



INSTITUTO BME

**MÁSTER EN INTELIGENCIA ARTIFICIAL
APLICADO A LOS MERCADOS FINANCIEROS**

Diseño de una red neuronal con topología evolutiva

Autor
Marta Villagrán Prieto

Dirigido por
Guillermo Meléndez Alonso

Madrid
Diciembre 2021

Resumen

El aprendizaje profundo o *Deep Learning* permite a modelos computacionales que están compuestos por diferentes capas de procesamiento, aprender una representación de los datos de entrada con múltiples niveles de abstracción. Estos métodos han mejorado sustancialmente el estado del arte en técnicas como el reconocimiento visual y del habla, la detección de objetos y el descubrimiento de nuevos fármacos y genomas. El Deep Learning descubre nuevas estructuras intrincadas en grandes conjuntos de datos utilizando la técnica del descenso por gradiente para indicar cómo el modelo debe adaptar sus parámetros internos para calcular la representación de cada capa en función de la anterior. Esta previa definición de la topología del modelo, es precisamente una de sus limitaciones iniciales, ya que depende del conocimiento previo del autor o meramente del azar. En la actualidad, el proceso conjunto del Deep Learning está dividido en dos etapas, una inicial de diseño de la arquitectura y otra de entrenamiento de la misma. El objetivo fundamental de este TFM consiste en aunar ambas fases en una única de modo que se permita la construcción de una red neuronal evolutiva.

De este modo, *FluidNet* permite optimizar simultáneamente el proceso de búsqueda de la mejor topología de la red así como de sus pesos relativos. El término de *FluidNet* se emplea con el objetivo de extrapolar una propiedad muy importante de los fluidos al ámbito de la Inteligencia Artificial. Las moléculas de este tipo de sustancias no se mantienen en posiciones fijas, como ocurre con las de los sólidos, sino que se pueden mover libremente deslizándose unas sobre las otras y esto es lo que impide que la materia en dichos estados tenga forma propia y por ende, adquiera la del recipiente que lo contiene. Con este proyecto se pretende conquistar esta propiedad también en el paradigma de las redes neuronales dotándolas para que adopten la topología de los datos que reciben, con el fin de estar permanentemente aprendiendo de las entradas de su entorno.

Abstract

Deep learning enables computational models that are composed of different processing layers to learn a representation of the input data with multiple levels of abstraction. These methods have substantially improved the state of the art in techniques such as visual and speech recognition, object detection, and the discovery of new drugs and genomes. Deep Learning discovers new intricate structures in large datasets using the gradient descent technique to indicate how the model should adapt its internal parameters to compute the representation of each layer based on the previous one. This prior definition of the model's topology is precisely one of its initial limitations, since it depends on the author's prior knowledge or merely on chance. At present, the whole Deep Learning process is divided into two stages, an initial architecture design stage and a training stage. The main objective of this project is to combine both phases into a single one in order to allow the construction of an evolutionary neural network.

In this way, *FluidNet* allows to simultaneously optimize the process of finding the best topology of the network as well as its relative weights. The term *FluidNet* is used with the aim of extrapolating a very important property of fluids to the field of Artificial Intelligence. The molecules of this type of substances do not remain in fixed positions, as is the case with solids, but can move freely sliding over each other and this is what prevents the matter in these states from having its own shape and therefore, from acquiring the shape of the vessel that contains it. This project aims to conquer this property also in the paradigm of neural networks by providing them to adopt its topology regarding the data they receive, in order to be permanently learning from the inputs of their environment.

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	4
1.3. Metodología	4
1.4. Estructura de la memoria	5
2. Construcción de la base de datos	7
2.1. Obtención de los datos	8
2.1.1. Homogeneización y limpieza de los datos	8
2.1.2. Selección de ajustes aplicados	10
2.2. Etiquetado de los datos	11
2.2.1. Tratamiento de las imágenes	14
2.2.2. Aplicación de Transfer Learning	15
2.2.3. Diseño de un algoritmo de Self Learning para aprendizaje semi supervisado	20
2.3. Manipulación de los datos	22
2.3.1. Estructura de datos del MNIST	22
2.3.2. Estructura de datos de las series temporales	24
3. Diseño de la red neuronal evolutiva	35
3.1. Introducción a las redes neuronales	35
3.2. Entrenamiento de una red neuronal	36
3.2.1. Fase forward	36
3.2.2. Fase backward	37
3.2.3. Algoritmos de optimización estudiados	37
3.3. Principales dificultades a resolver	40
3.4. Explicación de las alternativas planteadas para la resolución	41
3.4.1. Resolución del problema con Keras	41
3.4.2. Generalización de la idea de Transfer learning	45

3.5.	Explicación de la solución final adoptada	48
3.5.1.	Arquitectura de la red	50
3.5.2.	Definición de la tarea de aprendizaje	61
3.5.3.	Diseño experimental	63
3.5.4.	Métricas de evaluación	66
4.	Diseño del algoritmo genético	69
4.1.	Adaptación de los AG al diseño de redes evolutivas	70
4.2.	Estructura general de un AG	72
4.3.	Población inicial	73
4.4.	Evaluación de los individuos	76
4.5.	Selección de los padres	77
4.6.	Reproducción de la población	79
4.6.1.	Crossover	79
4.6.2.	Mutación	82
4.6.3.	Elitismo	82
4.7.	Criterios de parada	83
4.8.	Justificación del uso de AG	84
5.	Análisis de los resultados	85
5.1.	Resultados sobre el conjunto de datos del MNIST	86
5.2.	Resultados sobre el conjunto de datos de patrones técnicos	94
5.3.	Entrenamiento de los modelos en la nube	102
6.	Conclusiones y futuros desarrollos	103
Bibliografía		105
7.	Bibliografía	105

Índice de figuras

1.1.	<i>Paralelismo de una neurona biológica y artificial</i>	2
1.2.	<i>Comportamiento de los fluidos</i>	3
2.1.	<i>Precios sin normalizar</i>	10
2.2.	<i>Precios normalizados</i>	11
2.3.	<i>Patrón con un doble suelo</i>	12
2.4.	<i>Explicación de los conceptos de un doble techo</i>	13
2.5.	<i>Imagen en crudo</i>	15
2.6.	<i>Imagen preprocesada</i>	15
2.7.	<i>Conexión ResNet</i>	17
2.8.	<i>Histograma de valores para y_{train}</i>	18
2.9.	<i>Histograma de valores para y_{test}</i>	18
2.10.	<i>Matriz de confusión</i>	19
2.11.	<i>Esquema de los distintos enfoques del DL</i>	21
2.12.	<i>MNIST Dataset</i>	23
2.13.	<i>MNIST Dataset original</i>	23
2.14.	<i>MNIST Dataset con rotación</i>	23
2.15.	<i>MNIST Dataset con traslación</i>	24
2.16.	<i>MNIST Dataset con zoom</i>	24
2.17.	<i>MNIST Dataset original</i>	24
2.18.	<i>MNIST Dataset con data augmentation</i>	25
2.19.	<i>Selección de datos de la clase minoritaria con SMOTE</i>	26
2.20.	<i>Creación de datos sintéticos con SMOTE</i>	27
2.21.	<i>Balanceo de las clases con SMOTE</i>	28
2.22.	<i>Estructura de un VAE</i>	29
2.23.	<i>Discontinuidad del espacio latente</i>	30
2.24.	<i>K-L Divergencia aplicada a dos distribuciones de probabilidad</i>	31
2.25.	<i>Arquitectura del VAE diseñado</i>	32
2.26.	<i>Evolución del loss en cada época</i>	33

2.27. Generación de datos sintéticos con VAE	34
3.1. Visualización algoritmo descenso por gradiente	38
3.2. Arquitectura de la red neuronal asociada a la matriz M	42
3.3. Visualización de la operación assign	44
3.4. Visualización de la operación de igualar	44
3.5. Arquitectura de la red neuronal problemática asociada a la matriz M	46
3.6. Solución de la arquitectura de la red neuronal asociada a la matriz M	47
3.7. Visualización de la matriz binaria según el nivel de sparsity	48
3.8. Visualización de la matriz de pesos según el nivel de sparsity	48
3.9. Evolución de las conexiones del cerebro	49
3.10. Transformación de los pesos de la capa de entrada	52
3.11. Transformación de los pesos de la capa de intermedia (I)	53
3.12. Transformación de los pesos de la capa de intermedia (II)	53
3.13. Transformación de los pesos de la capa de entrada	55
3.14. Transformación de los pesos de la capa de salida	55
3.15. Creación de una capa intermedia de pesos	57
3.16. Ajuste de los pesos de la capa de salida	57
3.17. Aplicación del padding	58
3.18. Pico abrupto en el accuracy durante el entrenamiento	59
3.19. Obtención de la distribución de probabilidad normal	61
4.1. Procedimiento de búsqueda de la red evolutiva	71
4.2. Densidad de la población de los individuos	72
4.3. Esquema del procedimiento evolutivo	74
5.1. Modelo de la red neuronal con estructura fija	88
5.2. Evolución del loss para una red densa con MNIST	89
5.3. Convergencia del modelo para una red densa con MNIST	89
5.4. Evolución del accuracy para una red densa con MNIST	90
5.5. Evolución del loss para una red evolutiva con MNIST	91
5.6. Convergencia del modelo para una red evolutiva con MNIST	92
5.7. Evolución del accuracy para una red evolutiva con MNIST	92
5.8. Estructura de la red densa para los patrones	95
5.9. Datos sin ruido	97
5.10. Datos con ruido	97
5.11. Matriz de confusión del modelo de Keras sobre los patrones	97
5.12. Datos sin ruido	99
5.13. Datos con ruido	99

5.14. <i>Matriz de confusión del modelo de Keras sobre los patrones</i>	99
5.15. <i>Efecto del sobreajuste en la simulación I</i>	100
5.16. <i>Evolución de la precisión en la red evolutiva</i>	101

Índice de tablas

2.1.	Parámetros entrenables según la complejidad del modelo	13
2.2.	Comparación con el estado del arte	17
3.1.	Doble impacto de reducir el número de neuronas de una capa	50
3.2.	Escenario I: red con el mismo número de capas	51
3.3.	Escenario II: red con menor número de capas	54
3.4.	Escenario III: red con mayor número de capas	56
3.5.	Resumen de las mejores distribuciones ajustadas	61
3.6.	Hiperparámetros del entrenamiento	66
4.1.	10 individuos de una población inicial	75
4.2.	Umbrales de los individuos	75
5.1.	Parámetros del data generator	87
5.2.	Hiperparámetros de la red densa para el MNIST	87
5.3.	Análisis de sensibilidad para el MNIST	91
5.4.	Hiperparámetros de la red densa para los patrones	96
5.5.	Análisis de sensibilidad de la red evolutiva para los patrones	98
5.6.	Análisis de sensibilidad de la red evolutiva para los patrones	102

Capítulo 1

Introducción

1.1. Motivación

Santiago Ramón y Cajal, a finales del siglo XIX, descubrió que la parte fundamental del cerebro son las neuronas: esas células que transmiten entre sí impulsos eléctricos formando un complejo circuito del que resulta lo que somos, lo que sentimos y lo que pensamos. Nacemos con 100.000 millones de ellas. El cerebro: el resultado más sofisticado y misterioso del universo conocido. Es por tanto que las redes neuronales se conciben como el arte de imitar al cerebro humano.

Antes de explicar cómo se ha materializado dicha analogía conviene entender mínimamente cómo funciona el cerebro humano. El premio Nobel español, Ramón y Cajal, descubrió a finales del siglo XIX la estructura celular del sistema nervioso y encontró así el funcionamiento de las neuronas. Cada una de ellas consta de un cuerpo celular, un árbol de ramificaciones y un axón. El canal de entrada de información son las dendritas, el procesador es el cuerpo o soma y el axón representa el canal de salida. El cerebro tiene miles de millones de neuronas y aprende al reforzar las conexiones entre ellas. Ramón y Cajal defendía también que las neuronas se interconectaban entre sí de forma paralela. Dicha conexión, llamada sinapsis (se realiza de forma eléctrica y también química) y permite la transmisión de información entre ellas.

Desde que un ser humano nace hasta su adolescencia su cerebro está en continua evolución. En esta etapa se generan muchas sinapsis neuronales como resultado de los aprendizajes que va adquiriendo, a la vez que entrena la plasticidad neuronal que representa la capacidad del cerebro para reorganizarse y formar nuevas cone-

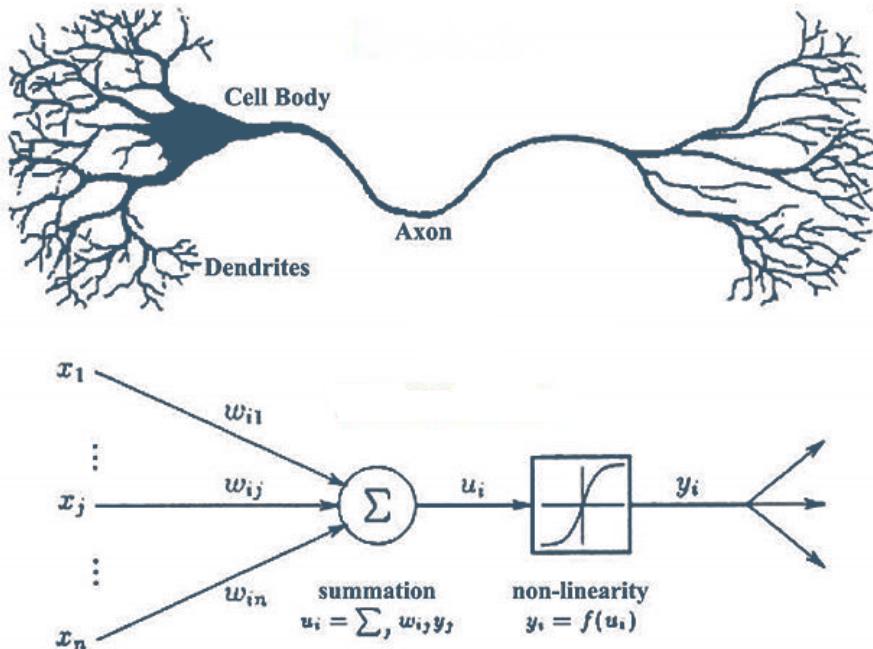


Figura 1.1. Paralelismo de una neurona biológica y artificial

xiones neuronales con el objetivo de adaptarse a su entorno. Durante los primeros años de vida, en el cerebro infantil se crean nuevas sinapsis a una velocidad sorprendente que puede alcanzar hasta las 40.000 conexiones neurales por segundo. Esto le permite al niño conocer su entorno y adquirir nuevos conocimientos a una gran velocidad, el problema es que con el paso del tiempo muchos de esos aprendizajes dejan de ser útiles. De esta manera, en el cerebro infantil se mantienen muchas sinapsis que no son funcionalmente necesarias y que en lugar de facilitar el procesamiento de la información, lo ralentizan.

Para evitar que esto ocurra y garantizar un procesamiento eficiente de la información tiene lugar la poda sináptica, que no es más que el proceso mediante el cual se eliminan las conexiones más débiles entre las neuronas que se crearon durante los primeros años de vida. Básicamente, se trata de un proceso regulador que garantiza una organización sináptica más eficiente, eliminando las estructuras innecesarias del cerebro para aumentar la superficie de recepción de los neurotransmisores. Este desarrollo natural del ser humano mejora el procesamiento de la información. La poda sináptica contribuye a optimizar el funcionamiento de la estructura neuronal

ya que al eliminar conexiones inútiles, favorece la consolidación de una red cerebral mucho más eficiente que permite que se creen nuevas sinapsis con mayor facilidad. Por otra parte, contribuye a la adaptación del entorno. Durante la poda cerebral desaparecen las sinapsis que el niño ya no utiliza para dejar paso a la creación de nuevas conexiones que le permitan adaptarse con más facilidad a su nuevo entorno.

De esta forma, si se pretende emular el comportamiento del cerebro humano, resulta fundamental incluir en los algoritmos algún mecanismo que permita imitar este fenómeno biológico. Las redes neuronales deben ser evolutivas en su entrenamiento, al igual que lo es el ser humano a lo largo de su vida. Deben inicializarse con estructuras generales que permitan aprender rápidamente nuevas tareas y mediante modelizaciones cerebrales que degradan y regeneran nuevas conexiones, permitir a lo largo de su entrenamiento centrarse en tareas más complejas o que requieren de mayor atención eliminando así tareas más simples construidas durante las primeras épocas del entrenamiento. Por lo tanto, el objetivo fundamental de este proyecto consiste en diseñar así una red neuronal evolutiva que trate de replicar este mecanismo natural biológico.

Para materializar esta idea se toma como fuente de inspiración el comportamiento que tienen algunas sustancias de la materia, concretamente los fluidos. Estos no tienen forma propia, sino que adquieren la del recipiente que lo contiene, como se puede observar en la Figura 1.2. En este proyecto se persigue algo similar llevado al mundo de la ciencia de los datos: diseñar una red neuronal cuya estructura evolucione con cada dato de entrada nuevo para así perfeccionar el rendimiento de su tarea de aprendizaje. De ahí sale la idea de FluidNet, una red neuronal con topología evolutiva.

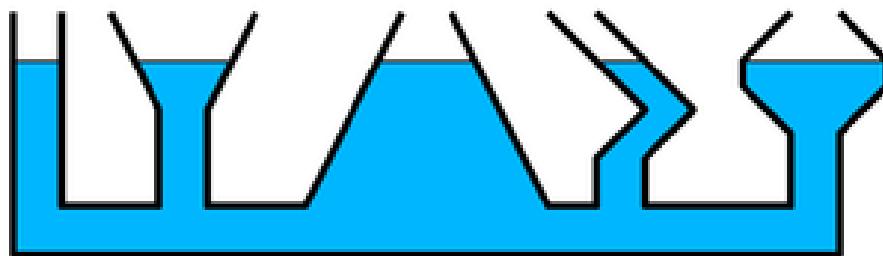


Figura 1.2. Comportamiento de los fluidos

1.2. Objetivos

Por todo lo expuesto anteriormente el trabajo propuesto se enfoca en la consecución de los siguientes objetivos:

1. Diseñar una red neuronal, con topología evolutiva, que sea capaz de ajustar no sólo sus parámetros internos, sino también su estructura de capas con cada nuevo dato de entrada.
2. Mejorar el rendimiento de la red neuronal de capas densas utilizada como benchmark en la tarea de clasificación de patrones técnicos.

1.3. Metodología

A continuación, se presenta el procedimiento desarrollado para la consecución de cada uno de los objetivos descritos anteriormente. Mediante la revisión del escaso estado del arte disponible, se profundizará en las investigaciones que se han desarrollado en este entorno, tanto desde el punto de vista de nuevos avances en el campo puro de las redes neuronales, como en el ámbito de los algoritmos genéticos. Con ello, se podrá estudiar la viabilidad de los distintos avances alcanzados y tomarlos como semillas para poder generar los frutos deseados.

El diseño de la estructura de las redes neuronales ha sido el epicentro de controversias en el mundo del *Deep learning*. Por ello, es necesario conocer los entresijos que se esconden tras la toma de dichas decisiones. Para ello, se estudiarán las bases que sustentan la elección de cada uno de los parámetros y las diferentes vías de proliferación que se han propagado en dicho entorno. Para el desarrollo de la implementación de la red neuronal evolutiva se revisará toda la documentación disponible en TensorFlow para conocer en profundidad el abanico de herramientas disponibles, aunque finalmente se optó por construir de forma íntegra el modelo en cuestión. Asimismo, para el diseño del algoritmo genético se implementarán funciones propias, dotadas de gran flexibilidad, para su adaptación al problema a solventar. Se explorarán diferentes tipos de mecanismos de transmisión genética y se incorporarán en el desarrollo final los que proporcionen los resultados más satisfactorios aplicados sobre el conjunto de test.

En relación con la construcción de la base de datos, se desarrollará de forma integral el proceso de obtención de los datos, preparación y etiquetado, tanto de

las imágenes a clasificar como de las series temporales a tratar. De este modo, se descarta cualquier ineficiencia en el modelo por el sesgo en los datos. Por último, para validar la calidad del modelo aportado, se analizarán los resultados sobre dos conjuntos de datos diferentes, los introducidos anteriormente, y la base de datos del MNIST disponible en TensorFlow.

1.4. Estructura de la memoria

El presente trabajo está configurado atendiendo al siguiente esquema. En el segundo capítulo se presentará cómo se ha construido la base de datos para detectar los patrones del doble suelo, asentando las bases de forma jerarquizada en cada paso dado, desde la propia obtención de las series temporales a partir de *Yahoo Finance*, hasta el etiquetado de las imágenes mediante el uso de técnicas como el Transfer Learning. En el tercer capítulo se expone de forma integral el diseño de la red neuronal evolutiva, aportando no sólo la idea final fructífera, sino todas las alternativas exploradas hasta llegar a la anterior. En el cuarto capítulo se explica cómo se ha diseñado el algoritmo genético que permite orientar la búsqueda de las mejores topologías de cada red neuronal. Finalmente, en el quinto capítulo se analizan los resultados obtenidos y se compara el rendimiento del modelo con una red densa sin carácter evolutivo.

Capítulo 2

Construcción de la base de datos

Una vez que se ha descrito de manera unívoca el problema a resolver conviene explicar detalladamente el proceso para la construcción de la base de datos. Se puede llegar a afirmar que los datos son los verdaderos "sentidos" de la Inteligencia Artificial. De hecho, un estudio de Dataiku ha demostrado que más del 40 % de las empresas encuestadas consideran que la limpieza y preparación de los mismos es la parte más difícil y la que requiere mayor tiempo en un proyecto de IA. Por ello, los datos son el auténtico combustible de la IA, sin ellos, y sin su buen octanaje, el motor de las redes neuronales genera revoluciones carentes de sentidos.

De este modo, en este capítulo se hará una exhaustiva descripción de las fases seguidas hasta la conquista del verdadero valor de los datos buscados. Como se ha introducido en el capítulo anterior, se utilizan dos bases de datos diferentes: una propia y otra de fuente abierta. Para la construcción de la propia base de datos se sigue el tradicional proceso de obtención, preparación y homogeneización de los mismos, considerando su naturaleza de series temporales, y aplicando los ajustes aprendidos a lo largo del máster. Posteriormente se etiquetarán las imágenes y se empleará un algoritmo semi supervisado para generar mayor volumen de datos, mediante el diseño de una red neuronal aplicando la técnica del Transfer Learning. Con esta última fase se daría por concluido la construcción de la base de datos para el algoritmo de detección de patrones técnicos. Por último, se explicarán las técnicas empleadas para manipular los datos ya disponibles para poder comparar la eficacia de la red neuronal evolutiva diseñada. A continuación, se procede a explicar cada una de los pasos previamente descritos

2.1. Obtención de los datos

Se parte de una estructura de series temporales comprendidas entre los años 2011 y 2021 compuesta por 2555 valores que cotizan en cuatro de los cinco mercados de Estados Unidos, concretamente en el Dow Jones, S&P 500, Nasdaq 100 y Nasdaq. La fuente de los datos está proporcionada por *Yahoo Finance* e incluye la información del precio de apertura, cierre, máximo y mínimo de cada una de las sesiones que comprenden el intervalo muestreado. Además, también se incorpora el volumen de las transacciones efectuadas en cada uno de los días. En este escenario se prescindió de la tasa de cambio medida respecto al euro ya que todos los valores cotizaban en la misma unidad monetaria.

Se ha tomado por referencia el mercado americano por el volumen de información disponible en el mismo. Resulta importante matizar un concepto, con relación al **sesgo de supervivencia** en este caso, como el objetivo del TFM no es diseñar un algoritmo de inversión, realmente no importa que haya valores que han desaparecido de los índices anteriores porque no se va a emplear el modelo para diseñar una estrategia de inversión. Como ya se ha comentado, lo importante es tener acceso a un gran volumen de datos para poder construir una base de datos de calidad que permita verificar si la red neuronal evolutiva mejor los resultados de una red puramente secuencial.

2.1.1. Homogeneización y limpieza de los datos

En primer lugar, es importante señalar que todos los activos poseen el mismo número de variables (OHLC), pero sin embargo, el número de filas depende de los datos disponibles según el histórico de cotizaciones presente. Por lo tanto, resulta crucial que todos los valores estén referenciados a los mismos días. Es importante advertir que la cotización de un activo puede no estar disponible debido principalmente a dos razones: la primera, que no existiera o que no estuviera presente en el índice, y la segunda que ya perteneciendo se suspendiera su cotización por algún motivo en concreto. De este modo, se han diseñado dos estrategias diferentes según los escenarios anteriores: si el activo no pertenece aún al índice se rellenan sus campos con NaNs, pero si falta un dato de un día en concreto, se completa con la cotización del día anterior. Por último, es importante matizar que en este caso, el problema de los días festivos y los fines de semana no aplica en este ámbito por pertenecer todas las acciones índices del mercado americano.

Por otra parte, es importante señalar que no se operan con las evoluciones reales de las cotizaciones de los precios, pues contienen demasiado ruido que realmente distorsiona la información. Por ello, se filtran los datos mediante un filtro paso bajo en dos sentidos. El primero de ellos, se realiza para suavizar las tendencias y así poder generar abstracciones más precisas de la volatilidad de los precios. La introducción de dicho filtro genera un retraso general en todas las señales, por ello, para evitarlo se pasa un segundo filtro en la dirección opuesta que contrarreste este efecto secundario.

Por último, como se ha comentado en la sección de justificación del problema, las variables de entrada de la red neuronal son las cotizaciones de diferentes activos en distintos días en una ventana móvil de 40 días. Para garantizar que todos los datos son comparables, resulta esencial manejar algún tipo de normalización. En este sentido se han aplicado dos técnicas diferentes:

- **Ratio de rentabilidad normalizada:** el modelo seguido en este caso se puede formular matemáticamente atendiendo a la siguiente ecuación:

$$O_{N_{pu}}[k] = \frac{O_N[k]}{O_N[k - N + 1]} \times 100 \quad (2.1)$$

$$C_{N_{pu}}[k] = \frac{C_N[k]}{O_N[k - N + 1]} \times 100 \quad (2.2)$$

Donde :

- $C_{N_{pu}}[k]$ y $O_{N_{pu}}[k]$ representan los precios de cierre y apertura.
- $O_N[k - N + 1]$: representa el precio de apertura N días antes.
- $C_N[k]$ y $O_N[k]$ representan los precios de cierre y de apertura en cada una de las sesiones que comprenden el intervalo estudiado.

- **Rentabilidad logarítmica del OHLC:** en este caso, se computa atendiendo a la siguiente ecuación:

$$R_i = \log \left(\frac{P_i}{P_{i-1}} \right) \quad (2.3)$$

Donde:

- R_i : representa el rendimiento logarítmico.

- P_i : representa alguna componente del OHLC en un día en concreto.

Generalmente en el ámbito financiero se prefiere utilizar el segundo enfoque ya que aporta mayor estabilidad en el tiempo, cumplen la propiedad aditiva de los logaritmos y el efecto de la variación del precio es simétrico en ambas direcciones. Estas razones justifican la elección de este ratio para homogeneizar la cotizaciones de los diferentes activos.

A continuación, se adjuntan en las Figuras 2.1 y 2.2 el efecto positivo de aplicar esta transformación logarítmica.

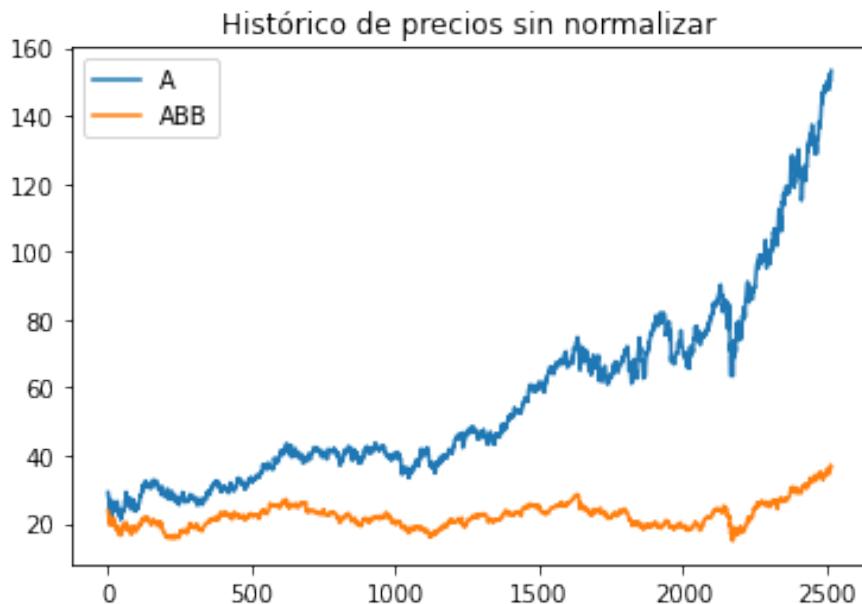


Figura 2.1. Precios sin normalizar

2.1.2. Selección de ajustes aplicados

Una vez que las cotizaciones están limpias, homogeneizadas y normalizadas, resulta esencial revisar los ajustes propios de las series temporales. Esto consiste en detectar si se ha producido algún split o contrasplit en un determinado valor. Es importante corregirlo ya que aunque el precio de la acción cambie la valoración de la empresa es sustancialmente la misma, pues lo único que cambia es el número de acciones en circulación. Para ello, se ha establecido como umbral

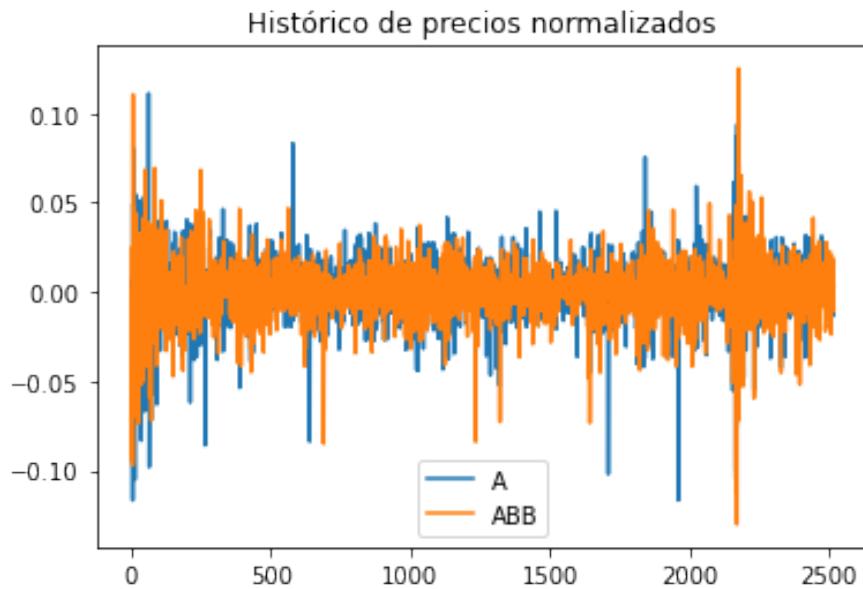


Figura 2.2. Precios normalizados

de corte un 0,69 en el ratio de rentabilidad logarítmica presentada anteriormente. Si dicha variable supera el umbral se asume que el precio se ha duplicado debido a un contrasplit. Por el contrario, si dicha rentabilidad es inferior a -0,69 se establece que se ha producido un split.

2.2. Etiquetado de los datos

Una vez que se han analizado las principales transformaciones que se le han aplicado a la fuente origen de los datos, conviene explicar como se ha construido a partir de la misma, la base de datos que se emplea para alimentar a cada una de las capas de la red neuronal en cuestión. Éstas se enmarcan dentro de la disciplina del aprendizaje supervisado, por lo tanto, esto obliga a tener que determinar cuál es la salida a predecir de la red. En el caso de interés, como ya se ha comentado, se trata de un problema de clasificación en el que el objetivo principal consiste en detectar si una figura determinada tiene o no un doble suelo. Por lo tanto, es necesario etiquetar manualmente el conjunto de series temporales previamente tratadas.

En este sentido, se ha diseñado un bucle de etiquetado que tomando por entrada el número de ejemplos a clasificar, y la longitud de la ventana de la serie temporal a analizar, permite al usuario desarrollar la tarea en cuestión. En este caso, se ha decidido fijar la dimensión de la ventana a dos meses (40 días hábiles). Por lo tanto, en cada iteración del bucle se seleccione aleatoriamente un valor y una fecha, se construye la ventana de precios de los dos meses siguientes y se pregunta al usuario que clasifique si dicho histórico contiene el patrón buscado. A continuación, se muestra en la Figura 2.3 un ejemplo de la misma.



Figura 2.3. Patrón con un doble suelo

Este proceso se ha de completar con absoluta atención pues es lo que determina esencialmente la tasa de acierto final de la red neuronal. Si los datos no se etiquetan correctamente, será imposible que luego el modelo de DL pueda generar cualquier abstracción de la realidad de los mismos y por lo tanto, perderá completamente su capacidad de generalización frente a datos futuros. Para ser consistentes en el criterio de determinación del patrón se han establecido una serie de reglas:

- El hecho de que no se produzca el pullback no es discriminatorio
- Se debe alcanzar el precio objetivo en un plazo mayor a 5 días desde la ruptura de la línea clavicular.

A continuación, se adjunta la Figura 2.4 en la que se visualizan los conceptos introducidos anteriormente.

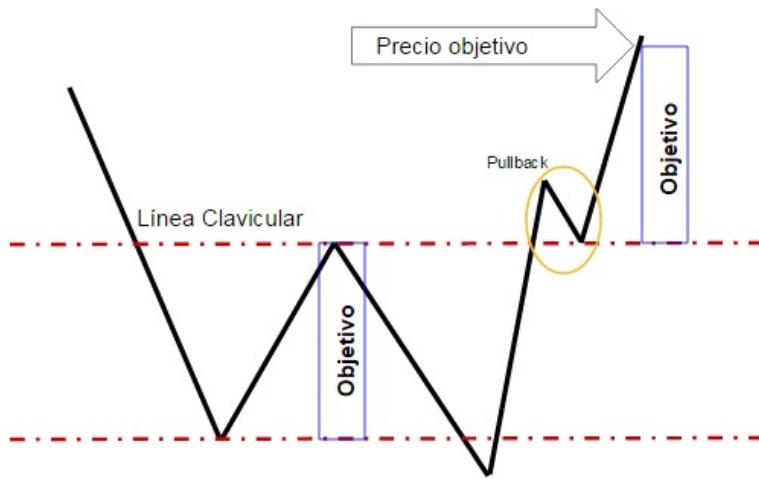


Figura 2.4. Explicación de los conceptos de un doble techo

El principal problema que tiene el DL es que sus modelos deben de ser entrenados con gran cantidad de datos para ajustar correctamente los parámetros internos de los mismos y evitar que se produzca overfitting. Un buen benchmark para evaluar el número específico de datos que es necesario es el número de parámetros entrenables por la red. A mayor complejidad de la misma, mayor volumen de datos es necesario. A continuación, la Tabla 2.1 muestra el número de parámetros a entrenar según se varía la estructura de la red. Ésta sirve como orientación para determinar la cantidad de datos mínima que se debería disponer.

Número de capas	Número de neuronas	Parámetros entrenables
3	128	133,633
3	16	13,121
5	128	166,657
5	88,2	13,665

Tabla 2.1. Parámetros entrenables según la complejidad del modelo

Por lo tanto, en el caso de optar por el modelo de menor complejidad, sería recomendable disponer como mínimo de 13.000 patrones clasificados. Por eso, se

ha decidido etiquetar a mano 5.000 gráficos y para completar el resto de los casos se ha diseñado un algoritmo semi supervisado mediante la técnica del CoTraining. Por último, resulta crucial recordar que el objetivo de este TFM no consiste en predecir si se va a producir un patrón, sino que el enfoque se centra únicamente en saber detectar si en una figura dada ha sucedido un doble suelo o no.

2.2.1. Tratamiento de las imágenes

Como se ha indicado anteriormente, para el desarrollo del algoritmo semi supervisado se va aplicar la técnica de aprendizaje por transferencia que se explicará posteriormente más detalladamente. A modo de breve resumen, el Transfer Learning consiste en aprovechar gran cantidad de información relacionada con la resolución de un problema y utilizarla sobre otro distinto. La idea del aprendizaje por transferencia proviene de un curioso fenómeno que consiste en que muchas redes neuronales profundas entrenadas con imágenes naturales aprenden características similares. Se trata de texturas, esquinas, bordes y manchas de color en las capas iniciales. Estas características de las capas iniciales parecen no ser específicas de un conjunto de datos o de una tarea en particular, sino que son generales en el sentido de que son aplicables a muchos conjuntos de datos y tareas. Estas características estándar encontradas en las capas iniciales parecen darse independientemente de la función de coste exacta y del conjunto de datos de la imagen natural. Esta es la razón fundamental por la que se aplicará esta técnica para la extracción de características de las figuras.

Sin embargo, como se ha incidido al principio del capítulo, resulta esencial que los datos de entrada sean de calidad para que realmente aporten valor. En este caso, se emplean como entradas distintas figuras que representan las velas japonesas de una determinada acción a lo largo de una ventana de 40 días. Para que la red neuronal sea capaz de identificar el patrón buscado, es necesario ajustar el grosor de las velas y su transparencia y centrar las figuras evitando la generación de demasiado píxeles en blanco como fondo. Además, también se ajustan su tamaño para que sean cuadradas ya que no parece intuitivo aplicar una convolución a una imagen rectangular. Por último, es importante quitarle los ejes y la rejilla porque no aportan valor a la tarea en cuestión. A continuación, se adjunta en las siguientes Figuras 2.5 y 2.6 los cambios introducidos de manera visual.



Figura 2.5. Imagen en crudo



Figura 2.6. Imagen preprocesada

2.2.2. Aplicación de Transfer Learning

Tal y como se ha introducido anteriormente, el Transfer Learning es una técnica que permite la reutilización de modelos complejos de DL. Actualmente existen dos diferentes maneras de aplicarlo a un proyecto en concreto:

1. **Feature Extraction:** se congelan ciertas capas del modelo que sirven para extraer características y se añaden nuevas capas que desarrollan la actividad de clasificación deseada.
2. **Fine-Tuning::** se congela el modelo entero y se entrena con un learning rate

más pequeño para que se adapte a los cambios en los datos.

En este caso específico se ha optado por la primera opción, es decir, se emplean las capas convolucionales de la red compleja para la extracción de características de la imagen y posteriormente, se diseña un clasificador acorde con el problema a resolver. Es importante recordar que las capas convolucionales más cercanas a la capa de entrada del modelo aprenden características de bajo nivel, como las líneas y los bordes, las capas del medio aprenden características complejas y abstractas que combinan las características de bajo nivel extraídas de la entrada, y las capas más cercanas a la salida interpretan las características extraídas en el contexto de una tarea de clasificación. Teniendo claro esto se puede elegir un nivel de detalle para la extracción de características de un modelo preentrenado existente. En este caso, como la nueva tarea difiere bastante de el dataset que se empleó para entrenar la red, conviene utilizar solo el nivel inferior e intermedio de capas y luego montar el clasificador.

Una vez que se ha entendido el concepto de Transfer Learning y se ha explicado cómo se va a adaptar al problema en cuestión, resulta interesante presentar la red que se ha utilizado para extraer las características de las imágenes. Se trata de ResNet152 que fue diseñada por un equipo de investigación de Microsoft en 2015 y se convirtió en la ganadora de ILSVRC 2015 en clasificación, detección y localización de imágenes, así como ganadora de MS COCO 2015 en detección y desegmentación. ResNet introduce la conexión de salto (o conexión de acceso directo) para ajustar la entrada de la capa anterior a la siguiente sin ninguna modificación de la entrada. Esta red se entrenó con el dataset de Imagenet que cuenta con cerca de 15 millones de imágenes con alrededor de 22.000 categorías diferentes. ResNet resuelve principalmente el problema del desvanecimiento y/o explotación del gradiente. Son precisamente esas conexiones directas mencionadas anteriormente las que permiten resolver el problema anterior. A continuación, se adjunta en la Figura 2.7 la representación de las mismas.

Se definen tres tipos de conexiones de salto cuando las dimensiones de entrada son menores que la de salida:

- El atajo realiza un mapeo de identidad, con un relleno extra de cero para dimensiones crecientes. Por lo tanto, no hay parámetros adicionales.
- El atajo de proyección se utiliza sólo para dimensiones crecientes, los otros ataños son de identidad. Se necesitan parámetros adicionales.

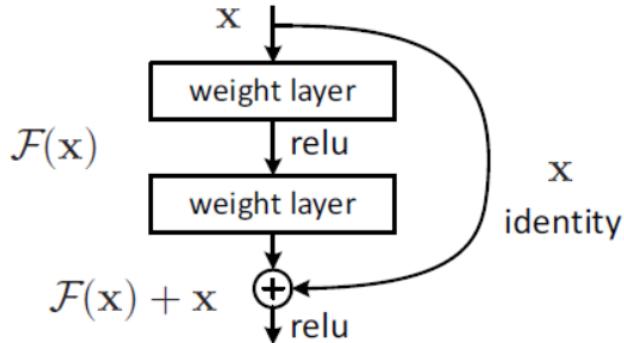


Figura 2.7. Conexión ResNet

Modelo	top-1 err.	top-5 err.
VGG [40] (ILSVRC'14)	—	8,43 [†]
GoogLeNet [43] (ILSVRC'14)	—	7,89
VGG [40] (v5)	24,4	7,1
PReLU-net [12]	21,59	5,71
BN-inception [16]	21,99	5,81
ResNet-34 B	21,84	5,71
ResNet-34 C	21,53	5,60
ResNet-50	20,74	5,25
ResNet-101	19,87	4,60
ResNet-152	19,38	4,49

Tabla 2.2. Comparación con el estado del arte

- Todos los atajos son proyecciones. Los parámetros adicionales son más que los de el punto anterior.

A continuación, se adjunta en la Tabla 2.2 la comparación del estado del arte de distintas redes utilizadas en la competición descrita, justificándose de este modo el empleo de la ResNet152.

Por lo tanto, la red que se utiliza para el algoritmo de Self-Training está diseñada empleando como modelo base la ResNet152 seguida de un clasificador conformado por una capa densa de 10 unidades y la capa final de salida con función de activación softmax. Es importante advertir, que se los datos de partida no están balanceados y por lo tanto, no se puede emplear el accuracy como métrica para valorar la eficacia del método. En las siguientes Figuras 2.8 y 2.9 se muestra la

proporción de datos etiquetados de cada clase.

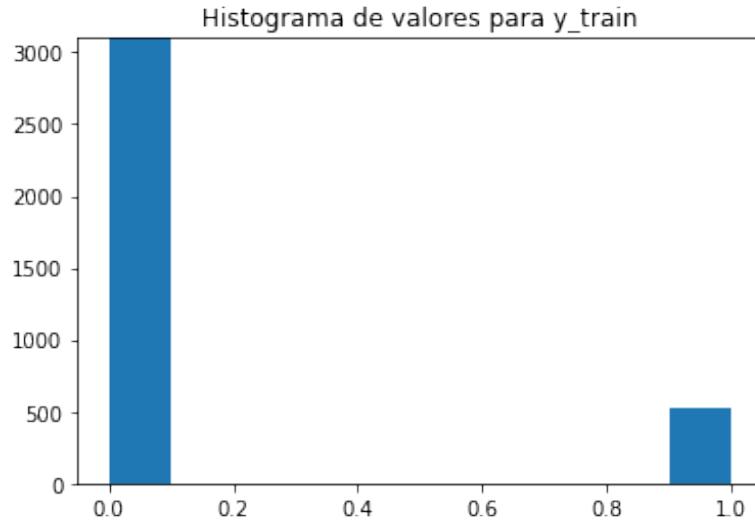


Figura 2.8. Histograma de valores para y_{train}

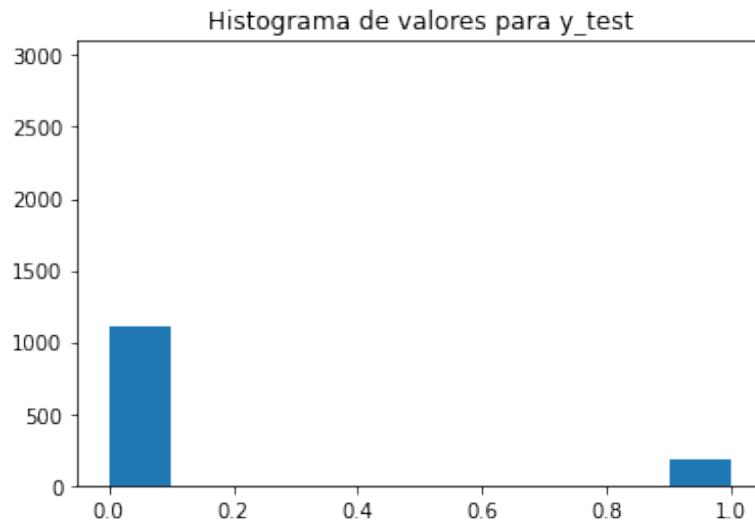


Figura 2.9. Histograma de valores para y_{test}

Por lo tanto en lugar del accuracy se ha construido la matriz de confusión adjunta en la Figura 2.10 para determinar el grado de satisfacción con la solución

encontrada. El entrenamiento se ha efectuado realizando una partición del train y test del 70/30 % respectivamente e imponiendo la restricción de que la clase minoritaria guarde la misma proporción en ambos subconjuntos.

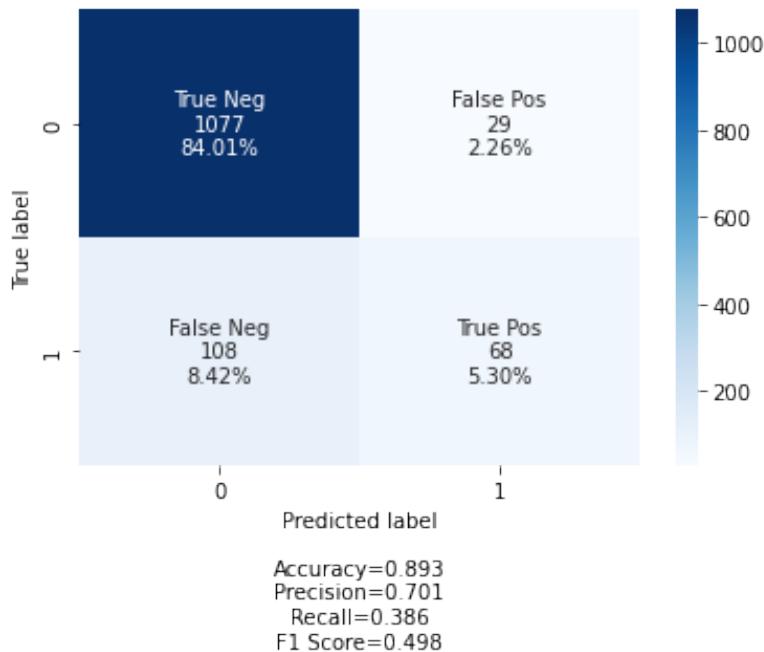


Figura 2.10. Matriz de confusión

A diferencia de las métricas de evaluación estándar que tratan todas las clases como igual de importantes, los problemas de clasificación desequilibrados suelen valorar los errores de clasificación con la clase minoritaria como más importantes que los de la clase mayoritaria. Por ello, se necesitan métricas de rendimiento que se centren en la clase minoritaria, lo que supone un reto porque es en la clase minoritaria donde se carecen de las observaciones necesarias para entrenar un modelo eficaz. Habitualmente, se suele emplear como métrica el F1-score que combina las medidas de precisión y recall en un sólo valor. Esta elección es aceptable cuando las penalizaciones de Falsos Positivos y Falsos negativos tienen el mismo peso. En el caso que se centre más la atención en la precisión, conviene emplear F-Beta score. En el caso que concierne, no hay ninguna penalización extra por el hecho de que el modelo se equivoque en un sentido en específico, esto se debe fundamentalmente a que el algoritmo no se va a emplear como estrategia de inversión ya que no tiene la capacidad de anticiparse al futuro. Sin embargo, es importante matizar que si se

empleará para predecir si va a ocurrir un doble suelo, sería recomendable usar la segunda métrica, ya que en este caso el coste de oportunidad de no invertir cuando sucede un doble suelo es menor que el riesgo que supone hacerlo cuando realmente no hay ningún patrón.

Teniendo estos conceptos claros, conviene analizar los resultados obtenidos. Se alcanza un F1-Score del 0.5 y una precisión del 0.7. No son resultados extraordinarios pero sí aceptables para desarrollar la tarea deseada.

2.2.3. Diseño de un algoritmo de Self Learning para aprendizaje semi supervisado

A diferencia de los dos enfoques más tradicionales (supervised learning y unsupervised learning), los algoritmos semi supervisados emplean pocos datos etiquetados y muchos datos no etiquetados como parte del conjunto de entrenamiento. Dichos algoritmos tratan de explorar la información estructural que contienen los datos no etiquetados con el objetivo de generar modelos predictivos que funcionen mejor que los que sólo utilizan datos etiquetados. Como sucede en el caso de interés, el etiquetado es una tarea ardua de conseguir, requiere tiempo y mucha intervención humana.

La primera fase (descrita en la sección anterior) consiste en entrenar un clasificador, en este caso el modelo de Transfer Learning previamente descrito, con los datos de entrenamiento etiquetados. Posteriormente, el clasificador es usado para predecir los datos no etiquetados y sus predicciones de mayor fiabilidad son añadidas al conjunto de entrenamiento. Finalmente, el clasificador se vuelve a reentrenar con el nuevo conjunto de datos. Este proceso se repite de forma iterativa hasta que se han etiquetado todos los datos disponibles. Teniendo en cuenta que en las condiciones más desfavorables, según lo visto en la Tabla 2.1, se necesitan del orden de 130.000 ¹ parámetros, entonces hace falta que el algoritmo diseñado etique al menos ese mismo número de datos. Por lo tanto, al empezar el porcentaje de los datos es el siguiente:

- 9 % de datos pertenecientes al conjunto de entrenamiento y etiquetados.
- 2 % de los datos pertenecientes al conjunto de test y etiquetados.

¹Por falta de tiempo sólo se etiquetaron 20.000 adicionales con el algoritmo desarrollado

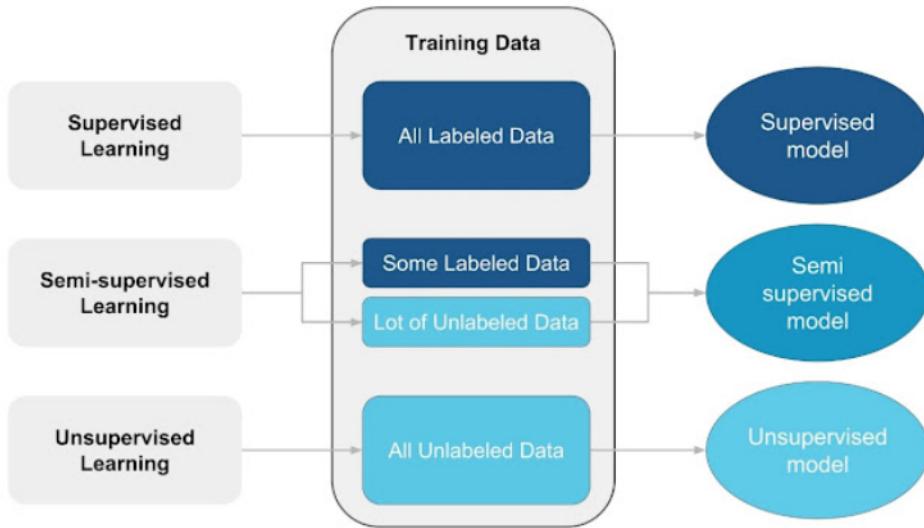


Figura 2.11. Esquema de los distintos enfoques del DL

- 89 % de los datos sin etiquetar.

Para llegar a la totalidad de datos etiquetados se establecerá que en cada pasada el algoritmo debe de etiquetar un ratio de 1:4 de los datos de entrenamiento etiquetados (y pseudo etiquetados) disponibles. Es decir, al iniciar el algoritmo se deberán etiquetar 3750 imágenes. En la siguiente iteración se dispondrá de un total de 18750 imágenes etiquetadas y se deberán etiquetar unas 4686 figuras nuevas. Con este ratio se consigue acelerar el proceso de self learning y además, garantizar que la proporción de datos a etiquetar se mantiene fija en el tiempo.

A continuación, se adjunta el pseudo código del algoritmo planteado.

Algorithm 1 Self-training

```

1: repeat
2:    $m \leftarrow$  train model ( $L$ )
3:   for  $x \in U$  do
4:     if  $\max m(x) > \tau$  then
5:        $L \leftarrow L \cup \{(x, p(x))\}$ 
6:   until no more predictions are confident

```

2.3. Manipulación de los datos

Tal y como se presentó en el capítulo de introducción, para comparar la eficacia de la red evolutiva frente a una red densa común, es necesario manipular los datos para dificultar a los modelos el desempeño de su tarea de clasificación. Es importante advertir que aunque se entrene la red con datos transformados, realmente esto permite a la misma obtener la información desde otros ángulos y perspectivas lo que mejora su capacidad de abstracción frente a datos futuros. A continuación, se presentan las técnicas implementadas en ambos conjuntos de datos.

2.3.1. Estructura de datos del MNIST

El dataset de MNIST es un acrónimo de *Modified National Institute of Standards and Technology dataset*. Contiene 70.000 imágenes de 28x28 píxeles en blanco y negro de dígitos escritos a mano entre 0 y 9. La tarea principal consiste en clasificar las imágenes según el número que representen. Se trata de un dataset que se utiliza como benchmark para modelos del estado del arte de DL. En el caso de interés se procede de forma similar, es decir, para garantizar la bondad del método implementado, se aplica el modelo evolutivo a un dataset fijo y conocido para descartar que la red construida no funcione debido a un problema en la construcción de la base de datos. Además, un aspecto importante a tener en cuenta es que resulta más intuitivo valorar los resultados sobre un dataset que esté balanceado, como es el caso, que en uno que exista una clara clase minoritaria.

En este sentido, para transformar los datos del MNIST se va a aplicar la técnica de *Data augmentation* que consiste en incrementar el volumen de datos añadiendo nuevas instancias de los mismos aplicando ligeras transformaciones, creando así nuevos datos sintéticos a partir de los primitivos. En el caso del preprocesado de las imágenes existen diferentes técnicas a aplicar, entre ellas algunas de las siguientes. Primero se adjunta en la Figura 2.14 el conjunto de datos del MNIST original y en las Figuras 2.13, 2.15 y 2.16 algunas de las posibilidades a desarrollar.

- Aplicar una rotación:
- Aplicar traslaciones verticales y horizontales
- Aplicar zoom

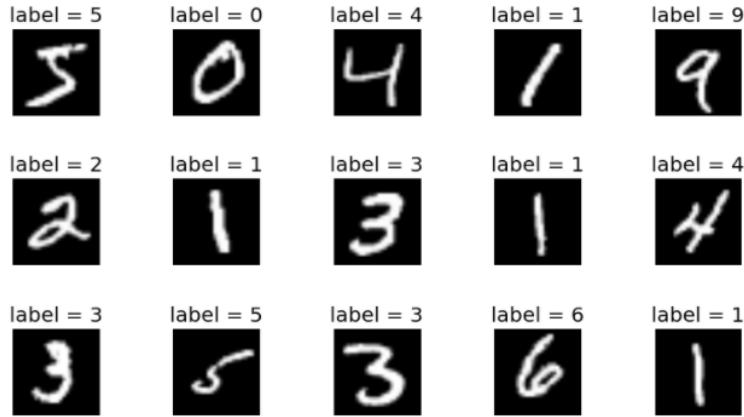


Figura 2.12. *MNIST Dataset*

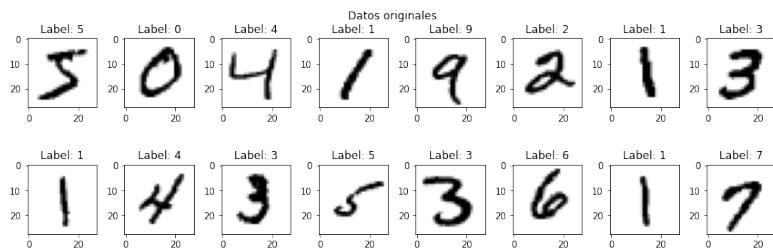


Figura 2.13. *MNIST Dataset original*

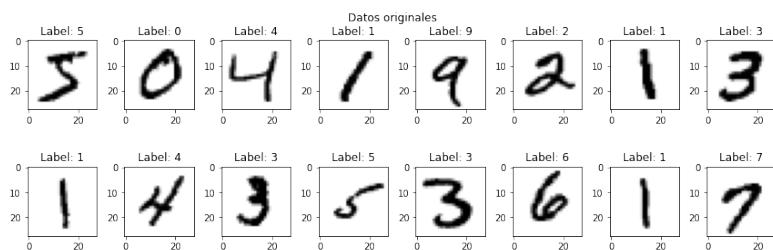


Figura 2.14. *MNIST Dataset con rotación*

Finalmente, se muestra en las Figuras 2.17 y 2.18 la combinación de las técnicas introducidas anteriormente.

De este modo, durante el entrenamiento de la red se alimenta el modelo con datos sintéticos que retan la tarea del clasificador.

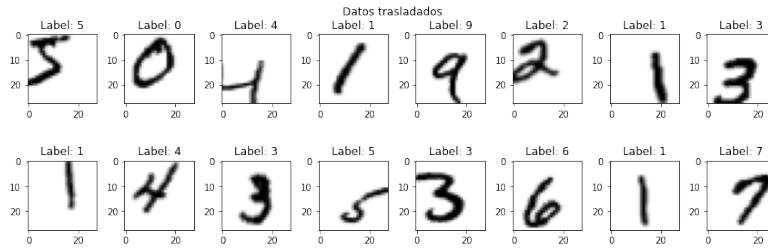


Figura 2.15. MNIST Dataset con translación

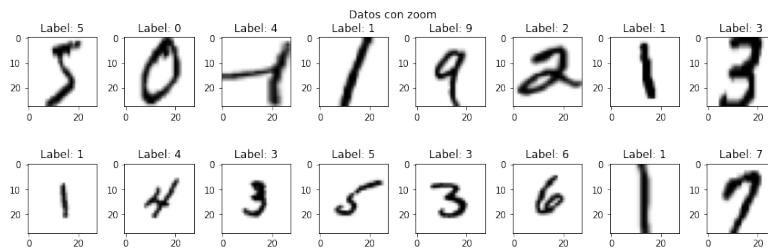


Figura 2.16. MNIST Dataset con zoom

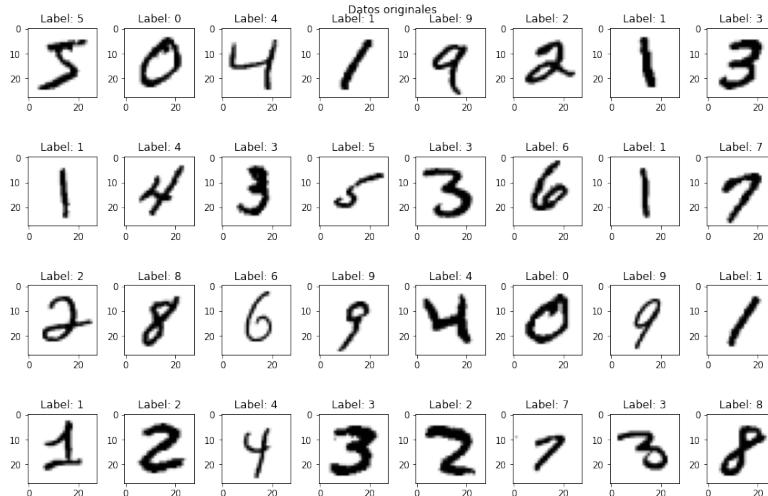


Figura 2.17. MNIST Dataset original

2.3.2. Estructura de datos de las series temporales

En el caso de resolver la tarea de identificar si una figura posee un doble suelo, no tiene sentido aplicar las transformaciones descritas anteriormente. Aunque en ninguno de los dos escenarios se va a trabajar directamente con las imágenes porque

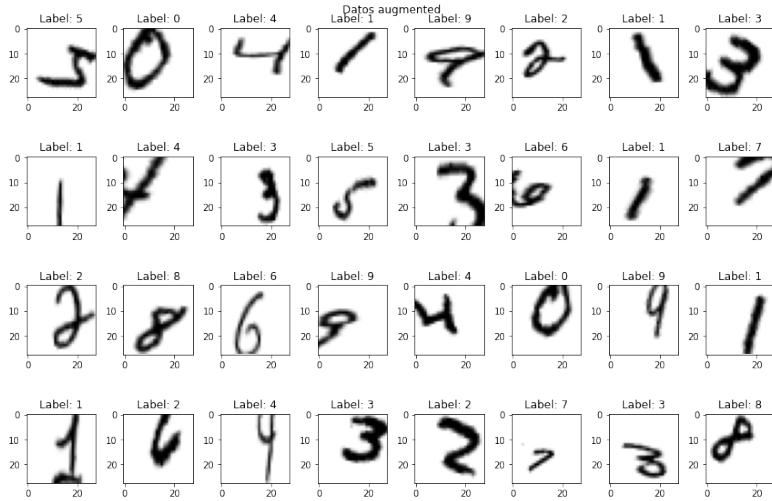


Figura 2.18. MNIST Dataset con data augmentation

se han implementado redes densas y no convolucionales, sí que conviene matizar que en los gráficos pertenecientes al dataset del MNIST se puede observar cierta invarianza (relación de orden constante) entre los elementos contiguos, que no se aprecian en una figura de unas velas japonesas. Además, dado que se introducen directamente como entrada las cotizaciones de un valor en una ventana móvil determinada, no parece muy eficiente transformar dichos valores a imágenes para aplicarles la transformación y posteriormente, tener que descodificarlos de nuevo para convertirlos en cotizaciones. Por ello, se presentan los siguientes enfoques:

1. SMOTE (Synthetic Minority Over-sampling Technique):

Se trata de un enfoque de sobremuestreo en el que la clase minoritaria es sobremuestreada creando ejemplos sintéticos en lugar de hacerlo con sustituciones, esto ayuda al clasificador a crear reglas de decisiones más generales y menos específicas. El algoritmo SMOTE selecciona dos instancias similares utilizando K-vecinos más cercanos y *bootstrapping* y genera muestras sintéticas a partir de instancias de la clase minoritarias, tal y como se puede observar en la Figura 2.19.

Por lo tanto, con la aplicación de este algoritmo se resuelve la tarea de inserción de ruido a las series temporales de los activos y se obtiene como ganancia adicional una mejora sustancial en el balanceo de las clases. Esta fue una de las principales razones por las que se decidió explorar los resultados de es-

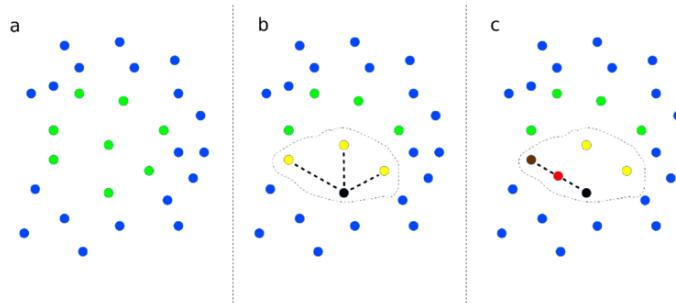


Figura 2.19. Selección de datos de la clase minoritaria con SMOTE

te método. El enfoque es eficaz porque se crean nuevos ejemplos sintéticos de la clase minoritaria que son plausibles, es decir, que están relativamente cerca en el espacio de características de los ejemplos existentes de la clase minoritaria. Una desventaja general del enfoque es que los ejemplos sintéticos se crean sin tener en cuenta la clase mayoritaria, lo que puede dar lugar a ejemplos ambiguos si hay un fuerte solapamiento de las clases, de ahí la importancia de las reglas introducidas para etiquetar los patrones del doble suelo.

En el paper original [23] se sugería la utilización de SMOTE con random undersampling de la clase mayoritaria. De este modo, se ha creado un pipeline en el que se incluyen ambas técnicas y se aplica secuencialmente al dataset original. De este modo, primero se realiza un sobremuestreo de la clase minoritaria para tener un 10 % del número de ejemplos de la clase mayoritaria y, a continuación, se diseña un submuestreo aleatorio para reducir el número de ejemplos de la clase mayoritaria para tener un 30 % más que la clase minoritaria. La razón de la elección de estos ratios no es arbitraria, sino que está para generar un balance adecuado entre generar datos sintéticos de calidad y balancear las clases. El primer objetivo es más importante, de ahí que se fije el porcentaje de undersampling en un valor inferior al habitual (50 %)

Para validar la calidad de los datos generados se incluyen las siguientes gráficas en la Figura 2.20. Se representan la reconstrucción del precio a partir de los datos sintéticos creados. Resulta importante matizar que los datos de partida representaban los rendimientos logarítmicos calculados a partir del

precio de cierre para un determinado valor en una ventana de 40 días. De este modo, para retornar a magnitud de cotizaciones es preciso realizar la transformación inversa, aplicando primero la suma acumulativa y luego una exponencial. En este punto es importante tener en cuenta que al aplicar las diferencias, siempre se pierde el primer dato de partida, y por eso, al reconstruir el histórico se queda en el rango de $[-1, 1]$. Se trata simplemente de un problema de escala que no afecta para nada a la forma del histórico construido, que es lo importante en este caso. Se puede apreciar claramente como en las etiquetas clasificadas como '1' se aprecia un claro doble suelo al principio, lo que pone de manifiesto la precisión del método aplicado.

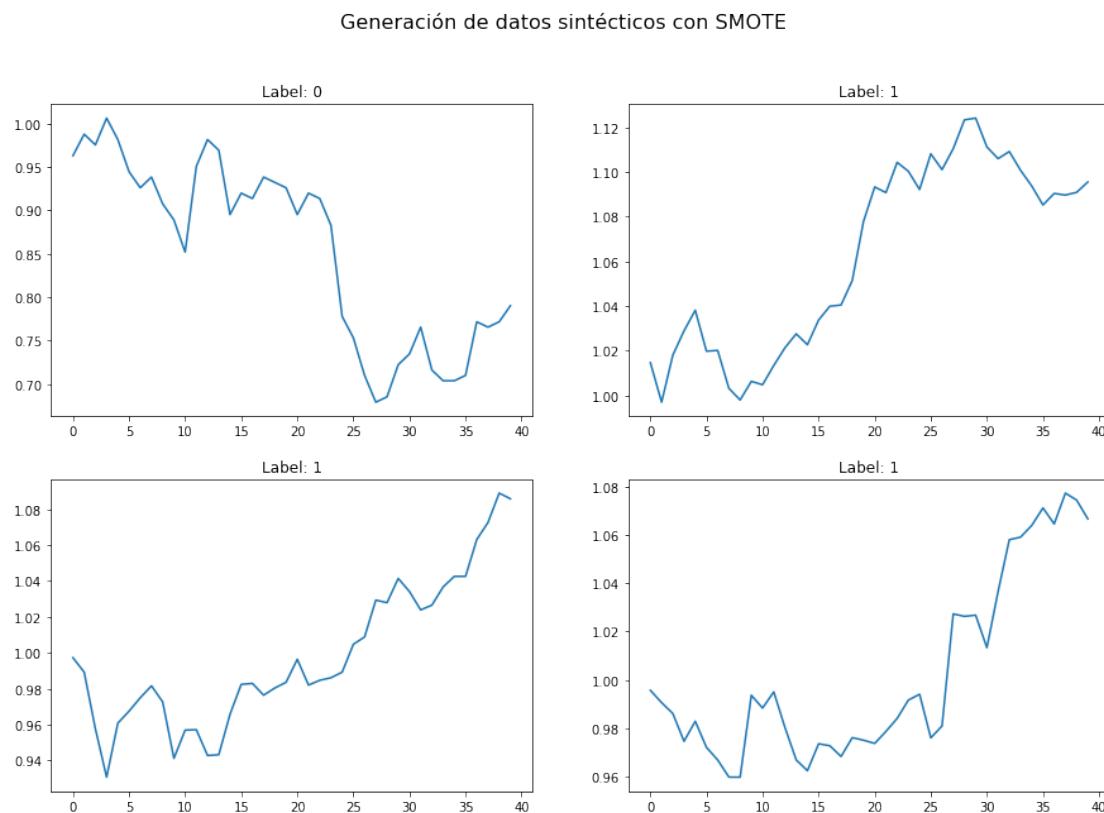


Figura 2.20. Creación de datos sintéticos con SMOTE

Finalmente, se muestra en la Figura 2.21 como se consigue equilibrar las instancias de cada clase tras la aplicación del citado algoritmo.

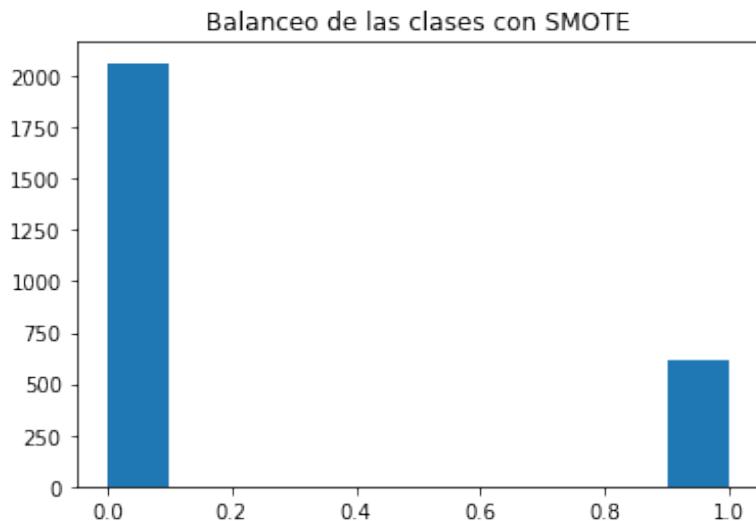


Figura 2.21. Balanceo de las clases con SMOTE

2. **Variational Autoencoder:** Los Variational Autoencoders (VAE) son modelos de aprendizaje que combinan redes neuronales con distribuciones de probabilidad. Su principal uso es el de construir modelos generativos que sean capaces de producir datos sintéticos que siguen los mismos patrones de datos de los que se alimentan. Normalmente, se han empleado para generar imágenes que se asemejan, por ejemplo, características conocidas como caras, vehículos y viviendas. De forma global un modelo generativo se puede entender como un modelo de Machine Learning que tras entrenarse con un conjunto de datos genera otros que se parecen a los de entrada. La diferencia sustancial que introducen los VAE con respecto a modelos generativos puramente al azar es que permiten influir en la dirección específica en la que explorar las variaciones posibles tomando como referencia los datos del conjunto de entrenamiento.

En cuanto a su estructura, un VAE se construye a partir de un autoencoder formado por dos redes: un encoder y un decoder a los que se le añade una función de pérdida que mide la calidad de reconstrucción de los datos de entrada. En esta secuencia el encoder se trata de una red neuronal que transforma sus entradas en una representación interna de dimensión muy inferior a la de entrada (subespacio latente) con el fin de obligarle a aprender una compresión eficiente que extraiga las principales propiedades de los datos de

entrada. Posteriormente, dicha abstracción intermedia alimenta al decoder para tratar de reconstruir la entrada original, tal y como se muestra en la Figura 2.22.

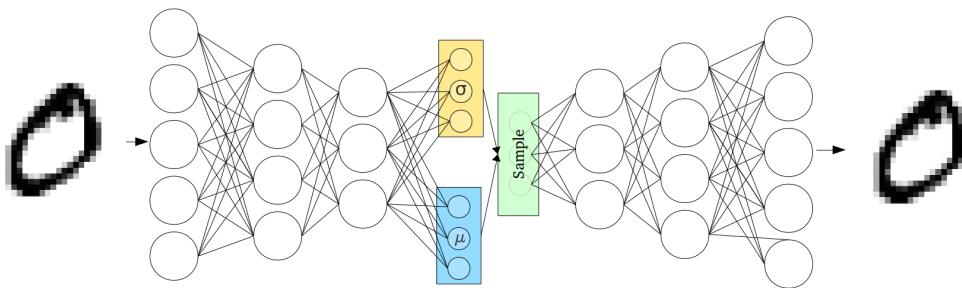


Figura 2.22. Estructura de un VAE

El proceso definido hasta el momento es general de todos los autoencoders, no específico de los VAE. Sin embargo, si no se tienen en cuenta algunos detalles adicionales, los autoencoders generales presentan algunos problemas que no facilitan su uso como generadores de variantes. El principal inconveniente que se ha constatado es que, en la mayoría de los casos, las representaciones internas que se obtienen (la representación en el espacio latente) forman un espacio que no es continuo, sino formado por diversas bolsas aisladas que agrupan en su interior representaciones de datos de entrada similares (clases). La Figura 2.23 pone de manifiesto tal contrariedad. De este modo, como se busca un algoritmo generador, el problema que plantea este hecho es que si el espacio de representación intermedia tiene discontinuidades y se toma una muestra de estas zonas intermedias, entonces el decoder producirá una salida muy poco realista. Los VAE precisamente tienen como objetivo eliminar ese problema construyendo explícitamente un espacio latente que ha de ser continuo, permitiendo generar objetos por medio de la interpolación de representaciones latentes de datos de entrada.

Para conseguir este efecto, un VAE considera que la representación intermedia no viene dada por medio de un vector de tamaño prefijado (la dimensión que se requiera en el espacio latente), sino por medio de dos vectores del mismo tamaño pero con significados bien distintos: un vector de medias, $\vec{\mu} = (\mu_1, \dots, \mu_n)$, y un vector de desviaciones estándar, $\vec{\sigma} = (\sigma_1, \dots, \sigma_n)$, que conjuntamente forman un vector de variables aleatorias descritas por medio

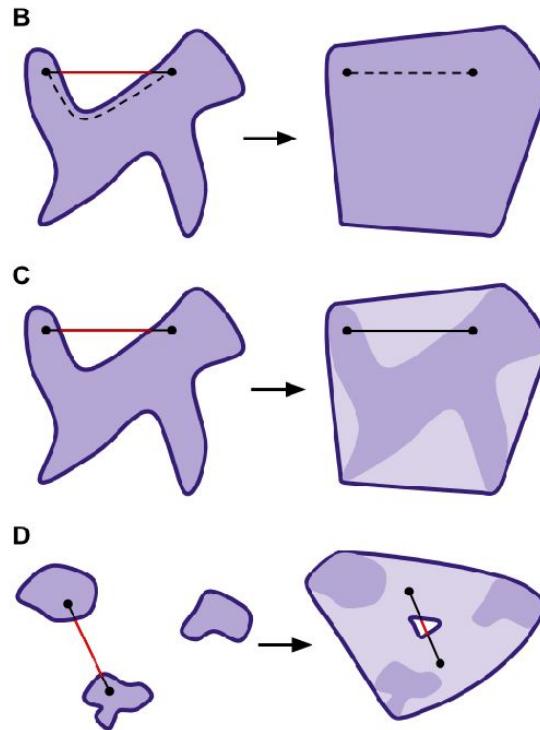


Figura 2.23. Discontinuidad del espacio latente

de distribuciones normales, $(N(\mu_1, \sigma_1), \dots, N(\mu_n, \sigma_n))$.

A partir de este vector de distribuciones normales se samplea un vector que se pasa por el decoder para generar la salida deseada. Cuando se entrena este modelo de forma repetida sobre una cantidad de datos con razonable calidad, el decoder va asociando áreas completas y no sólo puntos aislados como ocurriría con los autoencoders tradicionales. De hecho, la principal diferencia con estos es que en los VAE se puede interpretar el espacio latente, no como un espacio de codificación sino como una distribución de probabilidad. Como consecuencia, es necesario definir una función de pérdida especial que sea capaz de computar la "distancia" entre ambas distribuciones. Con el fin de acometer esta función, se introduce un factor, denominado KL-divergencia (Divergencia de Kullback-Leibler) que, en vez de medir la distancia entre puntos, mide la diferencia existente entre dos distribuciones de probabilidad, aplicando la siguiente fórmula que se añade al término de la función de coste

tradicional. La Figura 2.24 muestra la representación gráfica de dicho cálculo matemático.

$$KL = \sum_{i=1}^n \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1 \quad (2.4)$$

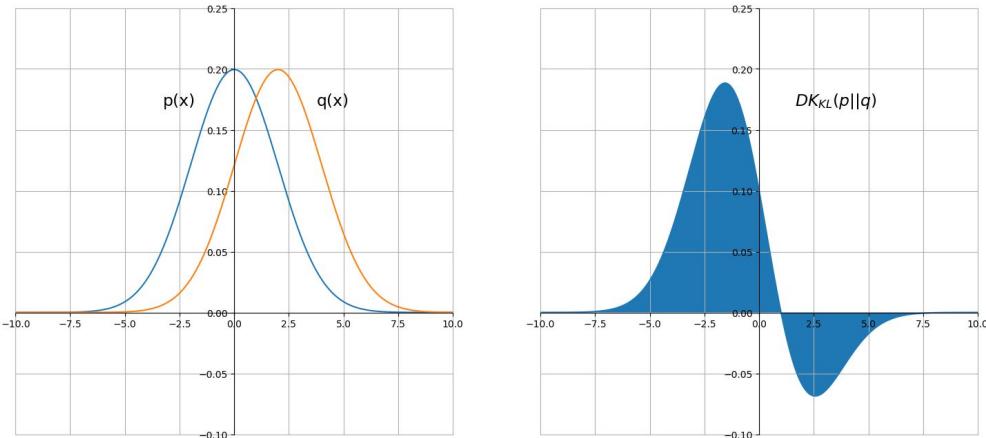


Figura 2.24. *K-L Divergencia aplicada a dos distribuciones de probabilidad*

Una vez que han sido explicadas las bases teóricas que sustentan la justificación de aplicación del método descrito, conviene indicar cómo se ha adaptado el algoritmo a las necesidades del problema en cuestión. El objetivo es generar datos sintéticos plausibles a partir de las cotizaciones de un determinado activo en una ventana compuesta por 40 días. Para ello, se ha diseñado la arquitectura del VAE, tal y como se aprecia en la Figura 2.25. Se pretende hacer una reducción de dimensionalidad de los 40 datos de entrada a 10 para luego poder reconstruirlos. Además, se emplean una combinación simétrica en el encoder y decoder de una capa LSTM de 20 unidades y una densa de 32. La razón fundamental de esta arquitectura es dotar al sistema de memoria pues los datos a tratar pertenecen a serie temporales.

A continuación, se adjunta en la Figura 2.26 la evolución de la función de pérdida en cada época y para validar la capacidad del modelo se generan figuras diseñadas por el modelo, tal y como se procedió en la sección anterior. Los resultados se visualizan en la Figura 2.27.

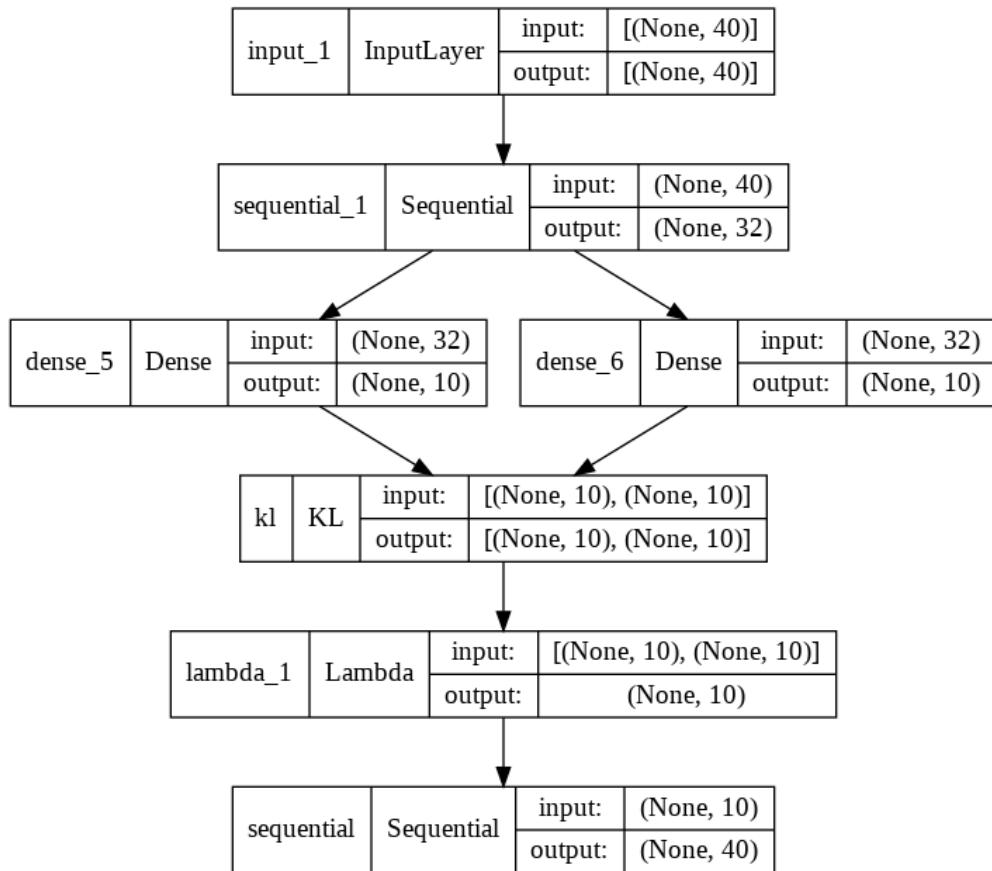


Figura 2.25. Arquitectura del VAE diseñado

A la vista de los resultados presente se puede afirmar que no se obtienen los resultados deseados. La principal causante es la falta de datos para entrenar correctamente el modelo implementado. Por este motivo, se procederá en el futuro con la técnica de SMOTE presentada al principio de esta sección.

A lo largo de este capítulo se ha hecho una revisión exhaustiva de todo el pipeline necesario para construir el dataset que permitirá alimentar la red neuronal diseñada. Se han incluido todas las alternativas que se han ido programando así como sus principales ventajas y desventajas de implementación. A modo de conclusión, se van a trabajar con dos base de datos, una open source correspondiente con el dataset del MNIST y otra de creación propia. En la construcción de esta última una de las tareas más desafiantes es el etiquetado de sus clases, por ello,

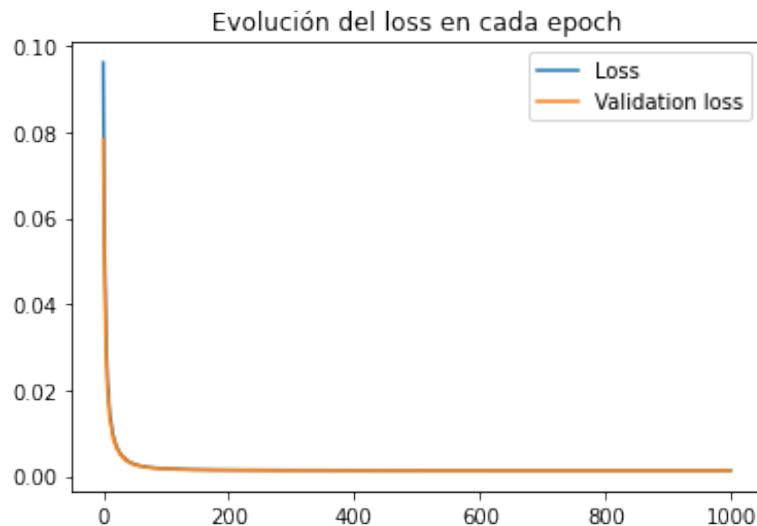


Figura 2.26. Evolución del loss en cada época

se ha desarrollado un algoritmo semi supervisado que complementa el etiquetado manual ya desarrollado. Finalmente, se han expuestos métodos para meter ruidos a los datos originales en los dos escenarios contemplados y como consecuencia de los análisis desarrollados se ha decidido apostar por implementar la técnica del *data augmentation* para los datos del MNIST y el SMOTE para las series temporales.

Generación de datos sintéticos con VAE

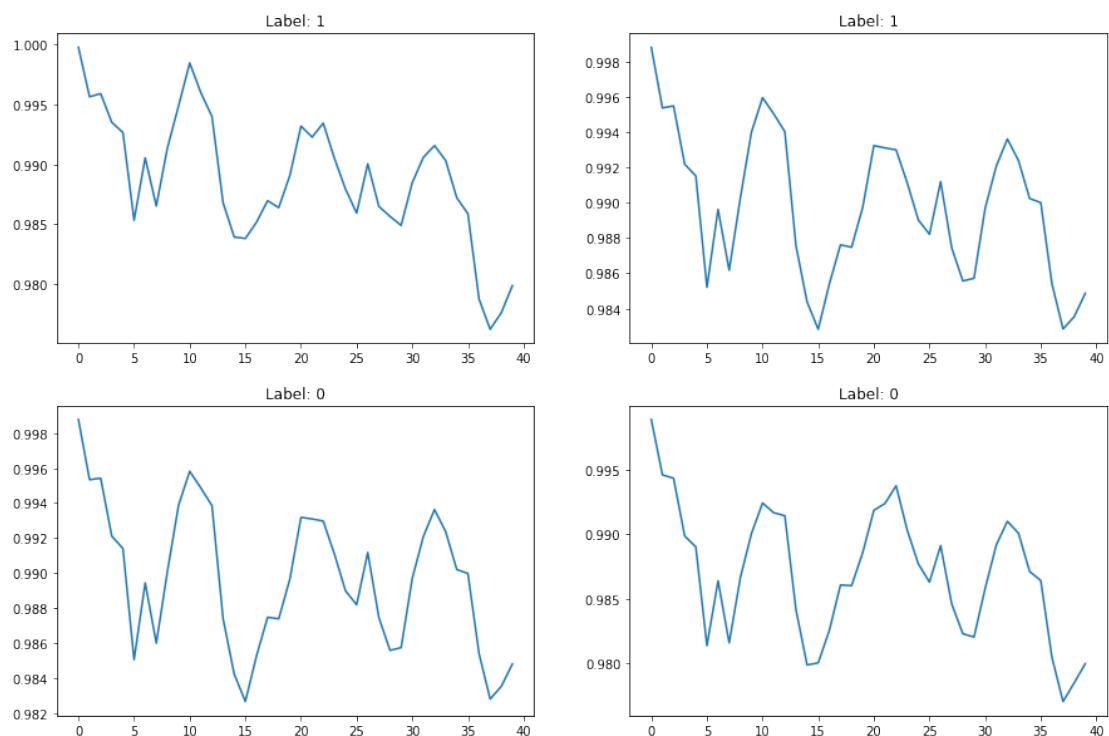


Figura 2.27. Generación de datos sintéticos con VAE

Capítulo 3

Diseño de la red neuronal evolutiva

El objetivo principal de este capítulo es explicar de forma detallada todos los pasos seguidos para la construcción de la red neuronal evolutiva, siendo, pues, el epicentro de este trabajo. Como punto de partida se introducirán las nociones básicas de la arquitectura y el algoritmo interno de las redes neuronales. Posteriormente, se enunciarán las principales dificultades y retos del problema y a continuación, se abordarán las alternativas desarrolladas hasta la implementación de la solución final. De ésta última se hará una profunda revisión de la tarea de aprendizaje a desarrollar, la arquitectura final propuestas y las simulaciones del diseño experimental, así como unas consideraciones adicionales a tener en cuenta.

3.1. Introducción a las redes neuronales

Las redes neuronales son modelos no lineales que aprenden representaciones en un conjunto de datos mediante el empleo de neuronas con numerosas interconexiones y el uso de algoritmos para mejorar su rendimiento. Se componen de un número determinado de elementos denominados *neuronas* que se agrupan en elementos denominados *capas*, éstas son el elemento más básico de una red neuronal y su finalidad principal es aplicar una transformación a los datos recibidos en x con el fin de predecir un valor continuo o categórico \hat{y} en la última capa. Las redes neuronales contienen tres tipos de capas según la posición en la que se encuentren dentro de la red: *input layer*, como capa de entrada, una o varias capas ocultas o *hidden layers* y finalmente la capa de salida o *output layer*. La primera de ellas se encarga de introducir en el modelo la información mientras que las restantes aplican transformaciones necesarias para predecir/clasificar la salida deseada. Por

último, en cuanto a las capas existen también diversas variantes según la forma de interconexión entre sus neuronas y entre ellas. En el presente trabajo se profundizará en el empleo de las capas densas, o *fully connected layers* que, como su nombre indica, está caracterizada por establecer relaciones entre todas las neuronas de las capas contiguas.

En base a un diseño genérico donde hay J neuronas en la capa input, H neuronas en la capa oculta y una neurona en la capa final, se define la función de la red neuronal como:

$$f(x; \theta) = g_o \left(b_o + \sum_{h=1}^H w_{ko} g_k \left(b_k + \sum_{j=1}^J w_{jk} x_j \right) \right) \quad (3.1)$$

donde x_j es un vector que recoge una observación muestral de la variable j -ésima, g_r es una función correspondiente a la neurona r -ésima (k si pertenece a la capa oculta y o si pertenece a la capa final), b_r es el sesgo u offset de la neurona r -ésima y $w_{rs} \in \mathbb{R}$ es una ponderación que transmite un valor de la neurona r a la neurona s . Los parámetros w y b , que pertenecen a θ , se estiman por máxima verosimilitud. Por último, a la función $g_r()$ se le denomina función de activación, y su elección depende de la tarea que se pretenda resolver. En secciones posteriores se discutirá su efecto sobre el rendimiento de la tarea.

3.2. Entrenamiento de una red neuronal

Una vez explicado los elementos básicos que componen una red neuronal, conviene profundizar en la forma de aprendizaje de las mismas. Éste se basa en el cálculo de gradientes mediante el algoritmo de *backpropagation* que se basa en la noción de que como las redes neuronales son funciones compuestas, gran parte de las derivadas parciales de la función de coste respecto de un parámetro cualquier dependen necesariamente de las derivadas que provienen de otros parámetros previos. Dicho algoritmo se compone de dos partes fundamentales: *forward* y *backward* que se explicarán a continuación.

3.2.1. Fase forward

En esta etapa se calcula la predicción de la red neuronal y se computa el valor de la función de coste asociada. Al final de esta fase, se calcula de manera inmediata $\frac{\partial \tilde{J}}{\partial o}$, que es la derivada parcial de la función de coste regularizada (\tilde{J}) con

respecto del output de la última neurona de la red.

3.2.2. Fase backward

El objetivo de esta fase es determinar el gradiente de la función de coste con respecto a cada uno de los parámetros de la red neuronal (w_i y b_i) empleando la regla de la cadena desde la última capa hasta la inicial, y acumulando sucesivamente los gradientes. Consideremos una red neuronal con h_1, h_2, \dots, h_k neuronas en las capas ocultas y una neurona o en la output layer, cuya predicción es utilizada por la función de pérdida \tilde{J} . Adicionalmente, cada ponderación w_{ij} puede ser interpretada como un parámetro que conecta dos neuronas, h_{r-1} y h_r , por lo que el parámetro que une a ambas sería $w_{(h_{r-1}, h_r)}$. Asumiendo que sólo existiese un único camino de o a h_r (es decir, que sólo hay un conjunto único de neuronas entre estas dos a través de las que se transmite la información), se puede calcular el gradiente de la función de pérdida con respecto de cualquier parámetro de la forma que sigue:

$$\frac{\partial \tilde{J}}{\partial w_{(h_{r-1}, h_r)}} = \frac{\partial \tilde{J}}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1}, h_r)}} \quad \forall r \in 1 \dots k \quad (3.2)$$

Para entender de forma visual lo que sucede conviene explicar que la dirección del gradiente es perpendicular a cualquiera de las posibles funciones en J y que está perfectamente alineada con el mínimo global, tal y como se puede observar en la Figura 3.1

3.2.3. Algoritmos de optimización estudiados

A continuación resulta importante explicar los algoritmos de optimización que se han implementado para las redes.

- 1. Stochastic Gradient Descent (SGD):** El stochastic gradient descent (SGD) es una variante del gradient descent y se define como sigue:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \tilde{J}_i}{\partial \theta_t} \quad (3.3)$$

El prefijo stochastic proviene del hecho de que este algoritmo evalúa la función en todas las observaciones muestrales de forma aleatoria antes de terminar el epoch. Una consecuencia de utilizar la función L_i , en lugar de la suma

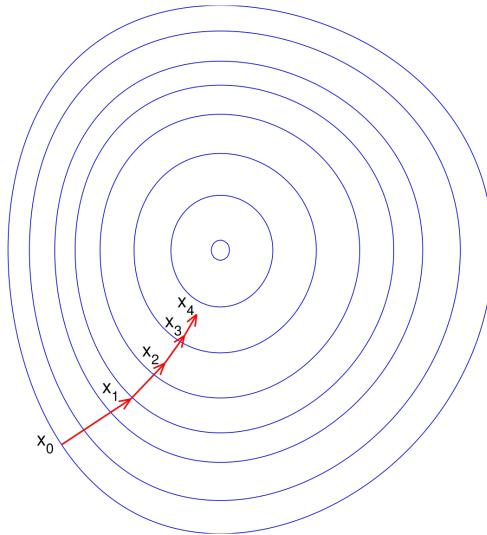


Figura 3.1. Visualización algoritmo descenso por gradiente

total L , es el hecho de que el cálculo del gradiente es menos preciso en cuanto a la estimación de la dirección correcta para la minimización de la función de coste, especialmente en las primeras actualizaciones de los parámetros del modelo (debido a que $\theta_{t=1}$ suele escogerse de manera aleatoria y en \mathbb{R}^n). Sin embargo, a largo plazo (i.e. un número de actualizaciones elevado), la pérdida de precisión respecto del gradient descent suele disminuir, y el menor requerimiento de memoria en el ordenador por parte del SGD ha contribuido a que este algoritmo sea más popular.

Una versión del SGD que permite equilibrar entre uso de memoria y precisión de estimación del gradiente es el mini-batch stochastic gradient:

$$\theta_{t+1} = \theta_t - \eta \sum_{i=1}^m \frac{\partial \tilde{J}_i}{\partial \theta_t} \quad (3.4)$$

donde m es el tamaño del batch aleatorio extraído de una muestra de tamaño N . Las actualizaciones se realizan para un total de $\frac{N}{m}$ submuestras hasta recorrer toda la muestra. Este fue el primer optimizador diseñado en la construcción de las redes evolutivas.

2. **Adam:** El algoritmo Adam se define según la siguiente ecuación:

$$w_i \leftarrow w_i - \frac{\eta_t}{\sqrt{A_i}} F_i; \quad \forall i \quad (3.5)$$

donde A_i es un factor de normalización, η_t es un learning rate ajustado por un corrector, y F_i es un nuevo componente que introduce una media exponencial para el momento de primer orden de $\frac{\partial \tilde{J}}{\partial \theta}$. En el caso de F_i , se tiene que:

$$F_i \leftarrow \rho_f F_i + (1 - \rho_f) \left(\frac{\partial \tilde{J}}{\partial w_i} \right) \quad \forall i \quad (3.6)$$

donde ρ_f es un hiperparámetro para hacer una media exponencial. La finalidad de F_i en el algoritmo Adam es introducir inercia en la estimación del gradiente, de tal forma que le permita al algoritmo superar mínimos locales en el problema de optimización. Las medias exponenciales contenidas en A_t y F_t pueden causar inestabilidad debido a la inicialización de ambos factores en 0 al comenzar la estimación del modelo (esto es, en $t = 0$). Por esta razón, la introducción de η_t pretende corregir este desequilibrio.

Tanto η , ρ como ρ_f son hiperparámetros escogidos manualmente. Para ρ y ρ_f , [40] recomiendan escoger los valores 0,999 y 0,9, respectivamente. El algoritmo Adam ofrece resultados competitivos respecto al otro algoritmo anterior, como se representará más adelante.

Aquí es importante matizar un aspecto crucial. En las ecuaciones anteriores, se inserta el \leftarrow y no $=$ porque al actualizar los pesos no se igualan los valores, sino que se asignan para no perder la trazabilidad y permitir la conexión entre los grafos computacionales de cada una de las capas.

El siguiente paso consiste en explicar como encajan todas las piezas descritas anteriormente. Como se introdujo en el capítulo de introducción, previo al entrenamiento de una red neuronal es necesario fijar su estructura, ésta se congela y permanece constante a lo largo del entrenamiento y únicamente los algoritmos anteriores optimizan los parámetros internos de la misma (w y b). Por lo tanto, en cada iteración, la red neuronal aplica las dos fases descritas anteriormente de *forward pass*, para computar el valor de la función de coste y la derivada de la misma y *backward pass*, para actualizar los pesos de cada capa según el método de descenso por gradiente con cualquiera de los dos optimizadores anteriores.

3.3. Principales dificultades a resolver

Previo al diseño de la red neuronal, resulta crucial entender a fondo el problema que se pretende resolver. Precisamente, lo que se busca conseguir es no tener que predefinir una estructura estática para una red neuronal, sino que ésta se vaya adaptando con cada dato de entrada para así mejorar sustancialmente su capacidad de abstracción. En la actualidad existen diversas técnicas que combinan el uso de redes neuronales con AG, la originalidad de este trabajo consiste en combinar ambos modelos durante el entrenamiento de la red neuronal, y no efectuar una optimización a posteriori de los hiperparámetros, es decir, lo que se pretende es que durante el entrenamiento no sólo se optimicen los pesos de la red, sino también su arquitectura. Para abordar el problema se plantean dos alternativas:

- La primera, consiste en optimizar ambos requisitos simultáneamente con un único AG, es decir, que se encargue de manera síncrona de optimizar el número de neuronas por capa y la complejidad de la red, así como cuantificar los pesos relativos de dichas conexiones con un método diferente al tradicional descenso por gradiente.
- La segunda, consiste en realizar una simbiosis de ambas técnicas, es decir, por una parte que el algoritmo de *backprop* se encargue de optimizar los parámetros de la red, y que de forma paralela el AG trate de buscar la arquitectura óptima.

En el presente trabajo se ha optado por el desarrollo de la segunda opción, existen dos razones fundamentales de peso. La primera de ellas es que el tiempo de ejecución de este tipo de algoritmos heurísticos crece exponencialmente con el número de variables en cuestión. La segunda, consiste precisamente en la forma de codificar dichas variables, de las técnicas tradicionales (binaria, entera o real) ninguna de ellas parece satisfacer los requisitos del problema ya que sería necesario disponer conjuntamente de una máscara binaria que se encargue de seleccionar las neuronas que se activan y las que no, y según lo seleccionado cuantificar las conexiones entre ellas mediante sus pesos. De este modo, el AG diseñado se encarga de ir activando y desactivando neuronas/capas y el algoritmo de *backprop* emplea la técnica del descenso por gradiente para ir optimizando los parámetros internos de cada una de las arquitecturas que se generan en la población de individuos. En este punto álgido resulta crucial indicar que la pieza fundamental de este trabajo consiste en velar por un flujo continuo de información de los pesos de cada una de las redes que se prueban, es decir, en el momento en el que el AG se activa hereda

la información de la red madre que se está entrenando y por lo tanto, toma esos pesos de partida para seguir entrenándose. Este es el matiz que lo diferencia de las otras técnicas disponibles hasta el momento, no obstante, se profundizará en los fundamentos del algoritmo diseñado en el siguiente capítulo.

3.4. Explicación de las alternativas planteadas para la resolución

Como se ha indicado anteriormente la clave de este proyecto consiste en diseñar una red neuronal que permita tener una estructura variable a lo largo del entrenamiento. De forma intuitiva, la forma más sencilla de implementarlo consiste en partir de una estructura genérica que represente las máximas dimensiones del problema, y a partir de ahí, mediante una máscara binaria ir activando/desactivando elementos según las órdenes del AG. Por ejemplo, si se acota el número máximo a 3 capas ocultas de 6 unidades cada una, se tendrían como datos de partida una matriz binaria de dimensiones 6×3 cuyos elementos representan si la neurona de la posición a_{ij} está activada (1) o no (0). Particularizándolos a unos valores en concreto, si se tiene la siguiente matriz M como máscara binaria, el resultado sería el representado en la Figura 3.2. Las neuronas que están en negro simbolizan que están apagadas y que por lo tanto, no transmiten secuencialmente la información. Es importante matizar que lógicamente dicha matriz sólo aplica a las capas ocultas ya que no parece tener sentido no incluir todas las entradas disponibles en el modelo ya que para eso se disponen de otras técnicas de reducción de dimensionalidad más sofisticadas.

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \quad (3.7)$$

Este es el preámbulo que sirve como columna vertebral de las principales alternativas exploradas. A continuación, se explican en detalle cada una de ellas.

3.4.1. Resolución del problema con Keras

Como punto de partida, se intentó desarrollar la red evolutiva bajo el paraguas de Keras, por la facilidad que ofrece a la hora de implementar modelos de DL.

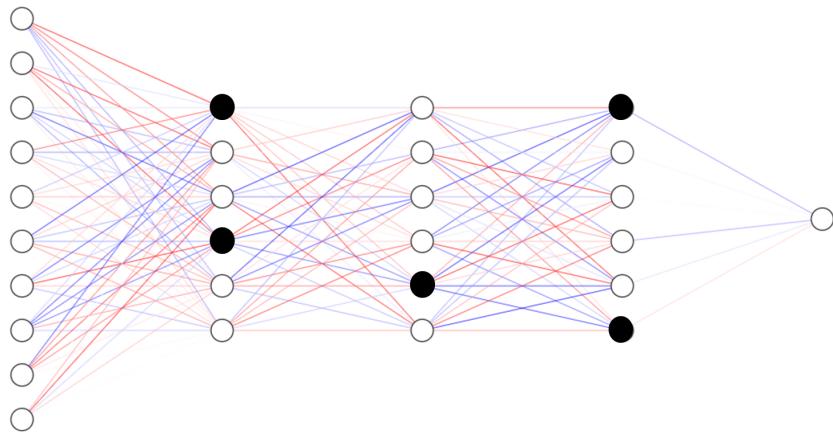


Figura 3.2. Arquitectura de la red neuronal asociada a la matriz M

Sin embargo, esa simplicidad tiene el precio de mermar la flexibilidad del usuario. El objetivo es diseñar un pipeline que permita entrenar una red neuronal madre que sea capaz de transmitir información a otras redes neuronales hijas cuando se active el AG. La pregunta aquí es evidente, ¿cuándo se debe dar luz verde a la actuación del AG? Básicamente, la idea consiste en hacer que el AG actué cuando el entrenamiento de la red superior se está agotando.

Actualmente, Keras proporciona una serie de *callbacks of early stopping* que permiten suspender el entrenamiento de una determinada red neuronal cuando se cumple una cierta condición, aunque no permite explícitamente la implementación de la idea sugerida, se podría construir *callbacks* propios para su desarrollo. Sin embargo, el principal problema que se encuentra aquí es la manera de heredar la información a lo largo del entrenamiento. Por ejemplo, supongamos que se parte de una red neuronal primitiva de 3 capas ocultas con 15 neuronas por capa, llegados a cierto punto del entrenamiento, se detecta que la red está dejando de aprender y ello hace activar el *trigger* del AG, lo que se pretende, pues, es que el AG aporte otras posibles combinaciones de estructuras que hereden la información del entrenamiento y que permitan explorar nuevas topologías y una vez optimizado, terminar el bucle de comunicación, introduciendo la nueva estructura en el entrenamiento de la red de nivel superior.

Teniendo claro el pipeline a seguir resulta crucial introducir el concepto de tensor y la forma en la que se construyen en Tensorflow. Los tensores son la base de

todos los grafos computacionales que permiten conectar los nodos de cada una de las capas de las redes neuronales. Las conexiones parecen triviales, pero en ellas se esconden ciertas dificultades relacionadas con la dimensión de los tensores. De forma genérica cada tensor tiene un nombre, un tipo, un rango y unas dimensiones:

- El nombre identifica de forma única al tensor en estos grafos computacionales.
- El tipo de datos hace referencia a si se trata de, por ejemplo un *tf.float32*, *tf.int64* o *tf.string*
- El rango en el ámbito de Tensorflow, que es diferente al del ámbito de las matemáticas, hace referencia al número de dimensión de un tensor. Por ejemplo, un escalar tiene rango 0, un vector 1 y una matriz 2.
- Las dimensiones o forma se corresponden con el número de elementos dispuestos en cada rango.

Precisamente en la definición de la forma del tensor, si se indaga en los detalles de Tensorflow se observa que se permite la representación de tres tipos diferentes de tensores atendiendo a su forma:

- **Forma totalmente conocida:** son exactamente los ejemplos descritos anteriormente en los que se conoce el rango y el tamaño para cada dimensión.
- **Forma parcialmente conocida:** en este caso, se conoce el rango pero se dispone de un tamaño desconocido para una o más dimensiones. Es por ejemplo, la forma de operar cuando se entrena un modelo de DL por batches.
- **Forma desconocida y rango conocido:** en este escenario se dispone del rango del tensor pero no de las dimensiones de cada rango.
- **Forma y rango desconocidos:** este supuesto es el más opaco porque no se conoce nada del tensor.

Como se puede intuir, el objetivo consiste en crear tensores de forma variable que permitan adaptarse a las diferentes arquitecturas que el AG va proponiendo. Aquí es donde Keras y Tensorflow, en general, presentan su principal limitación: las variables (*w* y *b*) tienen una forma fija, totalmente o parcialmente conocida, a lo largo del entrenamiento. Por ejemplo, si se parte de una red neuronal primitiva que tiene 5 entradas, 2 capas ocultas, con 128 neuronas cada una y una capa de

salida, a la hora de crear el modelo de Keras, independientemente del tipo de capas que se utilicen, Tensorflow creará para cada uno de los pesos de las capas unos tensores de dimensiones, 128×5 , 128×128 y 1×128 respectivamente. El principal problema es que si se desea asignar nuevos tensores de diferentes dimensiones a dichas variables, Tensorflow no lo permite. Es importante aquí recalcar que la operación intrínseca que realiza Tensorflow al actualizar los pesos por el método del descenso del gradiente, es la asignación y no la igualación. Dicha operación toma el tensor original y el nuevo, actualiza el original con el nuevo valor y devuelve la referencia del tensor original. Si por el contrario, se iguala sin asignar, se crea otro tensor constante (que no se puede entrenar) y que por lo tanto, no apunta a la referencia del tensor original. Las siguientes Figuras 3.3 y 3.4 que se generaron en Tensorboard muestran visualmente este fenómeno.

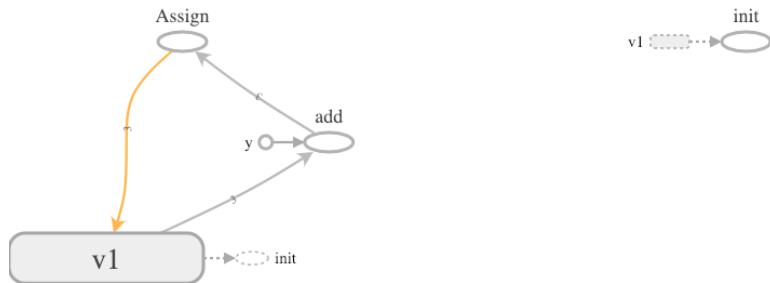


Figura 3.3. Visualización de la operación assign



Figura 3.4. Visualización de la operación de igualar

Esta es la razón fundamental por la que se decidió desarrollar de forma íntegra el diseño de la red neuronal evolutiva manualmente, para poder diseñar pesos con

forma y estructura variable. De aquí la complejidad adicional del proyecto desarrollado al no poder implementar modelos de DL ya establecidos. No obstante, sí que se empleo el marco de Tensorflow para computar las derivadas y los gradientes de forma eficiente, pero la lógica interna del modelo es íntegramente código propio.

3.4.2. Generalización de la idea de Transfer learning

Como ya se ha introducido previamente, la idea fundamental consiste en, a partir de una matriz binaria M poder activar/desactivar neuronas según las arquitecturas que vaya explorando el AG. Una de las piezas fundamentales del Transfer Learning es precisamente dotar a los modelos de la habilidad de poder congelar ciertas capas para, en lugar de tener que optimizar de nuevo los parámetros de las mismas, poder reusar la información aprendida por otros modelos. Una primera aproximación a la resolución del problema, consiste en extender ese concepto de congelar capas al nivel más elemental, es decir, al de las neuronas. De esta forma, si una neurona se bloquea deja de aprender durante el entrenamiento porque sus parámetros, w y b , no se actualizan en ningún momento.

Una vez entendido el procedimiento que se desea implementar se presenta los principales inconvenientes del desarrollo del mismo así como los obstáculos encontrados a lo largo del camino. En la actualidad, Keras permite que ciertas capas completas no se entrenen, sin embargo, no ofrecen dicha funcionalidad a nivel de neurona, que es lo que se pretende conseguir en este ámbito. De este modo, se desarrolló el código propio para simular el entrenamiento personalizado. Este procedimiento consta de dos etapas fundamentales:

1. Inicializar los pesos de las neuronas apagadas a cero, para evitar que tengan impacto en el cálculo de la salida deseada.
2. En cada iteración, forzar que los pesos de dichas neuronas sigan valiendo cero, y de esta manera, evitar el aprendizaje del mismo.

Estas dos reglas son de aplicación si algunas de las neuronas de las capas están apagadas, pero no todas, porque de aplicarlas se perdería la conexión entre los grafos. Por ejemplo, si se establece que el número máximo de capas ocultas es 7 y con 10 neuronas por capa como máximo, el AG en su búsqueda del óptimo podría dar con la siguiente solución potencial, representada por la matriz M :

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \quad (3.8)$$

Tal y como se puede observar en la Figura 3.5 si se aplicara de forma ciega las dos reglas anteriores, en la cuarta capa se perdería completamente la conexión con la salida. Por lo tanto, en este escenario se toma como fuente de inspiración el concepto de residuos introducido en la red Resnet. Es decir, cuando todos los pesos de una capa están desactivados, se pasa directamente la información a la capa contigua aplicando la matriz identidad. El efecto en términos prácticos es la supresión completa de la capa en cuestión, la Figura 3.6 visualiza la estructura de dicha topología en cuestión.

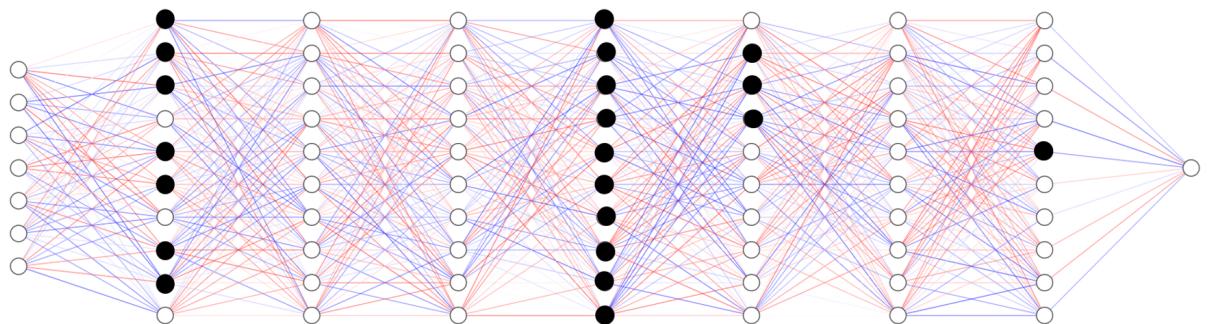


Figura 3.5. Arquitectura de la red neuronal problemática asociada a la matriz M

El principal problema de este método es el concepto de *sparsity*, es decir, que se tengan matrices de grandes dimensiones pero con gran cantidad de ceros y por lo tanto, realmente aporten poco valor. Esto, a parte de no ser eficiente computacionalmente en términos de memoria presenta graves problemas en el cálculo del gradiente. Como ya se introdujo en la sección anterior, los gradientes de las redes neuronales se calculan empleando la regla del *backpropagation*. Para ello, éste

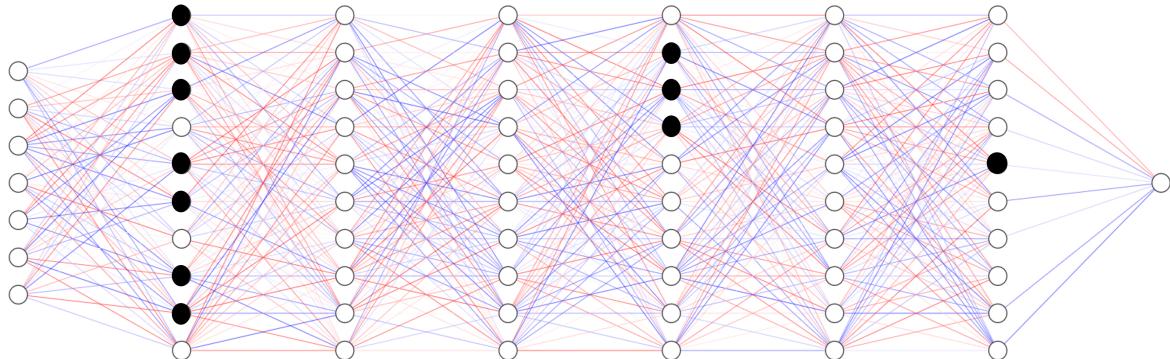


Figura 3.6. Solución de la arquitectura de la red neuronal asociada a la matriz M

calcula las derivadas de la red moviéndose capa por capa desde la final hasta la inicial. Por la regla de la cadena, las derivadas de cada capa se multiplican hacia abajo en la red para calcular las derivadas de las capas iniciales. Sin embargo, al utilizar muchas capas ocultas, las derivadas pequeñas se multiplican entre sí y esto hace que el gradiente disminuya exponencialmente a medida que se propaga hacia las capas iniciales. Un gradiente pequeño implica que los pesos y los sesgos de las capas no se actualizan eficazmente en cada entrenamiento y por lo tanto, ello conduce a la inexactitud general en toda la red. Esto se conoce como el problema del desvanecimiento del gradiente y tras numerosas pruebas se detectó que era lo que ocasionaba la falta en este escenario. Para recalcar el problema de los datos dispersos, si por ejemplo, se desea partir de una estructura general de 10 capas ocultas con 128 neuronas como máximo por capa, se tiene un total de 1280 elementos en dicha matriz binaria. Sin embargo, si una de las soluciones aportadas por el AG consiste en una red neuronal de 2 capas ocultas con 10 neuronas cada una, se tiene que sólo un 1,56 % de la matriz realmente tiene información. De este modo, al forzar al 98,44 % restantes de los elementos a mantenerse estáticos esto acaba finalmente colapsando el paso de información de la red al completo. A continuación, la Figura 3.7 muestra dicho fenómeno, a su vez, también se incluye en la Figura 3.8 el resultado de aplicar dicha transformación a una matriz de pesos determinada.

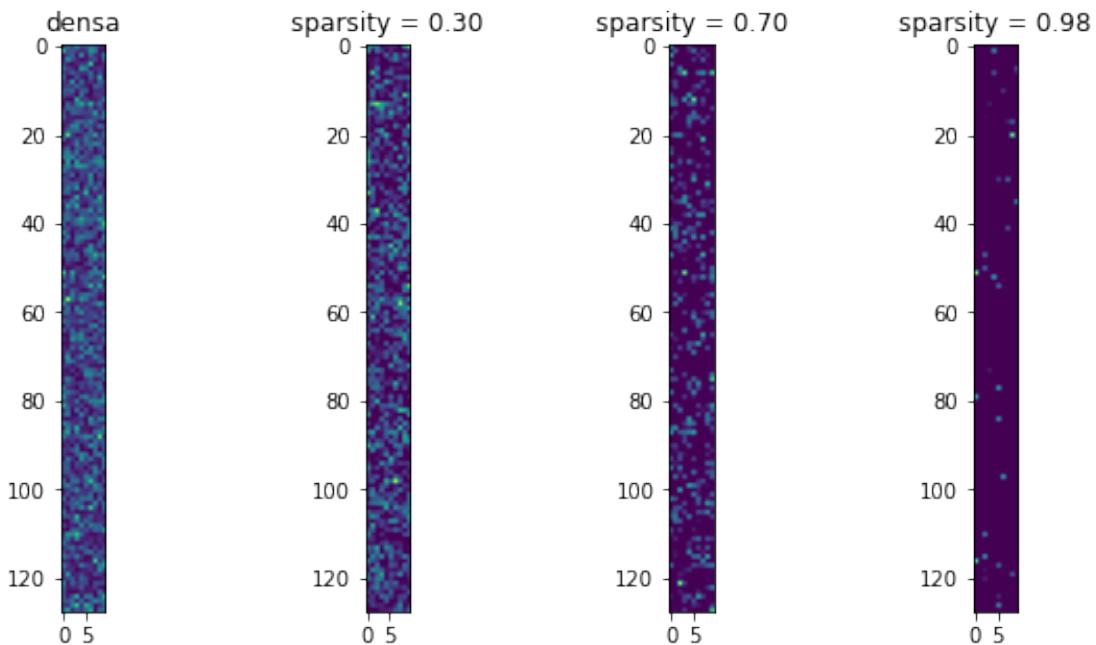


Figura 3.7. Visualización de la matriz binaria según el nivel de sparsity

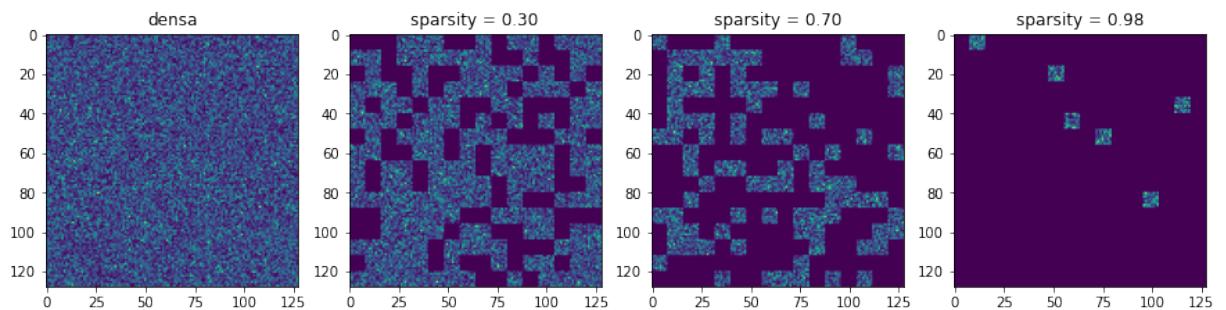


Figura 3.8. Visualización de la matriz de pesos según el nivel de sparsity

3.5. Explicación de la solución final adoptada

Para encontrar las soluciones óptimas es preciso buscar la raíz del problema que se pretende resolver. Las redes neuronales artificiales tratan de emular el comportamiento del cerebro humano, por lo que para abordar este problema desde la perspectiva correcta, lo ideal sería entender cómo funcionan las conexiones en el cerebro, así como su evolución en el tiempo. Por lo tanto, la solución aceptada

parte de tomar a la biología como fuente de inspiración.

Siguiendo esta línea de pensamiento, la poda La poda sináptica es un proceso que tiene lugar en el cerebro y que consiste en la eliminación de conexiones sinápticas entre neuronas. También se conoce como poda neuronal o poda de axones. Este proceso en el organismo persigue eliminar conexiones poco utilizadas para asegurar que la capacidad cerebral está disponible para conexiones utilizadas de forma frecuente. A diferencia de la apoptosis, la neurona no muere sino que los axones con conexiones ineficientes se retractan. El número de sinapsis en bebés y en niños es aproximadamente el doble que en adultos, lo que permite a los bebés aprender rápidamente nuevas tareas a medida que crecen y se desarrollan. El proceso de poda sináptica comienza típicamente durante la adolescencia y continua hasta la edad adulta. Esto hace al cerebro del adulto capaz de centrarse en tareas más complejas o que requieren de más atención eliminando asociaciones simples construidas durante la etapa infantil, aumentando así la eficiencia de las transmisiones neuronales. A continuación, se muestra en la Figura 3.9 el fenómeno descrito.

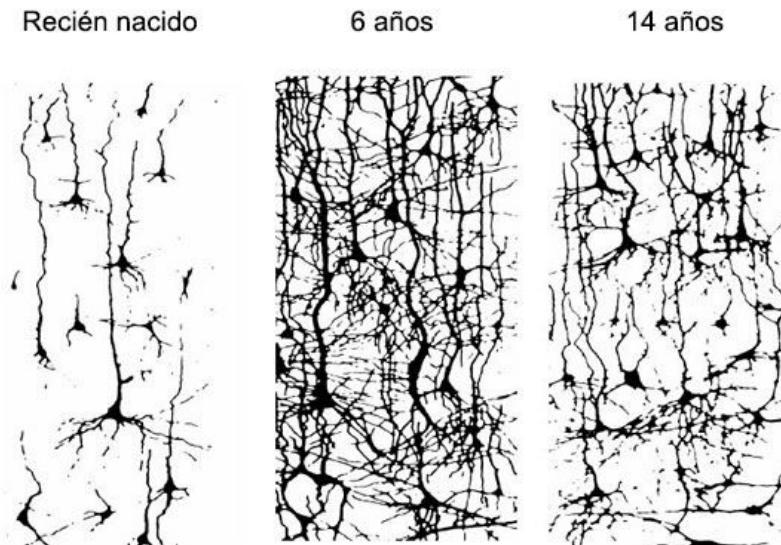


Figura 3.9. Evolución de las conexiones del cerebro

Por lo tanto, lo que se pretende es diseñar una estructura que trate de replicar este comportamiento natural del cerebro. Antes se ha comentado el concepto

de sparsity en los datos y su clara ineeficiencia para el modelo en cuestión. En este sentido, las redes convolucionales se presentan como una clara ventaja para modelar dicho aspecto. Por lo tanto, para resolver el problema se presenta una generalización del uso de Kernels en la aplicación de filtros para la extracción de características de imágenes. En términos generales, la idea consiste en focalizar la atención en las conexiones entre las neuronas y no en las propias neuronas en sí, tal y como lo hace el cerebro de forma natural. En la siguiente sección se explica en detalle cómo se logra implementar una topología dinámica que supere las limitaciones previamente comentadas.

3.5.1. Arquitectura de la red

Como se ha indicado anteriormente, la clave del método consiste en centrar la atención en las conexiones de las neuronas de cada una de las capas. De forma genérica lo que se pretende conseguir es que dada una nueva topología de una red neuronal determinada, ésta sea capaz de heredar la información del entrenamiento de otra red madre. Es importante matizar que el hecho de *apagar* una neurona de una capa i tiene un triple impacto en dichas conexiones:

- Reduce en una unidad el número de filas de los pesos $i-1$
- Reduce en una unidad el número de filas de los sesgos $i-1$
- Reduce en una unidad el número de columnas de los pesos i

De este modo, si por ejemplo se parte de una red neuronal que tiene la siguiente estructura de capas [10, 3, 4, 1], (entrada, ocultas y salida, respectivamente) y se desea pasar la información a otra red neuronal cuya topología es [10, 3, 10, 1], el hecho de aumentar en 6 unidades la segunda capa oculta tiene el siguiente impacto en los pesos, representado en la Tabla 3.1. Además se debe tener en cuenta que no las nuevas arquitecturas pueden diferir en el número de capas ocultas entre sí.

Capa	Dimensión pesos originales	Dimensión pesos nuevos
Entrada	3×10	3×10
Oculta 1	4×3	10×3
Salida	1×4	1×10

Tabla 3.1. Doble impacto de reducir el número de neuronas de una capa

Antes se ha hablado de extender la idea de las redes convolucionales ya que se ha tomado como fuente de inspiración la forma compacta en la que una convolución permite ir reduciendo las dimensiones de las imágenes que se toman por entrada. Se habla de generalización ya que el concepto aplicado no es completamente similar pero, guarda una estrecha relación con el método en el que los Kernels de este tipo de redes aplican los filtros. En el caso de interés, se distinguen tres escenarios globales diferentes: la nueva red contiene el mismo número de capas ocultas, pero diferentes números de neuronas en cada una de ellas; la nueva red posee un menor número de capas y el restante, que la nueva red disponga de más capas intermedias. A continuación, se explica de forma esquemática cada uno de los supuestos anteriores.

Red con el mismo número de capas ocultas

En este caso, la estructura de la red puede verse modificada por la diferencia en el número de neuronas de cada una de las capas. Se parte de una red neuronal con la siguiente topología: [3, 7, 6, 10] y se desea llegar a la siguiente: [3, 5, 8, 10]. Como se puede observar la primera capa intermedia posee menos neuronas, mientras que la segunda posee un mayor número. A continuación, se adjunta en la Tabla 3.2 los pesos esperados para cada una de las redes:

Capa	Dimensión pesos originales	Dimensión pesos nuevos
Entrada	7×3	5×3
Oculta 1	6×7	8×5
Salida	10×6	10×8

Tabla 3.2. Escenario I: red con el mismo número de capas

Para llegar a la solución deseada se acometen los siguientes pasos:

1. En primer lugar, como la nueva red tiene un menor número de neuronas en la primera capa oculta, se aplica una máscara binaria aleatoria que selecciona las filas que se van a heredar. Tal y como se puede observar en la Figura 3.10, se parte de una matriz de pesos de 7×3 y tras seleccionar las neuronas que se quedan activas se obtiene una matriz de 5×3
2. En segundo lugar, se aplica la misma lógica para la capa oculta, teniendo en cuenta el doble impacto ya descrito en los pesos. En la Figura 3.11 se muestra como se modifican las filas de los pesos afectados $i - 1$, mientras

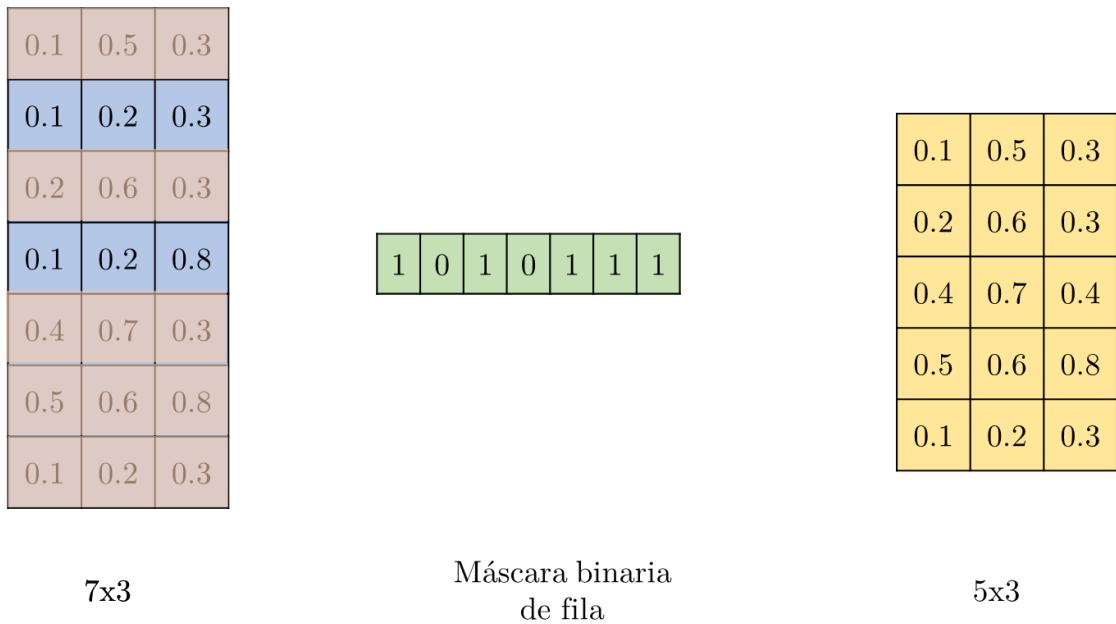
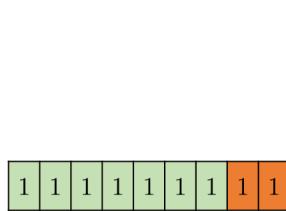


Figura 3.10. Transformación de los pesos de la capa de entrada

que en la Figura 3.12 se visualiza la transformación aplicada a las columnas de la capa i , en este caso, como la nueva topología dispone de un mayor número de neuronas es necesario añadir una columna adicional a la matriz de pesos existentes. Se han contemplado dos opciones posibles, por una parte, la inicialización aleatoria de dichos pesos, tal y como se opera al principio de las redes neuronales, y por otra, muestrear la distribución de probabilidad de los pesos disponibles y obtener los pesos a partir de la misma. Posteriormente, se profundizará en detalle en las dos alternativas viables.

0.1	0.5	0.3	0.1	0.2	0.3	0.4
0.1	0.2	0.3	0.1	0	0.3	0.9
0.2	0.6	0.3	0.1	0.2	0.5	0.4
0.1	0.2	0.8	0.1	0.3	0.3	0.4
0.4	0.7	0.3	0.1	0.2	0.3	0.4
0.1	0.2	0.3	0.1	0.4	0.3	0.4

6x7



Máscara binaria de fila

0.1	0.5	0.3	0.1	0.2	0.3	0.4
0.1	0.2	0.3	0.1	0	0.3	0.9
0.2	0.6	0.3	0.1	0.2	0.5	0.4
0.1	0.2	0.8	0.1	0.3	0.3	0.4
0.4	0.7	0.3	0.1	0.2	0.3	0.4
0.1	0.2	0.3	0.1	0.4	0.3	0.4
rnd						
rnd						

8x7

Figura 3.11. Transformación de los pesos de la capa de intermedia (I)

0.1	0.5	0.3	0.1	0.2	0.3	0.4
0.1	0.2	0.3	0.1	0	0.3	0.9
0.2	0.6	0.3	0.1	0.2	0.5	0.4
0.1	0.2	0.8	0.1	0.3	0.3	0.4
0.4	0.7	0.3	0.1	0.2	0.3	0.4
0.1	0.2	0.3	0.1	0.4	0.3	0.4
rnd						
rnd						

8x7



Máscara binaria de columna

0.1	0.5	0.1	0.2	0.3
0.1	0.2	0.1	0	0.3
0.2	0.6	0.1	0.2	0.5
0.1	0.2	0.1	0.3	0.3
0.4	0.7	0.1	0.2	0.3
0.1	0.2	0.1	0.4	0.3
rnd	rnd	rnd	rnd	rnd
rnd	rnd	rnd	rnd	rnd

8x5

Figura 3.12. Transformación de los pesos de la capa de intermedia (II)

Red con menor número de capas ocultas

En este escenario, la nueva topología de la red neuronal contiene menos capas que la original. En este sentido hay que tener especial cuidado con un aspecto: hay que elegir cuidado qué capas podar. En este trabajo se ha optado por dar preferencia a las relaciones establecidas con las entradas y las salidas, pues son las primeras capas y las últimas las que permiten obtener la información más genérica y más específica, respectivamente. Para ilustrar este mecanismo se parte de una red neuronal que tiene la misma topología que la red primitiva (pero con 6 neuronas a la salida) y que desea evolucionar hasta la siguiente: [3, 10, 6], en la Tabla 3.3 se incluyen los pesos esperados tras la transformación. Como se puede observar se pierde la información de la capa intermedia.

Capa	Dimensión pesos originales	Dimensión pesos nuevos
Entrada	7×3	10×3
Oculto 1	6×7	-
Salida	6×6	6×10

Tabla 3.3. Escenario II: red con menor número de capas

Red con mayor número de capas ocultas El procedimiento para llegar a la solución requerida es el siguiente:

1. En primer lugar, se ajustan los pesos para la capa de entrada, tal y como se puede observar en la Figura 3.13.
2. En segundo lugar, se copian los pesos de la últimas capas para obtener los parámetros de la capa de salida. Dicho resultado se visualiza en la Figura 3.14

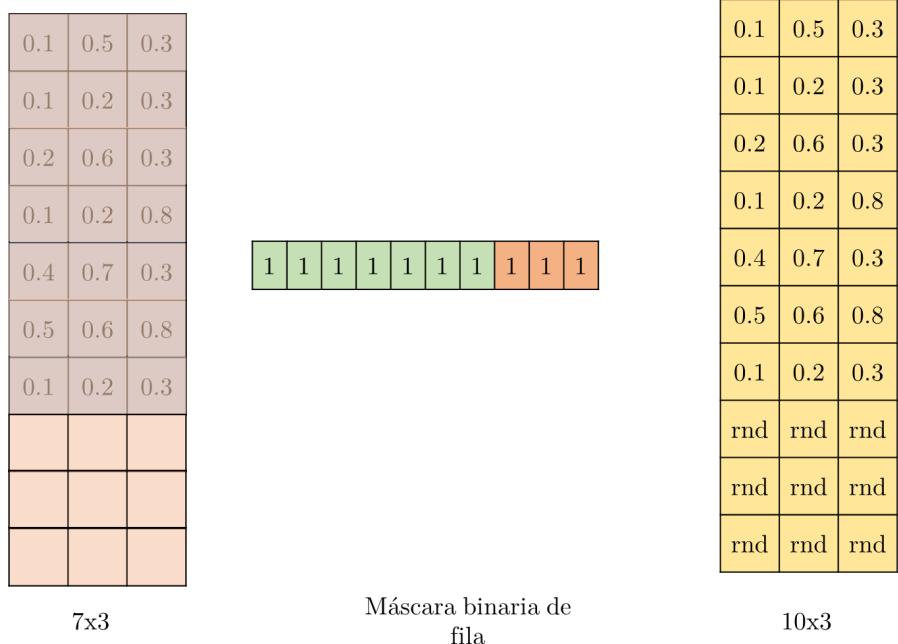


Figura 3.13. Transformación de los pesos de la capa de entrada

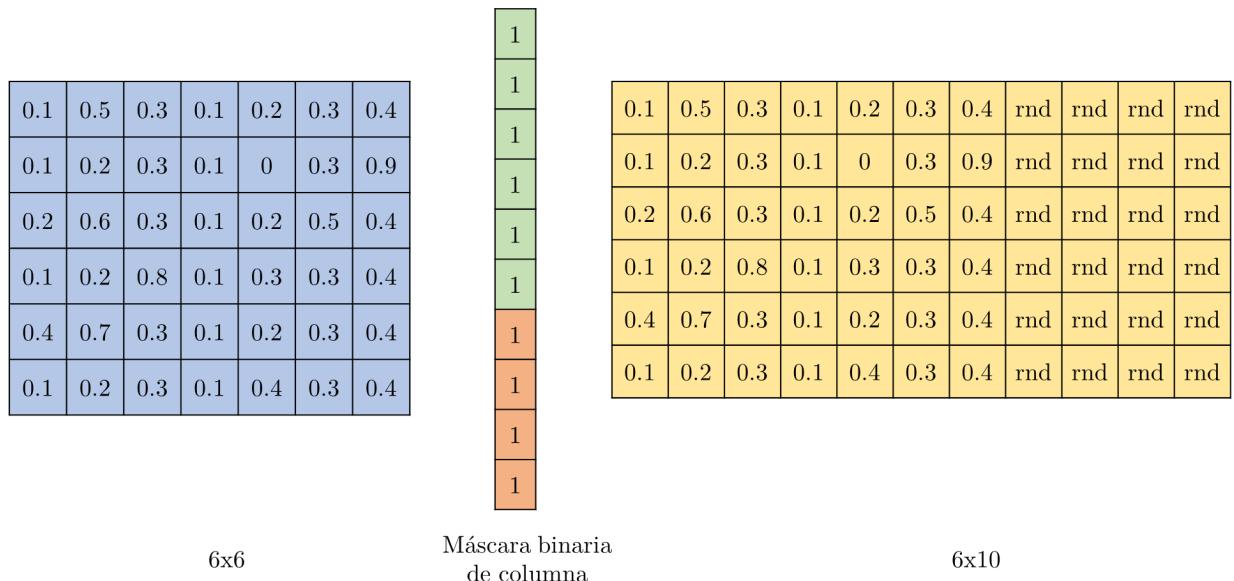


Figura 3.14. Transformación de los pesos de la capa de salida

Red con mayor número de capas ocultas

Por último, el tercer caso a explorar es cuando la red que evoluciona dispone de un mayor número de capas ocultas que la primitiva. En este supuesto, al igual que en el anterior, es de especial importancia la forma en la que se deciden inyectar nuevas capas intermedia a la solución final. En este escenario se ha optado por hacerlo en las capas ocultas, sin afectar ni a la capa de entrada ni a la de salida. Es decir, dicha adición de complejidad no se efectúa de forma secuencial sino que se congela la información de la capa de salida y se añaden nuevas conexiones a las capas ocultas. Para ilustrar este supuesto se parte de la misma red que en los casos anteriores y se desea llegar a una estructura con las siguientes capas: [3, 5, 8, 8, 6]. Tal y como se ha procedido en los casos anteriores, se adjunta en la Tabla 3.4 las dimensiones de los pesos tras las transformaciones aplicadas.

Capa	Dimensión pesos originales	Dimensión pesos nuevos
Entrada	7×3	5×3
Oculta 1	6×7	8×5
Oculta 2	-	8×8
Salida	6×6	6×8

Tabla 3.4. Escenario III: red con mayor número de capas

De este modo el procedimiento seguido en este caso es el siguiente:

1. En primer lugar, los pasos para ajustar los pesos de la capa de entrada y la capa oculta 1, son semejantes al procedimiento desarrollado en la primera sección.
2. En segundo lugar, para insertar una nueva capa, basta con aplicar alguna de las transformaciones previamente enunciadas (muestreo de distribuciones o inicialización aleatoria). En este caso, tal y como se puede observar en la Figura 3.15 se opta por la segunda opción.
3. Por último, en la Figura 3.16 se adjunta cómo se actualizan las dimensiones de los pesos de la capa de la salida.

Para poder desarrollar toda estructura resulta imprescindible entender las siguientes líneas de código. Los pesos y los sesgos se guardan en diccionarios que cuelgan de la clase principal *FluidNetwork*, la clave está en que para una de las capas iniciales durante el setup crearlas mediante variables cuya forma no esté definida, para

rnd							
rnd							
rnd							
rnd							
rnd							
rnd							
rnd							
rnd							

8x8

Figura 3.15. Creación de una capa intermedia de pesos

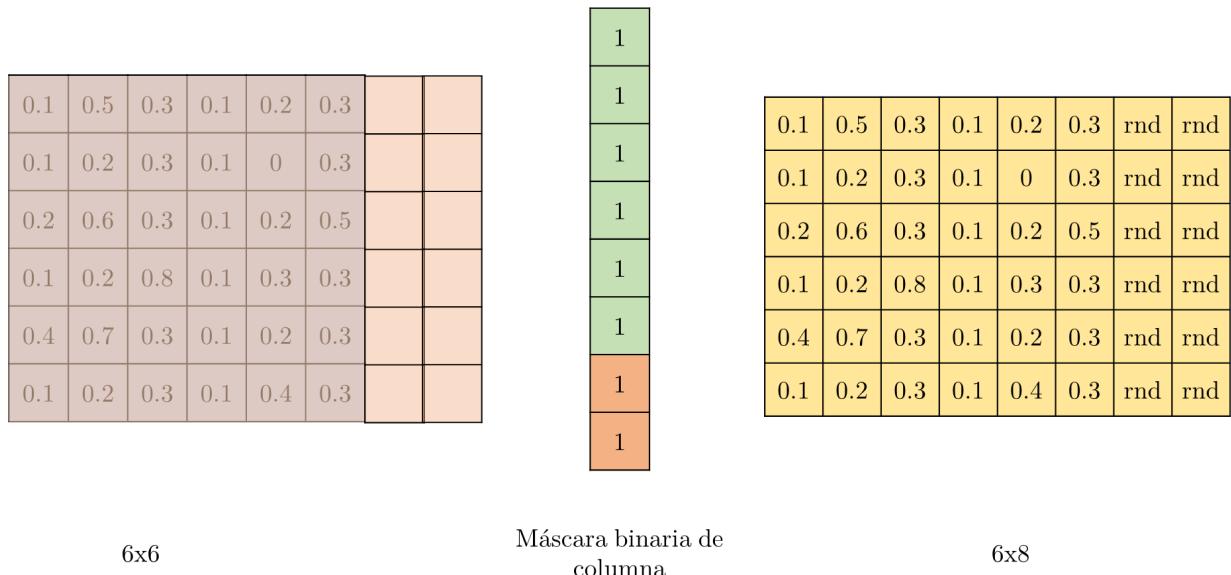


Figura 3.16. Ajuste de los pesos de la capa de salida

así poder implementar modificaciones futuras sin dañar las conexiones entre los grafos. Por ello, la piedra angular de este proyecto es precisamente la función que partiendo de una estructura determinada de una red neuronal es capaz de adaptar

sus parámetros internos a otra estructura objetivo, de aquí, el nombre de *FluidNet* ya que la red va adoptando una forma u otra según las solicitudes requeridas.

```

1 self.W[i] = tf.Variable(1.0, shape=tf.TensorShape(None))
2 self.b[i] = tf.Variable(1.0, shape=tf.TensorShape(None))
3 self.W[i].assign(tf.Variable(tf.random.normal(shape=(self.layers[i],
4                                         self.layers[i-1]))))
5 self.b[i].assign(tf.Variable(tf.random.normal(shape=(self.layers[i],1))))

```

Como se ha comentado anteriormente se emplea como fuente de inspiración los avances desarrollados en las redes neuronales convolucionales. En este tipo de redes existe un parámetro denominado *padding* que se emplea para llenar los bordes de las imágenes con ceros para poder aplicar los filtros de los kernels. A continuación, la Figura 3.17 presenta de manera visual la forma de operar.

En este proyecto se debe de proceder de forma análoga cuando se tiene mayores

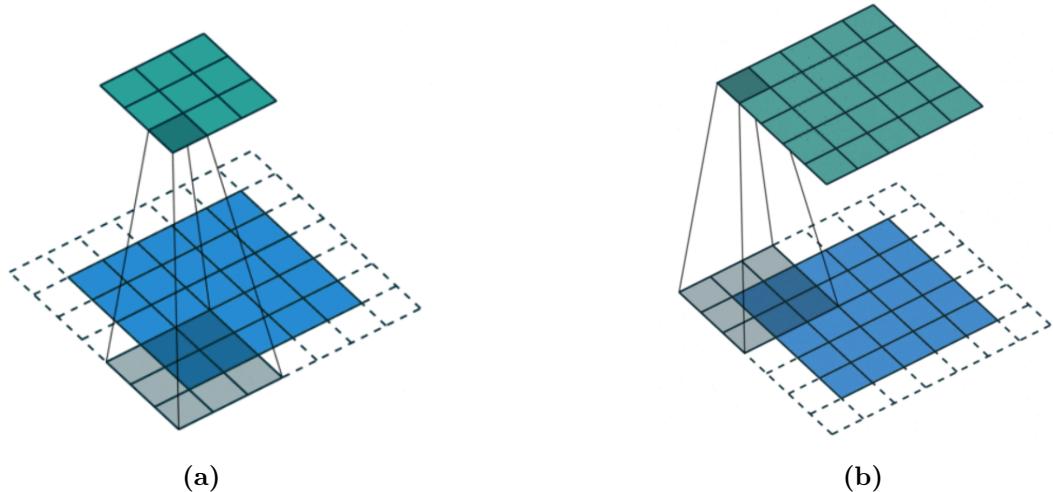


Figura 3.17. Aplicación del padding

dimensiones en la nueva topología en comparación con la primitiva. Para ello, como se introdujo brevemente al principio de esta sección, se han desarrollado seguido dos técnicas, la proporción de pesos obtenidos según el enfoque es uno de los hiperparámetros del modelo.

- Inicializar los pesos de los nuevos huecos aleatoriamente

- Muestrear la distribución de los pesos antiguos y obtener los pesos a través de la misma

Ambas estrategias presentan ciertas ventajas e inconvenientes. Básicamente representan la clásica confrontación entre la explotación y la exploración de nuevas soluciones. Con respecto a la inicialización aleatoria, La principal ventaja que ofrece es su rapidez de ejecución, además, con esta estrategia se fomenta la exploración del AG en busca de nuevas soluciones. Sin embargo, no dejan de ser datos introducidos de forma sintética sobre una matriz de pesos que previamente se estaba optimizando. Por lo tanto, esto puede generar saltos repentinos a lo largo del entrenamiento de las redes neuronales hijas, ya que heredan información de la red primitiva que se manipula con datos completamente nuevos que no tienen ninguna relación con los parámetros internos que se han ido actualizando mediante la técnica del descenso por gradiente. La Figura 3.18 muestra este fenómeno que ocurrió durante el entrenamiento de la red neuronal evolutiva. En este escenario, para llenar los huecos de los pesos se implementó únicamente la inicialización aleatoria, como se puede observar, al recibir la red valores totalmente aleatorios, pierde completamente la exactitud y se vuelve muy ineficiente.

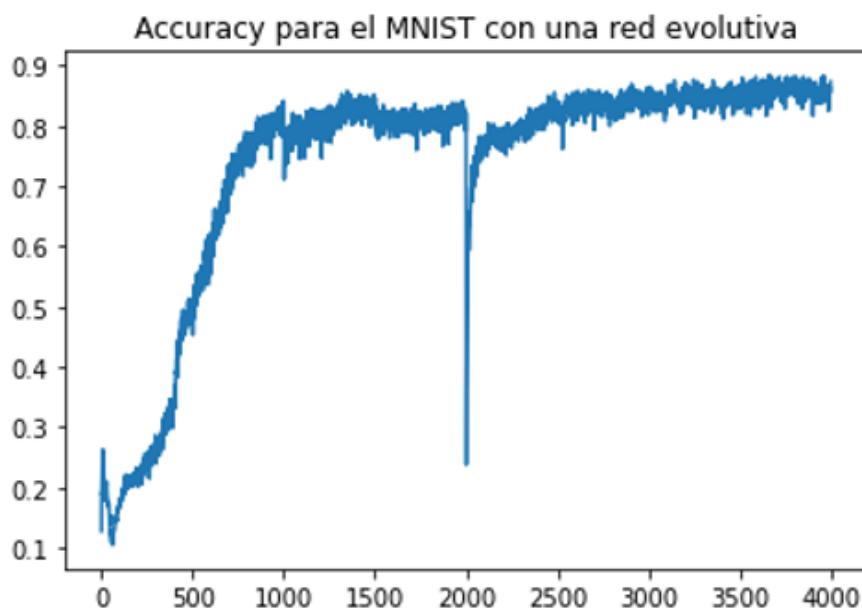


Figura 3.18. Pico abrupto en el accuracy durante el entrenamiento

Para encontrar la mejor distribución que encaja con los pesos anteriores se

emplea el módulo *Fitter* de Python. Esta clase en el backend utiliza la librería *Spicy* que permite explorar cuál es la mejor distribución (de entre las 80 disponibles) que se ajustan a unos datos determinados. Esto incrementa considerablemente el tiempo de ejecución, por ello, se decidió incluir únicamente las distribuciones de probabilidad más comunes:

- Cauchy
- Chi-cuadrado
- Exponencial
- Exponencial con potencia
- Gamma
- Lognormal
- Normal
- Rayleigh
- Uniforme
- Ley de potencia

Al instanciar el objeto e invocar el método `.fit()` se obtiene el ranking de las 5 mejores distribuciones de probabilidad que se ajustan a los datos de entrada. Hay diferentes métricas para ordenar, en este proyecto se ha utilizado el Criterio de información bayesiano (BIC). Para comprobar el correcto funcionamiento del método presentado, se muestreó la distribución de probabilidad de los primeros pesos de la red neuronal que se inicializan de forma aleatoria. Tal y como se puede observar, en la gráfica adjunta en la Figura 3.19 se ajustan perfectamente a la distribución normal esperada de media 0 y desviación estándar 1.

Por otro lado, también se adjunta en la Tabla 3.5 el resumen del resultado del modelo, así como los scores para las mejores distribuciones encontradas. Una vez que se determina la distribución que mejor se ajustan a los datos y sus parámetros, se puede tomar una muestra de la misma de una dimensión determinada para llenar la matriz de los pesos en cuestión.

A modo de conclusión, en esta sección se ha explicado de forma integral el algoritmo diseñado para crear redes neuronales con estructuras evolutivas, se han

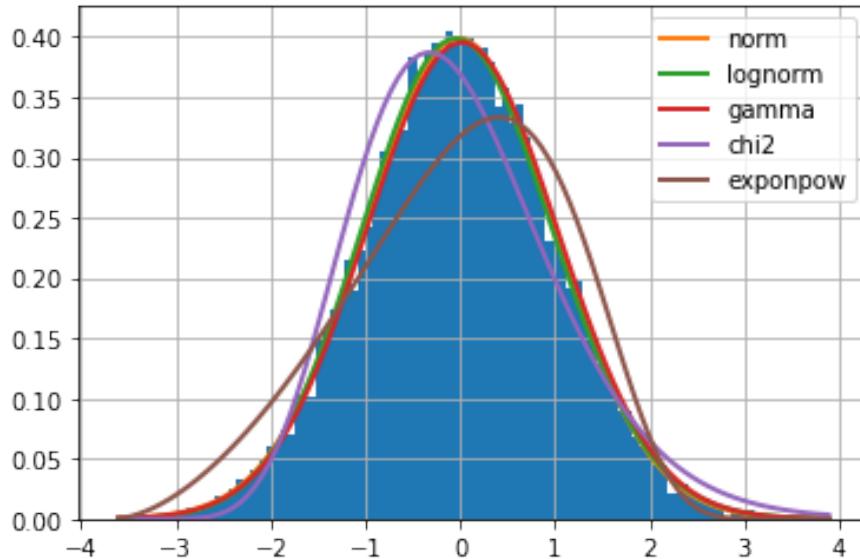


Figura 3.19. Obtención de la distribución de probabilidad normal

Distribución	RMSE	AIC	BIC	KL
norm	0,006947	667,948131	-240392,018547	inf
lognorm	0,007749	666,986517	-238592,548073	inf
gamma	0,009500	662,134767	-235252,984028	inf
chi2	0,083577	848,642065	-199627,061286	inf
exponpow	0,177132	899,103717	-187320,630900	inf

Tabla 3.5. Resumen de las mejores distribuciones ajustadas

presentado tres escenarios de aplicación particular y, por último, se han indicado las dos alternativas disponibles que permiten llenar los pesos de las matrices siguiendo como fuente de inspiración el método de *padding* empleado en las redes convolucionales.

3.5.2. Definición de la tarea de aprendizaje

Como se introdujo en el capítulo de introducción, el objetivo fundamental de este proyecto consiste en detectar si en un determinado gráfico de velas se produce el patrón de un doble suelo. Es importante matizar que no se pretende predecir si va a suceder dicho patrón, sino únicamente detectar si una determinada imagen lo

posee o no. Lo ideal sería haber empleado las redes convolucionales, en lugar de las densas, ya que éstas primeras emulan el comportamiento de las neuronas de la corteza visual primaria (V1) de un cerebro biológico y por lo tanto, son idóneas para las tareas de visión artificial. Sin embargo, el reto de construir una red completamente evolutiva desde cero ya era suficientemente desafiante y por lo tanto, se redujo el campo de aplicación a las redes densas. No obstante, se plantea para futuros desarrollos del proyecto su extensión este tipo de redes más eficientes para la tarea en cuestión.

De este modo, la red toma como entradas el precio de cierre normalizado y ajustado, según las técnicas presentadas en el capítulo anterior, a lo largo de una ventana temporal de 40 días. Como salida se obtiene una variable binaria que indica si hay un patrón o no en dicha figura. Para comparar la eficacia de una red evolutiva frente a otra que no lo es, se aplicó la técnica de SMOTE, ya introducida, al conjunto de datos de entrada del modelo y se analizaron los resultados obtenidos.

No obstante, también se trabajó con el dataset del MNIST. La razón de peso es que el dataset de los gráficos de velas presentado anteriormente, a parte de ser de elaboración propia, está muy no balanceado y ello tiene severas repercusiones en el comportamiento de la red. Por ello, se quería también comparar la validez del modelo con uno de los dataset más utilizados como benchmark en el ámbito de la IA. En este supuesto, la entrada al modelo son las imágenes del MNIST previamente normalizadas entre 0 y 1 y transformadas a arrays. De este modo, la red toma (*None*, 784) entradas y emite como salida la clasificación de los dígitos en cuestión.

En ambos casos, se utiliza como función de activación de las capas ocultas la *Relu* por ser considerada computacionalmente la más eficiente. Para la capa de salida se emplea la *softmax* ya que el objetivo de la tarea de la aprendizaje es clasificar una figura, y ésta da por salida la distribución de probabilidad de las clases posibles. Para la creación de la red neuronal evolutiva se ha diseñado una clase de objeto en Python notada por *FluidNetwork* que a partir de un vector de capas determinado, diseña la red neuronal madre cuya estructura evoluciona a lo largo del entrenamiento. Dicho vector de capas tiene que tener en el primer elemento el número de entradas y en el último el número de clases posibles a clasificar. De esta forma se consigue que sea un modelo completamente extrapolable a cualquier conjunto de datos.

3.5.3. Diseño experimental

Una vez que se ha explicado la tarea de aprendizaje conviene profundizar en los métodos disponibles para la clase creada, pues éstos son la base de cualquier diseño experimental que se desarrolle con la red. Adicionalmente al método `_init_`, que sirve para instanciar objetos con una determinada secuencia de capas, se tienen los siguientes:

- **Setup:** inicializa los pesos y sesgos de la red de forma aleatoria según la estructura de la red deseada.
- **Info:** muestra un resumen similar al proporcionado por los modelos de Keras en el que se adjuntan las características de cada capa así como, el número total de parámetros entrenables por el algoritmo.
- **Forward_pass:** implementa la parte forward del algoritmo de descenso por gradiente. Es importante advertir no aplica a la salida la función softmax directamente, ya que sino al calcular la función de coste con:

```
1 loss = tf.nn.softmax_cross_entropy_with_logits(Y,A)
```

produce resultados erróneos. Es decir, devuelve la Z y no la A , ($A = f(Z)$) resultado de aplicar la función de activación.

- **Compute_loss:** se encarga de calcular el valor de la función de coste en un punto determinado del entrenamiento.
- **Train_on_batch:**

Este método merece especial atención. El descenso por el gradiente es el método dominante a la hora de optimizar los modelos de DL, sin embargo, en la práctica existen diferentes variantes de aplicación: el gradiente estocástico y el calculado en batches. El descenso de gradiente estocástico, a menudo abreviado SGD, calcula el error y actualiza el modelo para cada ejemplo del conjunto de datos de entrenamiento. Por el contrario, el algoritmo de descenso por gradiente en batches calcula el error para cada ejemplo en el conjunto de datos, pero sólo actualiza el modelo después de que se hayan evaluado todos los ejemplos del entrenamiento. De este modo, este método realiza las actualizaciones del modelo al final de cada época. En este proyecto se ha elegido esta opción ya que presenta las siguientes ventajas:

- Al procesar un menor número de actualizaciones esto hace que la variante del descenso por gradiente sea más eficiente desde el punto de vista computacional
- A su vez, la menor frecuencia de actualización da lugar a un gradiente de error más estable y puede dar lugar a una convergencia más estable en algunos problemas.
- Por último, la separación del cálculo de los errores de predicción y de la actualización del modelo hace que el algoritmo se preste a implementaciones basadas en el procesamiento paralelo.

En este sentido el tamaño del batch es un hiperparámetro del modelo a optimizar. En este trabajo se han empleado conjuntos de 128 y 256 elementos. De este modo, este método se encarga de precisamente de aplicar el descenso por gradiente en batches. Para ello se sigue la siguiente secuencia:

1. Para cada batch se abre un ámbito de *GradientTape()*
 2. Dentro de ese ámbito se llama al *forward pass* y se calcula el valor de la función de pérdida
 3. Fuera del ámbito se recupera el valor del gradiente de los pesos con respecto a la función de pérdida
 4. Finalmente, se optimizan los parámetros según el optimizador deseado
- **Update_params:** implementa el optimizador para actualizar los pesos. Permite la elección del optimizador *Adam*, cuyos resultados se discutirán en capítulos posteriores.
 - **Predict:** se encarga de predecir la salida de clasificación para un conjunto de entrada dado.
 - **AG_update:** este método es el que se encarga de copiar la información del entrenamiento de una red madre a sus hijas y aportarles de este modo, su herencia genética en términos de pesos y sesgos. Se trata, pues, de la pieza clave de este algoritmo.
 - **Activate_neurons:** procede con la realización de los ajustes a nivel de neurona para transformar los pesos de una red madre a una red hija. Como ya se ha comentado, permite que dichos pesos se inicialicen de forma aleatoria o que se muestreen a partir de una distribución de probabilidad que se ajuste a los datos de los pesos de las capas disponibles.

- **Sample_distribution:** realiza la tarea de, dada un índice de una capa determinada, encontrar la distribución de probabilidad que mejor se ajuste a los pesos y sesgos de la misma según el criterio de menor BIC. A continuación, tras obtener los parámetros óptimos y la distribución de probabilidad más realista, muestrea un número finito de elementos definidos por las dimensiones del vector a llenar.
- **Update_keys:** este método ajusta los diccionarios de w y b tras realizar las transformaciones evolutivas.
- **Train:** se encarga de ejecutar el entrenamiento completo de la red neuronal, así como, gestionar las activaciones del AG para permitir que la estructura de la red vaya evolucionando con nuevos datos de entrada.

En relación con las técnicas de regularización disponibles, conviene explicar que no se ha incluido ninguna en específico, pues realmente el simple hecho de pasar de una estructura de [10, 30, 40, 1] a [10, 20, 40, 1] ya aplica de manera indirecta un Dropout del 33.33 % a las neuronas de la primera capa oculta. Es por tanto, que la propia red neuronal, sometida a los dictámenes del AG, es la que determinará si es necesario su aplicación o no.

Por otra parte, con respecto a la separación del conjunto de datos para entrenamiento y test, el modelo se entrena con un 70 % de los datos disponibles y que utilice los restantes para validar los resultados. En este sentido es importante velar por el hecho de que el entrenamiento sea significativo. Para ello, cada vez que se crea una nueva red neuronal a partir de una primitiva, ésta se entrena con una partición aleatoria del conjunto de entrenamiento diferente a la de su madre, para evitar cualquier tipo de sesgo en los datos. No se ha hecho uso de técnicas más sofisticadas como la validación cruzada, ya que las métricas en ambos conjuntos eran muy similares y por lo tanto, no ha sido necesario comparar diferentes modelos entre sí. Por último, si que es importante matizar, que para la separación de los datos en train y test se ha impuesto la condición de que se mantuviera la misma proporción de cada una de las clases en ambos conjuntos; esto resulta crucial ya que en el caso de los datos de los patrones el dataset está muy poco balanceado y se pueden generar comportamientos atípicos no deseados.

Por último, la Tabla 3.6 incluye los valores predefinidos de los hiperparámetros utilizados en el entrenamiento.

Hiperparámetro	Valor
Épocas	4000
Learning rate	0,001
Número máximo de capas ocultas	5
Número mínimo de capas ocultas	2
Número mínimo de neuronas por capas	10
Número máximo de neuronas por capas	128
Batch size	128
β_1 para Adam	0,9
β_2 para Adam	0,999
Pesos que se muestran	1 %

Tabla 3.6. Hiperparámetros del entrenamiento

3.5.4. Métricas de evaluación

Un clasificador es tan bueno como así lo sean las métricas que se empleen para evaluarlo. Estos indicadores dependen simultáneamente del tipo de tarea de aprendizaje y de los datos que ingesta el modelo. En general, la selección de las métricas correctas se trata de una de las piezas claves de los modelos de DL, este paso se vuelve aún más crucial e importante en el caso que se trabaje con datos no balanceados, ya que la mayoría de las métricas parten de la hipótesis que las clases a predecir se distribuyen de forma homogénea. De forma general, las métricas se utilizan para constatar el rendimiento del modelo y en el caso de los problemas de clasificación, éstas implican la comparación de la etiqueta de la clase esperada con la etiqueta de la clase predicha por el algoritmo. La selección de los hiperparámetros presentados anteriormente e incluso el tratamiento de los datos, se guía por la búsqueda del mejor resultado en la métrica de evaluación definida, aplicada sobre el conjunto liberado como test. Por lo tanto, aquí es donde radica la importancia y la trascendencia de la elección: si no se toman por referencia buenos indicadores, el modelo se optimizará conforme a unos criterios no deseados y no se alcanzará el objetivo definido en la tarea de aprendizaje.

Existen métricas estándar muy utilizadas para evaluar los modelos predictivos de clasificación, como el *accuracy* o el error de clasificación. Las métricas estándar funcionan bien en la mayoría de los problemas, por lo que son ampliamente adoptadas. Pero todas las métricas hacen suposiciones sobre el problema o sobre lo que es importante en el problema. Por ejemplo, indicar la tasa de acierto en un problema de clasificación con datos muy desequilibrados puede ser peligrosa-

mente engañoso, ya que de forma natural, el modelo acertaría la mayoría de los casos con un simple algoritmo aleatorio. Esta métrica resulta útil para evaluar el rendimiento de la red neuronal con los datos del MNIST, pero resulta claramente ineficiente para detectar los patrones del doble suelo. A diferencia de las métricas de evaluación estándar que tratan todas las clases como igual de importantes, los problemas de clasificación no balanceados suelen valorar los errores de clasificación con la clase minoritaria como más importantes que los de la clase mayoritaria. Por ello, se necesitan métricas de rendimiento que se centren en la clase minoritaria, lo que supone un reto porque es en la clase minoritaria donde se carecen de las observaciones necesarias para entrenar un modelo eficaz.

Existen cientos de métricas para evaluar los modelos de clasificación, en este proyecto se seguirá la taxonomía propuesta por Cesar Ferri et al en su artículo [?] en el que establece tres grupos fundamentales de métricas de umbrales, ranking y probabilidad. Las primeras, sirven para cuantificar el error cometido en la predicción y engloban algunos indicadores como el accuracy, la precisión, el recall, el F1-score o el F-beta score. Las segundas, focalizan la atención en evaluar cómo de efectivo es el clasificador a la hora de separar las clases y definen algunas métricas como la curva ROC o el AUC (área bajo la misma). Por último, las tercera, están destinadas a cuantificar la incertidumbre del clasificador en sus predicciones y por lo tanto, son útiles para problemas en los que se está menos interesado en las predicciones de clase incorrectas frente a las correctas y más en la incertidumbre que tiene el modelo en las predicciones y en penalizar aquellas predicciones que son erróneas pero de alta confianza; entre ellas se encuentra el Logloss, el BrierScore y el BrierSkillScore.

Dada la enorme cantidad de alternativas posibles, se ha seguido las recomendaciones sugeridas en el artículo citado anteriormente. Para ello es preciso entender cuál es el objetivo final del clasificador y qué error debe ponderar más el fallo del mismo. Por ejemplo, en una prueba de diagnóstico de cáncer de mama, la sensibilidad es una medida de lo bien que la prueba puede identificar los verdaderos positivos y la especificidad es una métrica que evalúa lo bien que una prueba puede detectar los verdaderos negativos. Para todo este tipo de pruebas de diagnóstico y de cribado suele haber un equilibrio entre estas dos métricas, de modo que una mayor sensibilidad implica una menor especificidad y viceversa. Aquí es donde entra en juego el factor de la ponderación de los errores, ya que en este caso se debe penalizar más que se indique a un paciente que no tiene cáncer cuando realmente sí que lo padece, y por lo tanto, el número de falsos negativos es más importante

que el de falsos positivos.

Para el caso de interés se debe de estimar la penalización de cometer ambos errores, entendiendo en profundidad su aplicación práctica con los datos en cuestión. Un falso positivo implicaría que el modelo no detecta un patrón de doble suelo cuando realmente sí que lo hay, mientras que un falso negativo, es el error que se comete al identificar un patrón de doble suelo en una figura arbitraria. En el primer caso, la pérdida que se genera es el coste de oportunidad de no invertir cuando se va a producir al alza, mientras que en el segundo, dicho error es más severo ya que el modelo daría una orden de compra y realmente la figura no sigue ningún patrón específico. De este modo, si el objetivo de la tarea de aprendizaje fuese predecir futuros patrones técnicos, se debería dar mayor importancia a los falsos positivos y por tanto, se debería emplear como métrica el F-beta score. Sin embargo, el propósito del algoritmo no es predecir si dada una figura se va a completar un patrón, sino si dicha figura lo contiene. Por lo tanto, para acercar más el proyecto a la realidad se usará la precisión como métrica discriminatoria.

Capítulo 4

Diseño del algoritmo genético

La profundización en los estudios en el área de la IA han propiciado la aparición de métodos de búsqueda de algoritmos con clara analogía con teorías de carácter biológico y físico. Dentro de este marco conceptual se ubican los Algoritmos Genéticos, en adelante AG, cuyo núcleo central se define por la creación de un conjunto aleatorio de individuos que se somete a una búsqueda guiada, mediante un procedimiento iterativo para así alcanzar la solución óptima. En general, se tratan de algoritmos basados en la dinámica de la selección natural y en la mecánica de la genética humana. Dicha teoría tiene como premisa fundamental la supervivencia del individuo más apto y su forma de alcanzarlo reside en la adaptación del ente a su entorno.

La implementación de los algoritmos genéticos se desarrolla bajo el esquema evolutivo defendido por el científico Charles Darwin, a través del cual, se crea una población inicial de individuos que son sometidos a una serie de transformaciones que combinan un intercambio de información estructurado con secuencias aleatorizadas para así fusionar en un único procedimiento una de las genialidades de las genialidades de las búsquedas humanas. Cada uno de estos ajustes lleva asociad una puntuación en una escala de bondad o *fitness* que será que determine si el gen ha de proliferar o no.

Los pilares fundamentales que rigen este algoritmo iterativo no son fenómenos ajenos a las características socio-económicas que definen la sociedad actual. De hecho, se puede encontrar un claro paralelismo entre ellos, por ejemplo, la competencia en los mercados se puede relacionar con la lucha por la supervivencia; la reproducción de los individuos se asemeja a la fusión de las distintas sociedades y empresas, y por último, la diversidad de la población manifiesta un componente

intrínseco en ambos ámbitos.

Los primeros atisbos de las ideas que subyacen a los AG se encuentran en los artículos de Holland y autores posteriores como Golberg y De Jong contribuyeron enormemente en la construcción de los cimientos de este algoritmo. Holland estableció una agenda amplia y ambiciosa para comprender los principios subyacentes de los sistemas adaptativos, sistemas que son capaces de auto modificarce en respuesta a sus interacciones con los entornos en los que deben funcionar. En opinión de Holland, la característica clave de dichos sistemas es el uso exitoso de la competencia y la innovación para dotar a los algoritmos de la capacidad de responder dinámicamente a eventos imprevistos y entornos cambiantes. Se observó que los modelos simples de la evolución biológica capturaba estas ideas a través de las nociones de la supervivencia de los más aptos y la producción continua de nuevos descendientes. La diligencia del AG, como la de cualquier otro algoritmo de optimización, se basa en el mecanismo para equilibrar los dos objetivos básicos que se persiguen: explorar espacios de soluciones óptimas y simultáneamente, permitir la búsqueda de otros entornos prometedores.

4.1. Adaptación de los AG al diseño de redes evolutivas

Una vez entendidas las bases teóricas sobre las que se sustentan los primeros atisbos de los AG, se procede a explicar el enfoque mediante el que se ha conseguido diseñar este tipo de algoritmos para permitir el desarrollo de redes neuronales con topología evolutiva. En el caso de los AG, se suele emplear el término de *individuo* para notar a las posibles soluciones del problema de optimización que se resuelve. Resulta fundamental entender qué simbolizan en este caso este conjunto de entes. En este caso, cada uno de los individuos representan una nueva red neuronal, que hereda la información de los pesos de una red madre (de topología diferente a ella), y que se entrena en la tarea de clasificación para comprobar si su estructura es más eficiente que la estructura que ha heredado de sus progenitores. Para ilustrar mejor el concepto, supongamos que en cierto punto del entrenamiento de la red neuronal de partida, se estima que se está agotando su capacidad de aprender, de este modo, se activa el AG y éste genera automáticamente una población de redes neuronales hijas que, partiendo de la información del entrenamiento de la red primitiva, proliferan hacia otro tipo de estructuras para acelerar el entrenamiento de

la red principal. A continuación, se adjunta en la Figura 4.1 el esquema descrito previamente.

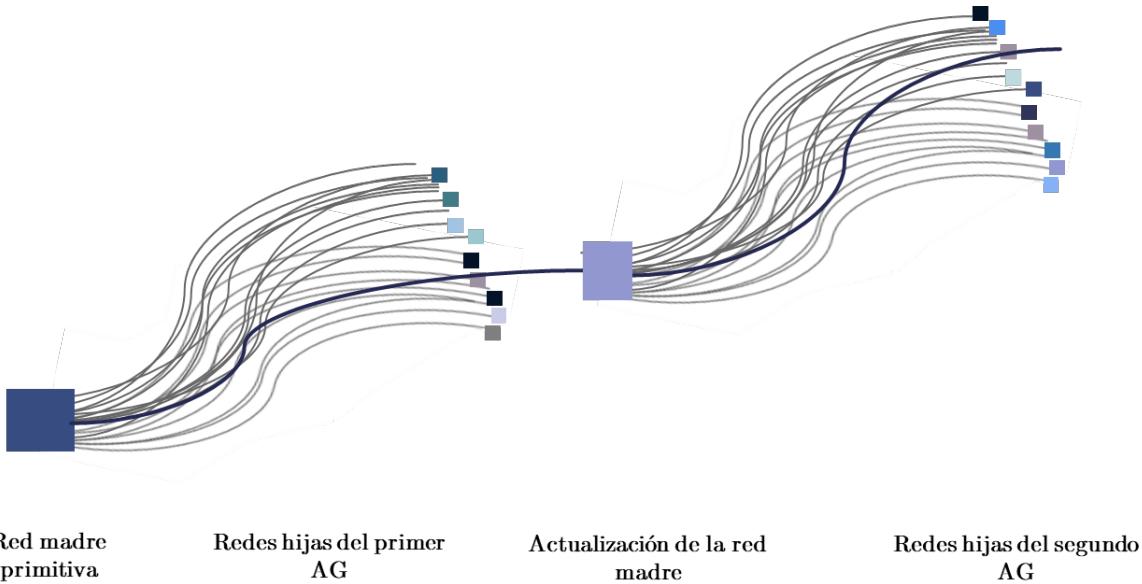


Figura 4.1. Procedimiento de búsqueda de la red evolutiva

Así, se obtienen todas las posibles soluciones en un tablero como el de la Figura 4.2 donde se visualiza de forma conceptual como aquellos individuos con mejor *fitness* tienen mayor densidad poblacional. No obstante, también se favorece la proliferación de nuevas soluciones para impulsar el mecanismo de exploración heurístico.

A modo de conclusión, los individuos de la población del AG son redes neuronales con diferentes estructuras que heredan los pesos del entrenamiento de una red neuronal superior. Dichas redes hijas se entranan a lo largo de un número determinado de épocas para detectar si hay alguna topología que permita a la red madre mejorar su rendimiento en la tarea de clasificación de patrones. En las siguientes secciones se profundizará en el mecanismo subyacente que controla la búsqueda guiada de las soluciones.

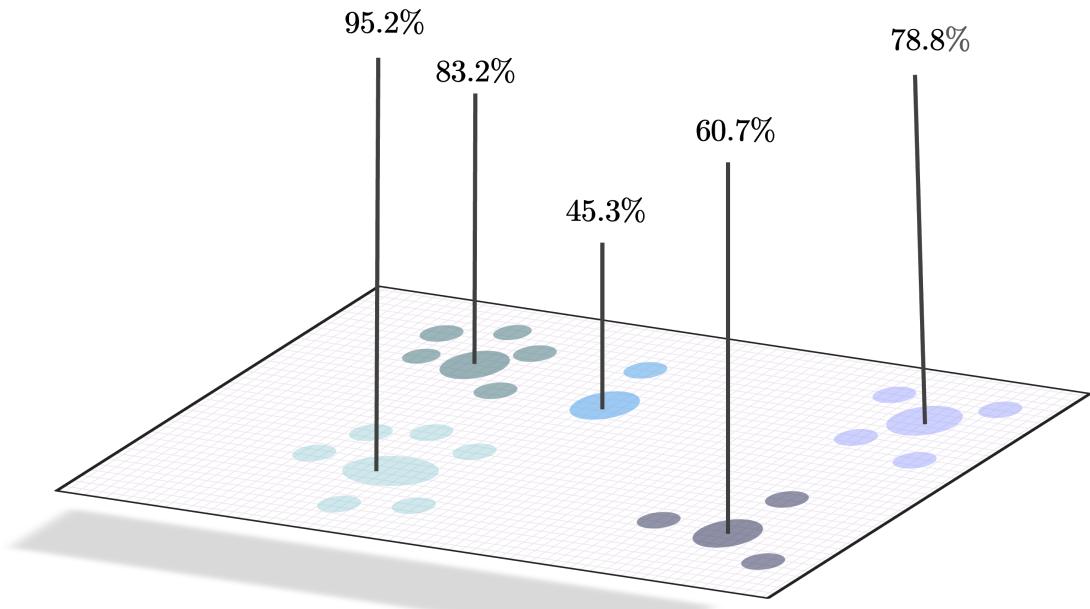


Figura 4.2. Densidad de la población de los individuos

4.2. Estructura general de un AG

Se pueden aplicar los AG a una gran variedad de problemas de optimización que no están adaptados a las condiciones de optimización clásicas (funciones no diferenciables, discontinuas o altamente no lineales). Por otra parte, también son capaces de resolver con gran facilidad problemas de programación de entero mixto, en las que algunas variables, como es el caso, están restringidas a tomar valores enteros, pues esto depende fundamentalmente de la forma en la que se codifiquen los genes de cada individuo. Para simular un proceso evolutivo el algoritmo requiere de:

- Una población inicial de posibles soluciones debidamente representadas a través de individuos.
- Un proceso de selección basado en la aptitud de los individuos.
- Un proceso de transformación, es decir, un mecanismo de construcción de nuevas soluciones a partir de las disponibles en la población anterior.

Con todas estas cláusulas previas se pueden sintetizar, a modo de esquema, las etapas transitadas por el algoritmo durante la ejecución de cada una de las iteraciones:

1. Se genera la semilla aleatoria de una población inicial diversa, capaz de explorar todo el espacio del problema.
2. Se evalúa cada individuo que compone la población primaria.
3. Se seleccionan los genes que se usarán como padres para la evolución de la especie, así como, los “élite” que encabezan la generación actual. Para computar las puntuaciones se realizan dos pasos:
 - a) Se calcula el valor de la función de aptitud que asigna puntuaciones “crudas” del fitness de cada individuo
 - b) Se escalan dichas puntuaciones para acotarlas dentro de un rango más manejable, y, por último, se transforman en los “valores expectativos”.
4. Se realiza el cruce y la mutación para actualizar la población.
5. Se comprueban los requisitos de parada del algoritmo. Si se cumplen, se da por finalizado el proceso, de lo contrario, se retoma el punto 2.

A continuación, se expondrá en detalle cada uno de los elementos mencionados anteriormente. Previo a ello, se adjunta en la Figura 4.3 un esquema que pretende ilustrar las fases de un AG.

4.3. Población inicial

Uno de los factores predominantes que determinan el rendimiento del algoritmo pasa por la selección correcta de la población inicial. Se pueden definir distintos parámetros con multitud de modalidades cuya manipulación puede desempeñar un papel fundamental en la convergencia del AG.

En primer lugar, es necesario establecer la codificación a utilizar. Cada individuo que conforma la población viene representado por un cromosoma que, a su vez, está compuesto por tantos genes como variables tenga el problema. Es decir, un individuo no se define por su ranking obtenido en la función de aptitud, sino

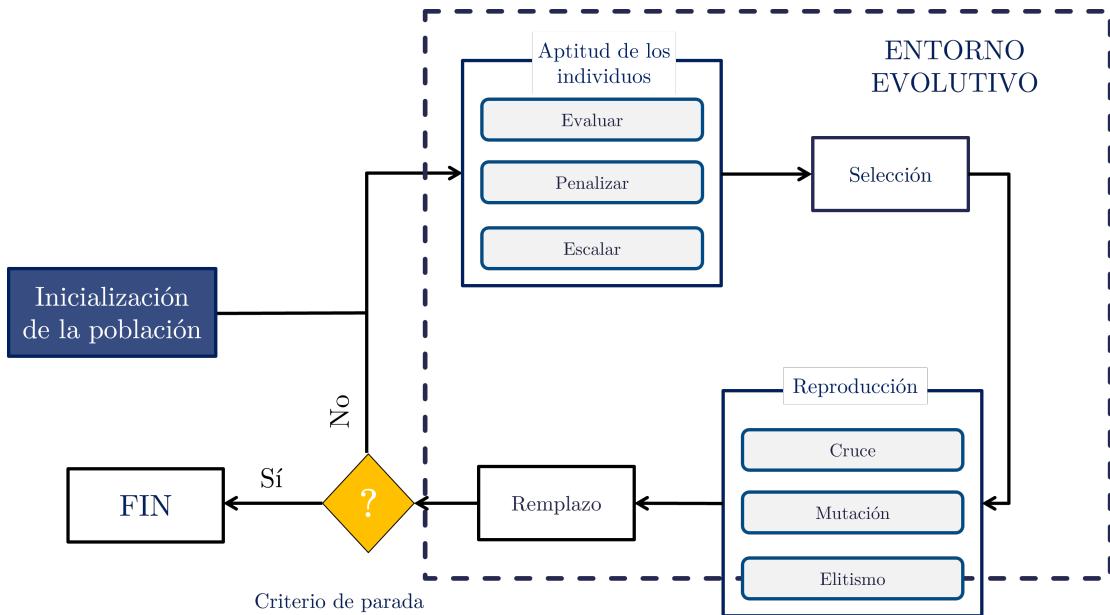


Figura 4.3. Esquema del procedimiento evolutivo

que viene representado por la combinación de parámetros necesarios para alcanzar dicha puntuación. Por todo ello, lo primero que hay que decidir es cómo codificar dichos genes. Puesto que se trata de un problema con una restricción de enteros, la única vía libre resulta en asignar datos del tipo *int*. En este sentido, cada individuo se codifica por las combinaciones de capas intermedias de las redes a estudiar. Por ejemplo, en la Tabla 5.1 se incluyen los primeros 10 individuos generados para la población inicial en el caso de trabajar con los datos del MNIST. Tal y como se puede observar, se permite la creación de distintas redes neuronales con diferentes números de capas y número de neuronas por capa. Es importante advertir que sólo se trabajan con las capas ocultas ya que tanto la capa de entrada como la de salida dependen únicamente del conjunto de datos utilizado.

Una vez establecido el parámetro anterior, resulta crucial analizar el número de individuos que deben conformar la población. Parece intuitivo pensar que un tamaño muestral reducido no permite explorar todo el espacio de estudio, y, que, por contra, un número elevado de individuos puede encarecer el coste computacional. Por lo tanto, para encontrar el equilibrio entre ambos requisitos se calcula el número de individuos mediante la siguiente fórmula:

Capa I	Capa II	Capa III	Capa IV
39	84	-	-
94	87	-	-
87	82	33	44
70	107	-	-
11	74	-	-
124	12	69	-
63	125	-	-
72	96	41	18
35	27	93	31
97	103	-	-

Tabla 4.1. 10 individuos de una población inicial

$$N_{pop} = \frac{N_{var} \times 50}{5 + N_{var}} \quad (4.1)$$

En ambos la ecuación da lugar a 25 individuos ya que el número de variables del AG es el número máximo de capas menos el número mínimo. Si la distancia promedio entre los individuos es considerable, entonces se favorece la pluralidad de los genes. Este parámetro se puede controlar delimitando el rango inicial de la población, así como, los umbrales máximos y mínimos que pueden tomar las variables implicadas. En este sentido, si se conoce intuitivamente de antemano valores que son impracticables o que se encuentran muy alejados de la solución, conviene restringir aquellas variables involucradas, para así evitar que el AG explore zonas prohibidas y propiciar una búsqueda más profunda en aquellas que susciten mayor interés. En el caso de este proyecto, los valores delimitadores se adjuntan en la Tabla 4.2

Hiperparámetro	Umbral superior	Umbral inferior
Capas ocultas	2	7
Neuronas por capas	10	128

Tabla 4.2. Umbrales de los individuos

4.4. Evaluación de los individuos

La función de aptitud o fitness es la encargada de clasificar a los individuos de la población y de describir la bondad de la solución obtenida. Su correcta definición juega un rol fundamental en el funcionamiento del AG. Entendiendo que cada individuo de la población representa una nueva red neuronal a entrenar, se puede afirmar que la función fitness en este caso es, precisamente, el entrenamiento de cada red neuronal hija a partir del descenso por gradiente en batches ya explicado previamente.

Como ya se ha comentado anteriormente, el objetivo de la tarea de aprendizaje es clasificar si una imagen contiene un patrón o no y para valorar el rendimiento del modelo se emplea la métrica de precisión. Sin embargo, al optimizar las redes neuronales mediante el método del descenso por gradiente, en la función de pérdida introducida, no se tiene en cuenta la métrica que se pretende optimizar. Dicho indicador sólo se monitoriza pero los parámetros de la red no se ajustan para maximizar su rendimiento. Esto se debe fundamentalmente a que se tratan de métricas no diferenciables y que por lo tanto, no se pueden optimizar mediante el tradicional método del descenso por gradiente. En general, las funciones de pérdida tienen dos propiedades: son globalmente continuas y diferenciables. Esto significa básicamente que la función que se utiliza no puede saltar, debe de estar definida en cada punto, no tiene giros bruscos ni tangentes verticales. No obstante, como se ha indicado anteriormente, una de las mayores ventajas de los AG es que las funciones de coste pueden ser no diferenciables. Esto supone una gran ventaja para la red evolutiva, ya que permite de manera indirecta ajustar mejor su estructura conforme a lo que se espera de ella.

De este modo, cuando se entrena la red neuronal hija sobre un conjunto aleatorio de entrenamiento, se evalúa su resultado sobre el conjunto de test y es precisamente la métrica elegida según el dataset, (accuracy, para el MNIST y precisión, para los patrones de doble suelo), la que representa el valor de la aptitud de cada individuo de la población. Por lo tanto, este diseño de una red neuronal evolutiva no sólo aporta la clara ventaja de que la red puede ir transformando su topología con cada dato nuevo, sino que dicha transformación está orientada hacia su verdadera misión de aprendizaje, sin necesidad de tener que definir funciones de pérdida diferenciables.

Por último, tras definir todas las pormenorizaciones diseñadas para el correcto

funcionamiento del AG, conviene indicar que una ventaja adicional que se obtiene derivada del método es que se permite la computación en paralelo para el cálculo de la función objetivo de cada individuo de la población. Esto reduce drásticamente el tiempo de computación ya que en lugar de tener que evaluar a los 50 individuos de forma secuencial, se puede realizar el cálculo distribuyendo cada red hija en un núcleo de la CPU diferente. Para ello, se hizo uso del módulo multiprocessing de Python. Este ofrece dos clases alternativas para implementar el código en paralelo que se diferencian en la forma en la que gestionan la memoria internamente:

- **Process:** asigna todas las tareas en la memoria de una sola vez. En este caso, asignaría en memoria todos los futuros entrenamientos de las redes hijas que se tienen que completar.
- **Pool:** asigna en memoria sólo los procesos en ejecución, es decir, toma el número de procesos de trabajos definidos en el pool y los ejecuta. Lanzar muchos procesos utilizando la clase Process es prácticamente inviable ya que podría romper el sistema operativo. De ahí surge Pool, que se encarga de distribuir los trabajos y recoger los resultados de todos los procesos generados en presencia de un número mínimo de procesos de trabajo. El único inconveniente que presenta es que no funciona en el intérprete interactivo ni en las clases de Python. Requiere que el módulo `__main__` sea importable por los hijos.

La ventaja de permitir una gestión más eficiente de la memoria fue la razón fundamental por la que se escogió esta clase para paralelizar la función objetivo. Para ello, se hizo uso del método `map()` que permite trocear el número de procesos en tareas separadas. En este caso el término proceso hace referencia al entrenamiento de cada red neuronal hija.

4.5. Selección de los padres

El método de selección influye de forma notoria en la velocidad de convergencia del algoritmo. Si se ejecuta de forma sospechosamente rápida, resulta un claro síntoma de inefficiencia del programa, pues, probablemente, se halla quedado estancado en un mínimo local pero no global. Existen múltiples modalidades de selección, sin embargo, todas comparten la misma filosofía: elegir a nuevos candidatos según una distribución de probabilidad que esté sesgada hacia los individuos

mejores dotados.

Existen diferentes métodos para la selección de los padres:

- **Selección proporcional a la aptitud:** un individuo se convierte en padre con una probabilidad proporcional a su aptitud. Por lo tanto, aquellas redes con mejor fitness tendrán mayor probabilidad de reproducirse y propagar su estructura a la próxima generación. Dentro de este método destaca la selección por ruleta. Este principio sigue una búsqueda lineal a través de una rueda de ruleta con las ranuras de la misma ponderada según la proporción de los valores del fitness de cada individuo. Este es el método más acertado para el problema en cuestión ya que no descarta a ningún individuo.
- **Selección basada en rangos:** en este caso, la población se ordena según los valores objetivos y la probabilidad de que cada individuo sea seleccionado para reproducirse depende de su aptitud normalizada por la aptitud total de la población. La clasificación introduce una escala uniforme en toda la población y proporciona una forma sencilla y eficaz de controlar la presión selectiva. Dentro de este enfoque destaca la selección por torneo en el se generan subconjuntos de k individuos que compiten según su condición de fitness.
- **Selección basada en umbrales:** es un método de selección artificial que utilizan los padres para poblaciones grandes o la selección masiva. En la selección de truncamiento, los individuos se clasifican según su aptitud. Solo los mejores individuos son seleccionados como padres.

Por lo tanto, el método implementado es el siguiente:

1. Se computa el fitness para cada individuo de la población
2. Se normalizan los valores entre 0 y 1
3. Se genera un índice aleatorio para evaluar la aptitud de cada individuo
4. Se genera un número aleatorio y se compara con la aptitud del primer elemento del vector del punto anterior. De este modo, el valor de fitness actúa como umbral, si el aleatorio es menor el individuo candidato pasa a ser padre de la siguiente generación
5. Este procedimiento iterativo se repite hasta que se tengan dos padres para dar lugar a su descendencia

4.6. Reproducción de la población

La robustez de los AG es fruto de su capacidad de atinar con óptimos globales en un panorama multimodal. La potencia se atribuye en gran parte al arte con el que se reproducen gracias a la función de Crossover. Ésta posibilita un intercambio estructurado, aunque aleatorizado, de material genético entre soluciones habilitando que aquellas más virtuosas se conviertan en óptimas. La población no contiene exclusivamente un conjunto de entes, sino que ellos representan una simple parte alícuota de todo el contenido (noción y clasificaciones) que aportan los alelos en cada generación.

De esta forma, el AG explota esta riqueza de material informativo mediante tres métodos de reproducción: el primero de ellos, Crossover, permite que el AG localice las zonas donde se hallan los mínimos locales, mientras que el segundo, Mutación, posibilita que el AG amplíe su búsqueda para explorar nuevos espacios con el fin de determinar el mínimo global. Por último, el tercero, la clonación, garantiza la mejoría del AG en cada generación. A continuación, se detallará cada una de estas vías de proliferación.

4.6.1. Crossover

La justificación principal del cruce reside en la premisa que, si se toman dos individuos aptos y se intercambian sus genes aleatoriamente, cabe la posibilidad que se genere una nueva especie cuyos cromosomas hereden toda la bondad de sus padres. Claro está, que no existe una certeza absoluta que se mejore la especie, pero ello no implica necesariamente que se esté contaminando la nueva inserción pues, habilita la posibilidad de que se generen nuevos fenotipos no contemplados hasta el momento. Existen diferentes alternativas para codificarlo, en este trabajo, se ha optado por el cruce uniforme, pues es el único compatible con las condiciones del problema.

En la actualidad existen diferentes mecanismos para implementarlo:

- **Cruzamiento de un único punto:** a partir de ese punto se intercambian las cadenas de los progenitores.
- **Cruzamiento de N puntos:** el nº de puntos de ruptura es constante.
- **Cruzamiento segmentado:** el nº de puntos de ruptura es variable.

- **Cruzamiento uniforme:** para cada posición, se decide al azar si se intercambian las posiciones (cuando la codificación es binaria).
- **Cruzamiento aleatorio:** Para cada posición, se decide al azar si se hereda de un progenitor o del otro (cuando la codificación es binaria).

Ninguna de las estrategias anteriores parecía adecuadas para el problema en cuestión, de forma que se implementó una mecanismo de cruzamiento ad-hoc. Es importante tener en cuenta la naturaleza del problema en cuestión. En este sentido, los genes de cada uno de los individuos de la población representan las neuronas que se activan en cada capa de una red neuronal. De este modo, se puede probar que aquellas capas cuyos gradientes sean más pequeños, son las que mejor han convergido hacia la solución final. Esta es la intuición que subyace bajo la nueva técnica de cruzamiento desarrollada, es decir, se parte de la hipótesis que las capas que tienen gradientes más estables en sus pesos, son las que mejor se han adaptado al problema en cuestión. Sin embargo, esto puede dar lugar a óptimos locales, ya que puede que esa convergencia sea prematura. De este modo, para evitar este fenómeno se hace el cociente del gradiente entre el fitness de cada red neuronal. A continuación se muestra el código que lo implementa, se definen dos funciones: la de permutación que cruza los genes y la del cálculo de los gradientes ponderados para cada capa, ambas son métodos de una clase denominada AG, que contiene todas las funcionalidades descritas hasta ahora.

```
1  def permutation(self, nchild):  
2      c = []  
3      fathers_to_select = self.selection()  
4      if len(fathers_to_select) < 2:  
5          return []  
6      else:  
7          p1 = self.pop_layers_prev[fathers_to_select[0]]  
8          p2 = self.pop_layers_prev[fathers_to_select[1]]  
9          w_dW1 = np.abs(self.compute_db_weight(fathers_to_select[0]))  
10         w_dW2 = np.abs(self.compute_db_weight(fathers_to_select[1]))  
11         for j in range(2):  
12             c[j] = []  
13             if np.max([len(p1), len(p2)]) == self.num_min_layers:  
14                 n = self.num_min_layers  
15             else:
```

```

16     # child's length
17     n = np.random.randint(self.num_min_layers, (np.max([len(p1), len(p2)])))
18     for i in range(n):
19         if i < min([len(p1)-1, len(p2)-1]):
20             idx = i
21             aux = np.argmin([w_dW1[idx], w_dW2[idx]])
22             if aux == 0:
23                 c[j].append(p1[idx])
24             else:
25                 c[j].append(p2[idx])
26         else:
27             if len(p1) > len(p2):
28                 idx = i
29                 c[j].append(p1[idx])
30             else:
31                 idx = i
32                 c[j].append(p2[idx])
33
34     return c

```

```

1 def compute_db_weight(self, j):
2     mean_dW = []
3     net = self.pop_net[j]
4     fitness = self.pop_fitness[j]
5     for i in range(1, net.L):
6         mean_dW.append(np.mean(net.dW[i]))
7     return mean_dW/fitness

```

Una vez que se han expuesto las bases sobre las que se asienta este método, conviene analizar la cantidad de individuos que se generan mediante este proceso. Generalmente, se suele asignar una proporción muy elevada, en torno al 80 % de la población (sin contar con los niños élite), y el resto se generan mediante mutación. Garantizar la convergencia del algoritmo es la principal justificación por la que se emplea un porcentaje tan elevado.

4.6.2. Mutación

La mutación es un proceso que consiste en alterar alguno de los genes del individuo de manera aleatoria. Esto permite que la población no se estanque con demasiada rapidez en determinadas zonas del espacio, propiciando la búsqueda de diversidad, redireccionando al azar otros entornos para así evitar caer en óptimos locales.

Las mutaciones sólo se producen con una probabilidad muy baja, con el fin de evitar demasiadas distorsiones cuando el proceso de búsqueda está analizando una zona comprometedora. Si la probabilidad fuera demasiado alta, entonces el AG desaprovecharía toda su estructura jerarquizada de almacenamiento de la información y se convertiría en un proceso completamente dominado por la aleatoriedad. En este proyecto se ha establecido un umbral del 20 %.

4.6.3. Elitismo

uno de los indicadores que sirven para detectar si el algoritmo está convergiendo es comparar el valor de la función fitness de la media de la población con el del mejor individuo. Una operación similar computa la función de mutación para ir reduciendo su porcentaje de aleatoriedad en cada iteración.

Es importante que para garantizar que el proceso de búsqueda nunca dará un paso hacia atrás en cuanto a la calidad de la mejor solución, salvar automáticamente a los mejores individuos de cada generación, los denominados “élite”, que se convierten en los primeros en constituir la población para la siguiente iteración. El número de individuos privilegiados que prevalecen representa una fracción muy reducida del total, aproximadamente un 5 %, pero suficiente para amparar la evolución hacia la búsqueda de soluciones mejores. En este caso, se ha programado para que sólo pase directamente a la siguiente generación la mejor red neuronal de esa iteración. En este punto, resulta crucial el problema que se está resolviendo: se permite que la mejor red pase automáticamente, pero vuelve a partir de cero, porque sino la primera red siempre tendría ventaja frente al resto por entrenarse a lo largo de más iteraciones. Además, también se ha estipulado que para cada nueva iteración del AG se vuelva a realizar una partición aleatoria del conjunto de entrenamiento y test, para evitar este sobreaprendizaje prematuro.

Si éste fuera un número más representativo, la población se estandarizaría en las

primeras soluciones y estaría colapsada por los “súper individuos” que asfixiarían a los menos dotados lo que conllevaría a la conquista de una convergencia prematura.

4.7. Criterios de parada

La pregunta inicial que incita el análisis de este apartado es cómo es posible definir una condición adecuada de parada para un algoritmo que es puramente evolutivo. Esta interrogativa que roza casi la filosofía merece ser estudiada en detalle. Lógicamente, existen variables que se pueden controlar y que fomentan un tipo de parada u otro. Fundamentalmente, los factores que afectan son: el número máximo de generaciones, la tolerancia relativa al cambio y el número máximo de generaciones “estancadas”. De las tres variables mencionadas anteriormente, sólo la primera de ellas puede ser controlada sin depender absolutamente de las particularidades del problema analizado. En cambio, las restantes son mucho más sensibles a las singularidades de cada caso.

La principal dificultad que surge es establecer los parámetros adecuados para garantizar la convergencia del algoritmo, y ello pasa por la selección de un tamaño muestral apropiado. El número de generaciones, como ya se analizó en el primer apartado, depende del número de variables que conforman el problema. Con respecto a la tolerancia relativa al cambio, este es un parámetro que permite que se detenga el AG cuando los cambios relativos en la evaluación de la función fitness son inferiores a una determinada tolerancia. Por último, el número máximo de generaciones “estancadas” hace referencia al caso en el que el mejor valor de aptitud permanece constante en un número considerable de iteraciones. Estudiar esta variable de control puede resultar muy útil ya que reduce notablemente el coste computacional del algoritmo. En el problema en cuestión se tomaron como criterios de paradas la disyuntiva entre el número máximo de iteraciones (establecido en 30) y el número máximo de generaciones sin mejores (fijado en 15). La razón de que se empleen valores tan bajos se debe a que sino se aumenta considerablemente el tiempo de ejecución. Cuando se lanzaron las pruebas en la nube, se aumentaron los márgenes de ambos umbrales.

4.8. Justificación del uso de AG

Los AG son algoritmos de optimización con un claro potencial, muy superiores a los métodos iterativos que descienden de la técnica del uso del gradiente para la localización de mínimos locales. Por otra parte, también vencen a los algoritmos de muestreo arbitrario gracias a su habilidad para direccionar la búsqueda hacia regiones relativamente prospectivas en el espacio de exploración.

Fundamentalmente, se optó por la elección de este algoritmo ya que no exige que la función objetivo sea continua ni derivable, y, sobre todo, porque no requiere que se defina ninguna función explícitamente, sino que permite el uso de datos históricos para encontrar patrones sujetos a las condiciones que se desean maximizar.

Por último, a pesar de que el éxito de un AG depende de la naturaleza del problema en cuestión, sí se puede afirmar que es capaz de encontrar una solución bastante aceptable en unos tiempos muy competitivos. Esto se debe fundamentalmente a que, como profesa Goldberg en el “Teorema de esquemas”, los AG son capaces de situar automáticamente un número considerables de individuos en las regiones con mayor probabilidad de éxito.

Capítulo 5

Análisis de los resultados

En este capítulo se hará una revisión detallada de todos los resultados obtenidos para así poder valorar el rendimiento del modelo diseñado. Existen dos herramientas claves que permiten el desarrollo del modelo ya explicado. La primera, consiste en habilitar la posibilidad de que la red neuronal cambie su topología a lo largo del entrenamiento para que vaya aprendiendo con cada dato de entrada nuevo. La segunda, hace relación al momento en el que se establece que el AG debe entrar en funcionamiento para mejorar el rendimiento de la red neuronal madre. En este proyecto se ha establecido un trigger que habilita la bandera del AG a lo largo de puntos uniformemente repartidos durante su entrenamiento. Por lo tanto, cuando se activa el AG comienza un mecanismo de transmisión de todos los parámetros internos de la red primitiva (pesos y sesgos) hacia sus redes neuronales hijas, que se entrena un número de épocas reducido y devuelve la información a la red neuronal madre, completando así el proceso de remodelación del conocimiento adquirido. Para valorar el rendimiento del modelo se compara el resultado de una red evolutiva frente a una con estructura fija, aplicando las técnicas a los dos conjuntos de datos ya introducidos.

Es importante advertir que sólo se incluyen los resultados de los modelos válidos, pero que para llegar hasta este punto se tuvieron que desechar todas las ideas ya presentadas en el Capítulo 3. No obstante, hay un factor que merece ser estudiado en detalle. Tras haber programado todo el diseño de la red neuronal evolutiva, se probó su funcionamiento sobre el conjunto de datos del MNIST y las métricas obtenidas no fueron nada satisfactorias. Tras analizar minuciosamente los resultados obtenidos y compararlos con un modelo de Keras construido con secuencias de capas densas, se llegó a la conclusión de que los resultados no eran eficientes porque no se estaba empleando el optimizador adecuado. Como punto

inicial se optimizaban los parámetros de la red empleando el tradicional método del descenso por el gradiente por lotes. Sin embargo, en la literatura se recomienda utilizar el algoritmo Adam ya que presenta un comportamiento más adecuado frente a diferentes problemas. De forma general, los optimizadores actualizan los parámetros de peso para minimizar la función de pérdida. La función de pérdida actúa como guía para el terreno, indicando al optimizador si se está moviendo en la dirección correcta para llegar al fondo del valle, el mínimo global. El principal problema encontrado es que se caía de forma recursiva en óptimos locales.

Por ello, se decidió programar la implementación del Adam desde cero, lo que aumentó considerablemente la dificultad del problema. El Adam se trata de un algoritmo con tasa de aprendizaje adaptativa que combina el impulso con Ada-delta o RMSprop. En este sentido, si la red neuronal hija hereda la información de los pesos de la red primitiva, también debe recibir el estado de los momentos de primer y segundo orden del optimizador. Por lo tanto, todo el procedimiento descrito a lo largo del Capítulo 3 para la transformación de los pesos de una red a otra, debe de ser implementado también para permitir heredar el estado de los dos momentos del optimizador. En el caso de que la red neuronal hija tenga mayor número de elementos que la madre, se rellenan las matrices con un padding puro (inicialización a cero y no aleatoria).

$$\begin{aligned}\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}\end{aligned}\tag{5.1}$$

A continuación se presentan los resultados sobre ambos conjuntos de datos.

5.1. Resultados sobre el conjunto de datos del MNIST

Para el caso del MNIST, se recuerda que para dificultar la tarea de aprendizaje se utiliza un conjunto de muestras reducidas y además se aplican las técnicas de *data augmentation*, con los siguientes parámetros. Resulta preciso comentar que no se activan los parámetros de normalización ya que los datos de entrada ya estaban acotados entre 0 y 1.

Parámetro	Valor
featurewise center	False
featurewise std normalization	False
rotation range	20
width shift range	0.2
height shift range	0.2
horizontal flip	True

Tabla 5.1. Parámetros del data generator

Para el diseño de la red neuronal con estructura fija se parte de un modelo secuencial definido en Keras con las siguientes características presentadas de forma visual en la Figura 5.1.

Asimismo, se presentan en la Tabla 5.2 los hiperparámetros con los que se entrenó el modelo.

Hiperparámetro	Valor
Optimizador	Adam
Batch Size	128
Learning rate	0.001
Número de épocas	100

Tabla 5.2. Hiperparámetros de la red densa para el MNIST

Para analizar los resultados se hará uso de dos gráficas fundamentales:

1. Evolución del loss: para garantizar que no se ha producido overfitting en el modelo. En esta gráfica se hará zoom en las últimas épocas para garantizar que los modelos han convergido completamente.
2. Evolución del accuracy: para valorar el rendimiento de la tarea de aprendizaje aplicado al conjunto de validación.

A continuación, se adjuntan en las Figuras 5.5, 5.6 y 5.7 los resultados obtenidos para el modelo entrenado en Keras con las imágenes con ruido.

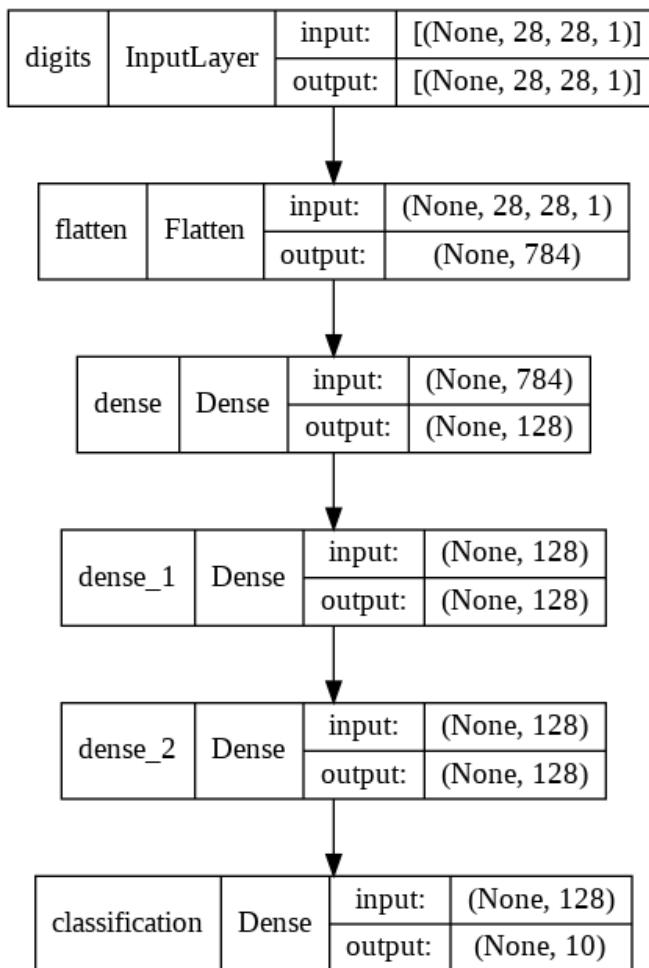


Figura 5.1. Modelo de la red neuronal con estructura fija

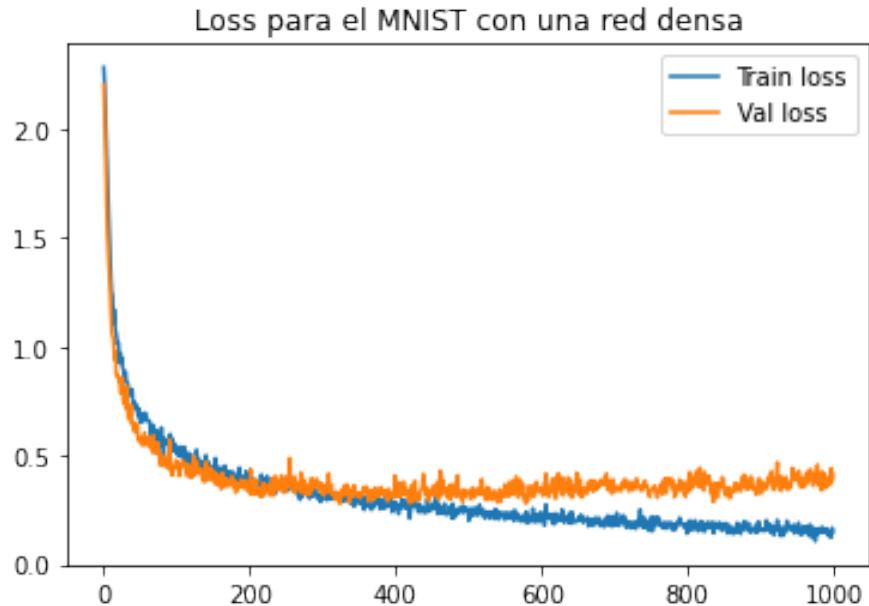


Figura 5.2. Evolución del loss para una red densa con MNIST

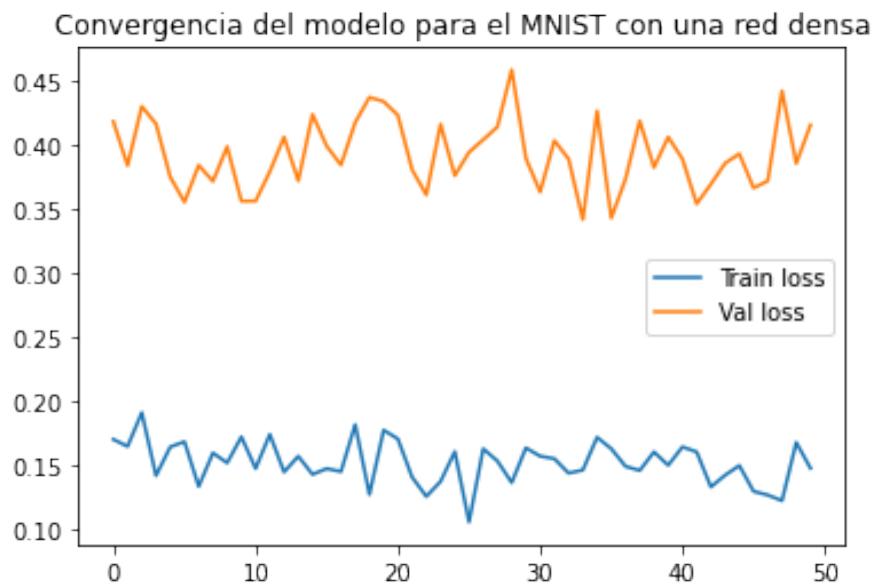


Figura 5.3. Convergencia del modelo para una red densa con MNIST

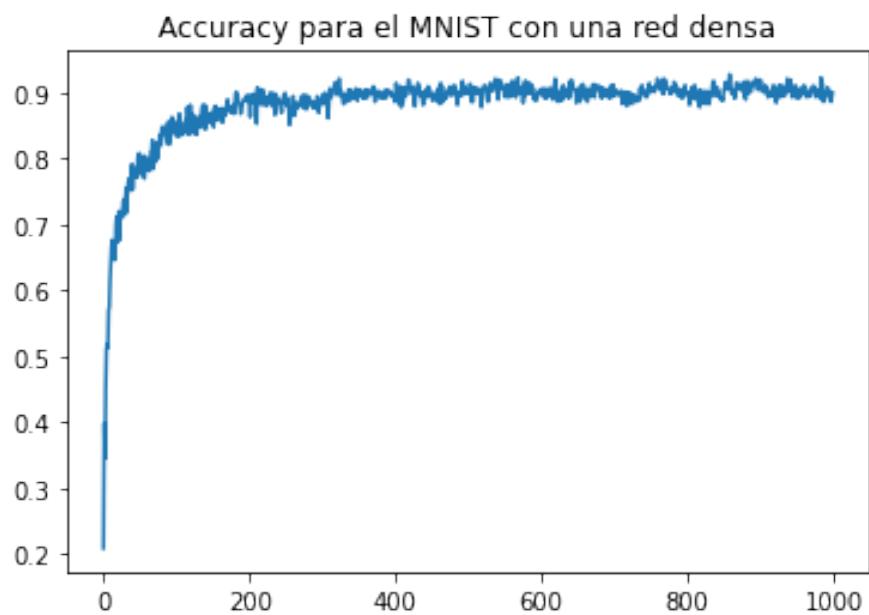


Figura 5.4. Evolución del accuracy para una red densa con MNIST

A continuación, se presenta el análisis de sensibilidad implementado para la red evolutiva. Se lanzaron las sucesivas simulaciones incluidas en la Tabla 5.6. La columna de estructura hace referencia a la red neuronal madre de partida y la de epochs AG al número de épocas con las que se entrena cada red neuronal hija.

Simulación	Optimizador	Batch	Epochs	lr	Estructura	Epochs AG	Accuracy
I	Adam	128	3000	0.001	2 de 128	20	0.82
II	Adam	128	3000	0.01	2 de 128	20	0.76
III	Adam	128	3000	0.01	2 de 128	100	0.78
III	Adam	128	6000	0.01	3 de 128	20	0.85
IV	Adam	256	6000	0.01	3 de 128	20	0.88

Tabla 5.3. Análisis de sensibilidad para el MNIST

A continuación, se incluyen las gráficas para la mejor simulación encontrada (IV).



Figura 5.5. Evolución del loss para una red evolutiva con MNIST

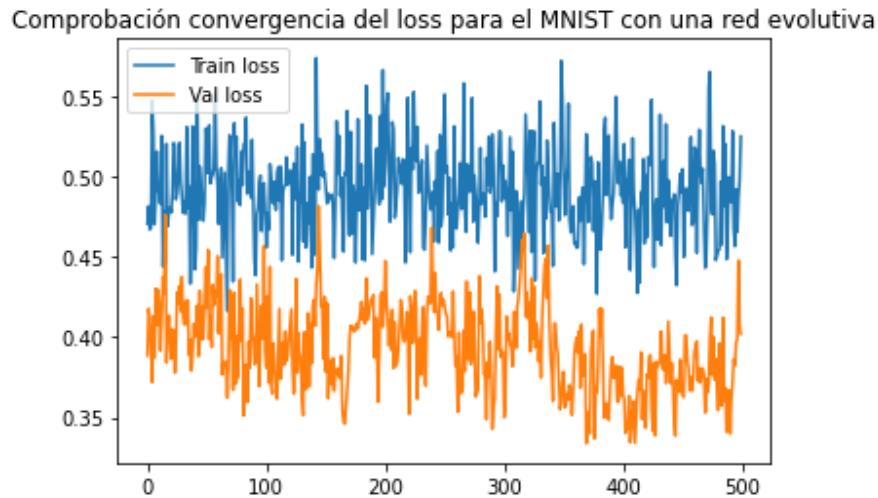


Figura 5.6. Convergencia del modelo para una red evolutiva con MNIST

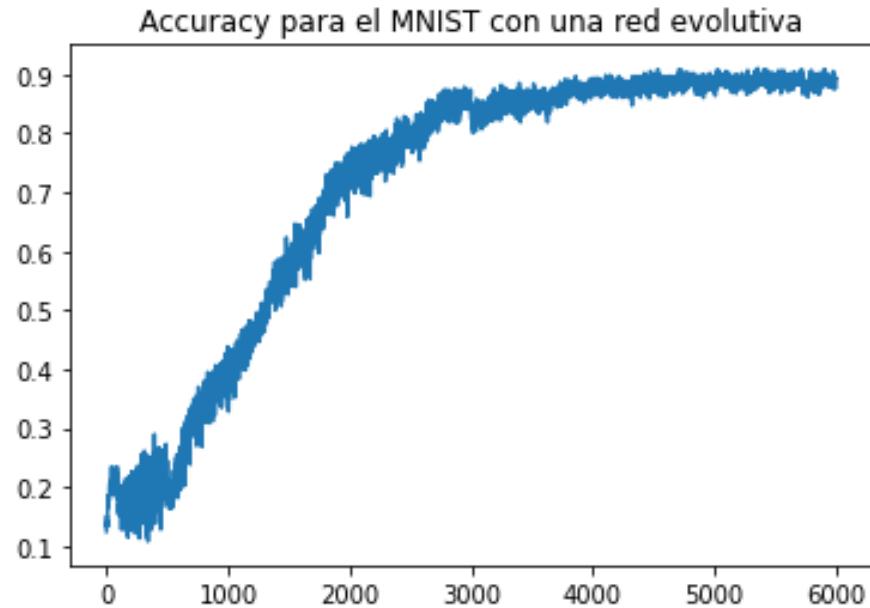


Figura 5.7. Evolución del accuracy para una red evolutiva con MNIST

Este análisis merece ser enfocado en dos direcciones. La primera de ellas, unidireccional en la que se evalúa si la red neuronal es capaz de mejorar un modelo de Keras. La segunda de ella, multidireccional en la que se estudia el impacto de cada uno de los parámetros determinantes de la red evolutiva. Con respecto al primer enfoque, los resultados obtenidos con las 4 simulaciones efectuadas son muy parecidos a los que se obtendrían entrenando simplemente un modelo de Keras, por lo tanto, no añaden ningún valor adicional. Con respecto a la segunda línea se pueden observar ciertos detalles:

- La convergencia del modelo es muy lenta. Esto se debe fundamentalmente a que cada vez que se activa la bandera del AG se cambia la estructura de la red neuronal y por lo tanto, esa adaptación a las nuevas topología requiere un tiempo adicional de entrenamiento. Además, al no ser un modelo entrenado directamente con Keras no es tan eficiente en términos computacionales.
- Si se observa la gráfica del accuracy se nota que tiene pequeñas fluctuaciones y oscilaciones a lo largo del entrenamiento. En una primera idea, se pensó que se debía al learning rate y al tamaño del batch, pues si éste primero es muy grande y el segundo pequeño, los pesos se actualizan con mayor frecuencia y esto puede dar lugar a ruidos indeseados. Sin embargo, fue un fenómeno presente en todas las figuras pintadas durante las 4 simulaciones descritas. Por lo tanto, puede deberse al propio mecanismo interno de la red neuronal evolutiva, ya que los pesos se optimizan en una doble dirección: con respecto al descenso por el gradiente y con respecto a las nuevas topologías desarrolladas. Sin embargo, se puede observar que en ningún caso es inestable el entrenamiento, por lo que dicho efecto perturbante no es un motivo de demasiada preocupación.
- En relación al número de épocas que se entrena cada red neuronal hija, se llegó a un resultado bastante sorprendente. Es indiferente que se entrenen con 20 épocas a que lo hagan con 100. Esto puede deberse a un factor determinante del algoritmo. Supongamos que tenemos como partida una red neuronal que tiene la siguiente topología: [10, 100, 50, 10] y se quiere pasar a la siguiente: [10, 100, 128, 128, 128, 10]; como se puede observar difieren notoriamente y por lo tanto, la red hija, al tratar de heredar la información de la red madre va a tener claras complicaciones, aunque se muestren los pesos. Este fenómeno ha de ser controlado para permitir que se permite avanzar en la exploración del AG.

- Por último, se pueden diseñar dos enfoques de partida. Pasar de una estructura inicial simple a otras más complejas o por el contrario, empezar con una estructura más compleja e ir quitando conexiones a medida que se avanza en el entrenamiento. Tras los resultados se puede observar que es ligeramente mejor el segundo enfoque. No obstante, se necesitarían un mayor numero de casos para verificarlo. Ambas ideas de manera intuitiva pueden funcionar de forma correcta, la primera trataría de capturar inicialmente los patrones más generales de los datos y conforme va avanzando en el entrenamiento focalizarse en detalles más concretos. Por el contrario, la segunda emularía de forma más fidedigna el comportamiento de la poda sináptica ya explicado.

Existen otros hiperparámetros que pueden ser analizados como el número de iteraciones máximas del AG y el número de iteraciones sin mejoras; el número de neuronas por capas y el número máximo de capas a partir de las cuales el AG puede encontrar nuevas soluciones óptimas; el learning rate de las redes hijas que se entrena cuando salta el AG y la proporción de pesos que se desean muestrear a partir de una distribución de probabilidad conocida. Por lo tanto, esto conduce a afirmar que el método diseñado tiene mucho margen de mejora y que sería recomendable hacer una búsqueda más exhaustiva de los hiperparámetros anteriores.

5.2. Resultados sobre el conjunto de datos de patrones técnicos

En esta sección se presentan los resultados obtenidos tras la aplicación del modelo al conjunto de series temporales para detectar patrones en la Figura. Al reto adicional de la complejidad del modelo desarrollado se le suma la dificultad de manejar un conjunto de datos severamente no balanceado. Por ello, se ha querido presentar en primer lugar los resultados del MNIST porque son más fáciles de interpretar.

Como herramienta adicional de análisis se emplean las matrices de confusión, así como una serie de indicadores como precisión, recall y F1-score que permiten realizar un sofisticado estudio de los resultados obtenidos. No se interpreta en ningún caso la evolución del accuracy ya que al ser un dataset no balanceado carece completamente de sentido.

En este escenario se utiliza como benchmark la siguiente red neuronal construida en Keras y cuyas características se adjuntan en la Figura 5.8.

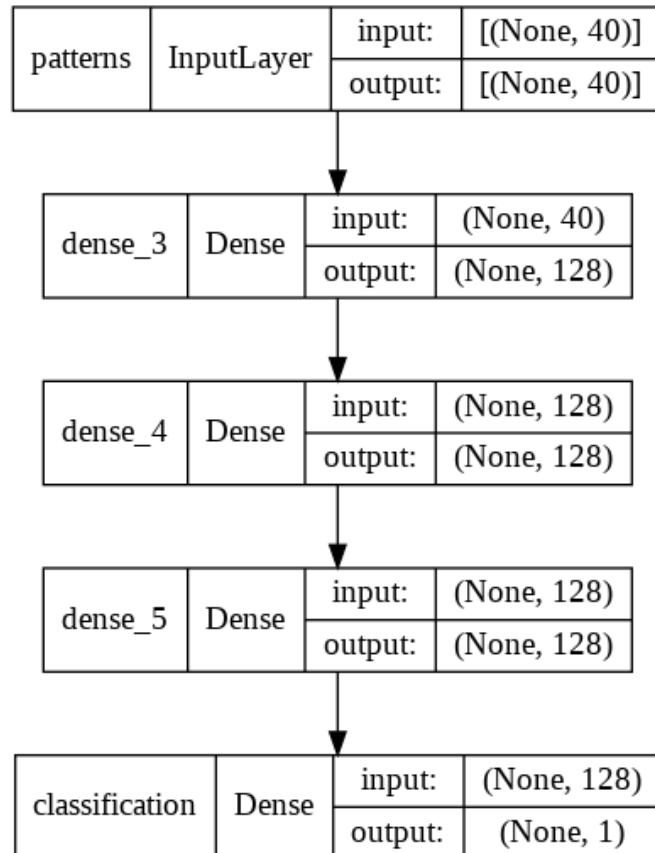


Figura 5.8. Estructura de la red densa para los patrones

Se recuerda que para valorar la calidad de estas soluciones se parte del precio de cierre normalizado de una ventana de 40 días. A su vez, dicha serie temporal sufre las debidas modificaciones tras aplicarle la técnica de SMOTE y es el resultado de ésta la que se introduce como input de ambas redes neuronales.

Asimismo, se presentan en la Tabla 5.4 los hiperparámetros con los que se entrenó el modelo.

Para presentar los resultados, se incluyen tanto los que se obtuvieron aplicando el ruido a las series temporales como los que no. Se incluyen de forma adicional a diferencia de en el caso del MNIST, ya que en este escenario la tarea de clasificación es más compleja, debido al número de muestras tan pequeñas disponibles de la clase minoritaria. De este modo, se incluyen en la Figura 5.11 los resultados obtenidos.

Hiperparámetro	Valor
Optimizador	Adam
Batch Size	128
Learning rate	0.001
Número de épocas	100

Tabla 5.4. Hiperparámetros de la red densa para los patrones

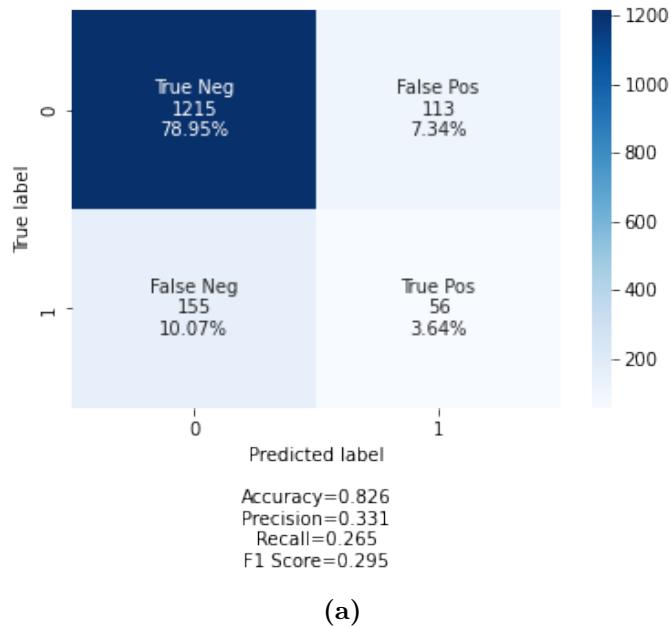


Figura 5.9. Datos sin ruido

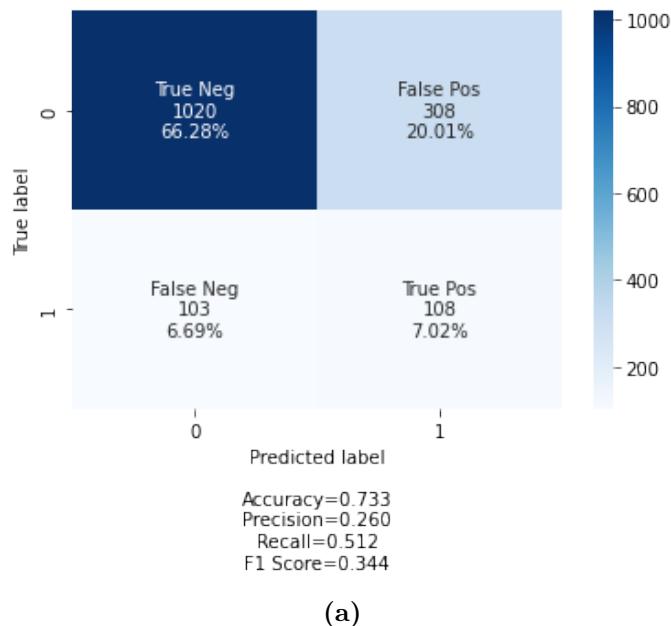


Figura 5.10. Datos con ruido

Figura 5.11. Matriz de confusión del modelo de Keras sobre los patrones

Tal y como se puede observar, al aplicarle el ruido a las series temporales se dificulta la tarea de aprendizaje y el modelo implementado empeora considerablemente. De todos las métricas disponibles se va a utilizar la precisión como benchmark, pues el objetivo fundamental es acertar el mayor número de veces en la predicción de dobles suelos.

A continuación, se adjuntan los hiperparámetros con los que se entrenó la red neuronal evolutiva.

Simulación	Optimizador	Batch	Epochs	lr	Estructura	Epochs AG	Precisión
I	Adam	256	3000	0.01	3 de 128	20	0.17
II	Adam	256	500	0.01	2 de 128	20	0.26
III	Adam	256	500	0.001	3 de 128	20	0.29

Tabla 5.5. Análisis de sensibilidad de la red evolutiva para los patrones

Una vez introducidos los hiperparámetros del modelo, se adjuntan en la Figura 5.14 las matrices de confusión para la simulación III.

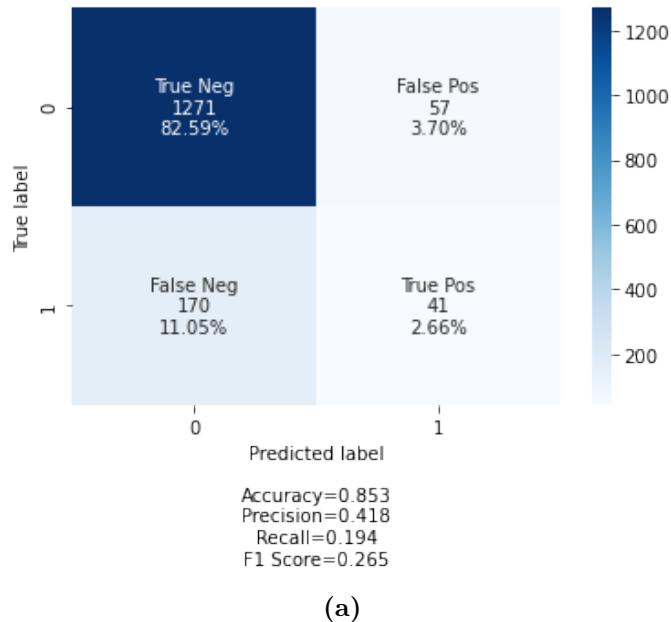


Figura 5.12. *Datos sin ruido*

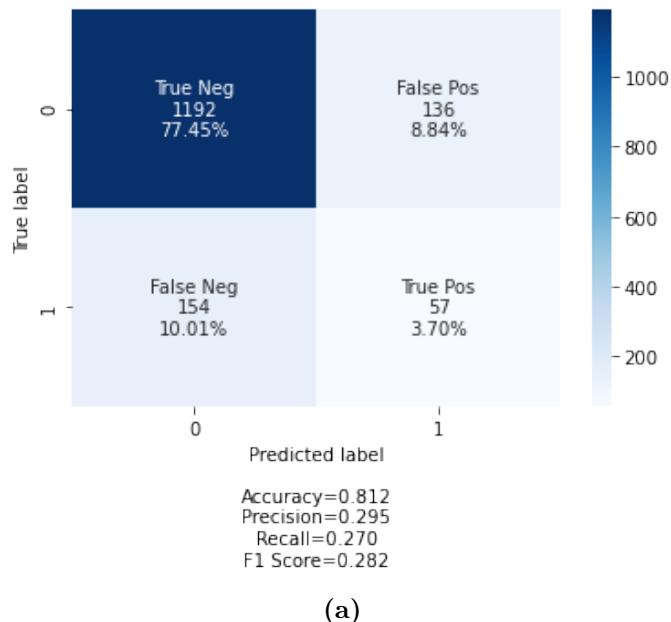


Figura 5.13. *Datos con ruido*

Figura 5.14. *Matriz de confusión del modelo de Keras sobre los patrones*

Del mismo modo que en la sección anterior el análisis de los resultados debe de ser enfocado las dos vertientes ya comentadas. Con respecto a la mejora de la tarea de aprendizaje, en este caso si que se puede afirmar que la solución aportada es mejor que el modelo diseñado por Keras ya que se obtiene una precisión del 0.418 frente a 0.331 con los datos sin ruido y del 0.295 frente a 0.26 con los datos con ruido. Esto se debe fundamentalmente a que en este caso el objetivo fundamental es detectar patrones de doble suelo, para ello la red construida en Keras trata de minimizar una función de coste que guarda relación con la métrica, pero que no es la métrica en sí ya que esta no es derivable. Por el contrario, la red evolutiva, no únicamente capaz de aprender con cada dato de entrada, sino que dicho entrenamiento se enfoca en la dirección de maximizar directamente el objetivo buscado.

Por otra parte, con respecto al análisis de sensibilidad, como primera aproximación se partió de los mismos hiperparámetros que fueron óptimos para el escenario anterior. Sin embargo, tal y como se puede observar en la Figura 5.15 el modelo estaba profundamente sobre-entrenado. Es por eso que se decidió bajar el número de épocas de la red neuronal madre.

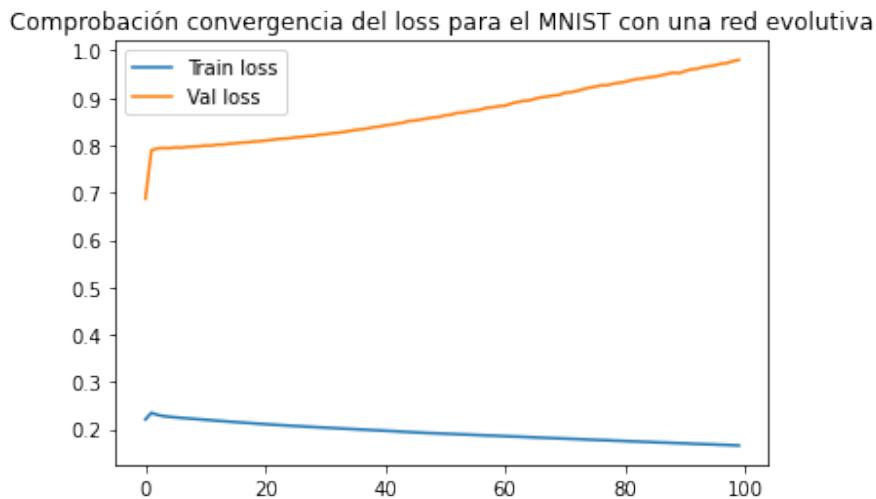


Figura 5.15. Efecto del sobreajuste en la simulación I

Por último, se observó un comportamiento extraño en cuanto a la evolución de la precisión, adjunto en la Figura 5.16 que se deja a futuros desarrollos su posterior análisis.

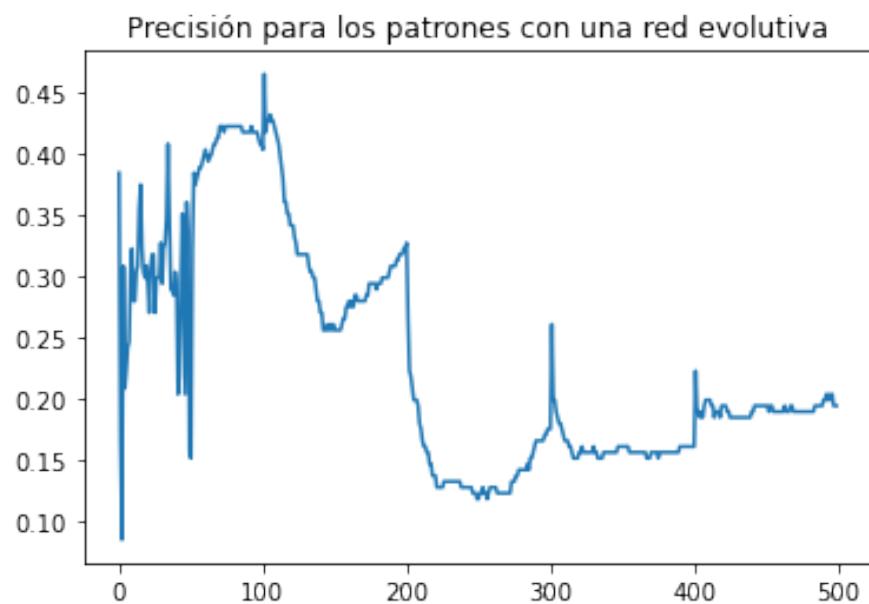


Figura 5.16. Evolución de la precisión en la red evolutiva

5.3. Entrenamiento de los modelos en la nube

Dada la gran carga computacional que requieren todas las simulaciones entrena das se hizo uso de la infraestructura de Cloud proporcionada por AWS, concretamente del servicio de SageMaker, que está especializado en entornos de IA. Se tomaron dos herramientas fundamentales: el uso de instancia de notebooks y la creación de un pipeline de entrenamiento predefinido.

En el primer caso, se utilizaron los siguientes recursos para las siguientes tareas: las dos primeras están relacionadas con el cálculo optimizado y la aceleración, mientras que las dos segundas se utilizan en entornos de sobre-carga de memoria.

Tarea	Recurso
Entrenar simulaciones para MNIST	<i>ml.c5.2xlarge</i>
Entrenar simulaciones para los patrones	<i>ml.c5.xlarge</i>
Diseñar el pipeline de la transformación de los datos	<i>ml.r5.4xlarge</i>
Entrenar el modelo de Self-Training	<i>ml.r5.4xlarge</i>

Tabla 5.6. Análisis de sensibilidad de la red evolutiva para los patrones

En el segundo caso, se utilizó una imagen de Docker para entrenar y desplegar el algoritmo personalizado en cuestión. Una vez creada la imagen se entrena el modelo en AWS SageMaker y se puede observar en el apartado de training jobs el resultado de la ejecución de los mismos. Aunque se desarrolló el pipeline para la implementación, no se llegó a utilizar dado que los notebooks eran más manejables y permitían hacer cambios *in-situ* en el código. Por otra parte, se parallelizó el cálculo de la función objetivo y por lo tanto, si se desea mejorar el rendimiento del AG basta con tener una máquina con múltiples CPU que procesen cada individuo en paralelo. Por último, con respecto al despliegue del modelo, se pretende crear un módulo en python que sea descargable mediante GitHub y de esta forma se facilita la distribución a todos los usuarios. En este caso, no tiene sentido crear ninguna aplicación web ya que el propósito del proyecto es diseñar una red evolutiva aplicable a cualquier conjunto de datos, pero como se ha observado, los hiperparámetros del modelo dependen enormemente del dataset y es esta la razón fundamental por la que se diseña un *package* de Python.

Capítulo 6

Conclusiones y futuros desarrollos

Con este trabajo se ha presentado una nueva forma de entender el concepto de red neuronal, desafiando las bases que sustentan el entrenamiento tradicional y proponiendo un nuevo método que permita hacer un entrenamiento más holístico optimizando al mismo tiempo la estructura de la red y sus parámetros internos. Se han validado los resultados en dos datasets de características muy distintas y se ha realizado un análisis de sensibilidad de los mismos en ambos casos.

Se ha probado que el modelo de herencia de los pesos da resultados que podrían llegar a ser bastante satisfactorios, pero es necesario realizar una búsqueda más exhaustiva de los hiperparámetros del mismo. Asimismo, para comprobar si realmente el método añade valor se debería diseñar un AG que tratara de optimizar las estructuras de los modelos de Keras, sin modificarlas durante el entrenamiento, para así estimar si el hecho de que la red se vaya transformando durante su tarea de aprendizaje realmente aumenta el rendimiento del modelo.

Por último, con respecto a los futuros desarrollos, sería muy recomendable extender el método desarrollado a otro tipos de redes neuronales que son más efectivas para clasificar las imágenes, como las redes convolucionales. También se deja como línea abierta de investigación la estimación del agotamiento de la capacidad de aprendizaje de una red neuronal evolutiva, pues si se perfecciona cuándo el AG debe ser activado, el modelo en conjunto dará resultados mejores.

Capítulo 7

Bibliografía

- [1] Editorial, Hybrid learning machine, Neurocomputing, 72, 2729-2730 (2009).
- [2] W. Deng, R. Chen, B. He, Y. Liu, L. Yin and J. Guo, A novel two-stage hybrid swarm intelligence optimization algorithm and application, Soft Computing, 16, 10, 1707-1722 (2012).
- [3] M.H.Z. Fazel and R. Gamasae, Type-2 fuzzy hybrid expert system for prediction of tardiness in scheduling of steel continuous casting process, Soft Comput. 16, 8, 1287-1302 (2012).
- [4] D. Jia, G. Zhang, B. Qu and M. Khurram, A hybrid particle swarm optimization algorithm for high-dimensional problems, Comput. Ind. Eng. 61, 1117-1122 (2011).
- [5] M. Khashei, M. Bijari and H. Reza, Combining seasonal ARIMA models with computational intelligence techniques for time series forecasting, Soft Comput. 16, 1091-1105 (2012).
- [6] U. Guvenc, S. Duman, B. Saracoglu and A. Ozturk, A hybrid GAPSO approach based on similarity for various types of economic dispatch problems, Electron Electr. Eng. Kaunas: Technologija 2, 108, 109-114 (2011).
- [7] G. Acampora, C. Lee, A. Vitiello and M. Wang, Evaluating cardiac health through semantic soft computing techniques, Soft Comput. 16, 1165-1181 (2012).
- [8] M. Patruikic, R.R. Milan, B. Jakovljevic and V. Dapic, Electric energy fore-

casting in crude oil processing using support vector machines and particle swarm optimization. In the Proceedings of 9th Symposium on Neural Network Application in Electrical Engineering, Faculty of Electrical Engineering, university of Belgrade, Serbia (2008).

[9] B.H.M. Sadeghi, ABP-neural network predictor model for plastic injection molding process. *J. Mater. Process Technol.* 103, 3, 411-416 (2000).

[10] T.T. Chow, G.Q. Zhang, Z. Lin and C.L. Song, Global optimization of absorption chiller system by genetic algorithm and neural network, *Energ. Build.* 34, 1, 103-109 (2002)

[11] D.F. Cook, C.T. Ragsdale and R.L. Major, Combining a neural network with a geneticalgorithm for process parameter optimization. *Eng. Appl. Artif. Intell.* 13, 4, 391-396(2000).

[12] S.L.B. Woll and D.J. Cooperf, Pattern-based closed-loop quality control for the injection molding process, *Polym. Eng. Sci.* 37, 5, 801-812 (1997).

[13] C.R. Chen and H.S. Ramaswamy, Modeling and optimization of variable retort temperature (VRT) thermal processing using coupled neural networks and genetic algorithms, *J. Food Eng.* 53, 3, 209-220 (2002).

[14] J. David, S. Darrell, W. Larry and J. Eshelman, Combinations of Genetic Algorithms and Neural Networks:A Survey of the State of the Art. In: proceedings of IEEE international workshop on Communication, Networking Broadcasting; Computing Processing (Hardware/Software), 1-37 (1992).

[15] Y. Nagata and C.K. Hoong, Optimization of a fermentation medium using neural networks and genetic algorithms. *Biotechnol. Lett.* 25, 1837-1842 (2003).

[16] V. Estivill-Castro, The design of competitive algorithms via genetic algorithmsComputing and Information, 1993. In: Proceedings Fifth International Conference on ICCI, 305-309 (1993).

[17] B.K. Wong, T.A.E. Bodnovich and Y. Selvi, A bibliography of neural network applications research: 1988-1994. *Expert Syst.* 12, 253-61 (1995).

- [18] P. Werbos, The roots of the backpropagation: from ordered derivatives to neural networks and political fore-casting, New York: John Wiley and Sons, Inc (1993).
- [19] D.E. Rumelhart and J.L. McClelland, In: Parallel distributed processing: explorations in the theory of cognition, vol.1. Cambridge, MA: MIT Press (1986).
- [20] L. Gu, W. Liu-ying'v, C. Gui-ming and H. Shao-chun, Parameters Optimization of Plasma Hardening Process Using Genetic Algorithm and Neural Network, J. Iron Steel Res. Int. 18, 12, 57-64 (2011).
- [21] G. Zhang, B.P. Eddy and M.Y. Hu, Forecasting with artificial neural networks: The state of the art, Int. J. Forecasting, 14, 35-62 (1998).
- [22] G.R. Finnie, G.E. Witting and J.M. Desharnais, A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning and regression models, J. Syst. Softw. 39, 3, 281-9 (1997).
- [23] R.B. Boozarjomehry and W.Y. Svrcek, Automatic design of neural network structures, Comput. Chem. Eng. 25, 1075- 1088 (2001).
- [24] M.M. Fischer and Y. Leung, A genetic-algorithms based evolutionary computational neural network for modelling spatial interaction data, Ann. Reg. Sci. 32, 437-458 (1998).
- [25] M. Versace, R. Bhatt and O. Hinds, M. Shiffer, Predicting the exchange traded fund DIA with a combination of genetic algorithms and neural networks, Expert Syst. Appl. 27, 417- 425(2004).
- [26] J. Cheng and Q.S. Li, Reliability analysis of structures using artificial neural network based genetic algorithms, Comput. Methods Appl. Mech. Engrg. 197, 3742-3750 (2008).
- [27] S.M.R. Longhmanian, H. Jamaluddin, R. Ahmad, R. Yusof and M. Khalid, Structure optimization of neural network for dynamic system modeling using multi-objective genetic algorithm, Neural Comput. Appl. 21, 1281-1295 (2012).
- [28] M. Mitchell, An introduction to genetic algorithms. MIT press: Massachusetts (1996).

setts (1998).

[29] D.A. Coley, An introduction to genetic algorithms for scientist and engineers, Worldscientific publishing Co. Pte. Ltd: Singapore (1999).

[30] A.F. Shapiro, The merging of neural networks, fuzzy logic, and genetic algorithms, *Insur. Math. Econ.* 31, 115-131 (2002).

[31] C. Huang, L. Chen, Y.Chen and F.M. Chang, Evaluating the process of a genetic algorithm to improve the back-propagation network: A Monte Carlo study. *Expert Syst.Appl.* 36, 1459-1465 (2009).

[32] D. Venkatesan, K. Kannan and R. Saravanan, A genetic algorithm-based artificial neural network model for the optimization of machining processes, *Neural Comput. Appl.* 18, 135-140 (2009).

[33] D.J. Montana and L. D. Davis, Training feedforward networks using genetic algorithms, In: Proceedings of the International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 762-767 (1989).

[34] H. Xiao and Y. Tian, Prediction of mine coal layer spontaneous combustion danger based on genetic algorithm and BP neural networks, *Procedia Eng.* 26, 139-146(2011). [35] S. Zhou, L. Kang, M. Cheng and B. Cao, Optimal Sizing of Energy Storage System in Solar Energy Electric Vehicle Using Genetic Algorithm and Neural Network. SpringerVerlag Berlin Heidelberg 720-729 (2007).

[36] R.S. Sexton, R.E. Dorsey and N.A. Sikander, Simultaneous optimization of neural network function and architecture algorithm, *Decis. Support Syst.* 36, 283-296 (2004). [37] Y.A. Abbas and M.M. Aqel, Pattern recognition using multilayer neural-genetic algorithm, *Neurocomputing*, 51, 237-247 (2003).

[38] D. Guyer and X. Yang, Use of genetic artificial neural networks and spectral imaging for defect detection on cherries, *Comput. Electr. Agr.* 29, 179-194 (2000).

[39] R. Furtuna, S. Curteanu, F. Leon, An elitist non-dominated sorting genetic algorithm enhanced with a neural network applied to the multi-objective optimization of a polysiloxane synthesis process, *Eng. Appl. Artif. Intell.* 24, 772-785 (2011).

[40] G. Yazc, O. Polat and T. Yildrm, Genetic Optimizations for Radial Basis

Function and General Regression Neural Networks, Springer-Verlag Berlin Heidelberg 348-356 (2006).

[41] B. Curry and P. Morgan, Neural networks: a need for caution, *Int. J. Manage. Sci.* 25, 123-133 (1997).

[42] C. Su and T. Chiang, Optimizing the IC wire bonding process using a neural networks/genetic algorithms approach, *J. Intell. Manuf.* 14, 229-238 (2003).

[43] D. Season, Non linear PLS using genetic programming, PhD thesis submitted to school of chemical engineering and advance materials, University of Newcastle, (2005).

[44] A.J. Jones, genetic algorithms and their applications to the design of neural networks, *Neural Comput. Appl.* 1, 32- 45(1993).

[45] G.E. Robins, M.D. Plumbley, J.C. Hughes, F. Fallside and R. Prager, Generation and adaptation of neural networks by evolutionary techniques (GANNET). *Neural Comput. Appl.* 1, 23-31 (1993).

[46] C. Harpham, C.W. Dawson and M.R. Brown, A review of genetic algorithms applied to training radial basis function networks, *Neural Comput. Appl.* 13, 193-201 (2004).

[47] J.H. Holland, Adaptation in natural and artificial systems, MIT Press: Massachusetts. Reprinted in 1998 (1975).

[48] M. Sakawa, Genetic algorithms and fuzzy multiobjective optimization, Kluwer Academic Publishers: Dordrecht(2002).

[49] D.E. Goldberg, Genetic algorithms in search, optimization, machine learning. Addison Wesley: Reading (1989).

[50] P. Guo, X. Wang and Y. Han, The enhanced genetic algorithms for the optimization design, In: IEEE International Conference on Biomedical Engineering Information, 2990- 2994(2010).

[51] M. Hamdan, A heterogeneous framework for the global parallelization of genetic algorithms, *Int. Arab J. Inf. Tech.* 5, 2, 192199 (2008).

- [52] C.T. Capraro, I. Bradaric, G.T. Capraro and L.T. Kong, Using genetic algorithms for radar waveformselection, Radar Conference, RADAR '08. IEEE, 1-6 (2008).
- [53] S.N. Sivanandam andS.N. Deepa, Introduction to genetic algorithms, Springer Verlag: New York (2008).
- [54] S. Haykin, Neural networks and learning machines (3rd edn), Pearson Education, Inc. New Jersey(2009).
- [55] D. Zhang andL. Zhou, Discovering Golden Nuggets: Data Mining in Financial Application,IEEE Trans. Syst. Man Cybern. C Appl. Rev. 34,4, 513-522 (2004).
- [56] D.E. Goldberg, Simple genetic algorithms and the minimal deceptive problem, In L. D. Davis, ed.,Genetic Algorithms and Simulated Annealing. Morgan Kaufmann (1987).
- [57] D.M. Vose andG.E. Liepins, Punctuated equilibria in genetic search, Compl. Syst. 5, 31-44 (1991).
- [58] T.E. Davis and J.C. Principe, A simulated annealinglike convergence theory for the simple genetic algorithm. In R. K. Belew and L. B. Booker, eds., Proceedings of the Fourth International Conference on Genetic Algorithms. Morgan Kaufmann (1991).
- [59] L.D. Whitley, An executable model of a simple genetic algorithm, In L. D. Whitley, 2ed., Foundations of Genetic Algorithms, Morgan Kaufmann (1993).
- [60] Z. Laboudi andS. Chikhi, Comparison of genetic algorithms and quantum genetic algorithms, Int. Arab J. Inf. Tech. 9,3, 243-249(2012).
- [61] Z. Michalewicz, Genetic algorithms + data structures = evolution program, Springer Verlag: New York (1999).
- [62] S.D. Ahmed, F. Bendimerad, L. Merad andS.D. Mohammed, Genetic algorithms based synthesis of three dimensional microstrip arrays, Int. Arab J. Inf.

Tech.1, 80-84 (2004).

[63] H. Aguir, H. BelHadjSalah and R. Hamblin, Parameter identification of an elasto-plastic behaviour using artificial neural networks/genetic algorithm method, Mater Design, 32, 48-53 (2011).

[64] F. Baishan, C. Hongwen, X. Xiaolan, W. Ning and H. Zongding, Using genetic algorithms coupling neural networks in a study of xylitol production: medium optimization, Process Biochem. 38, 979-985 (2003).

[65] K.Y. Chan, C.K. Kwong and Y.C. Tsim, Modelling and optimization of fluid dispensing for electronic packaging using neural fuzzy networks and genetic algorithms, Eng. Appl. Artif. Intell. 23, 18-26 (2010).

