eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Diseño e implementación de una librería de neuro-evolución para aprendizaje profundo con Pytorch

Autor

Ignacio Belzunegui Gabilondo

Director

Roberto Santana Hermida

Índice general

Índice de figuras

Índice de tablas

Resumen

La neuro-evolución es una técnica de inteligencia artificial que optimiza los hiperparámetros de redes neuronales empleando algoritmos evolutivos. Se parte de una población inicial de redes y se someten a una serie de transformaciones hasta obtener el resultado deseado o agotar el número de evoluciones establecidas.

El objetivo de este proyecto es la creación de una librería de neuro-evolución para redes neuronales creadas a través del módulo *Pytorch*, la cual ha sido llamada *Evotorch*.

A lo largo de este documento se presentan las bases teóricas sobre las que se sustentan las redes neuronales y los algoritmos genéticos, los cuales conforman el núcleo de la funcionalidad de este proyecto. También se explican los módulos desarrollados, así como su finalidad, los elementos que los conforman y el grado de relación existente entre ellos.

Uno de los principios fundamentales de esta librería es la posibilidad de que el usuario pueda calibrar los hiperparámetros de las redes neuronales de manera manual. El otro es que *Evotorch* obtenga el calibrado que le reporte un resultado óptimo a partir de la aplicación de un algoritmo genético ajustado a sus exigencias. En este trabajo se compararán los resultados que aportan las redes configuradas manualmente por un lado y las obtenidas mediante neuro-evolución en un conjunto de datos reales por otro, pudiendo evaluar el grado en que el algoritmo genético ha ayudado a obtener un resultado mejor.

1. CAPÍTULO

Introducción

Las redes neuronales son ampliamente utilizadas hoy en día para solucionar problemas en los que se requiere que un sistema aprenda de manera automática, como por ejemplo en la clasificación de imágenes o en la detección de patrones. Un gran inconveniente que presentan estas redes es la selección de sus hiperparámetros y su arquitectura.

Existen cientos de factores que pueden determinar qué valores son buenos para un determinado hiperparámetro, y el cambio de valor en uno o varios puede desembocar en un resultado completamente distinto. Una solución a este problema es el empleo de algoritmos neuro-evolutivos, utilizados para evolucionar las arquitecturas que ofrezcan mejores resultados. Estos algoritmos emulan la selección natural, partiendo de un conjunto de posibles soluciones al problema (en este caso, obtener la red neuronal que mejores resultados aporte) representados como arquitecturas de red. Estas soluciones se evolucionan mediante diferentes técnicas (un tema que será abordado en el **Capítulo 2**) hasta que se obtiene el resultado deseado o el algoritmo llega a un determinado número de iteraciones.

Actualmente son muchas las herramientas que existen para desarrollar redes neuronales: *Keras* [30], *Tensorflow* [61], *Theano* [62], etc. Una de estas herramientas es la librería *Pytorch* [47], la cual está obteniendo mucha popularidad últimamente.

Uno de los mayores problemas a los que se enfrentan los desarrolladores de redes neuronales en *Pytorch* es la falta de un módulo que les proporcione los mejores hiperparámetros a sus redes neuronales. A día de hoy tienen que calibrar los valores de dichos hiperparámetros a mano para poder afrontar diferentes problemas con sus redes.

1.1. Objetivo del proyecto

El objetivo de este proyecto ha sido el desarrollo de una librería de neuro-evolución en el lenguaje de programación *Python* que haga uso del formalismo de la librería *Pytorch*. Esta librería, a la cual se ha llamado *Evotorch*, permite la evolución de redes **multicapas** y redes **convolucionales**.

Evotorch ha nacido como una solución al problema expuesto anteriormente. Mediante el empleo de algoritmos genéticos (**Apartado 1.5)** obtiene el mejor calibrado de los hiperparámetros de una red neuronal para un conjunto de datos definidos por el usuario. Esta librería permite crear una red neuronal hecha enteramente en *Pytorch* y evolucionarla de acuerdo a una evolución personalizada.

1.2. Conceptos fundamentales

Las redes neuronales son modelos computacionales inspirados en el funcionamiento del cerebro, más concretamente en las conexiones entre las neuronas. Se trata de capas paralelas de nodos interconectados entre sí con conexiones ponderadas. En la **Figura 1.1** podemos observar su estructura.

Una red neuronal está compuesta de tres partes fundamentales:

1. **Capa de entrada:** Se trata de la primera capa de todas. En esta cada neurona representa el valor de una característica de la instancia que se pasa como entrada.
2. **Capas ocultas:** Se trata de las capas que se encuentran entre la primera y la última capa de la red neuronal. Constituyen la parte de la red donde se llevan a cabo los cálculos que transforman los valores de entrada en valores de salida; en otras palabras, donde la red realiza las operaciones que resultarán en una predicción a partir de unos datos de entrada.
3. **Capa de salida:** Se trata de la última capa de la red, donde se genera la predicción.

En cada neurona se realiza una combinación lineal de los pesos por los valores de entrada. A este resultado se le suma un valor de sesgo y al valor resultante se le aplica una función de activación. Existen diferentes funciones de activación. Algunas de las más representativas y que se utilizan en este proyecto son las siguientes:

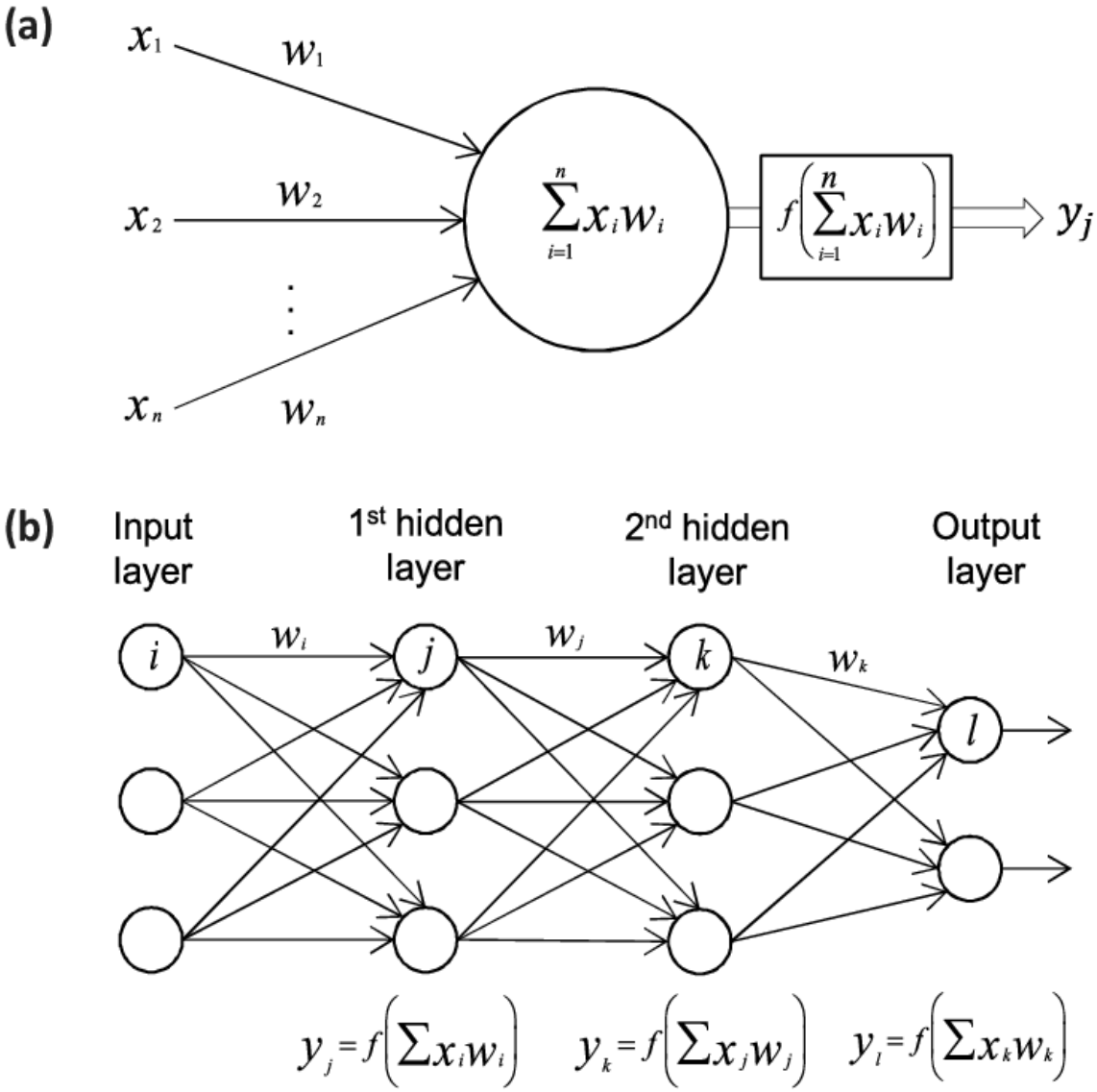

Figura 1.1: Estructura de una red neuronal.
En el apartado *a* podemos observar como se aplica la función de activación a la función de red en una neurona. En el apartado *b* podemos observar las capas y de qué manera se relacionan entre sí. En cada neurona del apartado *b* se lleva a cabo la operación de *a*. Imagen obtenida en [65].

- **Función de activación sigmoide**

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

  Los valores muy grandes se saturan en 1 y los muy pequeños en 0. Sirve para representar probabilidades, las cuales siempre vienen en un rango de 0 a 1.

- **Función de activación Unidad Rectificada Lineal (RELU)**

$$f(x) = \begin{cases} 0 & \text{para } x < 0 \\ x & \text{para } x \geq 0 \end{cases}$$

  Se comporta como una función lineal cuando es positiva y constante a 0 cuando el valor de entrada es negativo.

Si bien estas son las más importantes, *Evotorch* permite emplear más.

De esta manera el resultado que una neurona pasa a la siguiente es:

$$h(x) = g(f_{red}(x)) \tag{1.1}$$

siendo g la función de activación.

La función **h** toma el nombre de **función de decisión**. En la literatura se suele representar con la notación:

$$h(x;\theta,b) = g(\theta^T x + b) \tag{1.2}$$

siendo g la función de activación, θ el vector de pesos y b el sesgo.

Ésta será la representación que tendrá de ahora en adelante. A partir de la función de decisión obtenemos la **función objetivo**. En un problema de regresión podría tomar un valor como este:

$$J(\theta,b) = \sum_{i=1}^{m} (h(x^{(i)};\theta,b) - y^{(i)})^2 \tag{1.3}$$

siendo m el número de elementos o *batch*.

A partir de este concepto nace el *loss function* o función de pérdida, la cual nos indica cuánto se aleja un valor predicho del real, es decir, cuán lejos o cerca ha estado una predicción del valor que debería haber predicho. Cuanto menor sea el valor de esta función mejor habrá sido la predicción, y cuanto mayor sea, peor.

Las implementaciones de la función de pérdida son numerosas y varían en función del problema. En este trabajo se han empleado principalmente dos: **verosimilitud logarítmica negativa** (negative log likelihood) para problemas de **clasificación** y **error medio cuadrático** (mean squared error) para los de **regresión**.

- **Verosimilitud logarítmica negativa**

$$-\sum_{j=1}^{M} y_j \log \hat{y}_j \tag{1.4}$$

  - $\hat{\mathbf{y}}$: Vector con las probabilidades de cada posible clase. El valor de cada posición representa la probabilidad de que se prediga la clase equivalente a su índice.
  - $\mathbf{y}$: Vector que contiene la clase real. La posición cuyo índice corresponde al de la clase real vale 1, mientras que el resto 0 (*one hot-encode)*.
  - **M**: Entero que representa el número de clases.

- **Mean squared error**
  Siendo Z el vector con los valores predichos e Y el vector de los verdaderos valores:

$$ECM = \frac{1}{n}\sum_{i=1}^{n}(Z_i - Y_i)^2 \tag{1.5}$$

Como se puede observar, la función de decisión está compuesta por otras funciones de decisión, por lo tanto podemos representarla mediante la siguiente notación:

$h(x) = g(w_1 h_1(x) + w_2 h_2(x) + ... + w_n h_n(x) + b)$

donde w_i es el peso que pondera la conexión entre la neurona de la capa actual y la neurona de índice i de la capa anterior, el valor que ésta última transmite, $h_i(x)$, es a su vez otra función de decisión (**Ecuación 1.2 y Figura 1.1**), b es el sesgo y g es la función de activación de la capa actual.

Basándonos en los principios recién expuestos cualquier cálculo puede realizarse con los pesos, sesgos en cada neurona y número de neuronas en cada capa adecuados. Es evidente que se presenta un problema fundamental: ¿cómo se puede determinar el peso que ha de tener cada conexión y el sesgo de cada neurona? ¿Cómo podemos saber cuántas neuronas y cuántas capas necesitamos?

Para responder a la primera pregunta se hace uso de dos técnicas interrelacionadas llamadas **Algoritmo de descenso del gradiente y Backpropagation**. Para responder a la segunda, se usa otra llamada **Neural Architecture Search**, la cual constituye el foco principal de este trabajo.

El aprendizaje de una red neuronal consiste en determinar los valores óptimos de sus parámetros (pesos y sesgos). Para ello se utilizan el algoritmo de descenso del gradiente o una extensión de este último llamado *Adam* (Adaptive moment estimation) [55]. Para calcular estos valores óptimos de una manera eficiente se hace uso del algoritmo de propagación hacia atrás (*backpropagation*) [34].

Cabe destacar que a lo largo de este documento se hablará de **hiperparámetros** y **parámetros** de una red neuronal.

- **Parámetros**: Conjunto de atributos de la red neuronal consistente en los pesos y sesgos de las neuronas. Habitualmente el usuario de la red no realiza la calibración de los parámetros de manera directa; con este objetivo se usan los algoritmos de optimización y de *backpropagation*.
- **Hiperparámetros**: Conjunto de atributos que definen la etapa de entrenamiento de una red neuronal. Son los atributos calibrados por los usuarios, aquellos que la red por sí misma no puede modificar (al contrario que los parámetros).

1.3. Tipos de redes neuronales

En este trabajo hemos abordado las dos redes neuronales más empleadas actualmente: las redes neuronales **multicapa** y las **convolucionales**.

1.3.1. Redes neuronales multicapa

Es el tipo de red neuronal más extendido y el más simple. Está compuesto por capas donde todas las neuronas están interrelacionadas (que no intrarrelacionadas) entre sí. Es decir, cada neurona en la capa l está conectada con todas las neuronas de la capa $l+1$, pero no con las neuronas de la capa l.

Es el tipo de red que hemos descrito a lo largo de la introducción de esta sección y es la base desde la que parten las redes convolucionales.

1.3.2. Redes neuronales convolucionales

Las redes neuronales convolucionales suelen ser empleadas con objeto de detectar patrones en imágenes, aunque sus usos son mucho más extensos.

Este tipo de red neuronal son utilizados en problemas de clasificación y regresión cuando se reciben una enorme cantidad de datos de entrada que necesitan ser reducidos, como por ejemplo al recibir los píxeles de una imagen como valores de entrada, los cuales tienden a ser del orden de cientos o miles.

La *Red Neuronal Convolucional* o *Convolutional Neural Network* (CNN) [15] contiene varias capas ocultas dispuestas de manera secuencial. Es decir, las primeras capas detectan patrones simples desde líneas y curvas mientras que las capas más profundas pueden reconocer patrones más complejos como siluetas, caras e incluso patrones mucho más intrincados.

Cabe destacar que por lo general las CNNs suelen utilizarse con imágenes, donde cada píxel representa un valor de entrada. Sin embargo, la utilización de las redes convolucionales se extiende a otros ámbitos. La estructura de los valores analizados por una red convolucional son los tensores (matrices en el caso de imágenes en un solo canal).

Una CNN posee dos tipos de capas:

- **Capas de convolución**: Se encarga de obtener un mapeado de los diferentes filtros en los datos de entrada. En un mapa se establece una relación entre los datos de entrada y los filtros.
- **Capas de pooling**: Obtiene un nuevo mapa a partir de los valores definidos dentro de unas ventanas en el mapa producido en la capa convolucional (pudiendo haber sometido sus datos a una función de normalización o no) cuyo tamaño predefine el usuario (generalmente de tamaño 2). Dividimos el mapa en estas ventanas, y a partir de los valores que éstas posean en su interior conseguiremos una nueva matriz donde un valor (si fuera imagen, píxel) se corresponderá al valor obtenido de haber aplicado uno de los dos métodos posibles de pooling. Los dos métodos son:
  - **Max pooling**: Obtiene el valor máximo en cada ventana.
  - **Average pooling**: Obtiene el valor medio en cada ventana.

  De esta manera, lograremos un mapa de menor tamaño y que conserva las características más importantes.

A continuación procederemos a explicar la convolución y sus aspectos más importantes.

Definimos la convolución como una operación matemática en la que una función se aplica a otra. De esta manera, en una CNN aplicamos un filtro a una matriz para obtener otra donde esté mapeada la aparición del filtro en la misma. Siendo nuestro objetivo hallar patrones en una matriz, en primer lugar definimos los *kernels*: ventanas de un tamaño menor a la matriz a analizar cuyos valores definen los patrones a mapear. Acto seguido, definimos también el *stride*, es decir, de cuanto en cuanto desplazaremos el kernel en la matriz a mapear. El proceso es sencillo, ubicamos en la matriz una ventana con las mismas dimensiones que el *kernel* y llevamos a cabo la siguiente operación:

$$\frac{\sum_{i,j} M_{i,j} * K_{i,j}}{n} \tag{1.6}$$

siendo n el número de elementos dentro del kernel, M la matriz a mapear y K el kernel. Cada nuevo valor del mapa resultante poseerá un valor definido por esta ecuación. Si la ventana de la matriz llega a salirse de la misma (es decir, no abarca todos los valores) llevamos a cabo un proceso de nombre *padding* (relleno), consistente en rellenar los huecos vacíos con 0-s. El número de kernels de cada capa convolucional es un parámetro a definir a la hora de diseñar la red.

Cabe aclarar que una red neuronal convolucional posee los mismos elementos que la red neuronal multicapa, es decir, tiene capas de entrada, ocultas y de salida, y éstas también están completamente conectadas entre sí. Lo que diferencia a una CNN de la multicapa es la existencia de las capas de convolución (*convolution layers*) y agrupación (*pooling layers*). Generalmente estas capas están dispuestas de manera secuencial, en el orden en el cual acaban de ser expuestas. Frecuentemente, se somete cada valor de cada mapa obtenido en la capa de convolución a la función *Relu* a fin de convertir los valores negativos en 0-s. Acto seguido, estos datos normalizados se introducen en la capa de pooling.

Cabe recordar que para que la red neuronal sea capaz de aprender a reconocer patrones (en el caso de una imagen, por ejemplo, siluetas) se necesitará una cantidad de datos (por ejemplo, imágenes) del orden de cientos o miles por clase. Esta magnitud tan grande se debe a que la red tiene que ser capaz de generalizar las características; debe poder abstraer los atributos generales que identifican a una instancia como parte de una clase.

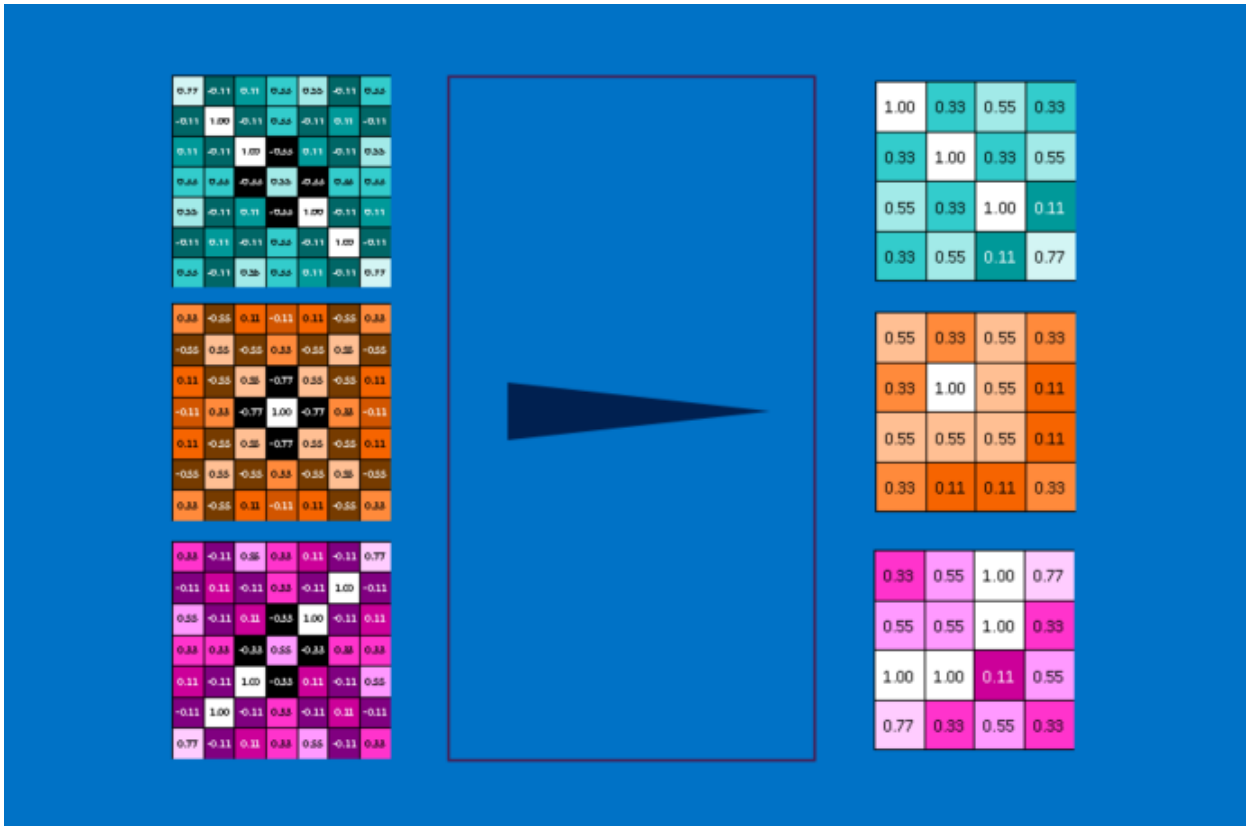

Figura 1.2: Resultado del pooling. Imagen obtenida de [52].

1.4. Pytorch

En esta sección se presentará la librería empleada para crear las redes neuronales, *Pytorch* [47]. Se trata de una librería de aprendizaje automático basada en la librería *Torch*. Posee todas las funciones, clases y atributos necesarios para definir las redes neuronales, trabajar con los gradientes de las mismas, entrenarlas, validarlas, testearlas y actualizar sus pesos y sesgos.

Además, también permite definir los tipos de datos con los que se trabajará, definir las funciones de activación, de optimización y de pérdida, obtener los parámetros de las redes, guardarlas, cargarlas y leer los datos (tanto numéricos como imágenes) con los que se trabajará.

Esta librería proporciona dos elementos de alto nivel que constituyen el eje vertebrador de la misma:

- Tensores (*Tensors*): Representación de la entidad algebraica del mismo nombre.
- Redes neuronales profundas (*deep neural networks*).

1.4.1. Tensores

Se emplean para almacenar y operar con colecciones de números multidimensionales, rectangulares y homógeneos. Su análogo en la conocida librería *numpy* son los arrays, con la particularidad de que los tensores de *Pytorch* pueden hacer uso de la plataforma *CUDA* de *NVIDIA* para realizar operaciones en paralelo. Los tensores son la manera de representar la información numérica dentro de las redes neuronales. La representación de arrays, matrices y números individuales se da en tensores. A continuación se presentarán los módulos principales de la librería.

1.4.2. Principales módulos

- **Autograd**: Proporciona clases y funciones que implementan diferenciación automática de funciones arbitrarias escalares.

  Un elemento guarda las operaciones que se han llevado a cabo para después reproducirlas y poder realizar un conjunto de operaciones hacia atrás (*backward*) y calcular los gradientes.

- **Optim**: Implementa varios algoritmos de optimización utilizados en el aprendizaje de redes neuronales, como Adam o SGD.

- **nn**: Implementa todas las clases, funciones y atributos necesarios para desarrollar redes neuronales (nn significa *neural network*).

1.5. Algoritmos genéticos

Los algoritmos genéticos [24] son uno de los paradigmas de algoritmos evolutivos, los cuales tratan de encontrar soluciones a problemas de búsqueda y optimización imitando a la evolución de las especies en la naturaleza: solo sobrevive el más fuerte, el que mejores resultados aporta. Con este objetivo, los candidatos a ser la solución del problema a tratar, que forman parte de una población, son sometidos a transformaciones (generalmente aleatorias) y a un proceso de selección para determinar el mejor subconjunto de soluciones.

A la hora de afrontar problemas de carácter algorítmico (definidos como secuencias de pasos a seguir para cumplir un objetivo) hacemos uso de objetos/instancias, las cuales están codificadas de una determinada manera, es decir, poseen una serie de atributos con sus respectivos valores. El valor de los parámetros de estos objetos suele determinar el resultado de la solución del problema. Extrapolando este principio a las redes neuronales, una combinación de hiperparámetros puede devolver un resultado mejor que otra, por lo que nuestro objetivo es encontrar la combinación de valores para los hiperparámetros que garantice el entrenamiento óptimo de una red neuronal en el problema definido.

En este tipo de algoritmos disponemos de n posibles soluciones a un problema, donde cada una de ellas toma el nombre de *individuo*. En el caso de la representación de soluciones complejas, como las redes neuronales, cada individuo puede ser una lista u objeto. Al conjunto de posibles soluciones se le llama *población*. Sobre cada individuo en la población se aplica una *función de evaluación*, la cual nos indica la calidad de la solución representada por el individuo en el problema a tratar.

Una vez evaluada toda una población, siempre que no se haya cumplido una condición de parada, esta se somete a una transformación, es decir, **evoluciona**. Esto se da mediante la aplicación de operadores de *mutación*, *cruce* y *selección*, que definiremos más adelante. Las poblaciones, por lo tanto, varían a lo largo del tiempo debido al efecto de estos

operadores; a cada una de estas poblaciones se les llama *descendencia*, y al punto de la evolución en el que se encuentran se le denomina *generación*.

Normalmente el usuario establece cuántas generaciones quiere en su algoritmo genético, debido a que quizás éste jamás llegue a hallar una solución adecuada o para evitar un tiempo de computación excesivo. El objetivo de un algoritmo genético es encontrar el individuo que reporte un mejor resultado en el problema a tratar.

A continuación se repasarán los distintos elementos que conforman a un algoritmo genético:

- **Individuo**: Representación de una posible solución al problema.
- **Población**: Conjunto de soluciones al problema (conjunto de individuos).
- **Descendientes**: Conjunto de soluciones al problema que han sido sometidas a una modificación durante la ejecución del algoritmo.
- **Generación**: Fase dentro de la ejecución del algoritmo genético en la cual se aplican los operadores de selección y recombinación para generar los descendientes de la población actual.
- **Función de evaluación**: Función objetivo utilizada para evaluar la calidad de la solución representada por cada individuo. A este resultado se le llama *valor de adecuación* o *fitness*.
- **Operador de mutación**: Se encarga de modificar una solución de manera aleatoria de acuerdo a un algoritmo previamente determinado.
- **Operador de cruce**: Se encarga de recombinar la información de dos o más soluciones para crear una o más nuevas soluciones.
- **Operador de selección**: Función en la que se determina qué individuos seguirán dentro de la evolución (los más aptos) y cuáles no (los menos aptos).

1.6. DEAP

Distributed Evolutionary Algorithm in Python (*DEAP*) es un framework creado por François-Michel De Rainville, Félix-Antoine Fortin y Marc-André Gardner que incorpora algoritmos evolutivos en *Python* [16]. La principal ventaja de esta librería es que permite crear

a los individuos y a las poblaciones de manera dinámica, mediante un módulo de nombre *creator*, al mismo tiempo que posee estándares de definición de los algoritmos que permiten interactuar de forma complementaria a los objetos definidos por los usuarios y a los predefinidos por el módulo, dando lugar a una modularidad que reduce los errores e incrementa la personalización.

Esta simbiosis entre métodos y estándares predeterminados con los propios nos permite trabajar en los aspectos personalizados de la evolución (como en la definición de los individuos o en las funciones de mutación y cruce) mientras hacemos uso de determinados datos/métodos nativos del módulo *DEAP* para operaciones de carácter más general (como por ejemplo la función de selección, cuyo funcionamiento es independiente de la forma en la que nuestro individuo ha sido construido).

Este framework ha sido empleado debido a que es muy intuitivo y a que permite combinar datos/métodos definidos por los usuarios con los de *DEAP* de forma simple dando pie a una mayor modularización del código.

A continuación se presentarán brevemente sus módulos principales.

1.6.1. Creator

Este módulo permite crear clases personalizadas para emplearlas en los algoritmos genéticos, de manera que se ajusten a los requerimientos del problema. La peculiaridad que presenta este módulo a la hora de generar clases es la posibilidad de crearlas en base a otra. Es decir, que la nueva posea sus características inherentes más las que ya posee la clase base (lo que se conoce como herencia). Generalmente utilizaremos este módulo para crear la clase de un individuo y de una población.

1.6.2. Base

Este módulo proporciona las estructuras necesarias para construir los algoritmos genéticos. Dentro de este módulo están contenidos *Toolbox* y *Fitness*, imprescindibles para llevar a cabo la organización del algoritmo.

- **Toolbox**: Contiene los operadores relativos a la evolución que se emplearán a lo largo de la ejecución del algoritmo genético. Mediante su función *register* podremos registrar una función bajo un alias e introducirle los argumentos como argumentos fijos (definiendo su valor en el momento de declarar la función) o dinámicos

```
(...)
>>> def funcion(var1, var2):
...     return var1 + var2
>>> herramientas = Toolbox() # creamos la instancia de Toolbox
>>> herramientas.register("AliasFuncion", funcion)
>>> resultado = herramientas.AliasFuncion(1,2)
>>> print(resultado)
3
```

Figura 1.3: Ejemplo del uso de *toolbox.register*

(definiendo su valor al implementar la instancia en algún momento del algoritmo). Con *unregister* realizamos el proceso inverso, eliminando de esta manera ese atributo. Cabe destacar que en esta librería, y más al hacer uso de este submódulo, los llamados atributos son realmente funciones, siendo los valores que devuelven los verdaderos atributos. Este concepto es ilustrado en la **Figura 1.3**:

En este caso, *resultado* sería el verdadero atributo puesto que es la materialización de la función *AliasFuncion*, la cual es empleada para definir un atributo.

- **Fitness**: Este submódulo tiene como objetivo definir la función de evaluación, la cual nos indica cuán buena es una solución; su calidad. Nos indica el valor que alcanza el *fitness* (aptitud) de un individuo así como si dicho valor es válido o no.

1.6.3. Tools

Este módulo posee los operadores predeterminados de la librería, empleados para seleccionar un individuo en base a su aptitud, mutarlo, cruzarlo con otro y seleccionar los mejores descendientes, entre otras funciones.

El alto número de operadores genéticos implementados permiten el rápido desarollo de algoritmos evolutivos; entre los operados más empleados, el módulo incluye *tools.clone* y *tools.initRepeat* entre otros.

- ***tools.clone***: Se encarga de clonar los valores de un contenedor a fin de poder guardarlo en otro sin que los elementos de ambos compartan el mismo espacio de memoria y que estos estén en dos espacios diferentes.
- ***tools.initRepeat***: Se encarga de repetir la función que se le pasa como segundo parámetro tantas veces como se le haya indicado en su último parámetro y guardarlo en el contenedor que se le pasa como su primer parámetro.

2. CAPÍTULO

Estado del arte

En esta sección se abordará el estado del arte en la materia concerniente a las competencias de *Evotorch*. Actualmente, debido al aumento del volumen de información que ha de ser tratado y al incremento del uso de las redes neuronales, se están buscando diferentes maneras de obtener estas optimizaciones de la forma más eficiente posible, lo que nos lleva a encontrarnos con diferentes paradigmas para hacer frente a este problema, siendo el más popular el *Neural Architecture Search* [63] [72]. Se conoce como *Neural Architecture Search* a la técnica para la búsqueda automatizada de parámetros e hiperparámetros que proporcionan el funcionamiento óptimo de las redes neuronales.

Partiendo de la premisa de que este proyecto consiste en una librería que permite la creación de redes neuronales y de la evolución de los hiperparámetros de las mismas, se abordará el análisis del estado del arte de dos maneras distintas:

En primer lugar, en base al estudio realizado por Thomas Elsken, Jan Hendrik Metzen y Frank Hutter [63] se describirá la situación del *Neural Architecture Search* atendiendo a tres principios distintos:

1. El espacio de búsqueda: Las diferentes arquitecturas que pueden ser representadas.
2. La estrategia de búsqueda: Los distintos métodos y criterios que emplearemos para explorar el espacio de búsqueda (se ha de tener en cuenta que el número de elementos a elegir suele ser exponencialmente grande, e incluso ilimitado).
3. La estrategia de estimación del rendimiento: Los diversos métodos y principios que

se emplearán para determinar el rendimiento (funcionamiento) de una arquitectura concreta.

Para cada principio, se comentará a qué problemas se han hecho frente y mediante qué técnicas.

En segundo lugar, se hablará acerca de *Evoflow*, la librería de neuro-evolución que ha servido como inspiración para *Evotorch*.

2.1. Neural Architecture Search

Evotorch hace uso de algoritmos genéticos para obtener la arquitectura que mejor se adapta a un problema concreto, sin embargo, el empleo de este tipo de algoritmos no es la única manera que existe para desempeñar la optimización de arquitecturas. Podemos definir la arquitectura de una red neuronal como el conjunto de los siguientes aspectos: el número de capas en la red, el número de neuronas en cada capa y las conexiones que se establecen entre las neuronas de las diferentes capas. Asimismo, definimos la estructura de una red neuronal como la disposición y las reglas de interacción de los elementos dentro de la arquitectura de la red.

A lo largo de esta sección nos referiremos en múltiples ocasiones a los hiperparámetros de una red neuronal: cabe mencionar que este concepto tiene dos acepciones distintas: existen los hiperparámetros de arquitectura y los de entrenamiento.

2.1.1. Espacio de búsqueda

Partiendo de la definición de arquitectura en una red neuronal, definimos el espacio de búsqueda como las arquitecturas que un algoritmo de NAS puede hallar. Cada estructura posee su propio espacio de búsqueda.

El espacio de búsqueda más común es el de la arquitectura de aquellas redes neuronales que siguen el modelo de estructura de cadena (*chain-structure*). En esta estructura las capas de la red (independientemente de su tipo) se disponen de manera secuencial, donde la capa L_i recibe como datos de entrada los datos de salida de la capa L_{i-1}, y transmite como datos de salida los que serán los datos de entrada de la capa L_{i+1}, siendo N el número de capas y $0 < i < N$. Esta es la estructura que siguen las redes neuronales en *Evotorch*. El espacio de búsqueda viene parametrizado por los siguientes elementos:

1. El número máximo de capas (en la amplia mayoría de problemas este número será ilimitado).

2. El tipo de operación que la capa realice (como pooling o convolución).

3. Los hiperparámetros asociados a las operaciones de las capas (como el tamaño del kernel en el caso de una convolución).

Los trabajos más recientes realizados en NAS [9], [19], [73], [20], [50] incorporan elementos de diseño modernos tales como la omisión de conexiones (*skip connections*), las cuales dan lugar a estructuras como las redes de ramas múltiples (*multi-branch networks*). A diferencia de las redes con estructura de cadena, en las redes de ramas múltiples las capas pueden no estar situadas secuencialmente, de forma que una capa L_i puede recibir como datos de entrada los datos de salida de varias capas L_j, y transmitir como datos de salida los que serán los datos de entrada de varias capas L_k donde N es el número de capas, $0 < i < N$, $0 \leq j \leq n-1$ y $i+1 \leq k \leq N$.

Las formas especiales de este tipo de arquitectura son:

1. Las redes neuronales con estructura de cadena (puesto que se puede elegir que cada capa reciba como datos de entrada únicamente los datos de salida de una única capa).

2. Las redes residuales (*residual networks*).

3. Redes densas (*DenseNets*).

Las diferencias entre las estructuras de cadena y de rama múltiple pueden ser apreciadas en la **Figura 2.1**. La estructura de cadena es la forma de red neuronal más común, y es la que emplea *Evotorch*. Un ejemplo de red neuronal de múltiples ramas puede ser encontrado en [2].

2.1.2. Estrategia de búsqueda

Son múltiples las diferentes estrategias que se pueden emplear para explorar el espacio de búsqueda: optimización Bayesiana, métodos evolutivos, aprendizaje por refuerzo y métodos básados en gradiente.

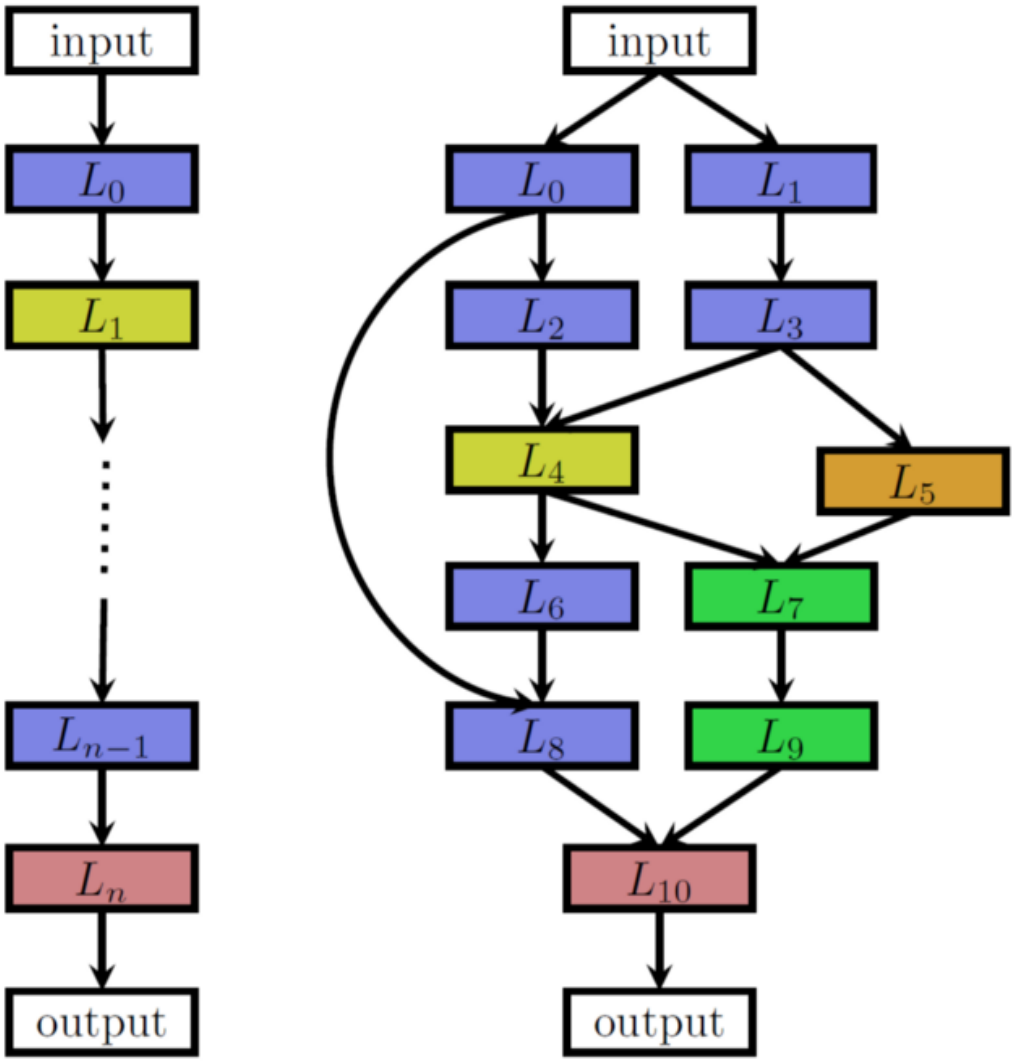

Figura 2.1: Izquierda: Estructura en cadena | Derecha: Estructura de rama múltiple. Imagen tomada de [63].

La optimización Bayesiana [53], [21] es uno de los métodos más empleados para optimizar hiperparámetros. Sin embargo cuenta con un gran inconveniente: se basa en gran medida en procesos Gaussianos y se centra en problemas de optimización contínuos de dimensionalidad pequeña.

La optimización bayesiana ha gozado de cierta popularidad desde 2013 debido a logros en arquitecturas de visión [6], consideradas de las mejores existentes.

Asimismo, mediante este procedimiento se ha obtenido también uno de los mejores rendimientos hasta la fecha en el conjunto de datos *CIFAR-10* [32] sin aumento de datos [18], junto con las primeras redes neuronales calibradas de manera automática para ganar a expertos humanos en tareas específicas [42]. El algoritmo presentado en [68] hace uso de este método.

Otro método muy empleado (el cual de hecho utiliza *Evotorch*) son los métodos evolutivos. Estos hacen uso de algoritmos evolutivos para optimizar las arquitecturas de las redes. La amplia mayoría de los proyectos que hacen uso de este tipo de métodos usan el paradigma de algoritmos genéticos [1], [57], [56] (*Evotorch* lo hace) para obtener las arquitecturas y la propagación hacia detrás (*backpropagation*) para optimizar los pesos. Se tiene constancia de que los algoritmos genéticos ya fueron propuestos con este objetivo a finales de la década de los 80 [44]. Actualmente es muy común emplear estos procedimientos junto a los métodos basados en gradiente [51], [58], [38], [50], [43], [70], [20]; los primeros se utilizan para optimizar la arquitectura de la red mientras que estos últimos se encargan de optimizar los pesos y los sesgos de esa misma red. En el proyecto [26] se presenta un algoritmo que hace uso de esta técnica. Otros algoritmos evolutivos usados en neuro-evolución incluyen programación genética [59], gramática evolutiva [36], [3] y estrategias evolutivas [40].

El aprendizaje por refuerzo (ApR) [7] es otro método que puede ser empleado para la optimización de arquitecturas, donde la generación de una arquitectura es la acción del agente, siendo el espacio de acción igual al espacio de búsqueda. La recompensa del agente consiste en una estimación del rendimiento de la arquitectura calculada a partir de información no vista por la red neuronal previamente. El objetivo del agente es maximizar la recompensa.

A la hora de representar las políticas del agente de un ApR pueden surgir algunos inconvenientes. En algunos problemas [73] se emplea una red neuronal recurrente para muestrear de forma periódica un *string* que a su vez codifica la arquitectura de la red neuronal. Otro enfoque empleado para representar estas políticas es el de formular el algoritmo de *Neu-*

ral Architecture Search como un proceso de decisión secuencial [11]. En el trabajo [17] podemos hallar una implementación de esta técnica.

2.1.3. Estrategia de estimación del rendimiento

El objetivo de las estrategias de búsqueda es obtener las arquitecturas que reporten los mejores resultados con información que la red neuronal no haya utilizado en su entrenamiento (generalmente conjuntos de datos de validación). Para poder llevar a cabo esta acción, se ha de obtener una estimación del rendimiento de la red (valoración de la bondad de los resultados que da). Por lo general el proceso de estimación que se suele llevar a cabo es el de entrenar las redes con conjuntos de datos de entrenamiento, validarlas mediante conjuntos de datos de validación y valorar el resultado que ha devuelto esta validación. Esta es la estrategia que sigue *Evotorch*, puesto que es la más común.

Una de las maneras en las que se puede disminuir el tiempo de estimación es inicializando los pesos en una red basándonos en una red que ha sido entrenada previamente. El morfismo de red (*network morphism*) [67] permite modificar la arquitectura de una red sin modificar ninguna de las funciones que esta emplea para obtener un resultado. De esta manera podemos obtener un rendimiento alto sin tener que entrenar toda la red desde cero. El morfismo de red incluye también emplear pocas épocas para continuar el entrenamiento de la red. En esta librería [33] se hace uso de la optimización Bayesiana junto con el morfismo de red; la explicación teórica del proyecto se halla en [28].

Existe la posibilidad de poder estimar el rendimiento de una arquitectura aprendiendo la curva de extrapolación [60], [18], [31], [4], [49]. En [18] se sugiere extrapolar las curvas de aprendizaje iniciales y acabar aquellos procesos que han predicho un rendimiento insuficiente para poder acelerar la búsqueda.

Otra de las formas en las que se puede reducir el tiempo de estimación es con la búsqueda de arquitectura de red de un capítulo (*One Shot Architecture Search*) [54], [10], [46], [39], [5], [13], [71]. En este método se tratan todas las arquitecturas como grafos de un supergrafo denominado *one shot* donde estos comparten pesos entre arquitecturas que contienen aristas de este supergrafo en común. Solo han de entrenarse los pesos de un solo modelo *one-shot*, de manera que las arquitecturas pueden ser evaluadas sin un entrenamiento diferente entre ellas, por lo que heredan los pesos de este supermodelo *one-shot*. En el proyecto descrito en [25] podemos encontrar una implementación de un tipo de *One Shot Architecture Search*.

2.2. Evoflow

Esta librería [22] ofrece todas las clases, funciones y datos necesarios para evolucionar mediante algoritmos evolutivos los hiperparámetros de redes neuronales profundas (*DNN*), es decir, redes neuronales MLP (MultiLayer Perceptron), CNN (Convolutional Neural Network) y TCNN (Temporary Convolution Network). *Evotorch* se inspira conceptualmente en ella, pues pretende replicar algunas de las funcionalidades de *Evoflow* en *Pytorch*, teniendo en cuenta las particularidades de *Pytorch* y añadiendo cambios en la representación y los operadores genéticos.

Evoflow permite tres modos de aprendizaje:

1. **Entrenamiento simple**: El usuario necesita especificar el problema a tratar simplemente configurando las funciones de pérdida y aptitud.

2. **Aprendizaje semipersonalizado**: Si el tipo de problema es diferente a clasificación o regresión, el usuario puede definir funciones de pérdida y evaluación simples. La función de pérdida ha de ser parte del entorno *Tensorflow*, es decir, el usuario deberá elegir una de las disponibles o implementarla.

3. **Aprendizaje completamente personalizado**: El usuario debe definir las funciones de pérdida y aptitud junto con la construcción del modelo.

Además del aprendizaje de modelos tradicionales, también se permite nuevas estructuras de modelos con múltiples redes. Para ello han de definirse nuevas clases de descriptores y redes.

3. CAPÍTULO

Planificación del proyecto

Este capítulo constará de tres partes diferenciadas: la primera será la explicación de los objetivos del proyecto, la segunda describirá la gestión del tiempo y las tareas, y la tercera presentará el análisis de los riesgos.

3.1. Objetivos del proyecto

El principal objetivo de este proyecto es el desarrollo de una librería de *Python* que implemente un algoritmo de neuro-evolución con el fin de obtener el óptimo modelado de los hiperparámetros y de la arquitectura de una red neuronal brindando compatibilidad con *Pytorch*.

1. En primer lugar se implementarán funciones que a partir de un conjunto de hiperparámetros definan arquitecturas correspondientes a redes neuronales multicapas y convolucionales.

2. En segundo lugar se llevará a cabo el desarrollo de la librería utilizando como base las librerías *Pytorch* y *DEAP*, siguiendo los principios de diseño de la librería *Evoflow*.

3. En tercer lugar se validará la librería en un conjunto de bases de datos con problemas de clasificación y regresión evolucionando arquitecturas que resuelvan de manera eficiente estos problemas.

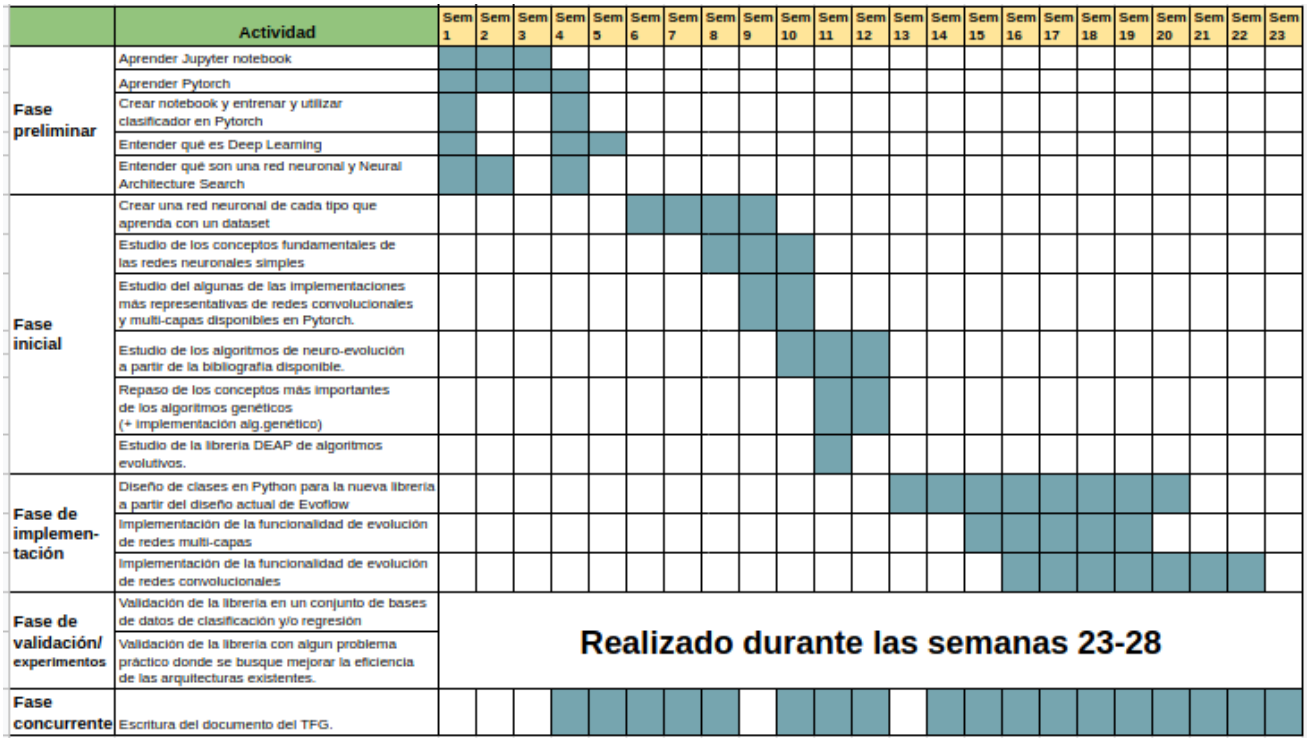

Figura 3.1: Diagrama Gantt con las tareas realizadas entre la semana 1 y la semana 23 (última semana de julio)

3.2. Gestión del tiempo y tareas

En esta sección se incluye un diagrama Gantt (**Figura 3.1**) que abarca las tareas realizadas entre la primera semana y la semana número 23. La razón principal por la que se ha decidido realizarlo de esta manera es debido a la necesidad de exponer un documento que abarque todo el trabajo realizado para este proyecto al mismo tiempo que sea legible y que posea un tamaño adecuado.

Junto al diagrama Gantt podemos encontrar la **Figura 3.2** con la lista de tareas realizadas, las horas que ha llevado terminar cada tarea y las que se habían predicho que tomaría.

3.3. Análisis de riesgos

En este apartado se detallará los principales riesgos a los que está expuesto este proyecto así como sus soluciones.

1. Un riesgo que se podría acentuar con esta situación es la avería del dispositivo

Tareas	*Tiempo real (horas)*	*Tiempo estimado (horas)*
Fase preliminar		
Aprender Jupyter notebook	7	6
Aprender Pytorch	9	6
Crear notebook y entrenar y utilizar clasificador en Pytorch	6	6
Entender qué es Deep Learning	3	6
Entender qué son una red neuronal y Neural Architecture Search	5,5	6
Fase inicial		
Crear una red neuronal de cada tipo que aprenda con un dataset	55	40
Estudio de los conceptos fundamentales de las redes neuronales simples	22	20
Estudio del algunas de las implementaciones más representativas de redes convolucionales y multi-capas disponibles en Pytorch.	8	10
Estudio de los algoritmos de neuro-evolución a partir de la bibliografía disponible.	9	8
Repaso de los conceptos más importantes de los algoritmos genéticos: (+ implementación alg.genético)	15,5	10
Estudio de la librería DEAP de algoritmos evolutivos.	14	10
Fase de implementación		
Diseño de clases en Python para la nueva librería a partir del diseño actual de Evoflow	76	55
Implementación de la funcionalidad de evolución de redes multi-capas	20	30
Implementación de la funcionalidad de evolución de redes convolucionales	22	30
Fase de validación/experimentos		
Validación de la librería en un conjunto de bases de datos de clasificación y/o regresión	24	50
Validación de la librería con algun problema práctico donde se busque mejorar la eficiencia de las arquitecturas existentes.	30	35
Fase concurrente		
Escritura del documento del TFG.	81,5	70
Total	407,5	398

Figura 3.2: Tareas realizadas junto con su tiempo de realización y el tiempo predicho

empleado para trabajar. En este caso se llevaría con la mayor brevedad posible a una tienda para su reparación haciendo uso de un dispositivo auxiliar para sustituirlo durante su ausencia; si el dispositivo averiado no pudiera volver a ser utilizado el resto del proyecto se llevará a cabo en el dispositivo auxiliar.

2. Otro riesgo es la pérdida de la información debido al incorrecto funcionamiento del dispositivo empleado para el trabajo. Para hacer frente a este inconveniente se alojará la información en la nube. Para el código y la documentación se empleará *Dropbox*, para la hoja de cálculo donde se monitereen las horas y tareas *Google Drive* y para la memoria, *Overleaf*. Se realizarán copias de seguridad de las carpetas que constituyen el proyecto tras terminar cada sesión de trabajo en un *pendrive* y en el ordenador donde se trabaja.

3. La ausencia de conjuntos de datos de calidad para llevar a cabo el aprendizaje de las redes neuronales y su posterior validación puede suponer un problema. Este problema ha sido resuelto haciendo uso de las bases de datos que se presentan en el **Capítulo 5**.

4. Los altos requerimientos de tiempo y memoria asociados al costo computacional de la neuro-evolución también suponen un riesgo. Para paliar esta situación se reducirá el volumen de cálculos a realizar ajustando los parámetros evolutivos a valores que resulten en una evolución llevada a cabo en un tiempo razonable.

5. Un riesgo considerable es la limitación temporal a la hora de realizar los experimentos. Para solucionar este problema se reducirá el número de problemas a resolver en los experimentos y se limitará su complejidad.

6. La pandemia del COVID-19 supone un contratiempo en el desarrollo de este proyecto de manera indirecta, debido a que el correcto funcionamiento de la sociedad en sus diferentes formas se ha visto altamente comprometido. Esta situación ha derivado en la demora y suspensión de actividades que apoyan el correcto progreso del proyecto tales como la reparación del ordenador o sus componentes en caso de avería o el mantenimiento de servidores empleados para la comunicación y custodio de los datos. Afortunadamente este incidente no altera de manera significativa el desarrollo del proyecto por lo que se puede seguir con el mismo de la forma prevista. No obstante, las reuniones programadas para llevarse a cabo de manera presencial han debido realizarse virtualmente a través del software de texto, voz y vídeo *Skype*.

4. CAPÍTULO

Diseño de la librería Evotorch

4.1. Principios de diseño

En el desarrollo de la librería se han seguido un conjunto de principios de diseño generales inspirados en la librería *Evoflow*:

- **Modularidad**.
- **Uso de descriptores para representar la arquitectura de una red**.
- **Aprendizaje de la red como parte de la evolución**.
- **Personalización de los hiperparámetros de las red neuronales**.
- **Personalización de la evolución**.
- **Elección personalizada de los hiperparámetros a evolucionar**.

La **Figura 4.1** muestra un esquema con las principales características de *Evotorch*.

4.2. Componentes conceptuales de la librería

A lo largo de la creación de este proyecto son dos los módulos principales que han sido elaborados.

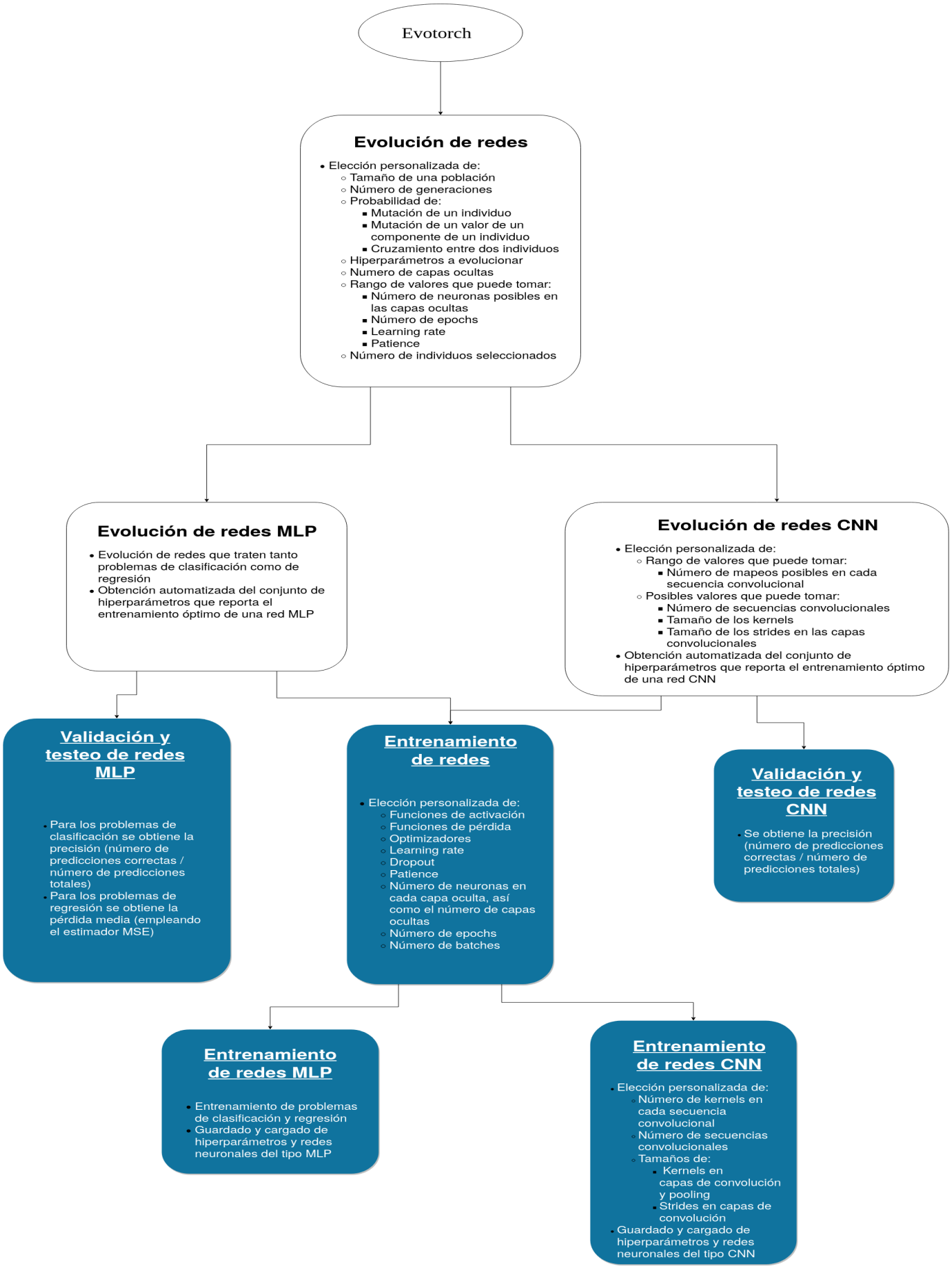

Figura 4.1: Características de *Evotorch*

1. **Network**: Agrupa las clases, funciones y datos relativos a las redes neuronales.
2. **GenAlgs (Genetic Algorithms)**: Contiene las clases, funciones y datos relativos a los algoritmos genéticos.

Son tres los elementos principales que conforman los pilares del diseño de *Evotorch*: los descriptores, las redes y los algoritmos genéticos.

1. **Descriptores**: Describen las arquitecturas de las redes y de los AG (Algoritmos Genéticos). Son empleados para definir aquellos elementos numéricos, booleanos o alfabéticos que, por norma general, son introducidos como argumentos en otros componentes, determinan el tamaño de una iteración o componente, definen el modo en el que un proceso ha de ejecutarse, el número de elementos que un contenedor debe guardar en su interior o el índice de un componente que se pretende utilizar. Existen sendos descriptores para cada tipo de red y AG.
2. **Redes**: Constituyen las implementaciones tangibles de las redes neuronales multicapa y convolucionales. Hacen uso de sus respectivos descriptores para modelar sus hiperparámetros y sus parámetros en *Pytorch*. Se encargan de realizar predicciones en base a unos datos de entrada. Las funcionalidades que incluyen son el entrenamiento y testeo de la red a partir de conjuntos de datos introducidos como datos de entrada. Existen dos tipos de redes: la MLP (MultiLayer Perceptron) y la CNN (Convolutional Neural Network).
3. **AG**: Constituyen las implementaciones tangibles de los algoritmos genéticos. Hacen uso de sus respectivos descriptores para modelar sus parámetros. Se encargan de obtener los valores óptimos de los hiperparámetros de una red neuronal para que ésta pueda hacer la predicción más precisa posible para un dataset determinado. Para llevar a cabo este cometido evolucionan los hiperparámetros definidos por el usuario, siguiendo los criterios paramétricos que éste haya definido.

4.3. Diseño de clases

Antes de comenzar a profundizar acerca del diseño de las clases que componen los módulos de este proyecto, cabe tener presente el esquema que detalla la manera en que se relacionan las principales clases de cada módulo. La **Figura 4.2** visualiza la relación entre los diferentes elementos que componen *Evotorch*. Esta imagen es de gran utilidad pues

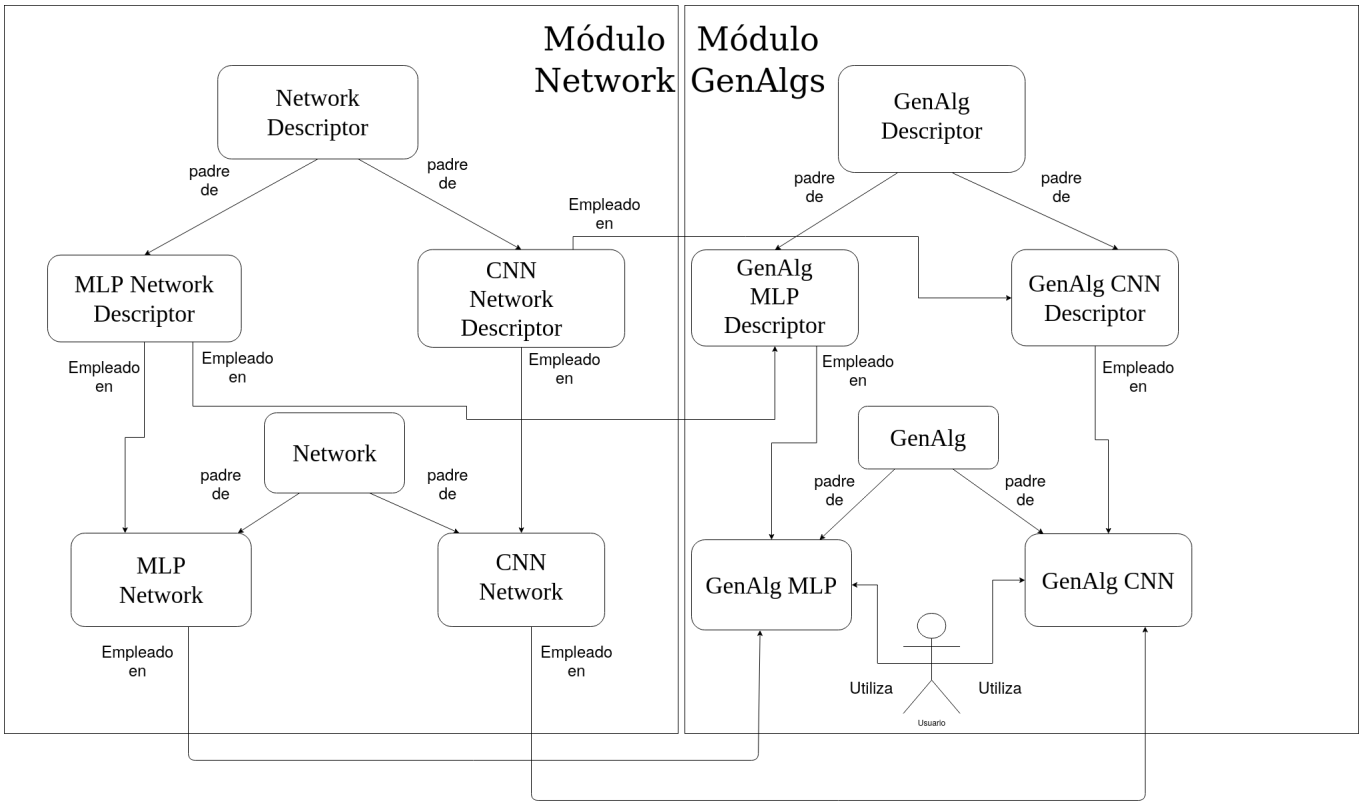

Figura 4.2: Relación entre los elementos de *Evotorch*

nos permite visualizar de manera simple la relación entre el módulo *Network* y el módulo *GenAlgs*. Aunque la figura muestra únicamente los módulos para las redes MLP y CNN *Evotorch* ha sido concebido para incorporar otros tipos de redes neuronales.

Network puede ser utilizado con independencia de *GenAlgs*, es decir, el usuario puede crear redes neuronales haciendo uso del primero sin necesidad de emplear los algoritmos genéticos para ello, sin embargo, la orientación del diseño de *Evotorch* es la evolución de los hiperparámetros de las redes, y es partiendo de este principio como están diseñadas las mismas.

4.3.1. Network

Este módulo es el encargado de implementar una red neuronal haciendo uso de clases, funciones y datos de *Pytorch*. Permite al usuario crear una red neuronal a partir de los datos que él introduzca, así como guardarla o cargar una. *Evotorch* permite crear dos redes neuronales distintas: MLP (*MultiLayer Perceptron*) (**Apartado 1.3.1**) y CNN (*Convolutional Neural Network*) (**Apartado 1.3.2**).

Aclaración de algunas decisiones estructurales del módulo

- En el módulo *Network* existen tres variables globales que definen las funciones de pérdida (*loss functions*), funciones de activación (*activation functions*) y optimizadores (*optimizers*) disponibles.

  Estas variables consisten en diccionarios donde cada posición guarda una función que se puede emplear en algún aspecto concreto de la red neuronal. Cabe mencionar que a lo largo de esta sección a las funciones de pérdida y a los optimizadores se les llama funciones, sin embargo, en *Pytorch* estos son clases, mientras que las funciones de activación sí son funciones.

  El motivo por el que se ha optado por esta decisión ha sido la posibilidad de poder guardar/leer las funciones elegidas por el usuario dentro de un archivo de texto. Para definir qué funciones utilizar el usuario simplemente ha de indicar una palabra (la cual se corresponde a una de las claves de los diccionarios recién expuestos) que designe a la función que desee en el argumento correspondiente en el constructor del descriptor. Al leer el fichero de texto donde los hiperparámetros de la red son guardados la representación de estas funciones se realiza por medio de estas palabras, llamadas **referencias**.

Las variables son:

- *criterion_funcs*: define las funciones de pérdida disponibles y sus referencias, que son: Negative Log Likelihood Loss → *nlll*, Mean Square Error → *mse*, Cross Entropy Loss → *crossentl*.
- *activation_funcs*: define las funciones de activación disponibles y sus referencias, que son: Relu → *relu*, Sigmoid → *sigmoid*.
- *optimization_funcs*: define las funciones de optimización disponibles y sus referencias, que son: Adam → *adam*, Stochastic Gradient Descent → *sgd*.

- Para definir redes de forma precisa en este proyecto se hace uso de la **herencia**, donde una subclase (clase hija) hereda los atributos y métodos de su superclase (clase padre). En el contexto de la programación, una superclase es aquella que define los atributos y métodos que una subclase heredará de ella, con independencia de los atributos y métodos que ésta última ya posea de por sí. En otras palabras, la subclase tiene los mismos atributos y métodos que la superclase además de otros que la clase padre no posee.

- Los datasets empleados para entrenar, validar y testear han de consistir en listas compuestas por sublistas de dos elementos: 1) Un array con los valores correspondientes a los atributos (en formato numérico). 2) Un número correspondiente a la etiqueta adjunta a esos atributos (en formato numérico).

  La representación de los datos, por lo tanto, sería la siguiente:
  [[[atributo, atributo, ..., atributo] , etiqueta] , [[atributo, atributo, ..., atributo] , etiqueta], ..., [[atributo, atributo, ..., atributo] , etiqueta]]

  Cabe destacar que las listas de atributos y las etiquetas han de ser tensores de *Pytorch*, y que en los problemas de regresión resulta importante normalizar los datos.

- En las redes CNN se hace uso de un elemento de *Pytorch* llamado *nn.Sequential*, es decir, una secuencia. En este contenedor se introducen capas de manera secuencial (de ahí su nombre) de forma que los datos de entrada son tratados por la primera capa, cuyos datos de salida pasan a ser los de entrada de la siguiente. Este patrón se repite hasta llegar a la última capa, donde se obtienen sus datos de salida. En este documento nos referiremos a estas secuencias por el nombre de **secuencias convolucionales**. Cabe destacar que una de estas secuencias es la sucesión de una capa de convolución, la aplicación de la función *Relu* y una capa de *MaxPooling* respectivamente, como se muestra en la **Figura 4.3**.

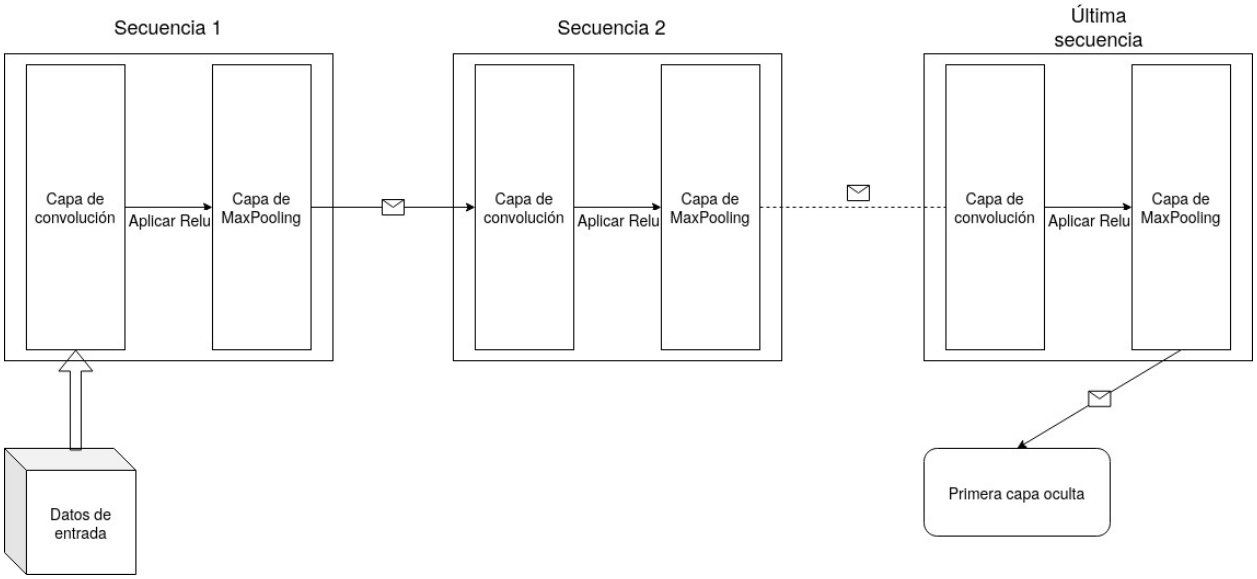

Figura 4.3: Secuencias convolucionales

Clase Network_Descriptor

Todas las redes neuronales, independientemente de si son MLP o CNN, poseen descriptores. Estos atributos definen los hiperparámetros de una red, es decir, variables ajustables de la red neuronal empleadas para definir su entrenamiento.

Dependiendo del tipo de red podemos hallar unos determinados hiperparámetros u otros, sin embargo, algunos son comunes a ambos. En la clase **Network_Descriptor** definimos aquellos hiperparámetros que tanto las redes MLP y CNN comparten, puesto que esta clase es la superclase de las clases *MLP_Descriptor* y *CNN_Descriptor*, las cuales definen los descriptores de las redes MLP y CNN respectivamente.

De esta manera, en *Network_Descriptor* se definen los siguientes hiperparámetros:

- **hidden_fc_layers**: Lista de números enteros que contiene cuantas neuronas poseerá cada capa oculta. De esta manera, su longitud define cuantas capas ocultas tendrá la red.
- **input_dim**: Número entero que indica el número de neuronas en la capa de entrada, es decir, cuantos atributos posee cada instancia.
- **output_dim**: Número entero que indica cuántas neuronas tiene la capa de salida, es decir, cuántas clases tiene que predecir la red en los problemas de clasificación. En los problemas de regresión, al tener que predecir un único valor, vale 1.

- **act_functions_refs**: String o lista de strings que indica qué funciones de activación poseerá cada capa oculta de la red.
    - Se puede utilizar la misma función de activación en todas las capas ocultas indicando "*relu_all*" o "*sigmoid_all*".
    - Si se desean funciones de activación distintas en cada capa, deberá tomar el valor de una lista donde cada posición, que representa la capa oculta correspondiente a su índice, tome valores "*relu*" o "*sigmoid*".
- **batch_size**: Número entero que define el tamaño del batch.
- **dropout**: Lista de números reales en (0,1) donde cada posición de la lista representa el dropout de la capa oculta correspondiente a su índice.
- **learning_rate**: Número real que indica la tasa de aprendizaje de la red.
- **epochs**: Número entero que indica el número de veces que se entrenará todo el conjunto de entrenamiento.
- **optim_ref**: String que indica la referencia de la función de optimización del módulo *torch* que lleva a cabo el algoritmo de optimización.
- **criter_ref**: String que indica la referencia de la función de pérdida del módulo *torch*.
- **patience**: Número entero que indica la paciencia de la red neuronal.

Para cada atributo existe un valor por defecto inicializado por el constructor.

Clase MLP_Descriptor

Esta subclase de *Network_Descriptor* define los descriptores de los Perceptrones multicapa (*MultiLayer Perceptron*) y contiene un atributo adicional al de su clase padre. Esta variable es *mode*.

- mode: String que indica qué tipo de problema tiene que enfrentar la red neuronal: clasificación o regresión. Su valor es "*clssf*" en los problemas de clasificación y "*rgrss*" en los de regresión.

Clase CNN_Descriptor

Esta subclase de *Network_Descriptor* contiene los mismos atributos que los de su clase padre además de cuatro atributos adicionales:

- **conv_layers**: Lista de enteros donde se definen el número de kernels de cada capa convolucional.
- **kernel_sizes**: Lista de listas de enteros de tamaño dos donde se indican el tamaño del kernel en cada capa de convolución y pooling respectivamente. El tamaño de esta lista de listas debe ser menor en una unidad al de *conv_layers*.
- **conv_stride_sizes**: Lista de enteros donde se indica el tamaño del stride en cada capa de convolución. El tamaño de esta lista debe ser menor en una unidad al de *conv_layers*.

Clase Network

Al igual que con los descriptores, tanto las redes MLP como las CNN tienen ciertos atributos en común. En la clase Network se definen los atributos y métodos que tanto las clases *MLP_Network* como *CNN_Network* comparten, por lo tanto es la superclase de las clases que representarán a las redes MLP y CNN.

Cabe destacar que *Network* es a su vez una subclase de *torch.nn.module* (hereda sus atributos), una clase que implementa la base sobre la que se sostiene una red neuronal en *Pytorch*.

Los atributos que posee son los siguientes:

- **descriptor**: Instancia del descriptor, donde están definidos todos los hiperparámetros. Nos permitirá instanciar los objetos necesarios en la construcción de la red.
- **dropout**: Lista con las instancias de las capas de dropout con los porcentajes definidos en el descriptor.
- **act_functions**: Lista con las instancias de las funciones de activación en cada capa oculta.
- **criterion**: Instancia del algoritmo de *loss function* o función de pérdida que ha sido referenciado en el descriptor.

Las función más relevante que posee es: **predict** $\rightarrow$ Devuelve una predicción a partir de los datos de entrada (x).

Clase MLP_Network

Esta subclase de *Network* define todos los atributos y métodos propios de las perceptrones multicapa.

Los atributos propios de esta clase son:

- **hidden_fc_layers**: Son las capas ocultas de la red. El número de neuronas en cada una es obtenido a través del descriptor.
- **output**: Capa de salida de la red neuronal.
- **optimizer**: Instancia del algoritmo de optimización que hemos referenciado en el descriptor.

Los métodos de la clase son los siguientes:

- **forward**: Realiza un *forward-pass* (permite calcular una clasificación/predicción a partir de unos datos de entrada (x)).
- **training_MLP**: Entrena la red neuronal a partir de un conjunto de datos de entrenamiento (*trainloader*) y los hiperparámetros de la misma red.
- **testing_MLP**: Estima el desempeño de la red neuronal a partir de un conjunto de datos de prueba (*testloader*). En los problemas de clasificación devuelve la precisión de la red, mientras que en los problemas de regresión la media de las diferencias entre los valores predichos y los reales (aplicando *MSE*).

Clase CNN_Network

Esta subclase de *Network* define todos los atributos y métodos propios de las redes convolucionales. Los atributos propios de esta clase son:

- **conv_layers**: Lista de secuencias convolucionales. El número de kernels en cada una es obtenido a través del descriptor.

- **_to_linear**: Número entero que indica el número de neuronas que posee la primera capa oculta.
- **hidden_fc_layers**: Son las capas ocultas de la red. El número de neuronas en cada una es obtenido a través del descriptor.
- **output**: Capa de salida de la red neuronal.
- **optimizer**: Instancia del algoritmo de optimización que hemos definido en el descriptor.

Los métodos de la clase son los siguientes:

- **convs**: Aplica las secuencias de convolución, *RELU* y pooling definidos en *conv_layers* a los datos de entrada (x).
- **forward**: Realiza un *forward-pass* a partir de los datos de entrada (x).
- **training_CNN**: Entrena la red neuronal a partir de un conjunto de datos de entrenamiento (*trainloader*) y los hiperparámetros de la misma red.
- **testing_CNN**: Estima el desempeño de la red neuronal a partir de un conjunto de datos de prueba (*testloader*).

4.3.2. GenAlgs

Este módulo es el encargado de implementar un algoritmo genético haciendo uso de clases, funciones y datos de *DEAP*. Permite al usuario crear un algoritmo genético.

Evotorch permite crear dos tipos de algoritmos genéticos distintos: *GenAlg_MLP* y *GenAlg_CNN*.

Al usuario se le permite decidir qué conjunto de hiperparámetros puede evolucionar. Para las redes MLP son: el número de neuronas en cada capa oculta, la tasa de aprendizaje, el número de épocas, la paciencia, el número de capas ocultas, las funciones de activación y los dropouts. Para las redes CNN, además de las mismas que las de las redes MLP, son: el número de secuencias convolucionales, el número de kernels en cada capa convolucional, el tamaño de los kernels en cada capa de convolución y pooling y el tamaño del stride en cada capa de convolución.

Clase GenAlg_Descriptor

Esta clase contiene a los atributos y métodos comunes de los algoritmos genéticos que se dedican a evolucionar tanto redes MLP como con redes CNN. Sus atributos son:

- **toolbox**: Declaración del *toolbox* que se empleará (por defecto es *base.Toolbox*).
- **neuron_min_lim** y **neuron_max_lim**: Respectivamente mínimo y máximo número de neuronas que puede tener una capa oculta.
- **lr_min_lim** y **lr_max_lim**: Respectivamente menor y mayor valor que puede poseer la tasa de aprendizaje.
- **epochs_min_lim** y **epochs_max_lim**: Respectivamente menor y mayor valor que puede poseer el número de épocas.
- **patience_min_lim** y **patience_max_lim**: Respectivamente menor y mayor valor que puede poseer la paciencia.
- **fc_layer_numb_min_lim** y **fc_layer_numb_max_lim**: Respectivamente menor y mayor valor que puede poseer el número de capas ocultas.
- **dropout_min_lim** y **dropout_max_lim**: Respectivamente menor y mayor valor que puede poseer el dropout de una capa.
- **pop_numb**: Tamaño de la población.
- **mutation_prob**: Probabilidad de que se produzca una mutación en un componente de un atributo del individuo.
- **mutation_happen_prob**: Probabilidad de que un individuo sufra una mutación. La diferencia respecto al anterior atributo es que esta probabilidad afecta a todo el individuo, mientras que la anterior afecta a un componente de uno de sus atributos.
- **mutate_all_hparams**: Indica si se desea mutar todos los atributos de un individuo que el usuario ha decidido evolucionar.
- **mate_happen_prob**: Probabilidad de cruzamiento.
- **mate_all_hparams**: Indica si se desean cruzar todos los atributos de un individuo.
- **tournsize**: Tamaño de torneo para la selección.

- **generation_numb**: Número de generaciones.
- **evol_hparams**: Lista de strings que indican qué hiperparámetros se van a evolucionar.
- **wanted_fitness**: Valor de la aptitud que se desea igualar o mejorar.

Clase GenAlg_MLP_Descriptor

Esta subclase de GenAlg_Descriptor define los descriptores de los algoritmos genéticos que evolucionan a las redes neuronales multicapa y contiene un atributo adicional al de su clase padre, que es: **nn_hparameters** $\rightarrow$ Descriptor *MLP_Descriptor* donde se indican qué valores tomarán los hiperparámetros de las redes que serán evolucionadas. Los valores que se definan para los hiperparámetros que serán evolucionados no son relevantes debido a que el valor de los mismos cambiará a lo largo de la ejecución del algoritmo genético.

Clase GenAlg_CNN_Descriptor

Esta subclase de *GenAlg_Descriptor* define los descriptores de los algoritmos genéticos que evolucionan a las redes neuronales convolucionales y contiene seis atributos adicionales al de su clase padre.

Estos atributos son:

- **kernels_min_lim** y **kernels_max_lim**: Respectivamente mínimo y máximo número de kernels en cada secuencia convolucional.
- **possible_kernels_conv** y **possible_kernels_pool**: Listas de enteros que representan los posibles valores que pueden tomar el tamaño del kernel en cada capa de convolución y el tamaño del kernel en cada capa de pooling respectivamente.
- **possible_strides_conv**: Lista de enteros que representa los posibles valores que los strides pueden tomar en cada capa de convolución.
- **nn_hparameters**: Descriptor *CNN_Descriptor* donde se indica el valor de los hiperparámetros que tomarán las redes neuronales. El valor que se definan para los hiperparámetros que serán evolucionados no es relevante debido a que el valor de los mismos cambiará a lo largo de la ejecución del algoritmo genético.

Clase Individual

En nuestra representación cada individuo codifica cada uno de los atributos de un descriptor de MLP tal y como han sido explicados en la **Sección 4.3.1** en el apartado **Clase MLP_Descriptor**.

La clase *Individual_CNN* es una subclase de *Individual* que posee atributos extras relacionados con los hiperparámetros propios de una red convolucional. Los atributos extra se corresponden con aquellos específicos del descriptor de la CNN y explicados en la **Sección 4.3.1** en el apartado **Clase CNN_Descriptor**.

GenAlg

En esta clase se definen los métodos y atributos comunes tanto a los algoritmos genéticos de las redes MLP como de las CNN.

Sus atributos son el descriptor del algoritmo genético (*descriptor*) y el objeto que contiene la implementación de los operadores genéticos en *DEAP* (*toolbox*). *GenAlg* incorpora como único método una función que selecciona el mejor individuo de la población.

Clase GenAlg_MLP

Subclase de *GenAlg* que define todos los atributos y métodos adjuntos a los algoritmos genéticos que se dedican a evolucionar los hiperparámetros de las redes MLP.

No tiene ningún atributo adicional al de su superclase.

Sus métodos son los siguientes:

- **func_mutation_MLP**: Función que se encarga de mutar los componentes de un individuo. La mutación se realiza con una probabilidad dada (*prob*) y existe la opción de mutar todos los atributos usando una variable booleana (*mutate_all_hparams*).
- **func_mate_MLP**: Función que se encarga de cruzar los componentes de los atributos de dos individuos. El cruzamiento se realiza intercambiando los componentes situados en las posiciones que abarcan desde 1 hasta i del primer individuo con los que están en las posiciones que abarcan desde i hasta n del segundo individuo (siendo n el número de componentes del atributo de este último individuo). El valor de

i es aleatorio y posee un valor entre 1 y la longitud del atributo con menor cantidad de componentes. Si el atributo está conformado por un único componente, se intercambian sus valores entre ambos individuos. Cabe mencionar que los atributos cuyos números de componentes han de coincidir son cruzados a la par a partir del mismo índice; este es el caso de *hidden_fc_layers*, *dropout* y *act_functions_ref*, las cuales siempre han de ser del mismo tamaño.

- **execute_NN**: Función que se encarga de entrenar una red neuronal (*network*) con un dataset de entrenamiento (*trainloader*) y de devolver el resultado que ésta red reporta al ser validada con su dataset de validación (*validloader*).

- **func_fitness**: Función que devuelve el *fitness* o aptitud de la red, es decir, el resultado que esta reporta tras tomar como descriptor los hiperparámetros evolucionados de un individuo junto con los predefinidos por el usuario y haber sido entrenada y validada por unos datasets de entrenamiento y validación respectivamente. En primer lugar se instancia una red neuronal cuyos hiperparámetros posean los valores del individuo que se recibe como dato de entrada (*individual*), acto seguido se entrena esta red con el conjunto de entrenamiento recibido por entrada (*trainloader*) y se testea con el conjunto de validación (*validloader*), recibido también como dato de entrada. La función devuelve la precisión (en problemas de clasificación) o pérdida media (en problemas de regresión) de la predicción de la red con las instancias del conjunto de datos de validación. Si se produce un error durante la instanciación de la red, su entrenamiento o su validación, la función devolverá $-\infty$ para un problema de clasificación y ∞ para un problema de regresión. El objetivo de la devolución de estos valores es asegurar que el individuo que produce el error jamás sea considerado como el mejor en la evolución. Se consideran situaciones que podrían desembocar en error que el tamaño del *batch* sea superior al del conjunto de datos de entrenamiento o validación, que el formato de la información sea incorrecto o que falten datos en las instancias de los conjuntos.

- **simple_genetic_algorithm**: Función que se encarga de llevar a cabo la evolución y de escribir en un fichero el conjunto de hiperparámetros que mejores resultados aportan al problema. Esta evolución se realiza en base al calibrado paramétrico establecido por el usuario en la instancia de la clase y en base a los datasets de entrenamiento (*trainloader*) y validación (*validloader*) introducidos. Si el usuario declara un valor *None* para el atributo *wanted_fitness* el algoritmo no poseerá ningún criterio de parada, es decir, se detendrá únicamente tras haber alcanzado el número

máximo de generaciones. Si por el contrario, el valor de *wanted_fitness* es distinto de *None*, entonces la evolución se detendrá cuando la aptitud del mejor individuo sea igual o mejor que la establecida en *wanted_fitness*.

Clase GenAlg_CNN

Subclase de *GenAlg* que define todos los atributos y métodos adjuntos a los algoritmos genéticos que se dedican a evolucionar los hiperparámetros de las redes CNN. No tiene ningún atributo adicional al de su superclase.

Sus métodos son: *func_mutation_CNN*, *func_mate_CNN*, *initialize_NN*, *execute_NN*, *func_fitness* y *simple_genetic_algorithm*, que tienen una funcionalidad similar a los presentados en la sección anterior. A continuación se explican las diferencias fundamentales con los métodos correspondientes de la clase *GenAlg_MLP*:

- **func_mutation_CNN**: Además de mutar los componentes de los atributos comunes a los individuos MLP y CNN, también muta valores exclusivos de los individuos CNN, tales como el número de kernels en cada secuencia convolucional, el número de secuencias convolucionales, los tamaños de los kernels de las capas de convolución y pooling o el tamaño del stride en las capas convolucionales.

- **func_mate_CNN**: Además de cruzar los componentes de los atributos comunes a los individuos MLP y CNN, también cruza aquellos atributos exclusivos de los individuos CNN. El procedimiento es similar al de *func_mate_MLP*, sin embargo, todos los atributos CNN son cruzados, pues el cruzamiento de un único atributo podría resultar en un número de componentes que no se correspondan con la cantidad de componentes que posean el resto de los atributos CNN. Esto se debe a que la lista con el número de kernels en cada secuencia convolucional siempre ha de ser mayor en una unidad a aquellas con los tamaños de los kernels y de los strides.

- **execute_NN**: Posee la misma funcionalidad que su homóloga en *GenAlg_MLP*, con la salvedad de que las redes que se entrenan y validan en esta función son redes CNN.

- **func_fitness**: Esta función realiza la misma función que su homóloga en *GenAlg_MLP* con la diferencia de que la red que se construye es CNN y de que se realiza por medio de un individuo de tipo CNN. Una diferencia sustancial con su versión en *GenAlg_MLP* es que esta función es más propensa a devolver $-\infty$, pues son más

los incidentes que pueden desembocar en errores al tratarse de redes convolucionales. Algunos de estos incidentes son que el número de kernels en cada secuencia sea demasiado grande, que el tamaño del stride sea demasiado alto para una capa convolucional concreta o que el tamaño de un kernel supere a las dimensiones de los datos de entrada.

Si en alguna secuencia convolucional concreta los datos devueltos tuvieran tamaño 1 o fueran menores al tamaño del kernel de convolución o pooling, *Evotorch* eliminaría la última secuencia convolucional (y por extensión la última lista de longitud dos con los tamaños de los kernels y el último número en la lista de strides para cada capa convolucional). Sin embargo, esta funcionalidad no siempre evita que se produzcan los errores mencionados, debido a que el número de kernels, sus tamaños y los de los strides en cada capa convolucional deben estar en sintonía.

- **simple_genetic_algorithm**: Esta función posee la misma funcionalidad que el método con su mismo nombre en *GenAlg_MLP*, con la diferencia de que se evolucionan redes CNN en vez de MLP, y por ende, devuelve los hiperparámetros de una red CNN.

5. CAPÍTULO

Experimentos

En este capítulo se presentan los experimentos que se han llevado a cabo para validar el correcto funcionamiento de *Evotorch*. En primer lugar se explicarán los objetivos principales de esta sección en relación a la totalidad del proyecto, en segundo lugar se hablará acerca de las bases de datos empleadas, en tercer lugar se procederá a explicar el diseño de los experimentos y por último se mostrarán los resultados obtenidos y se comentarán.

5.1. Objetivos

El objetivo de este capítulo es evaluar *Evotorch* como una librería adecuada para obtener el calibrado óptimo de los hiperparámetros de una red neuronal. Esta evaluación se realizará para una serie de problemas (bases de datos) diferentes.

Esta librería se centra en la eficiencia, es decir, en obtener los mejores resultados en el menor tiempo posible. Las redes neuronales encargadas de resolver problemas de clasificación (las redes MLP con modo "*clssf*" y las CNN) devolverán un valor numérico real $p \in [0,1]$ que indica la precisión y que corresponde a la siguiente fórmula:

$$\mathrm{p} = \frac{nc}{n} \tag{5.1}$$

donde nc es el número de instancias correctamente clasificadas, n es el número total de instancias y p es la precisión.

El objetivo es que este valor sea lo más grande posible (idealmente 1).

Las redes neuronales encargadas de resolver problemas de regresión (las redes MLP con modo "*rgrss*") devolverán un valor numérico contínuo que indica la pérdida media y que corresponde a la siguiente fórmula:

$$p_{media} = \frac{\sum_{i=0}^{n}(vp_i - vr_i)^2}{n} \tag{5.2}$$

donde *vp* es el valor predicho, *vr* el valor real, *n* es el número total de instancias y p_{media} es la pérdida media.

El objetivo es que este valor sea lo más pequeño posible (idealmente 0).

5.2. Especificaciones técnicas del hardware empleado

El ordenador con el que se han realizado los experimientos ha sido un *HP Notebook 10* haciendo uso del sistema operativo *Ubuntu 18.04*. Las especificaciones del hardware, obtenidas mediante los comandos de bash *lshw -short* y *lscpu*, son las siguientes:

- **CPU**:
  - Nombre del modelo: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
  - CPU(s): 4
  - Arquitectura: x86_64
  - Caché L1d: 32K; Caché L1i: 32K; Caché L2: 256K; Caché L3: 4096K
- **GPU**: Sun XT [Radeon HD 8670A/8670M/8690M / R5 M330 / M430 / R7 M520]
- **Memoria**:
  - 64KiB L1 caché; 512KiB L2 caché; 4MiB L3 caché; 64KiB L1 caché; 8GiB Memoria de sistema

Pytorch tiene compatibilidad con la plataforma de computación en paralelo *CUDA* de *NVIDIA*, pero al no poseer una tarjeta gráfica de dicha compañía, no se han podido realizar los cálculos en paralelo, lo que ha afectado enormemente a los resultados obtenidos debido a su alto coste computacional y temporal. Debido a ello, los valores de los parámetros evolutivos (como el tamaño de la población, el número de generaciones, el número máximo de épocas posibles o el tamaño de los conjuntos de datos) han tenido que ser reducidos para poder obtener a cambio tiempos de ejecución razonables.

5.3. Validación y bases de datos

En este apartado se describirá el tipo de validación utilizada y se realizarán la descripciones de las bases de datos.

Cabe destacar que se utiliza $\frac{2}{3}$ de los datos para entrenamiento. El resto se divide en partes iguales entre los conjuntos de validación y prueba.

Cabe mencionar que a no ser que se indique lo contrario, no se ha realizado ningún cambio a las bases de datos originales, es decir, no se les ha eliminado/añadido ninguna columna ni se les ha alterado ningún dato.

5.3.1. Bases de datos de experimentos con redes MLP

Para validar las redes MLP se han utilizadado 3 bases de datos para clasificación y 3 para regresión.

5.3.2. Clasificación: Champiñones

Este conjunto de datos incluye descripciones de muestras correspondientes a 23 especies de champiñones en el hongo de la familia *Agaricus* y *Lepiota*, extraídas de la Guía de campo de la Sociedad Audubon de hongos de América del Norte en 1981. Se tienen en cuenta 22 características de los champiñones para determinar si estos son venenosos o no. La base de datos está balanceada y contiene 8124 instancias. Este conjunto de datos puede ser encontrado en [37].

5.3.3. Clasificación: Enfermedad dermatológica

El diagnóstico diferencial de las enfermedades *erythemato-squamous* es un problema importante en los problemas de dermatología. Las enfermedades pertenecientes a este grupo son: *psoriasis*, *seboreic dermatitis*, *lichen planus*, *pityriasis rosea*, *cronic dermatitis* y *pityriasis rubra pilaris*. El objetivo es determinar qué enfermedad padece el paciente entre las 6 disponibles en función de los 33 atributos de su diagnóstico. La base de datos no está balanceada y posee 366 instancias. Este conjunto de datos puede ser encontrado en [27].

5.3.4. Clasificación: Mapeo de tipo de bosque

En este conjunto de datos [29] el problema consiste en la clasificación de diferentes tipos de bosque según la información que aportan sus características espectrales en longitudes de onda infrarojas cercanas a las del espectro visible utilizando imágenes obtenidas por un satélite *ASTER*. La clase obtenida puede ser usada para identificar y/o cuantificar los servicios de ecosistemas (como pueden ser las reservas de carbón o la protección de la erosión) del bosque. Se tienen en cuenta 27 atributos y se analizan 326 instancias. La base de datos no está balanceada. Este conjunto de datos puede ser encontrado en [29].

5.3.5. Regresión: Calidad del aire

En este conjunto de datos [66] se puede hallar las respuestas promediadas por hora de una matriz de 5 sensores químicos de óxido de metal integrados dentro de un dispositivo con sensores químicos que evalúan la calidad del aire. Este artefacto fue colocado en un área contaminada en una carretera dentro de una ciudad italiana. La información fue recogida entre marzo de 2004 y febrero de 2005 representando el más largo registro de calidad de aire mediante sensores químicos existente hasta la fecha. Para poder tratar los datos de manera más eficiente, se han eliminado las columnas *Date* y *Time*, por lo que se han empleado 14 atributos para el análisis de las 9358 instancias existentes. Este conjunto de datos puede ser encontrado en [66].

5.3.6. Regresión: Predicción del área quemada de un bosque

Uno de los mayores problemas medioambientales de la actualidad son los incendios forestales. Uno de los países más afectados por este problema es Portugal, donde 2,7 millones de hectáreas se han quemado entre 1980 y 2005. En esta base de datos [14] se predice el área de bosque quemada en el Noreste de Portugal haciendo uso de la información meteorológica recolectada así como del *Fire Weather Index* canadiense, el cual utiliza cuatro parámetros principales para predecir el riesgo de un incendio descontrolado: temperatura, humedad relativa, lluvia y viento. Para poder tratar los datos de manera más eficiente, se han eliminado las columnas *month* y *day*. El número de instancias de la base de datos es 517, y se han empleado 10 atributos. Este conjunto de datos puede ser encontrado en [14].

5.3.7. Regresión: Bolsa de valores de Estambul

El presente conjunto de datos [8] incluye los retornos de la Bolsa de Estambul para 9 índices internacionales. Los conjuntos de datos incluyen los retornos de la Bolsa de Estambul con los siguientes índices: *ISE* (basado en TL o Lira turca), *ISE* (basado en dolares estadounidenses), *SP*, *DAX*, *FTSE*, *NIKKEI*, *BOVESPA*, *MSCE_EU* y *MSCI_EM*. En esta base de datos se predice el valor del retorno de la Bolsa de Estambul con el índice internacional MSCI de los mercados emergentes (MSCI_EM). Para poder tratar los datos de manera más eficiente, se ha eliminado la columna *Date* y se ha representado los números decimales mediante puntos en vez de comas. El número de instancias que posee este conjunto de datos es de 536, y se han empleado 8 atributos. Este conjunto de datos puede ser encontrado en [8].

5.3.8. Bases de datos de experimentos con redes CNN

Para validar las CNN se han utilizadado 2 bases de datos haciendo uso del módulo *torchvision* [48] que *Pytorch* proporciona. En cada una de ellas se emplearán 1500 instancias en el conjunto de entrenamiento y 200 tanto en los de validación como en los de testeo.

5.3.9. CNN: MNIST

Este conjunto de datos [35] posee imágenes en blanco y negro de dígitos numéricos escritos a mano que van desde el 0 hasta al 9. En esta base de datos se predice el número correspondiente a una imagen a partir de los valores de sus píxeles, siendo las imágenes de tamaño 28*28. Este conjunto de datos puede ser encontrado en el módulo *torchvision*.

5.3.10. CNN: Fashion MNIST

Este conjunto de datos [69] posee imágenes en blanco y negro de artículos de moda agrupados en 10 categorías. En esta base de datos se predice la categoría correspondiente a una imagen a partir de los valores de sus píxeles, siendo las imágenes de tamaño 28*28.

Este conjunto de datos puede ser encontrado en el módulo *torchvision*.

5.4. Diseño de los experimentos

En esta sección se introducirán los principios seguidos para el diseño de los experimentos así como una descripción de las configuraciones de hiperparámetros establecidas para las redes neuronales y los algoritmos genéticos.

En primer lugar se evaluará el desempeño de redes neuronales cuyos parámetros han sido calibrados por un ser humano. Es decir, se simulará que un experto en la materia elige por sí mismo qué combinación de hiperparámetros puede desembocar en el mejor resultado. Se utilizarán combinaciones de hiperparámetros lo suficientemente diferenciadas entre sí para proveer a los experimentos de una mayor diversidad en los casos a evaluar.

En segundo lugar se evaluará la capacidad de *Evotorch* para evolucionar los hiperparámetros para poder obtener resultados óptimos en los diferentes problemas. Cabe destacar que debido a las características del hardware, el volumen de datos a tratar y al coste computacional de las operaciones a llevar a cabo, el tiempo de ejecución no permitirá calibrar ciertos parámetros evolutivos (como el tamaño de la población, el número de generaciones, el límite en el número de épocas, el límite en el número de neuronas, etc.) de forma que reporten los resultados más precisos. Es decir, en este trabajo se reportan los mejores resultados que los recursos disponibles nos permiten obtener, sin embargo, puede darse el caso de que en ciertos experimentos un incremento en la magnitud de ciertos parámetros evolutivos (con el incremento del tiempo de ejecución que tiene aparejado) derive en soluciones mejores que las expuestas en el presente documento. Para cada experimento con configuraciones y redes MLP se definirán 6 redes manuales y 6 configuraciones de las redes a evolucionar distintas, siendo cada una ejecutada 5 veces. En el caso de los experimentos con redes CNN, serán 3 redes manuales y 3 configuraciones distintas, las cuales también se ejecutarán 5 veces cada una.

5.4.1. Redes manuales

Las redes manuales que se emplearán en cada experimento se describen en las **Tablas 5.1-5.3**.

Los valores de los hiperparámetros *input_dim*, *output_dim* y *batch_size* varían en función del problema a tratar.

En el problema *Champiñones input_dim* toma valor 22 y *output_dim* 2, en *Enfermedad dermatológica* poseen 34 y 6 respectivamente y en *Tipo de bosque* 27 y 4 respectivamente.

Redes manuales						
Hiperparámetro	**Red_m_1**	**Red_m_2**	**Red_m_3**	**Red_m_4**	**Red_m_5**	**Red_m_6**
hidden_fc_layers	[15, 20, 25]	[30, 36, 40]	[10, 10]	[17, 17, 20]	[50, 50]	[16, 16, 32, 32]
act_functions	[r, r, s]	[s, r, s]	[r, r]	[r ,s, s]	[s, r]	[s, r, r, r]
dropout	[0.5, 0.5, 0.5]	[0.4, 0.5, 0.6]	[0.5, 0.5]	[0.3, 0.4, 0.5]	[0.5, 0.5]	[0.4, 0.6, 0.8, 0.5]
learning_rate	0.0001	0.001	0.000001	0.00001	0.001	0.01
epochs	10	6	21	8	5	12
optim_ref	adam	sgd	adam	adam	sgd	adam
criter_ref	nlll	crossentl	crossentl	nlll	nlll	crossentl
patience	15	8	10	15	6	10

Tabla 5.1: Redes manuales para los problemas MLP de **clasificación**. En las funciones de activación se usa *r* para *relu* y *s* para *sigmoid*.

Redes manuales						
Hiperparámetro	**Red_m_1**	**Red_m_2**	**Red_m_3**	**Red_m_4**	**Red_m_5**	**Red_m_6**
hidden_fc_layers	[15, 20, 25]	[30, 36, 40]	[10, 10]	[17, 17, 20]	[50, 50]	[16, 16, 32, 32]
act_functions	[r, r, s]	[s, r, s]	[r, r]	[r ,s, s]	[s, r]	[s, r, r, r]
dropout	[0.5, 0.5, 0.5]	[0.4, 0.5, 0.6]	[0.5, 0.5]	[0.3, 0.4, 0.5]	[0.5, 0.5]	[0.4, 0.6, 0.8, 0.5]
learning_rate	0.001	0.01	0.00001	0.0001	0.01	0.1
epochs	10	6	21	8	5	12
optim_ref	adam	sgd	adam	adam	sgd	adam
patience	15	8	10	15	6	10

Tabla 5.2: Redes manuales para los problemas MLP de **regresión**. En las funciones de activación se usa *r* para *relu* y *s* para *sigmoid*. El valor de *criter_ref* en todas las redes es *mse*.

En los problemas *Calidad del aire*, *Predicción del área quemada en un bosque* y *Bolsa de valores de Estambul input_dim* toma valores 14, 10 y 8 respectivamente, mientras que *output_dim* vale 1.

En los problemas *Champiñones* y *Calidad del aire* el *batch_size* en las redes tiene valor 40, en los problemas *Enfermedad dermatológica* y *Tipo de bosque* valor 4, mientras que en *Predicción del área quemada en un bosque* y *Bolsa de valores de Estambul* 8.

Redes manuales			
Hiperparámetro	**Red_m_1**	**Red_m_2**	**Red_m_3**
conv_layers	[1,16,32]	[1,32,64]	[1,8,16,32]
kernel_sizes	[5,2],[5,2]	[3,2],[3,2]	[5,2],[5,2], [7,2]
conv_stride_sizes	[1,1]		[1,1,1]

Tabla 5.3: Redes manuales para los problemas de clasificación de imágenes. El *batch_size* para las tres redes es 20, y los valores de los hiperparámetros compartidos con las redes MLP de clasificación son los mismos que los correspondientes a esas redes.

5.4.2. Configuraciones de los parámetros no evolucionados

Las configuraciones que se utilizarán en los experimentos con redes MLP pueden hallarse en la **Tabla 5.4**.

Las configuraciones 1, 2 y 3 tanto en MLP como en CNN comparten los mismos valores para sus atributos comunes.

Las tres configuraciones de las CNN comparten los mismos valores para los parámetros *kernels_min_lim* y *kernels_max_lim*, *possible_kernels_conv*, *possible_kernels_pool*, *possible_strides_conv* y *possible_conv_layers_sizes*: (10,150), [3,5,7], [2], [1,2] y [3,4,5] respectivamente. Los hiperparámetros a evolucionar de *evol_hparameters* son: [hddn_fc_lyrs, lr, epcs, pat, hddn_fc_lyrs_sz, act_fncs, drpt, cnvl_lyrs, krnl_szs, cnv_strd_szs] para la primera configuración, [hddn_fc_lyrs, epcs, drpt, cnvl_lyrs, krnl_szs] para la segunda y [hddn_fc_lyrs, lr, cnvl_lyrs, cnv_lrys_sz] para la tercera.

5.5. Resultados de los experimentos

En esta sección se presentarán los resultados de los experimentos atendiendo a 4 cuestiones fundamentales:

1. Importancia de la optimización de los hiperparámetros de las redes.
2. Diferencias en el grado de dificultad.
3. Desempeño de los algoritmos evolutivos.
4. Eficacia de la librería.

Configuraciones						
Hiperparámetro	**Evol_1**	**Evol_2**	**Evol_3**	**Evol_4**	**Evol_5**	**Evol_6**
neuronas por capa	(1, 50)	(50, 100)	(20, 30)	(100, 200)	(30, 60)	(1, 50)
lr	(0.00001, 0.1)	(0.0001, 0.001)	(0.0000004, 0.01)	(0.000001, 0.3)	(0.00002, 0.001)	(0.00001, 0.1)
epochs	(12, 40)	(3, 13)	(2, 50)	(1, 30)	(4, 40)	(10, 50)
patience	(5, 10)	(3, 13)	(7, 14)	(5, 18)	(4, 20)	(6, 15)
Nº capas ocultas	(2, 3)	(1, 3)	(3, 4)	(2, 3)	(2, 3)	(1, 2)
dropout	(0.2, 0.9)	(0.4, 0.8)	(0.3, 0.7)	(0.2, 0.9)	(0.5, 0.7)	(0.4, 0.9)
prob mutación	0.4	0.5	0.5	0.4	0.3	0.4
prob ocurr mutación	0.5	0.6	0.4	0.5	0.7	0.5
mutar todos	False	False	True	False	False	False
prob ocurr cruzamiento	0.2	0.4	0.2	0.5	0.1	0.3
cruzar todos	False	True	False	False	False	False
Tamaño torneo	3	4	3	4	3	3
Evol. hidden_fc-layers	✓	✓	✓	✗	✓	✓
Evol. lr	✓	✗	✓	✓	✓	✓
Evol. epocs	✓	✓	✗	✓	✗	✓
Evol. patience	✓	✗	✗	✗	✓	✗
Evol. Nº capas ocultas	✓	✗	✗	✗	✓	✗
Evol. act_functions	✓	✗	✗	✓	✗	✓
Evol. dropout	✓	✓	✗	✗	✓	✗

Tabla 5.4: Configuraciones para las redes MLP. Los valores de los parámetros *toolbox*, *pop_numb*, *generation_numb* y *wanted_fitness* son: base.Toolbox, 20, 30 y None respectivamente para todas las configuraciones.

5.5.1. Importancia de la optimización de los hiperparámetros de las redes

En la **Sección 5.4.2** se muestran los valores de las configuraciones de los diferentes algoritmos evolutivos (**Tabla 5.4**), así como los valores que toman por defecto los hiperparámetros de las redes que no serán evolucionados dentro de dichos algoritmos (**Tablas 5.1-5.3**). Las **Figuras 5.1-5.8** muestran la media de la mejor solución obtenida en cada generación de los diferentes algoritmos (izquierda) y la comparación con las redes manuales en el conjunto de prueba (derecha). Las **Figuras 5.1-5.3** muestran los resultados obtenidos en los problemas de clasificación MLP, las **Figuras 5.4-5.6** los resultados obtenidos en los problemas de regresión MLP, y las **Figuras 5.7 y 5.8** los obtenidos en los problemas CNN.

Una configuración adecuada de estos valores puede desembocar en redes cuyo desempeño es óptimo, como puede observarse en las **Figuras 5.1-5.8**.

Sin embargo, un calibrado incorrecto de estos valores puede producir unas redes que, pese a haber sido sometidas a configuraciones durante 30 generaciones a lo largo de 5 ejecuciones distintas con poblaciones de 20 individuos, no devuelvan valores mejores que los de las redes calibradas a mano. Este es el caso de la configuración 5 en los problemas *Calidad del aire* (imagen de la derecha en la **Figura 5.4**) , *Predicción del área quemada de un bosque* (imagen de la derecha en la **Figura 5.5**) y *Bolsa de valores de Estambul* (imagen de la derecha en la **Figura 5.6**), de la configuración 6 en el problema *Predicción del área quemada de un bosque* y de la configuración 2 en el problema *Fashion MNIST* (imagen de la derecha en la **Figura 5.8**). El hecho de que en estos problemas haya habido redes evolucionadas que hayan obtenido resultados mejores que sus homólogos manuales es un indicio de que la optimización evolutiva de los parámetros de las redes juega un papel determinante a la hora de obtener resultados adecuados.

Estas soluciones nos demuestran que es de vital importancia tener en cuenta qué valores han de definir la configuración a fin de que ciertos atributos (como el número de neuronas, la tasa de aprendizaje, el número de capas ocultas o la paciencia) no sean demasiado grandes o pequeños, pues pueden afectar totalmente en el aprendizaje de las redes, y por consiguiente, en su desempeño.

5.5.2. Diferencias en el grado de dificultad

Los resultados obtenidos muestran que no todos los conjuntos de datos determinan problemas con la misma dificultad. Los problemas de clasificación cuentan con conjuntos de datos con una muy reducida cantidad de valores atípicos (*outliers*); estos son números que se alejan mucho de la media dentro del conjunto de números (en el caso de conjuntos de datos, columnas/atributos) donde están presentes.

No es el caso de los conjuntos de datos de los problemas de regresión, donde existe una mayor cantidad de valores atípicos que determinan que la varianza entre los atributos sea mayor, y por ende, exista una menor cohesión entre la información. Esto produce que los entrenamientos tengan una mayor propensión al sobreajuste u *overfitting*, y por tanto, que la predicción de los valores tienda a un rango de valores concretos.

Esto es claramente visible en los resultados de la evolución de las redes que se utilizan para problemas de regresión, donde se puede observar claramente el efecto del sobreajuste en los algoritmos evolutivos 1, 2, 3, 4, y 6 (imágenes de la izquierda en las **Figuras 5.4-5.6**) y en el algoritmo evolutivo 2 en las **Figuras 5.7 y 5.8**.

5.5.3. Desempeño de los algoritmos evolutivos

Pese a la alteración producida por el sobreajuste en los problemas de regresión, los resultados obtenidos en cada conjunto de datos son los adecuados, pues la configuración de las redes ha proporcionado resultados mejores a los de las redes manuales cuando los valores de los algoritmos genéticos han sido adecuadamente calibrados (imágenes de la derecha en las **Figuras 5.1-5.8**). Por otra parte, en cada generación las mejores aptitudes se han mantenido o han mejorado respecto a las anteriores. Este hecho demuestra de manera empírica y fehaciente que el enfoque evolutivo permite obtener mejores configuraciones de los hiperparámetros en base a los parámetros evolutivos que le han sido introducidos, y que en cada generación los individuos son al menos tan buenos como los de la generación anterior, si no mejores. Este principio es el pilar fundamental sobre el que se fundamenta *Evotorch*, y como se puede comprobar en las imágenes de la izquierda en las **Figuras 5.1-5.8**, se cumple correctamente. Las ejecuciones han permitido además evaluar de manera sistemática que *Evotorch* es estable en tanto no han ocurrido errores de ejecución.

5.5.4. Eficacia de la librería

Los resultados obtenidos nos permiten confirmar que la librería desempeña su función adecuadamente, y que, con conjuntos de datos debidamente tratados, puede encontrar el calibrado de hiperparámetros para una red neuronal que desemboque en un entrenamiento óptimo por parte de la misma, como puede ser analizado en las imágenes de la izquierda en las **Figuras 5.1-5.8**.

Nuestros experimentos han demostrado que *Evotorch* puede ser empleado por los usuarios para hallar de forma eficaz los hiperparámetros que un mejor aprendizaje proporcionan en el marco del problema que se esté tratando, lo que constituye el foco principal de este proyecto.

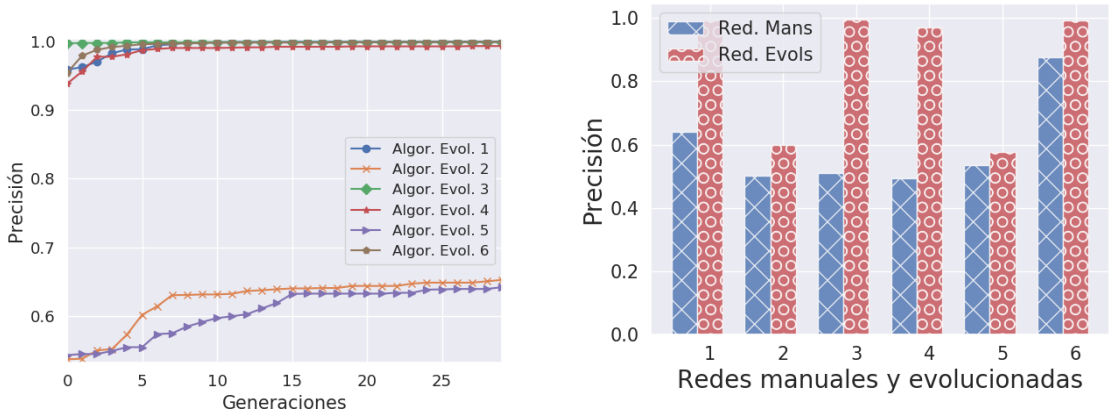

Figura 5.1: Izquierda: Evolución de la mejor precisión a lo largo de las generaciones. Derecha: Comparación precisiones redes manuales y evolucionadas. Problema: *Champiñones*.

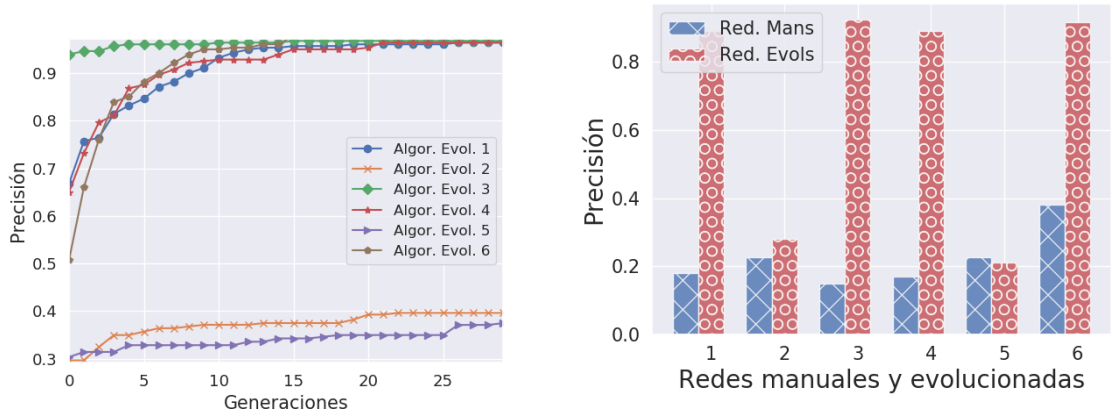

Figura 5.2: Izquierda: Evolución de la mejor precisión a lo largo de las generaciones. Derecha: Comparación precisiones redes manuales y evolucionadas. Problema: *Enfermedad dermatológica*.

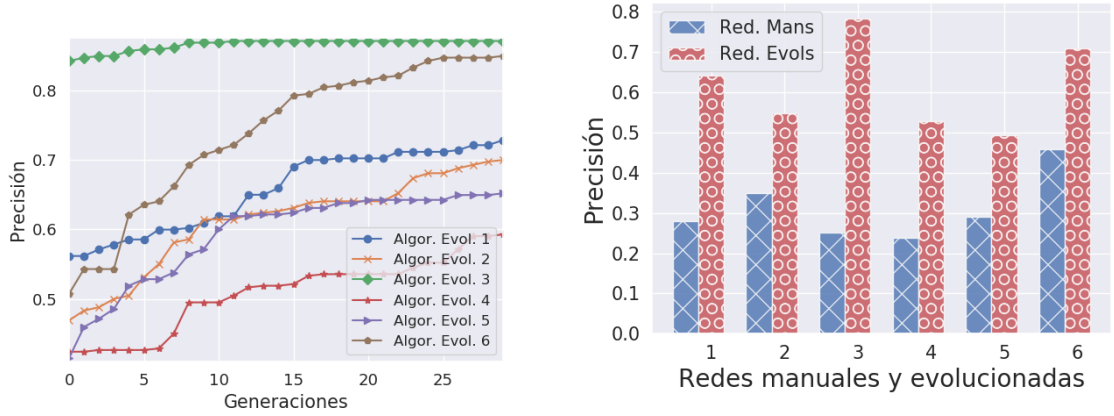

Figura 5.3: Izquierda: Evolución de la mejor precisión a lo largo de las generaciones. Derecha: Comparación precisiones redes manuales y evolucionadas. Problema: *Tipo de bosque*.

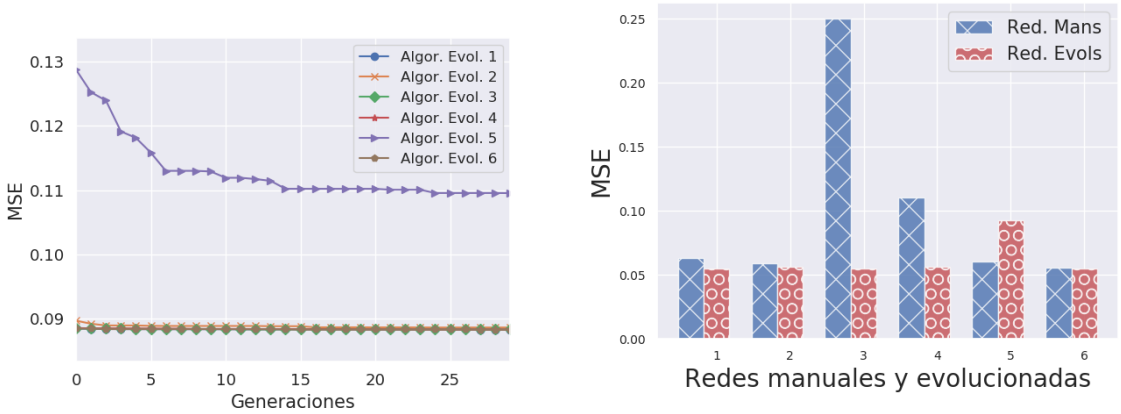

Figura 5.4: Izquierda: Evolución de la mejor MSE a lo largo de las generaciones. Derecha: Comparación MSEs redes manuales y evolucionadas. Problema: *Calidad del aire*.

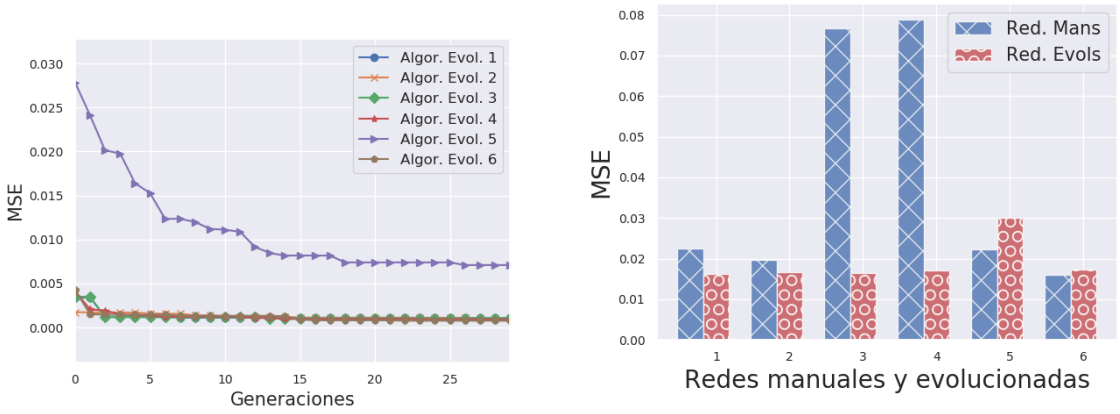

Figura 5.5: Izquierda: Evolución de la mejor MSE a lo largo de las generaciones. Derecha: Comparación MSEs redes manuales y evolucionadas. Problema: *Predicción área quemada bosque*.

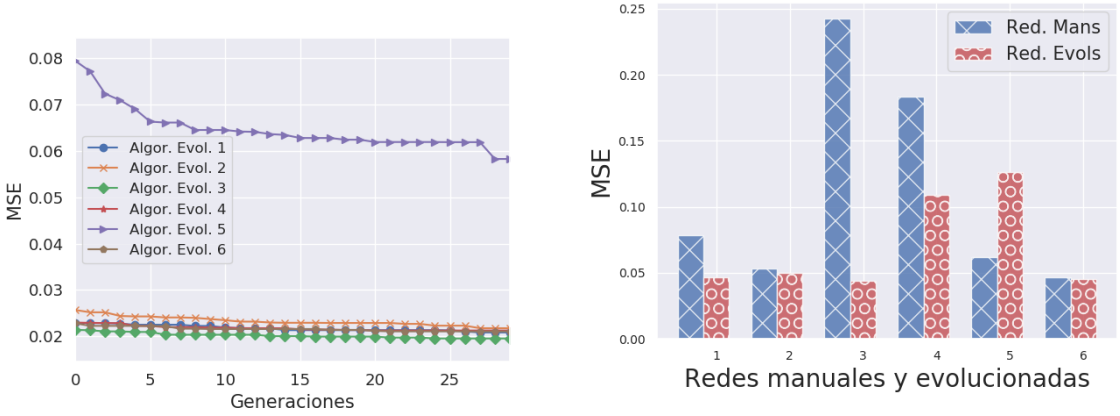

Figura 5.6: Izquierda: Evolución de la mejor MSE a lo largo de las generaciones. Derecha: Comparación MSEs redes manuales y evolucionadas. Problema: *Bolsa de Valores Estambul*.

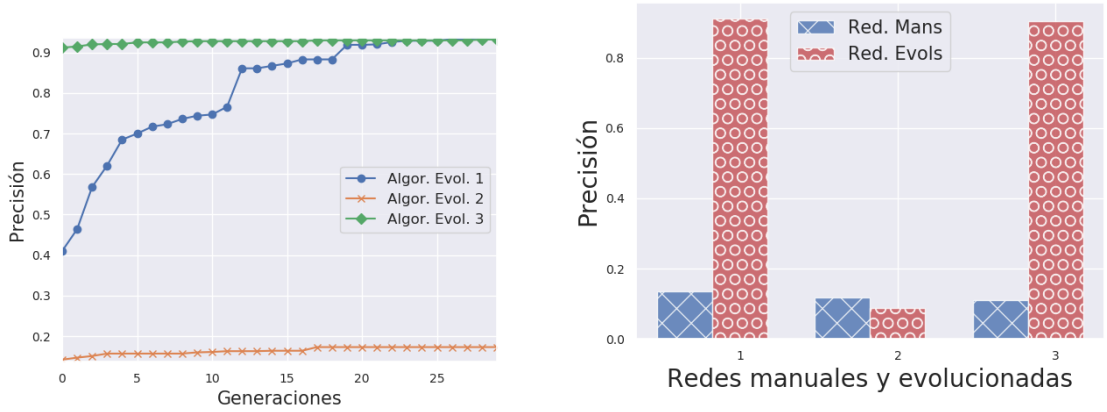

Figura 5.7: Izquierda: Evolución de la mejor precisión a lo largo de las generaciones. Derecha: Comparación precisiones redes manuales y evolucionadas. Problema: *MNIST*.

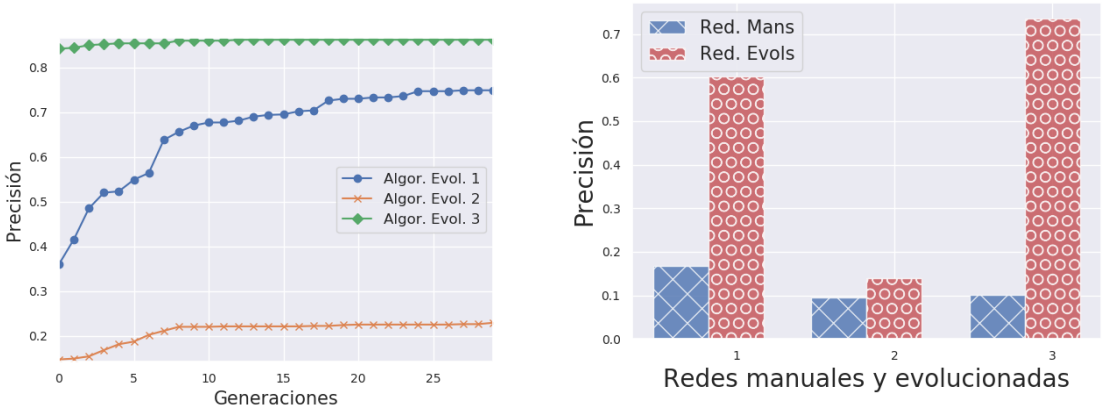

Figura 5.8: Izquierda: Evolución de la mejor precisión a lo largo de las generaciones. Derecha: Comparación precisiones redes manuales y evolucionadas. Problema: *Fashion MNIST*.

6. CAPÍTULO

Conclusiones

En este capítulo se presentarán las conclusiones extraídas a partir de la creación y evaluación de la librería de neuro-evolución *Evotorch*.

En primer lugar se ha llevado a cabo un estudio acerca de los diferentes algoritmos de neuro-evolución que existen a día de hoy (**Capítulo 2**), habiendo elegido el paradigma de los algoritmos genéticos como la mejor manera de evolucionar las redes neuronales. Este objetivo ha sido llevado a cabo mediante la creación del módulo *GenAlgs* (**Capítulo 4**) asistido por *DEAP*. En segundo lugar y en relación a este objetivo, se ha realizado una materialización de las redes neuronales haciendo uso de los aspectos más eficaces del módulo *Pytorch*, enfocándolo desde la perspectiva de la personalización y de la posibilidad de evolucionar de manera sencilla y eficaz los aspectos fundamentales que conforman a una red y a su aprendizaje. Este propósito ha sido cumplido mediante la creación del módulo *Network* (**Capítulo 4**).

Así mismo, para asegurar el correcto funcionamiento de la librería y de que ésta es capaz de resolver problemas reales, se han realizado experimentos (**Capítulo 5**) que han mostrado que la evolución de una serie de redes neuronales ha desembocado en la solución de los problemas planteados con una precisión adecuada. En esta línea, cabe remarcar que las limitaciones en el hardware y la propia naturaleza de los conjuntos de datos (los cuales han de pasar por un pre-procesado debido a que son datos medidos por terceros y por tanto la relación entre los datos puede no ser del todo lineal) han limitado algunos resultados en los experimentos. Un pre-procesado de los datos podría haber sido recomendable, pues podría haber ayudado sustancialmente a un aprendizaje más correcto por parte de las

redes; sin embargo, el tiempo que ésto hubiese requerido habría sido muy grande. Además, las redes evolucionadas han demostrado solventar los problemas expuestos en los experimentos. De esta manera han demostrado que son capaces de trabajar adecuadamente con cualquier base de datos debidamente tratada. Por ello, podemos concluir que los experimentos han sido un éxito, pues muestran como la evolución de los hiperparámetros resulta finalmente en una red neuronal que ofrece un resultado óptimo.

El mayor inconveniente a la hora de realizar este proyecto han sido las limitaciones en el hardware: al tratarse de datos muy voluminosos los tiempos de ejecución han sido muy altos a la hora de realizar los experimentos, probar las evoluciones y probar las redes, lo cual sin duda ha afectado al tiempo de realización del presente proyecto. Dos soluciones complementarias entre sí se presentan para afrontar esta cuestión: en primer lugar hacer uso de *CUDA* para dotar a *Evotorch* de paralelismo (lo cual no ha podido llevarse a cabo debido a la no tenencia de una GPU de marca *NVIDIA* y por tanto a la imposibilidad de testar su funcionamiento) y en segundo lugar emplear *Google Colab*.

Otro aspecto que podría haber ayudado a examinar el funcionamiento de la librería de manera más eficaz es el empleo de bases de datos de mayor volumen (pues cuantos más datos posea una red neuronal para aprender, validar y testear, mayor serán las precisiones de sus resultados), pero los problemas recién expuestos no nos han permitido poder introducir bases de datos de mayor tamaño.

Una posible mejora para la librería sería la introducción del modo de regresión para las redes convolucionales, o el empleo de más paradigmas de neuro-evolución para llevar a cabo la evolución de las redes neuronales. Sin embargo las limitaciones temporales han impedido el desarrollo de estas funcionalidades. No obstante, la arquitectura modular de *Evotorch* permite a cualquier usuario introducir por su cuenta cualquier nueva característica que considere conveniente de manera complementaria a las ya existentes.

Este trabajo ha contribuido a la adquisición de diversos conocimientos en el área del aprendizaje profundo, los algoritmos genéticos, las redes neuronales, la programación modular, la herencia y la planificación y desarrollo de proyectos. Asimismo, su realización permite a diferentes desarrolladores de diversos ámbitos disponer de una herramienta que les permita definir y evolucionar sus redes neuronales de una manera fácil, intuitiva y personalizada, al mismo tiempo que les permite añadir cualquier código que vean pertinente de una forma accesible y sencilla.

Bibliografía

[1] ANGELINE, P. J., SAUNDERS, G. M., AND POLLACK, J. B. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks 5*, 1 (1994), 54–65.

[2] ASLANI, S., DAYAN, M., STORELLI, L., FILIPPI, M., MURINO, V., ROCCA, M. A., AND SONA, D. Multi-branch convolutional neural network for multiple sclerosis lesion segmentation. https://www.groundai.com/project/multi-branch-convolutional-neural-network-for-multiple-sclerosis-lesion-segmentation/1, 2018.

[3] ASSUNÇÃO, F., LOURENÇO, N., MACHADO, P., AND RIBEIRO, B. DENSER: Deep evolutionary network structured representation. *CoRR abs/1801.01563* (2018).

[4] BAKER, B., GUPTA, O., RASKAR, R., AND NAIK, N. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823* (2017).

[5] BENDER, G., KINDERMANS, P.-J., ZOPH, B., VASUDEVAN, V., AND LE, Q. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning* (2018), pp. 550–559.

[6] BERGSTRA, J., YAMINS, D., AND COX, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning* (2013), pp. 115–123.

[7] BOADA, M. J. L., BOADA, B. L., AND LÓPEZ, V. D. Algoritmo de aprendizaje por refuerzo continuo para el control de un sistema de suspensión semi-activa. *Revista Iberoamericana de Ingeniería Mecánica 9*, 2 (2005), 77.

[8] BOZDOGAN, H., AKBILGIC, O., AND BALABAN, M. E. A novel hybrid rbf neural networks model as a forecaster. Artículo: https://doi.org/10.1007/s11222-013-9375-7 | Base de datos: https://archive.ics.uci.edu/ml/datasets/ISTANBUL+STOCK+EXCHANGE, 2014.

[9] BROCK, A., LIM, T., RITCHIE, J. M., AND WESTON, N. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344* (2017).

[10] BROCK, A., LIM, T., RITCHIE, J. M., AND WESTON, N. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344* (2017).

[11] CAI, H., CHEN, T., ZHANG, W., YU, Y., AND WANG, J. Efficient architecture search by network transformation. *arXiv preprint arXiv:1707.04873* (2017).

[12] CAI, H., YANG, J., ZHANG, W., HAN, S., AND YU, Y. Path-level network transformation for efficient architecture search. *arXiv preprint arXiv:1806.02639* (2018).

[13] CAI, H., ZHU, L., AND HAN, S. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[14] CORTEZ, P., AND MORAIS, A. A data mining approach to predict forest firesusing meteorological data. Artículo: http://www3.dsi.uminho.pt/pcortez/fires.pdf | Base de datos: https://archive.ics.uci.edu/ml/datasets/Forest+Fires, 2007.

[15] DE LIMA, R. H. R., POZO, A., AND SANTANA, R. Automatic design of convolutional neural networks using grammatical evolution. In *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)* (2019), IEEE, pp. 329–334.

[16] DEAP. Deap documentation. https://deap.readthedocs.io/en/master/, 2020.

[17] DENNYBRITZ. Reinforcement-learning. https://github.com/dennybritz/reinforcement-learning, 2016.

[18] DOMHAN, T., SPRINGENBERG, J. T., AND HUTTER, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence* (2015).

[19] ELSKEN, T., METZEN, J.-H., AND HUTTER, F. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528* (2017).

[20] ELSKEN, T., METZEN, J. H., AND HUTTER, F. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081* (2018).

[21] FELGAER, P. Optimización de redes bayesianas basado en técnicas de aprendizaje por inducción. *Reportes Técnicos en Ingeniería del Software 6*, 2 (2004), 64–69.

[22] GARCIARENA, U. Evoflow. https://github.com/unaigarciarena/EvoFlow, 2019.

[23] GOLDBERG, D., AND HOLLAND, J. Genetic algorithms and machine learning. machine learning.

[24] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[25] GUO, Z., ZHANG, X., MU, HAOYUAN, HENG, WEN, LIU, ZECHUN, WE, YICHEN, SUN, AND JIAN. Single-path-one-shot-nas. https://github.com/ShunLu91/Single-Path-One-Shot-NAS, 2019.

[26] HARVITRONIX. Evolve a neural network with a genetic algorithm. https://github.com/harvitronix/neural-network-genetic-algorithm, 2017.

[27] ILTER, N., GUVENIR, H. A., AND DEMIRÖZ, G. Learning differential diagnosis oferythemato-squamous diseases using voting featureintervals. Artículo: http://www.cs.bilkent.edu.tr/~guvenir/publications/AIM_13_3_147.pdf | Base de datos: https://archive.ics.uci.edu/ml/datasets/Forest+type+mapping, 2011.

[28] JIN, H., SONG, Q., AND HU, X. Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282 9* (2018).

[29] JOHNSON, B., TATEISHI, R., AND XIE, Z. Using geographically weighted variables for image classification. *Remote sensing letters 3*, 6 (2012), 491–499.

[30] KERAS. Keras documentation. https://keras.io/, 2019.

[31] KLEIN, A., FALKNER, S., SPRINGENBERG, J. T., AND HUTTER, F. Learning curve prediction with bayesian neural networks.

[32] KRIZHEVSKY, A., NAIR, V., AND HINTON, G. The cifar-10 dataset, 2009.

[33] LAB, D. Autokeras. https://autokeras.com/, 2019.

[34] LE, Q. V. A tutorial on deep learning part 1: Nonlinear classifiers and the backpropagation algorithm. *Google Brain, Google Inc.* (2015), 1–13.

[35] LECUN, Y., AND CORTES, C. MNIST handwritten digit database.

[36] LIMA, R. H., AND POZO, A. T. Evolving convolutional neural networks through grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2019), pp. 179–180.

[37] LINCOFF, G. H., AND KNOPF, A. A. Mushroom records are drawn from the audubon society field guide to north american mushrooms. Base de datos: http://www.randalolson.com/data/benchmarks/mushroom.csv.gz, 1981.

[38] LIU, H., SIMONYAN, K., VINYALS, O., FERNANDO, C., AND KAVUKCUOGLU, K. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).

[39] LIU, H., SIMONYAN, K., AND YANG, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[40] LOSHCHILOV, I., AND HUTTER, F. CMA-ES for hyperparameter optimization of deep neural networks. *CoRR abs/1604.07269* (2016).

[41] MATEOS, A. Algoritmos evolutivos y algoritmos genéticos. *Inteligencia en Redes de Comunicaciones, Ingeniería de Telecomunicación. Universidad Carlos III de Madrid* (2004).

[42] MENDOZA, H., KLEIN, A., FEURER, M., SPRINGENBERG, J. T., AND HUTTER, F. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning* (2016), pp. 58–65.

[43] MIIKKULAINEN, R., LIANG, J., MEYERSON, E., RAWAL, A., FINK, D., FRANCON, O., RAJU, B., SHAHRZAD, H., NAVRUZYAN, A., DUFFY, N., ET AL. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2019, pp. 293–312.

[44] MILLER, G. F., TODD, P. M., AND HEGDE, S. U. Designing neural networks using genetic algorithms. 379–384.

[45] NARANJO, E. Índices bursátiles: Definición y principales referentes internacionales. https://www.21tradingcoach.com/es/formaci%C3%B3n-gratuita/introducci%C3%B3n-a-los-mercados-financieros/111-%C3%ADndices-burs%C3%A1tiles-definici%C3%B3n-y-principales-referentes-internacionaless.

[46] PHAM, H., GUAN, M. Y., ZOPH, B., LE, Q. V., AND DEAN, J. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268* (2018).

[47] PYTORCH. Pytorch documentation. https://pytorch.org/docs, 2019.

[48] PYTORCH. torchvision. https://pytorch.org/docs/stable/torchvision/, 2019.

[49] RAWAL, A., AND MIIKKULAINEN, R. From nodes to networks: Evolving recurrent neural networks. *arXiv preprint arXiv:1803.04439* (2018).

[50] REAL, E., AGGARWAL, A., HUANG, Y., AND LE, Q. V. Aging evolution for image classifier architecture search.

[51] REAL, E., MOORE, S., SELLE, A., SAXENA, S., SUEMATSU, Y. L., TAN, J., LE, Q., AND KURAKIN, A. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041* (2017).

[52] ROHRER, B. How do convolutional neural networks work? https://e2eml.school/how_convolutional_neural_networks_work.html, 2016.

[53] RÍOS, N. R. T., LITARDO, F. E. T., AND SALTOS, J. W. S. Optimización de redes bayesianas basado en técnicas de aprendizaje por inducción. Revista Publicando.

[54] SAXENA, S., AND VERBEEK, J. Convolutional neural fabrics. 4053–4061.

[55] SINGARIMBUN, R. N., NABABAN, E. B., AND SITOMPUL, O. S. Adaptive moment estimation to minimize square error in backpropagation algorithm. In *2019 International Conference of Computer Science and Information Technology (ICoSNIKOM)* (2019), IEEE, pp. 1–7.

[56] STANLEY, K. O., D'AMBROSIO, D. B., AND GAUCI, J. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life 15*, 2 (2009), 185–212.

[57] STANLEY, K. O., AND MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. *Evolutionary computation 10*, 2 (2002), 99–127.

[58] SUGANUMA, M., SHIRAKAWA, S., AND NAGAO, T. A genetic programming approach to designing convolutional neural network architectures. 497–504.

[59] SUGANUMA, M., SHIRAKAWA, S., AND NAGAO, T. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference* (New York, NY, USA, 2017), GECCO '17, ACM, pp. 497–504.

[60] SWERSKY, K., SNOEK, J., AND ADAMS, R. P. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896* (2014).

[61] TENSORFLOW. Tensorflow documentation. https://www.tensorflow.org/, 2019.

[62] THEANO. Theano documentation. http://deeplearning.net/software/theano/, 2017.

[63] THOMAS ELSKEN, JAN HENDRIK METZEN, F. H. Neural architecture search: A survey. *Journal of Machine Learning Research 20 (2019) 1-21* (2019), 1–13.

[64] UPV/EHU. Algoritmos genéticos. *Universidad Pública del Pais Vasco (UPV/EHU)*, 1–29.

[65] VIEIRA, S., PINAYA, W. H., AND MECHELLI, A. Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications, 2017.

[66] VITO, S. D., MASSERA, E., PIGA, M., MARTINOTTO, L., AND FRANCIA, G. D. On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. Artículo: https://www.sciencedirect.com/science/article/abs/pii/S0925400507007691?via%3Dihub | Base de datos: https://archive.ics.uci.edu/ml/datasets/Air+Quality, 2008.

[67] WEI, T., WANG, C., RUI, Y., AND CHEN, C. W. Network morphism. 564–572.

[68] WHITE, C., NEISWANGER, W., AND SAVANI, Y. Bananas: Bayesian optimization with neural architectures for neural architecture search. https://github.com/naszilla/bananas, 2019.

[69] XIAO, H., RASUL, K., AND VOLLGRAF, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[70] XIE, L., AND YUILLE, A. Genetic cnn. 1379–1388.

[71] XIE, S., ZHENG, H., LIU, C., AND LIN, L. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926* (2018).

[72] ZOPH, B., AND LE, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[73] ZOPH, B., VASUDEVAN, V., SHLENS, J., AND LE, Q. V. Learning transferable architectures for scalable image recognition. 8697–8710.