

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
Matrícula: 39.022.782
Email: gerardo.collante@unc.edu.ar
Cel: 54 (03574) 650490

Supervisor
Dr. Ing. Orlando MICOLINI
26 de octubre de 2021

Agradecimientos

En primer lugar quiero agradecer profundamente a todas las personas que formaron parte de este viaje que culmina. Cada una de ellas aportó su granito de arena para que hoy me sea posible hacer realidad el objetivo más importante de mi vida. Especialmente a mis papás Daniel y Gloria, que ofreciendo su tiempo y esfuerzo me concedieron la dicha de poder formarme como profesional.

También quiero agradecer a mi supervisor Dr. Ing. Orlando Micolini por impartir conocimiento, brindar consejos y transmitir su experiencia a lo largo de todo el proceso del proyecto integrador.

Finalmente, el reconocimiento a la Universidad Nacional de Córdoba y la Facultad de Ciencias Exactas, Físicas y Naturales por estos invaluos años de saber y formación. La universidad pública me proveyó los recursos para que en el presente día me encuentre en condiciones de finalizar mi carrera de grado y graduarme como ingeniero, en virtud de ello mi gratitud eterna.

Índice general

1 Motivación	6
2 Objetivos	7
2.1 Objetivos principales	7
2.2 Objetivos secundarios	7
3 Clasificación de modelos de inteligencia artificial	8
3.1 Inteligencia Artificial	8
3.1.1 Aprendizaje automático	8
3.2 Preprocesamiento	10
3.3 Aprendizaje supervisado	10
3.3.1 Clasificación	11
3.3.2 Regresión	12
3.4 Aprendizaje no supervisado	15
3.4.1 Detección de anomalías	15
3.4.2 Reducción de dimensionalidad	16
3.4.3 <i>Clustering</i>	18
3.5 Selección de algoritmo en base al <i>dataset</i>	20
3.5.1 Algoritmos supervisados	21
3.5.2 Algoritmos no supervisados	23
4 Redes neuronales	25
4.1 Relación con la biología	26
4.2 Modelos artificiales	27
4.3 Funciones de activación	27
4.4 Arquitecturas de redes <i>feedforward</i>	30
4.5 Redes multicapa	31
4.6 Función pérdida	31
4.7 Descenso de gradiente	33
4.8 Backpropagation	35
4.9 Descenso de gradiente estocástico (SGD)	39
4.10 Sobreajuste y bajo-ajuste	40
4.11 Regularización	41
4.12 Los cuatro ingredientes de una red neuronal	41
4.12.1 Conjunto de datos	41
4.12.2 Función de pérdida	42
4.12.3 Modelo/Arquitectura	42

4.12.4	Método de optimización	42
5	Redes neuronales convolucionales	42
5.1	Convolución 1D	43
5.2	Convolución 2D	45
5.2.1	<i>Padding</i>	47
5.2.2	<i>Stride</i>	48
5.3	Tipos de capas	49
5.3.1	Convolución	49
5.3.2	Activación	52
5.3.3	<i>Fully-connected</i>	52
5.3.4	<i>Pooling</i>	52
5.3.5	<i>Batch Normalization</i>	53
5.3.6	<i>Dropout</i>	55
5.4	<i>WaveNet</i> y capas convolucionales causales dilatadas	55
6	Redes neuronales recurrentes	59
6.1	Arquitecturas	60
6.2	Funcionamiento	62
6.3	Entrenamiento	65
6.4	Desvanecimiento del gradiente	66
6.5	Tipos de <i>RNNs</i>	69
6.5.1	LSTM	69
6.5.2	GRU	74
6.6	Secuencias de entradas y salida	74
7	Optimización de hiperparámetros	77
7.1	Algoritmos	78
7.1.1	Búsqueda exhaustiva del espacio	78
7.1.2	Modelos subrogados	79
7.1.3	Reducciones sucesivas a la mitad	91
8	Series temporales	97
8.1	Clasificación del modelo	97
8.1.1	Tipo de variables de entrada	97
8.1.2	Objetivo	98
8.1.3	Estructura	99
8.1.4	Cantidad de variables utilizadas como características .	99
8.1.5	Horizonte de pronóstico	99
8.1.6	Estático <i>vs</i> Dinámico	100
8.1.7	Uniformidad en el tiempo	100
8.2	Modelos disponibles	101

8.2.1	Prophet	101
8.2.2	GluonTS	101
8.2.3	sktime	102
8.2.4	Conclusión	102
9	Entropía de una serie temporal	103
9.1	Introducción	103
9.2	Información y complejidad	103
9.2.1	Estadísticas de regularidad	103
9.2.2	Teoría de la información	105
9.3	Medición de la aleatoriedad	111
9.3.1	Entropía aproximada	111
9.3.2	Entropía de muestra	118
9.3.3	Diferencias entre ApEn y SampEn:	120
10	Desarrollo	120
10.1	Breve introducción	120
10.1.1	Organización de la usina	120
10.1.2	Nomenclatura de defectos	121
10.1.3	Gravedad	121
10.1.4	Tipo	122
10.1.5	Puntos de reconocimiento de defectos	122
10.2	Procesamiento de datos	123
10.2.1	Recolección de datos	123
10.2.2	Limpieza de datos	123
10.2.3	Preprocesamiento de datos	124
10.2.4	Análisis y visualización de datos	124
10.2.5	Imputación de datos	132
10.2.6	Formateo de datos	140
11	Modelos	148
11.1	Métricas	148
11.1.1	MAPE	148
11.1.2	MASE	149
11.2	Framework	150
11.2.1	Callbacks	151
11.3	Arquitecturas	152
11.3.1	Hiperparámetros de capas	152
11.3.2	Tipos	154
11.4	Entrenamiento	156
11.4.1	Datos de prueba	156

11.4.2	Desplazamiento temporal	158
11.4.3	Tipo de horizonte	159
11.4.4	Resultados	162
11.5	Ajuste de hiperparámetros	166
11.5.1	<i>Keras-tuner</i>	166
11.5.2	Espacio de búsqueda	168
11.5.3	Resultados <i>dataset</i> polución	170
11.5.4	Conclusión	173
11.5.5	Resultados <i>dataset</i> FSI	174
11.5.6	Cambio en preprocesamiento de datos	179
11.5.7	Conclusión	182
11.6	Entropía	183
11.6.1	Rendimiento de los algoritmos	183
11.6.2	<i>Dataset</i> de polución	184
11.6.3	<i>Dataset</i> FSI	185
11.6.4	Comparación de <i>datasets</i>	185
12	Conclusiones	187

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 una gran proporción de las mejoras en todos los campos se deberán al avance de la inteligencia artificial en cada uno de ellos. 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 Objetivos

2.1 Objetivos principales

Crear un modelo de *Machine Learning* cuyo fin será el pronóstico de la cantidad de defectos graves de las unidades en la línea de producción de una planta automotriz.

2.2 Objetivos secundarios

- **Clasificación de modelos de inteligencia artificial**

Uno de los objetivos que tiene el Laboratorio de Arquitectura de Computadoras es formar alumnos en el campo de la IA aplicada. Evaluar los diversos algoritmos disponibles de *Machine Learning*, explorando todas las posibilidades en aras de identificar el modelo que mejor se adapte al problema a afrontar.

- **Redes neuronales *feedforward*, convolucionales y recurrentes**

Analizar los principios fundamentales de las redes neuronales, así como sus elementos, algoritmos y arquitecturas que posibilitan su funcionalidad. Se explorará cada uno de sus tipos con el objetivo de reconocer sus fortalezas y debilidades según el tipo de problema a afrontar.

- **Tuning de hiperparámetros**

Establecer y evaluar la mejor configuración de parámetros para una determinada arquitectura de red con el fin de optimizar su desempeño.

- **Series temporales**

Dado que el problema a afrontar posee características temporales, se procede a describir y clasificar a las series temporales.

- **Entropía**

Calcular de forma teórica la entropía de una serie de datos con la finalidad de cuantificar su aleatoriedad sin ser relevante su origen.

- **Desarrollo**

Examinar el problema a afrontar en su contexto, para así determinar los requerimientos del mismo. Posteriormente haciendo uso de las herramientas para manipulación de datos obtener un formato acorde de la serie temporal para alimentar los modelos.

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.

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]

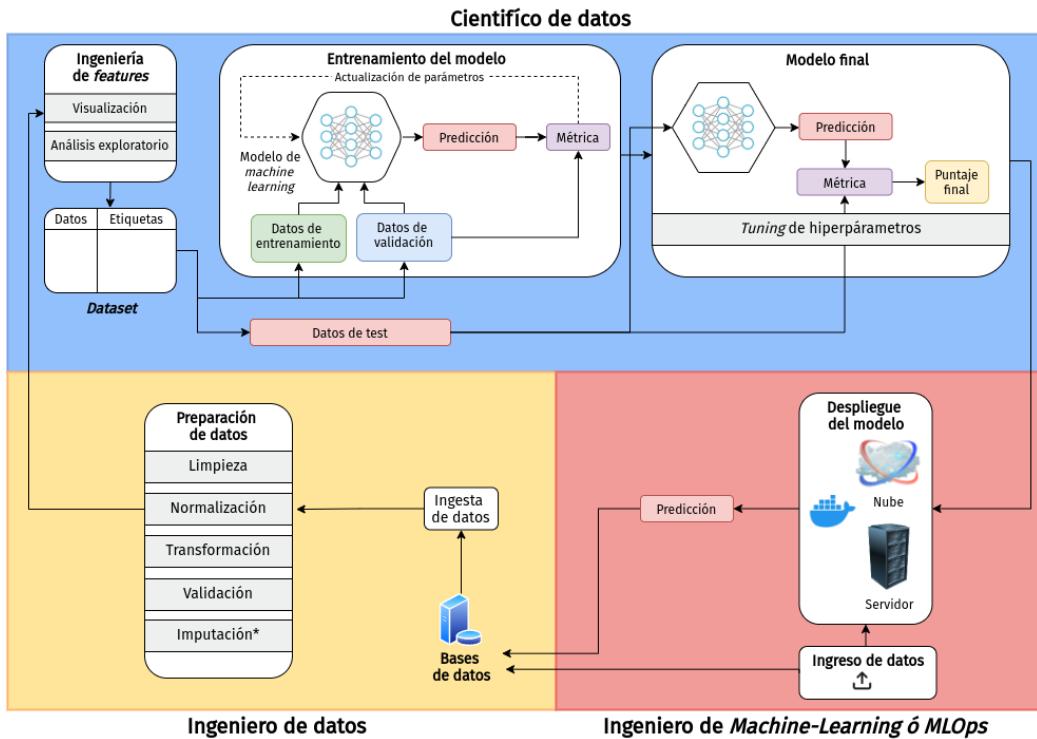


Figura 1: Diagrama de flujo de una aplicación de *Machine Learning*.

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 varias etapas. La industria se ha encargado de definir agrupar las tareas de estas etapas en 3 roles como se pueden observar en la Fig. 1, las cuáles se irán 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, con faltantes o ruidosos, entre otras cuestiones.

Una vez obtenido el *dataset* de entrada es primordial investigar, limpiar y transformar los datos con diversas técnicas. Una vez que los datos estén en condiciones procedemos al entrenamiento de nuestro modelo, con el propósito de lograr 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 comprendido para el aprendizaje automático, siendo ideal para tareas donde el modelo necesita predecir resultados. 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?”.

Cada una de estas piezas de datos que alimentan el modelo se denominan *features* (características), que son esencialmente una propiedad individual medible como puede ser la edad o el peso de una persona.

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 prueba para el cual la etiqueta debe ser inferida.

El algoritmo de aprendizaje debe ajustar el modelo predictivo al *dataset* de entrenamiento y usamos el *dataset* de prueba para evaluar la capacidad de generalización. El aprendizaje supervisado es ideal para tareas donde el modelo necesita predecir resultados.

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 y se intenta categorizar el elemento. Por tanto, se proporciona una distinción clara entre categorías y se denominan variables categóricas.

Clasificación de variables categóricas según tipo:

- **Ordinales:** tienen asociado un orden. *e.g.* las tallas de las camisetas “ $XL > L > M > S$ ”.
- **Nominales:** no implican un orden. *e.g.* no podemos asumir (en general) “naranja > azul > verde”

Tipo de clasificación según cantidad de variables:

- **Binaria:** se elige entre dos categorías, *e.g.* clasificar un mail entre Spam y No Spam.
- **Multiclasa:** se elige entre más de dos categorías, *e.g.* clasificar un conjunto de imágenes de frutas, donde habrá manzanas, naranjas y peras.

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).

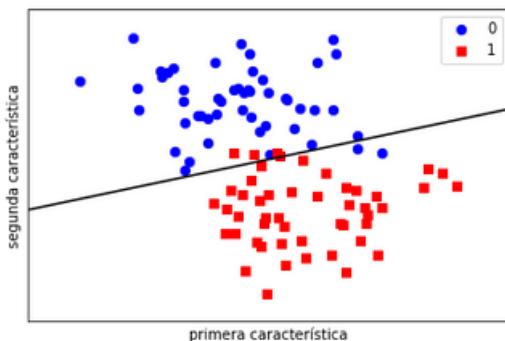


Figura 2: Ejemplo de clasificación binaria.

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.
Regresión logística	<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.
Árbol de decisión	<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. 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.

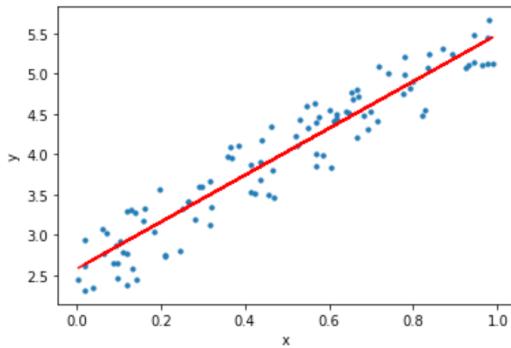


Figura 4: Ejemplo de regresión lineal.

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

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

	Ventajas	Desventajas
Ridge	<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 teniendo 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 inefficientes 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 que parece ser inconsistente con el resto del conjunto.

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.

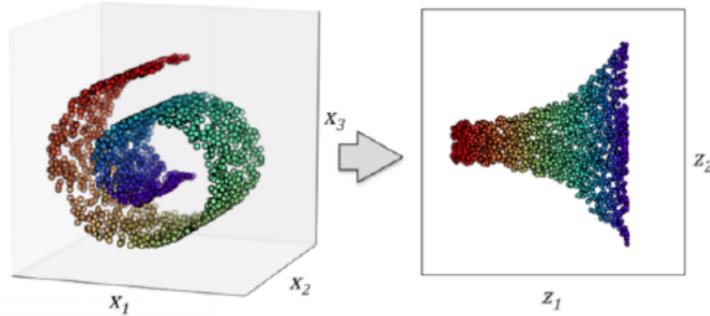


Figura 7: Reducción de dimensionalidad de 3D a 2D.

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 en 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 dendrograma los diferentes clusters.

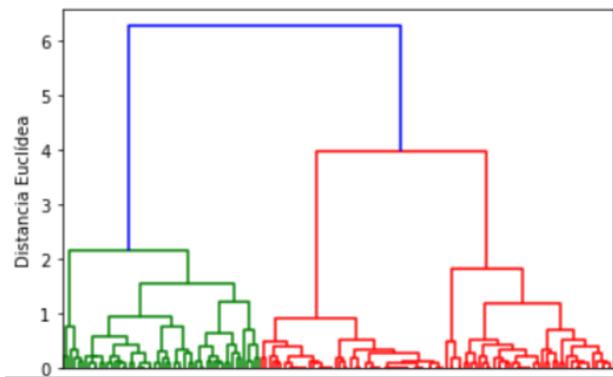


Figura 9: Dendrograma generado por *clustering* jerárquico.

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 hiperpará-

metros adicionales, como lo son la cantidad mínima de puntos y un radio ϵ , para lograr un óptimo funcionamiento.

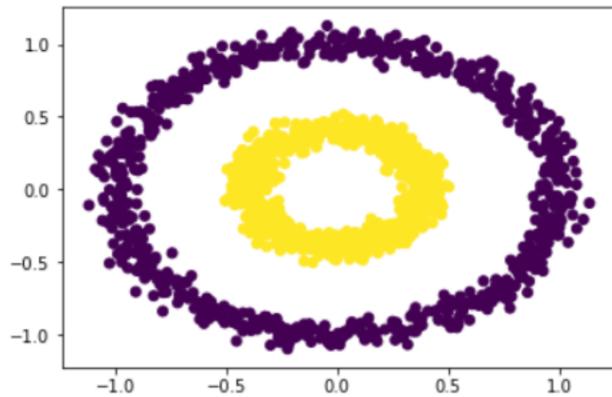


Figura 10: Clustering basado en densidades.

Enumeramos en la Fig. 11 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 que hacer con nuestros datos en función a nuestro objetivo.

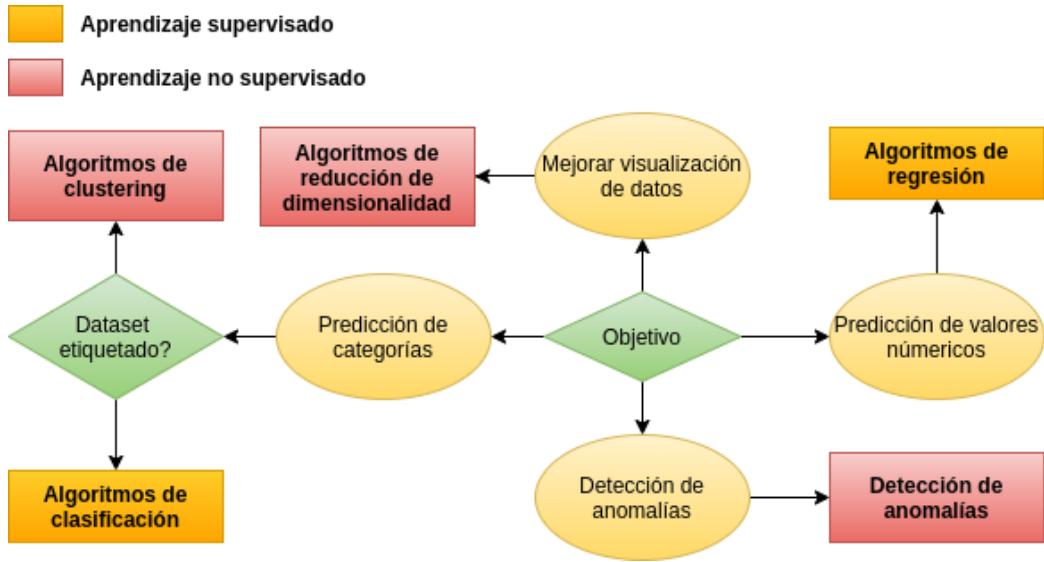


Figura 12: Diagrama general de algoritmos de aprendizaje supervisado y no supervisado.

3.5.1 Algoritmos supervisados

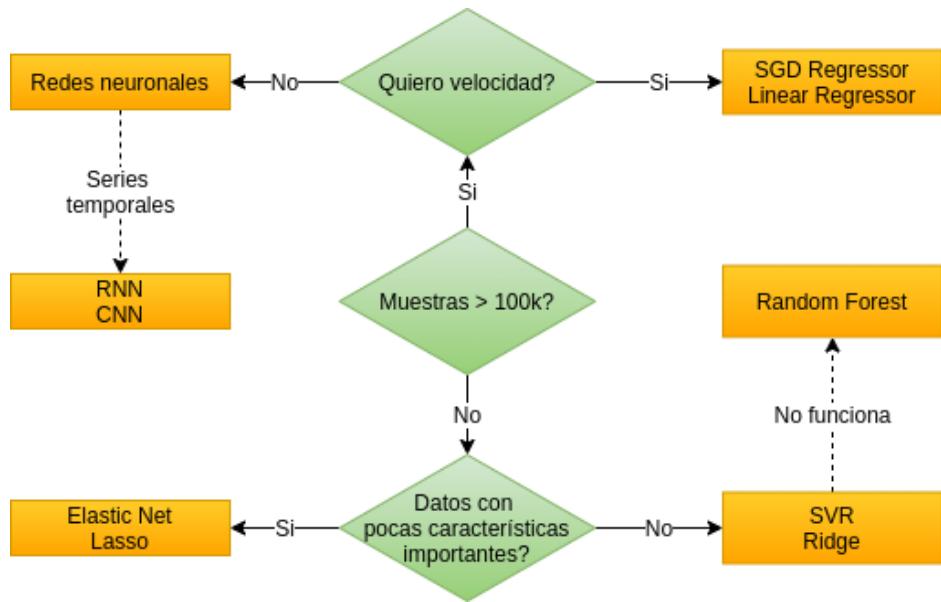


Figura 13: Diagrama general de los algoritmos de regresión.

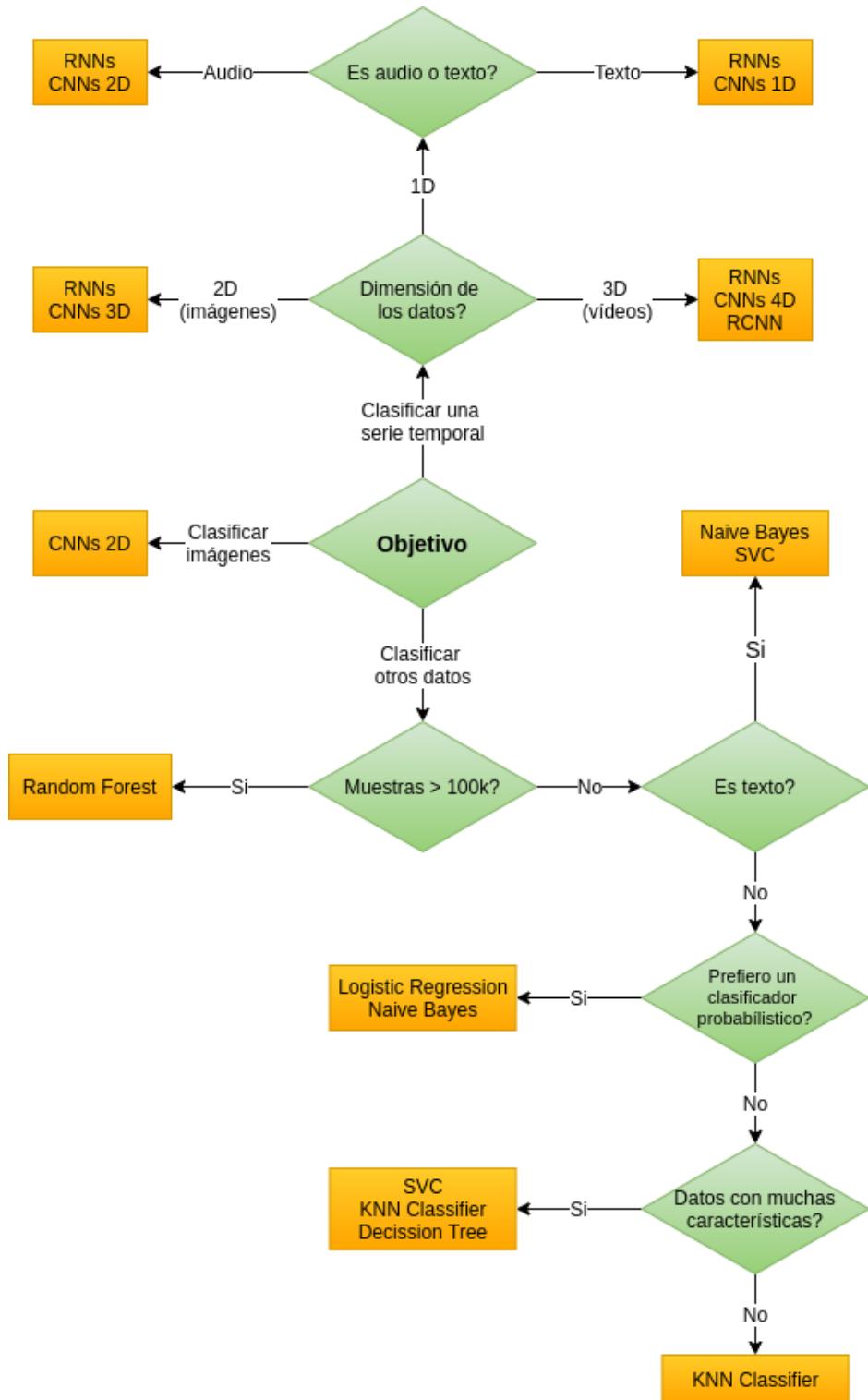


Figura 14: Diagrama general de los algoritmos de clasificación.

3.5.2 Algoritmos no supervisados

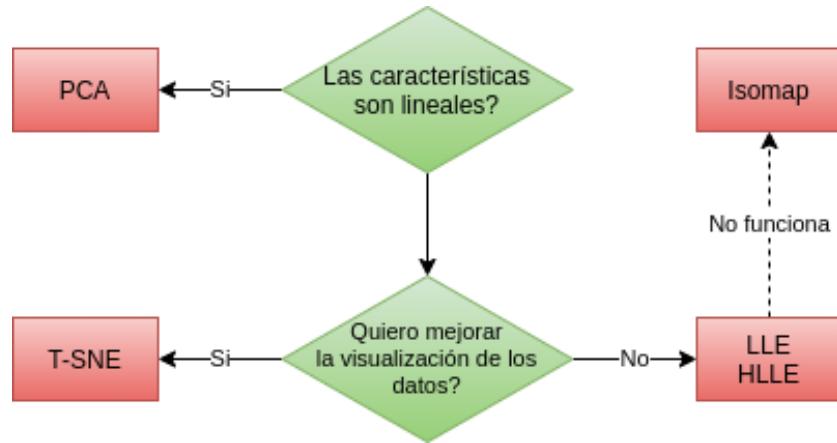


Figura 15: Diagrama general de los algoritmos de reducción de dimensión.

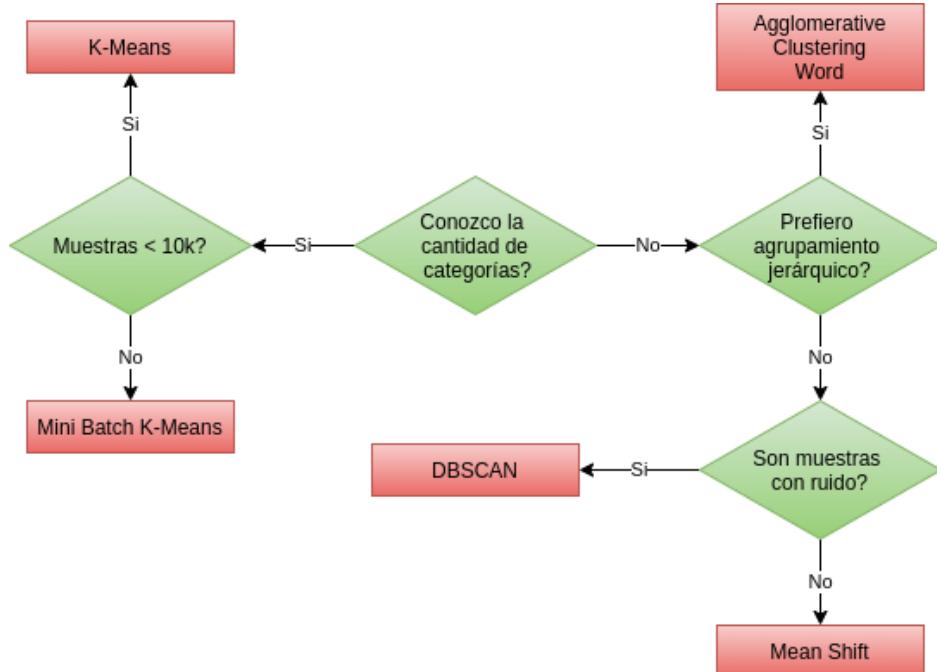


Figura 16: Diagrama general de los algoritmos de *clustering*.



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

4 Redes neuronales

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

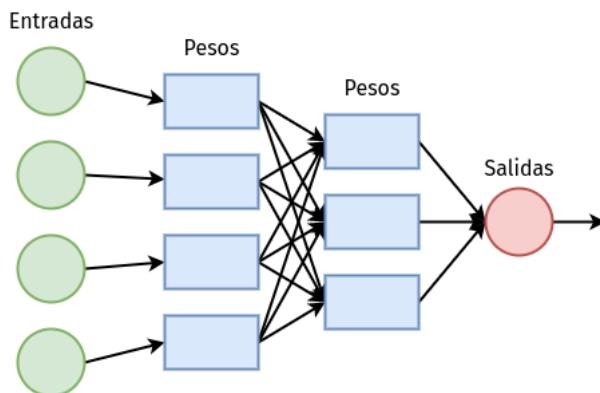


Figura 18: Arquitectura simple de red neuronal. [7]

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 otras 10000 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).

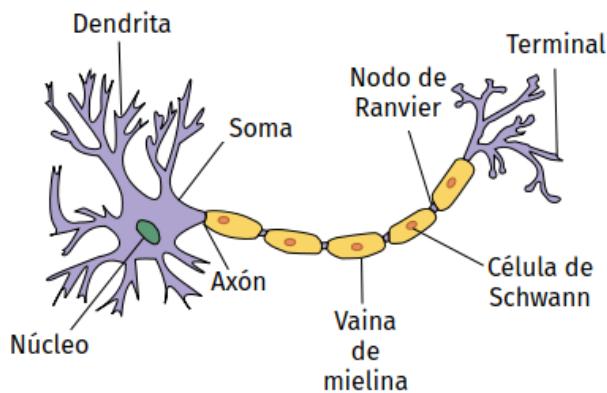


Figura 19: Estructura de una neurona biológica. [?]

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.

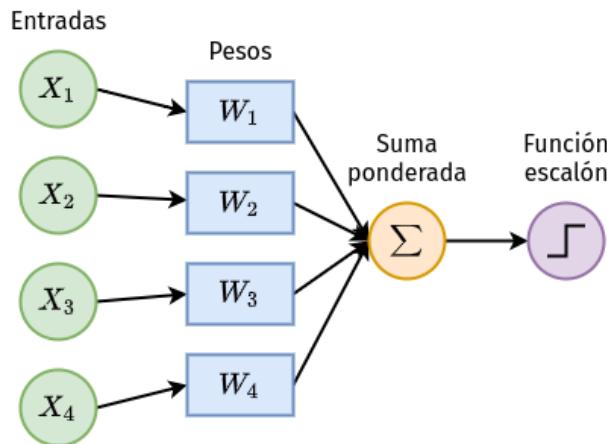


Figura 20: Simple NN.

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 $f(\text{net})$, donde $\text{net} = \sum_{i=1}^n w_i x_i$

4.3 Funciones de activación

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

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

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.

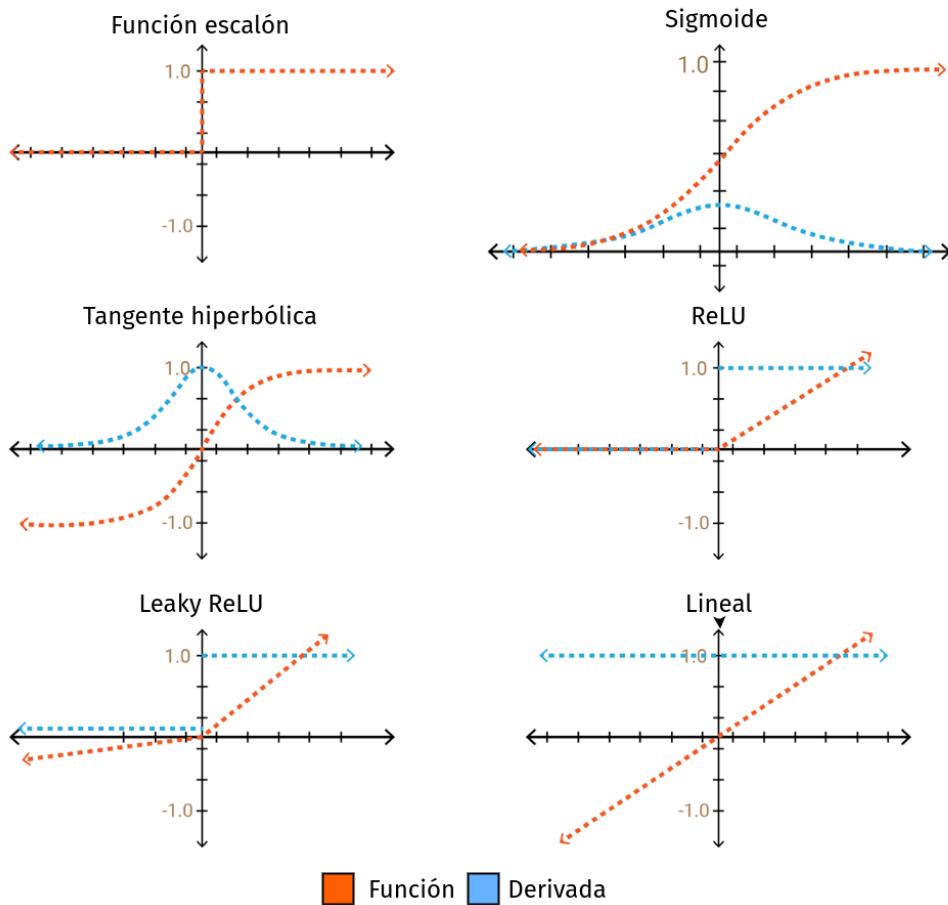


Figura 21: **Arriba-izquierda:** Función escalón. **Arriba-derecha:** Función sigmoidea. **Medio-izquierdo:** Tangente hiperbólica. **Medio-derecho:** activación ReLU (función activación más usada en *Deep Learning*). **Abajo-izquierdo:** Leaky ReLU, variante de ReLU que permite valores negativos. **Abajo-derecho:**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

Una de las funciones de activación más usadas en la historia de la literatura de NN es la función sigmoidea, que se define a continuación:

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

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 sigmoidea 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 a la sigmoidea) 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 sigmoidea 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 sigmoidea 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.

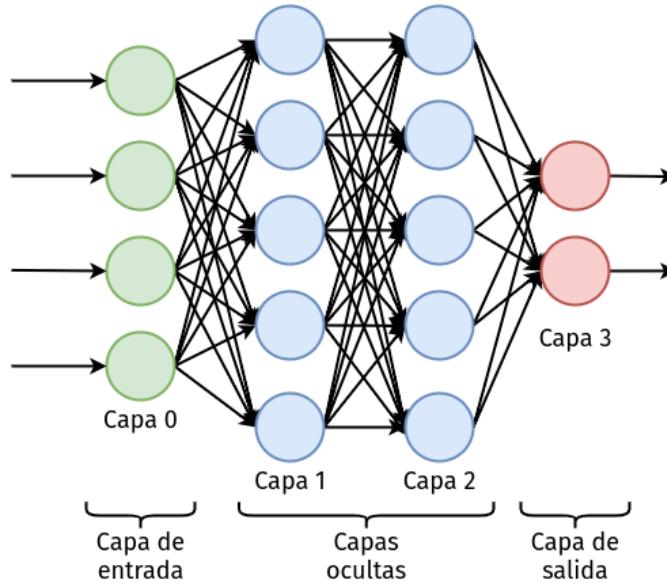


Figura 22: Un ejemplo de una red neuronal *feedforward*.

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 ítem 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 un solo ítem con 2 características, lo que significa 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 multivariable 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.

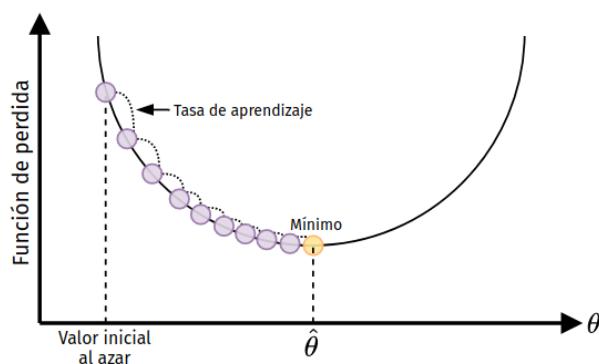


Figura 23: Representación gráfica del descenso de gradiente.

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.

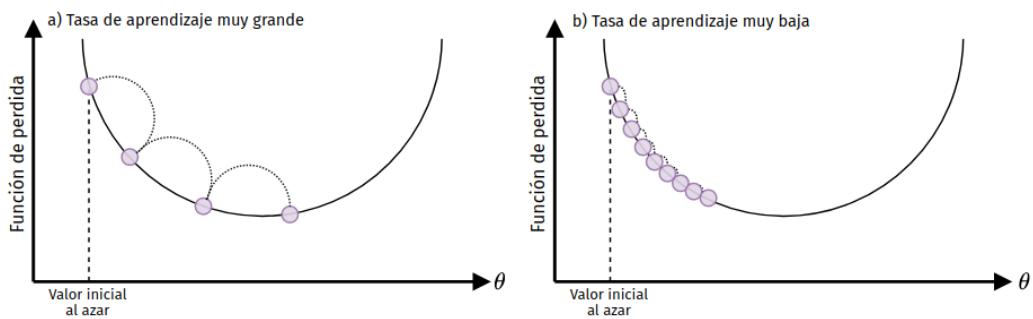


Figura 24: Ajuste de la tasa de aprendizaje.

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.

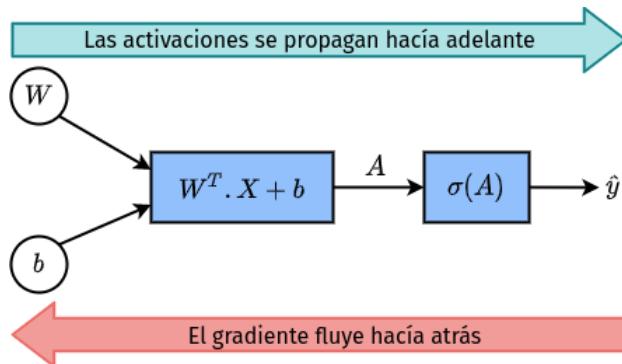


Figura 25: Flujo de cálculo de las variables.

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

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

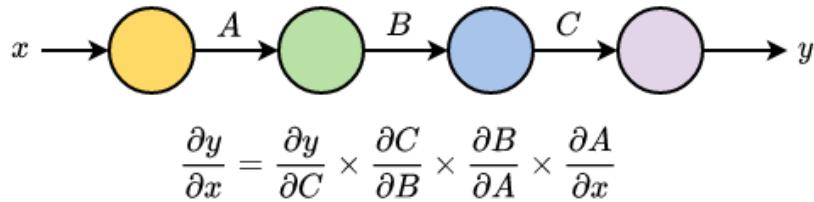


Figura 26: Representación de grafo simple.

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

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

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.

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

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

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

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

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)

El descenso de gradiente puede ser excepcionalmente lento en datasets muy grandes ya que en cada iteración se requiere calcular una predicción por cada punto en nuestro *dataset* de entrenamiento, antes que actualicemos 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.

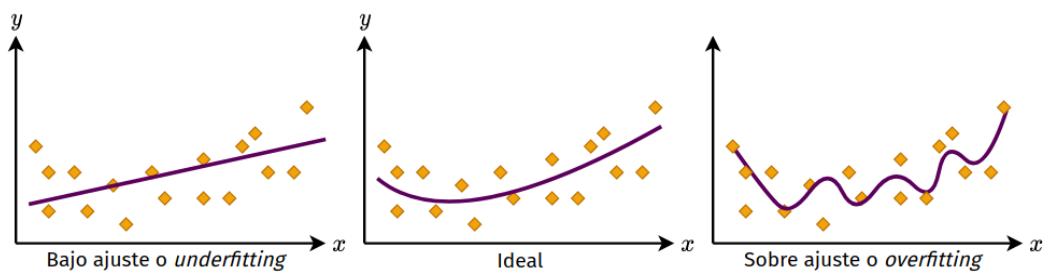


Figura 27: Distintas representaciones del ajuste en un mismo modelo.

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 el 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 entrena 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 $[a_1 \ a_2 \ \dots \ a_n]$.

Si las funciones f y g son representadas como vectores $a = [a_1 \ a_2 \ \dots \ a_m]$ y $b = [b_1 \ b_2 \ \dots \ b_n]$, entonces $f * g$ es un vector $c = [c_1 \ c_2 \ \dots \ c_{m+n-1}]$

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*.

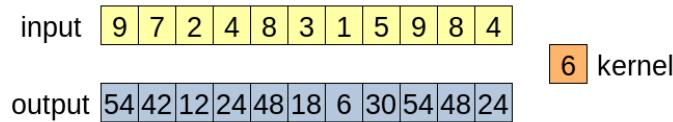


Figura 28: Vectores de convolución unidimensional con kernel simple.

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.

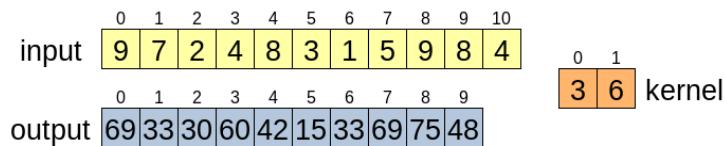


Figura 29: Vectores de convolución unidimensional con kernel doble.

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

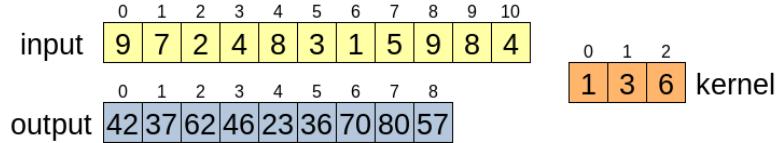


Figura 30: Vectores de convolución unidimensional con kernel triple.

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 izquierda a derecha y de arriba hacia abajo toda la matriz I .

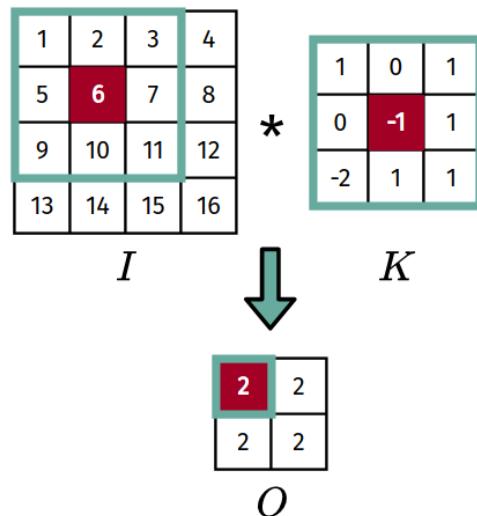


Figura 31: Vectores de convolución bidimensional con kernel 3×3 .

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.

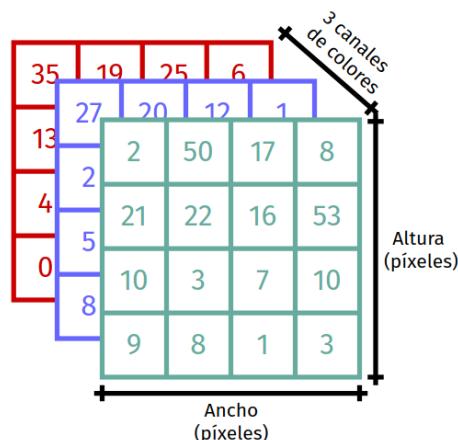


Figura 32: Imagen RGB $4 \times 4 \times 3$.

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.

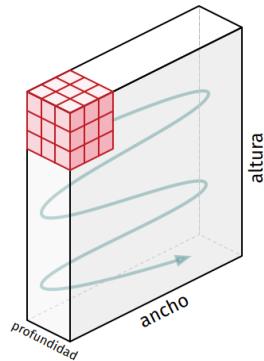
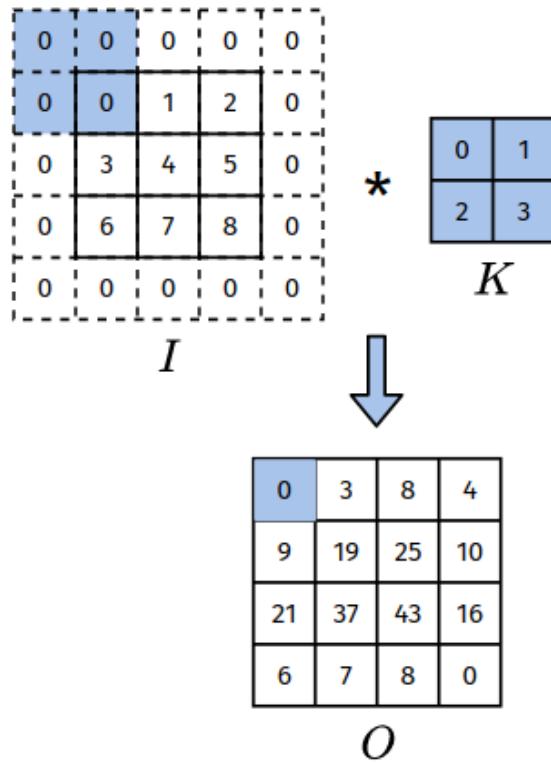


Figura 33: Movimiento del *kernel*.

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 .

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].

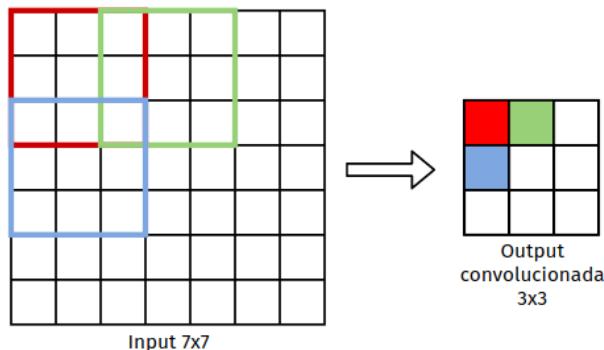


Figura 35: Convolución con *stride* de 2.

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*).

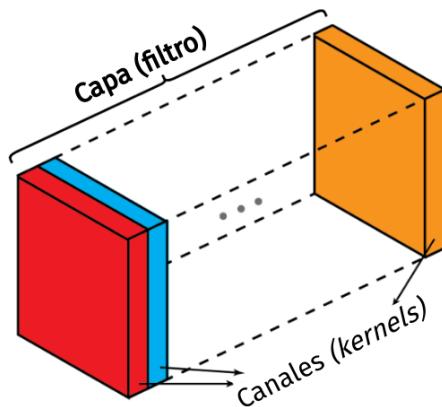


Figura 36: Una capa está compuesta de una colección de *kernels*.

La línea que divide filtro y *kernel* es un poco difusa. 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).

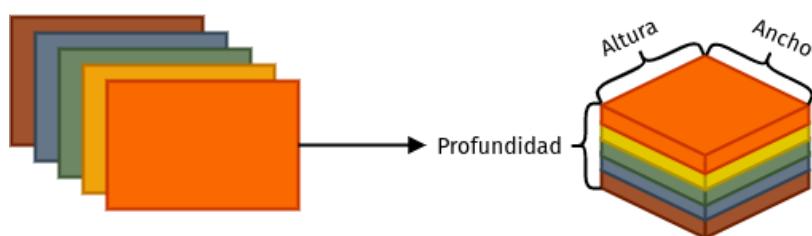


Figura 38: Luego de obtener los K mapas de activación, son apilados juntos para formar el volumen de entrada a la siguiente capa en la red.

Cada entrada en el volumen de salida es, por tanto, una salida de una neurona que "observa" 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.

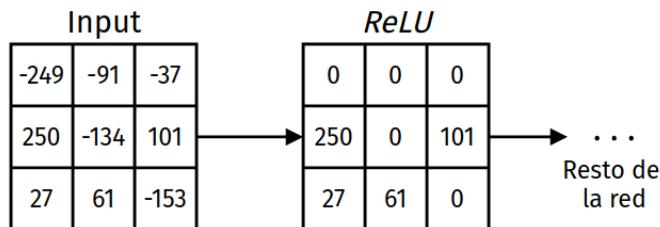


Figura 39: Un ejemplo de un volumen de entrada desplazándose a través de una ACT ReLU.

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 (APOOL)*: devuelve el promedio de todos los valores de la parte de la imagen cubierta por el *kernel* (figura 41 abajo).

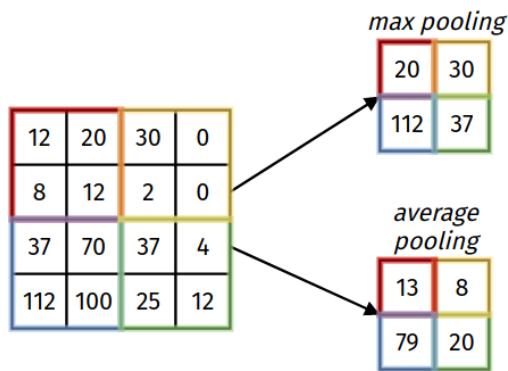


Figura 40: Tipos de *pooling*.

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*

Las capas de normalización por lotes o *Batch Normalization* (BN), como su nombre indica, se utilizan para normalizar las activaciones de un volumen de entrada determinado antes de pasarla a la siguiente capa de la red.

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* (D0) 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 D0 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).

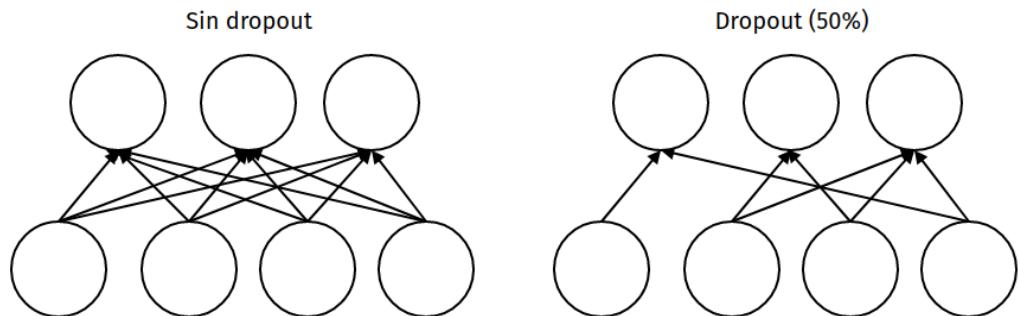


Figura 41: **Izquierda:** Dos capas FC sin D0. **Derecha:** Las mismas dos capas luego de realizar *dropout* sobre la mitad de las conexiones.

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]

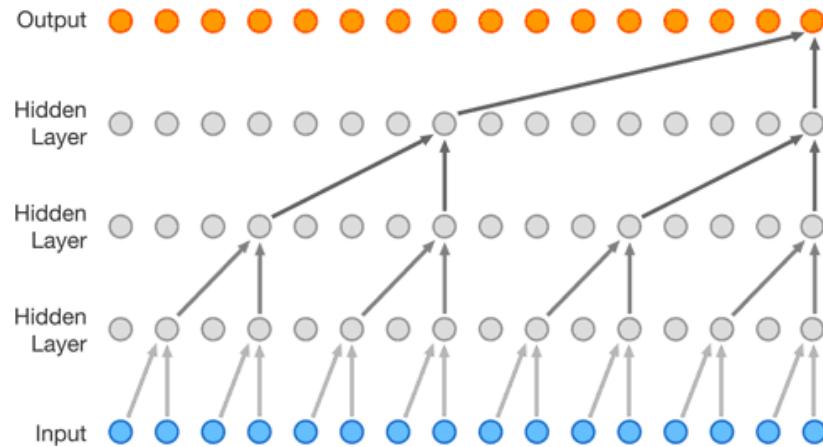


Figura 42: Paso de la información a través de la capa de convolución causal dilatada. [18]

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 desplazando las salidas convolucionales tradicionales en varios pasos de tiempo.

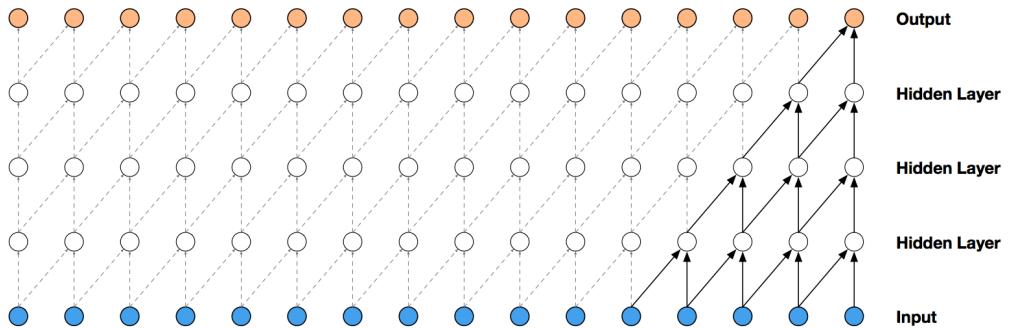


Figura 43: Visualización de una pila de capas causales convoluciones. [18]

Las convoluciones causales proporcionan la herramienta adecuada para manejar el flujo temporal, pero necesitamos una modificación adicional para manipular 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.

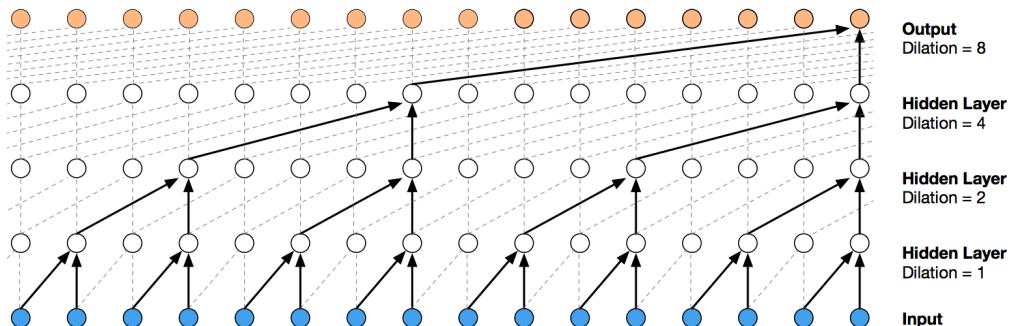


Figura 44: Visualización de una pila de capas causales convoluciones dilatadas. [18]

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 ó 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 (*e.g.* una secuencia de palabras o una secuencia de imágenes) 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 poseen la 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, donde cada rectángulo es un vector y cada flecha 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*.

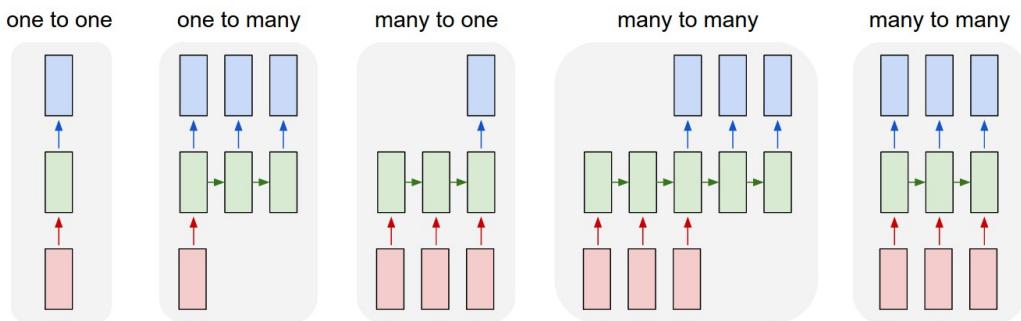


Figura 45: Tipos de arquitecturas para una *RNN*. [20]

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*" o subtítulo de imagen 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, 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 .

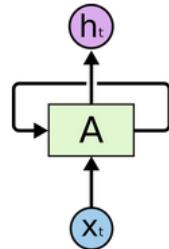


Figura 46: Unidad funcional de *RNN*. [21]

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.

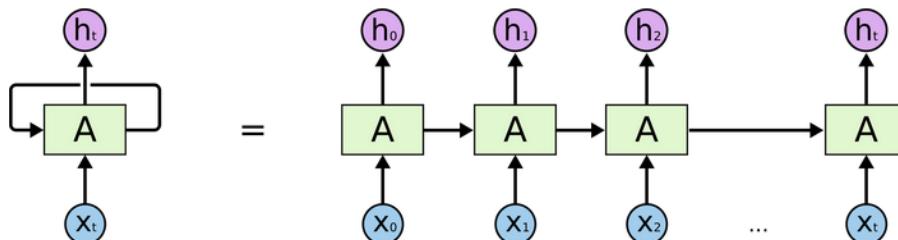


Figura 47: Una *RNN* desenrollada. [21]

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.

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.

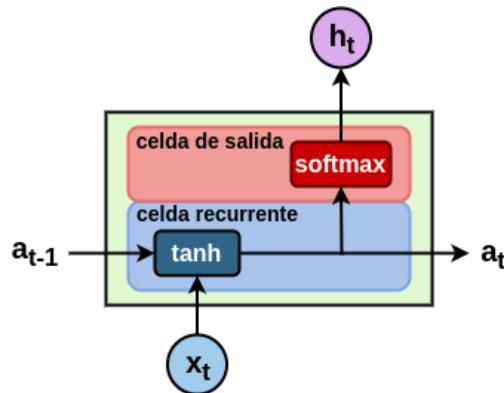


Figura 48: Unidad funcional *RNN* detallada. [21]

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 pediremos que modele la distribución de probabilidad del siguiente carácter en la secuencia dada una secuencia de caracteres anteriores. Esto nos permitirá generar texto nuevo, de a un carácter a la vez.

Como ejemplo práctico, suponga que solo tenemos un vocabulario de cuatro letras posibles **helo** y queremos entrenar a un *RNN* en la secuencia de entrenamiento **hello**.

Esta secuencia de entrenamiento es de hecho una fuente de 4 ejemplos de entrenamiento separados:

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

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

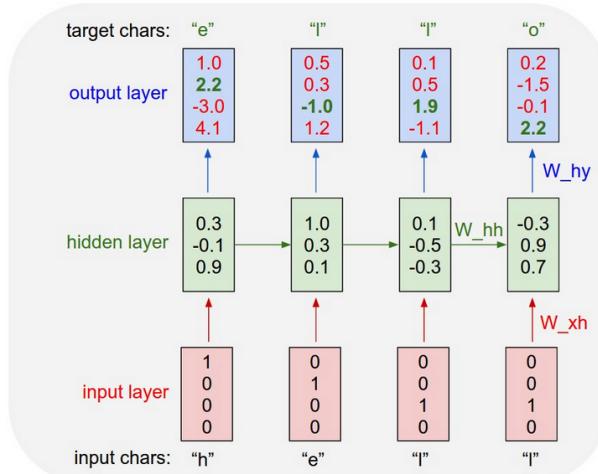


Figura 49: Visualización del flujo de información a través de una *RNN*. [20]

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.

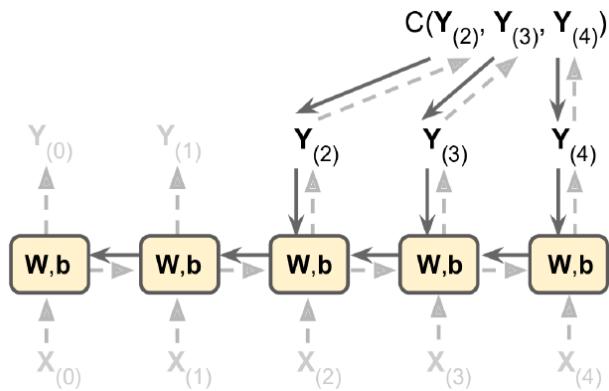


Figura 50: *Backpropagation* a través del tiempo.[22]

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].

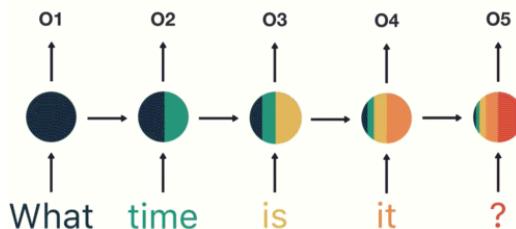
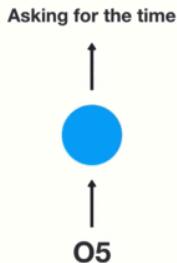
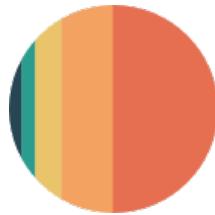


Figura 51: *RNN* siendo alimentada con las palabras de la oración. [23]

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. Se repite hasta que quedar 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*.[23]

Pero prestemos atención a la Fig. 53. Es posible que haya notado la irregular distribución de colores en los estados ocultos. Eso es para ilustrar un inconveniente con los *RNN* conocido como memoria a corto plazo, causado por el 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.

Figura 53: Estado oculto final de la *RNN*.[23]

La memoria a corto plazo es

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.

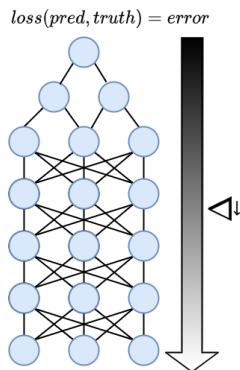


Figura 54: Desvanecimiento del gradiente desde las capas superiores a las inferiores.

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.

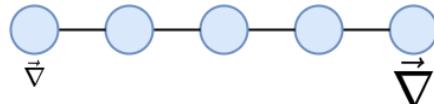


Figura 55: El gradiente se achica a medida que se propaga hacia atrás en el tiempo.

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.

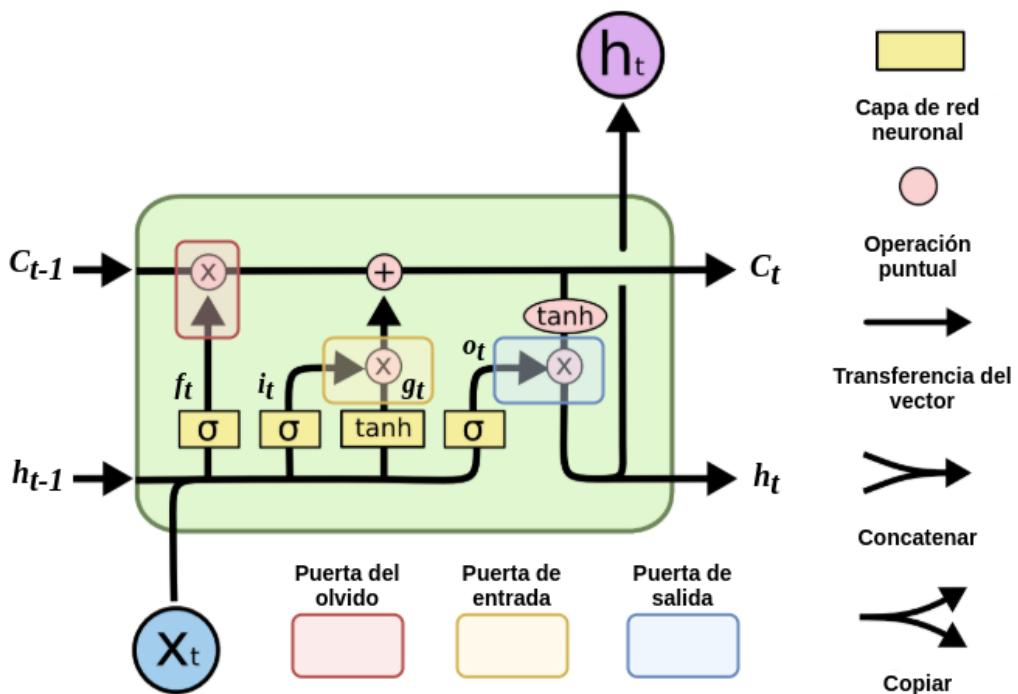


Figura 56: Celda LSTM.

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.

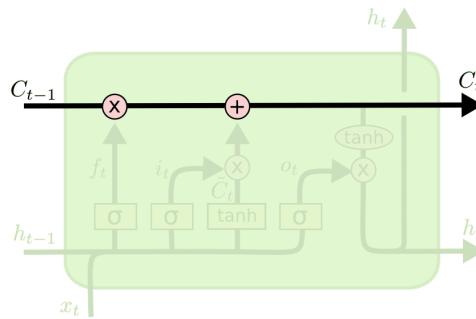


Figura 57: Celda de estado. [21]

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".

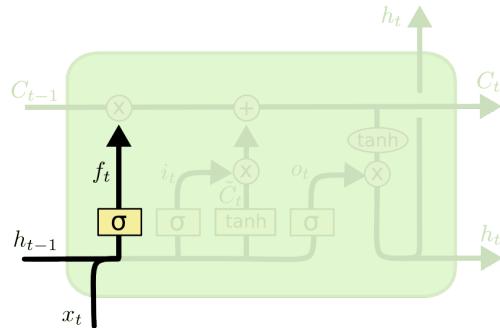


Figura 58: Puerta del olvido. [21]

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.

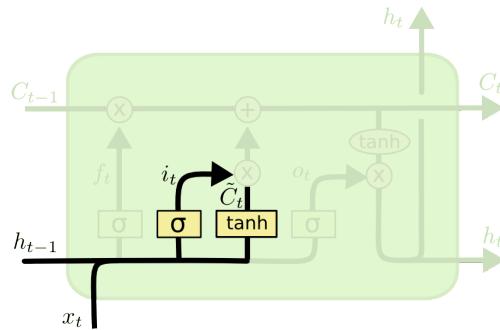


Figura 59: Puerta de entrada. [21]

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.

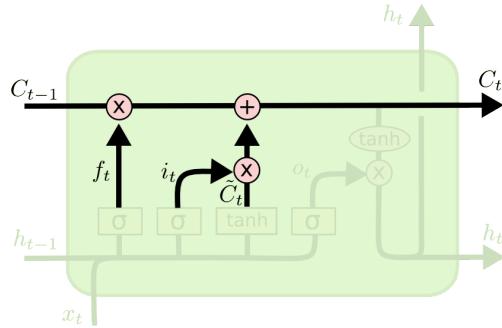


Figura 60: Actualización de la celda. [21]

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.

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

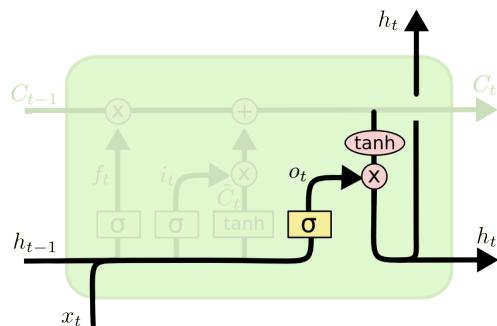


Figura 61: Puerta de salida. [21]

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 (puerta de entrada), almacenarla en el estado a largo plazo, aprender a preservarla durante el tiempo que sea necesario (puerta del olvido) y aprender a extraerla siempre que sea necesario (puerta de salida).

6.5.2 GRU

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

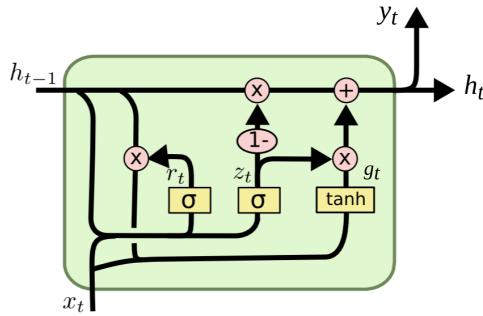


Figura 62: Celda GRU. [21]

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 describirlas un poco más y ejemplificar en que casos sería provechosa su utilización. Tomaremos como referencia la Figura 63.

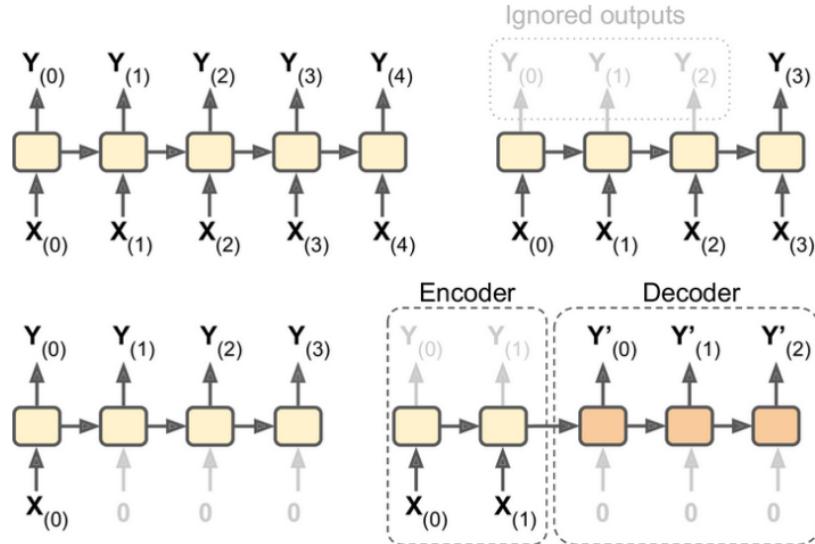


Figura 63: **Superior-izquierda:** *seq-to-seq*. **Superior-derecha:** *sec-to-vector*. **Inferior-izquierda:** *vector-to-sec*. **Inferior-derecha:** *encoder-decoder*. [22]

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 Optimización de hiperparámetros

Los parámetros que definen la arquitectura del modelo se denominan hiperparámetros y, por lo tanto, este proceso de búsqueda de la arquitectura del modelo ideal se denomina ajuste de hiperparámetros o *hyperparameters tuning*. [25]

Los hiperparámetros no son parámetros del modelo y no se pueden entrenar directamente a partir de los datos. Mientras que los parámetros del modelo especifican cómo transformar los datos de entrada en la salida deseada a través del entrenamiento optimizando una función de pérdida, los hiperparámetros definen cómo está realmente estructurado nuestro modelo.

Los métodos de ajuste de hiperparámetros se relacionan con cómo muestramos posibles candidatos a la arquitectura del modelo a partir del espacio de posibles valores de hiperparámetros. Esto se denomina a menudo "buscar" en el espacio de hiperparámetros los valores óptimos.

En general, este proceso incluye:

1. Definir los modelos.
2. Definir el rango de valores posibles para todos los hiperparámetros, es decir el espacio de búsqueda o *search space*.
3. Definir un algoritmo para obtener valores de hiperparámetros.
4. Definir un criterio evaluativo para juzgar el modelo.

La optimización de hiperparámetros se representa en forma de ecuación como:

$$x^* = \underset{x \in X}{\operatorname{argmin}} f(x)$$

Aquí $f(x)$ representa una puntuación objetivo para minimizar (usualmente una métrica), evaluada en el set de validación; x^* es el conjunto de hiperparámetros que produce el menor valor para la función objetivo, y x puede tomar cualquier valor en el dominio X . En términos simples, queremos encontrar los hiperparámetros del modelo que producen la mejor puntuación en la métrica del conjunto de validación.

El problema con la optimización de hiperparámetros es que evaluar la función objetivo para encontrar la puntuación es extremadamente costoso. Cada vez que probamos diferentes hiperparámetros, tenemos que entrenar un modelo con los datos de entrenamiento, hacer predicciones sobre los datos de validación y luego calcular la métrica de validación. Con una gran cantidad de hiperparámetros y modelos complejos, como conjuntos o redes neuronales profundas que pueden tardar días en entrenarse, y el problema no escala.

7.1 Algoritmos

Los algoritmos de optimización de hiperparámetros se pueden dividir en tres categorías principales:

- **Búsqueda exhaustiva del espacio:** estos métodos son una primera opción atractiva para la optimización, ya que son muy fáciles de codificar, se pueden ejecutar en paralelo y no necesitan ningún tipo de ajuste. Su inconveniente es que no hay garantía de encontrar un mínimo local con cierta precisión, excepto si el espacio de búsqueda se muestrea a fondo. Esto no supone ningún problema si el modelo es muy rápido de ejecutar y el número de hiperparámetros es bajo. Si el modelo tarda mucho en ejecutarse utilizando una gran cantidad de recursos computacionales estos métodos son ineficaces, ya que no utilizan la información obtenida de todos los intentos anteriores.
- **Modelos subrogados:** un modelo subrogado de la función de hiperparámetros puede ajustarse a los intentos anteriores y decir dónde podría estar el mínimo local. Estos métodos se denominan optimización basada en modelos secuenciales (SMBO, *Sequential Model-Based Optimization*).
- **Reducciones sucesivas a la mitad:** son algoritmos que se encargan de seleccionar dos configuraciones de hiperparámetros e ir las descartando tempranamente para así converger a la mejor.

Para todos los algoritmos, se debe definir un espacio de búsqueda de antemano para así establecer límites para todos los hiperparámetros.

7.1.1 Búsqueda exhaustiva del espacio

Búsqueda por grilla (*Grid search*)

El espacio de búsqueda de cada hiperparámetro se discretiza, y el espacio de búsqueda total se discretiza como los productos cartesianos de ellos. Luego, el algoritmo lanza un aprendizaje para cada una de las configuraciones de hiperparámetros y selecciona la mejor al final. Es un problema paralelo (siempre que uno tenga la potencia de cálculo necesaria para entrenar varios modelos al mismo tiempo) pero sufre el problema de la dimensionalidad (el número de configuraciones a probar es exponencial con respecto al número de hiperparámetros a ser optimizado).

Búsqueda aleatoria (*Random search*) Una variación del algoritmo anterior, que muestrea aleatoriamente el espacio de búsqueda en lugar de discretizarlo con una cuadrícula cartesiana. El algoritmo no tiene fin. En su lugar, se debe especificar un presupuesto de tiempo (en otras palabras, una número de *trials* o pruebas).

Este algoritmo también sufre el problema de la dimensionalidad para alcanzar una densidad de muestreo fija preestablecida. Una de las ventajas de la búsqueda aleatoria es que si dos hiperparámetros están poco correlacionados, la búsqueda aleatoria permite encontrar con mayor precisión los óptimos de cada parámetro, como se muestra en la fig. 64.

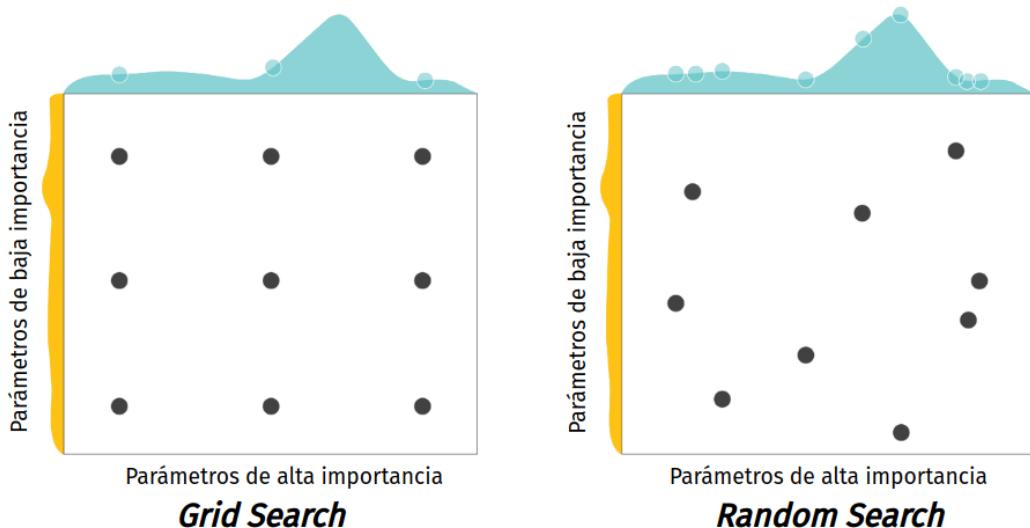


Figura 64: Comparación de *Grid Search* vs *Random Search*.

7.1.2 Modelos subrogados

Optimización bayesiana (*Bayesian Optimization*)

Tenemos una función $f(x)$, que es costosa de calcular, no es necesariamente una expresión analítica y no conoce su derivada. Y nuestro objetivo es encontrar los mínimos globales. [26]

Esta es sin duda una tarea difícil, más incluso que otros problemas de optimización dentro del aprendizaje automático. El descenso de gradiente, por ejemplo, tiene acceso a las derivadas de una función y aprovecha los atajos matemáticos para una evaluación de expresiones más rápida.

Alternativamente, en algunos escenarios de optimización, la función es simple de evaluar. Si podemos obtener cientos de resultados para variantes de una entrada x en unos pocos segundos, se puede emplear una simple *grid search* con buenos resultados.

Desafortunadamente, no contamos con estas bondades y estamos limitados en nuestra optimización por varios frentes, en particular:

- **Es costoso de calcular.** Idealmente, podríamos consultar la función lo suficiente como para replicarla, pero nuestro método de optimización debe funcionar con una muestra limitada de entradas.
- **Se desconoce la derivada.** Por tanto no podemos aplicar el descenso del gradiente y sus derivados.
- **Necesitamos encontrar los mínimos globales.** Esta es una tarea difícil incluso para un método sofisticado como el descenso de gradiente, entonces nuestro modelo de alguna manera necesitará un mecanismo para evitar quedar atrapado en los mínimos locales.

La solución es **optimización bayesiana** (*Bayesian Optimization*), que proporciona un marco elegante para abordar problemas que se asemejan al escenario descrito para encontrar el mínimo global en el menor número de pasos.

Construyamos un ejemplo hipotético de la función objetivo $c(x)$, o el costo de un modelo dado una entrada x . Por supuesto, el aspecto de la función estará oculto al optimizador; la verdadera forma de $c(x)$ es la de la fig 65.

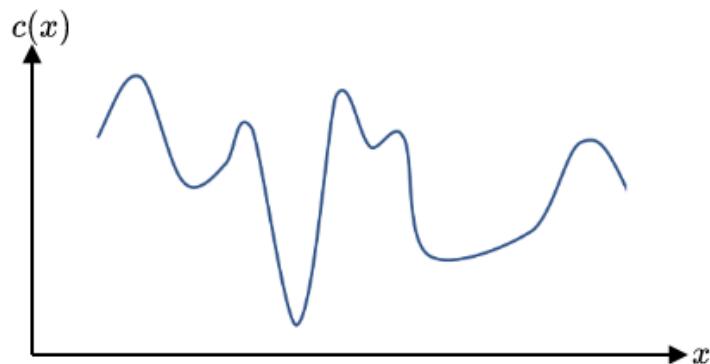


Figura 65: Representación gráfica de la función $c(x)$.

La optimización bayesiana aborda esta tarea mediante un método conocido como optimización subrogada. Por contexto, una maternidad subrogada es la práctica por la que, previo acuerdo con otra persona, una mujer se queda embarazada, lleva la gestación a término y da a luz a un bebé para esa otra persona o pareja [27]; en ese contexto, una función subrogada es una aproximación de la función objetivo y se forma sobre la base de puntos muestrados tal como muestra la fig. 66.

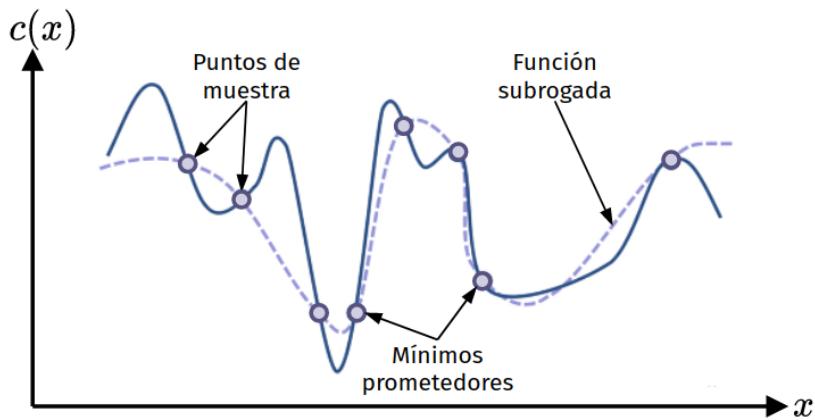


Figura 66: Función subrogada.

Basándonos en la función subrogada, podemos identificar qué puntos son mínimos prometedores. Decidimos tomar más muestras de estas regiones prometedoras y actualizar la función subrogada en consecuencia (fig. 67).

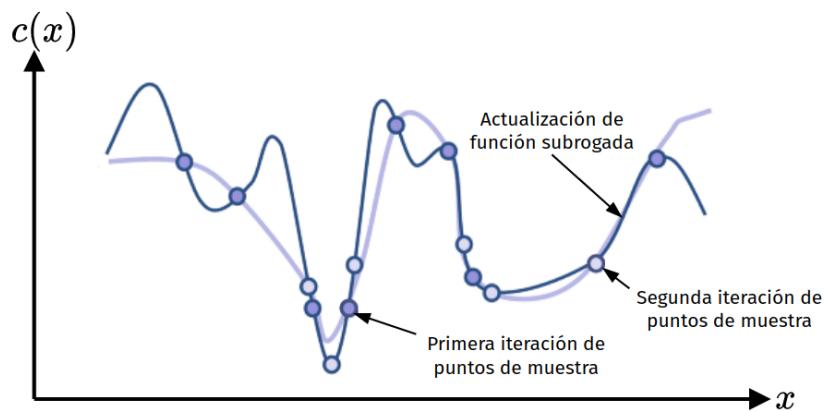


Figura 67: Actualización de función subrogada.

En cada iteración, continuamos examinando la función subrogada actual, aprendemos más sobre las áreas de interés mediante el muestreo y actualizamos la función. Tenga en cuenta que la función subrogada se expresará matemáticamente de una manera significativamente menos costosa de evaluar (por ejemplo, $y = x$ es una aproximación de una función más costosa, $y = \arcsin\left(\frac{1-\cos^2 x}{\sin x}\right)$ dentro de un cierto rango).

Después de un cierto número de iteraciones, estamos destinados a llegar a un mínimo global, a menos que la forma de la función sea muy irregular (en el sentido de que tiene grandes oscilaciones hacia arriba y hacia abajo) por tanto se debería hacer más foco sobre los datos que en la optimización.

Este enfoque no hace ninguna suposición sobre la función (excepto que es optimizable en primer lugar), no requiere información sobre derivadas y es capaz de utilizar el razonamiento de sentido común mediante el uso ingenioso de una función de aproximación continuamente actualizada. La costosa evaluación de nuestra función objetivo original no es un problema en absoluto.

La esencia de los modelos bayesianos es la actualización de una creencia previa ("antes") utilizando nueva información para producir una creencia posterior ("después") actualizada. Esto es exactamente lo que hace la optimización subrogada en este caso, por lo que se puede representar mejor a través de sistemas, fórmulas e ideas bayesianos.

La función subrogada generalmente está representada por **procesos gaussianos**, que se puede considerar como una tirada de dados que devuelven funciones ajustadas a puntos de datos dados (por ejemplo, \sin , \log) en lugar de números del 1 al 6 (fig. 68). El proceso devuelve varias funciones, que tienen probabilidades asociadas.

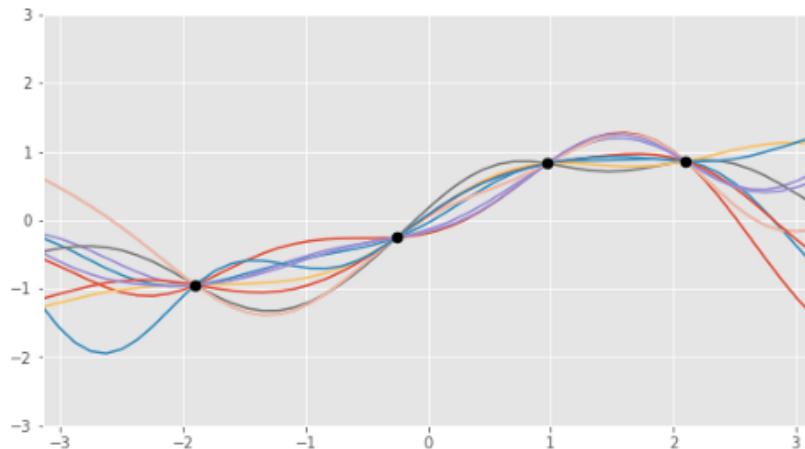


Figura 68: Varias funciones generadas por procesos gaussianos para cuatro puntos de datos. [26]

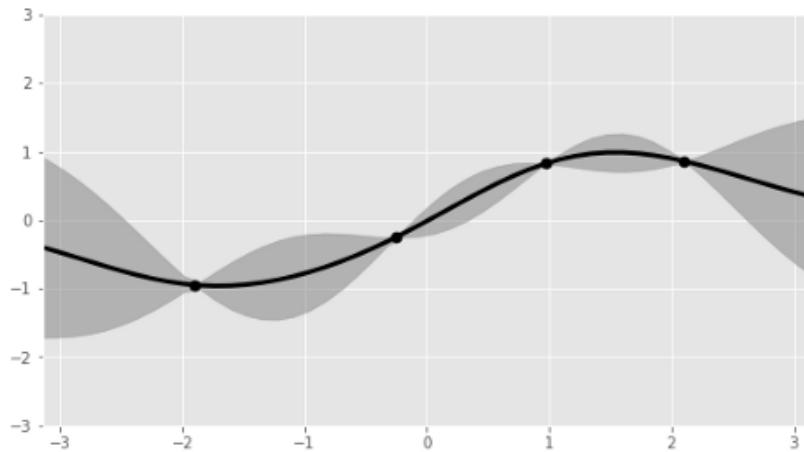


Figura 69: Las funciones agregadas. [26]

Hay una buena razón por la que se utilizan procesos gaussianos, y no algún otro método de ajuste de curvas, para modelar la función subrogada: es de naturaleza bayesiana. Un proceso gaussiano es una distribución de probabilidad, como una distribución de los resultados finales de un evento (por ejemplo, 1/2 probabilidad de lanzar una moneda), pero sobre todas las funciones posibles.

Por ejemplo, podemos definir el conjunto actual de puntos de datos como 40% representable por la función $a(x)$, 10% por la función $b(x)$, etc. Al representar la función sustituta como una distribución de probabilidad, se puede actualizar con nueva información a través de procesos bayesianos intrínsecamente probabilísticos. Quizás cuando se introduce nueva información, los datos solo son representables en un 20% por la función $a(x)$. Estos cambios se rigen por fórmulas bayesianas.

La función subrogada, representada como una distribución de probabilidad se actualiza con una **función de adquisición**. Esta función es responsable de impulsar la propuesta de nuevos puntos para probar, en una compensación de exploración y explotación:

- **Eplotación:** busca muestrear dónde el modelo subrogada predice un buen objetivo. Esto está aprovechando los lugares prometedores conocidos. Sin embargo, si ya hemos explorado una determinada región lo suficiente, la explotación continua de información conocida producirá poca ganancia.
- **Exploración:** busca muestrear en lugares donde la incertidumbre es alta. Esto asegura que no quede ninguna región importante del espacio sin explorar; es posible que los mínimos globales se encuentren allí.

Una función de adquisición que fomente demasiada explotación y muy poca exploración hará que el modelo resida solo un mínimo que encuentre primero (generalmente local). Una función de adquisición que fomente lo contrario no se quedará en mínimos, locales o globales, en primer lugar. Dando buenos resultados en un delicado equilibrio.

La función de adquisición, que denotaremos $a(x)$, debe considerar tanto la explotación como la exploración. Las funciones de adquisición comunes incluyen la mejora esperada y la probabilidad máxima de mejora, todas las cuales miden la probabilidad de que una entrada específica pueda dar resultados en el futuro, dada la información sobre el anterior (el proceso gaussiano). Existen diversas funciones de adquisición tales como *Upper Confidence Bound (UCB)*, *Probability of Improvement (PI)*, *Expected Improvement (EI)*, etc. [28]

Juntemos las piezas. La optimización bayesiana se puede realizar como tal:

1. Inicializar una distribución previa de función subrogada del proceso gaussiano.
2. Elija varios puntos de datos x de modo que se maximice la función de adquisición $a(x)$ que opera en la distribución anterior actual.
3. Evalúe los puntos de datos x en la función de costo objetivo $c(x)$ y obtenga los resultados, y .
4. Actualice la distribución previa del proceso gaussiano con los nuevos datos para producir un posterior (que se convertirá en el anterior en el siguiente paso).
5. Repita los pasos 2 a 5 para varias iteraciones.
6. Interprete la distribución actual del proceso gaussiano (que es poco costoso de realizar) para encontrar los mínimos globales.

La optimización bayesiana se trata de poner ideas probabilísticas detrás de la idea de optimización subrogada. La combinación de estas dos ideas crea un sistema poderoso con muchas aplicaciones, desde el desarrollo de productos farmacéuticos hasta vehículos autónomos. Sin embargo, más comúnmente en el aprendizaje automático, la optimización bayesiana se usa para la optimización de hiperparámetros. Por ejemplo, si estamos entrenando un clasificador de aumento de gradiente, hay docenas de parámetros, desde la tasa de aprendizaje hasta la profundidad máxima de capas de la red, la cantidad de neuronas de cada capa entre otras por nombrar algunos ejemplos. En este caso, x representa los hiperparámetros del modelo y $c(x)$ representa el rendimiento del modelo.

La principal motivación para utilizar la optimización bayesiana se encuentra en escenarios en los que es muy costoso evaluar la salida, tal como el entrenamiento de múltiples arquitecturas de modelos para encontrar la mejor.

Para dejar clara la idea:

- La optimización subrogada utiliza una función subrogada o aproximada para estimar la función objetivo a través del muestreo.
- La optimización bayesiana coloca la optimización sustituta en un marco probabilístico al representar funciones sustitutas como distribuciones de probabilidad, que pueden actualizarse al obtener nueva información.
- Las funciones de adquisición se utilizan para evaluar la probabilidad de que la exploración de un determinado punto en el espacio produzca un rendimiento "bueno" dado lo que se conoce actualmente de la exploración y la explotación anteriores, equilibrando la exploración.
- Utilice la optimización bayesiana principalmente cuando la función objetivo sea costosa de evaluar, comúnmente utilizada en el ajuste de hiperparámetros.

Ahora extrapolemos el conocimiento adquirido al ajuste de hiperparámetros teniendo en cuenta que la optimización bayesiana crea un modelo de probabilidad de la función objetivo. [29]

La función objetivo real es una función fija. Supongamos que se parece a la figura 70, pero obviamente desconocemos su forma.

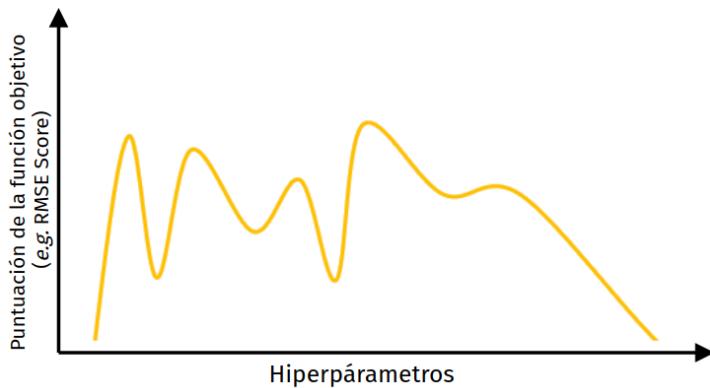


Figura 70: Función objetivo real.

Dado que calcular la métrica de nuestro modelo es costoso, digamos que solo tenemos 10 muestras, representada como círculos negros en la figura 71.

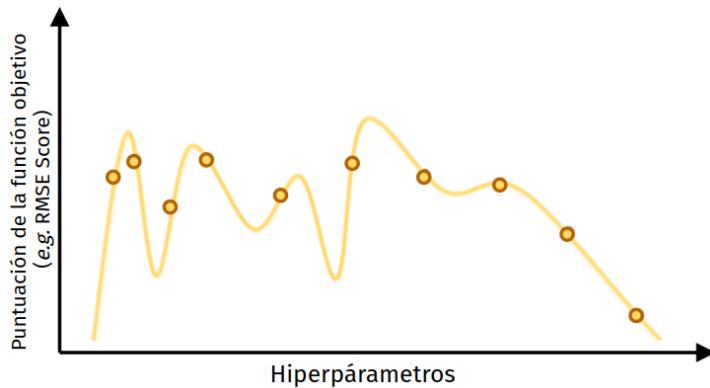


Figura 71: Muestras de la función objetivo real.

Usando estas 10 muestras, necesitamos construir un modelo subrogado (también llamado modelo de superficie de respuesta) para aproximarnos a la función objetivo real. Observemos la fig. 72, el modelo subrogado se representa como la línea azul y la sombra azul representa la desviación.

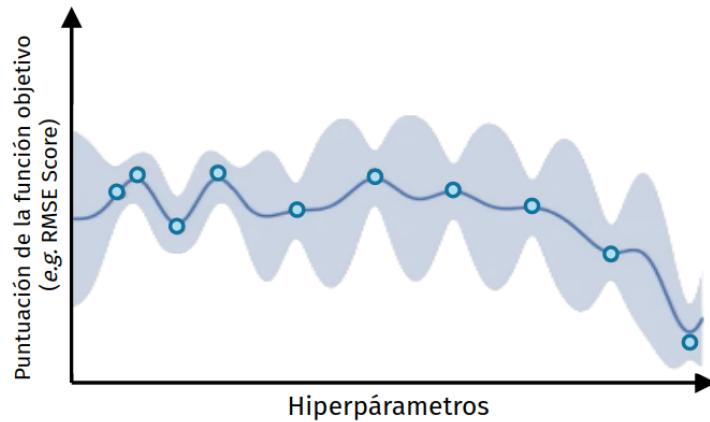


Figura 72: Modelo subrogado.

Un modelo (o función) subrogado es un modelo probabilístico que asigna hiperparámetros a una probabilidad de la puntuación de una métrica en la función objetivo real. Entonces queda formalmente definido como:

$$P(y|x)$$

Dónde y representa la puntuación de la métrica y x el conjunto de hiperparámetros seleccionado.

Ahora tenemos 10 muestras de la función objetivo, para decidir qué parámetro probar como la undécima muestra necesitamos construir la función de adquisición. El siguiente hiperparámetro de elección es donde se maximiza la función de adquisición.

En la fig. 73 abajo, el tono verde es la función de adquisición y la línea recta roja es donde se maximiza. Por lo tanto, el hiperparámetro correspondiente y su puntuación de función objetivo, representada como un círculo rojo, se utiliza como la undécima muestra para actualizar el modelo subrogado (fig. 73 arriba).

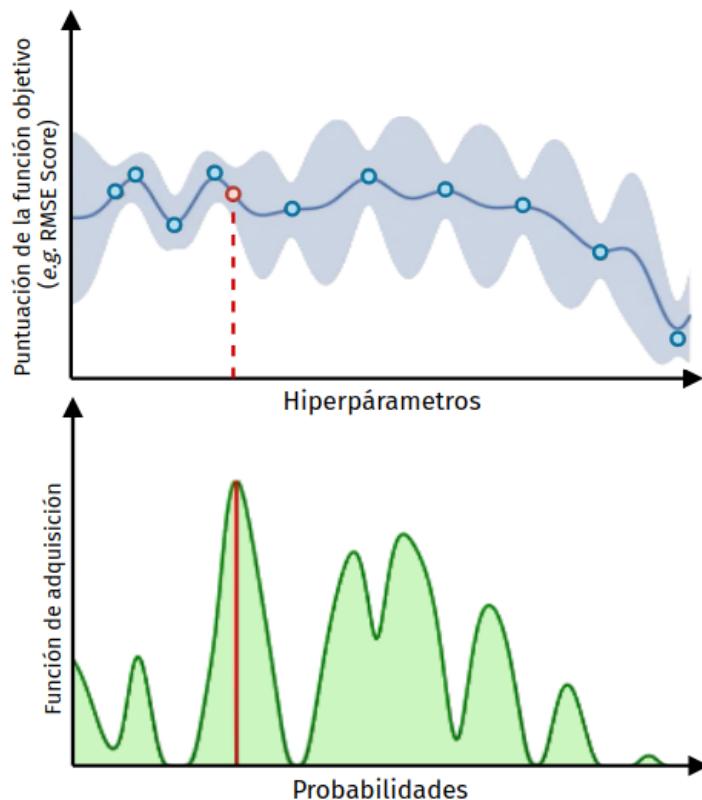


Figura 73: **Arriba:** modelo subrogado. **Abajo:** función de adquisición.

Luego de usar una función de adquisición para determinar el siguiente hiperparámetro, se obtiene la puntuación de la función objetiva real de este nuevo hiperparámetro. Dado que el modelo subrogado se entrena en $P(y|x)$, la adición de un nuevo punto de datos actualiza el modelo subrogado.

Ahora se deben repetir los pasos anteriores hasta que se alcance el tiempo máximo o la iteración máxima. Ahora posee una aproximación precisa

de la función objetivo real y puede encontrar fácilmente el mínimo global de las muestras evaluadas en el pasado para así completar la optimización bayesiana.

Finalmente vamos a desglosar el siguiente pseudo-código propuesto en "*Algorithms for Hyper-Parameter Optimization*" [30].

```

SMBO( $f, M_0, T, S$ )
1    $\mathcal{H} \leftarrow \emptyset,$ 
2   For  $t \leftarrow 1$  to  $T$ ,
3        $x^* \leftarrow \operatorname{argmin}_x S(x, M_{t-1}),$ 
4       Evaluate  $f(x^*)$ ,  $\triangleright$  Expensive step
5        $\mathcal{H} \leftarrow \mathcal{H} \cup (x^*, f(x^*)),$ 
6       Fit a new model  $M_t$  to  $\mathcal{H}$ .
7   return  $\mathcal{H}$ 
```

Figura 74: El pseudocódigo genérico de la optimización basada en modelos secuenciales.

SMBO significa optimización basada en modelos secuenciales, que es otro nombre de optimización bayesiana. Es "secuencial" porque los hiperparámetros se agregan para actualizar el modelo subrogado uno por uno; está "basado en modelos" porque se aproxima a la función objetivo real con un modelo subrogado que es más barato de evaluar.

Otras representaciones en el pseudocódigo:

- \mathcal{H} : historial de observación del par(puntuación, hiperparámetro).
- T : número máximo de iteraciones.
- f : función objetiva real (la puntuación de la métrica del modelo).
- M : modelo subrogado, que se actualiza cada vez que se agrega una nueva muestra.
- S : función de adquisición.
- x^* : el siguiente hiperparámetro elegido para evaluar.

La secuencia se desarrolla de la siguiente manera:

- Primero, iniciamos un modelo subrogado y una función de adquisición.
- Línea 3: luego, para cada iteración, encuentre el hiperparámetro x^* donde se encuentra el mínimo de la función de adquisición S . La función de adquisición es una función del modelo subrogado, lo que significa que se construye utilizando el modelo subrogado en lugar de la verdadera función objetivo real.
- Línea 4: Obtenga la puntuación de la función objetivo de x^* para ver cómo se desempeña realmente este punto

- Línea 5: Incluya el par (hiperparámetro x^* , puntuación de función objetivo real) en el historial de otras muestras.
- Línea 6: entrene el modelo subrogado utilizando el último historial de muestras.

- Repita hasta alcanzar el número máximo de iteraciones. Al final, se devuelve el historial de (hiperparámetro, puntuación de función objetiva real). Tenga en cuenta que el último registro no es necesariamente el mejor puntaje logrado. Tendría que ordenar la puntuación para encontrar el mejor hiperparámetro.

En la fig. 75 advertimos que tanto al inicio de la optimización bayesiana como de búsqueda aleatoria no hay mejoras. La optimización bayesiana necesita tiempo para la construcción de un buen modelo que conduzca a configuraciones de mejor rendimiento. Por tanto una vez que aumentamos los presupuestos, el modelo recopila cada vez más información sobre el espacio de búsqueda, lo que genera ventajas sobre la búsqueda aleatoria [31].

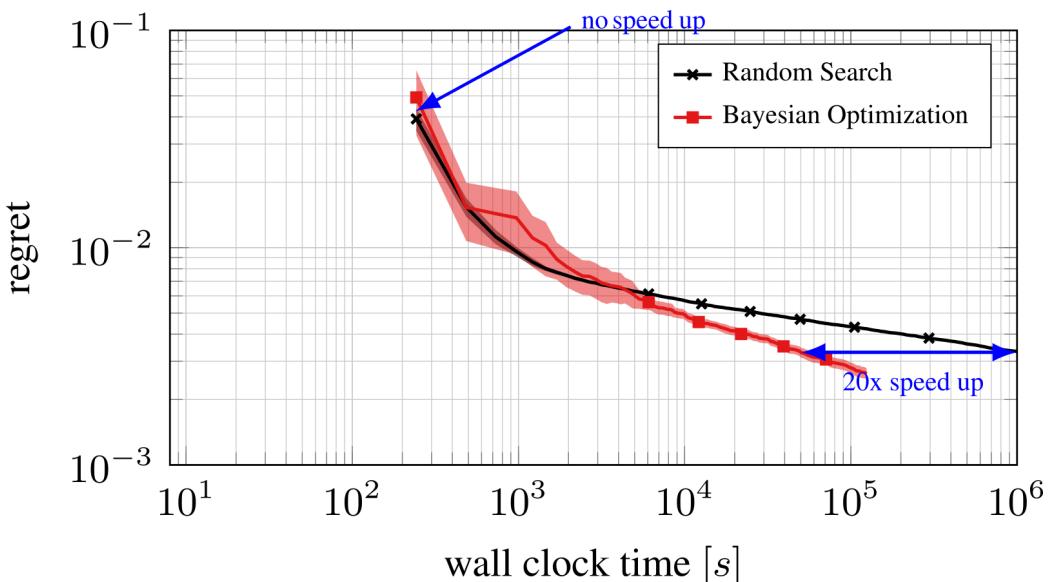


Figura 75: Rendimiento de Búsqueda Aleatoria *vs* Optimización Bayesiana. [31]

7.1.3 Reducciones sucesivas a la mitad

Definamos un par de conceptos que nos serán útiles para plasmar la idea. Al conjunto de valores para cada hiperparámetro se denomina configuración. Se entiende por presupuesto a la cantidad de recursos disponibles, ya sea para entrenar un modelo una determinada cantidad de épocas (*epochs*) o probar diversas configuraciones del mismo (*trials*).

En la práctica las configuraciones prometedoras tienden a obtener puntuaciones más altas en relación con las peores, incluso en las primeras etapas del proceso. A este hecho se aferran los algoritmos de sucesivas reducciones a la mitad *SuccessiveHalving*.

Y proceden de la siguiente forma:

1. Muestra aleatoriamente un conjunto de n configuraciones de hiperparámetros.
2. Evalua el desempeño de todas las configuraciones restantes actualmente.
3. Desecha la mitad inferior de las peores configuraciones de puntuación y duplica el presupuesto para el resto hasta que alcanza el presupuesto máximo.
4. Vuelva al paso 2 y repita hasta que quede una configuración.

La secuencia se ejemplifica en la fig. 76 donde cada línea paralela al eje de ordenadas es una iteración y cada línea de color corresponde a una configuración.

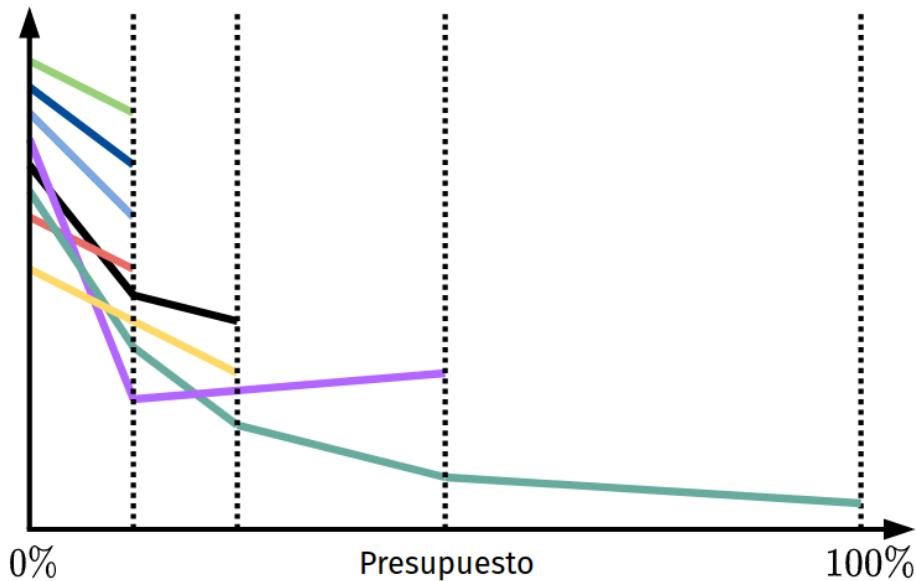


Figura 76: Secuencia de algoritmo de *SuccessiveHalving*.

En lugar de perder mucho tiempo de entrenamiento en configuraciones que no nos llevarán a ninguna parte, *SuccessiveHalving* los descarta lo antes posible. Por lo tanto, se puede asignar más tiempo de entrenamiento, es decir, recursos, a modelos más potencialmente valiosos.

Sin embargo, el *Successive Halving* sufre lo que se denomina el *trade-off* n vs B/n [32], dónde:

- B es el presupuesto total que se asignará para la búsqueda de hiperparámetros.
- n es el número de configuraciones que se explorarán.
- B/n es la cantidad promedio de recursos asignados a una configuración específica.

Para un presupuesto dado, no está claro si buscar muchas configuraciones (n grande) por un tiempo pequeño, o explorar pocas configuraciones pero asignándoles muchos recursos (B/n grande).

Para un problema dado, si las configuraciones de hiperparámetros se pueden discriminar rápidamente ya sea si:

- el conjunto de datos converge rápidamente.
- las malas configuraciones se revelan rápidamente.
- el espacio de búsqueda de hiperparámetros no se elige lo suficientemente bien como para que uno elegido al azar sea muy malo.

entonces n debe elegirse grande.

Por el contrario si:

- son lentos para diferenciar.
- el espacio de búsqueda es pequeño.
- se busca la mejor configuración con alta confianza.

entonces B/n debe ser grande (a expensas del número de configuraciones probadas).

Los inconvenientes de estas estrategias son los siguientes:

- Si n es grande, algunas buenas configuraciones que pueden tardar en converger al principio se eliminarán pronto.
- Si B/n es grande, entonces se asignarán muchos recursos a las malas configuraciones, aunque podrían haberse detenido antes.

El algoritmo que veremos a continuación elimina este dilema al considerar varios valores posibles de n para un B fijo, en esencia realizando una *Grid Search* sobre el valor factible de n .

Hyperband

El algoritmo de *Hyperband* se presenta en *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization* [33] y el enfoque se basa en una estrategia de detención temprana para algoritmos iterativos de algoritmos de aprendizaje automático.

El principio subyacente del procedimiento explota la intuición de que si una configuración de hiperparámetros está destinada a ser la mejor después de una gran cantidad de iteraciones, es más probable que funcione en la mitad superior de las configuraciones después de una pequeña cantidad de iteraciones. Es decir, incluso si el rendimiento después de un pequeño número de iteraciones es muy poco representativo del rendimiento absoluto de las configuraciones, su rendimiento relativo en comparación con muchas alternativas entrenadas con el mismo número de iteraciones aproximadamente se mantiene.

Hyperband es un método que tiene como objetivo encontrar los mejores hiperparámetros para un modelo y un presupuesto (*budget*) determinado.

Además *Hyperband* es una extensión del *Successive Halving*, y el problema con estos algoritmos es que frecuentemente no podemos saber cuál es la compensación correcta entre el número de pruebas y el número de épocas. En ciertos casos, algunas configuraciones de hiperparámetros pueden tardar más en converger, por lo que comenzar con muchas pruebas pero una pequeña cantidad de épocas no será lo ideal; en otros casos, la convergencia es rápida y la cantidad de pruebas es el cuello de botella. [34]

Ahí es donde entra en juego *Hyperband*, que es básicamente una *Grid Search* sobre la estrategia de asignación óptima. Entonces, en cada prueba individual, el conjunto de hiperparámetros se elige al azar.

Hyperband se basa en dos bucles `for` anidados. El bucle exterior itera de 0 a s_{max} y ejecuta un procedimiento de *Successive Halving* cada vez. El bucle interno también ejecuta *Successive Halving*, (lo denominamos `bracket`) repitiéndose s veces. Comienza con n modelos que se ejecutan con un presupuesto r , y en cada ciclo, el número de modelos se reduce en η mientras que el mismo factor aumenta el presupuesto. [35]

HYPERBAND algorithm for hyperparameter optimization.

```

input      :  $R, \eta$  (default  $\eta = 3$ )
initialization:  $s_{\max} = \lfloor \log_\eta(R) \rfloor, B = (s_{\max} + 1)R$ 
1 for  $s \in \{s_{\max}, s_{\max} - 1, \dots, 0\}$  do
2    $n = \lceil \frac{B}{R \cdot \eta^s} \rceil, r = R\eta^{-s}$ 
   // begin SUCCESSIVEHALVING with  $(n, r)$  inner loop
3    $T = \text{get\_hyperparameter\_configuration}(n)$ 
4   for  $i \in \{0, \dots, s\}$  do
5      $n_i = \lfloor n\eta^{-i} \rfloor$ 
6      $r_i = r\eta^i$ 
7      $L = \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$ 
8      $T = \text{top\_k}(T, L, \lfloor n_i/\eta \rfloor)$ 
9   end
10 end
11 return Configuration with the smallest intermediate loss seen so far.

```

Figura 77: Secuencia de algoritmo de *SuccessiveHalving*.

Dentro del algoritmo:

- η , el factor de poda, determina la cantidad de modelos eliminados en cada **bracket**. Por ejemplo, en el procedimiento de *Successive Halving* original donde $\eta = 2$, se podría comenzar con 64 modelos, mantener los mejores 32 modelos después del primer soporte, luego los mejores 16 modelos, y así sucesivamente.
- η , en azul, es el número de modelos considerados en cada bucle. Observe que la fórmula incluye una división y un operador de techo para obtener un valor entero. Siguiendo el ejemplo anterior, sería 64.
- n^i , en rojo, es el número de modelos utilizados en la iteración i . Observe que se redondea con un operador de piso.
- La cantidad sin nombre en verde es el número de modelos guardados de una iteración a la otra. Nuevamente, se redondea con un operador de piso.

Dado que la mayoría de estos valores están redondeados, esperamos que sean fracciones, no números enteros exactos. La fórmula de η está diseñada para consumir tanto presupuesto como sea posible en cada bucle de *Hyperband*.

En la fig. 78, puede ver que el algoritmo de *HyperBand* se ejecutará dividiendo a la mitad sucesivamente en 5 asignaciones de recursos. $s = 4$ comienza su primera ronda con 81 *trials*, ofreciendo a cada uno una única época y luego descarta de forma iterativa $\frac{2}{3}$ de los intentos hasta que queda uno y se entrena durante 81 épocas.

En $s = 0$, el algoritmo *Hyperband* básicamente está ejecutando una búsqueda aleatoria con 5 pruebas, cada uno entrenado en el número máximo de épocas.

i	$s = 4$		$s = 3$		$s = 2$		$s = 1$		$s = 0$	
	n_i	r_i								
0	81	1	27	3	9	9	6	27	5	81
1	27	3	9	9	3	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								

Figura 78: Cantidad de épocas y pruebas a medida que avanza el algoritmo.

En la práctica, *HyperBand* funciona muy bien para presupuestos pequeños a medianos y, por lo general, supera la búsqueda aleatoria y la optimización bayesiana con bastante facilidad en ese entorno. Sin embargo, su convergencia está limitada por su dependencia de configuraciones dibujadas al azar: con presupuestos más grandes, su ventaja sobre la Búsqueda Aleatoria disminuye. Un ejemplo de esto se da en la figura 79 a continuación. [31]

Dado que la Optimización Bayesiana se comporta de manera similar a la Búsqueda Aleatoria al principio, *Hyperband* exhibe la misma ventaja sobre éste para presupuestos pequeños a medianos. Para presupuestos más grandes, el modelo de Optimización Bayesiana normalmente guía la búsqueda a regiones de mejor rendimiento y supera a *Hyperband*.

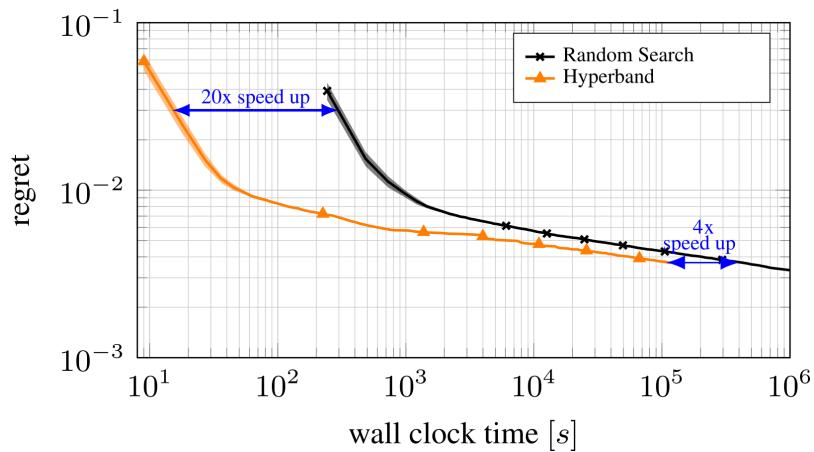


Figura 79: Comparación de Búsqueda Aleatoria *vs* *HyperBand*.

8 Series temporales

Una serie temporal o cronológica es una sucesión de datos medidos en determinados momentos y ordenados cronológicamente. [36]

8.1 Clasificación del modelo

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

8.1.1 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.

8.1.2 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.

8.1.3 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.

8.1.4 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).

8.1.5 Horizonte de pronóstico

El horizonte de pronóstico es el período de tiempo en el futuro para el cual 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.

8.1.6 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.

8.1.7 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.

8.2 Modelos disponibles

Se realizó una investigación de los modelos de *forecasting* disponibles de forma *open-source* 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.

Serán explorados los 3 más populares de Github.

8.2.1 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. [37]

- 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.

8.2.2 GluonTS

Gluon Time Series (**GluonTS**) es el kit de herramientas desarrollado por *AWSlabs* para el modelado probabilístico de series de tiempo, que se cen-

tra 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. [38]

8.2.3 `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: [39]

- **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.

8.2.4 Conclusión

Dada el carácter de investigación del proyecto integrador se optó por no utilizar estos modelos y crearlos desde *scratch* (terminología utilizada en el argot para describir un proyecto creado casi desde 0 a partir de un *stack* seleccionado de tecnologías).

9 Entropía de una serie temporal

La entropía aproximada (*Approximate Entropy*) y la entropía de muestra (*Sample Entropy*) son dos algoritmos para determinar la regularidad de series de datos en función de la existencia de patrones. A pesar de sus similitudes, las ideas teóricas detrás de esas técnicas son diferentes. [40]

9.1 Introducción

El objetivo de la entropía aproximada (ApEn) y la entropía muestral (**SampEn**) es estimar la aleatoriedad de una serie de datos sin ningún conocimiento previo sobre la fuente que genera el conjunto de datos. El objetivo perseguido por ApEn y **SampEn** es determinar con qué frecuencia se encuentran diferentes patrones de datos en el conjunto de datos. Para lograr ese objetivo, se requiere una medida matemática del nivel de aleatoriedad. Esta métrica se basa en el concepto de entropía de la teoría de la información, una magnitud que cuantifica la incertidumbre de una medida.

La conexión entre el contenido de la información y la aleatoriedad proporciona una forma de cuantificar el nivel de aleatoriedad de un conjunto de datos de una manera puramente matemática, sin asumir ningún modelo o hipótesis subyacente sobre el proceso que genera los datos.

9.2 Información y complejidad

9.2.1 Estadísticas de regularidad

En muchas situaciones, el uso de la palabra aleatorio contrasta con la idea matemática de la aleatoriedad de un número.

Chaitin [41] propuso el experimento de lanzar una moneda 20 veces y escribir un uno si sale cara y un cero si sale cruz. Desde una perspectiva probabilística, el experimento podría producir las siguientes dos series con la misma probabilidad (una sobre 2^{20}), por lo tanto, considerando ambas series como aleatorias :

A: 0101010101010101

B: 01101000110111100010

Si bien es cierto que las dos series se han producido aleatoriamente con la misma probabilidad y tienen los mismos valores de media y varianza, es fácil continuar con el patrón descrito en la serie A, pero tendríamos que trabajar más para poder predecir el siguiente número de la serie B. La serie

A se construye alternando ceros y unos, mientras que la serie B no tiene un patrón claro.

El análisis de aleatoriedad es, por tanto, una limitación esencial de los métodos de estadística de momento y no puede utilizarse para garantizar matemáticamente la aleatoriedad de una serie.

El origen de esta limitación radica en el hecho de que el enfoque probabilístico clásico de analizar los momentos de diferentes órdenes no analiza la aleatoriedad de una serie sino la aleatoriedad del proceso de generación de la serie. Una mayor frecuencia de un número en una serie, una asimetría de la distribución, o cualquier otra medida similar, son herramientas para estudiar el proceso de generación de los datos, y el orden de su generación no tiene ninguna influencia. En consecuencia, como afirmó *Chaitin*, el uso de esos métodos no puede probar la aleatoriedad de un número dado, simplemente porque la idea subyacente de probabilidad es analizar el método de generación.

En esencia, dado que la entropía cuantifica la cantidad de información, también mide el grado de aleatoriedad en el sistema. Sería deseable que una de las características a la hora de medir la aleatoriedad de una serie fuera establecer una jerarquía de grados de aleatoriedad, una cuantificación del nivel de aleatoriedad para que la pregunta no sea si una serie de datos es aleatoria o no, sino lo aleatorio que es. Habrá una situación con máxima aleatoriedad, totalmente impredecible, pero también habrá otras situaciones con diferentes grados de aleatoriedad por lo que deberíamos poder ordenar las series de datos según su complejidad. Una serie de datos simple (no compleja) tendría patrones obvios, como la serie A mencionada anteriormente, mientras que para una serie compleja esos patrones serían completamente inexistentes. Nos centraremos en analizar la aleatoriedad de la serie mediante el estudio de los patrones de números utilizando una formulación matemática fundamental sin asumir ningún modelo.

La idea subyacente es la siguiente. Imagínense los días en que se usaba el telégrafo, e imagine a dos investigadores, uno que lee las temperaturas en Nueva York y envía telegramas a su amigo en Chicago, un meteorólogo experto que quiere obtener datos del año anterior para su investigación. La teoría de la codificación se ocupa del contenido de información de los telegramas y la longitud mínima requerida para enviar el mensaje de un lugar a otro. Suponga que la persona en Nueva York escribe un 1 si la temperatura sube y un 0 si baja, escribiendo un número por hora y haciendo que la serie anual sea extremadamente larga. Si el clima se comportara como una serie sinusoidal perfecta, el telegrama consistiría en la serie de números: Los valores fueron 01....

Sin embargo, la persona en Nueva York puede simplificar el telegrama (y ahorrar dinero con él, recuerde que el costo dependía de la cantidad de palabras) diciéndole a su amigo: '**Los valores fueron 01 repetidos 4380 veces**'.

El punto esencial a tener en cuenta es que la información es la misma en los dos mensajes; sin embargo, la longitud del código es mínima en el segundo caso, lo que lo hace óptimo (y más económico). Por el contrario, suponga que el clima no se comporta de tan buena manera y que la secuencia de números no tiene sentido para la persona en Nueva York. No hay otra opción que enviar la lista completa de números en un telegrama de la forma: '**Los valores fueron 01101101000100111010101010010101011010101111010...**'.

Como no hay un patrón discernible en esos datos, no hay forma de reducir la longitud del telegrama, pagando así un costo más alto por ello. El punto esencial aquí es que **no podemos comprimir una secuencia de números totalmente aleatorios en un mensaje más corto**, lo que ofrece una manera de definir la **aleatoriedad total de una serie como un mensaje que requiere la máxima longitud para ser transmitido**. Una vez definida la aleatoriedad de esta forma, en función de su contenido de información y la longitud del código necesario para enviar el mensaje, podemos plantearnos cómo medirlo en diferentes grados.

Como hemos visto en el ejemplo anterior, la existencia de patrones dentro de una serie es el núcleo de la definición de aleatoriedad, por lo que es apropiado establecer una jerarquía de aleatoriedad en base a los diferentes patrones y sus repeticiones.

9.2.2 Teoría de la información

En esta sección definimos y resumimos aquellos conceptos de TI necesarios para comprender la lógica detrás de ApEn, enfocándonos en variables discretas.

Originalmente, la TI se diseñó como una teoría para analizar el proceso de envío de mensajes a través de un canal ruidoso y comprender cómo reconstruir el mensaje con una baja probabilidad de error. Como se mencionó anteriormente, una de las ideas detrás de TI es el hecho de que la información en un mensaje se puede medir cuantitativamente.

En nuestra era digital, los mensajes se componen de bits, pero no todos los bits son útiles: algunos de ellos son redundantes, algunos de ellos son errores, y así sucesivamente. Así que cuando comunicamos un mensaje, queremos tanta información útil como sea posible para pasar.

En la TI, transmitir un poco de información significa reducir la incertidumbre del receptor en un factor de 2, *e.g.*, digamos que el clima es completamente aleatorio, con una probabilidad de 50/50 de ser soleado o lluvioso todos los días (fig. 80). Si una estación meteorológica te dice que va a llover mañana, entonces en realidad han reducido tu incertidumbre en un factor de dos. Había dos opciones igualmente probables, ahora sólo hay una. [42]

Así que el canal del tiempo en realidad le envió un solo poco de información útil. Y esto es cierto sin importar cómo codificaron esta información. Si lo codificaron como una cadena, con 5 caracteres, cada uno codificado en 1 byte, entonces en realidad le enviaron un mensaje de 40 bits, pero sólo se comunicó 1 bit de información útil.

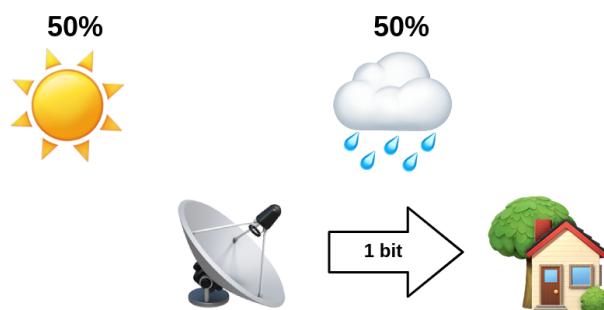


Figura 80: Representación gráfica de la situación propuesta.

Ahora supongamos que el clima tiene en realidad 8 estados posibles los cuales son definidos como los símbolos del alfabeto (dónde la longitud del alfabeto es S), todos igualmente probables (fig. 81, ignorar porcentajes). Ahora, cuando la estación meteorológica te da el tiempo de mañana, están dividiendo tu incertidumbre por un factor de 8, que es 2^3 . Así que se enviaron 3 bits de información útil. Es fácil encontrar el número de bits de información que en realidad se comunicaron calculando el logaritmo binario del factor de reducción de incertidumbre, *i.e.* $\log_2(8)$.

Pero, ¿qué pasa si las posibilidades no son igualmente probables? Digamos que el 75% tiene posibilidades de sol y el 25% de probabilidad de lluvia. Si la estación meteorológica te dice que va a llover mañana, entonces tu incertidumbre ha bajado en un factor de 4, *i.e.* $\log_2(4) = 2$ que serían 2 bits de información. La reducción de la incertidumbre es sólo la inversa de la probabilidad del evento, en este caso la inversa del 25% ($\frac{1}{4}$) es 4. Ahora $\log(\frac{1}{x}) = -\log(x)$, por lo que la ecuación para calcular el número de bits simplifica a $-\log_2(0.25)$.

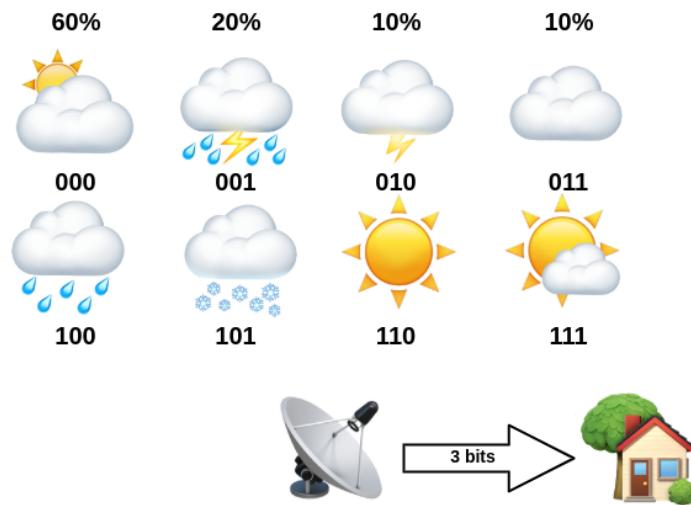


Figura 81: Representación gráfica de transmisión de 8 estados.

Por tanto la fórmula para medir la cantidad de información se define:

$$H = -p \log_2(p)$$

dónde:

- p : es la probabilidad de aparición de un determinado símbolo.

Ahora, si la estación meteorológica te dice que mañana estará soleado, entonces tu incertidumbre no ha bajado mucho. ¿Cuánta información vas a obtener de la estación meteorológica, en promedio? Bueno, hay un 75% de probabilidades de que sea soleado mañana, así que eso es lo que la estación meteorológica te diría y eso es 0.41 bits de información. Luego hay un 25% de probabilidades de que llueva, en cuyo caso la estación meteorológica te lo dirá, y esto te dará 2 bits de información. Así que en promedio, si calculamos $0.75 \cdot 0.41 + 0.25 \cdot 2 = 0.81$, *i.e.* obtenemos 0.81 bits de información de la estación meteorológica, todos los días. Así que lo que acabamos de calcular se llama **entropía**. Para finalizar hagamos el mismo cálculo para la fig. 81: $(0.6)(-\log_2(0.6)) + (0.2)(-\log_2(0.2)) + (0.1)(-\log_2(0.1)) - (0.1)(-\log_2(0.1)) = 1.57$, es decir que de los 3 bits enviados obtenemos el 52% de la información.

La entropía es una buena medida de lo inciertos que son los acontecimientos y de manera más formal la definimos como la medida de información de una sola variable aleatoria.

La idea de una variable aleatoria en este contexto se refiere a la interpretación probabilística: cada uno de los posibles resultados de la variable tiene una probabilidad definida de ocurrencia y el valor asociado con el resultado es aleatorio. Matemáticamente la definimos:

Si X puede tomar los valores $\{x_1, \dots, x_n\}$ y $p(x)$ es la probabilidad asociada con esos valores dado $x \in X$, la entropía se define como:

$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x)$$

Esperemos que la ecuación de la entropía ahora tenga sentido completo: mide la cantidad promedio de información obtenida cuando se consulta el clima cada día, o de forma más general la cantidad promedio de información obtenida de una muestra extraída de una p de distribución de probabilidad dada. Le dice lo impredecible que es esa distribución de probabilidad. Si vivís en medio de un desierto donde está soleado todos los días, en promedio no se obtendrá mucha información de la estación meteorológica y la entropía estará cerca de cero. Por el contrario, si el clima varía mucho, la entropía será mucho mayor.

Es importante comprender que la entropía se basa en la probabilidad de aparición de cada símbolo, *i.e.*, la entropía no es una función de los valores de la serie en sí, sino una función de sus probabilidades.

En general, el investigador ignora la distribución de probabilidad real de cada símbolo y usa la frecuencia de ocurrencia en la serie de datos como una aproximación de la probabilidad real. Es una buena idea interpretar los conceptos fundamentales de TI antes de continuar:

- La **información** es la disminución de la ambigüedad con respecto a un fenómeno, un incremento de nuestro conocimiento cuando observamos un evento específico. Las apariciones de una variable aleatoria con baja probabilidad son más sorprendentes y agregan más información. Imagine una serie de datos cuyos valores cambian tan poco que el siguiente valor es prácticamente el mismo que el anterior. Si hay un cambio brusco y el valor aumenta drásticamente (esta variable aleatoria que tiene un valor alto como resultado es un evento con baja probabilidad), hay una gran ganancia de información.

Matemáticamente, la información es proporcional a la inversa de la función de probabilidad del evento, $\log_2 \frac{1}{p_1}$. Por tanto, la **información** es un evento que induce a los agentes racionales a cambiar sus creencias y expectativas sobre una situación particular, con las restricciones de la nueva información.

En el ejemplo anterior, los eventos muy probables (una serie plana de datos) brindan poca información a los agentes, mientras que los resultados de baja probabilidad (un aumento repentino en el valor) brindan más información que cambiará sus creencias sobre el proceso detrás de los datos.

- La **incertidumbre** es algo posible pero desconocido. Es posible que tirando un dado consigamos el número cinco, pero tirar un dado no implica obtener el número cinco. Tenemos incertidumbre sobre el número cinco, pero no lo tendríamos para el número 214 en un dado de seis caras porque es un evento imposible.

En el contexto de las TI, la incertidumbre es la cantidad de información que esperamos que revele un experimento. Si la distribución de probabilidades es uniforme, como en el caso de los dados ($p_i = \frac{1}{n}$), existe una incertidumbre máxima y el resultado es más impredecible.

Por otro lado, si la distribución de probabilidad está más concentrada (un dado con una alta probabilidad de obtener un seis), entonces el resultado es más predecible y hay una menor incertidumbre. Respecto al concepto anterior, **baja incertidumbre significa más información**.

- La **entropía** (H_x) es una medida de la cantidad de incertidumbre asociada con una variable X cuando solo se conoce su distribución.

Es una medida de la cantidad de información que esperamos aprender de un resultado (observaciones) y podemos entenderla como una medida de uniformidad. **Cuanto más uniforme sea la distribución, mayor será la incertidumbre y mayor la entropía**. Esta definición es importante porque la entropía se utilizará en última instancia para medir la aleatoriedad de una serie. La entropía H es una función de la distribución de probabilidad $\{p_1, p_2, \dots\}$ y no una función de los valores de la variable en la serie particular $\{x_1, x_2, \dots\}$.

La teoría de la información y el concepto de entropía proporcionan una forma matemática de determinar la información contenida en un mensaje. Esta información es independiente del conocimiento del receptor y está determinada por la fuente de información a través de la función de probabilidad de cada uno de los símbolos.

El resultado de un experimento contiene la cantidad de información $H(p)$, o dicho de otra manera, $H(p)$ es una medida de la incertidumbre sobre un evento específico antes de observarlo.

Como podemos ver en la fig. 82, la entropía depende de la cantidad de aleatoriedad representada por p , donde la situación $p = \frac{1}{2}$ es aleatoria desde una perspectiva probabilística y da una entropía máxima $H = 1$.

Con la definición de Shannon, eventos con alto o baja probabilidad no contribuyen mucho al valor de la medida, como se muestra en el gráfico para valores de $p = 0$ o $p = 1$.

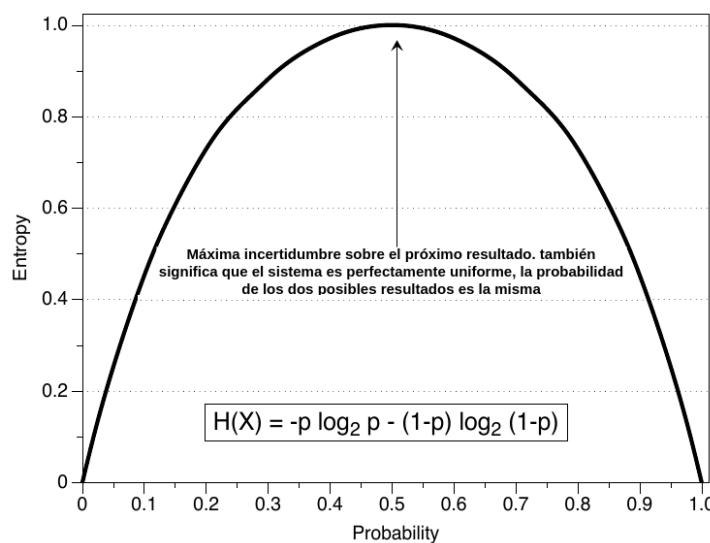


Figura 82: Entropía de una variable X la cual toma un valor uno con una probabilidad p y el valor cero con una probabilidad $(1 - p)$. [40]

En el apartado anterior definimos la entropía como una medida para cuantificar la información de un mensaje que es válida de forma general, independientemente del proceso que lo genera.

Tipos de procesos

- **Procesos estocásticos:** sucesión de variables aleatorias relacionadas entre sí por una regla de evolución, representando un sistema que evoluciona con el tiempo.
- **Procesos deterministas:** un sistema en el que no hay aleatoriedad en la determinación de los siguientes estados del sistema. Si conocemos con precisión el estado inicial del sistema podemos hacer una predicción exacta de cualquier estado futuro del sistema.

La cuestión de evaluar la aleatoriedad de una serie de datos es, por tanto, una cuestión de caracterizar el sistema como estocástico o determinista. Más precisamente, en qué medida los datos se comportan como estocásticos y cuánto determinismo existe; estaríamos hablando de grados de aleatoriedad.

Si no hay un proceso determinista subyacente al analizar una serie particular de datos, implicará una verificación del movimiento aleatorio más estricto. Sin embargo, si hay ciclos, tendencias y patrones, entonces la serie de datos no sería totalmente aleatoria. Será necesario cuantificar tanto la situación totalmente aleatoria como los diferentes grados de aleatoriedad de forma inequívoca. En el caso particular de los procesos estocásticos, el interés no radica en medir la entropía total sino en medir la entropía del proceso, es decir, cómo cambia la entropía de la serie a lo largo del tiempo.

Si pensamos en una serie de datos, nuestro interés sería medir el aumento de información que obtenemos cuando conocemos un nuevo valor. Esta medida está determinada por la tasa de entropía del proceso estocástico X_t con $x \in K$, donde K es el alfabeto (cuando existe el límite):

$$H(X) = \lim_{T \rightarrow \infty} \frac{1}{T} H(X_1, X_2, \dots, X_T)$$

Por tanto, la medición de la tasa de entropía ofrece una forma de medir hasta qué punto un proceso es estocástico y, en última instancia, cuantificará la aleatoriedad de una serie. Podemos relacionar este concepto con la discusión anterior sobre el telégrafo. La tasa de entropía es el número esperado de bits (medida de entropía cuando usamos la base 2 para el logaritmo) por símbolo necesarios para describir el proceso. Si el proceso es completamente estocástico, entonces el número de bits es máximo (la tasa de entropía es máxima). Comparando la tasa de entropía máxima de nuestro sistema con el máximo número de bits dado un alfabeto, podemos determinar el grado de aleatoriedad de una serie sin asumir nada sobre el proceso de generación de datos. Así que tanto en sistemas estocásticos como deterministas, la tasa de entropía puede estar relacionada con la aleatoriedad del sistema.

9.3 Medición de la aleatoriedad

9.3.1 Entropía aproximada

La formulación para el análisis de la complejidad de un sistema fue desarrollada para analizar sistemas caóticos, y los intentos de utilizar esas técnicas para series temporales limitadas, ruidosas y estocásticamente derivadas no tuvieron mucho éxito.

El número de puntos necesarios para caracterizar correctamente un sistema dinámico está en el rango $[10^d, 30^d]$, donde d es la dimensión del sistema. Por lo tanto, se requiere una gran cantidad de datos para caracterizar completamente incluso los sistemas de baja dimensión.

ApEn es un parámetro que mide la correlación, persistencia o regularidad en el sentido de que los valores bajos de **ApEn** reflejan que el sistema es muy persistente, repetitivo y predictivo, con patrones aparentes que se repiten a lo largo de la serie, mientras que los valores altos significan independencia entre los datos, un número bajo de patrones repetidos y aleatoriedad. Esta definición coincide con la intuición de que los sistemas que tienen una probabilidad más aleatoria tienen mayor entropía, como se ve en las secciones anteriores, siendo sistemas estocásticos puros aquellos que tenían una tasa de entropía más alta (número máximo de bits dado un alfabeto).

En los sistemas binarios, el valor máximo de ApEn sería el registro 2, y un valor inferior al indicado que la serie en análisis contiene patrones que se repiten y, por lo tanto, no es totalmente aleatorio.

ApEn mide la probabilidad logarítmica de que los patrones cercanos permanezcan cerca en la siguiente comparación incremental. La idea principal detrás del desarrollo fue que no es un algoritmo para determinar completamente la dinámica de un sistema, más bien es adecuado para clasificar sistemas y estudiar la evolución de su complejidad: por consiguiente concluimos que no es necesario reconstruir completamente la dinámica del sistema para clasificarlo.

En el algoritmo ApEn, debemos seleccionar un par (m, r) como parámetros de entrada. Estadísticamente, m sería el equivalente a dividir el espacio de los estados en celdas de ancho r , para estimar las probabilidades condicionales del orden m -ésimo. En la interpretación que se indica a continuación, m es la longitud de la ventana de las diferentes comparaciones vectoriales, y r es un filtro de ruido de facto (superposición de ruido mucho menor en magnitud que r apenas afecta al cálculo).

ApEn no intenta reconstruir toda la dinámica, sino que sólo discrimina de una manera estadísticamente válida, y a cambio, podemos deshacernos de los límites $m \rightarrow \infty$ y $r \rightarrow 0$, haciendo que el enfoque sea adecuado para el análisis de datos. Veremos cómo elegir los valores de m y r para calcular ApEn, pero es apropiado comprender cómo afectan esos cambios al sistema.

Si tenemos m más alto y r más pequeño describiremos detalles de parámetros más nítidos (probabilísticos). Sin embargo, al tratar con procesos estocásticos, el análisis de probabilidades condicionales hace que grandes valores de m o valores mínimos de r produzcan estimaciones estadísticamente deficientes.

Formalmente, el algoritmo para determinar la entropía aproximada de una secuencia es el siguiente:

Dada una secuencia de números $u = \{u(1), u(2), \dots, u(N)\}$ de longitud N , un entero no negativo m , con $m \leq N$ y un número real positivo r , definimos los bloques $x(i) = \{u(i), u(i+1), \dots, u(i+m-1)\}$ y $x(j) = \{u(j), u(j+1), \dots, u(j+m-1)\}$, y calcular la distancia entre ellos como $d[x(i), x(j)] = \max_{k=1,2,\dots,m}(|u(i+k-1) - u(j+k-1)|)$.

Luego calculamos el valor $C_i^m(r) = (\text{número de } j \leq N-m+1 \text{ tal que } d[x(i), x(j)] \leq r)/(N-m+1)$. El numerador de C_i^m cuenta, dentro de la resolución r , el número de bloques de valores consecutivos de longitud m que son similares a un bloque determinado.

Calculando:

$$\phi^m(r) = \frac{1}{N-m+1} \sum_{i=1}^{N-m+1} \log C_i^m(r) \quad (30)$$

Podemos definir $\text{ApEn}(m, r, N)(u) = \phi^m(r) - \phi^{m+1}(r)$, con $m \leq 1$ y $\text{ApEn}(0, r, N)(u) = \phi^1(r)$.

$\text{ApEn}(m, r, N)(u)$ mide la frecuencia logarítmica con la que los bloques de longitud m que están unidos permanecen juntos para la siguiente posición, o se colocan de manera diferente.

El valor negativo de ApEn se define como:

$$-\text{ApEn}(m, r, N)(u) = \phi^{m+1}(r) - \phi^m(r) \quad (31)$$

$\text{ApEn}(m, r, N)$ es el estimador estadístico del parámetro $\text{ApEn}(m, r)$:

$$\text{ApEn} = \lim_{N \rightarrow \infty} [\phi^m(r) - \phi^{m+1}(r)] \quad (32)$$

$\text{ApEn}(m, r, N)$ es una familia de estadísticas y las comparaciones entre sistemas están destinadas con valores fijos de m y r , y, si es posible, con el mismo número de observaciones N debido al sesgo que mencionaremos más adelante. ApEn mide la probabilidad de que las ejecuciones de patrones que están cerca para las observaciones m permanezcan cerca en las siguientes comparaciones incrementales. Una mayor probabilidad de permanecer cerca, lo que implica regularidad, produce valores ApEn más pequeños y, por el contrario valores más altos.

Se explicará el algoritmo a través de un ejemplo. Supongamos:

$$u = \{85, 80, 89, 85, 80, 89, \dots, 85, 80, 89\}$$

Con $N = 51$, una serie numérica claramente periódica y predecible. Seleccionamos $m = 2$ y $r = 3$.

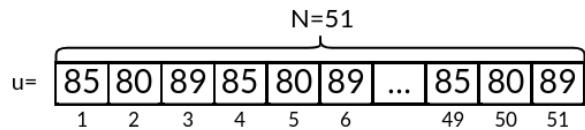


Figura 83: Serie numérica utilizada como ejemplo.

Definiremos pequeños bloques de longitud m , a los que nombraremos:

$$x(1), x(2), x(3), x(4), \dots$$

Donde $x(1) = [u(1), u(2)]$, $x(2) = [u(2), u(3)]$ y de esta forma definiremos los sucesivos bloques como visualizamos en la figura 84.

$$\begin{array}{ll} X(1) = \boxed{\begin{matrix} 85 & 80 \\ u(1) & u(2) \end{matrix}} & X(2) = \boxed{\begin{matrix} 80 & 89 \\ u(2) & u(3) \end{matrix}} \\ X(3) = \boxed{\begin{matrix} 89 & 85 \\ u(3) & u(4) \end{matrix}} & X(4) = \boxed{\begin{matrix} 85 & 80 \\ u(4) & u(5) \end{matrix}} \end{array}$$

Figura 84: Bloques de longitud m .

El siguiente paso a realizar en el algoritmo es calcular:

$$d[x(1), x(2)] = \max_a |u(a) - u^*(a)| = \max_a \{|u(1) - u(2)|, |u(2) - u(3)|\}$$

En este paso hay que tener cuidado ya que tenemos que calcular el valor absoluto de la resta cruzada de ambos bloques, esto queda claro en la figura 85. Una vez que obtenemos ambos valores nos quedamos que quedar con el mayor (resultado en rojo), que tendrá que ser comparado con r . En caso que sea menor, será un MATCH pero en este caso fue NO MATCH.

$$\left| \begin{array}{c} u(1) \\ \boxed{85} \\ u(2) \end{array} \right. - \left| \begin{array}{c} u(2) \\ \boxed{80} \\ u(3) \end{array} \right. \right| = \boxed{5} \quad \left| \begin{array}{c} u(1) \\ \boxed{85} \\ u(2) \end{array} \right. - \left| \begin{array}{c} u(3) \\ \boxed{89} \\ u(4) \end{array} \right. \right| = \boxed{9} < \boxed{3} \Rightarrow \text{NO MATCH}$$

Figura 85: Módulo de la resta entre bloques.

De forma similar calculamos:

$$d[x(1), x(1)] = \max\{[u(1)-u(1), u(2)-u(2)]\} = \max\{0, 0\} = 0 < r \Rightarrow \text{MATCH}$$

$$d[x(1), x(3)] = \max\{[u(1)-u(3), u(2)-u(4)]\} = \max\{4, 5\} = 5 > r \Rightarrow \text{NO MATCH}$$

$$d[x(1), x(4)] = \max\{[u(1)-u(4), u(2)-u(5)]\} = \max\{0, 0\} = 0 < r \Rightarrow \text{MATCH}$$

Nuestro objetivo es contar todos los MATCH para $x(j)/1 \leq j \leq N - m + 1$, en nuestro caso $N - m + 1 = 50$. Para el caso de $j = 1$, el número total de MATCH es 17 sobre un total de 50 iteraciones. Recordemos que la notación indica $C_i^m(r)$:

$$\begin{aligned} C_1^2(3) &= \frac{17}{50} \\ C_2^2(3) &= \frac{17}{50} \\ C_3^2(3) &= \frac{16}{50} \\ C_4^2(3) &= \frac{17}{50} \\ &\vdots \end{aligned}$$

Ahora procedemos a realizar la sumatoria de los logaritmos de los coeficientes calculados anteriormente como se indica en la ecuación 30:

$$\phi^2(3) = \frac{1}{50} \sum_{i=1}^{50} \log(C_i^2(3)) \approx -1.0982$$

Sin embargo todavía es necesario realizar estos cálculos para $m = 3$ ya que como se indica en la ecuación 31 necesitamos $\phi^{m+1}(r)$.

Repetimos la operación de módulo de la resta entre bloques, solo que ahora debemos decidir el mayor entre 3 resultados y comparar ese resultado con r para ver si es MATCH. Se computa a continuación:

$$\begin{aligned} C_1^3(3) &= \frac{17}{49} \\ C_2^3(3) &= \frac{16}{49} \\ C_3^3(3) &= \frac{16}{49} \\ C_4^3(3) &= \frac{17}{49} \\ &\vdots \end{aligned}$$

Entonces:

$$\phi^3(3) = \frac{1}{49} \sum_{i=1}^{49} \log(C_i^3(3)) \approx -1.0982$$

Finalmente:

$$\text{ApEn} = \phi^2(3) - \phi^3(3) \approx 0.000010997$$

Este valor es muy pequeño, lo que implica que la secuencia es regular y predecible, lo cual es consistente con la observación.

Propiedades de ApEn

- Es independiente de cualquier modelo, lo que la hace adecuada para el análisis de series de datos sin tener en cuenta nada más que los datos.
- Es muy estable a grandes e infrecuentes valores atípicos (*outliers*). Los valores atípicos grandes son eventos de baja probabilidad, que hacen una pequeña contribución a la medida general. Estos acontecimientos, sin embargo, son críticos en las estadísticas de momento, ya que las diferencias entre los valores y la media se cuantifican. La variabilidad puede detectar desviaciones del valor medio, pero no se preocupa por la regularidad de los datos. El mejor análisis de una serie de datos sería el uso combinado de estadísticas de regularidad y estadísticas de momentos. En este sentido, se ha demostrado que ApEn distingue lo normal de los datos anormales en los casos en que los enfoques estadísticos de momento no mostraron diferencias significativas.

- Debido a la construcción de ApEn y sus cimientos en TI, su valor no es negativo. Es finito para procesos estocásticos y para procesos deterministas con ruido.
- Tiene un valor de 0 para series perfectamente regulares.
- Toma el valor $\ln 2$ para series binarias totalmente aleatorias y, en general, el valor $\ln k$ para alfabetos de k símbolos.
- No se altera mediante traducciones o escalado aplicado uniformemente a todos los términos.
- La no linealidad provoca un mayor valor.
- Valores recomendados:
 - m debe ser bajo, $m = 2$ o 3 son opciones típicas.
 - r debe ser lo suficientemente grande como para tener un número suficiente de secuencias de vectores x a una distancia r de la mayoría de los vectores especificados, para garantizar estimaciones razonables de probabilidades condicionales. Los valores recomendados están generalmente en el rango de desviación estándar de 0.1 a 0.25 de la serie de datos en análisis.
- $\text{ApEn}(m, r)$ crece con una disminución de r como $\log(2r)$, exhibiendo una variación infinita con r , lo que implica una gran variación en el valor de la estadística $\text{ApEn}(m, r, N)$ con r .
- El número de datos necesarios para discriminar entre sistemas está en el rango de 10^m a 30^m , como en el caso de la teoría del caos, pero ya que m es generalmente un valor bajo, incluso una serie con un pequeño número de datos como $N = 100$ es adecuado para el análisis.
- Los sistemas con una $\text{SNR} < 3$ (relación señal-ruido), es decir, situaciones en las que el ruido es sustancial, comprometerían la validez de los cálculos de ApEn.
- La mayor utilidad surge cuando los medios y las desviaciones estándar de los sistemas muestran pocos cambios con la evolución del sistema.
- Para comparar diferentes series de datos, se recomienda normalizar estas series con respecto a su desviación estándar antes de la comparación.
- El algoritmo hace uso del vector de datos $\{x_1, x_2, \dots, x_T\}$ en lugar de utilizar las probabilidades asociadas con la aparición de cada resultado $\{p_1, p_2, \dots, p_T\}$. ApEn es directamente aplicable sin saber ni asumir nada sobre el conjunto de datos o saber nada sobre el proceso que genera los valores.
- Requiere mediciones igualmente espaciadas a lo largo del tiempo.

9.3.2 Entropía de muestra

En la práctica, ApEn tiene dos implicaciones importantes.

1. La coherencia relativa no está garantizada, y dependiendo del valor de m son los resultados puede ser diferente.
2. El valor de ApEn depende de la longitud de la serie de datos

Por ello se definió SampEn, una estadística que no tiene autoconteo. SampEn(m, r, N) es el valor negativo del logaritmo de la probabilidad condicional de que dos secuencias similares de puntos m permanezcan similares en el siguiente punto $m+1$, contando cada vector sobre todos los demás vectores excepto en sí mismo. Implica que SampEn mantiene la consistencia relativa y también es principalmente independiente de la longitud de la serie.

Según los autores de SampEn, la eliminación del autoconteo está justificada dado que "la entropía se concibe como una medida de la tasa de producción de información, y en este contexto comparar datos consigo mismos no tiene sentido". SampEn no utiliza el mismo enfoque de patrón que ApEn para determinar su valor, utilizando en su lugar toda la serie juntas, lo que requiere solo que un vector de plantilla encuentre una coincidencia de longitud $m+1$ que se va a definir. Contrasta con ApEn, donde cada vector de plantilla tiene que encontrar una coincidencia que definir.

$B_i^m(r) = \frac{1}{N-m-1} \times [\text{número de vectores } x_m(j) \text{ a la distancia } r \text{ de } x_m(i) \text{ sin permitir el autoconteo, donde } j = 1, N-m]$

$B_i^m(r) = \frac{1}{N-m-1} \sum_{j=1, j \neq i}^{N-m} [\text{número de veces que } d[|x_m(j) - x_m(i)|] < r]$
y sumando todos los vectores de plantilla:

$B^m(r) = \frac{1}{N-m} \sum_{i=1}^{N-m} B_i^m(r) = \frac{1}{N-m-1} \frac{1}{N-m} \sum_{i=1}^{N-m} \sum_{j=1, j \neq i}^{N-m} [\text{número de veces que } d[|x_m(j) - x_m(i)| < r]]$

De la misma manera, definimos el número total de coincidencias calculando para cada vector modelo:

$A_i^m(r) = \frac{1}{N-m-1} \times [\text{número de vectores } x_{m+1}(j) \text{ a la distancia } r \text{ de } x_{m+1}(i) \text{ sin permitir el autoconteo, donde } j = 1, N-m]$

y agregándolas como:

$A^m(r) = \frac{1}{N-m} \sum_{i=1}^{N-m} A_i^m(r) = \frac{1}{N-m-1} \frac{1}{N-m} \sum_{i=1}^{N-m} \sum_{j=1, j \neq i}^{N-m} [\text{número de veces que } d[|x_{m+1}(j) - x_{m+1}(i)| < r]]$

Por lo tanto, $B^m(r)$ es la probabilidad de que dos secuencias sean similares para los puntos m (posibles), mientras que $A^m(r)$ es la probabilidad de que dos secuencias sean similares para $m+1$ puntos (coincidencias). Puesto que el número de coincidencias es siempre menor o igual que el número de vectores posibles, la relación $A^m(r)/B^m(r)$ es una probabilidad condicional menor que la unidad.

El parámetro **entropía de muestra** se define como $SampEn(m, r) = \lim_{N \rightarrow \infty} \{ \log[A^m(r)/B^m(r)] \}$, valor que se estima a partir de la estadística $SampEn(m, r, N) = \log[A^m(r)/B^m(r)]$.

Si tomamos como ejemplo la serie utilizada para demostrar ApEn con $m = 2$ y $r = 3$:

$$u = \{85, 80, 89, 85, 80, 89, \dots, 85, 80, 89\}$$

La manera de computar si es MATCH o no es igual a ApEn, pero hay que tener en cuenta que no se permite el autoconteo, además considerar que debido al cálculo final de $\frac{B}{A}$ serán omitidas todas las multiplicaciones por constantes ya que se cancelan al dividir.

Se calcula $B_i^m(r)$:

$$\begin{aligned} B_1^2(3) &= 15 \\ B_2^2(3) &= 14 \\ B_3^2(3) &= 14 \\ B_4^2(3) &= 15 \\ &\vdots \end{aligned}$$

Y en el caso de $A_i^m(r)$:

$$\begin{aligned} A_1^2(3) &= 15 \\ A_2^2(3) &= 15 \\ A_3^2(3) &= 16 \\ A_4^2(3) &= 15 \\ &\vdots \end{aligned}$$

En el caso de SampEn se debe calcular B :

$$B^2(3) = \sum_{i=1}^{49} B_i^2(3) = 768$$

Luego computamos A :

$$A^2(3) = \sum_{i=1}^{49} A_i^2(3) = 752$$

Para finalizar calculamos:

$$\text{SampEn} = \log(B/A) = \log(752/768) = 0.021053$$

9.3.3 Diferencias entre ApEn y SampEn:

- SampEn no permite el autoconteo ($j \neq i$) mientras que ApEn sí.
- La suma de todos los vectores de plantilla está dentro del logaritmo en SampEn y fuera en ApEn. Implica que SampEn considera la serie completa y si una plantilla encuentra una coincidencia, SampEn ya está definido, mientras que ApEn necesita una coincidencia para cada plantilla.
- A priori, parece que el uso de SampEn elimina muchos de los problemas asociados con ApEn, siendo, según sus autores, útil para cuantificar la regularidad en un sistema más eficazmente.

10 Desarrollo

10.1 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. Estos defectos serán entrada del modelo que se creará con el objetivo de pronosticar la cantidad de defectos graves en un futuro a corto plazo.

10.1.1 Organización de la usina

La fábrica está compuesta 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)
- División de abastecimiento de piezas (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, y CALI se encarga de verificar los defectos con mayor recurrencia de los departamentos. La figura 86 nos esclarece esta idea.

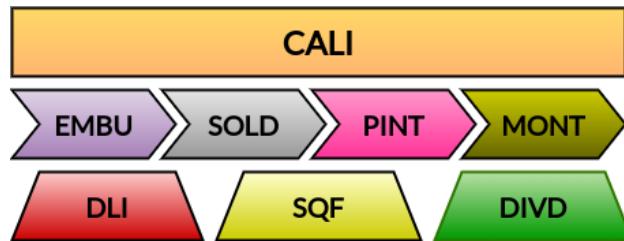


Figura 86: Diagrama de departamentos de la fábrica.

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.

10.1.2 Nomenclatura de defectos

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

$$\text{DEF} = \text{ELE} + \text{INC} + \text{LOC}$$

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.

10.1.3 Gravedad

Además los defectos se categorizan según la gravedad (GVD) del mismo en:

- V1: graves.
- V2: leves.
- V3: imperceptibles (para nuestro análisis serán descartados).

10.1.4 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.

10.1.5 Puntos de reconocimiento de defectos

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 en segundo para 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 encontrados en los puntos de reconocimiento 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, prueba de manejo, entre otros con objeto de encontrar defectos que en línea no sería posible.
- **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 un identificador único del vehículo.

10.2 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 87.



Figura 87: Procesamiento de datos segmentado en etapas.

En el campo de ciencia de datos la comunidad ha consensuado de facto que el estándar es utilizar como entorno de trabajo **Jupyter-Notebook**, que soporta Python como lenguaje de programación. Existen otros lenguajes reconocidos como R pero poseen mayor difusión en comunidades académicas.

Las librerías que serán parte del proyecto son las siguientes:

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

10.2.1 Recolección de datos

Etapa de búsqueda y recolección de los datos que alimentarán el modelo. En nuestro caso se dividió en recolectar en la organización los datos de defectos, que fueron obtenidos desde 4 auditorias diferentes que fueron presentadas en 10.1.5.

10.2.2 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.

10.2.3 Preprocesamiento de datos

En este punto es donde debemos prestar atención a los detalles al inspeccionar nuestro *dataset*.

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 nombrarlos.

Posteriormente fue necesaria la detección de incongruencias, advirtiendo 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*.

Una vez realizadas estas operaciones *dataset* se encuentra en condiciones para su análisis.

10.2.4 Análisis y visualización de datos

Para obtener una perspectiva más amplia del problema a tratar es conveniente analizar los datos disponibles. El *dataset* se compone de 229934 filas que cada una representa un defecto y 17 columnas.

Algunas estadísticas *grossos modo*, que podemos obtener:

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

Si verificamos el origen de los datos nos percataremos que casi la totalidad de ellos provienen de DPU, disputándose el pequeño margen restante PESD, CAPS y SAVES.

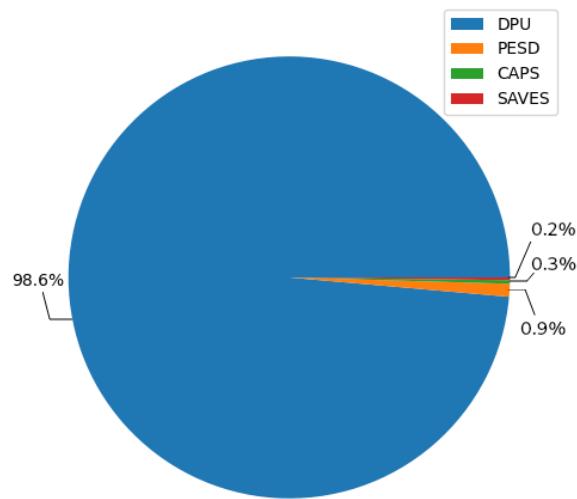


Figura 88: Distribución de los defectos según AUDI.

No obstante, no se debe perder el propósito el cual es el pronostico de la cantidad de defectos graves que ocurrirán en función a todos los defectos registrados con anterioridad. En la fig. 90 observamos que la gran mayoría de defectos registrados son V2 lo que podría ser útil para nuestro modelo.

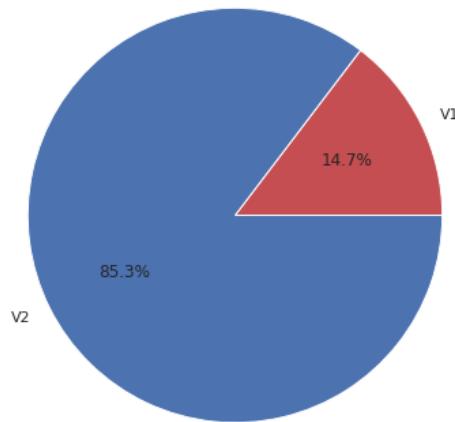


Figura 89: Distribución de los defectos según gravedad.

La fig. 90 nos muestra que los defectos principalmente ocurren en 3: PINT, SOLD y MONT, un poco más atrás en la participación está SQF.

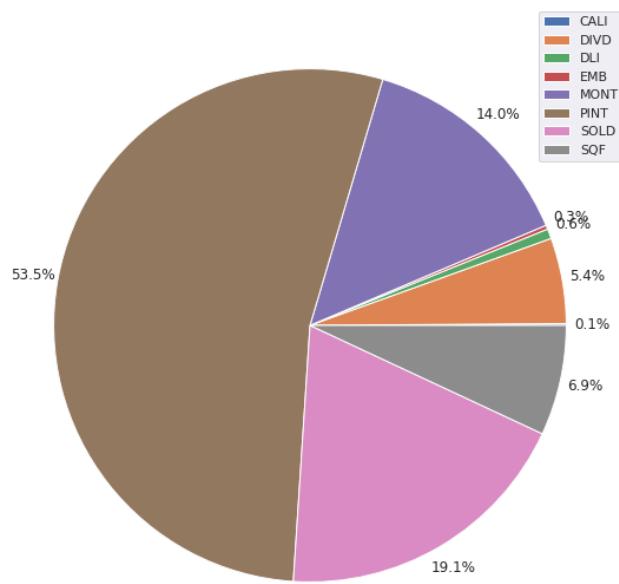


Figura 90: Distribución de los defectos por DPTO.

La fig. 91 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.

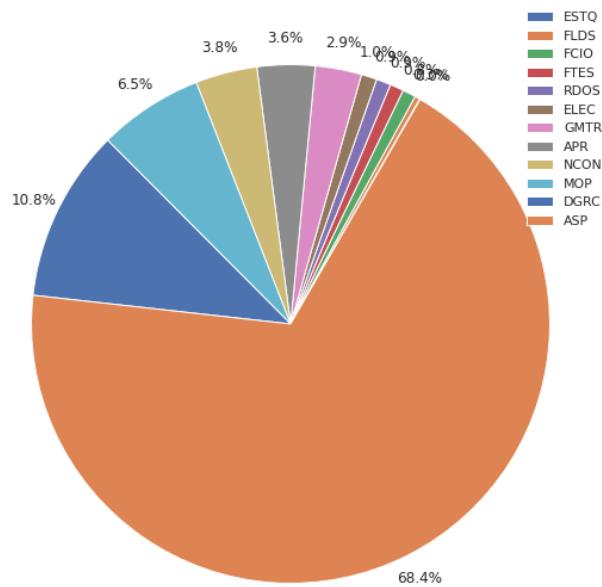


Figura 91: Distribución de los defectos por TIPO.

Mediante un mapa de calor (fig. 92) se cruzaron las características TIPO y DPTO. Advertimos una cierta correlación entre algunos tipos de defectos y los departamentos, a tal punto que en algunos departamentos ni siquiera existen determinados tipos de defectos. Sería beneficioso que el modelo pueda aprender esta característica para maximizar sus posibilidades.

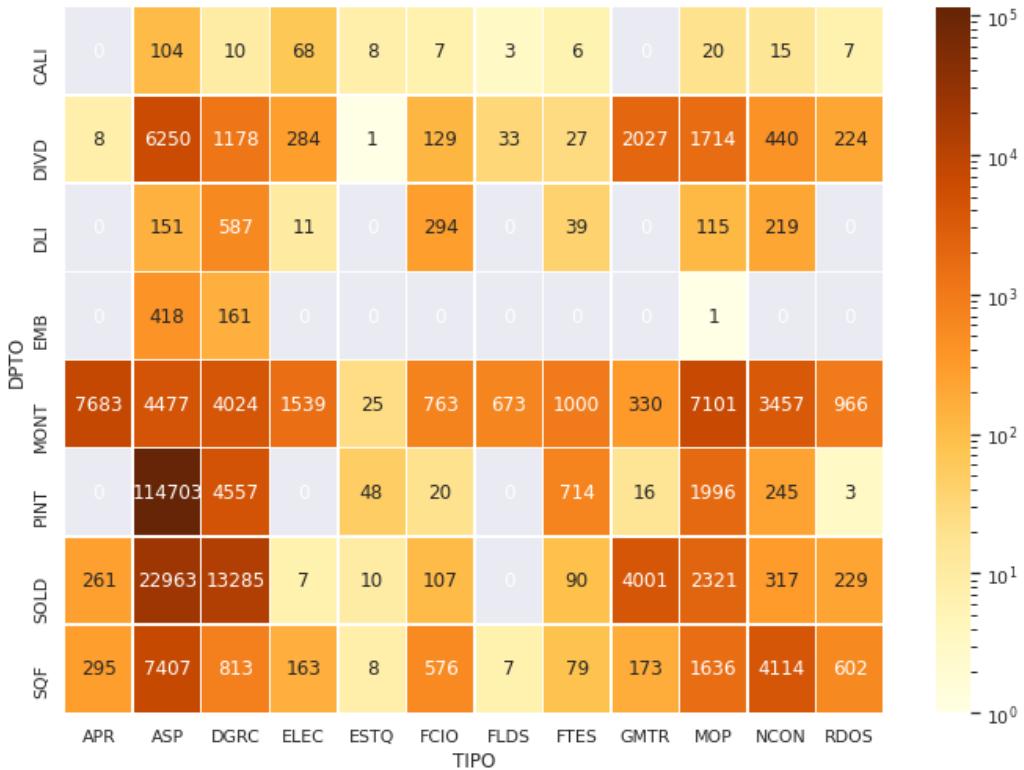


Figura 92: Cantidad de defectos cruzando DPTO y TIPO.

Pese a esto no se debe creer que aunque una combinación tenga una gran cantidad de defectos, la mayoría de estos serán V1. La fig. 93 nos muestra la densidad de V1 sobre el total de defectos, permitiéndonos establecer que determinadas combinaciones de TIPO y DPTO son más probables que sean V1 (incluso algunas poseen valores considerablemente más altos que las demás).

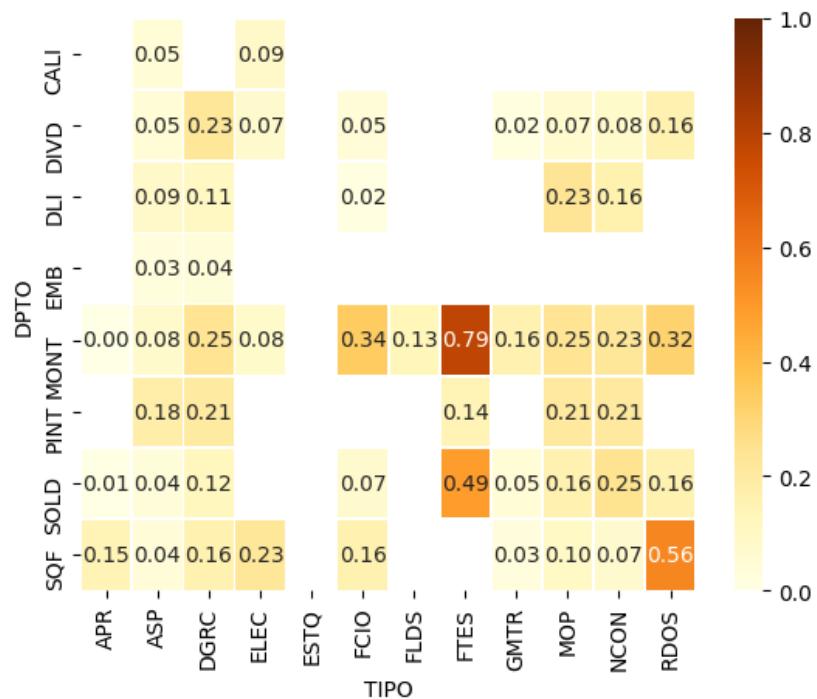


Figura 93: Fracción de V1 por cantidad de defectos.

El análisis visual más revelador es el de la figura 94. 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).

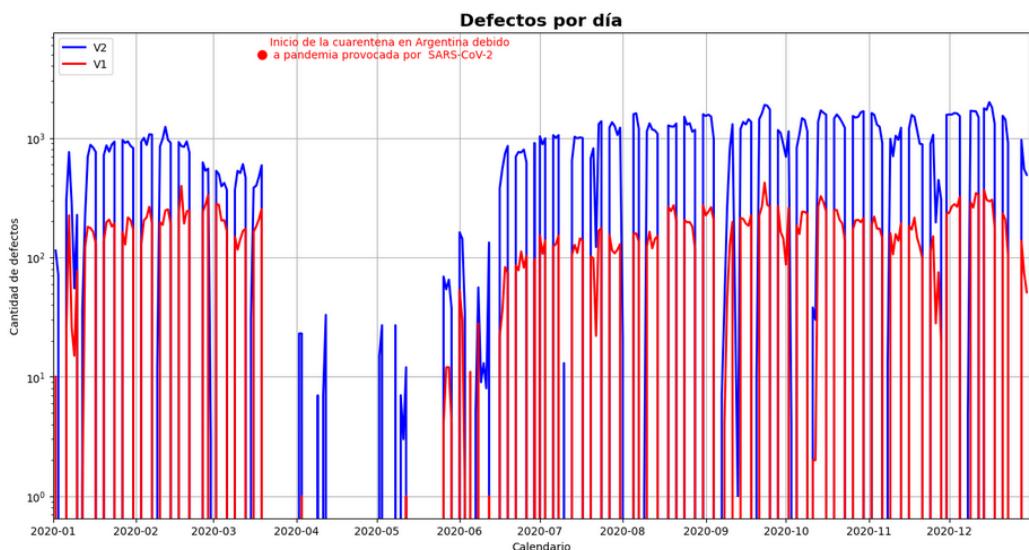


Figura 94: Defectos diarios agrupados por GVD.

Aún así, no podemos notar un patrón específico debido al ruido, por consiguiente incrementamos la ventana de tiempo a una semana (fig. 95). 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.

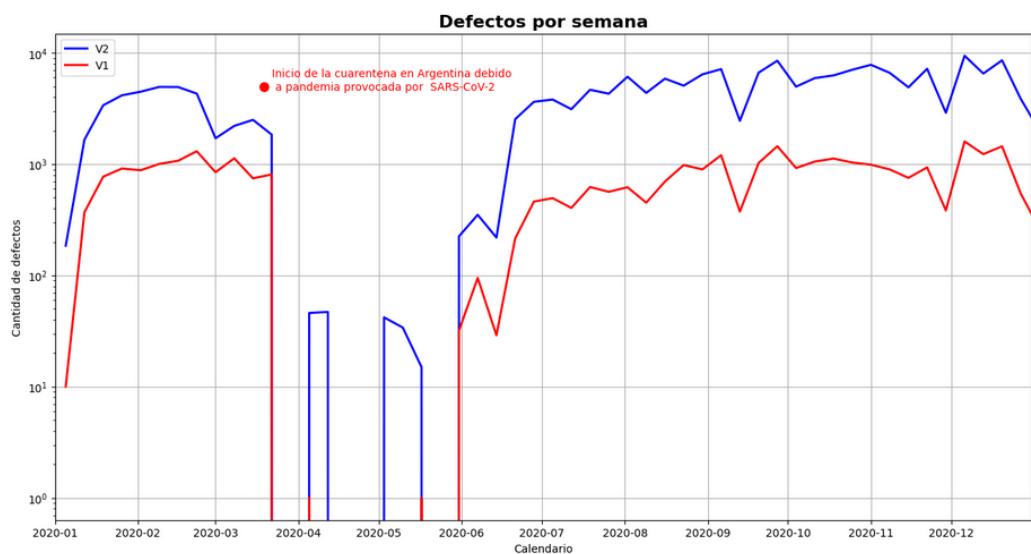


Figura 95: Defectos semanales agrupados por GVD.

Si al gráfico anterior lo desglosamos por auditoria y gravedad (fig. 96) veremos que en las demás auditorias que no sean DPU no hay patrones claros definidos, una de las causas es la baja densidad de datos disponibles.

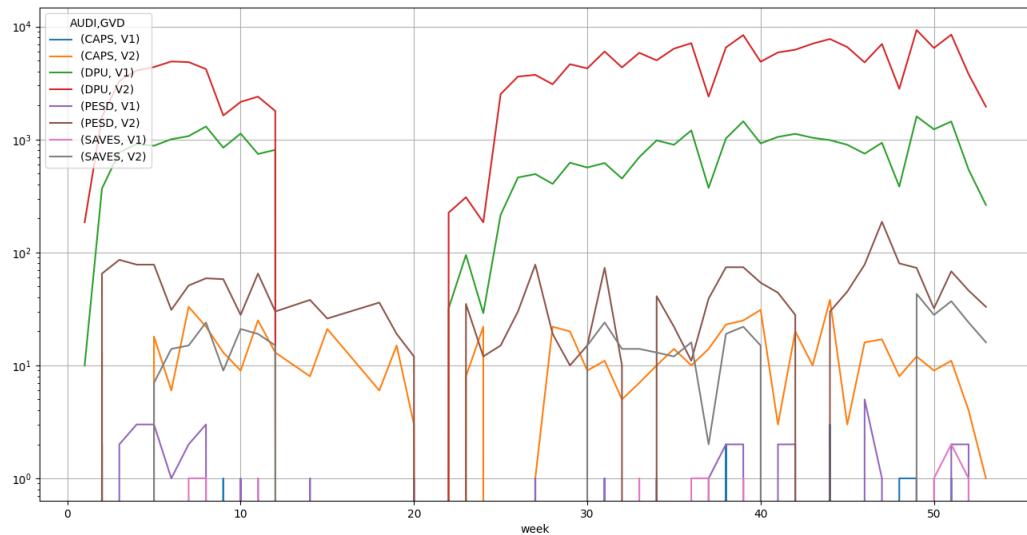


Figura 96: Defectos semanales agrupados por AUDI/GVD.

10.2.5 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. 97, 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.

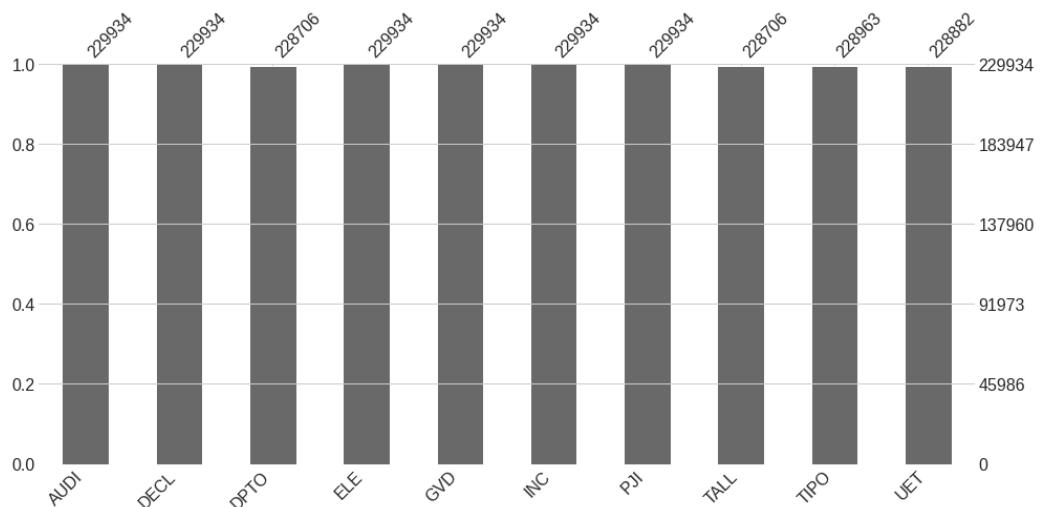


Figura 97: Datos faltantes por *feature*.

scikit-learn

Dado que el proceso de imputación no es crítico, se optó por utilizar la librería *sk-learn* (en lugar de redes neuronales) que nos provee de algoritmos ya implementados. Además nos permite a través de *pipelines* realizar un preprocesamiento de los datos para posteriormente entrenar al modelo.

Luego de revisar los valores faltantes se decidió utilizar 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 escogieron como *features* 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) [43].

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. 98 se ejemplifica parte del proceso.

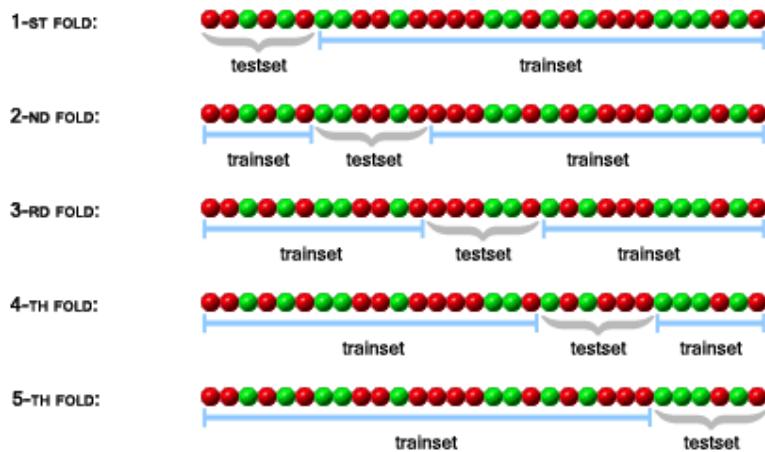


Figura 98: Ejemplo de validación cruzada de 5 veces con 30 muestras.

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.

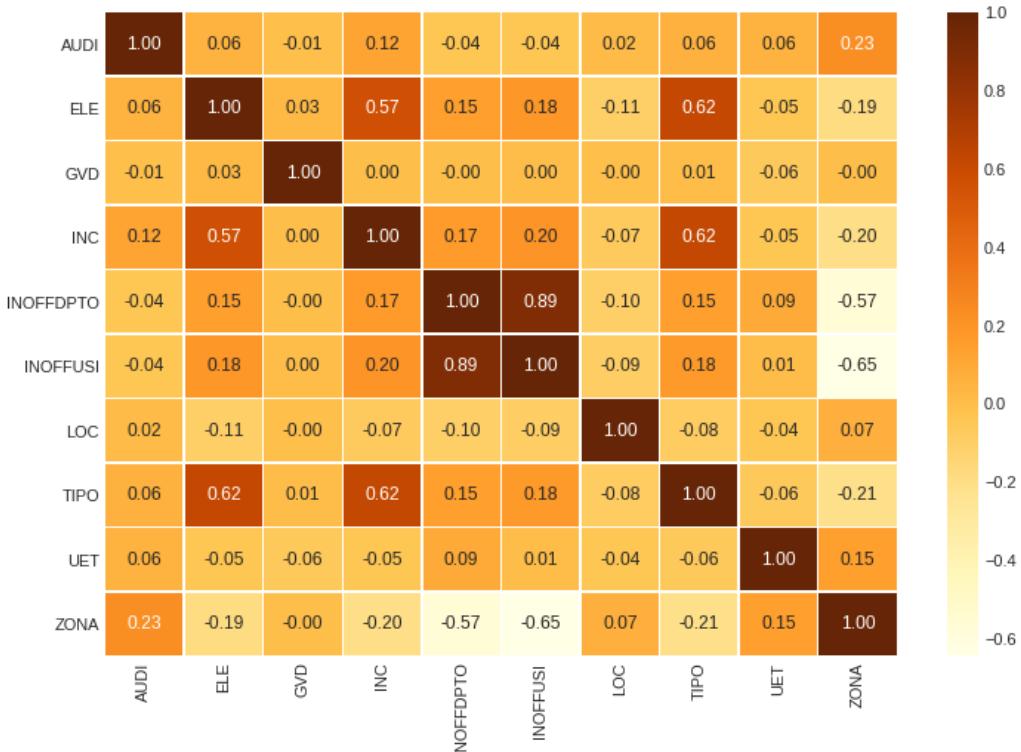
Por ello para obtener una estimación más exacta de la precisión, tiene sentido repetir la validación cruzada varias veces y tomar el promedio final después de cada iteración como estimación de precisión resultante.

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. [44]

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 99: Matriz de correlación de nuestro *dataset*.

En la fig. 99 pondremos atención 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 seleccionó ELE, INC, LOC, GVD y TIPO.

En nuestro caso debido a que todas las *features* son de tipo categoría usamos la codificación `one-hot encoder`, como observamos en la fig. 100.

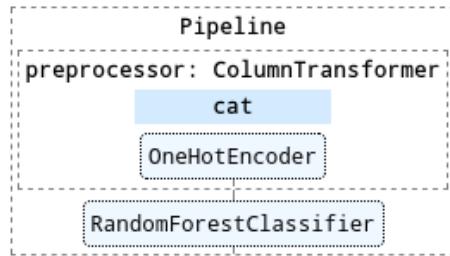


Figura 100: Gráfico de caja del rendimiento de cada modelo para TIPO.

Predicción de TIPO

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. 101).

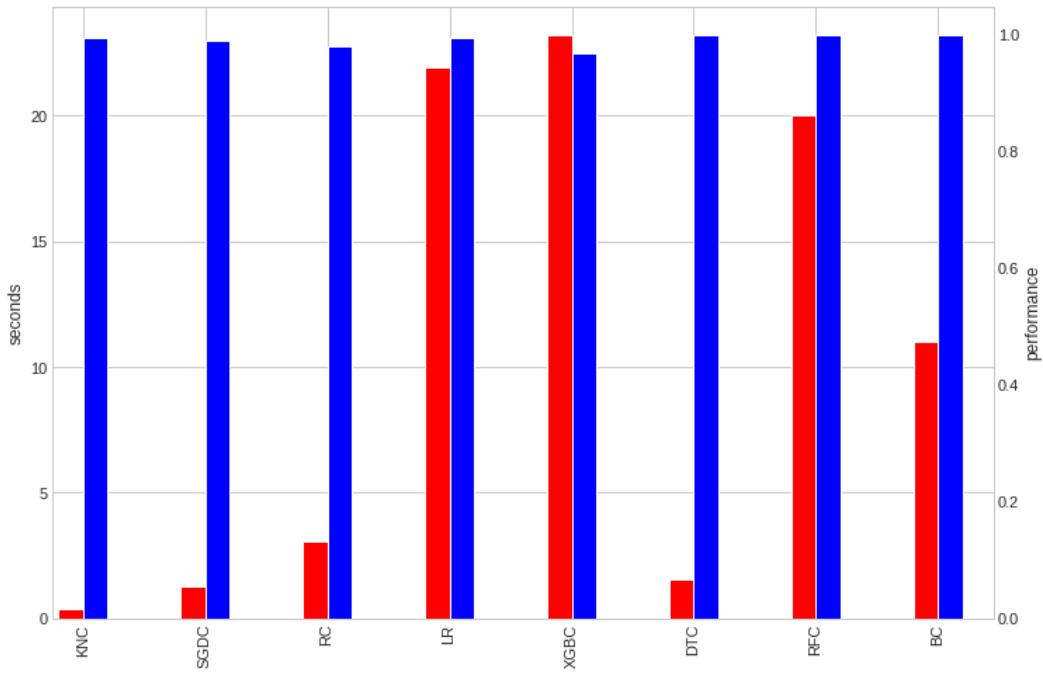


Figura 101: Gráfico de caja del rendimiento de cada modelo para TIPO.

Modelo	Tiempo de ejecución [seg]	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 TIPO

Los resultados de rendimiento de cada modelo se presentan en el cuadro 2 y los interpretaremos de manera visual en la fig. 102. Hay una leve ventaja de `RandomForestClassifier` por sobre los demás modelos, así que será el seleccionado para imputar los datos faltantes de TIPO.

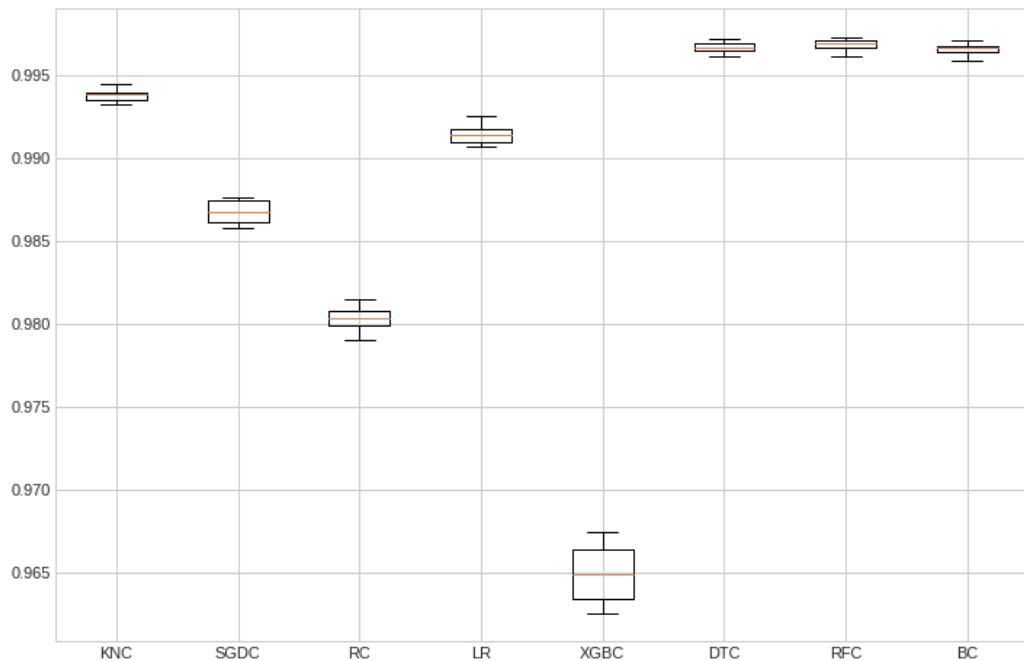


Figura 102: Gráfico de caja del rendimiento de cada modelo para TIPO.

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, el propósito de esto es quitar el componente de azar de los modelos y poder realizar las pruebas sobre los mismos datos.

Los mejores rendimientos se obtenían con ELE, INC y LOC, ergo se descartó GVD y TIPO.

Una vez más DTC logra la mejor relación entre tiempo de entrenamiento y rendimiento como detallamos en la fig. 103. Si nos basamos en rendimiento, RFC es superior (fig. 104).

Finalmente imputaremos los datos en nuestro dataset con el mejor clasificador para cada *feature*, materializando la idea de completar los datos faltantes.

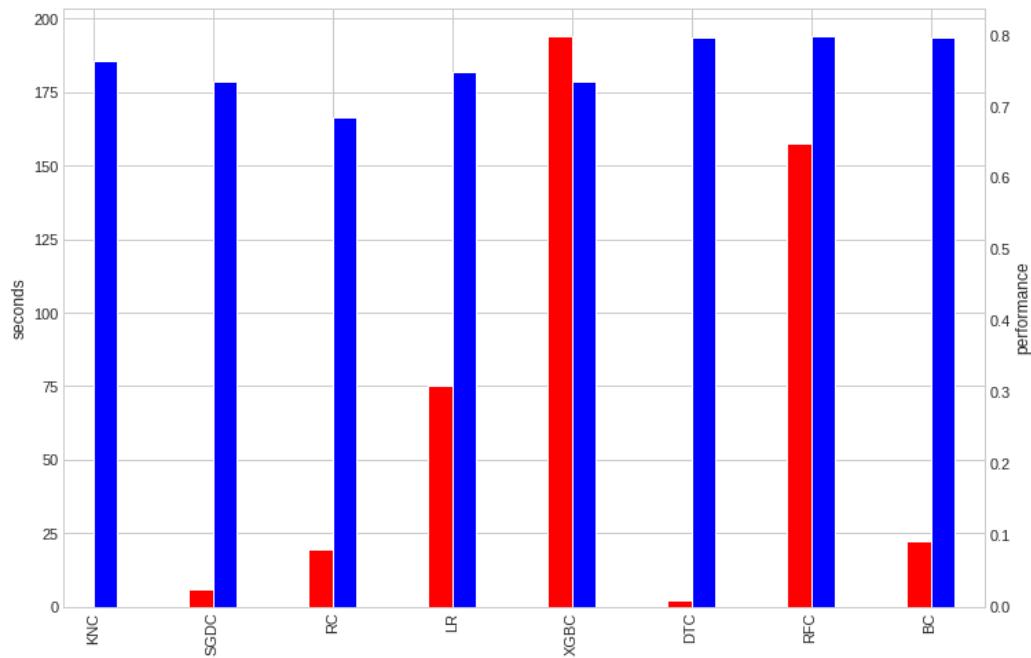


Figura 103: Gráfico de caja del rendimiento de cada modelo para TIPO.

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

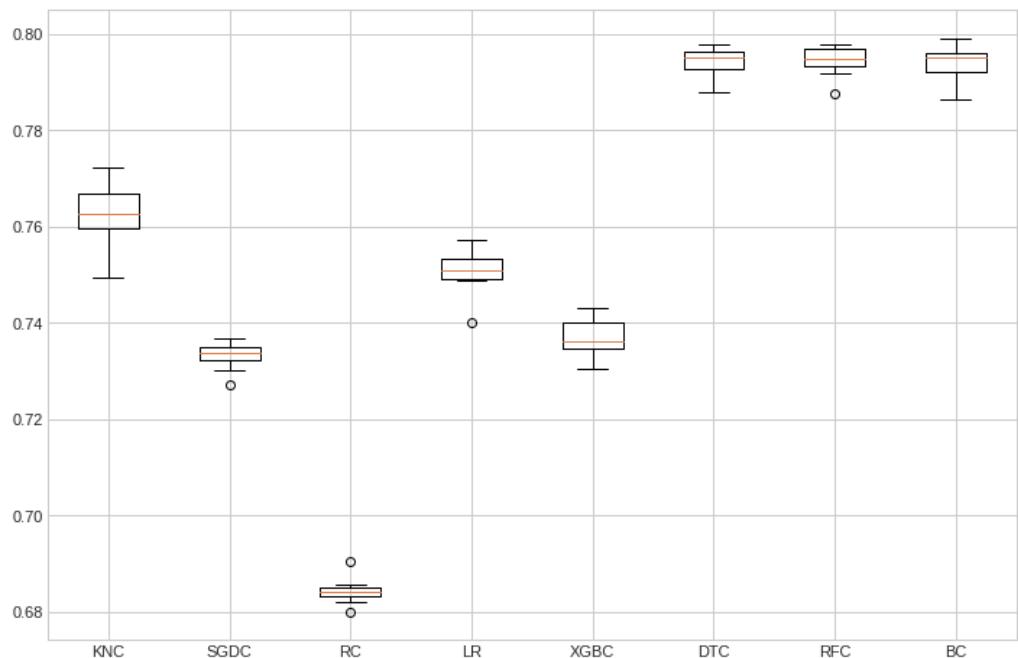


Figura 104: Gráfico de caja del rendimiento de cada modelo para TIPO.

10.2.6 Formateo de datos

El paso final antes de alimentar nuestro modelos es obtener rodajas (*slices*) de tiempo con funciones de agregación de tipo `sum`, `count`, entre otras de nuestros datos.

Serie temporal univariada

Pero como observamos en la Fig. 105, 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.

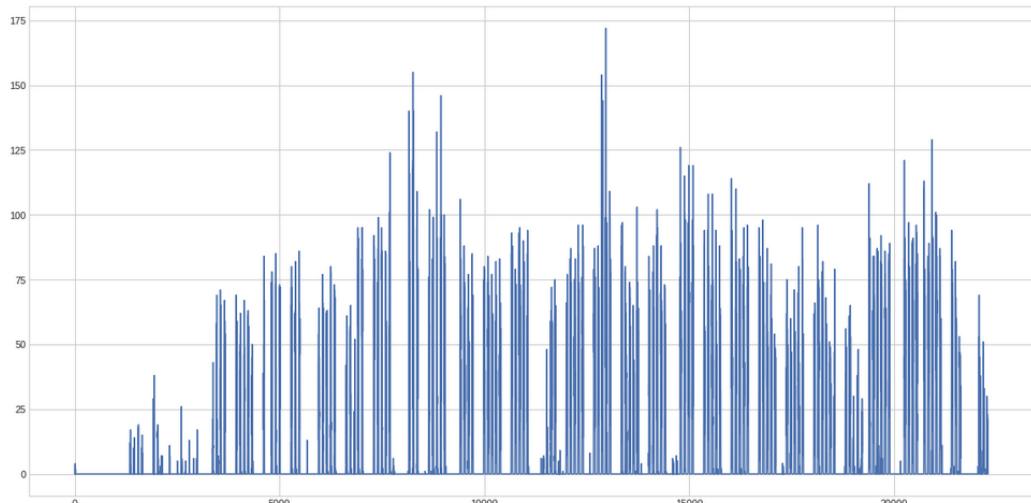


Figura 105: Gráfico temporal de cantidad de defectos.

Por esta razón se tomarán 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. 106 donde aplicamos la función escalón, notamos los huecos que tenemos en nuestra serie, por ello se aplicarán diversas funciones sobre la misma para obtener la serie continua deseada.

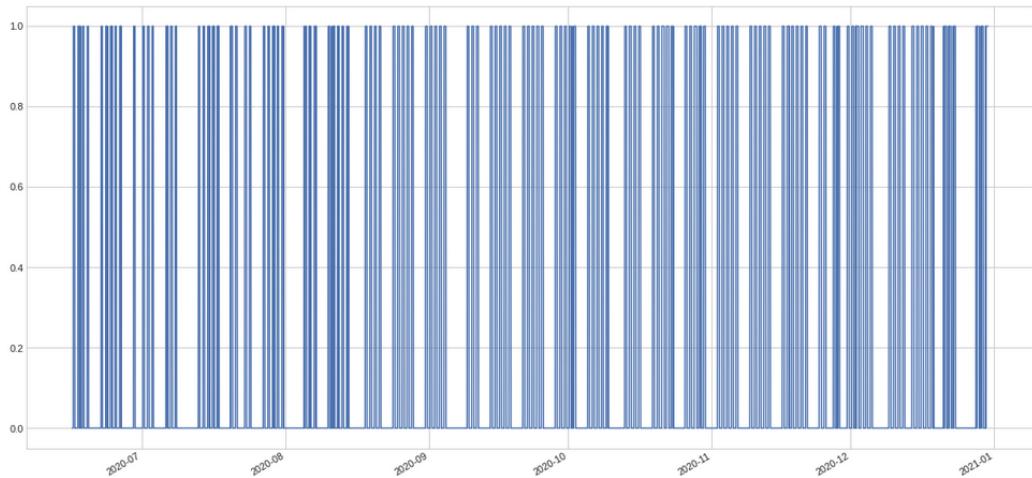


Figura 106: Gráfico temporal de detección de intermitencia.

Una vez aplicado los filtros necesarios obtenemos la siguiente serie de la Fig. 107. En la Fig. 108 si bien siguen apareciendo ceros, aparecen en menor cantidad siendo originados en horarios de producción mínima como el horario de inicio y finalización.

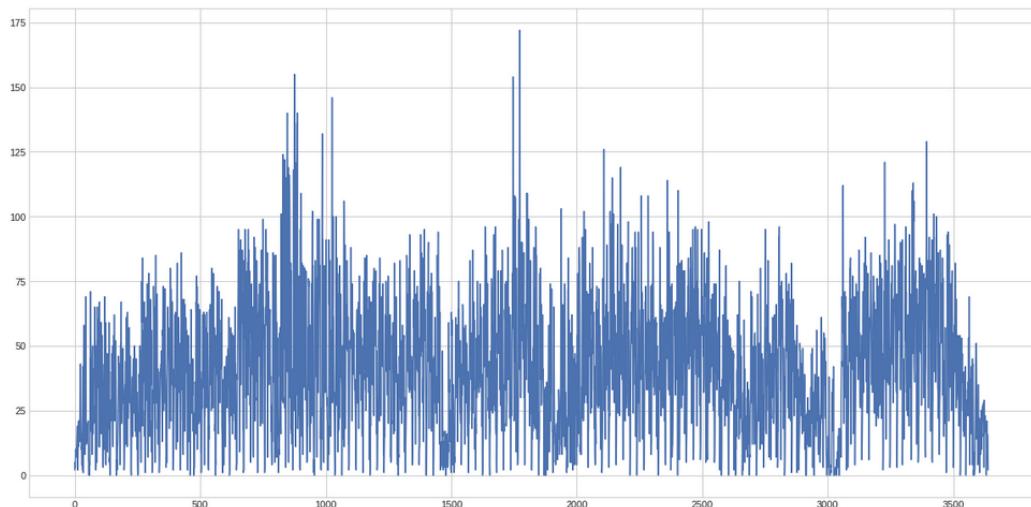


Figura 107: Gráfico temporal de cantidad de defectos procesado.

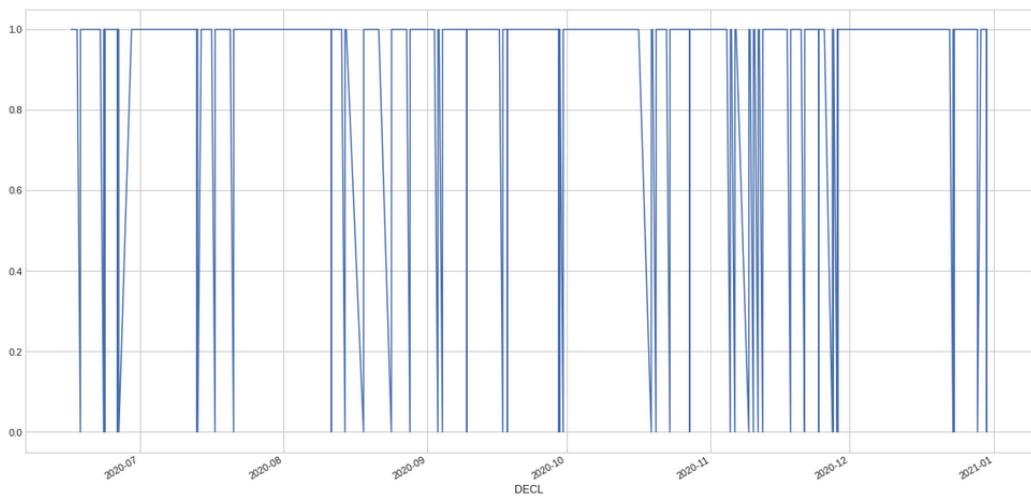


Figura 108: Gráfico temporal de detección de intermitencia procesado.

Serie temporal multivariada

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

En la fig. 109 observamos la cantidad de defectos que luego son desglosados por departamento, lo que nos daría 8 *features*. En la fig. 110 los defectos son desglosados por tipo, lo que nos daría 11 *features*. Lo que resultaría en 19 *features* que van a alimentar nuestro modelo.

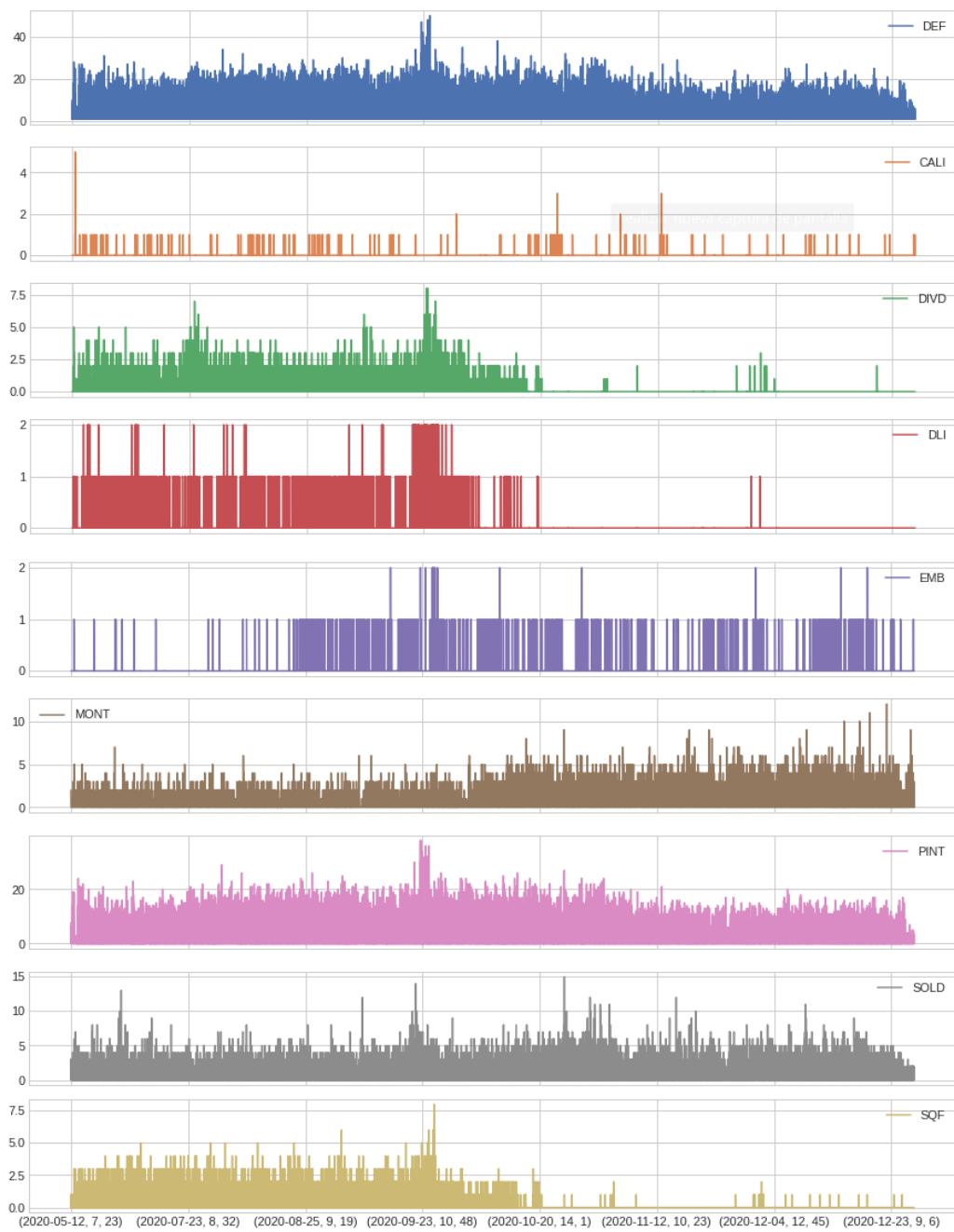


Figura 109: Gráfico temporal por departamentos.

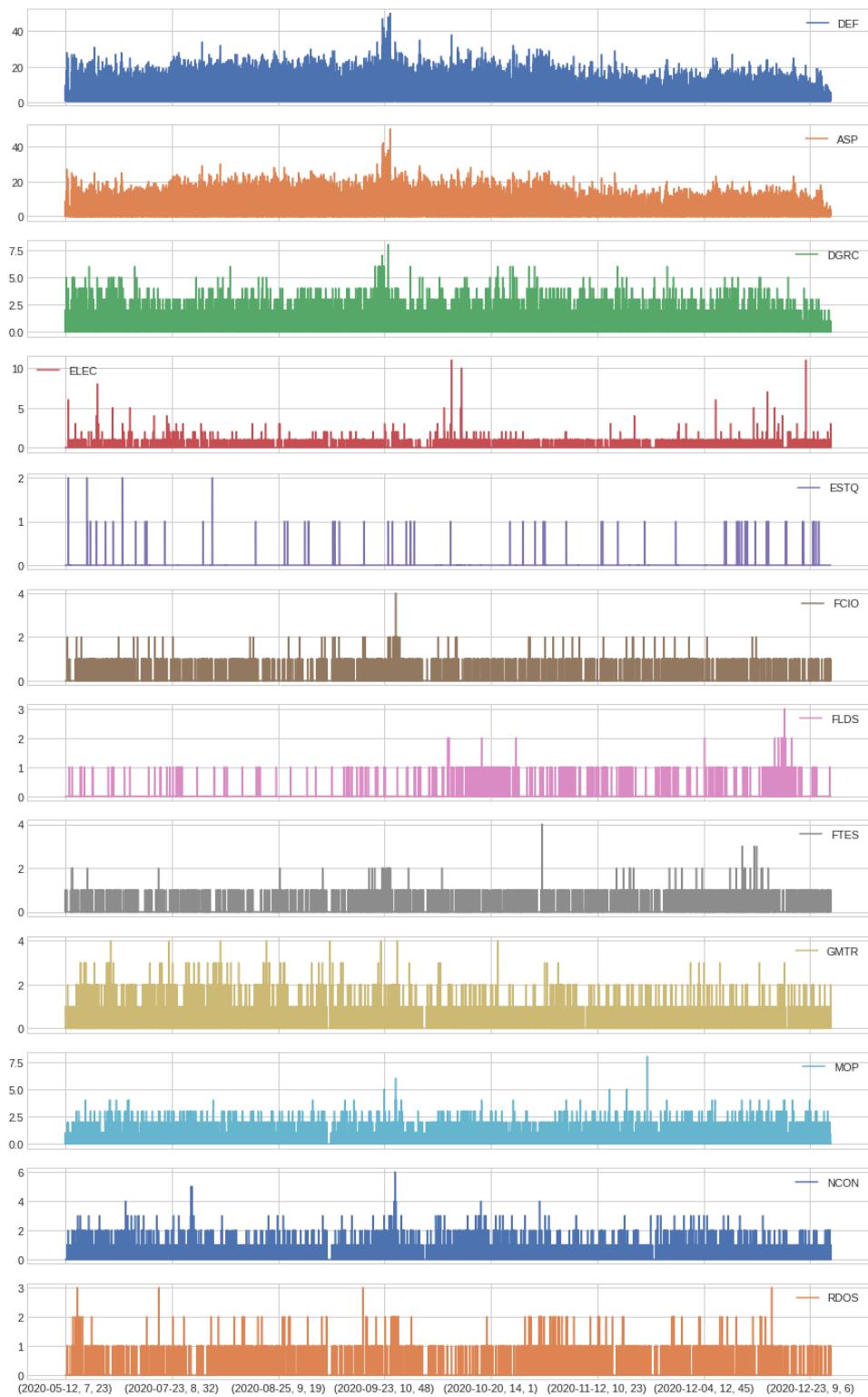


Figura 110: Gráfico temporal por tipo de defecto.

Tal como se realizó en la serie temporal univariada, se aplican los mismos pasos a estas series temporales de lo cual resultan las figuras 111 y 112.

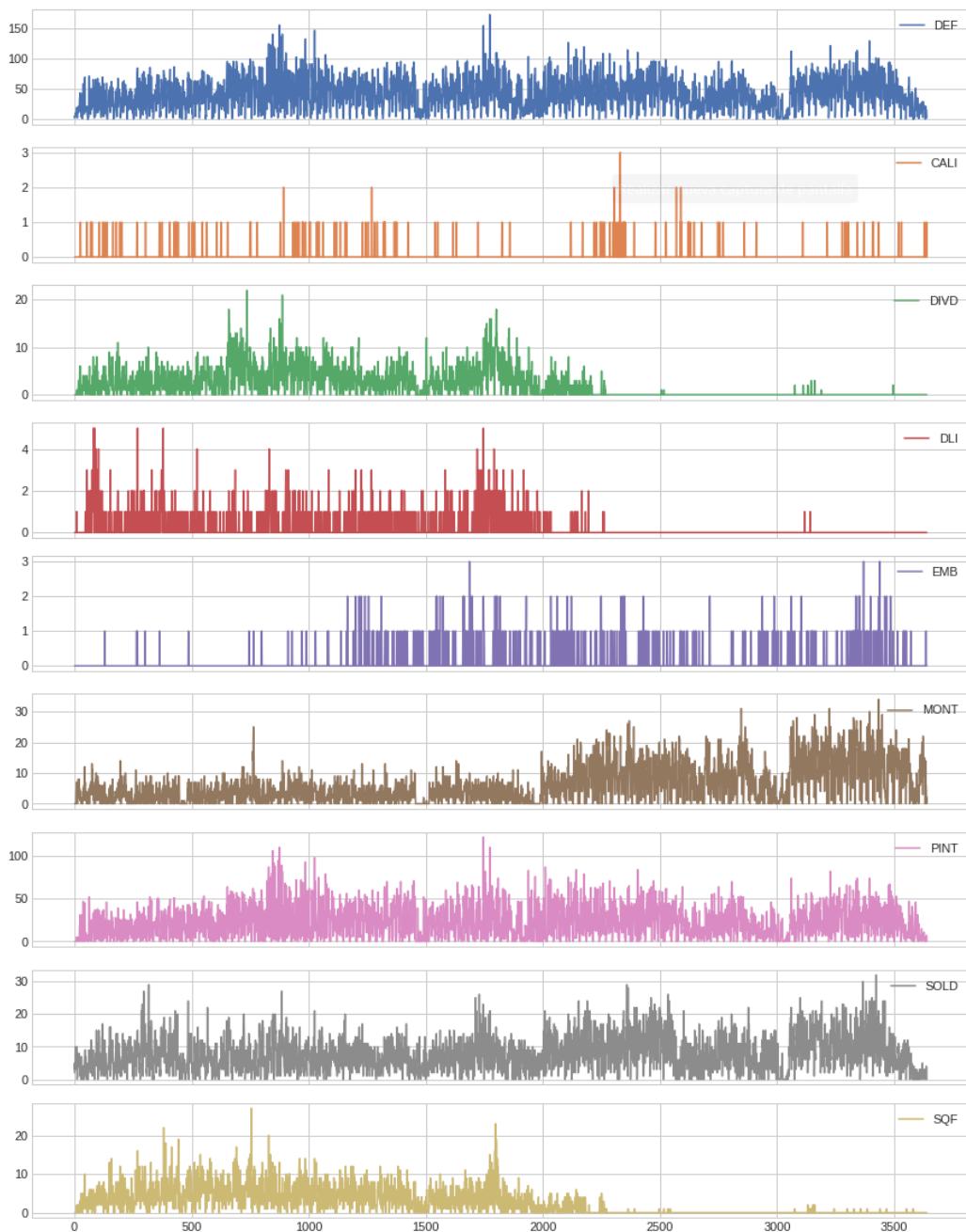


Figura 111: Gráfico temporal procesado por departamentos.

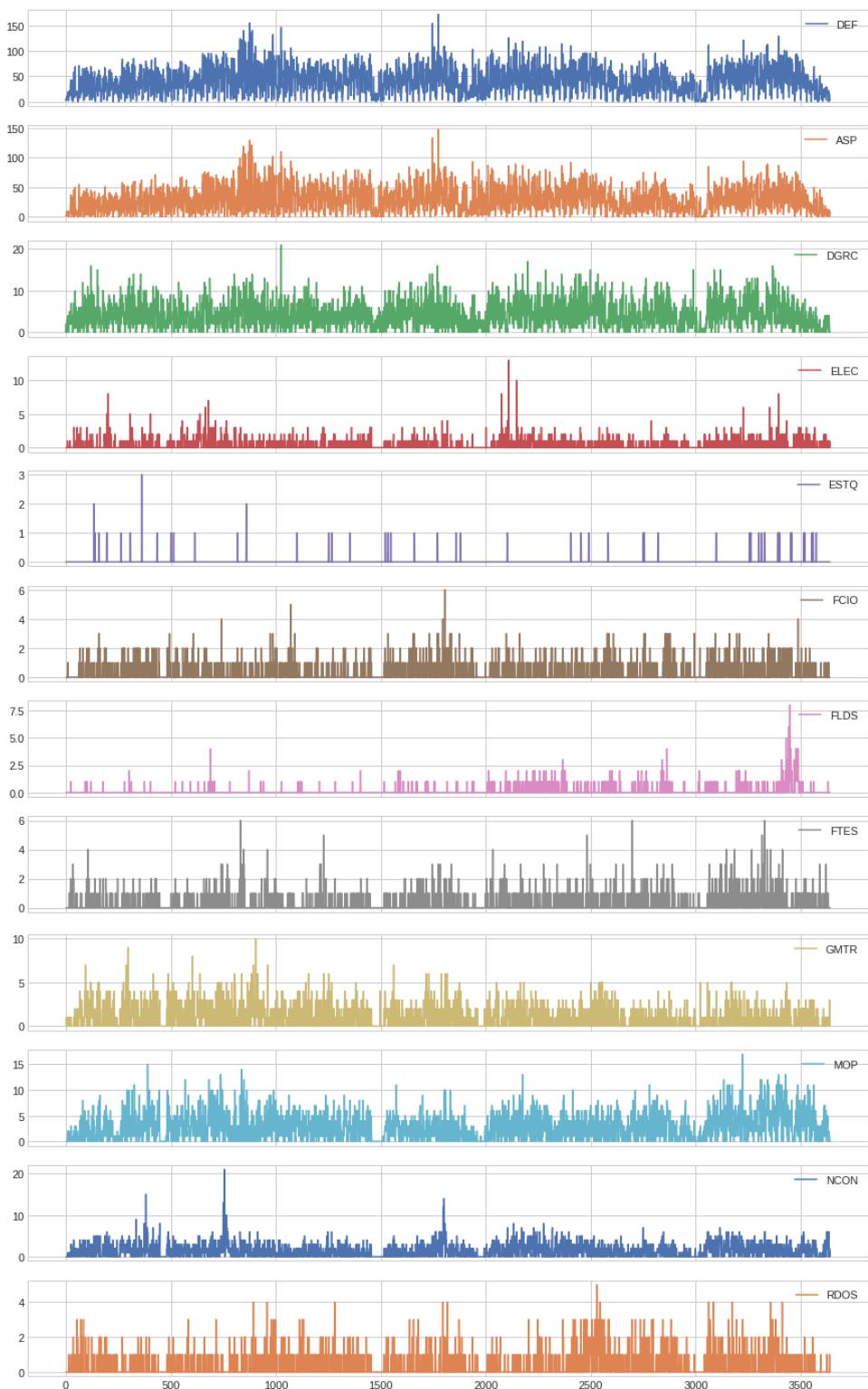


Figura 112: Gráfico temporal procesado por tipo de defecto.

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

11 Modelos

Los modelos serán creados combinando las diferentes capas que se han definido con anterioridad.

11.1 Métricas

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

11.1.1 MAPE

Una de las métricas más habituales para medir la precisión del pronóstico de un modelo es MAPE, que significa error porcentual absoluto medio (*mean absolute percentage error*). [45]

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 generalmente debido a su fácil interpretación y simple explicación. 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.

11.1.2 MASE

En el pronóstico de series temporales, el MASE es el error medio absoluto escalado (*Mean Absolute Scaled Error*). Es una métrica 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*. [46]

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*. Un valor mayor a uno indica que el algoritmo está funcionando mal en comparación al *naive forecast*.
- MASE es inmune al problema que enfrenta MAPE cuando la salida de la serie de tiempo real en cualquier paso de tiempo es cero. 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. 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.

11.2 *Framework*

Al momento de definir el *framework* de *Machine Learning* podemos encontrar 2 alternativas que se disputan el mayor porcentaje del mercado:

- *TensorFlow*: es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por *Google* para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. Usualmente se utiliza como *back-end* de otro *framework* llamado *Keras* que posee una API más amigable con el usuario.
- *PyTorch*: es una biblioteca de aprendizaje automático de código abierto basada en la biblioteca de *Torch*, utilizado para aplicaciones del campo de la visión artificial y del procesamiento natural del lenguaje, principalmente desarrollado por el Laboratorio de Investigación de Inteligencia Artificial de *Facebook*.

Los datos recopilados por la página *Papers With Code* [47] en la fig. 113 nos muestra una clara tendencia de los investigadores a preferir *PyTorch* sobre los demás *frameworks*.

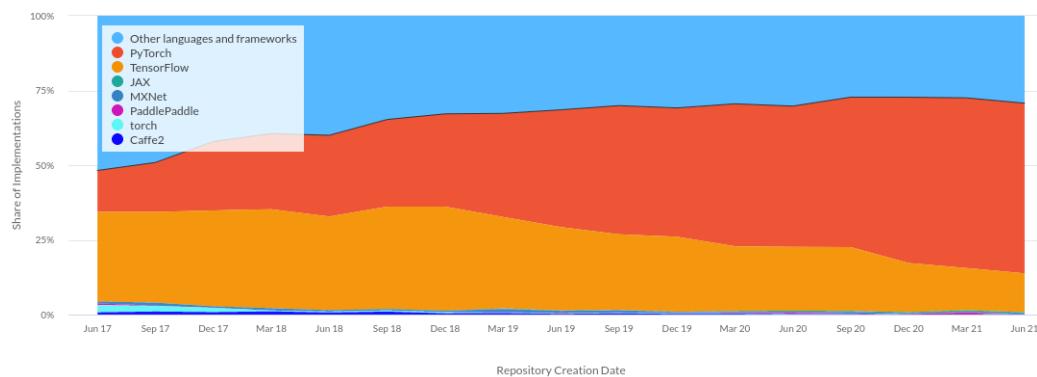


Figura 113: Porcentaje de implementaciones de cada *framework* en *papers*.

A pesar de esto para nuestra tarea debido a su facilidad de uso se escogió *Keras*, biblioteca de redes neuronales de código abierto cuyo autor principal es el ingeniero François Chollet.

Está especialmente diseñada para posibilitar la experimentación en un corto plazo con redes de Aprendizaje Profundo. Sus fuertes se centran en ser amigable para el usuario, modular y extensible. [48]

11.2.1 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. [49]

- **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 la métrica no disminuye, el entrenamiento termina. Esto es ventajoso para ahorrar tiempo de entrenamiento.
- **ReduceLROnPlateau** : reduce la tasa de aprendizaje cuando una métrica ha dejado de mejorar. Los modelos usualmente 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 brindado excelentes resultados.

- **ModelCheckpoint:** trabaja en conjunto 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:
 - Mantener solo el modelo que ha logrado el "mejor desempeño" hasta ahora, o guardar el modelo al final de cada época sin importar el desempeño.
 - La frecuencia a la que debería guardar. Actualmente, se 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.

11.3 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 se denomina arquitectura de tipo lineal. A modo informativo en *TensorFlow* es posible crear arquitecturas no-lineales que admitan múltiples flujos de datos entre las diferentes capas.

11.3.1 Hiperparámetros de capas

En la figura 114 podemos observar las capas disponibles para crear nuestra arquitectura.

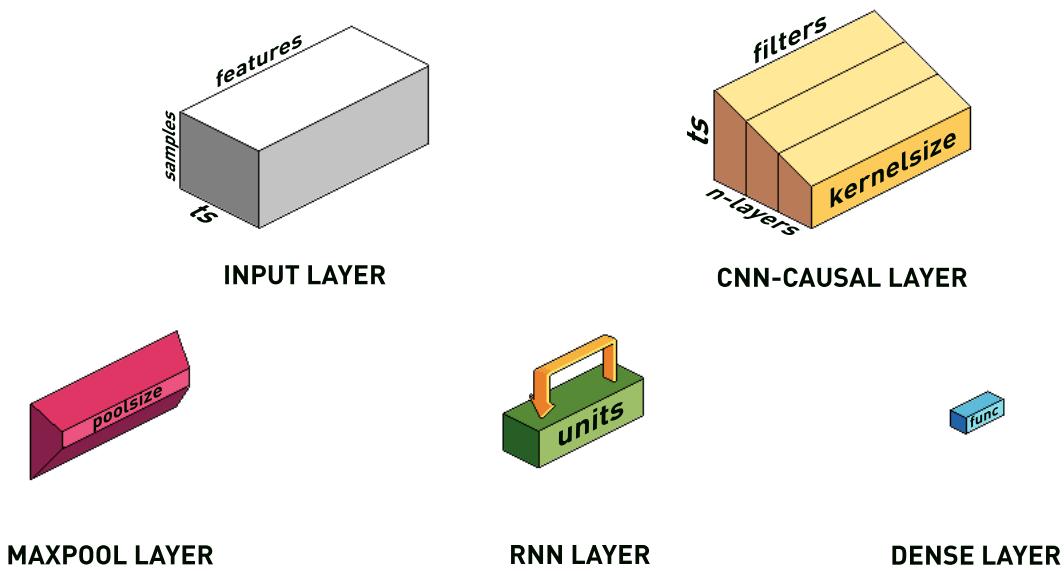


Figura 114: Tipos de capas disponibles.

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.

11.3.2 Tipos

En función a las capas explicadas en el marco teórico se ensayarán 5 arquitecturas de modelo representadas de forma visual en la fig. 115, con el objetivo de identificar la de mejor desempeño utilizando como parámetro la métrica MASE.

- 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.

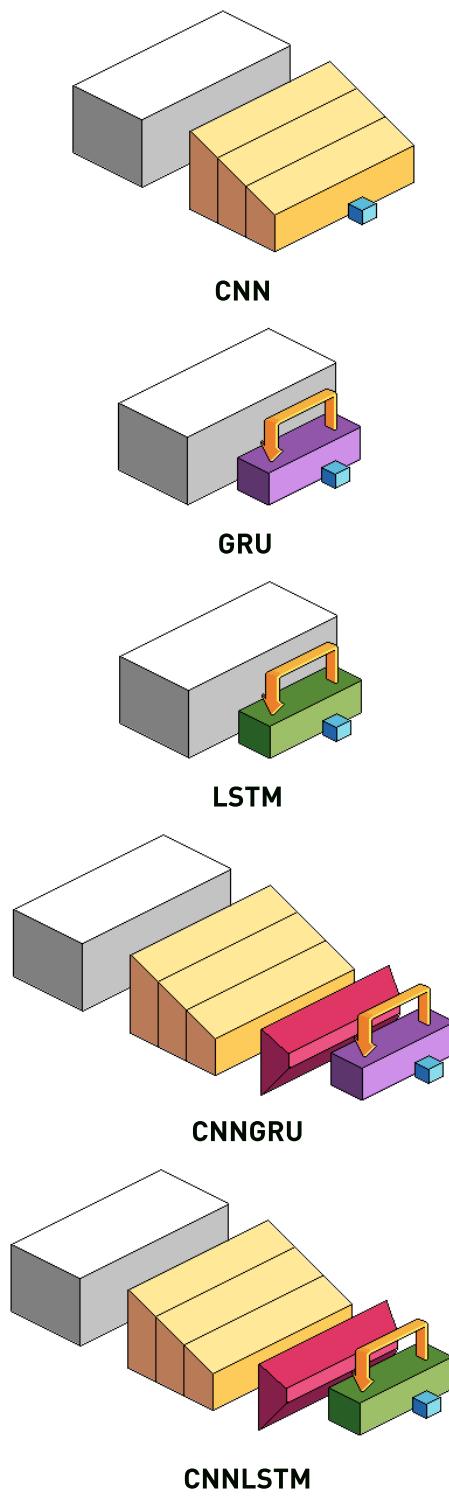


Figura 115: Tipos de arquitecturas.

11.4 Entrenamiento

11.4.1 Datos de prueba

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.

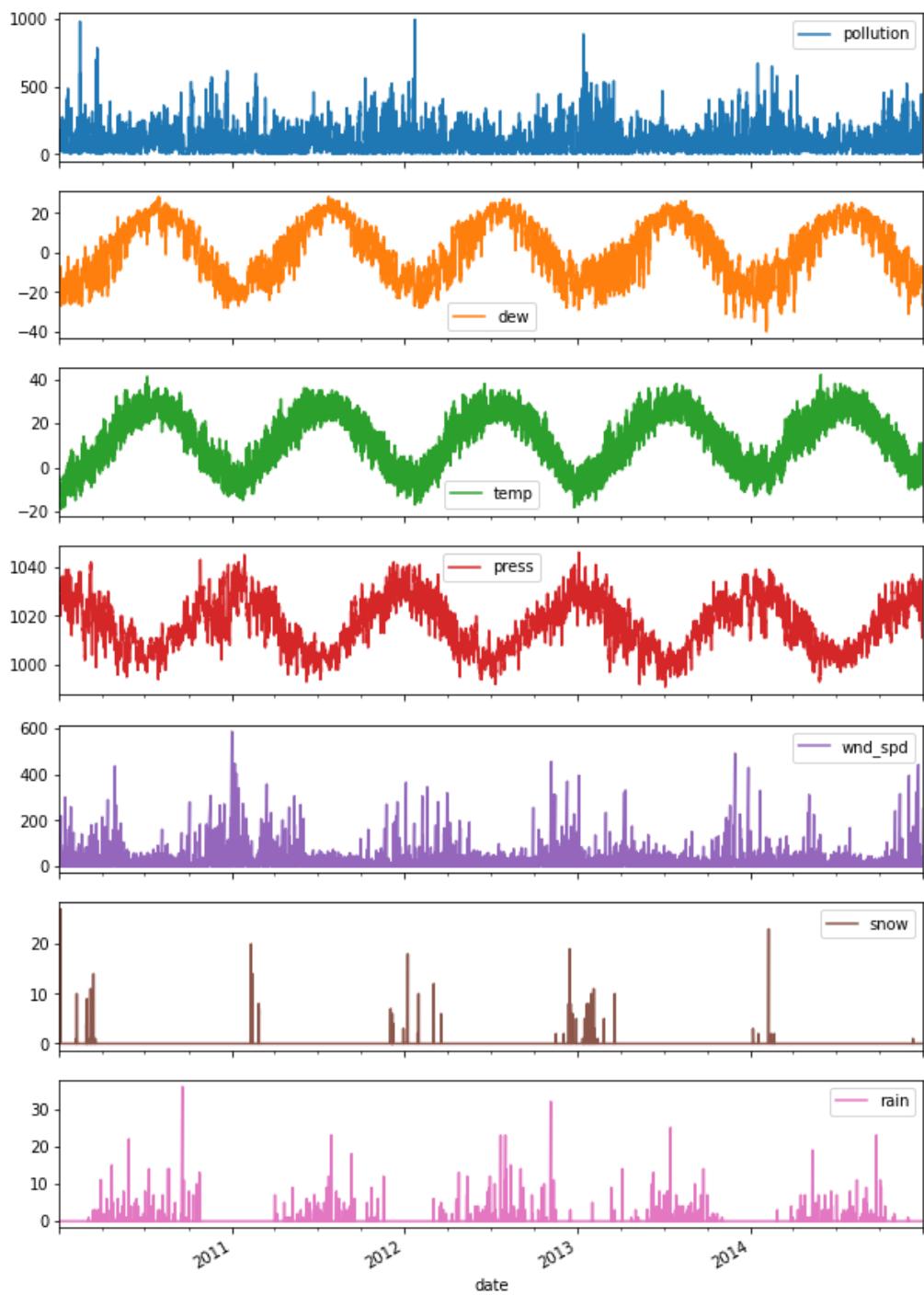
Para esta tarea haremos uso de un *dataset* de los niveles de polución en el aire. Contiene detalles sobre diferentes parámetros que son monitoreados cada hora durante cinco años en la embajada de Estados Unidos en Beijing, China.

Los datos incluyen la fecha y hora, la contaminación denominada concentración de PM2.5 y la información meteorológica. La lista completa de características en los datos 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.

Bajo estos datos se enmarcará un problema de pronóstico en el que dadas las condiciones climáticas y la contaminación de las horas previas, pronosticamos la contaminación de la hora siguiente.

En la fig. 116 podemos observar que nuestros datos son bastante armoniosos notando 3 ondas sinusoidales, característica que el modelo explotará (pero la mayoría de las series temporales en la vida real no serán tan convenientes).

Figura 116: *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.

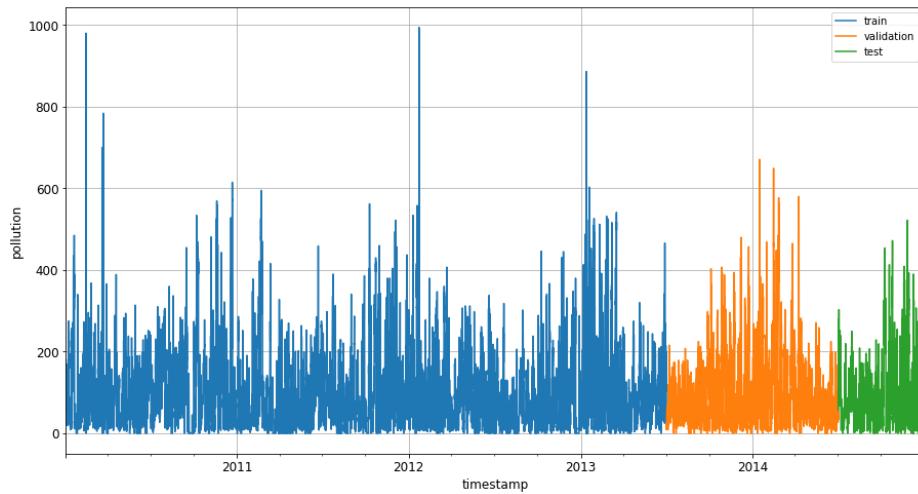


Figura 117: *Dataset* de contaminación del aire.

11.4.2 Desplazamiento temporal

Una operación importante a realizar en nuestro dataset es el desplazamiento temporal o *time-shifting* 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 118. 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*.

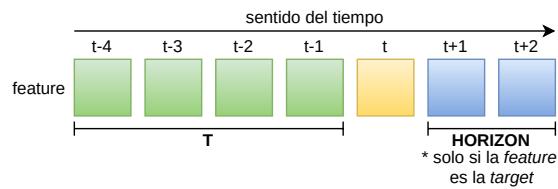


Figura 118: Desplazamiento temporal visualizado.

11.4.3 Tipo de horizonte

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. 119 [50]. Básicamente tomamos los T *timesteps* de nuestras *features* para alimentar a la arquitectura (que fueron presentadas en la sección 11.3), en este caso compuesta de celdas RNN, que luego la salida alimenta una neurona que determina el valor de t+1.

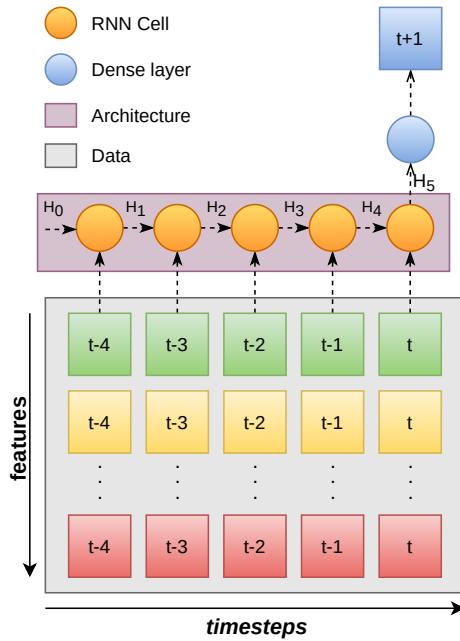


Figura 119: Modelo de horizonte único.

Horizonte múltiple

Observemos la fig. 120 [51]. Si determinamos que nuestra *target feature* será de horizonte múltiple, debemos aumentar la complejidad de nuestra arquitectura lo que se traduce en menor personalización 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.

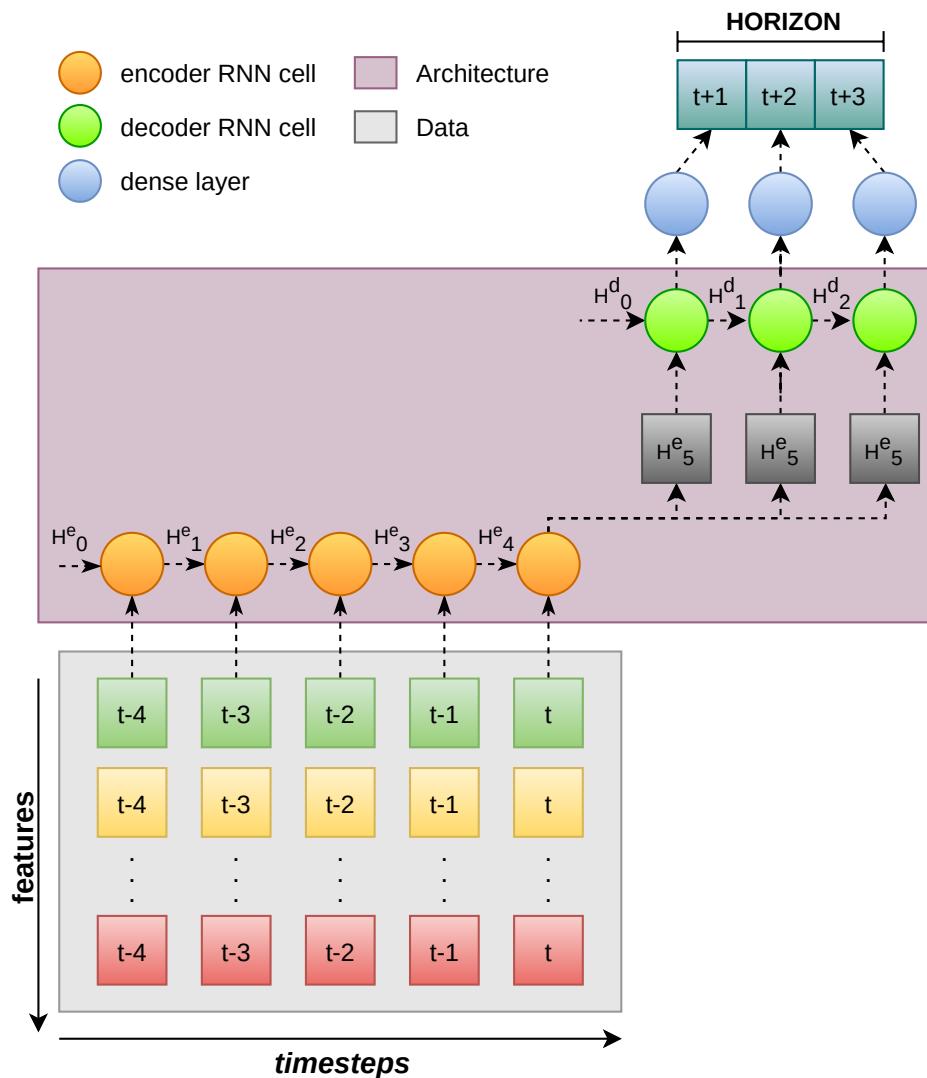


Figura 120: Modelo de horizonte múltiple.

En la fig. 121 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.

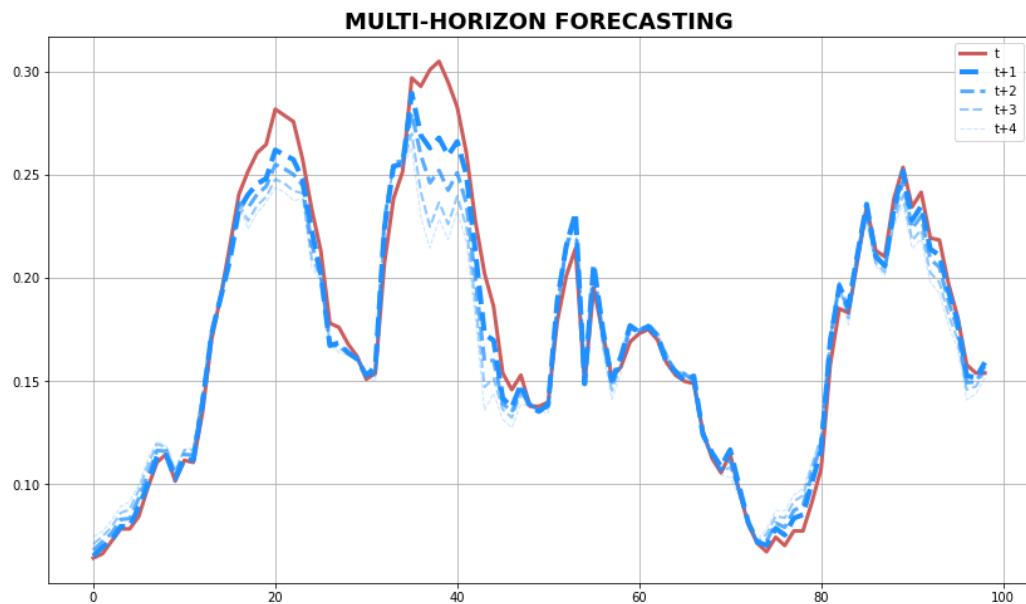


Figura 121: Pronóstico de horizonte múltiple.

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.

11.4.4 Resultados

Se entrenaron los 4 modelos principales (posteriormente se mostrarán los resultados de la CNN causal dilatada, ya que requiere un tratamiento diferente). Los siguientes hiper-parámetros se seleccionaron luego de varios ensayos donde se comprobó que como valores estándar tenían un rendimiento aceptable:

- EPOCHS=200
- UNITS=250
- BATCH_SIZE=64
- T=8
- HORIZON=1
- CNN_LAYERS=3

Se obtuvieron los resultados de la fig. 122. Los saltos que observamos en el `val_loss` (el más notorio en `CNNLSTM`) se deben a la callback de `ReduceLROnPlateau` (sección 11.2.1), siendo una función de *Keras* muy provechosa para nuestro objetivo.

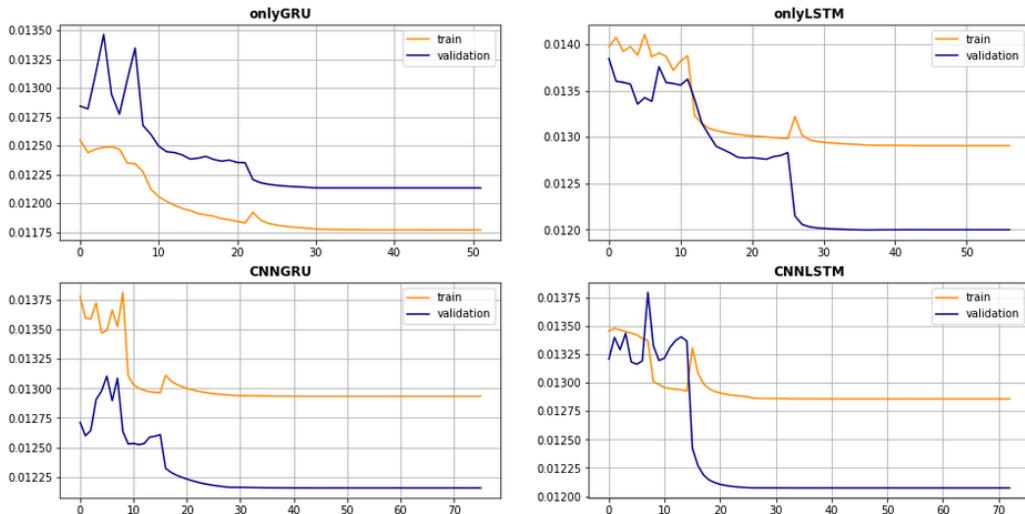


Figura 122: Resultados de entrenamiento de los modelos GRU/LSTM.

Para el entrenamiento de modelos compuestos exclusivamente de capas CNN dilatadas causales se utilizaron los siguientes hiper-parámetros:

- KERNEL_SIZE=2
- DEEP_LAYERS=1 + i donde $i \in \mathbb{N} : 1 \leq i \leq 8$.
- dilation_rate= i^2 donde $i \in \mathbb{N} : 1 \leq i \leq 8$.

En la fig. 123 representamos cada uno de los entrenamientos de los modelos, en los cuáles fuimos variando en cada modelo el valor de `dilation_rate`.

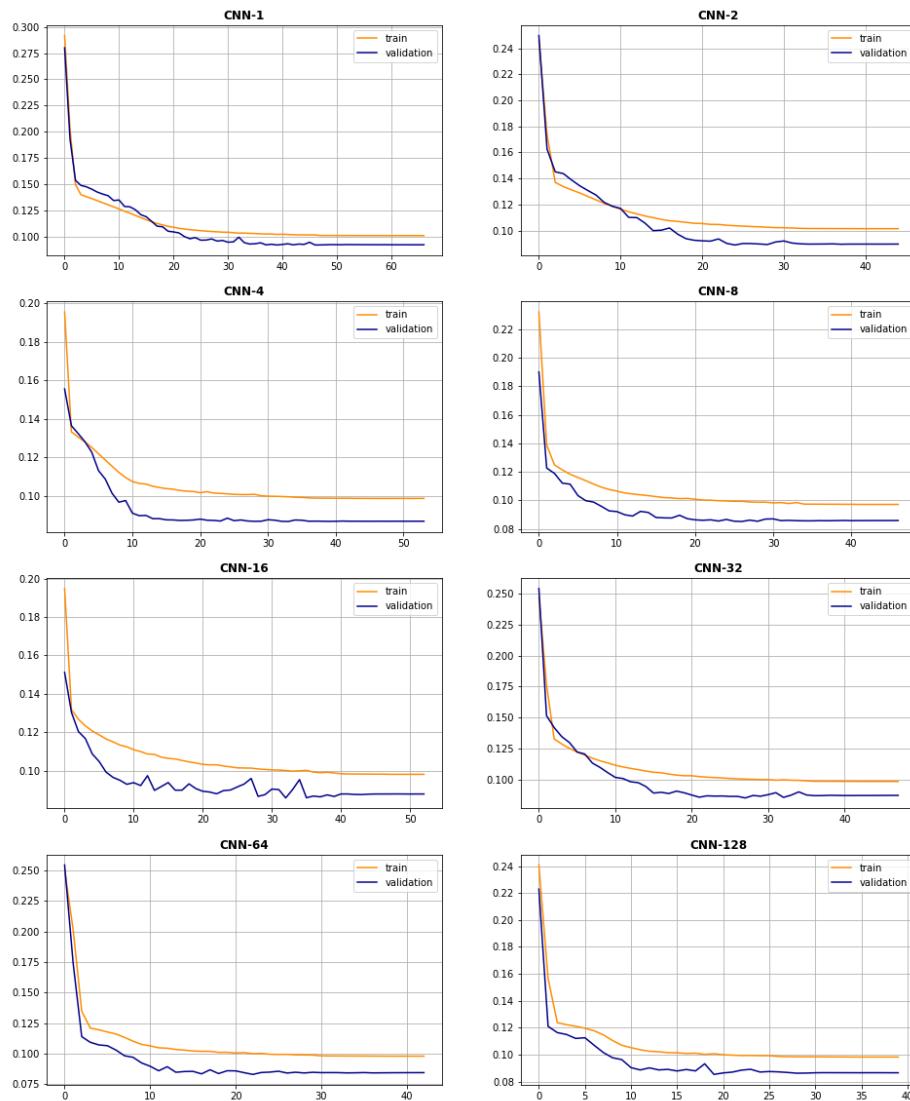


Figura 123: Resultados de entrenamiento de los modelos CNN.

Las diferencias que podemos notar entre la fig. 122 y la fig. 123 es que los modelos que poseen un componente RNN tienden a ser mucho más caóticos en su entrenamiento pero si la cantidad de épocas es suficiente el resultado es satisfactorio. En cambio, los modelos CNNs puros reflejan un entrenamiento más armonioso y continuo.

En la tabla 5 y en la fig. 124 verificamos que el modelo con el mejor MASE es el CNNGRU, sin embargo esto lo realizamos con valores fijos y estándares para los hiper-parámetros de los modelos.

model	MAE	MASE
GRU	0.0121	0.2894
LSTM	0.0123	0.2777
CNNGRU	0.0123	0.2775
CNNLSTM	0.0123	0.2793
CNN-1	0.0144	0.6526
CNN-2	0.0124	0.3476
CNN-4	0.0126	0.3242
CNN-8	0.0133	0.4389
CNN-16	0.0124	0.3172
CNN-32	0.0127	0.3318
CNN-64	0.0126	0.3597
CNN-128	0.0122	0.3121

Cuadro 5: Métricas de los modelos.

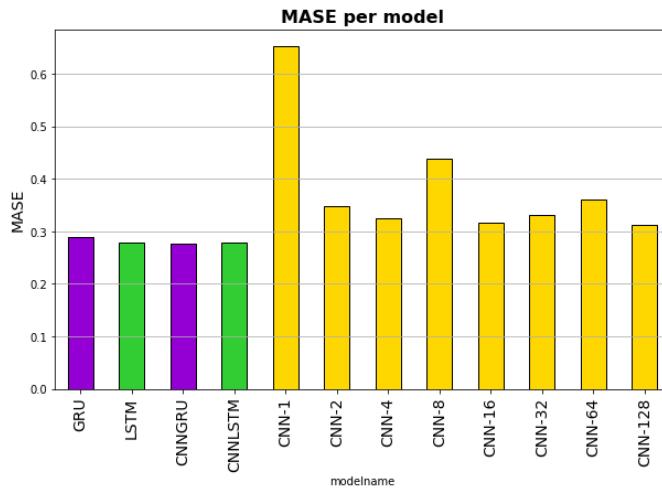


Figura 124: MASE por modelo.

Para una mejor visualización de los resultados, en la fig. 125 se toman las 200 primeras muestras. La precisión del modelo es alta, pronosticando valores muy cercanos a los reales. Por tanto, para estos datos el modelo ha resultado eficaz.

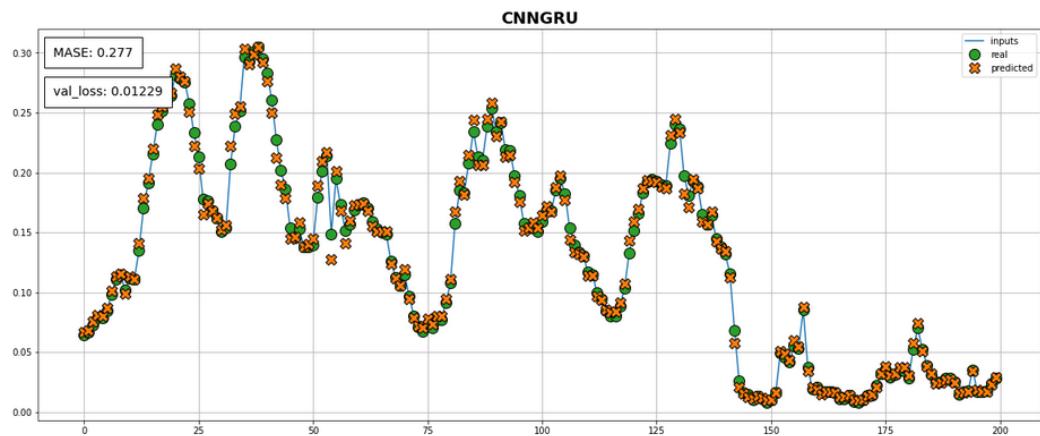


Figura 125: Pronóstico del mejor modelo.

11.5 Ajuste de hiperparámetros

Para obtener el mejor modelo es necesario explorar diversas configuraciones a través del ajuste de hiperparámetros (sección 7).

Para esta labor existen diversas librerías detalladas en la tabla 6 [52]. Entre todas ellos nuestra decisión será *Keras-tuner*, esencialmente por su facilidad de uso con la librería *Keras* (*framework* seleccionada de *Machine Learning*).

Frameworks HPO	Algoritmos			Frameworks ML					GPU
	HB	BO	RS	PT	TF	SL	XGB	LGBM	
<i>Ray Tune</i>	Si	Si	No	Si	Si	Si	Si	Si	No
<i>Optuna</i>	Si	Si	No	Si	Si	No	No	No	Si
<i>Hyperopt</i>	No	No	Si	Si	Si	Si	Si	No	No
<i>Scikit-Optimize</i>	Si	Si	Si	No	No	Si	No	No	No
<i>Keras-tuner</i>	Si	Si	Si	No	Si	No	No	No	Si
<i>Microsofts NNI</i>	No	Si	No	Si	Si	Si	No	No	Si

Cuadro 6: Comparación de los diferentes *frameworks* disponibles para optimización de hiper-parámetros.

Aconimos utilizados:

- HB: *HyperBand*.
- BO: *Bayesian Optimization*.
- RS: *Random Search*.
- PT: *PyTorch*.
- TF: *TensorFlow*.
- SL: *Scikit-Learn*.
- XGB: *XgBoost*.
- LGBM: *LightGBM*.
- MN: *MxNet*.

11.5.1 *Keras-tuner*

La librería se compone de diversas clases y métodos que nos permitirán ajustar los hiperparámetros de nuestros modelos creados en *Keras*. [53]

Clase HyperParameters

Sirve como contenedor de hiperparámetros, una instancia de *HyperParameters* contiene información sobre el espacio de búsqueda y los valores actuales de cada hiperparámetro.

Además nos provee diversos métodos para decidir de que manera queremos explorar el espacio de búsqueda de un determinado hiperparámetro:

- **Boolean**: elige entre `True` o `False`.
- **Choice**: elige entre una lista de opciones.
- **Fixed**: fijo, no permite cambiar su valor.
- **Int**: explora números enteros entre el valor inicial y final, es posible seleccionar el modo y el paso.
- **Float**: *ídem* al ítem anterior, pero con números reales.

Clase Oracle

Cada clase `Oracle` (óraculo) implementa un algoritmo de ajuste de hiperparámetros particular. Un `Oracle` se pasa como argumento a un `Tuner` (clase cuyo objetivo es realizar la búsqueda de hiperparámetros). `Oracle` le informa al `Tuner` qué hiperparámetros deben probarse a continuación.

La mayoría de las clases de `Oracle` se pueden combinar con cualquier subclase de `Tuner` definida por el usuario. Si no se necesita una subclase de `Tuner` (el caso más común), también se proporciona una serie de clases que empaquetan un `Tuner` y un `Oracle` juntos (cómo veremos a continuación).

Clase Tuner

Su propósito es realizar la búsqueda de hiperparámetros en el espacio de búsqueda definido, y como se aclaró en la sección anterior es posible crearlos de forma personalizada pero los algoritmos ofrecidos son más que suficientes para nuestra tarea (analizados en la sección 7.1):

- Búsqueda Aleatoria (descartado).
- Optimización Bayesiana: parámetro a destacar:
 - `max_trials`: número total de pruebas (configuraciones de modelo) para probar como máximo.
- *Hyperband*: parámetros a destacar:
 - `factor`: factor de reducción (η) para el número de épocas y el número de modelos para cada `bracket`.
 - `hyperband_iterations`: el número de veces que se iterará sobre el algoritmo *Hyperband* completo. Una iteración ejecutará aproximadamente $s_{max}(\log_\eta s_{max})^2$ épocas acumulativas en todas las pruebas (notación tomada de la sección 7.1.3). Se recomienda establecer esto en un valor tan alto como esté dentro de su presupuesto de recursos.

11.5.2 Espacio de búsqueda

Para cada tipo de modelo se definió un espacio de búsqueda determinado en función a su arquitectura.

RNN

Se limitó a explorar la cantidad de unidades en las capas recurrentes e intercambiar la función de activación de la capa final.

```

1 # RNN layer
2 for i in range(8,256,16):
3     units = i
4 # Dense layer
5 for j in ['relu', 'sig', 'tanh']:
6     dense = j

```

CNN

En este caso se debió crear una capa CNN *input* y sobre la misma ir apilando las demás capas CNN que fueron variando en cantidad desde 1 a 3. Además se ajustaron los hiperparámetros de filtros y tamaño del kernel en cada una de ellas.

```

1 # For first layer
2 for i in range(8,64,16):
3     filters_0 = i
4     for j in range(2,8,2):
5         kernel_size_0
6
7 # Others layers
8 for i in range(1,3):
9     cnn_i = stack_cnnlayer(dilation_date=2**i)
10    for j in range(8,64,16):
11        filters_i = j
12        for k in range(2,8,2):
13            kernel_size_i = k
14
15 # Model compile
16 for lr in [1e-2, 1e-3, 1e-4]:
17     model_lr = lr

```

CNNRNN

Se realizó una configuración mixta de los hiperparámetros de las otras arquitecturas, dónde la única novedad es la búsqueda en los hiperparámetros de la capa MaxPooling1D.

```
1 # First CNN layer
2 for i in range(8,64,16):
3     filters_0 = i
4     for j in range(2,8,2):
5         kernel_size_0
6
7 # Others CNN layers
8 for i in range(1,3):
9     cnn_i = stack_cnnlayer(dilation_date=2**i)
10    for j in range(8,64,16):
11        filters_i = j
12        for k in range(2,8,2):
13            kernel_size_i = k
14
15 # Pooling layer
16 for i in range(2,8,2):
17     pool_size = i
18
19 # RNN layer
20 for i in range(8,256,16):
21     units = i
22
23 # Model compile
24 for lr in [1e-2, 1e-3, 1e-4]:
25     model_lr = lr
```

11.5.3 Resultados *dataset* polución

Ya definido el espacio de búsqueda de los modelos, se procede a la búsqueda del mejor conjunto de hiperparámetros obteniendo la tabla 7 (recordar que se utiliza como métrica MAE). De la misma podemos sacar varias conclusiones interesantes.

- El modelo con la mejor puntuación fue CNNGRU en combinación con el algoritmo *Hyperband*, aunque el segundo lugar (LSTM/Optimización Bayesiana) le sigue muy de cerca. De hecho todos los modelos están muy parejos.
- Hay una clara tendencia a obtener mejores resultados con combinaciones definidas como CNNGRU/HB/**sig** y LSTM/BO/**tanh**.
- Es interesante notar que a pesar de su arquitectura simple LSTM logró batir los resultados de otros modelos.
- Nuestro tope de unidades RNN fue seteado en 256, observamos que los modelos con mejor puntaje están muy cerca de ese valor. Existe la posibilidad de obtener mejores resultados en caso de haber definido un espacio de búsqueda mayor para ese hiperparámetro.

model	algo	layers	CNN				MAXPOOL		RNN	DENSE	score
			f_0	ks_0	f_1	ks_1	f_2	ks_2			
CNNGRU	HB	3	40	8	56	4	56	6	6	232	sig 0.011961
LSTM	BO	-	-	-	-	-	-	-	248	tanh 0.011973	
CNNGRU	HB	3	40	8	56	4	56	6	6	232	sig 0.011980
LSTM	BO	-	-	-	-	-	-	-	248	tanh 0.011996	
CNNGRU	HB	3	40	8	56	4	56	6	6	232	sig 0.011996
LSTM	BO	-	-	-	-	-	-	-	248	tanh 0.012001	
CNNGRU	HB	1	56	6	-	-	-	-	6	88	sig 0.012003
LSTM	BO	-	-	-	-	-	-	-	248	tanh 0.012004	
CNNGRU	HB	1	40	6	-	-	-	-	6	120	sig 0.012004
CNNGRU	HB	2	40	6	40	4	-	-	6	216	sig 0.012008

Cuadro 7: Resultados del ajuste de hiperparámetros.

Se pondrá atención en los rendimientos por arquitectura de ambos algoritmos de ajuste de hiperparámetros utilizados discriminando los análisis por las dos métricas propuestas (MAE y MASE).

MAE

Notamos que exceptuando LSTM, el algoritmo *Hyperband* se desempeña mejor en cada uno de los modelos restantes (tabla 8 y fig. 126).

algo/model	CNN	CNNGRU	CNNLSTM	GRU	LSTM
<i>BayesianOptimization</i>	0.01260	0.01224	0.01222	0.01218	0.01197
<i>Hyperband</i>	0.01213	0.01196	0.01203	0.01204	0.01202

Cuadro 8: Mejor puntuación por algoritmo y arquitectura utilizando MAE como métrica.

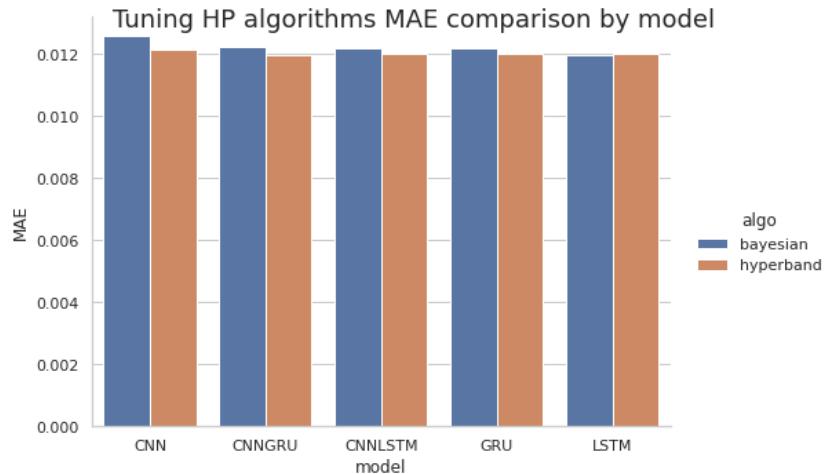


Figura 126: Comparación de algoritmos de ajuste de hiperparámetros por modelo y arquitectura utilizando como métrica MAE.

MASE

Es arduo crear una función de pérdida MASE ya que ésta recibe como parámetro muestras previas, añadiendo una complejidad que no fue posible superar en su implementación (*Keras* no soportaba esta característica). En consecuencia utilizamos **MAE** como métrica para entrenar nuestros modelos, sin embargo **MASE** es nuestra principal métrica a la hora de definir cual es el mejor modelo para el objetivo propuesto.

Concluimos que la combinación que posee el mejor rendimiento en nuestro *set* de datos es **CNN** con Optimización Bayesiana (tabla 9 y fig. 127).

algo/model	CNN	CNNGRU	CNNLSTM	GRU	LSTM
<i>BayesianOptimization</i>	0,208	0,300	0,306	0,243	0,297
<i>Hyperband</i>	0,253	0,332	0,290	0,628	0,304

Cuadro 9: Mejor resultado por algoritmo y arquitectura utilizando como métrica MASE.

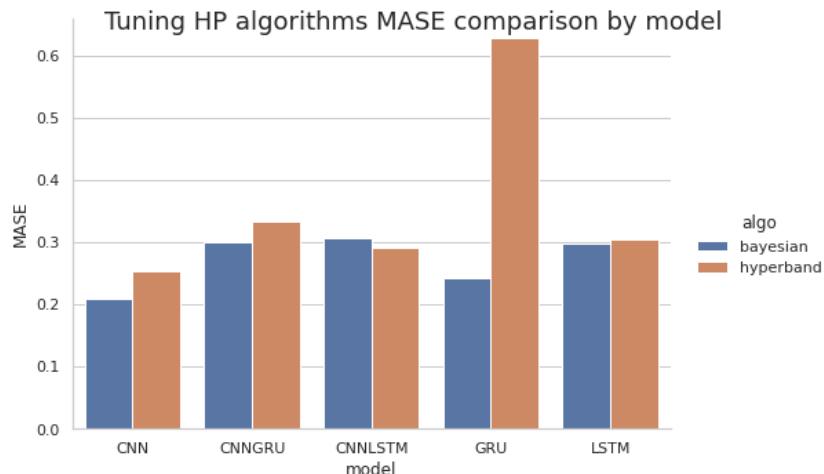


Figura 127: Comparación de algoritmos de ajuste de hiperparámetros por modelo y arquitectura utilizando como métrica MAE.

En la fig. 128 podemos notar la excelente precisión del modelo al predecir los instantes $t+1$, justificando de manera visual el análisis realizado.

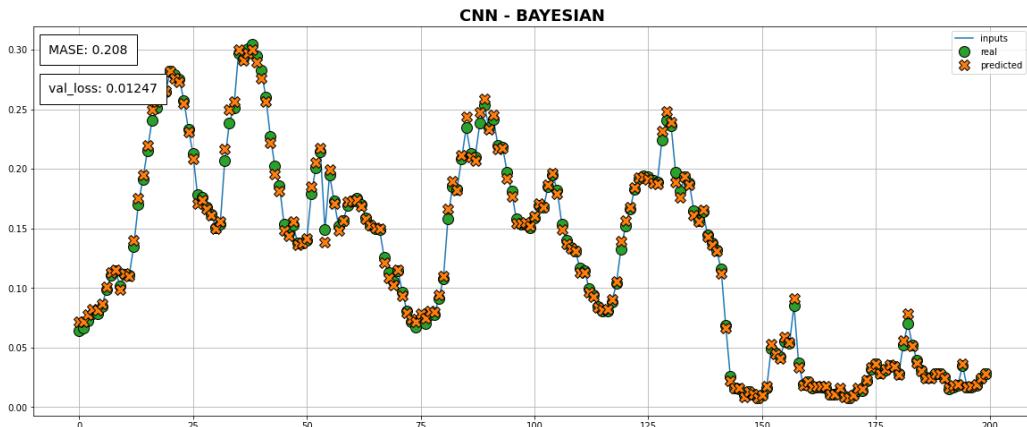


Figura 128: Pronóstico del mejor modelo una vez realizado el ajuste de hiperparámetros.

Para destacar la utilidad del ajuste de hiperparámetros recordemos la tabla 5. En la misma notamos que el mejor modelo (CNNGRU) logró un MASE de 0.278 mientras que el modelo (CNN) de mejor rendimiento una vez aplicado el ajuste de hiperparámetros obtuvo 0.208, una mejora de casi 8 puntos. También es interesante notar que antes del ajuste de hiperparámetros los modelos CNN estaban muy lejos de los mejores rendimientos, pero luego se posicionaron como la mejor opción.

11.5.4 Conclusión

Si bien el ajuste de hiperparámetros es más costoso computacionalmente que el entrenamiento de los modelos, si la necesidad radica en obtener el mejor modelo posible con nuestros recursos definitivamente es un paso indispensable.

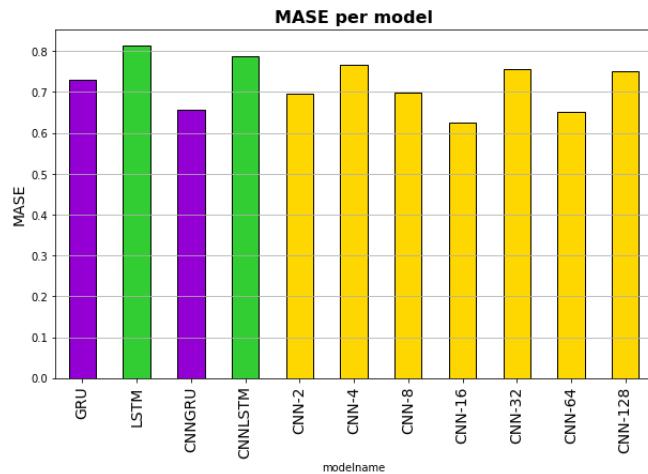
11.5.5 Resultados *dataset FSI*

Los mismos procedimientos realizados sobre nuestra *dummy data* serán aplicados a nuestro *dataset* de FSI.

Los modelos obtenidos sin búsqueda de hiperparámetros arrojan los resultados de la tabla 10 que son interpretados visualmente en la fig. 129. Si comparamos los gráficos entre ambos *datasets*, tenemos un rendimiento 8 veces peor en promedio (MAE) que con el *dataset* de polución, esto claramente repercutirá en la precisión de pronóstico de nuestros modelos.

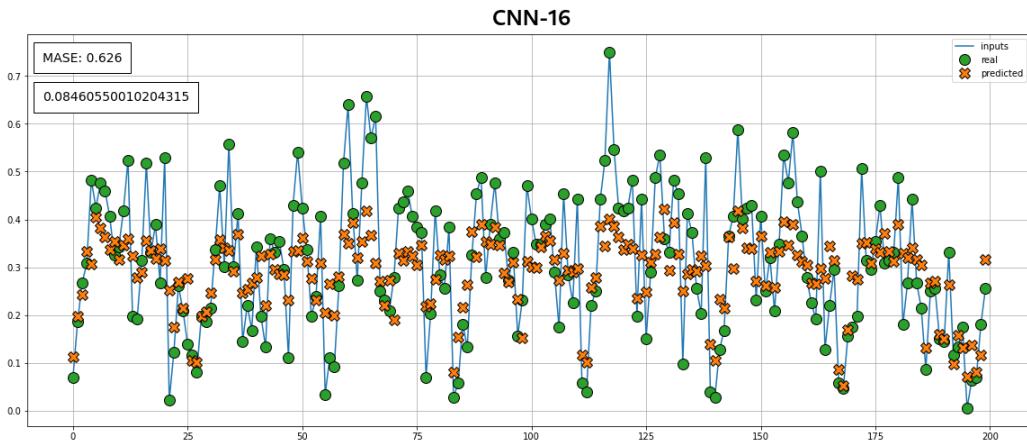
model	val_loss	MASE
GRU	0.0874	0.7308
LSTM	0.0891	0.8138
CNNGRU	0.0865	0.6575
CNNLSTM	0.0841	0.7877
CNN-1	0.0867	0.6812
CNN-2	0.0891	0.6949
CNN-4	0.0888	0.7675
CNN-8	0.0844	0.6984
CNN-16	0.0846	0.6260
CNN-32	0.0839	0.7574
CNN-64	0.0847	0.6516
CNN-128	0.0878	0.7501

Cuadro 10: Métricas de los modelos para el *dataset* FSI.

Figura 129: MASE por modelo con *dataset* FSI.

El mejor modelo con un MASE de 0.626 es CNN-16 seguido de cerca por CNNGRU. Se observa que los modelos LSTM funcionan peor que los demás pero no podemos sacar conclusiones claras aún.

De la fig. 130 notamos que a nuestro modelo no tiene problemas en predecir mínimos pero le es casi imposible predecir máximos debido a los cambios abruptos de la serie temporal.

Figura 130: Mejor modelo para el *dataset* FSI.

Obtenemos la tabla 7 (recordar que se utiliza como métrica MAE). De la misma podemos sacar varias conclusiones interesantes.

- El modelo con la mejor puntuación fue CNNGRU en combinación con el algoritmo *HyperBand*, y en segundo lugar (*LSTM/HyperBand*) apenas por detrás.
- 2 capas de CNN fueron suficientes, e incluso podría haberse utilizado solo una.
- Las unidades de las capas RNN varían demasiado sin definir un rango de valores claros.
- Sin lugar a dudas la mejor función de activación fue la sigmoide.

model	algo	layers	CNN				MAXPOOL		RNN	DENSE	score
			f_0	ks_0	f_1	ks_1	f_2	ks_2	p_s	units	
CNNGRU	HB	1	56	4	24	6	40	4	6	120	sig 0.080426
CNNGRU	HB	2	56	4	40	4	40	4	6	200	sig 0.080560
CNNLSTM	HB	2	56	2	56	6	24	6	4	168	sig 0.080621
CNNGRU	HB	1	56	6	24	6	56	6	8	184	sig 0.080632
CNNLSTM	HB	1	40	8	24	4	24	6	8	232	sig 0.080693
CNNLSTM	HB	1	24	4	56	4	24	8	6	152	sig 0.080752
CNNLSTM	HB	1	56	6	56	2	56	4	8	248	sig 0.080915
CNNLSTM	HB	1	24	4	40	6	24	4	6	184	sig 0.080979
CNNGRU	HB	2	56	6	24	4	56	4	8	248	sig 0.081028
CNNGRU	HB	1	40	8	40	2	8	6	8	200	sig 0.081080

Cuadro 11: Resultados de ajuste de hiperparámetros del *dataset* FSI.**MAE**

Para esta tarea utilizaremos de soporte la tabla 12 y la fig. 131. Excepto GRU, *Hyperband* obtiene un mejor rendimiento en cada uno de los modelos restantes.

algo/model	CNN	CNNGRU	CNNLSTM	GRU	LSTM
<i>BayesianOptimization</i>	0.08309	0.08138	0.08138	0.08558	0.08346
<i>Hyperband</i>	0.08196	0.08043	0.08062	0.09082	0.08389

Cuadro 12: Mejor puntuación por algoritmo y arquitectura utilizando MAE como métrica en *dataset* FSI.

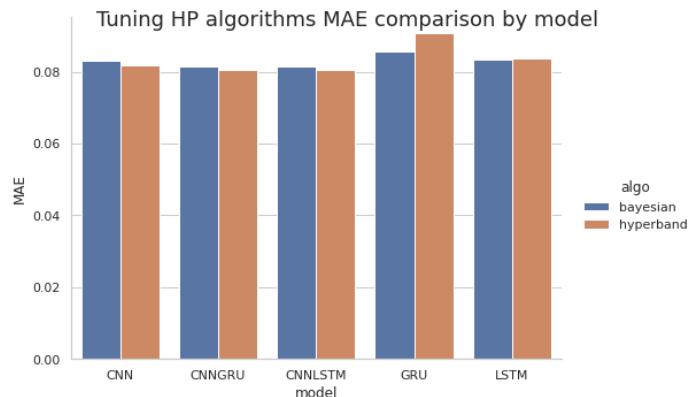


Figura 131: Comparación de algoritmos de ajuste de hiperparámetros por modelo y arquitectura utilizando como métrica MAE.

MASE

Utilizando la tabla 13 y la fig. 132 concluimos que la combinación que posee el mejor rendimiento en nuestro *set* de datos es CNN con *HyperBand*. En el *dataset* de polución el mejor rendimiento también se obtuvo a través de una CNN pero el algoritmo de ajuste de hiperparámetros fue *Bayesian*. Aunque en nuestros dos *datasets* hayamos obtenido que el mejor modelo es CNN no se puede asegurar que no habrá arquitecturas que funcionen mejor con otras series temporales.

algo/model	CNN	CNNGRU	CNNLSTM	GRU	LSTM
<i>BayesianOptimization</i>	0.661	0.659	0.707	0.730	0.754
<i>Hyperband</i>	0.627	0.707	0.704	0.744	0.752

Cuadro 13: Mejor resultado por algoritmo y arquitectura utilizando como métrica MASE en *dataset* FSI.

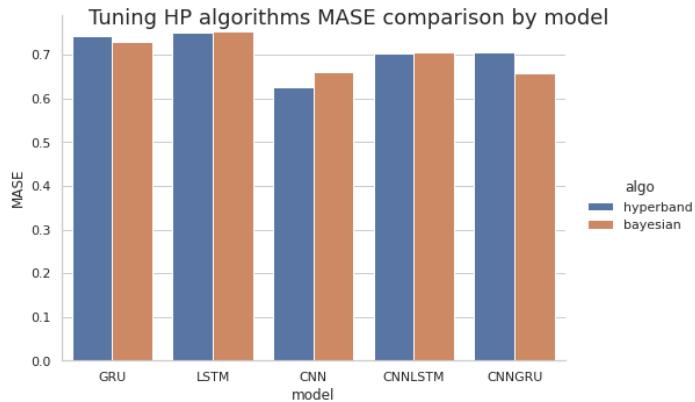


Figura 132: Comparación de algoritmos de ajuste de hiperparámetros por modelo y arquitectura utilizando como métrica MASE para el *dataset* FSI.

En la fig. 133 resaltamos lo antes mencionado, nuestro modelo carece de la capacidad de predecir los valores máximos de la función, pero puede predecir los valores mínimos. Si pensamos a cada capa como un amplificador podemos intuir que quizás una de las mismas se esté saturando incapacitando a las siguientes a llegar a los valores máximos. Se realizará un análisis ajustando los datos de entrada al modelo con diferentes *scalers*.

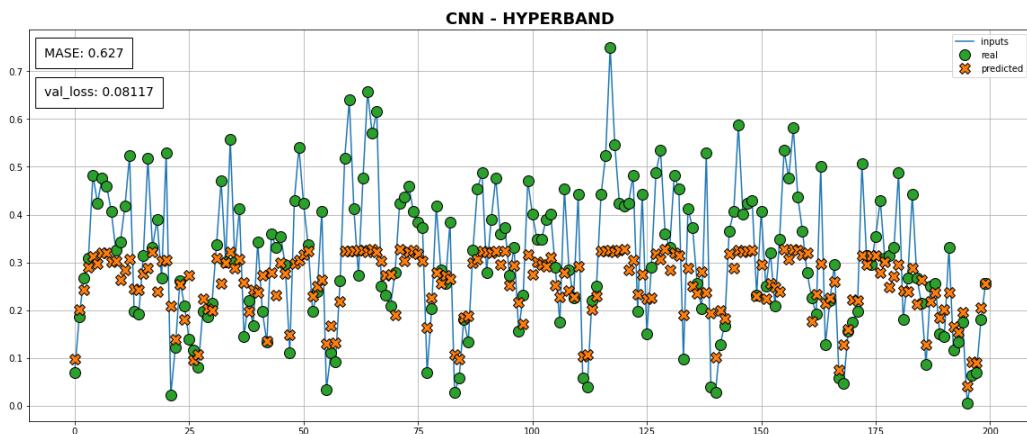


Figura 133: Pronóstico del mejor modelo para el *dataset* FSI una vez realizado el ajuste de hiperparámetros.

Este inconveniente lo hace disfuncional a nuestra tarea pero en la próxima sección intentaremos descubrir que diferencia a ambos *datasets* para que uno

funcione tan bien y el otro no.

Comparemos los valores antes y después del ajuste de hiperparámetros, en la tabla 10 obtuvimos con CNN-16 un MASE de 0.626 y la tabla 13 nos dice que el mejor modelo fue un CNN con MASE de 0.627 *i.e.* valores casi idénticos pero además peores para el post-ajuste.

11.5.6 Cambio en preprocessamiento de datos

Como mencionamos anteriormente, para evitar la saturación en nuestra red variaremos forma en que la misma recibe nuestro *dataset* modificando dos parámetros:

- *Scaler*: algoritmo de preprocessamiento de datos cuyo objetivo usualmente es lograr un *dataset* más homogéneo y sin *outliers*.
- Factor de escala (s_f): simplemente multiplicaremos toda la red por este valor para obtener amplitudes menores.

Scikit-learn nos provee de varios *scalers* pero seleccionamos los siguientes:

- **MinMax**: escala cada característica individualmente de modo que se encuentre en el rango seleccionada (*e.g.* 1 y 0).
- **PowerTransformer**: aplica una transformación de potencia en función de las características para hacer que los datos sean más *gaussianos*.
- **RobustScaler**: escala características usando estadísticas que sean robustas a *outliers*.
- **StandardScaler**: estandariza las características eliminando la media y escalando a la varianza unitaria.

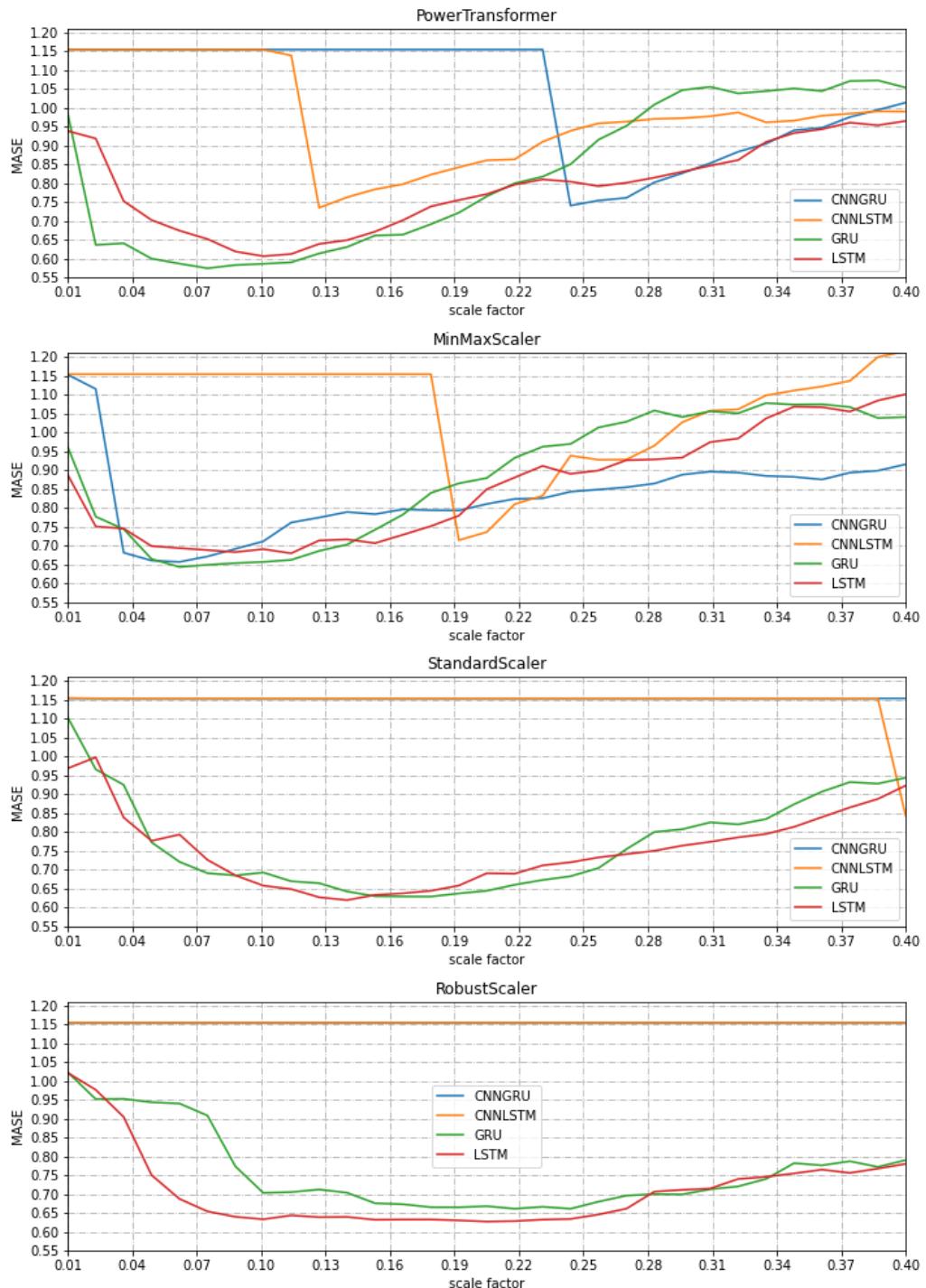
Para el factor de escala se definió un rango entre 0.01 y 0.40 tras varios ensayos.

En la tabla 14 se corrobora que el *scaler* de mejor rendimiento es **PowerTransformer** (notar los bajos valores de s_f). Con respecto a nuestros resultados originales (en MASE) se obtienen casi 5 puntos de mejora.

<i>scaler</i>	s_f	model	MASE
PowerTransformer	0.075	GRU	0.5749
PowerTransformer	0.088	GRU	0.5836
PowerTransformer	0.101	GRU	0.5868
PowerTransformer	0.062	GRU	0.5873
PowerTransformer	0.114	GRU	0.5907
PowerTransformer	0.049	GRU	0.6004
PowerTransformer	0.101	LSTM	0.6070
PowerTransformer	0.114	LSTM	0.6128
PowerTransformer	0.127	GRU	0.6142
PowerTransformer	0.088	LSTM	0.6194

Cuadro 14: Top 10 de mejores *scalers* según MASE obtenido.

Los modelos RNN puros funcionan mucho mejor que los que poseen algún componente CNN (fig. 134), una posibilidad es que las capas CNN se sobrecarguen haciendo que el modelo diverja. Además según el *scaler* hay determinadas zonas de valle en las cuales los modelos funcionan mejor. Lo recomendable sería una vez determinado el mejor par *scaler*-zona de mayor rendimiento, explorar utilizando ajuste de hiperparámetros.

Figura 134: Factor de escala *vs* MASE según *scaler*.

Los resultados son esclarecedores, si bien seguimos sin llegar a los picos ahora al menos nos aproximamos (fig. 135) y no tenemos ese techo que no nos permitía escalar (fig. 133). Otra manera de mejorar el rendimiento de nuestros modelos puede ser variar las configuraciones del preprocesamiento de los datos. Sin embargo esto es tedioso y arduo de automatizar.

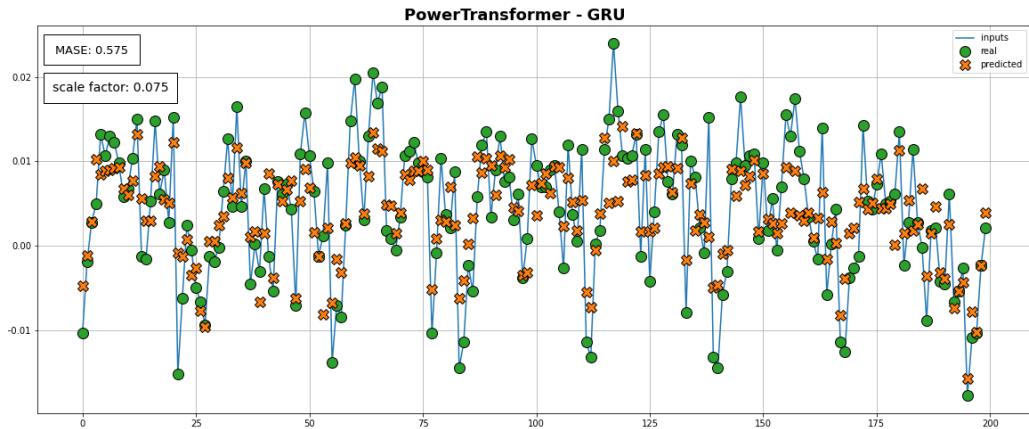


Figura 135: Prónostico utilizando como *scaler* PowerTransformer y modelo GRU.

11.5.7 Conclusión

No siempre aplicar ajuste de hiperparámetros nos dará resultados mejores a los originales, ya que esto dependerá de cada serie temporal a pronosticar. Pero si intuimos lo que puede estar sucediendo en la red podemos desarrollar soluciones innovadoras.

11.6 Entropía

Intentaremos encontrar una causa al dispar rendimiento entre nuestro *dataset* de polución del aire y el de defectos. Para ello haremos uso de lo revisado en la sección 9, dónde mencionamos que la entropía de los datos influye directamente en la capacidad de predicción de nuestros modelos. Mientras más entropía tengan los datos, más difícil será que el modelo pueda aprender correctamente sus patrones, si es que los hubiere.

Ambos *datasets* serán procesados por los dos algoritmos disponibles para esta tarea: ApEn y SampEn.

11.6.1 Rendimiento de los algoritmos

Al realizar los ensayos quedó al descubierto la demora de los algoritmos para ejecutar los cálculos sobre la totalidad del *dataset*, lo que llevó a buscar una solución. *Python* por defecto no posee librerías que hagan uso intensivo de GPU (un caso es *scikit-learn*, que a pesar de brindar diversos algoritmos de Aprendizaje Automático sólo utiliza CPU para sus cálculos) y *NumPy* (librería por defecto para operaciones matriciales y tensoriales) no es la excepción.

CuPy

Es una biblioteca de matrices de código abierto que proporciona computación acelerada por GPU con *Python* a través de CUDA. [54]

CUDA es una plataforma de computación paralela y un modelo de programación desarrollado por *Nvidia* para computación general en unidades de procesamiento gráfico (GPU). Con CUDA, los desarrolladores pueden acelerar drásticamente las aplicaciones informáticas aprovechando la potencia de las GPU. [55] La implementación de CuPy sobre código creado para la librería NumPy es muy simple.

Comparación de rendimiento

Se divisa que a medida que la cantidad de muestras aumenta, el tiempo de ejecución de NumPy crece exponencialmente en contraste con CuPy que crece de forma lineal (fig. 136). En el caso límite que el algoritmo utiliza la totalidad del *dataset* llegamos a tener un *speedup* de 8X. Por tanto concluimos que CuPy promete lo que cumple.

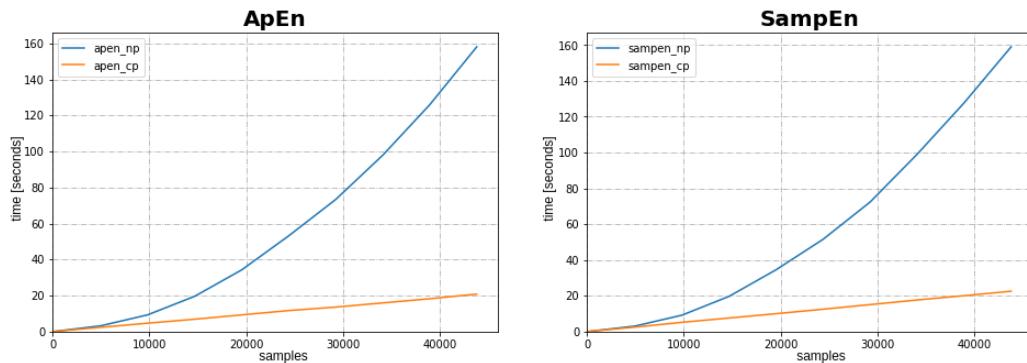
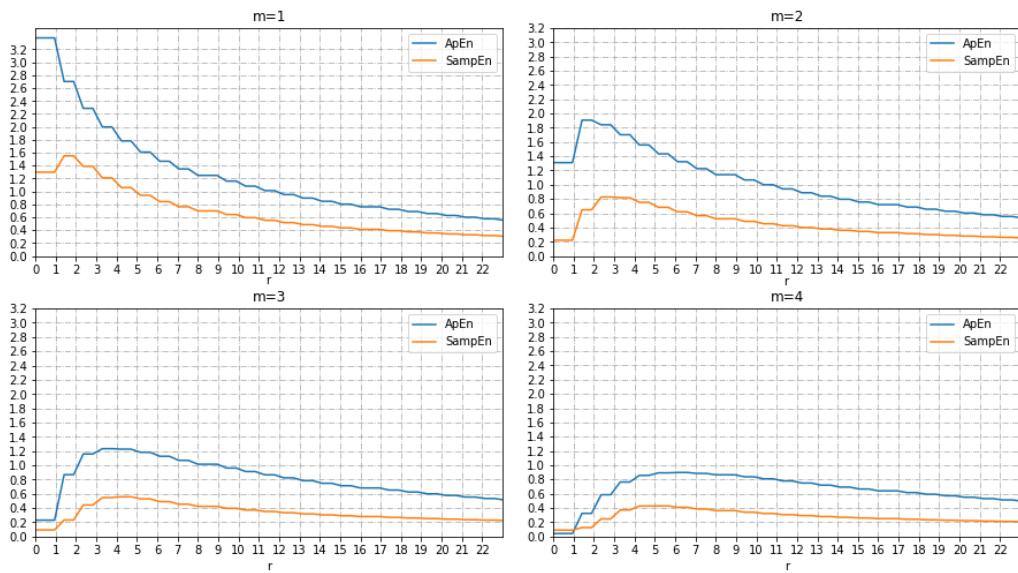


Figura 136: Comparación de NumPy vs CuPy.

11.6.2 Dataset de polución

Vamos a calcular ApEn y SampEn sobre el *dataset* de polución con $0 \leq m \leq 4$ y $0 \leq r \leq 0.25$ (recordar que éste valor debe ser multiplicado por la desviación estándar de la serie temporal σ). Los resultados obtenidos son visualizados en la fig. 137.

Figura 137: Gráfico de la entropía del *dataset* de polución.

Podemos deducir que tenemos un pico de entropía claro en cada uno de las figuras, que a medida que m aumenta este pico se va desplazando hacia la derecha aumentando el valor de r que lo intersecta.

11.6.3 Dataset FSI

Sin embargo, esta información carece de valor si no la comparamos con nuestro *dataset* principal, por tanto observemos los resultados obtenidos para el mismo en la fig. 139.

La forma escalonada se debe a la baja densidad de datos en comparación al otro *dataset*. Si bien se divisa que los valores de ApEn y SampEn a priori parecen mayores, no podemos sacar conclusiones claras por lo que se requiere otro tipo de análisis.

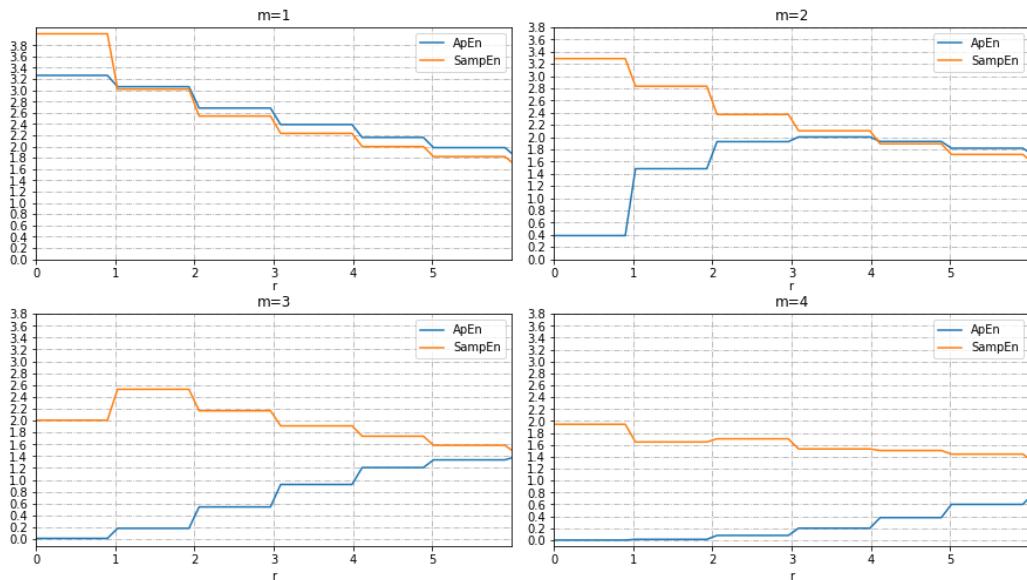


Figura 138: Gráfico de la entropía del *dataset* FSI.

11.6.4 Comparación de *datasets*

Tomaremos el mayor resultado de cada m de cada modelo y algoritmo, conformando la tabla 15. Interpretaremos estos valores a través de la fig. 139, que nos muestra que según ApEn ambas series temporales muestran una entropía similar, cosa que podemos desmentir al visualizar los ploteos de cada una.

Sin embargo, SampEn nos ayuda a exponer de manera empírica que la entropía de la serie temporal de FSI es mayor a la de polución. En este caso SampEn funcionó mejor que ApEn a la hora de comparar la entropía de ambas series temporales.

m	score	dataset	algo
1	3.264	fsi	ApEn
2	2.003	fsi	ApEn
3	1.388	fsi	ApEn
4	0.777	fsi	ApEn
1	4.003	fsi	SampEn
2	3.286	fsi	SampEn
3	2.527	fsi	SampEn
4	1.946	fsi	SampEn
1	3.377	pollution	ApEn
2	1.906	pollution	ApEn
3	1.233	pollution	ApEn
4	0.897	pollution	ApEn
1	1.551	pollution	SampEn
2	0.828	pollution	SampEn
3	0.558	pollution	SampEn
4	0.427	pollution	SampEn

Cuadro 15: Tabla para comparación de *datasets*.

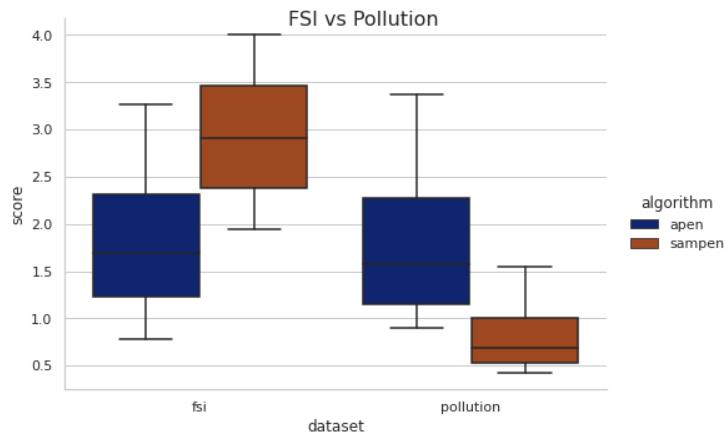


Figura 139: Gráfico de la entropía del *dataset* FSI.

12 Conclusiones

El proceso de construcción del sistema de predicción de defectos, requirió el estudio y selección de las técnicas de inteligencia artificial apropiadas para la ejecución del proyecto, las cuales luego de un cuidadoso estudio se identificaron que las más aptas eran redes neuronales de dos tipos: convolucionales y recurrentes.

La recolección de datos significó una ardua tarea ya que los mismos se encontraban dispersos en diversas fuentes. Con el análisis de datos, se identificaron que se montaron un total de 10780 vehículos, con una media de 19.6 defectos. Del total un 14.7% pertenece a defectos graves y el resto a defectos leves. Por otro lado, se identificó que el departamento con mayor cantidad de defectos con más del 50% es **PINTURA** y dentro de las familias fue **ASPECTO** con casi un 70%. Por lo cual se distingue una gran correlación entre estas dos variables.

La mayor proporción de defectos graves fue en la combinación **MONTAJE-FALTANTES**, denotando que la cantidad de defectos no está íntimamente relacionado con la gravedad de los mismos.

Las técnicas de inteligencia artificial evaluadas originalmente si bien no fueron seleccionadas para el modelo definitivo, fueron de utilidad para imputar datos faltantes en las columnas **TIPO** y **UET** con una precisión del 99.73% y 79.74% respectivamente.

Se especificó la arquitectura para cada modelo y las métricas para su evaluación. Cada modelo se entrenó con datos de prueba y datos objetivo para contrastar y verificar su correcta ejecución. El mínimo error obtenido para los datos de prueba utilizando como métrica **MASE** arrojó como resultado 27.75% para un modelo convolucional-recurrente. Visualmente se constata que el modelo es capaz de predecir con exactitud los valores objetivo. Para los datos objetivo el error crece a 62.6% siendo esto reflejado en los gráficos.

Se aplicó búsqueda de hiperparámetros a cada modelo para acceder a la máxima eficacia posible, limitando cada configuración a un espacio de búsqueda predefinido. El modelo convolucional-causal a través de hiperparámetros obtenidos mediante búsqueda bayesiana disminuyó el error a 20.80% en los datos de prueba.

Los modelos para los datos objetivo pese a que diversas estrategias fueron puestas a prueba en el preprocesamiento de datos solo lograron reducir el error en un pequeño porcentaje. El modelo recurrente con un *scaler PowerTransformer* obtuvo el mejor desempeño con un error equivalente a 57.5%.

Si bien los modelos de redes neuronales son potentes para el pronóstico de series temporales, es factible que su rendimiento se vea afectado por la calidad de los datos. Un modo de estimar este aspecto es a través del cálculo de la entropía de la serie temporal, que nos brinda indicios de la condición de nuestros datos sin preocuparse por su origen. Consecuentemente se comprobó que la entropía en los datos objetivo era superior a la de los datos de prueba.

El despliegue del modelo se enfrentaba a diversos inconvenientes técnicos. Uno de ellos fue la incapacidad de disponibilizar los datos en tiempo real debido a la necesidad de implementar sistemas de bases de datos de mayor accesibilidad y rendimiento. Inclusive era imprescindible construir una nueva infraestructura para soportar el *pipeline* de datos, incurriendo en costos que la organización no consideraba solventar. En base a los puntos considerados previamente, desde una etapa temprana se determinó que este documento revestiría carácter de investigación.

Para posibles trabajos futuros se sugiere tener en cuenta las siguientes recomendaciones:

1. Explorar en mayor profundidad el preprocesamiento de los datos constatando la repercusión de estos cambios en el desempeño del modelo.
2. Si la organización lo requiere, incluir metadata externa a nuestros datos originales para así obtener un *dataset* más diverso.
3. Experimentar con otras herramientas *open-source* disponibles para el pronóstico de series temporales.

Desde un punto de vista funcional, los resultados han sido satisfactorios para el comitente, ya que los modelos cuentan con rendimientos adecuados para la naturaleza del problema abordado. En lo personal todo el recorrido a través del campo *Machine Learning* ha sido muy satisfactorio, adquiriendo habilidades sobre herramientas muy valoradas profesionalmente.

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] Cognethi. C 4.1 | 1D Convolution | CNN | Object Detection | Machine Learning | EvODN, Aug 2019. URL https://www.youtube.com/watch?v=yd_j_zdLDWs. [Online; accessed 4. Jan. 2021].
- [13] StackOverflow. How did they calculate the output volume for this convnet example in Caffe?, Jan 2021. URL <https://stackoverflow.com/questions/32979683/how-did-they-calculate-the-output-volume-for-this-convnet-example-in-caffe>. [Online; accessed 4. Jan. 2021].
- [14] Louis N Andrianaivo, Roberto D'Autilia, and Valerio Palma. Architecture recognition by means of convolutional neural networks. *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, 2019.
- [15] Sumit Saha. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. *Medium*, Oct 2020. ISSN 3211-6453. URL <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b116>
- [16] 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] Will Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. Towards Data Science, India, Jul 2018. ISBN 978-817227805. URL <https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization>
- [26] Andre Ye. The Beauty of Bayesian Optimization, Explained in Simple Terms. *Medium*, Oct 2020. URL <https://towardsdatascience.com/the-beauty-of-bayesian-optimization-explained-in-simple-terms-81f3ee13b10f>.
- [27] Colaboradores de los proyectos Wikimedia. Vientre de alquiler (práctica) - Wikipedia, la enciclopedia libre, Mar 2021. URL [https://es.wikipedia.org/w/index.php?title=Vientre_de_alquiler_\(pr%C3%A1ctica\)&oldid=134152025](https://es.wikipedia.org/w/index.php?title=Vientre_de_alquiler_(pr%C3%A1ctica)&oldid=134152025). [Online; accessed 4. Apr. 2021].
- [28] Apoorv Agnihotri and Nipun Batra. Exploring Bayesian Optimization. *Distill*, 5(5):e26, May 2020. ISSN 2476-0757. doi: 10.23915/distill.00026.
- [29] Wei Wang. Bayesian Optimization Concept Explained in Layman Terms. *Medium*, Apr 2020. URL <https://towardsdatascience.com/bayesian-optimization-concept-explained-in-layman-terms-1d2bcdeaf12f>.
- [30] James Bergstra, R. Bardenet, Balázs Kégl, and Y. Bengio. Algorithms for hyper-parameter optimization. 12 2011.
- [31] BOHB: Robust and Efficient Hyperparameter Optimization at Scale, Apr 2021. URL https://www.automl.org/blog_bohb. [Online; accessed 5. Apr. 2021].

- [32] Aloïs Bissuel. *Hyper-parameter optimization algorithms: a short review.* Criteo R&D Blog, Apr 2019. ISBN 978-244752590. URL <https://medium.com/criteo-engineering/hyper-parameter-optimization-algorithms-2fe447525903>.
- [33] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization, 2018.
- [34] Noam (xn-45q. xn-9kq.) Rosenberg. Hyper Parameter Tuning — A Tutorial - Towards Data Science. *Medium*, Aug 2020. ISSN 7065-5254. URL <https://towardsdatascience.com/hyper-parameter-tuning-a-tutorial-70dc6c552c54>.
- [35] Alexandre Abraham. A (Slightly) Better Budget Allocation for Hyperband | data from the trenches. *Medium*, Feb 2021. URL <https://medium.com/data-from-the-trenches/a-slightly-better-budget-allocation-for-hyperband-bbd45af14481>.
- [36] Shubham Agrawal. Time Series Modeling, Mar 2019. URL https://rstudio-pubs-static.s3.amazonaws.com/478704_10185309918043d4a279e1e1545694e0.html. [Online; accessed 5. Apr. 2021].
- [37] Prophet, Jan 2021. URL <https://facebook.github.io/prophet>. [Online; accessed 8. Feb. 2021].
- [38] GluonTS - Probabilistic Time Series Modeling — GluonTS documentation, May 2020. URL <https://ts.gluon.ai>. [Online; accessed 8. Feb. 2021].
- [39] alan-turing institute. sktime, Feb 2021. URL <https://github.com/alan-turing-institute/sktime>. [Online; accessed 8. Feb. 2021].
- [40] Alfonso Delgado-Bonal and Alexander Marshak. Approximate entropy and sample entropy: A comprehensive tutorial. *Entropy*, 21(6):541, 2019.
- [41] (77) Gregory Chaitin | Universidad de Buenos Aires - Academia.edu, Apr 2021. URL <https://uba.academia.edu/GregoryChaitin>. [Online; accessed 12. Apr. 2021].
- [42] Aurélien Géron. A Short Introduction to Entropy, Cross-Entropy and KL-Divergence, Feb 2018. URL <https://www.youtube.com/watch?v=ErfnhcEV108>. [Online; accessed 13. Apr. 2021].

- [43] ProClassify User's Guide - Cross-Validation Explained, Jun 2006. URL <https://genome.tugraz.at/proclassify/help/pages/XV.html>. [Online; accessed 1. Feb. 2021].
- [44] 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].
- [45] 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].
- [46] Ashish Ahuja. Mean Absolute Scaled Error (MA-SE) in Forecasting - Ashish Ahuja - Medium. *Medium*, Jan 2021. URL <https://medium.com/@ashishdce/mean-absolute-scaled-error-mase-in-forecasting-8f3aec21968>.
- [47] Papers with Code - Papers With Code : Trends, Jul 2021. URL <https://paperswithcode.com/trends>. [Online; accessed 12. Jul. 2021].
- [48] 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].
- [49] Keras Team. Keras documentation: Callbacks API, Mar 2021. URL <https://keras.io/api/callbacks>. [Online; accessed 23. Mar. 2021].
- [50] Azure. DeepLearningForTimeSeriesForecasting, Mar 2021. URL https://github.com/Azure/DeepLearningForTimeSeriesForecasting/blob/master/2_RNN.ipynb. [Online; accessed 29. Mar. 2021].
- [51] Azure. DeepLearningForTimeSeriesForecasting, Mar 2021. URL https://github.com/Azure/DeepLearningForTimeSeriesForecasting/blob/master/3_RNN_encoder_decoder.ipynb. [Online; accessed 29. Mar. 2021].
- [52] Best Tools for Model Tuning and Hyperparameter Optimization - neptune.ai, May 2021. URL <https://neptune.ai/blog/best-tools-for-model-tuning-and-hyperparameter-optimization>. [Online; accessed 12. Jul. 2021].

- [53] Keras Tuner, Nov 2019. URL <https://keras-team.github.io/keras-tuner>. [Online; accessed 5. Apr. 2021].
- [54] CuPy, Apr 2021. URL <https://cupy.dev>. [Online; accessed 16. Apr. 2021].
- [55] CUDA Zone, Apr 2021. URL <https://developer.nvidia.com/cuda-zone>. [Online; accessed 16. Apr. 2021].

