



UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Facultad de Ciencias

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y
MATEMÁTICAS

TRABAJO DE FIN DE GRADO

Análisis y modelado de series temporales con métodos estadísticos y basados en Deep Learning

Presentado por:

Miguel Lentisco Ballesteros

Tutores:

José Manuel Benítez Sánchez

Ciencias de la Computación e Inteligencia Artificial

Miguel Lastra Leidinger

Lenguajes y Sistemas Informáticos

Curso académico 2019-2020

Análisis y modelado de series temporales con métodos estadísticos y basados en Deep Learning

Miguel Lentisco Ballesteros

Miguel Lentisco Ballesteros *Análisis y modelado de series temporales con métodos estadísticos y basados en Deep Learning.*

Trabajo de fin de Grado. Curso académico 2019-2020.

**Responsables de
tutorización**

José Manuel Benítez Sánchez
*Ciencias de la Computación e Inteligencia
Artificial*

Miguel Lastra Leidinger
Lenguajes y Sistemas Informáticos

Doble Grado en Ingeniería
Informática y Matemáticas

Escuela Técnica Superior
de Ingenierías Informática
y de Telecomunicación
Facultad de Ciencias

Universidad de Granada

DECLARACIÓN DE ORIGINALIDAD

D. Miguel Lentisco Ballesteros

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2019-2020, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 12 de agosto de 2020

Fdo: Miguel Lentisco Ballesteros

Resumen

Este trabajo está dirigido principalmente a la investigación de problemas poco tratados e investigados, donde aún queda mucho por explorar, en el área de series temporales con modelos de redes neuronales LSTM. Nos centraremos en tres problemas bien diferenciados: selección de modelos, detección de anomalías y predicción de series discretas. Antes de abordarlos, veremos una introducción para tener un entendimiento básico del funcionamiento de las redes neuronales y concretamente de las redes LSTM. Después repasaremos los conceptos sobre series temporales más generales que serán el dominio de nuestros problemas. Finalmente acabamos nuestros conceptos previos con las métricas que usaremos para valorar los modelos utilizados. Ahora sí pasaremos a los problemas: en la selección de modelos investigaremos una nueva heurística frente a la validación clásica y discutiremos su efectividad. Para la detección de anomalías ofrecemos un detector sencillo pero funcional junto a unos métodos para crear instancias anómalas a partir de las normales. Finalmente estudiaremos la predicción de series temporales discretas obteniendo éstas de series continuas mediante métodos de discretización.

PALABRAS CLAVE redes neuronales LSTM series temporales selección de modelos validación hiperparámetros detección de anomalías detector perturbación series temporales discretas predicción discretización

Summary

The main goal of this project is to study and research problems where there have been little research on, focusing on the domain of time series and resolving these problems using models based on LSTM neural networks architectures.

CHAPTER 1 We present a brief background of the project with an introduction to its basis, the motivation that led to its realization, objectives to accomplish and the structure of this document.

CHAPTER 2 First of all, we need to introduce to the reader the basic concepts that we are going to use in the development of the main parts. We start in the DeepLearning area with a short tour in the neural networks' history to learn about its origins that lead to the most basic but functional neural network: the feed-forward neural network. We explain in detail how this model works allowing us to understand the majority of neural networks models used nowadays, as these networks share the fundamentals to a large degree of similarity. Finally we list a few most well-known neural network architectures, focusing in detail at the LSTM cells.

CHAPTER 3 About time series [EN CONSTRUCCIÓN]

CHAPTER 4 Finally, we make a quick review of the metrics that we are going to use to validate the models developed in the project: for classification problems the accuracy, F-score and the precision-recall curves; for regression the usual error metrics like mean squared error or mean absolute error.

PART I The first part of this project is about model selection in classification problems that is classically done with the cross-validation and the hold-out-test validation. We study a new heuristic called Perturbated Validation that do not use a train/test partition but introduces tiny perturbations and measures how the model behaves to them. We experiment with several machine learning models, including a LSTM neural network in a big dataset of time series, both with model selection and hyperparameters selection, and we analyze the results to validate the effectiveness of this heuristic.

PART II In this part we address the anomaly detection problem: detecting atypical behavior in time series. We comment about the big obstacles in this type of problems: the lack of datasets with labeled data, for its use in the training of models based on supervised learning; and the models developed are too specific for the problem they are solving thus they can not be deployed in other areas. We provide a simple yet useful model based in LSTM architecture that can detect several types of anomalies, provided a well known pattern in the data, and several methods to modify the normal series in order to create artificially samples of anomalies that we use to validate our model. We evaluate the detector and the methods

using them with both a real and syntetic dataset, analyzing the adjust of the detector and the quality of the perturbations.

PART III Finally we consider an area that has not been very worked on: prediction of discrete time series. First of all, we study a few methods to transform continous to discrete series, analyzing its results. Then we try to predict this series using LSTM neural networks and study its performance.

KEYWORDS neural networks LSTM time series model selection validation
hyperparameters anomaly detection detector perturbation discrete time series prediction discretization

Índice general

Índice de figuras	15
Índice de tablas	19
1. Introducción	21
1.1. Motivación	21
1.2. Objetivos	22
1.3. Estructura	22
2. Aprendizaje profundo	23
2.1. Origen	23
2.1.1. Neurona de McCulloch-Pitts	23
2.1.2. Perceptrón	24
2.1.3. Perceptrón MultiCapa	26
2.2. Redes Neuronales Hacia Adelante	29
2.2.1. Descripción	29
2.2.2. Estructura	29
2.2.3. Propagación hacia adelante	32
2.2.4. Gradiente Descendente	33
2.2.5. Propagación hacia atrás	35
2.2.6. Error y sobreajuste	37
2.2.7. Teorema de aproximación universal	40
2.3. Clasificación	42
2.3.1. Aprendizaje	42
2.3.2. Arquitectura	42
2.3.3. Redes Neuronales Convolucionales	43
2.3.4. Autocodificador	45
2.3.5. Redes Generativas Antagónicas	46
2.3.6. Redes Neuronales Recurrentes	47
2.4. LSTM	49
2.4.1. Estructura general	49
2.4.2. Variantes	52
2.4.3. Recorte	54
3. Series Temporales	55
3.0.1. Descomposición STL	55
4. Métricas	57
4.1. Clasificación	57
4.1.1. Etiquetas	57
4.1.2. Probabilidades	58
4.2. Regresión	60

Índice general

I. Selección de modelos	61
5. Introducción	63
6. Selección de modelos	65
6.1. Selección clásica	65
6.2. Perturbated Validation	66
6.2.1. Funcionamiento	66
6.2.2. Definición	68
6.3. Implementación	68
7. Experimentación	71
7.1. Modelos	71
7.1.1. LSTM	71
7.1.2. SVM	73
7.1.3. Árboles de decisión	73
7.1.4. Random Forest	74
7.1.5. k -NN	75
7.2. Experimentación previa	76
7.3. Elección del mejor modelo	77
7.3.1. Descripción	77
7.3.2. Resultados	81
7.3.3. Análisis	85
7.4. Hiperparametrización	88
7.4.1. Descripción	88
7.4.2. Resultados	89
7.4.3. Análisis	89
8. Conclusiones y trabajo futuro	95
II. Detección de Anomalías	97
9. Introducción	99
10. Sistema LSTM para detección de series anómalas	101
10.1. Autoencoder LSTM	101
10.2. Detección	102
11. Alteraciones	103
11.1. Ruido gaussiano	104
11.2. Pulso gaussiano-sinusoidal	105
11.3. Modificación STL	107
11.3.1. Estacionalidad	108
11.3.2. Tendencia	108
12. Experimentación	113
12.1. Dataset real	113
12.1.1. Descripción	113

12.1.2. Entrenamiento	113
12.1.3. Validación	116
12.2. Dataset sintético	124
12.2.1. Descripción	124
12.2.2. Entrenamiento	124
12.2.3. Validación	127
13. Conclusiones y trabajo futuro	135
III. Predicción de Series Temporales Discretas	137
14. Introducción	139
A. Documentación	141
A.1. Selección de Modelos	141
A.1.1. Perturbated Validation	141
A.1.2. Clasificador LSTM	143
A.1.3. Clasificadores	146
A.2. Detección de anomalías	150
A.2.1. Alteración de series	150
A.2.2. Detector	153
A.2.3. Cálculo Curva Precision-Recall	156
B. Software	159
B.1. Archivos del proyecto	159
B.2. Lenguajes utilizados	159
B.3. Librerías utilizadas	160
Bibliografía	161

Índice de figuras

2.1.	Neurona McCulloch-Pitts con entrada x y umbral θ	24
2.2.	Perceptrón con entrada x , pesos w y umbral θ	25
2.3.	Ejemplo de perceptrón separando linealmente dos conjuntos de datos. Al separar es posible asignar a cada región una etiqueta, siendo equivalente separar a clasificar.	26
2.4.	Esta f no se puede aprender con un solo perceptrón.	27
2.5.	Cada perceptrón aprende un hiperplano distinto.	27
2.6.	Funciones AND y OR como perceptrones.	28
2.7.	Perceptrón que implementa $AND(h_1\bar{h}_2, \bar{h}_1h_2)$	28
2.8.	Añadimos dos perceptrones $AND(h_1, \bar{h}_2)$ (en azul) y $AND(\bar{h}_1, h_2)$ (en rojo).	29
2.9.	Los dos perceptrones iniciales h_1 (azul) y h_2 (rojo).	29
2.10.	Estructura genérica de una FFNN con L capas, dimensiones $d^{(\ell)}$, entrada x , salida y y funciones de activación σ	30
2.11.	Funciones de activación.	31
2.12.	Ejemplo de Gradiente Descendente en una parábola.	34
2.13.	Diferentes resultados según la tasa de aprendizaje μ	34
2.14.	Efecto del <i>overfitting</i> y <i>underfitting</i>	39
2.15.	Sobreajuste entrenando una red neuronal, tenemos que parar antes de que las curvas diverjan.	40
2.16.	Ejemplo de aplicación de convolución 2D.	43
2.17.	Ejemplo de aplicación de convolución 3D.	43
2.18.	Ejemplo de <i>max-pooling</i> y <i>average-pooling</i>	44
2.19.	Ejemplo de estructura de CNN.	44
2.20.	Ejemplo de estructura de un autocodificador	45
2.21.	Ejemplo de estructura de un autocodificador variacional	46
2.22.	Ejemplo de generador de caras usando un VAE entrenado con caras reales.	46
2.23.	Estructura de una red GAN con dígitos.	47
2.24.	Estructura de una RNN.	47
2.25.	Estructura de una RNN desenrollada.	48
2.26.	Estructura de una neurona en una RNN.	48
2.27.	La información de x_1 y x_2 no llega para h_{t+1}	49
2.28.	Ejemplo de puerta.	49
2.29.	Esquema de célula LSTM.	50
2.30.	Puerta del olvido f_t	50
2.31.	Puerta de entrada i_t e información de la neurona \tilde{C}_t	51
2.32.	Estado celular C_t	51
2.33.	Puerta de salida o_t	52
2.34.	Variante que introduce cauces de C_{t-1} con las puertas.	53
2.35.	Variante que sustituye la puerta de entrada por la negación de la del olvido.	53
2.36.	Estructura de la variante GRU.	54
4.1.	Ejemplo de curva ROC.	59

Índice de figuras

4.2. Ejemplo de curva PR	59
6.1. Funcionamiento de la validación cruzada.	65
6.2. Idea general de funcionamiento del <i>PV</i>	67
6.3. Tres funciones distintas de clasificación con varios modelos, recogiendo <i>PV</i> , <i>accCV</i> y <i>acctest</i> y la función aprendida.	67
7.1. Arquitectura del clasificador LSTM con series de longitud 1460 y 10 clases. . .	72
7.2. Ejemplo de SVM lineal.	73
7.3. Árbol de decisión con reglas para si debería jugar al tenis.	74
7.4. Construcción de Random Forest.	74
7.5. Ejemplo de <i>k</i> -NN. Con <i>k</i> = 3 se toma la clase B, con <i>k</i> = 6 la A.	75
7.6. Funcionamiento de <i>DTW</i> frente $\ \cdot\ _2$	76
7.7. Resultados <i>acc</i> y recta de regresión al calcular <i>PV</i> en <i>FreezerRegularTrain</i> versión TS.	82
7.8. Relaciones entre <i>PV</i> y otras métricas.	83
7.9. Distribución de valores del <i>PV</i>	84
7.10. Resultados del grupo 1.	86
7.11. Resultados del grupo 2 (la escala ha cambiado).	87
7.12. Resultados hiperparámetros LSTM.	90
7.13. Resultados hiperparámetros SVM.	91
7.14. Resultados hiperparámetros Random Forest.	92
7.15. Resultados hiperparámetros <i>k</i> -NN.	93
10.1. Arquitectura del modelo <i>autoencoder</i>	102
11.1. Muestreos de la distribución gaussiana con distintos tamaños y σ	104
11.2. Perturbaciones con ruido gaussiano con distintas longitudes y σ en el dataset <i>Beef</i>	105
11.3. Ejemplos de pulsos gaussianos-sinusoidales con distintos tamaños y σ	106
11.4. Perturbaciones con pulso gaussiano-sinusoidal con distintas longitudes y σ en el dataset <i>ECG5000</i>	107
11.5. Dataset de cosenos.	108
11.6. Modificación de la estacionalidad de cosenos.	109
11.7. Modificación de estacionalidad con distintas longitudes y σ	109
11.8. Modificación de la tendencia de cosenos.	110
11.9. Modificación de tendencia con distintas longitudes y σ	111
12.1. Algunas series de <i>TwoLeadECG</i>	113
12.2. Entrenamiento del modelo LSTM.	114
12.3. Reconstrucciones en <i>train</i>	115
12.4. Reconstrucciones en <i>test</i>	115
12.5. Histogramas de error en <i>train</i> y <i>test</i>	116
12.6. Curvas sensibilidad-precisión en <i>train</i> y <i>test</i>	116
12.7. Alteraciones con ruido gaussiano a $\sigma = 0.75$	117
12.8. Alteraciones con ruido gaussiano a $\sigma = 0.9$	118
12.9. Alteraciones con ruido gaussiano a $\sigma = 1$	118
12.10. Alteraciones con ruido gaussiano a $\sigma = 1.25$	119

12.11	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con ruido gaussiano en <i>train</i>	120
12.12	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con ruido gaussiano en <i>test</i>	120
12.13	Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 0.75$	121
12.14	Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 0.9$	121
12.15	Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 1$	122
12.16	Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 1.25$	122
12.17	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con pulso gaussiano-sinusoidal en <i>train</i>	123
12.18	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con pulso gaussiano-sinusoidal en <i>test</i>	124
12.19	Algunas series del <i>dataset Cosenos</i>	125
12.20	Entrenamiento del modelo LSTM en <i>Cosenos</i>	125
12.21	Reconstrucciones en <i>train</i>	126
12.22	Reconstrucciones en <i>test</i>	126
12.23	Histogramas de error en <i>train</i> y <i>test</i>	127
12.24	Modificación de la estacionalidad a $\sigma = 0.01$	128
12.25	Modificación de la estacionalidad a $\sigma = 0.05$	128
12.26	Modificación de la estacionalidad a $\sigma = 1.8$	129
12.27	Modificación de la estacionalidad a $\sigma = 2$	129
12.28	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la estacionalidad en <i>train</i>	130
12.29	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la estacionalidad en <i>test</i>	131
12.30	Modificación de la tendencia a $\sigma = -1$	131
12.31	Modificación de la tendencia a $\sigma = 0.001$	132
12.32	Modificación de la tendencia a $\sigma = 3$	132
12.33	Modificación de la tendencia a $\sigma = 4$	133
12.34	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la tendencia en <i>train</i>	134
12.35	Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la tendencia en <i>test</i>	134

Índice de tablas

7.1.	Datasets del experimento.	79
7.2.	Datasets del experimento (continuación).	80
7.3.	Resultados de Adiac.csv versión TS.	82
7.4.	Diferentes resultados de cada modelo.	84

1. Introducción

En las dos últimas décadas, gracias al uso de las GPUs [OJ04] se ha hecho posible un uso potente y extendido del **aprendizaje profundo** (*deep learning*), una de las subáreas del **aprendizaje automático** (*machine learning*) más conocidas y usadas actualmente debido a su gran versatilidad de aplicación en diversos tipos de problemas, su fácil diseño y su buen rendimiento. Este campo intenta resolver problemas muy diversos que no tienen una solución fácil de diseñar **manualmente** (reconocimiento en imágenes, conducción autónoma, domótica...), mediante el uso de las **redes neuronales** (*Neural Networks*) [LMP43, Heb49]: algoritmos inspirados en el sistema neuronal del cerebro que, como el propio nombre indica, se estructuran mediante capas formadas por *neuronas* que están *conectadas* entre sí formando así una red compleja donde se propaga la información de una capa a otra.

Dentro de las muy variadas y complejas arquitecturas y tipos de capas que se han ido desarrollando destacamos las **redes neuronales recurrentes** (*recurrent neural networks*, RNN) [Hop82, Jor97], redes neuronales que introducen una novedad: la extracción y guardado de información de una entrada en un instante determinado (x_t) para su uso posterior (x_{t+1}); en otras palabras, la red utiliza no solo la entrada/dato actual sino también tiene en cuenta **relaciones de las entradas anteriores**, creándose de esta manera una especie de dependencia temporal.

Este tipo de redes se ha utilizado mucho en tareas donde existe esta dependencia temporal, donde destacan mayormente las **series temporales** (sucesiones de datos ordenados por tiempo) [WR17] donde existen una gran variedad de problemas como por ejemplo: predicción de valores (mercados, n° de pacientes, texto predictivo), clasificación de series (electrocardiogramas) o detección de anomalías (sensores, tráfico servidores).

La más conocida dentro de este tipo de arquitecturas es la *Long Short Term Memory* (LSTM) [HS97] que intenta solventar un gran problema dentro de las RNN: la pérdida de dependencias temporales **a largo plazo**. Implementa un tipo especial de *neurona* llamada **célula de memoria** para solucionar esto, permitiendo aprender u olvidar información del pasado según convenga [WR17].

1.1. Motivación

En este campo tan extenso, existen muchos problemas que se han tratado poco o que son muy recientes y todavía no se han investigado; de aquí surge nuestra idea de ahondar en estos dominios de problemas de series temporales tan poco explorados. Nos centraremos así en dar soluciones a estos problemas mediante arquitecturas de redes neuronales LSTM para aprovechar la especialización en cuanto a la extracción y uso de información de patrones y dependencias temporales que generalmente existen en las series.

1. Introducción

1.2. Objetivos

Los objetivos fundamentales se centrarán en el estudio, investigación y obtención de posibles resultados en problemas pocos tratados actualmente mediante el uso de técnicas basados en *Deep Learning*, concretamente modelos de redes neuronales con arquitecturas que utilizan capas LSTM.

Trataremos tres problemas diversos, cada uno desarrollados en una parte independiente:

1. Selección de modelos ([Parte I](#)).
2. Detección de anomalías ([Parte II](#)).
3. Predicción de series temporales discretas ([Parte III](#)).

1.3. Estructura

Comenzamos con una introducción en [Capítulo 1](#) donde desarrollamos brevemente el contexto del trabajo, junto con la motivación, los objetivos a alcanzar y la estructura del documento.

Hacemos un repaso de los conceptos previos más relevantes para poder entender este trabajo: sobre el aprendizaje profundo en [Capítulo 2](#), series temporales en [Capítulo 3](#) y las métricas utilizadas para valorar nuestros modelos en [Capítulo 4](#).

Después estudiamos en cada parte el problema concreto que hemos ido desarrollando: en [Parte I](#) abordamos el problema de selección de modelos, investigando una nueva propuesta de métrica de validación de modelos llamada *PV* (*Perturbated Validation*) que estudiamos frente a la validación clásica.

En [Parte II](#) nos centramos en la detección de anomalías, modelando una detector capaz de clasificar series anómalas y también una serie de métodos capaces de alterar las series normales para crear otras anómalas.

La última parte en [Parte III](#) valoramos el comportamiento de las redes neuronales en series temporales discretas, un ámbito poco frecuente de uso con redes neuronales.

Se han incluido además dos apéndices con la documentación más relevante de las clases y funciones implementadas ([Apéndice A](#)) y otro con información sobre el software desarrollado ([Apéndice B](#)).

2. Aprendizaje profundo

El **aprendizaje profundo** o *deep learning* es un subárea del aprendizaje automático que trata de solucionar problemas complejos mediante las llamadas **redes neuronales**. El uso de estas redes se ha hecho extremadamente popular en la última década gracias a su facilidad de diseño, gran diversidad de arquitecturas y buenos rendimientos que son aplicables en diferentes dominios de problemas.

Introduciremos primero el origen de estas redes, explicando el funcionamiento completo de la red más básica y visitando el zoológico de las arquitecturas generales más usadas en los problemas actuales.

2.1. Origen

2.1.1. Neurona de McCulloch-Pitts

El desarrollo de este campo empieza con la **bioinspiración** del sistema nervioso humano: una red densa y compleja formada por unidades simples llamadas neuronas. Para comunicarse entre sí, cada neurona procesa los impulsos nerviosos que recibe de otras mediante las dendritas, y devuelve el pulso hacia otras con el axón.

La **Neurona de McCulloch-Pitts** [MP43] es un modelo basado en este mismo funcionamiento, que sirvió como base para todo el desarrollo en el campo del aprendizaje profundo. Considera un vector de entrada $\mathbf{x} = (x_1, \dots, x_M)^T$ que son los impulsos enviados por otras neuronas y recibidos por las dendritas, teniendo en cuenta que se puede estar recibiendo la señal (1) o no (0). La neurona devuelve un pulso si la suma de señales recibidas supera un cierto umbral de activación θ (actualmente llamado el término de sesgo o *bias*).

Formalizamos la definición en Def. 2.1 y representamos visualmente en Figura 2.1.

Definición 2.1 (Neurona de McCulloch-Pitts). Sea un vector de entrada $\mathbf{x} \in \{0,1\}^M$ con $M \in \mathbb{N}$, y $\theta \in \mathbb{R}$ el umbral de activación, la neurona calcula la función h dada por la siguiente expresión:

$$h(\mathbf{x}) = \sigma \left(\sum_{i=1}^M x_i - \theta \right),$$

donde σ es la función de activación dada por

$$\sigma(t) = \begin{cases} 1, & \text{si } t \geq 0 \\ 0, & \text{si } t < 0 \end{cases}.$$

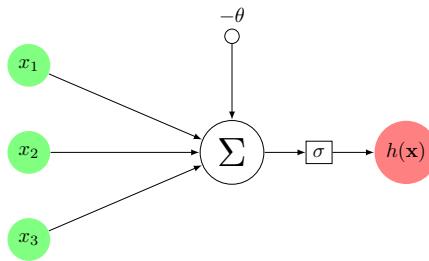


Figura 2.1.: Neurona McCulloch-Pitts con entrada \mathbf{x} y umbral θ .

2.1.2. Perceptrón

Con este modelo es posible representar alguna función muy sencilla pero no es útil para aprender funciones con un poco más de complejidad (observamos que el modelo está determinado por θ , un único parámetro libre). Por ello surge una generalización de este modelo llamado el **perceptrón** [Ros58].

Este modelo incorpora una novedad fundamental: **pesos** que afectan a las entradas para regular las relaciones y las intensidades de estas. Al establecer conexiones más complejas entre las entradas se consigue que la neurona pueda aprender funciones que no sean tan simples como ocurría con la neurona de McCulloch-Pitts. También se admiten ahora las entradas reales, aumentando el dominio de entrada.

Definimos el perceptrón en Def. 2.2 y representamos visualmente en Figura 2.2 viendo que lo que cambia simplemente son los pesos asociadas a las entradas, que actúan como reguladores de las conexiones.

Definición 2.2 (Perceptrón). Sea un vector de entrada $\mathbf{x} \in \mathbb{R}^M$ con $M \in \mathbb{N}$, un vector de pesos $\mathbf{w} \in \mathbb{R}^n$ y $\theta \in \mathbb{R}$ el umbral de activación, el perceptrón calcula la función h dada por la siguiente expresión:

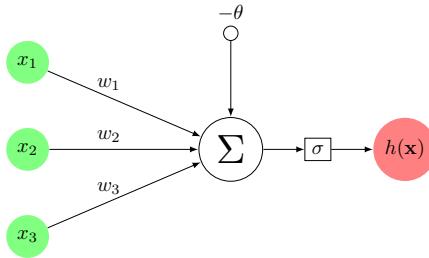
$$h(\mathbf{x}) = \sigma \left(\sum_{i=1}^M w_i x_i - \theta \right),$$

donde σ es la función de activación dada por

$$\sigma(t) = \begin{cases} 1, & \text{si } t \geq 0 \\ 0, & \text{si } t < 0 \end{cases}.$$

Considerando $\mathbf{x} = (x_1, \dots, x_M, 1)^T$ y $\mathbf{w} = (w_1, \dots, w_M, -\theta)^T$ podemos compactar la expresión, teniendo entonces $h(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x})$. Observando detenidamente esta expresión vemos que lo que se tiene realmente es un modelo lineal: un **hiperplano** de \mathbb{R}^M al que luego se le aplica una función no lineal.

Sabemos que un hiperplano divide \mathbb{R}^M en dos regiones conexas, de manera que la función no lineal lo que está realizando realmente es asignar a cada región un valor distinto. Esta idea tan simple de funcionamiento nos permite pensar que el perceptrón sea capaz de solucionar **problemas de clasificación binarios** (Def. 2.3).

Figura 2.2.: Perceptrón con entrada \mathbf{x} , pesos \mathbf{w} y umbral θ .

Estos problemas consisten fundamentalmente en inferir la relación que existe entre los datos de unas variables o **características** y las **etiquetas** asociadas a cada dato, que se identifican con una de las dos clases $+1$ o -1 . Esta relación es la **función de etiquetado** desconocida f que deseamos aprender, donde sabemos cómo se comporta en una **muestra** dada y que usaremos para que los modelos puedan aprender una función aproximada h . El objetivo entonces se convierte en conseguir que $h \approx f$ (ya veremos adelante como medimos esto).

Como nota adicional si consideramos la muestra extraída (X, \mathbf{y}) , podemos notar que $X \in \mathcal{M}_{N \times M}(\mathbb{R})$ e $\mathbf{y} \in \mathcal{M}_{N \times 1}(\mathbb{N})$ y que

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}.$$

Definición 2.3 (Problema de clasificación binario). Sea $M \in \mathbb{N}$ el número de características, y $N \in \mathbb{N}$ el número de observaciones, consideramos el espacio $\mathcal{X} \times Y \subseteq \mathbb{R}^M \times \{-1, +1\}$ del que obtenemos una muestra $(X, \mathbf{y}) \in \mathcal{X}^N \times Y^N$ que sigue una distribución de probabilidad \mathcal{P} desconocida y el espacio de funciones del modelo dado \mathcal{H} . El problema consiste en encontrar un $h \in \mathcal{H}$ de tal manera que $h \approx f$, siendo f la función de etiquetado:

$$\begin{aligned} f : \mathcal{X} &\rightarrow Y \\ \mathbf{x} &\mapsto f(\mathbf{x}) \in \{-1, +1\}. \end{aligned}$$

Efectivamente, sin más que cambiar los valores de la función de activación σ de $\{0, 1\}$ a $\{-1, +1\}$ el perceptrón es capaz de aprender funciones capaces de solucionar estos problemas. De hecho generalmente en el perceptrón se adopta en vez de la función σ anteriormente comentada, la función signo.

Es fácil ver que el problema de clasificación binaria es equivalente a encontrar una función que separe dos conjuntos de datos agrupados por la clase, es decir a la separabilidad de dos conjuntos, que es lo que está haciendo el hiperplano \mathbf{w} del perceptrón ([Figura 2.3 \[Wik\]](#)).

Gracias al Teorema de Convergencia del Perceptrón [[Nov63](#)] tenemos un resultado fuerte sobre la clase de funciones que puede aprender el perceptrón: se prueba que el modelo es

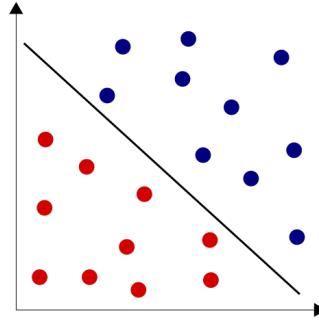


Figura 2.3.: Ejemplo de perceptrón separando linealmente dos conjuntos de datos. Al separar es posible asignar a cada región una etiqueta, siendo equivalente separar a clasificar.

capaz de aprender siempre una función h que separa perfectamente dos conjuntos de datos que sean linealmente separables. Es decir, el perceptrón encuentra una solución muy buena $h \approx f$ ya que coinciden completamente, al menos en la muestra obtenida.

La pregunta restante es cómo conseguir encontrar la función h , cuya respuesta está en el **proceso de aprendizaje**. Este proceso es un método que tiene cada modelo de usar la información de la muestra recogida para poder aprender la función de etiquetado, que en el caso del perceptrón será una simple regla que actualiza los pesos por cada dato de la muestra (2.1).

$$\mathbf{w}_{(t+1)} = \begin{cases} \mathbf{w}_{(t)}, & \text{si } \text{signo}(\mathbf{w}_{(t-1)}^T \mathbf{x}_t) \neq y_t \\ \mathbf{w}_{(t)} + y_t \mathbf{x}_t, & \text{en caso contrario} \end{cases}. \quad (2.1)$$

A pesar de todo, en cuanto los datos dejan de ser linealmente separables el perceptrón no tiene por qué converger siquiera a una solución. Si bien se puede arreglar esto, quedándose con la mejor solución hasta el momento por ejemplo, las soluciones encontradas no suelen ser muy buenas. Por esta razón se necesitaba encontrar una manera de mejorar este modelo.

2.1.3. Perceptrón MultiCapa

Basándonos [AMMIL12] explicaremos el proceso lógico que se lleva a cabo para poder aumentar la potencialidad del modelo del perceptrón.

Consideremos la siguiente función de la figura (Figura 2.4, [AMMIL12]) que es bastante simple aunque está claro que la función f no se puede aprender con un solo perceptrón (tomando una muestra vemos que los datos no son separables).

Ahora bien, usemos dos perceptrones que aprendan cada uno de los hiperplanos separadores (Figura 2.5, [AMMIL12]) y fijémonos en las áreas de las clases. La función f tiene la clase $+1$ cuando h_1 y h_2 tienen $(+1, -1)$ ó $(-1, +1)$; y la clase -1 cuando $(+1, +1)$ ó $(-1, -1)$. Este comportamiento es análogo a la función booleana XOR , por lo que podemos representarla como si lo fuera en base a las salidas de h_1 y h_2 ($f = XOR(h_1, h_2)$) siendo $+1$ Verdadero y -1 Falso. Usando notación booleana podemos reescribir $f = h_1 \bar{h}_2 + \bar{h}_1 h_2$.

Como estamos usando funciones $OR(+)$ y $AND(\cdot)$ para obtener f , el siguiente paso lógico es intentar expresar estas funciones como perceptrones. Es fácil ver que podemos obtenerlas de la siguiente manera (2.2).

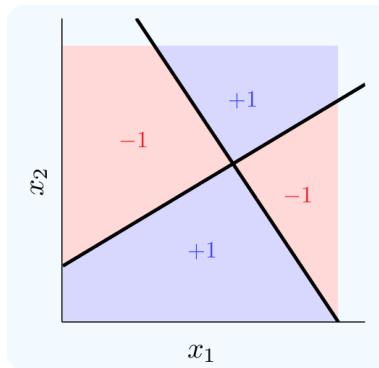
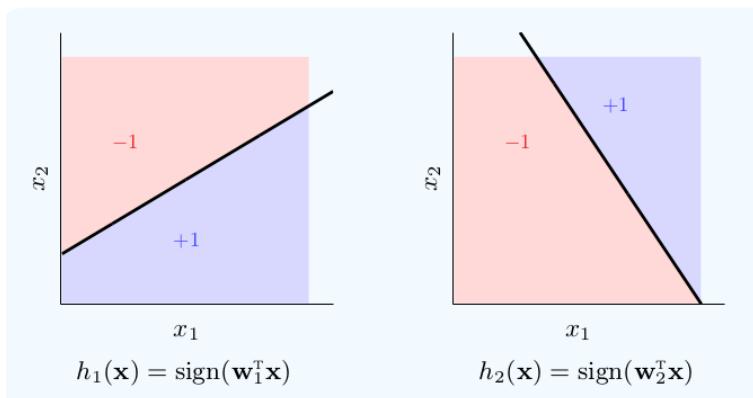
Figura 2.4.: Esta f no se puede aprender con un solo perceptrón.

Figura 2.5.: Cada perceptrón aprende un hiperplano distinto.

2. Aprendizaje profundo

$$\begin{aligned} AND(h_1, h_2) &= \text{signo}(h_1 + h_2 + 1.5), \\ OR(h_1, h_2) &= \text{signo}(h_1 + h_2 - 1.5). \end{aligned} \quad (2.2)$$

Representamos estas funciones en forma de perceptrones, considerando la forma compacta ($\mathbf{x} = (x_1, \dots, x_M, 1)$, $\mathbf{w} = (w_1, \dots, w_M, -\theta)$) que ya habíamos comentado con la Neurona de McCulloch-Pitts (Figura 2.6, [AMMIL12]).

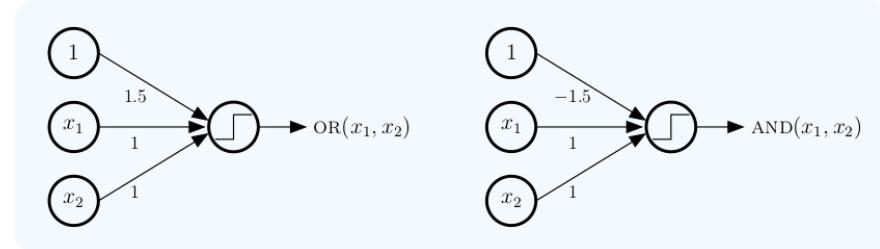


Figura 2.6.: Funciones *AND* y *OR* como perceptrones.

Usando estos perceptrones podemos ir implementando poco a poco la función f : primero nos encontramos con una *OR* que tiene como entradas $h_1 \bar{h}_2$ y $\bar{h}_1 h_2$ (Figura 2.7, [AMMIL12]).

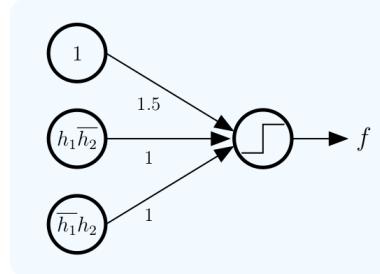
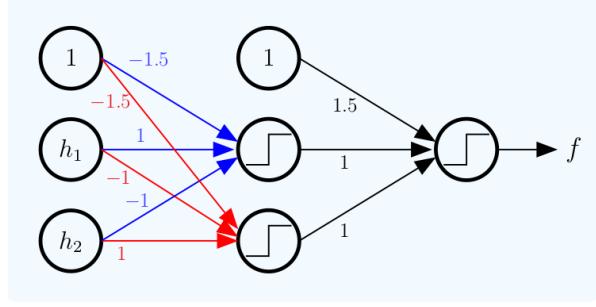
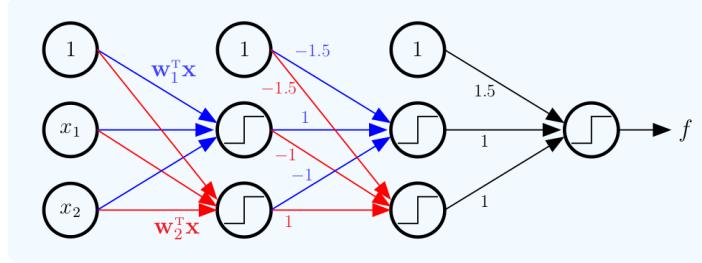


Figura 2.7.: Perceptrón que implementa $AND(h_1 \bar{h}_2, \bar{h}_1 h_2)$.

Ahora tendríamos que usar dos perceptrones *AND* para implementar $h_1 \bar{h}_2$ y $\bar{h}_1 h_2$. Como ambos perceptrones usarán la misma entrada ($\mathbf{h} = (h_1, h_2, 1)$) se pueden representar los dos a la vez mediante dos vectores de pesos distintos (que son lo que determinan al perceptrón). Finalmente hay que tener en cuenta la negación de las entradas que se puede conseguir fácilmente cambiando el signo del peso asociado (Figura 2.8, [AMMIL12]).

Ya casi hemos acabado: falta obtener h_1 y h_2 , pero si recordamos que se tenía que $h_1 = \text{signo}(\mathbf{w}_1^T \mathbf{x})$ y $h_2 = \text{signo}(\mathbf{w}_2^T \mathbf{x})$ que eran los dos perceptrones iniciales. Los incorporamos a nuestro esquema que ya estaría completo (Figura 2.9, [AMMIL12]).

Analizando este modelo vemos que ha sido formado juntando perceptrones, uniendo las salidas de uno con la entrada de otro como si añadiésemos una **capa** tras otra de perceptrones. Debido a esto, este modelo se conoce como el **Perceptrón MultiCapa** (*MultiLayer Perceptron*, MLP) [RHW85] y es de hecho la primera estructura de red neuronal básica y sobre la que se vertebrará toda la estructura del aprendizaje profundo.

Figura 2.8.: Añadimos dos perceptrones $AND(h_1, \bar{h}_2)$ (en azul) y $AND(\bar{h}_1, h_2)$ (en rojo).Figura 2.9.: Los dos perceptrones iniciales h_1 (azul) y h_2 (rojo).

2.2. Redes Neuronales Hacia Adelante

2.2.1. Descripción

Estudiamos las **redes neuronales hacia delante** (*feedforward neural networks*, FFNN) que son las estructuras de redes neuronales más básicas y funcionales, donde sus procesos de funcionamiento y aprendizaje son el núcleo central de las redes neuronales, por lo que al estudiarlos en detalle nos permitirán entender *a grosso modo* cualquier arquitectura utilizada actualmente.

Como hemos visto, este modelo surge con el perceptrón multicapa que se puede considerar un caso particular de esta aunque también se puede denominar como el mismo tipo de red. Usaremos como base los resultados y explicaciones de [AMMIL12] para detallar exhaustivamente comentando su estructura, algoritmos de cómputo y aprendizaje.

2.2.2. Estructura

2.2.2.1. Arquitectura

Veamos primero una estructura general de ejemplo en [Figura 2.10](#). Consideramos una red con L capas, etiquetadas con $\ell = 0, 1, \dots, L$ donde $\ell = 0$ se corresponde con la capa de **entrada**, de las variables, que no se suele contar como una capa propia por lo que cuando se dice que una red tiene L capas estamos diciendo que tiene $L - 1$ capas intermedias ($0 < \ell < L$), llamadas **capas ocultas** y la capa $\ell = L$ de **salida** que es el resultado de la red.

Cada capa tiene una dimensión concreta, que denotaremos por $d^{(\ell)}$ y que indica que la capa ℓ tiene $d^{(\ell)} + 1$ nodos o neuronas (incluimos el nodo 1 del sesgo θ) excepto para la capa de salida que se puede obviar. En particular se tiene que $d^{(0)} = M$, el número de variables y

2. Aprendizaje profundo

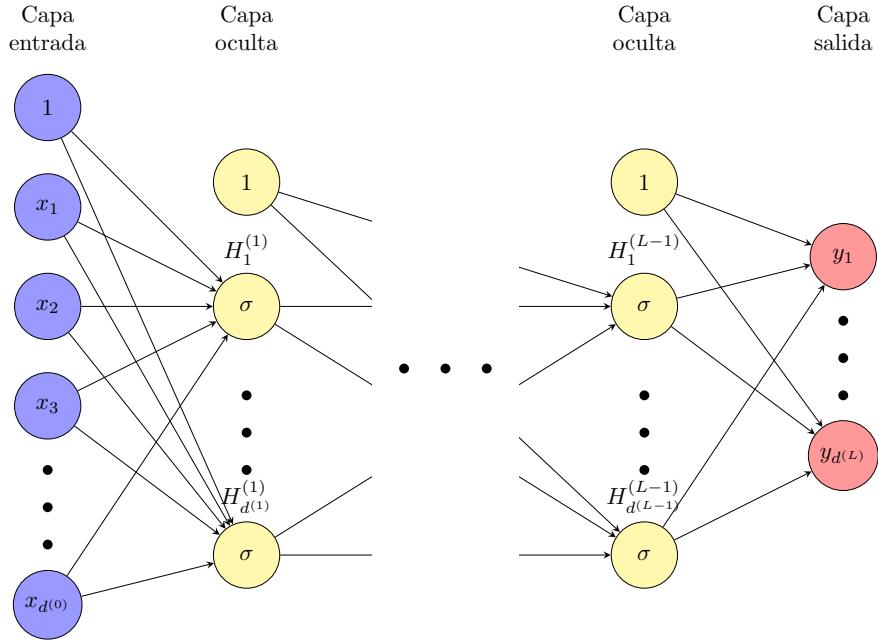


Figura 2.10.: Estructura genérica de una FFNN con L capas, dimensiones $d^{(\ell)}$, entrada x , salida y y funciones de activación σ .

que $d^{(L)}$ será la dimensión de la salida, que ya no tiene que ser obligatoriamente un único nodo; de hecho, la salida ya no tiene que tomar valores discretos ya que podemos considerar otras funciones de activación que veremos un poco más adelante.

Observando este diagrama en profundidad notamos que cada neurona de una capa estará conectada por un peso (las conexiones) a todas las neuronas de la siguiente capa, formándose una red **densa**, nombre por la que también se suele conocer este tipo de capa. Esta estructura de conexiones de entrada a salida, capa por capa, es lo que le da nombre a este tipo de red puesto que los resultados se van pasando *hacia adelante* siempre.

Adicionalmente, tengamos que en cuenta que si cada neurona está asociada con unos pesos, toda la capa ℓ que está formada por $d^{(\ell)}$ (sin contar la de sesgo) estará determinada por todos los pesos de cada neurona, que podemos representar como una matriz (2.3).

$$W^{(\ell)} = \left[\mathbf{w}_1^{(\ell)} \dots \mathbf{w}_{d^{(\ell)}}^{(\ell)} \right] \in \mathcal{M}_{(d^{(\ell-1)}+1) \times d^{(\ell)}}(\mathbb{R}). \quad (2.3)$$

2.2.2.2. Función de activación

Respecto las funciones de activación σ , también llamadas funciones **de transformación** ya que son funciones no lineales, se deja de usar la función signo() en las capas ocultas debido a su discontinuidad en pos de otras funciones aproximadas más *suaves* que realizan una función igual o muy parecida; las más famosas y usadas son las siguientes (funciones dibujadas Figura 2.11, [Mou16]):

1. Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

2. Tangente hiperbólica:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

3. ReLU (REctifier Linear Unit):

$$\text{ReLU}(x) = x^+ = \max(0, x).$$

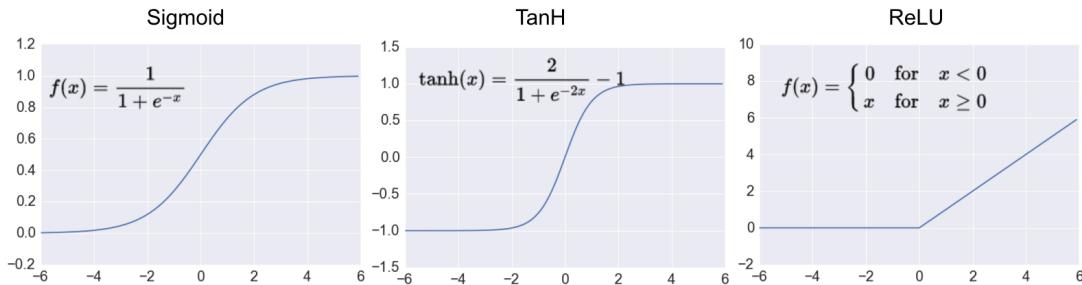


Figura 2.11.: Funciones de activación.

Donde sí podremos usar signo() será en la capa de salida cuando queramos obtener un resultado respecto una clase, sin embargo recordemos que ahora podemos tener una dimensión de salida cualquiera y no estamos limitados a aprender funciones discretas. De hecho, si consideramos otras capas de activación para la salida podemos aprender distintos tipos de funciones según el problema que tengamos entre manos.

Por ejemplo, para los problemas de **regresión** que son simplemente problemas de clasificación donde la f pasa de ser discreta a continua ($f : \mathcal{X} \rightarrow Y, Y \subseteq \mathbb{R}^n$) podemos usar directamente la función identidad ($f(x) = x$); para los problemas de **regresión logística**, que considera una función que en vez de asignar una clase directamente asigna la **probabilidad** ($\mathcal{X} \rightarrow Y, Y \subseteq [0, 1]$), se puede usar la función $\tanh()$ o *sigmoide*.

De hecho, cuando tenemos un problema de clasificación con n clases en vez de calcular una clase directamente se calcula la probabilidad **asociada** a cada clase y cuando se hace una clasificación de una única clase se toma la de mayor probabilidad. Esto es posible utilizando en la capa de salida la función de activación **softmax** Def. 2.4, que toma un vector real de tamaño n y lo normaliza en una distribución de probabilidad.

Definición 2.4 (Softmax). La función softmax es una función $\sigma : \mathbb{R}^n \rightarrow \mathbb{R}^n$ definida de la siguiente forma:

$$\sigma(\mathbf{x}) = \sum_{i=1}^n e^{x_i} \begin{pmatrix} e^{x_1} \\ \vdots \\ e^{x_n} \end{pmatrix}.$$

2. Aprendizaje profundo

En cualquier caso, el modelo \mathcal{H}_{nn} queda determinado por la **arquitectura** de la red neuronal, es decir, especificar el número de capas y la dimensión de cada capa: fijar L y $\mathbf{d} = [d^{(0)}, d^{(1)}, \dots, d^{(L)}]$. Por tanto cada función de hipótesis $h \in \mathcal{H}_{nn}$ estará representada por los pesos de las conexiones entre las capas que representaremos con matrices: $\mathbf{W} = \{W^{(1)}, W^{(2)}, \dots, W^{(L)}\}$.

2.2.3. Propagación hacia adelante

Antes de explicar el algoritmo de cálculo de $h(\mathbf{x})$, indiquemos una breve notación: sea $\ell \in \{1, 2, \dots, L\}$, consideramos la capa ℓ que recibe el resultado $\mathbf{s}^{(\ell)} \in \mathcal{M}_{d^{(\ell)} \times 1}(\mathbb{R})$ de la multiplicación de la capa anterior (2.4) de los datos de entrada $\mathbf{x}^{(\ell-1)} \in \mathcal{M}_{(d^{(\ell-1)}+1) \times 1}(\mathbb{R})$ con los pesos $W^{(\ell)} \in \mathcal{M}_{(d^{(\ell-1)}+1) \times d^{(\ell)}}(\mathbb{R})$

$$\mathbf{s}^{(\ell)} = (W^{(\ell)})^T \mathbf{x}^{(\ell-1)}, \quad (2.4)$$

a los que le aplica la función de activación σ obteniendo la salida $\mathbf{x} \in \mathcal{M}_{d^{(\ell)} \times 1}(\mathbb{R})$ (2.5)

$$\mathbf{x}^{(\ell)} = \begin{bmatrix} 1 \\ \sigma(\mathbf{s}^{(\ell)}) \end{bmatrix}. \quad (2.5)$$

La **propagación hacia adelante** (*forward propagation*) es el algoritmo para poder calcular el resultado de la red $h(\mathbf{x})$. A partir de los datos de entrada $\mathbf{x}^{(0)}$ se calcula la siguiente salida de la capa que se **propaga** a la siguiente capa y así hasta llegar al final siguiendo la siguiente cadena (2.6)

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{W^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\sigma} \mathbf{x}^{(1)} \xrightarrow{W^{(2)}} \dots \xrightarrow{\sigma} \mathbf{x}^{(L)} = h(\mathbf{x}). \quad (2.6)$$

Formalizamos este proceso mediante el algoritmo **Algoritmo 1** [AMMIL12]. Aunque en principio solo necesitamos la última salida $h(\mathbf{x}) = \mathbf{x}^{(L)}$ devolvemos todas las entradas \mathbf{x}' y salidas \mathbf{s} ya que los necesitaremos más adelante.

Algoritmo 1: ForwardPropagation(\mathbf{x})

```

// Inicializamos con la entrada
1  $\mathbf{x}^{(0)} \leftarrow \mathbf{x};$ 
// Propagamos para cada capa
2 para  $\ell = 1, \dots, L$  hacer
    // Calculamos la entrada de la capa
    3    $\mathbf{s}^{(\ell)} \leftarrow (W^{(\ell)})^T \mathbf{x}^{(\ell-1)};$ 
    // Se devuelve la salida de la capa
    4    $\mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \sigma(\mathbf{s}^{(\ell)}) \end{bmatrix};$ 
5 fin
// Devolvemos las entradas y salidas
6  $\mathbf{x}' \leftarrow [\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(L)}];$ 
7  $\mathbf{s} \leftarrow [\mathbf{s}^{(1)}, \dots, \mathbf{s}^{(L)}];$ 
Resultado:  $\mathbf{x}', \mathbf{s}$ 

```

2.2.4. Gradiente Descendente

Lo último que nos falta para poder empezar a usar estos modelos es un mecanismo de aprendizaje, el algoritmo para que la red pueda aprender la función f mediante la muestra de datos que tenemos, es decir buscar una $h \in \mathcal{H}_{nn}$ tal que $h \approx f$, o lo que es lo mismo $h(\mathbf{x}) \approx f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}$.

Está claro que cuanto más se aproxime h a f mejor será, aunque necesitamos una manera de formalizar esta similitud para poder trabajarla. Recordemos que tenemos una muestra que podemos suponer que es suficientemente grande y representativa de la distribución subyacente que se puede usar para poder valorar la diferencia entre h y f . La idea que surge es valorar con una función de **error** o **pérdida** las diferencias entre f y h para todos los valores de la muestra, dando un valor indicativo con el que podemos trabajar.

Este tipo de funciones generalmente son de la forma $E_{\mathcal{D}} : \mathcal{H} \rightarrow \mathbb{R}_0^+$ que toman una función del conjunto de hipótesis del modelo y se valora en base a un criterio impuesto según el problema que se quiera resolver usando la muestra \mathcal{D} ; por ejemplo para problemas de regresión uno querría usar el conocido **error cuadrático medio**. Fijando una función de error, como queremos la función h que menor $E_{\mathcal{D}}(h)$ tenga el problema de aprendizaje se ha convertido en un problema de **minimización**, teniendo en cuenta que para que la función sea de variable real tomaremos la representación de h por los pesos \mathbf{w} (veremos cómo extenderlo a \mathbf{W}).

Es importante entender que se puede imponer una función de error para minimizar en un modelo pero que luego el problema original que se quiere resolver sea distinto, de manera que la **valoración** del modelo se realizará de otra manera; concretamente usaremos las **métrica** para valorar la bondad de los modelos, que veremos con detalle más adelante en [Capítulo 4](#).

Nuestro problema de aprendizaje se ha convertido en un problema de minimización donde para resolverlo podemos recurrir a los métodos numéricos existentes como el **Gradiente Descendente** (*Gradient descent*, GD). [[Cur44](#)].

El gradiente descendente ([Def. 2.5](#)) es un método de minimización cuya idea principal se basa en ir moviéndose hacia el mínimo *bajando* poco a poco tomando la dirección contraria (negativa) del gradiente ([Figura 2.12](#), [[Mol19](#)]). Como la dirección del gradiente indica donde aumenta la función localmente, al tomar la dirección contraria después de suficientes iteraciones, esperamos idealmente llegar al menos a un mínimo local (pudiera acabar en puntos de silla) aunque lo deseable será llegar al global si lo hubiera. Bajo ciertas condiciones podemos asegurar la convergencia de este método, por ejemplo si la función es convexa y el gradiente es lipschiziano [[SSBD14](#)].

Definición 2.5 (Gradiente descendente). Sea $E : \mathbb{R}^m \rightarrow \mathbb{R}$ una función diferenciable, un punto inicial $\mathbf{w}_{(0)}$ y la tasa de aprendizaje $\mu \in \mathbb{R}^+$. El gradiente descendente es un método iterativo de primer orden para encontrar mínimos que sigue la siguiente regla de actualización:

$$\mathbf{w}_{(t)} = \mathbf{w}_{(t-1)} - \mu \nabla E(\mathbf{w}_{(t-1)}), \forall t \in \mathbb{N}.$$

El establecer un valor para la **tasa de aprendizaje** (*learning rate*) no es una tarea para nada trivial, de hecho en [Figura 2.13](#) podemos ver las implicaciones de no escoger una tasa adecuada: si es muy baja puede resultar en una convergencia **muy lenta**, y si es alta puede que **no converga** o lo haga saltándose mínimos **mejores**, también podría pasar que el error *explote* (diverge) [[Rud16](#)].

2. Aprendizaje profundo

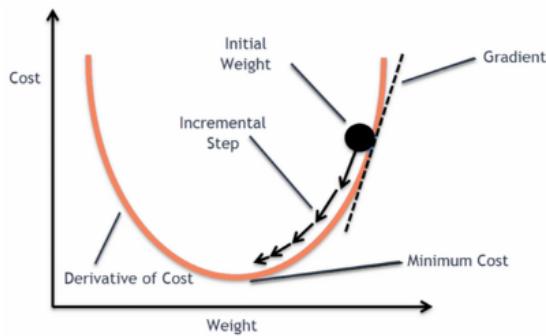


Figura 2.12.: Ejemplo de Gradiente Descendente en una parábola.

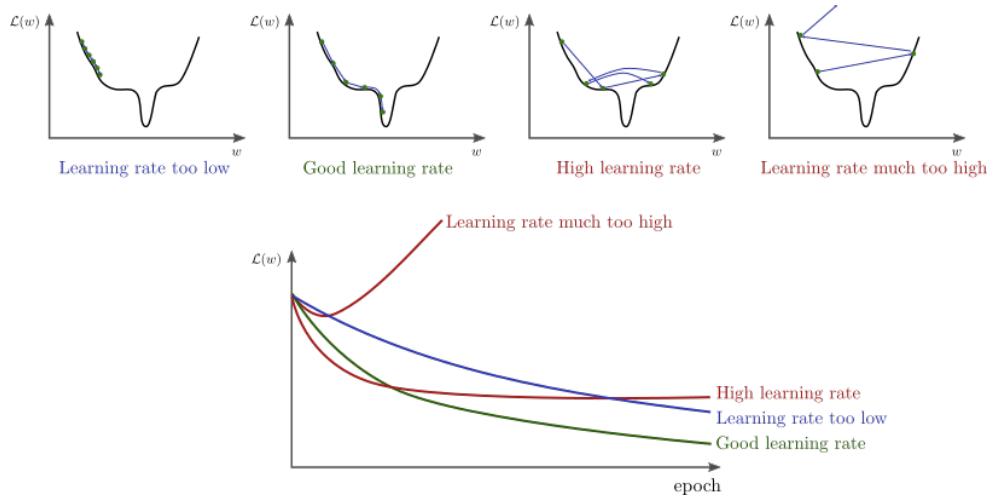


Figura 2.13.: Diferentes resultados según la tasa de aprendizaje μ .

En principio ya tendríamos un método para poder intentar que nuestra red pueda aprender pero tenemos un problema general a la hora de calcular $\nabla E_{\mathcal{D}}$: si nuestra muestra de datos \mathcal{D} es muy grande esto puede provocar que el cálculo sea muy costoso en tiempo e incluso supere los tamaños de memoria del *hardware* y esto solo para un paso, por lo que para realizar las múltiples iteraciones necesarias para la convergencia resulte completamente inviable [Rud16].

La solución a esto la aporta el **Gradiente Descendente Estocástico** (*Stochastic Gradient Descent*, SGD) [RM51] que es igual que el Gradiente Descendente, solo que en vez de tomar todo el conjunto de datos de golpe \mathcal{D} se hace una partición en subconjuntos de tamaño n llamados **lotes** o *batches* y se actualizan los pesos para cada *batch*. La aleatoriedad que se introduce al escoger los *batches* es interesante ya que nos puede ayudar en la búsqueda de mínimos, aportándonos un poco de exploración para encontrar mejores mínimos en la función.

Técnicamente se llama SGD cuando $n = 1$, es decir, cuando actualizamos la red dato a dato, cuando $1 < n < N$ se suele llamar al método **Gradiente Descenso con Mini-lotes** (*Mini-batch Gradient Descent*) aunque también se usa SGD indistintivamente.

Respecto al tamaño del *batch* n , el rango de valores típico de suele estar en [32, 256] aunque dependerá del modelo y problema concretos que se esté considerando, se suele probar con los valores típicos y se van adaptando para obtener un buen funcionamiento.

En principio ya estaríamos listos para que nuestra red aprendiera, pero merece la pena mencionar que en la actualidad se utilizan métodos derivados más complejos que intentan mejorar el funcionamiento del SGD para una convergencia más rápida y encontrar mejores mínimos. Listamos unos cuantos de los más conocidos y populares: **Adadelta**, **RMSprop**, **Adam**, **AdaMax** y **Nadam** [Rud16]. No existe un método absoluto que gane a todos, de hecho SGD sigue siendo una opción viable, por lo que dependiendo del problema se puede decidir escoger uno y otro; aun así uno de los optimizadores más populares y que se suele escoger para las redes neuronales es **Adam**.

2.2.5. Propagación hacia atrás

2.2.5.1. Sensibilidades

Vamos a aplicar SGD para que nuestro modelo aprenda, para ello necesitamos $\nabla E_{\mathcal{D}}(\mathbf{w})$ pero aquí $\mathbf{w} = \{W^{(1)}, \dots, W^{(L)}\}$ por lo que hay que calcular las derivadas parciales respecto **todas las matrices**. Considerando que el error en la muestra no es más que la media de los errores de cada dato podemos reescribir como (2.7)

$$E_{\mathcal{D}}(\mathbf{w}) = \frac{1}{N} \sum_{d \in \mathcal{D}} E_d(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N e_i. \quad (2.7)$$

Y por tanto las derivadas parciales del error respecto cada matriz de \mathbf{w} serán como (2.8)

$$\frac{\partial E_{\mathcal{D}}}{\partial W^{(\ell)}} = \frac{1}{N} \sum_{i=1}^N \frac{\partial e_i}{\partial W^{(\ell)}}. \quad (2.8)$$

Es posible usar métodos numéricos para obtener cada $\frac{\partial e}{\partial W^{(\ell)}}$ pero la complejidad del algoritmo para la derivada respecto cada peso sería de $O(Q^2)$ siendo Q el número de pesos, que se hace computacionalmente inviable. Surge así el algoritmo de la **propagación**

2. Aprendizaje profundo

hacia atrás (*backpropagation*) que nos permite calcular estas derivadas eficientemente con una complejidad de $O(Q)$.

Este algoritmo se basa en la **regla de la cadena** para expresar las derivadas parciales de la capa ℓ en base a las de la capa $(\ell + 1)$. Para describir el funcionamiento, definamos primero el **vector de sensibilidad** (Def. 2.6) que trata de medir como cambia e respecto \mathbf{s}^ℓ .

Definición 2.6 (Vector de sensibilidad). Sea $\ell \in \mathbb{N}$, el vector de sensibilidad de la capa ℓ notado por $\boldsymbol{\delta}^{(\ell)}$, es el gradiente del error e respecto la señal de entrada en la capa $\mathbf{x}^{(\ell)}$, es decir:

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial e}{\partial \mathbf{s}^{(\ell)}}.$$

Usando la regla de la cadena, y recordando que $\mathbf{s}^{(\ell)} = (W^{(\ell)})^T \mathbf{x}^{(\ell-1)}$ expresamos las derivadas parciales con la sensibilidad (2.9) [AMMIL12]

$$\frac{\partial e}{\partial W^{(\ell)}} = \frac{\partial e}{\partial \mathbf{s}^{(\ell)}} \cdot \frac{\partial \mathbf{s}^{(\ell)}}{\partial W^{(\ell)}} = \mathbf{x}^{(\ell-1)} (\boldsymbol{\delta}^{(\ell)})^T. \quad (2.9)$$

Además, considerando que $\mathbf{x}^{(\ell)} = \sigma(\mathbf{s}^{(\ell)})$ y volviendo a aplicar la regla de la cadena podemos obtener la sensibilidad de la capa (ℓ) en función de la de $(\ell + 1)$ para cada coordenada (2.10), que extendemos al vector entero (2.11) [AMMIL12]

$$\delta_i^{(l)} = \frac{\partial e}{\partial s_i^{(l)}} = \sum_{j=1}^{d^{(\ell)}} \left(\frac{\partial e}{\partial s_j^{(\ell+1)}} \cdot \frac{\partial s_j^{(\ell+1)}}{\partial x_i^{(\ell)}} \right) \cdot \frac{\partial x_i^{(\ell)}}{\partial s_i^{(l)}} = \sum_{j=1}^{d^{(\ell)}} (\delta_j^{(\ell+1)} w_{ij}^{(\ell)}) \sigma'(s_i^{(\ell)}), \quad (2.10)$$

$$\boldsymbol{\delta}^{(\ell)} = \sigma'(\mathbf{s}^{(\ell)}) \otimes \left[W^{(\ell+1)} \boldsymbol{\delta}^{(\ell+1)} \right]_1^{d^{(\ell)}}. \quad (2.11)$$

\otimes denota la multiplicación componente a componente y $[\mathbf{v}]_1^{d^{(\ell)}}$ indica que se toman las componentes $1, \dots, d^{(\ell)}$ del vector \mathbf{v} que simplemente indica que no contamos la componente del nodo de sesgo (el nodo 1).

Recapitulando: las salidas $\mathbf{x}^{(\ell)}$ se pueden obtener simplemente calculando el resultado de la red (propagación hacia adelante), lo que resta por calcular son las sensibilidades $\boldsymbol{\delta}^{(\ell)}$. Replicamos el algoritmo de la propagación hacia adelante con modificaciones: en vez de ir obteniendo y propagando las salidas hacia adelante (se usa $\mathbf{x}^{(\ell)}$ para $\mathbf{x}^{(\ell+1)}$), lo haremos con las sensibilidades propagándolas *hacia atrás* (se usa $\boldsymbol{\delta}^{(\ell+1)}$ para $\boldsymbol{\delta}^{(\ell)}$).

Empezaremos calculando $\boldsymbol{\delta}^{(L)}$ directamente con la definición y después iremos propagando hacia atrás las sensibilidades. Por ejemplo, si consideramos el error cuadrático medio con un modelo cuya salida sea un único valor tendríamos $e(\mathbf{x}; h) = (h(\mathbf{x}) - y)^2$ y podemos calcular la sensibilidad fácilmente (2.12)

$$\boldsymbol{\delta}^{(L)} = \frac{\partial e}{\partial \mathbf{s}^{(L)}} = \frac{\partial}{\partial \mathbf{s}^{(L)}} (\mathbf{x}^{(L)} - y)^2 = 2(\mathbf{x}^{(L)} - y) \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} = 2(\mathbf{x}^{(L)} - y) \sigma'(\mathbf{s}^{(L)}). \quad (2.12)$$

Después habrá que calcular la derivada de la función de activación elegida, por ejemplo para $\sigma(s) = \tanh(s)$ tenemos que $\sigma'(\mathbf{s}^{(L)}) = 1 - \sigma(\mathbf{s}^{(L)})^2$; para la lineal (la identidad) $\sigma(\mathbf{s}^{(L)}) = \mathbf{s}^{(L)}$ se tendrá simplemente que $\sigma'(\mathbf{s}^{(L)}) = 1$.

Describimos el método de *backpropagation* por el cual calculamos las sensibilidades $\boldsymbol{\delta}^{(\ell)}$ en Algoritmo 2.

Algoritmo 2: *BackPropagation(x, y)*

```

// Hacemos propagación hacia adelante para obtener  $\mathbf{x}$  y  $\mathbf{s}$ 
1  $[\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(L)}], [\mathbf{s}^{(1)}, \dots, \mathbf{s}^{(L)}] \leftarrow ForwardPropagation(\mathbf{x});$ 
// Inicializamos las sensibilidades
2  $\boldsymbol{\delta} \leftarrow \frac{\partial e}{\partial \mathbf{s}^{(L)}}(\mathbf{x}, y);$ 
// Propagamos hacia atrás
3 para  $\ell = L - 1, \dots, 1$  hacer
    // Calculamos la sensibilidad
    4  $\boldsymbol{\delta}^{(\ell)} \leftarrow \sigma'(\mathbf{s}^{(\ell)}) \otimes [W^{(\ell+1)} \boldsymbol{\delta}^{(\ell+1)}]_1^{d^{(\ell)}}$ 
5 fin
// Devolvemos las sensibilidades
6  $\boldsymbol{\delta} \leftarrow [\boldsymbol{\delta}^{(1)}, \dots, \boldsymbol{\delta}^{(L)}];$ 
Resultado:  $\boldsymbol{\delta}$ 

```

2.2.5.2. Aprendizaje

Por fin podemos definir el algoritmo SGD [Algoritmo 3](#) por el cual podemos hacer que la red aprenda: solo necesitamos pasar como parámetros la muestra (X, y) , la tasa de aprendizaje μ y el tamaño de los mini-batches tam_{batch} . Obviamente se habrá fijado la arquitectura de la red junto a la función de error a minimizar y las funciones de activación.

Esta es la base completa de las redes neuronales, las distintas arquitecturas que se han desarrollado a lo largo de los últimos años no cambian demasiado en los aspectos fundamentales por lo que sabiendo todo lo que hemos visto podemos hacernos una idea de cómo se extiende a los casos más complejos.

2.2.6. Error y sobreajuste

También tenemos que comentar que a la hora de entrenar las redes neuronales hay que tener en cuenta un aspecto fundamental en el aprendizaje automático: el **compromiso** entre el **sesgo** (*bias*) y la **varianza** (*variance*).

Cuando hablamos de sesgo estamos generalmente hablando del error del modelo en el conjunto de nuestra muestra, notado típicamente por E_{in} y en principio es el error que queremos reducir en el proceso de aprendizaje de nuestros modelos. Sin embargo tenemos un problema, si intentamos ajustarnos perfectamente a los datos casi anulando efectivamente el sesgo pudiera ser que en otra muestra distinta de datos observemos que el modelo no ajusta bien estos nuevos datos.

Este efecto es el conocido **sobreajuste** (*overfitting*) que se da cuando el error de entrenamiento E_{in} no coincide (generalmente es mucho más bajo) con E_{out} que es el error del modelo fuera de nuestra muestra. Este efecto ocurre debido al término de la **varianza** del error, que para nuestro modelo probablemente sea muy alta, y que nos indica generalmente el grado de variabilidad del error para una muestra distinta de datos.

Cuanto más queramos bajar el sesgo, más nos ajustaremos a los datos y provocaremos que aumente la varianza y por tanto tengamos sobreajuste; si no bajamos el sesgo a costa de no aumentar la varianza entonces el modelo pudiera estar mal ajustado (*underfitting*). Tenemos

2. Aprendizaje profundo

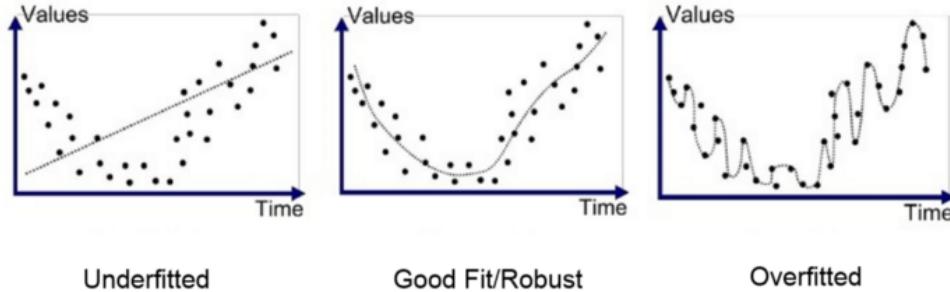
Algoritmo 3: *AprendizajeRed($X, \mathbf{y}, \mu, tam_{batch}$)*

```

// Inicializamos los pesos aleatorios
1  $[W^{(1)}, \dots, W^{(L)}] \leftarrow PesosAleatorios();$ 
// Iteramos hasta que se cumpla el criterio de parada
2 repetir
    // Dividimos aleatoriamente en mini-batches de tamaño  $tam_{batch}$ 
    3  $n_{batches} \leftarrow M // tam_{batch};$ 
    4  $[(X_1, \mathbf{y}_1), \dots, (X_{n_{batches}}, \mathbf{y}_{n_{batches}})] \leftarrow ParticionAleatoria(X, \mathbf{y});$ 
    // Por cada mini-batch actualizamos los pesos
    5 para  $i = 1, \dots, n_{batches}$  hacer
        // Inicializamos los gradientes a 0
        6  $[G^{(1)}, \dots, G^{(L)}] \leftarrow 0 \cdot [W^{(1)}, \dots, W^{(L)}];$ 
        // Calculamos el gradiente de todo el mini-batch
        7 para  $(x, y) \in (X_i, \mathbf{y}_i)$  hacer
            // Forwardpropagation y backpropagation
            8  $[\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(L)}] \leftarrow ForwardPropagation(\mathbf{x});$ 
            9  $[\boldsymbol{\delta}^{(1)}, \dots, \boldsymbol{\delta}^{(L)}] \leftarrow BackPropagation(\mathbf{x}, y);$ 
            // Actualizamos el gradiente de cada capa
            10 para  $\ell = 1, \dots, L$  hacer
                11  $G^{(\ell)}(\mathbf{x}) \leftarrow \mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^T;$ 
                12  $G^{(\ell)} \leftarrow G^{(\ell)} + \frac{1}{tam_{batch}} G^{(\ell)}(\mathbf{x});$ 
                13 fin
            14 fin
            // Actualizamos los pesos de cada capa
            15 para  $\ell = 1, \dots, L$  hacer
                16  $W^{(\ell)} \leftarrow W^{(\ell)} - \mu G^{(\ell)};$ 
                17 fin
            18 fin
    19 hasta que CondicionParada();
    // Devolvemos los pesos
    20  $\mathbf{w} \leftarrow [W^{(1)}, \dots, W^{(L)}];$ 
Resultado:  $\mathbf{w}$ 

```

que encontrar un **compromiso** entre ambos términos, un equilibrio adecuado para poder alcanzar un buen modelo que resuelva nuestro problema; podemos ver un ejemplo de lo que hablamos en una problema de regresión [Figura 2.14](#) [Bha18].



[Figura 2.14.](#): Efecto del *overfitting* y *underfitting*.

El proceso del sobreajuste está relacionado con la **complejidad** del modelo, que nos indica la capacidad del conjunto de hipótesis \mathcal{H} para aprender funciones, siendo una medida común la **Dimensión de Vapnik-Chervonenkis** (dimensión VC) d_{VC} [[VC15](#)]. La idea principal es que cuanto más alta es esta dimensión, el modelo es capaz de aprender funciones más complejas y por tanto es capaz de ajustar bien los datos; sin embargo si la complejidad es mucho mayor que la de la función entonces es cuando se produce el efecto del sobreajuste.

Para evitar esto utilizamos técnicas de **regularización** que se encargan de imponer condiciones sobre el modelo reduciendo así el conjunto de hipótesis \mathcal{H} considerado y disminuyendo su complejidad, que se traduce a un decremento de la varianza a costa de un pequeño aumento del sesgo.

Dentro de este campo hay muchas técnicas que podemos aplicar a las redes neuronales: reducción de la dimensión en profundidad y/o anchura, parada temprana (*early stopping*) que trata de parar el entrenamiento bajo ciertas condiciones, *dropout* [[Her18](#), [HSK⁺12](#)] que desactiva aleatoriamente algunas neuronas de las capas mientras la red se entrena, o añadir términos de penalización al error (*weight decay*) siendo los más conocidos la regularización L1 (Lasso) ([2.13](#)) y L2 ([2.14](#)) [[KH92](#)].

$$E_{reg} = E_{in} + \frac{\lambda}{N} \sum_{\ell=1}^L \|W^{(\ell)}\|_1, \quad \lambda \in \mathbb{R}. \quad (2.13)$$

$$E_{reg} = E_{in} + \frac{\lambda}{N} \sum_{\ell=1}^L \|W^{(\ell)}\|_2^2, \quad \lambda \in \mathbb{R}. \quad (2.14)$$

Para acabar, en las redes neuronales generalmente se deja una muestra de datos que no se usa para entrenar para obtener una estimación del error fuera de la muestra E_{out} que se llama conjunto de **validación** E_{val} o de **test** E_{test} y que podemos usar para comprobar si tenemos sobreajuste o no. Si vamos dibujando E_{in} y E_{val} conforme vamos entrenando la red (cada época) podemos observar cuando hay sobreajuste y parar en ese momento ([Figura 2.15](#) [[Des18](#)]).

2. Aprendizaje profundo

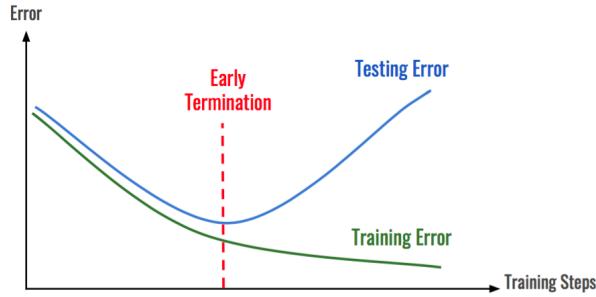


Figura 2.15.: Sobreajuste entrenando una red neuronal, tenemos que parar antes de que las curvas diverjan.

2.2.7. Teorema de aproximación universal

Finalmente veamos un resultado matemático que nos permite afirmar con seguridad la fuerte capacidad de las redes neuronales para aprender funciones: el **teorema de aproximación universal**.

Este teorema tiene dos versiones: el caso de anchura indeterminada, que es la forma clásica, [Cyb89] y el de profundidad indeterminada [LPW⁺17].

El caso de anchura arbitraria (**Teorema 2.1**) nos dice que **cualquier función continua** en un subconjunto compacto de R^k se puede aproximar con tanta precisión como se quiera con una red neuronal *feed-forward* con una capa oculta y función de activación sigmoide aunque de anchura libre.

Teorema 2.1 (Teorema de aproximación universal (anchura indeterminada)). *Sea $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ una función no constante, acotada y continua a la que llamamos función de activación, I_n el hipercubo unidad n -dimensional $[0, 1]^n$. Dado un $\epsilon > 0$ y cualquier función $f \in C(I_n)$ entonces $\exists N \in \mathbb{N}$ entero, $\exists \alpha_i, \theta_i \in \mathbb{R}$ constantes reales y $\exists w_i \in \mathbb{R}^n$ vectores reales para $i = 1, \dots, N$ tal que la función definida como:*

$$F(\mathbf{x}) = \sum_{i=1}^N \alpha_i \sigma(w_i^T \mathbf{x} + \theta_i)$$

es una realización aproximada de la función f , es decir:

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon, \forall \mathbf{x} \in I_n.$$

En otras palabras, las funciones de la forma $F(\mathbf{x})$ son densas en $C(I_n)$. Este resultado también se cumple sustituyendo I_n con cualquier subconjunto compacto de \mathbb{R}^n .

Esta es la versión clásica donde en principio la función σ debe ser sigmoide, pero en [LLPS93] se demuestra que el teorema es cierto para cualquier función φ no polinómica. El teorema también se puede extender a cualquier red con profundidad fija, es decir, con un número de capas ocultas fijo, pero con anchura indeterminada.

Para probar este teorema, primero necesitamos definir cuando una función es **discriminatoria** Def. 2.7 [Cyb89].

Definición 2.7 (Función discriminatoria). Sea $M(I_n)$ el espacio de las medidas con signo regulares finitas en I_n , una función $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ es **discriminatoria** para una medida $\mu \in M(I_n)$

si:

$$\int_{I_n} \sigma(\mathbf{w}^T \mathbf{x} + \theta) d\mu(x) = 0, \forall \mathbf{w} \in \mathbb{R}^n, \forall \theta \in \mathbb{R} \implies \mu = 0.$$

Ahora probamos el Teorema 1 de [Cyb89] que es igual que **Teorema 2.1** pero considerando que σ es una función discriminatoria continua.

Teorema 2.2. *Sea $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ una función discriminatoria no constante. Dado cualquier $\epsilon > 0$, $f \in C(I_n)$ entonces $\exists N \in \mathbb{N}$ entero, $\exists \alpha_i, \theta_i \in \mathbb{R}$ constantes reales y $\exists \mathbf{w}_i \in \mathbb{R}^n$ vectores reales para $i = 1, \dots, N$ tal que la función definida como:*

$$F(\mathbf{x}) = \sum_{i=1}^N \alpha_i \sigma(\mathbf{w}_i^T \mathbf{x} + \theta_i)$$

es una realización aproximada de la función f , es decir:

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon, \forall \mathbf{x} \in I_n.$$

En otras palabras, las funciones de la forma $F(\mathbf{x})$ son densas en $C(I_n)$.

Demostración. Sea $S \subset C(I_n)$ el conjunto de funciones de la forma de $F(\mathbf{x})$, es trivial que S es un subespacio lineal de $C(I_n)$. Queremos demostrar que la clausura de S es $C(I_n)$.

Probemos esto por contradicción: supongamos que $R := \overline{S} \neq C(I_n)$ entonces R es un subespacio cerrado propio de $C(I_n)$. Por consecuencia del Teorema de Hahn-Banach (Proposición 3.5 [Rud73]), existe un funcional lineal acotado L en $C(I_n)$ tal que $L \neq 0$ pero $L(R) = L(S) = 0$.

Por el Teorema de Representación de Riesz [Rudo6], este funcional lineal acotado es de la forma

$$L(h) = \int_{I_n} h(\mathbf{x}) d\mu(\mathbf{x}), \forall h \in C(I_n), \mu \in M(I_n).$$

En particular, como $\sigma(\mathbf{w}^T \mathbf{x} + \theta)$ está en R , $\forall \mathbf{w} \in \mathbb{R}^n, \forall \theta \in \mathbb{R}$ se debe cumplir que

$$\int_{I_n} \sigma(\mathbf{w}^T \mathbf{x} + \theta) d\mu(\mathbf{x}) = 0, \forall \mathbf{w} \in \mathbb{R}^n, \forall \theta \in \mathbb{R}.$$

Sin embargo σ era discriminatoria, así que tendríamos que $\mu = 0$ y por tanto $L = 0$ llegando a contradicción. Por tanto, el espacio S debe ser denso en $C(I_n)$. \square

Finalmente veamos el Lema 1 de [Cyb89] que nos dice que las funciones sigmoides continuas son discriminatorias.

Lema 2.1. *Cualquier función sigmoide acotada y medible σ es discriminatoria. En particular, cualquier función sigmoide es discriminatoria.*

La demostración de **Teorema 2.1** queda así demostrada uniendo **Teorema 2.2** y **Lema 2.1**.

En la otra versión del teorema, de profundidad indeterminada (**Teorema 2.3**), prueba que para toda función Lebesgue-integrable en el espacio de entrada n -dimensional respecto la distancia L^1 puede ser aproximada por una red neuronal de dimensiones de capa fija $n+4$ (anchura fija) y función de activación ReLU pero con número de capas ocultas libre.

2. Aprendizaje profundo

Teorema 2.3 (Teorema de aproximación universal (profundidad indeterminada)). *Para cualquier función Lebesgue-integrable $f : \mathbb{R}^n \rightarrow \mathbb{R}$ y cualquier $\epsilon > 0$ existe una red neuronal hacia adelante con función de activación ReLU y anchura $d_m \leq n + 4$ tal que la función F que representa satisface que*

$$\int_{\mathbb{R}^n} |f(\mathbf{x}) - F(\mathbf{x})| d\mathbf{x} < \epsilon.$$

Finalmente cabe añadir que en ambos casos solo se prueba la existencia, no se da una manera para poder construir los pesos ni del tamaño, por lo que el teorema solo nos indica la potencia del espacio de funciones del modelo de las redes neuronales.

2.3. Clasificación

Veamos cómo podemos clasificar las redes neuronales en función del tipo de problema que resuelven o de la arquitectura principal que tienen integrada.

2.3.1. Aprendizaje

Por ser el aprendizaje profundo una subárea del aprendizaje automático, cualquier modelo de red neuronal se puede clasificar atendiendo al tipo de problema que resuelve, donde existen tres ramas principales:

- Aprendizaje **supervisado**: cuando el problema que queremos resolver tiene asociado a los datos unas **etiquetas**. Se pretende aprender la f desconocida que asigna los datos con las etiquetas mediante la minimización de alguna función de error. Los problemas que ya hemos visto de clasificación y regresión son típicos de aprendizaje supervisado.
- Aprendizaje **no supervisado**: cuando no hay etiquetas, solo nos dan los datos. En este tipo de problemas generalmente se busca patrones y relaciones en los datos, por ejemplo el **agrupamiento** o *clustering* es un problema típico de aprendizaje no supervisado ya que intenta agrupar los datos en distintos tipos.
- Aprendizaje **por refuerzo**: distinto de los dos anteriores, en estos problemas se intenta enseñar a un modelo como realizar una tarea recompensando o penalizando las acciones que realiza. Crear IAs capaces de jugar al ajedrez, Go o incluso videojuegos entraría dentro de esta categoría.

2.3.2. Arquitectura

Veamos aquí distintas arquitecturas con capas más complejas y variadas que han ido surgiendo para intentar abordar problemas de distintos dominios con otros enfoques muy ingeniosos. Obviamos las redes neuronales clásicas que ya hemos explicado y que en cierto sentido podrían considerarse como la red más básica.

2.3.3. Redes Neuronales Convolucionales

Las **Redes Neuronales Convolucionales** (*Convolutional Neural Networks, CNN*) [LB⁺95] es un tipo de arquitectura orientada al **tratamiento de imágenes** por lo que se suele usar bastante en problemas cuyos datos sean imágenes de cualquier tipo, en particular prolifera su uso en problemas supervisados de clasificación de imágenes.

Esta arquitectura se basa fundamentalmente en el uso de las **convoluciones discretas** (2.8) con las imágenes y un **tensor** (que puede ser 2D o 3D) denominado **núcleo** (*kernel*).

Definición 2.8 (Convolución N-D discreta). Sean dos funciones discretas $f, g : \mathbb{Z}^N \rightarrow \mathbb{R}$ definimos la convolución N-D discreta de f con g en el punto $\mathbf{n} = (n_1, \dots, n_N)$ como

$$(f * g)(\mathbf{n}) = (f * g)(n_1, \dots, n_N) = \sum_{m_1=-\infty}^{+\infty} \dots \sum_{m_N=-\infty}^{+\infty} f(m_1, \dots, m_N)g(n_1 - m_1, \dots, n_N - m_N)$$

Esta idea surge de la aplicación de filtros a las imágenes, que se consiguen aplicando estos núcleos convolucionados con la imagen. Los pesos tradicionales que estábamos aprendiendo en las redes se convierten en los valores del núcleo, y lo que se intenta es que la red clasifique imágenes (por ejemplo saber que es un gato o no) **extrayendo características** que se obtienen al aplicar estos filtros.

Un ejemplo de convolución 2D lo tenemos en [Figura 2.16](#) ([Pel20]), donde observamos como la convolución realmente es pasar una matriz en este caso, por la imagen multiplicando coordenada a coordenada y sumando todo el resultado. Esto es extensible fácilmente al caso 3D [Figura 2.17](#) ([Ban18]) que es el más usado debido a que las imágenes vienen en 3 canales (RGB) y por tanto pasan de ser matrices (2D) a tensores 3D.

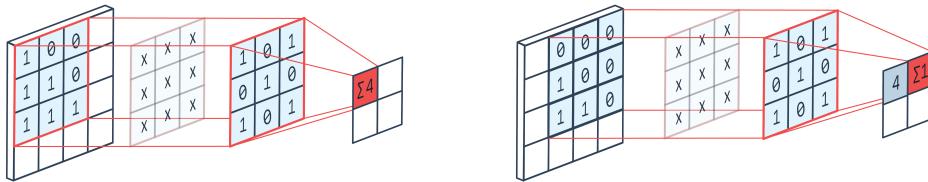


Figura 2.16.: Ejemplo de aplicación de convolución 2D.

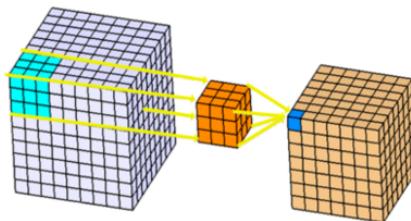


Figura 2.17.: Ejemplo de aplicación de convolución 3D.

Notamos que si las imágenes son muy grandes al aplicar diversas capas de filtros el número de pesos puede elevarse bastante y hacerse computacionalmente inviable, por lo que

2. Aprendizaje profundo

se suele aplicar una técnica de discretización/reducción de dimensión llamada *max-pooling* o *average-pooling* [Gra14]. Esta técnica es igual que aplicar un filtro con 1 solo que en vez de hacer la suma se toma o bien el máximo o la media de los valores, consiguiendo reducir la dimensión de la imagen tomando representantes de cada área a la que se le aplica; un ejemplo de esto está en [Figura 2.18](#) [Y+19].

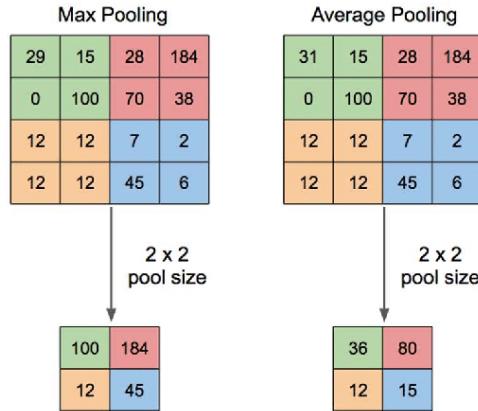


Figura 2.18.: Ejemplo de *max-pooling* y *average-pooling*.

Generalmente la estructura de una CNN tiene dos partes: la primera parte se corresponde al **extractor de características** que obtiene estas características aplicando filtros mediante el uso de convoluciones con *pooling*, y la segunda es el **clasificador** que es una red densa (como si fuese una *feed-forward neural network*) que utiliza las características extraídas anteriormente para clasificar las imágenes.

Un ejemplo de esta estructura para clasificar imágenes de dígitos lo tenemos en [Figura 2.19](#) [Sah18].

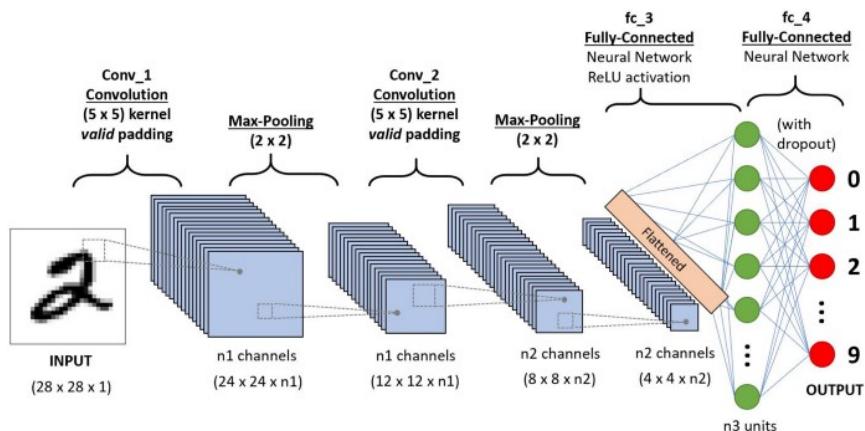


Figura 2.19.: Ejemplo de estructura de CNN.

2.3.4. Autocodificador

Los **autocodificadores** (*autoencoder, AE*) [MRG⁺86] son redes neuronales modeladas típicamente para problemas de aprendizaje no supervisado que intentan replicar la entrada en la salida, intentando que en el proceso la red aprenda las características fundamentales de los datos consiguiendo así una **representación** comprimida de los datos (están codificados).

Veamos la estructura (Figura 2.20, [SC20]) que generalmente se divide en dos partes:

1. **Codificador** (*encoder*): la parte de la red que recibe la entrada y va disminuyendo el tamaño de las salidas hasta llegar al vector donde queda codificado la entrada.
2. **Descodificador** (*decoder*): la parte de la red que recibe el vector codificador y va aumentando su tamaño hasta llegar a la salida donde reconstruye la entrada original.

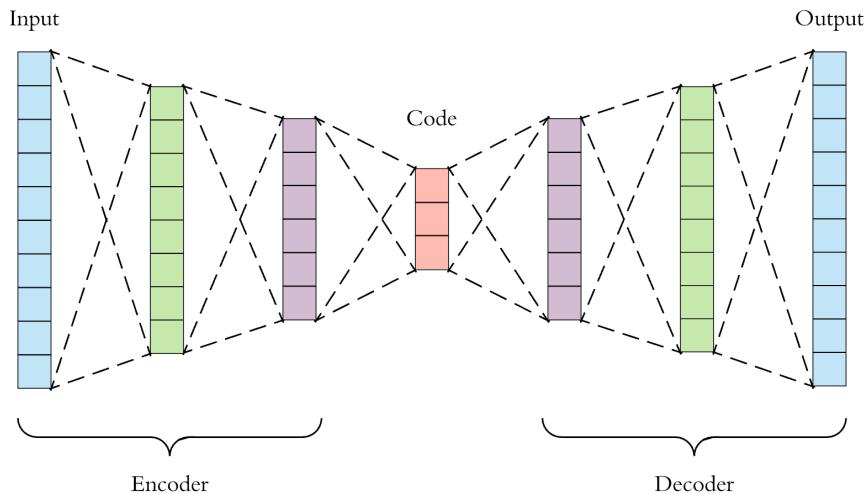


Figura 2.20.: Ejemplo de estructura de un autocodificador

Este tipo de redes son útiles cuando necesitamos aprender algún patrón o estructura de los datos, o cuando necesitamos comprimir en una dimensión reducida datos muy grandes.

También mencionamos los **autocodificadores variacionales** (*variational autoencoders, VAE*) que lo que cambian respecto los autocodificadores es el tipo de codificación que se está haciendo, en vez de un vector cualquiera se busca obtener una representación con **distribuciones de probabilidad gaussiana**, para ello se toman dos vectores uno de medias $\mu = (\mu_1, \dots, \mu_n)^T$ y otro de varianzas $\sigma^2 = (\sigma_1^2, \dots, \sigma_n^2)^T$ de manera que si los juntamos obtenemos un vector de variables aleatorias que siguen distribuciones gaussianas $\mathcal{N}(\mu, \text{diag}(\sigma^2))$ de las que podemos obtener una muestra que reconstruimos en la salida Figura 2.21 [SC20].

Para que el entrenamiento se haga correctamente se debe introducir un término nuevo a la función de pérdida que teníamos para la reconstrucción de la entrada: la **Divergencia de Kullback-Leiber** [KL51] que *a grosso modo* mide la diferencia entre dos distribuciones de probabilidad. Para poder efectuar la diferencia necesitamos saber cual es la distribución latente, pero esta es desconocida así que imponemos la hipótesis de que sea una distribución normal $\mathcal{N}(\mathbf{0}, \mathbf{I})$ y ya podemos obtener la divergencia como (2.15).

2. Aprendizaje profundo

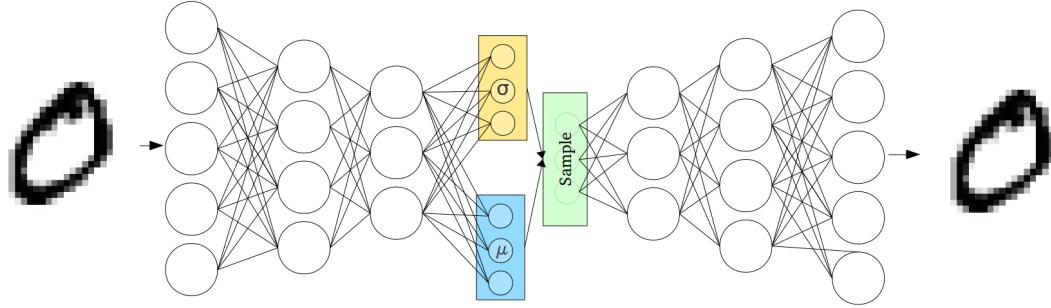


Figura 2.21.: Ejemplo de estructura de un autocodificador variacional

$$D_{KL}(\mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2)) || \mathcal{N}(\mathbf{o}, \mathbf{1})) = \frac{1}{2} \sum_{i=1}^n \left(\sigma_i^2 + \mu_i^2 - 1 - \log(\sigma_i^2) \right). \quad (2.15)$$

Gracias a este diseño, estos modelos son muy útiles para cuando se quieren generar nuevos datos sintéticos que sigan los mismos patrones de los datos donde fueron entrenados, haciendo que parezcan datos auténticos. Por ejemplo podríamos crear un generador de caras en Figura 2.22 [SC20].



Figura 2.22.: Ejemplo de generador de caras usando un VAE entrenado con caras reales.

2.3.5. Redes Generativas Antagónicas

Las **Redes Generativas Antagónicas** (*Generative Adversarial Nets*, GAN) [GPAM⁺14] es uno de los tipos más novedosos de redes neuronales actualmente cuyo objetivo es crear muestras sintéticas nuevas a partir de una muestra real. Esta arquitectura está compuesta por dos redes distintas: la red **generadora** (*generator*) y la red **discriminadora** (*discriminator*). La generadora toma ruido aleatorio como entrada y trata de producir una salida que sea muy parecida a los datos reales, por otro lado la discriminadora toma la entrada del generador y una muestra real del *dataset* y su objetivo es acertar cual es la verdadera (Figura 2.23, [Sil18]).

Podemos entender este tipo de red parecido a un juego de suma cero ya que la función de pérdida que intenta minimizar uno es la contraria al otro, por lo que se tiene una especie

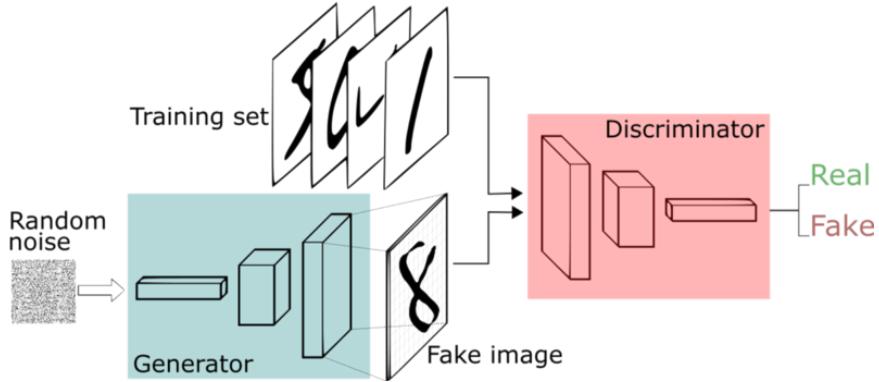


Figura 2.23.: Estructura de una red GAN con dígitos.

de *lucha* en el entrenamiento. El generador empieza mejorando los datos fabricados lo que provoca que el discriminador le cueste más diferenciar y se entrene para mejorar su capacidad de discriminación que hace que el generador tenga que hacer mejores fabricaciones; y así cíclicamente, dándonos al final de entrenamiento un generador que devuelve muestras casi imposibles de diferenciar de una real.

Como ya ocurría con los VAE estos modelos son muy útiles para crear datos nuevos que sean prácticamente reales o también para la creación de videos y fotos falsas, modificando las caras de las personas por otras (las famosas *Deepfake* [NNN⁺20]).

2.3.6. Redes Neuronales Recurrentes

Finalmente vemos el tipo de arquitectura en el que nos centraremos, las **Redes Neuronales Recurrentes** (*Recurrent Neural Networks*, RNN) [Elm90] tratan de aprovechar la dependencia temporal que presentan ciertos tipos de datos (frases, música, series temporales...) mediante un flujo de información de entradas anteriores hacia las posteriores (Figura 2.24 [Ola15]).

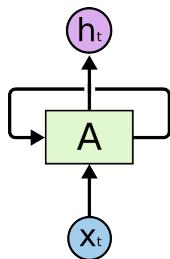


Figura 2.24.: Estructura de una RNN.

Esta estructura nos viene a decir que para una cierta entrada x_t el modelo A nos proporciona una salida h_t pero además se pasa a si mismo, retroalimentándose, una información obtenida en función de x_t que se tendrá en cuenta para entradas posteriores (x_{t+1}, x_{t+2}, \dots). Aunque en principio pueda parecer una estructura complicada se puede *desenrollar* en función del tiempo, quedándonos una estructura muy simple (Figura 2.25, [Ola15]).

Queda claro así cómo según bajo ciertas entradas anteriores, que es lo que proporciona el

2. Aprendizaje profundo

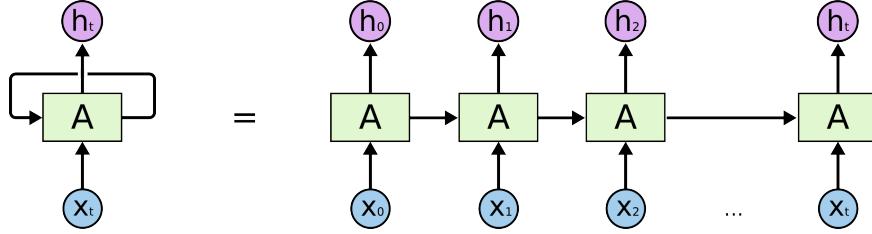


Figura 2.25.: Estructura de una RNN desenrollada.

contexto, la salida de una entrada posterior puede variar enormemente. Así, en las RNN se diseña una neurona especial para enviar la información de manera relativamente sencilla: se une la entrada x_t con la salida de la entrada anterior h_{t-1} , que llamaremos **estado oculto**, a una neurona normal como las conocemos (pesos más una función de activación). La fórmula de cada neurona quedará terminada por los pesos W y la fórmula (2.16) (Figura 2.26, [Ola15]) tomando como función de activación \tanh .

$$h_t = \tanh \left(W^T [h_{t-1} \quad x_t \quad 1]^T \right). \quad (2.16)$$

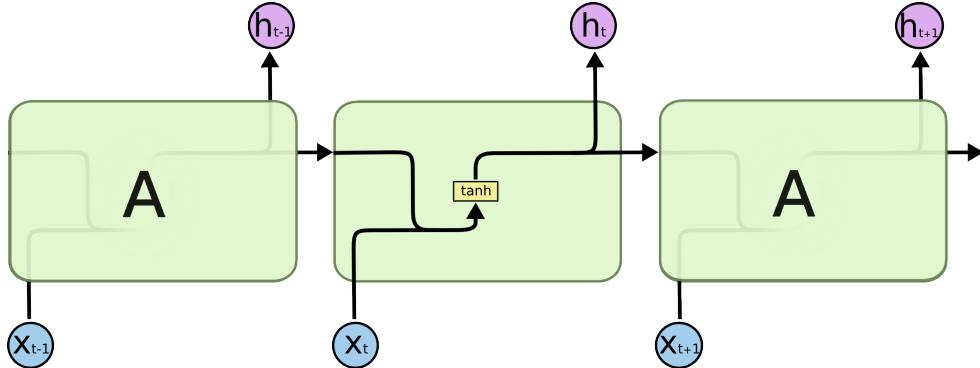


Figura 2.26.: Estructura de una neurona en una RNN.

Sin embargo hay un gran problema: cierta información útil para una entrada **muy posterior** en el tiempo probablemente no llegue debido a que hay un salto temporal muy grande entre las entradas (Figura 2.27, [Ola15]), es decir, el modelo no capta bien las **dependencias temporales a largo plazo** (*long-term dependencies*) [BSF94].

Este obstáculo es consecuencia directa del **problema del desvanecimiento del gradiente** (*vanishing gradient problem*) que ocurre al entrenar redes neuronales muy profundas, ya que, recordando que para obtener el gradiente íbamos propagando el error desde la última capa hasta la primera capa, el gradiente va disminuyendo poco a poco y si la red es muy extensa entonces va disminuyendo a cero provocando que las capas dejen de actualizarse. Lo que viene a decirnos resumidamente es que la información se pierde rápidamente en el tiempo, por lo que no podemos dar información de entradas muy separadas. También es posible que pase el caso contrario, que el gradiente **explote** (*explodes*) (aumente hasta valores que sobrepasan el límite de representación de la máquina); en cualquier caso las RNN suelen ser modelos inestables debido a esto [GBC16].

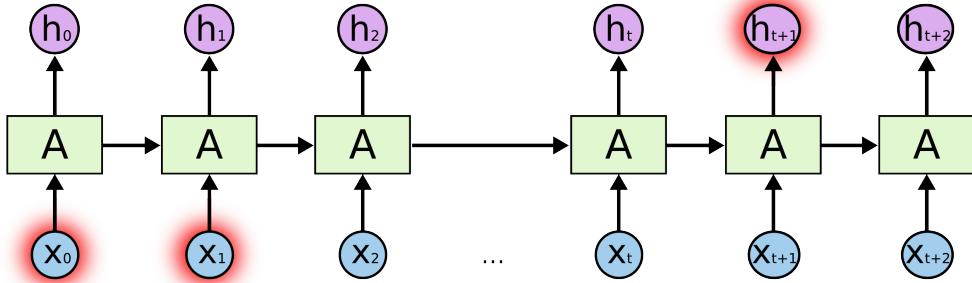


Figura 2.27.: La información de x_1 y x_2 no llega para h_{t+1} .

Para resolver este problema surgen las redes *Long Short Term Memory* (LSTM) [HS97], que consiguen aprender la información tanto a corto como a largo plazo. Estas redes serán las que dedicaremos más atención, y sobre las que nos basaremos para construir nuestros modelos de aprendizaje profundo en el desarrollo del trabajo.

2.4. LSTM

2.4.1. Estructura general

La idea que añade una neurona LSTM para solucionar las dependencias temporales a largo plazo es permitir el paso de la información, recordando u olvidándola, mediante **puertas** (*gates*) y añadiendo otro flujo de información además del estado oculto h_t : el estado celular C_t . Este estado intenta emular una especie de *memoria* que irá llevando información relevante desde el principio de la secuencia de entradas hasta mucho más adelante, evitando la pérdida a largo plazo de las RNN.

Primero veamos como está constituida una puerta (Figura 2.28, [Ola15]): una neurona normal que recibe una entrada y con función de activación **sigmoide**, cuya salida realiza una operación de multiplicación coordenada a coordenada con la información que se está regulando. Como la función sigmoide devuelve valores entre $[0, 1]$, al multiplicarlos por la información nos permite, o bien olvidar cuando los valores sean cercanos a 0 ó recordarlos si son cercanos a 1.

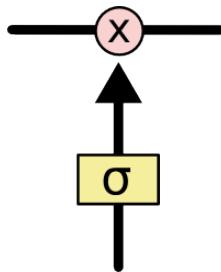


Figura 2.28.: Ejemplo de puerta.

Fijémonos ahora en la estructura general de una neurona o célula LSTM (Figura 2.29,

2. Aprendizaje profundo

[Ola15]) y vayamos paso por paso para entender todas las operaciones que tienen lugar. Apreciamos cómo el estado celular viene dado por el flujo superior de la célula, mientras que el estado oculto entra y sale por el inferior y consideraremos además que $\sigma \equiv \text{sigmoide}$.

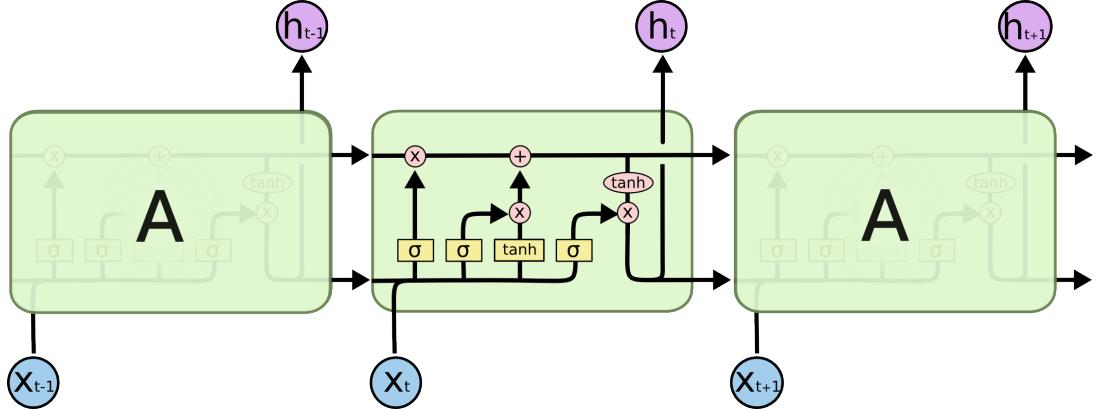


Figura 2.29.: Esquema de célula LSTM.

Primero nos encontramos con la **puerta del olvido** (*forget gate*), la puerta que se encarga de decidir si la información del estado celular anterior c_{t-1} debería olvidarse o no en base a el estado oculto anterior h_{t-1} y la entrada actual x_t ; de esta manera controlamos qué información de las variables anteriores recordamos. El valor de la puerta f_t queda determinado por los pesos W_f y en base a (2.17) (Figura 2.30 [Ola15]).

$$f_t = \sigma \left(W_f^T [h_{t-1} \quad x_t \quad 1]^T \right). \quad (2.17)$$

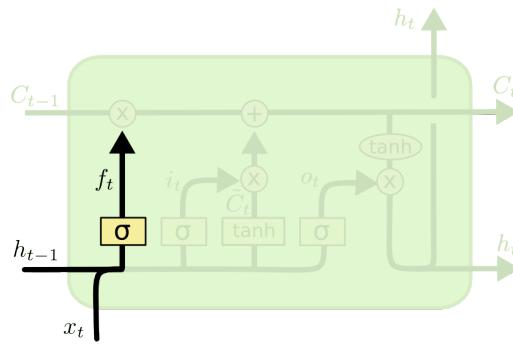


Figura 2.30.: Puerta del olvido f_t .

A continuación tenemos la **puerta de entrada** (*input gate*) que es la que determina si la nueva información en base a la entrada actual x_t y el estado oculto anterior h_{t-1} , se debe incluir o no en el estado celular; indicándonos si esta entrada es relevante o no. Tenemos por un lado el valor de la puerta i_t y el valor de la neurona que tiene la información \tilde{C}_t donde ambas quedan determinadas por sus pesos W_i , $W_{\tilde{C}}$ y con las fórmulas (2.18), (2.19) respectivamente (Figura 2.31 [Ola15]).

$$i_t = \sigma \left(W_i^T [h_{t-1} \ x_t \ 1]^T \right), \quad (2.18)$$

$$\tilde{C}_t = \tanh \left(W_{\tilde{C}}^T [h_{t-1} \ x_t \ 1]^T \right). \quad (2.19)$$

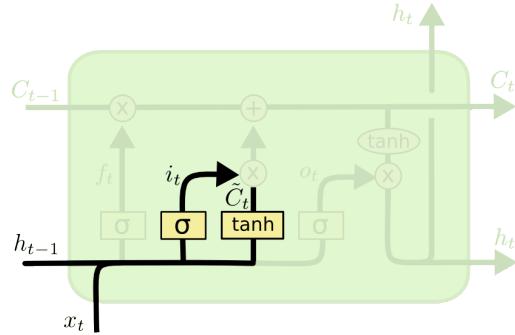


Figura 2.31.: Puerta de entrada i_t e información de la neurona \tilde{C}_t .

Llegamos al **estado celular** que es el valor que lleva la información importante a lo largo de la secuencia y que queda determinada por su entrada anterior C_{t-1} junto a la puerta del olvido f_t y la información actual \tilde{C}_t junto a la puerta de entrada i_t . Su valor C_t queda determinada por los valores anteriores en base a la fórmula (2.20) (Figura 2.32 [Ola15]).

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t. \quad (2.20)$$

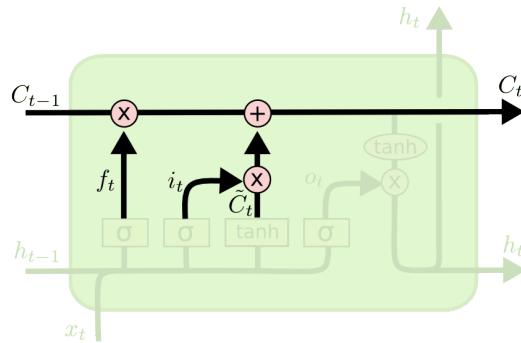


Figura 2.32.: Estado celular C_t .

Finalmente acabamos con la **puerta de salida** (*output gate*) que regula la información del estado celular C_t que devuelve como salida la célula, es decir, el estado oculto h_t . Tenemos el valor de la puerta o_t , determinado por la última matriz de pesos W_o , y el estado celular h_t calculados con las fórmulas (2.21) (2.22) respectivamente (Figura 2.33 [Ola15]).

$$o_t = \sigma \left(W_o^T [h_{t-1} \ x_t \ 1]^T \right), \quad (2.21)$$

$$h_t = o_t \tanh (C_t). \quad (2.22)$$

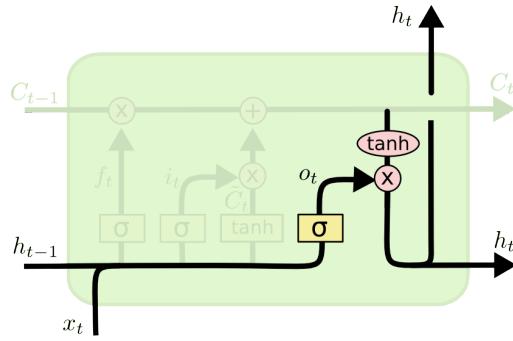


Figura 2.33.: Puerta de salida o_t .

Observamos que cada célula LSTM tiene 4 neuronas normales con sus pesos respectivos que aprender, siendo de estas 3 para puertas (olvido, entrada y salida) y la cuarta para la información de la propia célula. Este tipo de neurona es mucho más compleja y con muchos mas parámetros libres que aprender que la versión que habíamos considerado en las RNN pero permite flujos de información mucho más ricos al permitir olvidar o memorizar a voluntad y por supuesto que se pueda transmitir información importante desde tiempos muy separados.

2.4.2. Variantes

De esta célula se han creado muchas variantes que generalmente suelen ser modificaciones pequeñas que son ciertamente interesantes. Por ejemplo, [GSoo] añade conexiones del estado celular anterior C_{t-1} hacia las puertas de manera que los valores quedan actualizados así (2.23) (Figura 2.34 [Ola15]).

$$\begin{aligned} f_t &= \sigma \left(W_f^T [C_{t-1} \ h_{t-1} \ x_t \ 1]^T \right), \\ i_t &= \sigma \left(W_i^T [C_{t-1} \ h_{t-1} \ x_t \ 1]^T \right), \\ o_t &= \sigma \left(W_o^T [C_{t-1} \ h_{t-1} \ x_t \ 1]^T \right). \end{aligned} \quad (2.23)$$

Otra modificación de [GSoo] considera unificar las puertas de olvido f_t y entrada i_t de manera que si una puerta considera olvidar la información la otra puerta recuerda; de esta manera solo recordamos algo nuevo si olvidamos lo pasado, y solo recordamos lo pasado si olvidamos lo nuevo. Las fórmulas de C_t y i_t actualizadas se quedan así (2.24) (Figura 2.35 [Ola15]).

$$\begin{aligned} i_t &= 1 - f_t, \\ C_t &= f_t C_{t-1} + (1 - f_t) \tilde{C}_t. \end{aligned} \quad (2.24)$$

Finalmente nombramos una de las variantes más conocidas, la **Unidad Recurrente con Puertas** (*Gated Recurrent Unit, GRU*) [CVMG⁺14] que intenta simplificar la célula LSTM para reducir los cálculos y conseguir un entrenamiento mucho más rápido mediante la fusión el

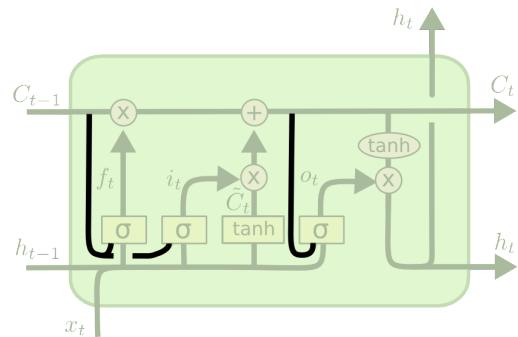


Figura 2.34.: Variante que introduce cauces de C_{t-1} con las puertas.

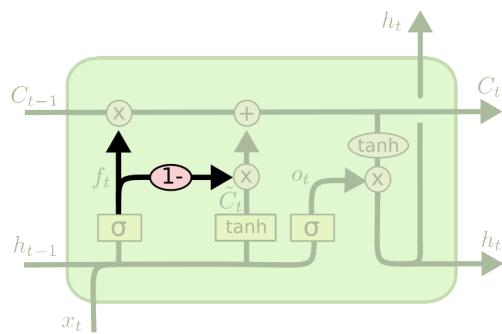


Figura 2.35.: Variante que sustituye la puerta de entrada por la negación de la del olvido.

2. Aprendizaje profundo

estado celular con el oculto, así como también las puertas del olvido y de entrada (formando la puerta de actualización z_t), quitando la puerta de salida y añadiendo una nueva llamada de reinicio r_t . Ahora la célula solo devuelve el estado oculto h_t en función de la siguientes fórmulas (2.25) junto a la nueva estructura (Figura 2.36, [Ola15]).

$$\begin{aligned} z_t &= \sigma \left(W_z^T [h_{t-1} \ x_t]^T \right), \\ r_t &= \sigma \left(W_r^T [h_{t-1} \ x_t]^T \right), \\ \tilde{h}_t &= \tanh \left(W_h^T [r_t h_{t-1} \ x_t]^T \right), \\ h_t &= (1 - z_t) h_{t-1} + z_t \tilde{h}_t. \end{aligned} \quad (2.25)$$

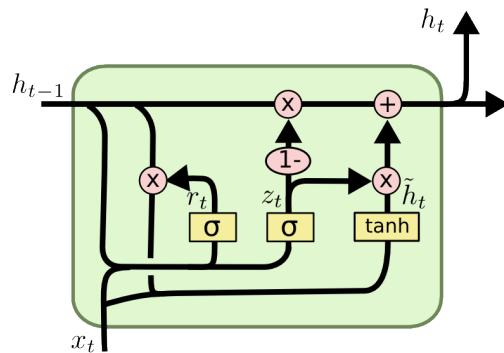


Figura 2.36.: Estructura de la variante GRU.

Sin embargo, en [GSK⁺16] se hace una comparación de rendimiento entre todas variantes y se observó que no había un modelo que fuese especialmente mejor, todas se comportaban más o menos igual de bien, si bien [GSC99] observaron que la versión de LSTM añadiendo un término de error de 1 en la puerta del olvido la hacía la versión más potente de todas las consideradas [GBC16]. Bajo estas razones tomaremos en nuestros modelos las versiones LSTM normales.

2.4.3. Recorte

A pesar de que la arquitectura LSTM evita enormemente el problema del desvanecimiento del gradiente, es posible encontrar en algún caso de entrenamiento que el gradiente explote (la norma del gradiente diverge). Si ocurre esto una técnica muy interesante de utilizar es el método del **recorte de gradiente** (*clipping gradient*) [PMB13], que se encarga de ponerle un *tope* a la norma del gradiente cuando supera cierto valor.

Cada vez que se obtiene el gradiente \mathbf{g} se aplica la siguiente regla (2.26) antes de actualizar los pesos, evitando así que la red use valores enormes del gradiente para actualizar los pesos, haciendo que la red se inestabilice y ya no sirva [GBC16].

$$\mathbf{g}' = \begin{cases} \mathbf{g} & \text{si } \|\mathbf{g}\| \leq v \\ v \frac{\mathbf{g}}{\|\mathbf{g}\|} & \text{en caso contrario.} \end{cases} \quad (2.26)$$

3. Series Temporales

[EN CONSTRUCCIÓN]

3.0.1. Descomposición STL

En las series se pueden observar ciertos componentes comunes con patrones diferenciados en los que se pueden descomponer para hacer un mejor análisis y manipulación de estas [HA18]:

1. **Tendencia:** patrones de crecimiento o disminución a largo plazo.
2. **Estacionalidad:** patrones afectados por factores estacionales (un día concreto del año, una estación como verano) que suelen tener una frecuencia fija.
3. **Ciclo:** fluctuaciones que no parecen tener una frecuencia fija, es decir, que son efímeros.

Estas descomposiciones suelen realizarse en las componentes mencionadas: la **tendencia-cíclica** S_t (se suele decir solamente tendencia), la **estacionalidad** T_t y los **restos** R_t (lo que sobra). Además podemos considerar un modelo aditivo (3.1) donde se suman las componentes o multiplicativo (3.2); donde se multiplican [HA18].

$$y_t = S_t + T_t + R_t \quad (3.1)$$

$$y_t = S_t \cdot T_t \cdot R_t \quad (3.2)$$

La descomposición STL (*Sesonal and Trend decomposition using Loess*) [CCMT90] es uno de los métodos más conocidos para realizar esta decomposición, donde solo necesita que se le pase la frecuencia de la componente estacional para realizar el método.

4. Métricas

Enumeramos y explicamos distintas **métricas**: funciones que nos sirven para medir la bondad del ajuste de los modelos en distintos problemas. Es importante no confundir la métrica usada para medir el rendimiento de los modelos, con la función de error o pérdida que usa cada modelo concreto y trata de minimizar, ya que se puede minimizar distintas funciones de error pero que impliquen un buen resultado en la métrica; aun así es posible que coincida la métrica con la función de error.

Según la tipología del problema podemos encontrar distintas métricas donde veremos las más importantes de los problemas de clasificación y regresión.

4.1. Clasificación

En los problemas de clasificación nos encontramos con aprendizaje **supervisado** ya que generalmente tenemos unas etiquetas asociadas a los datos que queremos predecir correctamente. Según el modelo puede tener dos tipos de enfoque para los que tendremos distintos tipos de métricas: predicción de etiquetas (clasificador "duro") o predicción de probabilidades asociadas a las clases (clasificador "flexible").

Planteamos antes el problema general de clasificación con n clases ([Def. 4.1](#)).

Definición 4.1 (Problema de clasificación multiclase). Sea $M \in \mathbb{N}$ la longitud de las series, $N \in \mathbb{N}$ el número de series y n el número de clases, se considera el espacio $\mathcal{X} \times \mathcal{Y} \subseteq \mathbb{R}^M \times \{0, 1, \dots, n-1\}$ del que se extrae una muestra $(\mathbf{X}, \mathbf{y}) \in \mathcal{X}^N \times \mathcal{Y}^N$ que sigue una distribución \mathcal{P} desconocida y el espacio de funciones del modelo dado \mathcal{H} .

El problema consiste en encontrar un $h \in \mathcal{H}$ de tal manera que $h \approx f$, siendo f la función de etiquetado:

$$\begin{aligned} f : \mathcal{X} &\rightarrow \mathcal{Y} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \in \{0, 1, 2, \dots, n-1\}. \end{aligned}$$

Como nota, en los problemas de clasificación binarios se suele indicar una clase como la positiva (+1) y la otra como la negativa (-1); normalmente la positiva es la que tiene más relevancia para ser detectada que la negativa, aunque no tiene por qué. Además, en clasificación multiclase se puede considerar la clase más importante, si la hubiera, como la positiva y el resto la negativa; o también considerar para cada clase como si fuese la positiva y tomar la media de todos los resultados.

4.1.1. Etiquetas

Si nuestro clasificador devuelve etiquetas, suponiendo que la muestra tiene las clases balanceadas y que importan lo mismo, la métrica más usada es el acierto (*accuracy*) que mide simplemente la proporción de correspondencia de las etiquetas predichas con las originales.

4. Métricas

Sea la función aprendida por un modelo $h \in \mathcal{H}$, definimos acc como 4.1. Considerando que $acc \in [0, 1]$ se puede también usar la función de pérdida asociada como 4.2, aunque es más frecuente usar acc .

$$acc(h) = \frac{1}{n} \sum_{i=1}^N [[h(\mathbf{x}_i) = y_i]], \quad (4.1)$$

$$E(h) = 1 - acc(h). \quad (4.2)$$

Cuando la clase positiva importa más que la negativa tenemos que medir de otra manera para tener en cuenta que una clase influye más que la otra. Para ello introducimos los los conceptos de **precisión** (*precision*) y **sensibilidad** (*recall*):

1. **Precisión:** definido en (4.3), evalúa la proporción de las clases positivas correctamente clasificadas frente a todas las etiquetadas como la clase positiva. Si el modelo es muy preciso nos indica que cuando el modelo etiqueta un dato con la clase positiva casi seguro que lo sea.
2. **Sensibilidad:** definido en (4.4), evalúa la proporción de las clases positivas correctamente clasificadas frente a todas las clases positivas que hay, bien o mal clasificadas. Si el modelo es muy sensible nos indica que clasifica correctamente la mayoría de datos con etiqueta positiva.

$$precision = \frac{verdaderos_positivos}{verdaderos_positivos + falsos_positivos}, \quad (4.3)$$

$$recall = \frac{verdaderos_positivos}{verdaderos_positivos + falsos_negativos}. \quad (4.4)$$

Usando estos dos valores se obtiene la métrica F_β (4.5), donde variamos el β usado según la matriz de costes asociados a los falsos positivos y negativos. Generalmente se usa F_1 cuando cuestan igual, $F_{0.5}$ si los falsos positivos cuestan más y F_2 si son los falsos negativos los que más cuestan.

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}. \quad (4.5)$$

4.1.2. Probabilidades

Aunque el clasificador devuelve probabilidades generalmente necesitamos dar una respuesta con etiquetas, por lo que se suele dar un **umbral** para el que clasificar como una clase (problema binario) o tomar la clase con mayor probabilidad si se están prediciendo probabilidades para todas las clases.

En cualquier caso las dos métricas más usadas generalmente son las que miden el **área bajo la curva** de las funciones **precision-recall** (*PR – AUC*) y **Receiver Operating Characteristic** (*ROC – AUC*). En ambos casos se crea una curva en el cuadrado $[0, 1] \times [0, 1]$ donde se integra para obtener el área que deja la curva, haciendo que ambas métricas tomen valores en $[0, 1]$ siendo mejor el modelo cuanto más alto valga.

Cuando la clase positiva importa igual que la clase negativa se toma *ROC – AUC* que mide el comportamiento del modelo obteniendo las tasas de falsos positivos (*falsos_positivos*)

y verdaderos positivos (*verdaderos_positivos*) para cada umbral de probabilidad que se va poniendo al modelo. Se suele imprimir el gráfico con la curva que forma comparando con el clasificador aleatorio para compararlo (Figura 4.1 [sl2oe]).

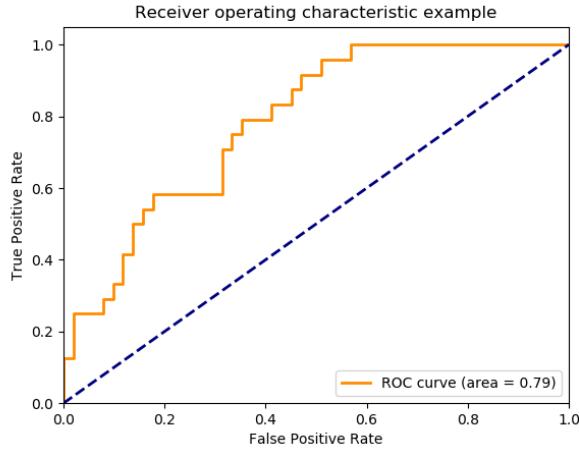


Figura 4.1.: Ejemplo de curva ROC.

Por otro lado, si la clase positiva tiene mayor importancia entonces se coge $PR - AUC$ que en este caso mide la precisión y la sensibilidad del modelo frente a distintos umbrales de probabilidad. También se incluye un modelo aleatorio para hacer la comparación, donde encontramos un ejemplo en Figura 4.2 [sl2oc].

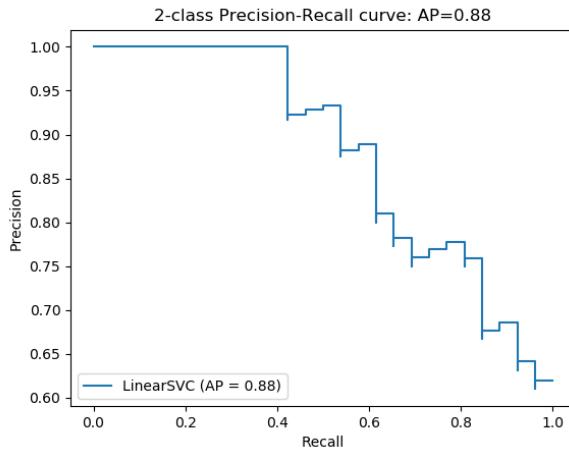


Figura 4.2.: Ejemplo de curva PR.

4.2. Regresión

Un problema de regresión no es más que un problema de clasificación donde las etiquetas no son discretas sino continuas ([Def. 4.2](#)).

Definición 4.2 (Problema de regresión). Sea $M \in \mathbb{N}$ la longitud de las series y $N \in \mathbb{N}$ el número de series y n la dimensión de las etiquetas, se considera el espacio $\mathcal{X} \times \mathcal{Y} \subseteq \mathbb{R}^M \times \mathbb{R}^n$ del que se extrae una muestra $(X, Y) \in \mathcal{X}^N \times \mathcal{Y}^N$ que sigue una distribución \mathcal{P} desconocida y el espacio de funciones del modelo dado \mathcal{H} .

El problema consiste en encontrar un $h \in \mathcal{H}$ de tal manera que $h \approx f$, siendo f la función de etiquetado:

$$\begin{aligned} f : \mathcal{X} &\rightarrow \mathcal{Y} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \in \mathbb{R}^n. \end{aligned}$$

Generalmente se usan las siguientes métricas principales, donde todas toman valores en $[0, +\infty)$ siendo cuanto más bajo mejor:

1. **Error Medio Absoluto** (*Mean Absolute Error*, MAE): definido en [\(4.6\)](#), se suele usar para no tener en cuenta el efecto de los valores atípicos ya que es menos sensible a estos.
2. **Error Cuadrático Medio** (*Mean Squared Error*, MSE): definido en [\(4.7\)](#), es una de las métricas mayormente usadas y añade importancia a errores grandes, ya que se aplica un cuadrado.
3. **Raíz del Error Cuadrático Medio** (*Root Mean Squared Error*, RMSE): definido en [\(4.8\)](#), hace que las medidas del MSE vuelvan a la unidad, dando una mejor interpretabilidad.

$$MAE(h) = \frac{1}{n} \sum_{i=1}^N \|\mathbf{y}_i - h(\mathbf{x}_i)\|_1, \quad (4.6)$$

$$MSE(h) = \frac{1}{n} \sum_{i=1}^N \|\mathbf{y}_i - h(\mathbf{x}_i)\|_2^2, \quad (4.7)$$

$$RMSE(h) = \sqrt{MSE(h)}. \quad (4.8)$$

Parte I.

Selección de modelos

Estudiamos una nueva heurística alternativa a la selección clásica de modelos: *Perturbation Validation (PV)*. Este método intenta medir si la función aprendida por un modelo se ajusta correctamente a la función que se está intentando aprender, intentando ser un **método complementario** a la selección en base a la validación clásica de usar particiones entrenamiento/test junto a la validación cruzada. Este método nuevo es bastante útil para la **búsqueda de hiperparámetros** del modelo, donde la validación clásica a veces no es capaz de discernir entre varios modelos con un rendimiento parecido, pero que alguno puede estar sobreajustando mucho más los datos.

En [Capítulo 5](#) se introduce el contexto bajo el que surge el *PV* y que se va a intentar obtener, definiéndolo formalmente y describiendo su implementación en [Capítulo 6](#). Finalmente, realizamos los experimentos para comprobar su utilidad, junto con los resultados y las conclusiones en [Capítulo 7](#).

5. Introducción

La **selección de modelos** es una tarea muy importante en la resolución de problemas del aprendizaje automático (y profundo) ya que de entre varios modelos propuestos como solución querremos escoger el mejor.

Para valorar esto recurrimos a las **métricas**, aplicaciones que miden cuantitativamente, usando algún criterio, la semejanza entre la función f que estamos intentando aprender y la función h aprendida por el modelo.

Usando una métrica podemos comparar los resultados entre distintos modelos y escoger el que mayor valor arroje, sin embargo aquí entra el problema de la **validación**. Si solo nos basamos con el resultado de la métrica en el mismo conjunto donde hemos entrenado los datos, puede que estemos sobreestimando bastante el valor real debido al efecto del **sobreajuste**.

Recordemos que este efecto se daba cuando el modelo se ajustaba demasiado a los datos de la muestra actual obteniendo un sesgo muy bajo, sin embargo, vimos que también la varianza aumentaba mucho, provocando que para otra muestra distinta de la distribución el modelo obtuviese un rendimiento mucho peor al de la muestra donde fue entrenada.

Por esta razón es necesaria una manera más realista de medir los modelos para evitar el sobreajuste, surgiendo así la **división clásica** entrenamiento/test. Este método consiste en, o bien obtener dos muestras independientes de la distribución o dividir una única muestra en dos antes de manipular los datos. La muestra de entrenamiento se usa para entrenar el modelo y la muestra de test para valorar los modelos, ya que es una nueva muestra que no ha visto el modelo anteriormente.

Aunque este método ya resuelva nuestro problema también se puede querer hacer la selección de los modelos antes de realizar la valoración en el conjunto de test para, por ejemplo, quedarnos con un subconjunto menor de modelos de un gran número de modelos considerados para reducir tiempo de cómputo innecesario. También entra aquí la **selección de hiperparámetros** que es una selección de modelos solo que considerando el mismo modelo base y lo que cambia entre modelos es algún hiperparámetro (por ejemplo el número de neuronas en una red neuronal).

Para resolver esto hay generalmente dos alternativas: hacer una partición más del conjunto de entrenamiento, quedándonos en total con el de entrenamiento, test y el de **validación**; o bien utilizar el método de **validación cruzada**.

La validación cruzada consiste *a grosso modo* en dividir la muestra de entrenamiento en k conjuntos del mismo tamaño y repetir k veces la validación usando un conjunto como *test* y el resto de entrenamiento. Este método intenta dar un valor más robusto que usando un único conjunto de validación, y además puede indicar una idea de cómo de variable será en función a la varianza de los resultados.

Estos son los métodos clásicos que se han usado generalmente para obtener valores de la métrica **realistas** y que nos permiten comparar modelos y por tanto, elegir los mejores en base a esto. Sin embargo, esta clase de validaciones confía que en las muestras de datos recogidas de la distribución es **suficientemente representativa** de la distribución subyacente, y que han sido recogidos **sin sesgo** alguno [ZHG⁺19].

5. Introducción

Estos problemas se puede presentar perfectamente cuando el problema es muy grande y es muy difícil obtener muchas muestras, o cuando se ha cometido algún error al tomar las muestras que nadie se ha percatado. Por ejemplo, en la detección de caras en imágenes, debido a la gran cantidad de posibilidades de formas en las que podrían estar, se necesita una muestra muy grande que abarque múltiples y diversas maneras para poder tener una representación buena de la distribución y además que no haya sesgos (solo salen caras en fotos con buena iluminación, o de gente con tez blanca...).

Por estas razones se presenta un nuevo método para valorar modelos basado en una heurística: *PV (Perturbated Validation)* [ZH^{G+}19]. Este método **no** utiliza las mismas técnicas clásicas de utilizar un conjunto aparte sino que considera todo el conjunto entero, sin divisiones. En líneas generales trata de valorar si el modelo ha entendido realmente **la relación subyacente de los datos** midiendo como varía el comportamiento del modelo frente a **perturbaciones graduales** en los datos.

Nuestro objetivo será ver el funcionamiento de esta heurística, su comportamiento en casos reales, mediante una experimentación utilizando muchos modelos y conjuntos de datos para observar y analizar si nos resulta de utilidad y puede aportar algo nuevo en el paradigma actual de selección de modelos.

6. Selección de modelos

Planteamos el problema de la selección de modelos para problemas de clasificación multiclase en dominios de series temporales: sea $M \in \mathbb{N}$ la longitud de cada serie temporal, N el número de series temporales, n el número de clases y k el número de modelos a ajustar, consideraremos el espacio $\mathcal{X} \times \mathcal{Y} \subseteq \mathbb{R}^m \times \{0, 1, \dots, n-1\}$ del que se obtiene una muestra $(X, y) \in \mathcal{X}^N \times \mathcal{Y}^N$ (datos X y etiquetas y) que sigue una distribución de probabilidad \mathcal{P} desconocida. Sean también el espacio de hipótesis de los modelos considerados $\mathcal{H} = \{\mathcal{H}_1, \dots, \mathcal{H}_k\}$ de los que se ha obtenido una función cada uno $h_i \in \mathcal{H}_i, i = 1, \dots, k$ que intentan aproximar a la función de etiquetado f definida por:

$$\begin{aligned} f : \mathcal{X} &\rightarrow \mathcal{Y} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \in \{0, 1, \dots, n-1\}. \end{aligned}$$

Se pide valorar bajo un criterio determinado la aproximación de las funciones de hipótesis a la función de etiquetado ($f \approx h_i$) de manera que se establezca un orden entre los modelos, permitiéndonos seleccionar los modelos con mejor valoración bajo este criterio.

6.1. Selección clásica

Consideramos la selección de modelos clásica: primero tomamos la muestra o *dataset* (X, y) de la realizamos una partición en un conjunto de entrenamiento (*train*) y test usando la proporción usual de 80/20 % respectivamente, manteniendo la proporción de clases. Después realizaremos el método de Validación Cruzada con k pliegues (*Cross Validation with k folds*, k -CV) [Sto74] en el conjunto de entrenamiento: como explicamos previamente, consiste en dividir el conjunto de entrenamiento en k subconjuntos de tamaño igual que mantienen la proporción de clases y repetir k veces la validación usando un subconjunto como test y el resto como entrenamiento (Figura 6.1, [NLWH18]).

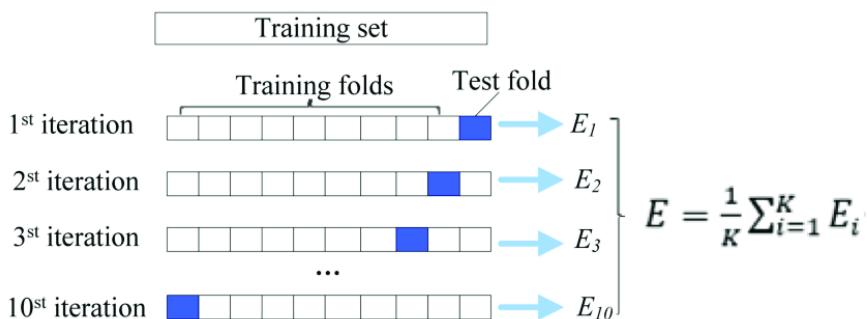


Figura 6.1.: Funcionamiento de la validación cruzada.

6. Selección de modelos

La validación en este caso consiste en entrenar el modelo en el subconjunto de entrenamiento y obtener el valor de la métrica en el subconjunto test; una vez se obtienen los k valores de la métrica se toma el valor final como la media de los valores aunque también se suele devolver la desviación típica o la varianza para tener una idea de cómo puede variar este valor (en qué intervalo de confianza se mueve la métrica).

Para problemas de clasificación consideraremos la métrica *accuracy* acc que ya vimos en [Capítulo 4](#) que básicamente mide el porcentaje de etiquetas predichas correctamente. Utilizando esta métrica junto a la validación cruzada para cada modelo se obtendrá acc_{CV} , y en base a este valor podremos hacer la selección de modelos tomando en cuenta que es un estimador de la métrica en el conjunto test acc_{test} ; por tanto lo esperable es que el mejor acc_{CV} será probablemente el mejor acc_{test} .

6.2. Perturbated Validation

6.2.1. Funcionamiento

La idea general del *PV* es ir midiendo la métrica acc del modelo conforme se van añadiendo perturbaciones graduales en función de una ratio de error r en la muestra de datos, tomando el valor del *PV* como la **pendiente** absoluta de la recta de regresión con los puntos formados por las tasas de acierto y error (acc, r).

Si un modelo es bueno y capta bien la relación de los datos con las etiquetas entonces el acc debería ir decrementando conforme aumenta el ratio de error, ya que no se deja llevar por los datos perturbados. Si por el contrario el modelo es muy complejo y se ajusta demasiado a los datos, provocando **sobreajuste**, veremos que la métrica no cambia casi nada aunque se perturben muchos datos. Finalmente si el modelo no tiene buenos resultados en ningún caso, la variación también será pequeña. ([Figura 6.2 \[ZHG⁺19\]](#)).

Así, el *PV* intenta medir la correspondencia entre la función de etiquetado real con la aprendida a partir del modelo y del que se esperan dos consecuencias:

- Para un valor muy alto la función aprendida h debe ser muy parecida a f y por tanto debería tener valores muy altos de acc en cualquier conjunto.
- Para un valor muy bajo la función no se parece, donde pueden pasar dos cosas:
 - Si acc_{train} es bajo, entonces es que el modelo tiene un ajuste directamente malo y en acc_{test} también lo será.
 - Si acc_{train} es alto, entonces el modelo probablemente ha sobreajustado los datos y se tenga que acc_{test} sea mucho menor y haya una gran diferencia con acc_{train} .

Sin embargo, puede haber casos en los que tengamos un *PV* bajo y obtengamos acc_{test} bastantes altos, esto es consecuencia de cómo sean las muestras obtenidas de la distribución desconocida, si tienen alguno de los problemas que hemos mencionado (sesgo o representatividad insuficiente) entonces no se verá reflejado el problema en el conjunto de test y los modelos funcionarán igual de bien.

Veamos esto con un ejemplo de [\[ZHG⁺19\]](#) [Figura 6.3](#) donde se han creado tres *datasets* sintéticos con unas funciones de etiquetado bien definidas, en forma de lunas, círculos y lineal, y se han entrenando con distintos modelos obteniendo tres valores de las métricas utilizadas, *PV* (*PV*), *CV* (acc_{CV}) y *TA* (acc_{test}) junto con la funciones aprendidas de cada modelo (se corresponde el valor de la etiqueta con un color).

6.2. Perturbated Validation

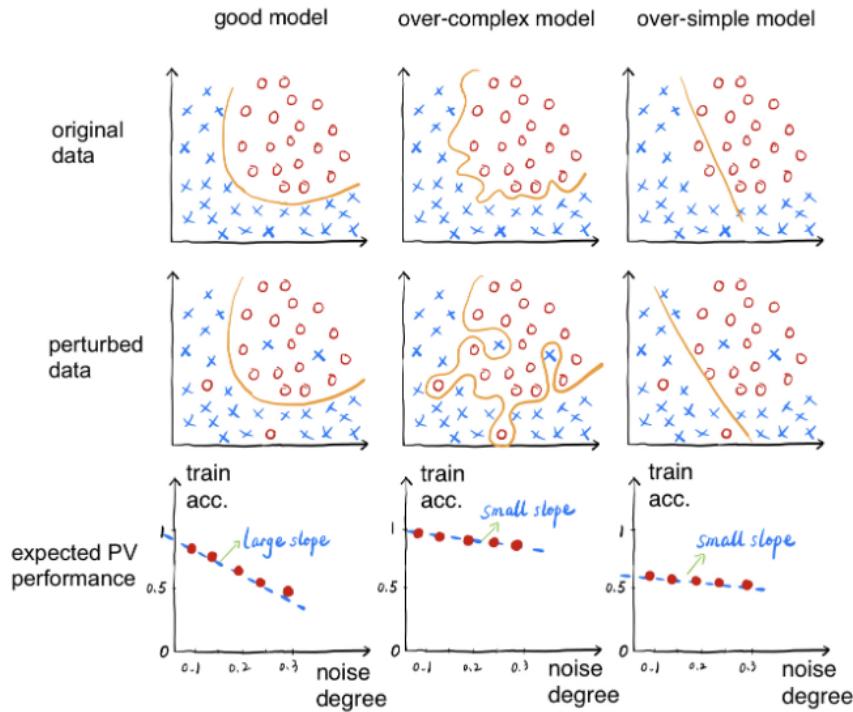


Figura 6.2.: Idea general de funcionamiento del PV.

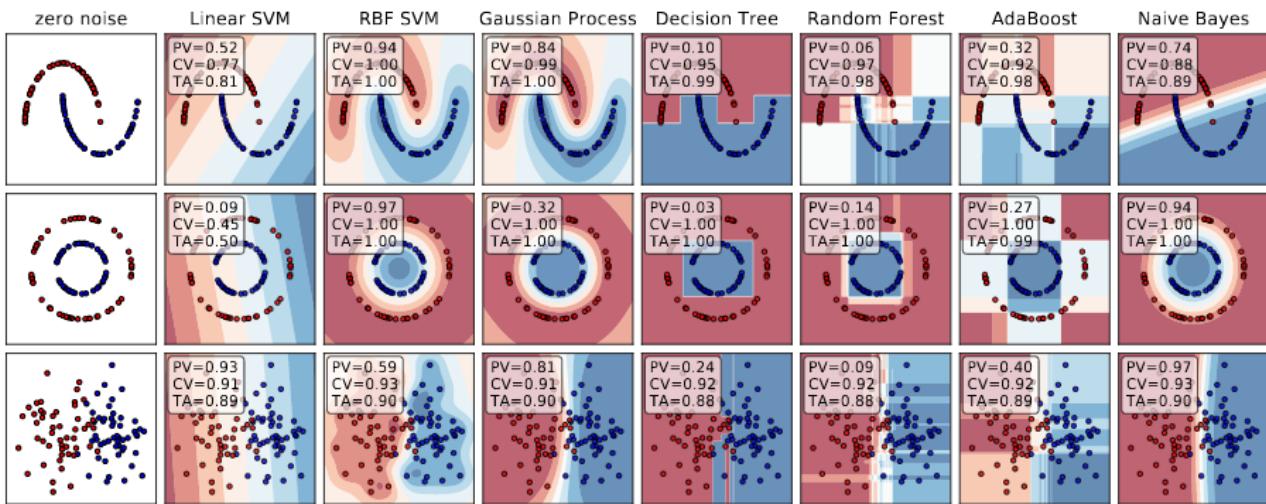


Figura 6.3.: Tres funciones distintas de clasificación con varios modelos, recogiendo PV , acc_{CV} y acc_{test} y la función aprendida.

6. Selección de modelos

Observando los valores PV vemos cómo son mucho más altos cuando la función aprendida del modelo se corresponde con la *forma* original de la función y esto se corresponde con valores casi perfectos de CV y TA. Por otro lado, se asignan valores bajos de PV para los modelos con funciones con una forma más complicada (Random Forest por ejemplo) o cuando no es capaz de captar bien la relación de los datos (Linear SVM), aunque en el primer caso se sigue teniendo valores casi perfectos de CV y TA reflejando lo que habíamos comentado previamente.

Desde luego entre varios modelos con valores CV en torno a 1 podríamos hacer otra selección con quien tenga un mayor valor de PV ya que preferimos quedarnos con un modelo que lo hace igual de bien que el resto pero la función que ha aprendido se adapta mucho más a la **distribución real de los datos**.

6.2.2. Definición

Sea $r \in [0, 1]$ el ratio de etiquetas perturbadas para cada clase (para mantener las distribuciones), consideramos una sucesión creciente de ratios $\{r_i\}_{1 \leq i \leq n}$ e \mathbf{y}_{r_i} el conjunto de etiquetas con un r_i porcentaje de etiquetas perturbadas por clase. Finalmente notamos $acc(S)$ como el máximo valor de la métrica obtenido con \mathcal{H} en $S = (X, \mathbf{y}_{r_i})$, es decir (6.1) [ZHG⁺19].

$$acc(S) = \max_{h \in \mathcal{H}} acc(h), \text{ con } S = (X, \mathbf{y}_{r_i}). \quad (6.1)$$

Con todos estos elementos ya podemos definir formalmente la heurística PV (Def. 6.1, [ZHG⁺19]).

Definición 6.1 (Perturbated Validation). *PV* es el valor absoluto del coeficiente de la regresión lineal (la pendiente) entre la variable independiente, ratio de etiquetas perturbadas r_i , frente la variable dependiente, acierto en el *dataset* perturbado $acc(S_{r_i})$, es decir,

$$PV = \left| \frac{\sum_{i=0}^n (r_i - \bar{r})(acc(S_{r_i}) - \overline{acc(S_r)})}{\sum_{i=0}^n (r_i - \bar{r})^2} \right|.$$

6.3. Implementación

Se ha implementado en una clase en *Python* denominada *PV* que nos permite evaluar los modelos usando la descripción mencionada anteriormente. La documentación se puede encontrar en [Apéndice A](#), aunque merece la pena describir el algoritmo general con pseudo-código: [Algoritmo 4](#) para crear los conjuntos de etiquetas perturbados y [Algoritmo 5](#) para calcular el *PV* para un modelo dado.

Para crear los *datasets* perturbados tomamos los errores de manera que sean **equidistantes**, por lo que solo hace falta pasar como argumento el número de perturbaciones deseadas n , y los errores de inicio y fin err_1 y err_n . A la hora de elegir los índices aleatoriamente se toman de manera que todos sean **diferentes** y de la clase correspondiente en el bucle. La clase que se escoge también se asegura que no sea la que tiene actualmente, de manera que la serie tenga efectivamente una **etiqueta distinta** asociada. Devolvemos las etiquetas perturbadas Y_i como los índices de error r_i necesarios para calcular el *PV*.

Algoritmo 4: $PV(\mathbf{y}, n, err_{ini}, err_{fin})$

```

    // Inicializamos los ratios de error
1 para  $i = 1, \dots, n$  hacer
2    $r_i \leftarrow err_{ini} + (i - 1) \cdot \frac{err_{fin} - err_{ini}}{n - 1};$ 
3 fin
    // Para cada ratio creamos unas etiquetas perturbadas
4 para  $i = 1, \dots, n$  hacer
5   // Copiamos las etiquetas originales
6    $\mathbf{y}^{(i)} \leftarrow \mathbf{y};$ 
7   // Perturbamos para cada clase
8   para  $j = 1, \dots, n_{clases}$  hacer
9     // Perturbamos el % respecto el número de etiquetas de esa clase
10     $n_{per} \leftarrow r_i \cdot \sum_{k=1}^{|y|} [[y_k = j]];$ 
11    // Hacemos una permutación de los índices de la clase para escogerlos
12      aleatoriamente
13     $\sigma \leftarrow Permutacion (\{i \in \mathbb{N} : y_i = j\});$ 
14    // Para cada índice se le pone una clase aleatoria distinta de la
15      original
16    para  $k = 1, \dots, n_{per}$  hacer
17       $\mathbf{y}_{\sigma(k)}^{(i)} \leftarrow Aleatorio (\{1, 2, \dots, j - 1, j + 1, \dots, n_{clases}\});$ 
18    fin
19 fin
20 fin

```

Resultado: $r_1, \dots, r_n, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n)}$

6. Selección de modelos

Algoritmo 5: Calcular-PV(*modelo*, X , $\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n)}$, r_1, \dots, r_n)

```

// Para cada conjunto entrenamos el modelo y obtenemos el accuracy
1 para  $i = 1, \dots, n$  hacer
2    $modelo_{copia} \leftarrow copiar(modelo);$ 
3    $modelo_{copia}.fit(X, \mathbf{y}^{(i)});$ 
4    $acc_i \leftarrow modelo_{copia}.score(X, \mathbf{y}^{(i)});$ 
5 fin
// Calculamos las medias de los errores y el accuracy
6  $\bar{r} \leftarrow \frac{1}{n} \sum_{i=1}^n r_i;$ 
7  $\bar{acc} \leftarrow \frac{1}{n} \sum_{i=1}^n acc_i;$ 
// Calculamos el PV
8  $PV \leftarrow \left| \frac{\sum_{i=0}^n (r_i - \bar{r})(acc_i - \bar{acc})}{\sum_{i=0}^n (r_i - \bar{r})^2} \right|;$ 

```

Resultado: PV

Para obtener el PV introducimos el modelo con el resultado del algoritmo anterior, obteniendo un acc_i asociado con cada entrenamiento y medición de rendimiento del modelo (notar que se copia el modelo para que se entrene desde cero). Finalmente obtenemos el PV para el modelo pasado.

7. Experimentación

7.1. Modelos

Enumeramos y explicamos brevemente los distintos modelos utilizados en la experimentación. En el primer estudio usaremos una serie de modelos con hiperparámetros fijos donde se comentan más detalladamente los que se escogen, aunque se volverán a mencionar en la siguiente parte. Para el segundo se probará a encontrar la mejor configuración de hiperparámetros para cada modelo, por lo que se indicará en cada caso que se irá variando.

7.1.1. LSTM

Como tenemos que crear una arquitectura para clasificación en muchísimos *datasets* para comprobar la selección tampoco nos preocuparemos demasiado por sacar el mayor rendimiento posible, así que realizamos un diseño simple pero funcional. Hemos implementado la clase en *Python*.

Utilizando la misma idea que en las redes convolucionales, extraemos características usando una convolución 1-D de 64 filtros (de salida) y tamaño de núcleo 8 y función de activación ReLU junto a una capa MaxPooling1D con tamaño de núcleo 4 para reducir la dimensión. Después aplicamos una capa LSTM de 80 células quedándonos con el ultimo estado devuelto, junto a una capa Densa de 300 neuronas con activación ReLU que incluye *BatchNormalization* y *Dropout* (técnicas de regularización) donde llegamos finalmente a la última capa de clasificación con n neuronas y activación softmax, siendo $n \in \mathbb{N}$ el número de clases del problema ([Figura 7.1](#)).

Minimizamos mediante **ADAM** (modificación de SGD) la función de pérdida **entropía cruzada categórica** (*categorical crossentropy*, [Def. 7.1](#)) que se encarga de medir la diferencia entre dos distribuciones de probabilidad discretas; en nuestro caso estamos aprendiendo la probabilidad de asignar a cada clase que queremos que sea casi 1 para la etiqueta asignada.

Definición 7.1 (Entropía cruzada categórica). Sea n el número de clases e \mathbf{y} el vector de probabilidades real, dado $\hat{\mathbf{y}}$ el vector de probabilidades obtenido por un modelo, la entropía cruzada categórica es una función $E : [0, 1]^n \rightarrow \mathbb{R}$ definida por:

$$L(\hat{\mathbf{y}}; \mathbf{y}) = \sum_{i=1}^n y_i \log(\hat{y}_i).$$

El entrenamiento se hará durante 300 épocas, con un tamaño de *batch* de 128, usando un 10 % de los datos para el conjunto de validación y añadiendo una parada temprana cuando el acc_{val} no mejore un 0.1 en 50 épocas, guardando los pesos con mejor resultado.

Esta configuración permite ser relativamente flexible para diferentes *datasets* ya que dejamos muchas épocas para datos más complejos, y una parada temprana para cuando no haya mejora en los simples.

7. Experimentación

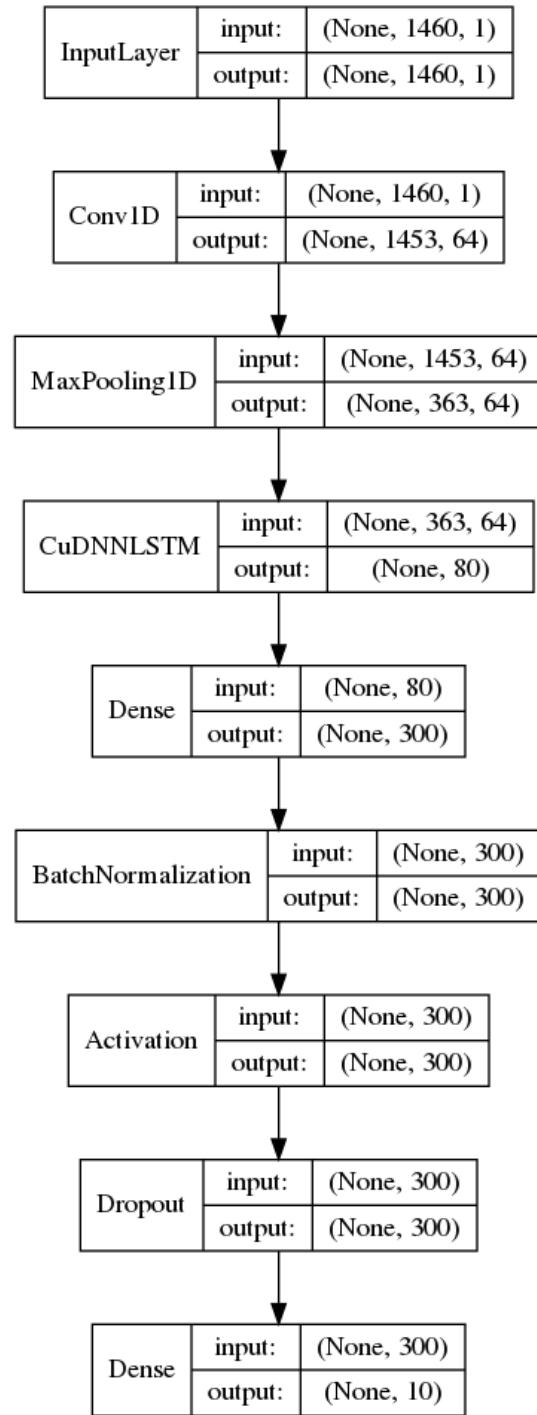


Figura 7.1.: Arquitectura del clasificador LSTM con series de longitud 1460 y 10 clases.

7.1.2. SVM

Las Máquinas de Soporte Vectorial (*Support Vector Machine*, SVM) [CV95] implementadas en *Python* por la librería *scikit-learn* en [SL2of]. Este método se encarga de buscar el mejor hiperplano que **separe** los datos por sus clases, entendiendo como mejor el que deje a **más distancia** los puntos más cercanos (siendo estos los puntos de soporte) que tenga ya que este modelo generalizará mejor y por tanto tendrá menor **sobreajuste** (Figura 7.2 [Jav]).

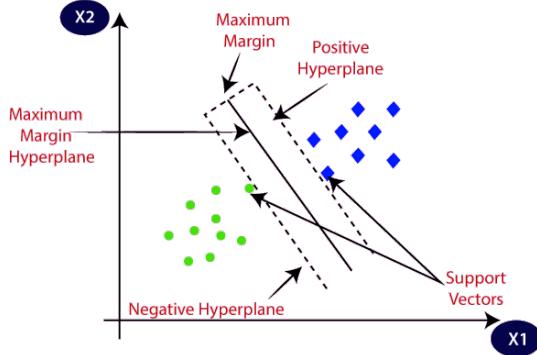


Figura 7.2.: Ejemplo de SVM lineal.

Además se puede incluir una transformación no lineal en los datos para intentar conseguir una mejor separación en otros espacios, mediante la denominada función *kernel*. Las más usadas son las siguientes, siendo γ un hiperparámetro:

1. Polinómica de grado d y radio r : $(\gamma < x, x' > +r)^d$.
2. Base radial (*Radial Basis Function*, RBF): $\exp(-\gamma ||x - x'||_2^2)$.
3. Sigmoide de radio r : $\tanh(\gamma < x, x' > +r)$.

Para nuestro modelo fijamos el parámetro de regularización C a 1 (cuanto más vale, menos fallos permite), usando como función *kernel* RBF y dejamos que γ tome un valor heurístico a $1/(M * \text{Var}(X))$ siendo M el número de características y X los datos.

7.1.3. Árboles de decisión

Los árboles de decisión son un modelo muy simple que se puede entender como un conjunto de **reglas** asociadas a las características (Figura 7.3), o como una serie de hiperplanos paralelos a los ejes que partitionan el espacio. Estas reglas nos permiten **entender** porqué el árbol llega a la predicción, cosa que no suele suceder en la mayoría de modelos.

En general el gran problema que presentan es el alto **sobreajuste** que tienen: a costa de tener un error/sesgo casi nulo se tiene demasiada varianza. Por ejemplo, para dos muestras de datos un poco distintas pueden dar lugar a dos árboles completamente distintos.

Usamos unos cuantos subtipos/implementaciones usualmente utilizados:

- **CART**: Classification and Regression Trees [BFSO84] implementado en *Python* en la librería *scikit-learn* [SL2oa]. Fijamos la profundidad máxima a 20 para evitar sobreajuste.

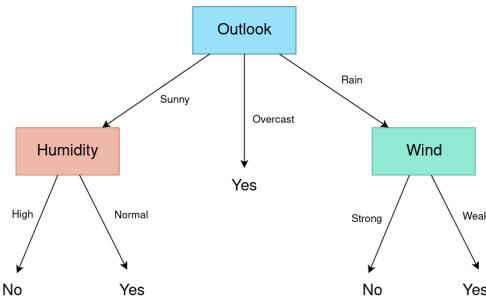


Figura 7.3.: Árbol de decisión con reglas para si debería jugar al tenis.

- **C4.5**: [Qui14] implementado en *R* en el paquete *partykit* [HBH⁺20].
- **C5.0**: mejora de C4.5 [Qui19] implementado en *R* en el paquete *C50* [KWC⁺20]. Se añade un modelo sin *boosting* y otro con 10 árboles.
- **RPart**: Recursive partitioning for classification trees [BFSO84] implementado en *R* en el paquete *rpart* [ATR19].
- **CTree**: Conditional inference tree [HHZo6, HHZ15] implementado en *R* en el paquete *ctree* [Hot20].

Excepto CART, para el resto se ha usado la librería *rpy2* [Gau20] para usar en *Python* las clases en *R*.

7.1.4. Random Forest

Random Forest [Ho95, Bre01] con implementación en *Python* de la librería *scikit-learn* [SL20d]. Un modelo no lineal que construye una **agrupación** (*ensemble*) de árboles de decisión mediante la técnica de *bootstrap/bagging*. La idea detrás de este método es **evitar** el sobreajuste ocasionado por la alta varianza que tiene un solo árbol de decisión, utilizando un gran cantidad de ellos y además entrenándolos cada uno con sola muestra aleatoria con reemplazamiento (Figura 7.4 [OA18]).

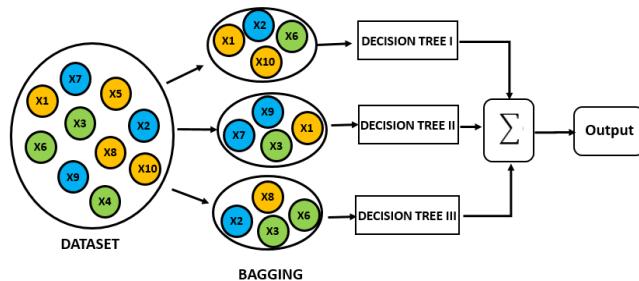


Figura 7.4.: Construcción de Random Forest.

Como en los árboles, el espacio se partitiona por hiperplanos paralelos a los ejes, considerando que ahora se toma el voto mayoritario de la agrupación de todos. Sin embargo se pierde la interpretabilidad clara que pudiera tener un solo árbol, si bien podemos intentar observar las características más importantes según la cantidad de veces que aparecen en los árboles.

Usando una regla heurística se fija el número de características de cada árbol a \sqrt{M} siendo M el número de características del *dataset* y usamos el criterio de la impureza Gini [RM05] para escoger las características que dividen al árbol.

Escogemos para el modelo 200 árboles con una profundidad máxima de 20, que según el *dataset* podrá ser poco o mucho pero en principio lo ponemos así para evitar en la medida de lo posible un sobreajuste constante.

7.1.5. k -NN

k -Nearest Neighbors (k -NN) [CH67] usando como base la implementación en *Python* de la librería *scikit-learn* [SL20b]. Es un modelo no lineal simple que cambia el enfoque de los modelos anteriores. El entrenamiento consiste únicamente en **copiar** los datos, de manera que para clasificar un punto considera los k vecinos (datos) **más cercanos** de los que fueron guardados, en función de la métrica que se le haya indicado. Por tanto lo que realiza el modelo es una partición del espacio cuya forma estará determinada por la distribución de los datos y el k elegido; un ejemplo lo tenemos en Figura 7.5 [M18].

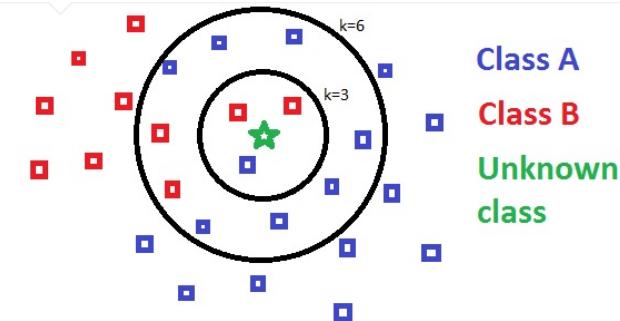


Figura 7.5.: Ejemplo de k -NN. Con $k = 3$ se toma la clase B, con $k = 6$ la A.

Para nuestro modelo consideramos la métrica usual euclídea $\|\cdot\|_2$ y usamos la regla heurística que recomienda usar $k \approx \sqrt{N}$ siendo N el número de series del *dataset* de entrenamiento.

Como estamos en el contexto de las series temporales, se ha decidido añadir otro modelo k -NN pero en vez de usar la métrica euclídea usamos la *semi-métrica Dynamic Time Warping* (DTW) [BC94]. Esta *semi-métrica* [Jai18] mide la distancia con el **alineamiento óptimo** entre dos series temporales, de manera que es muy útil para medir series que exhiben ciertos patrones pero que tienen **pequeñas variaciones** (por ejemplo, una pequeña amplitud o desplazamiento) entre sí. Si suponemos que las series con misma etiqueta siguen una cierta forma bajo esta distancia deberían obtener valores muy bajos, si lo unimos con el algoritmo k -NN tenemos un modelo que puede funcionar bastante bien.

Ejemplificamos visualmente lo que calcula DTW comparando con la métrica euclídea en Figura 7.6 [VNZ12]; observamos que para cada punto se busca la mejor coincidencia con la

7. Experimentación

otra serie, consiguiendo una buena correspondencia entre series parecidas.

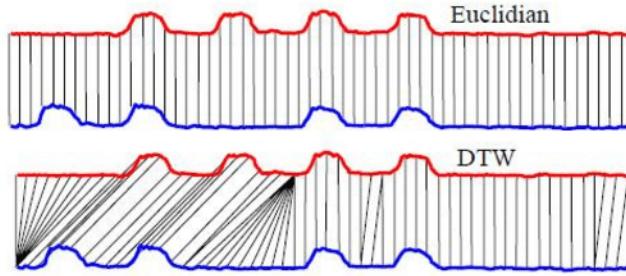


Figura 7.6.: Funcionamiento de DTW frente $\|\cdot\|_2$.

Sin embargo, DTW es muy **costoso** de usar, puesto que para cada punto de la serie se calcula una distancia ($\|\cdot\|_1$ o $\|\cdot\|_2$) con el resto de puntos de la otra serie, y si tenemos en cuenta que en k -NN se calculan las distancias de cada serie que se quiera etiquetar con todas las series de entrenamiento el tiempo empleado en cómputos se dispara. Por ello para nuestro modelo usaremos una implementación eficiente en R [Leo20] junto al uso de una ventana de tamaño 5 (solo se buscan las coincidencias en un entorno de radio 5). Finalmente fijamos $k = 3$ ya que esperamos una mejor agrupación con DTW (podríamos haber cogido $k = 1$ pero si hacemos esto provocaríamos que $acc_{train} = 1$, sobreajustando demasiado) y evitamos aumentar aún más los cálculos.

El modelo $k - NN$ euclídeo para tomar el k automático se ha implementado en *Python* y para k -NN + DTW se ha usado otra vez la librería *rpy2* para usar el paquete de *R* en *Python*.

7.2. Experimentación previa

Antes de hacer un trabajo de experimentación extenso se hicieron varias pruebas para determinar la eficacia previa del *PV*, no se detalla en concreto ya que fueron pequeños casos (unos cuantos *datasets* y clasificadores) de los experimentos grandes que comentaremos en [Capítulo 7](#) para poder corroborar rápidamente los resultados en [ZHG⁺19] y comprobar la viabilidad de los experimentos. Se pudieron notar los siguientes aspectos:

1. La **forma** de los puntos: como se indicaba en [ZHG⁺19], los puntos $\{r_i, acc_i\}$ se alinean con una recta con pendiente no positiva generalmente, los puntos a veces pueden estar un poco por encima/debajo de ella pero se ve claramente la tendencia de recta.
2. El **tamaño** del *dataset*: un *dataset* muy pequeño (depende de lo complicada que sea la distribución original) puede dar lugar a que el *PV* sea muy inestable entre ejecuciones debido a la aleatoriedad de escoger y cambiar etiquetas. No es especialmente preocupante puesto que para que los modelos aprendan correctamente se necesita una cantidad importante de datos.
3. El **intervalo** donde tomar los r_i (errores): la propuesta original de [ZHG⁺19] indicaba el intervalo $[0, 0.5]$ pero se obtuvieron resultados que indicaban que a partir de 0.4 las alteraciones eran demasiado fuertes. Nuestras observaciones concuerdan con la nueva versión de [ZHG⁺19], que restringieron a $[0, 0.3]$.

4. Valores **válidos** de n (número de *datasets* perturbados): [ZHG⁺¹⁹] indica usar solamente 3, aunque es posible tomar cualquier valor mayor que este. Cuantos más puntos más robusto será el valor PV aunque computacionalmente será más costoso, por lo que un valor de 5 es más que razonable para obtener resultados buenos.
5. **Rango** de valores del PV : como reflejan en [ZHG⁺¹⁹] la mayoría de valores del PV obtenidos se encuentran en el intervalo $[0, 1]$ si bien se obtienen a veces valores por encima. Para los valores en $[1, 2]$, en [ZHG⁺¹⁹] se propone corregirlos al $[0, 1]$ penalizándolos con la transformación (7.1). En nuestro caso dejaremos los datos originales sin transformar, para notar explícitamente que se han obtenido estos valores por encima de los usuales.

$$PV_{corregido} = 1 - |PV - 1|. \quad (7.1)$$

6. **Interpretabilidad** del PV : el PV nos indica en líneas generales si el modelo ha entendido correctamente el patrón subyacente en los datos. Observamos que conforme **aumenta** PV , el valor de las métricas acc_{train} y acc_{test} también lo hace y la tendencia del sobreajuste disminuye.
7. **Resultados** del PV : observamos que modelos que obtienen un $PV \approx 0$ suelen tener un acc_{train} bastante alto mientras que en acc_{CV} y acc_{test} bajan bastante, haciéndose evidente el sobreajuste que existe. Además, un modelo puede tener mejor PV que otro pero no tiene por qué mantenerse el orden con acc_{test} ; lo que sí observamos es que conforme aumenta PV , la tasa de acc_{train} y acc_{test} aumentan y el sobreajuste disminuye.
8. **Cambio en las perturbaciones**: se intentó probar a perturbar los datos en sí en vez de las etiquetas (para poder trasladar el PV a problemas con métricas continuas o problemas no supervisados) con distintas pruebas, por ejemplo ruido gaussiano; pero no tuvieron el efecto deseado y al no tener unos resultados muy claros se decidió descartar esta línea.

Como los resultados de [ZHG⁺¹⁹] parecían corresponder con estos preliminares dimos el visto bueno para pasar a la experimentación extendida para comprobar de una manera más extendida y analizar detalladamente la eficacia y los resultados del PV .

7.3. Elección del mejor modelo

7.3.1. Descripción

7.3.1.1. Objetivos

En este experimento evaluamos la nueva heurística PV frente a las medidas clásicas de validación acc_{CV} y acc_{test} . De [ZHG⁺¹⁹] y la experimentación previa sabemos que aunque un modelo obtenga un mejor valor de PV frente a otro (sobre todo cuando los PV rondan valores muy bajos) esto no implicará con una seguridad absoluta que el acc_{test} sea mejor también, sobretodo si el rango del PV es bajo (en un entorno cercano al 0).

Por tanto lo que esperamos en este experimento es:

7. Experimentación

1. Que un *PV* alto indique que el modelo sea **bueno**: esperamos que un modelo sea bueno si la función aprendida se **corresponde con la función de la distribución** de los datos. Sin embargo como desconocemos esta función no podemos compararlas para comprobarlo directamente pensamos que si el modelo ha aprendido la función correcta deberá obtener una **buena tasa de acierto** en **cualquier conjunto** de datos, es decir deberá obtener un valor en ambas métricas acc_{train} y acc_{test} y ser muy iguales, ya que no debería haber **sobreajuste**.
2. Si $PV \approx 0$ el modelo **sobreajustará**: si el $PV \approx 0$ entendemos que el modelo se ha pasado con la complejidad o bien no ha conseguido entender nada de la función de etiquetado. En cualquier caso esperamos que en otra muestra distinta el modelo lo haga mucho peor, notando una tasa de acc_{test} peor y probablemente con una gran diferencia con acc_{train} si el modelo se ha pasado con la complejidad.

7.3.1.2. Datasets

Utilizamos la base de datos "*UEA & UCR Time Series Classification*"[\[ABK20\]](#) que reúne 158 *dataset* de series temporales estandarizadas para problemas de clasificación. Cogeremos los 114 *datasets* de series temporales **unidimensionales** para evitar altos tiempos en entrenamiento, mezclamos la partición ya hecha en *train/test* ya que muchos *datasets train* tienen **muy pocas** series y hay un gran desequilibrio, y volvemos a partir pero con un 80/20 %. Finalmente nos quedaremos con los que el *dataset train* cumpla las siguientes condiciones:

1. Tiene al menos **100 series**: aunque puede depender de la distribución, en general necesitamos muchos datos para que los modelos aprendan. Además un número tan pequeño de series puede ocasionar una gran inestabilidad a la hora de obtener el *PV*.
2. Tiene al menos **5 series por clase**: si no se tiene al menos 5 elementos por clase (se podría poner un umbral mayor incluso) como al hacer el *PV* alteramos las etiquetas el error que introducimos sería demasiado fuerte.
3. La clase mayoritaria no tiene una **proporción mayor al 70 %**: usamos la métrica *acc* por lo que evitamos *datasets* con un gran desequilibrio entre número de instancias por clase.

Además de considerar los *datasets* originales en medida temporal (TS), también tomaremos su versión en medidas de complejidad (CMFTS). La lista final de los 95 *datasets* se encuentra detallada en [Tabla 7.1](#) y [Tabla 7.2](#).

7.3.1.3. Clasificadores

Ya comentábamos previamente los modelos utilizados, así que los listamos simplemente ahora:

- **LSTM**: Con la arquitectura ya indicada, se entrena como máximo 300 épocas con tamaño de *batch* a 128 y un 10% de validación, parando si no hay una mejora de al menos de 0.1 en acc_{val} en 50 épocas y quedándose con la configuración que mejor acc_{val} haya tenido.
- **SVM**: $C = 1$, función de *kernel* RBF y $\gamma = 1/(M * Var(X))$ (M longitud series y $Var(X)$ varianza de las series).

7.3. Elección del mejor modelo

Nombre	Tamaño Train	Tamaño Test	Nº clases	Longitud TS	Longitud CMFTS
ACSF1	160	40	10	1460	38
Adiac	624	157	37	176	38
ArrowHead	168	43	3	251	38
BME	144	36	3	128	38
CBF	744	186	3	128	38
ChlorineConcentration	3445	862	3	166	37
CinCECGTorso	1136	284	4	1639	38
Computers	400	100	2	720	38
CricketX	624	156	12	300	38
CricketY	624	156	12	300	38
CricketZ	624	156	12	300	38
Crop	19200	4800	24	46	36
DiatomSizeReduction	257	65	4	345	36
DistalPhalanxOutlineAgeGroup	431	108	3	80	36
DistalPhalanxOutlineCorrect	700	176	2	80	36
DistalPhalanxTW	431	108	6	80	36
ECG200	160	40	2	96	36
ECG5000	4000	1000	5	140	38
ECGFiveDays	707	177	2	136	38
ElectricDevices	13309	3328	7	96	36
EOGHorizontalSignal	579	145	12	1250	38
EOGVerticalSignal	579	145	12	1250	38
EthanolLevel	803	201	4	1751	37
FaceAll	1800	450	14	131	38
FacesUCR	1800	450	14	131	38
FiftyWords	724	181	50	270	38
Fish	280	70	7	463	37
FordA	3936	985	2	500	38
FordB	3556	890	2	500	38
FreezerRegularTrain	2400	600	2	301	38
FreezerSmallTrain	2302	576	2	301	38
Fungi	163	41	18	201	38
GunPoint	160	40	2	150	38
GunPointAgeSpan	360	91	2	150	38
GunPointMaleVersusFemale	360	91	2	150	38
GunPointOldVersusYoung	360	91	2	150	38
Ham	171	43	2	431	38
HandOutlines	1096	274	2	2709	37
Haptics	370	93	5	1092	38
Herring	102	26	2	512	37
HouseTwenty	127	32	2	2000	38
InlineSkate	520	130	7	1882	37
InsectEPGRegularTrain	248	63	3	601	38
InsectEPGSmallTrain	212	54	3	601	38
InsectWingbeatSound	1760	440	11	256	38
ItalyPowerDemand	876	220	2	24	36
LargeKitchenAppliances	600	150	3	720	38
Lightning7	114	29	7	319	38

Tabla 7.1.: Datasets del experimento.

7. Experimentación

Nombre	Tamaño Train	Tamaño Test	Nº clases	Longitud TS	Longitud CMFTS
Mallat	1920	480	8	1024	38
MedicalImages	912	229	10	99	38
MelbournePedestrian	2920	730	10	24	36
MiddlePhalanxOutlineAgeGroup	443	111	3	80	36
MiddlePhalanxOutlineCorrect	712	179	2	80	36
MiddlePhalanxTW	442	111	6	80	36
MixedShapesRegularTrain	2340	585	5	1024	38
MixedShapesSmallTrain	2020	505	5	1024	38
MoteStrain	1017	255	2	84	36
NonInvasiveFetalECGThorax1	3012	753	42	750	38
NonInvasiveFetalECGThorax2	3012	753	42	750	38
OSULeaf	353	89	6	427	38
PhalangesOutlinesCorrect	2126	532	2	80	36
Plane	168	42	7	144	38
PowerCons	288	72	2	144	38
ProximalPhalanxOutlineAgeGroup	484	121	3	80	35
ProximalPhalanxOutlineCorrect	712	179	2	80	35
ProximalPhalanxTW	484	121	6	80	35
RefrigerationDevices	600	150	3	720	38
ScreenType	600	150	3	720	38
SemgHandGenderCh2	720	180	2	1500	38
SemgHandMovementCh2	720	180	6	1500	38
SemgHandSubjectCh2	720	180	5	1500	38
ShapeletSim	160	40	2	500	38
ShapesAll	960	240	60	512	38
SmallKitchenAppliances	600	150	3	720	38
SmoothSubspace	240	60	3	15	30
SonyAIBORobotSurface1	496	125	2	70	36
SonyAIBORobotSurface2	784	196	2	65	36
StarLightCurves	7388	1848	3	1024	38
Strawberry	786	197	2	235	38
SwedishLeaf	900	225	15	128	38
Symbols	816	204	6	398	38
SyntheticControl	480	120	6	60	36
ToeSegmentation1	214	54	2	277	38
Trace	160	40	4	275	38
TwoLeadECG	929	233	2	82	36
TwoPatterns	4000	1000	4	128	38
UMD	144	36	3	150	38
UWaveGestureLibraryAll	3582	896	8	945	38
UWaveGestureLibraryX	3582	896	8	315	38
UWaveGestureLibraryY	3582	896	8	315	38
UWaveGestureLibraryZ	3582	896	8	315	38
WordSynonyms	724	181	25	270	38
Worms	206	52	5	900	38
WormsTwoClass	206	52	2	900	38
Yoga	2640	660	2	426	38

Tabla 7.2.: Datasets del experimento (continuación).

- **CART**: profundidad máxima de 20.
- **C4.5**.
- **C5.0**.
- **C5.0 + boosting**: *boosting* con 10 árboles.
- **RPart**.
- **CPart**.
- **Random Forest**: 200 árboles con profundidad máxima de 20.
- **k -NN euclídeo**: $k = \sqrt{N}$ siendo N el número de series de entrenamiento.
- **k -NN + DTW**: $k = 3$ y tamaño de ventana a 5.

Notamos que estos modelos no estarán adecuados con la mejor configuración de hiperparámetros para cada *dataset* pero se puede seguir haciendo la comparación mediante las métricas.

7.3.1.4. Métricas

Las métricas a medir son:

- **PV**: la nueva heurística a medir, tomando $n = 5$, $r_1 = 0.1$ y $r_n = 0.3$.
- acc_{train} : acierto en el conjunto de entrenamiento.
- acc_{CV} : acierto por validación cruzada con 5 particiones en el *dataset* de entrenamiento.
- acc_{test} : acierto en el conjunto test.

Guardaremos también los resultados de los acc_i obtenidos al calcular el PV.

7.3.2. Resultados

Hemos guardado dos tipos de resultados, disponibles en el repositorio del proyecto:

- Los valores de cada métrica para cada clasificador en un archivo de tipo *csv* (Nombre-Dataset.csv). Un ejemplo formateado a tabla sería [Tabla 7.3](#).
- Una gráfica mostrando los puntos (r_i, acc_i) obtenidos para cada clasificador para calcular el PV, junto con la regresión lineal obtenida. Un ejemplo lo tenemos en [Figura 7.7](#).

Unimos todos los datos obtenidos para poder visualizar la relación entre la heurística PV y las métricas existentes, a los que añadimos unas rectas de regresión para mostrar la tendencia de los datos, aunque en un caso se ha tenido que usar una parábola de regresión ([Figura 7.8](#)). Además se ha añadido la distribución en un histograma de los valores del PV ([Figura 7.9](#)).

Finalmente hemos recogido en una tabla ciertos valores interesantes de cada modelo para estudiar sus comportamientos individuales: los valores máximos, mínimos, media, desviación típica y número de valores por encima de 1 de PV; la media de acc_{train} y acc_{test} y finalmente la media y desviación típica de los errores cuadráticos medios de la regresión lineal obtenida en el cálculo del PV ([Tabla 7.4](#))

7. Experimentación

Classifier	PV	acc_{train}	acc_{CV}	acc_{test}	acc_1	acc_2	acc_3	acc_4	acc_5
RBF-SVM	0.1763	0.1378	0.1385	0.1592	0.1298	0.1234	0.1202	0.1154	0.0897
Random Forest	0.0	1.0	0.6856	0.7389	1.0	1.0	1.0	1.0	1.0
CART	0.0833	0.9968	0.5012	0.5223	1.0	1.0	0.9936	0.9872	0.9856
C4.5	0.4647	0.9263	0.5481	0.5987	0.859	0.867	0.8189	0.8045	0.774
C5.0	0.4615	0.9263	0.5562	0.6306	0.8622	0.851	0.8349	0.8029	0.7708
C5.0-Boosting	0.0833	1.0	0.6443	0.7261	0.9952	0.9952	0.9952	0.9856	0.9792
CTree	0.9423	0.5577	0.3942	0.465	0.4311	0.4199	0.3814	0.3141	0.2484
RPart	0.9327	0.6635	0.4791	0.5096	0.6154	0.5545	0.5401	0.4792	0.4199
KNN	0.5481	0.5192	0.4023	0.4586	0.4551	0.4535	0.4183	0.3814	0.3542
3NN+DTW	0.0288	0.8846	0.4616	0.5159	0.8798	0.8718	0.8766	0.8606	0.8926
LSTM	1.5513	0.391	0.4343	0.3376	0.4503	0.0417	0.2292	0.0417	0.0625

Tabla 7.3.: Resultados de Adiac.csv versión TS.

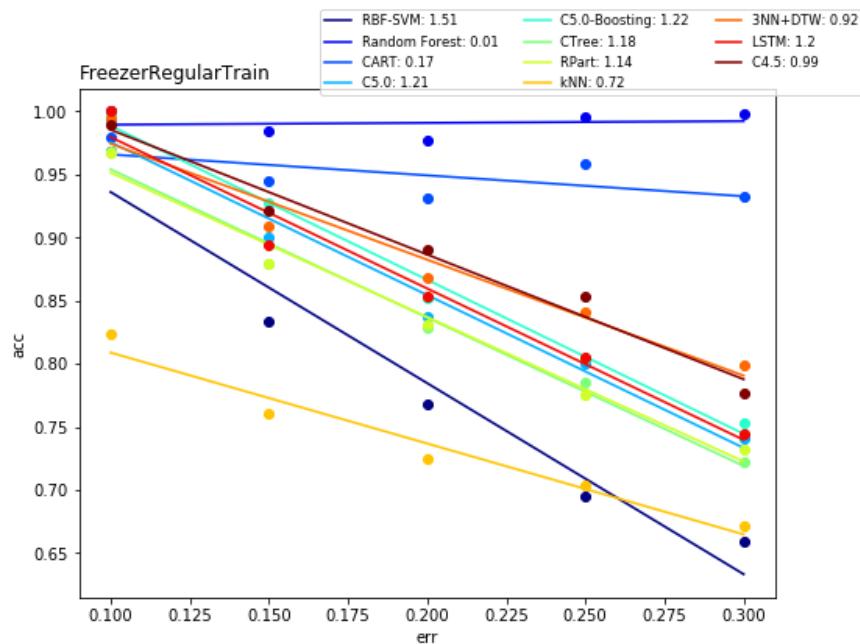


Figura 7.7.: Resultados acc y recta de regresión al calcular PV en FreezerRegularTrain versión TS.

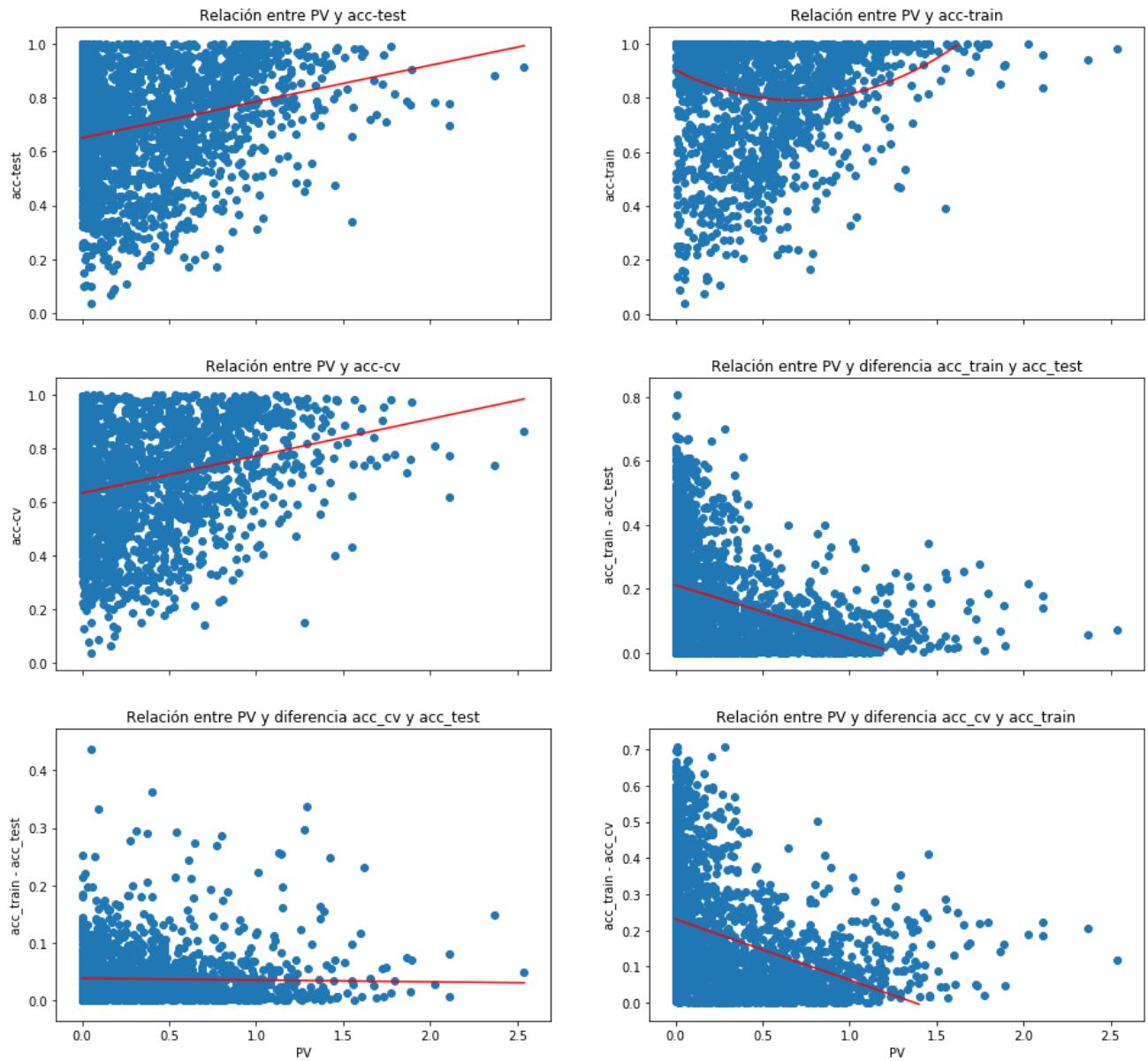


Figura 7.8.: Relaciones entre PV y otras métricas.

7. Experimentación

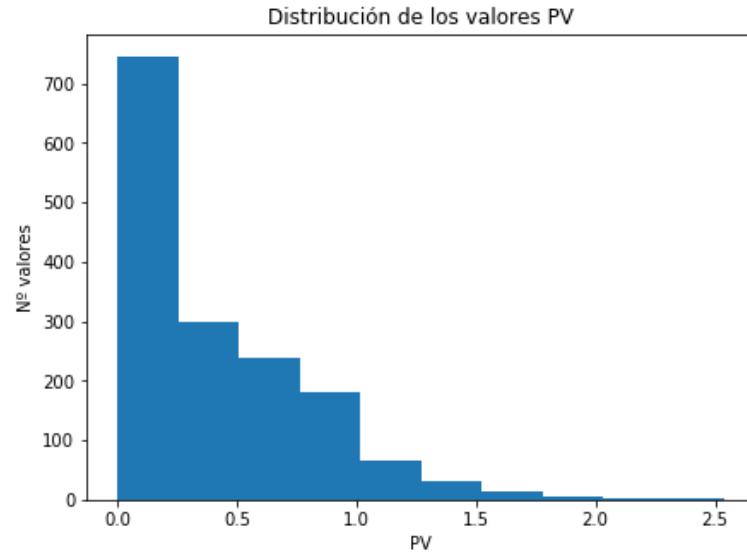


Figura 7.9.: Distribución de valores del PV .

Classifier	min PV	max PV	\bar{PV}	σ_{PV}	$ PV > 1 $	\overline{acc}_{train}	\overline{acc}_{test}	\overline{MSE}	σ_{MSE}
RBF-SVM	0.000	1.111	0.511	0.302	6	69.866 %	65.329 %	0.0001	0.0003
Random Forest	0.000	0.285	0.017	0.048	0	99.955 %	79.916 %	0.0000	0.0000
CART	0.000	0.858	0.060	0.110	0	99.608 %	70.883 %	0.0001	0.0002
C4.5	0.000	2.112	0.440	0.435	26	94.114 %	71.776 %	0.0012	0.0032
C5.0	0.006	1.747	0.392	0.369	18	93.478 %	72.474 %	0.0007	0.0014
C5.0-Boosting	0.000	2.025	0.363	0.512	33	98.033 %	77.784 %	0.0008	0.0022
CTree	0.000	2.366	0.731	0.342	28	72.754 %	67.755 %	0.0006	0.0013
RPart	0.000	0.995	0.518	0.248	0	77.188 %	67.841 %	0.0003	0.0004
KNN	0.008	0.977	0.519	0.253	0	69.065 %	66.173 %	0.0002	0.0004
3NN+DTW	0.000	0.763	0.246	0.186	0	90.052 %	72.121 %	0.0002	0.0003
LSTM	0.007	2.540	0.593	0.472	37	67.746 %	62.959 %	0.0050	0.0059

Tabla 7.4.: Diferentes resultados de cada modelo.

7.3.3. Análisis

Observando [Figura 7.8](#) podemos comprobar que las hipótesis que queríamos comprobar estaban en lo correcto: cuando PV aumenta, la tendencia de acc_{test} y acc_{train} es creciente indicando que los modelos son **buenos** en cuanto a **mejor ajuste** y menor **sobreajuste**; notándose además que cuando $PV \approx 0$ observamos un gran cúmulo de valores de $acc_{train} \approx 1$ mientras que acc_{test} toma valores en toda la franja $[0, 1]$ manifestándose en el **fuerte sobreajuste** que existe en ciertos modelos (razón por la que hemos usado una parábola de regresión).

Además entre PV y acc_{CV} , como acc_{CV} es un estimador de acc_{test} , se tiene una relación muy idéntica entre PV y acc_{test} . También es notable el hecho de que conforme aumenta PV la diferencia entre acc_{CV} y acc_{test} disminuya ligeramente, indicando que acc_{CV} se vuelve mejor estimador (menor varianza) conforme más vale sea PV .

Por otro lado, queda claro que PV no nos marca el modelo que mayor acc_{test} tenga, observamos que hay valores altos de acc_{test} a pesar de que el PV sea bajo; incluso con valores más grandes de PV no se nos puede asegurar nada, pero probablemente tengan unos muy buenos resultados.

De [Figura 7.9](#) notamos que la mayoría de valores se concentran en el intervalo esperado $[0, 1]$ con una pequeña franja en $[1, 1.5]$ que era posible también. Únicamente se han detectado una pequeña cantidad de puntos (una veintena) en $[1.5, 2]$, y únicamente 5 puntos en $[2, 2.6]$, que aunque en principio son valores atípicos respecto de la distribución del PV los estudiaremos más adelante para intentar averiguar las causas de estos valores.

En [Tabla 7.4](#) tenemos distintos datos que resaltan: lo más notable es los resultados de los modelos C4.5, C5.0 (sin y con *boosting*), CTree y LSTM que obtienen unos valores max PV bastante por encima de 1, que se une con una mayor dispersión σ_{PV} , una cantidad alta de valores atípicos $|PV > 1|$ y unos mayores errores de ajuste en las rectas de regresión para calcular el PV tanto en la media MSE como en la dispersión σ_{MSE} .

Todos estos indicadores nos están diciendo que para estos modelos se obtienen algunas veces, al calcular el PV , unos valores de acc que no se ajustan correctamente con una recta provocando así unos valores más erráticos del PV que ocasionan los altos valores del PV . Esto nos provoca un interés en separar y analizar el comportamiento de ambos grupos por separado.

Por otro lado, también se muestra claramente el problema de los árboles y del 3-NN con el alto sobreajuste manifestándose con un alto valor de \overline{acc}_{train} pero que se distancia mucho de \overline{acc}_{test} , cosa que no ocurre en el resto de modelos.

Finalmente para analizar la cuestión de estos modelos inestables vamos a separar en dos grupos y observar los resultados obtenidos pero en cada grupo ([Figura 7.10](#), [Figura 7.11](#)).

En ambos grupos, como ya veníamos viendo de los resultados globales, se mantienen las relaciones del PV que habíamos comentado, sin embargo vemos como el grupo 1 muestra unos resultados más estables:

- Excepto 5 valores que están en $[1, 1.1]$, todos los valores PV se encuentran en el intervalo esperado $[0, 1]$.
- La correlación entre PV y acc_{test} es muy fuerte para SVM, RPart y k -NN, donde los datos forman casi una recta de pendiente 1. Para DTW se toma también la tendencia pero con una curva arqueada.
- En el caso de Random Forest y CART se nota mucho más el **fuerte sobreajuste** a los que tienden estos modelos que obtienen valores muy pequeños de PV , propiciando valores perfectos de acc_{train} pero muy dispersos en acc_{test} .

7. Experimentación

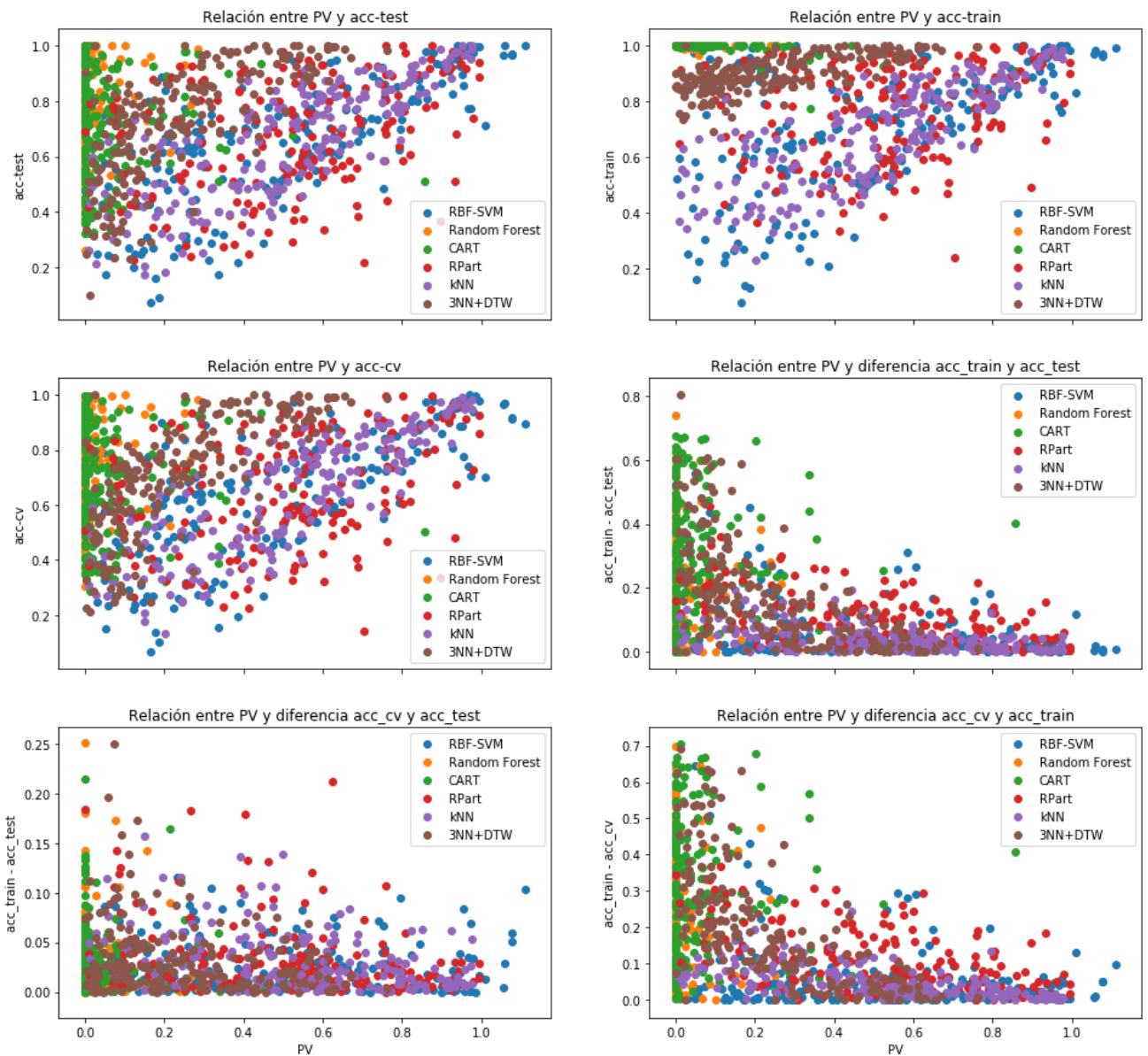


Figura 7.10.: Resultados del grupo 1.

7.3. Elección del mejor modelo

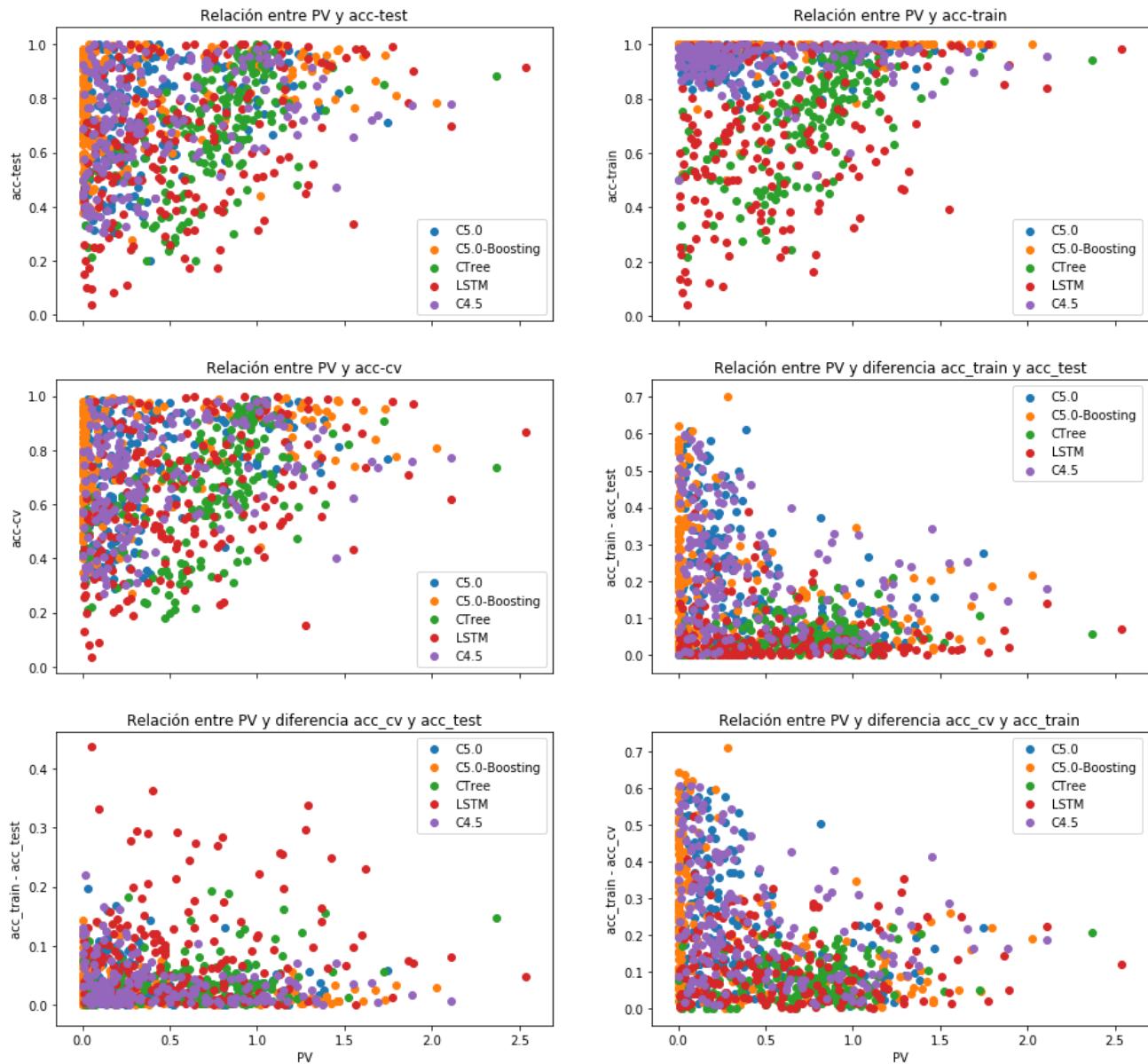


Figura 7.11.: Resultados del grupo 2 (la escala ha cambiado).

7. Experimentación

En el grupo 2 vemos unos resultados parecidos en general aunque aquí ya se observan los valores atípicos más frecuentes, que llegan incluso por encima de 2. De hecho los datos de LSTM presentan valores peores (puntos con PV altos pero acc_{test} bajos, sobreajustes muy por encima de los normales) probablemente de una alta varianza que obtenga soluciones inestables. También sea efecto de usar una configuración y entrenamiento fijados para un montón de modelos diferentes, ya que las redes neuronales necesitan estudiar si el entrenamiento está siendo adecuado y converge a una solución correctamente.

Para acabar, resumamos lo obtenido: nuestras hipótesis sobre el PV se cumplen, si bien parece que hay ciertos modelos como el LSTM que pueden presentar una mayor inestabilidad a la hora de obtener el PV por lo que habría que tener cuidado y comprobar que se calcula correctamente. No podemos considerar sustituir la selección clásica con el PV ya que como hemos visto, no tiene por qué tomar el mejor modelo en base al menor error en una muestra distinta. Sin embargo sí podemos utilizarlo como **selección complementaria**, ayudándonos a hacer una criba más entre modelos con rendimientos muy parecidos asegurándonos así de que escogemos un modelo que ha aprendido mejor la **relación de los datos**.

7.4. Hiperparametrización

7.4.1. Descripción

7.4.1.1. Objetivos

Comprobada la utilidad de la heurística PV , en el contexto de la elección de los **mejores hiperparámetros** de un modelo es especialmente útil ya que normalmente nos encontramos con muchos valores para los que los modelos obtienen un acc_{cv}/acc_{test} **muy parecidos** mientras que la complejidad va cambiando. Esto es de especial interés, ya que lo ideal es buscar un modelo con un buen ajuste y que generalice lo mejor posible para evitar tiempos de computación innecesarios y sobreajustes que pudieran darse.

Buscaremos así si PV destaca en la búsqueda de la mejor configuración de hiperparámetros posible para un modelo dado, frente a la métrica clásica acc_{CV} . Para ello escogeremos unos cuantos *datasets* y unos cuantos modelos típicos buscando los hiperparámetros óptimos de cada modelo en cada dataset, y veremos cual es el comportamiento entre las métricas frente a la variación de los hiperparámetros.

7.4.1.2. Datasets

Cogemos cuatro *dataset* de "UEA & UCR Time Series Classification" [ABK20] ([Tabla 7.1](#), [Tabla 7.2](#)): *ChlorineConcentration*, *PowerCons*, *SmoothSubspace*, *ProximalPhalanxTW*.

7.4.1.3. Clasificadores

Hemos seleccionado cuatro clasificadores de los que habíamos usado en el anterior experimento que tienen hiperparámetros fáciles de comprender. En principio usamos las mismas configuraciones que el anterior experimento, variando solamente los hiperparámetros que comentamos:

- **LSTM**: el espacio de hiperparámetros es el número de neuronas de la capa LSTM donde cuantas más neuronas se tengan más complejo es el modelo. El espacio de búsqueda serán 10 enteros equidistantes en el intervalo [10, 110].

- **SVM:** variamos el hiperparámetro C que nos indica cuánto se permite ignorar el error de clasificación, cuanto más alto menos fallos se permite por lo que baja el error a costa de menor generalización y mayor tiempos de cálculo. Para la búsqueda tomamos 15 puntos equidistantes en el intervalo $[-4, 4]$ como exponentes para 10 (escala logarítmica).
- **Random Forest:** variamos la profundidad máxima de los árboles que creamos. Cuanto menor sea, los árboles son más simples y tienen mayor error pero reducen la varianza y también los tiempos de cómputo. Tomamos el espacio $\{1, 2, \dots, 14\}$.
- **k -NN:** el hiperparámetro a cambiar es el número de vecinos considerado k que en este caso cuanto más alto mas *simple* será pero añadirá más tiempos de cálculo. Tomamos 20 enteros equidistantes en el intervalo $[2, 60]$ (excluimos el 1 ya que siempre sobreajusta).

7.4.1.4. Métricas

Mediremos para cada configuración de cada modelo los siguientes valores:

- PV : con $n = 5$, $r_1 = 0.1$ y $r_n = 0.3$.
- acc_{CV} : con 5 particiones.
- acc_{test}

7.4.2. Resultados

Mostramos una imagen por cada modelo, mostrando a la vez los resultados en los cuatro *datasets* donde se han dibujado las tres curvas correspondientes a cada métrica obtenida.

Los resultados se muestran en orden para LSTM ([Figura 7.12](#)), SVM ([Figura 7.13](#)), Random Forest ([Figura 7.14](#)) y k -NN ([Figura 7.15](#)).

7.4.3. Análisis

Primero analicemos los resultados modelo a modelo. Para LSTM observamos que en los 4 *datasets* para acc_{CV} se obtienen valores muy similares donde no hay ninguno que resalte especialmente, mientras que PV presenta una mayor variación. Además los valores más grandes de PV parecen ir ligados con los más altos de acc_{test} aunque habría que tener cuidado, debido a la posible inestabilidad que habíamos comentado sobre el modelo LSTM.

En SVM, las curvas obtenidas aquí son más *suaves* y vemos como PV capta bien los cambios importantes de acierto: cuando las métricas empiezan a cambiar substancialmente, esto se refleja con un crecimiento en el PV pero a diferencia de las métricas que vuelven a estabilizarse, PV vuelve a decrecer. En todo caso parece que los valores máximos del PV parecen coincidir con las métricas clásicas, pero en cuanto el modelo es muy complejo o simple se castiga con valores bajos.

Para Random Forest pasa algo muy parecido a SVM: en este caso las métricas clásicas van unidas pero no se da ningún valor al hecho de que estamos aumentando innecesariamente la profundidad y por tanto, la complejidad del modelo. PV es capaz él solo de determinar cual es el hiperparámetro óptimo que nos da el mejor rendimiento sin pasarse de complejidad.

En k -NN tenemos unas curvas de PV más *espinosas* pero se repiten los mismos hechos que hemos observado.

7. Experimentación

LSTM

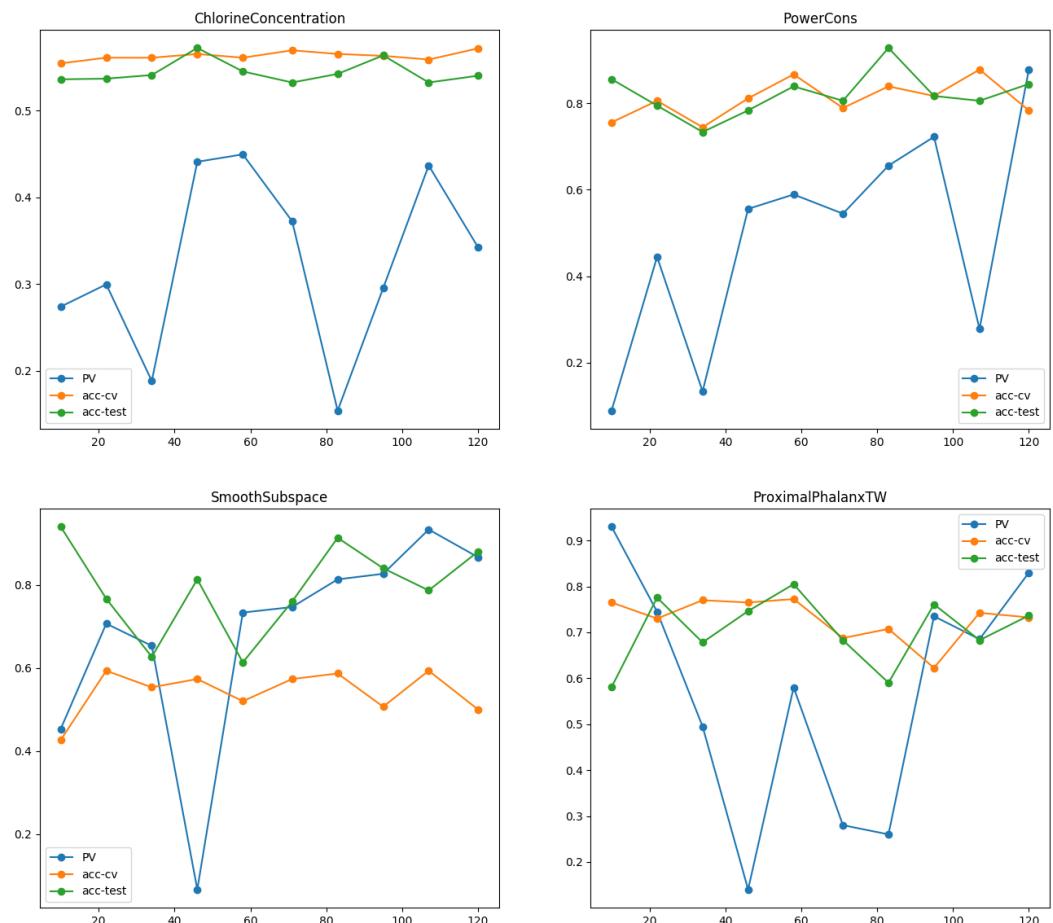


Figura 7.12.: Resultados hiperparámetros LSTM.

SVM

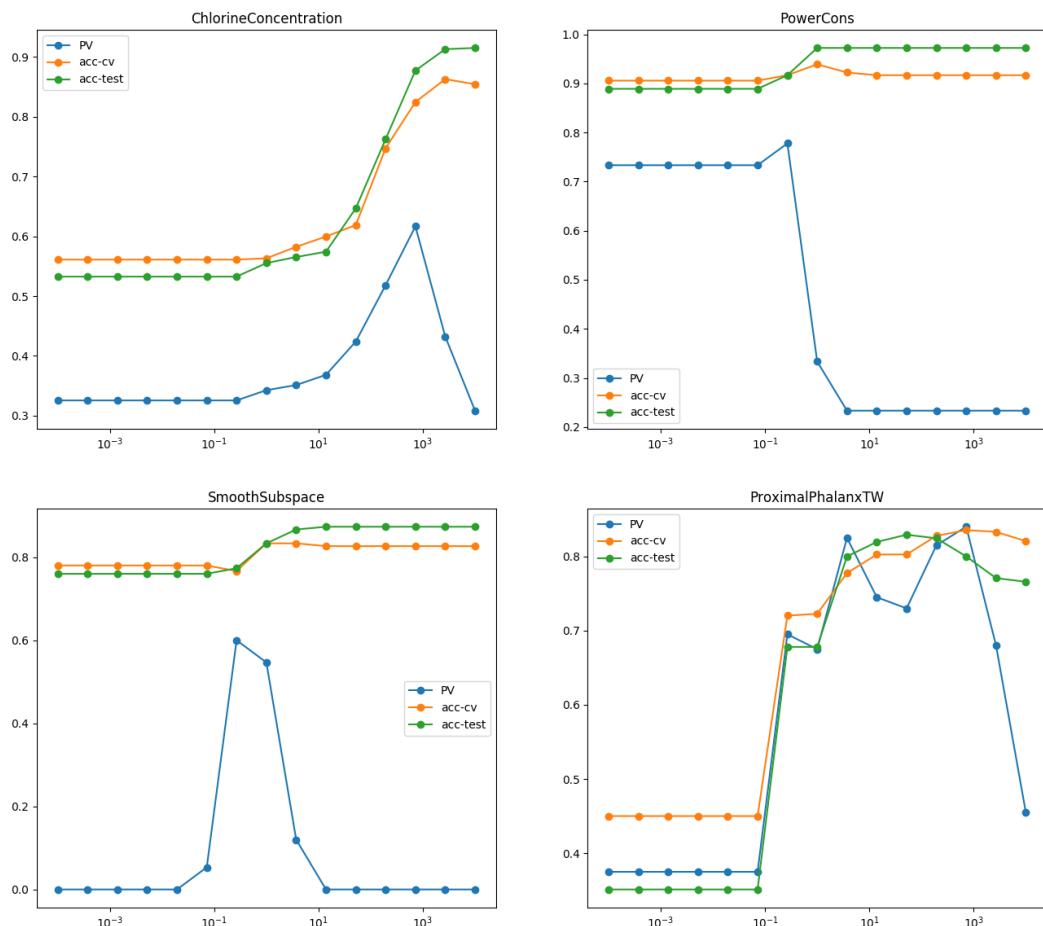


Figura 7.13.: Resultados hiperparámetros SVM.

7. Experimentación

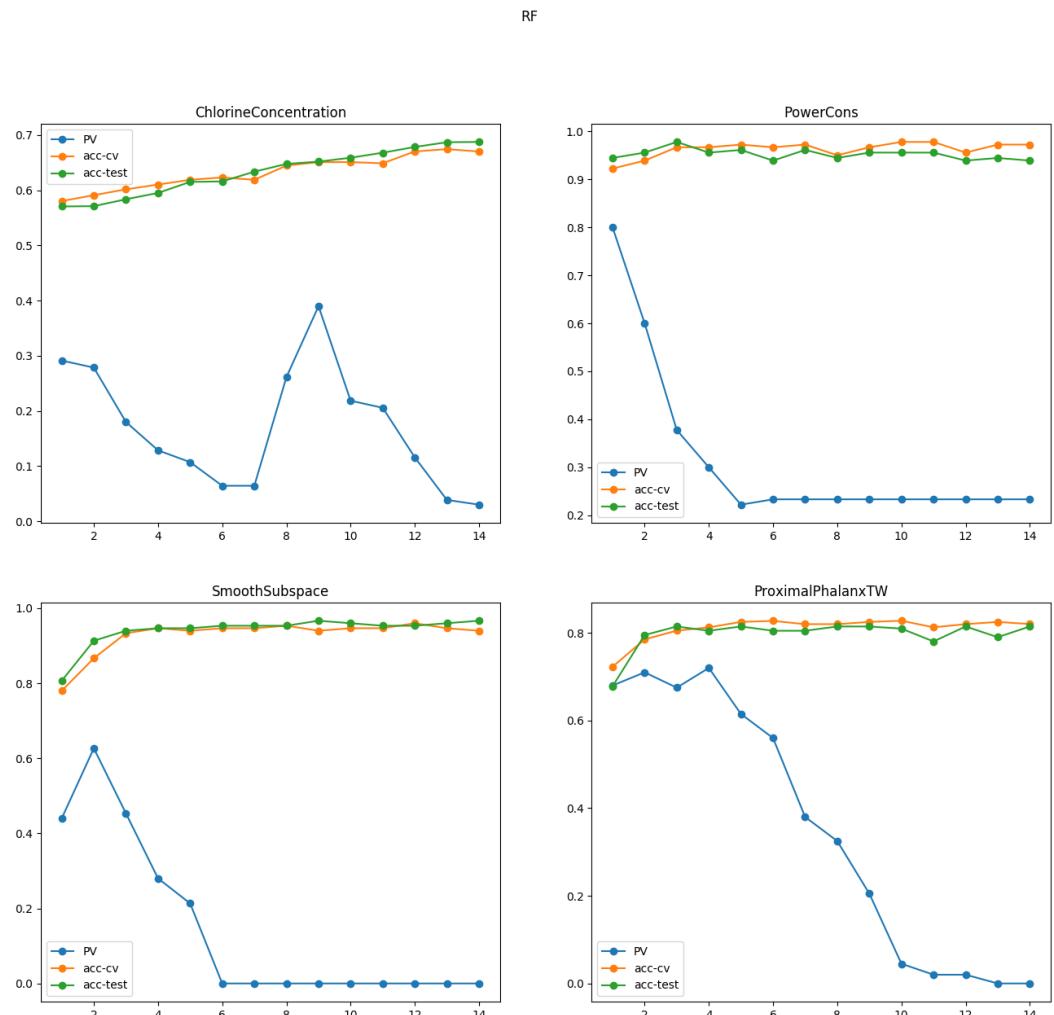


Figura 7.14.: Resultados hiperparámetros Random Forest.

KNN

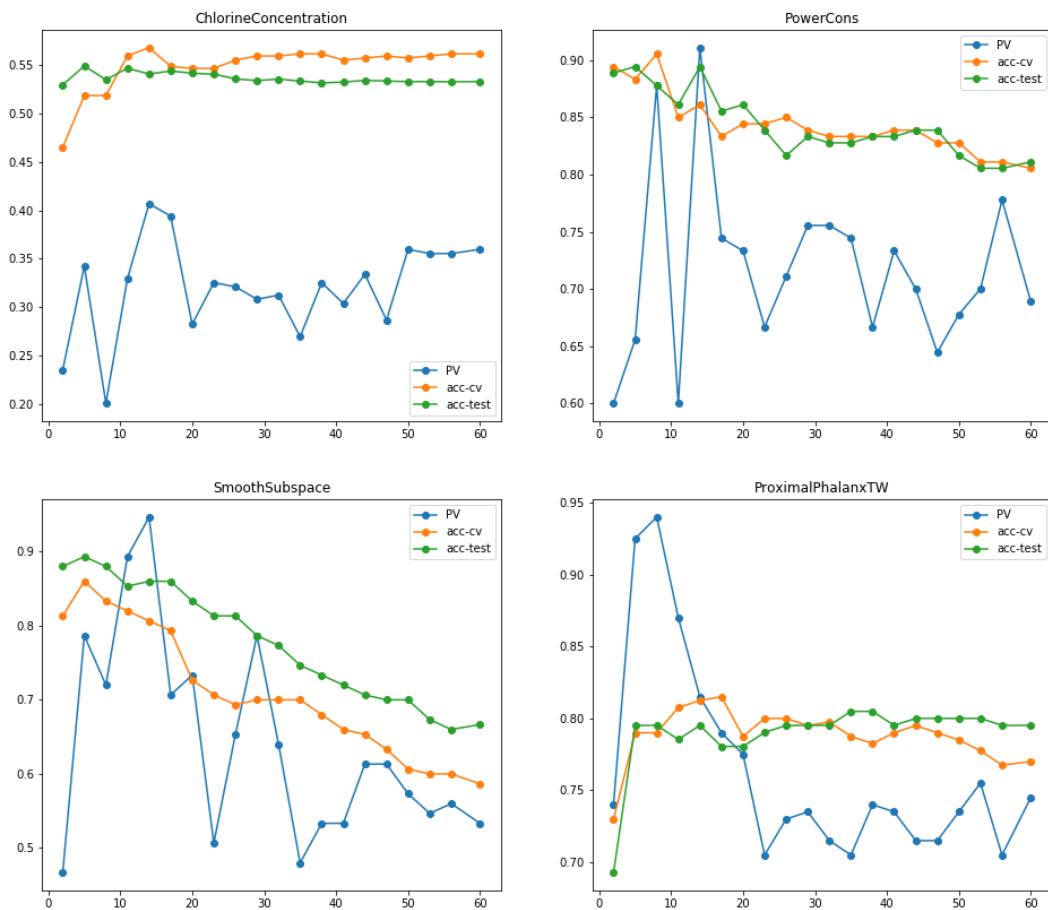


Figura 7.15.: Resultados hiperparámetros k -NN.

7. Experimentación

Recapitulando, vemos que PV es capaz de **discriminar** mejor que acc_{CV} permitiendo considerar no solo el buen ajuste del modelo sino también la complejidad del modelo donde acc_{CV} puede mantener valores iguales. Además los valores máximos de PV se parecen corresponder con los máximos de acc_{test} más o menos y que pudiera reemplazarse directamente por acc_{CV} , pero también se pueden usar **conjuntamente** para tener una mayor seguridad si se quiere obtener el mejor ajuste posible. Finalmente confirmamos la relación del anterior experimento de a mayor PV , mayor acc_{test} .

8. Conclusiones y trabajo futuro

Hemos comprobado que PV puede ser una nueva métrica que puede ayudar en el proceso de selección de modelos como método de validación complementario, ya que nos permite discernir entre los modelos con mayor ajuste estimado el que mejor está adaptándose a la relación subyacente de los datos haciendo que sea el modelo más robusto.

También resalta su eficacia en la búsqueda de hiperparámetros, que capta donde se realizan los cambios importantes en la métrica y permite ser discernir los hiperparámetros al considerar también la complejidad del espacio, cosa que no tiene en cuenta la métrica clásica.

Sin embargo, hay que tener cuidado con modelos que presentan una alta varianza a la hora de aprender las soluciones, como es el caso de las redes neuronales que puede haber problemas de aprendizaje o convergencia. En este caso habría que hacer un diseño y entrenamiento mejor adaptado que una estructura estándar y comprobar que el PV se calcula de una manera correcta.

En un desarrollo posterior sería interesante ver si se pueden mejorar los resultados del PV cambiando los propios parámetros como el número de puntos tomados, o el rango de errores. Otra línea de trabajo sería pensar en alguna forma de extender correctamente este método a los problemas de regresión.

Parte II.

Detección de Anomalías

Modelamos un **detector de anomalías** para series temporales basado en arquitectura LSTM que sea fácilmente desplegable en diversos dominios de los que se quiera reconocer patrones normales y detectar los anómalos. Además proporcionamos **métodos de alteraciones** de series temporales para poder crear muestras con las que poder realizar la validación del modelo, así como también para poder crear *datasets* con datos anómalos que suplan en cierta medida la falta de estos. Usaremos las propias alteraciones para validar nuestro detector y comprobar: el buen rendimiento del modelo y la gran utilidad de las alteraciones para validar.

En [Capítulo 9](#) introducimos los problemas actuales en el dominio de detección de anomalías, donde aportamos un sistema de detección de anomalías LSTM fácil de usar y extender a muchos dominios en [Capítulo 10](#) y también los métodos de alteración de series para transformarlas en anómalas en [Capítulo 11](#). Probamos experimentalmente ambos resultados mediante dos experimentos: uno en un *dataset* real y otro artificial, comentando los resultados y analizándolos en [Capítulo 12](#). Finalmente damos las conclusiones y posible trabajo futuro en [Capítulo 13](#).

9. Introducción

La detección de anomalías en series temporales consiste en el análisis de series que trata de descubrir si hay algo raro en ellas, tanto como si se señalan ciertos puntos o la serie entera. Llamamos a estos datos como **anómalos** ya que se diferencian, ya sea por forma, intensidad o cualquier otro criterio, del patrón regular esperable de las series temporales, a las que llamamos series **normales**.

En la actualidad existen varios problemas fundamentales respecto los problemas de detección de anomalías [AMH16]:

1. Sin modelos **universales**: los modelos realizados suelen ser muy específicos sobre el problema que resuelven, evitando una fácil traslación a un dominio de problema distinto.
2. La falta de **datos** públicos: en la actualidad hay muy pocos *datasets* etiquetados que nos permitan **validar** los modelos para poder medir su eficacia así como para poder usar técnicas de aprendizaje **supervisado**.

Para resolver lo primero, modelamos un **sistema de detección** de series temporales anómalas muy simple basado sobre una arquitectura LSTM que puede desplegarse, a falta de que se ajuste con los parámetros ideales, en cualquier dominio donde las series normales se comportan regularmente mediante un **patrón**. Patrones como estos son esperables en series que se miden cada cierto **periodo** de tiempo como por ejemplo electrocardiogramas, sensores, servidores...

Para el segundo problema proporcionamos una serie de técnicas que nos permiten modificar series ya existentes mediante **perturbaciones**, creando así otras nuevas que pueden servir para proporcionar nuevos *datasets* artificiales con ejemplos anómalos que permitan validar los modelos de detección. Aunque sean artificiales intentan suplir esta **falta de datos** para poder al menos dar una idea de cómo se comportan los modelos ante perturbaciones de las series, que están parametrizadas para ver el rendimiento a distintas escalas de valores.

Analizaremos **experimentalmente** con ejemplos como aprende y se comporta el sistema, validándolo usando los métodos comentados para alterar las series. **Analizaremos la eficacia** del sistema frente a **distintos tipos e intensidades** de alteraciones, observando la eficacia del detector y de las perturbaciones.

10. Sistema LSTM para detección de series anómalas

Supongamos que las series vienen de una distribución desconocida \mathcal{P} que podemos dividir en dos tipos: las normales y las anómalas. Las normales son las señales que podemos esperar con una **probabilidad muy alta**, mientras que las anómalas son las que su **frecuencia es inusual**.

El sistema que diseñamos intenta aprender la distribución de estas señales normales, su patrón, de manera que una señal será anómala si no se parece a lo que ha aprendido el sistema. Así diferenciamos dos partes fundamentales en nuestro modelo:

1. **Autoencoder LSTM** que aprende este patrón mediante reconstrucción de la entrada.
2. **Detector de anomalías** en base a la distribución de los errores entre las series normales y sus reconstrucciones.

10.1. Autoencoder LSTM

El *autoencoder LSTM* se encarga de aprender mediante aprendizaje **no supervisado** cómo devolver la misma señal que se proporciona como entrada, forzando al sistema a que aprenda el patrón subyacente de los datos.

Para la arquitectura de este modelo usaremos una estructura *autoencoder*, sin la subdimensión codificada ya que no necesitamos esta representación, que está formada por dos partes: el **codificador** (*encoder*) y el **descodificador** (*decoder*). El *encoder* obtiene de la serie características mediante capas LSTM, cuyo tamaño va decreciendo hasta llegar al subespacio de codificación de la series, que saltaremos hasta pasar directamente al *decoder* que se encarga de usar capas LSTM con tamaño creciente hasta llegar al tamaño original de las series mediante una capa completamente conectada (*Dense*) que se encargará de reproducir los valores de la serie. Además todas las capas LSTM tienen como función no lineal de activación *ReLU*.

Por ejemplo para un *dataset* cuyas series tienen longitud 82 y solo se tiene una característica esta sería la arquitectura [Figura 10.1](#). Si tuviese otra longitud o número de características lo único que cambiaría sería la dimensión de la entrada y la salida, ajustándose a la del *dataset* concreto.

Además se pueden modificar ciertos hiperparámetros como el número de neuronas de las capas LSTM, el parámetro de regularización de las capas, la tasa de aprendizaje, el número de épocas, el tamaño de los *batches*. El método para actualizar la red es el usual de **Adam**, usando como **función de pérdida** el Error Cuadrático Medio (*Mean Squared Error* [\(4.7\)](#)) que es el usual para problemas de regresión y que se adecúa a nuestro problema ya que estamos prediciendo valores continuos (los de la serie) y queremos que lo minimice para que reconstruya la entrada en la salida.

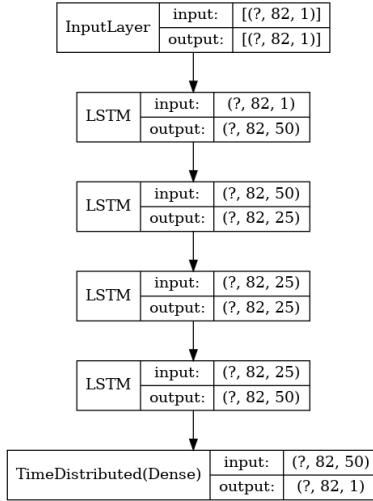


Figura 10.1.: Arquitectura del modelo *autoencoder*.

10.2. Detección

La detección está basada en la **distribución de los errores** (por ejemplo, el error cuadrático medio) de distribución obtenidos en el conjunto de entrenamiento de las señales normales con las reconstrucciones del modelo LSTM. Podemos clasificar una serie **anómala** cuando el error de reconstrucción supere una cierto **umbral** fijado por nosotros. Para ser más flexibles y evitar ser tan **discriminativos**, devolveremos **probabilidades** para dar una idea más clara de cuan grave puede ser un error.

Estas probabilidades estarán basadas en la **función de probabilidad** de la distribución de los errores, de manera que para errores poco frecuentes o mucho más grandes de los vistos, tendremos una probabilidad asignada cercana a 100 % de ser anómala. Obviamente para muchas series normales se tendrán probabilidades altas, de ahí que se quiera tomar un **umbral** alto para asegurarnos que lo que se detecte sea realmente una anomalía.

La función de probabilidad como ya sabemos se puede obtener integrando la función de densidad, que **estimaremos** con una cantidad razonable de datos con cualquiera de los métodos ya existentes. Por lo que esta parte fundamentalmente se encargará de **aprender una función de densidad** basada en los errores de reconstrucción del *dataset* de entrenamiento.

Cabe añadir que estamos partiendo del hecho de que solo tenemos series normales, **descubriremos** tanto la forma como la frecuencia de las anomalías que suele ser las circunstancias en muchos problemas reales, por lo que la probabilidad que devolvemos no es más que una **estimación**. Conforme se obtengan nuevas muestras se puede ir **mejorando el sistema**, incorporando la distribución de las series anómalas, ajustando así la probabilidad para que sea mucho más exacta.

Finalmente, si observamos que los errores suelen ser mucho más grandes que los de entrenamiento se podría considerar usar un umbral fijo, aunque en nuestro caso **alteraremos muy poco** las series por lo que es esperable que muchos errores puedan ser muy sutiles y verse casi como series normales, a lo que nos interesa que el intervalo de probabilidades se mueva dentro de los errores de las series normales.

11. Alteraciones

Proporcionaremos cuatro tipos distintos de alteraciones para series temporales: dos basados en **ruido gaussiano** y los otros dos en la **descomposición STD**. Estas alteraciones intentan ser lo más **genéricas** posibles para poder ser aplicadas en un **alto rango** de dominios de problemas y lo más **naturales** posibles, intentando imitar las anomalías reales que se dan.

Si consideramos que a veces las anomalías como tales están consideradas así bajo un **criterio concreto** de un dominio específico del problema es muy difícil intentar tomarlas todas en cuenta; sin embargo sí hay generalmente unas pocas anomalías típicas que se suele querer detectar:

1. **Pico**: se dice que hay un *pico* en la serie cuando en un periodo muy pequeño de tiempo toma un valor muy alto o muy bajo de lo usual en comparación en un entorno local.
2. **Cambio estacional**: cuando en una serie donde se ve un patrón estacional (cada cierto periodo se repite una estructura) en uno de los periodos la intensidad de la señal cambia, aumentando o decrementándose.
3. **Fluctuación**: cuando en la señal tiene un cambio de forma (fluctúa) en un periodo corto de tiempo, aunque no tiene por qué variar mucho en cuanto a la intensidad.

Intentaremos reproducir estas anomalías mediante métodos que produzcan alteraciones **parecidas**, siendo además **regulables** a gusto del usuario para adaptarse a lo que se necesite.

Además, para todos las alteraciones es necesario un algoritmo que nos **seleccione un tramo de la serie** para poder aplicar la alteración. Implementamos **Algoritmo 6** que al pasarle una serie X nos devuelve un trozo de ésta determinada por el punto donde empieza y su longitud. El algoritmo es muy flexible puesto que podemos indicar una longitud máxima y/o mínima y excluyendo los bordes hasta cierto punto. En líneas generales el algoritmo escoge el tramo aleatoriamente bajo los criterios impuestos, y siempre que sea posible según la longitud de la serie.

Algoritmo 6: random_slice(*X, max_length, min_length, border*)

```
// Obtenemos la longitud máxima posible
1 max_length ← min(max_length, X.length – 2 · border);
// Escogemos un tamaño aleatorio
2 length ← round(distribucion_uniforme(min_length, max_length));
// Ajustamos la posición máxima posible
3 max_pos ← X.length – max(border, length);
// Tomamos la posición aleatoriamente
4 pos ← round(distribucion_uniforme(border, max_pos));
Resultado: pos, length
```

La posible aleatoriedad que introducimos está hecha para poder generar **muchísimas alteraciones** de golpe que sean distintas entre sí automáticamente, generando efectivamente muchas muestras anómalas para validación.

11.1. Ruido gaussiano

La distribución **normal** o **gaussiana** nos es muy útil para intentar simular un *pico* ya que su forma de campana se asemeja mucho a un cambio de intensidad muy alto y es además de forma continua; sin más que muestrear la función de densidad en el intervalo $[-3\sigma, 3\sigma]$ podemos obtener el 99.74 % de la distribución.

Para hacer que el muestreo sea significativo, esto es, que la mayoría de puntos tomados tengan un valor no nulo, tomamos $\sigma = \text{longitud}/6$. Finalmente se multiplica todo otra vez por un parámetro σ que se le pasa al algoritmo para controlar el grado de **intensidad** de la alteración.

Si queremos picos muy **pronunciados** solo tendremos que tomar una longitud de alteración pequeña y un σ alto. Para otros *picos* más suaves aumentamos un poco la longitud y disminuimos el σ . En cualquier caso, independientemente de la longitud total de la serie, la longitud de la perturbación no debería ser muy grande puesto que estaríamos afectando a un porcentaje mayor de la serie (ya no sería una anomalía puntual) y la perturbación se aplana ya que la desviación de la gaussiana se vuelve mayor.

Podemos ver distintos muestreos con varios tamaños y σ en [Figura 11.1](#), observándose que efectivamente se tiene un muestreo correcto, con intensidad moldeable por σ y que según la longitud deseada se puede hacer un cambio más brusco o suave.

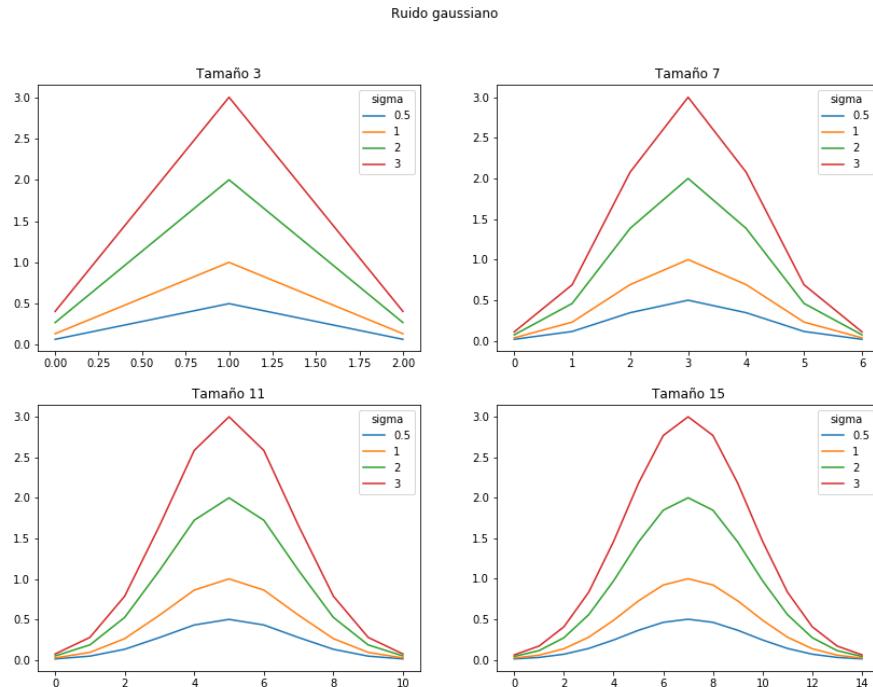


Figura 11.1.: Muestreos de la distribución gaussiana con distintos tamaños y σ .

Finalmente, la serie alterada que se devuelve es la original X sumando al tramo seleccionado anteriormente el muestreo que hemos obtenido. Se ha incluido también una opción para

que aleatoriamente se reste en vez de sumar, para crear un pico invertido.

Tomando el *dataset Beef* de [ABK2o], representamos varios ejemplos de las series con las perturbaciones aplicadas, escogiendo longitudes aleatorias en $[3, 40]$ y σ en $[0.7, 3]$ en Figura 11.2. Vemos como tenemos un amplio rango de distintas perturbaciones, con picos muy empinados o más pequeños y otras más suaves.

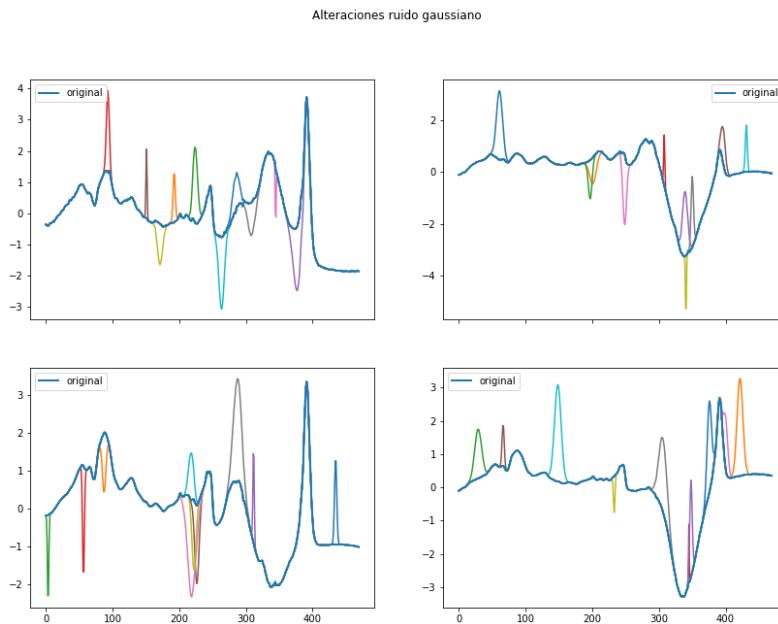


Figura 11.2.: Perturbaciones con ruido gaussiano con distintas longitudes y σ en el dataset *Beef*.

11.2. Pulso gaussiano-sinusoidal

Introducimos la alteración de **pulso gaussiano-sinusoidal** como alteración de la forma de la señal, imitando una **fluctuación** en el tramo de la señal. Este pulso no es nada más que un pulso electromagnético con forma sinusoidal que es amortiguado con una gaussiana, es decir, una señal sinusoidal con amplitud creciente desde los extremos hasta el centro de la señal.

Igual que hacíamos antes, se toma el muestreo y se multiplica por el σ aplicado, sumando esta alteración al tramo elegido aleatoriamente.

Unos cuantos ejemplos de pulsos gaussianos-sinusoidal los tenemos en Figura 11.3, que indican que por lo menos necesitamos un tamaño considerable para que el muestreo sea significativo (que se muestre el carácter sinusoidal) y en cualquier caso podemos modificar la intensidad con σ .

Finalmente en el *dataset ECG5000* de [ABK2o], repetimos ejemplificando las perturbaciones tomando longitudes en el intervalo $[7, 11]$ y con σ en $[0.5, 1.5]$ en Figura 11.4. Cambiamos

11. Alteraciones

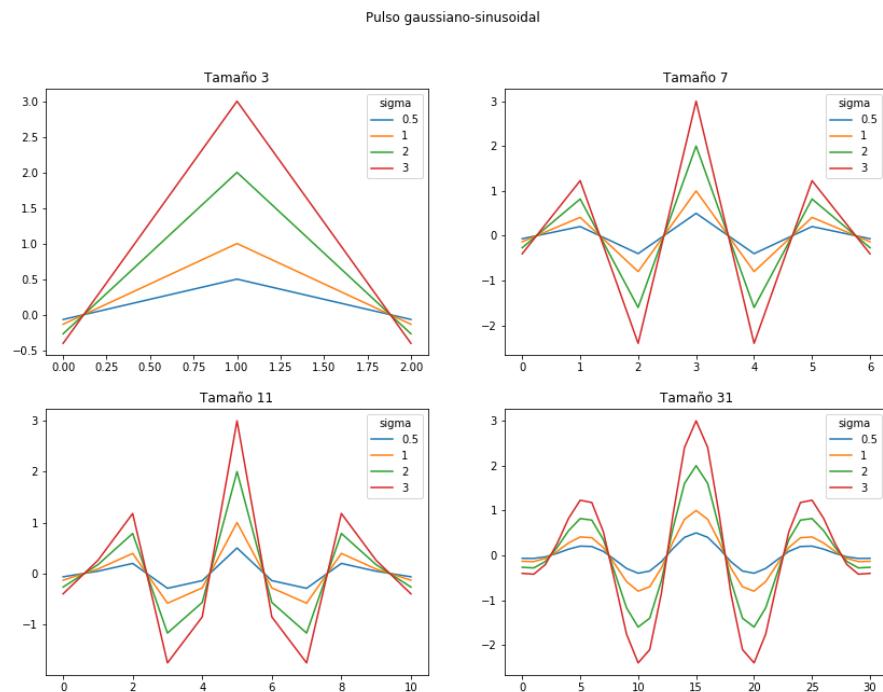


Figura 11.3.: Ejemplos de pulsos gaussianos-sinusoidales con distintos tamaños y σ .

efectivamente la forma de la señal introduciendo pequeñas oscilaciones de los valores, que pueden ser más o menos bruscas según se necesite.

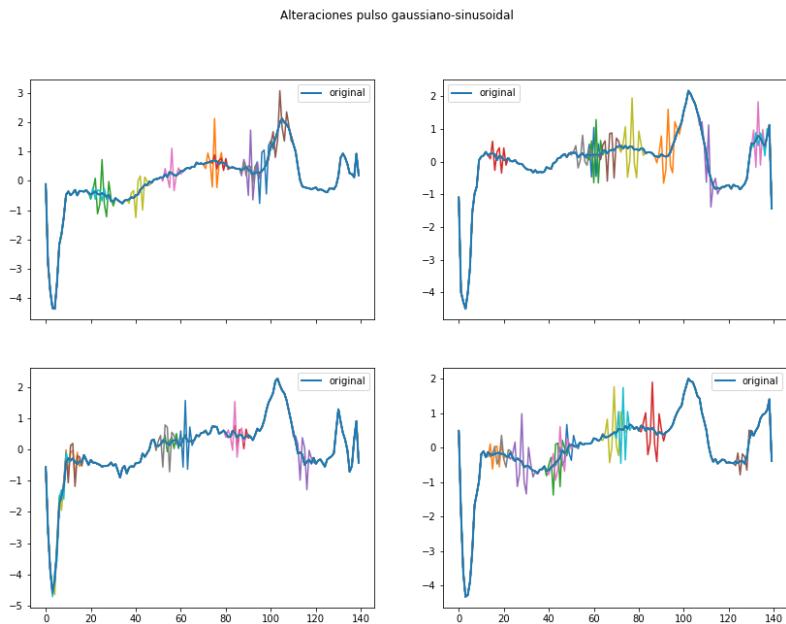


Figura 11.4.: Perturbaciones con pulso gaussiano-sinusoidal con distintas longitudes y σ en el dataset *ECG5000*.

11.3. Modificación STL

Para intentar emular el **cambio estacional** utilizaremos la técnica de descomposición STL y modificaremos las series de **tendencia** y **estacionalidad**.

Estas técnicas están dirigidas especialmente cuando el *dataset* muestra cierta tendencia estacional para poder **descomponerla** correctamente, considerando que los métodos necesitan el **periodo** de esta estacionalidad que tendremos que estimar más o menos para poder realizar las modificaciones bien.

Para poder hacer perturbaciones de este estilo que se apliquen correctamente hemos creado un *dataset* artificial formado por muestras de 100 valores de la función **coseno**, que aporta la componente **estacional**. Después se ha hecho un muestreo del intervalo $[0, 1]$ con 100 puntos equidistantes, sumándolo a la serie para darle la **tendencia**. A continuación se ha añadido ruido gaussiano aleatorio para darle ruido y que las muestras sean diferentes; y finalmente se han estandarizado.

Visualizamos unas 50 muestras Figura 11.5 y comprobamos que efectivamente que tenemos la función coseno con una tendencia al alza y con cierto ruido para cada serie.

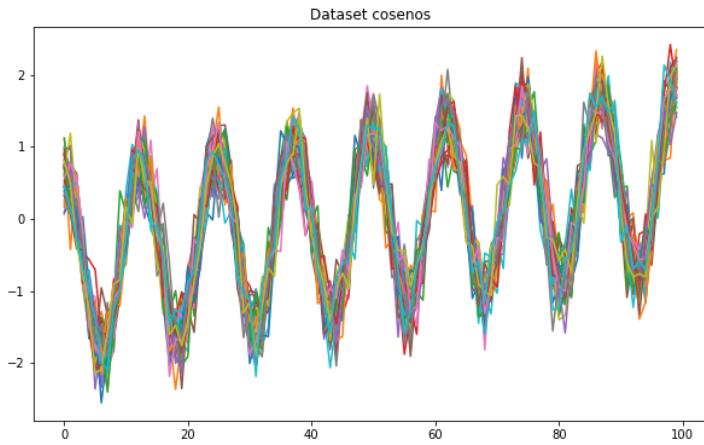


Figura 11.5.: Dataset de cosenos.

11.3.1. Estacionalidad

Esta modificación consiste simplemente en obtener la parte de estacionalidad de la descomposición STL de la serie y en amplificarla o disminuirla multiplicando por el parámetro σ , haciendo un cambio en la serie pero **sin cambiar su forma**.

Ejemplificamos lo que estamos haciendo en [Figura 11.6](#), probando con distintos períodos y σ . Notamos como si se toma un valor de periodo incorrecto se pueden obtener resultados no deseados, en este caso el periodo que parece que mejor encuentra la descomposición es 13. Por otro lado vemos como conseguimos lo que queremos, teniendo en cuenta que la serie original está con $\sigma = 1$ amplificamos cuando $\sigma > 1$ y amortiguamos con $\sigma < 1$.

Para ver el efecto en las series completas, cogemos el periodo de 13 que hemos visto que recoge bien la descomposición y variamos la longitud de las perturbaciones entre [3, 13] y σ entre [0, 2.5] para que pueda amplificar o disminuir la señal, cuyos resultados están en [Figura 11.7](#)

En este caso vemos que son muy variadas las perturbaciones pero que generalmente cumplen su propósito al aumentar o decrementar la serie en ciertos momentos sin alterar la forma de las señales.

11.3.2. Tendencia

Procedemos igual que la estacionalidad, pero en este caso cambiamos la tendencia. Hacemos la descomposición STL de la serie y aumentamos o amortiguamos multiplicando por un parámetro σ que cambia la serie sin modificar su forma.

Algunos ejemplos los tenemos en [Figura 11.8](#) que volvemos a probar con varios períodos y σ . Se confirma que el periodo 13 es el que mejor se ajusta a la descomposición, y que la tendencia aumenta o disminuye según σ considerando que el caso original es $\sigma = 1$.

Finalmente vemos los efectos al alterar la tendencia en [Figura 11.9](#) cambiando la longitud en [3, 13] y σ en [0, 5] (hemos aumentado el sigma para que se notasen mejor los efectos).

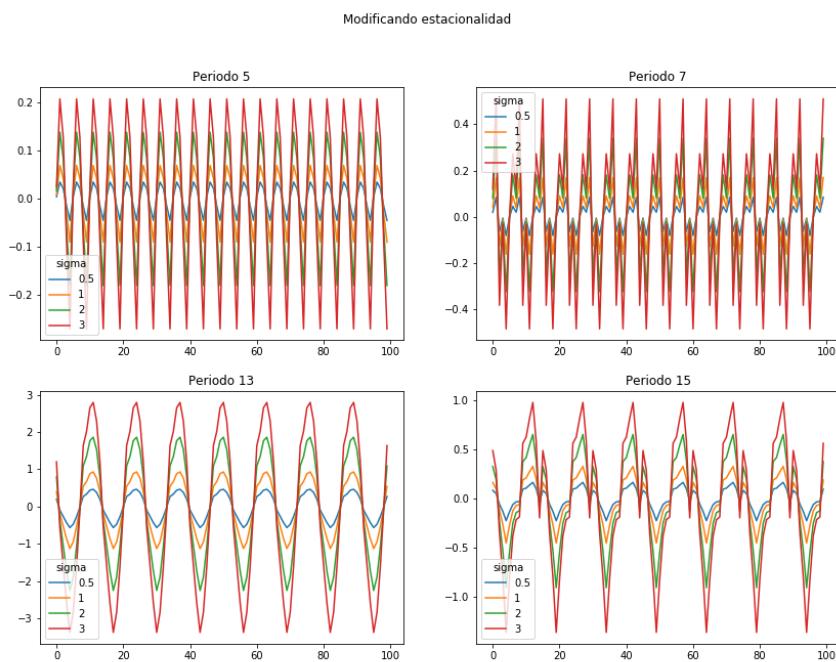


Figura 11.6.: Modificación de la estacionalidad de cosenos.

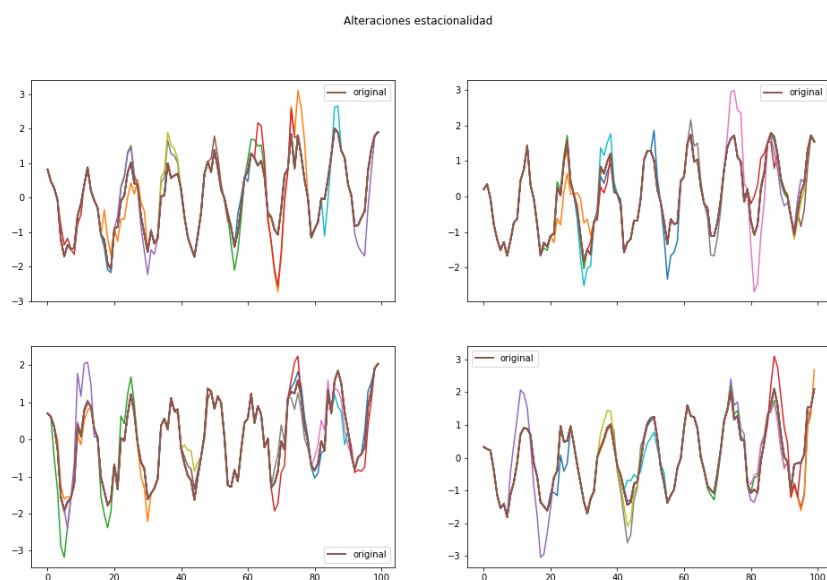


Figura 11.7.: Modificación de estacionalidad con distintas longitudes y σ .

11. Alteraciones

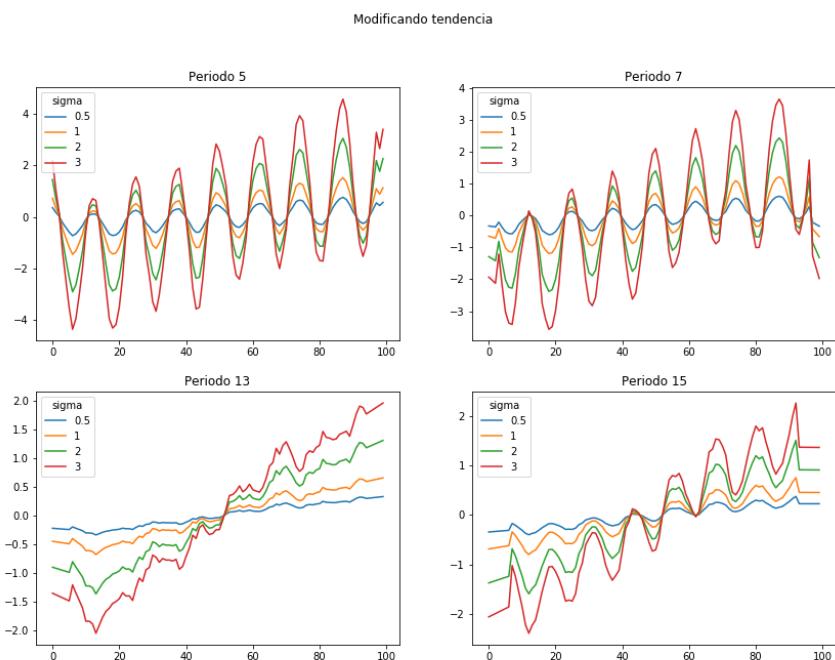


Figura 11.8.: Modificación de la tendencia de cosenos.

Notamos que la tendencia se suaviza o potencia conforme σ , por lo que hemos conseguido los efectos deseados.

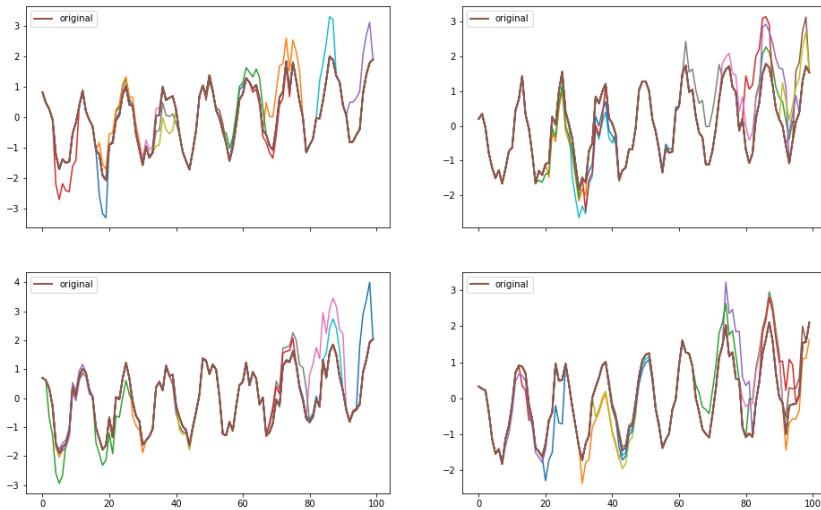


Figura 11.9.: Modificación de tendencia con distintas longitudes y σ .

Cabe añadir que se consigue un efecto parecido tanto en **tendencia** como en **estacionalidad** al estar cambiando la intensidad de una parte de la serie sin cambiar su forma, pero los cambios son un poco distintos: con la estacionalidad se puede cambiar las partes más grandes a más, y las más pequeñas a menos; con la tendencia cambia todo a más o a menos independientemente, solo importa la parte de la tendencia.

12. Experimentación

Comprobaremos **empíricamente** el sistema de detección y los métodos de perturbación introducidos mediante la experimentación en un *dataset* real para probar los ruidos gaussianos, y un *dataset* sintético con tendencia y estacionalidad clara para probar los basados en la descomposición STL.

12.1. Dataset real

12.1.1. Descripción

Escogemos el *dataset* *TwoLeadECG* de [ABK20] que originalmente viene de un problema de clasificación de varias señales de electrocardiogramas. Mezclamos la partición existente, hacemos nosotros una división 80/20 % y nos quedamos solo con la clase o que consideramos nuestra clase normal; el resto se consideran series anómalas que también podremos usar para validar el modelo.

Inspeccionamos 10 series del *dataset* para ver si ciertamente hay algún patrón normal que muestren las series en [Figura 12.1](#). Efectivamente hay una forma clara que nos muestra que es posible que nuestro modelo pueda aprenderla.

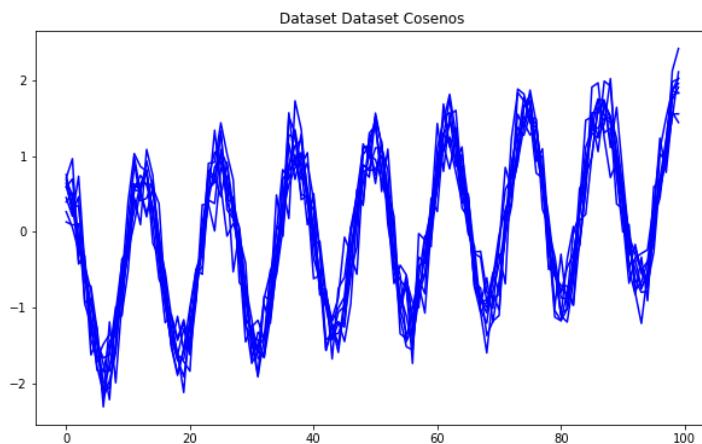


Figura 12.1.: Algunas series de *TwoLeadECG*.

12.1.2. Entrenamiento

Usando la arquitectura comentada previamente entrenamos el modelo durante 300 épocas que después de varias pruebas comprobamos que una buena configuración nos resulta con

12. Experimentación

un número de neuronas a [50, 25, 25, 50] (una cantidad bastante baja y razonable en cuestión de tiempo de entrenamiento), sin regularización (provoca que aumente mucho la función de pérdida hasta desbordarse), con una tasa de aprendizaje de 0.001 (para evitar ciertos problemas de convergencia encontrados), tamaño de *batch* de 128 y con un conjunto de validación usando el 10 %.

Mostramos el resultado del entrenamiento de la función de pérdida MSE en [Figura 12.2](#). Realiza un buen entrenamiento sin sobreajuste (Val y Train están juntas), donde converge rápidamente y hemos ido siguiendo entrenando para que se ajustase lo mejor posible al patrón de las series.

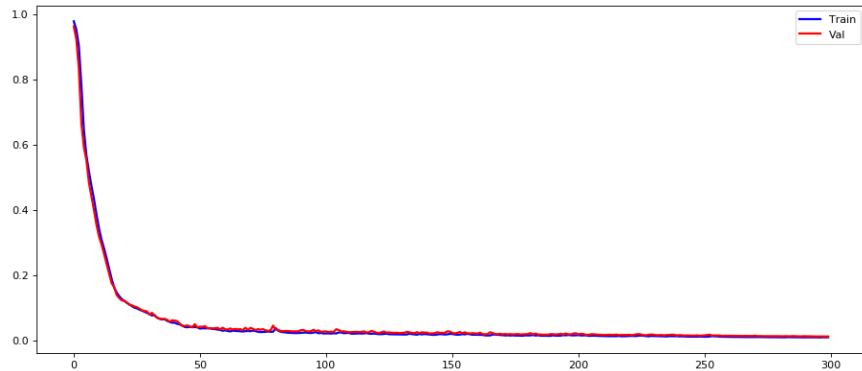


Figura 12.2.: Entrenamiento del modelo LSTM.

Para validar los resultados veamos algunas series tanto en *train* como en *test* junto con sus reconstrucciones para ver que tal se comporta el modelo. Mostramos cuatro reconstrucciones de *train* en [Figura 12.3](#) y cuatro de *test* en [Figura 12.4](#).

En general vemos que las reconstrucciones son casi idénticas, habiendo un poco de diferencia en el primer tramo de las series, que si bien no se ajusta exactamente está bastante cerca. Después de varias configuraciones no se ha conseguido perfilar totalmente, por lo que el resultado obtenido nos parece correcto.

Ahora pasamos a estudiar la distribución de errores MSE entre las series y sus reconstrucciones con el modelo, tanto en *train* como en *test* ([Figura 12.5](#)). A partir de estos datos usamos una estimación de la función de densidad que representamos también.

Observamos que los errores se distribuyen parecido a una distribución gaussiana (esperable del ruido blanco) aunque sí que hay un pequeño aumento a la izquierda, probablemente debido al ajuste que se da en el primer tramo que habíamos visto. En cualquier caso, los errores son bastante pequeños y parecen que se distribuyen igual tanto en *train* como en *test* (funciones de densidad estimada parecidas), por lo que el ajuste parece indicarnos que es bueno.

Por lo tanto ya tenemos el aprendizaje del modelo: el *autoencoder* LSTM que ha aprendido a reconstruir las series y el detector que se encarga de asignar probabilidades a las series en función del error de reconstrucción usando la función de densidad estimada en el conjunto *train*.

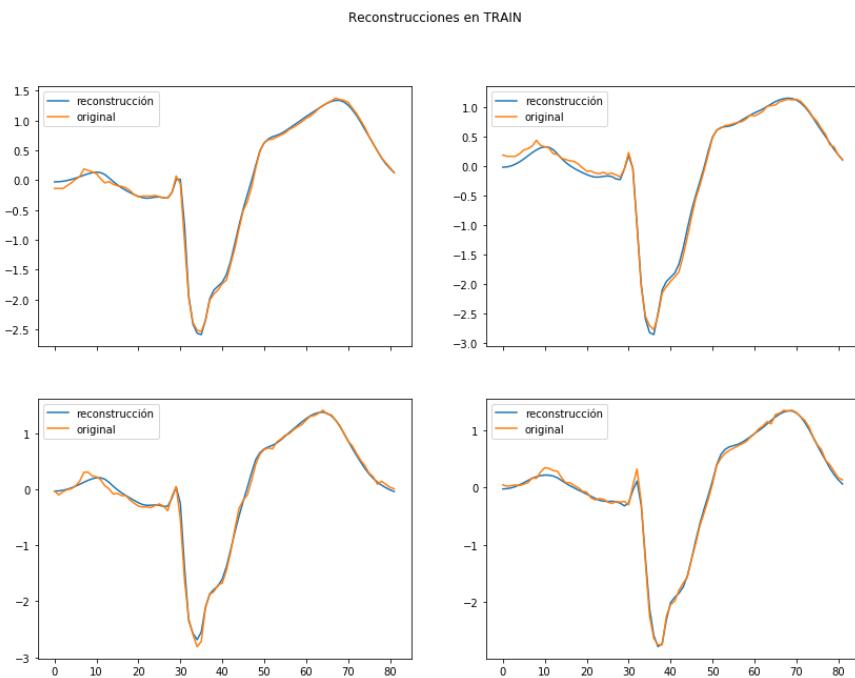


Figura 12.3.: Reconstrucciones en *train*.

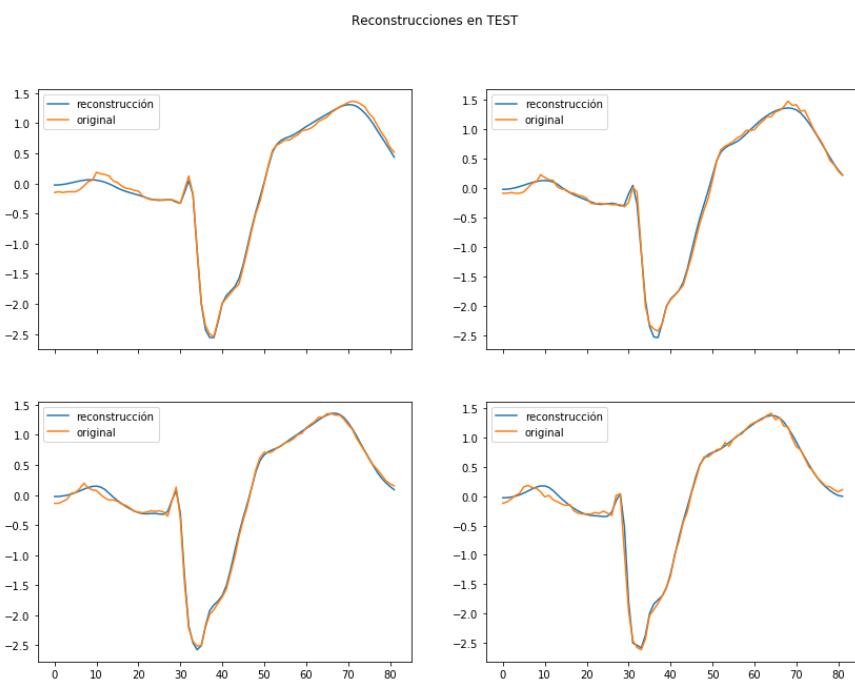


Figura 12.4.: Reconstrucciones en *test*.

12. Experimentación

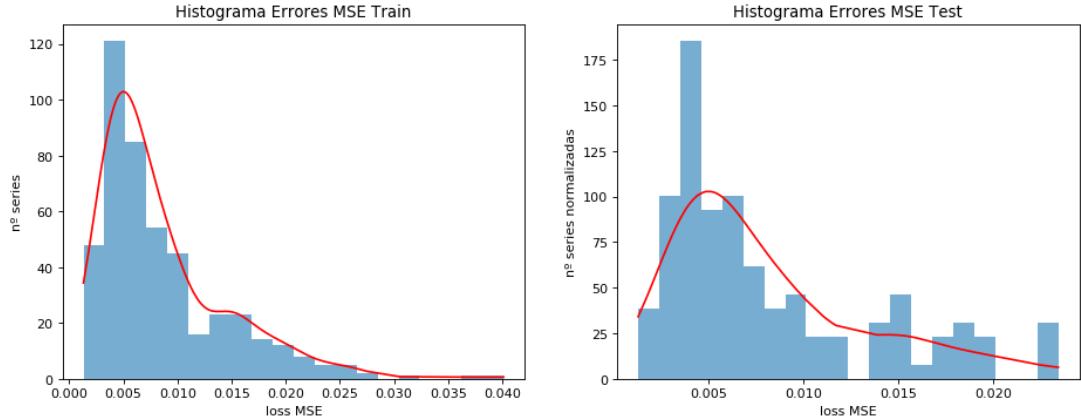


Figura 12.5.: Histogramas de error en *train* y *test*.

12.1.3. Validación

Para la validación usaremos primero las clases anómalas para ver si consigue detectar correctamente otras señales (que podría ser un ejemplo real para detectar anomalías en electrocardiogramas de personas) y después usaremos nuestros métodos para crear anomalías nuevas.

La métrica para validar nuestros resultados será la *PR-AUC* o *PR* ya que el modelo calcula probabilidades y consideramos que la clase positiva (detectar como anomalía) es más importante que la clase negativa (que no lo sea), ya que generalmente es preferible detectar los errores aunque sea una falso positivo por el mayor coste asociado a los falsos negativos.

12.1.3.1. Clases anómalas

Mostramos la curva de sensibilidad-precisión del detector junto a la curva del clasificador aleatorio (*no-skill*) proporcional a las clases que se usa como clasificador base para comparar. Los resultados de *train* y en *test* los tenemos en Figura 12.6, que obtiene un valor de *PR* = 0.926 y *PR* = 0.939 respectivamente.

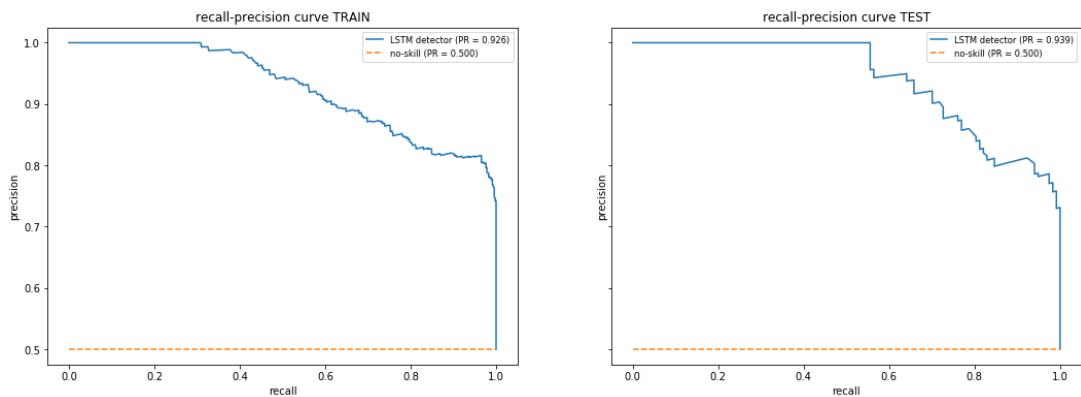


Figura 12.6.: Curvas sensibilidad-precisión en *train* y *test*.

Vemos por los resultados de la métrica y comparando con el clasificador base que nuestro detector se comporta bastante bien, obteniendo una puntuación bastante elevada. Observamos que podemos obtener una sensibilidad muy alta a costa de perder muy poca precisión, en cualquier caso el umbral usado dependería del problema real concreto.

12.1.3.2. Alteraciones

Procedemos a crear muestras anómalas con nuestros métodos, aunque como hemos visto en el patrón de la serie no parece que haya una componente cíclica clara, por lo que no usaremos los métodos de descomposición STL.

Empezaremos por el **ruido gaussiano**, donde aplicamos perturbaciones con tamaños entre $[3, 11]$ y σ en $\{0.75, 0.9, 1, 1.25\}$. Tanto en tamaño como en σ tenemos unos valores bastante bajos para que las alteraciones sean muy leves y poner a prueba el modelo.

Mostramos 4 ejemplos para cada σ en Figura 12.7, Figura 12.8, Figura 12.9 y Figura 12.9.

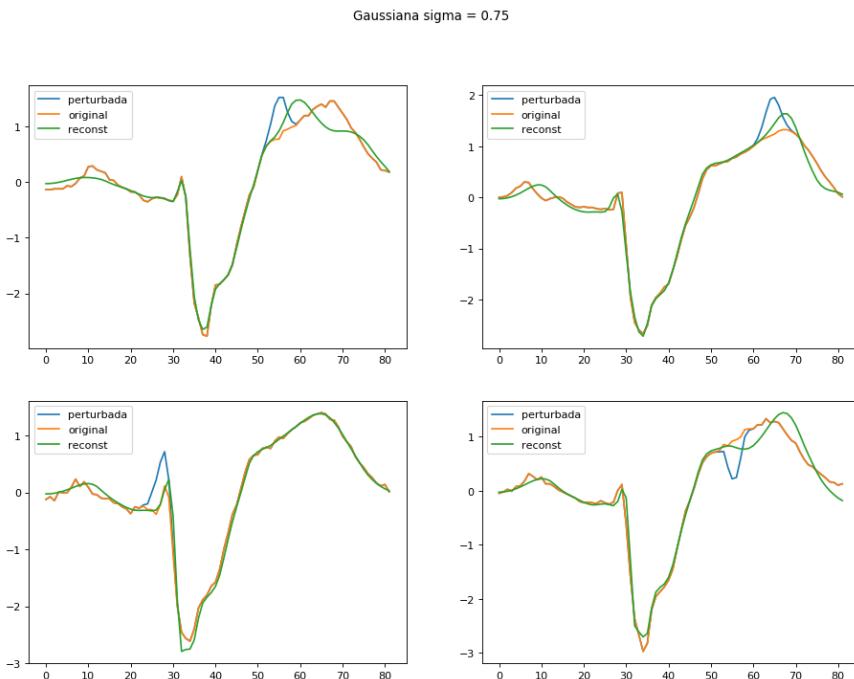


Figura 12.7.: Alteraciones con ruido gaussiano a $\sigma = 0.75$.

Observamos que generalmente las reconstrucciones son buenas hasta que llega al tramo de la perturbación, a partir de ahí la señal construida empieza a comportarse de manera muy distinta a la señal original añadiendo más error que permite identificar a la señal como anómala.

También notamos que la mayoría de perturbaciones no son especialmente bruscas, incluso según con la posición de la alteración, a veces no se diferencia casi nada de la serie original.

Ahora pongamos a validar el modelo con la métrica PR , para ello crearemos tanto para *train* como para *test* una parte de anomalías con tamaño proporcional a cada uno, respectivamente. Tomaremos los ratios $\{0.05, 0.1, 0.2, 0.3\}$ para ver cuál es el efecto con muy pocas muestras y aumentando hasta que el tamaño de anomalías sea representativo de las posibles

12. Experimentación

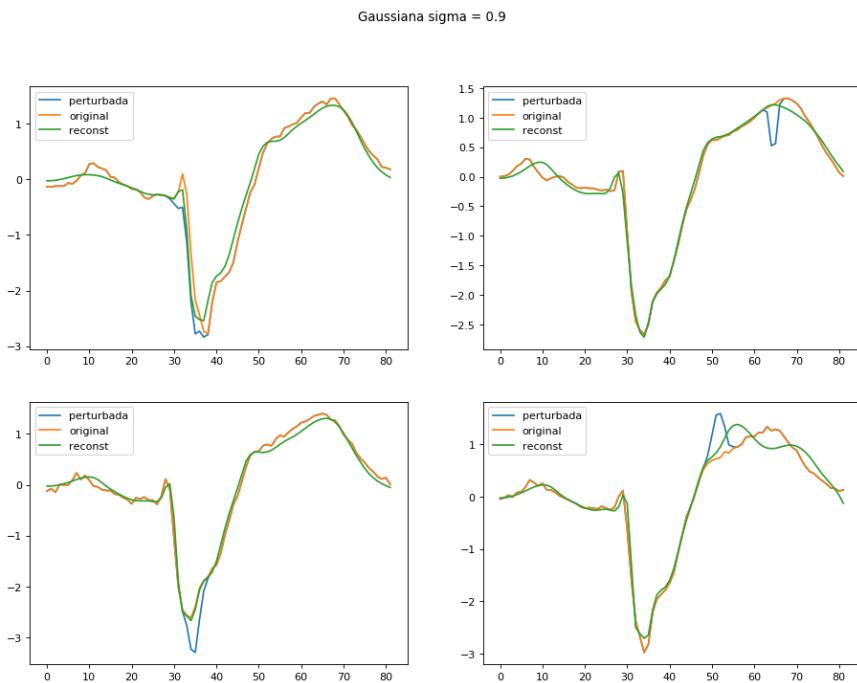


Figura 12.8.: Alteraciones con ruido gaussiano a $\sigma = 0.9$.

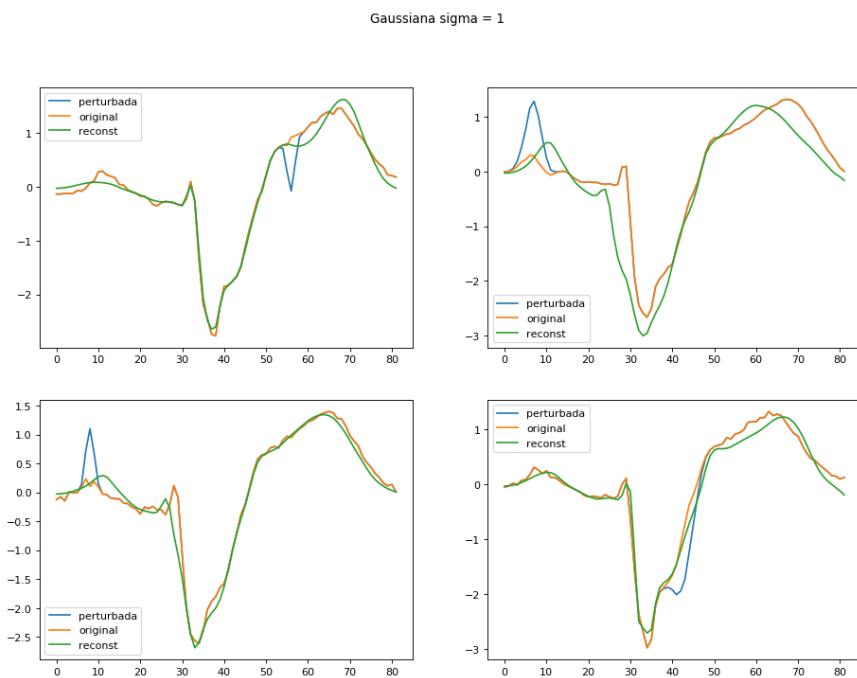


Figura 12.9.: Alteraciones con ruido gaussiano a $\sigma = 1$.

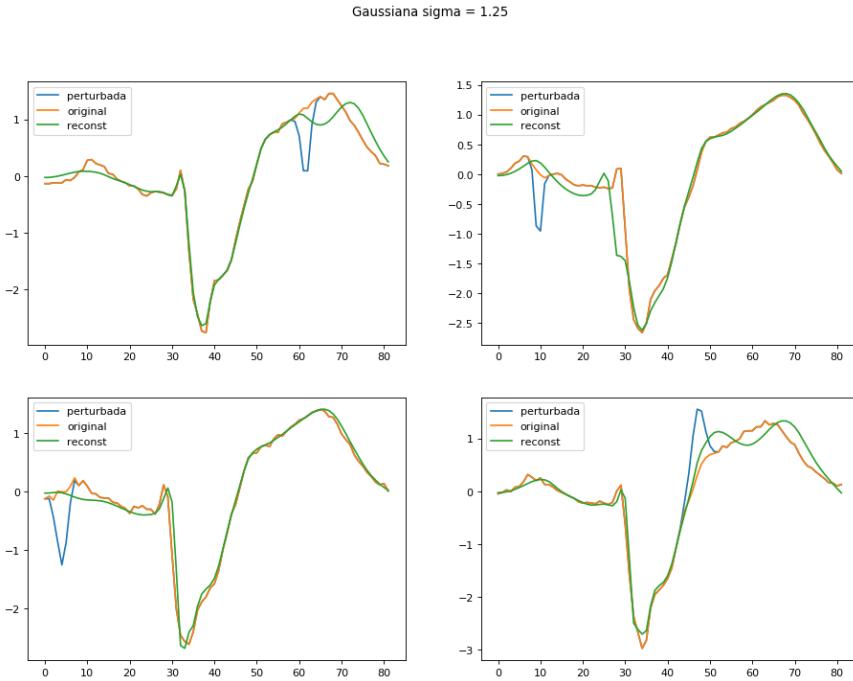


Figura 12.10.: Alteraciones con ruido gaussiano a $\sigma = 1.25$.

perturbaciones que se pueden aplicar. Finalmente por cada ratio tomaremos los σ en $\{0.75, 0.9, 1, 1.25\}$ para notar la diferencias a distinta escala.

Los resultados de *train* los tenemos en Figura 12.11 y de *test* en Figura 12.12.

Se refleja claramente que a cuanta mayor proporción de anomalías mejor se comporta el modelo (mayor valores de PR), aunque probablemente se estabilice entre $[0.2, 0.3]$ que probablemente se deba a la aleatoriedad existente a la hora de escoger la posición y la longitud de la perturbación. También parece que conforme es más intensa la perturbación no parece influir tanto el ratio.

En cualquiera de los casos, todos los resultados indican que el modelo se comporta mejor que el clasificador base (tanto visualmente como con valores de la métrica) por lo que por lo menos el detector funciona en cierta manera. Los peores resultados que puede que se vean están en *test* con 0.05 pero seguramente debido a que son muy pocas anomalías.

Viendo el comportamiento más estable en los ratios de 0.2 y 0.3 queda constatado que el modelo es capaz de detectar bastante bien estas pequeñas anomalías en las escalas más pequeñas que hemos visto que son bastante suaves. Cuando la escala ya empieza a ser más evidente, pero sin ser picos exageradamente grandes, el detector consigue valores de la métrica casi perfectos.

Vistos los resultados del ruido, pasamos al **pulso gaussiano-sinusoidal**, donde operamos de la misma manera que antes, aplicando perturbaciones con tamaño [5, 11] (para que se vea la forma de la señal) y σ en $\{0.75, 0.9, 1, 1.25\}$ que van desde pequeñas alteraciones a más obvias.

Mostramos 4 ejemplos para cada σ en Figura 12.13, Figura 12.14, Figura 12.15 y Figura 12.15.

12. Experimentación

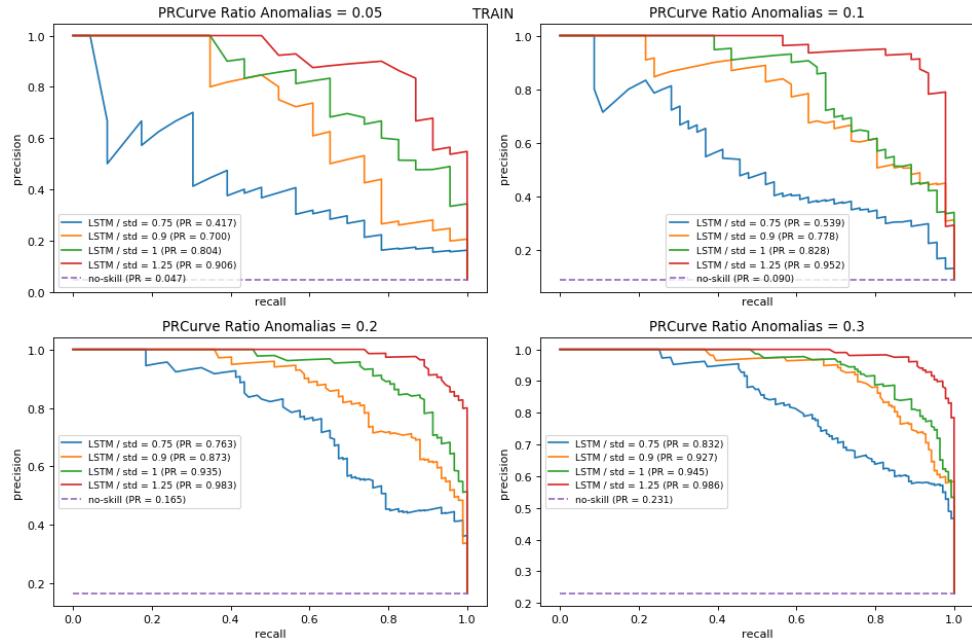


Figura 12.11.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con ruido gaussiano en *train*.

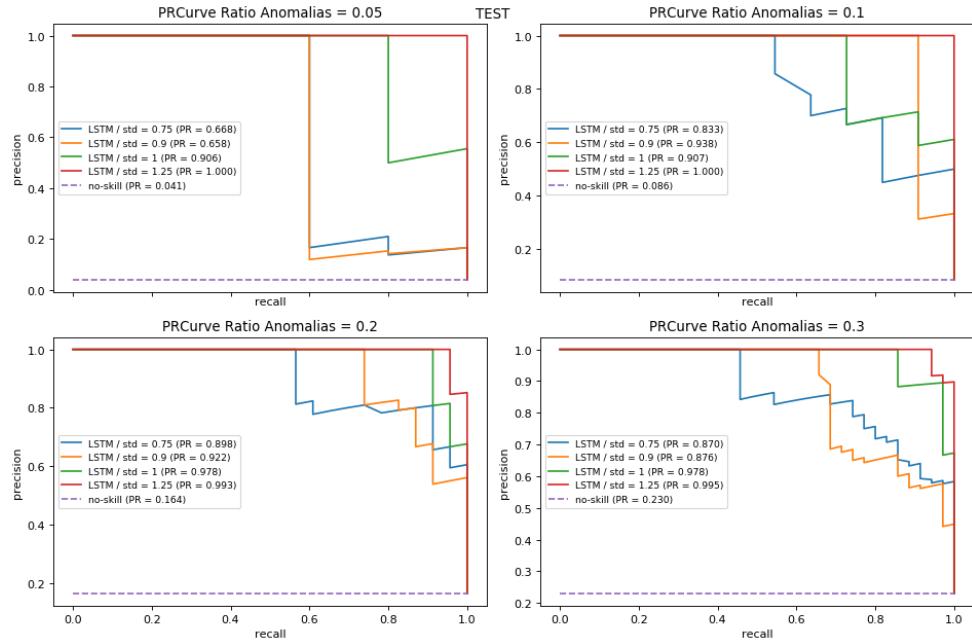
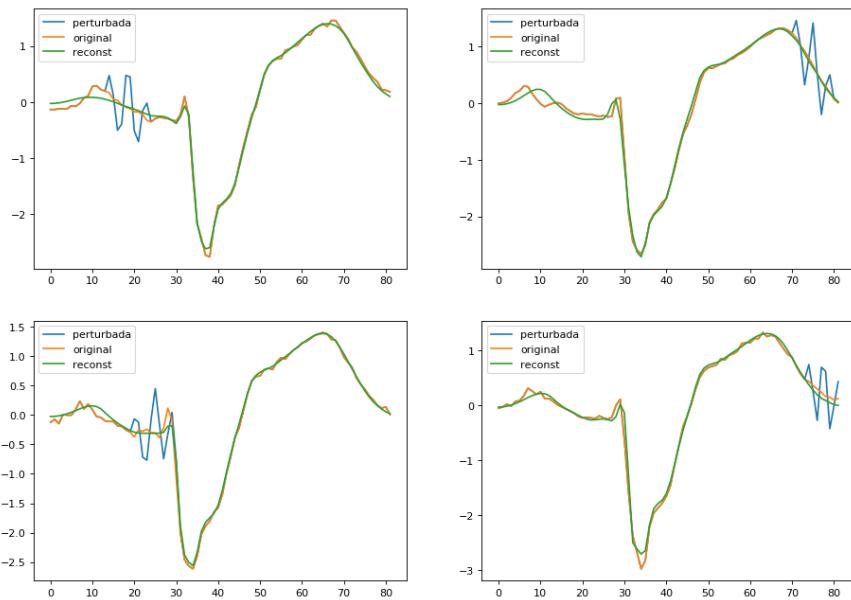
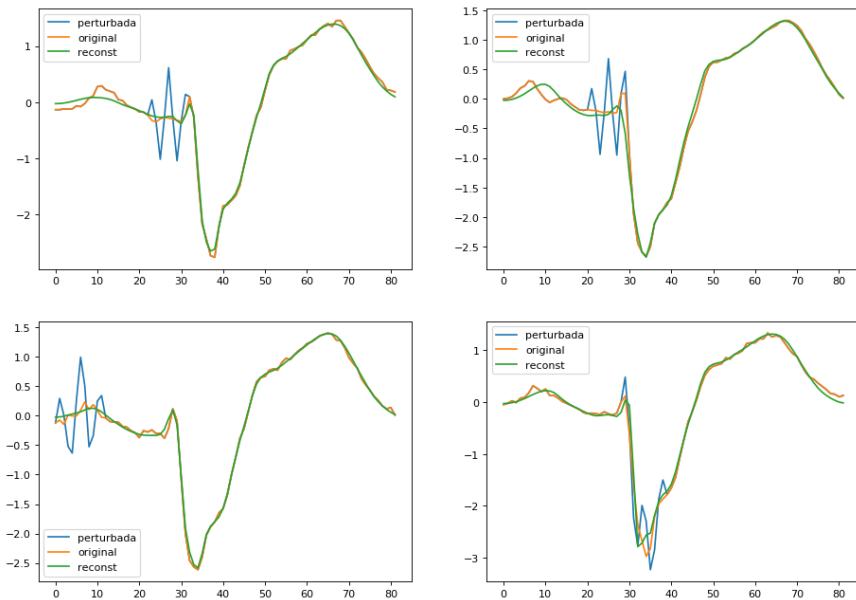


Figura 12.12.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con ruido gaussiano en *test*.

Figura 12.13.: Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 0.75$.Figura 12.14.: Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 0.9$.

12. Experimentación

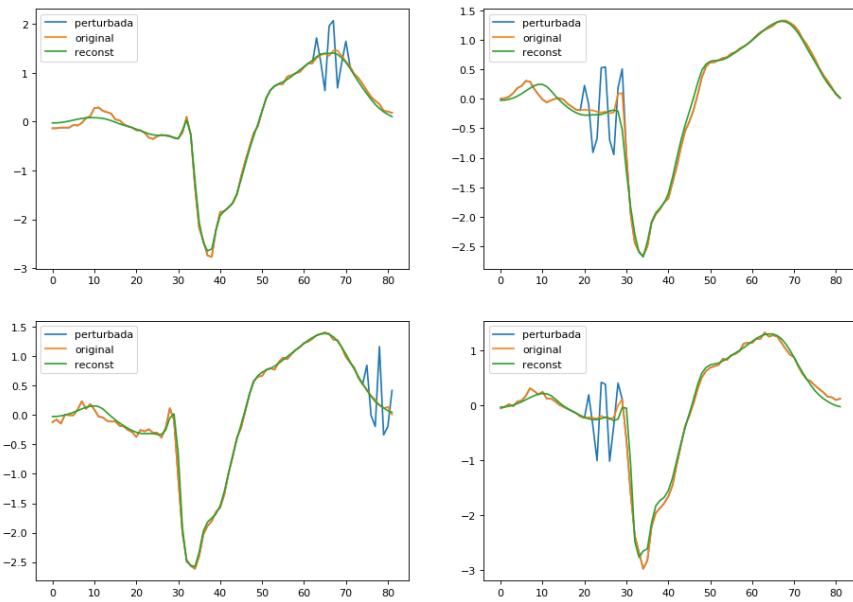


Figura 12.15.: Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 1$.

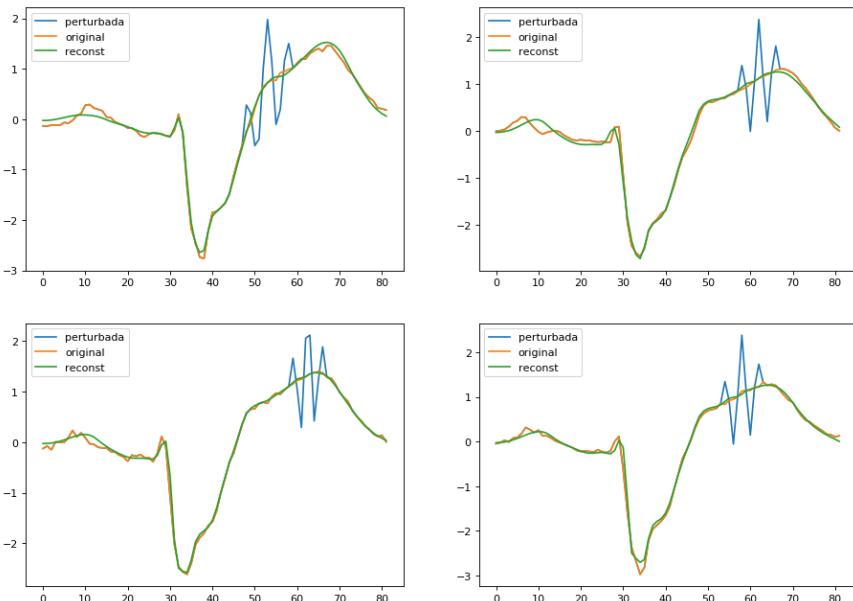
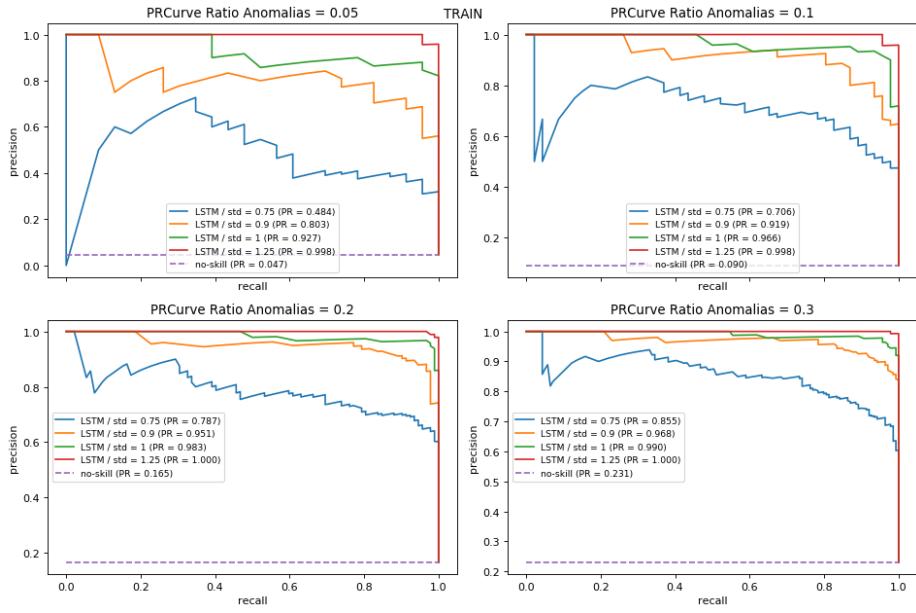


Figura 12.16.: Alteraciones con pulso gaussiano-sinusoidal a $\sigma = 1.25$.

Se aprecia correctamente los pulsos que cambian la forma de la señal, sin tener unos cambios muy bruscos en el rango de valores de la señal original, notándose más cuando $\sigma = 1.25$. También es digno de notar que, a diferencia con el ruido gaussiano, aquí las alteraciones cambian poco la reconstrucción debiéndose quizás al carácter sinusoidal de la perturbación aunque no podemos asegurar nada.

Repetimos de nuevo el cálculo del PR con los mismos tamaños, escalas de σ y ratios de proporción; obteniendo [Figura 12.17](#) en *train* y [Figura 12.18](#) en *test*.



[Figura 12.17.](#): Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con pulso gaussiano-sinusoidal en *train*.

El comportamiento respecto el ratio es igual que el ruido gaussiano, da mejores resultados con mejor ratio, aunque se nota que menos la escala más pequeña de σ el resto obtienen muy buenos resultados casi perfectos notándose más exageradamente en *test*.

Los resultados vuelven a indicar que el detector consigue realizar un buen trabajo a todas las escalas y con muy pocas muestras anómalas, de hecho parece que se consiguen muchos mejores resultados que el ruido gaussiano, aunque también puede deberse a que hayamos aumentado el tamaño mínimo de las perturbaciones.

Cuando la escala es la más pequeña ($\sigma = 0.75$) se comporta bastante peor aunque solo con el menor ratio. Sin embargo, no es de extrañar puesto que aunque estemos fluctuando la forma de la señal en esa escala los cambios son muy pequeños y no se diferencian demasiado de la original. A pesar de todo esto el sistema es capaz de hacer un buen trabajo frente al clasificador base y va mejorando bastante las puntuaciones con mayor proporción de anomalías.

Resumiendo, el sistema es capaz de detectar bastante bien las clases existentes que habíamos clasificado como anómalas y también las alteraciones artificiales que hemos creado a distintas escalas, tamaños y proporciones; indicando el buen rendimiento de este detector.

12. Experimentación

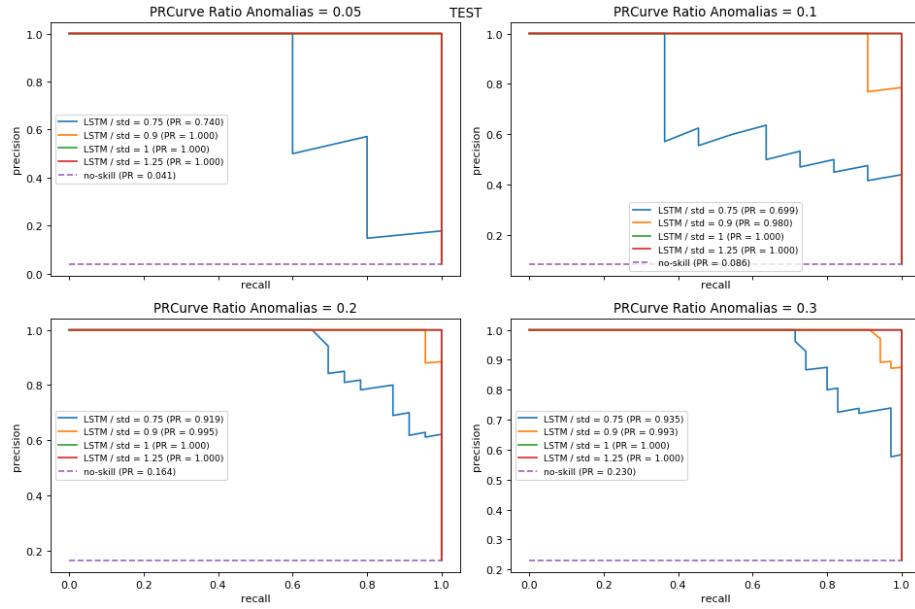


Figura 12.18.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad con pulso gaussiano-sinusoidal en *test*.

12.2. Dataset sintético

12.2.1. Descripción

Para demostrar los métodos basados en descomposición STL, usaremos el *dataset* basado en funciones coseno que habíamos creado a modo de ejemplo anteriormente. Recordamos que muestreábamos una función coseno, le añadíamos tendencia ascendente y ruido blanco; finalmente se estandarizaban. Volvemos a dejar un 20 % para *test*.

Inspeccionamos 10 series del *dataset* para ver si hay un patrón, aunque ya sabemos que sí lo va a haber. En Figura 12.19 vemos claramente el patrón, aunque observamos el ruido introducido provoca pequeñas oscilaciones.

12.2.2. Entrenamiento

Repetimos usando el modelo que teníamos entrenando durante 400 épocas con un número de neuronas a [20, 10, 10, 20] (la función es muy sencilla por lo que necesitamos muchas neuronas), sin regularización, con tasa de aprendizaje de 0.001, tamaño de *batch* de 128 y con un conjunto de validación usando el 10 %.

El resultado del entrenamiento mostrando la función de pérdida MSE se muestra en Figura 12.20. Nos muestra que no hay sobreajuste puesto que ambas curvas van unidas y el entrenamiento es correcto puesto que el error baja casi a 0; en este caso hemos entrenado muchas épocas para bajar al máximo posible los errores.

Observemos algunas series en *train* y *test* con varias reconstrucciones para ver el comportamiento del modelo en Figura 12.21 y Figura 12.22 respectivamente.

Las reconstrucciones parecen bastante buenas ya que aprenden el modelo sinusoidal sub-

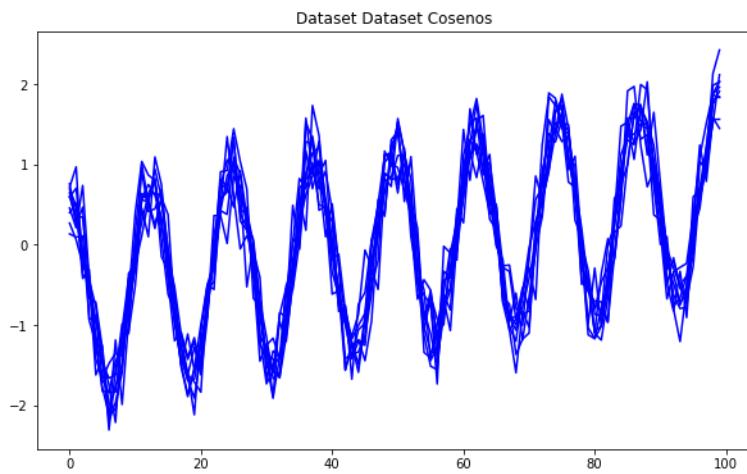


Figura 12.19.: Algunas series del dataset *Cosenos*.

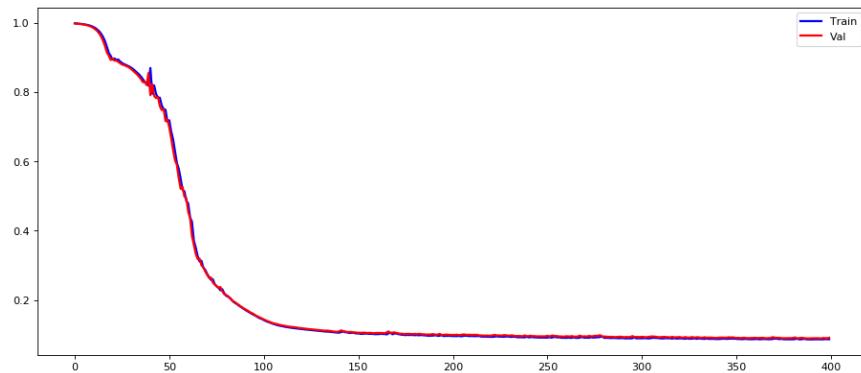


Figura 12.20.: Entrenamiento del modelo LSTM en *Cosenos*.

12. Experimentación

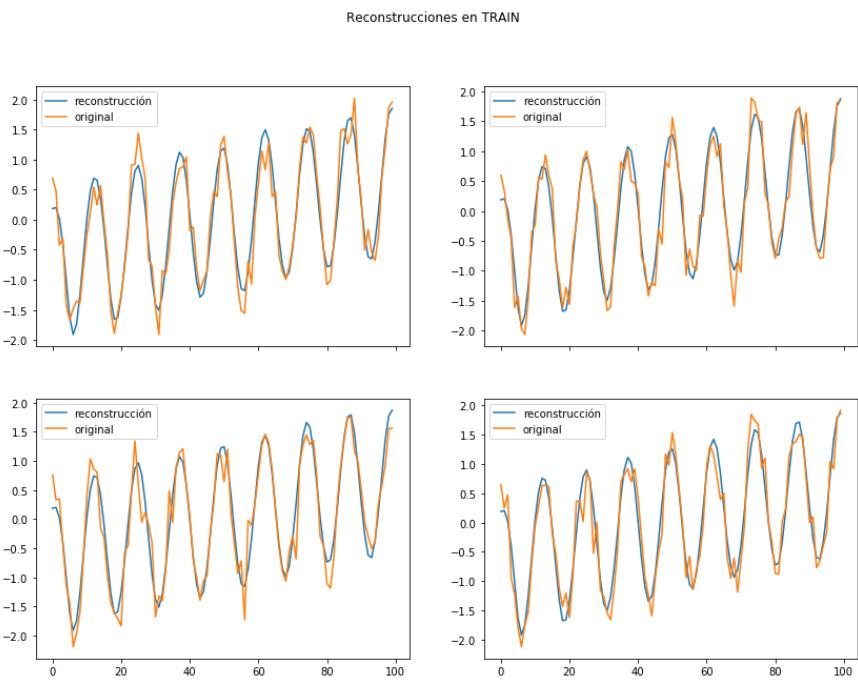


Figura 12.21.: Reconstrucciones en *train*.

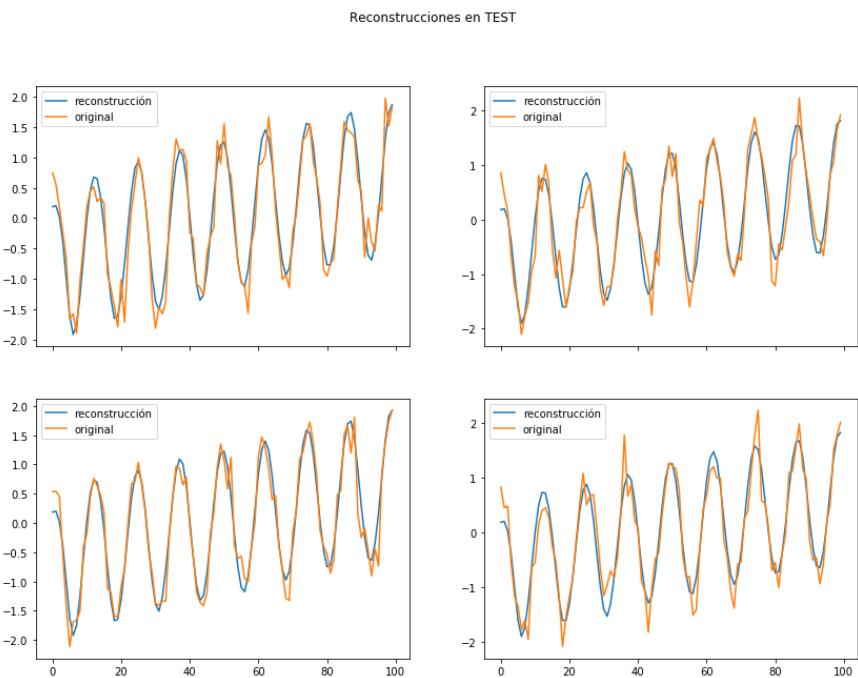


Figura 12.22.: Reconstrucciones en *test*.

yacente sin el ruido blanco, las diferencias que observamos son generalmente debidas a esto por lo que podemos considerar que el modelo ha aprendido correctamente el patrón de los datos.

Estudiamos ahora la distribución de errores MSE entre las series y las reconstrucciones en *train* y *test* en Figura 12.23, junto con la estimación de la función de densidad.

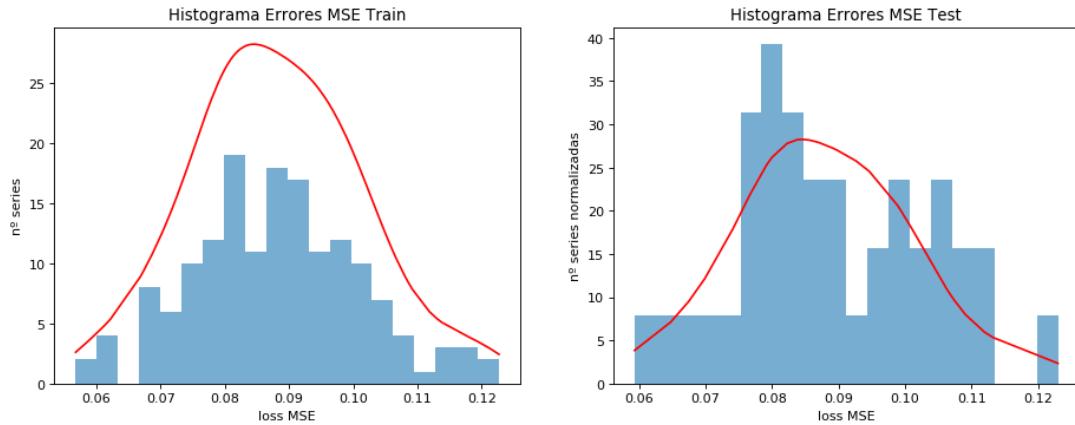


Figura 12.23.: Histogramas de error en *train* y *test*.

Como habíamos visto que el modelo había aprendido correctamente el patrón, los errores esperados se comportan como una gaussiana, es decir como ruido blanco por lo que probablemente no queda más que pueda extraer el modelo. El comportamiento en *train* y *test* es el mismo, por lo que hace una buena generalización.

Habiendo obtenido la función de densidad estimada, el modelo ya ha sido entrenado y listo para poner a prueba frente a las alteraciones.

12.2.3. Validación

Para este conjunto de datos usaremos la validación usando las alteraciones basadas en la descomposición STL para comprobar su eficacia y a la vez el rendimiento del detector ante este tipo de anomalías.

Volvemos a usar como métrica *PR* para valorar el modelo.

12.2.3.1. Estacionalidad

Empezamos a crear anomalías alterando la componente de la **estacionalidad** de las series con tamaños entre $[7, 11]$ y σ en $\{0.01, 0.05, 1.8, 2.0\}$ para que se aprecien las alteraciones en al menos medio periodo y para amortiguar la estacionalidad ($\sigma < 1$) y amplificarla ($\sigma > 1$).

Mostramos 4 ejemplos para cada σ en Figura 12.24, Figura 12.25, Figura 12.26 y Figura 12.27 respectivamente.

Conseguimos el efecto deseado: para $\sigma < 1$ amortiguamos la estacionalidad y para $\sigma > 1$ se amplifica, modificando la serie de una manera sinusoidal debido al carácter propio de su estacionalidad. La reconstrucción se mantiene casi sin alteraciones por lo que queda en evidencia clara la anomalía que se muestra, aunque debido al ruido que había en el *dataset* no es fácil determinar si es una anomalía completamente.

12. Experimentación

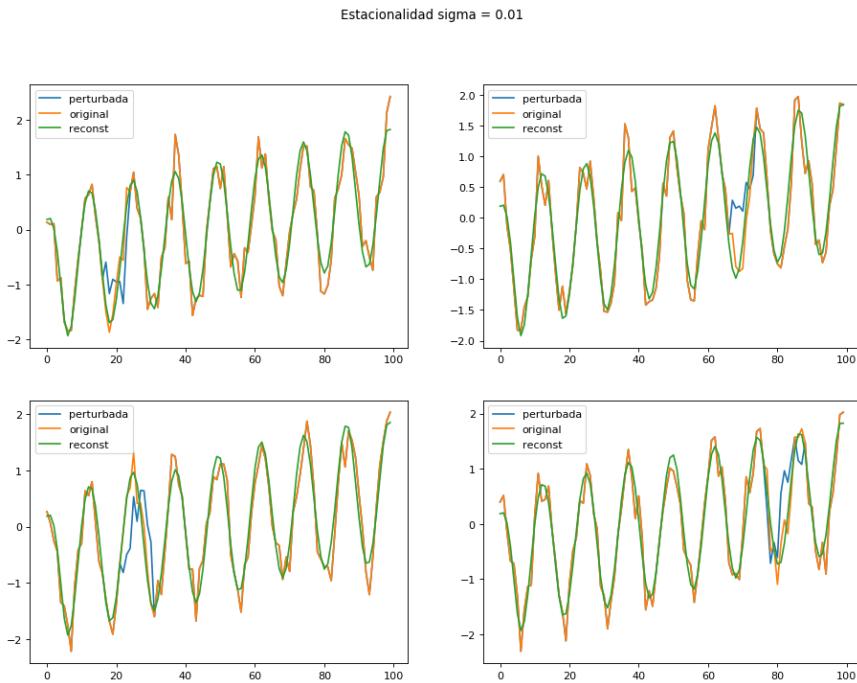


Figura 12.24.: Modificación de la estacionalidad a $\sigma = 0.01$.

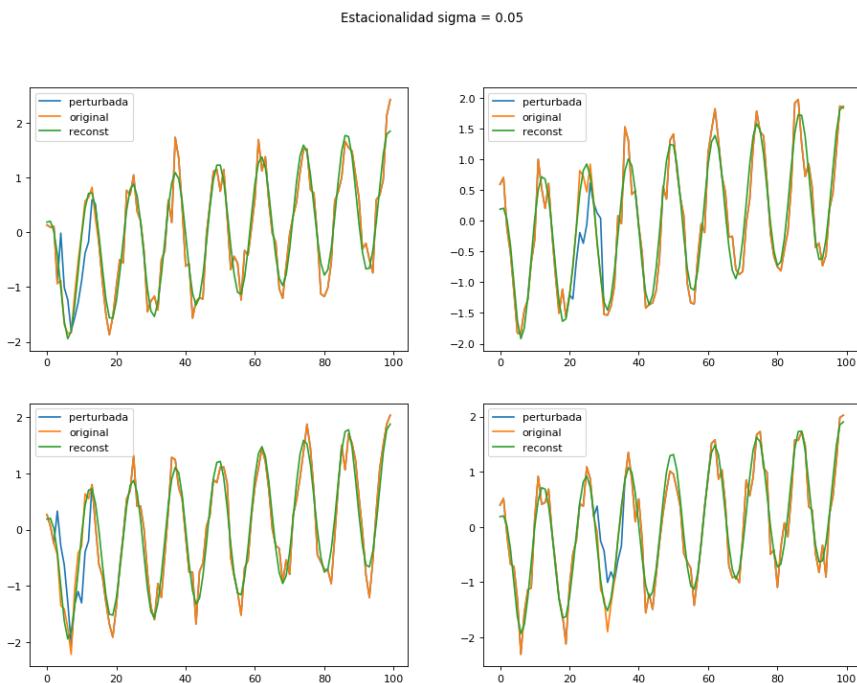


Figura 12.25.: Modificación de la estacionalidad a $\sigma = 0.05$.

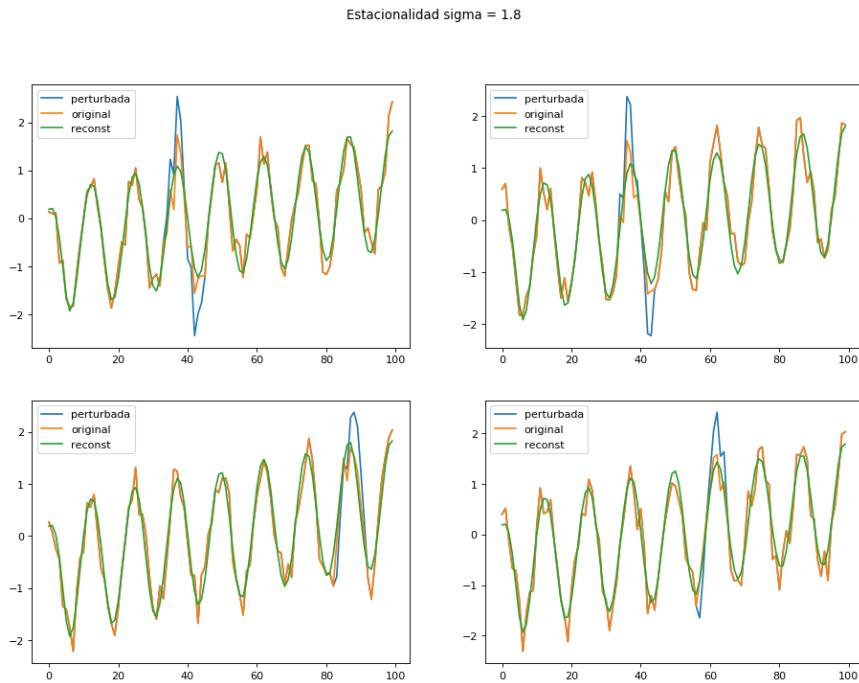


Figura 12.26.: Modificación de la estacionalidad a $\sigma = 1.8$.

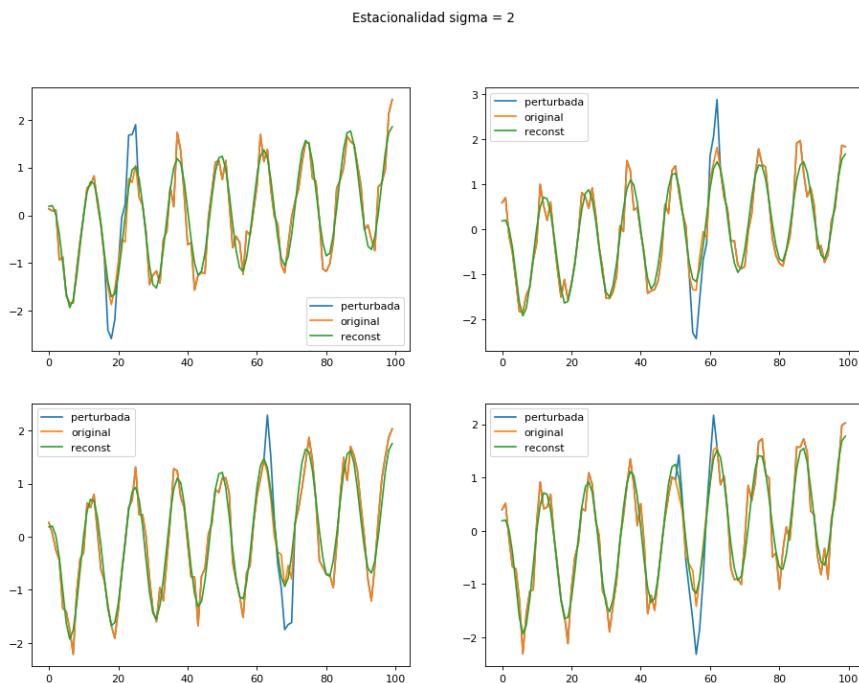


Figura 12.27.: Modificación de la estacionalidad a $\sigma = 2$.

12. Experimentación

Validemos calculando la métrica PR de la misma manera que hicimos con el otro *dataset*: tomando *train* y *test* y creando alteraciones con un número proporcional al tamaño de cada uno respectivamente, en concreto los ratios $\{0.05, 0.1, 0.2, 0.3\}$. La configuración de las alteraciones es la misma que la de los ejemplos.

Obtenemos los resultados de *train* en Figura 12.28 y *test* en Figura 12.29.

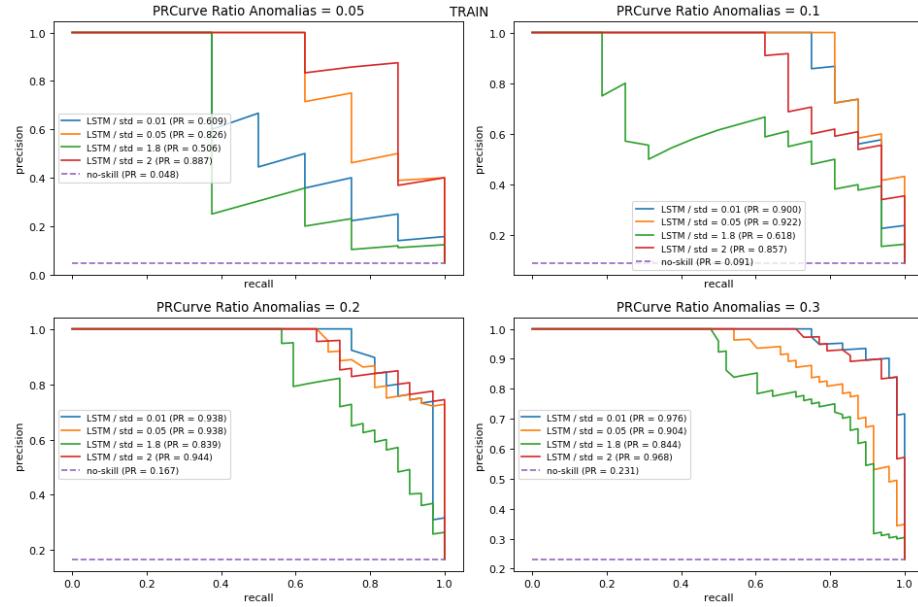


Figura 12.28.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la estacionalidad en *train*.

Se obtiene un resultado parecido en el otro *dataset*: conforme aumenta el número de perturbaciones los valores de las métricas van aumentando hasta estabilizarse; y en *test* es un poco inestable con un ratio bajo debido al menor número de series.

En todos los casos el detector es capaz de obtener un rendimiento superior al clasificador aleatorio, consiguiendo tasas de la métrica muy elevadas desde un ratio del 0.1 por lo que podemos decir el modelo consigue detectar la mayoría de estas anomalías siendo realmente efectivo.

Para los σ de amortiguación de señal vemos que el comportamiento es muy parecido entre los dos valores, en los de amplificación el valor 2 parece que crea anomalías más obvias que 1.8. En cualquier caso, las perturbaciones realizadas parecen ser casi como el ruido gaussiano que hemos introducido, siendo solo un poco más fuertes, y aun así el detector es capaz de detectarlas lo que nos indica la gran sensibilidad que tiene.

12.2.3.2. Tendencia

Repetimos lo que hemos hecho pero ahora modificando la componente de la **tendencia** de las series con tamaños entre $[7, 11]$ y σ en $\{-1, 0.001, 3.5, 4\}$ que han sido tomados para invertir la tendencia ($\sigma < 0$), amortiguarla ($0 < \sigma < 1$) o amplificarla ($\sigma > 1$).

Mostramos 4 ejemplos para cada σ en Figura 12.30, Figura 12.31, Figura 12.32 y Figura 12.33 respectivamente.

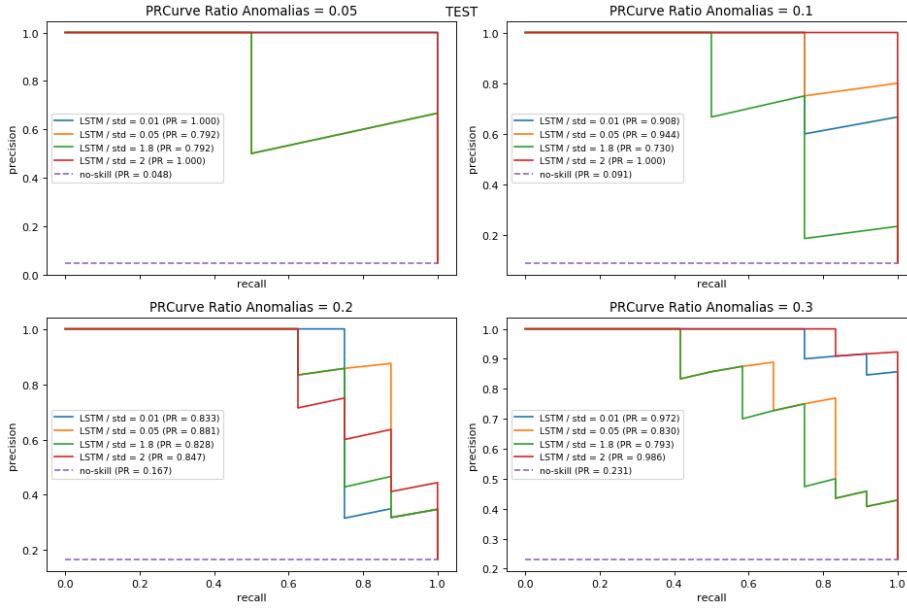


Figura 12.29.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la estacionalidad en *test*.

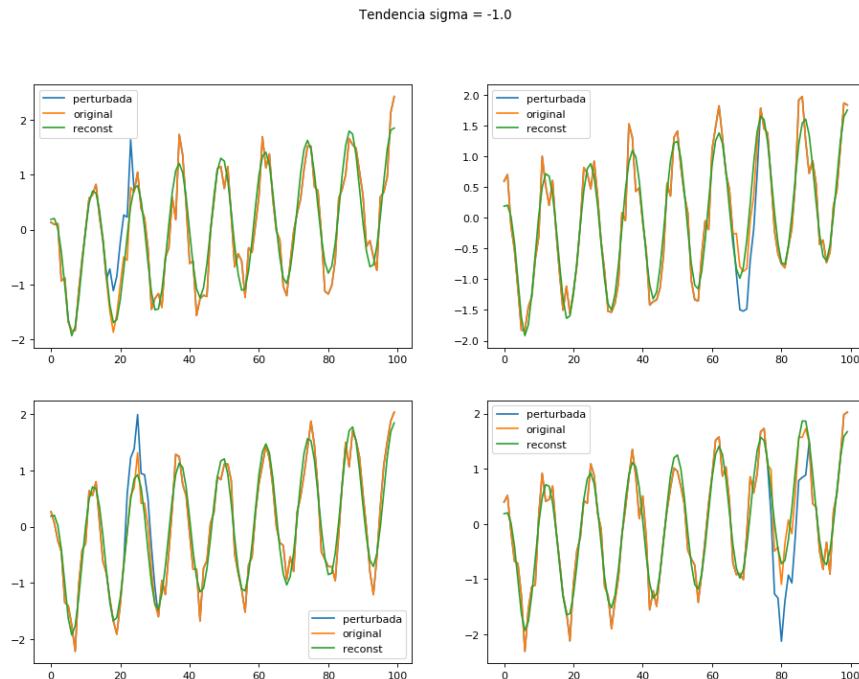


Figura 12.30.: Modificación de la tendencia a $\sigma = -1$.

12. Experimentación

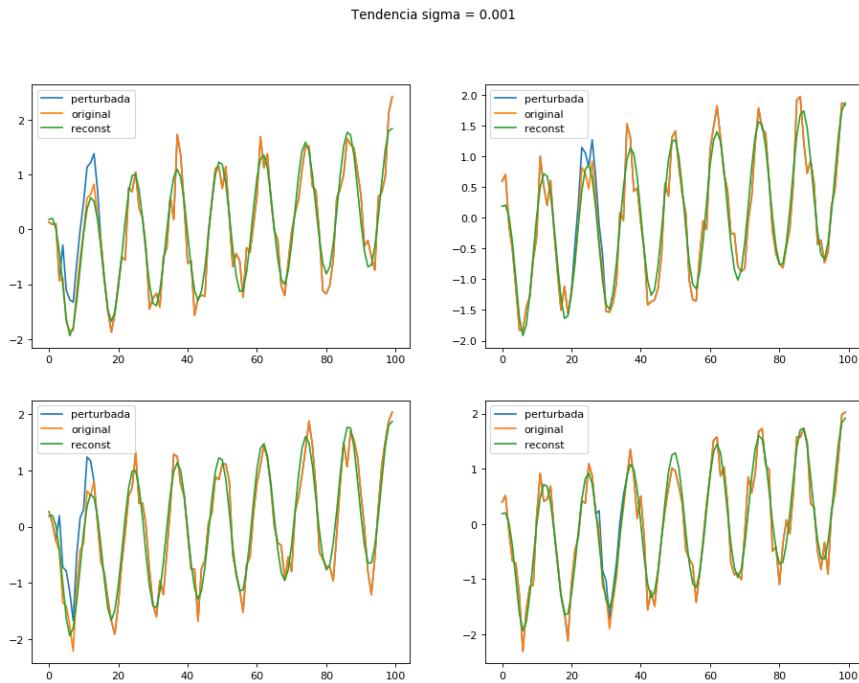


Figura 12.31.: Modificación de la tendencia a $\sigma = 0.001$.

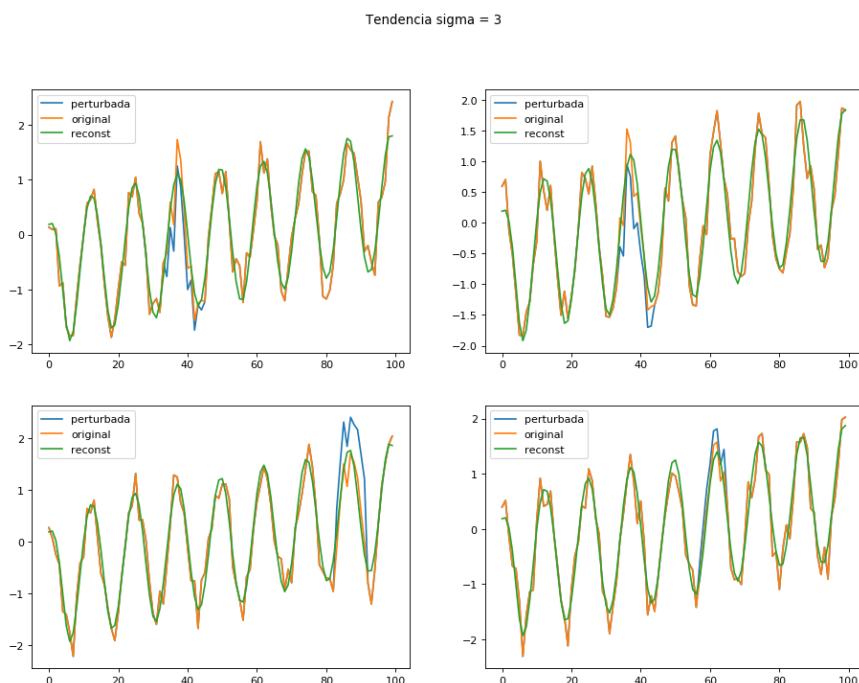


Figura 12.32.: Modificación de la tendencia a $\sigma = 3$.

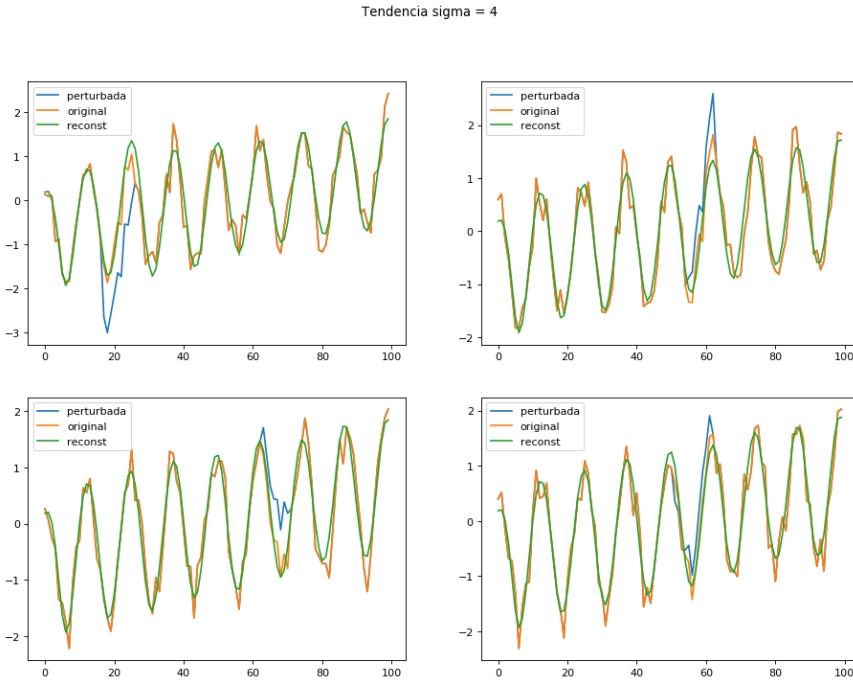


Figura 12.33.: Modificación de la tendencia a $\sigma = 4$.

Como en este caso la tendencia es como una recta que toma valores en $[-0.5, 0.5]$ las alteraciones se notarán más en los extremos ya que conforme nos acercamos al centro de las series la tendencia se anula, haciendo que cualquier variación sea imperceptible.

Con $\sigma = -1$ conseguimos invertir la tendencia y como vemos, en las zonas extremas de las series se puede notar más estas alteraciones; en todo caso como la tendencia no era muy grande en valor absoluto las modificaciones suelen ser pequeñas.

Para $\sigma = 0.001$ amortiguamos la tendencia ocasionando que aumente en las zonas donde era negativa y disminuya para las positivas pero volvemos a notar que estas modificaciones son casi indistinguibles, parece que solo se aprecian en los extremos.

Con $\sigma = 3,4$ amplificamos la tendencia pero volvemos a notar que las perturbaciones no se aprecian mucho aunque estas perturbaciones parecen ser más fuertes que el resto de casos.

Ahora pasamos a validar calculando la métrica PR con la configuración anterior y los mismos ratios que hemos realizado anteriormente. Los resultados obtenidos de *train* y *test* están en Figura 12.34 y Figura 12.35 respectivamente.

Como era de esperar debido a que la tendencia se anula por el centro de las series, la mayoría de estas perturbaciones son casi imperceptibles y se nota debido a las caídas abruptas de las curvas, provocando así un comportamiento en general peor que el resto de alteraciones. Aun así, excepto 0.001, para el resto de σ el modelo es capaz de captar notablemente las alteraciones cuando el ratio de proporciones se estabiliza.

Destaca el peor rendimiento de $\sigma = 0.001$ aunque no es ninguna sorpresa, ya que las modificaciones muchas veces son casi imperceptibles debido a que puede mezclarse con ruido perfectamente o casi no hay efecto al estar cerca del centro. A pesar de ser el peor caso, el detector sigue realizando un mejor trabajo que el detector aleatorio.

12. Experimentación

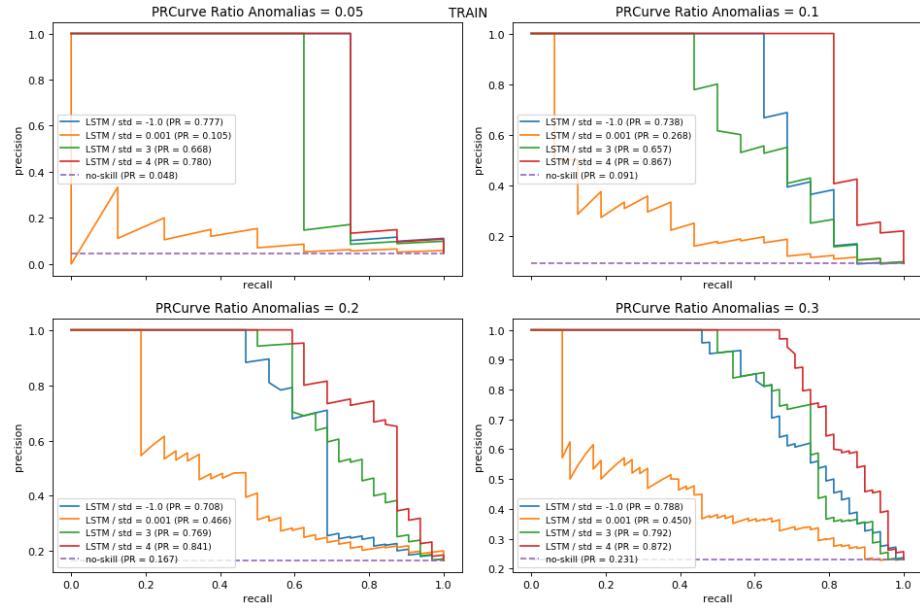


Figura 12.34.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la tendencia en *train*.

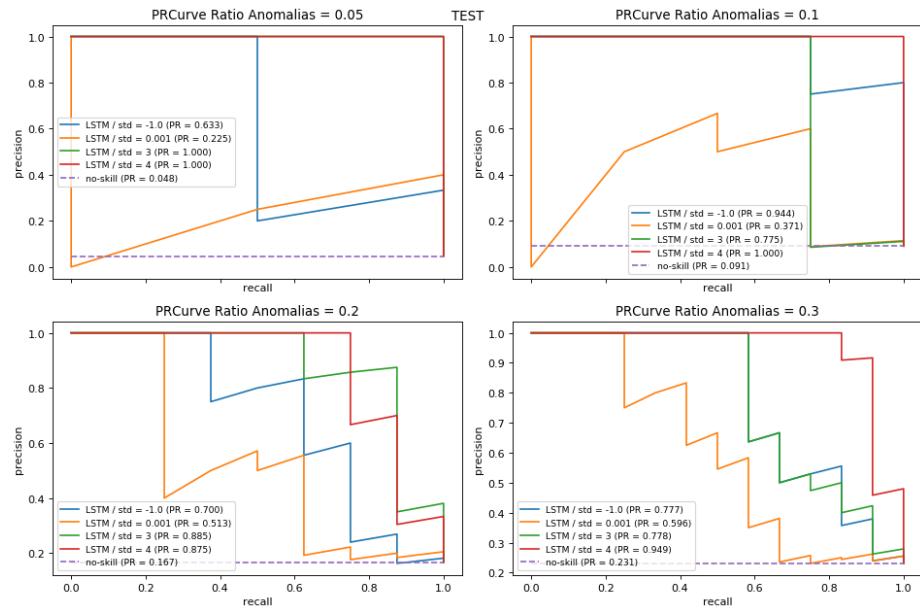


Figura 12.35.: Curvas Sensibilidad-Precisión para distintas proporciones de número de anomalías y de intensidad modificando la tendencia en *test*.

13. Conclusiones y trabajo futuro

Mediante una arquitectura muy simple, capaz de ser desplegada en distintos dominios, hemos conseguido construir un detector capaz de detectar bastante bien distintas anomalías en cuanto a forma y amplitud, siendo estas alteraciones muy sutiles ya que a veces era difícil distinguirlas del ruido blanco propio que podemos encontrar generalmente.

Para ampliaciones futuras sería muy interesante realizar arquitecturas que permitan codificar distintos patrones y no solamente uno como en nuestro caso, ya que suele haber señales regulares que exhiben varios patrones claramente distintos en tramos de las series. Una modificación simple para intentar resolver esto podría consistir en añadir más decodificadores y que cada uno aprendiese un patrón distinto.

También podría estudiarse cómo el detector pudiera mejorar incorporando a la función de probabilidad estimada interna, la distribución de las señales anómalas permitiendo corregir estas probabilidades para una mejor exactitud.

Respecto a los métodos de alteraciones, hemos visto que son maneras muy simples que nos permiten modificar las series de distintas formas permitiéndonos valorar los detectores cuando no tenemos ningún tipo de ejemplo anómalo y siendo perfectamente modificables por los parámetros de longitud y σ para adaptarlos a los datos concretos que se tengan.

En un desarrollo futuro podría verse muchas más alteraciones que incorporar a esta lista, creando así un conjunto de herramientas amplio capaz de crear *datasets* de anomalías muy variados. Así, podríamos estudiar si estas muestras serían capaces de entrenar detectores basados directamente como aprendizaje supervisado y ampliar el conjunto de modelos usados actualmente, siendo actualmente la mayoría de tipo no supervisado.

Parte III.

Predicción de Series Temporales Discretas

14. Introducción

A. Documentación

En este apéndice se deja la documentación en estilo *python* de la documentación de las distintas clases, métodos y funciones más importantes implementados en el proyecto.

A.1. Selección de Modelos

Las clases implementadas para la parte de selección de modelos que se pueden encontrar en la carpeta *PV/src*.

A.1.1. Perturbated Validation

Esta clase y sus métodos se encuentran en el archivo *PV.py*.

PV Clase PV que implementa el método para calcular y manejar la heurística PV. Guarda los datos de las series originales, las perturbaciones realizadas, los ratio de error, el nombre del dataset que se está perturbando, y valores auxiliares para imprimir gráficas del cálculo del PV.

```
class PV:  
    """  
        Clase que implementa Perturbation Validation (PV).  
  
        Attributes  
        _____  
        X : np.array  
            Dataset  
        y : np.array  
            Conjunto de etiquetas perturbadas  
        ds_name : str  
            Nombre del dataset  
        errs : np.array  
            Errores tomados  
        counter : int  
            Contador auxiliar  
        fig : Figure  
            Figura actual  
        ax : Axes  
            Ejes actuales  
    """
```

A. Documentación

Constructor Constructor de la clase PV que necesita los datos originales, el número de perturbaciones, el nombre del dataset, y el inicio y fin de los ratio de error. Crea los conjuntos de etiquetas perturbadas.

```
def __init__(self, X, y, n_pv = 5, ds_name = "", err_ini = 0.1,
             err_fin = 0.3):
    """
        Inicializa la clase creando las etiquetas perturbadas.
    
```

Las perturbaciones se realizan tomando un %err de cada clase, poniendole otra etiqueta distinta.

Se toman "n_pv" puntos entre [err_ini , err_fin].

Parameters

X : np.array
Dataset
y : np.array
Etiquetas
n_pv: int
Número de puntos/errores
ds_name: str
Nombre del databases
err_ini : float
Error inicial
err_fin : float
Error final

"""

Cálculo PV Método para calcular el valor PV de un modelo dado.

```
def get_pv(self, clf, clf_name = "", plot = True):
    """
        Calcula el PV score para el clasificador.
    
```

Parameters

clf : Classifier
Clasificador
clf_name : str
Nombre del clasificador

Returns

pv : float
PV score
accs : list(float)

```

    accs obtenidos
"""

Dibujar cálculo PV Método para representar en una gráfica los valores de la métrica acc obtenidos en el cálculo de PV junto a la recta de regresión obtenida.

def plot_pv(self, errs, accs, poly, pv, clf_name = ""):
    """
        Dibuja los puntos y la recta de regresión en la figura actual.

    Parameters
    -----
    errs : np.array
        Errores
    accs : np.array
        acc obtenidos
    poly : np.array
        Recta de regresión
    pv : float
        Valor PV
    clf_name : str
        Nombre del clasificador
"""

```

Guardar gráfica Método para guardar en una imagen .png el gráfico del método *plot_pv*.

```

def save_graph(self, name_fig):
    """
        Guarda el gráfico de los resultados en un .png

    Parameters
    -----
    name_fig : str
        Nombre (ruta) de la imagen a guardar.
"""

```

A.1.2. Clasificador LSTM

Esta clase y sus métodos se encuentran en el archivo *LSTM.py*.

LSTM La clase LSTM que implementa el clasificador LSTM. Guarda el modelo LSTM, el número de clases, la longitud de las series, opciones de entrenamiento y para gráficas de entrenamiento.

```

class LSTM(BaseEstimator):
    """
        Implementación de una red neuronal con capas LSTM.

```

A. Documentación

Attributes

```
counter : int, static
    Valor auxiliar para ruta de imagen
model : Sequential
    Modelo red neuronal
history : list
    Historial del entrenamiento
n_clases : int
    Número de clases de las etiquetas
input_shape : tuple
    Forma de los datos
epochs: int
    Número de épocas para entrenamiento
verbose : int
    Información sobre el entrenamiento
save_hist : boolean
    Si guardar las gráficas de los entrenamientos
"""

```

Constructor Constructor de la clase LSTM que guarda opciones de entrenamiento.

```
def __init__(self, epochs, n_neurs = 80, verbose = 0, save_hist = False,
             n_clases = -1):
    """

```

Inicializamos la red LSTM.

Attributes

```
epochs : int
    Número de épocas para entrenamiento
n_neurs : int
    Número de neuronas LSTM
verbose : int
    Información sobre el entrenamiento
save_hist : boolean
    Si guardar las gráficas de los entrenamientos
n_clases : int
    Número de clases a predecir
"""

```

Creación del modelo Método para crear el modelo LSTM.

```
def create_model(self):
    """

```

Crea el modelo LSTM.

Compilar el modelo Compila el modelo con el optimizador ADAM y la función de pérdida entropía cruzada categórica.

```
def compile_model(self):
    """
        Compila el modelo con optimizador ADAM y función de pérdida
        categorical_crossentropy .
    """
```

Entrenamiento Método para entrenar el modelo con el conjunto de datos, con las épocas guardadas, validación al 10% y con parada temprana.

```
def fit(self, X, y):
    """
        Entrenamos el modelo .
    """
```

Parameters

X : numpy.array	Datos de entrenamiento
y : numpy.array	Etiquetas de entrenamiento

Cálculo métrica Método para calcular la métrica *acc* en el conjunto de datos pasado.

```
def score(self, X, y):
    """
        Calcula el acc con los datos que se le pasan .
    """
```

Parameters

X : numpy.array	Datos test
y : numpy.array	Etiquetas test

Returns

acc : float	accuracy obtenida
-------------	-------------------

Guardar gráfica de entrenamiento Método para guardar el historial de entrenamiento en una imagen.

```
def save_history(self):
    """
```

A. Documentación

```
    Guarda el historial en una imagen.  
    """
```

A.1.3. Clasificadores

Los clasificadores adicionales que usamos para comparar modelos, comparten dos métodos generales: entrenamiento y cálculo de la métrica.

Entrenamiento Método que se encarga de entrenar el modelo usando una muestra de datos.

```
def fit(self, X, y):  
    """  
        Entrena el modelo.  
  
    Parameters  
    _____  
        X : numpy.array  
            Datos de entrenamiento  
        y : numpy.array  
            Etiquetas de entrenamiento  
    """
```

Cálculo de métrica Método que se encarga de calcular la métrica (*accuracy*) de un modelo en un conjunto de datos.

```
def score(self, X, y):  
    """  
        Calcula el acc con los datos que se le pasan.  
  
    Parameters  
    _____  
        X : numpy.array  
            Datos test  
        y : numpy.array  
            Etiquetas test  
  
    Returns  
    _____  
        acc : float  
            accuracy obtenida  
    """
```

A.1.3.1. C4.5

Esta clase y sus métodos se encuentran en el archivo *RClassifiers.py*.

C45 La clase C45 que usa el árbol de decisión C4.5 que guarda el modelo.

```
class C45(BaseEstimator):
    """
        Implementa el árbol de decisión C4.5.

        Attributes
        -----
        model : clasificador en R
            El clasificador (clase en R)
    """
```

A.1.3.2. C5.0

Esta clase y sus métodos se encuentran en el archivo *RClassifiers.py*.

C50 La clase C50 que usa el árbol de decisión C5.0 que guarda el modelo, y también el valor del *boosting*.

```
class C50(BaseEstimator):
    """
        Implementa el árbol de decisión C5.0 (con boosting o no).

        Attributes
        -----
        model : clasificador en R
            El clasificador (clase en R)
        boosting : int
            El valor del boosting
    """
```

Constructor Constructor de la clase C50 que se le pasa el número de *boosting* que se necesite.

```
def __init__(self, boosting = 10):
    """
        Inicializa el clasificador.

        Parameters
        -----
        boosting : int
            El valor del boosting
    """
```

A.1.3.3. Recursive Partitioning Tree

Esta clase y sus métodos se encuentran en el archivo *RClassifiers.py*.

A. Documentación

RPart Clase que usa el árbol RPart.

```
class RPart(BaseEstimator):  
    """
```

Implementa el árbol de decisión RPart (Recursive Partitioning Tree).

Attributes

*model : clasificador en R
El clasificador (clase en R)*

```
"""
```

A.1.3.4. Condicional Tree

Esta clase y sus métodos se encuentran en el archivo *RClassifiers.py*.

CTree La clase CTree implementa el uso del árbol de decisión Condicional Tree.

```
class CTree(BaseEstimator):  
    """
```

Implementa el árbol de decisión CTree (Conditional Inference Tree).

Attributes

*model : clasificador en R
El clasificador (clase en R)*

```
"""
```

A.1.3.5. k-NN

Esta clase y sus métodos se encuentran en el archivo *KNN.py*.

Clase KNN Clase que implementa el clasificador k-NN que se le puede pasar el *k* fijo o que lo calcule automáticamente tomado como la raíz cuadrada del número de datos.

```
class KNN(BaseEstimator):  
    """
```

Implementa el clasificador KNN (K–Nearest neighbors).

Parameters

*k : int
Número de vecinos
model : KNeighborsClassifier
Modelo k-NN
metric : str, metric
Métrica que usar con KNN
n_jobs : int*

```
    """ Número de procesadores usados
```

Constructor Constructor de la clase KNN que necesita el número de vecinos, la métrica y el número de procesadores.

```
def __init__(self, k = None, metric = "euclidean", n_jobs = 1):
    """
        Inicializa el clasificador.

    Parameters
    -----
    k : int
        Número de vecinos
    metric : str, metric
        Métrica que usar con KNN
    n_jobs : int
        Número de procesadores
    """
```

A.1.3.6. *k*-NN + DTW

Esta clase y sus métodos se encuentran en el archivo *RClassifiers.py*.

DTW Clase que implementa el clasificador *k*-NN con métrica DTW, que guarda los datos de entrenamiento, el número de vecinos y el tamaño de la ventana para aplicar DTW.

```
class DTW(BaseEstimator):
    """
        Clase que implementa K–Nearest Neighbors con la distancia DTW
        usando la implementación del paquete "IncDTW".

    Attributes
    -----
    data : R.DataFrame
        Datos transformados en un objeto dataframe de R
    k : int
        Número de vecinos
    window_shift : int
        Tamaño de la ventana para aplicar DTW
    """
```

Constructor Constructor de la clase DTW que necesita el número de vecinos y el tamaño de la ventana para el cálculo de la métrica DTW.

```
def __init__(self, k = 1, window_shift = 5):
    """
        Constructor de la clase, debe hacerse solo una vez por dataset.
```

Parameters

```
k : int  
    Números de vecinos  
window_shift : int  
    Tamaño de la ventana para aplicar DTW  
"""
```

A.2. Detección de anomalías

Funciones y clases relativas a la parte de detección de anomalías que se encuentran en la carpeta *AD/src*.

A.2.1. Alteración de series

Funciones para la creación de anomalías en base a las series normales implementadas en el archivo *alteraciones.py*.

Tramo aleatorio Función para escoger un tramo aleatorio de la serie en función a la longitud indicada (máxima, mínima, fija).

```
def random_slice(x, max_length = None, min_length = None,  
                  length = None, pos = None, border = 0):  
    """
```

Se encarga de elegir un tramo aleatorio de una serie que queda determinado por una posición y longitud, de manera que el tramo elegido es [posición, posición + longitud].

Se puede determinar una longitud máxima o mínima, o incluso especificar una longitud o posición fijada. También se puede indicar si excluir los extremos (añadir borde).

Parameters

```
x : np.numpy  
    Serie temporal que alterar  
max_length : int, None  
    Longitud máxima de la perturbación  
min_length : int, None  
    Longitud mínima de la perturbación  
length : int, None  
    Longitud fija de la perturbación  
pos : int, None  
    Posición fija de la perturbación  
border : int  
    Borde para excluir la perturbación
```

Returns

pos : int
Posición de la perturbación
length : int
Longitud de la perturbación

"""

Ruido gaussiano Método para alterar un tramo aleatorio de la serie añadiendo ruido gaussiano mediante un parámetro σ que controla la intensidad de esta perturbación, y la longitud máxima y mínima de esta.

```
def gaussian_noise(x, max_length, min_length = 3, std = 3, neg = False,
                    border = 0, neg_random = True):
```

Crea una perturbación de ruido gaussiano añadiendo en un tramo aleatorio un muestreo de la función de densidad normal. Se puede controlar la intensidad.

Además se puede activar aleatoriamente (50%) o de manera fija que la alteración gaussiana sea negativa.

Parameters

x : np.numpy
La serie para alterar
max_length : int
Longitud máxima de la alteración
min_length : int
Longitud mínima de la alteración
std : float
Controla la intensidad de la alteración
neg : boolean
Si invertir la señal gaussiana
border : int
El borde para excluir la perturbación
neg_random : boolean
Si se invierte aleatoriamente las señales

Returns

x : np.numpy
Una copia de la señal perturbada

"""

A. Documentación

Pulso sinusoidal-gaussiano Método para alterar un tramo aleatorio de la serie añadiendo un pulso sinusoidal-gaussiano mediante su frecuencia fc , un parámetro σ que controla la intensidad de la perturbación y la longitud máxima y mínima de esta.

```
def gaussian_sine_pulse(x, max_length, min_length = 3, fc = 1.5, std = 3,
                         border = 0):
    """
    Crea una perturbación con un pulso sinusoidal-gaussiano añadido en un
    tramo aleatorio. Se puede controlar la intensidad y la frecuencia
    del pulso.
    """

    Parameters
```

*x : np.numpy
La serie para alterar
max_length : int
Longitud máxima de la alteración
min_length : int
Longitud mínima de la alteración
fc : float
Frecuencia de la señal del pulso
std : float
Controla la intensidad de la alteración
border : int
El borde para excluir la perturbación*

Returns

*x : np.numpy
Una copia de la señal perturbada*

Estacionalidad Método para alterar un tramo aleatorio de la serie modificando la estacionalidad de la descomposición STL (dada con un periodo) por un parámetro σ que controla la intensidad y la longitud máxima y mínima de esta.

```
def modify_season(x, period, max_length, min_length = 3, std = 1, border = 0):
    """
    Crea una perturbación multiplicando por un real la estacionalidad
    de un tramo aleatorio de la serie. Se necesita el periodo para
    realizar la descomposición STL.
    """

    Parameters
```

*x : np.numpy
La serie para alterar
period : int
Periodo de repetición de la serie para descomposición STL*

```

max_length : int
    Longitud máxima de la alteración
min_length : int
    Longitud mínima de la alteración
std : float
    Controla la intensidad de la alteración
border : int
    El borde para excluir la perturbación
"""

```

Tendencia Método para alterar un tramo aleatorio de la serie modificando la tendencia de la descomposición STL (dada con un periodo) por un parámetro σ que controla la intensidad y la longitud máxima y mínima de esta.

```

def modify_trend(x, period, max_length = 3, std = 1, border = 0):
    """
        Crea una perturbación multiplicando por un real la tendencia
        de un tramo aleatorio de la serie. Se necesita el periodo para
        realizar la descomposición STL.
    
```

Parameters

```

x : np.numpy
    La serie para alterar
period : int
    Periodo de repetición de la serie para descomposición STL
max_length : int
    Longitud máxima de la alteración
min_length : int
    Longitud mínima de la alteración
std : float
    Controla la intensidad de la alteración
border : int
    El borde para excluir la perturbación
"""

```

A.2.2. Detector

Clase y sus métodos implementados para crear el detector de anomalías, implementado en *detector.py*

Clase LSTM_AD Clase que implementa el detector de anomalías basado en autoencoder LSTM. Mantiene el modelo LSTM, la probabilidad estimada y otros parámetros de entrenamiento.

```

class LSTM_AD:
    """
        Clase que implementa un detector de anomalías usando
    
```

A. Documentación

un modelo autoencoder con capas LSTM.

Attributes

```
model : keras.Sequential
        Autoencoder LSTM
n_neur : int
        Número de neuronas base para las capas
alpha : float
        Parámetro de regularización L2
lr : float
        Learning rate
epochs : int
        Número de épocas de entrenamiento
mode : int
        Si incluir espacio de codificación (1) o no (2)
hist : keras.Historial
        Historial de entrenamiento
kernel : scipy.gaussian_kde
        Distribución de errores estimada
"""

```

Constructor Constructor de la clase LSTM_AD que guarda los parámetros relativos al entrenamiento y al modo de arquitectura.

```
def __init__(self, n_neur = 32, alpha = 0, lr = 0.001, epochs = 300,
            mode = 2):
"""

```

Constructor de la clase

Parameters

```
n_neur : int
        Número de neuronas base para las capas
alpha : float
        Parámetro de regularización L2
lr : float
        Learning rate
epochs : int
        Número de épocas de entrenamiento
mode : int
        Si incluir espacio de codificación (1) o no (2)
"""

```

Creación del modelo Función para crear la arquitectura del modelo autoencoder LSTM.

```
def create_model(self, X):
"""

```

Crea la arquitectura del autoencoder LSTM con los atributos de la clase.

Parameters

*X : np.numpy
Series temporales*

"""

Compilación Función para compilar el modelo autoencoder LSTM.

def compile_model(self):

"""

Compila el modelo con ADAM añadiendo un clip de 1, learning rate especificado y minimizando el error cuadrático medio.

"""

Entrenamiento Función para entrenar el modelo autoencoder LSTM.

def load_model(self, path):

"""

Carga el modelo de unos pesos guardados en un archivo

Parameters

*path : str
Ruta donde está el archivo de los pesos*

"""

Reconstrucción Función para obtener las reconstrucciones de un conjunto de series temporales.

"""

Obtiene las reconstrucciones del autoencoder para las series.

Parameters

*X : numpy.array
Datasets de series temporales*

Returns

*reconstrucciones : numpy.array
Reconstrucciones de las series temporales*

"""

A. Documentación

Estimar distribución Función para estimar la distribución de los errores de reconstrucción.

```
def fit_kernel(self, X):
    """
        Ajustamos la distribución de los errores de reconstrucción
        con los datos de entrenamiento.

    Parameters
    -----
    X : numpy.array
        Dataset de series temporales
    """
```

Calcular probabilidades Función para obtener las probabilidades de ser serie anómala para un conjunto de series temporales.

```
def predict_prob(self, X):
    """
        Devolvemos las probabilidades de ser serie anómala para
        cada serie del dataset

    Parameters
    -----
    X : numpy.array
        Dataset de series temporales

    Returns
    -----
    probs : numpy.array
        Probabilidades de anomalía para cada serie
    """
```

A.2.3. Cálculo Curva Precision-Recall

Las funciones para calcular la métrica AUC-PR (curva precisión-recall) que se encuentran en el archivo *calc_pr.py*.

Contar anomalías Se cuentan el número de anomalías detectadas en función de las probabilidades de las series de ser anómalas y de un umbral de probabilidad al partir del cual se considera que es anómala.

```
def count_anomalies(probs, threshold):
    """
        Cuenta cuantas anomalías hay en función a la probabilidad de serlo
        y un umbral de probabilidad.

    Parameters
    -----
```

```

probs : np.numpy
    Array con probabilidades de cada serie de ser anómala
threshold : float
    Umbral de probabilidad a partir del cual se considera anómala

Returns
_____
n_anomalies : int
    Número de anomalías detectadas
"""

```

Calcular sensibilidad Se calcula la sensibilidad del modelo en base a las probabilidades de las series anómalas y un umbral.

```

def calc_recall(probs_anomalies, threshold):
    """
        Calcula la sensibilidad (recall) de un modelo en base a las
        probabilidades de las series anómalas.

    Parameters
    _____
    probs_anomalies : np.numpy
        Array con probabilidades de anomalías de las series anómalas
    threshold : float
        Umbral de probabilidad

    Returns
    _____
    recall : float
        Sensibilidad del modelo
    """

```

Calcular precisión Se calcula la precisión del modelo en base a las probabilidades de las series anómalas y normales junto a un umbral.

```

def calc_precision(probs_normal, probs_anomalies, threshold):
    """
        Calcula la precisión de un modelo en base a las probabilidades
        de las series anómalas y normales.

    Parameters
    _____
    probs_normal : np.numpy
        Array con probabilidades anomalías de las series normales
    probs_anomalies : np.numpy
        Array con probabilidades anomalías de las series anómalas
    threshold : float
        Umbral de probabilidad

```

A. Documentación

Returns

precision : float
Precisión del modelo

"""

Curva Precision-Recall Se calcula la métrica *PR* tomando el área debajo de la curva Precision-Recall integrando en el cuadrado $[0, 1]^2$. Además se imprime una figura mostrando la curva que se forma.

```
def recall_precision_curve(X_normal, X_anomalies, model, clf_name = "clf",
                           title = "recall-precision_curve", axis = None,
                           plot = True):
```

"""

Calcula la métrica PR y además muestra la curva Precision-Recall del modelo.

Parameters

X_normal : np.numpy
Series normales
X_anomalies : np.numpy
Series anómalas
model : detector
Detector de anomalías
clf_name : str
Nombre del detector
title : str
Título de la gráfica
axis : matplotlib.axis
Objeto para imprimir las gráficas
plot : boolean
Si imprimir cosas opcionales de la gráfica

Returns

pr_score : float
Valor de la métrica PR

"""

B. Software

B.1. Archivos del proyecto

Encontramos el [proyecto](#) en la plataforma *Github*, con la siguiente estructura:

- *PV*: Parte selección de modelos
 - *resultados*: contiene los resultados del experimento en .csv y .png para TS Y CMFTS.
 - *src*: archivos fuente.
 - *PV.py*: contiene la clase PV.
 - *KNN.py*: contiene la clase KNN.
 - *LSTM.py*: contiene la clase LSTM.
 - *RClassifiers.py*: contiene las clases DTW, C45, C50, CPart y RPart.
 - *Utils.py*: funciones auxiliares.
 - *experimento_PV.py*: experimento para selección de modelos.
 - *experimento_hiper.py*: experimento para los hiperparámetros.
- *AD*: Parte detección de anomalías
 - *models*: contiene los pesos de los modelos obtenidos.
 - *src*: archivos fuente.
 - *alteraciones.py*: métodos para alterar series.
 - *calc_pr.py*: métodos para calcular la métrica PR.
 - *detector.py*: detector LSTM.
 - *experimento_AD.py*: experimento anomalías.
- *doc*: archivos referentes a la memoria del proyecto.
- *Datasets*: datasets con las versiones TS y CMFTS.

B.2. Lenguajes utilizados

La mayoría de código ha sido implementado en *Python 3.6.8*, habiéndose implementado alguna función y usando paquetes de *R 3.4.4*.

B. Software

B.3. Librerías utilizadas

Listamos las librerías utilizadas en el proyecto. En *Python 3.6.8*:

- *matplotlib 3.1.1*: para imprimir gráficos.
- *numpy 1.18.3*: para trabajar con arrays, matrices, tensores.
- *pandas 0.24.1*: cargar y guardar *datasets*.
- *Keras 2.2.5*: utilizada para implementar modelos de redes neuronales LSTM.
- *scikit-learn 0.20.2*: se han usado distintos clasificadores, métricas y preprocesado de *datasets*.
- *scipy 1.2.1*: para señales gaussianas, pulso sinusoidal-gaussiano y estimación de distribución.
- *statsmodels 0.9.0*: descomposición STL.
- *rpy2 3.0.0*: interfaz para utilizar *R* con *Python*.

En *R 3.4.4*:

- *RWeka 0.4.42*: árbol de decisión C4.5
- *C50 0.1.3*: árbol de decisión C5.0
- *rpart 4.1.13*: árbol de decisión RPart.
- *partykit 1.2.3*: árbol de decisión CTree.
- *IncDTW 1.1.3.1*: función para calcular la métrica DTW.

Bibliografía

Las referencias se listan por orden alfabético. Aquellas referencias con más de un autor están ordenadas de acuerdo con el primer autor.

- [ABK20] William Vickers Anthony Bagnall, Jason Lines and Eamonn Keogh. The UEA & UCR Time Series Classification Repository, 2020. URL: www.timeseriesclassification.com. [Citado en págs. 78, 88, 105, and 113]
- [AMH16] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016. [Citado en pág. 99]
- [AMMIL12] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data*, volume 4. AMLBook New York, NY, USA:, 2012. [Citado en págs. 26, 28, 29, 32, and 36]
- [ATR19] Beth Atkinson, Terry Therneau, and Brian Ripley. Recursive partitioning and regression trees, 2019. URL: <https://cran.r-project.org/web/packages/rpart/rpart.pdf>. [Citado en pág. 74]
- [Ban18] Shivam Bansal. 3D Convolutions : Understanding + Use Case, 2018. URL: <https://www.kaggle.com/shivamb/3d-convolutions-understanding-use-case>. [Citado en pág. 43]
- [BC94] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, USA:, 1994. [Citado en pág. 75]
- [BFSO84] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984. [Citado en págs. 73 and 74]
- [Bha18] Anup Bhande. What is underfitting and overfitting in machine learning and how to deal with it., 2018. URL: <https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76>. [Citado en pág. 39]
- [Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. [Citado en pág. 74]
- [BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994. [Citado en pág. 48]
- [CCMT90] Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning. Stl: A seasonal-trend decomposition. *Journal of official statistics*, 6(1):3–73, 1990. [Citado en pág. 55]
- [CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967. [Citado en pág. 75]
- [Cur44] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944. [Citado en pág. 33]
- [CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. [Citado en pág. 73]
- [CVMG⁺14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. [Citado en pág. 52]

Bibliografía

- [Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. [Citado en págs. 40 and 41]
- [Des18] Julien Despois. Memorizing is not learning! — 6 tricks to prevent overfitting in machine learning., 2018. URL: <https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42>. [Citado en pág. 39]
- [Elm90] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990. [Citado en pág. 47]
- [Gau20] Laurent Gautier. Interface to R running embedded in a Python process., 2020. URL: <https://rpy2.github.io/>. [Citado en pág. 74]
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>. [Citado en págs. 48 and 54]
- [GPAM⁺14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. [Citado en pág. 46]
- [Gra14] Benjamin Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014. [Citado en pág. 44]
- [GSoo] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000. [Citado en pág. 52]
- [GSC99] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999. [Citado en pág. 54]
- [GSK⁺16] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016. [Citado en pág. 54]
- [HA18] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018. [Citado en pág. 55]
- [HBH⁺20] Kurt Hornik, Christian Buchta, Torsten Hothorn, Alexandros Karatzoglou, David Meyer, and Achim Zeileis. C4.5 R Implementation, 2020. URL: <https://cran.r-project.org/web/packages/RWeka/RWeka.pdf>. [Citado en pág. 74]
- [Heb49] Donald Olding Hebb. *The organization of behavior: a neuropsychological theory*. J. Wiley; Chapman & Hall, 1949. [Citado en pág. 21]
- [Her18] John A Hertz. *Introduction to the theory of neural computation*. CRC Press, 2018. [Citado en pág. 39]
- [HHZo6] Torsten Hothorn, Kurt Hornik, and Achim Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical statistics*, 15(3):651–674, 2006. [Citado en pág. 74]
- [HHZ15] Torsten Hothorn, Kurt Hornik, and Achim Zeileis. ctree: Conditional inference trees. *The Comprehensive R Archive Network*, 8, 2015. [Citado en pág. 74]
- [Ho95] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995. [Citado en pág. 74]
- [Hop82] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982. [Citado en pág. 21]
- [Hot20] Torsten Hothorn. CTree R Implementation, 2020. URL: <https://www.rdocumentation.org/packages/partykit/versions/1.2-9/topics/ctree>. [Citado en pág. 74]

- [HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. [Citado en págs. 21 and 49]
- [HSK⁺12] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. [Citado en pág. 39]
- [Jai18] Brijnesh J Jain. Semi-metrification of the dynamic time warping distance. *arXiv preprint arXiv:1808.09964*, 2018. [Citado en pág. 75]
- [Jav] JavaTpoint. Support vector machine algorithm. URL: <https://www.javatpoint.com/machine-learning-support-vector-machine-algorithm>. [Citado en pág. 73]
- [Jor97] Michael I Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997. [Citado en pág. 21]
- [KH92] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992. [Citado en pág. 39]
- [KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951. [Citado en pág. 45]
- [KWC⁺20] Max Kuhn, Steve Weston, Mark Culp, Nathan Coulter, and Ross Quinlan. C5.0 R Implementation, 2020. URL: <https://cran.r-project.org/web/packages/C50/C50.pdf>. [Citado en pág. 74]
- [LB⁺95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995. [Citado en pág. 43]
- [Leo20] Maximilian Leodolter. Incremental calculation of dynamic time warping, 2020. URL: <https://cran.r-project.org/web/packages/IncDTW/IncDTW.pdf>. [Citado en pág. 76]
- [LLPS93] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993. [Citado en pág. 40]
- [LMP43] HD Landahl, Warren S McCulloch, and Walter Pitts. A statistical consequence of the logical calculus of nervous nets. *The bulletin of mathematical biophysics*, 5(4):135–137, 1943. [Citado en pág. 21]
- [LPW⁺17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, pages 6231–6239, 2017. [Citado en pág. 40]
- [M18] Sanjay. M. MachineLearning — KNN using scikit-learn, 2018. URL: <https://towardsdatascience.com/knn-using-scikit-learn-c6bed765be75>. [Citado en pág. 75]
- [Mol19] Rekha Molala. The Ascent of Gradient Descent, 2019. URL: <https://blog.clairvoyantsoft.com/the-ascent-of-gradient-descent-23356390836f>. [Citado en pág. 33]
- [Mou16] Adil Moujahid. A Practical Introduction to Deep Learning with Caffe and Python, 2016. URL: <http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/>. [Citado en pág. 30]
- [MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. [Citado en pág. 23]
- [MRG⁺86] James L McClelland, David E Rumelhart, PDP Research Group, et al. Parallel distributed processing. *Explorations in the Microstructure of Cognition*, 2:216–271, 1986. [Citado en pág. 45]
- [NLWH18] Mengting Niu, Yanjuan Li, Chunyu Wang, and Ke Han. Rfamyloid: a web server for predicting amyloid proteins. *International Journal of Molecular Sciences*, 19(7):2071, 2018. [Citado en pág. 65]

Bibliografía

- [NNN⁺20] Thanh Thi Nguyen, Coung M. Nguyen, Dung Tien Nguyen, Duc Thanh Nguyen, and Saedi Nahavandi. Deep Learning for Deepfakes Creation and Detection: A Survey. *arXiv:1909.11573*, 2020. [Citado en pág. 47]
- [Nov63] Albert B Novikoff. On convergence proofs for perceptrons. Technical report, STANFORD RESEARCH INST MENLO PARK CA, 1963. [Citado en pág. 25]
- [OA18] Johanna Orellana Alvear. Arboles de decision y random forest, 2018. URL: <https://bookdown.org/content/2031/ensambladores-random-forest-parte-i.html>. [Citado en pág. 74]
- [OJo4] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004. [Citado en pág. 21]
- [Ola15] Christopher Olah. Understanding LSTM Networks, 2015. URL: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. [Citado en págs. 47, 48, 49, 50, 51, 52, and 54]
- [Pel20] Pelatrlon. 2d convolution block, 2020. URL: <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/2d-convolution-block>. [Citado en pág. 43]
- [PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013. [Citado en pág. 54]
- [Qui14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014. [Citado en pág. 74]
- [Qui19] J. Ross Quinlan. C5.0 C Implementation, 2019. URL: <https://www.rulequest.com/see5-info.html>. [Citado en pág. 74]
- [RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985. [Citado en pág. 28]
- [RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. [Citado en pág. 35]
- [RM05] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers-a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005. [Citado en pág. 75]
- [Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. [Citado en pág. 24]
- [Rud73] Walter Rudin. *Functional analysis*. McGraw-Hill, New York, 1973. [Citado en pág. 41]
- [Rudo06] Walter Rudin. *Real and complex analysis*. Tata McGraw-hill education, 2006. [Citado en pág. 41]
- [Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. [Citado en págs. 33 and 35]
- [Sah18] Sumit Saha. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, 2018. URL: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>. [Citado en pág. 44]
- [SC20] Fernando Sancho Caparrini. Variational autoencoder, 2020. URL: <http://www.cs.us.es/~fsancho/?e=232>. [Citado en págs. 45 and 46]
- [Sil18] Thalles Silva. An intuitive introduction to Generative Adversarial Networks (GANs), 2018. URL: <https://www.freecodecamp.org/news/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394/>. [Citado en pág. 46]
- [SL20a] Scikit-Learn. CART Python Implementation, 2020. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>. [Citado en pág. 73]

- [SL2ob] Scikit-Learn. K Nearest Neighbors Classifier Python Implementation, 2020. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>. [Citado en pág. 75]
- [sl2oc] scikit learn. Precision-recall (pr), 2020. URL: https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html. [Citado en pág. 59]
- [SL2od] Scikit-Learn. Random Forest Python Implementation, 2020. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>. [Citado en pág. 74]
- [sl2oe] scikit learn. Receiver operating characteristic (roc), 2020. URL: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html. [Citado en pág. 59]
- [SL2of] Scikit-Learn. SVM Classifier Python Implementation, 2020. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>. [Citado en pág. 73]
- [SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. [Citado en pág. 33]
- [Sto74] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974. [Citado en pág. 65]
- [VC15] Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pages 11–30. Springer, 2015. [Citado en pág. 39]
- [VNZ12] Petr Volny, David Novák, and Pavel Zezula. Employing subsequence matching in audio data processing. *arXiv preprint arXiv:1204.2541*, 2012. [Citado en pág. 75]
- [Wik] Wikipedia. Lineare separierbarkeit. URL: https://de.wikipedia.org/wiki/Lineare_Separierbarkeit. [Citado en pág. 25]
- [WR17] Haohan Wang and Bhiksha Raj. On the origin of deep learning. *arXiv preprint arXiv:1702.07800*, 2017. [Citado en pág. 21]
- [Y⁺19] Muhamad Yani et al. Application of transfer learning using convolutional neural network method for early detection of terry’s nail. In *Journal of Physics: Conference Series*, volume 1201, page 012052. IOP Publishing, 2019. [Citado en pág. 44]
- [ZHG⁺19] Jie M Zhang, Mark Harman, Benjamin Guedj, Earl T Barr, and John Shawe-Taylor. Perturbation validation: A new heuristic to validate machine learning models. *arXiv preprint arXiv:1905.10201*, 2019. [Citado en págs. 63, 64, 66, 68, 76, and 77]

