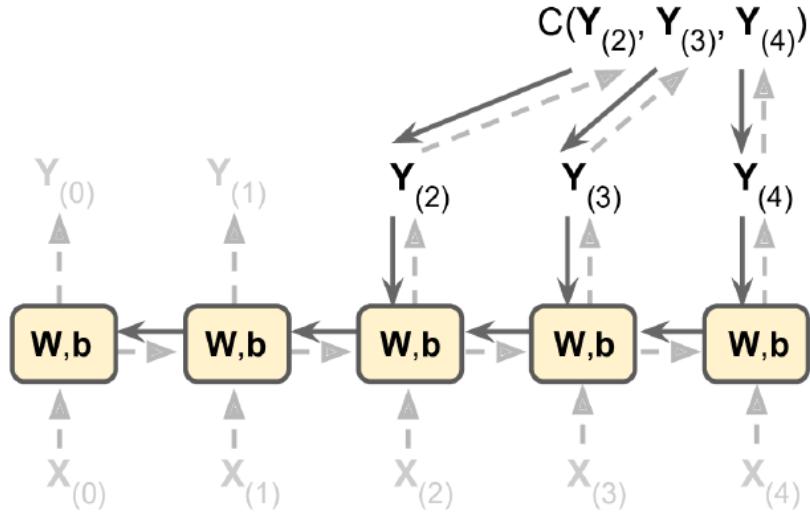


Figura 50: *Backpropagation* a través del tiempo.

Primero realizamos una pasada hacia 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 $C(Y_{(0)}, Y_{(1)}, \dots, Y_{(T)})$ (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. 50. Los gradientes de esa función de costo luego son propagados hacia atrás a través de la red desenrollada (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 hacia 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 $Y_{(1)}$).

6.4 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. 51 intentaremos predecir la intención del usuario tomando como entrada la oración "What time is it?". [23].

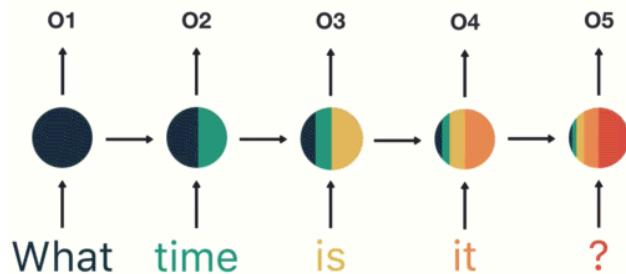


Figura 51: *RNN* siendo alimentada con las palabras de la oración.

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. 52).

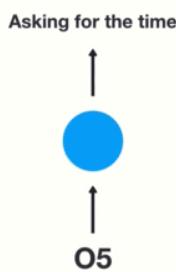


Figura 52: Predicción de la *RNN*.

Pero prestemos atención a la Fig. 53. 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.

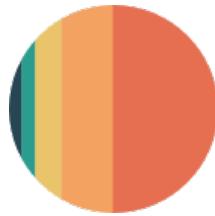


Figura 53: Estado oculto final de la *RNN*.

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.

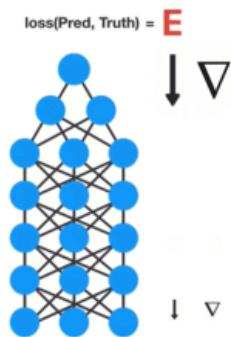


Figura 54: Desvanecimiento del gradiente desde las capas superiores a las inferiores.

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.

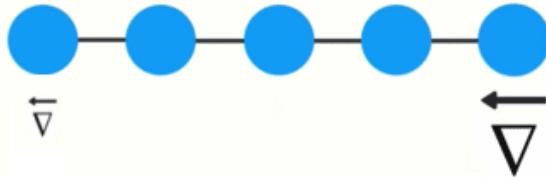


Figura 55: El gradiente se achica a medida que se propaga hacia atrás en el tiempo.

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.

6.5 Tipos de *RNNs*

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. Las otras se denominan *Gated Recurrent Units* o *GRU*.

6.5.1 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. [21]

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.

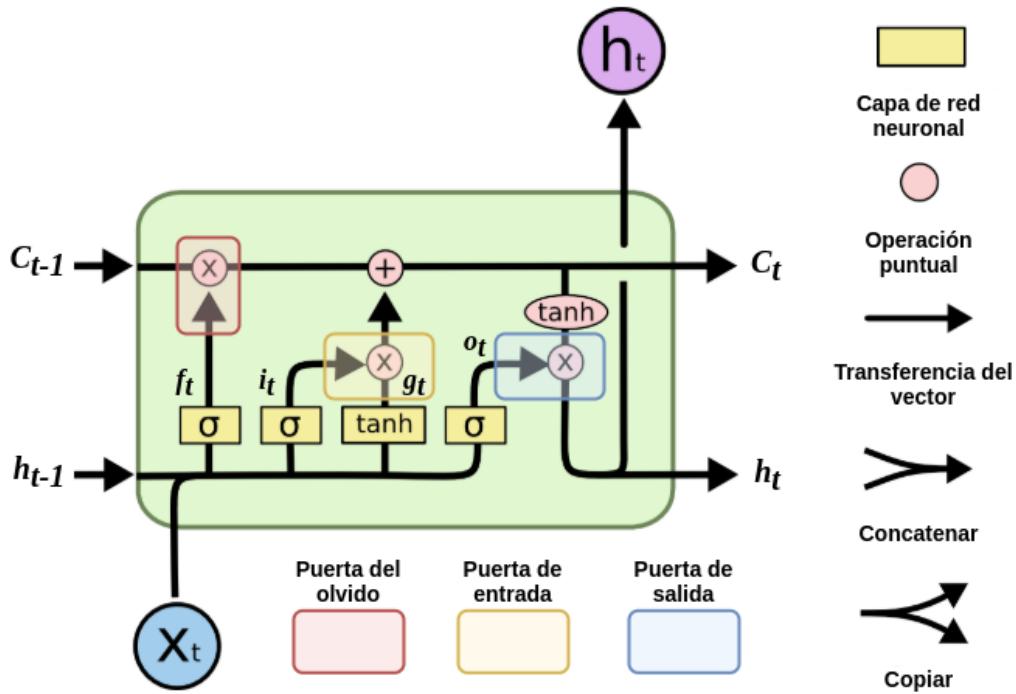


Figura 56: Celda LSTM.

El diagrama completo del LSTM lo podemos observar en la Fig.56, 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. 57. 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.

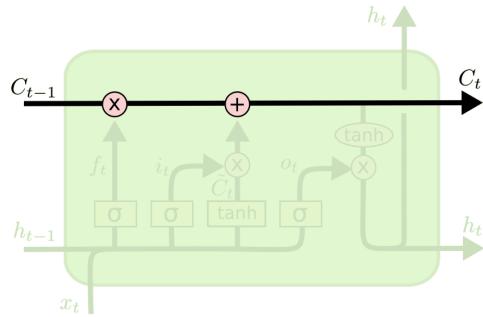


Figura 57: Celda de estado.

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. 58). 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".

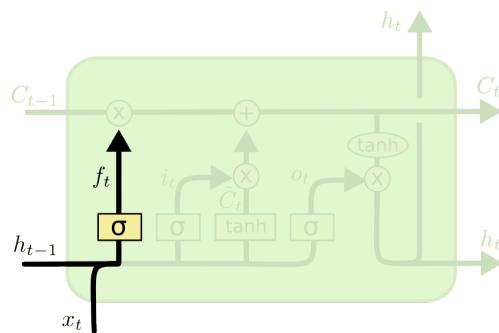


Figura 58: Puerta del olvido.

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.

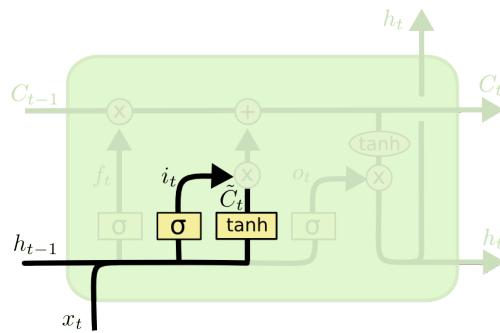


Figura 59: Puerta de entrada.

Una capa sigmoidea llamada **puerta de entrada** (Fig 59) 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.

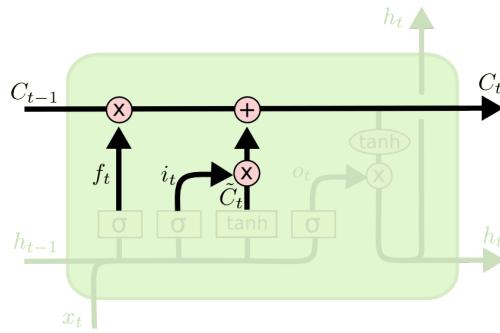


Figura 60: Actualización de la celda.

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.

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

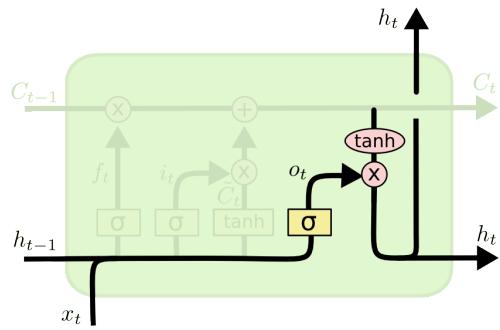


Figura 61: Puerta de salida.

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. 56, 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 [22]:

- 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 [24]), 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.

6.5.2 GRU

Otra variante popular es la celda GRU (Unidad Recurrente Cerrada, *Gated Recurrent Unit*). [21]

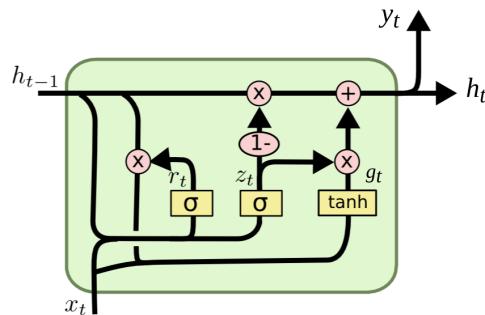


Figura 62: Celda GRU.

Hace algunas simplificaciones [22]:

- 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:

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

6.6 Secuencias de entradas y salida

Si bien en la Sección 6.1 ya desglosamos los diferentes tipos de arquitecturas que puede tener una *RNN*, sería de utilidad ahora que ya poseemos un

marco teórico aceptable describir las un poco más y ejemplificar en que casos sería provechosa su utilización. Tomaremos como referencia la Figura 63 [22].

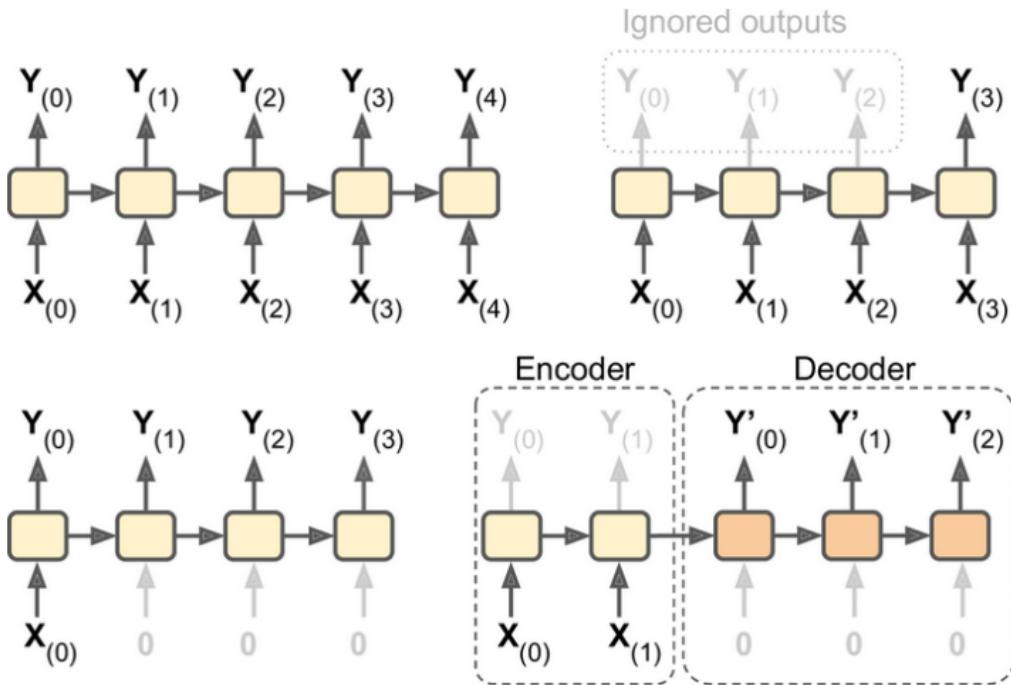


Figura 63: Superior-izquierda: *seq-to-seq*. Superior-derecha: *sec-to-vector*. Inferior-izquierda: *vector-to-sec*. Inferior-derecha: *encoder-decoder*.

Secuencia a secuencia

Una *RNN* puede tomar simultáneamente una secuencia de entradas y producir una secuencia de salidas (red **superior-izquierda**). Por ejemplo, este tipo de red es útil para predecir series de tiempo como los precios de las acciones: usted le da los precios de los últimos N días y debe generar los precios desplazados un día en el futuro (es decir, de $N - 1$ días hace hasta mañana).

Secuencia a vector

Alternativamente, puede alimentar a la red con una secuencia de entradas e ignorar todas las salidas excepto la última (red **superior-derecha**). En otras palabras, esta es una red de secuencia a vector. Por ejemplo, podría alimentar a la red con una secuencia de palabras correspondiente a una crítica de película y la red generaría una puntuación de sentimiento (*e.g.*, de 0 [odio]

a 1 [amor]).

Vector a secuencia

Por el contrario, puede alimentar la red con una sola entrada en el primer paso de tiempo (y ceros para todos los demás pasos de tiempo) y dejar que genere una secuencia (red **inferior-izquierda**). Por ejemplo, la entrada podría ser una imagen y la salida podría ser un título para esa imagen.

Secuencia a secuencia con retardo

Por último, podría tener una red de secuencia a vector, llamada *encoder*, seguida de una red de vector a secuencia, llamada *decoder* (consulte la red de la parte inferior derecha). Por ejemplo, esto se puede utilizar para traducir una oración de un idioma a otro. Alimentaría la red con una oración en un idioma, el *encoder* convertiría esta oración en una representación de vector único y luego el *decoder* decodificaría este vector en una oración en otro idioma. Este modelo de dos pasos, llamado *encoder-decoder*, funciona mucho mejor que intentar traducir sobre la marcha con un único *RNN* secuencia a secuencia (red **Superior-izquierda**), ya que las últimas palabras de una oración pueden afectar las primeras palabras de la traducción, por lo que debe esperar hasta que haya escuchado la oración completa antes de traducirla.

7 Series temporales

Una serie temporal o cronológica es una sucesión de datos medidos en determinados momentos y ordenados cronológicamente.

Poole et al. [1]

7.1 Clasificación del modelo

Los datos de entrada se pueden subdividir aún más para comprender mejor su relación con la variable de salida.

7.1.1 Tipo de variables de entrada

Una variable de entrada es:

- **endógena** si se ve afectada por otras variables del sistema y la variable de salida depende de ella.
- **exógena** si es independiente de otras variables del sistema y la variable de salida depende de ella.

En pocas palabras, las variables endógenas están influenciadas por otras variables del sistema (incluidas ellas mismas), mientras que las variables exógenas no lo están y se consideran fuera del sistema.

Por lo general, un problema de pronóstico de series de tiempo tiene variables endógenas (por ejemplo, el resultado es una función de cierto número de pasos de tiempo anteriores) y puede tener o no variables exógenas.

A menudo, las variables exógenas se ignoran debido al fuerte enfoque en las series de tiempo. Pensar explícitamente en ambos tipos de variables puede ayudar a identificar datos exógenos que se pasan por alto fácilmente o incluso características de ingeniería que pueden mejorar el modelo.

7.1.2 Objetivo

Los problemas de modelado predictivo de:

- **regresión** son aquellos en los que se predice una cantidad.
- **clasificación** son aquellos en los que se predice una categoría.

7.1.3 Estructura

Es útil trazar cada variable en una serie temporal e inspeccionar la trama en busca de posibles patrones.

- **No estructurado:** No hay un patrón dependiente del tiempo sistemático obvio o discernible en una variable de serie temporal.
- **Estructurado:** Patrones sistemáticos dependientes del tiempo en una variable de serie temporal (por ejemplo, tendencia y/o estacionalidad).

Podemos pensar en una serie sin patrón como **desestructurada**, ya que no existe una estructura discernible dependiente del tiempo.

Alternativamente, una serie de tiempo puede tener patrones obvios, como una **tendencia** o **ciclos estacionales estructurados**.

A menudo podemos simplificar el proceso de modelado identificando y eliminando las estructuras obvias de los datos, como una tendencia creciente o un ciclo repetido. Algunos métodos clásicos incluso le permiten especificar parámetros para manejar estas estructuras sistemáticas directamente.

7.1.4 Cantidad de variables utilizadas como características

Una serie temporal según la cantidad de variables medidas que serán utilizadas como entrada al modelo puede ser:

- **Univariada:** una única variable.
- **Multivariada:** múltiples variables.

Es importante clasificar nuestra serie temporal en alguna de estos dos ya que los modelos a aplicar difieren de forma considerable en complejidad (multivariadas).

7.1.5 Horizonte de pronóstico

El horizonte de pronóstico es el período de tiempo en el futuro para el cual se nos propusimos predecir. Estos generalmente varían desde horizontes de pronóstico a corto plazo (menos de tres meses) hasta horizontes a largo plazo (más de dos años).

Sin embargo eso es una cuestión relativa a como fueron medidos los datos de nuestro *dataset* (segundos, minutos, horas, días, etc). Dado que nuestro *dataset* está indexado según la marca de tiempo (*timestamp*) nosotros podemos decidir cuantos pasos hacía adelante vamos a realizar nuestra predicción.

- **Un paso:** requiere predecir el próximo paso de tiempo futuro.
- **Varios pasos:** requiere predecir más de un paso de tiempo futuro.

Cuantos más pasos de tiempo se proyecten en el futuro, más desafiante será el problema dada la naturaleza agravada de la incertidumbre en cada paso de tiempo previsto.

7.1.6 Estático vs Dinámico

Se refiere a la actualización del modelo para dar nuevos pronósticos.

Estático: El modelo de pronóstico se ajusta una vez y se usa para hacer predicciones.

Dinámica: El modelo de pronóstico se ajusta a los nuevos datos disponibles antes de cada predicción.

7.1.7 Uniformidad en el tiempo

- **Contiguo:** Las observaciones se hacen uniformes a lo largo del tiempo. Muchos problemas de series de tiempo tienen observaciones contiguas, como una observación cada hora, día, mes o año.
- **Discontiguo:** Las observaciones no son uniformes a lo largo del tiempo. La falta de uniformidad de las observaciones puede deberse a valores perdidos o corruptos. También puede ser una característica del problema cuando las observaciones solo están disponibles esporádicamente o en intervalos de tiempo cada vez más o menos espaciados.

En el caso de observaciones no uniformes, es posible que se requiera un formato de datos específico al ajustar algunos modelos para que las observaciones sean uniformes a lo largo del tiempo.

7.2 Modelos disponibles

Se realizó una investigación de los modelos de *forecasting* disponibles de forma gratuita y los resultados se reflejan en el Cuadro 1. Se realizará una breve síntesis de cada uno de ellos, sin embargo no se puede responder a la pregunta de cual es mejor ya que según el tipo de problema a abordar alguno tendrá mejor rendimiento que el otro.

Librería	Creador/es	Estrellas	Versión	Licencia	Framework ML
Prophet	Facebook	12000	0.7.1	MIT	<i>PyTorch</i>
Skttime	ATI	3400	0.5.1	BSD-3-Clause	-
GluonTS	AWSlabs	1700	0.6.4	Apache-2.0	<i>mxnet</i>
pmdarima	Alkaline-ml	780	1.8.0	MIT	-
arch	bashtage	630	4.15	Propia	-
DCRNN	liyaguang	600	?	MIT	<i>TensorFlow</i>
Skttime-dl	ATI	340	0.1.0	BSD-3-Clause	<i>Keras</i>

Cuadro 1: Comparativo de modelos de series temporales.

Se explorarán los 3 más populares de Github

7.2.1 Prophet

Prophet es un modelo desarrollado por *Facebook AI* para pronosticar datos de series de tiempo basado en un modelo aditivo donde las tendencias no lineales se ajustan a la estacionalidad anual, semanal y diaria, más los efectos de las vacaciones. [25]

- Funciona mejor con series de tiempo que tienen fuertes efectos estacionales y varias temporadas de datos históricos.
- Robusto ante los datos faltantes y los cambios de tendencia, y normalmente maneja bien los valores atípicos.
- Incluye muchas posibilidades para que los usuarios modifiquen y ajusten los pronósticos. Puede utilizar parámetros interpretables por humanos para mejorar su pronóstico agregando su conocimiento del dominio.
- Está pensado para series temporales bursátiles en primera instancia, pero es posible utilizarlo en otros campos.

7.2.2 GluonTS

Gluon Time Series (GluonTS) es el kit de herramientas desarrollado por *AWSlabs* para el modelado probabilístico de series de tiempo, que se centra en modelos basados en el aprendizaje profundo. Proporciona utilidades para cargar e iterar sobre conjuntos de datos de series de tiempo, modelos de última generación listos para ser entrenados y bloques de construcción para definir sus propios modelos y experimentar rápidamente con diferentes soluciones. [26]

7.2.3 sktime

Kit de herramientas desarrollado por *Alan Turing Institute*, que proporciona algoritmos de series temporales especializados y herramientas compatibles con *scikit-learn* para construir, ajustar y validar modelos de series de tiempo para múltiples problemas de aprendizaje, que incluyen: [27]

- **Pronóstico de series temporales:** predecir el mañana dado datos pasados.
- **Clasificación de series temporales:** dada una serie temporal, asignar una etiqueta,

No utiliza aprendizaje profundo, pero *ATI* está desarrollando una librería que si lo hace denominada **sktime-dl**. Se encuentra en fase de desarrollo actualmente.

7.2.4 Conclusión

Debido a la naturaleza de nuestra serie temporal (que es de tipo intermitente) no podemos ajustarnos a algunos de estos modelos ya que necesitan series temporales continuas.

8 Desarrollo

8.1 Breve introducción

Los datos recabados provienen de la línea de fabricación de vehículos utilitarios en una usina de automotores, en la cual se utiliza un sistema de diversas auditorias para identificar defectos.

8.1.1 Organización de la usina

La usina se compone de departamentos, talleres y unidades de trabajo en ese orden jerárquico. A continuación listaremos los departamentos:

- Calidad (CALI)
- Embutición (EMBU)
- Ingeniería (DLI)
- Logística (SQF)
- Montaje (MONT)
- Pintura (PINT)
- Soldadura (SOLD)
- Soldadura (DIVD)

La línea de fabricación consta de 4 departamentos que intervienen directamente en el ensamblaje del vehículo:

EMBU → SOLD → PINT → MONT

Los demás departamentos son una parte vital de la producción pero intervienen indirectamente.

Cabe destacar a su vez que cada departamento posee talleres, que a su vez poseen unidades de trabajo obteniendo una estructura jerárquica de tipo árbol.

8.1.2 Nomenclatura de defectos

Un defecto (DEF) en su esencia se compone de un elemento (ELE), un incidente (INC) y opcionalmente una localización (LOC).

DEF = ELE + INC + LOC

Tanto ELE como INC están definido inequívocamente por un código de 4 caracteres mientras que LOC puede variar, lo que lleva a que un defecto posea al menos 8 caracteres.

8.1.3 Tipo

A su vez los defectos están categorizados por familia o tipo de defectos:

- APR: aprietes.
- ASP: aspecto.
- DGRC: degradaciones.
- ELEC: eléctrico.
- ESTQ: estanqueidad.
- FCIO: funcionamiento.
- FLDS: fluidos.
- FTES: faltantes.
- GMTR: geometría.
- MOP: modo operatorio.
- NCON: no conforme.
- RDOS: ruidos.

8.1.4 Puntos de captaje

Los defectos son detectados en diversos puntos de captaje que pueden definir (o no) a qué departamento o taller pertenece el defecto con un doble objetivo. En primer lugar para que los vehículos que ya poseen el defecto detectado sean derivados a puntos de retoque para ser reparados. Y además analizar y ejecutar diversas estrategias con el fin atacar el problema raíz para así erradicar el defecto.

Los defectos detectados provienen de las siguientes fuentes de datos:

- **Defectos por unidad (DPU)**: los defectos se detectan en puntos de captaje a lo largo de toda la línea de producción.
- **Carrocería pintura-soldadura (CAPS)**: se realiza una auditoria al final de la línea de pintura sobre una muestra de carrocerías.

- **Plan estático-dinámico (PESD)**: una vez finalizado el proceso de fabricación del vehículo, se le realizan pruebas de estanqueidad, *test-drive* y otros con objeto de encontrar defectos que en línea no serían detectables.
- **SAVES**: Auditoria de 5 minutos de una muestra aleatoria de vehículos en línea final.

Sin embargo solo obtenemos datos a partir de los puntos de captaje de **SOLD** dado que en **EMBU** las partes aún no poseen el identificador único del vehículo.

8.2 Procesamiento de datos

Es vital realizar el procesamiento de los datos para alimentar el modelo, para ello se ha dividido la tarea en las etapas que se muestran en la figura 64.



Figura 64: Procesamiento de datos segmentado en etapas.

El entorno de trabajo será **Jupyter-Notebook** que soporta Python como lenguaje de programación y las librerías utilizadas serán:

- **Pandas**: manejo de tablas y datos.
- **Numpy**: operaciones matriciales.
- **Matplotlib, Seaborn**: visualización de datos.

8.2.1 Recolección de datos

Etapa de búsqueda y recolección de los datos que serán utilizados para el modelo. En nuestro caso se dividió en recolectar en la organización los datos de defectos y paradas de línea (se obtuvieron para los departamentos **SOLD**, **PINT** y **MONT**).

Los datos fueron obtenidos desde 4 auditorias diferentes que fueron presentadas en 8.1.4.

8.2.2 Limpieza de datos

Consiste en inspeccionar los datos obtenidos en búsqueda de posibles errores o datos que quizás es posible obtener pero por diversas razones se han perdido y debemos intentar recuperar.

Ambos casos sucedieron, por tanto se procedió a recuperar los datos cuando fue posible, y en los que no se procedió al descarte debido a que no eran útiles para la tarea.

8.2.3 Preprocesamiento de datos

En este punto es donde debemos ponernos finos a la hora de inspeccionar nuestro *dataset* para homogeneizar los datos, detectar incongruencias o datos sin sentido.

En primer lugar se realizó una homogeneización de los datos, *i.e.* quizás el mismo dato se puede estar refiriendo a lo mismo pero tienen *tags* diferentes, por tanto es necesario revisar los datos columna por columna. Esto sucedió debido a que provenir los datos de diversas fuentes cada una tenía una forma de tratarlos.

Luego fue necesario la detección de incongruencias, y ahí nos topamos con inconvenientes tales como departamentos inexistentes. Además todos los defectos que no hayan sido catalogados en el campo GVD pasaron a ser V2.

Debido a la estructura jerárquica de la usina, si tenemos definido la UET a la que pertenece el defecto, escalando hacia arriba podemos obtener el taller y el departamento correspondiente. Esta operación se realizó para todo el *dataset*.

Finalmente tenemos nuestro *dataset* en condiciones para su análisis.

8.2.4 Análisis y visualización de datos

Haremos una parada en nuestro procesamiento de datos para visualizarlos y dimensionar en qué posición nos encontramos. Nuestro *dataset* se compone de 229934 filas que cada una representa un defecto y 17 columnas.

Algunas estadísticas a *grosso modo* que podemos sacar del mismo son:

- 10780 vehículos fueron fabricados durante 2020.
- 5224 defectos diferentes fueron registrados en el sistema.
- Se registran en promedio 19,6 defectos por vehículo.

Además si verificamos de dónde provienen nuestros datos caeremos en cuenta que casi la totalidad de ellos provienen de DPU, disputándose el pequeño margen restante PESD, CAPS y SAVES.

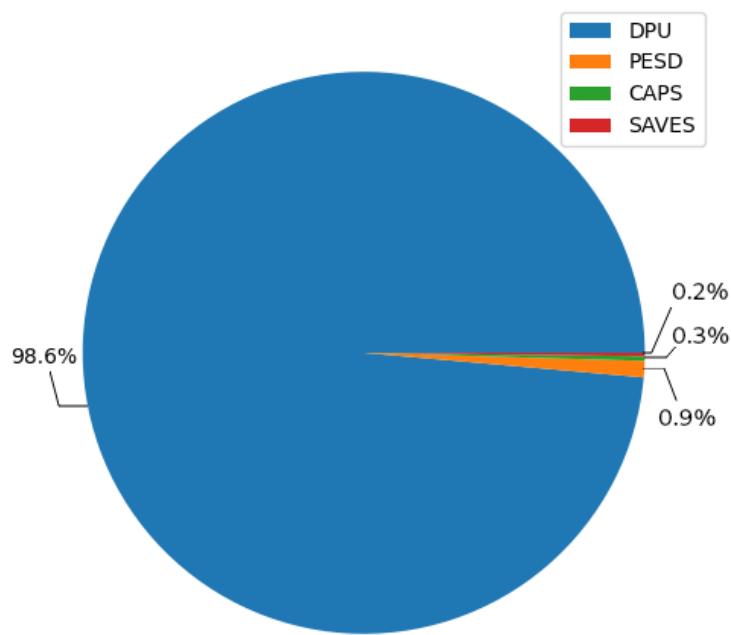


Figura 65: Distribución de los defectos según AUDI.

Sin embargo no debemos perder el foco que es predecir la cantidad de defectos graves que ocurrirán en función a todos los defectos anteriores. En la fig. 67 observamos que la gran mayoría de defectos registrados son V2 lo que podría ser útil para nuestro modelo.

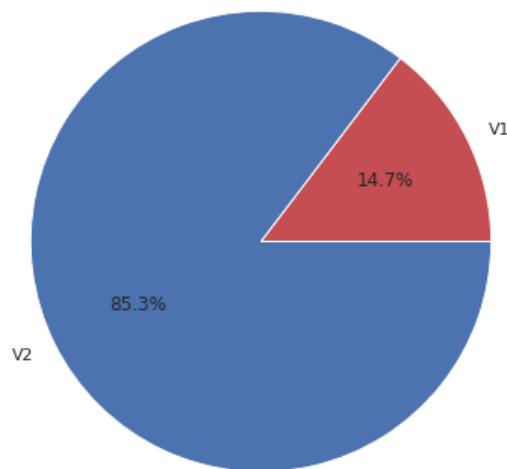


Figura 66: Distribución de los defectos según gravedad.

La distribución por departamento nos muestra que los defectos principalmente ocurren en 3: PINT, SOLD y MONT, un poco más atrás en la participación está SQF.

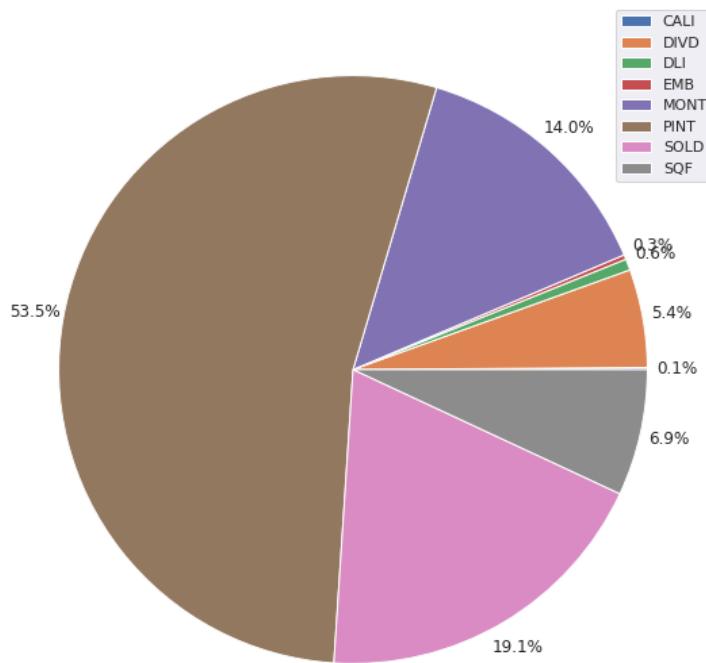


Figura 67: Distribución de los defectos por DPTO.

La distribución resume que casi 7 decimos de los defectos totales son de tipo ASP, seguido muy por detrás por DGRC y en tercer lugar los de MOP.

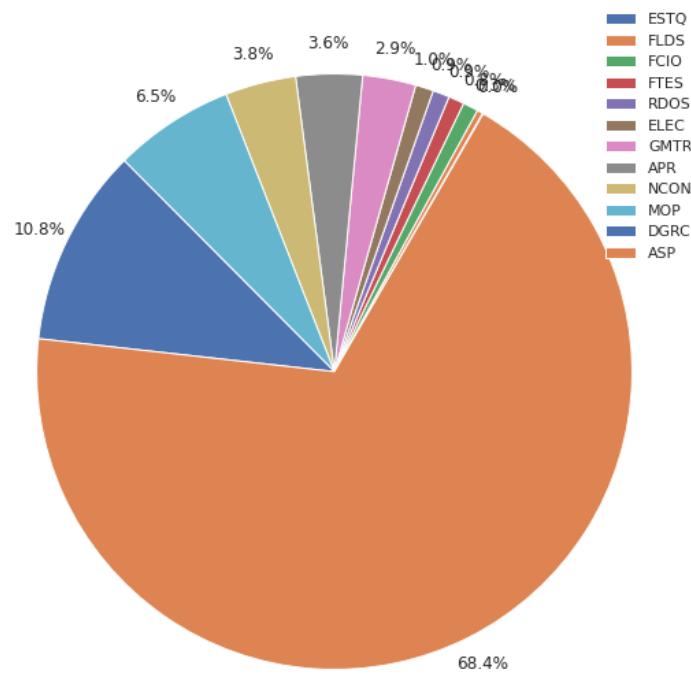


Figura 68: Distribución de los defectos por TIPO.

A través de un mapa de calor (fig. 69) intentaremos cruzar esta información para obtener de manera visual la correlación que existe entre los departamentos y los tipos de defectos. Notamos que existe una cierta correlación entre algunos tipos de defectos y los departamentos, a tal en punto que en algunos departamentos ni siquiera existen determinados tipos de defectos. Sería provechoso que nuestro modelo pueda aprender esta característica para maximizar sus posibilidades.

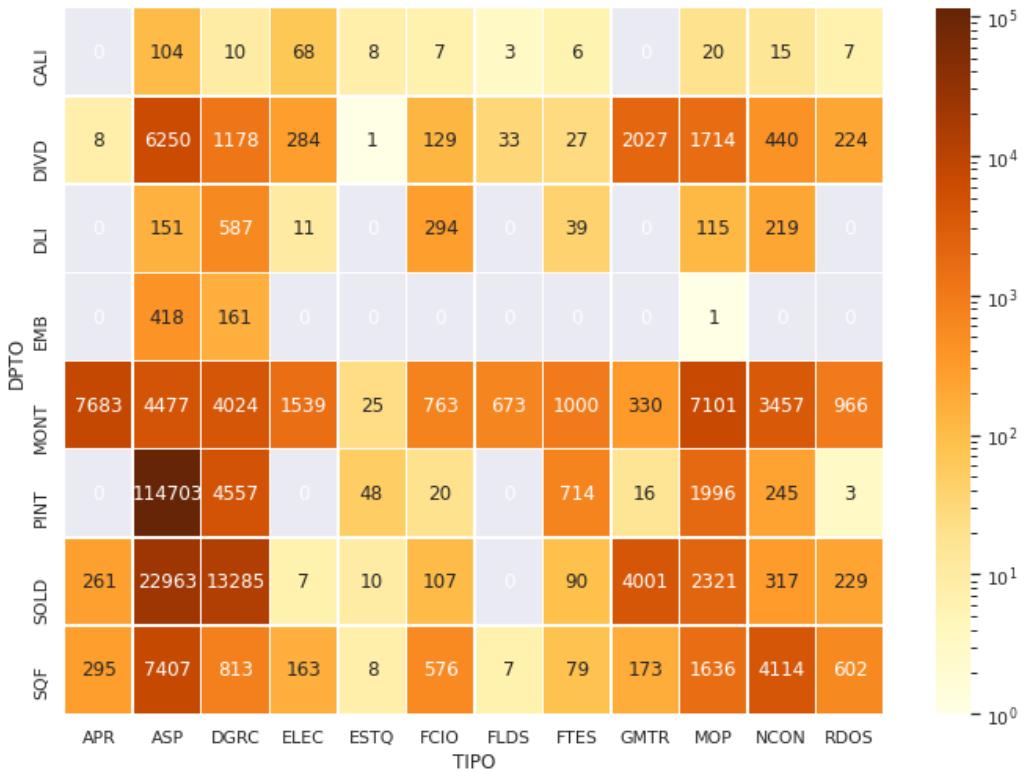


Figura 69: Cantidad de defectos cruzando DPTO y TIPO.

Sin embargo no hay que pensar que debido a que un determinado departamento y defecto posee muchos defectos la muchos son V1, la fig. 70 se encarga de ilustrar el concepto. Así dilucidamos que hay determinadas combinaciones de TIPO y DPTO que son más probable que sean V1.

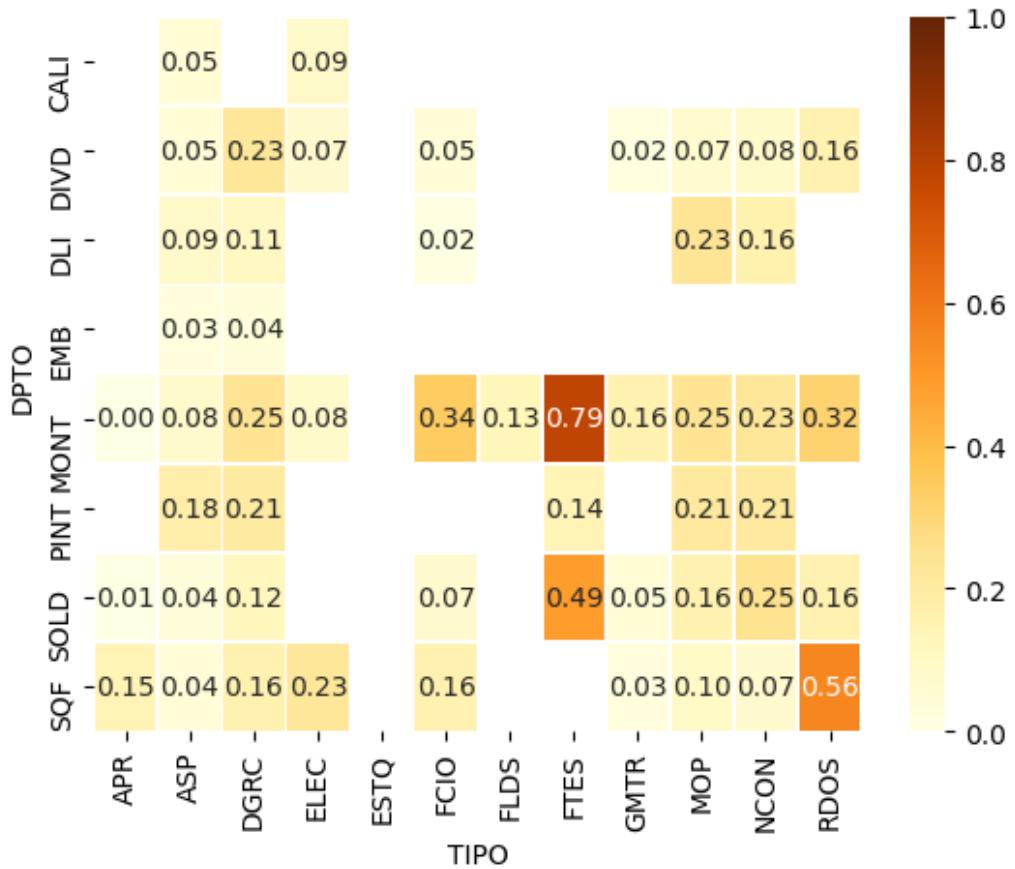


Figura 70: Fracción de V1 por cantidad de defectos.

Sin embargo el análisis más importante de manera visual que podemos realizar es el gráfico temporal de la cantidad de defectos según su gravedad como se muestra en la figura 71.

El 19 de marzo de 2020 se decretó la cuarentena debido a la pandemia provocada por el virus *SARS-CoV-2*, por tanto no se fabricaron vehículos. Los defectos registrados durante esa fecha y el reinicio de la producción en junio se deben a la recuperación de vehículos por retoques. Por esto para que el flujo de los datos hacia el modelo sea continuo solo se tomará desde junio a diciembre (6 meses).

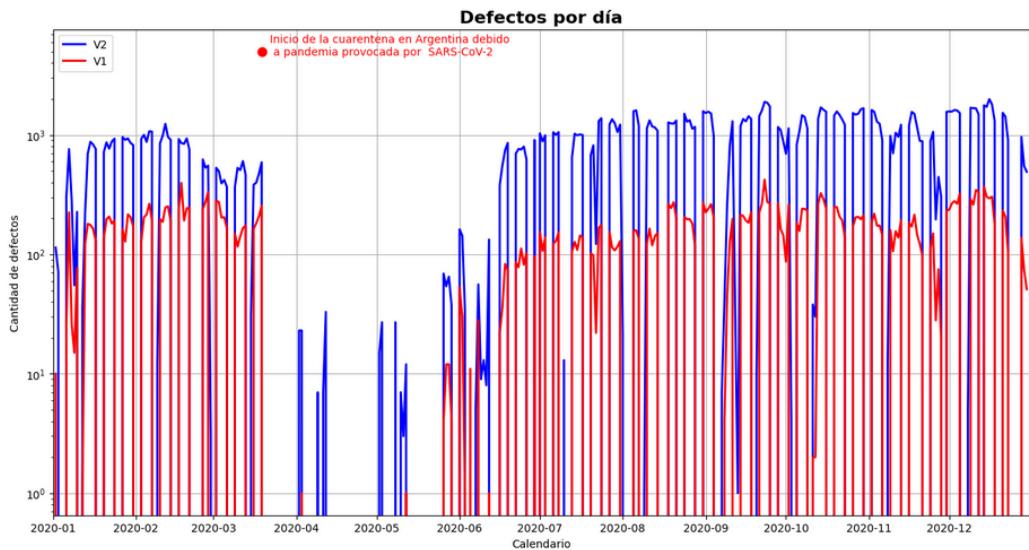


Figura 71: Defectos diarios agrupados por GVD.

Sin embargo, aún no podemos notar un patrón específico debido al ruido, por tanto aumentaremos la ventana de tiempo a una semana (fig. 72). Afortunadamente para nuestro modelo (aunque la escala del eje de ordenadas es logarítmico) la cantidad de defectos V2 sigue la curva de V1 de forma muy similar. Esto es positivo ya que podemos intuir que existe una correlación entre los mismos del cual nuestro modelo podrá obtener provecho.

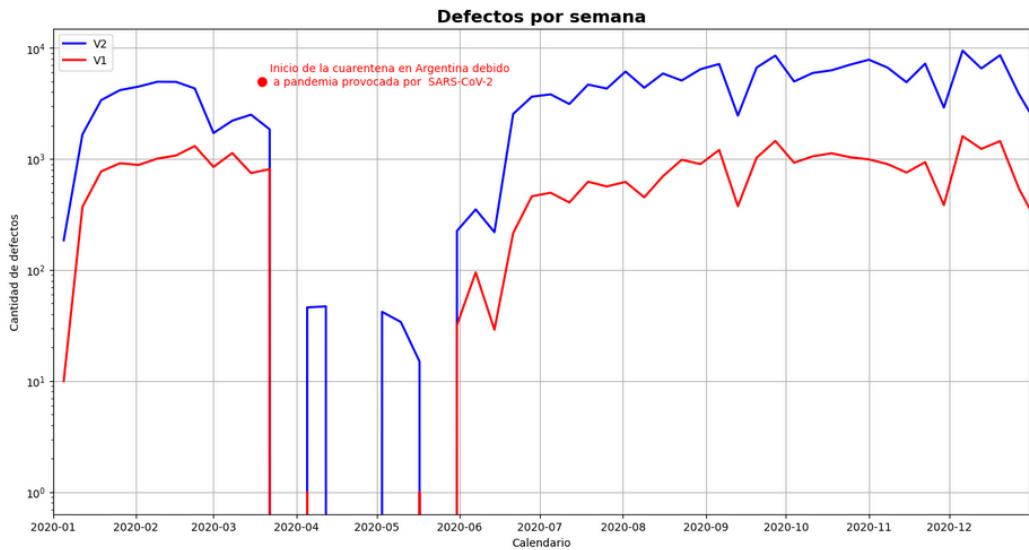


Figura 72: Defectos semanales agrupados por GVD.

Si al gráfico anterior lo desglosamos por auditoria y gravedad (fig. 73) veremos que en las demás auditorias que no sean DPU no hay patrones claros definidos, quizás se deba a la baja densidad de datos.

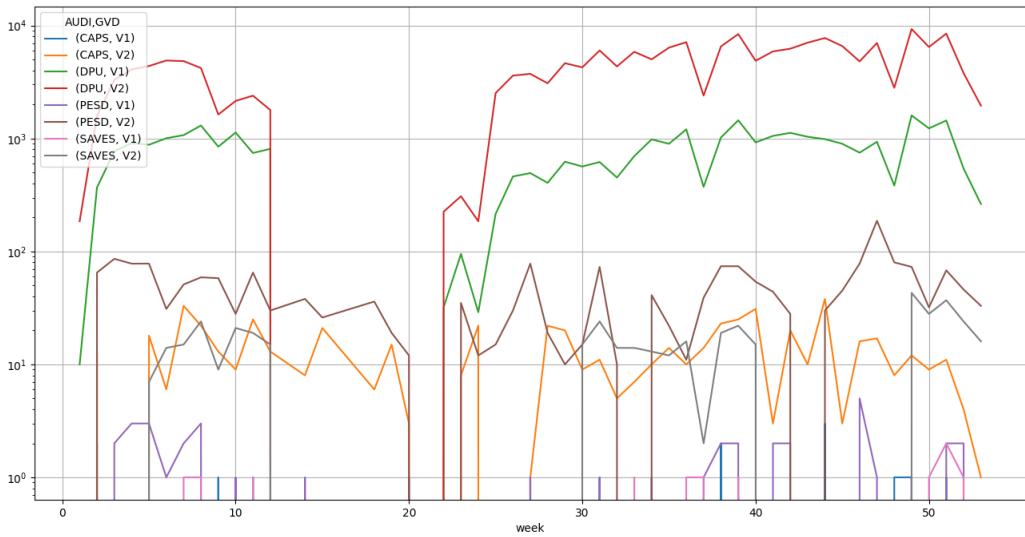


Figura 73: Defectos semanales agrupados por AUDI/GVD.

8.2.5 Imputación de datos

En estadística, la imputación es el proceso de reemplazar los datos faltantes (*missing values*) con valores sustituidos. En nuestro *dataset* tenemos 2 *features* con faltantes, una es UET (lo que conlleva a faltantes de DPTO y TALL) y la otra es TIPO.

Como se observa en la fig. 74, los datos faltantes no son demasiados, más precisamente 1052 de UET y 1228 de TIPO. Debido a la gran cantidad de datos disponibles podemos llenar estos valores con algún algoritmo de *Machine Learning* visto con anterioridad.

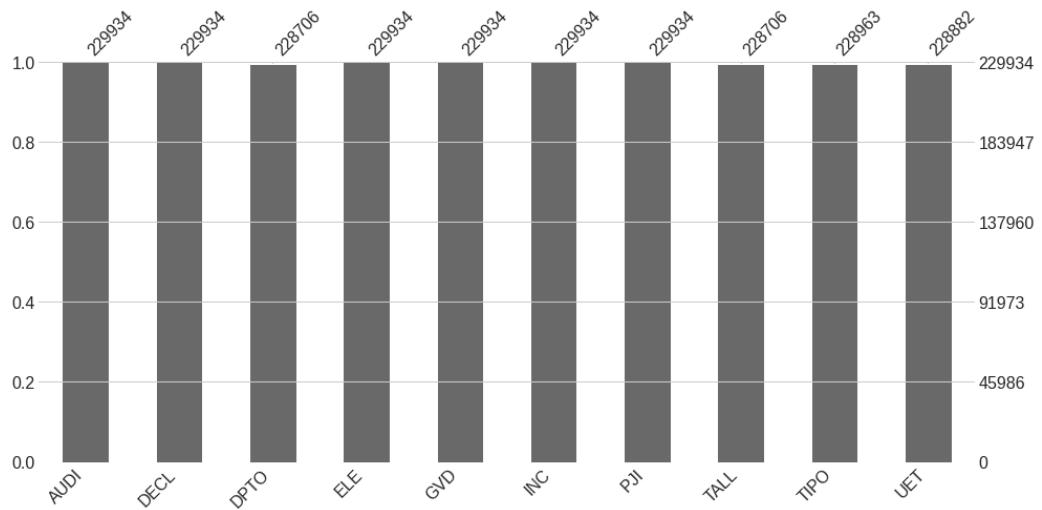


Figura 74: Datos faltantes por *feature*.

Luego de revisar los valores faltantes se optó por usar como estimador diversos tipos de clasificadores para así seleccionar el de mejor rendimiento, para posteriormente llenar los valores.

Los siguientes clasificadores fueron seleccionados:

- *KNeighborsClassifier* (KNC)
- *SGDClassifier* (SGDC)
- *RidgeClassifier* (RC)
- *LogisticRegression* (LR)
- *XGBClassifier* (XGBC)

- *DecisionTreeClassifier* (DTC)
- *RandomForestClassifier* (RFC)
- *BaggingClassifier* (BC)

Se optó por utilizar como *feature* ELE, INC, LOC y GVD, dejando como *target* TIPO.

Estimación de la precisión del modelo

Para estimar la precisión de los modelos seleccionados se utilizó *K-fold Cross-Validation* (KFCV) [28].

Una iteración de KFCV se realiza de la siguiente manera:

1. Se genera una permutación aleatoria del conjunto de muestra y se divide en K subconjuntos (pliegues o *folds*) de aproximadamente el mismo tamaño.
2. De esos K subconjuntos, un solo subconjunto se retiene como datos de validación para probar el modelo (este subconjunto se llama *testset*), y los restantes $K - 1$ subconjuntos juntos se utilizan como datos de entrenamiento (*trainset*).
3. Luego, se entrena un modelo en el *trainset* y se evalúa su precisión en el *testset*.
4. El entrenamiento y la evaluación del modelo se repiten K veces, y cada uno de los K subconjuntos se utiliza exactamente una vez como *trainset*.

En la fig. 75 se ejemplifica parte del proceso.

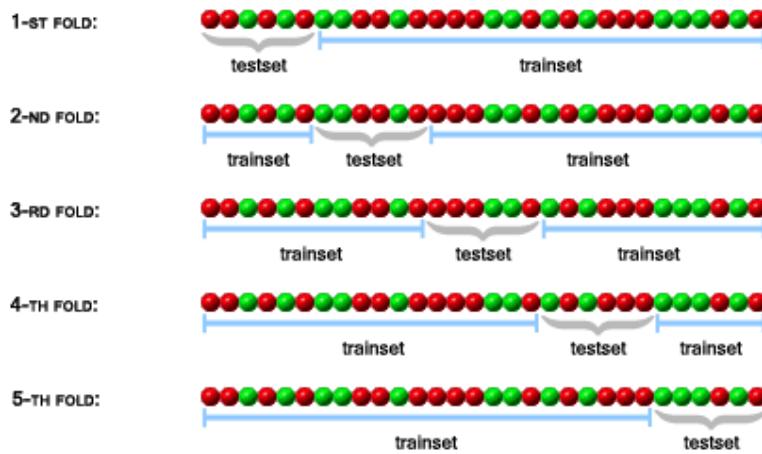


Figura 75: Ejemplo de validación cruzada de 5 veces con 30 muestras.

La estimación de precisión resultante depende de la permutación aleatoria que se generó al comienzo del proceso, ya que afecta la forma en que se partitiona el conjunto de muestras.

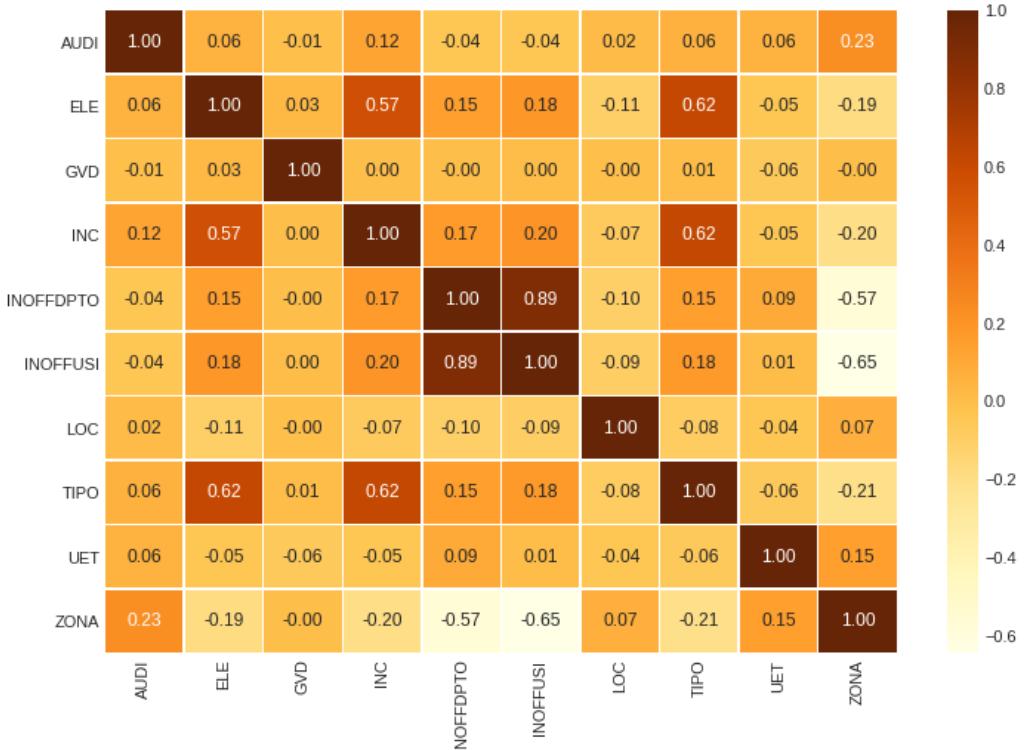
Por lo tanto, para obtener una estimación más exacta de la precisión, tiene sentido repetir la validación cruzada varias veces y tomar el promedio de todas las estimaciones de precisión obtenidas después de cada iteración como estimación de precisión resultante.

Matriz de correlación

En un *dataset* con muchos atributos, el conjunto de valores de correlación entre pares de sus atributos forma una matriz que se denomina matriz de correlación.

Existen varios métodos para calcular un valor de correlación. El más popular es el coeficiente de correlación de *Pearson*. Sin embargo, debe notarse que mide solo la relación lineal entre dos variables. En otras palabras, es posible que no pueda revelar una relación no lineal. El valor de la correlación de *Pearson* varía de -1 a $+1$, donde ± 1 describe una correlación positiva/-negativa perfecta y 0 significa que no hay correlación. [29]

La matriz de correlación es una matriz simétrica con todos los elementos diagonales iguales a $+1$. Nos gustaría enfatizar que una matriz de correlación solo brinda información sobre la correlación y NO es una herramienta confiable para estudiar la causalidad.

Figura 76: Matriz de correlación de nuestro *dataset*.

En la fig. 76 nos posaremos sobre las dos *features* que nos propusimos predecir.

- **TIPO:** en este caso notamos que las features ELE y INC, tienen una relación notable con TIPO, por tanto esas serán utilizadas y el resto descartadas.
- **UET:** aquí las relaciones están mucho más difusas, por tanto usaremos *features* que estén en las filas a las cuales queremos predecir su UET. Por tanto se usará ELE, INC, LOC, GVD y TIPO.

scikit-learn

Los algoritmos utilizados están implementados en la librería *sk-learn*, que nos permite a través de *pipelines* realizar un preprocessamiento de los datos para posteriormente entrenar al modelo. En nuestro caso debido a que todas las *features* son de tipo categoría usamos la codificación `one-hot encoder`, como observamos en la fig. 77.

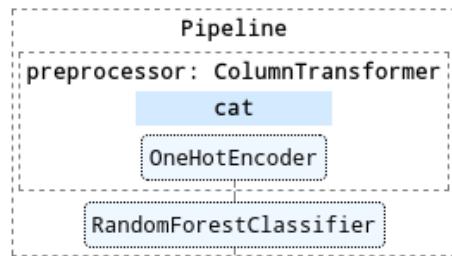


Figura 77: Gráfico de caja del rendimiento de cada modelo para TIP0.

Predicción de TIP0

El tiempo de ejecución se considera un parámetro importante debido a que si nuestro *dataset* creciera y poseemos el mismo poder de cómputo quizás deberíamos optar por entrenar un modelo que dé un buen rendimiento por poco tiempo de entrenamiento, *e.g.* DTC (fig. 78).

Modelo	Tiempo de ejecución	Rendimiento	Rend. por tiempo
KNC	0,37	99,42 %	2,71
SGDC	1,24	98,76 %	0,79
RC	3,08	97,99 %	0,32
LR	21,95	99,25 %	0,05
XGBC	23,20	96,72 %	0,04
DTC	1,53	99,72 %	0,65
RFC	20,05	99,73 %	0,05
BC	11,00	99,71 %	0,09

Cuadro 2: Ranking de clasificadores para TIP0

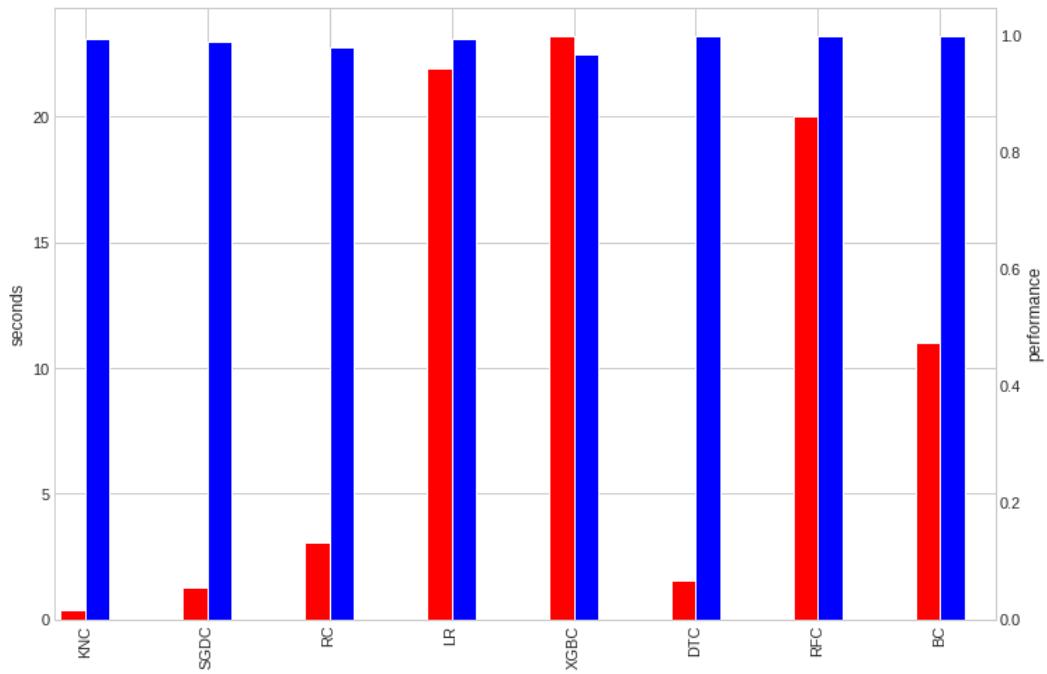


Figura 78: Gráfico de caja del rendimiento de cada modelo para TIPO.

Los resultados de rendimiento de cada modelo se presentan en el cuadro 2 y los interpretaremos de manera visual en la fig. 79. Hay una leve ventaja de `RandomForestClassifier` por sobre los demás modelos, así que será el seleccionado para imputar los datos faltantes de TIPO.

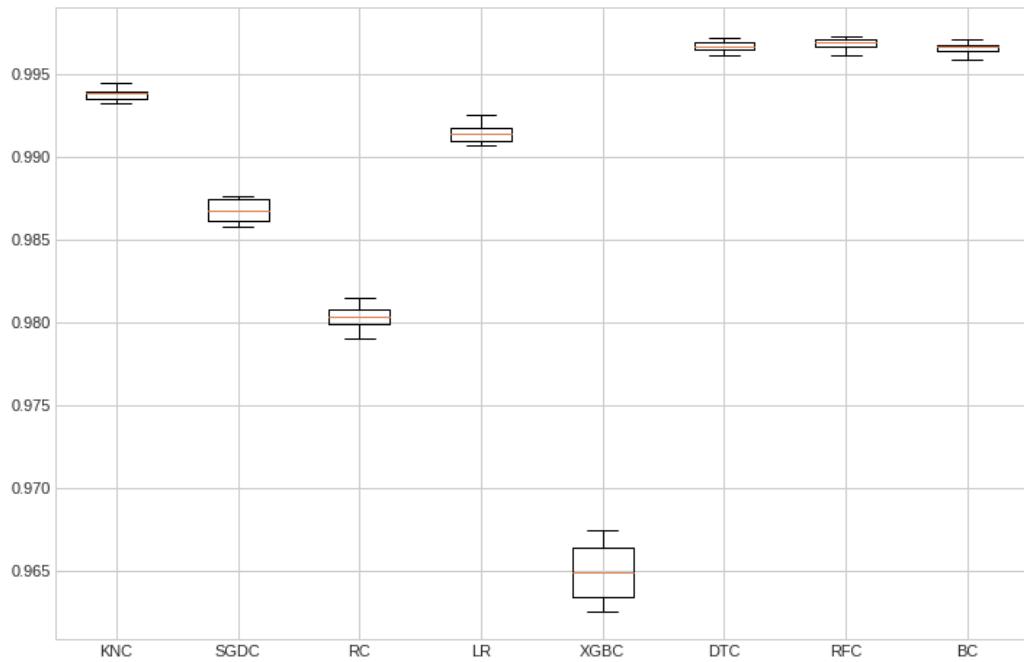


Figura 79: Gráfico de caja del rendimiento de cada modelo para TIPO.

Predicción de UET

Utilizando un *miniset* (*i.e.* una fracción de nuestro *dataset*) se fueron probando diferentes *features* para nuestro modelo (debido a que las pruebas requieren menor tiempo de cálculo y se supone que la muestra representa al menos la mayor parte del *dataset*). Tener en cuenta que para que este enfoque funcione es necesario setear el `random-state` de las funciones con una *seed* que es un número entero, esto es para quitar el componente de azar de los modelos y poder realizar las pruebas sobre los mismos datos.

Así llegamos a la conclusión que los mejores rendimientos se obtenían con ELE, INC y LOC, por tanto descartamos GVD y TIPO.

Una vez más DTC se lleva la mejor relación entre tiempo de entrenamiento y rendimiento como observamos en la fig. 80. El mejor rendimiento lo obtiene RFC (fig. 81).

Finalmente imputaremos los datos en nuestro dataset con el mejor clasificador para cada *feature*, así logrando el objetivo de completar los datos faltantes.

Modelo	Tiempo de ejecución	Rendimiento	Rendimiento por tiempo
KNC	0,42	76,45 %	1,80
SGDC	6,01	73,39 %	0,12
RC	19,45	68,54 %	0,04
LR	74,98	74,88 %	0,01
XGBC	193,92	73,52 %	0,00
DTC	2,29	79,60 %	0,35
RFC	157,46	79,74 %	0,01
BC	22,36	79,61 %	0,04

Cuadro 3: Ranking de clasificadores para UET

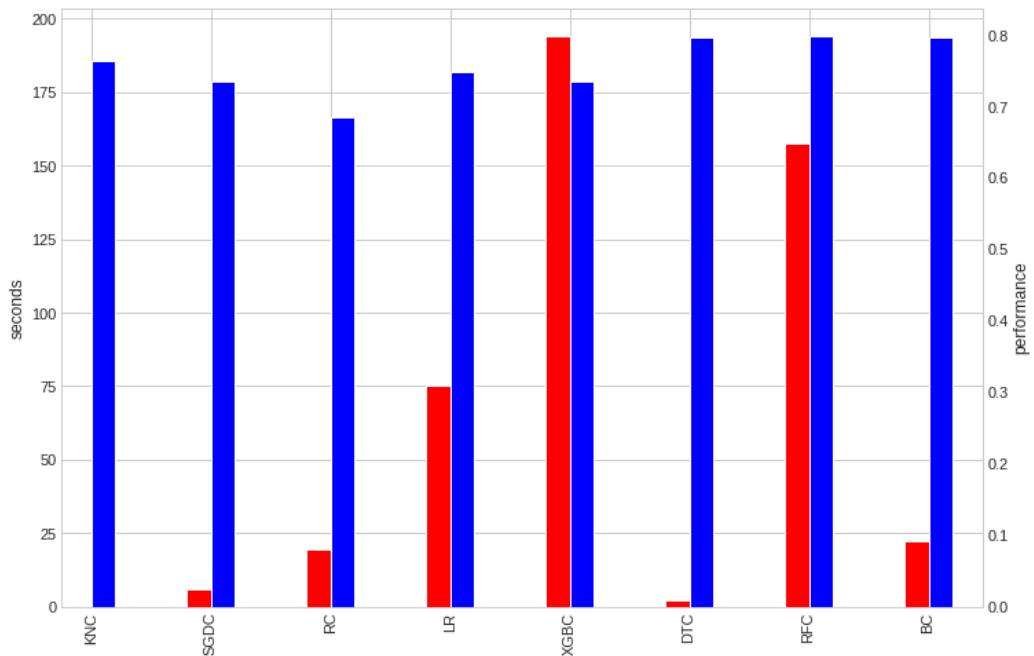


Figura 80: Gráfico de caja del rendimiento de cada modelo para TIPO.

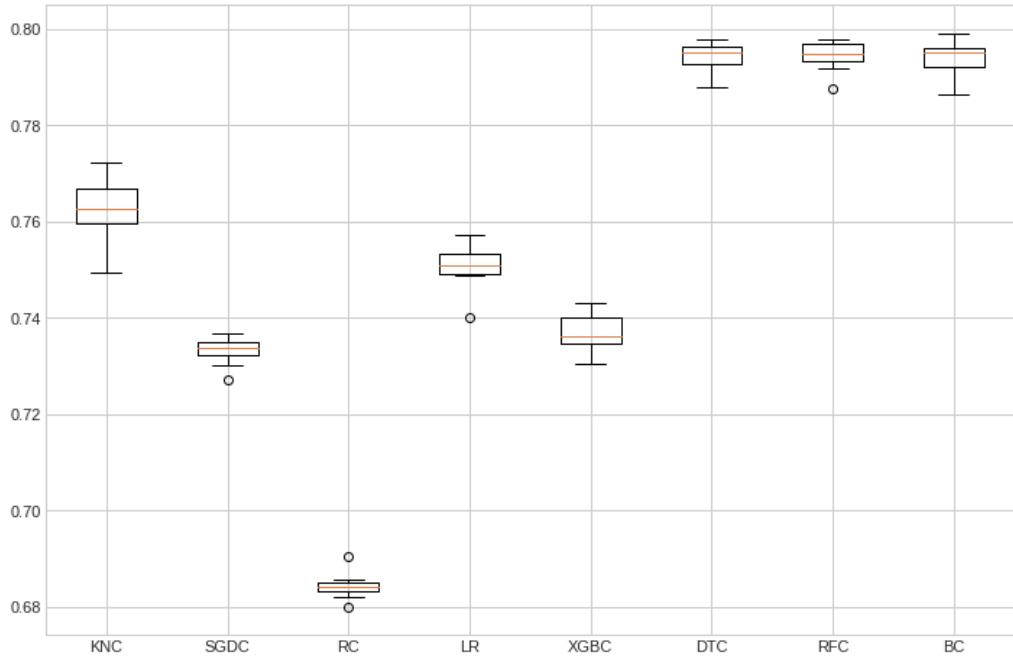


Figura 81: Gráfico de caja del rendimiento de cada modelo para TIPO.

8.2.6 Formateo de datos

El paso final antes de alimentar nuestro modelos es obtener rodajas de tiempo con funciones de agregación de tipo `sum`, `count`, etc de nuestros datos.

Serie temporal univariada

Pero como observamos en la Fig. 82, nuestra serie temporal dista mucho de ser continua. Esto es un inconveniente ya que se ha demostrado que a los modelos les cuesta mucho aprender sobre series temporales intermitentes.

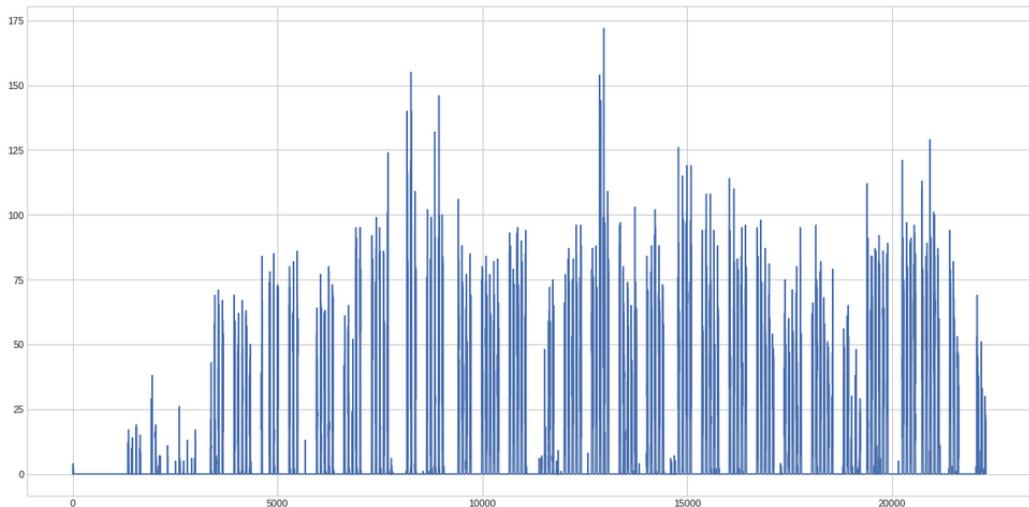


Figura 82: Gráfico temporal de cantidad de defectos.

Por tanto tomaremos las siguientes consideraciones:

- tomar el periodo posterior a la cuarentena por *Sars-CoV-2*.
- quitar los días sin o con baja producción.
- quitar horarios no laborales.

En la Fig. 83 donde aplicamos la función escalón, notamos los huecos que tenemos en nuestra serie, por tanto aplicaremos diversas funciones sobre la misma para obtener la serie continua deseada.

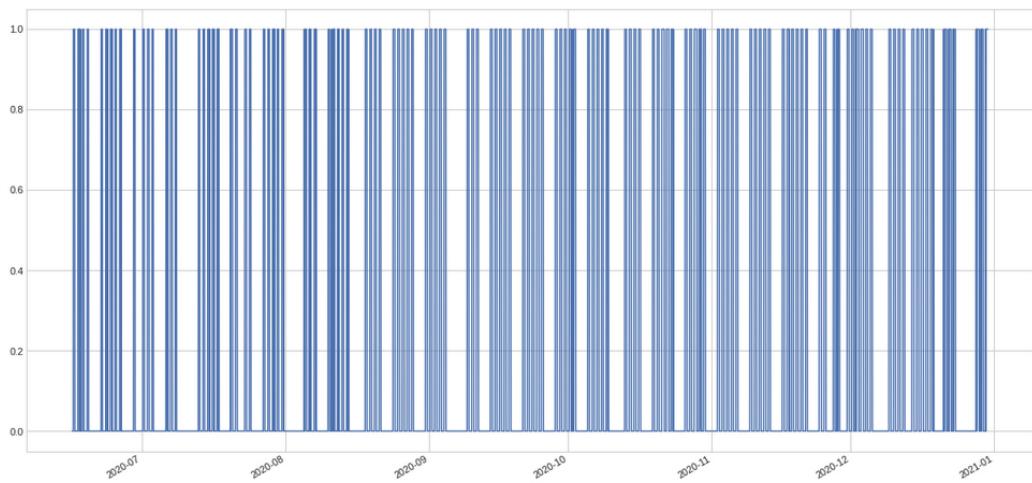


Figura 83: Gráfico temporal de detección de intermitencia.

Una vez aplicado los filtros necesarios obtenemos la siguiente serie de la Fig. 84. En la Fig. 85 si bien siguen apareciendo ceros son mucho menores y se originan en horarios de producción mínima como el horario de inicio y finalización.

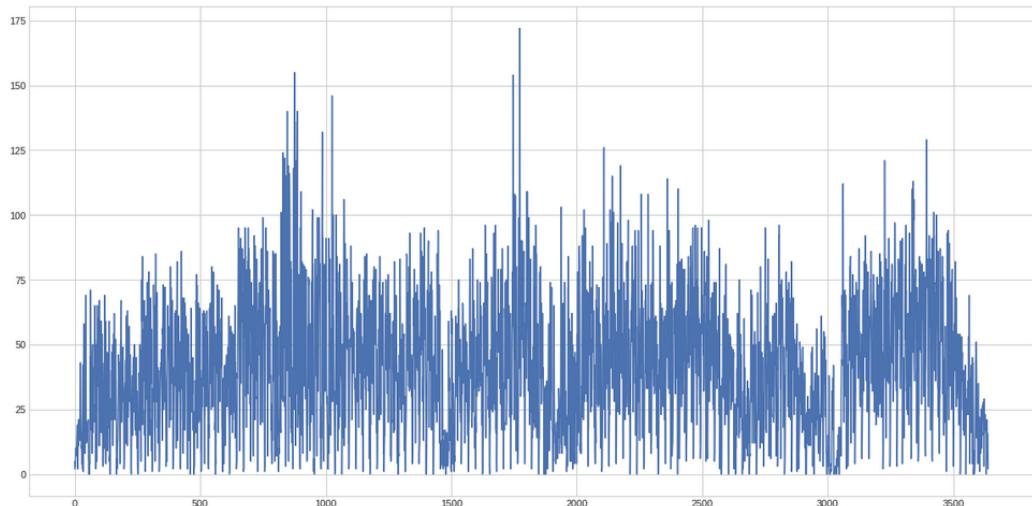


Figura 84: Gráfico temporal de cantidad de defectos procesado.

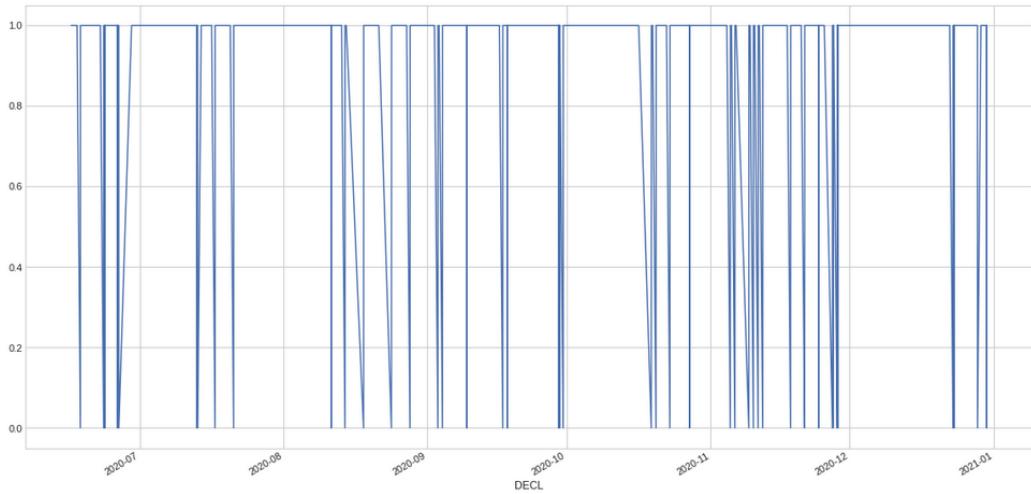


Figura 85: Gráfico temporal de detección de intermitencia procesado.

Serie temporal multivariada

El mismo análisis realizado para la serie univariada debe extrapolarse a la cantidad de dimensiones que consideremos necesarias.

En la fig. 86 observamos la cantidad de defectos que luego son desglosados por departamento, lo que nos daría 8 *features*. En la fig. 87 los defectos son desglosados por tipo, lo que nos daría 11 *features*. Lo que resultaría en 19 *features* que van a alimentar nuestro modelo.

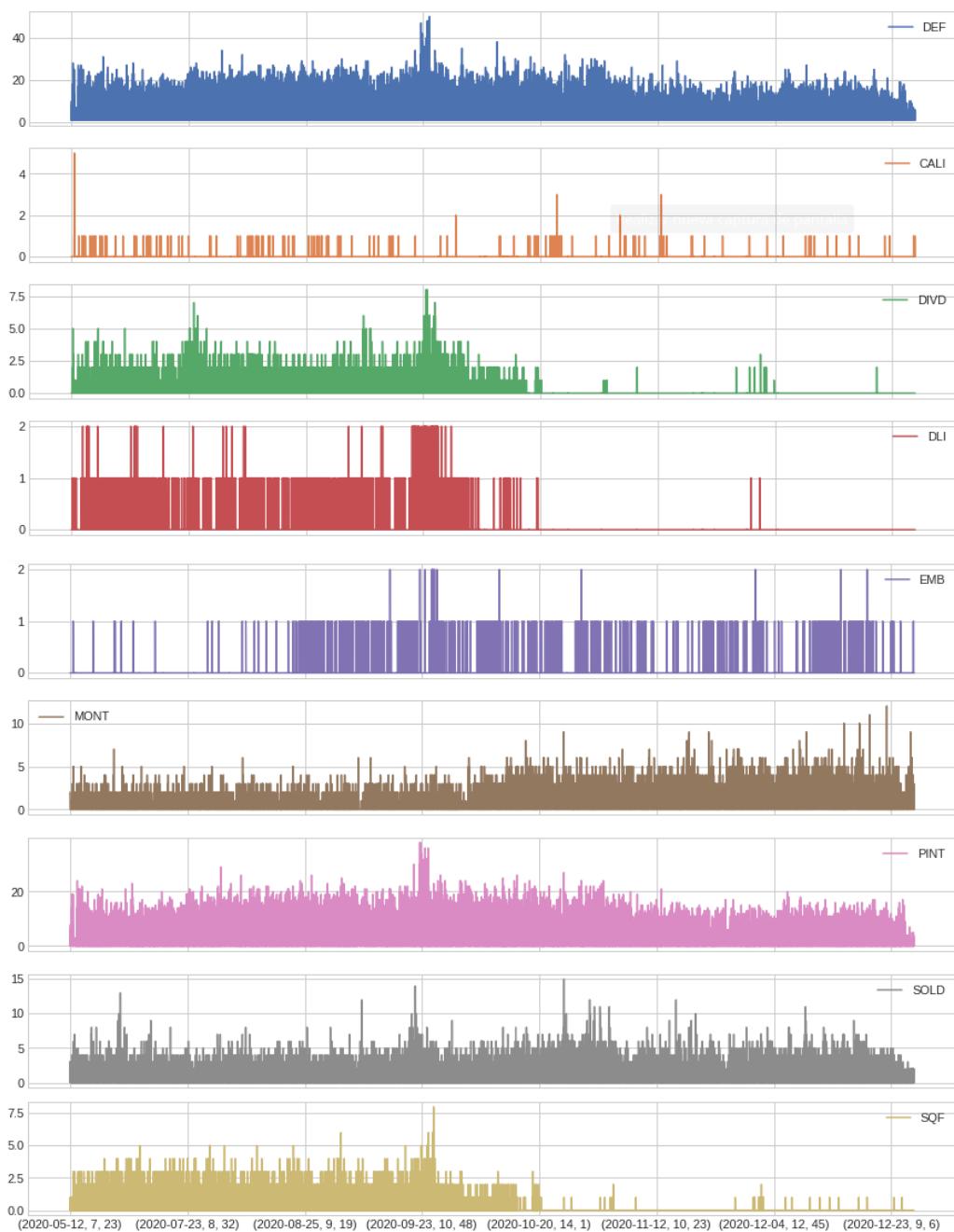


Figura 86: Gráfico temporal por departamentos.



Figura 87: Gráfico temporal por tipo de defecto.

Tal como se realizó en la serie temporal univariada, se aplican los mismos pasos a estas series temporales de lo cual resultan las figuras 88 y 89.

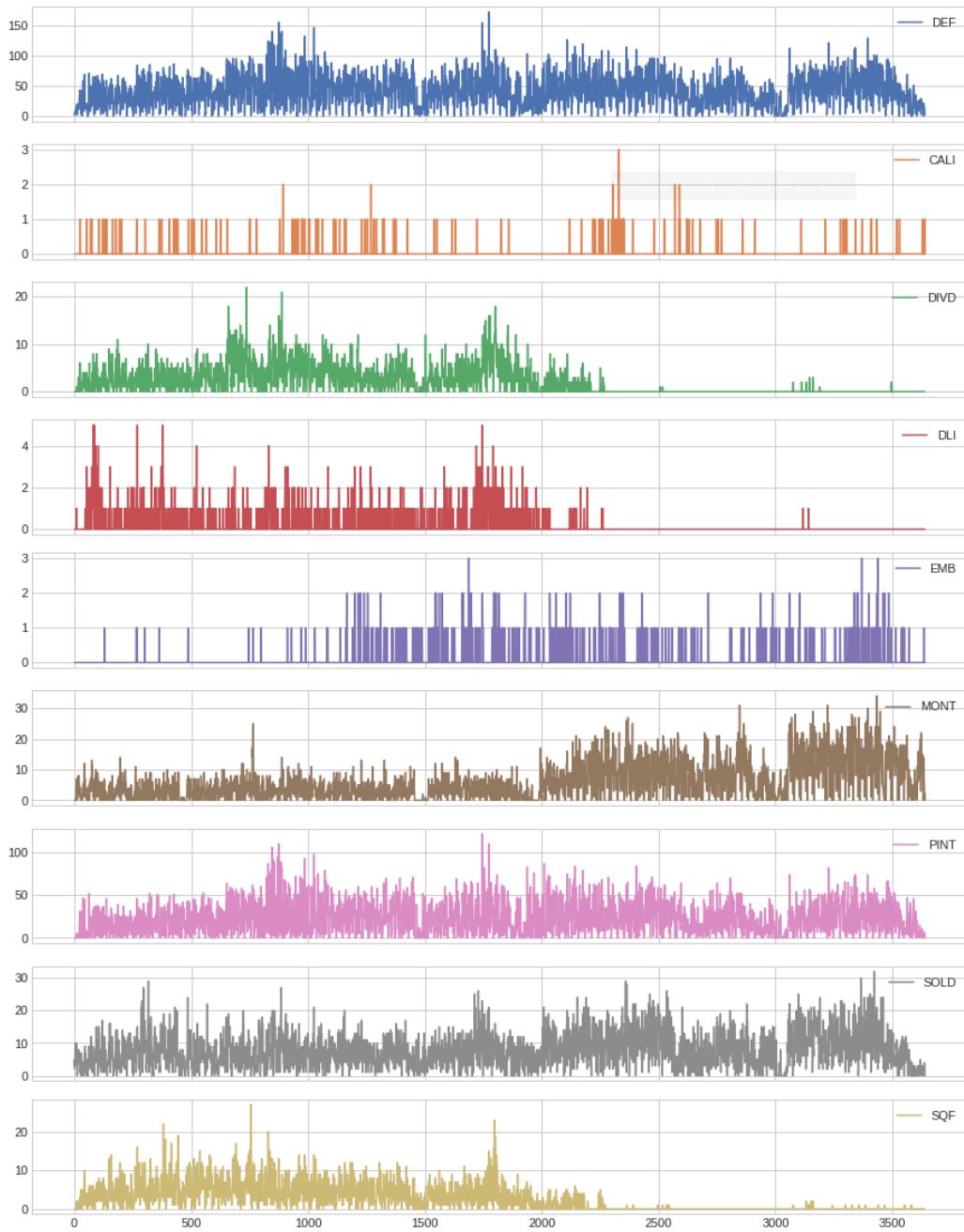


Figura 88: Gráfico temporal procesado por departamentos.



Figura 89: Gráfico temporal procesado por tipo de defecto.

Ya tenemos los *datasets* definitivos formateados de una manera correcta para alimentar nuestros modelos.

9 Modelos

Los modelos serán creados combinando las diferentes capas que se han visto con anterioridad, en dónde la tabla nos resumirá cada capa en una breve descripción además detallando los parámetros de cada una de ellas. Los parámetros de cada capa son importantes ya que en nuestra búsqueda del mejor modelo en la próxima sección implementaremos el *tuning* de hiperparámetros.

9.1 Métricas

Es importante definir métricas para determinar el rendimiento de nuestro modelo de una manera empírica, más allá de que nuestro modelos serán entrenados utilizando como métrica de pérdida MAE (error absoluto medio).

9.1.1 MAPE

Una de las métricas más comunes que se utilizan para medir la precisión del pronóstico de un modelo es **MAPE**, que significa error porcentual absoluto medio (*mean absolute percentage error*). [30]

La fórmula para calcular MAPE es la siguiente:

$$\text{MAPE} = \frac{100}{N} \times \sum_{i=1}^N \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

dónde:

- y_i : valores de los datos reales.
- \hat{y}_i : valor de datos pronosticados.
- N : cantidad de puntos.

MAPE se usa comúnmente porque es fácil de interpretar y de explicar. Por ejemplo, un valor MAPE de 11,5% significa que la diferencia promedio entre el valor pronosticado y el valor real es 11,5%. Cuanto menor sea el valor de MAPE, mejor podrá un modelo pronosticar valores.

Sin embargo esta métrica no nos será de utilidad ya que si alguno de los valores reales es 0 (como sucede) entonces el valor será incalculable por la división por 0.

9.1.2 MASE

En el pronóstico de series de tiempo, el **MASE** es el error medio absoluto escalado (*Mean Absolute Scaled Error*) es una medida para determinar la efectividad de los pronósticos generados a través de un algoritmo al comparar las predicciones con el resultado de un enfoque de pronóstico ingenuo o *naive forecast*. [31]

El *naive forecast* se genera en cualquier paso equiparando el pronóstico actual con la salida del último paso de tiempo. Por ejemplo, la predicción de las ventas de una empresa al comienzo de un mes se realiza comparándola con las ventas reales del último mes sin considerar ningún patrón estacional.

El **MAE** para *naive forecast* se calcula de la siguiente forma:

$$\text{MAE}_{\text{naive}} = \frac{1}{N-1} \sum_{i=2}^N |y_i - y_{i-1}|$$

Para determinar la efectividad de un algoritmo de pronóstico, el **MASE** se calcula de la siguiente manera:

1. Calcular el **MAE** (*Mean Absolute Error*) para los pronósticos del algoritmo.

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |\hat{y}_i - y_i|$$

2. MASE está dado por el ratio entre el **MAE** del algoritmo y el **MAE** del *naive forecast*.

$$\text{MASE} = \frac{\text{MAE}}{\text{MAE}_{\text{naive}}}$$

Características

- MASE da una indicación de la eficacia del algoritmo de pronóstico con respecto a un *naive forecast*. Su valor mayor que uno indica que el algoritmo está funcionando mal en comparación con el *naive forecast*.
- MASE es inmune al problema que enfrenta el Error de porcentaje absoluto medio MAPE cuando la salida de la serie de tiempo real en cualquier paso de tiempo es cero. En esta situación, MAPE da una salida infinita, que no es significativa. Sin embargo, se observa que para una serie de tiempo con todos los valores iguales a cero en todos los pasos, la salida de MASE tampoco se definirá, pero tales series de tiempo no son realistas.

- MASE es independiente de la escala del pronóstico, ya que se define utilizando la proporción de errores en el pronóstico. Esto significa que los valores de MASE serán similares si pronosticamos series de tiempo de alto valor, como el número de paquetes de tráfico de Internet que cruzan un router por hora, en comparación con el número de peatones que cruzan un semáforo ocupado cada hora.

9.2 *Framework*

Se utilizó *Keras*, una biblioteca de redes neuronales de código abierto escrita en *Python*. Es capaz de ejecutarse sobre *TensorFlow*, *Microsoft Cognitive Toolkit* o *Theano*. Su autor principal y mantenedor ha sido el ingeniero de François Chollet.

Está especialmente diseñada para posibilitar la experimentación en más o menos poco tiempo con redes de Aprendizaje Profundo. Sus fuertes se centran en ser amigable para el usuario, modular y extensible. [32]

9.2.1 **Callbacks**

Una característica muy interesante que nos ofrece *Keras* son las *callbacks*. Se define como un objeto que puede realizar acciones en varias etapas del entrenamiento (por ejemplo, al comienzo o al final de una época, antes o después de un solo lote, etc.). Repasaremos algunas de las que fueron utilizadas en el entrenamiento del modelo. [33]

- **EarlyStopping**: detiene el entrenamiento cuando una métrica monitoreada haya dejado de mejorar. Asumimos que el objetivo de un entrenamiento es minimizar la pérdida, *e.g.* la métrica a monitorear sería `loss` y el modo sería `min`. Un ciclo de entrenamiento verificará al final de cada época si la pérdida ya no está disminuyendo, considerando el `min_delta` y la paciencia (cantidad de épocas sin mejorar). Una vez que se encuentra que ya no disminuye, el entrenamiento termina. Esto es de utilidad para ahorrar tiempo de entrenamiento.
- **ReduceLROnPlateau**: reducir la tasa de aprendizaje cuando una métrica ha dejado de mejorar. Los modelos a menudo se benefician de la reducción de la tasa de aprendizaje en un factor de 2 a 10 una vez que el aprendizaje se estanca. Esta *callback* monitorea una métrica y si no se ve ninguna mejora en un número de épocas de "pacienza", la tasa de aprendizaje se reduce. Ha demostrado ser de gran utilidad.

- **ModelCheckpoint**: se usa junto con el entrenamiento para guardar un modelo o pesos (en un archivo de punto de control) en algún intervalo, por lo que el modelo o los pesos se pueden cargar más tarde para continuar el entrenamiento desde el estado guardado. Algunas opciones que ofrece esta devolución de llamada incluyen:
 - Ya sea para mantener solo el modelo que ha logrado el "mejor desempeño" hasta ahora, o si para guardar el modelo al final de cada época sin importar el desempeño.
 - La frecuencia a la que debería guardar. Actualmente, la devolución de llamada permite guardar al final de cada época o después de un número fijo de lotes de entrenamiento.
 - Si solo se guardan los pesos o se guarda todo el modelo.

9.3 Arquitecturas

Se define como la arquitectura del modelo a la disposición del mismo en lo que concierne a sus capas y al flujo de la información a través de las mismas.

En *Keras* es posible definir un modelo **Sequential**, lo que significa que cada salida alimentará a la siguiente y no habrá bifurcaciones entre el flujo de los tensores. También se podría resumir que **Sequential** agrupa una pila lineal de capas, lo que para nuestro propósito es suficiente. Esto sería una arquitectura de tipo lineal.

Utilizando *TensorFlow* es posible crear arquitecturas no-lineales que admitan múltiples flujos de datos entre las diferentes capas.

9.3.1 Hiperparámetros de capas

En la figura 90 podemos observar las capas disponibles para crear nuestra arquitectura.

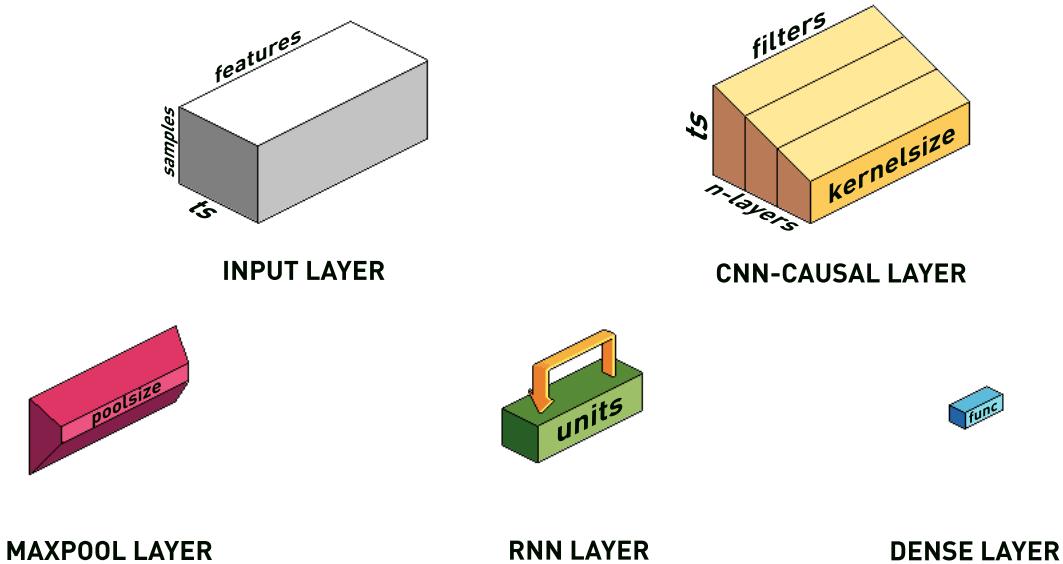


Figura 90: Tipos de capas disponibles.

Detallaremos los hiperparámetros que será posible *tunear* de cada capa.

- **INPUT:**
 - **ts**: *timestamps*.
 - **features**: características.
 - **samples**: cantidad de muestras.
- **CNN-CAUSAL**
 - **n-layers**: cantidad de capas, podemos escalar en la profundidad de CNNs.
 - **filters**: filtros.
 - **kernelsize**: tamaño del *kernel*.
 - **ts = timestamps** o rodajas de tiempo.
- **GRU y LSTM**
 - **units**: unidades de la celda.

- MAXPOOL:
 - `poolszie`: tamaño de *pool*.
- FC
 - `func`: función de activación.

9.3.2 Tipos

Por tanto en función a las capas explicadas en el marco teórico con anterioridad serán probados 5 arquitecturas de modelo representadas de forma visual en la fig. 91 para determinar cual es la mejor utilizando como métrica MASE con respecto a los datos de prueba.

- CNN: CNN → FC
- LSTM: LSTM → FC
- GRU: GRU → FC
- CNNLSTM: CNN → MAXPOOL → LSTM → FC
- CNNGRU: CNN → MAXPOOL → GRU → FC

En el caso de las últimas dos redes se utilizó la capa de MAXPOOL para evitar el cuello de la botella de los datos, corroborando que no se afectó el rendimiento del modelo y mejoraron los tiempos de entrenamiento.

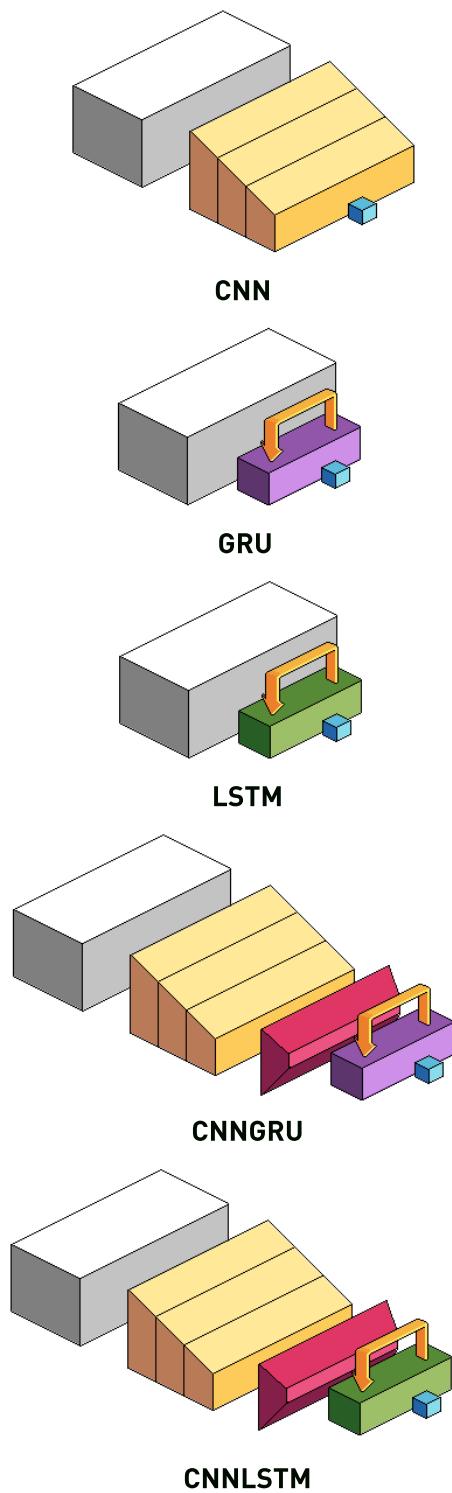


Figura 91: Tipos de arquitecturas.

9.4 Entrenamiento

Ya tenemos todos los ingredientes necesarios para empezar el entrenamiento de nuestra red neuronal, sin embargo sería coherente utilizar datos de prueba o *dummy data* para corroborar que todo funcione según lo esperado.

Usaremos un *dataset* de la calidad del aire, más precisamente uno que informa sobre el clima y el nivel de contaminación cada hora durante cinco años en la embajada de Estados Unidos en Beijing, China.

Los datos incluyen la fecha y hora, la contaminación denominada concentración de PM2.5 y la información meteorológica, incluido el punto de rocío, la temperatura, la presión, la dirección del viento, la velocidad del viento y la cantidad acumulada de horas de nieve y lluvia. La lista completa de características en los datos brutos es la siguiente:

- **pollution**: concentración de PM2.5, son partículas muy pequeñas suspendidas en el aire que tienen un diámetro de menos de 2.5 micras. Será nuestro *target*.
- **dew**: punto de rocío, es la más alta temperatura a la que empieza a condensarse el vapor de agua contenido en el aire, produciendo rocío, neblina, cualquier tipo de nube o, en caso de que la temperatura sea lo suficientemente baja, escarcha.
- **temp**: temperatura.
- **press**: presión.
- **wnd_dir**: dirección del viento combinada.
- **wnd_spd**: velocidad del viento acumulada.
- **snow**: horas acumuladas de nieve.
- **rain**: horas acumuladas de lluvia.

Podemos usar estos datos y enmarcar un problema de pronóstico en el que, dadas las condiciones climáticas y la contaminación de las horas anteriores, pronosticamos la contaminación para la próxima hora.

En la fig. 92 podemos observar que nuestros datos son bastante armónicos notando 3 ondas sinusoidales, lo que resultará en que nuestro modelo aprenda mejor (pero la mayoría de las series temporales en la vida real no serán tan convenientes).

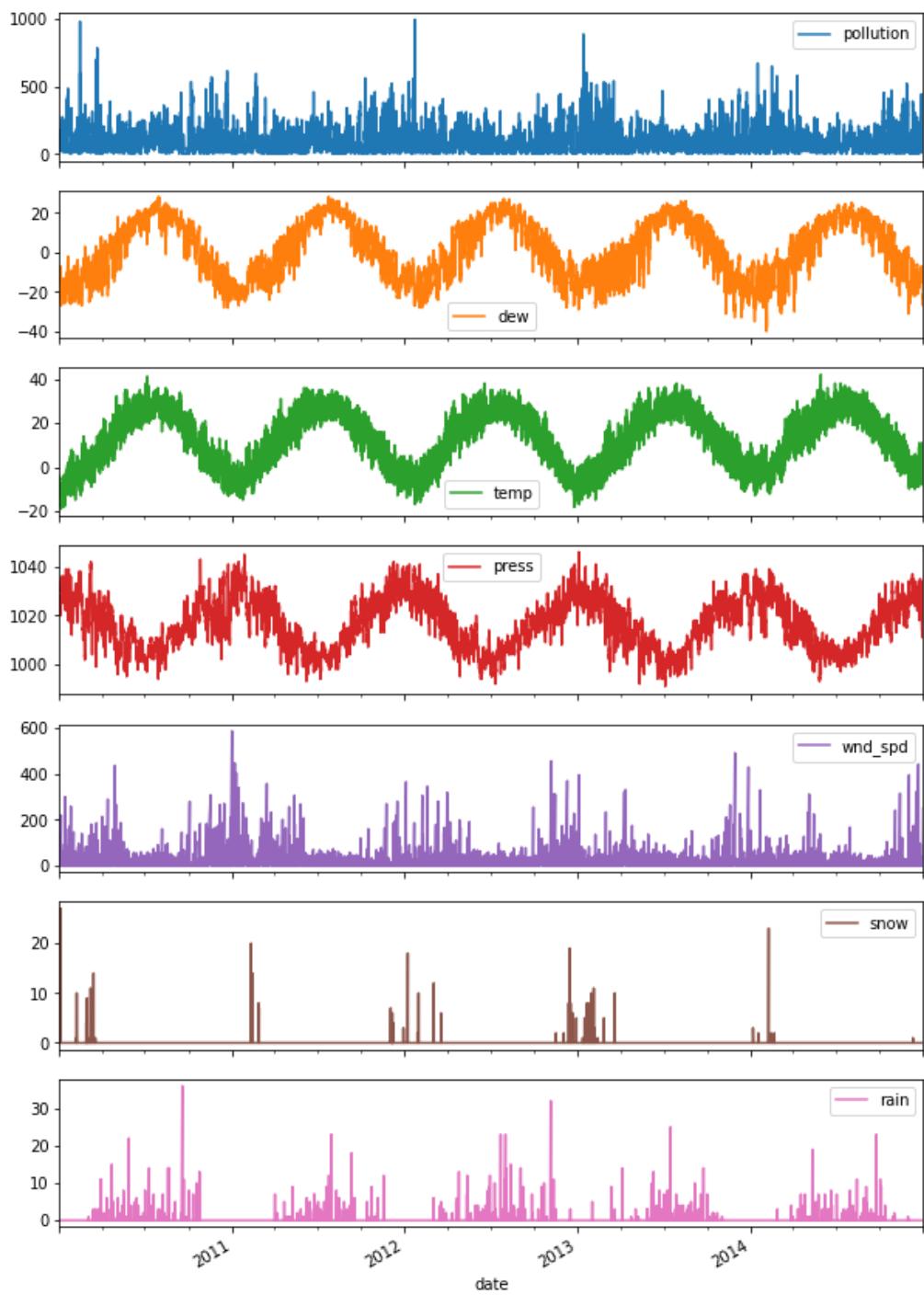


Figura 92: *Dataset* de contaminación del aire.

Luego deberemos dividir nuestro *dataset* como vimos con anterioridad en 3 sets en la siguiente proporción:

- **train** (*entrenamiento*): 70%.
- **validation** (*validación*): 20%.
- **test** (*prueba*): 10%.

Recordar que la métrica `val_loss` será aplicada sobre el set de validación y MASE sobre el set de prueba.

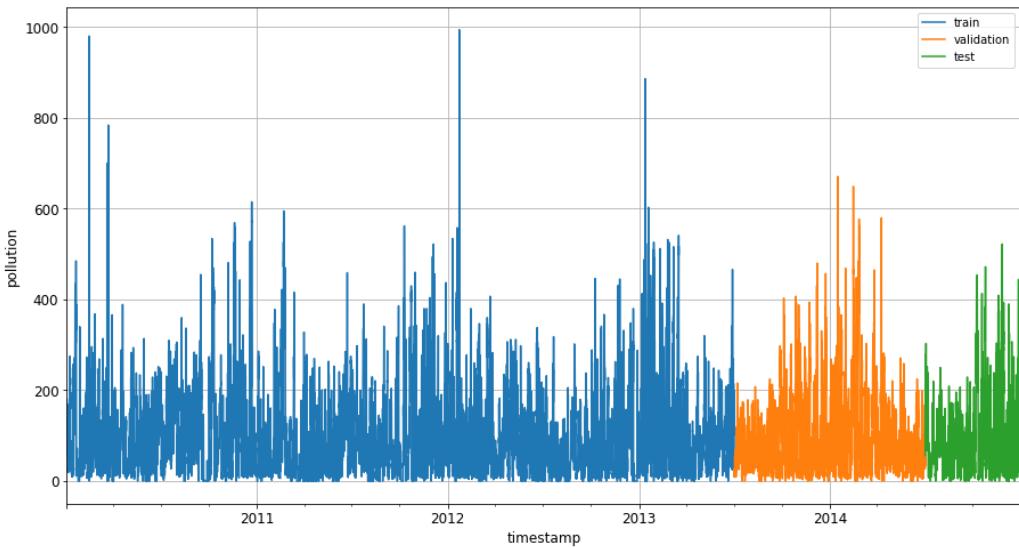
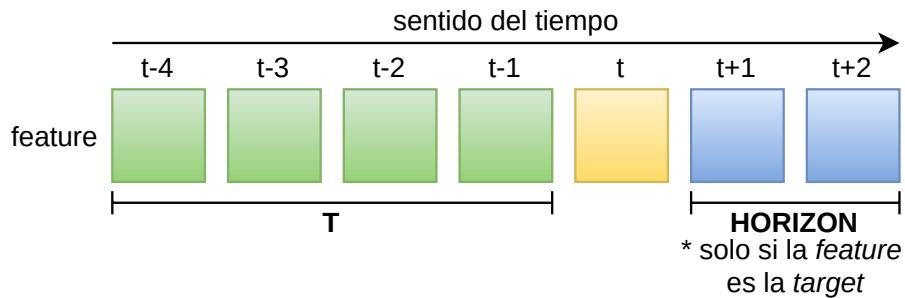


Figura 93: *Dataset* de contaminación del aire.

Una operación importante a realizar en nuestro dataset es el *shifteo* temporal para lograr que nuestra red pueda aprender sobre los *timesteps* pasados de nuestras features y comparar su salida contra los *timesteps* futuros en caso de ser la *target feature*.

Para aclarar esta idea pongamos nuestro enfoque en la figura 94. Definiremos T a la cantidad de *timesteps* que queremos que nuestro modelo utilice para extraer información pasada, además del actual.

Y definiremos HORIZON como al horizonte de pronóstico, *i.e.* la cantidad de *timesteps* futuros que queremos predecir de nuestra característica objetivo o *target feature*.

Figura 94: *Shifteo* temporal.

Un punto importante a aclarar es que es diferente la arquitectura a implementar según si nuestra *target feature* es de horizonte único o multi-horizonte.

Horizonte único

Veremos en una perspectiva más amplia el funcionamiento de nuestro modelo para tomar noción de donde nos encontramos y para esta tarea nos será útil la fig. 95. Básicamente tomamos los T *timesteps* de nuestras *features* para alimentar a la arquitectura (que fueron presentadas en la sección 9.3), en este caso compuesta de celdas RNN, que luego la salida alimenta una neurona que determina el valor de $t+1$.

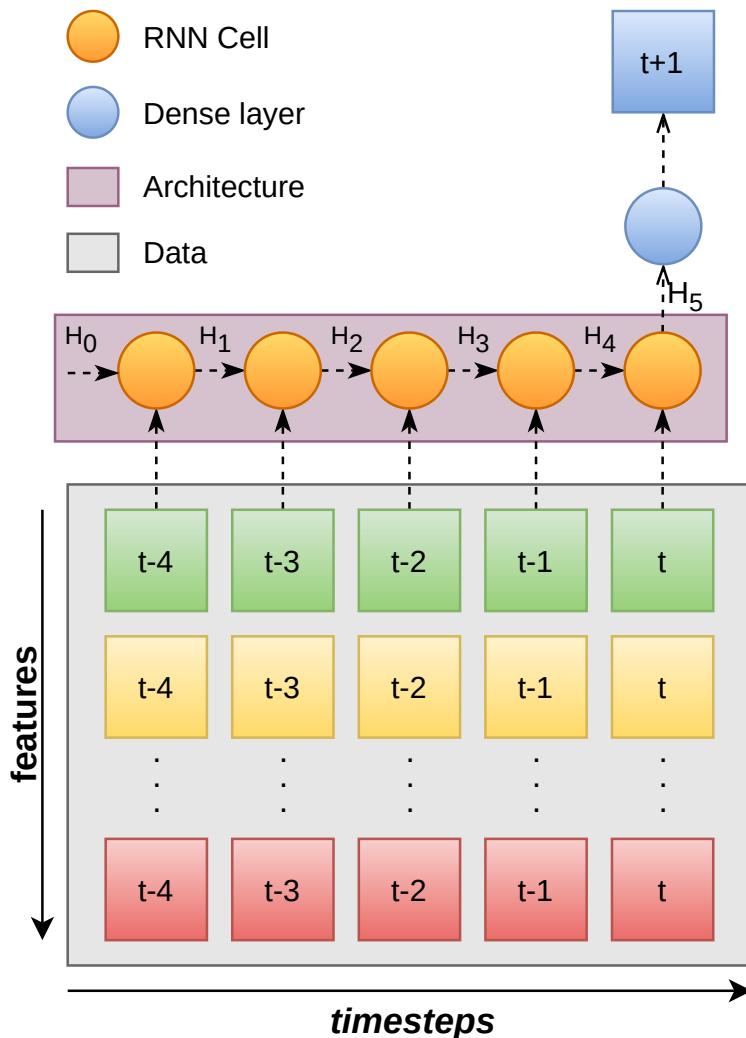


Figura 95: Modelo de horizonte único.

Horizonte múltiple

Observemos la fig. 96. Si determinamos que nuestra *target feature* será de horizonte múltiple, debemos aumentar la complejidad de nuestra arquitectura lo que se traduce en menor maniobrabilidad en lo que se refiere a adición de capas.

Debemos contar con dos capas **encoder** y **decoder**, donde una se encargará de procesar los datos de entrada y la otra de decodificar la salida de la primera para obtener la forma de tensor deseada a la salida de la red, lo que sería **t+1**, **t+2** y **t+3** en nuestro ejemplo.

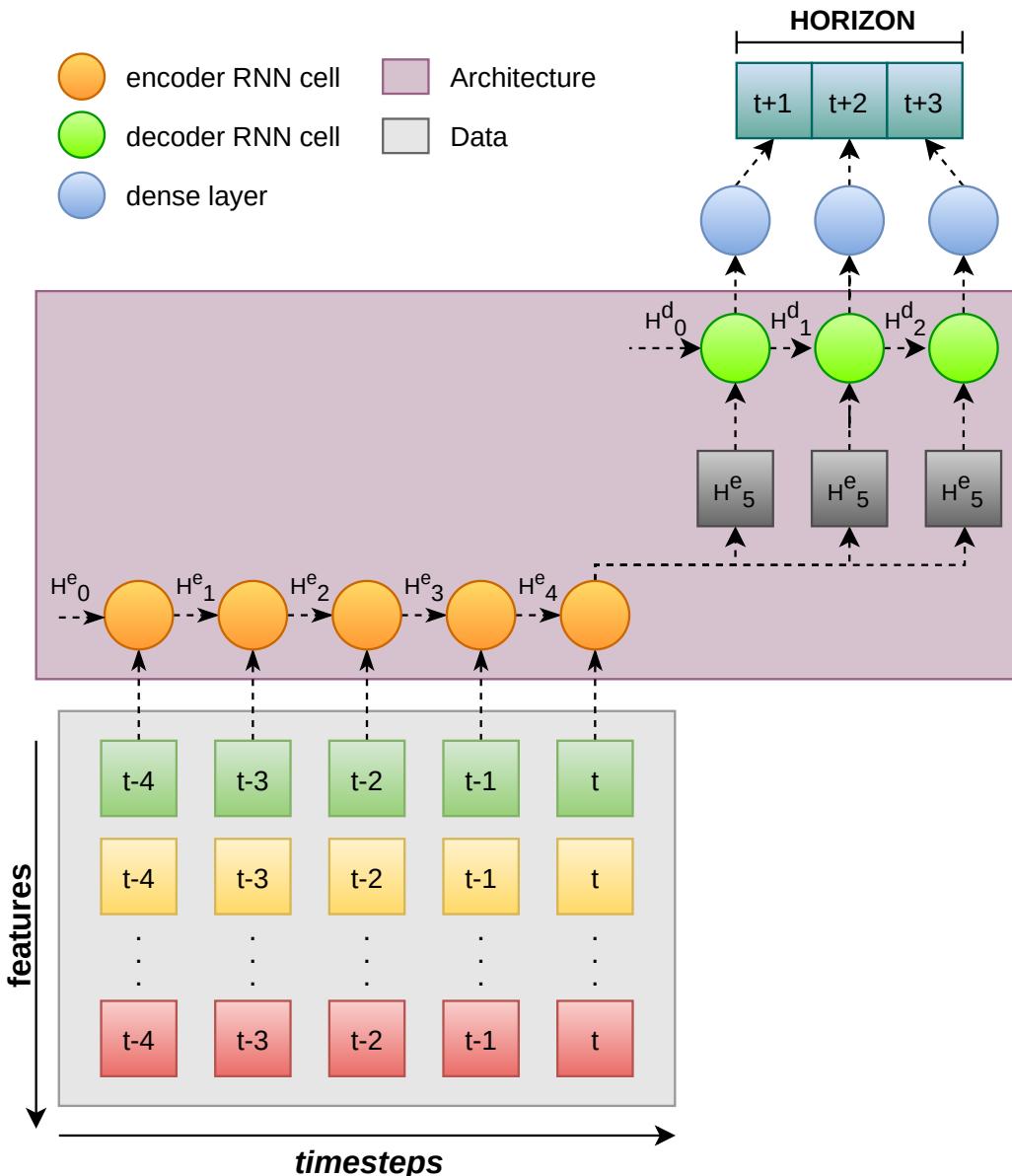


Figura 96: Modelo de horizonte múltiple.

En la fig. 97 podemos observar la salida de un pronóstico de horizonte múltiple con $T=16$ y $HORIZON=4$. Notamos que a medida que queremos predecir pasos más lejanos perdemos precisión, en la tabla 4 se corrobora de manera numérica esta observación.

Además cuando veamos los resultados de los modelos de horizonte único

notaremos que $t+1$ tiene mayor precisión que en este tipo de modelos.

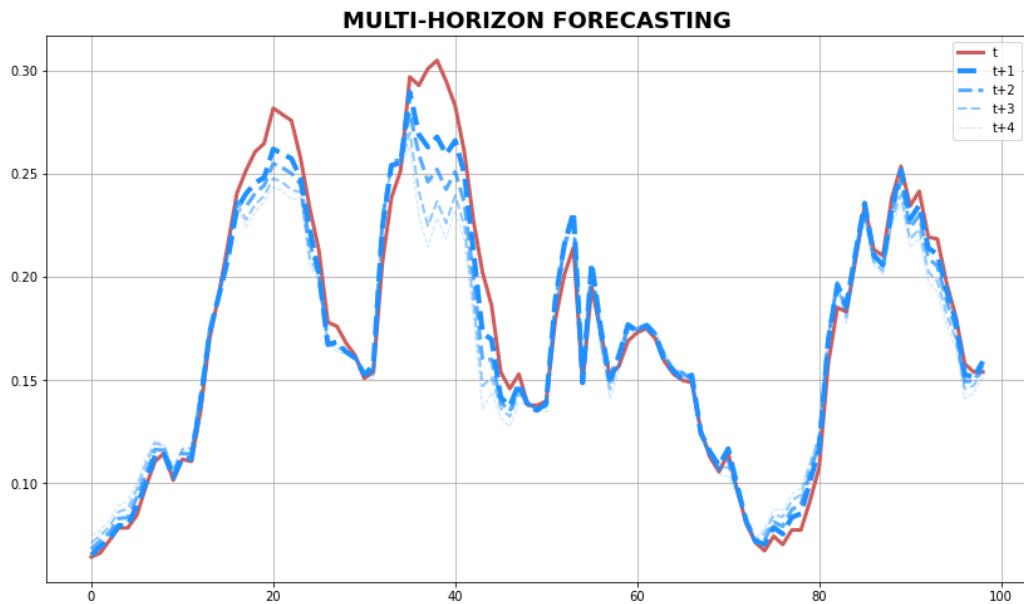


Figura 97: Pronóstico de horizonte múltiple.

t	MAE	MASE
t+1	0.00527	0.454
t+2	0.00728	0.628
t+3	0.00953	0.822
t+4	0.01167	1.006

Cuadro 4: Resultados del modelo de horizonte múltiple.

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*. Tektme, 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-based-image/>
- [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-3bd2b11e05d4>
- [16] 6.3. Padding and Stride — Dive into Deep Learning 0.16.0 documentation, Jan 2021. URL https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html. [Online; accessed 11. Jan. 2021].
- [17] Federico Peccia. Batch normalization: theory and how to use it with Tensorflow. *Medium*, Jan 2019. ISSN 1892-0173. URL <https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad>.
- [18] Deepmind. WaveNet: A Generative Model for Raw Audio, Sep 2016. URL <https://deepmind.com/blog/article/wavenet-generative-model-raw-audio>. [Online; accessed 11. Jan. 2021].
- [19] Joseph Eddy. Time Series Forecasting with Convolutional Neural Networks - a Look at WaveNet, Feb 2019. URL https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_conv. [Online; accessed 9. Jan. 2021].
- [20] 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].
- [21] Christopher Olah. Understanding LSTM Networks, Aug 2015. URL <https://colah.github.io/posts/2015-08-Understanding-LSTMs>. [Online; accessed 4. Jan. 2021].

- [22] 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.
- [23] 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>.
- [24] Producto de Hadamard (matrices) - Hadamard product (matrices) - qaz.wiki, Jan 2021. URL [https://es.qaz.wiki/wiki/Hadamard_product_\(matrices\)](https://es.qaz.wiki/wiki/Hadamard_product_(matrices)). [Online; accessed 4. Jan. 2021].
- [25] Prophet, Jan 2021. URL <https://facebook.github.io/prophet>. [Online; accessed 8. Feb. 2021].
- [26] GluonTS - Probabilistic Time Series Modeling — GluonTS documentation, May 2020. URL <https://ts.gluon.ai>. [Online; accessed 8. Feb. 2021].
- [27] alan-turing institute. sktime, Feb 2021. URL <https://github.com/alan-turing-institute/sktime>. [Online; accessed 8. Feb. 2021].
- [28] ProClassify User's Guide - Cross-Validation Explained, Jun 2006. URL <https://genome.tugraz.at/proclassify/help/pages/XV.html>. [Online; accessed 1. Feb. 2021].
- [29] 2 - How to Calculate a Correlation Matrix - Data Exploration for Machine Learning | Vertica, Nov 2020. URL <https://www.vertica.com/blog/in-database-machine-learning-2-calculate-a-correlation-matrix-a-data-explor>. [Online; accessed 4. Feb. 2021].
- [30] Info@numxl. com Spider Financial. MAPE - Error medio de porcentaje absoluto, Mar 2021. URL <https://support.numxl.com/hc/es/articles/215959443-MAPE-Error-medio-de-porcentaje-absoluto>. [Online; accessed 23. Mar. 2021].
- [31] Ashish Ahuja. Mean Absolute Scaled Error (MASE) in Forecasting - Ashish Ahuja - Medium. *Medium*, Jan 2021. URL <https://medium.com/@ashishdce/mean-absolute-scaled-error-mase-in-forecasting-8f3aec21968>.

- [32] Wikipedia. Keras - Wikipedia, la enciclopedia libre, Aug 2020. URL <https://es.wikipedia.org/w/index.php?title=Keras&oldid=128778152>. [Online; accessed 23. Mar. 2021].
- [33] Keras Team. Keras documentation: Callbacks API, Mar 2021. URL <https://keras.io/api/callbacks>. [Online; accessed 23. Mar. 2021].