uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática

Curso 2018-2019

Trabajo Fin de Grado

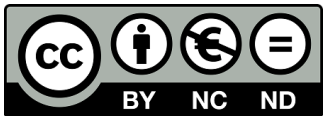
"Técnicas de computación evolutiva aplicadas al diseño de redes de neuronas profundas"

José Antonio Gómez Caudet

Tutor/es

Yago Sáez Achaerandio

05 de julio de 2019



Resumen

Encontrar topologías de redes de neuronas es un proceso de prueba y error que consume una gran cantidad de tiempo de personal altamente cualificado como pueden ser expertos en redes neuronales. Debido a los nuevos tipos de redes profundas que se emplean actualmente, como pueden ser las redes convolucionales y las redes recurrentes, este período de busca pasa a ser todavía mayor.

El uso de algoritmos genéticos para la creación de topologías de redes neuronales, de forma que se evite dedicar capital humano a su construcción, se denomina neuroevolución y es un campo relativamente inexplorado debido a que solo es posible a nivel tecnológico desde hace unos pocos años.

El objetivo fundamental del trabajo es el de desarrollar un sistema de generación de redes neuronales complejas teniendo solamente por entrada un conjunto de datos de entrenamiento, sin ningún tipo de pericia de un experto o ayuda externa. Para lograrlo se partirá desde el trabajo realizado por Baldominos et al. [1], en el que se emplea la neuroevolución para alcanzar un nuevo estado del arte sobre el *dataset* MNIST.

En el presente trabajo se propone un algoritmo genético que evoluciona un gran número de parámetros de la topología de redes neuronales convolucionales. Para comprobar la eficacia de la propuesta se evaluará sobre un conjunto de datos usado frecuentemente para *benchmarks* como el es CIFAR-10. Este *dataset* contiene imágenes de baja resolución que, tras el entrenamiento, la red deberá ser capaz de clasificar en función de lo que contengan.

Tras realizar sucesivas implementaciones, la versión más eficaz del algoritmo obtiene redes con un porcentaje de acierto de 88'3 %, lo que, si bien no establecen un nuevo estado del arte en la evaluación del conjunto de datos objetivo, sí supone un resultado competitivo, teniendo en cuenta los recursos disponibles, y muestra la gran capacidad del campo de estudio.

Por último, se extraerán conclusiones en función de los resultados obtenidos y se propondrán nuevas rutas para mejorar cuando los medios disponibles avancen.

Palabras clave: Neuroevolución, Algoritmos genéticos, Redes de neuronas convolucionales, CIFAR10

Índice general

Índice de figuras

Índice de tablas

1. Introducción

En este apartado se cubrirán las motivaciones detrás de este proyecto, los objetivos que se intentan alcanzar durante su desarrollo, y se realizará un breve resumen sobre el contenido y la estructura del presente documento.

1.1. Motivación del proyecto

El diseño de arquitecturas de redes de neuronas convolucionales para resolver una tarea dada es complejo y requiere un proceso de experimentación casi aleatorio con largos entrenamientos a cada prueba. Esta complejidad en el diseño de arquitecturas neuronales surje por el enorme número de parámetros implicados en su construcción y por la falta de patrones de diseño que ayuden a convertir el problema en una búsqueda dirigida con garantías.

La solución clásica al problema antes mencionado es el uso de capital humano experto en la materia que dedique su tiempo y esfuerzo a dicho proceso de prueba y error intentando mejorar la búsqueda mediante su pericia. Esto hace que la construcción de redes de neuronas convolucionales de calidad sea un proceso lento, tedioso y caro para las empresas y equipos de investigación interesados.

Debido a que la solución clásica no parece resolver completamente el problema del coste del diseño, se han empleado métodos meta-heurísticos para encontrar procesos de búsqueda más eficientes. De entre estos métodos meta-heurísticos destacan los algoritmos genéticos, que al aplicarse al problema de la construcción de redes de neuronas conforman el campo de la neuroevolución. Este es el campo en el que se encuadrará este proyecto.

Pese a que la neuroevolución existe desde hace alrededor de tres décadas, durante muchos años los resultados eran deficientes. Esto se debía a que el algoritmo evolucionaba muy pocos parámetros y las redes producidas eran de un tamaño muy reducido, con solo una capa oculta y unas pocas neuronas. Es a partir de los avances tecnológicos en *hardware* de 2014 que este campo empieza a dar resultados más interesantes, específicamente con el trabajo de Koutník et al [2].

Ya que el desarrollo de este campo es reciente y los resultados actuales son prometedores, la neuroevolución debe ser probada en múltiples áreas y con diferentes metodologías. El presente trabajo se centrará en la creación de redes neuronales convolucionales secuenciales para la clasificación de imágenes, una vertiente donde hay pocos trabajos relacionados.

1.2. Objetivos

El objetivo principal de este trabajo es el de diseñar, implementar y aplicar un algoritmo genético capaz de construir de forma automática redes de neuronas convolucionales que, una vez entrenadas, puedan resolver correctamente una tarea de reconocimiento de imágenes.

Debido a la complejidad inherente al dominio se partirá del algoritmo genético propuesto por Baldominos et al. en su trabajo *Evolutionary Convolutional Neural Networks: an Application to Handwriting Recognition* [1], en el que se realiza una codificación de redes secuenciales con el objetivo de aplicarse sobre el *dataset* MNIST, formado por imágenes de números escritos a mano.

El diseño e implementación consistirán en la creación de una codificación de individuos adecuada y en la implementación de las funciones que convengan para la correcta progresión del algoritmo. Para que la codificación sea correcta debe permitir representar una gran variedad de redes neuronales convolucionales con el menor número de individuos incorrectos. Las funciones a implementar son aquellas que permiten que el algoritmo avance hacia soluciones casi óptimas, como pueden ser el cruce o la mutación. Se deberá tener en cuenta el tiempo empleado para la resolución del problema, por lo que se podrán valorar soluciones que hagan uso de paralelización o de distinto hardware que mejore el rendimiento con el mínimo coste añadido.

La aplicación del algoritmo consistirá en una extensa experimentación que permita observar si este método es válido para la resolución de la tarea y, además, si obtiene los resultados de forma robusta entre distintas ejecuciones y no por simple azar. Es esta parte del trabajo la que será más costosa, pues los tiempos de entrenamiento del algoritmo son notablemente extensos.

Por último, solo queda indicar explícitamente cuál es la tarea a resolver. La red neuronal convolucional fruto de este proceso de evolución debe ser capaz de clasificar correctamente las imágenes contenidas en el *dataset* CIFAR-10. Este *dataset*, que será explicado con más detenimiento en los próximos apartados, está compuesto por una serie de imágenes clasificadas en un total de diez categorías que engloban desde vehículos hasta animales. La red antes mencionada debe alcanzar valores de precisión en la clasificación competitivos teniendo en cuenta el actual estado del arte.

1.3. Estructura del documento

Este documento está formado por un total de cinco apartados principales, que pasarán a ser descritos a continuación:

- Introducción: En este apartado se expondrá la motivación del proyecto, los objetivos que se persiguen al desarrollarlo y la estructura del presente documento.

- Planteamiento teórico: En este apartado se realizará una explicación a nivel teórico sobre las técnicas empleadas en este trabajo y sobre el estado del arte de todas las cuestiones involucradas. Estas cuestiones son la neuroevolución y la precisión alcanzada para la clasificación sobre el *dataset* CIFAR-10.

- Planificación del proyecto: En este apartado se tratarán todas las cuestiones que deben ser resueltas antes de comenzar con la construcción del algoritmo. Esto incluye una planificación temporal para el proyecto, un estudio sobre el marco legal y el entorno socioeconómico y un análisis de requisitos que perfile la aplicación que se debe construir para la correcta realización del trabajo.

- Diseño e implementación: En este apartado se profundizará en el algoritmo desarrollado para este trabajo. Se realizará una profunda exposición sobre el diseño decidido (su codificación y las funciones involucradas en el algoritmo), además de tratar la forma en la que se ha abordado la programación de redes neuronales convolucionales. Por último, se presentarán alternativas al diseño expuesto.

- Experimentación: En este apartado se recogerá de forma ordenada los resultados de las distintas redes obtenidas y se comparará con el estado del arte de la cuestión. De esta forma se podrá asegurar su correcto desempeño y su robustez.

- Conclusiones: En este apartado se analizarán los resultados de los anteriores apartados, se observará si se han obtenido los resultados esperados y especificados en el subapartado de objetivos y se extraerán las conclusiones pertinentes. Además, se propondrán líneas de trabajo futuras para trabajos similares.

2. Planteamiento teórico

En este apartado, que se dividirá en dos partes, se explicarán los fundamentos teóricos en los que se basa el presente trabajo y el estado del arte, es decir, qué progreso se ha logrado en las cuestiones que nos ocupan. En este caso hay dos cuestiones relevantes: la neuroevolución, concretamente la neuroevolución aplicada a clasificación de imágenes, y la clasificación sobre el *dataset* CIFAR-10.

2.1. Fundamentos teóricos

Como se ha comentado anteriormente, el campo en el que se ubica este trabajo es el de la neuroevolución. Esto implica que es imprescindible el conocimiento de los dos campos que lo conforman, es decir, de los algoritmos evolutivos y el de las redes de neuronas. Cada uno de estos temas tendrá su propio subapartado debido a que el lector debe familiarizarse con ellos para la correcta interpretación de los siguientes apartados de esta memoria.

2.1.1. Algoritmos evolutivos

Existe un campo de la inteligencia artificial llamado computación evolutiva. Este campo, profundamente inspirado en los principios de la evolución biológica darwinista de las especies, es utilizado para resolver problemas de búsqueda en espacios extensos y no lineales donde es complicado obtener una heurística que dirija dicha búsqueda. Son métodos conocidos por encontrar soluciones a problemas complejos en un tiempo reducido que, además, no requieren más información sobre el dominio que una puntuación de cómo de buena es una solución para resolver el problema dado. Es por esto que se les considera métodos meta-heurísticos.

Los principios de la computación evolutiva se basan en replicar una abstracción del comportamiento general de una población de individuos vivos. Ya que el objetivo es resolver un problema dado cada individuo de la población representará una solución más o menos acertada para el mismo. Los comportamientos de los individuos, que pasarán a ser funciones que se aplican sobre la población, provocarán que ésta evolucione hasta soluciones mejores de manera progresiva.

Dentro del campo de la computación evolutiva, dependiendo de las funciones que se apliquen a la población, los tipos de individuos u otros factores, se pueden distinguir distintos paradigmas tales como la programación evolutiva, las estrategias evolutivas o los algoritmos genéticos. Son estos últimos los que se explicarán en detalle, pues el presente

trabajo se basa en ellos.

Para explicar con detalle los algoritmos genéticos se explicará por un lado el concepto de individuo y por otro las funciones bioinspiradas que se aplican sobre la población:

- Individuo: Un individuo de una población en un algoritmo evolutivo es una posible solución para el problema. Dentro del programa, dicho individuo estará representado por una cadena binaria que representarán su genotipo. Tal y como ocurre en la naturaleza, ese genotipo del individuo se manifestará en un determinado fenotipo, que se adaptará mejor o peor al medio. Dentro de la metáfora biológica, el fenotipo será la solución representada por la cadena de unos y ceros que es el genotipo y la adaptación al medio será la calidad de esa solución para el problema dado.

  Por ejemplo, simplificando el problema de este trabajo, el genotipo de un individuo sería 100110. Dicho genotipo provocaría el fenotipo de una red neuronal cualquiera, incluso una inservible para el problema dado. Por último, la adaptación al medio de ese individuo sería el valor de acierto de la red (fenotipo) tras entrenarla para resolver el problema.

  El mayor problema que se encuentra en la creación del individuo es la elección de una correcta codificación binaria del genotipo, que debe adaptarse lo máximo posible a los principios de los algoritmos genéticos. Según estos principios, la codificación debe cubrir todas las posibles soluciones, solo debe representar soluciones factibles, todas las soluciones deben estar representadas por la misma cantidad de codificaciones, debe ser fácil de decodificar el genotipo para convertirlo en fenotipo y pequeños cambios sobre los genotipos de los individuos deben corresponderse con pequeños cambios en los fenotipos de los mismos. Estos principios no siempre se pueden cumplir, pero una correcta codificación debe ceñirse a ellos lo máximo posible.

- Funciones: Como se comentado anteriormente, las funciones no son más que abstracciones de los fenómenos biológicos que impulsan la evolución biológica. En los algoritmos genéticos nos encontramos los siguientes operadores:

  - Inicialización: Este operador se encarga de crear la población inicial de individuos. Además de esto, es conveniente que la creación de los individuos no sea puramente aleatoria, pues distribuir uniformemente los individuos por el espacio de selecciones factibles puede facilitar en gran medida el desempeño del algoritmo.
  - Selección: Esta función se encarga de crear un subconjunto de la población, que será el útil para reproducirse y crear la futura generación del algoritmo. La creación de este subconjunto se hará mediante la selección de los individuos más aptos de la población. Para ello se pueden usar distintos métodos tales como ruletas o torneos.

Los torneos, el método de cruce utilizado en este trabajo, se basan en elegir un pequeño subconjunto de individuos de la población de manera aleatoria. De este pequeño subconjunto, que formará el torneo, pasará a la población apta para la reproducción el individuo que lo gane, o lo que es lo mismo, el individuo que mejor haya resuelto el problema. Se repetirán los torneos hasta que la población intermedia está completa. El tamaño de esta población depende de si cada reproducción producirá un individuo o dos (dos es lo más común).

- Cruce: Operador que se encarga de crear nuevos individuos a partir de un par de los antiguos. Para ello, realiza cortes en el genotipo de cada individuo y los vuelve a ensamblar de forma intercalada.

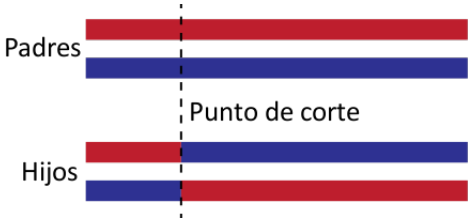


Fig. 2.1. Imagen resumen del cruce con un punto

Este operador permite realizar varios cortes dependiendo de los deseos del diseñador del algoritmo genético. El cruce actúa como explotación, buscando mejores soluciones en una zona concreta del espacio de búsqueda.

- Mutación: Este es un operador genético que realiza modificaciones aleatorias sobre el genotipo de los individuos con una probabilidad dada. Dado que la codificación es binaria aparecen dos tipos de mutación. La primera se basa en modificar un gen cambiando su valor y la segunda en intercambiar los valores de dos genes aleatorios. La mutación realiza una labor de exploración en el espacio de búsqueda, por lo que es una práctica habitual el reducir su probabilidad con el paso del tiempo.
- Elitismo: Este operador, que se encuentra entre los opcionales, se basa sencillamente en hacer pasar de una generación a la siguiente a un número determinado de individuos. Esos individuos serán los mejores de la anterior generación, lo que se podría considerar una élite. De esta forma se evita que los mejores individuos puedan desaparecer por las mutaciones o cruces, avanzando siempre la búsqueda hacia mejores soluciones.

El diseñador del algoritmo, además de crear la codificación y las funciones antes mencionada, debe solucionar los problemas intrínsecos a este tipo de algoritmos, muy similares a los de una población de seres vivos. El algoritmo genético, principalmente, debe evitar:

- Pérdida de la variabilidad genética: Al basarse el algoritmo en combinar los genotipos de la población ya existente, los individuos pueden acabar siendo muy similares

entre ellos. No es negativa mientras se produzca en los últimos momentos de la búsqueda.

- Convergencia prematura: Circunstancia que se da cuando se pierde la variabilidad genética en los primeros pasos del algoritmo. Esto se debe generalmente a que el algoritmo se ha encontrado un mínimo local y se ha estancado en él. Probablemente uno de los individuos iniciales lo ha encontrado y se ha convertido en un superindividuo, haciendo que el resto siempre salieran derrotados frente a él en los torneos y dirigiendo la búsqueda de una manera muy marcada. Puede ser útil emplear una tasa de mutación distinta o una mejor inicialización de la población.

- Epístasis: es uno de los principios deseables de la codificación, concretamente el que establece que pequeños cambios en el genótipo deben de producir pequeños cambios en el fenotipo. Esto hace al proceso menos heurístico y, por lo tanto, más aleatorio. Se soluciona buscando una mejor codificación.

- Deriva genética: Situación en la que los genes privilegiados se pierden por los operadores aleatorios que se utilizan en este tipo de algoritmo. Tiene una fácil solución si se introduce elitismo.

2.1.2. Redes de neuronas

Las redes de neuronas artificiales son un modelo de computación encuadrados en la inteligencia artificial subsimbólica y basados libremente en las neuronas humanas. Estos modelos se aplican a problemas de regresión, predicción, clasificación o clustering obteniendo resultados de bastante calidad. Como sistema de aprendizaje automático se encuadra dentro del aprendizaje supervisado, necesitando que los datos de aprendizaje contengan tanto los datos de entrada como la salida esperada.

Las redes de neuronas se basan, como no podía ser de otra manera, en la neurona computacional. La abstracción de la neurona no es más que una suma ponderada de los valores que recibe por las dentritas que se propaga por el axón después de sumarle un cierto umbral. Los pesos, uno por cada dentrita, y el umbral de dicha suma ponderada serán los valores a ajustar durante el periodo de aprendizaje de la neurona. Las redes neuronales más sencillas se limitan a esta abstracción, como pueden ser el perceptrón simple o el Adaline.

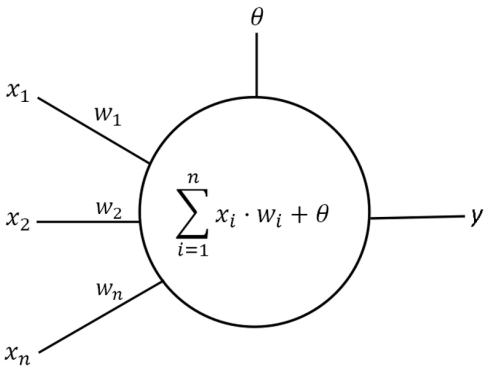


Fig. 2.2. Imagen de una neurona artificial

Cada tipo de red neuronal aprende de una determinada manera, pero es especialmente importante para aprendizajes de redes futuras el método que emplea Adaline. Adaline es una red neuronal especializada en regresiones, por lo que puede saber fácilmente cuánto error ha cometido en un resultado al compararlo con el valor esperado (recordemos que es un método supervisado). Si el lector se imagina una neurona de dos entradas, se puede establecer un mapa del error tridimensional en el que, dados dos pesos, la red comete un determinado error que se colocará en el eje Z. El método del aprendizaje del Adaline, que recibe el nombre de descenso del gradiente, se basa en calcular la pendiente del error en un punto determinado por los pesos y modificarlos para que el error baje. Este proceso se realizará de forma reiterativa hasta que la pendiente sea cero, lo que significa que se ha alcanzado un mínimo (local o absoluto) de la función de error.

El problema de las redes neuronales compuestas por una única neurona es que no son capaces de resolver gran parte de los problemas, como pueden ser aproximaciones a funciones no lineales o clasificar según la función XOR. Por ello es necesario crear redes más complejas, especialmente añadiendo más neuronas a la red en distintas capas y creando lo que se llamará perceptrón multicapa. A nivel de obtener los resultados de esta nueva red el proceso no se complica en exceso, sencillamente se realiza una concatenación de sumas ponderadas con el único añadido de aplicar una función no lineal a la salida de cada neurona (pues de no hacerlo todo el sistema sería equivalente a una única neurona lineal). Esta función que se aplica a cada neurona recibe el nombre de función de activación.

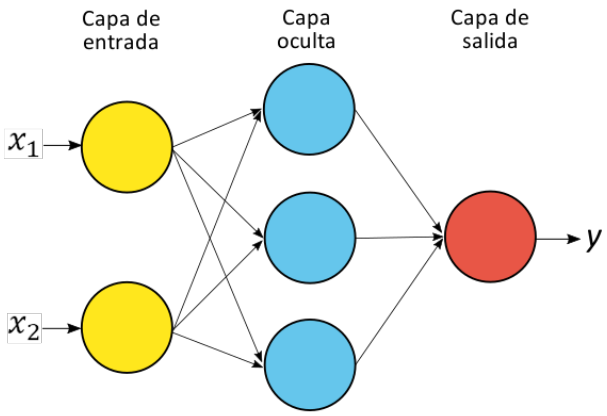


Fig. 2.3. Imagen de un perceptrón multicapa

En esta imagen podemos observar la capa de entrada, no mencionada anteriormente. Esta capa está formada por tantas neuronas como parámetros de entrada tenga la red, con la diferencia de que estas neuronas no realizan ninguna transformación sobre los datos. Su salida es igual que su entrada, o, lo que es lo mismo, son solamente útiles para representar visualmente las entradas.

Una vez obtenido el resultado de la red falta compararlo con la salida esperada y modificar los pesos para que la red aprenda. Este aprendizaje, notablemente más complejo que el del Adaline, recibe el nombre de algoritmo de retropropagación. Básicamente la red, una vez se ha calculado el error cometido, va recalculando los pesos y los umbrales de sus neuronas dependiendo de cuánto han aportado estas al resultado final. Esta situación es comparable a cómo se reparten las responsabilidades entre los distintos departamentos (neuronas) tras el fracaso de un producto en una empresa (la red). Este se ejecutará de forma iterativa a cada ejemplo de entrenamiento procesado.

A partir de este punto se desarrollan multitud de redes complejas con distintas funcionalidades específicas, pero de entre todas ellas son relevantes para este trabajo dos en particular: las redes neuronales convolucionales y las redes neuronales recurrentes.

Las redes neuronales convolucionales tienen como particularidad principal el hecho de que introducen capas convolucionales, que imitan el comportamiento de las neuronas de la corteza visual primaria del cerebro [3]. Una capa convolucional se define por su número de filtros o kernels (en este trabajo se usará indistintamente estos nombres) y el tamaño de dichos filtros. A nivel matemático lo que ocurre es que se realizan correlaciones cruzadas [3], que no convoluciones, entre la información contenida en el filtro aplicado y el segmento de la imagen que se está analizando.

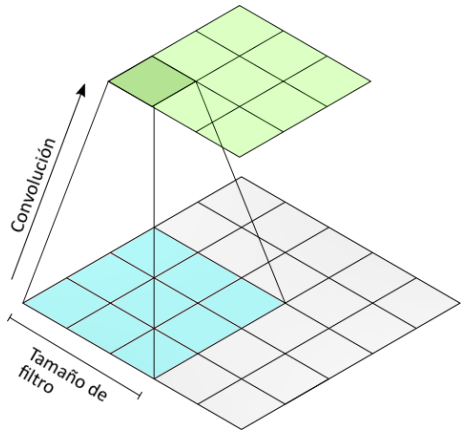


Fig. 2.4. Imagen de una convolución

Como se puede ver en la imagen anterior, el filtro (en azul) se superpone a la imagen (en gris) y realiza la convolución. El resultado (en verde oscuro) se almacena en su correspondiente espacio de la matriz de salida (en verde claro). Este proceso se realizará tantas veces como permita la imagen de entrada. Una vez se complete con un kernel se pasará al siguiente hasta hacerlo con todos los de la capa. Por lo tanto, la salida de la capa en total será un conjunto de salidas, una por cada filtro que tenga. Se puede ver en la imagen que

la salida tiene un tamaño inferior a la entrada, lo que no es en sí mismo un problema. En caso de que no se desee que sea así se rellenará la diferencia con ceros, lo que se conoce como *padding*.

Además de este tipo de capas se debe conocer la capa de *Max-pool*. En esta capa el filtro no realiza una convolución sino que sencillamente toma el mayor valor de los valores bajo el filtro. Sirve para reducir la resolución de la imagen y, con ello, permite aprender de forma más consistente al ser las imágenes más simples.

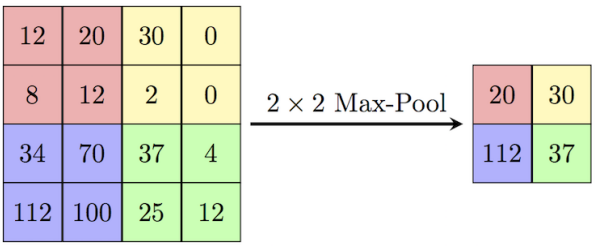


Fig. 2.5. Imagen resumen de una capa *Max-pool* [4]

El uso de capas convolucionales y de *Max-pool* solucionaría la necesidad de un preprocesado de los datos, pues serían las encargadas de encontrar los patrones ocultos dentro de los datos. Una vez atravesada la sección propiamente convolucional se pasaría a un perceptrón multicapa, a los que nos referiremos a partir de ahora como capas densas.

Las redes neuronales recurrentes más sencillas son muy similares a las redes neuronales densas (el perceptrón multicapa) pero con conexiones entre neuronas de la misma capa, conexiones hacia una capa anterior o conexiones de una neurona consigo misma. Este tipo de redes neuronales tienen la particularidad de tener una cierta memoria, lo que las hace ideales para tareas de predecir series temporales o de procesamiento de lenguaje natural. Hay varios tipos tales como *LSTM* (*Long short-term memory*), *GRU* (*Gated Recurrent Unit*) o redes recurrentes normales (*RNN*). Este tipo de neuronas se pueden usar como capas aisladas dentro de otras capas densas o en una red convolucional.

2.2. Estado del arte

En este apartado se realizará una breve explicación histórica de los temas más relevantes del trabajo con el objetivo de aclarar cómo ha sido su desarrollo a lo largo de los años y cuál es su estado actual.

El primer subapartado cubrirá el desarrollo histórico de la neuroevolución, los enfoques tomados en su desarrollo, sus limitaciones y el estado actual del campo. La mejor forma de abordar esto es mediante un relato cronológico que recorra estos temas en orden.

El segundo subapartado tendrá por tema el *dataset* sobre el que se realizará el trabajo, el CIFAR-10. Sobre este dataset se explicarán sus usos principales, su estructura y los mejores trabajos al respecto.

2.2.1. Neuroevolución

La neuroevolución se define, según la Enciclopedia del Aprendizaje Automático [5], como: "La neuroevolución es un método que modifica los pesos, topologías o conexiones de las redes neuronales para aprender una tarea específica. Se emplea la computación evolutiva para buscar los parámetros de la red que maximicen la función de *fitness* que mide el desempeño en una tarea dada".

El concepto de neuroevolución nació en la década de 1980, cuando una serie de investigadores tales como Miihlenbein y Kindermann [6] comenzaron a emplear algoritmos genéticos para evolucionar los pesos de las redes neuronales artificiales. Esta aproximación era muy útil en los casos en los que el algoritmo de retropropagación no era una buena opción.

Aunque sí hay trabajos tempranos sobre el uso de estos algoritmos para determinar la topología de las redes, como el de Miller et al. [7] en el año 1989, el campo empieza a enfocarse en esta dirección a mediados de la siguiente década. Con el paso del tiempo empezó a usarse en los nuevos tipos de redes de neuronas que se estaban creando, continuando el avance del campo hasta el presente. Sin embargo, la necesidad de usar estos métodos era discutible, ya que por compleja que fuera la red el número de parámetros no llegaba a ser excesivo.

La aparición de las redes neuronales profundas, entre las que se incluyen las convolucionales, aumentaron en gran medida el número de parámetros implicados en la creación de la red y en su aprendizaje. El proceso de búsqueda aleatoria que tendría que realizar el experto sería enorme para cubrir todos los parámetros anteriores más algunos nuevos como los tamaños de filtros, las conexiones recurrentes en varias capas, las funciones de activación o el tamaño de las capas de *Max-Pooling*.

El problema asociado a este tipo de redes es la complejidad de su proceso de aprendizaje, pues el entrenamiento de una red ejecutada sobre una CPU (aun siendo de gama alta) es muy largo. Si tenemos en cuenta que este tipo de algoritmos debe entrenar a miles de redes neuronales convolucionales se llega a la conclusión de que no es viable alcanzar la meta por este camino. Este es el estado en el que se ha encontrado el campo respecto a las redes de neuronas convolucionales desde su creación.

En el año 2012, Ciresan et al.[8] consiguieron refinar e implementar las redes convolucionales para ser ejecutadas en unidades de procesamiento gráfico (a partir de ahora GPU). Este avance logró acelerar notablemente el proceso de aprendizaje debido a la afinidad de estos dispositivos de *hardware* con el cálculo de matrices. Sin embargo, pese a acelerar el proceso, el rendimiento quedaba lejos de poder entrenar a un gran número de redes de este tipo en poco tiempo.

No fue hasta el año 2014 que el hardware permitió los rendimientos necesarios como para el desarrollo de esta vertiente de la neuroevolución, que comenzó con el trabajo de Koutník et al. [2], momento a partir del cual se exploró en profundidad el campo. Aún

así, el escaso tiempo que ha transcurrido desde entonces hasta el presente año hace que la neuroevolución con redes convolucionales sea un campo casi inexplorado. A nivel de topologías se pueden distinguir dos aproximaciones para el desarrollo. La primera busca crear redes secuenciales, más sencillas de representar y menos potentes. La segunda es evolucionar la estructura de la red como si fuera un grafo, lo que es muy complejo debido a las limitaciones de las cadenas binarias para realizar correctas codificaciones.

De entre todos estos trabajos que han aparecido en estos años son especialmente relevantes para este documento dos en concreto:

- Baldominos et al. [1] desarrollaron en 2019 un algoritmo genético aplicado sobre el MNIST, un *dataset* formado por dígitos manuscritos. Sus resultados establecieron un nuevo estado del arte para dicho conjunto de datos, sin ningún tipo de manipulación previa sobre los mismos. Este trabajo es relevante para este documento debido a que este trabajo es su continuación, aplicandolo tras realizar algunos cambios sobre un dominio distinto y más complejo como es el CIFAR-10. El trabajo proponía una codificación binaria reflejada (código Gray) para representar redes neuronales convolucionales secuenciales, evolucionando la topología y un gran número de parámetros de la red.
- Un equipo de Google Brain publicó en 2019 un algoritmo evolutivo sobre el mismo conjunto de datos que se emplea en este trabajo, el CIFAR-10. Los resultados del artículo no son relevantes para nuestro estudio, pues los investigadores realizaron *data augmentation* sobre las entradas para mejorar el entrenamiento. El artículo de Esteban et al.[9] es relevante por implementar una codificación capaz de explorar el espacio de búsqueda NASNet [10], especialmente diseñado para redes neuronales convolucionales clasificadoras de imágenes.

2.2.2. El *dataset* CIFAR-10

El conjunto de datos CIFAR-10, creado por el *Canadian Institute for Advanced Research* (CIFAR), es uno de los más usados para el entrenamiento de modelos de visión artificial. Está compuesto por un conjunto de 60000 imágenes, con un tamaño de 32x32 píxeles cada una. Al ser las imágenes en color, cada imagen tiene, a su vez, los tres canales necesacios (rojo, verde y azul). A cada imagen le corresponde una etiqueta que indica a qué clase pertenece, o, lo que es lo mismo, qué muestra la imagen.

Las imágenes se agrupan en diez clases: aviones, coches, aves, gatos, ciervos, perros, ranas, caballos, barcos y camiones. A todas las clases les corresponde un igual número de ejemplos, así que hay 6000 imágenes para cada una.

Algunos ejemplos son:

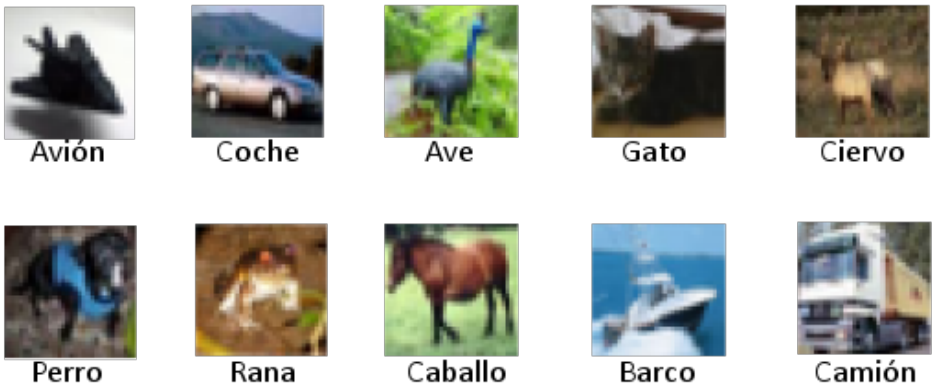


Fig. 2.6. Imágenes de ejemplo del *dataset* CIFAR-10

Por defecto, CIFAR-10 agrupa las imágenes y etiquetas en dos grupos diferenciados. El primero, formado por 50000 imágenes con sus respectivas etiquetas, supone el conjunto de entrenamiento. Este conjunto servirá para que el modelo aprenda lo necesario como para ser válido. El segundo, formado por 10000 imágenes con sus etiquetas, será el de *test*. Estos datos, a los que la red no accederá durante su entrenamiento, servirán para medir correctamente el desempeño de la red y si el conocimiento adquirido es generalizable a ejemplos no vistos.

El CIFAR-10 es un *dataset* de elevada complejidad. Esto se debe a múltiples factores tales como:

- Imágenes de baja resolución: Como ya se ha dicho anteriormente, cada imagen está formada por 32x32 píxeles. Esto hace que sean imágenes de muy baja resolución y que algunas sean difíciles de clasificar incluso para un ojo humano. Como es evidente, si es dificil de reconocer para una persona será terriblemente complicado para una máquina que intenta imitar su habilidad. Algunos ejemplo de este fenómeno serían:

Fig. 2.7. Imágenes extrañas del *dataset* CIFAR-10

  En la primera imagen, clasificada por los autores como un gato, se puede ver al animal con la mitad del cuerpo tapada y la mitad restante con una resolución tan baja que es dificil distinguir claramente su anatomía. En la segunda, la resolución hace

que sea complicado determinar los píxeles que pertenecen al ciervo, que se está camuflando en la naturaleza. Estas fotografías han sido extraídas de los primeros cien ejemplos de entrenamiento, por lo que se puede suponer que no son una excepción.

- Clases con imágenes poco similares: Algunas de las clases que agrupan las imágenes son excesivamente amplias. Por ejemplo, la clase de aves engloba tanto imágenes de pequeños pájaros como gorriones hasta imágenes de casuarios. Esto hace que miembros de la misma clase sean muy diferentes tanto en color como en forma y tamaño. Pero el problema no se limita a eso solamente. En otras clases, el cambio de perspectiva a la hora de tomar la imagen o la proximidad a algunos rasgos determinados (la cara de un gato, por ejemplo) también lo dificulta.

- Pocas imágenes: Teniendo en cuenta los dos puntos anteriores, tener un subconjunto destinado al entrenamiento de 50000 imágenes no parece ser suficiente. Esta cantidad limita en gran medida cuánto puede aprender la red sin realizar un sobreajuste sobre los datos de entrenamiento y, debido a lo dicho en los puntos anteriores, se necesita un largo entrenamiento para aprender todas las características desde diferentes ángulos e inclinaciones de cada clase de los datos.

Son los motivos anteriores los que provocan que un gran número de investigadores decidan realizar un preprocesamiento sobre los datos de entrada para obtener un número mayor de ellos. A este procesamiento se le conoce como *data augmentation* y consiste en realizar una serie de transformaciones sobre los datos de entrada (giros, simetrías, zooms...) y añadir los resultados de esas transformaciones a los datos de entrenamiento.

El presente trabajo no emplea ningún tipo de *data augmentation*, pues el objetivo del mismo no es encontrar un nuevo estado del arte para el conjunto de datos sino comprobar la eficiacia de la neuroevolución a la hora de obtener un modelo competitivo en el dominio solo introduciendo los datos sin tratar, sin ninguna intervención por parte de un experto. Este hecho hace que los artículos que supondrán el estado del arte para este trabajo deben cumplir la misma condición, no deben emplear *data augmentation*.

Algunos de los trabajos más destacables realizados sobre este conjunto de datos son:

Método	Nº Capas	C10
Network in Network [11]	-	10.41
All-CNN [12]	17	9.08
Deeply Supervised Net [13]	-	9.69
FractalNet [14]	21	10.18
ResNet [15]	110	13.63
ResNet (pre-activation) [16]	1001	10.56
DenseNet [17]	190	5.19

Tabla 2.1. Tabla resumen del estado del arte

3. Planificación del proyecto

En este apartado se estudiarán todos los aspectos previos a la realización del proyecto. Esto incluye realizar una planificación temporal que cubra toda la ventana de fechas disponible repartiendo correctamente los turnos para cada tarea, un estudio sobre el marco legal de las herramientas que se piensan usar, un análisis del entorno socioeconómico del proyecto y una especificación de requisitos del programa a desarrollar.

3.1. Planificación

Este proyecto tuvo su inicio en el mes de febrero de 2019, prolongándose su desarrollo hasta inicios de junio de 2016. Las fases a completar eran evidentes desde un principio, pero los tiempos dedicados a cada una debieron ser estudiados específicamente para evitar errores en el futuro.

Con el objetivo de que el apartado sea claro se comenzará con una exposición de la planificación temporal elegida inicialmente y se terminará con la planificación final que se siguió realmente por distintos contratiempos en el proyecto.

A la hora de realizar la planificación inicial se consideraron tres fases principales para el proyecto:

- Planificación e investigación: En esta fase se planificarían los siguientes pasos y se investigaría sobre el dominio del proyecto y sobre la tecnología relacionada con el mismo. Una vez estuvieran claras las opciones se eligirían que mejor se adaptasen al trabajo y se comprobaría la compatibilidad entre ellas. El tiempo estimado para esta fase sería de un mes.

- Diseño e implementación: Una vez resueltas las incógnitas de la fase anterior se pasaría a diseñar el software que se debe construir. Una vez diseñado se pasaría a la implementación usando las tecnologías oportunas. Esta fase tendría una longitud aproximada de mes y medio.

- Experimentación y documentación del proyecto: Por último se pasaría a realizar una exhaustiva experimentación sobre el proyecto y, de forma paralela, se escribiría la memoria sobre el trabajo realizado. Esta fase ocuparía lo restante del tiempo pensado, los dos meses que faltan hasta la entrega acordada.

Esta planificación queda representada en el siguiente diagrama Gantt:

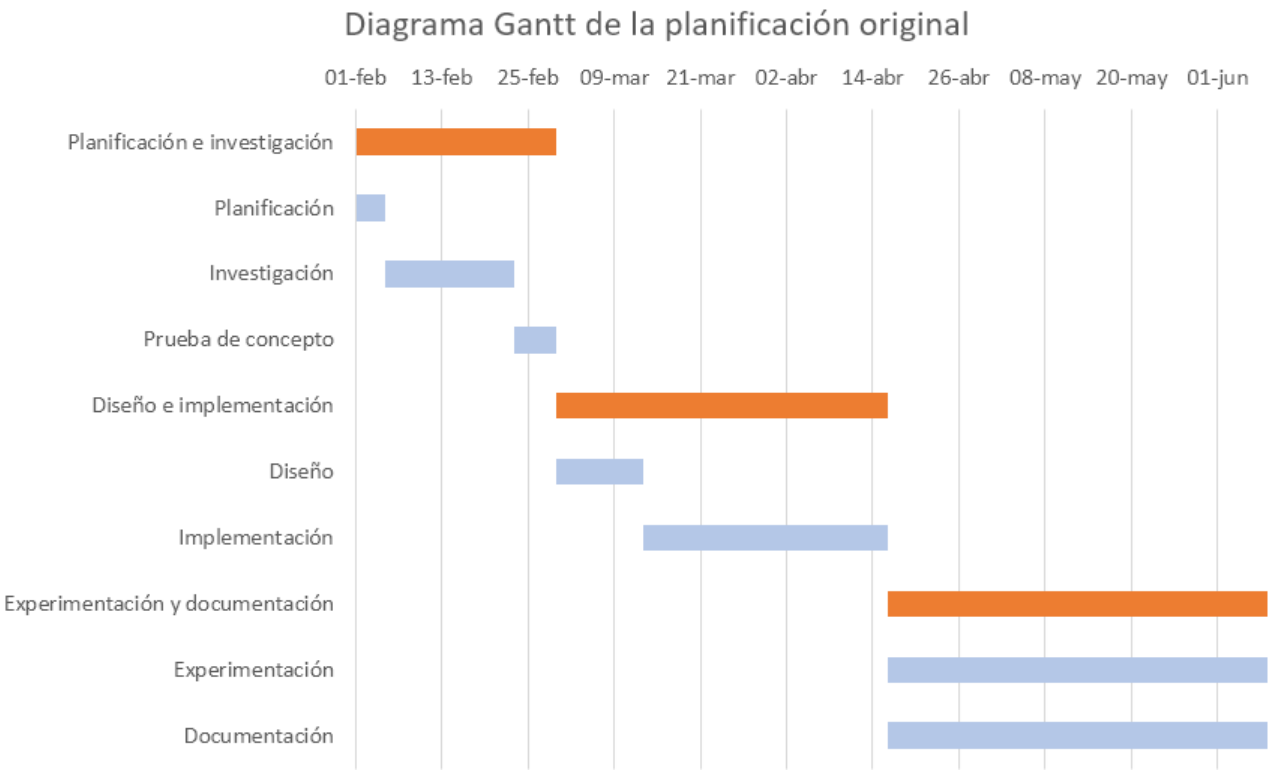


Fig. 3.1. Diagrama Gantt de la planificación original

En la figura se pueden observar los tiempos asignados para cada fase del proyecto (en naranja) y cuánto se dedica dentro de cada fase para cada una de las acciones principales (en azul).

Debido a una serie de contratiempos hubo que realizar una serie de cambios tanto de los tiempos asignados en la planificación como de las competencias de cada fase.

La primera fase comenzó en la fecha prevista pero terminó a inicios de abril debido a que el primer conjunto de datos elegido fue el MIMIC, un *dataset* de la universidad de Massachusetts que contiene información sobre pacientes médicos y enfermedades relacionadas. El objetivo original era relacionar las medidas de electrocardiogramas y tensión sistólica con episodios de inestabilidad hemodinámica. Hubo que cambiar el *dataset* objetivo debido a la falta de correlación entre los datos de entrada y los de salida. Esto provocó un enorme retraso, pues supuso una fase de "Prueba de concepto"mucho más larga, y este retraso se propagó al resto de fases.

La segunda fase comenzó una vez se habían resuelto todos los problemas anteriores y terminó poco después, a mediados de abril. Para solucionar el retraso se agilizó lo máximo posible esta fase, realizando una implementación no definitiva, que solamente sirvió como prototipo inicial sobre la que realizar posteriores mejoras en función de los resultados obtenidos en las pruebas. No se considera de esta fase el resto del desarrollo del programa.

Por último, la última fase se extendió aproximadamente por las fechas previstas. El

único cambio es que en esta fase, al ser la implementación solo un prototipo, hubo que realizar correcciones sobre el código desarrollado. Además, durante este período se tuvo acceso a otro ordenador que contaba con dos GPU, lo que facilitó el proceso de pruebas a costa de un pequeño esfuerzo a la hora de paralelizar el código desarrollado. Debido a los largos tiempos de espera entre pruebas, la realización de modificaciones sobre el código no supuso un sobrecoste en lo planeado.

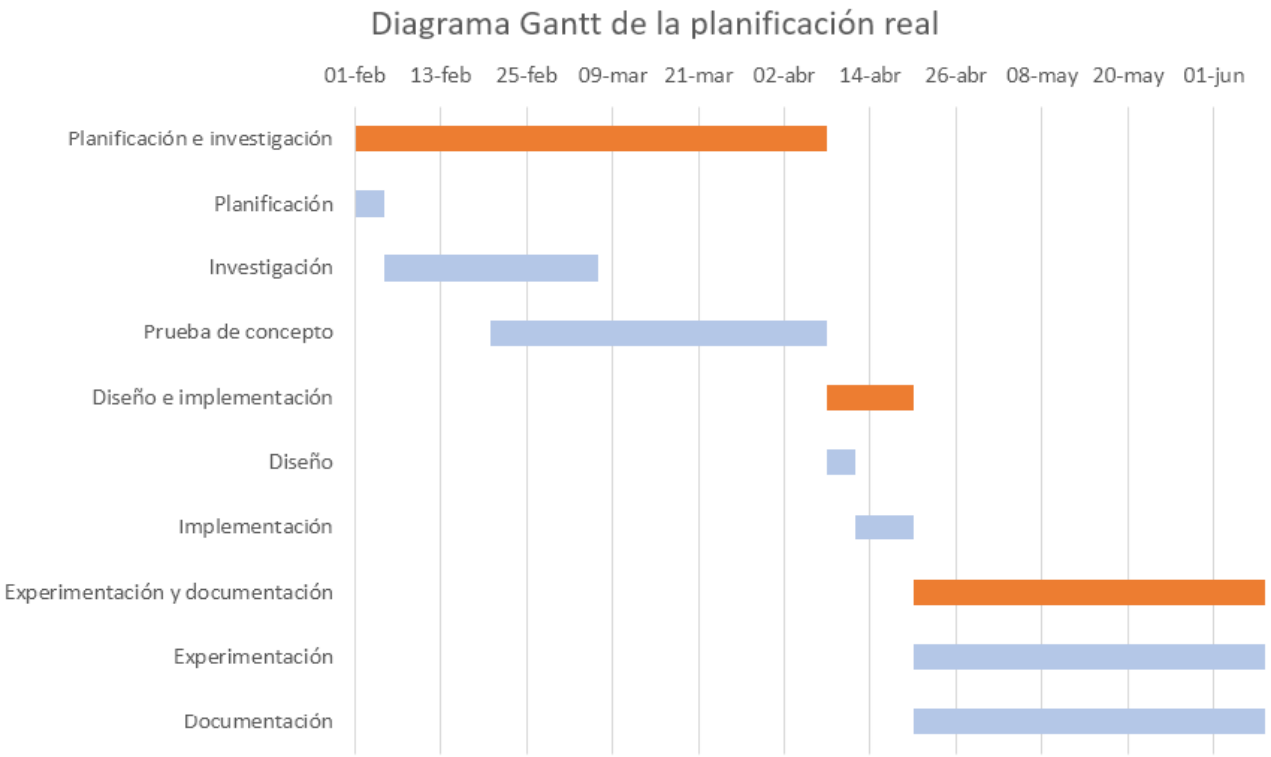


Fig. 3.2. Diagrama Gantt de la planificación real

3.2. Marco legal

En este apartado se tratarán las implicaciones legales en el uso de las tecnologías elegidas para la realización del presente proyecto. Pese a que se mencionarán estas tecnologías no se dedicará tiempo en este apartado a describir sus funcionalidades o a discutir el porqué de su elección.

Las licencias bajo las que se opera en las tecnologías empleadas son:

- Tensorflow: Licencia Apache 2.0[18], según la cuál se puede hacer uso de esta tecnología mientras se mantenga un aviso que indique que se ha usado código bajo esta licencia en el producto.
- Keras: Licencia MIT[19], concretamente X11. Esta licencia permite el uso de la tecnología mientras se incluya en las copias del *software* la nota de derechos reservados.

- Python: Licencia PSFL[20], una licencia de *software* libre permisiva que solo obliga a mantener el *copyright* sobre la versión de Python empleada.
- Jupyter Notebook: Licencia BSD[21], que comparte las mismas condiciones que la licencia MIT.

Al ser las cuatro licencias anteriores de *software* libres permisivas, los derechos que otorgan al usuario son iguales. Estos derechos incluyen la libertad de usar el *software* para cualquier propósito, distribuirlo, modificarlo, y distribuir versiones modificadas del mismo, estando el producto libre de regalías.

Por último, es necesario mencionar que el proyecto se ha realizado sobre equipos con sistema operativo Ubuntu, bajo licencia GLP 3.0[22]. Sin embargo, dicha licencia no puede aplicarse al presente proyecto debido a que no es un producto derivado del sistema operativo, pues no emplea código del mismo.

Emplear productos tecnológicos bajo este tipo de licencias permite crear gratuitamente software avanzado mientras se cumplan las restricciones antes mencionadas a la hora de distribuirlo, por lo que son condiciones muy favorables para el desarrollador.

3.3. Entorno socioeconómico

A la hora de desarrollar la aplicación deseada es necesario tener en cuenta los recursos necesarios, tanto físicos como laborales, para obtener el resultado deseado.

Como ya se ha indicado en el anterior apartado, las licencias bajo las que operan las tecnologías *software* sobre las que se cimenta este proyecto permiten su utilización sin necesidad de pagar ninguna clase de regalías, lo que facilita el proceso de desarrollo.

Sin embargo, al ser el *software* a desarrollar altamente costoso en términos computacionales hace necesario el uso de potentes equipos informáticos para el procesamiento. Los recursos *hardware* empleados son los componentes principales (CPU y GPU) de los dos ordenadores usados para esta práctica, por lo que se describirán por separado:

- Equipo Titán: Este equipo tiene por CPU un Intel Core i7-3930K, valorado en 469€, y una GPU Nvidia GTX Titan Xp, valorada en 1299€. Estos componentes suponen un precio total de 1768€.
- Equipo Lava: Este equipo tiene por CPU un Intel Core i7-6700, valorado en 259€, y dos GPUs Nvidia GTX 1080, valoradas en 795€(precio de salida del *hardware*, modelo ya descatalogado). Estos componentes supondrían un precio total para el equipo de 1849€.

Si bien es cierto que los componentes del equipo Titán son superiores en potencia, el hecho de que el equipo Lava tenga dos GPU permite la parelización a la hora de evaluar

las redes neuronales, lo que reduce el tiempo de ejecución a la mitad. Una configuración más eficiente para este trabajo sería un procesador de una gama más baja y el mayor número de tarjetas gráficas posibles, ya que el procesamiento que necesita el algoritmo es mínimo frente al cálculo matricial necesario, que es el principal cuello de botella del programa.

4. Diseño e implementación

En este apartado se describirá el diseño del algoritmo utilizado en este proyecto y la forma en la que se ha implementado. Por otra parte, se tratarán los mismos temas con respecto al creador de redes neuronales convolucionales y, con el objetivo de que el proceso sea lo más claro posible, se describirán las distintas alternativas al diseño elegido.

4.1. Análisis

En este apartado se diferenciará entre los tipos de requisitos que se van a considerar en este proyecto y, tras establecer la tabla con la que se definirán, se pasará a enumerarlos.

Los requisitos de usuario que se considerarán en este trabajo son los dos siguientes:

- Requisitos de capacidad o funcionales: Clase de requisitos que representan las necesidades de los usuarios para resolver un problema o lograr un objetivo. Indican la funcionalidad deseada.
- Requisitos de restricción o no funcionales: Grupo de requisitos que actúan como restricciones impuestas por los usuarios sobre cómo debe resolverse el problema encargado o cómo debe alcanzarse un cierto objetivo. Este grupo se puede subdividir en:
    - Requisitos de rendimiento: Se refieren a aspectos tales como tiempos de respuesta, capacidad de disco o necesidades de memoria.
    - Requisitos de implantación: Describen las necesidades relacionadas con el contexto (hardware, software, normas, ubicación...) en los que debe funcionar el sistema.

Para definir cada requisito se empleará la siguiente tabla:

Identificador			
Prioridad	————-	Fuente	————-
Necesidad		————-	
Claridad	————-	Verificabilidad	————-
Estabilidad		————-	
Descripción		————-	

Tabla 4.1. Tabla de ejemplo para requisitos

Cada uno de los apartados de la tabla se refiere a:

- Identificador: Identifica unívocamente a un requisito de usuario. Está formado por las siglas RNE (requisito de neuroevolución), las letras C o Rx (capacidad o restricción) y un número que irá aumentando a cada requisito, todo separado por guiones. En el caso de ser de restricción, la x podrá tomar el valor de R o I (rendimiento o implantación). Un ejemplo sería RNE-RR-2, que designa a un requisito de restricción de este proyecto, más concretamente uno de rendimiento.
- Prioridad: Medida útil para la planificación del proyecto, pues indica la urgencia con la que debe implementarse dicho requisito. Toma el valor de *Alta*, *Media* u *Baja*.
- Fuente: Indica el origen de cada requisito. Puede ser una normativa externa o el desarrollador/usuario.
- Necesidad: Indica si un requisito es o no negociable debido a su importancia y cómo afectaría su falta al proceso. Puede tomar el valor de *Esencial*, *Deseable* u *Opcional*.
- Claridad: Determina si el requisito es ambiguo o no. Representa dicha claridad mediante los valores *Alta*, *Media* u *Baja*.
- Verificabilidad: Indica hasta qué punto se ha verificado su incorporación al diseño. Toma el valor de *Alta*, *Media* u *Baja*.
- Estabilidad: Determina si un requisito se va a mantener estable en el tiempo. Su contenido es una medida aproximada del tiempo en el que se espera que no sea modificado. Toma el valor de *Alta*, *Media* u *Baja*.
- Descripción: Texto en el que se explica el propósito del requisito de forma breve, clara y completa.

4.1.1. Requisitos de capacidad

A continuación se describirán los requisitos de capacidad del *software* a desarrollar:

REV-C-1			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Media	Verificabilidad	Media
Estabilidad		Alta	
Descripción		El programa debe mejorar progresivamente el desempeño de las redes creadas hasta converger en un rango de acierto estable.	

REV-C-2			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Media	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe mostrar a cada generación el estado de la población.	

REV-C-3			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe crear redes neuronales a partir de las codificaciones de los invididuos.	

REV-C-4			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe mostrar si ha habido algún error durante el proceso de creación de las redes neuronales.	

REV-C-5			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe operar sobre el *dataset* de imágenes CIFAR-10.	

4.1.2. Requisitos de restricción

Los requisitos de rendimiento son:

REV-RR-6			
Prioridad	Baja	Fuente	Desarrollador
Necesidad		Deseable	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Media	
Descripción		La duración máxima de evaluación una generación deben ser las diez horas.	

REV-RR-7			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe disponer de forma exclusiva de la memoria de la GPU.	

REV-RR-8			
Prioridad	Media	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe mantenerse en funcionamiento de forma ininterrumpida durante semanas.	

Los requisitos de implantación son:

REV-RI-9			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe incluir las fases fundamentales de un algoritmo genético: inicialización, selección, cruce y mutación.	

REV-RI-10			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El equipo informático utilizado necesita una o varias GPU de alta gama.	

REV-RI-11			
Prioridad	Baja	Fuente	Desarrollador
Necesidad		Deseable	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		El programa debe estar paralelizado para emplear todas las GPU disponibles en el equipo.	

REV-RI-12			
Prioridad	Media	Fuente	Desarrollador
Necesidad		Deseable	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Media	
Descripción		Se repetirá la inicialización de los individuos hasta que todos sean válidos.	

REV-RI-13			
Prioridad	Baja	Fuente	Desarrollador
Necesidad		Deseable	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Media	
Descripción		La codificación debe provocar que, en cada generación, más del 70 % de los individuos sean válidos.	

REV-RI-14			
Prioridad	Baja	Fuente	Desarrollador
Necesidad		Deseable	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Media	
Descripción		Se implementará un sistema de nichos para evitar la convergencia prematura del sistema.	

REV-RI-15			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Deseable	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		La codificación de los individuos se realizará en código Gray.	

REV-RI-16			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Baja	
Descripción		El algoritmo evolucionará optimizando la topología de la red (número, características y tipos de capas).	

REV-RI-17			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Baja	
Descripción		El algoritmo evolucionará optimizando los valores de aprendizaje (razón de aprendizaje, tamaño de lote...).	

REV-RI-18			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción		La implementación se realizará en Python.	

REV-RI-19			
Prioridad	Alta	Fuente	Desarrollador
Necesidad		Esencial	
Claridad	Alta	Verificabilidad	Alta
Estabilidad		Alta	
Descripción:		La implementación de las redes se realizará utilizando las librerías Tensorflow y Keras.	

4.2. Alternativas de diseño

En esta sección se discutirán las distintas opciones de diseño posibles para el proyecto, justificando las elecciones tomadas para la implementación en base a las ventajas y desventajas de cada opción.

En los siguientes subapartados se tratarán por separado las alternativas en cuanto a lenguajes de programación y librerías de redes neuronales.

4.2.1. Lenguaje de programación

Debido a los conocimientos del programador se plantearon las siguientes alternativas para el lenguaje a elegir:

- Python: Un lenguaje interpretado con una sintaxis sencilla. Soporta programación orientada a objetos, imperativa y funcional. Destaca por ser sencillo de usar y por su enorme cantidad de librerías que facilitan trabajar con él, especialmente en relación con la inteligencia artificial. Por esta razón Python se ha convertido en el lenguaje más utilizado no solo para aprendizaje automático, sino en general [23]. En inconveniente principal de este lenguaje es su lentitud en la ejecución.

- JavaScript: Un lenguaje de programación interpretado, dinámico y orientado a objetos. Se usa principalmente para desarrollos web, lo que ha creado una enorme base de usuarios. Esa misma base de usuarios ha provocado que algunas de las más importantes bibliotecas de redes neuronales como Tensorflow hayan creado *ports*

para este lenguaje. Sus inconvenientes principales son el desconocimiento por parte del desarrollador, su velocidad y que el propósito principal del algoritmo no está alineado con la tarea a realizar.

- C: Un lenguaje de propósito general, conocido por la eficiencia del código que produce y por el enorme control que da al desarrollador tanto a alto como a bajo nivel. Los problemas de esta opción son, principalmente, los problemas de tipos a la hora de tratar con los datos y la poca oferta de librerías relacionadas con el aprendizaje automático.

- Java: Un lenguaje de programación de propósito general orientado a objetos. Es uno de los lenguajes más usados en la actualidad y cuenta con una buena cantidad de librerías dedicadas al aprendizaje automático. Sin embargo, es un lenguaje lento y poco intuitivo. Además, pese a tener librerías sobre el tema de este trabajo (redes neuronales) su comunidad está lejos de la de algunos de sus competidores.

Para elegir el lenguaje en el que se desarrollará el proyecto se deben tener en cuenta las necesidades de éste, los conocimientos del programador y la facilidad para obtener a información necesaria en caso de duda.

Son las razones antes descritas las que determinan la elección de Python como lenguaje de programación, ya que tiene una gran variedad de librerías sobre el tema (y muy actualizadas), una gran comunidad de *machine learning* cuyas dudas están recogidas en foros de Internet y es bien conocido por el desarrollador. Frente a su inconveniente referido a la velocidad de ejecución no es especialmente relevante para el proyecto, ya que el tiempo dedicado a la ejecución del algoritmo propiamente dicho es mínimo comparado con el dedicado a la evaluación de las redes neuronales creadas, que es independiente al lenguaje elegido.

4.2.2. Librerías de redes neuronales

Una vez elegido en el apartado anterior el lenguaje de programación que se utilizará para la implementación de este proyecto se debe estudiar el conjunto de alternativas posibles dentro de la variedad de librerías de redes neuronales.

Python ha adquirido en los últimos años una enorme popularidad, especialmente entre los programadores dedicados al *Machine Learning* y al análisis de datos. Esto ha provocado que haya una gran oferta de librerías gratuitas y de calidad para esta tarea.

Algunas de estas librerías tienen dependencias entre ellas, ocupándose unas de los cálculos subyacentes y otras de facilitar al desarrollador la creación de los modelos. Las librerías más relevantes son:

- Tensorflow: Librería encargada de la computación numérica creada por Google. Es muy flexible gracias a su forma de representar las redes neuronales como grafos y

recibe actualizaciones constantes.

- Theano: Librería para la manipulación numérica de matrices. Actualmente obsoleta por haber dejado de recibir actualizaciones.
- Keras: Librería que permite la implementación a alto nivel de las redes neuronales, facilitando la tarea respecto a lo que sería solo la librería de computación numérica. Es una librería sencilla, está actualizada y soporta Tensorflow y Theano. Recibe soporte de Google pero es poco flexible.
- Lasagne: Librería que sirve como interfaz para desarrollar redes neuronales en Theano, pero su comunidad se ha reducido notablemente por la falta de actualizaciones sobre esta librería.
- Pytorch: Librería desarrollada por Facebook. En este caso sirve tanto de librería de computación numérica como de interfaz que facilita la creación de los modelos. Su lanzamiento es reciente, por lo que está muy actualizada.

La tarea a desarrollar consiste en crear redes neuronales secuenciales sencillas de forma automática, por lo que no se requiere una enorme flexibilidad en la librería elegida. Lo determinante en la elección es que esté actualizada y que tenga una amplia comunidad.

Concretamente, se busca que la librería (o librerías) reciba actualizaciones para asegurar que la aplicación tenga una larga vida útil y que su rendimiento se pueda mejorar con el tiempo de la mano con los avances en el campo.

Respecto a la comunidad de usuarios, su número suele provocar que la actividad en foros de ayuda o errores sea mayor, lo que facilita su aprendizaje. Además, al querer crear una aplicación que genere las redes de forma automática se encontrarán los errores propios de distintos tipos de arquitectura, por lo que encontrar documentación sobre posibles soluciones es muy favorable para este proyecto.

Estos motivos llevan a descartar Theano, Tensorflow (se descarta su uso en solitario) y Lasagne, pero queda elegir entre Keras o Pytorch. Debido a la sencillez que proporciona Keras y a que permite realizar todas las funciones pensadas para este proyecto es la librería elegida para el trabajo. Ya que Theano está obsoleto, Keras operará sobre Tensorflow.

4.3. Diseño del algoritmo

En este subapartado se describirá el diseño elegido para el algoritmo genético. Para ello el documento se centrará en describir el funcionamiento a nivel global y los parámetros que optimizará el proceso de evolución.

El funcionamiento general del programa sigue las bases de los algoritmos evolutivos:

1. Inicialización de la población: Se crea la primera población de individuos de manera aleatoria. La inicialización influye en gran medida en la posterior evolución, por lo que se puede mejorar forzando a que la población inicial esté bien repartida por el espacio de búsqueda. Se debe elegir una determinada población para el algoritmo.

2. Evaluación: Consiste en aplicar la función de *fitness* a los individuos, de manera que tengan todos una puntuación asociada relacionada con su rendimiento para completar la tarea. En este caso se hará un programa encargado únicamente de dicha evaluación.

3. Elitismo: Se guarda el genotipo de los n individuos mejor valorados de la población, para que la aleatoriedad propia del algoritmo no los pueda eliminar y las futuras generaciones los tengan con seguridad entre su variedad genética. El desarrollador debe elegir el número de individuos que formarán la élite.

4. Selección: Se empleará el método de torneos, en los que se compara el desempeño de varios individuos y se añade el mejor a la población apta para la reproducción. Se debe elegir un determinado tamaño de torneo, que indicará cuántos individuos se baten en cada ronda.

5. Cruce: Se eligen dos individuos aleatorios de la población apta para la reproducción y se cruzan sus genotipos por uno o más puntos (número que debe determinar también el desarrollador). Este operador se considera de explotación.

6. Mutación: Se modifican genes de cada individuo con una cierta probabilidad. Dicha posibilidad debe ser ajustada por el desarrollador, pues una posibilidad demasiado alta hará imposible converger al algoritmo y una muy baja hará irrelevante a este operador. Es un operador de exploración y, después de ser completado, se unirá la nueva población con la antigua élite y el algoritmo volverá al paso 2.

Es decir, a partir de los puntos anteriores se concluye que el desarrollador debe ajustar del algoritmo genético el tamaño de la población, la función de evaluación, la tasa de elitismo, el tamaño del torneo de selección, el número de puntos del cruce y la probabilidad que tiene de mutar un gen. Además de esto debe indicar cuántas generaciones se ejecutará el algoritmo o si se detendrá por otros motivos.

El diseño del algoritmo quedaría resumido en la siguiente imagen:

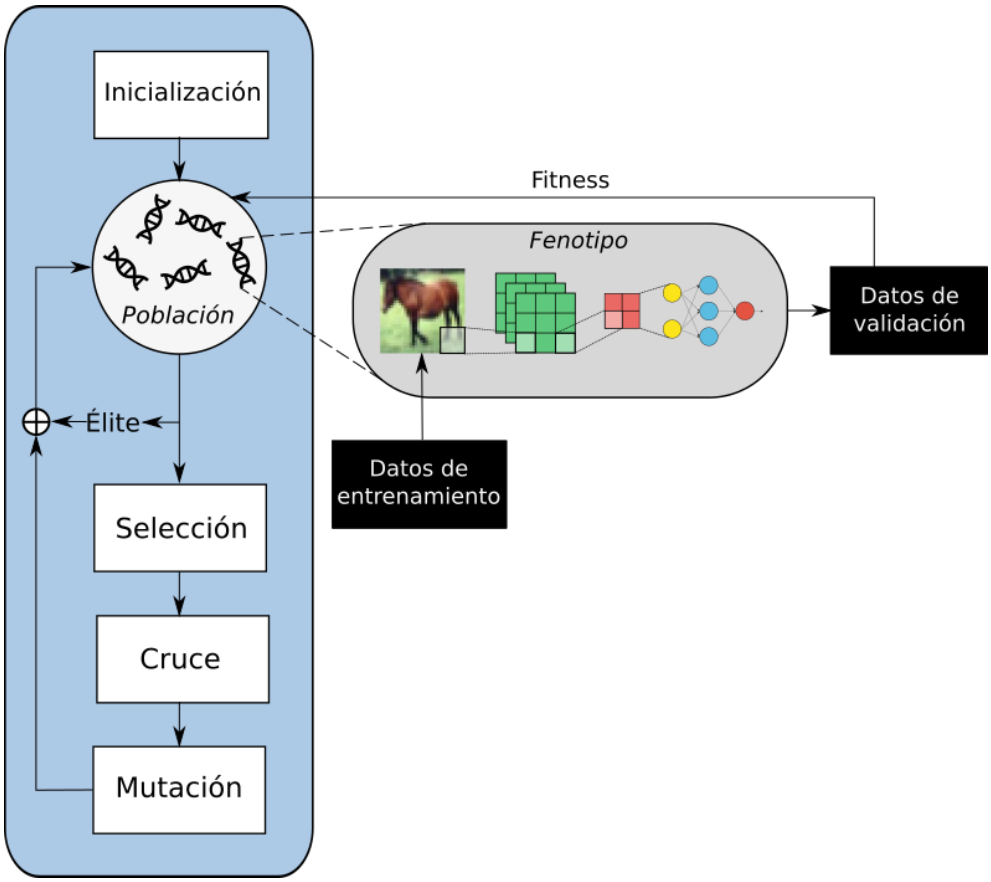


Fig. 4.1. Imagen resumen del algoritmo genético

Una vez se conocen los parámetros del algoritmo se pasará a describir los parámetros de la red de neuronas convolucionales que va a optimizar el algoritmo, explicando los que no se hayan tratado con anterioridad. Estos parámetros vienen determinados por el espacio de búsqueda diseñado en el trabajo de Baldominos et al[1], y son los siguientes:

- Parámetros generales de la red:
  - Tamaño del lote o *batch*: El tamaño de *batch* indica cuántos ejemplos de entrenamiento se van a introducir a la vez a la red, pues ésta puede realizar el aprendizaje de forma paralela. Un tamaño de lote bien ajustado mejora la velocidad de aprendizaje y la capacidad de aprendizaje de la red.
  - Regla de aprendizaje: Son distintas versiones del descencenso del gradiente, que se describió en el apartado 2, con propiedades que mejoran o empeoran el aprendizaje dependiendo del caso a tratar para la red.
  - Razón de aprendizaje.
- Parámetros de las capas convolucionales:
  - Número de capas convolucionales.

- Número de filtros en cada capa.
- Tamaño del filtro de cada capa.
- Función de activación de cada capa.
- Tamaño de la capa de *pooling* asociada a cada capa: El algoritmo deberá determinar si es necesaria una capa de *pooling* y, de serla, se determinará cuál es su tamaño.

- Parámetros de las capas densas:
  - Número de capas.
  - Tipo de capa para cada una: Indica si la capa es una capa densa normal, como sería en un perceptrón multicapa, o si es una capa recurrente. De ser el segundo caso variaría el patrón de conexión entre las capas.
  - Número de neuronas para cada capa.
  - Función de activación para cada capa.
  - Tipo de regularización de pesos para cada capa densa: Son penalizadores que se aplican sobre los pesos de la capa durante la optimización. Permiten prevenir el sobreajuste de la red durante el entrenamiento.
  - *Dropout* para cada capa: El *dropout* elimina con una probabilidad dada una neurona de la capa, provocando que la red tenga que realizar esa iteración de aprendizaje sin ella. De esta manera se reduce el sobreajuste y se consigue cierta redundancia en el conocimiento que almacenan las neuronas.

4.4. Implementación

En este apartado se tratará la forma en la que el desarrollador ha decidido implementar todas las funciones tratadas en los temas anteriores para el correcto funcionamiento del proyecto. Para ello se tratarán por separado las partes relevantes del proyecto que han exigido una especial dedicación.

Debido a que en la sección dedicada a la experimentación se hablarán de distintas variantes del algoritmo en algunos apartados se describirán distintos diseños probados para cumplir la función que se trata.

4.4.1. Codificación

La primera codificación es una leve modificación del cromosoma empleado en el trabajo de Baldominos et al.[1], que sigue la siguiente estructura binaria:

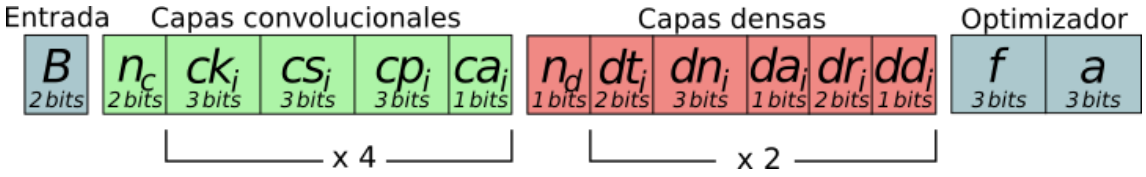


Fig. 4.2. Imagen resumen de la codificación

En la imagen se pueden observar tres colores que identifican las partes del cromosoma. La parte grisácea (tanto a la izquierda como a la derecha) representa ese conjunto de parámetros a los que se denominaba en el apartado anterior como parámetros generales. En verde se encuentran los parámetros propios de las capas convolucionales y en rojo aquellos que son propios de las capas densas.

Los algoritmos genéticos se ven favorecidos si los cambios pequeños en el genotipo conllevan pequeños cambios en el fenotipo. Por este motivo, ya que el dominio sobre el que se aplica la evolución lo complica, se ha decidido hacer uso de código binario reflejado o código Gray. Este código permite que dos números consecutivos se diferencien solo en un bit, ayudando a que los cambios sean más suaves.

A continuación se describirá en profundidad la estructura del cromosoma, indicando qué representa a cada uno de los cuadrados de la imagen:

- B: El tamaño de lote. Se le dedican dos bits y se calcula mediante $B = 25 \cdot 2^{bits}$, tomando los valores $\{25, 50, 100, 200\}$.
- n_c: Número de capas convolucionales, comenzando a contar a partir de uno. Se le dedican dos bits, por lo que cubre entre una y cuatro capas.
- ck_i Número de filtros de la capa i. Se le dedican tres bits, por lo que, al calcular la cantidad como $ck_i = 2^{bits+1}$, resulta en que puede tomar los valores $\{2, 4, 8, 16, 32, 64, 128, 256\}$.
- cs_i: Tamaño de filtro de la capa i. Se le dedican tres bits, por lo que, al calcular la cantidad como $cs_i = 2 + bits$, resulta en que puede tomar valores entre el dos y el nueve.
- cp_i: El tamaño de la capa de *pooling* tras la capa i. Con tres bits dedicados en el cromosoma, se calcula mediante $cp_i = 1 + bits$ y puede tomar valores entre el uno y el ocho. Dado que se fuerza que los filtros sean cuadrados, si el algoritmo elige como tamaño de filtro un uno es lo mismo que no aplicar *pooling* tras esa capa de convolución.

- ca_i: La función de activación de la capa de convolución. Dado que solo dispone de un bit oscila entre la función ReLU y la lineal.

Ya que el tamaño del cromosoma debe ser el mismo independientemente del individuo se leerán solo los parámetros sub i que indique el número de capas. Lo mismo ocurre para las capas densas que se verán a continuación.

- n_d: Número de capas densas. Se le dedica un solo bit, por lo que puede generar una o dos capas.
- dt_i El tipo de la capa i. Con dos bits dedicados puede tomar como valores {*RNN*, *LSTM*, *GRU*, *Dense*}.
- dn_i: Número de neuronas de la capa i. Se le dedican tres bits, por lo que, al calcular la cantidad como $dn_i = 2^{3+bits}$, resulta en que puede tomar los valores {8, 16, 32, 64, 128, 256, 512, 1024}.
- da_i: La función de activación de la capa. Dado que solo dispone de un bit oscila entre la función ReLU y la lineal.
- dr_i: La regularización aplicada sobre los pesos de la capa i. De dos bits, puede tomar como valores Sin regularización, L1, L2, L1 y L2. En el caso de haberla siempre se aplicará con un coeficiente de 0'1.
- dd_i: La probabilidad de *dropout* de la capa i. De un solo bits, puede tomar como valores con *dropout* o sin él. En el caso de haberlo siempre se aplicará con un coeficiente de 0'5.
- f: El método de descenso del gradiente empleado en la red. Con tres bits dedicados, el valor de este parámetro puede ser {*SGD*, *RMSprop*, *Nadam*, *Adagrad*, *Adamax*, *Adam*, *Adadelta*}. En este caso hay cierta redundancia con *SGD*, ya que Keras no proporciona ocho optimizadores diferentes.
- a: La razón de aprendizaje que tomará el optimizador. Con tres bits dedicados puede tomar los valores $\{10^{-5}, 5 \cdot 10^{-5}, 10^{-4}, 5 \cdot 10^{-4}, 10^{-3}, 5 \cdot 10^{-3}, 10^{-2}, 5 \cdot 10^{-2}\}$.

Dado que el número máximo de capas convolucionales es de cuatro y de capas densas dos, la longitud total del cromosoma será de 69 bits.

La segunda codificación no es más que una ampliación de la primera, ampliando el espacio de búsqueda para permitir hasta un total de ocho capas convolucionales. Esto conlleva que se debe aumentar la cantidad de bits que determinan el número de capas (de dos a tres) y aumentar el tamaño del cromosoma lo suficiente como para que pueda codificar también los parámetros propios de esas nuevas capas. En total, este nuevo cromosoma alcanza 110 bits de tamaño, dificultando más la búsqueda pero permitiendo el desarrollo de topologías más potentes.

La representación de este segundo cromosoma sería:

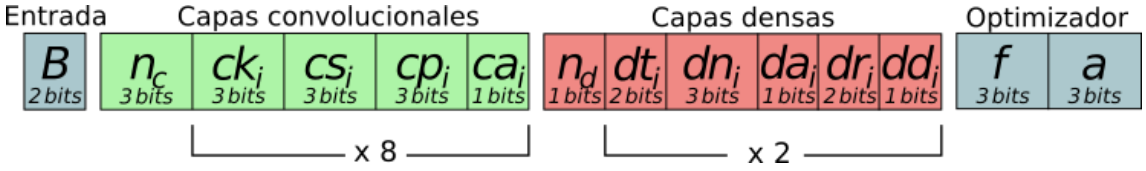


Fig. 4.3. Imagen resumen de la codificación ampliada

4.4.2. Inicialización de la población

Durante la inicialización de los individuos se generan de forma aleatoria cadenas binarias que configurarán los posteriores genotipos. Ya que el proceso evolutivo se basa en la población existente se requiere que dicha inicialización proporcione los individuos más aptos posibles frente al entorno (que desempeñen de forma aceptable la tarea) y lo más distintos posibles (que haya variedad genética).

En este problema se considerará que un individuo se comporta de una forma aceptable frente al problema cuando su fenotipo sea válido para la construcción de la red, más allá del resultado que logre ésta. Por otro lado, se considera variedad genética cuando la comparación entre dos genotipos proporciona una distancia Hamming suficiente.

La distancia Hamming no es más que el número de bits diferentes que se encuentran al comparar los cromosomas de dos individuos. Este valor, que tiene como valor máximo el tamaño del individuo, es útil para detectar si los individuos generados están cerca o lejos en el espacio de búsqueda.

Una vez determinados estos principios se pasará a describir las tres inicializaciones implementadas:

- Aleatoria en el espacio de soluciones válidas: Debido a la posibilidad de que haya individuos no válidos por fallo en las dimensiones de los datos procesados se fuerza a que los individuos inicializados sean todos del espacio de soluciones válidas. Para ello se desecha a los quse no lo sean y se repite la creación aleatoria de individuos hasta completar la población.

- Aleatoria simple: Como se tratará posteriormente, al añadir *padding* a los datos procesados se logra que no haya soluciones inválidas, por lo que se opta por eliminar la restricción anterior y se generan los individuos de forma más eficiente.

- Secuenciado simple inhibido: Debido a los aumentos en el espacio de búsqueda se mejora la distribución de los individuos mediante este método. En este caso, se elige una distancia Hamming mínima Δ y se genera un primer individuo de forma aleatoria. Luego se irán generando individuos aleatorios, que solo se aceptarán en el caso de que la distancia a los otros individuos generadas esté, como mínimo, a más de Δ. En el caso de que esté a más de esa distancia se incluye en el conjunto de

la población. Si no lo está se repite el proceso de generación de ese individuo. Este proceso se prolongará hasta que la población esté completa.

4.4.3. Evaluación

La evaluación de una generación algoritmo se realiza después de haber compuesto, ya sea por la inicialización o la reproducción de una generación anterior, una nueva población. Para ello se ejecuta una función de *fitness* sobre cada uno de los cromosomas y se almacena su resultado para su posterior utilización. Con el objetivo de facilitar posteriores modificaciones del proceso como puede ser una paralelización de la evaluación se ha decidido ejecutarlo en un proceso diferente.

La función de *fitness* se encarga de, tras recibir un cromosoma decodificado (la decodificación se realiza en el programa principal), construir la red que se ejecutará. Para ello se hace uso de las facilidades que proporciona Keras para crear topologías secuenciales de redes de neuronas convolucionales. La función regresará un valor de *fitness*, que no es más que el mayor porcentaje de acierto que ha alcanzado la red creada durante su aprendizaje.

Para crear un correcto generador de neuronas debe conocerse las restricciones de cada tipo de capa y la forma en la que estas tratan los datos que procesan. Estas particularidades en el procesamiento de las capas convolucionales es la que provoca la posibilidad de que una red pueda ser inválida, pues, como se ha podido observar el el apartado 2.1.2 (cuando se trata la convolución), la salida de la red tiene un tamaño menor que la entrada dependiendo del tamaño del filtro empleado. Por este motivo ocurre que en el procesamiento de algunas redes, tras pasar por varias convoluciones y *poolings*, se llegue a una dimensión de los datos negativa, algo imposible que inhabilita esa topología en cuestión. En el caso de que la red creada sea inválida se devolverá un valor de *fitness* de cero, es decir, no se penaliza la existencia de individuos defectuosos en la población.

La existencia de estos individuos inválidos para el problema genera dos problemas. El primero es que se están perdiendo individuos de la población, ya limitada en número por el hardware empleado, a la hora de realizar la búsqueda y el segundo es que este problema impide la creación de redes grandes. De hecho, como se verá en los siguientes apartados, no se generan tan siquiera individuos de cuatro capas por el problema de la reducción de dimensión de los datos. Para solucionar el problema se deben realizar algunas modificaciones sobre la función de *fitness*.

El *padding* es una técnica aplicable a las capas convolucionales que consiste en rodear con píxeles sin información (a nivel práctico son ceros) la imagen a procesar. El número de filas de ceros a añadir depende del tamaño del filtro de la capa, pero será el suficiente como para que la salida tenga el mismo tamaño que la entrada sin el *padding*. La diferencia entre ambas formas de convolución sería:

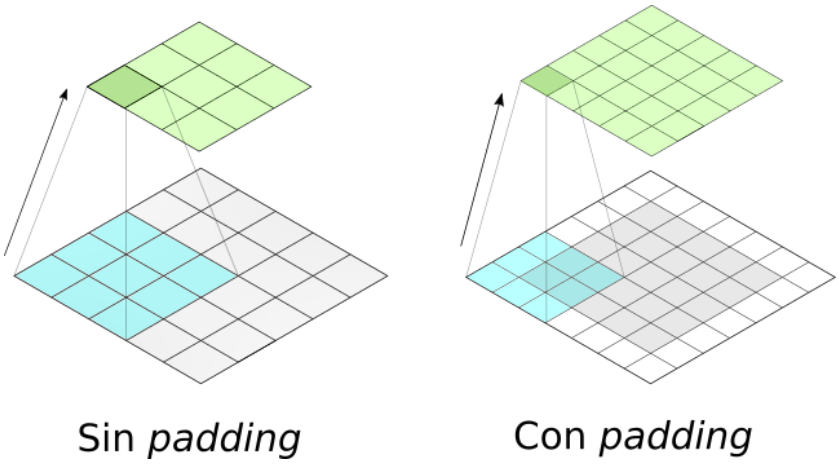


Fig. 4.4. Imagen resumen del funcionamiento del *padding*

En la anterior imagen se representa el filtro en azul, la imagen de entrada en gris, la salida en verde y en blanco los píxeles sin información fruto del padding. Como se puede ver, en las convoluciones normales la dimensión de los datos pasa de 5x5 a 3x3, mientras que al aplicar *padding* se mantiene la salida en 5x5. Es por este motivo que los expertos suelen usar el *padding*, pues es la única forma de crear modelos muy largos. Además, esta técnica mejora los resultados de las convoluciones por permitir más detalle en los límites de la imagen.

Al aplicar el *padding* a la creación de las redes neuronales de la función de *fitness* se desbloquean redes de cuatro capas (teniendo como base la primera codificación). El único inconveniente de esta técnica es que, al mover conjuntos de datos mayores y durante más capas, son necesarios más pesos en las capas convolucionales, ralentizando el aprendizaje.

Una vez se ha construído correctamente la red se debe determinar la forma de entrenar las redes. Usando los parámetros determinados por el algoritmo, la red aprenderá a partir de la mitad de los datos de entrenamiento y se medirá su eficacia mediante la mitad de los datos de *test*. De esta manera se aprenderá de 25000 ejemplos de entrada y se comprobará con otros 5000, elegidos ambos conjuntos aleatoriamente sobre el total.

Sobre el aprendizaje de las redes solo queda ajustar cuántos ciclos de aprendizaje debe pasar la red, lo que es posiblemente el parámetro que más afecta al tiempo de ejecución del algoritmo. De forma ideal se puede determinar que el mayor posible, pues siempre proporcionaría la mejor similitud con lo que sería una red completamente entrenada. Ya que el hardware es limitado, en este proyecto se han aumentado progresivamente el número de iteraciones para potenciar el proceso de búsqueda, encontrándose un límite temporal en torno a las 8 iteraciones por tarjeta gráfica en el equipo.

Por último queda describir la opción tomada para intentar agilizar el proceso de entrenamiento. Ya que se dispone de un equipo que monta dos GPU GTX 1080 se decidió paralelizar la evaluación, entrenando una red en cada tarjeta de forma simultánea. De esta manera se logra reducir el tiempo de entrenamiento a casi la mitad, lo que permite dedicar los ciclos de aprendizaje mencionados en el párrafo anterior.

4.4.4. Selección

Para la realización de los torneos de selección se eligen aleatoriamente t individuos, que se enfrentarán comparando sus puntuaciones. De esos individuos saldrá un único vencedor, que se sumará a la población intermedia apta para la reproducción.

El número de individuos que participan en cada torneo, t, ajusta un valor propio de los algoritmos genéticos llamado presión evolutiva. Un valor t bajo conllevará una nula presión evolutiva, pues se compararán muy pocos individuos al mismo tiempo y será más aleatorio quién entra en la población intermedia apta para la reproducción. Sin embargo un valor t muy alto hará casi seguro que se encuentren siempre los mejores individuos en el torneo, añadiéndose siempre a la población intermedia. Es por ello que en este trabajo se usará el tamaño de torneo más común por encontrarse entre ambas categorías, siendo éste $t=3$.

Sin embargo, para la comparación dentro del torneo no se usará el *fitness* de los individuos, sino su *fitness* ajustado. Para obtener este segundo valor se tendrá en cuenta tanto la puntuación devuelta por la función de evaluación como la similitud genética entre el individuo en cuestión y la población. De esta forma se asegura que no convergerá prematuramente en valores subóptimos y explorará en mayor profundidad el espacio de búsqueda.

El primer paso para determinar este *fitness* ajustado se debe establecer qué se entiende por similitud entre dos individuos, pues en este caso se calculará una distancia fenotípica (al contrario que la distancia Hamming, que era genotípica):

$$sim(i_i, i_j) = \begin{cases} \frac{|i_i \cap i_j|}{|i|}, & \text{si } n_c^{(i)} = n_c^{(j)} \text{ y } n_d^{(i)} = n_d^{(j)} \\ 0, & \text{en cualquier otro caso} \end{cases}$$

Concretamente, la anterior ecuación devuelve, en el caso de que sean dos redes con el mismo número de capas convolucionales y el mismo número de capas densas, cuál es el ratio de bits que comparten (bits compartidos / total de bits).

Una vez se ha definido el coeficiente de similitud entre dos individuos se calcula el *fitness* ajustado siguiendo la siguiente ecuación:

$$f_a(i_i) = f(i_i) \times \left(1 - \frac{\sum_{i_j \in P, j \neq i} sim(i_i, i_j)}{|P|-1}\right)$$

4.4.5. Cruce, mutación y elitismo

En este apartado se comentarán los operadores restantes, ya que ninguno de ellos es lo suficientemente complejo como para requerir un apartado propio. A continuación se describirán las funcionalidades y particularidades de cada uno:

- Cruce: Se ha implementado un cruce con tantos puntos de corte como indique un número aleatorio, que se genera entre entre un mínimo de tres y un máximo de diez. Una vez decidido el número de cortes se distribuyen aleatoriamente sobre los individuos a reproducir y se produce el cruce descrito en el apartado 2. Los dos individuos generados se añaden a la siguiente población.
- Mutación: Es una implementación sencilla en la que cada gen tiene una oportunidad de mutación del 1.5 %.
- Elitismo: Implementación sencilla del elitismo. Se almacena un solo individuo después de la evaluación de la población y se selecciona a la élite en base al *fitness* sin ajustar.

5. Experimentación

Una vez implementado todo el código que cubra las funcionalidades descritas en el apartado anterior se debe evaluar el rendimiento del sistema. Para ello se ejecutarán varios experimentos variando entre los operadores que hemos descrito como con varias implementaciones y modificando también el número de ciclos de aprendizaje por red.

Al modificar dichos parámetros se ha llegado a tres versiones del algoritmo evolutivo, teniendo cada una dedicada un subapartado más adelante en esta memoria. Se describirán los experimentos realizados, los resultados obtenidos y las conclusiones que se pueden extraer de ellos en el orden en el que fueron ejecutados.

Los experimentos consistirán en el algoritmo genético evolucionando progresivamente hasta que se den dos situaciones de parada. La primera sucederá cuando el algoritmo se mantenga veinte generaciones sin mejorar el resultado, mientras que la segunda será cuando el algoritmo alcance el número de iteraciones máximo, de cien generaciones.

Si bien es cierto que el mecanismo implementado del *fitness* ajustado mantiene el algoritmo en constante movimiento y sería posible que mejorase tras veinte generaciones sin mejora, la enorme cantidad de tiempo que es necesaria para cada generación (cuando el algoritmo está evaluando redes avanzadas puede alcanzar las 10 horas) provoca que resulte más eficiente detener la búsqueda.

En otros tipos de algoritmos evolutivos resultaría muy escaso un número máximo de generaciones de 100, pero en este caso el algoritmo converge hacia buenas soluciones en relativamente pocas iteraciones y, debido de nuevo a los tiempos necesarios para cada generación, que el algoritmo realice más de cien ciclos conlleva un tiempo excesivo.

Pese a lo expuesto en los dos párrafos anteriores, es cierto que en una implementación ideal en la que se disponga de un número superior de tarjetas gráficas podrían dilatarse estas condiciones, pero han debido acotarse en función al *hardware* disponible.

El algoritmo al ejecutarse no solo muestra a cada generación el estado de la población (imprimiendo cromosomas, *fitness* y *fitness* ajustado), sino que también almacena estos valores para su posterior consulta. De esta manera han quedado almacenadas en sus respectivas carpetas todos los resultados obtenidos para cada generación de cada experimento, algo que será muy útil para el correcto desarrollo de este apartado.

A la hora de evolucionar las redes neuronales se emplean aproximaciones, pues, como se ha comentado en el anterior apartado, se entrenan a bajas iteraciones y con solo la mitad de los datos disponibles. Es por este motivo que las aproximaciones serán empleadas para representar gráficamente el progreso de la evolución, mientras que el desempeño de la mejor red se obtendrá tras un entrenamiento completo por separado.

En cada uno de los apartados se incluirá una matriz de confusión para la mejor red

neuronal obtenida en el mismo. Una matriz de confusión es un método de mostrar de forma clara cómo se han clasificado los datos por parte de la red. Sigue el siguiente modelo:

		Valor predicho	
		Clase 1	Clase 2
Valor Real	Clase 1	Verdaderos Positivos	Falsos Negativos
	Clase 2	Falsos Positivos	Verdaderos Negativos

Tabla 5.1. Ejemplo para matrices de confusión

Debido a los datos del problema la matriz deberá representar diez clases, lo que hace imposible escribir qué representa cada una. Es por este motivo que se representará cada clase en función de su número asignado en el *dataset*, es decir:

- Clase 0: Aviones.
- Clase 1: Coches.
- Clase 2: Aves.
- Clase 3: Gatos.
- Clase 4: Ciervos.
- Clase 5: Perros.
- Clase 6: Ranas.
- Clase 7: Caballos.
- Clase 8: Barcos.
- Clase 9: Camiones.

Para medir el rendimiento de las redes generadas se hará uso de el porcentaje de acierto o *accuracy*. Para comparar el desempeño entre las distintas versiones del algoritmo se usará el *accuracy* obtenido por la mejor red neuronal que haya encontrado. La forma de calcular el porcentaje de acierto es:

$$accuracy = \frac{VP+VN}{FP+FN+VP+VN}$$

Es decir, el *accuracy* es el cociente entre el número de aciertos y el número total de predicciones.

5.1. Experimento 1

El primer algoritmo emplea la creación de redes neuronales sin usar *padding*, lo que ocasiona que pueda haber individuos inválidos (tal y como se ha explicado en el apartado 4). Es por este motivo por el que se emplea la inicialización aleatoria en el espacio de soluciones válidas, lo que ralentiza bastante la inicialización por el bucle aleatorio hasta completar la población inicial.

Dado que el trabajo del que parte este proyecto, realizado por Baldominos et al.[1], emplea como aproximaciones redes entrenadas durante cinco iteraciones se decidió replicar la aproximación. De esta manera, se realizaron varios experimentos con el objetivo de comprobar los siguientes factores:

- El algoritmo evoluciona correctamente las redes hacia topologías más eficaces.
- Ejecutar varias veces la misma configuración del algoritmo lleva siempre a resultados del mismo tramo, demostrando así que es un método consistente.
- Los resultados son competitivos.

Tras comprobar que el tiempo de ejecución era viable, sobre tres horas la generación, se realizaron cinco pruebas completas cuyos resultados se mostrarán en forma de tabla. En dicha tabla se muestran:

- ID Prueba: Identificador unívoco de la prueba. Indica tanto el experimento que se está realizando como el número de la prueba.
- Nº de generaciones: El número de generaciones que se han creado en esa prueba.
- Resultado algoritmo: El porcentaje de acierto calculado por el algoritmo del mejor individuo creado.
- Resultado final: El porcentaje de acierto del mejor individuo creado por el algoritmo tras haber sido entrenado por separado con todos los datos y todas las iteraciones necesarias. Solo se calculará para la prueba de mayor precisión en el campo "Resultado algoritmo".

ID Prueba	Nº de generaciones	Resultado algoritmo	Resultado final
Exp1_1	100	0.7777	——
Exp1_2	70	0.7760	——
Exp1_3	100	0.7794	0.7989

Tabla 5.2. Tabla de resultados del experimento 1

La representación gráfica de estos resultados es la siguiente:

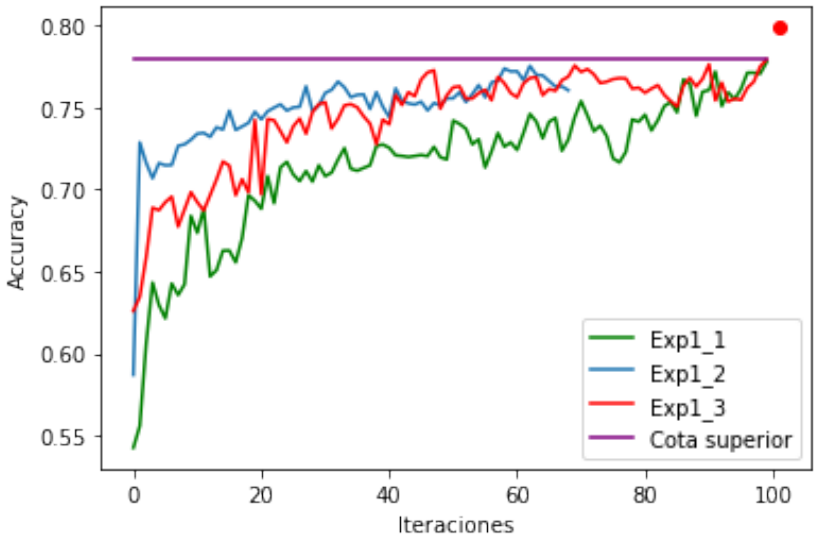


Fig. 5.1. Gráfica de resultados del experimento 1

En esta imagen se puede observar el progreso a lo largo de las generaciones de cada una de las pruebas, identificadas con los colores que aparecen en la leyenda. El punto en rojo indica el valor de *accuracy* obtenido en el entrenamiento completo, es decir, el contenido de la columna "Resultado final" de la tabla.

Como se puede observar, la duda existente sobre si el algoritmo evoluciona correctamente las redes queda resuelta. Los tres algoritmos, partiendo desde porcentajes de acierto distintos debido a la inicialización aleatoria, logran mejorar rápidamente sus resultados, como mínimo, en 15 puntos.

Además de esto se puede percibir que los tres algoritmos han llegado a valores extremadamente similares, lo que indica la robustez y consistencia del método, que no depende de la probabilidad para obtener resultados. Este hecho permitirá en próximas pruebas poder evitar repetir varias veces la misma experimentación, lo que supone un enorme ahorro de tiempo.

Por último queda comentar los resultados obtenidos. Si bien son superiores a los que se encuentran por Internet o a los obtenidos por el desarrollador en un proceso de prueba y error, al ser comparados con el actual estado del arte están lejos de ser competitivos. Es por esto que se deben desarrollar modificaciones que mejoren el desempeño de la búsqueda.

En el estudio de los resultados se observan dos características. La primera es la enorme convergencia a la que llega el algoritmo, quedando todos los resultados ligeramente por debajo de 0.78 (en morado). La segunda es la inexistencia en las poblaciones de redes de cuatro capas convolucionales, algo que permite la codificación empleada.

Como ya se ha mencionado en párrafos anteriores, la convergencia entre distintos resultados es positiva porque demuestra la robustez del algoritmo, pero es común que

los resultados oscilen más allá de las milésimas. Es por esto que surge la siguiente duda, ¿es posible que el número de ciclos de aprendizaje que se emplean suponga una cota superior a la puntuación aproximada obtenida por las redes y, por lo tanto, limiten la búsqueda? De ser esto cierto, al mantenerse todas las aproximaciones por debajo de la cota, diferenciándose entre ellas por milésimas, sería imposible evolucionar topologías que den resultados superiores.

Para comprobar si esta teoría es correcta se realizará una prueba más con esta configuración del algoritmo, con la única diferencia de que el número de ciclos de aprendizaje se subirá ligeramente de cinco a ocho. Si este aumento supone una diferencia significativa en los resultados se concluirá que las aproximaciones a bajas iteraciones conducen a resultados subóptimos. Una vez ejecutada la prueba la tabla de esta experimentación sería:

ID Prueba	Nº de generaciones	Resultado algoritmo	Resultado final
Exp1_1	100	0.7777	——
Exp1_2	70	0.7760	——
Exp1_3	100	0.7794	0.7989
Exp1_4	100	0.797	0.8145

Tabla 5.3. Tabla de resultados del experimento 1 ampliado

La representación gráfica de los resultados pasaría a ser:

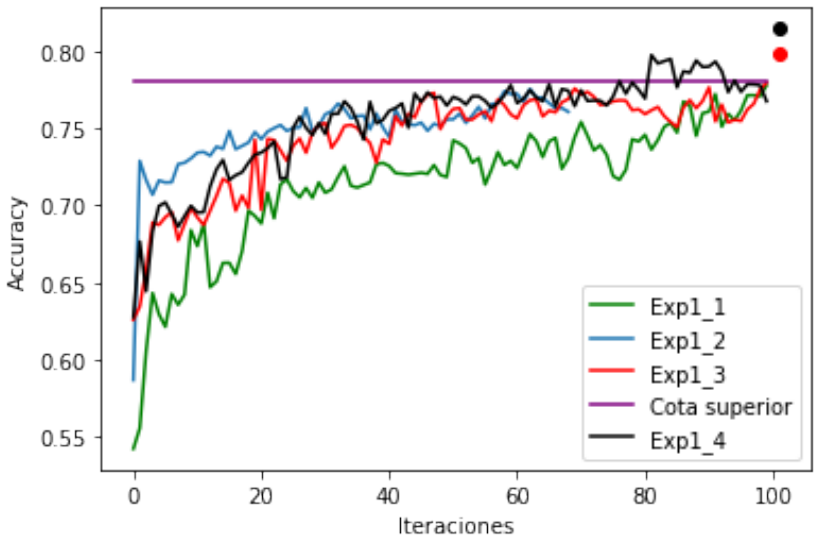


Fig. 5.2. Gráfica de resultados del experimento 1 ampliado

A la luz de estos nuevos resultados parece evidente que la teoría relacionada con el efecto negativo de las aproximaciones sobre el proceso de búsqueda era cierta. Se puede observar cómo la nueva prueba, en negro, sobrepasa esa cota superior que habían configurado las anteriores pruebas. Además, la versión entrenada completamente de esta prueba sobrepasa por casi dos puntos a la anterior. Estos resultados obligan a que las siguientes versiones del algoritmo empleen como aproximación, como mínimo, de ocho ciclos de entrenamiento.

Una vez obtenido el modelo completamente entrenado, que había sido encontrado empleando la aproximación de ocho ciclos de entrenamiento, la matriz de confusión resultante es la siguiente:

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
C0	855	10	32	13	15	3	7	7	39	19
C1	13	903	7	3	3	4	3	4	16	44
C2	46	8	726	40	60	37	44	28	8	3
C3	17	6	61	651	48	132	45	19	13	8
C4	15	1	52	44	789	21	31	34	8	5
C5	12	4	33	155	29	714	14	32	3	4
C6	6	2	29	32	23	16	886	3	2	1
C7	14	3	20	27	33	39	6	850	2	6
C8	48	20	13	5	3	7	5	5	880	14
C9	22	40	6	7	2	3	5	10	14	891

Tabla 5.4. Matriz de confusión del mejor resultado del experimento 1

A partir de esta tabla se puede ver cómo se han clasificado los ejemplos. Es especialmente relevante que el mayor número de fallos se encuentre en la clasificación entre perros y gatos, pues son las siluetas más similares dentro del conjunto de clases.

Por otro lado se encuentra la segunda característica observada: pese a encontrarse en el espacio de búsqueda, las redes de cuatro capas no aparecen en el conjunto de individuos. Esto puede deberse a las siguientes situaciones:

- Las redes de cuatro capas convolucionales no son útiles para este problema, por lo que el algoritmo las descarta.
- Las redes de cuatro capas convolucionales son útiles para el problema, pero por algún motivo el algoritmo las descarta.
- Las redes de cuatro capas convolucionales no aparecen por algún motivo relacionado con un fallo en la programación del generador automático de redes neuronales.

La primera posibilidad es imposible que sea, ya que este tipo de redes no aparecen tan siquiera entre los individuos peor puntuados. Esto indica o que no se generan estos individuos o que son rápidamente eliminados. Suponiendo que no se ha cometido ningún error en la programación queda estudiar las trazas de error en la construcción de las redes.

Al leer cuidadosamente los errores provocados en las primeras generaciones se llega a la conclusión de que cuantas más capas haya mayor es la probabilidad de que la reducción de la dimensionalidad fruto de las convoluciones y de los *pooling* lleve a que la dimensión acabe siendo negativa, lo que hace saltar el error. Además, la probabilidad de que se reincorporen es mínima, ya que una mutación sobre los genes encargados del número de capas convolucionales que lleve a las cuatro capas, con toda seguridad, lleva a una capa

no evolucionada (pues antes en la población nunca se ha tenido en cuenta), por lo que la red falla.

Ante el problema de las redes de neuronas inválidas por la dimensionalidad de los datos de entrada se perfilan dos soluciones:

- Implementar un sistema de reparación de individuos: Este algoritmo de reparación calcularía la dimensionalidad restante tras cada capa convolucional y, en el caso de que fuese negativa, reduciría de forma aleatoria en una unidad el tamaño de filtro convolucional o el tamaño de un filtro de *max-pooling*. Este proceso se realizaría iterativamente sobre cada individuo hasta que fuera válido. Tiene como ventaja que no aumenta el número de parámetros de la red, pero resulta poco eficiente de ejecutar y complejo.
- Emplear de forma generalizada *padding* con ceros: El *padding*, explicado en el apartado 4, es una práctica muy usada por expertos en redes de neuronas convolucionales, pues permite la implementación de modelos de mayor tamaño y mejora las convoluciones sobre los bordes de las entradas. Tiene como desventaja que aumenta el número de parámetros a entrenar.

Por los motivos antes expuestos se decide implementar *padding* con ceros tanto para las capas de convolución como para el *max-pooling*. Los experimentos que lo empleen pertenerán a los próximos apartados.

5.2. Experimento 2

Una vez se han descubierto los inconvenientes que encerraba la anterior versión del algoritmo, con el objetivo de mejorar el desempeño se han incluído las soluciones para el presente experimento. En este caso se realizarán dos pruebas (ambas con *padding*): la primera evolucionará aproximando el resultado a partir del resultado obtenido con ocho ciclos de aprendizaje y la segunda hará lo propio con dieciséis.

Este avance en el uso de mayores cantidades de ciclos de aprendizaje por red, con el que se busca obtener la mejor evolución posible, se realiza asumiendo un consumo de tiempo mucho mayor que en la anterior versión, aunque se ve mitigado por el uso de la paralelización. El uso de un mayor número de ciclos está completamente desaconsejado para un *hardware* similar al utilizado en este proyecto.

El resultado de la experimentación con este algoritmo es la siguiente:

ID Prueba	Nº de generaciones	Resultado algoritmo	Resultado final
Exp2_1	100	0.8345	——
Exp2_2	67	0.8566	0.8582

Tabla 5.5. Tabla de resultados del experimento 2

En ambos resultados se puede observar cómo la implementación del *padding* y el consecuente desbloqueo de las redes de cuatro capas convolucionales han supuesto una notable mejora sobre el resultado de la neuroevolución. También es cierto, que, una vez alcanzado este nivel de precisión, el hecho de entrenar la red con el total de los datos supone una menor mejora que en las versiones anteriores.

Ambas evoluciones se comparan de forma más clara en la siguiente representación gráfica:

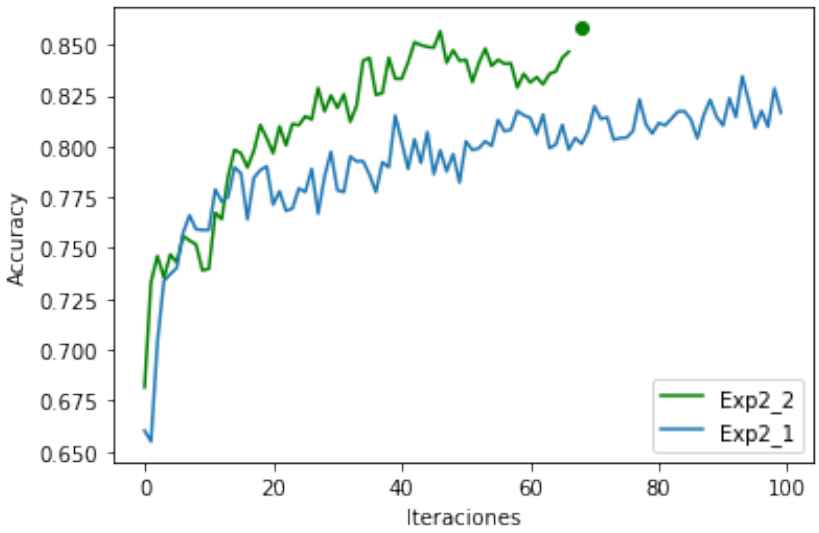


Fig. 5.3. Gráfica de resultados del experimento 2

Esta comparación vuelve a demostrar la importancia de usar elevados ciclos de aprendizaje para las aproximaciones de las redes neuronales. Es posible que aumentándolas todavía más se mejore el desempeño, pero con el *hardware* en uso es inviable por el momento.

Una vez obtenido el mejor resultado se puede realizar la matriz de confusión del experimento:

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
C0	898	6	23	13	8	3	7	6	21	15
C1	8	919	4	2	3	1	6	4	18	35
C2	42	1	795	32	50	24	38	13	3	2
C3	14	3	52	751	29	78	34	23	9	7
C4	9	1	23	32	851	15	34	30	4	1
C5	10	6	23	130	29	765	10	22	1	4
C6	5	0	26	40	16	7	895	3	3	5
C7	8	1	16	24	31	31	2	883	0	4
C8	37	6	6	9	1	4	5	0	916	16
C9	22	31	6	4	1	4	3	6	14	909

Tabla 5.6. Matriz de confusión del mejor resultado del experimento 2

Al comparar esta tabla con la del apartado anterior se puede ver cómo los números de falsas clasificaciones ha disminuido, aunque sigue siendo un problema para la red generada distinguir entre perros y gatos.

5.3. Experimento 3

En este único experimento el objetivo es superar el desempeño del mejor individuo del apartado anterior. Para ello se propone el uso de un nuevo cromosoma, lo que supone un espacio de búsqueda un 60 % mayor, en sustitución del que se ha empleado en el resto de la experimentación.

Tal y como se ha explicado en el apartado cuatro, este cromosoma expandido soporta redes de hasta ocho capas convolucionales, lo que supone mayor tiempo de evolución pero también una posible mayor potencia. El mayor inconveniente de este cambio es el espacio de búsqueda expandido en sí mismo, pues es un suceso común en los algoritmos genéticos que los espacios excesivamente grandes provocan que el proceso de búsqueda se desoriente, más teniendo en cuenta el número tan limitado de individuos y generaciones.

Los resultados de la evolución de este algoritmo (empleando 16 ciclos de entrenamiento por red) son de un *accuracy* de 86'9 %. Este resultado acaba convirtiéndose en un porcentaje de acierto de 88'3 % al realizar el entrenamiento al completo.

La gráfica que representa la evolución de esta configuración del algoritmo es la siguiente:

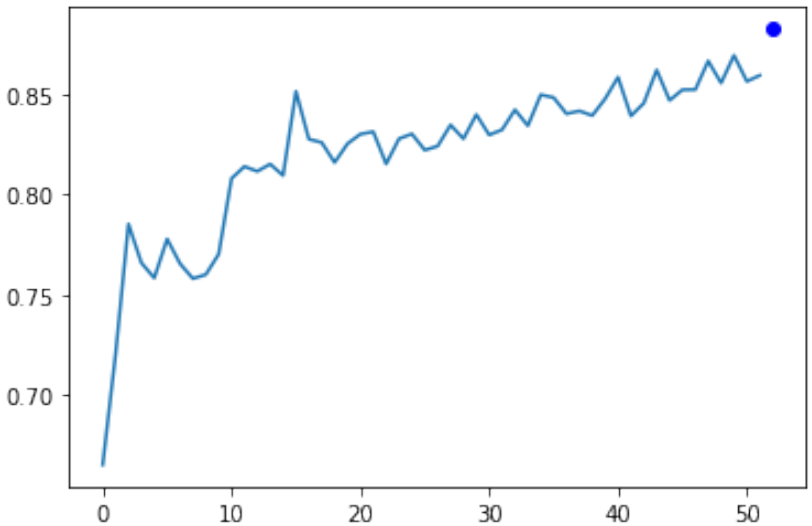


Fig. 5.4. Gráfica de resultados del experimento 3

Se puede ver cómo el proceso de neuroevolución ha pasado a ser notablemente más rápido, convergiendo hacia soluciones mejores que sus competidores en un menor número de iteraciones a pesar de tener un espacio de búsqueda mucho mayor. Con esto queda demostrada la utilidad de la nueva inicialización.

Además, los nuevos resultados demuestran como un aumento en el espacio de búsqueda ha supuesto la obtención de unos resultados notablemente superiores frente a la versión anterior del algoritmo.

Sin embargo, el tiempo asociado a una generación una vez que la población ha convergido hacia buenas soluciones es aún mayor. Esto hace imposible realizar un número elevado de generaciones, suponiendo cada generación más allá de la número 50 más de doce horas de procesamiento. Este hecho ha provocado que la experimentación no esté completa (aunque haya alcanzado buenos resultados) antes de la redacción del presente documento.

Su matriz de confusión asociada es:

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
C0	912	5	18	11	9	2	2	6	25	10
C1	10	940	1	1	1	1	1	2	10	33
C2	33	2	841	25	42	22	21	11	3	0
C3	12	2	51	749	36	89	31	20	5	5
C4	6	1	28	23	895	11	17	17	2	0
C5	3	1	28	95	31	807	10	22	1	2
C6	5	1	25	20	9	7	922	6	3	2
C7	6	0	15	17	34	22	1	903	0	2
C8	31	8	5	3	2	1	5	4	935	6
C9	13	32	7	5	1	2	0	2	12	926

Tabla 5.7. Matriz de confusión del resultado del experimento 3

En este caso ya se puede apreciar cómo la gran mayoría de los casos se clasifica de manera correcta, reduciendo el número de errores por debajo de la centena del gran problema que se crea con este *dataset*: la diferenciación entre perros y gatos.

5.4. Comparación entre los experimentos y con el estado del arte

Comparados los resultados de los anteriores tres experimentos se obtiene la siguiente gráfica:

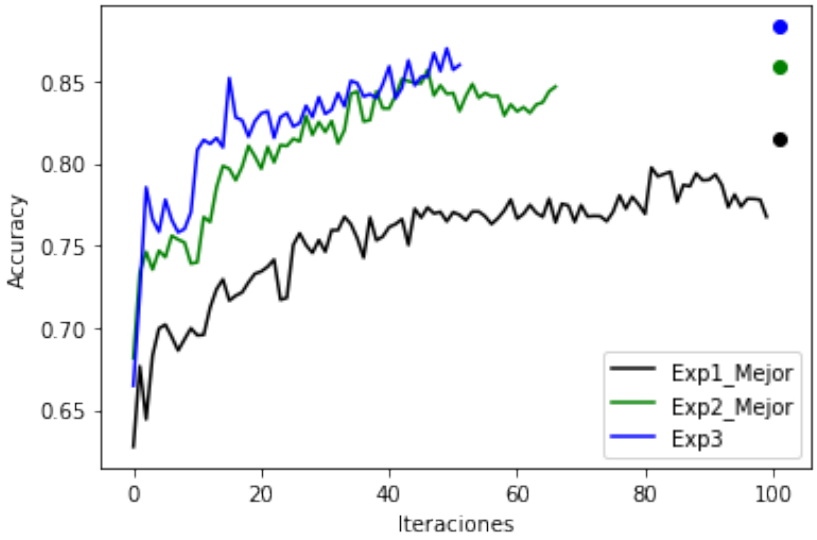


Fig. 5.5. Gráfica comparativa entre los experimentos

En esta gráfica comparativa se pueden apreciar las diferencias en las evoluciones seguidas en los tres experimentos. Claramente se aprecia que la última inicialización implementada (secuenciado simple inhibido) y la codificación extendida obtienen mejores resultados en un menor número de iteraciones. Esto hace al tercer experimento el mejor algoritmo genético encontrado para esta tarea.

Comparados los resultados del tercer experimento con el resto de trabajos relevantes mencionados en el estado del arte se obtiene la siguiente tabla:

Método	Profundidad	C10
Network in Network	-	10.41
All-CNN	17	9.08
Deeply Supervised Net	-	9.69
FractalNet	21	10.18
ResNet	110	13.63
ResNet (pre-activation)	1001	10.56
DenseNet	190	5.19
Este proyecto	6	11.7

Tabla 5.8. Tabla de comparación con el estado del arte

Como se indicó en los objetivos, el proyecto no podría buscar alcanzar un nuevo estado del arte usando neuroevolución, ya que el *hardware* empleado limita notablemente el proceso. Sin embargo, sí se ha alcanzado el objetivo de conseguir resultados competitivos aunque se emplee un equipo asequible economicamente, obteniendo un porcentaje de acierto comparable a algunos de los otros de los trabajos que se encuentran en la tabla.

6. Conclusiones

En este apartado se analizarán los resultados del anterior apartado, comparándolo con los resultados esperados y especificados en el subapartado de objetivos. A partir de dichos resultados se extraerán las conclusiones pertinentes. Además, se propondrán líneas de trabajo futuras para trabajos similares.

6.1. Conclusiones del trabajo

En los anteriores apartados se ha descrito en profundidad el diseño, implementación y aplicación del algoritmo genético para diseñar redes neuronales convolucionales para el conjunto de datos CIFAR-10. Todos los pasos descritos en los objetivos han sido completados satisfactoriamente, incluído el hecho de lograr resultados que se pudieran considerar competitivos.

Este método de desarrollo de redes neuronales no solo consigue resultados de notable calidad, sino que lo logra empleando redes de menor tamaño (como se puede ver en la tabla comparativa del apartado anterior), con entrenamientos muy breves y un nulo trabajo por parte del diseñador, pues solo tiene que poner en funcionamiento el algoritmo y esperar.

Esta facilidad a la hora de obtener redes de buena calidad sin ninguna intervención externa permite que alguien sin gran conocimiento en el tema o pericia en la creación de redes neuronales consiga resultados superiores o similares a los logrados por equipos de investigadores de primer nivel dedicados al dominio. Y, además, la ventaja de los algoritmos genéticos radica en que al variar el dominio es altamente probable que logre desenvolverse con una eficacia similar.

El mayor problema asociado a este método es el elevado coste computacional que requiere, ya que el uso de dos tarjetas gráficas de alta gama como han sido las dos GTX 1080 que poseía el equipo Lava no era suficiente finalizar una evolución completa de la última versión del algoritmo a pesar de que eran solamente 100 generaciones. Aun así, el mejor resultado obtenido se logró tras una ejecución de 13 días, un tiempo asumible. En el caso de desear que este algoritmo obtuviera mejores resultados se necesitarían más recursos o más tiempo para la ejecución.

El hecho de que incluso con las limitaciones mencionadas se hayan obtenido resultados de esta calidad indica que no solo la neuroevolución es un campo relevante y lleno de posibilidades sino que el espacio de búsqueda diseñado por Baldominos et al. [1] puede obtener excelentes resultados tras aplicarle al algoritmo y a la codificación los cambios tratados en este trabajo.

6.2. Trabajos futuros

Antes de comenzar con las posibles vías futuras para mejorar el presente trabajo se debe dejar claro que se considera que la última versión propuesta exprime practicamente al máximo las posibilidades dentro de las limitaciones de *hardware* mencionadas. Es por esto que no se aconseja realizar ninguna de las mejoras que se expondran a continuación hasta que haya más recursos disponibles.

Algunas de las posibles mejoras para el algoritmo son:

- Permitir al algoritmo realizar un alto número de generaciones: Como se ha mencionado en apartados anteriores, el hecho de que un algoritmo genético evolucione solamente durante 100 generaciones es algo escaso. Por este motivo, se propone que, una vez se disponga de recursos suficientes, se ejecute el algoritmo durante un número de generaciones elevado.

- Uso de un espacio de búsqueda todavía mayor: Ya que el algoritmo realiza de forma eficiente la búsqueda aunque se ha aumentado en más de un 60 % y este hecho ha mejorado los resultados, la posibilidad de que al seguir aumentando la posible complejidad de las redes construídas aumente también la calidad de los resultados obtenidos es alta.

- Uso de un mayor número de individuos: Es posible que, al aumentar el tamaño del espacio de búsqueda, el número de individuos utilizados no sea suficiente. Es por esto que se propone la posibilidad de aumentar el número de individuos para mejorar el proceso evolutivo.

- Modificación de la codificación para aplicar los conocimientos adquiridos en el trabajo de *Striving for simplicity:The all convolutional net* [12]: En este trabajo se demuestra no solo que las capas de *Max-Pooling* pueden ser sustituídas por capas de convolución con un *stride* (el desplazamiento que realiza el filtro sobre la imagen a la hora de realizar las convoluciones) mayor a la unidad, sino que esta sustitución logra mejorar el resultado. Es por esto que se puede modificar la codificación para cambiar el espacio reservado al *Max-Pooling* por un espacio que decida el *stride* a utilizar.

Además de estas modificaciones sobre el algoritmo, también podría usarse este método neuroevolutivo sobre un dominio más complejo, de forma análoga a como se pasó del MNIST al CIFAR-10. Algunos de los *dataset* que se podrían emplear son el CIFAR-100 o, en un futuro lejano en el que los recursos sean muy altos, el ImageNet.

Por último, también se podría emplear este algoritmo sobre un conjunto de datos de una naturaleza distinta al dominio de la visión artificial. Es posible que en ese caso haya que realizar mayores modificaciones sobre la codificación, siempre manteniendo la versatilidad que caracteriza a este espacio de búsqueda.

Bibliografía

[1] A. Baldominos, Y. Sáez y P. Isasi, "Evolutionary Convolutional Neural Networks: an Application to Handwriting Recognition", *Neurocomputing*, vol. 283, pp. 38-52, 2018.

[2] J. Koutník, J. Schmidhuber y F. Gomez, "Evolving deep unsupervised convolutional networks for vision-based reinforcement learning", en *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 541-548.

[3] A. Géron, *Hands on Machine Learning with Scikitt-Learn, Keras, and TensorFlow*. O'Reilly, 2019.

[4] Computer Science Wiki contributors, *Pictorial example of Max-pooling*, [Accedido a 6 Junio de 2019], 2018. [En línea]. Disponible en: https://computersciencewiki.org/index.php/File:MaxpoolSample2.png.

[5] C. Sammut y G. Webb, *Encyclopedia of Machine Learning and Data Mining*. Springer, 2017.

[6] A. Baldominos, Y. Sáez y P. Isasi, "On the automated, evolutionary design of neural networks: past, present, and future", *Neural Computing and Applications*, 2019.

[7] G. Miller, F. Todd, P. Hedge y S.U., "Designing neural networks using genetic algorithms", en *3rd International Conference on Genetic Algorithms*, 1989, pp. 373-384.

[8] D. Ciresan, U. Meier, J. Masci, L. M. Gambardella y J. Schmidhuber, "Flexible, High Performance Convolutional Neural Networks for Image Classification", en *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*, vol. 2, 2011, pp. 1237-1242.

[9] E. Real, A. Aggarwal, Y. Huang y Q. V Le, "Regularized Evolution for Image Classifier Architecture Search", 2018.

[10] B. Zoph, V. Vasudevan, J. Shlens y Q. V. Le., "Learning transferable architectures for scalable image recognition", en *CVPR*, 2018.

[11] M. Lin, Q. Chen y S. Ya, "Network in network", *ICLR*, 2014.

[12] J. T. Springenberg, T. B. A. Dosovitskiy y M. Riedmiller, "Striving for simplicity: The all convolutional net", *arXiv*, 2014.

[13] C. Lee, S. Xie, Z. Z. P. Gallagher y Z. Tu, "Deeplysupervised nets", *AISTATS*, 2015.

[14] G. Larsson, M. Maire y G. Shakhnarovich, "Fractalnet:Ultra-deep neural networks without residuals", *arXiv*, 2016.

[15] G. Huang, Y. Sun, Z. Liu, D. Sedra y K. Q. Weinberger, "Deep networks with stochastic depth", *ECCV*, 2016.

[16] K. He, X. Zhang, S. Ren y J. Sun, “Identity mappings in deep residual networks”, *ECCV*, 2016.

[17] G. Cornell, Z. Tsinghua, L. Maaten y K. Weinberger, “Densely Connected Convolutional Networks”, *arXiv*, 2018.

[18] The Apache Software Fundation, *Apache License, Version 2.0*, [Accedido a 6 Junio de 2019], 2004. [En línea]. Disponible en: https://www.apache.org/licenses/LICENSE-2.0.

[19] Massachusetts Institute of Technology, *The MIT License*, [Accedido a 6 Junio de 2019]. [En línea]. Disponible en: https://opensource.org/licenses/MIT.

[20] Python Software Foundation, *History and License*, [Accedido a 6 Junio de 2019], 2001. [En línea]. Disponible en: https://docs.python.org/3/license.html.

[21] Regents of the University of California, *New BSD License*, [Accedido a 6 Junio de 2019], 1999. [En línea]. Disponible en: https://opensource.org/licenses/BSD-3-Clause.

[22] Free Software Foundation, *GNU GENERAL PUBLIC LICENSE*, [Accedido a 6 Junio de 2019], 2007. [En línea]. Disponible en: https://www.gnu.org/licenses/gpl-3.0.en.html.

[23] S. Cass, “The 2018 Top Programming Languages”, *IEEE Spectrum*, 2018. [En línea]. Disponible en: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages.

Anexo I: Competencias en inglés

Introduction

Convolutional neural networks have positioned themselves as one of the most powerful methods of machine learning. Of all its variants, two stand out: the convolutional ones, which have a great ability to recognize images, and the recursives ones, used with data that have a temporal relationship.

The design of convolutional neural network architectures to solve a given task is complex and requires an almost random experimentation process with long workouts at each test. This complexity in the neuronal architectures design arises from the enormous number of parameters involved in its construction and the lack of design patterns.

The classic solution to the aforementioned problem is to hire experts in the field who dedicate their time and effort to this process of trial and error trying to improve the search through their expertise. This makes the construction of quality convolutional neural networks a slow, tedious and expensive process for companies and research groups.

Looking for ways to reduce the cost of design, meta-heuristic methods have been used to find more efficient search processes. Among these meta-heuristic methods, genetic algorithms stand out, which when applied to the problem of the neural network construction make up the field of neuroevolution. This is the field the project is focused on.

Although the field of neuroevolution has existed for around three decades, for many years the results were deficient. This was because the algorithm evolved very few parameters and the networks produced were of a small size, with only one hidden layer and a few neurons. From the technological advances in hardware of 2014 that this field begins to give competent results, specifically with the work of Koutník et al [2].

Since the development of neuroevolution is recent and the current results are promising, neuroevolution must be tested in multiple fields and with different methodologies. The present work will focus on the creation of sequential convolutional neural networks for the classification of images, a branch where there are few related works.

Objectives

The main objective of this work is to design, implement and apply a genetic algorithm capable of automatically building convolutional neuronal networks that, once trained, can correctly solve an image recognition task.

Due to the inherent complexity of the domain, this work will be based on the genetic algorithm propounded by Baldominos et al. in his work "Evolutionary Convolutional Neural Networks: an Application to Handwriting Recognition" [1], in which sequential networks are encoded in order to be applied to the MNIST dataset, formed by handwritten numbers images.

The design and implementation will consist in the creation of a correct coding of individuals and in the implementation of the functions that are convenient for the correct progression of the algorithm. Coding would be correct if it is able to represent a great variety of convolutional neural networks with the least number of incorrect individuals. The functions to be implemented are those that allow the algorithm to move towards almost optimal solutions, such as crossing or mutation. The time taken to solve the problem must be taken into account, so solutions that use parallelization or different hardware that improves performance with the minimum added cost can be evaluated.

The algorithm experimentation will consist in an extensive test that allows us to observe if this method is valid for the resolution of the task and, in addition, if it obtains results in a consistent way from different executions and not by simple chance. This part of the work that will be more expensive in a temporal way, since the training times of the algorithm are remarkably extensive.

Finally, it only remains to indicate explicitly what is the task to solve. The convolutional neuronal network resulting from this evolution process must be able to correctly classify the images contained in the CIFAR-10 dataset. This dataset is composed of a series of images classified into a total of ten categories that encompass from vehicles to animals. The aforementioned network must reach competitive values of precision in the classification taking into account the current state of the art.

Design of the solution

In this chapter we will describe the algorithm designed and the way it has been implemented. On the other hand, the same topics will be discussed about the convolutional neural networks creator.

Genetic algorithm

In this section, the design chosen for the genetic algorithm will be described. For this purpose, the document will focus on describing the global operation and the parameters that the evolution process will optimize.

The general operation of the program follows the typical steps of evolutionary algorithms:

1. Initialization of the population: The first population of individuals is created random. Initialization influences the subsequent evolution, so it can be improved by forcing the initial population to be well distributed in the search space. A certain population size must be chosen for the algorithm.

2. Evaluation: Consists in applying the fitness function to every individual, so they all have an associated score related to their performance in the task. In this case, a program will be made exclusively for this evaluation.

3. Elitism: The genotype of the n individuals best valued in the population is saved, so the algorithm randomness could not eliminate them and future generations will have them among their genetic variety. The developer must choose the number of individuals that will form the elite.

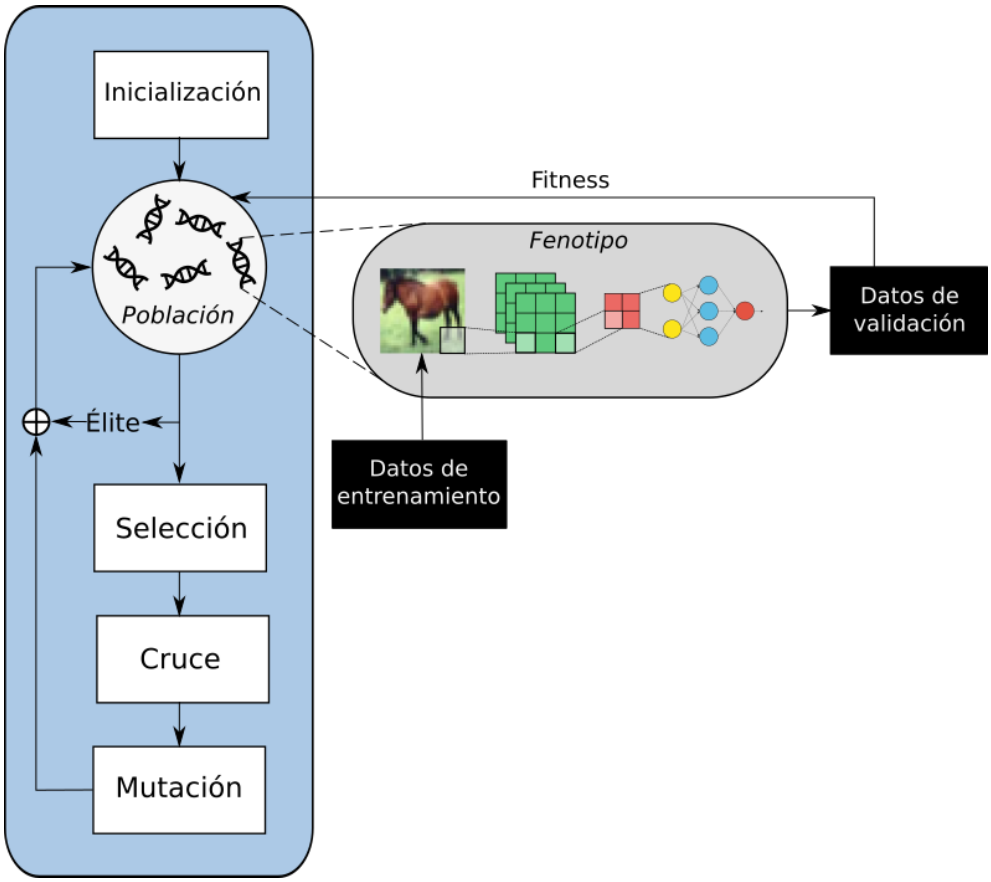
4. Selection: The tournament selection will be used, in which the performance of several individuals are compared and the best one is added to the population suitable for reproduction. A certain tournament size must be chosen, which will indicate how many individuals are clashed in each round.

5. Crossover: Two random individuals of the population suitable for reproduction are chosen and their genotypes are crossed by one or more points (number that must also be determined by the developer).

6. Mutation: Genes of each individual are modified with a certain probability. This possibility must be adjusted by the developer, because if it is too high it will make impossible to converge to the algorithm and if it is very low it will make this operator irrelevant. After being completed, the new population will join the old elite and the algorithm will return to step 2.

From the previous points it is concluded that the developer must adjust the population size, the evaluation function, the elitism rate, the tournament size, the number of crossing points and mutation rate. In addition to this you must indicate how many generations the algorithm will run or if it will stop for other reasons.

The algorithm design would be summarized in the following image:



Summary image of the genetic algorithm

Once the parameters of the algorithm are known, the parameters of the convolutional neural network that will optimize the algorithm will be described. These parameters are determined by the search space designed in the work of Baldominos et al [1], and are the following:

- General network parameters:
  - Batch size: The batch size indicates how many training examples are going to be introduced to the network at the same time, because the network can learn several samples in a parallel manner. A well-adjusted batch size improves the learning speed and learning capacity of the network.
  - Learning rule: There are different versions of gradient descent, the iterative method that reduces the network error, with properties that improve or worsen learning depending on the case to be treated.

        - Learning rate: It determines the speed of the gradient descent algorithm. An overly large learning rate will cause learning to not converge to minimums, while an excessively low ratio will cause the algorithm to get stuck at local minimums.

- Convolutional layers parameters:

    - Number of convolutional layers: The number of convolutional layers that the resulting network will have.
    - Number of filters in each layer: The number of filters of each convolutional layer that will be applied over the input.
    - Filter size of each layer: The filter size of each convolutional layer. It will force the filters to be square.
    - Activation function of each layer: The function that will be applied to the result of each convolutional layer.
    - Pooling size associated to each layer: The algorithm must determine if a pooling layer is necessary and, if so, its size will be indicated.

- Dense layers parameters:

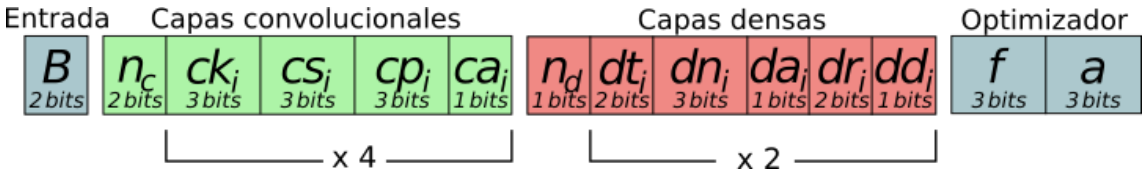
    - Number of layers: The number of dense layers that the resulting network will have.
    - Type of layer for each dense layer: Indicates if the layer is a normal dense layer, as it would be in a multilayer perceptron, or if it is a recurrent layer. If it is the second case, the connection pattern between the layers would vary.
    - Number of neurons for each dense layer.
    - Activation function for each dense layer.
    - Weight regularization type for each dense layer: These are penalties that are applied on the layer weights during the optimization. They prevent the overfitting during training.
    - Dropout rate for each layer: The dropout eliminates with a given probability a neuron of the layer, causing the network to have to perform that iteration of learning without it. In this way the overfit is reduced and a certain redundancy in the knowledge stored by the neurons is achieved.

Implementation

This section will discuss the way in which the developer has decided to implement all the aspects previously described about the genetic algorithm and the neural network creator (fitness function).

Some functions have several implementations depending the experiment where they have been used. All implementations will be explained for the correct understanding of the next sections.

About the coding, two variants have been implemented. The first one consist in a slight modification of the chromosome used in the work of Baldominos et al. [1], which follows this binary structure:



First coding summary

In the image you can see three colors that identify the parts of the chromosome. The gray part (both left and right) represents that set of parameters that were called in the previous section as general parameters. In green are the convolutional layers parameters and, in red, those parameters that are from the dense layers.

Genetic algorithms are favored if small changes in the genotype entail small changes in the phenotype. For this reason it has been decided to use reflected binary code or Gray code. This code allows two consecutive numbers to differ by only one bit, helping to make the changes smoother.

This first genotype has a length of 69 bits, which codificates all previously explained parameters in a range of values. Precisely, the coding is:

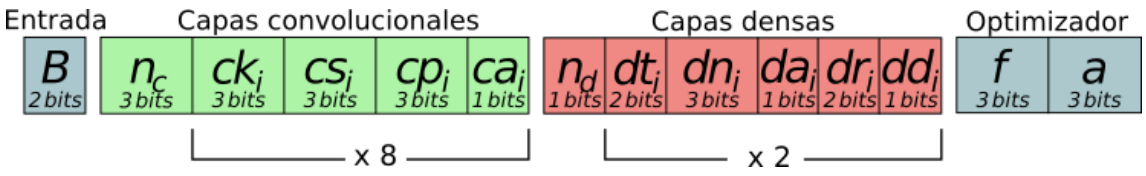
- B: The batch size (two bits) is calculated as $B = 25 \cdot 2^{bits}$, taking values {25, 50, 100, 200}.
- n_c: Number of convolutional layers, starting from one. It is codified by two bits, so the resulting network will have between one and four layers.
- ck_i Number of filters for the i convolutional layer. It has three bits dedicated and it is calculated as $ck_i = 2^{bits+1}$, taking values like {2, 4, 8, 16, 32, 64, 128, 256}.
- cs_i: The filter size for i convolutional layer (three layers) is calculated as $cs_i = 2 + bits$, taking values between two and nine.
- cp_i: Pooling size after the i convolutional layer. Codified by three bits, is is calculated as $cp_i = 1 + bits$ and it takes values between one and eight. If the algorithm determinates pooling size to be one it is the same as considerate it like no pooling layer.
- ca_i: The activación function for the i convolutional layer. It is determined by only one bit, so its possible values are the ReLU function and the lineal function.

Depending on the numbers of convolutional layers, the algorithm will only read the relevant bits for those layers. The same thing happens with the followings dense layers:

- n_d: Number of dense layers. With only one bit dedicated, it can generate one or two layers.
- dt_i The i layer type (two bits) can take as value $\{RNN, LSTM, GRU, Dense\}$.
- dn_i: Number of neurons at the i layer. With three bits dedicated, it is calculated as $dn_i = 2^{3+bits}$, taking values from {8, 16, 32, 64, 128, 256, 512, 1024}.
- da_i: The activation function of the i layer. It is determined by only one bit, so its possible values are the ReLU function and the lineal function.
- dr_i: The weight regularization for the i layer. Using two bits, it can take values from No regularization, L1, L2, L1 y L2. If the weight regularization is applied, it will be only with a 0'1 rate.
- dd_i: Dropout rate at i layer. Taking just one bit, it indicates if the layer applies dropout or not. If it is applied, its rate will be 0'5. puede tomar como valores con *dropout* o sin él. En el caso de haberlo siempre se aplicará con un coeficiente de 0'5.
- f: The gradient descent method used (three bits), taking values from the following list: $\{SGD, RMSprop, Nadam, Adagrad, Adamax, Adam, Adadelta\}$. In this parameter exist some redundancy (at *SGD*), since Keras do not support eight optimizers.
- a: The learning rate used by the network. With three bits dedicated, it can take values from $\{10^{-5}, 5 \cdot 10^{-5}, 10^{-4}, 5 \cdot 10^{-4}, 10^{-3}, 5 \cdot 10^{-3}, 10^{-2}, 5 \cdot 10^{-2}\}$.

The second coding is only an extension of the first, expanding the search space from four convolutional layers to eight. This means that you must increase the number of bits that determine the number of layers (from two to three) and increase the size of the chromosome enough so you can also code the parameters of those new layers. In total, this new chromosome reaches a size of 110 bits, making it more difficult to search but allowing the development of more powerful network topologies.

The representation of this second chromosome would be:



Second coding summary

During the initialization of the individuals, binary chains are generated randomly. Since the evolutionary process is based on the existing population, this initialization is required to provide the most suitable individuals possible in the environment (the individual performs the task in an acceptable way) and as different as possible compared to the rest of the population (genetic variety).

Once these principles are determined, the three implemented initializations will be described:

- Random valid initialization: Due to the possibility of invalid individuals, the initialization force them to be in the valid solutions space. To do so, those who are not valid are rejected and the random creation of individuals is repeated until the population is completed.
- Random initialization: As will be discussed later, adding padding to the processed data ensures that there are no invalid solutions, so individuals can be generated more efficiently.
- Simple inhibited sequencing: Due to the increases in the search space, the distribution of individuals is improved by this method. In this case, a minimum Hamming distance (counter of different bits between two individuals) Δ is chosen and a first individual is randomly generated. Then random individuals will be generated, which will only be accepted if the distance to the other individuals generated is at least more than Δ. In the event that it is more than that distance, it is included in the population as a whole. If it is not, the generation process of that individual is repeated. This process will continue until the population is complete.

The evaluation algorithm is performed after having composed, either by initialization or by reproduction of a previous generation, a new population. To do this, a fitness function is executed on each of the chromosomes and its result is stored for later use. In order to facilitate later modifications of the process, such as a parallelization of the evaluation, it has been decided to execute it as a different process.

The fitness function is in charge of building the network to be tested. For this purpose, the facilities provided by Keras are used to create sequential topologies of convolutional neural networks. The function will return a value of fitness, which the highest accuracy reached by the network.

There exist invalid individuals, so the more efficient way to eliminate that possibility is applying padding to the inputs. The padding is a technique applicable to the convolutional layers where the inputs are surrounded by pixels without information (those pixels are zeros) before the image is processed. The number of rows of zeros to add depends on the filter size of the layer, but it has to be enough for keeping the output size as input size.

Networks will be trained with one half of training samples and with a reduced number of iterations. This will cause the network result to be an approximation, which may have

an impact in the evolution process. That number of learning iterations will be discussed later on this document.

At the selection phase, it has been implemented a niching strategy. This method avoids the possibility of an early convergence making genetic variability a ratio that affects fitness value multiplying it. It will be applied a tournament size of three.

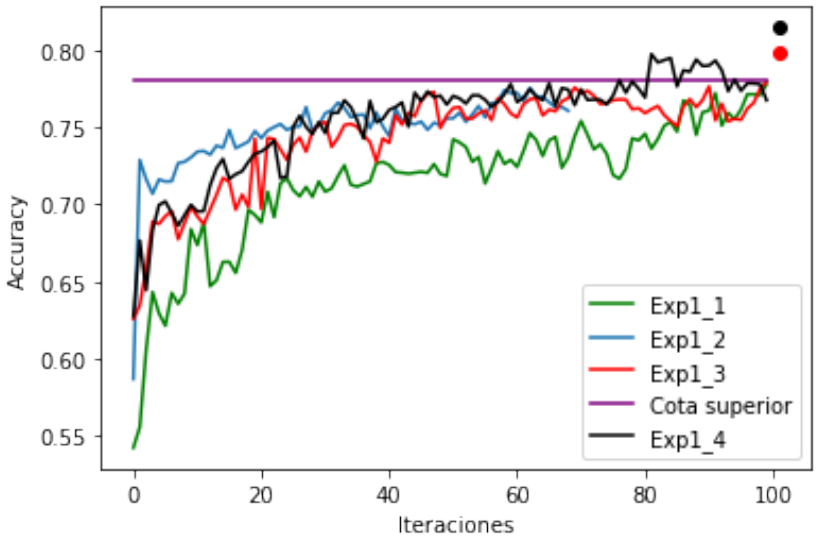
The other operators will have a typical configuration, using a multi-point random crossover, a mutation rate of 0'015 and just one individual as elite. At the crossover phase, the two descendants will be added to the new population.

Experiments

Once all the code has been implemented, the performance of the system should be evaluated. For this, several experiments will be executed varying between the operators that we have described previously and also modifying the number of learning cycles per network.

With those algorithm modifications three different versions have been created. The first one is similar to the one described at Baldominos et al., using almost the same codding. The second version applies padding at each convolutional layer, allowing larger networks, and the learning cycles are increased. The third one uses the extended coding.

The graphic representation of the results obtained by the first version is:

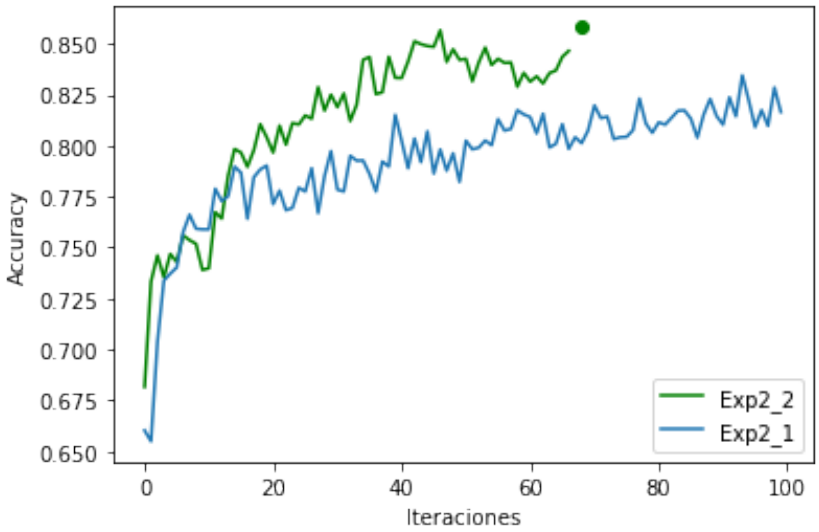


First experiment results

The first three tests evolve correctly, but they can not surpass an accuracy of 78 %. The fourth increases slightly the number of learning cycles, from five to eight, and manages to overcome the others until reaching a 79'7 %. Once correctly trained, the best network result is represented by the black point (the red dot corresponds to the correctly trained third test result), of 71'45 % accuracy.

After checking the four-layer networks are discarded by the algorithm and that increasing the number of learning cycles improves the evolution process, we modified the algorithm adding padding and increasing learning cycles even more.

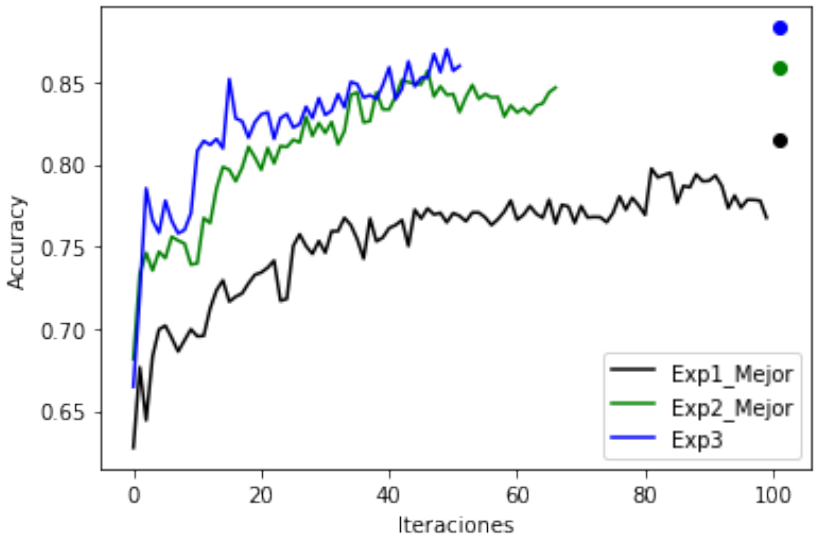
The graphic representation of the results obtained by the second version is:



Second experiment results

This time, the first test uses approximations with eight learning cycles, meanwhile second test uses sixteen iterations. As we can see, results are better that the previous experiment and the increase in iterations number also improves the accuracy reached. First test has an accuracy of 83'45 % and the second one has 85'66 % (85'82 % when it is well-trained).

Finally, the third experiment only changes the individual coding. This time, coding is 110 length, allowing networks to have even eight convolutional layers. Taking into account a searching space increase of 60 %, a better initialization is needed, so simple inhibited sequencing will be used. The comparison between all experiments could be seen in the following graphic:



Comparison between experiments results

As we can see, third experiment achieves better results in fewer generations, which indicates the new initialization effectiveness and the new coding capacity. This experiments increases accuracy until 86'9 %, being 88'3 % when it is well-trained.

Conclusions

In the previous sections, the genetic algorithm design, implementation and application has been deeply described. All the steps described in the objectives have been satisfactorily completed, including the achievement of results that could be considered competitive compared with the current state-of-art.

This method of developing neural networks not only achieves results of remarkable quality, but achieves it using smaller networks, with very short training and a null work by the designer, who just have to start the algorithm and wait for the result.

This facility allows someone without great knowledge in the subject or expertise in the creation of neural networks to achieve superior or similar results to those achieved by teams of first level researchers dedicated to the domain. Also, the advantage of genetic algorithms lies in the fact that it will find solutions of similar quality not matter what the domain is.

The biggest problem associated with this method is the high computational cost that it requires, since the use of two high-end graphics cards such as the two GTX 1080 as Lava computer possessed was not enough to complete a full evolution process of the latest version of the algorithm even though they were only 100 generations. Even so, the best result obtained was achieved after a 13 days execution, what is an assumable time. In the case of wanting this algorithm to obtain better results, more resources or more time for execution would be needed.

The fact that even with the mentioned limitations it has been obtained results of this quality level indicates that not only neuroevolution is a relevant field and full of possibilities but the search space designed by Baldominos et al. [1] can obtain excellent results after applying the changes discussed in this work.

Just before starting with the possible future ways to improve the present work it should be made clear that the last proposed version expresses practically the maximum possibilities within the limitations of *hardware* mentioned. This is the reason why it is not advised to make any of the improvements that will be discussed below until more resources are available.

These are possible ways to improve the algorithm:

- Keep evolutionary process working more generations.
- Make search space bigger.
- Increase population number.
- Modify the coding changing pooling layers by more convolutional layers with bigger stride, as indicated at “Striving for simplicity:The all convolutional net” [12].
- Testing it with more complex image recognition datasets as CIFAR-100 and ImageNet or using different domain datasets.

Anexo II: Estructura de la mejor red

La mejor red, obtenida mediante la tercera experimentación, alcanza un porcentaje de acierto del 88'3 %. Su arquitectura se muestra a continuación:

Capas	Tamaño de la salida	Configuración
Conv_1	32x32	256 filtros de tamaño 3. Función ReLU.
Conv_2	32x32	256 filtros de tamaño 3. Función ReLU.
MP_1	16x16	Un tamaño de *pooling* de 2.
Conv_3	16x16	256 filtros de tamaño 5. Función ReLU.
Conv_4	16x16	256 filtros de tamaño 6. Función ReLU.
Conv_5	16x16	256 filtros de tamaño 8. Función ReLU.
MP_2	8x8	Un tamaño de *pooling* de 2.
Conv_6	8x8	256 filtros de tamaño 6. Función ReLU.
MP_3	2x2	Un tamaño de *pooling* de 5.
Dense_1	-	128 neuronas con una función de activación ReLU.
Dense_2	-	256 neuronas con una función de activación lineal.
Dense_salida	-	Diez neuronas de salida con función de activación Softmax.