

UAX

TRABAJO FIN DE MÁSTER

**OMPE
23-24**

UNIVERSIDAD ALFONSO X EL SABIO

ESCUELA TÉCNICA SUPERIOR

MUIA Máster Universitario en Inteligencia Artificial



**UNIVERSIDAD
ALFONSO X EL SABIO**

TRABAJO DE FIN DE MÁSTER

Redes Neuronales Automodelables

**Carlos Bilbao Lara
Alberto Partida Rodríguez**

Julio, 2024



RESUMEN

A lo largo de este proyecto se expone qué es y cómo funciona un algoritmo evolutivo y las redes neuronales artificiales convolucionales. Además, se explica cómo se pueden unir ambas tecnologías y se ha implementado un algoritmo evolutivo que permite encontrar la red neuronal convolucional capaz de adaptarse a un dataset introducido por el usuario. Para ello se ha diseñado un mapa de características que representan una arquitectura de una red convolucional, y una función que es capaz de convertir esas características en un modelo que se pueda entrenar y usar. Con el fin de facilitar el uso de este tipo de tecnologías, se introduce una interfaz gráfica sencilla pero profesional que permite al usuario interactuar con el algoritmo. Estas comunicaciones se hacen mediante REST, el servidor está realizado en Python, con Flask, y el cliente en ReactJS. Además, se ha utilizado los cuadernos de Python para desarrollar y explicar cómo se ha implementado el algoritmo. Con el fin de llevar un control de versiones se ha utilizado la herramienta Git, junto a GitHub donde se encuentra el código del proyecto (disponible en el Anexo I). Tras ajustar el algoritmo, se han realizado unas pruebas utilizando el dataset CIFAR-10 debido a su simplicidad, pero al mismo tiempo siendo lo suficientemente complejo como para comprobar si el algoritmo funciona como se espera. Estas pruebas consisten en ejecutar el algoritmo 10 veces y medir el tiempo que tarda en ejecutarse el modelo, guardando los resultados de los mejores individuos de cada generación. Después de recopilar los datos, se realiza un análisis de estos en los que se expone que a pesar de que los resultados satisfactorios en términos de precisión, se ha observado que el algoritmo es lento. Para mejorar esta situación, se exponen una serie de soluciones, como usar un modelo distribuido que permita entrenar los individuos en paralelo.

Palabras clave

AutoML, convolucional, genético, evolutivo, red



ABSTRACT

This project explains what an evolutionary algorithm and convolutional artificial neural networks are and how they work. In addition, it is explained how both technologies can be united, this project also implements an evolutionary algorithm has been implemented to find the convolutional neural network capable of adapting to a dataset introduced by the user. For this purpose, a map of features that represent a convolutional network architecture has been designed, and a function that is able to convert those features into a model that can be trained and used. To facilitate the use of this type of technology, a simple but professional graphical interface is introduced that allows the user to interact with the algorithm. These communications are done through REST, the server is made in Python, with Flask, and the client in ReactJS. In addition, Python notebooks have been used to develop and explain how the algorithm has been implemented. In order to keep a version control, the Git tool has been used, together with GitHub where the project code is located (available in Annex I). After adjusting the algorithm, some tests have been performed using the CIFAR-10 dataset due to its simplicity, but at the same time being complex enough to check if the algorithm works as expected. These tests consist of running the algorithm 10 times and measuring the time it takes to run the model, saving the results of the best individuals of each generation. After collecting the data, an analysis of the data is carried out in which it is stated that despite the satisfactory results in terms of accuracy, it has been observed that the algorithm is slow. To improve this situation, a series of solutions are presented, such as using a distributed model that allows training individuals concurrently.

Keywords

AutoML, convolutional, genetic, evolution, network



ÍNDICE

Capítulo 1: Introducción al Trabajo Fin de Máster	6
1.1 Justificación del proyecto realizado.....	6
1.2 Presentación del proyecto y sus contenidos.....	7
1.3 Ejemplo Práctico de Uso	9
Capítulo 2: Objetivos del TFM.....	15
2.1 Objetivo general.....	15
2.2 Objetivos específicos.....	15
Capítulo 3: Marco Teórico.....	16
3.1 Algoritmo Genéticos, ¿qué son?	16
3.1.1 Algoritmo Genéticos: Teoría de la genética.	17
3.1.2 Algoritmo Genéticos: Ejemplos y aplicaciones.	20
3.1.3 Algoritmo Genéticos: Selección.	21
3.1.4 Algoritmo Genéticos: Cruce.....	22
3.1.5 Algoritmo Genéticos: Mutación.....	23
3.2 Redes Neuronales: Historia y bases	24
3.2.1 Redes Neuronales: CNNs	26
3.2.2 Redes Neuronales: Funciones de Coste	28
3.2.3 Redes Neuronales: Funciones de Activación	30
3.2.4 Redes Neuronales: Pooling	31
3.2.5 Redes Neuronales: Padding.....	33
3.2.6 CNNs: casos de usos.....	33
3.2.7 CNNs: Arquitecturas más conocidas	34
3.3 Redes Auto modelables: State of the art	36
3.3.1 Algoritmos Genéticos y Redes Neuronales	39
3.3.2 Ejemplos de Redes Auto modelables: AutoKeras	40
3.3.3 Ejemplos de Redes Auto modelables: Neural Network Intelligence.....	41
Capítulo 4: Marco Metodológico.....	42
4.1 Población y muestra	44
4.2 Objetivos de investigación	45
4.3 Variables intervinientes e instrumentos de evaluación	45
4.4 Diseño experimental.....	46
4.5 Modelo de análisis de datos	46



Capítulo 5: Presentación de análisis, resultados y discusión	49
5.1 Análisis descriptivo de la muestra	49
5.2 Resultados obtenidos.....	49
5.3 Discusión y análisis de los resultados	51
Capítulo 6: Conclusiones	54
6.1 Limitaciones.....	56
6.2 Prospectiva.....	¡Error! Marcador no definido.
6.3 Posibles mejoras/continuaciones de este proyecto..	¡Error! Marcador no definido.
6.4 Consideraciones finales.....	57
Bibliografía.....	59
1. ANEXO I	65



Capítulo 1: Introducción al Trabajo Fin de Máster

A lo largo de este proyecto se tratará y analizará la viabilidad, así como la utilidad de las Redes Neuronales Auto modelables, en específico, para las redes neuronales convolucionales, con el objetivo de realizar el análisis y clasificación de imágenes. Aunque si bien está centrado en este tipo de redes, puede ser extrapolado y fácilmente adaptado a cualquier otro tipo de redes.

1.1 Justificación del proyecto realizado

Este proyecto surgió como idea de evitar tener que hacer la tediosa fase de búsqueda de hiperparámetros y de la mejor arquitectura para cada dataset y el problema que al que un experto en estas tecnologías tiene que enfrentarse cada día. Por ello se buscó la manera de intentar automatizar estas tareas. A partir de aquí surgen muchas posibilidades para enfrentarse a esta automatización, como, por ejemplo, tener una pila de arquitecturas predefinidas, ejecutarlas y escoger la que mejor resultado dé, hacer búsqueda de hiperparámetros con metodologías como Grid Search y con ellos se ejecuta otra pila de arquitecturas predefinidas de nuevo, o por ejemplo; arquitecturas que se van adaptando y modificando al dataset buscando la mejor arquitectura posible usando algoritmos genéticos. Este proyecto se centra en esta última solución.

Es realmente interesante pensar en obtener un algoritmo lo más pulido posible de forma automatizada, ya que ayudará a la comunidad a poder buscar el mejor resultado posible en los problemas a los que se enfrentarán, aunque estas ejecuciones llevarán un tiempo, en función del hardware utilizado, ya que para un mismo dataset se entrenarán y evaluarán muchísimas arquitecturas.



Por esta razón, es fundamental permitir al usuario ajustar tanto la profundidad del algoritmo como la cantidad de descendencia generada. Esto le permite controlar el tiempo de espera para obtener resultados. En datasets más simples, es posible encontrar una arquitectura óptima o suficientemente buena en pocas iteraciones. Sin embargo, en datasets más complejos, puede ser necesario incrementar el número de iteraciones del algoritmo para alcanzar un resultado que sea lo suficientemente óptimo para el usuario.

También, se permite al usuario seleccionar el número de epochs (número de veces que se entrenará y evaluará el algoritmo) ya que pasa exactamente lo mismo que con el número de descendencia, para datasets simples, puede necesitar solo unas pocas epochs para converger en una alta precisión en la fase de evaluación, sin embargo; para datasets más complejos hará falta una cantidad más elevada de epochs.

Hoy en día ya existen ciertos modelos que buscan automatizar este proceso [1] como AutoKeras, este algoritmo buscará en su base arquitecturas predefinidas, bajo la API de AutoKeras una serie de arquitecturas que puedan ser la que mejor se adapte a priori al dataset. Tras ello, escogerá la mejor y la mutará para crear la siguiente configuración a evaluar. El diseño del algoritmo se basa en Hill-Climbing. En el capítulo 3 se explicará más en detalle este y otros algoritmos similares de automatización de la búsqueda de arquitecturas explicando y desarrollando el marco teórico del proyecto.

1.2 Presentación del proyecto y sus contenidos

Actualmente uno de los pasos más tediosos y largos en el desarrollo de un modelo de redes neuronales es generar unos hiperparámetros y una arquitectura que se adapten perfectamente al dataset, es decir; que permitan un entrenamiento relativamente rápido obteniendo una alta precisión en la fase de evaluación, sin sobre ajustarse; evitando que el modelo sea capaz de aprender de las imágenes de entrenamiento, pero no siendo capaz de generalizar ese conocimiento a imágenes no vistas anteriormente. Habitualmente este paso es relativamente lento ya que,



aunque se puede intuir, no se sabe exactamente como influirá al resultado del entrenamiento y testeo modificar una u otra capa, añadir más o menos neuronas, añadir o eliminar capas....

Por ello, este proyecto pretende ayudar en esta fase, automatizando todo este proceso de búsqueda, permitiendo realizar esa búsqueda manual de la mejor arquitectura e hiperparámetros a partir de algoritmos genéticos para la arquitectura.

Aunque si bien es cierto que se automatiza esta tarea y se va a reducir el tiempo necesario para encontrar la arquitectura ideal, el tiempo necesario para encontrar la arquitectura va a seguir siendo muy elevado debido a que entrenar y evaluar múltiples arquitecturas es lento (siempre dependiendo que hardware y que dataset se esté utilizando), por ello se expondrán algunas soluciones o mejoras a este proyecto, incluyendo una solución a este problema. Con el fin de entender, explicar y demostrar a solución planteada en este apartado se ha organizado el proyecto de la siguiente manera:

1. **Capítulo 2: Objetivos del TFM**, en este apartado se desarrollará los objetivos del TFM, incluyendo los objetivos generales como el desarrollo de una red auto modelable y específicos como el desarrollo de una aplicación web que dé al usuario la mejor arquitectura, y le permita realizar algunas configuraciones de la red, como la cantidad de descendientes con los que desea probar.
2. **Capítulo 3: Marco Teórico**, en este apartado se desarrollará qué son las redes neuronales, las redes neuronales convolucionales, los algoritmos genéticos, así como su funcionamiento y uso. También se desarrollará qué son las redes auto modelables y el estado del arte de esta tecnología.
3. **Capítulo 4: Marco Metodológico**, en este apartado se incluye las tecnologías utilizadas, la metodología de trabajo, los datasets utilizados para la observación del funcionamiento de la red auto



modelable, igualmente se describe el hardware, las herramientas utilizadas para desarrollar el proyecto y el modelo de análisis de datos para analizar los resultados obtenidos.

4. **Capítulo 5: Presentación de análisis, resultados y discusión**, en este apartado se presentarán los resultados obtenidos del modelo, así como una discusión acerca de estos y posibles mejoras del modelo. También se abordará el tema de los posibles usos del proyecto y adaptaciones para nuevos proyectos.
5. **Capítulo 6: Conclusiones**, finalmente se concluye con las limitaciones del proyecto, errores cometidos, posible prospectiva, es decir; como continuar con el proyecto y unas consideraciones finales incluyendo las reflexiones finales y personales acerca del mismo, autoevaluación y agradecimientos.
6. **Bibliografía**, contiene los enlaces y referencias usadas a lo largo de esta memoria con los diferentes, libros, artículos y páginas utilizadas y leídas para comprender el contenido y desarrollar el proyecto correctamente.

1.3 Ejemplo Práctico de Uso

Este proyecto está destinado a aquellas personas que desean crear una red neuronal convolucional con el fin de obtener la mejor puntuación posible en la evaluación de su Red Neuronal. Sin embargo, aquí se pueden dar dos situaciones, que el usuario no tenga amplios conocimientos en cómo crear, modificar y ajustar arquitecturas de redes neuronales convolucionales; o que no quiera o tenga mucho tiempo para probar manualmente ajustes y estar monitorizando la evolución de entrenamiento y evaluación de la red neuronal que ha construido. Por tanto, aquí el usuario se plantea, ¿no existirá una forma de obtener la mejor arquitectura posible fácilmente y sin tener que intervenir constantemente en su construcción?



En este punto es donde entra en juego este proyecto, para crear un ejemplo, supondremos que el usuario quiere crear una red neuronal convolucional que detecte qué hay en la imagen. Para ello, busca un dataset como CIFAR10 y se dispone a encontrar la mejor arquitectura. Sin embargo, no dispone de los conocimientos o del tiempo para estar monitorizando y ajustando la arquitectura. Por ello, decide usar este proyecto. Simplemente indicando un usuario, un email, cuantos epochs para entrenar las arquitecturas y la descendencia, el algoritmo generado en este proyecto se encargará de encontrar la mejor arquitectura posible y entregarle un resumen de esta para que pueda replicarla y usarla. Además, el usuario simplemente debe incluir el dataset con las imágenes, ya sea que estén dividido en train and test o simplemente en las carpetas con cada clase, es decir; la estructura de carpetas puede ser:

- path + cifar10\train\truck: siendo “path” la ubicación donde está guardado el proyecto, “cifar10” la carpeta que contiene el dataset, que en este caso contendrá en su interior las carpetas “train” y “test”. Cada una de estas 2 subcarpetas contendrá 1 carpeta por cada clase con las imágenes correspondientes a su nombre, en este caso, por ejemplo, contendrá las imágenes de camiones.
- path + cifar10\truck; similar al caso anterior, con la variación de que no tendrá la división en train y test, directamente la carpeta del dataset “cifar10” contiene las carpetas correspondientes a las clases con todas las imágenes de la misma.

Para ambos casos la aplicación funcionará correctamente.



AutoModelizer

Introduce a dataset to discover its best architecture

Search the best architecture

Introduce your data

Name
carlos

Email
bilbao@gmail.com

Number of descendency
10

Number of epochs
20

Number of classes
10

☒ Is it splitted into train and test?

Seleccionar archivo

Work your magic →

Check the results, please notice it could take several hours

Name
carlos

Email
bilbao@gmail.com

check results →

Learning Rate: 0.00001

Number of Convolutional Layers: 2

Kernel Sizes: 1, 5

Filters: 32, 128

Fully connected Layers: 0

Dropout: 0

Accuracy: 0.5637

Figura 1.3.1 Captura de pantalla de la aplicación web donde el usuario introduce la información deseada

¿Descendencia? Sí, con el objetivo de encontrar la mejor arquitectura posible, se utiliza un algoritmo genético cuyas bases teóricas se detallarán en el Capítulo 3. Un algoritmo genético en software trata de imitar la teoría de la evolución desarrollada por Darwin en 1859[3]. La teoría de la evolución describe un mecanismo natural para la evolución de las especies a lo largo de la historia, la selección natural. (Las bases de esta teoría que serán explicadas más en detalle en el capítulo 3) Las principales fases son la herencia, la variación genética y la selección natural.

Teniendo esto en cuenta y que el usuario ha introducido 10 generaciones y 20 epochs, el proceso que realizará el algoritmo será el siguiente:



1. Genera un dataset de entrenamiento y otro de testeo con el dataset introducido por el usuario.
2. Genera una población inicial de 10 arquitecturas.
3. Entrena 20 epochs (número que ha sido introducido por el usuario) cada arquitectura y las evalúa.
4. Las 4 arquitecturas más fuertes (con mejor puntuación en evaluación) se “reproducen” generando 4 nuevos individuos cuya genética (cantidad de capas, numero de filtros...) se basa en la de sus antecesores.
5. Se introduce aleatoriamente cambios en su genética (mutación genética).
6. Se crea una nueva población reemplazando las 4 peores arquitecturas por las nuevas que acaban de ser creadas y mantenemos las demás (selección natural).
7. Se repite desde los pasos del 3 al 6 otras 9 veces más para obtener un total de 10 generaciones (cantidad indicada anteriormente por el usuario).
8. Se entrega al usuario la arquitectura final con mejor puntuación de todas las generaciones.

Para poder hacer todo esto, se manifiesta las características de una red neuronal convolucional como “genética” es decir; un gen indica la cantidad de capas convolucionales, otro la cantidad de poolings, otro el número que se usará como tamaño del kernel, otro para las capas dropout.... El conjunto de estos genes conforma una arquitectura completa. Con el fin de encontrar la mejor arquitectura, toda la población se entrena con la misma cantidad de epochs y se evalúa, posteriormente las mejores genéticas se cruzarán,



generando un nuevo individuo con la genética de sus antecesores y se le realizara alguna mutación.

Tras alcanzar el máximo de descendencia indicado por el usuario, se le entregará la arquitectura con el mejor resultado obtenido durante el algoritmo genético. Para ello, usa el botón de “Check Results”, y el servidor buscará si la respuesta ya está disponible o si todavía está procesando la información.

Figura 1.3.2 Captura de pantalla que muestra la notificación cuando aún se está procesando la información



Figura 1.3.2 Captura de pantalla que muestra la arquitectura final óptima que ha encontrado el algoritmo (con el fin de acelerar el proceso, se ha usado 2 generaciones y 5 epochs para obtener este resultado)



Capítulo 2: Objetivos del TFM

A lo largo de este capítulo se describen los objetivos, tanto generales como específicos propuestos para completar este proyecto.

2.1 Objetivo general

- El objetivo general del proyecto es crear una aplicación web que permita a cualquier usuario obtener a través del backend la mejor arquitectura encontrada para el dataset del usuario.

2.2 Objetivos específicos

1. Investigar otras posibles soluciones en el mercado.
2. Investigar y definir que son los algoritmos evolutivos y redes neuronales convolucionales.
3. Desarrollar una arquitectura de redes neuronales, que permita generar un modelo en base a un vector.
4. Desarrollar un algoritmo evolutivo que permita ir cruzando y mutando las diferentes arquitecturas.
5. Desarrollar una interfaz en REACT.
6. Desarrollar una arquitectura REST-API para combinar el backend con el frontend.



Capítulo 3: Marco Teórico

A lo largo de este capítulo se explicará la base teórica de este proyecto los algoritmos genéticos, las redes neuronales, y explicando en detalle las CNNs, y por último, una explicación de cómo funcionan las aplicaciones basadas en microservicios como la que se ha desarrollado en el proyecto. Además, un último apartado que desarrolla el estado del arte de estas redes auto modificables.

3.1 Algoritmo Genéticos, ¿qué son?

Este tipo de algoritmos buscan recrear la evolución de las especies dentro de un algoritmo de software, simulando que las “especies” en este caso modelos de redes neuronales, tienen ciertos atributos o genes como la cantidad de capas, el tamaño de estas, el tipo de las capas...[2]. Darwin describió un mecanismo natural para la evolución de las especies a lo largo de la historia conocido como selección natural. Aunque comprendió que los seres vivos tienen ciertas variaciones en sus rasgos con respecto a sus antecesores, no fue hasta Gregor Mendel que desarrolló la teoría de la genética [3] a finales del siglo XIX, publicando sus descubrimientos en 1864, las leyes de la herencia genética publicadas en 1866 y finalmente en 1874 su última obra, la “Teoría mecánico-fisiológico de la evolución orgánica”, mismo año en el que falleció. Sin embargo, no fue hasta el siglo XX que finalmente los biólogos comprendieron que esta teoría explicaba correctamente aquellos conceptos de la teoría de la evolución que Darwin había dejado sin resolver: cómo se produce la herencia, por qué existen variaciones entre individuos en una misma población y la selección natural.

Los algoritmos genéticos, aplicados al contexto software, basan su premisa en que las características de un individuo están codificadas en su genética, por ejemplo, un guisante es verde, porque hay un gen que dice que expresa que es verde, una persona tiene el pelo rubio y no castaño, porque tiene un gen que expresa este rasgo. Integrando tanto la teoría de la evolución de



Darwin con la teoría genética de Mendel, se identifica 3 fases clave en los algoritmos genéticos:

- **Herencia:** los descendientes comparten la mayoría de las características genéticas de sus progenitores tal y cómo descubrió Mendel, aunque pueden existir ciertas mutaciones
- **Variación:** Darwin observó que los individuos dentro de una población varían en algunas características entre ellos. Tras los descubrimientos de Mendel, pasó a conocerse como Mutación Genética, es decir; aquellas diferencias que Darwin observó en los individuos de la misma población son la consecuencia de ciertas variaciones de los genes que ha obtenido de sus antecesores.
- **Selección Natural:** los individuos con las características más ventajosas para su supervivencia son los que acaban predominando mientras que otros acaban desapareciendo. La comprensión moderna de la genética ha permitido explicar cómo estas variaciones genéticas favorecen a ciertos individuos sobre los demás.

3.1.1 Algoritmo Genéticos: Teoría de la genética.

[4] Gregor Mendel, a través de experimentar y cruzar seres vivos, entre ellos su conocido experimento con guisantes, estableció los fundamentos de la herencia genética, conocidos hoy en día como las leyes de Mendel. Para poder entender sus leyes, se explica a continuación los experimentos que realizó Mendel para desarrollar la teoría de la genética.

En los experimentos de su huerto con los guisantes, descubrió que, si juntaba 2 guisantes de la misma raza “pura”, su descendencia sería idéntica, es decir; si cruzaba 2 guisantes verdes, su descendencia sería verde, y si juntaba 2 amarillos su descendencia sería amarilla.

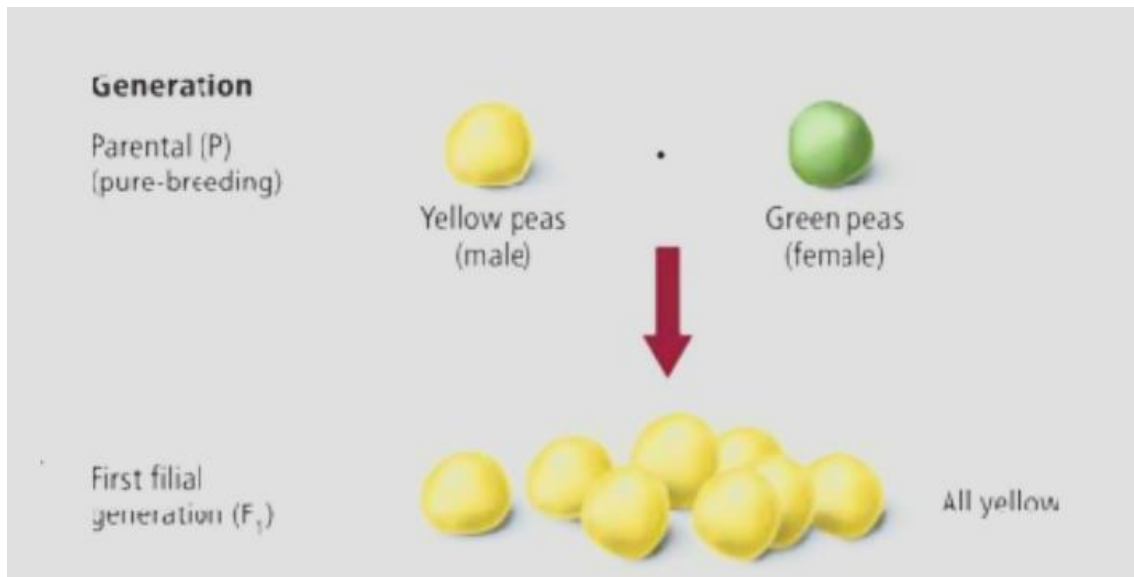


Figura 3.1.1.1 [4] Representación del cruce de 2 razas puras distintas, gráfica obtenida de "Miko, I. (2008). Gregor Mendel and the principles of inheritance. Nature Education" [4]

Al cruzar 2 guisantes puros, de diferentes razas, por ejemplo, uno amarillo y uno verde, el resultado al contrario de lo que se pudiera imaginar fue que todos eran amarillos.



Figura 3.1.1.2 Representación del cruce de la descendencia de 2 razas puras distintas gráficas obtenidas de “Miko, I. (2008). Gregor Mendel and the principles of inheritance. Nature Education” [4]

Sin embargo, si cruzamos esta segunda generación de guisantes que han salido todos amarillos, aparecerán algunos verdes en su descendencia, aproximadamente con una proporción de 3 a 1.

El hecho de que los guisantes sean todos amarillos en su primera generación se debe a que esta característica es dominante, es decir; tiende a aparecer más frecuentemente en sus descendencias. Por el contrario, el color verde es recesivo. Con todo ello, Mendel separó los fenotipos, que son las características observables, como el color del guisante, de los genotipos, que es el conjunto de genes en el ADN de un organismo. Por ejemplo, para un mismo gen P, existe también valor contrario p, si los dos antecesores tienen el mismo tipo como genotipo P, su descendencia tendrá el gen P y mostrará el fenotipo correspondiente. Si los antecesores son P y p, el fenotipo resultante será el dominante de los 2.




Genotype		
PP (homozygous)	Pp (heterozygous)	pp (homozygous)
Phenotype		
 Purple	 Purple	 White



Figura 3.1.1.3 [4] Representación del cruce de la descendencia de 2 razas puras distintas, gráfica obtenida de “Miko, I. (2008). Gregor Mendel and the principles of inheritance. Nature Education” [4]

Con todo ello, Mendel creó las leyes de la genética que son:

- **Ley de la uniformidad:** al cruzar una raza pura de una especie, con otra raza pura, la descendencia de la primera generación será física y genotípicamente iguales entre sí.
- **Ley de la segregación:** cada individuo posee dos alelos para cada gen, que se separan durante la formación del individuo, ya que cada gameto solo puede llevar 1 de los dos. Estas características se “escogen” aleatoriamente, y el fenotipo es el resultado de combinar estas características.
- **Ley de transmisión independiente:** la segregación de los rasgos hereditarios se da de forma independiente de unos individuos a otros, por lo que la genética de uno no afectará al patrón de herencia del otro

3.1.2 Algoritmo Genéticos: Ejemplos y aplicaciones.

Los algoritmos genéticos se utilizan en el software para buscar soluciones óptimas a problemas complejos mediante la simulación de procesos evolutivos. [5] Entre sus principales usos se encuentra:

- Optimización de redes y sistemas: mejorando el diseño y la operación de redes complejas como redes de telecomunicación y transporte
- Finanzas: para selección de carteras y estrategias de trading



- Gestión automatizada de equipamiento industrial
- Inteligencia Artificial: encontrando soluciones en el aprendizaje automático, sobre todo se aplica en optimización de juegos y planificación de tareas
- Ingeniería y diseño: optimizando procesos industriales como en aerodinámica y configuración de sistemas
- Medicina: ayudando en el diseño de nuevos fármacos, o en optimizar modelos de propagación de enfermedades y estrategias de salud pública.

3.1.3 Algoritmo Genéticos: Selección.

Escoger el tamaño de la población inicial puede ser determinante para que la ejecución del algoritmo sea óptima. Por ejemplo; si la población inicial no recoge suficientes diferencias genéticas, podrías no explorar todas las posibilidades, sin embargo; si es demasiado grande, el algoritmo se volverá demasiado lento. Del mismo modo, una población más grande puede ayudar a encontrar una solución óptima, pero si se utiliza una población más pequeña se encontraría una solución subóptima pero no necesaria [3].

Tras determinar la puntuación de cada individuo se procede a seleccionar aquellos individuos que se van a reproducir para generar una descendencia [7]. Hay una gran variedad de tipos de selección, pero entre las más destacadas encontramos:

- **Selección proporcional:** los individuos se seleccionan según su puntuación, a mayor puntuación, mayor probabilidad de ser elegido. Sin embargo, hay que tener cuidado con esto ya que puede dar una convergencia prematura



- **Selección por torneo:** se escoge un numero pequeño de individuos al azar y se escoge al mejor del grupo. Es simple y eficaz por lo que ha sido escogido para la implementación práctica de este proyecto.
- **Selección por rango:** los individuos se clasifican por su puntuación y según el rango en el que estén se les asigna una probabilidad de ser escogido.
- **Selección elitista:** se asegura de escoger a los mejores individuos de cada generación y que se conserven para la siguiente.
- **Selección estocástica:** similar a la selección proporcional, pero reduce la probabilidad de que sean escogidos los mejores individuos múltiples veces. Garantiza de este modo una distribución más uniforme

3.1.4 Algoritmo Genéticos: Cruce.

Consiste en combinar la genética de 2 individuos seleccionados anteriormente seleccionados para generar un nuevo individuo. Del mismo modo que antes también existen diferentes tipos de cruce, el objetivo de este paso es encontrar la mejor descendencia posible. [3] Existen diferentes tipos de cruce como:

- **Cruce de 1 punto:** se trata de copiar exactamente los mismos genes de uno de los padres hasta un punto de la cadena y a partir de ahí copia los genes del otro.
- **Cruce de 2 puntos:** similar al anterior, pero añadiendo otro punto en la cadena donde vuelva a copiar los genes del primero de los padres.
- **Cruce uniforme:** cada gen se copia de uno de los 2 padres aleatoriamente. Este es el que se ha escogido en la implementación de este proyecto



3.1.5 Algoritmo Genéticos: Mutación.

Tras haber generado un nuevo individuo en la fase de cruce, falta mutarlo. La mutación se hace con el fin de encontrar nuevas posibilidades de combinaciones genéticas que pueda favorecer la “supervivencia” del individuo. Gracias a ello, se previene al algoritmo de encontrar un máximo local.

Por tanto, el proceso de mutación consiste en la introducción de modificaciones en la cadena genética del individuo de forma aleatoria, atendiendo a un umbral. Este umbral determinará cuanto y cuantas veces se producirán mutaciones en la descendencia. Escoger este umbral dependerá de que se desee más, si la exploración o la explotación, con el peligro de tardar mucho en converger, o quedarse estancado en un óptimo local respectivamente.



3.2 Redes Neuronales: Historia y bases

[8] Las redes neuronales artificiales son un subconjunto de machine learning y están en el núcleo de los algoritmos de Deep Learning. Su nombre se debe que la idea del funcionamiento de este software se basa en imitar cómo funcionan las redes neuronales biológicas que tienen los seres humanos en el cerebro.

Aunque el boom de esta tecnología se ha dado recientemente, sobre todo con el descubrimiento al público general de tecnologías como CHAT-GPT o Dall-E 3, el inicio de las redes neuronales se remonta a 1940, cuando Warren McCulloch y Walter Pitts introdujo el concepto de red neuronal. En 1943, propusieron un modelo matemático simple para la actividad neuronal. [9] En 1951, se desarrolló lo que se considera la primera red neuronal artificial de la historia, creada por Minsky y Dean Edmunds. Por esta época, en el año 1950 Alan Turing desarrolló el conocido Test de Turing, un concepto de juego de imitación donde un humano habla con una inteligencia artificial sin saberlo, si el humano no se da cuenta de que no está hablando con una persona de verdad, esta pasa el test.

[8] Sin embargo, no fue hasta 1957 que Frank Rosenblatt desarrolló el perceptrón simple, una simple neurona que imita las capacidades de percepción del ser humano y que es conocida ya que es la base de las siguientes investigaciones de redes neuronales. El perceptrón simple es una unidad binaria de procesamiento que recibe una serie de entradas y mediante unos pesos (la importancia relativa que existe entre las entradas y la salida) y un umbral de resultado a esas entradas, el perceptrón devuelve 1 o 0.

Siguiendo con la investigación y el desarrollo de Frank Rosenblatt, casi una década después se creó el perceptrón “multilayer”, una extensión del perceptrón simple que utilizaba múltiples neuronas formando capas en 1965, cuyos valores de los pesos de cada neurona se modificaban y actualizaban manualmente. Debido a que esto era un trabajo tedioso, la investigación sobre redes neuronales se enfocó en desarrollar un algoritmo



que pudiera actualizar los pesos directamente mientras la red se entrenara. Fue en 1986 cuando David Rumelhart, Geoffry Hinton y Ronald Williams desarrollaron el algoritmo de “Backpropagation”, fundamental y usado hoy en día, este algoritmo permite a la red calcular el error obtenido en la salida y propagar hacia las capas anteriores pequeños ajustes en los pesos con el fin de obtener mejores resultados en la siguiente iteración.

Poco después en el año 1989, inspirándose en la visión de los animales, desarrollaron las redes neuronales convoluciones, o en inglés, Convolutional Neuronal Networks (CNNs). La principal innovación de estas redes fue la introducción de capas convolucionales en las redes neuronales artificiales, dotándolas de “visión” y permitiendo entrenar a las redes para reconocimiento de objetos en imágenes entre otros usos. Por ejemplo, con estas redes se puede enseñar a un dron que se encuentre volando por una ciudad a esquivar los obstáculos de la calle, como edificios, coches, señales....

En 1997 se creó otra de las arquitecturas que sirve de base de las redes más utilizadas y conocidas hoy, las Long Short Term Memory (LSTM), conocidas por albergar en su arquitectura una pequeña memoria a corto plazo que se utiliza como entrada de las capas. Hoy en día son ampliamente utilizadas para tareas de NLP (Natural Language Processing), clasificación de texto, generación de texto, resúmenes..., así como para predicciones de series temporales.

Casi una década después, en 2006 se desarrollaron las Deep Belief Networks. Fueron desarrolladas por Geoffry Hinton, incluso hizo una conferencia de Google Tech Talk en 2007 explicando como funcionaban estas redes y llamándolo como “The Next Generation of Neural Networks”. Principalmente se utilizan para tareas de aprendizaje no supervisado y preentrenamiento de modelos de Deep Learning.

Por el año 2014, se presentaron las redes GAN (Generative Adversarial Networks) cuyo modelo consistía en entrenar 2 redes neuronales artificiales a la vez, uno haciendo de Generador que usando datos basura genera muestras, y Discriminador que recibe las muestras del generador y del



conjunto real, teniendo que ser capaz de distinguir entre las generadas y las reales.

Por último, como redes más conocidas hoy en día, en 2017 se presentaron las redes Transformers, en el artículo Attention is All you Need [10]. El modelo fue un hito en el mundo del NLP al demostrar un rendimiento extraordinario en una gran variedad de tareas, incluyendo traducción automática de textos, y generación de texto. Entre los modelos comerciales más conocidos y que se han ganado el afecto de los consumidores están BERT de Google o GPT de OpenAI lo que ha provocado un gran auge del sector, tanto por descubrimiento entre los consumidores como el aumento en la inversión en innovación en esta área lo que provocará un gran número de nuevas arquitecturas e incluso nuevas clases de redes neuronales en el futuro.

Dado que para este tipo de tecnologías era necesario el uso de ordenadores y servidores con una gran capacidad de procesamiento, según se podía ir aumentando la capacidad, más y mejores arquitecturas se iban descubriendo, así como la cantidad de problemas y situaciones que se pueden resolver utilizando redes neuronales. Es por ello por lo que, aunque la investigación sobre redes neuronales artificiales lleve 80 años, en el último cuarto se ha avanzado muchísimo más y se han realizado una gran cantidad de descubrimientos sobre las arquitecturas de las redes y sus posibles aplicaciones, cambiando el mundo tal y como se conocía.

3.2.1 Redes Neuronales: CNNs

Las Redes Neuronales Convolucionales, o CNNs (Convolutional Neuronal Networks) en inglés se desarrollaron sobre el año 1989, con la intención de dotar de una visión como la de los animales a los ordenadores. Para ello se desarrollaron varias capas ocultas especializadas en las tareas de tratamiento y obtención de información de imágenes, entre ellas las capas de convolución, sampling y pooling.



Las capas convoluciones permiten a la red neuronal detectar desde las líneas más simples en una imagen hasta las formas más complejas como rostros. Para ello se necesita una gran cantidad de imágenes para que la red aprenda las formas y características de la imagen para clasificarlas correctamente, lo que supone un desafío. Además, son redes computacionalmente costosas debido a que necesitas 1 neurona por cada característica de la imagen, es decir, suponiendo que estamos clasificando imágenes de 28x28 píxeles y a color, necesitaremos en la primera capa 2532 neuronas ($28 \times 28 \times 3 = 2532$) y esta imagen sería de bastante baja calidad por lo que costaría hasta para un humano reconocer lo que hay en ella. Sin embargo, si ese número ya parece alto, realicemos el cálculo con una imagen HD, 1920x1080 píxeles y a color; la primera capa de la red necesitará 6.220.800 neuronas, es por ello por lo que se suele realizar un tratamiento a las imágenes para reducir su dimensionalidad.

Además de ello, con el fin de obtener más datos, y a su vez, reducir el posible overfitting (es decir; que el modelo se ha entrenado tanto que no es capaz de generalizar correctamente, y funciona bien con los datos de entrenamiento pero no con el resto), el dataset se suele someter, no solo a esa reducción de calidad de imagen sino también a transformaciones como rotarla, cortarla, subir la exposición, o bajarla, voltearla vertical o/y horizontalmente incluso añadir ruido, eso sí tratando que estas transformaciones permitan a la imagen seguir siendo realista y que se puedan encontrar esas mismas imágenes transformadas en la vida real. A esta técnica se la conoce como “data augmentation”.

Cada valor de cada píxel se encuentra entre 0 y 255, por lo que para simplificar cálculos y el aprendizaje en la red se estandariza y se pone el valor entre 0 y 1 dividiendo su valor por 255. Posteriormente la capa convolucional procede crea grupos de kernel, que consisten, en grupos de píxeles cercanos sobre los que se calcula su producto escalar, permitiendo reducir la complejidad de la red.

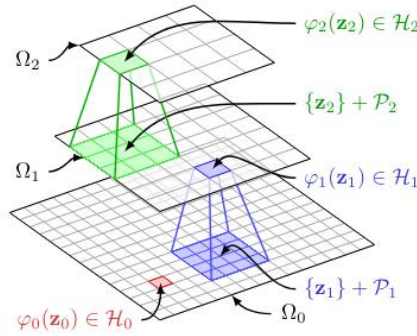


Figura 3.2.1.1 [11] Agrupación en kernel

Tras crear los kernel, se aplica una primera capa de convolución y en su salida, una función de activación, por ejemplo, ReLu que es la más extendida hoy en día. La idea de esta función de activación es discriminar los valores que no se quieran utilizar, y estandarizar aquellos que sí pasan ese valor de corte. Posteriormente se utilizan capas de subsampling o Max-Pooling para reducir la complejidad de la arquitectura. Resumidamente, el subsampling es una práctica común en la codificación de imágenes que, realizando un muestreo de frecuencia de la crominancia, se reduce el ancho de memoria necesario para procesar una imagen sin afectar a la calidad de esta; mientras que el Max-Pooling busca también reducir la complejidad buscando reducir la dimensión de la imagen aplicando filtros, habitualmente de 2x2 o 3x3, escogiendo el máximo valor de los pixeles de cada filtro y reduciendo los 2x2 o 3x3 pixeles a 1.

3.2.2 Redes Neuronales: Funciones de Coste

[8][12] Las funciones de coste determinan el error que existe entre el valor predicho por las redes neuronales y el valor real que debería ser. Por ejemplo, supongamos que estamos analizando que objeto sale en la imagen, si mostramos un coche y la red predice camión, se calcula el error con respecto a la solución esperada, que será menor y distinto a si predice árbol. De esta manera se intenta corregir a través de backpropagation los pesos y optimizar la red buscando el error sea lo más bajo posible y se vaya



reduciendo a lo largo del entrenamiento de la red. Entre las funciones más utilizadas hoy en día son:

- **Cross Entropy Loss:** esta función calcula la pérdida usando la entropía cruzada entre la entrada y el objetivo. Se utiliza para clasificación múltiple, es este tipo de casos la red devuelve un vector de valores con las probabilidades que hay de que pertenezca lo que se quiere clasificar a esa etiqueta. La función puede devolver desde 0 hasta infinito, donde se intentará buscar que sea 0 que quiere decir que la predicción realizada es correcta.
- **Binary Cross Entropy Loss:** especialización de Cross Entropy Loss para problemas de clasificación binaria, como, por ejemplo; si una imagen es un perro o un gato.
- **MSE Loss (Mean Squared Error):** la función mide la pérdida dado utilizando el error cuadrático medio entre cada uno de los elementos predichos y el objetivo. Se suele utilizar para resolver problemas de regresión en aprendizajes automáticos supervisados
- **Hinge Embedding Loss:** para medir la pérdida la función utiliza un tensor x y un tensor y con valores de 1 a -1 y una con la predicción y la otra con el valor real. Busca que la distancia entre las 2 entradas sean iguales, utilizando una medición por pares:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

Figura 3.2.2.1 [8] Fórmula de la función de HE Loss



3.2.3 Redes Neuronales: Funciones de Activación

[8] Las funciones de activación son aquellas que se encargan de definir cómo piensa una neurona de la red neuronal, ya que, son las que determinan cual será el valor de la salida de la neurona en función de la entrada que reciba. Generalmente el valor suele ir entre -1 y 1. Entre las más utilizadas hoy en día se encuentran:

- Función de Heaviside: también conocida como función de escalón unitario, fue la primera en ser utilizada en las redes neuronales debido a su simplicidad. Simplemente, devolverá 0 para todos los valores calculados hasta que sobrepasen el umbral por ejemplo de 0.5, si es superior a este número, devolverá 1.
- Función ReLU (Rectified Lineal Unit): transforma los valores introducidos, eliminando y devolviendo 0 para aquellos valores que sean negativos y devolviendo el mismo valor que se introdujo a la función si este es positivo

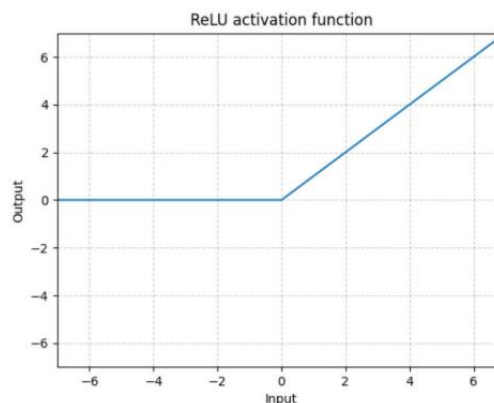


Figura 3.2.3.1 [8] Representación de la función de activación ReLU

- Funciones Sigmoideas: aplican 2 asíntotas horizontales impidiendo que los valores tomen valores inferiores o superiores al rango (-1, 1). De este modo se mitigan los efectos de los outliers.

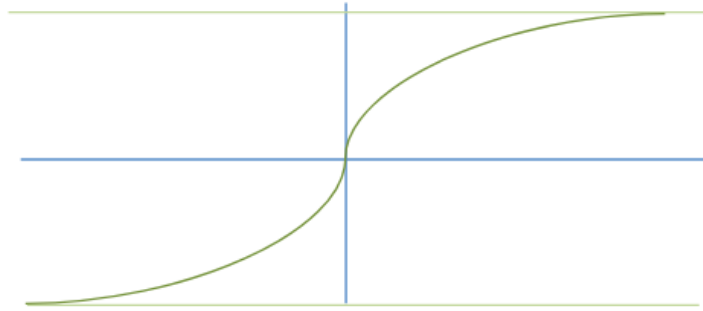


Figura 3.2.3.2 [8] Fórmula de las funciones sigmoideas

- Función logística: convierte en un valor entre (0, 1) casi cualquier entrada, especialmente útil en problemas de clasificación binaria como función de activación de la última capa.
- Función Softmax: o función exponencial normalizada, es una generalización de la función logística

3.2.4 Redes Neuronales: Pooling

[8][13] La capa de reducción o Pooling se utiliza principalmente en redes neuronales que trabajan con imágenes, en la mayoría de los casos, CNNs. Estas capas tratan de reducir las dimensiones espaciales de la entrada a la salida.

Es por ello que esta operación también se conoce como reducción de muestreo ya que el proceso conlleva la pérdida de información, pero agiliza en gran medida el proceso de computación en las siguientes capas, permitiendo también la reducción del sobreajuste. Por lo general, para realizar esta técnica se agrupan los píxeles según un filtro, que suele ser de 2x2 o 3x3, reduciendo el tamaño de la imagen a la mitad o un tercio.

Entre las técnicas de pooling más destacadas podemos encontrar el Max Pooling. Esta técnica recoge el máximo valor de cada filtro creando un mapa de características reducido, y es el más utilizado hoy en día.

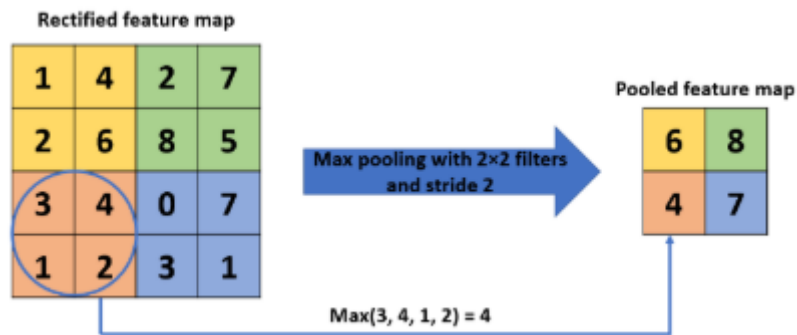


Figura 3.2.4.1 [13] Imagen del funcionamiento de Max Pooling con filtros de 2x2

Otro muy conocido, es el Average Pooling, que consiste en poner como valor del filtro, la media de los valores.

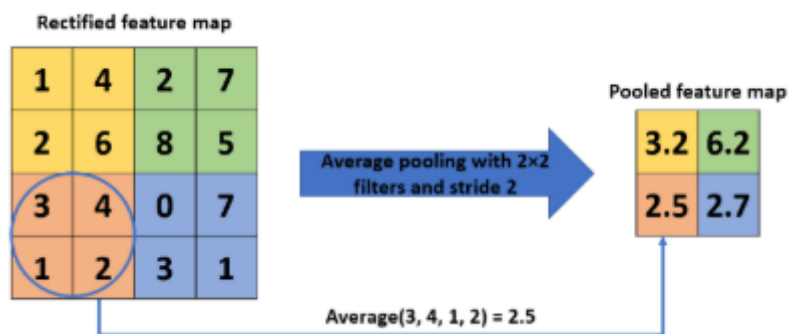


Figura 3.2.4.2 [13] Imagen del funcionamiento de Average Pooling con filtros de 2x2

Con un funcionamiento similar, se también se utiliza el Mixed Pooling, una técnica que combina Max Pooling con Average Pooling. Tiene como objetivo combinar las ventajas de ambos métodos para obtener representaciones más robustas y ricas. La idea es combinar ambos resultados y ponderarlos.



3.2.5 Redes Neuronales: Padding

[8][14] Padding es la técnica opuesta al Pooling, consiste en agregar 0 alrededor de la imagen, siendo útil en casos de que la información de la imagen no está en el centro, si no en borde u esquinas. Esta información no influye en el mapa de características, pero permite a la red neuronal captar mejor el resto de la información de la imagen.

Se suele combinar con una capa de Pooling para reducir el tamaño extra de la imagen que ha sido generado en la capa de Padding, pero de este modo dejándola centrada para mejorar el rendimiento de la red.

3.2.6 CNNs: casos de usos

Las redes neuronales convolucionales o CNNs por sus siglas en inglés, se utilizan y aplican en una amplia variedad de campos hoy en día:

- El principal uso hoy en día es la **visión por computador**, cuyas tareas van desde la clasificación de objetos, como por ejemplo indicar si lo que se muestra en una imagen es un perro o un gato; la detección de objetos, que no solo indica que objeto hay en la imagen, sino que también es capaz de indicar en que posición de la imagen está. También se utiliza dentro de este campo para la segmentación de imágenes, dividiendo los diferentes fondos u objetos que aparecen en la imagen, y para realizar reconocimientos faciales
- También se utiliza para **procesar video**, con análisis en tiempo real, incluyendo seguimiento de objetos o persona a través de un video, o para la vigilancia de actividades inusuales.



- Para la **industria automotriz**, como en los coches autónomos, para la identificación de señales de tráfico, peatones u otros vehículos, por ejemplo.
- En la **industria médica**, analizando imágenes como radiografías para la identificación de posibles enfermedades o lesiones como el cáncer, rotura de huesos.
- **Reconocimiento de texto**, se utilizan para el reconocimiento óptico de caracteres para convertir imágenes a texto
- Para la **Super Resolución** de imágenes, se utilizan para aumentar la resolución de imágenes, permitiendo mejorar la calidad y claridad de estas
- En la **Agricultura**, para monitorear el estado de los cultivos, detección de plagas y enfermedades de las plantas.
- En la **robótica**, para reconocer y manipular objetos en entornos complejos y de alta precisión
- **Industria retail**, donde se usan para la detección y reconocimiento de productos en tiendas mejorando la gestión de los inventarios, por ejemplo.

3.2.7 CNNs: Arquitecturas más conocidas

Desde que se desarrollaron las primeras redes neuronales profundas, se ha estado investigando cuantas capas, de que tipo, las cuantas neuronas en cada capa etc... eran las óptimas para resolver un problema, dando lugar a diferentes arquitecturas, que incluso compiten entre sí para ver cual es capaz de obtener mejores resultados como se verá a continuación. Entre las redes neuronales convolucionales más conocidas hoy en día nos encontramos con:



- **LeNet-5:** [15] red convolucional de 5 niveles creado en 1998, por Yann LeCun extremadamente popular debido a su arquitectura simple pero eficiente, especialmente en el reconocimiento de caracteres a mano, aunque se pueden encontrar múltiples papers donde se optimiza y se utilizan para otros fines como a la detección del COVID-19 en [16] o en [17] donde se ha utilizado para clasificar gases.
- **ImageNet:** [18] no es ningún modelo CNNs, pero se ha incluido en la lista debido a su importancia en esta industria. Destaca por que la base de datos se ha llegado a convertir en estándar en la visión por computación, debido a su gran cantidad de imágenes, 14 millones, etiquetadas, con una amplia variedad de categorías. Usan etiquetas organizadas jerárquicamente según el esquema de WordNet, permitiendo una clasificación estructurada y detallada. Muchos de los modelos que comentaremos a continuación utilizaron este dataset para entrenarse. Se usa en una competencia anual llamada ILSVRC donde compiten los mejores modelos de CNNs.
- **GoogleNet:** [19] una de las redes más famosas actualmente, su principal función consiste en el reconocimiento de imágenes a color y destaca por sus módulos Inception, que permite realizar convoluciones 1x1 reduciendo la dimensionalidad. Su arquitectura de 22 capas quedó primera en el ILSVRC de 2014 con bastante diferencia del resto de competidores, demostrando su eficiencia.
- **AlexNet:** [18] se considera el primer modelo en despertar el interés en la comunidad sobre las CNNs al ganar el desafío de ILSVRC en 2012 con más del 26% de diferencia sobre el resto de sus competidores. Consta de 5 capas convolucionales, 3 de max-pooling y 3 capas densas. Además, aplicaron técnicas de data augmentation, ReLu y dropout para conseguir esa puntuación.



- **ZFNet:** [20] creado en 2013, se basa obteniendo una precisión aún mejor en el concurso. Para ello usaron filtros de 7x7 en vez de los filtros de 11x11, ya que se dieron cuenta que los filtros más grandes perdían mucha información relevante.
- **VGG Net:** [21] desarrollada durante el año 2013, pero se presentó al concurso de 2014, quedando segunda este año, justo por detrás de GoogleNet. Sigue la línea de ZFNet de reducir el tamaño de los filtros, esta vez optaron por utilizar filtros de 3x3.
- **ResNet:** [22] creada por Microsoft en el año 2015, su nombre viene de Residual Network. Con el fin de mejorar el rendimiento de los anteriores ganadores de los concursos, optaron por simplemente añadir más capas, pero esto suponía un problema de rendimiento, conocido como problema del gradiente de fuga. Al haber demasiadas capas, el algoritmo de Backpropagation no es capaz de cambiar los pesos de las primeras ya que se va perdiendo en las capas al ir disminuyendo el gradiente. Es por ello por lo que decidieron basarse en Highway Network, que permitía establecer conexiones directas entre la última capa y las primeras para evitar esa desaparición del gradiente.

3.3 Redes Auto modelables: State of the art

Las redes auto modelables provienen de la rama del AutoML, o, Auto Machine Learning. A raíz de la necesidad de ir creando nuevas redes neuronales, para cada dataset, los científicos e ingenieros dedicados a este campo empezaron a desarrollar nuevas teorías y formas de conseguir crear las redes neuronales adaptadas al dataset con la menor intervención posible por parte del usuario, llegando así también a un público menos experto en esta materia.



Es por ello por lo que a partir del año 2010 comienzan a desarrollarse e investigarse posibles formas de automatizar el proceso de creación de modelos de machine learning, de los cuales se pueden destacar artículos como [23] “Neural Architecture Search with Reinforcement Learning” del 2016, que es un punto de inflexión sobre el uso del AutoML en redes neuronales. En este artículo utilizan reinforcement learning con el fin de encontrar la mejor arquitectura de redes neuronales a los datasets. Para ello utilizan el dataset CIFAR-10. Según ellos mismos, la búsqueda de la mejor arquitectura es autorregresiva lo que quiere decir que busca los mejores hiperparámetros de uno en uno, una idea tomada de la publicación de “Sutskever et al., 2014”. Sin embargo, el método que proponen ellos es diferenciales en que optimiza, no cada hiperparámetro si no la precisión de la red hija, algo que se asemeja a la BLEU Optimization de “Ranzato et al., 2015”; “Shen et al., 2016”, este método propone aprender y optimizarse en base a una recompensa. En resumen, proponen el uso de Reinforcement Learning para una pieza de su arquitectura llamada controlador, que es una red neuronal que propone posibles arquitecturas en base a su aprendizaje por refuerzo que son evaluadas para determinar que rendimiento tienen. En sus experimentos destacan que este algoritmo es capaz de proponer arquitecturas de redes neuronales muy rápido y con un rendimiento competitivo.

Por otro lado, en 2018 se publicó [24] “Efficient neural architecture search via parameters sharing” que aborda la importancia del uso de AutoML en las redes neuronales, para ello propone un nuevo método, ENAS (Efficient Neuronal Architecture Search), que trata de compartir los parámetros entre diferentes arquitecturas en la búsqueda y buscan optimizar la arquitectura a través del reinforcement learning. Como controlador utilizan un LSTM y de dataset CIFAR-10 y destacan que, a diferencia de otros modelos existentes con una solución similar, su solución es más eficiente y económica.

El reinforcement learning, así como variaciones que utilizan esta solución para encontrar la mejor arquitectura no son las únicas que se han investigado, por ejemplo [25] “Auto-Keras: An Efficient Neural Architecture Search System” en 2019, propone la búsqueda a través de un aprendizaje



bayesiano. Auto-Keras es un sistema diseñado para minimizar la intervención humana en el desarrollo de redes profundas, aplicando las siguientes técnicas:

- **NAS:** técnicas avanzadas de búsqueda de arquitecturas
- **Search Space:** sistema que define un espacio de búsqueda que incluye cantidad de capas, conexiones y parámetros
- **Optimización Bayesiana:** para un muestreo inteligente del espacio de posibles hiperparámetros identificando rápidamente configuraciones óptimas para la arquitectura.
- **Procesamiento Automatizado de Datos:** incluye tareas como la normalización, manejo de valores perdidos y transformación de características.
- **Uso de Transfer Learning:** utilizan arquitecturas predefinidas en su algoritmo como punto de partida para nuevas arquitecturas y soluciones.

Además, en los últimos años se han probado otros enfoques y tendencias en este campo entre los que destacamos:

- **Uso de Algoritmos genéticos,** que imitan la evolución biológica para explorar el espacio de posibles arquitecturas, como NNI del que hablaremos posteriormente, junto a este impresionante campo.
- **Aprendizaje por imitación,** donde se trata de buscar la similitud del problema con otro solucionado conocido y usar esa arquitectura
- **Modelos de reducción de dimensión,** utilizando técnicas de reducción de la dimensionalidad para hacer más manejable el espacio de búsqueda.



Un buen ejemplo de esto es [26] “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware”, que propone la búsqueda directamente en los procesos y el hardware, eliminando el uso de proxies y por tanto acelerando el proceso de búsqueda de una arquitectura óptima.

3.3.1 Algoritmos Genéticos y Redes Neuronales

[27] Dentro del campo del aprendizaje automático, o del machine Learning, también se busca que aprenda con la menor supervisión de un experto humano, para ello se han desarrollado muchas técnicas y algoritmos, sin embargo; una de las que es más interesante sin lugar a duda es la aplicación de la evolución y algoritmos genéticos.

Este tipo de algoritmos buscan recrear la evolución de las especies dentro de un algoritmo de software, simulando que las “especies” en este caso modelos de redes neuronales, tienen ciertos atributos o genes como la cantidad de capas, el tamaño de estas, el tipo de las capas.... Un conjunto de estas arquitecturas conforma una generación, que se pone a prueba con un dataset de testeo, después de haber sido entrenados. Aunque el conjunto de prueba si debe ser igual para todos los individuos, la cantidad o el dataset de entrenamiento pueden ser perfectamente un atributo más de la arquitectura.

Una vez han sido evaluadas, las que han obtenido la mejor puntuación se “reproducen” es decir, crean una nueva generación de individuos con características comunes a sus progenitores. Con el fin de abarcar una exploración de posibles arquitecturas, se introduce la mutación, es decir; con cada individuo hijo tiene ciertas características que no proviene de ninguno de sus antecesores. Se vuelven a entrenar, a evaluar y a cruzar... así tantas veces como se quiera explorar, o hasta que se encuentre un individuo que supere la puntuación deseada.

Gracias a este algoritmo, no se requiere intervención de ningún tipo para encontrar la mejor arquitectura a un dataset dado y explora espacios complejos de posibilidades que podrían no ser intuitivas para un operador



humano. Aunque, como desventaja, es posible que le lleve mucho tiempo en encontrar al individuo perfecto ya que, al funcionar con probabilidades en las mutaciones, puede ser que nunca salga la combinación perfecta para el dataset, por lo que es posible que, en vez de encontrar un máximo global en el algoritmo, acabe devolviendo un máximo local.

Entre redes auto modelables basadas en algoritmos genéticos podemos encontrar ejemplos TPOT y Neural Network Intelligence, creada por Microsoft de la cual hablaremos más tarde en el punto 3.3.3.

[28] Tree-Bases Pipeline Optimization Tool o TPOT, es una herramienta de código abierto desarrollada en Python que busca optimizar de manera automática algoritmos de Machine Learning. Aunque si bien es cierto que no está centrada en redes neuronales, se puede adaptar para este propósito como en [29] “TPOT-NN: augmenting tree-based automated machine learning with neural network estimators”. Esta herramienta destaca por el uso de algoritmos genéticos para seleccionar y combinar automáticamente algoritmos y preprocesamientos de datos con el fin de crear el mejor modelo para una tarea de aprendizaje supervisado. Además, es capaz de devolverte un pipeline que puedes revisar modificar y usar con tus datos para recrear el mejor modelo que ha encontrado el algoritmo.

3.3.2 Ejemplos de Redes Auto modelables: AutoKeras

[25] AutoKeras es una biblioteca de AutoML diseñada para facilitar la creación de modelos de Deep Learning, facilitando el proceso de diseño, selección de hiperparámetros y entrenamiento del modelo, permitiendo a usuarios sin amplios conocimientos en la materia crear modelos de alta calidad.

Estos modelos de Deep Learning se basan en la forma de construir modelos mediante técnicas de NAS, evaluando los rendimientos de múltiples modelos, incluso de algunas arquitecturas conocidas como GoogleNet o AlexNet y escoge la que mejor resultado haya dado. Tras ello, realiza un finetuning con los datos para refinar el modelo al mejor rendimiento posible.



3.3.3 Ejemplos de Redes Auto modelables: Neural Network Intelligence

[33] NNI es una plataforma de código abierto desarrollada por Microsoft, que se encarga de automatizar la optimización de hiperparámetros y redes neuronales. El usuario introduce que es lo que quiere hacer y define un espacio de búsqueda y el objetivo de la optimización, a partir de aquí el algoritmo se encarga de encontrar mediante múltiples algoritmos de búsqueda cuales es la mejor solución. Se destaca este modelo por ser prácticamente el único que realiza una búsqueda de arquitectura de la red mediante algoritmos evolutivos, aunque es cierto que no solo utiliza este algoritmo, sino que también usa NASNet, ENAS; DARTS y Network Morphism.

Se integra con ML, siendo compatible con otras librerías de ML como Tensorflow, Pytorch, Keras y scikit-learn, y se puede detener de manera temprana añadiendo ciertos parámetros para aquellos resultados que no parezcan prometedores. Además, está integrado tanto para servidores en local como para desplegarse en la nube.

Por último, existe una interfaz web que permite monitorizar y visualizar el progreso y resultado de las arquitecturas.



Capítulo 4: Marco Metodológico

A lo largo de este capítulo se exponen los recursos y herramientas utilizados durante del proyecto. Para desarrollar este proyecto, lo primero que se ha realizado es establecer una metodología de trabajo donde:

- Se ha buscado posibles soluciones para implementar un algoritmo evolutivo, finalmente se ha implementado un ciclo donde se crean 10 individuos en cada generación, que, mediante torneo, se escogen a los 4 mejores, y se generan 4 nuevos individuos cada generación que sustituirán a los 4 peores. Además, cada generación, entrenará el modelo y evaluará su precisión, con las mismas epochs para que partan de las mismas condiciones. Los nuevos individuos se generan mediante cruce, que, a través de probabilidades, se escogerán un atributo de un predecesor o de otro. Posteriormente, mediante un umbral de probabilidad bajo, cada atributo se somete a una posible mutación, es decir; se genera un nuevo valor para ese atributo.

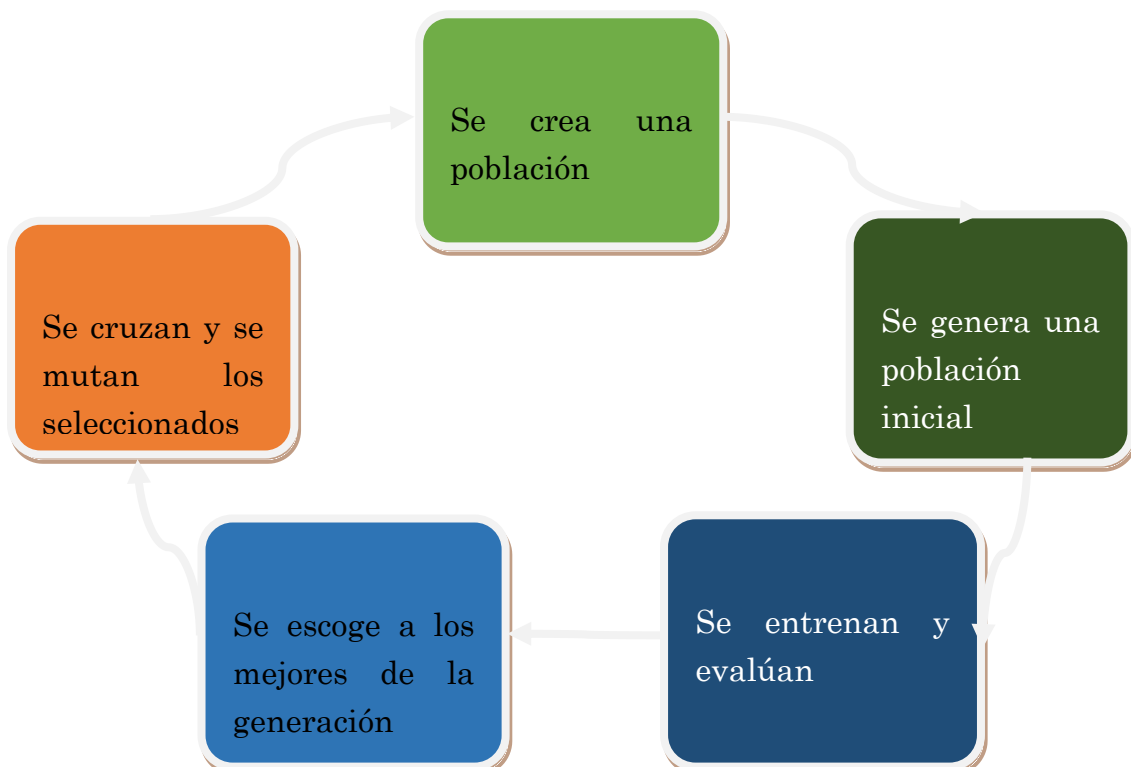




Figura 4.0.1 Ciclo del funcionamiento del algoritmo evolutivo

- Cuál es la mejor forma representar los atributos de una CNN, para poder ser combinados modificados por el algoritmo, que se reflejaran mediante diccionarios de Python. Entre estas, se escoge la cantidad de capas convolucionales, capas densas, tamaño de los filtros, tamaño del stride.... Para poder crear los modelos, se desarrolló una función genérica que, usando Pytorch, construye el modelo y lo devuelve para que el algoritmo pueda entrenarlo y evaluarlo. Al final del algoritmo, se guarda como un atributo más la puntuación obtenida.
- Qué métrica es la más útil para tener en cuenta. Dado que se busca que funcione lo mejor posible, la mejor métrica que podríamos tener en cuenta para una CNN es la precisión, ya que se busca que clasifique lo mejor posible las imágenes.
- Arquitectura del sistema, ya que la idea del proyecto es facilitar la creación de redes neuronales a usuarios no expertos en este ámbito, se ha desarrollado una arquitectura de microservicios, con un servidor en backend que procesa y ejecuta el algoritmo y una interfaz que permite a los usuarios utilizar con facilidad la aplicación.
- Herramienta de desarrollo, finalmente se desarrolló el proyecto en GitHub para llevar un control de versiones, Python, para desarrollar el algoritmo y los modelos, junto a Pytorch para las redes neuronales y Flask para el servidor. Con respecto a la interfaz se ha desarrollado en ReactJS. Respecto a los datasets de prueba, se ha utilizado CIFAR10, debido a que es un dataset relativamente

pequeño pero complejo lo que permite comprobar si el algoritmo está trabajando correctamente, dentro de un coste computacional y de tiempo permisibles. Como herramienta de desarrollo, se ha escogido VS Code por la facilidad que tiene para poder desarrollar tanto la interfaz como todo el backend en un solo IDE. Para el desarrollo y control de versiones se ha usado GitHub.

- Hardware, se ha utilizado un ordenador con gráfica NVIDIA RTX 3060 de 16GB VRAM, aprovechando sus capacidades con CUDA, procesador AMD 5 5600 X, 16 GB de memoria RAM.

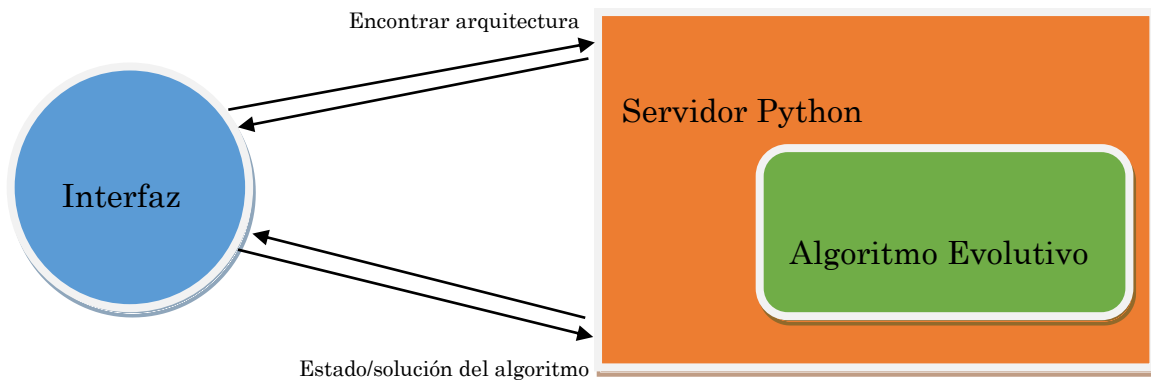


Figura 4.0.2 Esquema de la arquitectura implementada

4.1 Población y muestra

La población y muestra del estudio para este proyecto se ha utilizado el dataset de CIFAR-10 que debido a los límites del hardware disponible es el único dataset con una ejecución larga y “completa”, y que dispone de cierto nivel de complejidad.

Este dataset se compone de 60000 imágenes a color, clasificadas en 10 categorías con 6000 imágenes por categorías. Estas son aviones, automóviles, pájaros, gatos, ciervos, perros, ranas, caballos, barcos y



camiones. Para el entrenamiento se han utilizado 50000 imágenes del dataset y reservando 10000 imágenes para la fase de testeo.

4.2 Objetivos de investigación

El principal objetivo de este proyecto es desarrollar un algoritmo genético que sea capaz de optimizar redes neuronales convolucionales, y evaluar la viabilidad como método de Auto Machine Learning para redes neuronales. Específicamente se pretende:

- Desarrollar el algoritmo genético: diseñar y programar un algoritmo genético que pueda generar entrenar y evaluar redes neuronales convolucionales.
- Evaluar la viabilidad del algoritmo, examinar la eficiencia en términos de cuanta capacidad se necesita para lograr una alta precisión en tareas de clasificación de imágenes
- Estudiar posibles mejoras del algoritmo, como evitar el cuello de botella que se produce al entrenar a los individuos

4.3 Variables intervinientes e instrumentos de evaluación

En este proyecto mediremos la precisión de las arquitecturas que se van desarrollando a lo largo de la ejecución, poniendo el foco de atención en la arquitectura que se entrega como “mejor” al usuario.

Para ello se han utilizado trazas de log al final de cada ciclo de vida de la generación de arquitecturas que incluye datos como la precisión de cada modelo, y la arquitectura.



También se medirá el tiempo de ejecución del algoritmo, usando un cronómetro que empieza cuando se empieza a ejecutar el algoritmo y se para una vez se ha terminado.

4.4 Diseño experimental

Para probar el desarrollo de este algoritmo se ha decidido utilizar los siguientes métodos:

- Desarrollo aislado: un notebook de Python con todo el desarrollo y explicaciones necesario para comprender lo que se pretende realizar en el y con un dataset controlado ya que pertenece a la librería de Pytorch. En este entorno se desarrolla todo el algoritmo y se ajusta para que sea lo mejor posible.
- Implementación del servidor: tras haber investigado y desarrollado el algoritmo, se desarrolla el servidor aprovechando el código y las funcionalidades desarrolladas en el otro entorno con el fin de probar las condiciones en las que un usuario puede usar el algoritmo.

4.5 Modelo de análisis de datos

Para analizar los resultados del modelo, utilizaremos métodos cuantitativos, como la precisión del modelo y el tiempo de ejecución.

Para analizar los resultados obtenidos mediremos la precisión del mejor modelo de cada generación con el fin de ver como se va adaptando al dataset, guardando esta información en una colección:

```
population.sort(key=lambda x: x['fitness'], reverse=True)

# Preservamos el mejor individuo (elitismo)
mejor_individuo = population[0]
top_models.append(mejor_individuo)
```



Además, se medirá el tiempo que ha tardado en completarse cada ejecución del algoritmo. Para ello la mejor forma es coger la información directamente del editor de Visual Studio Code ya que nos proporciona el tiempo de ejecución en el trozo de código específico como, por ejemplo:



Figura 4.5.1 Ejemplo del tiempo de ejecución en el IDE de VS Code

Con el fin de entender mejor los datos utilizaremos la librería de matplotlib de Python para generar las imágenes que nos permitan visualizar estos resultados en gráficas:

```
# Extraemos los valores de fitness de cada modelo
fitness_values = [model['fitness'] for model in top_models]

# Creamos gráfico
plt.figure(figsize=(10, 6))
plt.plot(range(len(fitness_values)), fitness_values, marker='o',
         linestyle='-', color='b')
plt.title('Fitness del mejor modelo de cada generación')
plt.xlabel('Generación')
plt.ylabel('Valor de Fitness')
plt.grid(True)
plt.show()
```

Por limitaciones de hardware para este análisis se recogerán los datos de 13 generaciones. Cada una de estas generaciones estarán formadas por 10 individuos. De este modo recogeremos los datos de los mejores modelos de cada generación para mostrar la evolución del algoritmo. Son 13 generaciones por que a partir de ahí el hardware suele dar errores de falta de memoria, o empieza a tardar más de 1 hora por iteración (epoch) lo que vuelve inviable seguir con el experimento:

- CUDA out of memory. Tried to allocate 1.60 GiB. GPU 0 has a total capacity of 12.00 GiB of which 0 bytes is free. Of the allocated memory 17.65 GiB is allocated by PyTorch, and 63.76 MiB is



Epoch 1/7: 100%	<div></div>	391/391	[00:16<00:00,	24.34batch/s,	training_loss=4.219]
Epoch 2/7: 100%	<div></div>	391/391	[00:16<00:00,	24.42batch/s,	training_loss=3.993]
Epoch 3/7: 100%	<div></div>	391/391	[00:15<00:00,	24.70batch/s,	training_loss=3.790]
Epoch 4/7: 100%	<div></div>	391/391	[00:16<00:00,	24.42batch/s,	training_loss=3.611]
Epoch 5/7: 100%	<div></div>	391/391	[00:15<00:00,	24.69batch/s,	training_loss=3.793]
Epoch 6/7: 100%	<div></div>	391/391	[00:16<00:00,	24.39batch/s,	training_loss=3.710]
Epoch 7/7: 100%	<div></div>	391/391	[00:15<00:00,	24.81batch/s,	training_loss=3.574]
{ 'filters': [64, 64], 'kernel_sizes': [7, 3], 'learning_rate': 1e-05, 'fully_connect					
Epoch 1/7: 100%	<div></div>	391/391	[13:58<00:00,	2.14s/batch,	training_loss=3.419]
Epoch 2/7: 100%	<div></div>	391/391	[13:48<00:00,	2.12s/batch,	training_loss=2.909]
Epoch 3/7: 100%	<div></div>	391/391	[14:22<00:00,	2.21s/batch,	training_loss=2.976]
Epoch 4/7: 100%	<div></div>	391/391	[13:51<00:00,	2.13s/batch,	training_loss=2.629]
Epoch 5/7: 100%	<div></div>	391/391	[13:57<00:00,	2.14s/batch,	training_loss=2.438]
Epoch 6/7: 100%	<div></div>	391/391	[15:22<00:00,	2.36s/batch,	training_loss=2.314]
Epoch 7/7: 100%	<div></div>	391/391	[15:21<00:00,	2.36s/batch,	training_loss=1.792]

Figura 4.5.1 Ejemplo de cuando comienza a dispararse el tiempo de ejecución por el hardware (no está sacado de las pruebas realizadas si no de una ejecución rápida posterior para ejemplarizar el problema)

Por último, con el fin de obtener una media de tiempo relativamente real, ejecutaremos el algoritmo por 10 veces, tras haber completado las 13 generaciones.



Capítulo 5: Presentación de análisis, resultados y discusión

Durante este apartado observaremos los resultados obtenidos tras completar el proyecto, como, por ejemplo, si el algoritmo evoluciona con las generaciones, el tiempo que conlleva la ejecución del algoritmo dado el hardware utilizado durante el mismo.

5.1 Análisis descriptivo de la muestra

Para probar el algoritmo se ha utilizado el dataset de CIFAR-10, un dataset ampliamente utilizado en el mundo de las redes neuronales convolucionales debido a su simplicidad permitiendo que las ejecuciones sean relativamente rápidas, pero al mismo tiempo la complejidad necesaria para ver si el algoritmo realmente evoluciona. Entre sus características encontramos:

- 60000 imágenes a color
- 10 categorías: aviones, automóviles, pájaros, gatos, ciervos, perros, ranas, caballos, barcos y camiones
- 6000 imágenes por categoría
- 32x32 píxeles en cada imagen.

5.2 Resultados obtenidos

Tras la ejecución del algoritmo siguiendo las pautas del apartado 4.5, se ha obtenido la información necesaria y las gráficas para analizar el rendimiento del algoritmo obtenido durante este proyecto.

Comenzaremos analizando el rendimiento del modelo:

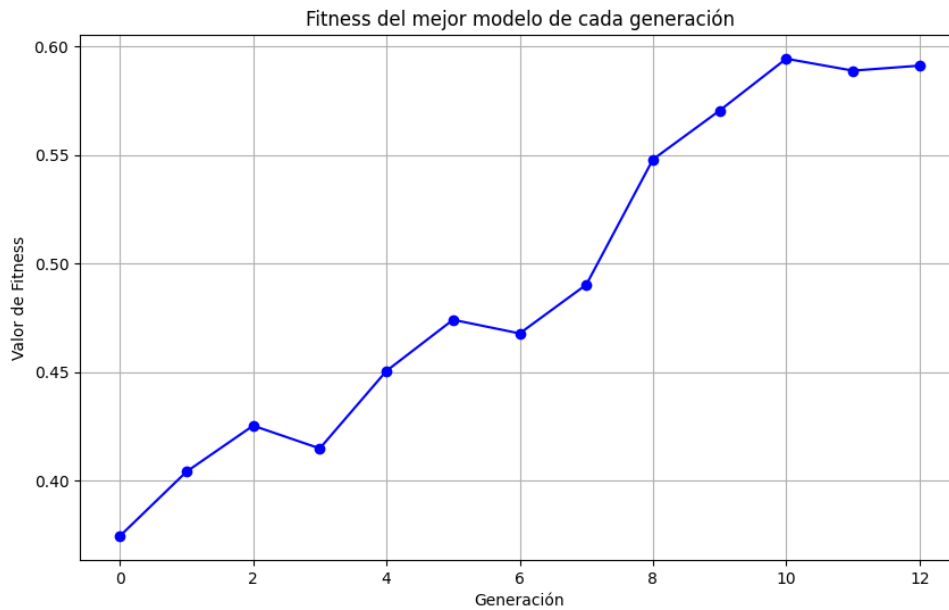


Figura 5.2.1 Imagen de la evolución de la precisión del mejor modelo obtenido en cada generación.

Como se puede observar en el gráfico generación a generación va encontrando mejores resultados, evolucionando desde el 0.3 de precisión al 0.6 en solo 12 generaciones. Aunque si bien es cierto que estos resultados podrían ser muy distintos si se hubiera entrenado cada individuo con más iteraciones, nos permite apreciar como la combinación y mutación de las arquitecturas permite generar individuos que aprenden más rápido y mejor.

Ahora analizaremos el tiempo que tarda en ejecutar el algoritmo completo siguiendo las pautas del apartado 4.5:

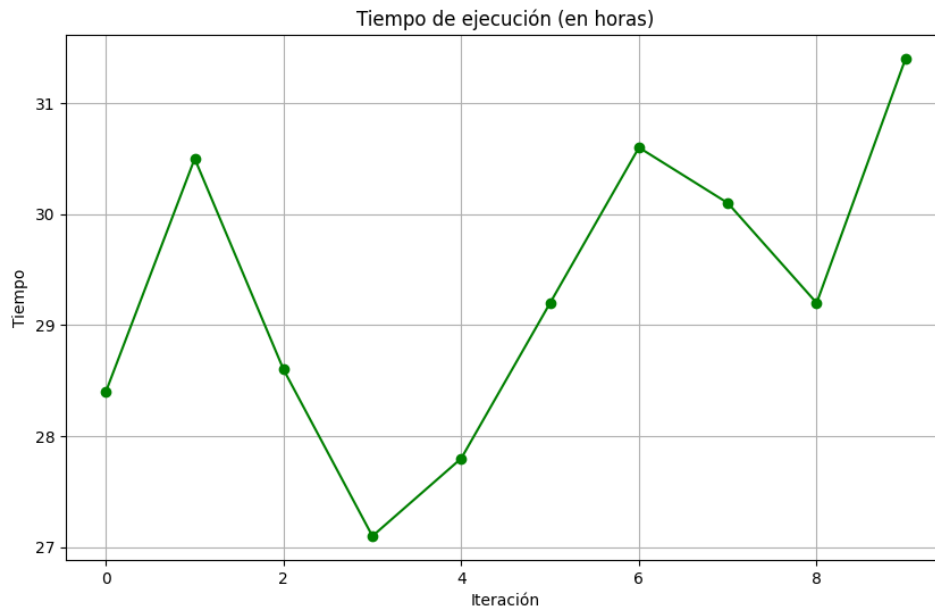


Figura 5.2.2 Imagen del tiempo de ejecución total del algoritmo en horas en las 10 iteraciones del algoritmo completas.

El tiempo de ejecución del algoritmo en los componentes hardware expuestos y bajo las condiciones software (epochs, generaciones, individuos) expresados en el punto 4.5 del capítulo anterior resultan realmente lentos, la media ronda las 29 horas (28.7). Esto indica un problema con el algoritmo, es lento.

5.3 Discusión y análisis de los resultados

Tras analizar los resultados en el apartado anterior, observamos que el algoritmo funciona y es capaz de generar nuevas generaciones que avanzan y obtienen mejores resultados en cuanto a precisión al menos se refiere.

Sin embargo, en un algoritmo también es importante el tiempo de ejecución de este, en este caso, como se puede observar es muy lento ya que el tiempo total que tarda es la suma total de los tiempos de entrenamiento y evaluación de cada individuo, de cada generación. Dicho de otro modo, supongamos que cada individuo, tarda exactamente 1 minuto por epoch,



para calcular cuánto tiempo necesita el algoritmo para aprender, habría que multiplicarlo por el número de epochs, después por la cantidad de individuos y a su vez por la cantidad de generaciones. En un caso de 25 epochs, 10 individuos por generación y 20 generaciones, el tiempo total sería de:

$$1 * 25 * 10 * 20 = 5000$$

En ese hipotético caso donde los tiempos son exactos y no varían, se tardarían 5000 minutos, es decir; 83.33 horas o 3.47 días.

El tiempo de ejecución del algoritmo para la búsqueda de arquitecturas de redes neuronales resulta considerablemente elevado. Aun así, si se compara con el tiempo requerido por un especialista humano con conocimientos en el área, la situación cambia.

Supongamos que un humano necesita aproximadamente tres cuartas partes del tiempo que toma el algoritmo para encontrar una arquitectura adecuada. Este especialista, debido a su experiencia, no requeriría probar 200 arquitecturas distintas. Es más probable que necesite evaluar alrededor de la mitad o incluso menos, considerando su capacidad para hacer ajustes informados basados en el análisis del dataset y la efectividad de diferentes configuraciones. El tiempo total que un especialista humano podría necesitar se estima en 42 horas de trabajo, la mitad tiempo medio del algoritmo, considerando que necesitará probar menos de la mitad arquitecturas al tener ya conocimientos y experiencias previas en el desarrollo de este tipo de redes, pero sumando que necesita para realizar los ajustes, y hacer el prueba y error de estos hasta encontrar la mejor arquitectura. Al dividir este tiempo por una jornada laboral estándar de 8 horas diarias, se obtiene un tiempo aproximado de 5.25 días laborales. Además, si el especialista deja entrenando una arquitectura al finalizar cada jornada, el tiempo total se reduciría a 5 horas, considerando los 25 minutos que toma cada arquitectura en entrenarse y evaluarse. (Es difícil tener un tiempo de un operario humano debido a que cada persona es diferente y tarda más o menos en realizar una misma tarea, es por ello por lo que se utiliza una estimación basada en el tiempo que ha tardado una



persona humana experta en realizar esta tarea, ajustado a las justificaciones anteriormente expuestas)

Por esta razón, comparando ambos tiempos, el algoritmo desarrollado puede suponer una ventaja significativa para ahorrar tiempos en el desarrollo de redes neuronales, sin embargo, aún es posible optimizarlo más. Para ello en el siguiente capítulo se expondrán algunas soluciones al cuello de botella creado a la hora de entrenar y evaluar las diferentes arquitecturas.



Capítulo 6: Conclusiones

El objetivo general del proyecto es crear una aplicación web que permita a cualquier usuario obtener a través del backend la mejor arquitectura encontrada para el dataset del usuario. Para ello se dividió el proyecto en ciertos objetivos específicos:

1. Investigar otras posibles soluciones en el mercado.
2. Investigar y definir que son los algoritmos evolutivos y redes neuronales convolucionales.
3. Desarrollar una arquitectura de redes neuronales, que permita generar un modelo en base a un vector.
4. Desarrollar un algoritmo evolutivo que permita ir cruzando y mutando las diferentes arquitecturas.
5. Desarrollar una interfaz en REACT.
6. Desarrollar una arquitectura REST-API para combinar el backend con el frontend.

Tras todo lo expuesto en este documento se puede concluir que se ha cumplido con éxito todos los objetivos del proyecto, tanto específicos como generales, ya que en el capítulo 3 se detalla el primer apartado de los objetivos específicos, mientras que, en el capítulo 4 se detalla como se ha desarrollado el algoritmo, así como la interfaz y la arquitectura API-REST que como se ve a continuación funciona correctamente tanto el algoritmo como la arquitectura:



```

127.0.0.1 - - [17/Jun/2024 21:38:15] "POST /result HTTP/1.1" 202 -
Datos eliminados si existian.
Parámetro 'bilbao@gmail.com' no encontrado.
127.0.0.1 - - [17/Jun/2024 21:38:58] "POST /result HTTP/1.1" 200 -
CUDA (GPU) está disponible en tu sistema.
127.0.0.1 - - [17/Jun/2024 21:40:25] "POST /upload HTTP/1.1" 202 -
*****
Generation: 0
*****
{'num_conv_layers': 2, 'filters': [64, 64], 'kernel_sizes': [5, 7], 'learning_rate': 0.1, 'fully_connected': 1, 'dropout': 0}
Epoch 1/20: 100% | 391/391 [00:39:00:00, 9.83batch/s, training_loss=58.391]
Epoch 2/20: 54% | 212/391 [00:18:00:16, 10.69batch/s, training_loss=32.074]P
Parámetro 'bilbao@gmail.com' no encontrado.
127.0.0.1 - - [17/Jun/2024 21:41:24] "POST /result HTTP/1.1" 200 -
Epoch 2/20: 100% | 391/391 [00:34:00:00, 11.23batch/s, training_loss=27.130]
Epoch 3/20: 91% | 354/391 [00:29:00:02, 13.52batch/s, training_loss=41.651]

```

Figura 6.1.1 trazas de ejemplo del servidor Python.

Además de una sencilla interfaz, pero con un diseño profesional accesible para cualquier usuario que desee encontrar la mejor arquitectura para su dataset:

AutoModelizer

Introduce a dataset to discover its best architecture

Search the best architecture

Introduce your data

Name

Email

Number of descendency

Number of epochs

Number of classes

☒ Is it splitted into train and test?

Work your magic →

Check the results, please notice it could take several hours

Name

Email

check results →

Learning Rate: 0.00001

Number of Convolutional Layers: 2

Kernel Sizes: 1, 5

Filters: 32, 128

Fully connected Layers: 0

Dropout: 0

Accuracy: 0.5637

Figura 6.1.1 Interfaz de la aplicación.



Cabe destacar que, aunque en este documento no se puede plasmar, la interfaz dispone de animaciones que hacen más atractivo su uso a los usuarios.

6.1 Limitaciones

El hardware del que se disponía ha limitado la cantidad de pruebas, así como posiblemente la calidad de estas para realizar unas pruebas más exhaustivas y con datasets aún más complejos, que permitiera observar si es capaz de funcionar este algoritmo aún en los casos más difíciles.

6.2 Futuras líneas de trabajo

Este proyecto se puede mejorar utilizando las capacidades en paralelo de tantos ordenadores o servidores como individuos haya en cada generación, de tal manera que optimizamos el tiempo de entrenamiento y evaluación del algoritmo, al tener como tiempo máximo de esta fase el más lento de las arquitecturas, en vez de la suma total de todos los tiempos.

Una posible solución sería usar o adaptar Pytorch Distribute Package, ya que se usa hoy en día en LLMs para entrenar el modelo en máquinas distribuidas y así mejorar los tiempos de entrenamiento.

Tras analizar el producto entero, se ha llegado a la conclusión de posibles mejoras que se podrían implementar en el proyecto a posteriori de su entrega:

- Implementar un sistema distribuido donde se entrenen los individuos: tras analizar el proyecto nos damos cuenta de que se tarda mucho en completar el algoritmo, debido a que existe un cuello de botella en la etapa de entrenamiento y evaluación de los individuos de cada generación. Esto se debe a que cada modelo se entrena después de haber completado el anterior, pero si se



distribuye y se entrenan en paralelo, se reduciría el tiempo, al individuo más lento de todos, en vez de a la suma de estos.

- Adaptarlo para que no solo funcione con CNNs: tras analizar el algoritmo se ha considerado factible modificarlo para que se adapte a cualquier tipo de dato. Para ello sería necesario modificar el tipo de atributos para que acepte otros tipos de capas o que sea capaz de generar otro tipo de arquitecturas como los de Autoencoders o LLMs.
- Seguimiento del entrenamiento: en vez de aportar los datos de los resultados al final, con el fin de hacer más usable la aplicación, se puede adaptar la interfaz para que muestre cual es el progreso que tiene actualmente el algoritmo.
- Permitir parar el algoritmo: introducir una opción de parar el algoritmo si al observar durante su entrenamiento que ya se ha llegado al límite de precisión que quiera el usuario para no malgastar recursos.

6.3 Consideraciones finales

Finalmente, observando los resultados del proyecto, el trabajo realizado se ha cumplido adecuadamente con los objetivos propuestos, tanto el desarrollo del algoritmo, como la implementación de un cliente servidor comunicado por REST, además de haber expuesto la base teórica en las que se basa el proyecto, así como otras soluciones existentes en el mercado que abordan el mismo problema. Todo este trabajo práctico se puede observar y utilizar en el repositorio de GitHub disponible en el Anexo I

A lo largo del TFM he podido disfrutar y aprender mucho sobre como poder investigar y desarrollar nuevas herramientas que permitan mejorar el futuro, y lo apasionante que es resolver y enfrentarse a nuevos retos como juntar 2 áreas tan apasionantes de la informática. He podido aprender cómo



funcionan los algoritmos genéticos y ser capaz de extrapolarlos a cualquier problema de optimización, no solo a redes neuronales convolucionales. Además, he podido aprender como extrapolar las características de las redes neuronales a un conjunto de datos, y crear funciones genéricas capaz de construir los modelos en base a los datos descriptivos del mismo. Por último, he podido aprender a desarrollar una interfaz web profesional y con todo lo requerido para que se integre correctamente con el algoritmo.

Por último, quisiera agradecer especialmente a mi tutor del TFM Don Alberto Partida Rodríguez por su ayuda, consejos y revisiones del proyecto, así como a todo el profesorado del Máster en Inteligencia Artificial de la Universidad Alfonso X por haberme enseñado lo emocionante que es este campo y las bases teóricas y prácticas necesarias para desarrollar por completo este proyecto. Además, no puedo terminar este proyecto sin reflejar mi deseo de que este modelo sirva de base e inspiración para otros y ayude al futuro de la industria del Deep Learning.



Bibliografía

- [1] Haifeng Jin, François Chollet, Qingquan Song, and Xia Hu. "AutoKeras: An AutoML Library for Deep Learning." the Journal of machine Learning research 6 (2023): 1-6 <https://jmlr.org/papers/v24/20-1355.html>
- [2] Arranz de la Peña, Jorge & Parra Truyol, Antonio (2007). Algoritmos Genéticos.
https://d1wqtxts1xzle7.cloudfront.net/35889597/Algoritmos_Geneticos_app-libre.pdf?1418170375=&response-content-disposition=inline%3B+filename%3DALGORITMOS_GENETICOS.pdf&Expires=1713384155&Signature=IRuY4748-itaLM5KmNMvp1OM3IBqw0W3JHpvvRf4wIkDwMZrJj9mD8RkGxei wvnCGv91bia10gW-7m2IbtZpOyzsO9ljhYRQWA0xN7-Kbm9DRFAjc80VbNANv47OoAUEnfuWP7je42jsaj6qo3OWIrFNoAKCCRQ1htc-upeJvpDrEm~ZGaZpqzYKzV7Jj9ZIUJgJrW5bT2XWTBA276U0qkTMfllzq~~sLCswdtOCB4IPGCikRytVhiipCDrQ8SbbyTVuEMm~MyKV~QyQy4aI2H9y~r6EltjJuWff9OnQ-mHylR5Wvlo8ZqYEOXCVvz297zlM2vKWnQ~zQ2OBbltnDw_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA
- [3] Suárez Fernando & Ordóñez, Adriana (2010) De Gregor Mendel y la docencia sin licencia
<https://revistas.javeriana.edu.co/index.php/vnimedica/article/download/16052/12846>
- [4] Miko, I. (2008). Gregor Mendel and the principles of inheritance. Nature Education, 1(1), 134. https://hagenetics.org/hh/wp-content/uploads/2015/04/371_Lec4.pdf
- [5] Roeva, O. (Ed.). (2012). Real-world applications of genetic algorithms. BoD—Books on Demand.



https://books.google.es/books?hl=es&lr=&id=L-KdDwAAQBAJ&oi=fnd&pg=PR11&dq=Real-World+Applications+of+Genetic+Algorithms&ots=jfocxvHUSp&sig=I7qVdhx4_BgB4os03oadya7MmY#v=onepage&q=Real-World%20Applications%20of%20Genetic%20Algorithms&f=false

- [6] Bodenhofer, U. (2003). Genetic algorithms: theory and applications. Lecture notes, Fuzzy Logic Laboratorium Linz-Hagenberg, Winter, 2004. <https://www.flil.jku.at/div/teaching/Ga/notes.pdf>
- [7] Schmitt, L. M. (2001). Theory of genetic algorithms. Theoretical Computer Science, 259(1-2), 1-61. <https://www.sciencedirect.com/science/article/pii/S0304397500004060>
- [8] Bilbao Lara, Carlos (2022). Modelización de una red neuronal capaz de detectar emoticonos en imágenes. Proyecto Fin de Carrera / Trabajo Fin de Grado, E.T.S. de Ingenieros Informáticos (UPM) 2-15 <https://oa.upm.es/71006/>
- [9] Park, W. J., & Park, J. B. (2018). History and application of artificial neural networks in dentistry. European journal of dentistry, 12(04), 594-601. https://www.thieme-connect.com/products/ejournals/html/10.4103/ejd.ejd_325_18
- [10] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
- [11] Mairal, J., Koniusz, P., Harchaoui, Z., & Schmid, C. (2014). Convolutional kernel networks. *Advances in neural information processing systems*, 27. https://proceedings.neurips.cc/paper_files/paper/2014/hash/81ca0262c82e712e50c580c032d99b60-Abstract.html



- [12] Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4, No. 4, p. 738). New York: springer.
<https://link.springer.com/book/10.1007/978-0-387-45528-0>
- [13] Gholamalinezhad, H., & Khosravi, H. (2020). Pooling methods in deep neural networks, a review. arXiv preprint arXiv:2009.07485.
<https://arxiv.org/pdf/2009.07485>
- [14] Dwarampudi, M., & Reddy, N. V. (2019). Effects of padding on LSTMs and CNNs. arXiv preprint arXiv:1903.07288.
<https://arxiv.org/abs/1903.07288>
- [15] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
<https://ieeexplore.ieee.org/abstract/document/726791>
- [16] Islam, M. R., & Matin, A. (2020, December). Detection of COVID 19 from CT image by the novel LeNet-5 CNN architecture. In *2020 23rd International Conference on Computer and Information Technology (ICCIT)* (pp. 1-5). IEEE.
<https://ieeexplore.ieee.org/abstract/document/9392723>
- [17] Wei, G., Li, G., Zhao, J., & He, A. (2019). Development of a LeNet-5 gas identification CNN structure for electronic noses. *Sensors*, 19(1), 217.
<https://www.mdpi.com/1424-8220/19/1/217>
- [18] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6), 84-90.
<https://dl.acm.org/doi/abs/10.1145/3065386>
- [19] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern*



- recognition (pp. 1-9). https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deep_With_2015_CVPR_paper.html
- [20] Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13* (pp. 818-833). Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-10590-1_53
- [21] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. <https://arxiv.org/abs/1409.1556>
- [22] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778). https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html
- [23] Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*. <https://arxiv.org/pdf/1611.01578>
- [24] Pham, H., Guan, M., Zoph, B., Le, Q., & Dean, J. (2018, July). Efficient neural architecture search via parameters sharing. In *International conference on machine learning* (pp. 4095-4104). PMLR. <http://proceedings.mlr.press/v80/pham18a.html>
- [25] Jin, H., Song, Q., & Hu, X. (2019, July). Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining* (pp. 1946-1956). <https://dl.acm.org/doi/abs/10.1145/3292500.3330648>



- [26] Cai, H., Zhu, L., & Han, S. (2018). Proxylessnas: Direct neural architecture search on task and hardware. arXiv preprint arXiv:1812.00332. <https://arxiv.org/abs/1812.00332>

- [27] Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2), 99-127. <https://ieeexplore.ieee.org/abstract/document/6790655>

- [28] Olson, R. S., Bartley, N., Urbanowicz, R. J., & Moore, J. H. (2016, July). Evaluation of a tree-based pipeline optimization tool for automating data science. In Proceedings of the genetic and evolutionary computation conference 2016 (pp. 485-492). <https://dl.acm.org/doi/abs/10.1145/2908812.2908918>

- [29] Romano, J. D., Le, T. T., Fu, W., & Moore, J. H. (2021). TPOT-NN: augmenting tree-based automated machine learning with neural network estimators. *Genetic Programming and Evolvable Machines*, 22, 207-227. <https://link.springer.com/article/10.1007/s10710-021-09401-z>

- [30] Gridin, I. (2022). Introduction to Neural Network Intelligence. In Automated Deep Learning Using Neural Network Intelligence: Develop and Design PyTorch and TensorFlow Models Using Python (pp. 1-30). Berkeley, CA: Apress. https://link.springer.com/chapter/10.1007/978-1-4842-8149-9_1



Anexos



1. ANEXO I

Con el fin de poder entender mejor y poder utilizar todo la investigación y desarrollo del proyecto, se proporciona un enlace a la plataforma de GitHub donde está el código, así como una guía de instalación y uso:

<https://github.com/Carlosbil/AutoModelizer>