Trabajo de Fin de Máster
Máster en Lógica, Computación e Inteligencia Artificial

# Optimización Automática de Hiperparámetros en Modelos de Aprendizaje Automático mediante PBIL

Autor: Roberto Andrés Vasco Carofilis

Tutores: Miguel A. Gutiérrez Naranjo y Miguel Cárdenas Montes

**Dpto. de Ciencias de la Computación e Inteligencia Artificial**
**Escuela Técnica Superior de Ingeniería Informática**
**Universidad de Sevilla**

Sevilla, 2019

Trabajo de Fin de Máster
Máster en Lógica, Computación e Inteligencia Artificial

# Optimización Automática de Hiperparámetros en Modelos de Aprendizaje Automático mediante PBIL

Autor:
Roberto Andrés Vasco Carofilis

Tutores:
Miguel A. Gutiérrez Naranjo y Miguel Cárdenas Montes

Dpto. de Ciencias de la Computación e Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

Sevilla, 2019

# Agradecimientos

Deseo expresar mi agradecimiento a los doctores Miguel Ángel Gutiérrez y Miguel Cárdenas, por su conocimiento y guía brindados durante la realización de este proyecto.
Y mi más grande agradecimiento a mi familia, quienes son mi soporte y mi inspiración en todo momento.

# Resumen

La optimización de hiperparámetros en redes neuronales profundas es una tarea crítica para el rendimiento final de una red. Sin embargo, conlleva un alto porcentaje de decisiones subjetivas basadas en el conocimiento previo de los diseñadores del sistema.

En este trabajo se presenta la implementación de Population-based incremental learning (PBIL); un método de optimización que combina algoritmos genéticos con aprendizaje competitivo, para la optimización automática de hiperparámetros en arquitecturas de aprendizaje profundo.

En el caso de estudio propuesto se aplica una combinación de preprocesamiento de series temporales mediante Seasonal Decomposition of Time Series by Loess (STL); un método clásico de descomposición de series temporales, y redes neuronales convolucionales para la predicción de valores futuros. En el pasado, esta combinación ha producido resultados prometedores, pero se ha visto afectada por el número incremental de hiperparámetros.

La arquitectura propuesta se aplica a la predicción del nivel del $^{222}Rn$ en el Laboratorio de Canfranc (España). Prediciendo los períodos de bajo nivel de $^{222}Rn$, se podría minimizar la contaminación potencial durante las operaciones de mantenimiento en los experimentos del laboratorio. En este documento se muestra como PBIL puede ser utilizado para la selección de hiperparámetros con un coste computacional razonable.

**Palabras clave:** Optimización automática de hiperparámetros / PBIL / Redes neuronales convolucionales / Descomposición STL / Predicción de series temporales / Niveles de $^{222}Rn$

# Abstract

The optimization of hyperparameters in deep neural networks is a critical task for the final performance of a network. However, it entails a high percentage of subjective decisions based on the prior knowledge of the system designers.

This paper presents the implementation of Population-based incremental learning (PBIL); an optimization method that combines genetic algorithms with competitive learning, for the automatic optimization of hyperparameters in deep learning architectures.

In the proposed case study a combination of Seasonal Decomposition of Time Series by Loess (STL); a classical method of time series decomposition, and convolutional neural networks for predicting future values is applied. In the past, this combination has produced promising results, but has been affected by the incremental number of hyperparameters.

The proposed architecture is applied to the prediction of the $^{222}Rn$ level at the Canfranc Laboratory (Spain). By predicting low level periods of $^{222}Rn$, potential contamination during maintenance operations in laboratory experiments could be minimized. In this document it is shown how PBIL can be used for the selection of hyperparameters with a reasonable computational cost.

**Keywords:** Automated hyperparameters tuning / PBIL / Convolutional neural networks / STL decomposition / Time series forecast / $^{222}Rn$ measurements

# Índice

# Índice de Figuras

# Índice de Tablas

# 1 Introducción

Ajustar los hiperparámetros en aprendizaje automático es una tarea tediosa pero crucial, ya que el rendimiento de un modelo puede depender de la elección de sus hiperparámetros. La optimización manual es un proceso que consume mucho tiempo, el cual puede invertirse en otras tareas del proceso de desarrollo. Este trabajo presenta un enfoque automático para la generación de hiperparámetros de un modelo de aprendizaje automático que, en pocas sesiones de entrenamiento, permite alcanzar un error mínimo local.

El método implementado se conoce como Aprendizaje Incremental Basado en Población ("PBIL", por sus siglas en inglés) y será adaptado para optimizar hiperparámetros de modelos de aprendizaje automático. PBIL combina algoritmos evolutivos con aprendizaje competitivo para optimizar un vector de probabilidades mediante el cual se generan nuevos individuos en una sucesión de generaciones, cada individuo representa una posible solución al problema. Con el enfoque propuesto se busca que PBIL sea capaz de generar combinaciones de hiperparámetros que se acerquen a un error mínimo local con el paso de pocas generaciones.

De manera resumida, durante la ejecución de PBIL con el nuevo enfoque planteado, se optimiza un vector de probabilidades (al cual se llamará $p$) a través de una sucesión de generaciones. Utilizando $p$ como base, se genera una población de individuos en cada generación. Cada individuo es un vector binario que representa una combinación de hiperámetros a partir de la cual se genera un modelo de aprendizaje automático. Cada modelo es evaluado mediante una función de *fitness* especificada, encargada de devolver la calidad de las predicciones realizadas por cada modelo. Los mejores individuos son tomados en cuenta para actualizar $p$ aumentando la probabilidad de que en futuras generaciones se obtengan las mismas características presentes en ellos. Se realiza el proceso contrario con los peores individuos, reduciendo la probabilidad de generar sus características. De esta manera, con cada nueva generación el algoritmo es capaz de

producir individuos mejores que sus predecesores, acercándose así al error mínimo objetivo.

Para probar el sistema, se realiza un análisis con los datos del nivel de $^{222}Rn$ de los Laboratorios Subterráneos de Canfranc (LSC) tomados durante un periodo de cinco años -desde julio de 2013 hasta junio de 2018-, a modo de caso de estudio. Tal como se explica en [21], la predicción de los niveles de $^{222}Rn$ en el ambiente tiene implicaciones relevantes para la calidad de los experimentos realizados en el laboratorio. Mediante la optimización automática de hiperparámetros se pretende mejorar la predicción de los valores futuros de $^{222}Rn$.

La arquitectura base del modelo sobre el cual se optimizan los hiperparámetros está compuesta por una primera parte correspondiente a una descomposición de la serie temporal (mediciones históricas del $^{222}Rn$) utilizando Descomposición Estacional de Series Temporales mediante Loess ("STL", por sus siglas en inglés); un método que descompone una serie temporal en tres componentes básicas, a modo de preprocesado de la serie temporal. Además, una segunda parte correspondiente a una Red Neuronal Convolucional ("CNN", por sus siglas en inglés) que toma por entrada las tres componentes de la serie temporal y predice su comportamiento futuro. Una Red Neuronal Convolucional es un tipo específico de Red Neuronal Artificial capaz de aprender características relevantes a diferentes niveles, a partir de las entradas proporcionadas.

La arquitectura *STL + CNN* ha sido probada en estudios previos y aplicada en análisis de series temporales [22]. Sin embargo, sus hiperparámetros necesitan ser cuidadosamente ajustados para obtener predicciones precisas, por lo cual sirve como un buen punto de referencia para comprobar experimentalmente la efectividad de PBIL en la optimización de hiperparámetros.

El documento está organizado de la siguiente manera: La sección 2 da una descripción de las diferentes técnicas de utilizadas; PBIL como método de optimización, STL para descomponer series temporales y CNN como modelo predictivo. En la sección 3 se repasan los diversos enfoques desde los que se ha abordado previamente la optimización automática de hiperparámetros. En la sección 4 se expone el algoritmo propuesto para optimizar hiperparámetros de modelos de aprendizaje automático mediante PBIL. En la sección 5 se dan los detalles del caso de estudio con el que se verificó la eficacia experimental de método expuesto. En la sección 6 se presentan los resultados obtenidos a partir del caso de estudio, y, finalmente en la sección 7, se presentan las conclusiones y propuestas generadas para trabajos futuros.

# 2 Técnicas utilizadas

En el presente capítulo se exponen las técnicas que han sido utilizadas durante la realización del proyecto. Por un lado se expone el funcionamiento del algoritmo PBIL, mediante el cual se realiza la optimización de hiperparámetros. Y, por otro lado, se describe la Descomposición STL y los fundamentos de las Redes Convolucionales; componentes del modelo de aprendizaje automático implementado.

## 2.1 Descomposición STL

La Descomposición Estacional y de Tendencias mediante Loess ("STL" por sus siglas en ingles) es un método versátil y robusto para descomponer series temporales mediante la utilización de Loess; un método para la estimación de relaciones no lineales [8]. La idea principal de la descomposición es, que, las series temporales pueden ser separadas en tres componentes fundamentales: una tendencia $T_t$, una componente estacional $S_t$ y una componente aleatoria $R_t$. De tal forma que una serie temporal $Y$ en un instante de tiempo $t$ se puede describir como: $Y_t = T_t + S_t + R_t$. STL es un algoritmo iterativo que progresivamente suaviza y mejora las estimaciones de la componente estacional y la tendencia, siendo resistente a observaciones extremas.

Las componentes fundamentales que se obtienen de la descomposición STL se pueden definir de la siguiente manera:

- *Componente de tendencia:* La tendencia subyacente de las métricas. Es decir, el patrón existente cuando los datos muestran subidas y bajadas que no responden a un período fijo.

- *Componente de estacionalidad:* Patrones que se repiten con un período de tiempo fijo (trimestres, meses, días de la semana). La componente posee una media nula.

- *Componente aleatoria:* La componente aleatoria, o "ruido", es decir, el residuo de las serie temporal original después de que se eliminan las series estacionales y de tendencia: $R_t = Y_t - (T_t + S_t)$.

La clave del enfoque STL es el suavizado mediante Loess. El suavizado de datos consiste en realizar una transformación de los mismos de manera que la serie resultante presente menos fluctuaciones que la original. Para un conjunto de mediciones $x_i$, Loess proporciona una estimación suavizada $g(x)$ en todos los valores de $x$. El concepto de Loess será extendido más adelante.

Para separar una serie temporal en sus tres componentes, el algoritmo de STL ejecuta dos bucles:

- Un bucle externo donde se asignan coeficientes o "pesos" de robustez a cada punto en los datos, dependiendo del tamaño del resto. Esto permite reducir o eliminar los efectos de los valores atípicos.

- Un bucle interno en el que el componente de tendencia y el componente estacional se actualizan de forma iterativa mediante el suavizado Loess.

El algoritmo puede configurarse para ejecutarse durante un número fijo de iteraciones para cada bucle, o puede configurarse para ejecutarse hasta que se cumpla un criterio de convergencia específico.

La serie temporal original se divide en subseries de ciclos (por ejemplo, si se trata de datos mensuales con una temporalidad anual, entonces habrá 12 subseries: todo enero será una serie temporal, todo febrero será otra, y así sucesivamente).

Las subseries se suavizan y luego se pasan por un *filtro de paso bajo*. La componente estacional es, entonces, la subserie suavizada menos el resultado del filtro de paso bajo. La componente estacional se resta de los datos en bruto. El resultado es suavizado mediante loess, y se obtiene la componente de tendencia. Lo que queda es la componente aleatoria, es decir, el resto.

En la Figura 2.1 se presenta un ejemplo de descomposición STL ejecutada sobre una serie temporal.

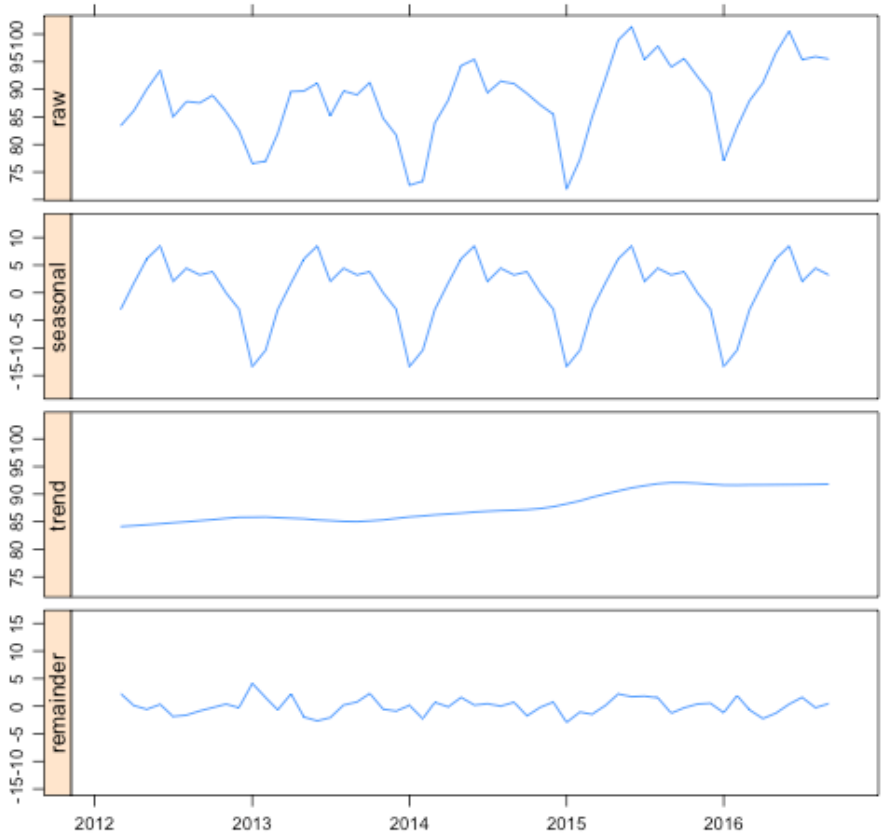



**Figura 2.1** Descomposición STL sobre una serie temporal de ejemplo..

A continuación se explica de manera formal el algoritmo seguido:

**1.** Inicializar componente de tendencia como: $T_v^0 = 0$ y $R_v^0 = 0$, para $v = 1,...,N$.

**2.** Bucle externo.- Calcular los pesos de robustez:

- Inicializar $\rho_v = 1$
- Calcular $R_v$
- Calcular pesos de robustez: $\rho_v = B(\frac{|R_v|}{h})$, donde $h = 6 * mediana(|R_v|)$ y $B$ es la función de pesos *bi-square*[1].

**3.** Bucle interno.- Calcular iterativamente la componente de tendencia y la componente estacional:

[1] Función bi-square: $B(u) = (1-u^2)^2$ para $0 \leq u \leq 1$ y $B(u) = 0$ para todos los demás $u$

- Cálculo de serie temporal sin tendencia: $Y_v - T_v^{(k)}$ donde $k$ es el número de iteración. Si el valor observado $Y_v$ es faltante, o "missing", entonces el valor de la serie temporal sin tendencia también será missing.

- Suavizado de subseries: La serie temporal sin tendencia se divide en subseries de ciclo, tal como se explicó previamente, y cada subserie se suaviza mediante Loess. Los valores suavizados producen una serie temporal correspondiente a una componente de tendencia provisional: $C^{k+1}$.

- Filtro de paso bajo: El filtro de paso bajo aplicado a $C_{k+1}$ produce $L_{k+1}$. Este filtro es la aplicación de dos medias móviles[2] con latencia 3, seguido por un suavizado loess.

- Calculo de la serie temporal suavizada sin tendencia: $S^{k+1} = C^{k+1} - L^{k+1}$ Corresponde a k+1-ésima estimación de la componente estacional. Es importante destacar que el filtro de paso bajo hace que la media de esta serie temporal sea casi nula.

- Calculo de serie temporal sin estacionalidad: $Y - S^{k+1}$.

- Suavizado de tendencia: Se suaviza la serie temporal sin estacionalidad mediante Loess. Dando como resultados $T^{k+1}$, es decir, la k+1-ésima estimación de la componente de tendencia.

### 2.1.1 Loess

El método fundamental del que depende la descomposición STL es el Suavizado Estimado Localmente de Gráficos de Dispersión ("Loess", por sus siglas en inglés). Loess es un método de construcción de modelos que se basa en métodos clásicos, como la regresión de mínimos cuadrados. Como su nombre lo indica, fue desarrollado inicialmente para el suavizado de diagramas de dispersión y es útil en situaciones en las que los procedimientos clásicos no funcionan bien o no se pueden aplicar de manera efectiva sin trabajo excesivo. Loess combina gran parte de la simplicidad de la regresión lineal de mínimos cuadrados con la flexibilidad de la regresión no lineal. Esto se logra ajustando modelos simples a subconjuntos localizados de los datos para construir una función que describa la parte determinista de los datos, punto por punto [11].

Si se está utilizando muestras de datos generados a partir de un fenómeno físico, es muy probable que exista ruido. El ruido puede ser añadido a la señal por el sensor que la

[2] La media móvil es calculada mediante el promedio de un subconjunto de los datos previos.

mide, o puede ser inherente a la naturaleza estocástica del fenómeno del que proceden los datos.

En la Figura 2.2 se puede observar un conjunto de puntos (en azul) que componen una señal ruidosa de ejemplo, y en rojo se aprecia su señal principal separada del ruido.

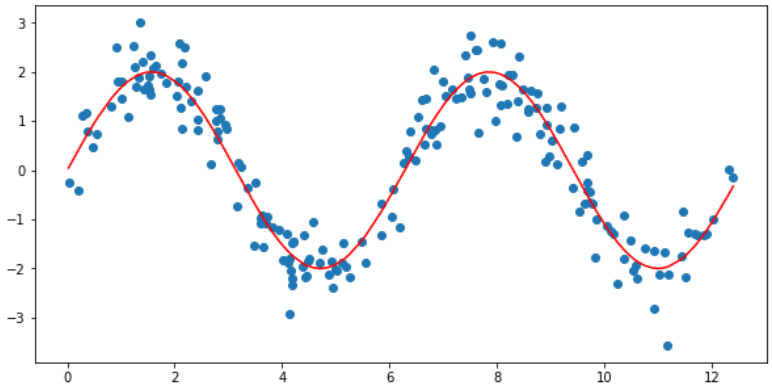



**Figura 2.2** Ejemplo de señal ruidosa, y su correspondiente señal subyacente.

Para recuperar la señal principal, Loess realiza el siguiente proceso:

Para cada valor de $x$, se estima el valor de $f(x)$ utilizando sus valores vecinos muestreados (conocidos). De forma similar al algoritmo k-Nearest Neighbor[3], donde k, el tamaño de la ventana, es un parámetro ajustable, y, en este caso particular, determinará la suavidad de la estimación resultante. Valores grandes de k resultarán en un sesgo más alto y los valores más bajos inducirán una mayor varianza.

El primer paso es recopilar el valor de $x$ para el cual se quiere estimar $y$. Se llamarán a estos $x'$ e $y'$. Alimentando el algoritmo Loess con $x'$, y usando los valores muestreados $x$ e $y$, se obtendrá una estimación $y'$. En este sentido, Loess necesita utilizar todo el conjunto de datos para la estimación.

Una vez se tiene $x'$, se debe encontrar sus $k$ vecinos más cercanos utilizando la distancia euclídea[4]. Se llamará $D$ al conjunto ordenado resultante.

El siguiente paso convierte el conjunto $D$ de $k$ distancias en un conjunto ordenado $W$ que contiene los pesos que se utilizarán más adelante en el proceso de regresión

[3] KNN es un algoritmo que busca en las observaciones más cercanas a la que se está tratando de predecir y clasifica el punto de interés basado en la mayoría de datos que le rodean, se utiliza una variable $k$ que representa el número de puntos vecinos que se tomará en cuenta para clasificar cada grupo

[4] Ecuación de la distancia euclídea entre dos puntos: $d_E(P_1,P_2) = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

lineal. Estos pesos se calculan utilizando una función de pesos especializada que asigna importancia a cada uno de los $k$ vecinos de $x$ según su distancia a $x$'.

Las pesos se calculan mediante la *función tricúbica*:

$$w(x) = \begin{cases} (1-|x|^3)^3 & |x| < 1 \\ 0 & |x| \geq 1 \end{cases} \tag{2.1}$$

Esta función devuelve valores positivos solo cuando $x$ está entre -1 y 1 . Fuera de este intervalo, la función es cero. En la Figura 2.3 se puede observar la apariencia de la función.

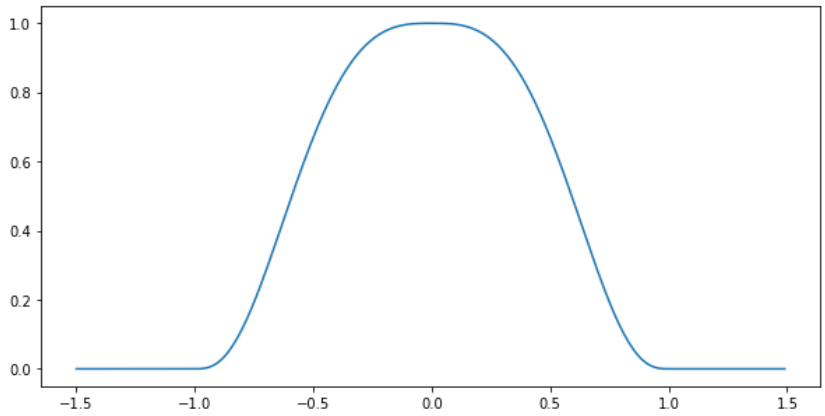



**Figura 2.3** Gráfica de la función tricúbica.

Debido a que esta función sólo tiene resultados positivos para $-1 < x < 1$, se debe normalizar la distancia dividiéndola para el valor máximo observado en $D$. Concretamente:

$$w(x) = \begin{cases} (1-\left|\frac{d(x,x')}{max_i d(x_i,x')}\right|^3)^3 & |x| < 1, x_i \in D \\ 0 & |x| \geq 1 \end{cases} \tag{2.2}$$

Donde, $d(x,x')$ es la distancia entre $x$; uno de los k vecinos más cercanos, y $x$'.

El efecto de la normalización es que las distancias más grandes se asocian con pesos más bajos. En un extremo, el punto correspondiente a la distancia máxima tendrá un peso de 0, y el punto con la distancia cero tendrá un peso de 1. Así se consigue dar mayor importancia a los datos de entrenamiento más cercanos de donde queremos que se calcule la predicción.

Hecho esto se puede calcular la estimación usando regresión lineal ponderada que se entrena con los valores $x$ de $D$, y sus valores de $y$ correspondientes.

Para cada punto que nos proponemos estimar ($x'$), el algoritmo Loess debe configurar un modelo de regresión lineal que calculará la salida correspondiente ($y'$), utilizando los $k$ vecinos más cercanos de $x'$ y un conjunto de pesos que califiquen su importancia.

Para la configuración de un modelo de regresión lineal ponderada en primer lugar se calcula un vector de parámetros lineales ($\beta$):

$$\beta = (X^T W X)^{-1} X^T W Y \tag{2.3}$$

Donde, la matriz W tiene todos los pesos calculados en diagonal, y todos los demás elementos establecidos en cero. X es la matriz que contiene todas las x observaciones, organizadas de la siguiente manera:

$$\begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_{n-1}^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_{n-1}^{(2)} \\ \dots & & & & \\ 1 & x_1^{(m)} & x_2^{(m)} & \dots & x_{n-1}^{(m)} \end{bmatrix}$$

Una vez se tiene el vector $\beta$, los nuevos valores de $y$ se pueden calcular utilizando la siguiente ecuación:

$$\hat{y} = \beta^T X \tag{2.4}$$

## 2.2 Population-Based Incremental Learning

El aprendizaje incremental basado en población ("PBIL", por sus siglas en ingles) es un método de optimización que combina algoritmos evolutivos con aprendizaje competitivo [1]. En un algoritmo evolutivo, una población de soluciones candidatas para un problema de optimización evoluciona hacia soluciones mejores durante varias iteraciones. Mientras que el aprendizaje competitivo es un método de aprendizaje comunmente utilizado para redes neuronales artificiales, en el cual los nodos compiten por el *derecho* de responder a un conjunto de los datos de entrada, es decir, habrán nodos que tengan una mayor relevancia que otros en el proceso de convertir los datos de entrada en salidas.

PBIL pertenece a los llamados algoritmos de estimación de distribución ("EDAs", por sus siglas en ingles). La principal diferencia con los algoritmos evolutivos estándar es que los EDAs no crean una población de soluciones a partir de la generación anterior utilizando mutaciones. En su lugar, los EDAs consideran la información global de toda la población para construir una distribución probabilística que se actualiza paso a paso siguiendo reglas basadas en el aprendizaje competitivo, las cuales se expondrán más adelante.

El núcleo de PBIL es un vector de probabilidades ($p$) en el cual cada elemento es un valor real en el rango [0,1]. A partir del vector $p$ se genera una población de individuos en cada iteración (generación) del algoritmo. Habiendo previamente seleccionado una métrica para determinar la calidad de un individuo (denominada "función de *fitness*"), se clasifican los mejores y los peores individuos de su respectiva generación. Posteriormente se actualiza el vector $p$ premiando las características de los mejores individuos y penalizando las características de los peores individuos. De esta manera, con el paso de cada generación aumenta la probabilidad de obtener mejores características y por lo tanto un mejor valor de fitness. Al final de cada generación, siguiendo una determinada probabilidad se realizan mutaciones a los elementos del vector. De esta manera, el algoritmo confluye hacia un mínimo local con cada generación [24].

De manera detallada, los pasos seguidos por PBIL son los siguientes:

**1.** Inicializar el vector de probabilidades $v$ con tamaño $n$. Cada posición con un valor de 0.5

**2.** Bucle generaciones.-

**2.**1. Bucle población.- Crear población individuo por individuo.

- Individuo $k_i$.- Siendo "random" la representación de un número generado aleatoriamente en un rango especificado. Se recorre un índice $j$ que va desde *0* hasta $n$:

  Si $random[0,1] > v_j$ entonces $k_{i,j} = 1$, sino $k_{i,j} = 0$.

- Evaluar $k_i$ mediante la función de fitness.

**2.**2. Clasificar los mejores individuos ($m_i$) y los peores individuos ($p_i$) de la generación.

**2.**3. Actualización del vector $v$ basado en cada uno de los individuos más aptos: $v_j = v_j + 2 \cdot (m_{i,j} - 0.5) \cdot LR$

**2.**4. Actualización del vector $v$ basado en cada uno de los individuos menos aptos: $v_j = v_j - 2 \cdot (p_{i,j} - 0.5) \cdot NLR$

**2.**5. Bucle mutación.- Recorrer todos los indices $j$ del vector $v$:

- Si $random(0,1] < MP$ entonces:

$$v_j = v_j \cdot (1.0 - MS) + random[0 \text{ o } 1] \cdot MS$$

Siendo,

- $LR$: tasa de aprendizaje utilizada para recomenzar las características de los mejores individuos.
- $NLR$: tasa de aprendizaje utilizada para penalizar las características de los peores individuos.
- $MP$: probabilidad de que ocurra una mutación en cada posición del vector $v$.
- $MS$: valor de influencia de la mutación en el vector de probabilidad.

## 2.3 Redes Neuronales Convolucionales

En el presente proyecto se utilizó un tipo de red neuronal denominada red neuronal convolucional. Para poder explicar su funcionamiento es necesario conocer ciertas definiciones previas, las cuales serán expuestas en la presente subsección.

### 2.3.1 Inteligencia artificial

Para poder entender el concepto de Inteligencia Artificial primero es necesario comprender, hasta cierto punto, el funcionamiento del cerebro. Este trabaja como un sistema de reconocimiento de patrones enormemente complejo. Posee en promedio 86 mil millones de neuronas [14] interconectadas mediante enlaces sinápticos. La activación de estas neuronas sigue procesos biofísicos que nos permiten realizar funciones complejas.

En los procesos cognitivos existe una separación de escalas entre la dinámica a nivel neuronal y la aparición del pensamiento abstracto. Esto representa una separación entre el “hardware” (neuronas y sinapsis) y el “software” (conciencia, operaciones abstractas) de nuestro cerebro, y es el punto de partida para concebir una representación de dichos procesos de forma digital y apartada de un entorno biológico [2].

A partir de estos principios se puede continuar el análisis indicando que la Inteligencia Artificial consiste en la manipulación de esquemas abstractos y expresiones lógicas mediante sistemas artificiales [16]. También se la puede definir como la disciplina que estudia la forma de diseñar procesos que exhiban características que comúnmente se asocian con el comportamiento humano inteligente [20].

### 2.3.2 Aprendizaje automático

Dentro del campo de la Inteligencia Artificial se encuentra el Aprendizaje Automático, rama que se enfoca en crear algoritmos capaces de generalizar comportamientos y reconocer patrones a partir de información suministrada en forma de ejemplos. De manera más concreta, busca generalizar un enunciado a partir de enunciados de casos particulares [5].

Se puede considerar que el aprendizaje automático intenta extraer conocimiento sobre algunas propiedades no observadas de un *objeto* basándose en propiedades que sí han sido observadas de ese mismo *objeto*.

Los diferentes algoritmos de Aprendizaje Automático se pueden agrupar dependiendo de su enfoque, el tipo de datos que reciben y generan, y el tipo de problema que deben resolver. Las tres categorías principales son:

- *Aprendizaje supervisado:* Abarca técnicas utilizadas para deducir una función a partir de datos de entrenamiento, en los cuales se presentan diferentes datos de entrada y los resultados esperados para esos datos. El objetivo de este tipo de aprendizaje es poder predecir el resultado correspondiente a cualquier nueva entrada, basándose en el conocimiento extraído de los datos de entrenamiento.

- *Aprendizaje no supervisado:* Abarca técnicas donde el modelo se ajusta de acuerdo a las observaciones, sin datos que especifiquen las salidas esperadas para los distintos datos de entrada, por lo que el agrupamiento se realiza en base a distintos criterios impuestos por el método.
  Comúnmente es utilizada para encontrar patrones que permitan separar y clasificar datos en diferentes grupos en función de los atributos de los ejemplos.

- *Aprendizaje por refuerzo:* Abarca un conjunto de técnicas utilizadas para aprender a través de ensayo y error. Es necesaria una comunicación que vaya desde el sistema al entorno y del entorno al sistema para poder recibir retroalimentación sobre las acciones realizadas, para, de esta manera, reforzar las acciones que reciban una respuesta positiva. En principio las respuestas positivas se reciben cuando una acción realizada ha acercado al sistema a su objetivo.

Dentro de los problemas que se tratan con aprendizaje automático se encuentran varios grupos, que se diferencian por el tipo de objeto que se intenta predecir, tenemos, por ejemplo:

- *Clasificación:* Se busca predecir la clase a la que pertenece un conjunto de datos. En esencia, los sistemas diseñados para clasificación deben retornar valores discretos. Un ejemplo son los sistemas diseñados para extraer los objetos presentes en una imagen.

- *Regresión:* La salida de estos sistemas es un valor continuo, el cual representa el atributo objetivo.
  Un ejemplo de esta categoría son los sistemas diseñados para predecir el precio de un inmueble a partir de la información de inmuebles previamente vendidos.

### 2.3.3 Redes neuronales

Las redes neuronales artificiales son un modelo computacional de aprendizaje automático basado en el modelo y funcionamiento de las redes neuronales biológicas.

En esencia representa una combinación de funciones no lineales para intentar aproximar alguna función, y haciendo uso de un conjunto de entrenamiento, este tipo de modelos son capaces de modificar los pesos involucrados en ella para minimizar el error entre la función implementada y la que se desea aproximar.

La amplia popularidad de las redes neuronales está asociada a sus numerosas ventajas, entre sus características están:

- Permiten el manejo de información redundante.

- Adquieren conocimiento a través de la experiencia, y al igual que en el cerebro, este conocimiento se almacena en el peso relativo de las conexiones interneuronales.

- Tienen un comportamiento altamente no lineal, lo que permite procesar información de fenómenos no lineales.

- Poseen una alta plasticidad y adaptabilidad, lo que les permite cambiar su comportamiento de forma dinámica junto con el medio.

- Brindan cierta independencia entre la complejidad del problema y la dimensionalidad de la red.

- Tienen gran tolerancia ante variables de entrada no relevantes y ruido, de esta forma, si una parte de la red no maneja información correcta la red en conjunto funcionará como un sistema robusto.

Las redes neuronales artificiales se pueden utilizar como método para resolver un problema siempre y cuando no sea necesario obtener una justificación del resultado obtenido, esto es debido a que son un modelo de aprendizaje de caja negra.

Cada nodo o "neurona" de la red está compuesta por una serie de entradas ($x_i$), cada una con su correspondiente peso ($w_i$) los cuales serán optimizados durante el proceso de aprendizaje y, por último, una salida $x_f$. Tal como se puede observar en la Figura 2.4.

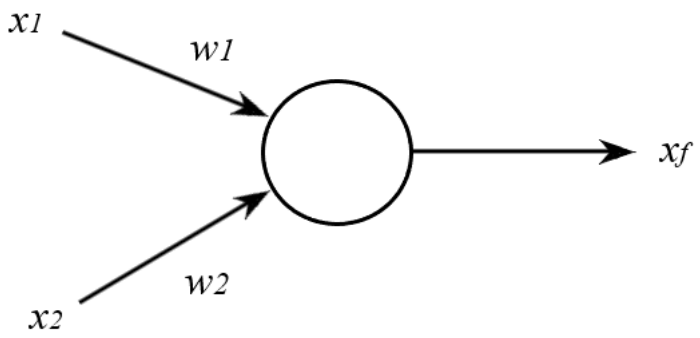



**Figura 2.4** Esquema de una neurona simple.

Las neuronas artificiales individuales intentan replicar el comportamiento de las neuronas en los cerebros biológicos, y, al igual que lo que se cree sobre éstos, las neuronas artificiales están organizadas en capas de manera jerárquica. Cada neurona está conectada con varias otras y el enlace entre cada una de ellas puede aumentar o disminuir el estado de activación de la neurona siguiente. En la Figura 2.5 se puede observar la estructura de una red neuronal sencilla.

Las neuronas se agrupan de manera jerárquica en unidades estructurales llamadas capas, que en conjunto conforman la red neuronal. Hay tres tipos de capas en una red neuronal:

- *Capa de entrada:* Se compone de neuronas que reciben datos del entorno.
- *Capas ocultas:* Se compone de neuronas ubicadas entre la capa de entrada y capa de salida, proporciona grados de libertad a la red neuronal con los cuales se le permite modelar de mejor manera el entorno o comportamiento deseado.

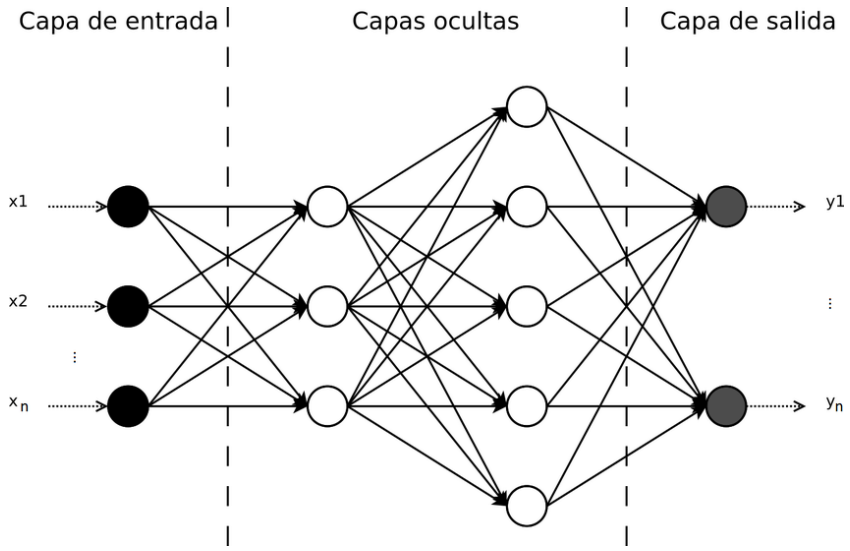



**Figura 2.5** Esquema de una Red Neuronal.

- *Capa de salida:* Se compone de neuronas encargadas de proporcionar la respuesta de la red neuronal.

Cada neurona de una red neuronal toma los valores que recibe de entrada, y los convierte a una salida que pasa a la siguiente capa de neuronas o a la salida final de la red, esto se conoce como *propagación hacia delante*. Para calcular la salida se utilizan dos funciones denominadas función de propagación y función de activación.

La función de propagación ($h_i$) para la i-ésima neurona artificial se define a partir del conjunto de entradas $x_i$ y los pesos sinápticos $w_{ij}$, con $j = 1,...,n$:

$$h_i(x_1,...,x_n,w_{i,1},...,w_{i,n})$$

La función de propagación más comúnmente utilizada consiste en la sumatoria de todas las entradas multiplicadas por los pesos de las conexiones, más un valor de sesgo o "bias", el cual es un parámetro que proporciona un grado de libertad adicional al modelo y evita salidas no deseadas cuando las entradas son cero.

El valor bias comúnmente es 1 o -1 y es multiplicado por un peso $w_b$ el cual será determinado por la red neuronal durante el entrenamiento, para más tarde añadirse a la función de propagación. El valor bias será representado con la letra "b", con lo cual la

función de propagación es la siguiente:

$$h_i(x_1,...,x_n,w_{i,1},...,w_{i,n}) = b \cdot w_b + \sum_{j=1}^{n} w_{ij}x_j \tag{2.5}$$

Por otro lado, la función de activación ($f_i$) puede activar o desactivar el resultado de la función de propagación en una neurona (darle un valor alto o bajo). Su resultado es la salida de la neurona ($y_i$).

$$y_i = f_i(h_i) = f_i(b \cdot w_b + \sum_{j=1}^{n} w_{ij}x_j) \tag{2.6}$$

La utilización de funciones de activación no lineales permiten que las redes puedan resolver problemas no triviales, necesitando solamente un pequeño número de nodos.

En la Tabla 2.1 se pueden observar las funciones de activación más comunes.

La estructura de las capas de la red neuronal define la arquitectura de la misma. De esta forma podemos diferenciar dos tipos de redes neuronales:

- *Redes monocapa:* Compuestas por una única capa de neuronas. Se utilizan típicamente en en tareas relacionadas en lo que se conoce como autoasociación. Por ejemplo, regenerar datos de entrada incompletos o distorsionados.

- *Redes multicapa:* Compuesta por varias capas de neuronas. En ellas una capa recibe las señales de entrada de una capa cercana y envía señales de salida a una capa posterior.

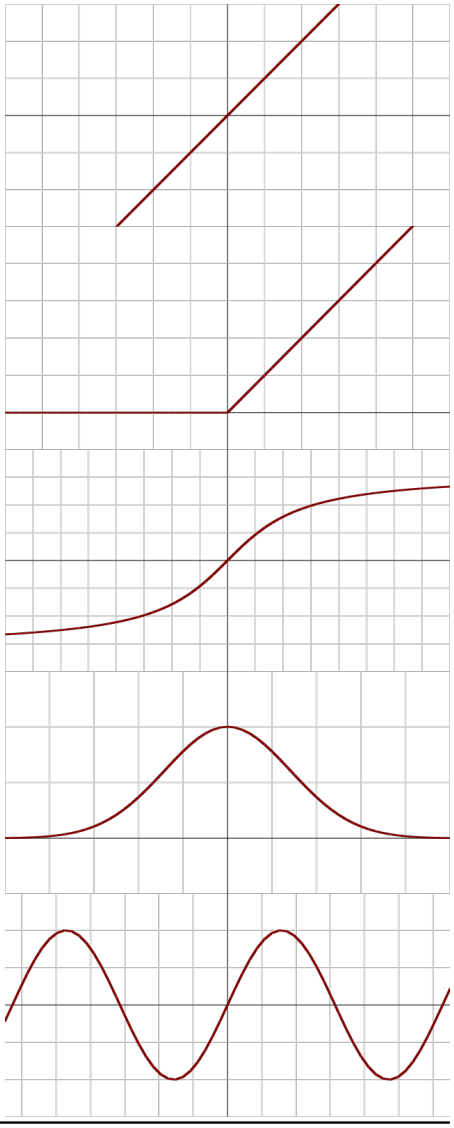
Así mismo, tomando en cuenta la dirección del flujo de datos se pueden diferenciar dos tipos de redes neuronales:

- *Redes unidireccionales:* La información circula en un único sentido a lo largo de la red neuronal, partiendo de la capa de entrada hasta la capa de salida.

- *Redes recurrentes:* La información puede circular entre las distintas capas de neuronas en cualquier sentido, sin excluir las capas de entrada y salida, puede presentar bucles, y, por lo tanto, retroalimentación. Existen modelos que aprovechan esta retroalimentación para desempeñar las funciones de una memoria básica, debido a que se pueden almacenar los estados anteriores más cercanos en el tiempo.

La interacción entre las diversas neuronas de una red produce respuestas en la capa de salida a partir de unos datos dados en la capa de entrada. Durante la etapa de entrenamiento, la red optimizará los valores de sus pesos para aproximar la función que relacione los datos de entrada con los datos de salida esperados.

**Tabla 2.1** Funciones de activación más comunes.

| | Función | Rango | Gráfica |
|---|---|---|---|
| Identidad | $y = x$ | $]-\infty,+\infty[$ |  |
| Rectificador (ReLu) | $y = \begin{cases} 0 & si \quad x \leq 0 \\ x & si \quad x > 0 \end{cases}$ | $[0,+\infty[$ | |
| Sigmoide | $y = \frac{1}{1+e^{-x}}$<br>$y = tgh(x)$ | $[0,+1]$<br>$[-1,+1]$ | |
| Gaussiana | $y = Ae^{-Bx^2}$ | $[0,+1]$ | |
| Sinusoidal | $y = Asen(\omega x + \varphi)$ | $[-1,+1]$ | |

### 2.3.4 Métodos de optimización de pesos

Los pesos de una red neuronal comúnmente se inicializan de manera aleatoria, y a medida que avanza el proceso de entrenamiento los pesos van optimizándose para mejorar el rendimiento de la red en su correspondiente tarea. El rendimiento de una red viene dado por una función de pérdida, encarga de cuantificar la distancia entre las salidas que esperamos que la red sea capaz de generar y las salidas reales generadas.

La función de pérdida más comúnmente utilizadas es el error cuadrático medio ("MSE" por sus siglas en inglés) el cual viene dado por la siguiente ecuación:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - p_i)^2 \tag{2.7}$$

Siendo:

- $N$: el número total de ejemplos evaluados.
- $y_i$: la i-ésima salida esperada.
- $p_i$: la i-ésima salida calculada.

Por otro lado, para problemas cuyas salidas se encuentran en el rango [0,1], es común utilizar la entropía binaria cruzada como función de pérdida:

$$L = -\frac{1}{N}\sum_{i=1}^{N} y_i log(p_i) \tag{2.8}$$

Siendo:

- $N$: el número total de ejemplos evaluados.
- $y_i \in [0,1]$: la i-ésima salida esperada.
- $p_i \in [0,1]$: la la i-ésima salida calculada.

Existe una gran variedad de funciones de pérdida que pueden ser aplicadas, y cuya utilidad dependerá de los datos manejados.

Una vez definida la función de pérdida, se puede realizar un proceso de entrenamiento, que tiene por objetivo ajustar los pesos de la red para mejorar el desempeño del modelo.

Entre los métodos de ajuste de pesos se pueden diferenciar los métodos de tipo gradiente y los métodos basados en algoritmos genéticos.

#### Métodos basados en algoritmos genéticos

Los métodos basados en algoritmos genéticos son técnicas de resolución de problemas inspirados en la naturaleza, basados en el principio darwiniano de supervivencia y reproducción de los individuos más aptos. Consisten en la creación de un determinado número de modelos (individuos) iniciales, que son optimizados a lo largo de varias generaciones aplicando "mutaciones" en sus parámetros, y evaluando el error en cada uno de ellos. Los individuos con menor error tienen mayor probabilidad de convertirse en la base de nuevos modelos, mientras que los individuos con mayor error desaparecen. Este tipo de métodos permiten un gran paralelismo para lograr encontrar una solución.

Para implementar un algoritmo genético es necesario en primer lugar generar una población inicial (generalmente de manera aleatoria), después de esto se debe evaluar la aptitud (fitness) de cada individuo. Un operador de selección será el encargado de decidir qué individuos contribuirán en la formación de la siguiente generación, simulando la selección natural al tomar únicamente los individuos con mayor aptitud de cada generación.

A continuación, dependiendo del método, se pueden aplicar operadores genéticos como el "cruce" cuya función es tomar aleatoriamente dos individuos que tengan un buen rendimiento para entrelazarlos intercambiando partes entre ellos, creando así nuevos descendientes.

Por último, se puede aplicar un operador de mutación que modifica algunas características de algunos individuos posibilitando así la búsqueda de soluciones alternativas a las que no se hubiera llegado sin este operador. El operador de mutación usualmente actúa con una probabilidad bastante baja y sirve para evitar los mínimos locales en la solución encontrada. El ciclo se repite mejorando generalmente los resultados obtenidos en cada generación.

Este algoritmo se conoce como el algoritmo genético canónico, sin embargo, en algoritmos particulares se pueden agregar nuevos operadores o descartar otros [4].

Los algoritmos genéticos se pueden utilizar cuando la función modelada sea no derivable, y por lo tanto no se puedan aplicar métodos de tipo gradiente. También se pueden aplicar en problemas matemáticos abstractos como el problema de la mochila, la coloración de grafos o en asuntos tangibles de ingeniería como el control de flujo en una línea de ensamble, reconocimiento y clasificación de patrones y optimización estructural. A

medida que avanza el estudio sobre este tipo de algoritmos sus campos de aplicación van en aumento.

#### Métodos de tipo gradiente

Los métodos de tipo gradiente calculan la variación de la función de pérdida al modificar los parámetros, a modo de derivada multidimensional, para así cambiar los parámetros de la red y mejorar sus resultados.

Se busca encontrar la configuración de pesos que se corresponda con el mínimo global de la función error, aunque en la mayoría de casos es suficiente encontrar un mínimo local aceptable.

De manera detallada, el principio general de estos métodos podría ser enunciado como sigue: dado un conjunto de pesos $W(t)$ para un instante de tiempo $t$ la dirección de máximo crecimiento de la función $E(W)$ en $W(t)$ viene dada por el gradiente $\nabla E(W)$. De modo que, la dirección de máxima disminución del error viene dada por la dirección contraria a la indicada por el gradiente. De esta forma es posible acercarse gradualmente al mínimo local.

$$W(t+1) = W(t) - \alpha \nabla E(W) \tag{2.9}$$

Donde $\alpha$ indica la tasa de aprendizaje, es decir, la amplitud del paso tomado en cada iteración. Si la tasa de aprendizaje es un valor demasiado pequeño el aprendizaje requerirá una gran cantidad de iteraciones, y, además, se corre el riesgo de que el algoritmo se estanque en un mínimo local que no sea suficientemente bueno. Por otro lado, si la tasa de aprendizaje es demasiado grande se pueden producir oscilaciones en torno al mínimo perseguido. En la Figura 2.6 se puede observar una representación de la manera en que un método de tipo gradiente se acerca a un error mínimo partiendo de un punto aleatorio.

El algoritmo de optimización de pesos utilizado en redes neuronales es la *Propagación hacia atrás* (Backpropagation), el cual es un método que implementa el descenso por el gradiente para optimizar los pesos de una red, mediante los siguientes pasos:

**1.** Inicializar los pesos de la red neuronal. Comúnmente se realiza de manera aleatoria.

**2.** Bucle- Para cada epoch (número de veces que se recorren todos los ejemplos):

**2.**1. Bucle- Para cada batch (subconjuntos generados aleatoriamente con los ejemplos de entrenamiento):

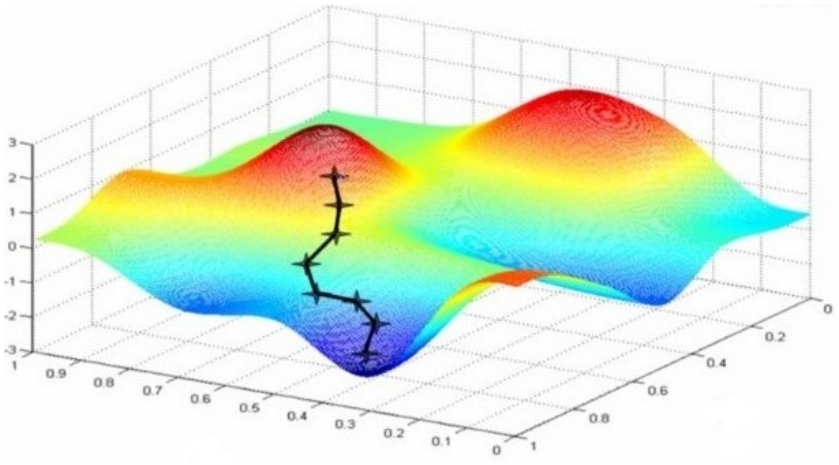


**Figura 2.6** Representación gráfica del descenso por el gradiente. Recuperado de [9].

**2.**1.1. Realizar la propagación hacia delante con las entradas de cada ejemplo y obtener la predicción $p$ de la red.

**2.**1.2. Comparar la salida obtenida con la salida esperada ($y$) y obtener el valor de pérdida.

**2.**1.3. Utilizando el promedio del error calculado en todos los ejemplos del batch, ajustar los pesos de las neuronas de la capa de salida, con la tasa de aprendizaje definida.

**2.**1.4. Repetir el proceso propagando el error (y actualizando los pesos) hacia las capas anteriores (propagación hacia atrás), hasta llegar a la primera capa de la red.

El descenso por el gradiente es, además, la base de otros optimizadores encargados de unir la función de pérdida y los pesos del modelo, actualizando los pesos en respuesta al error obtenido. Entre estos optimizadores podemos encontrar:

- *Adagrad:* Adagrad adapta la tasa de aprendizaje para cada parámetro de acuerdo con el historial de gradientes (pasos previos) para ese parámetro, a esto se le denomina *momentum*. Se realiza básicamente dividiendo el gradiente por la suma de los gradientes anteriores.
  Brinda un buen rendimiento para conjuntos de datos dispersos donde faltan muchos ejemplos de entrada. Sin embargo, el problema de Adagrad es que la tasa de aprendizaje adaptativa tiende a volverse bastante pequeña con el paso del tiempo.

- *RMSprop:* RMSprop es una versión especial de Adagrad, cuya diferencia es que divide la tasa de aprendizaje por una tasa exponencial de decaimiento especificada.

- *Adam:* Adam proviene de "adaptive moment estimation" o "estimación adaptativa del momento" en español, y es otro método que calcula una tasa de aprendizaje adaptativa para cada parámetro y utiliza una tasa de decaimiento al igual de que RMSprop. Tiene la particularidad de que usa la media móvil del gradiente, y no el gradiente en sí.

Entre las diversas tareas en las que un algoritmo de tipo gradiente puede aplicarse se puede mencionar la clasificación lineal y no lineal de una cantidad arbitraria de clases, análisis de series temporales, control de proceso, regresión lineal y no lineal, optimización de funciones, procesamiento de señales y modelado de cualquier función derivable.

### 2.3.5 Redes convolucionales

Las Redes Convolucionales son un tipo específico de Red Neuronal Artificial, en ellas cada neurona imita el comportamiento de las neuronas en la corteza visual primaria de un cerebro biológico. Sus fundamentos se basan en el Neocognitron, introducido en 1980 por Kunihiko Fukushima [12].

Debido a que su aplicación se realiza normalmente sobre matrices con dos o tres dimensiones, se utiliza mayormente para tareas de visión artificial donde se ha demostrado que ofrece resultados superiores a los obtenidos con otros modelos de redes neuronales. Sin embargo, también se ha probado su eficacia en campos tales como el modelado de series temporales y reconocimiento de voz [18].

Una Red Neuronal Convolucional ("CNN" por sus iniciales en inglés) está compuesta por una o más capas de convolución, a menudo con pasos de submuestreo intermedios (capas de reducción), y finalmente seguida por una o más capas de neuronas unidireccionales como en una red neural multicapa estándar. En una red neuronal estándar todas las neuronas están conectadas con todas las neuronas de sus capas adyacentes, sin embargo, en una red neuronal convolucional se aprovecha el patrón jerárquico de los datos y se consiguen ensamblar patrones complejos a partir patrones simples mediante conexiones locales. Siendo, de esta manera, menos complejas que una red neuronal estándar, lo cual facilita su entrenamiento.

En la Figura 2.7 se puede observar el esquema mencionado de una Red Convolucional.

En la capa de convolución se utiliza un número $n$ de filtros de convolución. Cada filtro debe tener como hiperparámetros el tamaño de su ventana (también llamada *kernel*) correspondiente, con un número de dimensiones igual al de la entrada de la capa. Es decir,

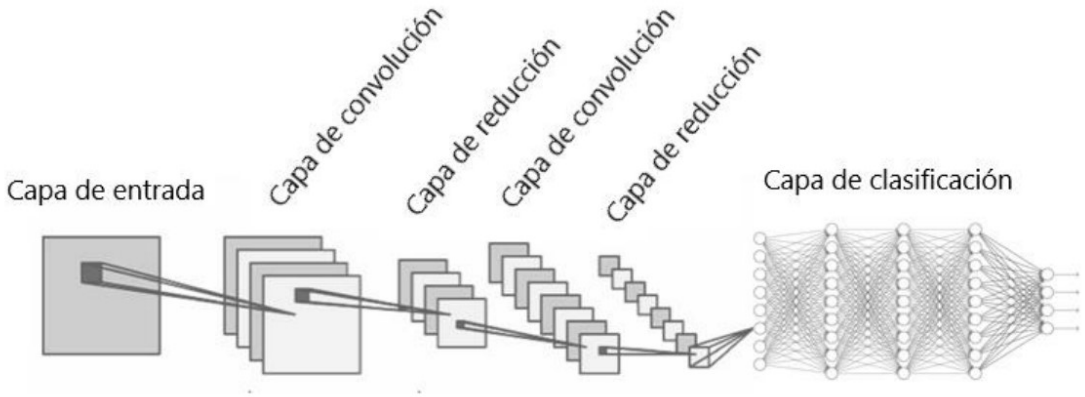



**Figura 2.7** Esquema de una Red Neuronal Convolucional.

en el caso de que la entrada de la capa sea una matriz bidimensional, se debe especificar un ancho y largo para la ventana del filtro, en cambio, en caso de que la entrada sea un vector unidimensional será necesario especificar únicamente un hiperpárametro.

Cada filtro de la capa de convolución recorre las entradas calculando el productos entre cada elemento del filtro y su elemento correspondiente en la ventana observada por el filtro en las entradas, posteriormente se suman los resultados obtenidos. De esta manera se genera elemento a elemento un mapa de características que se corresponde con los datos transformados por dicho filtro, tal como se puede observar en la Figura 2.8.

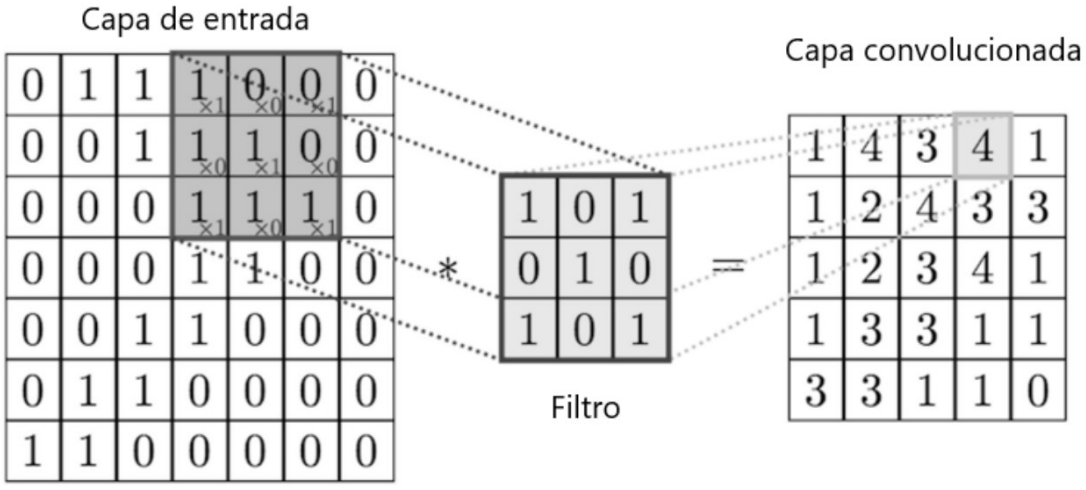



**Figura 2.8** Esquema de aplicación de filtro de convolución.

El filtro comienza en la esquina superior izquierda y se mueve de izquierda a derecha hasta analizar el ancho completo de la matriz. Continuando, salta hacia el principio (izquierda) de la imagen, debajo del punto de inicio del recorrido anterior. El proceso se repite hasta que se recorre toda la imagen.

Un desarrollo reciente [26] introduce un hiperparámetro más en las capas de convolución, llamado dilatación. Los filtros de convolución presentados hasta ahora son contiguos. Sin embargo, es posible tener filtros que tengan espacios entre cada celda, a esto se le denomina *dilatación*. Por ejemplo, en una dimensión, un filtro $w$ de tamaño 3 calcularía sobre la entrada $x$ lo siguiente:

$$w[0] \cdot x[0] + w[1] \cdot x[1] + w[2] \cdot x[2]$$

Esto representa un valor de dilatación de 0. Para una dilatación igual a 1, el filtro calcularía en cambio:

$$w[0] \cdot x[0] + w[1] \cdot x[2] + w[2] \cdot x[4]$$

En otras palabras, hay un espacio de 1 entre cada uno. Esto puede ser muy útil en algunos ajustes para usarse en conjunto con filtros de dilatación 0, ya que permite al sistema fusionar información espacial a través de las entradas de forma mucho más agresiva con menos capas. Por ejemplo, si se apilan dos capas de convolución 3x3 una encima de la otra, si no existe dilatación entonces cada neurona de la segunda capa será una función con respecto a una sección de 5x5 de la entrada de la red (diríamos que el campo receptivo efectivo de estas neuronas es 5x5). Si se aplica dilatación entonces este campo receptivo crecería mucho más rápido.

Después de obtener los mapas de características, se aplica una función de activación (comúnmente ReLu) sobre ellos, para después pasar a la siguiente capa de la red.

El objetivo de la operación de convolución es extraer las características de alto nivel. Convencionalmente, las primeras capas de convolución se encargan de capturar las características de bajo nivel tales como, en caso de que la entrada sea una imagen, los bordes, color, orientación, entre otras. A medida que se añaden capas de convolución la arquitectura se adapta a las características de alto nivel, lo que nos da una red capaz de distinguir estructuras complejas, tales como objetos dentro de una imagen, con un nivel de precisión elevado.

Por otro lado, la capa de reducción, también llamada capa de *pooling*, de manera similar a la capa convolucional, es responsable de reducir el tamaño espacial de las entradas. Esto permite disminuir la potencia computacional requerida para procesar los datos. Además, es útil para extraer los rasgos dominantes, manteniendo la efectividad del modelo. Esto permite que la red cuente con cierta tolerancia a pequeñas perturbaciones en los datos de

entrada. Por ejemplo, si dos imágenes son casi idénticas, pero varían únicamente por un traslado horizontal de algunos pixeles, el resultado debería ser esencialmente el mismo.

Existen dos métodos de reducción, por un lado esta el *Max Pooling* y otro el *Average Pooling*. Max Pooling devuelve el valor máximo de la porción de los datos de entrada cubierta por el kernel. Por otro lado, el Average Pooling devuelve la media de todos los valores de la parte de los datos de entrada cubierta por el Kernel.

En la Figura 2.9 se puede observar un ejemplo de la aplicación de Max Pooling sobre una entrada bidimensional.

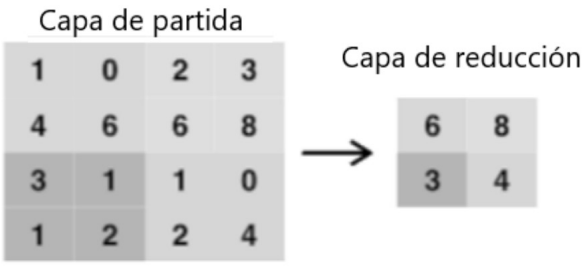



**Figura 2.9** Esquema de aplicación de filtro de reducción.

Una vez pasadas las capas de convolución y reducción, se redimensionan las salidas obtenidas a un solo vector unidimensional (proceso al que se le denomina "flattenning") que sirve de entrada para una red neuronal unidireccional simple. Tal como se puede observar en la Figura 2.10.

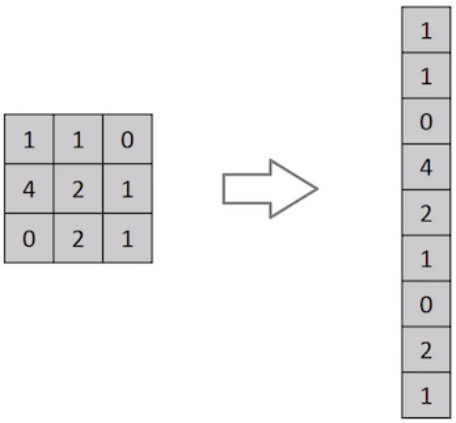



**Figura 2.10** Ejemplo de Flatenning de una matriz.

La utilización de una red neuronal estándar permite aprender combinaciones no lineales de las características de alto nivel representadas por la salida de las capas convolucionales.

En caso de que la tarea a realizar sea de clasificación entonces el número de salidas de la red debe ser igual al número de categorías a clasificar. La salida que tenga el mayor valor representará la categoría predicha por el sistema [15].

# 3 Estado del arte

El problema de la optimización de hiperparámetros en aprendizaje automático se ha abordado desde diversos puntos de vista. En la presente sección se hace un repaso de los enfoques más populares.

## 3.1 Barrido paramétrico

Uno de los enfoques más sencillos para la optimización de hiperparámetros es el barrido paramétrico, o también llamado "grid search" o "parametric sweep", consistente en probar mediante fuerza bruta todas las posibilidades dentro de un grupo de valores discretos definidos manualmente.

En el caso de la optimización de hiperparámetros se entrena un modelo con cada posible combinación de valores y se evalúa los modelos generados mediante una métrica definida. El método garantiza encontrar el mínimo global en el conjunto de hiperparametros predeterminado. Sin embargo, el mayor problema de este enfoque es el alto poder de calculo requerido, y su correspondiente tiempo de ejecución.

## 3.2 Búsqueda aleatoria

Una evolución del barrido paramétrico es la búsqueda aleatoria, la cual muestrea valores seleccionados aleatoriamente del dominio de los hiperparámetros. Al igual que el barrido paramétrico, la búsqueda aleatoria puede trabajar con valores discretos, pero también es aplicable para espacios continuos. La ejecución se detiene cuando se alcanza una

condición de parada definida, como, por ejemplo, un determinado número de modelos generados.

El enfoque es expuesto por Bergstra y Bengio en [3]. Los autores demuestran en su estudio que la búsqueda aleatoria es más eficiente en poder de calculo requerido que un barrido paramétrico, y, a su vez, la calidad de los resultados (con respecto a la métrica definida) obtenidos es en su mayoría semejante, y, en ciertos casos, superior. Tal como se puede observar en el ejemplo de la Figura 3.1.

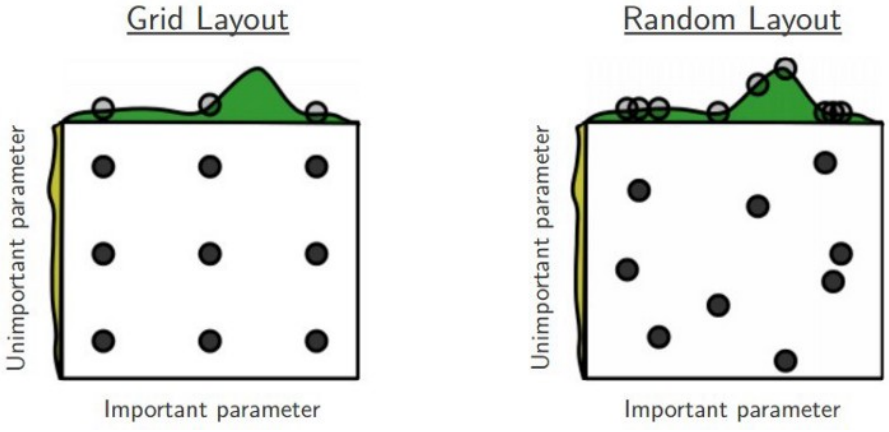



**Figura 3.1** Ventajas de utilizar búsqueda aleatoria frente al barrido paramétrico. Recuperado de [3].

## 3.3 Optimización bayesiana

La optimización bayesiana trata el ajuste de hiperparámetros como un problema de regresión. Dado un conjunto de características de entrada (los hiperparámetros), el ajuste de hiperparámetros optimiza un modelo para la métrica que se elija. Para solucionar un problema de regresión, el ajuste de hiperparámetros hace suposiciones sobre qué combinaciones de hiperparámetros tendrán probablemente los mejores resultados y entrena modelos para probar estos valores. Después de probar cada conjunto de valores de hiperparámetros, el método selecciona el siguiente conjunto de valores de hiperparámetros que va a probar en la siguiente iteración [25]. Su funcionamiento se puede describir con los siguientes pasos:

1. Selección de una medida previa en el espacio de posibles funciones de métodos de evaluación. Debido a que se desconoce la función objetivo, se trata la función como

una función aleatoria, a la cual se le especifica una distribución previa. La selección de la distribución previa permite al diseñador añadir al sistema sus creencias sobre el comportamiento de la función. A menudo se utiliza la distribución gaussiana multivariable.

2. Combinación de la distribución previa con ciertas observaciones dadas (configuraciones probadas) para obtener una nueva distribución en el objetivo, la cual será la estimación de donde reside la función verdadera.

3. Utilizar la nueva distribución para decidir dónde probar la siguiente configuración según una función de adquisición. La función de adquisición se encarga de retornar la configuración a probar en la que se conseguirá la mayor cantidad de información.

4. Evaluar la configuración seleccionada en el paso 3.

El número de ejecuciones simultáneas tiene un efecto sobre la eficacia del proceso de ajuste. Normalmente, un menor número de ejecuciones simultáneas puede provocar una mejor convergencia de muestreo, dado que el menor grado de paralelismo aumenta el número de ejecuciones que se benefician de las ejecuciones completadas previamente.

La optimización bayesiana ofrece una solución intermedia entre exploración y explotación, de modo que se minimice el número de evaluaciones:

- *Exploración:* Evaluación en los lugares en que la varianza es amplia, es decir, donde la estimación tenga mayor probabilidad de ser inexacta.

- *Explotación:* Evaluación en los lugares en que la estimación está más cerca del objetivo.

## 3.4 Optimización evolutiva

La optimización evolutiva considera ideas provenientes de algoritmos evolutivos para explorar un espacio de hiperparámetros [23]. El enfoque sigue un proceso inspirado en el concepto biológico de la evolución:

1. Crear una población inicial de soluciones aleatorias. Es decir, generar tuplas de hiperparámetros al azar.

2. Evalúar las tuplas de hiperparámetros y obtener su medida de rendimiento (por ejemplo, la precisión del modelo asociado).

3. Clasificar las tuplas de hiperparamétros con respecto a su rendimiento correspondiente.

4. Reemplazar las tuplas de hiperparámetros de peor rendimiento con nuevas tuplas de hiperparámetros generadas a través del cruce y la mutación.

5. Repetir los pasos 2 al 4 hasta que se alcance un rendimiento satisfactorio o el rendimiento del algoritmo deje de mejorar con cada iteración.

Uno de sus algoritmos más populares en esta categoría es el Covariance Matrix Adaptation Evolution Strategy (CMA-ES). En el cual, cada individuo representa un vector n-dimensional con valores reales actualizados mediante mutaciones y recombinaciones de ellos, en particular por recombinación intermedia global, la cual implica el cálculo del centro de masa de los individuos en la población inmediatamente anterior, así como, la mutación se realiza añadiendo vectores aleatorios con media cero, de modo que toda la matriz de covarianza sea adaptada durante la evolución, mediante el algoritmo descrito, para mejorar la estrategia de búsqueda [13].

## 3.5 Optimización basada en gradiente

Para algoritmos de aprendizaje específicos, es posible calcular el gradiente con respecto a los hiperparámetros y luego optimizar los hiperparámetros utilizando el descenso del gradiente. El primer uso de estas técnicas se centró en las redes neuronales [17]. Desde entonces, estos métodos se han extendido a otros modelos, como las máquinas de soporte vectorial [7] y regresión logística [10].

La optimización basada en gradiente calcula la variación del error al modificar los hiperparámetros, a modo de derivada multidimensional, para así cambiar sus valores y sus resultados. De manera más detallada, el principio general de los métodos basados en gradiente aplicados a la optimización de hiperparámetros, podría ser enunciado como sigue: dado un conjunto de hiperparámetros $H(t)$ para un instante de tiempo $t$, la dirección de máximo crecimiento de la función de error $E(W)$ en $H(t)$ viene dada por el gradiente $\nabla E(H)$. Luego, la dirección de máxima disminución del error viene dada por la dirección contraria a la indicada por el gradiente $\nabla E(H)$. De esta forma se consigue un acercamiento gradual hacia un mínimo local.

$$H(t+1) = H(t) - \alpha \nabla E(H) \tag{3.1}$$

Donde $\alpha$ indica la tasa de aprendizaje o amplitud del paso tomado en cada iteración.

## 3.6 Métodos basados en población

El entrenamiento basado en población (PBT) aprende tanto los valores de los hiperparámetros como los pesos de la red. Los múltiples procesos de aprendizaje operan de forma independiente, utilizando diferentes hiperparámetros. Los modelos de bajo rendimiento se reemplazan iterativamente con modelos que adoptan valores de hiperparámetro modificados de un mejor modelo. La modificación permite que los hiperparámetros evolucionen y elimina la necesidad de definir los hiperparámetros manualmente. El proceso no hace suposiciones con respecto a la arquitectura del modelo, las funciones de pérdida o los procedimientos de capacitación [19].

En este método cada conjunto de hiperparametros se considera un individuo de una población. En esencia, el algoritmo seguido por este enfoque es el siguiente:

1. Generar una población inicial de individuos con valores de hiperparámetros obtenidos aleatoriamente.

2. Para cada individuo, entrenar un modelo de aprendizaje automático con sus hiperparámetros correspondientes, durante un número pequeño de ciclos de entrenamiento.

3. Calcular el rendimiento de cada individuo.

4. Descartar los individuos con peor rendimiento y hacer que los individuos con mejor rendimiento perpetúen sus características, con pequeñas modificaciones. Una forma de conseguir esto es formando múltiples parejas de individuos seleccionados al azar. Se compara el rendimiento de cada individuo con respecto al de su pareja asociada, y se descartan los que tengan un peor rendimiento que su pareja.

5. Los individuos que participaron en el paso anterior y no fueron descartados generan una copia suya con mutaciones aleatorias en los valores de sus hiperparámetros.

6. Repetir los pasos 2 al 5 hasta que se llegue a una condición de parada, como, por ejemplo el número de iteraciones ejecutadas por el algoritmo.

# 4 Método Propuesto

El algoritmo propuesto en el presente trabajo utiliza las bases de PBIL para optimizar hiperparámetros en modelos de aprendizaje automático. Cabe mencionar que la implementación del algoritmo se realizó en el lenguaje de programación Python 3, y se utilizó la librería Keras para la creación de los modelos de aprendizaje automático.

Recordando los principios del descenso por el gradiente, podemos interpretar las combinaciones de parámetros de una red neuronal (pesos) como puntos en una función de error, la cual se recorre en busca de un error mínimo local durante el proceso de entrenamiento. Esta representación sirve para entender el objetivo del algoritmo propuesto. En este caso el objetivo a optimizar no son los pesos, sino los hiperparámetros del modelo, entendiéndose por hiperparámetros los valores que describen la arquitectura de un modelo y como será entrenado, por ejemplo; la tasa de aprendizaje, el número de filtros a utilizar (en el caso de una red convolucional), entre otros. Lo que se busca con el algoritmo expuesto en este proyecto es un método para moverse en el espacio multimensional que representa todas las posibles combinaciones de hiperparámetros en busca de un error mínimo local aceptable.

Utilizando PBIL es posible explorar el espacio multimensional desde varias posiciones de manera paralela. Sin embargo, PBIL se diseñó inicialmente para problemas en los cuales cada valor buscado es binario. Es por este motivo que una de las claves del algoritmo introducido es, por un lado, la manera en que se van a traducir los posibles hiperparámetros a vectores binarios optimizables por el algoritmo, y, por otro lado, la conversión del vector binario a hiperparámetros que puedan ser utilizados para crear modelos los cuales serán evaluados.

En primer lugar es necesario definir que hiperparámetros van a ser optimizados, el rango de valores posibles que pueden tomar, y el tamaño del *paso* con el que se recorrerá

el rango. Es decir *(valor mínimo, valor máximo, tamaño paso)*. Tal como se puede intuir, el número de posibles valores que puede tomar cada hiperparámetro es finito.

Para poder determinar el tamaño del vector de probabilidades $p$ es necesario calcular, para cada hiperparámetro, la cantidad mínima de valores binarios mediante los cuales se pueden representar todos sus posibles valores.

Se selecciona un hiperparámetro $h_k$ definido de la forma $(min, max, paso)$, para determinar el número de posibles valores que puede tomar se utiliza la ecuación:

$$n = \frac{max - min}{paso} \tag{4.1}$$

Posteriormente, se encuentra el mínimo múltiplo de 2 que contenga el valor $n$. Una vez hecho esto se calcula el número de bits necesarios para representar el mencionado múltiplo de 2. El método utilizado en la implementación para devolver el número de bits necesario para representar un valor entero es el siguiente[1]:

**Código 4.1** Método utilizado para devolver el número de bits necesarios para representar un número en nota decimal.

```
def num_bits(num):
    return len([int(x) for x in bin(num)[2:]])
```

Si tomamos, por ejemplo, el hiperparámetro que indica cuantas épocas (epoch) se utilizan para entrenar un modelo, definido de la siguiente manera:

$$epoch = (20, 64, 4)$$

El número de posible valores que puede tomar el hiperparámetro epoch es 11 (llamaremos a este valor *n_epoch*). Con lo cual, el mínimo número de bits necesarios para representarlo es 4, debido a que:

$$2^3 < n\_epoch < 2^4$$

El tamaño del vector de probabilidades $p$ se obtiene sumando el número de bits necesarios para representar todos los hiperparámetros[2].

[1] El método reservado bin en Python convierte un número entero a un binario en forma de cadena

[2] Es necesario almacenar (por ejemplo, en una lista) la identificación de cada hiperparámetro con el número de bits que le corresponden, para procesos posteriores.

Una vez definido el tamaño del vector $p$, es posible, siguiendo los principios de PBIL, inicializar las probabilidades del vector, y, posteriormente, crear la primera generación de individuos. Para crear un nuevo individuo se genera un vector aleatorio con el mismo tamaño del vector $p$ y se compara uno a uno los elementos de ambos vectores. Si un determinado valor del vector aleatorio es mayor a su valor de $p$ correspondiente entonces el valor del individuo en esa misma posición será 1, caso contrario es 0.

Cada individuo estará compuesto por un vector binario, el cual representa la concatenación de sus hiperparámetros codificados, y dará origen a un modelo asociado. Cada modelo se evalúa mediante una función de fitness, y, posteriormente, se refuerzan las características de los modelos que obtuvieron el mejor valor de fitness y se penalizan las características de los modelos con peor rendimiento.

Para poder generar y evaluar cada modelo es necesario decodificar el vector binario de los individuos. Para esto, se divide de manera ordenada el vector en subsecciones que tengan como tamaño el número de bits de cada hiperparámetro representado, posteriormente se transforma el valor a notación decimal y se ubica el resultado dentro su rango específico. En la Figura 4.1 se puede observar un ejemplo de como están concatenados los hiperparámetros en el vector binario de un individuo $x_i$.

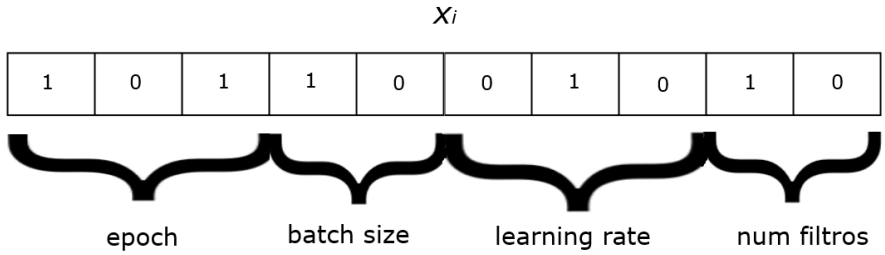



**Figura 4.1** Ejemplo de concatenación de cada hiperparámetro de un individuo en un vector binario.

Sin embargo, existe un inconveniente al realizar la decodificación de cada hiperparámentro. Para explicarlo de manera clara asumamos un ejemplo en el cual el único hiperparámetro a optimizar es el epoch, y la función de error viene dada como se muestra en la Figura 4.2:

Asumiendo que el error mínimo se encuentra en el epoch=16, y en la generación $k$ los mejores individuos conseguidos poseen un epoch=15 después de que el algoritmo consiguiera acercarse al mínimo. Si tomamos en cuenta que el vector que se está optimizando es binario, entonces para poder conseguir que en el la generación $k+1$ sea altamente probable generar individuos con epoch=16 entonces es necesario realizar una

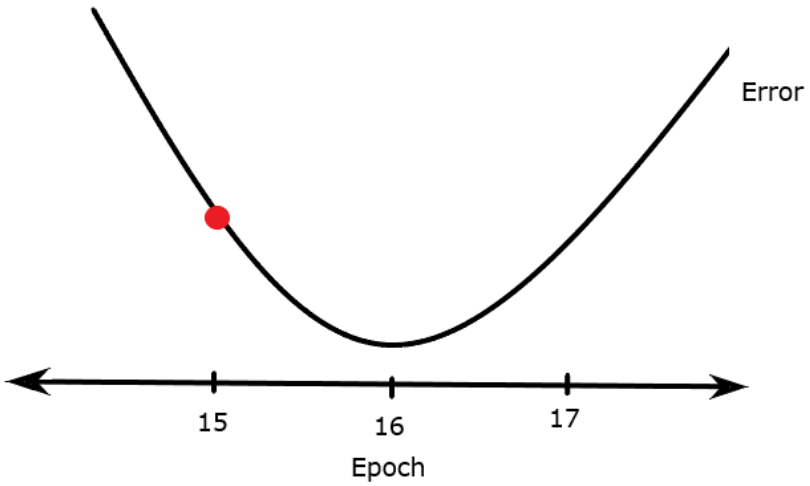



**Figura 4.2** Ejemplo de una función de error.

gran cantidad de cambios en el vector binario del hiperparámetro con respecto al punto anterior, a pesar de que ambos son entero contiguos. Tal como se muestra en la Figura 4.3. Este fenómeno es conocido como Hamming cliff [6].

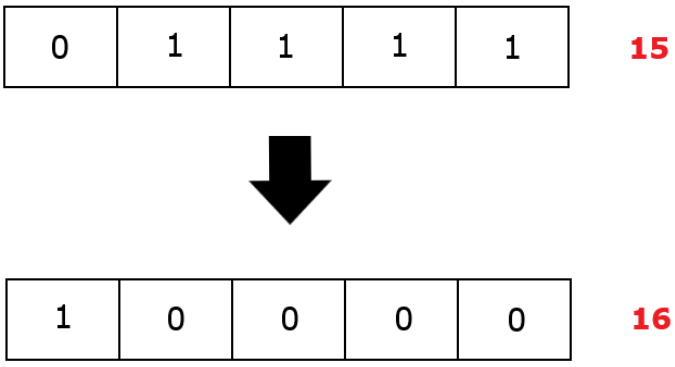



**Figura 4.3** Ejemplo de Hamming cliff.

Hamming cliff (o "acantilado Hamming") se forma cuando dos valores numéricamente adyacentes tienen representaciones de bits que están muy separadas, esto dificulta que el sistema sea capaz de llegar al mínimo. Para solucionar este problema se interpreta los vectores binarios como código Gray. El código Gray es un sistema de ordenamiento

binario en el que dos números consecutivos difieren solamente en uno de sus dígitos. En la Tabla 4.1 se puede observar la representación de los primeros 8 valores en ambos sistemas.

**Tabla 4.1** Primeros ocho valores de código binario y código Gray.

| Decimal | Binario | Gray |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0011 |
| 3 | 0011 | 0010 |
| 4 | 0100 | 0110 |
| 5 | 0101 | 0111 |
| 6 | 0110 | 0101 |
| 7 | 0111 | 0100 |

Para convertir un número en código binario natural a código Gray se aplican las siguientes consideraciones:

- El bit más significativo (BMS) del código Gray es siempre igual al BMS del código binario dado.
- Todos los demás bits del código Gray de salida se pueden obtener aplicando la función XOR entre cada bit del código binario y su bit anterior.

A continuación se muestra como, por ejemplo, se convierte el número binario 0111 a código Gray:

$$\begin{array}{cccc} 0 & 1 & 1 & 1 \\ & 0 & 1 & 1 \quad \dotplus \\ \hline 0 & 1 & 0 & 0 \end{array}$$

Una vez definida la codificación de los vectores binarios de cada individuo, podemos extraer cada subsección correspondiente a los hiperparámetros, y realizar una conversión de código Gray a binario, y posteriormente a decimal (llamaremos $d$ al número obtenido).

Cada posible valor decimal representa uno de los *pasos* dentro del rango definido del hiperparámetro. Recordando que cada hiperpámetro posee los atributos *(min,max,paso)*, se puede calcular el valor del hiperparámetro mediante la ecuación:

$$h = (d \cdot paso) + min \quad , \quad si \quad h > max \rightarrow h = max \tag{4.2}$$

Retomando el ejemplo del hiperparámetro epoch que tiene por atributos *(20,64,4)*, supongamos que tras la conversión del resultado a decimal se obtuvo el valor 8 en su respectiva subsección, con lo cual el valor del hiperparámetro epoch de ese individuo es:

$$h = (8 \cdot 4) + 20 = 52$$

De esta manera se extraen los valores de todos los hiperparámetros de un individuo, y con ellos se crea un modelo de aprendizaje automático, que posteriormente es entrenado y su correspondiente error (valor de fitness) es almacenado.

Los individuos con los mejores valores de fitness son utilizados para modificar el vector de probabilidades $p$ aumentando la probabilidad de obtener sus respectivas características (valores del vector binario). Así mismo, se utilizan los individuos con los peores valores de fitness para reducir la probabilidad de que se vuelvan a obtener cada una de sus características mediante una tasa de aprendizaje definida. En la Figura 4.4 se puede observar un ejemplo de la actualización del vector de probabilidades $p$ tomando en cuenta uno de los mejores individuos de una generación. Mientras menor sea el valor de cada $p_k$ mayor será la probabilidad de que un número generado aleatoriamente sea mayor a $p_k$, con lo cual será mas probable obtener un 1 en esa posición en nuevos individuos generados a partir de $p$.

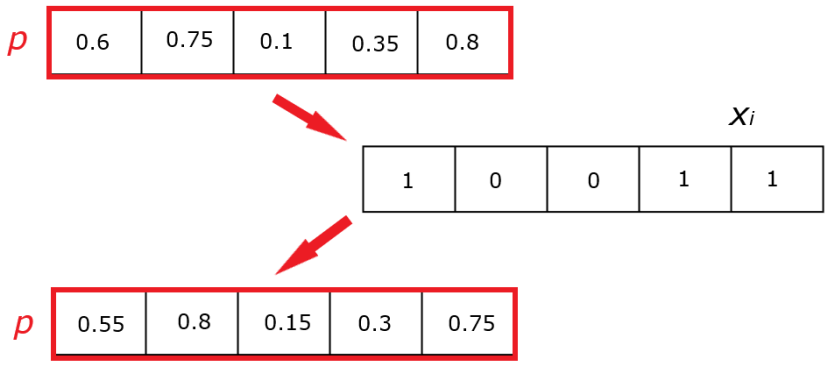



**Figura 4.4** Ejemplo de actualización del vector de probabilidades p tras analizar un individuo $x_i$ que obtuvo uno de los mejores valores de fitness de su generación, utilizando una tasa de aprendizaje de 0.05.

Se repite el proceso desde la creación de la población, un número determinado de generaciones, almacenando el mejor modelo de todas las generaciones producidas. El resultado final del algoritmo es el modelo con mejor valor de fitness de toda la ejecución.

De manera resumida se puede exponer el algoritmo de la siguiente manera:

**1.** Inicializar un vector de probabilidad $p$ de longitud $n$, donde cada valor sea igual a 0,5.

**2.** Generar vectores aleatorios $v_i$ de tamaño $n$. Donde $n$ se obtiene sumando el número de bits mínimos para representar cada posible valor de todos los hiperparámetros, el número de vectores a generar viene dado por el tamaño especificado de la población.

**3.** Para cada vector aleatorio, se genera un individuo $x_i$ relacionado, en la población. Un individuo es, en esencia, un vector de tamaño $n$ con valores binarios. Cada valor del individuo $x_i$ se genera de la siguiente manera:

$$\text{Si } v_{i,j} > p_j \text{ entonces } x_{i,j} = 1 \text{ de otro modo } x_{i,j} = 0. \text{ Donde } j = 1,...,n.$$

Cada individuo representa una combinación de hiperparámetros.

**4.** Descomponer cada individuo $x_i$ en subvectores que tengan por tamaño el mismo tamaño de cada respectivo hiperparámetro, e interpretar cada combinación de números binarios como código Gray, posteriormente traducir este valor a sistema decimal. Cada resultado obtenido representa un hiperparámetro.

**5.** Evaluar el modelo generado con los hiperparámetros obtenidos.

**6.** Si el mejor individuo de esta generación es mejor que los mejores individuos de las generaciones anteriores entonces guardarlo.

**7.** Recorrer los mejores individuos obtenidos en la generación y modificar el vector de probabilidad $p$ para que sea más probable obtener sus características en el futuro. Realizar el proceso en la dirección opuesta para los peores individuos.

**8.** Repetir los pasos 2-7 hasta que se complete el número de generaciones definido.

**9.** Retornar el mejor modelo almacenado.

# 5 Caso de Estudio

Para probar el método propuesto, se realizó un análisis del nivel $^{222}Rn$ de los Laboratorios Subterráneos de Canfranc (LSC) en un periodo de cinco años -de julio de 2013 a junio de 2018- como caso de estudio. Tal como se explica en [21], la predicción del $^{222}Rn$ en el medio ambiente tiene implicaciones relevantes para la calidad científica de los experimentos, y mediante la optimización de los hiperparámetros de un modelo de aprendizaje automático con arquitectura STL+CNN se pretende mejorar la predicción de los valores futuros de esta serie temporal.

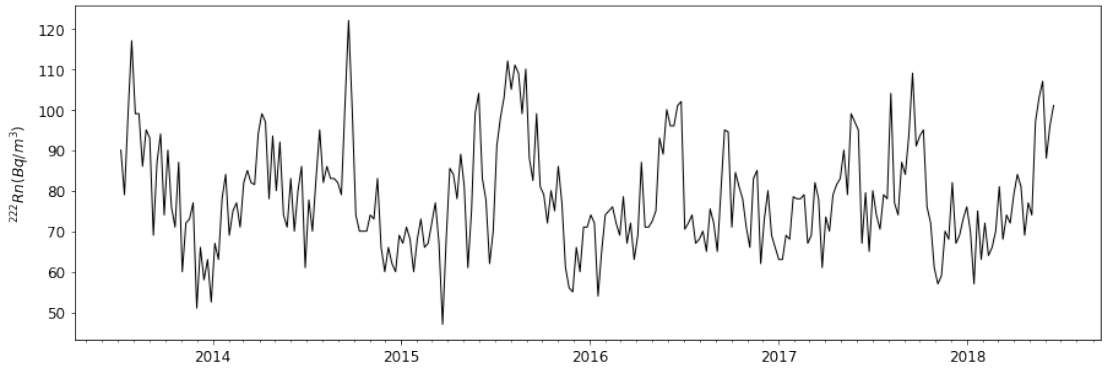
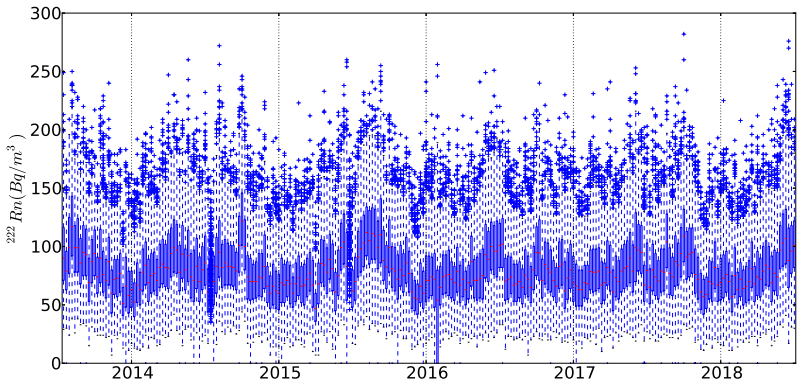



**Figura 5.1** Media semanal de $^{222}Rn$ en el Laboratorio Subterráneo de Canfranc, desde julio de 2013 hasta junio de 2018.

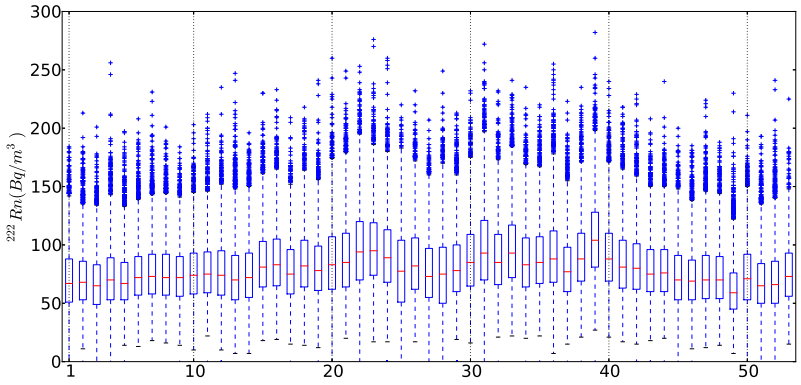
El conjunto de datos utilizado contiene 259 medianas semanales del nivel de $^{222}Rn$ en el Laboratorio Subterráneo de Canfranc (LSC), las cuales se pueden observar en la Figura 5.1. En [21] se presentan análisis previos con datos del $^{222}Rn$ que utilizan enfoques clásicos de aprendizaje profundo y estadística. Posteriormente, en [22] se muestran

algunos resultados de la aplicación de la STL junto con el enfoque de CNN. Para probar el enfoque de optimización de hiperparámetros propuesto en este trabajo, se realizaron dos análisis, cada una con dos hiperparámetros involucrados en el modelo $STL+CNN$ que serán optimizados automáticamente.

En la Figura 5.2, se muestran los boxplots semanales, cuyas medianas semanales se utilizan para crear la serie temporal con la cual se realiza el estudio (Figura 5.1).




(a) Boxplots semanales




(b) Boxplots de cada semana en años distintos

**Figura 5.2** Boxplots semanales de $^{222}Rn$ en la Sala A del LSC. La toma de datos corresponde al período comprendido entre julio de 2013 y junio de 2018.

El 70 % inicial de los datos de la serie temporal se utiliza para el entrenamiento de los modelos y el restante 30 % se utiliza para determinar su rendimiento. La arquitectura STL con CNN implica que la serie temporal original es descompuesta en sus tres componentes: estacionalidad, tendencia, y residuo. Estas tres componentes alimentarán al mismo tiempo

una Red Neuronal Convolucional que tenga como salida la predicción de los valores futuros de la serie temporal original. En principio, la Red Neuronal debe ser capaz de modelar con relativa facilidad las componentes de tendencia y estacionalidad, tratando de forma independiente la componente más compleja, es decir, el residuo, sin necesidad de utilizar una red excesivamente grande. Por lo tanto, la aplicación de STL funciona como una ayuda para que el sistema modele correctamente los datos en bruto.

Así mismo, la utilización de STL brinda al algoritmo planteado la posibilidad de optimizar hiperparámetros externos a la Red Convolucional, y de esta manera intervenir directamente en el preprocesado de los datos.

En la Figura 5.3 se puede observar la arquitectura de la Red Neuronal utilizada.

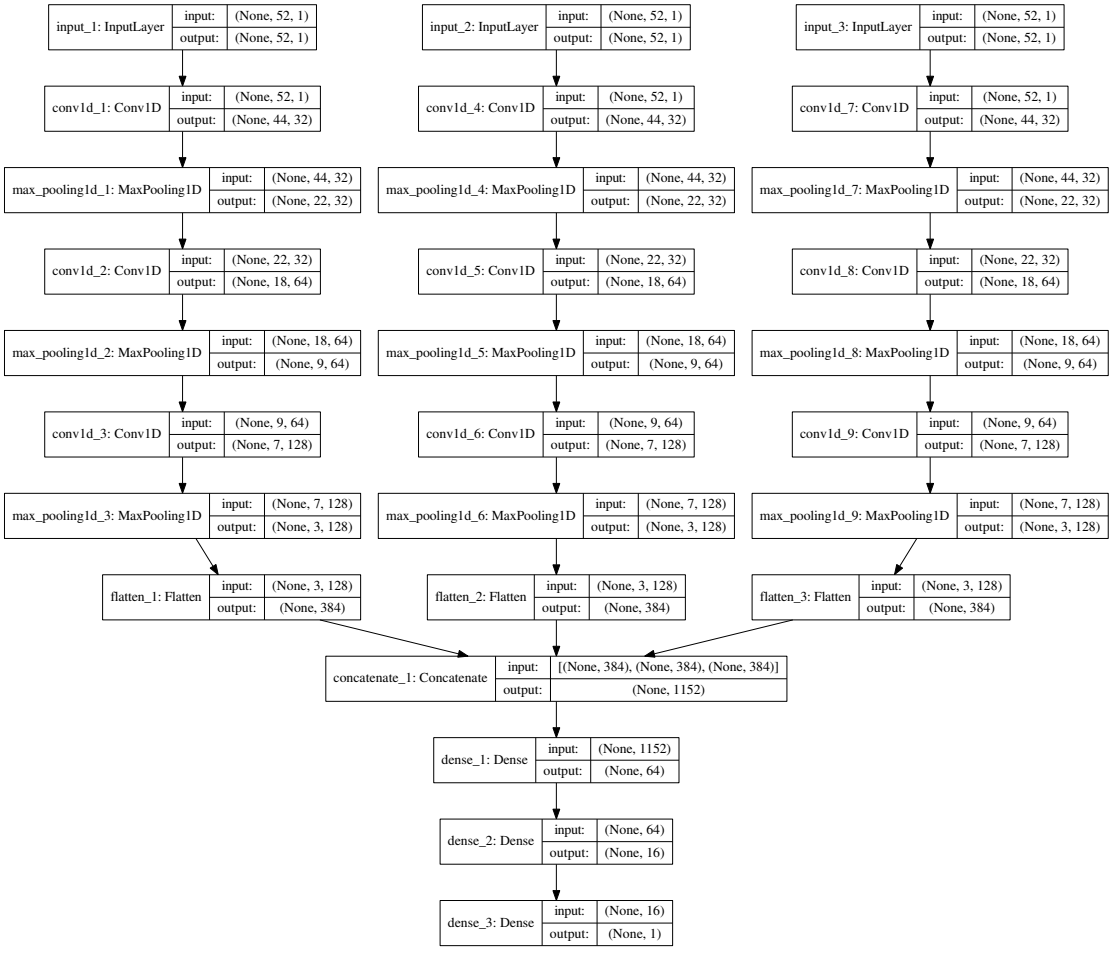



**Figura 5.3** Estructura de la Red Convolucional para la cual se optimizan los hiperparámetros seleccionados.

En el primer análisis se utilizan los hiperparámetros: período, que define el período de la componente de estacionalidad en la serie temporal para la descomposición STL, y el tamaño del batch utilizado en el proceso de entrenamiento de la CNN. Es necesario plantear una manera de determinar que efectivamente el algoritmo es capaz de acercarse a un error mínimo aceptable, por lo cual, se realiza un barrido paramétrico que pruebe todas las posibles combinaciones. Para el periodo se determinó como rango de valores posibles [30, 61] y para el tamaño del batch [40, 71], ambos con un tamaño de paso igual a 1, por lo que en total se entrenaron 1024 modelos en el barrido paramétrico. De esta manera, a través de fuerza bruta, se puede observar cuáles son las combinaciones que minimizan el MSE.

En la Figura 5.4 se puede observar el MSE obtenido a partir del barrido paramétrico para cada posible combinación de hiperparámetros utilizados en el primer análisis.

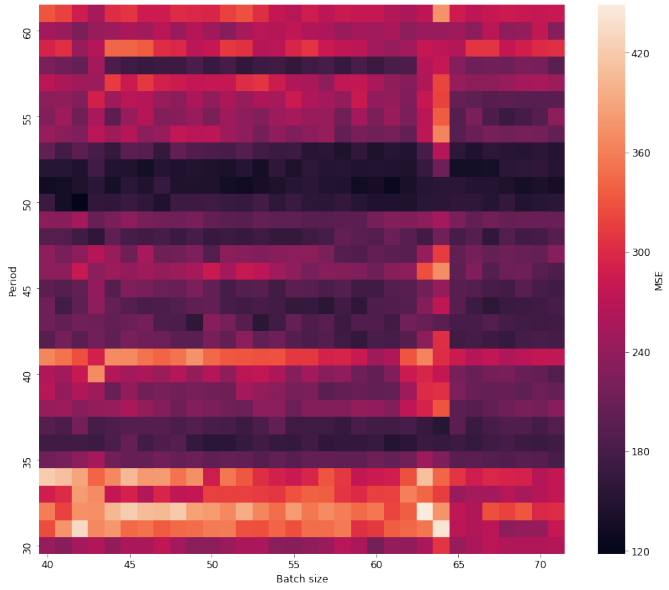



**Figura 5.4** Mapas de calor del MSE con respecto a los posibles valores de *período* (eje y) y *tamaño de batch* (eje x)..

Observando los datos se puede determinar que el *periodo* tiene una alta relación con el MSE obtenido, por lo que en el segundo análisis se utiliza nuevamente el periodo con el mismo rango, y como segundo hiperparámetro se utiliza un valor denominado *fraction* el cual especifica el porcentaje de datos a utilizar en el entrenamiento de la regresión Loess. El hiperparámetro *fraction* puede tomar valores entre [0,1], y el paso definido es de 0,1, por lo cual en total se entrenaron 320 modelos.

En la Figura 5.5 se puede observar los MSE obtenidos para el segundo análisis.

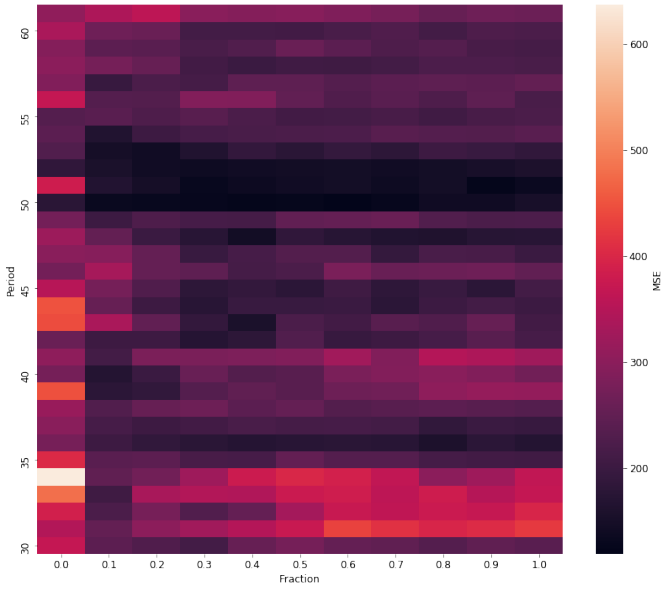



**Figura 5.5** Mapas de calor del MSE con respecto a los posibles valores de *período* (eje y) y *fraction* (eje x).

Cada ejecución en los análisis tomó un promedio de 4,59 segundos en un ordenador con 16 GB de RAM, CPU Intel Core i7-8750H y tarjeta gráfica Nvidia GTX 1070.

Posteriormente, los hiperparámetros fueron optimizados con el algoritmo de optimización descrito en la Sección 4 y su eficiencia fue verificada comparando sus resultados con los resultados del barrido paramétrico.

Para poder conseguir una alta reproducibilidad en los resultados, se definió una *semilla*[1] fija para el generador de números pseudoaleatorios utilizado por Keras en la generación de los pesos iniciales de todos los modelos. Sin embargo, pueden existir pequeñas variaciones en los resultados obtenidos a partir de unos mismos hiperparámetros, debido a que la ejecución se realiza en GPU. Dependiendo del proveedor, el software de una determinada GPU puede estar configurado para utilizar una sofisticada pila de librerías, y que algunas de ellas introduzcan su propia fuente de aleatoriedad, tal es el caso de las tarjetas Nvidia[2].

[1] Un generador de números pseudoaleatorios es una función matemática que genera una secuencia larga de números que son lo suficientemente aleatorios para un uso general, como es el caso de los algoritmos de aprendizaje automático. Dichos generadores requieren una semilla para iniciar el proceso, y es común usar el tiempo actual en milisegundos en la mayoría de las implementaciones. Esto para garantizar que se generen diferentes secuencias de números aleatorios cada vez que se ejecuta el código.

La semilla puede ser especificada con un número determinado, para garantizar que se genera la misma secuencia de números pseudoaleatorios cada vez que se ejecuta el código.

[2] Más información en: https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html

# 6 Resultados

Para poder demostrar de manera aplicada que el método propuesto aprende a producir mejores modelos con cada nueva generación, los resultados deben mostrar una clara tendencia a aglomerarse alrededor de un error mínimo.

Para la evaluación se realizaron 25 campañas PBIL independientes por cada combinación de hiperparámetros a analizar, cada campaña con 15 generaciones y cada generación con 10 individuos generados. Los resultados son comparados con el barrido paramétrico de cada análisis y de esta manera se determina la calidad de los modelos producidos con PBIL.

En la Figura 6.1 se presentan los boxplot de los mejores individuos de todas las campañas PBIL.

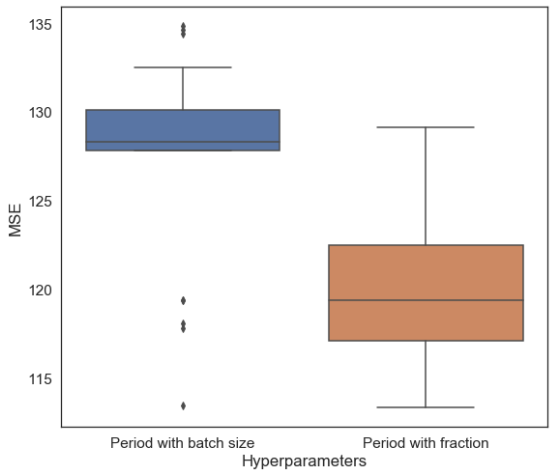



**Figura 6.1** Mejores individuos de cada campaña PBIL.

Los datos de las campañas PBIL y los barridos paramétricos de ambos análisis realizados se pueden observar a detalle en la Tabla 6.1.

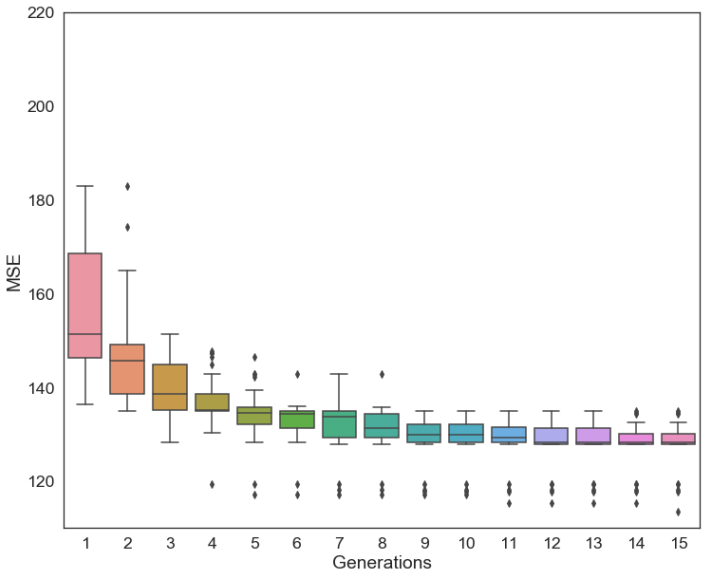
**Tabla 6.1** Valores obtenidos durante la ejecución de los barridos paramétricos, y los promedios (Avg) y desviaciones estándar (std) de 25 campañas PBIL independientes para cada análisis realizado.

| **Combinación de hiperparámetros** | | **Periodo + Batch size** | **Periodo + Fraction** |
|---|---|---|---|
| **Barrido paramétrico** | **Avg $\pm$ std** | 231.69 $\pm$ 61.24 | 242.85$\pm$ 71.28 |
| | **Mínimo Global** | 118.22 | 119.37 |
| **Campañas PBIL** | **1ª Generación** | 157.00 $\pm$ 13.80 | 160.42 $\pm$ 23.74 |
| | **2ª Generación** | 147.47 $\pm$ 11.86 | 136.86 $\pm$ 16.21 |
| | **3ª Generación** | 140.21 $\pm$ 6.10 | 131.18 $\pm$ 13.39 |
| | **4ª Generación** | 137.04 $\pm$ 6.39 | 126.39 $\pm$ 7.17 |
| | **5ª Generación** | 134.19 $\pm$ 6.52 | 123.58 $\pm$ 6.39 |
| | **6ª Generación** | 132.55 $\pm$ 5.24 | 122.49 $\pm$ 5.75 |
| | **7ª Generación** | 131.51 $\pm$ 5.95 | 122.23 $\pm$ 5.43 |
| | **8ª Generación** | 130.83 $\pm$ 5.79 | 122.06 $\pm$ 5.42 |
| | **9ª Generación** | 128.77 $\pm$ 5.26 | 121.09 $\pm$ 4.57 |
| | **10ª Generación** | 128.77 $\pm$ 5.26 | 120.81 $\pm$ 4.10 |
| | **11ª Generación** | 128.52 $\pm$ 5.38 | 120.69 $\pm$ 4.02 |
| | **12ª Generación** | 127.84 $\pm$ 5.47 | 120.22 $\pm$ 3.95 |
| | **13ª Generación** | 127.82 $\pm$ 5.46 | 120.18 $\pm$ 3.94 |
| | **14ª Generación** | 127.65 $\pm$ 5.39 | 120.12 $\pm$ 3.87 |
| | **15ª Generación (Mínimo de la campaña)** | 127.57 $\pm$ 5.58 | 119.97 $\pm$ 3.97 |

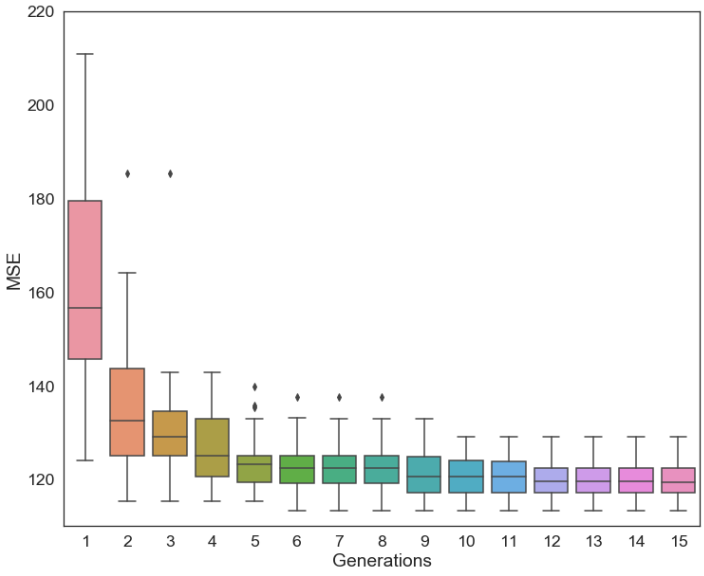
Según lo obtenido se demuestra experimentalmente la utilidad de PBIL como método de optimización de hiperparámetros. Se puede observar que el algoritmo es capaz de acercarse al error mínimo global en unas pocas generaciones, de la misma manera que con cada generación la desviación estándar tiende a reducirse, por lo que, además de que los valores medios de error mejoran, los resultados también tienden a aglomerarse.

La convergencia de los resultados se puede observar claramente en la Figura 6.2, donde se presentan los boxplot para ambas combinaciones de hiperparámetros, teniendo en cuenta los mejores individuos de cada generación en las 25 campañas. Se puede afirmar

por lo tanto, que el método propuesto en el presente proyecto es efectivo en su tarea de optimizar hiperparámetros.




(a) Mejores individuos en el análisis hecho con los hiperarámetros *periodo* y *tamaño de batch*




(b) Mejores individuos en el análisis hecho con los hiperarámetros *periodo* y *fraction*

**Figura 6.2** Boxplots con los mejores individuos en cada generación de las campañas PBIL.

# 7 Conclusiones y Trabajos Futuros

Una vez expuesto el procedimiento para la implementación de PBIL en la optimización automática de hiperparámetros y habiendo analizado los resultados obtenidos con respecto al caso de estudio, se pueden extraer varias conclusiones y propuestas para trabajos futuros.

## 7.1 Conclusiones

A lo largo del presente proyecto se ha expuesto un método para optimizar hiperparámetros en modelos de aprendizaje automático utilizando PBIL. Mediante la implementación del caso de estudio propuesto se comprobó que, ejecutando el algoritmo descrito, con un pequeño número de generaciones y un pequeño número de individuos por generación, es posible generar modelos con resultados cercanos o iguales al óptimo global, en un rango definido de valores.

Se puede concluir que la solución presentada es una buena alternativa para generar modelos optimizados para una determinada tarea, con resultados cercanos al mejor resultado posible y en un tiempo de ejecución razonable.

Además, tal como se puede observar, la combinación de métodos evolutivos con aprendizaje automático es un campo de investigación interesante, capaz de brindar resultados prometedores, y mejorar el rendimiento de una amplia variedad de sistemas.

El estudio de la optimización automática de hiperparámetros es un área compleja pero bastante importante para el desarrollo del aprendizaje automático, debido a que, la creación de métodos de optimización suficientemente potentes podrían dar pie a sistemas capaces de crear nuevos sistemas de forma completamente autónoma, superando los resultados que un desarrollador podría conseguir. Y, a su vez, permitiendo que en un

futuro se reduzca el esfuerzo necesario para el diseño de una arquitectura de aprendizaje automático competente, y se pueda enfocar la mayor parte del esfuerzo de desarrollo en el planeamiento del problema y otras tareas relacionadas.

## 7.2 Trabajos Futuros

Durante la realización del presente proyecto surgieron varias ideas para trabajos futuros, entre ellas:

- Debido a que la potencia computacional disponible es el único limite en el número de hiperparámetros a optimizar, es posible desarrollar un sistema complejo capaz de optimizar todos los hiperparámetros involucrados en la generación de un modelo. Es decir, la solución propuesta puede extenderse también a la arquitectura del propio modelo, determinando la cantidad óptima de capas, neuronas, e incluso el tipo de red neuronal a generar, entre otros. Para que el sistema sea capaz de generar una red neuronal altamente optimizada desde cero y sin intervención humana, siempre y cuando el sistema esté provisto de una medida de fitness asignable a cada individuo generado.

- En el presente proyecto no se buscó conseguir la mejor predicción posible para los niveles de $^{222}Rn$ del LSC, sino que el objetivo principal es la presentación de PBIL como método de optimización de hiperparámetros. Debido a lo cual, en trabajos futuros que traten a fondo el caso de estudio utilizado se pueden probar arquitecturas distintas, las cuales pueden ser optimizadas mediante el método propuesto en el presente proyecto, o incluso permitiendo que el sistema seleccione la mejor arquitectura de manera automática, tal como se menciona previamente.

- Basado en las ideas de los optimizadores expuestos en la subsección 2.3.4 se plantea la posibilidad de, en futuros proyectos, investigar la utilización de tasas de aprendizaje variables en PBIL, que se vayan adaptando a medida que el sistema aprende.

- Se planea la idea de desarrollar un método que permita definir automáticamente el óptimo número de individuos a utilizar en cada generación, con los cuales se modifique el vector de probabilidades, tomando en cuenta los resultados obtenidos por los individuos de una generación previa en PBIL. Por ejemplo, si los mejores dos individuos de una generación tienen resultados notablemente superiores a los demás individuos, es deseable que se utilice únicamente esos dos individuos para modificar el vector de probabilidades, debido a que, utilizar otros individuos con resultados muy inferiores puede ser contraproducente para el proceso de aprendizaje.

# Apéndice A

# Código de la implementación realizada

In [ ]:
```
import os
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib notebook
import random as rn
import math

from keras.backend import clear_session
from keras.models import Model
from keras.layers import Dense, Dropout, Activation, Flatten, concatenate, Input
from keras.layers import Convolution1D, MaxPooling1D
from keras.optimizers import Adam

from sklearn.metrics import mean_squared_error

import matplotlib as mpl
mpl.rcParams['xtick.labelsize'] = 'large'
mpl.rcParams['ytick.labelsize'] = 'large'
mpl.rcParams['axes.labelsize'] = 'large'

from stldecompose import decompose, forecast   #pip install stldecompose
from stldecompose.forecast_funcs import (naive,
                                         drift,
                                         mean,
                                         seasonal_naive)
```

In [ ]:
```
dataset = [90, 79, 99, 117, 99, 99, 86, 95, 93, 69, 87, 94, 74, 90, 76, 71, 87, 60, 72,
73, 77, 51, 66, 58, 63, 52.5, 67, 63, 78, 84, 69, 75,
77, 71, 82, 85, 82, 81.5, 94, 99, 97, 78, 93.5, 80, 92, 74, 71, 83, 70, 80, 86, 61,
77.6969111969112, 70, 83, 95, 82, 86, 83, 83, 82, 79, 101,
122, 100, 74, 70, 70, 70, 74, 73, 83, 66, 60, 66, 62, 60, 69, 67, 71, 68, 60, 68, 73,
66, 67, 72, 77, 67, 47, 68, 85.5, 84, 78, 89, 81, 61,
75, 99, 104, 83, 77.6969111969112, 62, 70, 91, 98, 103, 112, 105, 111, 109, 99, 110, 88,
82.5, 99, 81, 79, 72, 80, 75, 86, 77, 61, 56, 55, 66,
60, 71, 71, 74, 72, 54, 65, 74, 75, 76, 72, 69, 78.5, 67, 72, 63, 69, 87, 71, 71, 72.5,
75, 93, 89, 100, 96, 96, 101, 102, 70.5, 72, 74, 67,
68, 70, 65, 75.5, 72, 65, 80, 95, 94.5, 71, 84.5, 81, 78, 71, 66, 83, 85, 62, 73, 80,
69, 66, 63, 63, 69, 68, 78.5, 78, 78, 79, 67, 69, 82,
78, 61, 73.5, 70, 79, 81.5, 83, 90, 79, 99, 97, 95, 67, 79.5, 65, 80, 74, 70.5, 79, 78,
104, 77, 74, 87, 84, 94, 109, 91, 93.5, 95, 76, 72,
61, 57, 59, 70, 68, 82, 67, 69, 73, 76, 70, 57, 75, 63, 72, 64, 66, 70, 81, 68, 74, 72,
79, 84, 81, 69, 77, 74, 97, 103, 107, 88, 96, 101]

date = pd.date_range(start='7/1/2013', freq='W', periods=259)
df = pd.DataFrame(dataset, index=date)
```

In [ ]:
```
# Hiperparámetros a optimizar [Nombre del hiperpárametro, desde, hasta, con paso]
hiperparametros = [
    ["batch_size", 40, 71, 1],
    ["lo_frac", 0, 1, 0.1],
    ["period", 30, 61, 1],
    ["lo_delta", 0, 0.2, 0.01],
    ["sample_size", 52, 100, 2],
    ["epoch", 20, 140, 20],
    ["learning_rate", 0.001, 0.05, 0.005],
    ["lr_decay", 0.0, 0.001, 0.0001],
    ["kernel1", 1, 10, 1],
    ["kernel2", 1, 6, 1],
    ["kernel3", 1, 3, 1],
    ["filtro1", 10, 50, 5],
    ["filtro2", 30, 80, 10],
    ["filtro3", 60, 160, 20]
]

##Valores por defecto:
#lo_delta=0.01
#batch_size = 50
#period = 5
```

```
        #lo_frac = 0.6
        #sample_size = 52
        #epoch=100
        #learning_rate=0.001
        #lr_decay=0.0
        #kernel1 = 9
        #kernel2 = 5
        #kernel3 = 3
        #filtro1 = 32
        #filtro2 = 64
        #filtro3 = 128
```

In [ ]:
```
import gc

generaciones = pd.DataFrame(columns=['Generacion','MSE'])

def pbil_optimizacion(learn_rate, learn_rate_neg, num_pob,
num_mejores_para_actualizar_vec, num_peores_para_actualizar_vec, vec_len,
             ciclos_optimizacion, eval_f, eps=0.01, lista_vecs=None):
    """
    learn_rate: tasa de actualización del vector poblacional (vec) hacia cada uno de los
mejores individuos.
    learn_rate_neg: similar a learn_rate, pero para alejar el vector de los peores
individuos.
    num_pob: número de individuos en cada población
    num_mejores_para_actualizar_vec: ¿cuántos mejores individuos se utilizarán para
actualizar el vector poblacional?
    num_peores_para_actualizar_vec: ¿cuántos de los peores individuos se utilizarán para
actualizar el vector poblacional?
    vec_len: longitud del vector de población
    ciclos_optimizacion: numero de ciclos de optimizacion
    eval_f: función para la evaluación del fitness de cada individuo
    eps: el vector poblacional será alejado de los valores extremos (0, 1) sumando o
restando un valor eps determinado.
    lista_vecs: almacena los vectores de población de cada generación.
    """
    global generaciones
    secciones = [row[2] for row in hiperparametros_int_aux]

    # Inicialización del vector
    vec = crea_prob_inicial(vec_len)

    # Inicialización de la población
    poblacion = pob_ini(num_pob, vec_len)
    scores = scores_ini(num_pob)
    modelos = [None] * num_pob
    hipers = [None] * num_pob

    # Inicializar mejor resultado
    # [Fitness, vector binario, modelo entrenado, hiperparam. en decimales]
    mejor_de_todos = [float("inf"), None]

    lista_vecs = [vec]

    for i in range(ciclos_optimizacion):
        for j in range(num_pob):
            poblacion[j][0:secciones[0]] = get_num(vec[0:secciones[0]])
            anterior = secciones[0]
            indx=1
            while indx < len(secciones):
                poblacion[j][anterior:secciones[indx]+anterior] =
get_num(vec[anterior:secciones[indx]+anterior], indx= indx)
                anterior = anterior + secciones[indx]
                indx+=1

            # Evaluaciones de los individuos
            scores[j], modelos[j], hipers[j] = eval_f(poblacion[j])
        # Selección de los mejores individuos
```

```
            results_ordenados = sorted(zip(scores, poblacion, modelos, hipers), key=lambda
        x:x[0], reverse=False)
            mejor = results_ordenados[:num_mejores_para_actualizar_vec]
            worst = results_ordenados[-num_peores_para_actualizar_vec:]

            if mejor_de_todos[0] > mejor[0][0]:
                mejor_de_todos = (mejor[0][0], list(mejor[0][1]), mejor[0][2], mejor[0][3])

            print('Paso: {0}'.format(i))
            print('Vector de prob: {0}'.format(vec))
            print('Mejores: {0}'.format([(b[0]) for b in mejor]))
            print('Peores: {0}'.format([(w[0]) for w in worst]))
            print('Mejor de todos: {0}'.format((mejor_de_todos[0])))
            print("--------------")

            # Actualizar vector
            for v in mejor:
                vec += 2 * learn_rate * (v[1] - 0.5)
            for v in worst:
                vec -= 2 * learn_rate_neg * (v[1] - 0.5)

            # Corrección del vector si hay elementos fuera del rango [0, 1]
            for j in range(vec_len):
                if vec[j] < 0:
                    vec[j] = 0 + eps
                elif vec[j] > 1:
                    vec[j] = 1 - eps

            # Añadir vec
            lista_vecs.append(vec)

            generaciones = generaciones.append({'Generacion':i+1,
                                'MSE':mejor_de_todos[0]}, ignore_index=True)

            #generaciones.to_pickle('data_mayo/generationes_periodo_batchsize_nuevo_3.pkl')

        return mejor_de_todos
```

In [ ]:
```
#Inicialización y entrenamiento de cada modelo

def model_creation(df, lo_delta=0.01, batch_size = 50, period = 5, lo_frac = 0.6,
sample_size = 52,
                   epoch=100, learning_rate=0.001, lr_decay=0.0, kernel1 = 9, kernel2 =
5, kernel3 = 3,
                   filtro1 = 32, filtro2 = 64, filtro3 = 128):

    """
    df: DataFrame de pandas con los valores de la serie temporal.
    #Hiperparametros STL
    period: periodicidad más significativa de la serie temporal observada, en unidades
de 1 observación. Ejm: para indicar una fuerte periodicidad anual con observaciones
tomadas diariamente: period=365
    lo_frac: fracción de datos a utilizar en el entrenamiento de la regresión Lowess.
    lo_delta: distancia fraccional dentro de la cual se puede utilizar la interpolación
lineal en lugar de una regresión ponderada. Usar un lo_delta que no sea cero disminuye
significativamente tiempo de cálculo.
    #Hiperparametros modelo
    batch_size: número de ejemplos de entrenamiento utilizados en cada iteración del
entrenamiento.
    sample_size: número de observaciones previas que se tomarán como inputs.
    epoch: número de veces en que se procesarán todos los ejemplos del conjunto de datos
en una sesión de entrenamiento.
    learning_rate: medida en la que los pesos se actualizan con respecto a la nueva
información adquirida tras procesar cada batch.
    lr_decay: tasa de decaimiento del learning_rate en cada iteración
    kernel1-3: tamaño de la ventana (o detector de caracteristicas) que recorrerá los
datos en cada capa de convolución.
    filtro1-3: número de filtros a utilizar en cada capa de convolución.
```

```
    """


    import numpy as np
    np.random.seed(42) # para reproducibilidad
    import tensorflow as tf
    import keras
    from keras import backend as K
    tf.set_random_seed(1234)
    from tensorflow.python.util import deprecation
    deprecation._PRINT_DEPRECATION_WARNINGS = False

    ##DESCOMPOSICIÓN STL

    stl = decompose(df, period=period, lo_frac=lo_frac, lo_delta=lo_delta)

    seasonal, trend, random =  np.asarray(stl.seasonal.values),
np.asarray(stl.trend.values), np.asarray(stl.resid.values)

    #DEFINICIÓN DEL MODELO

    kernel_sz = [int(kernel1), int(kernel2), int(kernel3)]
    filtros = [filtro1,filtro2,filtro3]

    # CNN para seas
    inputseas = Input(shape=(sample_size, 1))
    convseas = Convolution1D(filtros[0], kernel_sz[0], activation='relu')(inputseas)
    convseas = MaxPooling1D(pool_size =2)(convseas)
    convseas = Convolution1D(filtros[1], kernel_sz[1], activation='relu')(convseas)
    convseas = MaxPooling1D(pool_size =2)(convseas)
    convseas = Convolution1D(filtros[2], kernel_sz[2], activation='relu')(convseas)
    convseas = MaxPooling1D(pool_size =2)(convseas)
    convseas = Flatten()(convseas)

    # CNN para trend
    inputtrend = Input(shape=(sample_size, 1))
    convtrend = Convolution1D(filtros[0], kernel_sz[0], activation='relu')(inputtrend)
    convtrend = MaxPooling1D(pool_size =2)(convtrend)
    convtrend = Convolution1D(filtros[1], kernel_sz[1], activation='relu')(convtrend)
    convtrend = MaxPooling1D(pool_size =2)(convtrend)
    convtrend = Convolution1D(filtros[2], kernel_sz[2], activation='relu')(convtrend)
    convtrend = MaxPooling1D(pool_size =2)(convtrend)
    convtrend = Flatten()(convtrend)

    # CNN para rand
    inputrand = Input(shape=(sample_size, 1))
    convrand = Convolution1D(filtros[0], kernel_sz[0], activation='relu')(inputrand)
    convrand = MaxPooling1D(pool_size =2)(convrand)
    convrand = Convolution1D(filtros[1], kernel_sz[1], activation='relu')(convrand)
    convrand = MaxPooling1D(pool_size =2)(convrand)
    convrand = Convolution1D(filtros[2], kernel_sz[2], activation='relu')(convrand)
    convrand = MaxPooling1D(pool_size =2)(convrand)
    convrand = Flatten()(convrand)

    merged = concatenate([convseas, convtrend, convrand],axis=1)
    out = Dense(64)(merged)
    out = Dense(16)(out)
    out = Dense(1, activation='linear')(out)

    final_model = Model(inputs=[inputseas, inputtrend, inputrand], outputs=[out])

    # Compilar
    adam = Adam(lr=learning_rate, decay=lr_decay)
    final_model.compile(loss="mse", optimizer=adam)

    # ENTRENAMIENTO DEL MODELO

    dataset = np.asarray(df.values)
```

```
        y = dataset[sample_size:]

        Xseas = np.atleast_3d(np.array([seasonal[start:start + sample_size] for start in
    range(0, seasonal.shape[0]-sample_size)]))
        Xtre = np.atleast_3d(np.array([trend[start:start + sample_size] for start in
    range(0, trend.shape[0]-sample_size)]))
        Xrand = np.atleast_3d(np.array([random[start:start + sample_size] for start in
    range(0, random.shape[0]-sample_size)]))

        test_size = int(len(dataset)*0.3)

        trainY, testY = y[:-test_size], y[-test_size:]

        trainXseas = Xseas[:-test_size]
        testXseas =  Xseas[-test_size:]

        trainXtre = Xtre[:-test_size]
        testXtre =  Xtre[-test_size:]

        trainXrand = Xrand[:-test_size]
        testXrand =  Xrand[-test_size:]

        nextSteps = np.empty((1,sample_size,1))
        nextSteps[0,:,:]= np.atleast_3d(np.array([dataset[start:start + sample_size]
                                                  for start in
    range(dataset.shape[0]-sample_size,dataset.shape[0]-sample_size+1)]))

        final_model.fit([trainXseas,trainXtre,trainXrand], trainY, epochs=epoch,
    batch_size=batch_size, verbose=0)

        pred = final_model.predict([testXseas,testXtre,testXrand])

        testScore = mean_squared_error(testY, pred)

        clear_session()
        gc.collect()

        return testScore, final_model
```

```
In [ ]: import numpy as nump

    ##Devuelve el numero de bits mínimos necesarios para representar un número decimal.

    def num_bits(num_dec):
        res= 2**(num_dec - 1).bit_length()
        if num_dec == res:
            num_dec+=1
            res= 1 if num_dec == 0 else 2**(num_dec - 1).bit_length()
        res= int(res/2)

        return len(entero_a_binario(res))

    #Toma un resultado obtenido en decimales y lo ubicar dentro del rango definido para su
    hiperparametro correspondiente.
    def resultado_a_rango(num_dec, hiperparametro):
        resul = num_dec * hiperparametro[3] + hiperparametro[1]
        if resul>hiperparametro[2]: resul = hiperparametro[2]
        return resul


    #Convierte una cadena de números binarios a código Gray.
    def binario_a_gray(num):
        return num ^ (num >> 1)

    #Convierte una cadena de números binarios en formato Gray a entero.
    def gray_a_entero(num_gray):
        num = binario_a_entero(num_gray)
        inv = 0;
```

```
    while(num):
        inv = inv ^ num;
        num = num >> 1;
    return inv;

def entero_a_binario(num):
    return [int(x) for x in bin(num)[2:]]

#Convierte una cadena binaria a una variable entera
def binario_a_entero(num_bit):
    out = 0
    for bit in num_bit:
        out = (out << 1) | bit
    return out

#Genera números aleatorios y los compara con los valores de un vector "prob"
def get_num(prob, indx=0):
    vec = [None] * len(prob)
    for i in range(len(prob)):
        vec[i]= rn.random() < prob[i]

    if indx ==1:
        if (gray_a_entero(vec)>10):
            vec = get_num(prob)
    return vec

#Creamos un array de longitud vec_len, correspondiente al vector de probabilidades.
#Al principio, la probabilidad de obtenerse un 1 o un 0 es 0,5.
def crea_prob_inicial(vec_len):
    return nump.full(vec_len, 0.5, dtype=float)

#Inicializa la población (con valores 0).
def pob_ini(num_pob, vec_len):
    return nump.zeros((num_pob, vec_len), dtype=int)

#Inicialización de los valores de la función fitness (comienzan valiendo None)
def scores_ini(num_pob):
    return [None for _ in range(num_pob)]

#Procesa el vector binario de un individuo y devuelve los valores de sus hiperparametros
en decimales.

def proc_bin(bin_vector):
    global hiperparametros, hiperparametros_int_aux
    flag=0
    dec_resul=[]
    for i, hiper in enumerate(hiperparametros):
        dec_resul.append(resultado_a_rango(gray_a_entero(bin_vector[flag : flag +
hiperparametros_int_aux[i][2]]),hiper))
        flag += hiperparametros_int_aux[i][2]

    return dec_resul

#Función encargada de procesar el vector binario de cada individuo, definir y entrenar
su modelo correspondiente, y devolver su MSE.

def fitness(bin_num):
    dec_resul = proc_bin(bin_num)

    hiper_inputs = {}
    for i, v in enumerate(dec_resul):
        hiper_inputs[hiperparametros_int_aux[i][0]] = v

    loss, modelo = model_creation(df, **hiper_inputs)

    gc.collect()

    return loss, modelo, hiper_inputs
```

```
        #Convertir la lista de hiperparámetros a una lista con el nombre, el número total de
        posibles valores, y el número de
        #bits mínimo necesario para representarlo

        hiperparametros_int_aux= []
        for h in hiperparametros:
            hiperparametros_int_aux.append([h[0], math.ceil((h[2]-h[1])/h[3]),
        num_bits(math.ceil((h[2]-h[1])/h[3]))])

In [ ]: mejor = pbil_optimizacion(learn_rate=0.02,
                                learn_rate_neg=0.02,
                                num_pob=15,
                                num_mejores_para_actualizar_vec =3,
                                num_peores_para_actualizar_vec =3,
                                vec_len=sum([h[2] for h in hiperparametros_int_aux]),
                                ciclos_optimizacion=20,
                                eval_f=fitness)
```

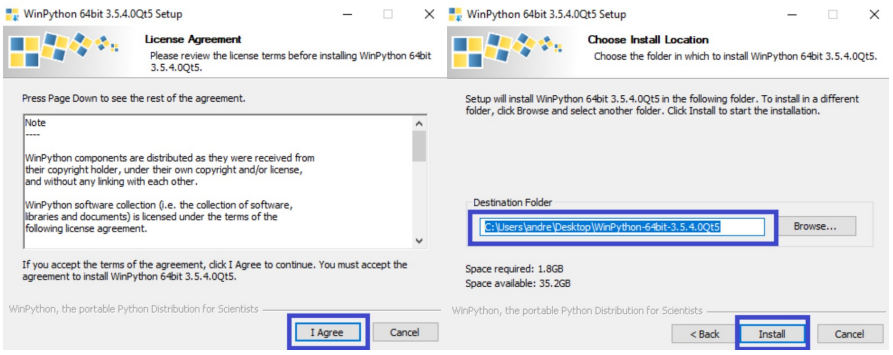
# Apéndice B

# Manual de uso

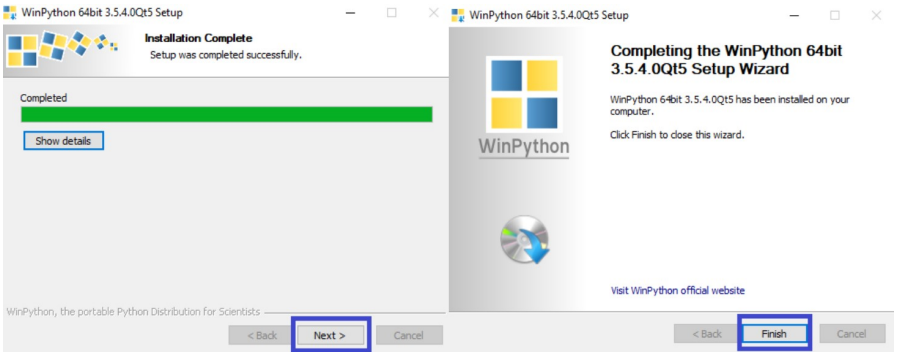
El código del sistema está desarrollado en el lenguaje de programación Python 3, en el entorno Jupyter Notebook. A continuación se detalla el procedimiento para poder ejecutar el código de la implementación explicada en el presente documento.

## B.1 Instalación

En el proyecto se utilizó el paquete de herramientas WinPython[1] en su versión 3.6.8 de 64 bits que trae preinstaladas la mayoría de programas y librerías necesarias. Para su instalación, se aceptan los Términos y Condiciones y se indica la ubicación donde se van a descomprimir los archivos.




[1] https://winpython.github.io/



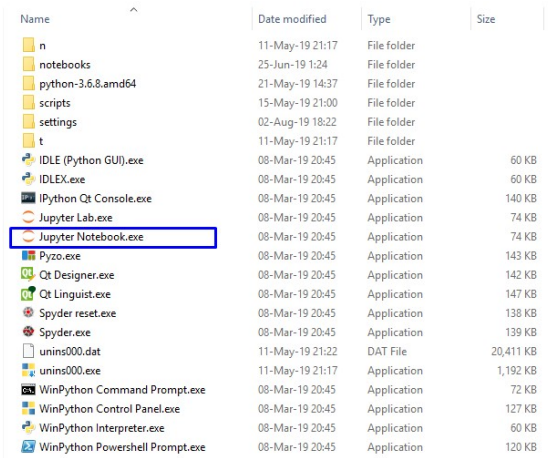

Para instalar las librerías faltantes, se ejecuta WinPython Command Prompt.exe ubicado en la nueva carpeta que se creó en la ubicación previamente indicada, y se ingresa el siguiente comando:

```
pip install tensorflow keras matplotlib math sklearn pandas
   stldecompose
```

## B.2 Ejecución

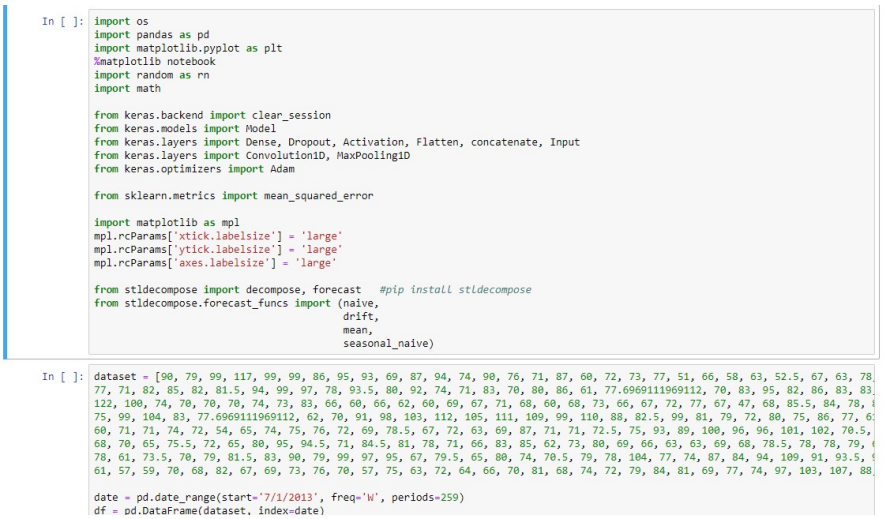
Para ejecutar el código es necesario mover el archivo "Hiperparameters Tunning via PBIL.ipynb" a la carpeta “notebooks” en el directorio de WinPython.

Finalmente para su ejecución es necesario abrir el archivo "Jupyter Notebook.exe".



Se abrirá una ventana en el navegador con el explorador de Jupyter Notebook, donde se debe abrir el archivo del proyecto

La interfaz de Jupyter consta de secciones de código que se pueden ejecutar de forma individual. Se debe ejecutar en orden cada sección utilizando las teclas Shift + Enter.



```
In [ ]: import os
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib notebook
        import random as rn
        import math

        from keras.backend import clear_session
        from keras.models import Model
        from keras.layers import Dense, Dropout, Activation, Flatten, concatenate, Input
        from keras.layers import Convolution1D, MaxPooling1D
        from keras.optimizers import Adam

        from sklearn.metrics import mean_squared_error

        import matplotlib as mpl
        mpl.rcParams['xtick.labelsize'] = 'large'
        mpl.rcParams['ytick.labelsize'] = 'large'
        mpl.rcParams['axes.labelsize'] = 'large'

        from stldecompose import decompose, forecast   #pip install stldecompose
        from stldecompose.forecast_funcs import (naive,
                                                 drift,
                                                 mean,
                                                 seasonal_naive)

In [ ]: dataset = [90, 79, 99, 117, 99, 99, 86, 95, 93, 69, 87, 94, 74, 90, 76, 71, 87, 60, 72, 73, 77, 51, 66, 58, 63, 52.5, 67, 63, 78,
        77, 71, 82, 85, 82, 81.5, 94, 99, 97, 78, 93.5, 80, 92, 74, 71, 83, 70, 80, 86, 61, 77.6969111969112, 70, 83, 95, 82, 86, 83, 83,
        122, 100, 74, 70, 70, 70, 74, 73, 83, 66, 60, 66, 62, 60, 69, 67, 71, 68, 60, 68, 73, 66, 67, 72, 77, 67, 47, 68, 85.5, 84, 78, 8
        75, 99, 104, 83, 77.6969111969112, 62, 70, 91, 98, 103, 112, 105, 111, 109, 99, 110, 88, 82.5, 99, 81, 79, 72, 80, 75, 86, 77, 6
        60, 71, 71, 74, 72, 54, 65, 74, 75, 76, 72, 69, 78.5, 67, 72, 63, 69, 87, 71, 71, 72.5, 75, 93, 89, 100, 96, 96, 101, 102, 70.5,
        68, 70, 65, 75.5, 72, 65, 80, 95, 94.5, 71, 84.5, 81, 78, 71, 66, 83, 85, 62, 73, 80, 69, 66, 63, 63, 69, 68, 78.5, 78, 78, 79, 8
        78, 61, 73.5, 70, 79, 81.5, 83, 90, 79, 99, 97, 95, 67, 79.5, 65, 80, 74, 70.5, 79, 78, 104, 77, 74, 87, 84, 94, 109, 91, 93.5, 8
        61, 57, 59, 70, 68, 82, 67, 69, 73, 76, 70, 57, 75, 63, 72, 64, 66, 70, 81, 68, 74, 72, 79, 84, 81, 69, 77, 74, 97, 103, 107, 88

        date = pd.date_range(start='7/1/2013', freq='W', periods=259)
        df = pd.DataFrame(dataset, index=date)
```

Todo el código se encuentra comentado y estructurado de una forma sencilla de entender para permitir una fácil modificación.

# Bibliografía


[1] Shumeet Baluja, *Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning*, Tech. Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, January 1994.

[2] Raúl Benítez, Gerard Escudero, Samir Kanaan, and David Masip Rodó, *Inteligencia artificial avanzada*, Editorial UOC, 2014.

[3] James Bergstra and Yoshua Bengio, *Random search for hyper-parameter optimization*, Journal of Machine Learning Research **13** (2012), 281–305.

[4] M Ing Paola Britos, *Entrenamiento de redes neuronales basado en algoritmos evolutivos*, Ph.D. thesis, Universidad de Buenos Aires, 2005.

[5] Fernando Caparrini, *Introducción al aprendizaje automático*, 2017.

[6] Uday K Chakraborty and Cezary Z Janikow, *An analysis of gray versus binary encoding in genetic search*, Information Sciences **156** (2003), no. 3-4, 253–269.

[7] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee, *Choosing multiple parameters for support vector machines*, Machine learning **46** (2002), no. 1-3, 131–159.

[8] Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning, *Stl: a seasonal-trend decomposition*, Journal of official statistics **6** (1990), no. 1, 3–73.

[9] Stevenson Contreras and Fernando De la Rosa, *Aplicación de deep learning en robótica móvil para exploración y reconocimiento de objetos basados en imágenes*, 2016 IEEE 11th Colombian Computing Conference (CCC), IEEE, 2016, pp. 1–8.

[10] Chuan-sheng Foo, Chuong B Do, and Andrew Y Ng, *Efficient multiple hyperparameter learning for log-linear models*, Advances in neural information processing systems, 2008, pp. 377–384.

[11] John Fox and Sanford Weisberg, *An r companion to applied regression*, Sage Publications, 2018.

[12] Kunihiko Fukushima, *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*, Biological cybernetics **36** (1980), no. 4, 193–202.

[13] Nikolaus Hansen and Andreas Ostermeier, *Completely derandomized self-adaptation in evolution strategies*, Evolutionary computation **9** (2001), no. 2, 159–195.

[14] Suzana Herculano-Houzel, *The human brain in numbers: a linearly scaled-up primate brain*, Frontiers in human neuroscience **3** (2009), 31.

[15] A. Karpathy, *Convolutional neural networks (cnns / convnets)*, 2017.

[16] Christof Koch, *Biophysics of computation: information processing in single neurons*, Oxford university press, 2004.

[17] Jan Larsen, Lars Kai Hansen, Claus Svarer, and M Ohlsson, *Design and regularization of neural networks: the optimal use of a validation set*, Neural Networks for Signal Processing VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop, IEEE, 1996, pp. 62–71.

[18] Yann LeCun, Yoshua Bengio, et al., *Convolutional networks for images, speech, and time series*, The handbook of brain theory and neural networks **3361** (1995), no. 10, 1995.

[19] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta, *A generalized framework for population based training*, CoRR **abs/1902.01894** (2019).

[20] Ramon Garcia Martinez, *Sistemas autonomos - aprendizaje automatico*, 1997.

[21] Ivan Mendez-Jimenez and Miguel Cardenas-Montes, *Modelling and forecasting of the 222rn radiation level time series at the canfranc underground laboratory*, pp. 158–170, 06 2018.

[22] _____, *Time series decomposition for improving the forecasting performance of convolutional neural networks*, pp. 87–97, 01 2018.

[23] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat, *Evolving deep neural networks*, CoRR **abs/1703.00548** (2017).

[24] Reza Rastegar and Arash Hariri, *The population-based incremental learning algorithm converges to local optima*, Neurocomputing **69** (2006), no. 13-15, 1772–1775.

[25] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams, *Practical bayesian optimization of machine learning algorithms*, Advances in Neural Information Processing Systems 25, 2012, pp. 2960–2968.

[26] Fisher Yu and Vladlen Koltun, *Multi-scale context aggregation by dilated convolutions*, arXiv preprint arXiv:1511.07122 (2015).