UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Búsqueda automática de arquitecturas profundas para imagen hiperespectrales

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Búsqueda automática de arquitecturas profundas para imagen hiperespectrales

Autor: Miguel Ángel González Jorge

Tutor: Juan Mario Haut Hurtado

Co-Tutor: Mercedes Eugenia Paoletti Ávila

AGREDECIMIENTOS

Este trabajo se ha realizado gracias al apoyo de muchas personas, primeramente a todos aquellos profesores que a lo largo del grado me han proporcionado los conocimientos necesarios, a mis familiares y amigos que diariamente me han animado a seguir, sin ellos este trabajo de investigación no podría haberse convertido en realidad.

Por último, me gustaría agradecer especialmente a mi director, Juan Mario Haut, por sus incansables horas de explicaciones y los conocimientos enseñados en la creación de este documento, convirtiéndose en un referente para mí. Así como a Mercedes Eugenia Paoletti, quien ha proporcionado enseñanzas a nivel técnico y personal. Gracias a ellos este documento puede estar en vuestras manos.

Resumen

Las imágenes hiperespectrales capturan un amplio rango del espectro electromagnético en cientos de bandas espectrales, formando una estructura tridimensional para su procesamiento y análisis. En el marco de la Observación Remota de la Tierra, estas imágenes se obtienen a través de técnicas de teledetección, donde los métodos basados en deep learning han demostrado su capacidad de clasificar de forma precisa esta información. En este contexto, las redes neuronales convolucionales son usadas ampliamente, estableciéndose como el estado-del-arte actual. Para operarlas se requiere de un esfuerzo significativo, tanto para diseñar una arquitectura adecuada que se adapte al problema, como para seleccionar los hiperparámetros del modelo, tales como: el número de capas y neuronas, tipo de operaciones, etc. En este TFG profundizaremos en el uso de diferentes técnicas para la búsqueda automática de las mejores arquitecturas neuronales para un problema dado, una tarea que previamente se realizaba de forma manual, proponiendo nuevos algoritmos sobre imágenes hiperespectrales ampliamente utilizadas por la comunidad.

Abstract

Hyperspectral imaging captures a wide range of the electromagnetic spectrum in hundreds of spectral bands, forming a three-dimensional structure that must be processed and analyzed. In the framework of Earth Observation, these images are obtained through remote sensing techniques, where deep learning-based methods have demonstrated their ability to accurately classify this information. In this context, convolutional neural networks are widely used, establishing themselves as the current state-of-the-art. To operate them, a significant effort is required both to design an appropriate architecture to suit the problem and to select the model hyperparameters, such as: the number of layers and neurons, type of operations, etc. In this project we will explore in depth the use of different techniques for the automatic search of the best neural architectures for a given problem, a task that was previously performed manually, proposing new algorithms on hyperspectral images widely used by the community.

Índice

Índice de tablas

Índice de figuras

1. INTRODUCCIÓN

La teledetección consiste en la adquisición de información acerca de objetos ubicados en la superficie terrestre, o de la propia cobertura terrestre, sin necesidad de emplear instrumentos que estén en contacto directo con la misma, a través de sensores remotos ubicados sobre plataformas aéreas o satelitales [1]. Para la recopilación de esta información, se produce un flujo de radiación que es captada y medida por los sensores. Dependiendo del tipo del sensor, podemos hablar de sensores pasivos, encargados de recopilar el flujo de radiación solar (reflejada por los objetos de la superficie) o el flujo de radiación terrestre (emitida por los propios objetos), o activos, que emiten su propio flujo, capturando el grado de absorción y reflexión de los objetos. Dicha radiación es entonces medida, codificada y almacenada en formato digital para su posterior procesamiento, generando lo que se conoce como datos o imágenes de teledetección. En particular, este trabajo se centrará en imágenes ópticas, capturadas con sensores pasivos que miden el comportamiento de los materiales terrestres ante la luz solar.

Este tipo de datos (datos ópticos) contiene una rica información espacial, espectral y temporal del área geográfica observada. Enfocándonos en la dimensión espectral, podemos clasificar dichos datos en tres grandes tipos dependiendo de la información espectral que contengan. Por un lado están las imágenes *RGB*, que codifican la información en tres bandas relativas a los colores rojo, verde y azul, siendo éstas las imágenes comunes que acostumbramos a ver. Aunque el ojo humano solo puede percibir las ondas comprendidas entre los 390 y 750 nm del espectro electromagnético, existen diferentes rangos de frecuencia (como el infrarrojo cercano -*near infrared*, NIR- y el infrarrojo de onda corta -*shortwave infrared*, SWIR-)que aportan una gran información fuera del espectro visible, lo que nos lleva a las imágenes multiespectrales, que contienen decenas de bandas espectrales discretas (normalmente menos de 30 bandas), y las imágenes hiperespectrales, que continen cientos de bandas estrechas y contínuas (normalmente entre 30 y cientos de bandas).

Las imágenes hiperespectrales proporcionan una mayor caracterización de los

materiales que componen la escena, pues cada material tiene una firma espectral única de acuerdo a sus características fisico-químicas que definen el patrón de absorción-reflexión de la luz solar capturado por el sensor hiperespectral (espectrómetro), permitiendo su identificación de manera inequívoca. Al mismo tiempo, se obtiene el contexto espacial mediante una malla 2D de píxeles que proporciona una abundante información estructural de la superficie capturada, generando como resultado volúmenes de datos tridimensionales que contienen información tanto espectral como espacial, con cientos e incluso miles de canales espectrales, en comparación con las tres únicas bandas de una imagen RGB. Toda esta información es captada mediante espectrómetros ubicados en satélites o plataformas aerotransportadas como por ejemplo *LandSat*, *Sentinel*, *AVIRIS* y *SPOT*, que además de contener los sensores y los medios de captura, almacenan y transmiten esos datos al segmento de tierra, donde serán procesados por los dispositivos de cómputo.

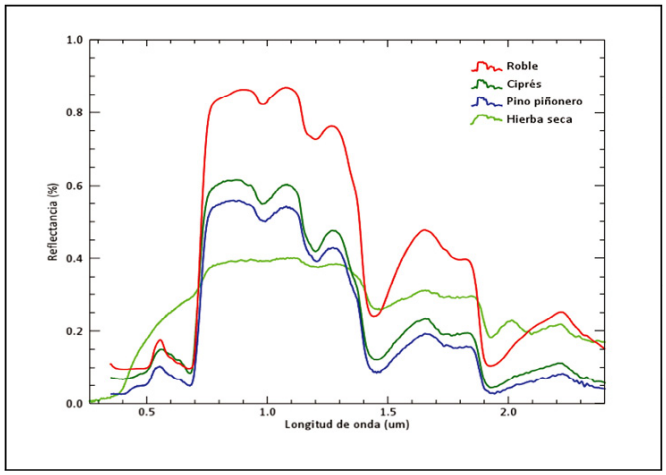


Figura 1: Distintas firmas espectrales de vegetación. Fuente: Biblioteca espectral del USGS. Spectral reflectance curves of some vegetation types.

En la figura 1 se observa la firma espectral de algunos tipos de vegetación obtenidas de imágenes hiperespectrales. Como se puede observar, cada tipo de vegetal exhibe una reflectividad diferente y única, lo que permite caracterizarlo fácilmente. Sin embargo, debido a la baja resolución espacial de este tipo de imágenes es posible que co-existan diferentes tipos de materiales en un mismo píxel, con lo que la firma espectral de ese

píxel será una combinación o mezcla de la firma espectral de estos materiales. Para obtener la representación pura de los materiales en ese píxel, existen métodos para el desmezclado de las firmas espectrales, como los autoencoders reductores de ruido, una técnica basada en el aprendizaje profundo.

Gracias al contenido de información que presentan las imágenes hiperespectrales, éstas tienen un valor importante para su uso en ciertas aplicaciones, como la creación de mapas topográficos, monitorización de la desertificación de ciertas zonas de la tierra, laerosión de zonas costeras, la geodesia para el estudio de perturbaciones del campo gravitatorio terrestre y la observación terrestre para detectar la deforestación, el estado de salud de tierras de cultivo, búsqueda de zonas mineras, etc. Esto ha tenido como consecuencia que existan una mayor cantidad de datos de este tipo, lo que precisa de la búsqueda de mejores técnicas para su almacenamiento y procesamiento.

En las primeras etapas de la búsqueda de algoritmos eficaces para la clasificación de imágenes hiperespectrales, muchos métodos se enfocaban en el análisis de la firma espectral de manera independiente, por lo cual, numerosos métodos de clasificación, como máquinas de vectores de soporte (*support vector machine* o *SVM*), regresión logística multinomial (*multinomial logistic regression* o *MLR*), y perceptrón multicapa (*multilayer perceptron* o *MLP*) han sido propuestos y ampliamente utilizados para este fin [2]. También existen otros enfoques de análisis centrados en la extracción de características o técnicas de reducción de dimensiones, como el análisis de componentes principales (*principal component analysis* o *PCA*). Sin embargo, los métodos anteriores presentan limitaciones al no considerar el contexto espacial de cada píxel, generando así en los mapas de clasificación el conocido ruido “sal y pimienta”.

En este aspecto, el uso de las redes neuronales artificiales (*artificial neural network* o *ANN*), especialmente las redes convolucionales (*convolutional neural network* o *CNN*) [3], ha crecido notablemente debido a la gran capacidad que tienen para la clasificación y el reconocimiento de patrones, pues establecen relaciones automáticas espectrales, espaciales y espectro-espaciales [4]. Su gran poder de generalización las ha posicionado como el estado-de-arte actual dentro del campo de la visión artificial y

del procesamiento de imágenes, alcanzando rendimientos y resultados de precisión nunca antes observados dentro de la clasificación de la imagen hiperespectral [5]. Sin embargo, el hecho de que este tipo de redes tengan un funcionamiento tipo *caja negra*[1] [6], no permite conocer al programador qué es lo que está haciendo exactamente el modelo neuronal en cada capa, ni cómo afecta el diseño, la arquitectura y los hiperparámetros al funcionamiento del mismo. Al final, esto se traduce en que la arquitectura de la red neuronal debe ser diseñada manualmente por expertos y sus hiperparámetros deben ser seleccionados por prueba y error para conseguir una solución óptima en una fase de diseño laboriosa, que consume mucho tiempo y que depende totalmente de los conocimientos del experto. Sin embargo, el diseño que tenga la red es la clave para una clasificación precisa sobre el conjunto de datos, haciendo necesario un exámen exhaustivo del modelo más allá de los conocimientos del programador. A causa de todo ello, se ha dedicado un gran esfuerzo para la construcción de arquitecturas neuronales eficaces, permitiendo reducir notablemente algunos problemas [7, 5].

Actualmente existen algunas estrategias para la búsqueda automática de arquitecturas neuronales (*NAS*), incluyendo la búsqueda aleatoria, optimización bayesiana, aprendizaje por refuerzo (*reinforcement learning*), métodos evolutivos y métodos basados en el gradiente [8, 9]. En este trabajo se explorarán tres nuevas soluciones para la búsqueda automática de arquitecturas óptimas de redes neuronales que ayuden a reducir el timpo de diseño del modelo neuronal:

- Método evolutivo utilizando un algoritmo genético propio, adaptado para la búsqueda de arquitecturas neuronales, clasificadores de imágenes hiperespectrales, capaz de generar modelos con una adecuada tasa de acierto.

- Búsqueda de arquitectura diferenciable (*Differentiable Architecture Search, DARTS*).

[1] Es un término técnico que se aplica a la forma en la que se estudia un sistema, principalmente en sus características de entrada y salida. Un algoritmo de caja negra es aquel donde el usuario no puede ver la forma interna del algoritmo.

- Búsqueda mejorada de arquitectura diferenciable para HSI (*Improved Differentiable Architecture Search for HSI, iDARTS HSI*, como una mejora de *DARTS*.

El resto de este documento se organiza de la siguiente manera. La sección II describe conceptos previos del aprendizaje profundo. La sección III proporciona una visión en profundidad de los objetivos principales que se abordan en este proyecto. La sección IV ofrece un breve resumen del estado del arte en cuanto a la búsqueda automática de arquitecturas neuronales. La sección V explica en detalle la búsqueda de arquitecturas con modelos mínimamente predefinidos, mediante el uso de las tres nuevas soluciones propuestas en este trabajo. La implementación de estos métodos se descibe en la sección VI, los resultados experimentales se informan en la sección VII, y en la sección VIII se presentan las conclusiones alcanzadas y prosibles trabajos futuros.

2. CONCEPTOS PREVIOS

Las redes neuronales artificiales son una abstracción computacional del funcionamiento de las neuronas biológicas de nuestro cerebro. A continuación se detalla una introducción desde el modelo neuronal más simple, el perceptrón, hasta redes neuronales más complejas, las redes convolucionales.

2.1. La neurona: el perceptrón simple

La neurona es la unidad básica de procesamiento que existe en una red neuronal, la cual tiene conexiones de entrada, por donde reciben estímulos externos que son los valores de entrada [10]. Con estos valores, la neurona genera un valor de salida, por lo que una neurona no es más que una función matemática. Ésta utiliza todos los valores de entrada para realizar una suma ponderada de ellos, donde la ponderación de cada una de las entradas viene dada por el peso asignado a cada una de las conexiones de entrada, a modo de peso sináptico. Análogamente, cada peso define con qué intensidad una variable de entrada afecta a la neurona.

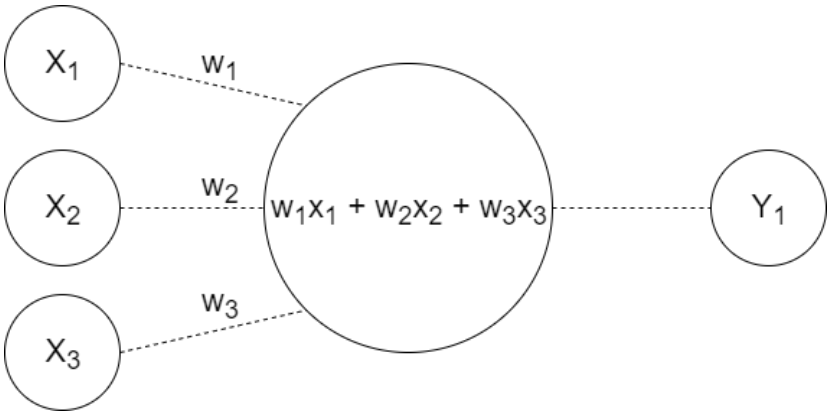


Figura 2: Estructura básica de una neurona.

Como puede verse en la figura 2, una neurona en su interior, realiza un modelo de regresión lineal, donde las variables de entradas definen una recta o un hiperplano, los cuales pueden variar su inclinación variando las entradas. También existe un término independiente, que permite mover verticalmente la función resultado, llamado sesgo

(*bias*), representado como otra conexión a la neurona, donde su valor de entrada siempre es 1, pudiéndose controlar manipulando el parámetro de sesgo.

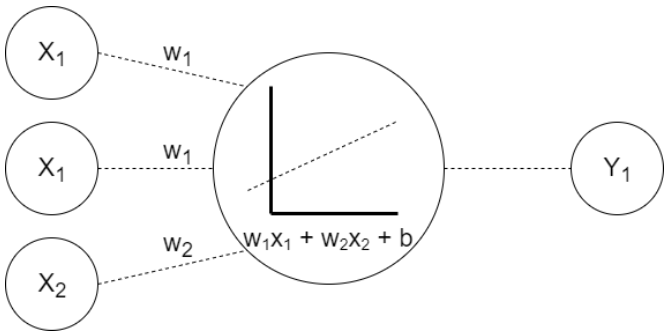


Figura 3: Estructura de una neurona siguiendo el modelo de regresión lineal.

$$y = \sum x_i w_i + b \tag{1}$$

Así, como se dijo anteriormente, el valor resultado de una neurona no es más que una suma ponderada de sus entradas más el valor de sesgo, representado en la ecuación 1, donde y es el valor de salida, x_i es la entrada actual de la neurona y w_i su ponderación.

A esta estructura de red neuronal con una sóla neurona en su capa, se le conoce como perceptrón. Usar un único perceptrón no garantiza la convergencia para problemas que no se puedan separar linealmente, por ejemplo, una nube de puntos distribuidos en forma de operación XOR[11].

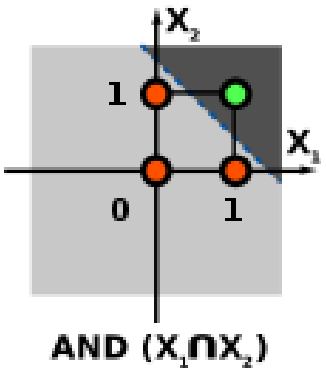


Figura 4: Puerta AND.

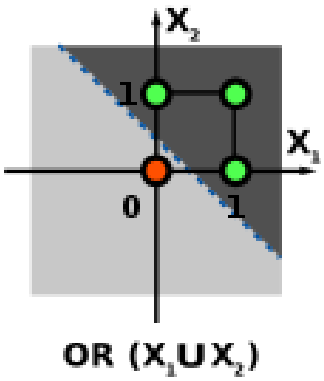


Figura 5: Puerta OR.

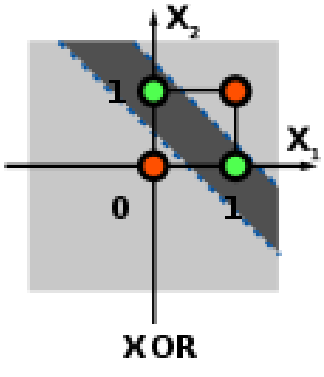


Figura 6: Puerta XOR.

Tabla 1: Nube de puntos de puertas lógicas.

2.2. La función de activación

Al conectar de forma secuencial varias neuronas, se obtiene lo que se conoce como aprendizaje jerarquizado, es decir, que cada neurona se especializa en algo cada vez más abstracto, dando lugar al aprendizaje profundo (*deep learning*).

Enlazar estas neuronas de forma secuencial, matemáticamente hablando es concatenar varias operaciones de regresión lineal, y está demostrado que sumar varias operaciones de regresión lineal, equivale a haber hecho una única operación de regresión lineal equivalente, lo que daría como resultado que la red colapse para formar una única neurona [12].

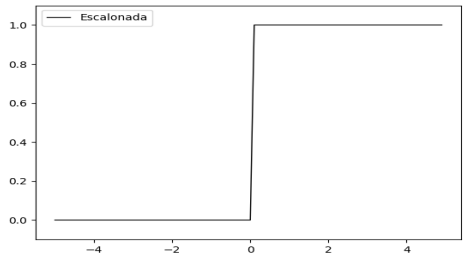
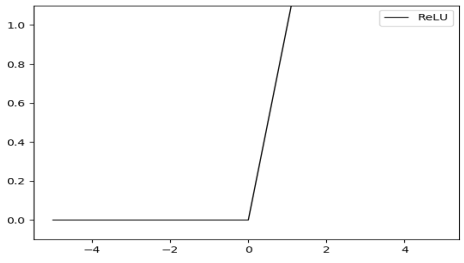
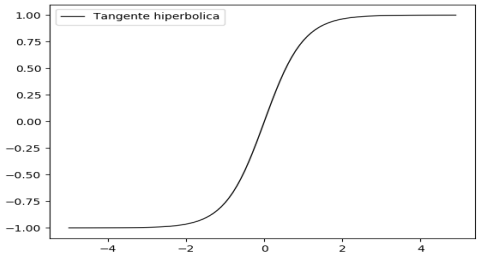
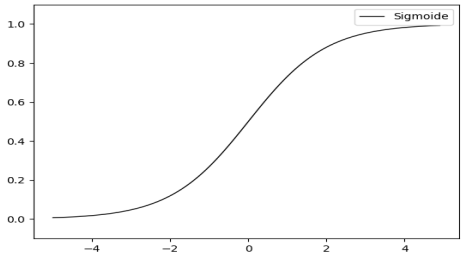
Escalonada	Unidad de rectificada lineal (*ReLU*)
	
$f(x) = \begin{cases} 0 & \text{para } x < 0 \\ 1 & \text{para } x \geq 0 \end{cases}$	$f(x) = \begin{cases} 0 & \text{para } x < 0 \\ x & \text{para } x \geq 0 \end{cases}$
Tangente hiperbólica	**Sigmoide**
	
$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

Tabla 2: Funciones de activación.

Para conseguir que la red no colapse, es necesario añadir una manipulación no

lineal a la salida de estas neuronas, lo que provoca que sufran una distorsión, la función de activación [13]. De este modo, podemos encadenar de forma efectiva la computación de varias neuronas. Pueden verse algunos ejemplos de funciones de activación ampliamente usadas en la tabla 2.

2.3. El perceptrón multicapa

El perceptrón multicapa es una red neuronal formada por varias capas, compuestas de neuronas, lo que permite resolver problemas que no son linealmente separables. Puede ser **totalmente conectado** (*fully-connected*), donde cada neurona de la capa actual es entrada de todas las neuronas de la capa posterior, o **parcialmente conectado**, donde cada neurona de la capa actual es entrada de una serie de neuronas (región) de la capa siguiente. Estas capas se pueden clasificar de tres formas:

Capa de entrada Formada por aquellas neuronas que introducen los patrones de entrada en la red. En esta capa no se produce procesamiento.

Capa oculta Se denominan a todas las capas cuyas entradas vienen de una capa anterior y su salida es entrada de una capa posterior.

Capa de salida Es la capa cuyo conjunto de neuronas se corresponde con la salida de toda la red.

Existen ciertas limitaciones en el perceptrón multicapa [10]:

No extrapola bien. Si la red es entrenada de manera incorrecta o insuficiente, las salidas pueden ser imprecisas.

La existencia de mínimos locales. Esto dificulta su entrenamiento, pues una vez alcanzado un mínimo, el entrenamiento se detiene aunque no haya alcanzado la tasa de convergencia fijada. Para evitar esto, se puede cambiar la topología de la red (número de capas y de neuronas), comenzar el entrenamiento con pesos iniciales diferentes, modificar los parámetros de aprendizaje, modificar el conjunto de entrenamiento o presentar el conjunto de entrenamiento en otro orden.

2.4. La retropropagación

El algoritmo utilizado para el entrenamiento de un perceptrón simple no es aplicable para el entrenamiento de redes neuronales más complejas, por lo que a través de esta necesidad se crea el algoritmo denominado propagación hacia atrás, el cual es un algoritmo utilizado para entrenar estas redes, también conocido como retropropagación del error [14]. La técnica utilizada para la ejecución del algoritmo es la llamada descenso del gradiente.

Para el cómputo del gradiente, calculamos el error del modelo en el punto en el que nos encontramos, y calculamos las derivadas parciales en dicho punto. Con esto se obtiene un vector de direcciones hacia donde el error se incrementa, lo que se llama el gradiente. Moviéndonos iterativamente en la dirección contraria, podemos ir reduciendo el error del modelo como si estuviéramos bajando una montaña, ver figura 7.

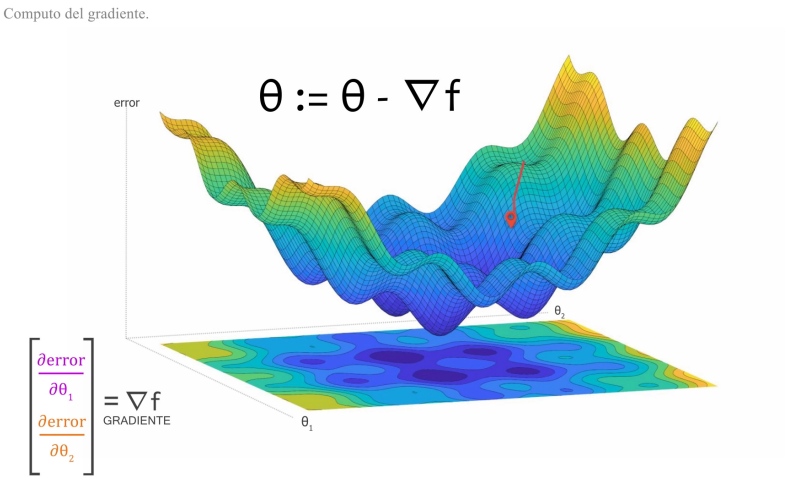


Figura 7: Representación del descenso del gradiente.

Cuando trabajamos con regresión lineal, calcular el vector gradiente es muy sencillo: para la regresión simple, tenemos sólo dos parámetros que afectan directamente al resultado del modelo: $y = w_0 + w_1 x$, con lo cual calcular el gradiente para cada uno de los parámetros es calcular cómo varía el coste aunte un cambio del parámetro w, lo que matemáticamente se resuelve como el cálculo de las derivadas

parciales: derivada parcial de la función de coste con respecto a cada uno de los parámetros: $\frac{\partial C}{\partial w}$ [12].

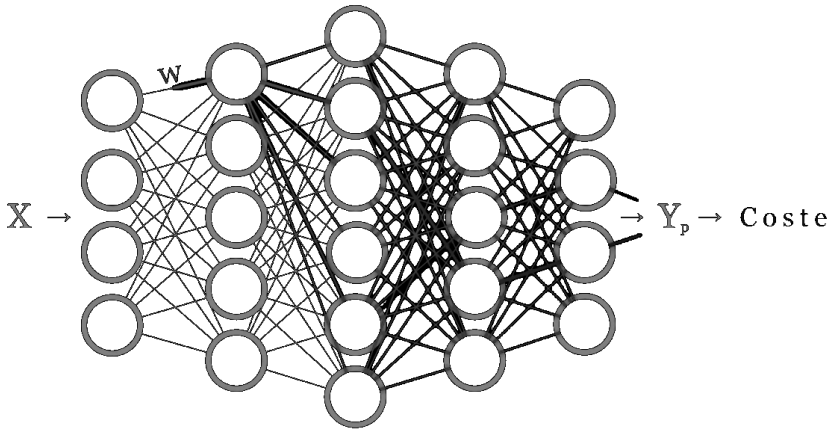


Figura 8: Influencia de pesos en el perceptrón multicapa.

En el perceptrón multicapa, el concepto es el mismo, cómo varía el coste cuando cambiamos un parámetro. Como puede observarse en la figura 8, el problema surge porque el cambio de un parámetro puede influir al resultado final a través de sus conexiones consecuentes. El efecto de ese parámetro también se ve controlado por el valor del resto de parámetros de las capas posteriores. Todo esto hace que el cálculo de las derivadas parciales sea muy complejo de calcular.

Calculada la función de coste, o error, ésta es propagada desde la última capa hacia las primeras capas. Tiene sentido hacerlo porque en un perceptrón multicapa el error de las capas anteriores depende directamente del error de las capas posteriores. Si el resultado de salida no depende mucho de una neurona de la capa anterior, entonces el error de las fases previas a estas, también deberan afectar poco al error final. Si por el contrario, el resultado depende mucho de ésta neurona, habrá que analizar las responsabilidades en las fases previas a ésta.

En función de cuánto se haya implicado cada neurona en generar el resultado final, se ponderan con un porcentaje del error, que será el utilizado para saber cuánto hay que modificar cada parámetro en la neurona. Este proceso se repite nuevamente, asumiendo que la capa anterior es la última, hasta llegar a la primera capa, lo que hace el cálculo

muy eficiente, pues sólo se propaga el error una única vez hacia atrás.

Los errores son usados para calcular las derivadas parciales de cada parámetro de la red, conformando así el vector gradiente, necesario por el algoritmo de descenso del gradiente, que minimiza el error, y por tanto, entrena la red.

Siguiendo el esquema anterior, la derivada de los parámetros de la última capa, suponiendo una red de L capas, corresponde a $\frac{\partial C}{\partial w^L}\frac{\partial C}{\partial b^L}$. Para resolver esta derivada es necesario analizar el camino que conecta el valor del parámetro y el coste final. El parámetro w forma parte del resultado de la suma ponderada $Z^L = w^L x + b^L$, que luego es pasado a la función de activación $a(Z^L)$, para ser finalmente evaluado por la función de coste $C(a(Z^L))$. El resultado final para evaluar corresponde a una composición de funciones, resolviéndose por la conocida regla de la cadena: la derivada de una composición de funciones se calcula multiplicando cada una de las derivadas intermedias:

$$\begin{aligned} \frac{\partial C}{\partial w^L} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L} \\ \frac{\partial C}{\partial b^L} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L} \end{aligned} \tag{2}$$

Suponiendo que la función de coste utilizada es el error cuadrático medio (ecuación 3), la derivada de la activación con respecto al coste viene definida por la ecuación 4.

$$C\left(a_j^L\right) = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2 \tag{3}$$

$$\frac{\partial C}{\partial a_j^L} = \left(a_j^L - y_j\right) \tag{4}$$

La segunda derivada a calcular, la activación con respecto a z, indica cómo varía la salida de la neurona cuando varía la suma ponderada de la neurona. Como el resultado z con respecto a la salida de la neurona viene determinado por la función de activación, esta derivada se calcula derivando la función de activación, que para el caso de una función de activación sigmoide corresponde a la ecuación 5, visualmente mostrado en la figura 9.

$$\frac{\partial a^L}{\partial z^L} = a^L\left(z^L\right) \cdot \left(1 - a^L\left(z^L\right)\right) \tag{5}$$

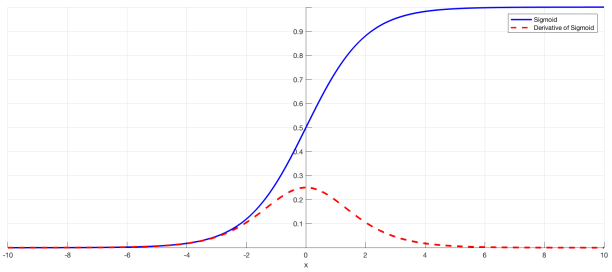


Figura 9: La función de activación sigmoide y su función derivada. Fuente: Towards Data Science, The Vanishing Gradient Problem. Article 69bf08b15484.

La derivada de la suma ponderada con respecto al término de sesgo, el cual es un término independiente, es 1. Con respecto a w, la derivada corresponde al valor de entrada de la neurona, el cuál es la salida de la neurona de la capa anterior: a_i^{L-1}.

La primera y segunda derivada ($\frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$) representan cómo varía el error en función del valor de z, es decir, cuánto se modifica el coste al producirse un cambio en esta neurona, por tanto, su responsabilidad. Por ello, la derivada $\frac{\partial C}{\partial z^L}$ se conoce como error imputado de la neurona, también llamado δ^L. Con las deduciones realizadas anteriormente, la ecuación 2 puede reestructurarse como la ecuación 6.

$$\begin{aligned} \delta^L &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \\ \frac{\partial C}{\partial b^L} &= \delta^L \\ \frac{\partial C}{\partial w^L} &= \delta^L a_i^{L-1} \end{aligned} \tag{6}$$

Una vez determinadas las derivadas necesarias para calcular los parámetros de la última capa (capa L), lo siguiente es proceder con la capa anterior, la capa $L-1$, que, al aplicar la regla de la cadena se obtiene siguiendo la ecuación 7.

$$\begin{aligned} \frac{\partial C}{\partial w^{L-1}} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}} \\ \frac{\partial C}{\partial b^{L-1}} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial b^{L-1}} \end{aligned} \tag{7}$$

Las dos primeras derivadas ($\frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$), correspondientes al error en la capa L, ya se han calculado. Las últimas derivadas ($\frac{\partial z^{L-1}}{\partial w^{L-1}}$ y $\frac{\partial z^{L-1}}{\partial b^{L-1}}$) son operadas de igual forma que para la última capa, lo que corresponde a la activación de la capa previa y 1 respectivamente. Como la derivada de la función de activación también se ha calculado en el paso anterior, sólo queda realizar una única derivada: $\frac{\partial z^L}{\partial a^{L-1}}$, que proporciona la información de cómo varía la suma ponderada de una capa cuando se varía la salida de una neurona en la capa previa.

El cálculo de esta última derivada corresponde la matriz de parámetros w^L que conecta ambas capas, lo que transfiere el error de una capa a la capa anterior. Reestructurando nuevamente para utilizar el error imputado de la neurona en la capa $L-1$, obtenemos la ecuación 8.

$$\delta^{L-1} = w^L \delta^L \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \tag{8}$$

Lo realizado en la capa $L-1$ es extensible al resto de capas restantes. Así, realizando un único pase hacia atrás, recorriendo sucesivamente todas las capas hacia atrás, se calculan todos los errores y las derivadas parciales de la red, con sólo 4 expresiones, necesarias para recorrer de atrás hacia el comienzo.

2.5. Redes convolucionales

Las redes convolucionales son muy similares a una red neuronal construida siguiendo el modelo del perceptrón multicapa, formada por neuronas que tienen pesos y sesgos, las cuales son entrenadas [15]. Cada neurona recibe algunas entradas, realizando un producto escalar y opcionalmente seguida de una no-linearidad. La red entera expresa una única función resultado: desde los datos en brutos de la imagen, los píxeles, hasta la clasificación final, conteniendo también una función de pérdida, por ejemplo SVM/Softmax, y una última capa fully-connected.

Este tipo de redes asume que las entradas son imágenes, lo que permite codificar algunas propiedades en el diseño de la arquitectura, haciendo que la función de

avance (*forward*) sea más eficiente de implementar y reduzca los parámetros de la red. Aprovechando ésto, las capas formadas por esta red, se disponen en tres dimensiones: ancho, alto y profundidad[2]. Las neuronas presentes en una capa, solo estarán conectadas a una pequeña región de la capa anterior, en lugar de un modelo que conecta totalmente con las neuronas anteriores.

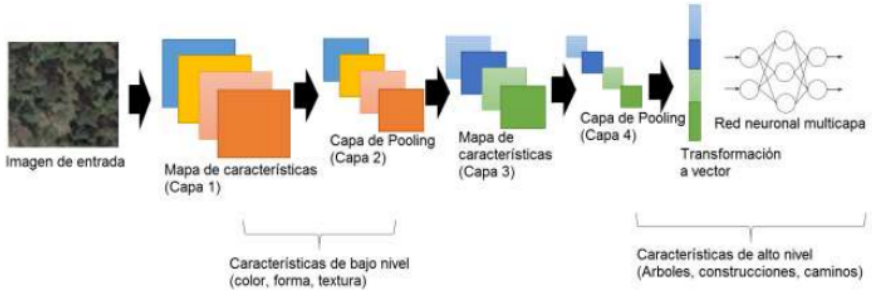


Figura 10: Ejemplo de topología de una red convolucional.

2.6. Tipos de capas en redes convolucionales

Las redes convolucionales están compuestas por una serie de capas que transforman un volumen de activación a otro, aplicando una función diferenciable[3].

Se utilizan tres tipos principales de capas para construir la arquitectura de una red convolucional: la **capa convolutiva**, capa de reducción o **capa de pooling**, y como ya se ha explicado, **capa fully-connected**. La apilación de estas capas forma la arquitectura de una red convolucional.

De esta forma, una red convolucional transforma la entrada original, capa por capa, desde los valores originales, hasta la obtención de una clasificación final. Para ello cada capa oculta sigue una jerarquía y se especializa en conceptos cada vez más abstractos, por ejemplo las primeras capas pueden detectar líneas, curvas hasta especializarse en capas más profundas que reconocen formas más complejas como rostros o siluetas de animales.

Algunas capas contienen parámetros y otras no, en particular, las capas

[2]Se refiere a una dimensión dentro del volumen de activación, no al número de capas dentro de la red.

[3]Enlace: https://es.wikipedia.org/wiki/Función_diferenciable

convolutivas y fully-connected, realizan transformaciones que no sólo corresponden a funciones de activación del volumen de entrada, sino también a parámetros (como los pesos y los sesgos de las neuronas), que son entrenados a través de la técnica de descenso del gradiente, de forma que las clasificaciones sean consistentes con las etiquetas del conjunto de entrenamiento para cada imagen. Por otro lado, las capas de rectificación y de pooling, implementan una función fija carente de pesos que aprender.

2.6.1. Capa convolucional

El objetivo principal de una red convolucional es reducir el volumen de datos de forma que sea más fácil de procesar, sin perder características que son necesarias para obtener una buena predicción.

El primer elemento involucrado en la realización de una operación de convolución es el filtro o *kernel*, representado por una matriz K de dimensión $N \times N$. Este kernel se desplaza de izquierda a derecha y de arriba a abajo con un cierto valor de paso, llamado *stride*, hasta recorrer toda la matriz de entrada, realizando la multiplicación de sus pesos por la porción P de la imagen sobre la que se encuentra el kernel K.

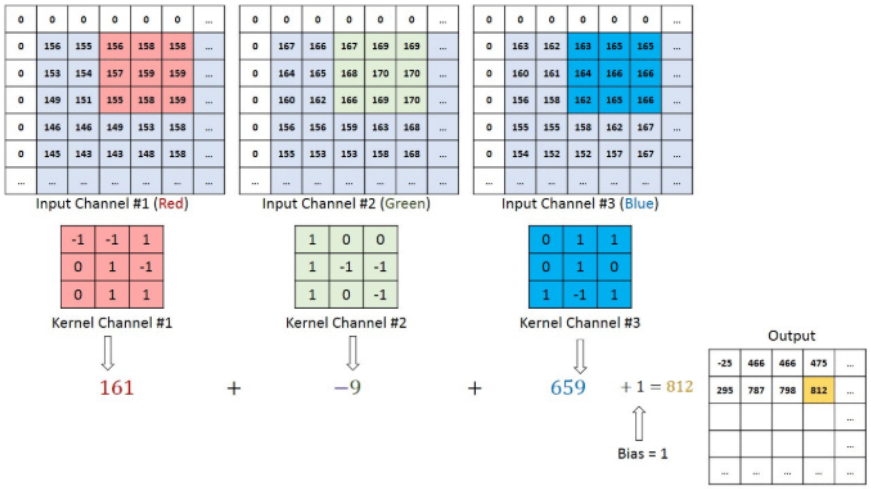


Figura 11: Operación de convolución en un volumen de datos de 3 dimensiones con tres kernels de 3x3. Fuente: Towards Data Science, a comprehensive guide to convolutional neural networks. The eli5 way. Article 3bd2b1164a53.

Como puede observarse en la figura 11, en el caso de imágenes con múltiples

canales (ej. *RGB*), el kernel tiene la misma profundidad que el volumen de entrada, así, la multiplicación de las matrices se realiza en pila ($[K_1, P_1]$, $[K_2, P_2]$, $[K_3, P_3]$) donde todos los resultados son sumados, junto al *bias*, para dar como resultado la salida de la *feature* convolución de un canal [16].

Existe un parámetro adicional llamado relleno o *padding*, que define cómo se maneja el borde de una muestra. Una convolución con *half-padding* mantendrá las dimensiones de salida espacial iguales a la entrada, mientras que las convoluciones sin relleno (*zero-padding*) recortarán algunos de los bordes si el kernel es mayor que 1.

Algunos de los tipos más conocidos de convoluciones son:

- Las **convoluciones dilatadas**, también llamadas convoluciones atroces, introducen otro parámetro en las capas de convolución llamado tasa de dilatación, que define el espacio entre los valores en un kernel. Un kernel de 3x3 con una tasa de dilatación de 2 tendrá el mismo campo de efecto que un kernel 5x5, mientras que sólo usando 9 parámetros (en lugar de 25). Se puede imaginar como tomar un kernel 5x5 y eliminar cada segunda columna y fila. Este tipo de convoluciones ofrecen un campo de vista mayor a cada filtro por el mismo coste computacional, siendo muy populares en el campo de la segmentación en tiempo real.

- Las **convoluciones separables** permiten dividir las operaciones del kernel en múltiples pasos. Expresando una convolución cualquiera como $y = conv(x,k)$ donde y es la salida de la imagen, x es la imagen de entrada, y k es el kernel, asumiendo que el kernel puede ser calculado como $k = k_1.\text{dot}(k_2)$, ésta sería una convolución separable, porque en lugar de realizar una convolución 2D con k, obtenemos el mismo resultado haciendo 2 convoluciones 1D con k_1 y k_2. Este tipo de convoluciones pueden usarse para crear algo muy similar a un espacio separable apilando una capa de kernel $1xN$ y $Nx1$, utilizado recientemente en una arquitectura llamada *EffNet*, mostrando resultados muy prometedores [17].

2.6.2. Capa de pooling

Similar a la capa de convolución, la capa de reducción o pooling es responsable de reducir el tamaño espacial de la convolución, lo que es útil para extraer características dominantes que son invariantes con respecto a su posición o rotación, manteniendo el proceso de entrenamiento del modelo efectivo[18]. Existen dos tipos de reducción:

- **MaxPooling**, que devuelve el valor máximo de la porción de la imagen cubierta por el kernel. También funciona como un supresor de ruido, ya que descarta las activaciones ruidosas por completo y también realiza la eliminación de ruido junto con la reducción de dimensionalidad.

- **Average Pooling**, que devuelve la media de todos los valores de la porción de la imagen cubierta por el kernel. Al contrario que MaxPooling, ésta simplemente realiza la reducción de dimensionalidad como mecanismo de reducción de ruido.

La capa de convolución y la capa de pooling forman juntas la i-ésima capa de una red convolucional. Dependiendo de la complejidad de las imágenes, este número de capas puede aumentar para capturar aún más detalles de bajo nivel, aunque con un mayor coste computacional.

2.6.3. Capa *fully-connected*

Una vez realizado los procesos anteriores, se realiza la vectorización, que convierte la matriz salida de la convolución en un vector unidimensional, el cuál será la entrada de una red neuronal *fully-connected* que actuará de clasificador.

2.6.4. Otros tipos de capa

Existen otros tipos de capas que no forman parte de una red convolucional estándar, como las capas recurrentes, utilizadas en redes convolucionales recurrentes, o bloques residuales (*skip-connections*), los cuales son utilizados en partes de este trabajo.

Las skip-connections son esencialmente conexiones de las primeras capas a capas posteriores mediante la adición o la concatenación directa. La skip-connection simplemente agrega la entrada, x, a la salida de dos capas resultantes, $F(x)$, por lo tanto, la salida general del bloque es $F(x)+x$. La intuición detrás de este tipo de conexiones es que tienen un flujo de gradiente ininterrumpido desde la primera capa hasta la última capa, que aborda el problema de la fuga del gradiente, donde a medida que se agregan más capas que utilizan ciertas funciones de activación a las redes neuronales, los gradientes de la función de pérdida se acercan a cero, lo que hace que la red sea difícil de entrenar. La representación de un bloque residual puede verse en la figura 12.

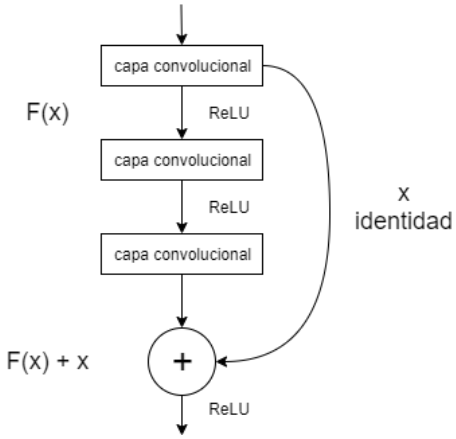


Figura 12: Bloque residual.

3. OBJETIVOS

Como se ha comentado en la sección I, existe un crecimiento en el uso de técnicas que involucran redes neuronales para la clasificación de imágenes hiperespectrales, debido a la gran capacidad que tienen para establecer relaciones y analizar patrones que abarcan el espacio y el espectro. En este contexto, existe un gran esfuerzo para la generación de arquitecturas neuronales que consigan mejores tasas de acierto, un trabajo que conlleva la disponibilidad de un profesional que, a través de un proceso de prueba y error, sea capaz de obtener un modelo óptimo, lo que nos lleva al objetivo general de este proyecto, la generación automática de modelos neuronales para la clasificación de imágenes hiperespectrales.

A través de diferentes técnicas innovadoras, como la búsqueda evolutiva, o métodos basados en el gradiente, se desean obtener modelos que sean óptimos para diferentes conjuntos de datos, comparables con el estado del arte actual. Para la programación que requieren estos métodos se hará uso de librerías y *frameworks* de redes neuronales de código abierto, como *TensorFlow*, *Keras* y *PyTorch* [19, 20].

Para avalar los resultados obtenidos, y obtener las futuras tablas comparativas, se llevará a cabo una experimentación continua en todos los subobjetivos con diferentes conjuntos de datos hiperespectrales. En este sentido, el alcance del objetivo general se guiará a través de una serie de objetivos específicos organizados en fases:

1. Estudiar el uso de algoritmos genéticos y métodos basados en el descenso del gradiente para la búsqueda de arquitectura neuronal. Profundizar en las técnicas actuales de la búsqueda de arquitectura neuronal adaptadas al procesamiento de datos hiperespectrales.

2. Primera fase de la implementación de un algoritmo genético adaptado a la búsqueda de arquitecturas neuronales siguiendo el modelo del perceptrón multicapa, que establece relaciones espectrales. (Generamos nuestro primer modelo automático).

3. Segunda fase de la implementación del algoritmo genético, capaz de realizar búsquedas siguiendo un modelo 2D de red convolucional, que establece relaciones espectro-espaciales. En esta fase se crean técnicas de corrección genética para la adecuada generación de arquitecturas convolucionales.

4. Evaluación con el conjunto de datos propuestos de la primera y segunda fase del algoritmo genético. Generación de resultados.

5. Implementar y evaluar DARTS como método de búsqueda de modelos neuronales para datos hiperespectrales.

6. Realizar implementación de Improved DARTS para datos hiperespectrales, adaptar la arquitectura de la red, la carga del conjunto de datos y las operaciones.

7. Analizar y comparar los resultados obtenidos. Esto ha permitido crear tablas y comparativas objetivas sobre los diferentes métodos.

4. ESTADO DEL ARTE

La clasificación de imágenes hiperespectrales es una tarea central en la comunidad de teledetección, donde los métodos basados en el aprendizaje profundo, especialmente las redes neuronales convolucionales, han mostrado su eficacia en los últimos años al clasificar este tipo de imágenes de manera satisfactoria.

Sin embargo, para obtener un buen rendimiento y una tasa de acierto significativa, se requiere de un esfuerzo importante para diseñar la arquitectura de red profunda, ya que los resultados obtenidos dependerán de este modelo. Por ello, el diseño manual de arquitecturas es costoso, lento, y podría no ajustarse bien para un conjunto de datos específico.

A causa de esto, hay un interés creciente en el diseño de métodos de búsqueda automática de modelos neuronales, el siguiente paso lógico en el aprendizaje de máquina automático, que puede englobarlo como un nuevo subcampo [21]. Hasta el momento, los métodos basados en la búsqueda automática han superado las arquitecturas diseñadas manualmente en tareas como la clasificación de imágenes [22, 23], la detección de objetos[22] o segmentación semántica[24].

Muchos de estos métodos aplican técnicas evolutivas [25] o aprendizaje por refuerzo [8] sobre un espacio de búsqueda discreto y no diferenciable, y es en estos últimos años cuando se proponen nuevas técnicas eficaces que buscan sobre un espacio continuo permitiendo la optimización por la técnica del descenso de gradiente, como el método propuesto en la búsqueda diferenciable de arquitectura, lo que ha permitido obtener modelos comparados con el estado del arte actual en esos conjuntos de datos.

5. METODOLOGÍA

5.1. Búsqueda de arquitectura neuronal

La creación de modelos neuronales es un proceso muy costoso de prueba y error, que consume mucho tiempo y precisa del diseño manual por parte de expertos. La búsqueda de arquitectura neuronal, *Neural Architecture Search*, es el proceso de automatizar la creación de modelos neuronales eficientes, clasificándose de acuerdo a tres dimensiones [26].

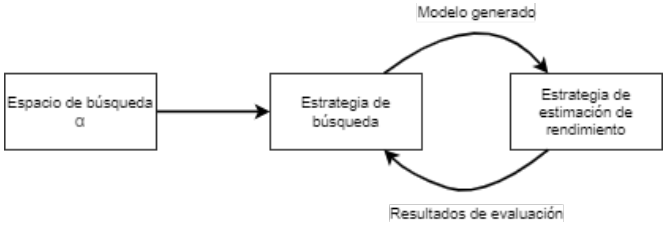


Figura 13: Ilustración de los métodos de la búsqueda neuronal.

- **El espacio de búsqueda**. Define las arquitecturas que pueden ser representadas. Incorporando conocimientos de propiedades típicas en arquitecturas bien conocidas para una tarea específica, puede reducir el tamaño de la búsqueda y simplificarla, aunque esto podría evitar encontrar nuevos modelos que van más allá del conocimiento actual.

- **La estrategia de búsqueda**. Detalla cómo se explora el espacio de búsqueda. Por un lado, se requiere encontrar rápidamente arquitecturas con buen rendimiento, mientras que por otro, debe evitarse la rápida convergencia a una región de arquitecturas cuasi-óptimas.

- **La estrategia de estimación**. El objetivo de la búsqueda de arquitectura neuronal es encontrar modelos con un alto rendimiento predictivo en datos nunca vistos. La opción más simple es realizar un entrenamiento y una validación estándar de la arquitectura en los datos, aunque es computacionalmente costoso y limita el número de arquitecturas que pueden ser exploradas.

Actualmente, existen diferentes estrategias para la búsqueda de arquitecturas neuronales, como la búsqueda aleatoria por fuerza bruta, optimización bayesiana, aprendizaje por refuerzo, el uso de métodos evolutivos y métodos basados en el descenso del gradiente [27, 28]. Algunas de estas estrategias pueden ser computacionalmente muy costosas. En este trabajo, estudiaremos dos técnicas, el uso de algoritmos genéticos, y un nuevo método de búsqueda diferenciable.

5.2. Algoritmos genéticos

Los algoritmos bioinspirados son aquellos basados en el comportamiento de la vida y su evolución, siguiendo la teoría evolutiva de Darwin [29]. Dentro de los algoritmos evolutivos se encuentran los algoritmos genéticos.

La dificultad presentada por este tipo de algoritmos es la existencia de problemas con gran cantidad de mínimos locales, o el hecho de que el óptimo global se encuentre muy aislado. Por lo tanto, deben proveer un mecanismo para evitar quedar atrapados en estos mínimos locales. Los elementos definidos en este tipo de algoritmos son su ploblación, la función objetivo, y los procesos de selección, cruce y mutación.

Figura 14: Procesos del algoritmo genético.

La población del algoritmo genético es el conjunto de individuos, capaces de interactuar entre sí con mecanismos de cruce (imitando la reproducción animal) y mutación, que generará una nueva cepa de individuos con nuevo material genético. Así, cada individuo será un cromosoma formado por genes. La disposición de esos genes marcarán la solución del problema para ese cromosoma.

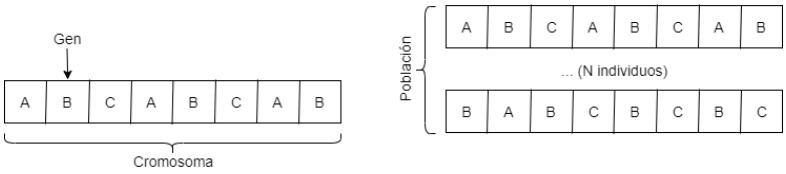


Figura 15: Individuo, un cromosoma, dentro de la población en un algoritmo genético formado por genes, como partes de la solución.

El cromosoma de nuestro algoritmo genético se diferencia según el tipo de arquitectura de red a buscar. Para el modelo *MLP*, el cromosoma quedará definido como una secuencia de información de cada capa, en la que se incluye el número de neuronas a utilizar en esa capa. Consecutivamente, incluirá información del optimizador a utilizar, la función de activación después de cada capa y el número de las capas en uso, desde 1 hasta *N*.

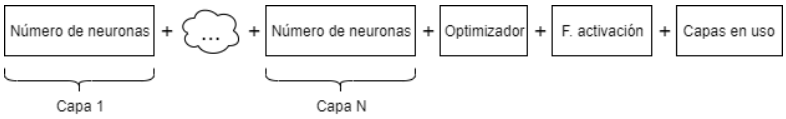


Figura 16: Composición del cromosoma de un individuo en el modelo MLP.

Para el modelo CNN2D, siguiendo la estructura del cromosoma del modelo *MLP*, el cromosoma estará formado por la información de cada capa, el optimizador a utilizar, la función de activación de cada capa y el número de las capas en uso. La información de cada capa se compone, como en el anterior caso, del número de neuronas o filtros a utilizar, y además, del tamaño del *kernel* y de la división *max-pool*.

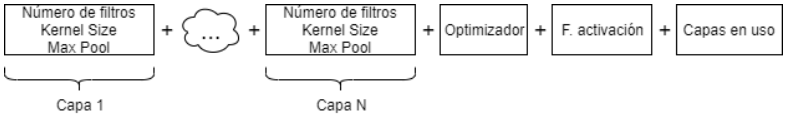


Figura 17: Composición del cromosoma de un individuo en el modelo CNN2D.

Elegir el tamaño de la población sustenta algunos problemas de decisiones: una población pequeña corre el riesgo de no cubrir adecuadamente el espacio de búsqueda de soluciones, mientras que el trabajar con poblaciones de gran tamaño puede acarrear problemas relacionados con el excesivo costo computacional, haciendo el algoritmo impracticable [30].

Los individuos de la primera generación, la población inicial, se generan aleatoriamente, de la forma más uniforme posible. Para ello, al igual que en las redes neuronales, existe una función de coste llamada *fitness*, que determina lo buena que es una solución y es definida necesariamente para que el algoritmo genético tenga éxito. Debe construirse con cierta regularidad, ya que para dos individuos que se encuentren cercanos en el espacio de búsqueda, sus valores de función objetivo deben ser muy similares.

La función de coste definida en este algoritmo genético es el porcentaje de acierto en el conjunto de validación para el modelo formado por un cromosoma. Para evaluar esta función de coste, se realiza un entrenamiento iterativo de la arquitectura, donde por cada paso, comprueba la convergencia de su pérdida. Si la convergencia falla o no mejora después de un número definido de pasos, se finaliza el entrenamiento del modelo, pasando a ser los resultados del último paso del entrenamiento los válidos para ese modelo.

Utilizar técnicas iterativas que permitan parar el entrenamiento del modelo, según la convergencia de su pérdida para reducir el tiempo de cómputo en el algoritmo genético, puede dar lugar a generar modelos no eficaces, pero que evolucionan rápidamente [28]. Si un cromosoma utiliza una función de activación lineal para su arquitectura, llegará rápidamente a altos porcentajes de acierto, aunque no evolucionará más allá de ello, pues es conocido que el uso de funciones de activación lineales no trabajan bien con datos no lineales, como en el caso de conjuntos de datos hiperespectrales. Esto hará que esa arquitectura gane posiciones debido a su rápida convergencia, pudiendo desecharse arquitecturas más prometedoras porque tienen una lenta convergencia. Por todo ello, el utilizar este tipo de técnicas para optimizar

el tiempo de cómputo, puede mermar el rendimiento final al encontrar soluciones cuasi-óptimas.

Para realizar el proceso de generar nuevos individuos es necesaria una función de selección de padres o función de selección proporcional a la función objetivo, cada individuo tiene una probabilidad de ser seleccionado como padre, que es proporcional a su valor de la función objetivo. Cuanto mejor es un individuo, más probabilidades tiene de crear hijos. Existen varios mecanismos para la selección como la ruleta, ruleta proporcional por función objetivo, muestreo estocástico, etc.

El mecanismo de selección utilizado aquí se basa en torneos. Se inicia seleccionando un número P de individuos aleatoriamente. Cada individuo tendrá su propio *fitness*, lo que, simulando el realismo de un torneo entre todos los candidatos, habrá una probabilidad p de que el mejor de los candidatos sea seleccionado, o en su defecto, aleatoriamente uno de los restantes. Este método implica que aquellos individuos con mayor *fitness* sean seleccionados para ser progenitores, aunque también posibilita la entrada de individuos objetivamente peores, lo que mantiene mayor diversificación del material genético y evita la monopolización de individuos evitando caer en mínimos locales.

Una vez creada una lista de individuos aptos para generar nuevos individuos, seleccionamos dos aleatoriamente de la lista de selección y los emparejamos entre sí. Esto es la función de cruce, encargada de efectuar búsquedas a lo largo del espacio de posibles soluciones.

Los operadores de cruce suelen ser basados en dos puntos o punto único de corte, generando una partición del primer individuo siendo la restante del otro, o bien para N puntos donde se realizarán N particiones conformando al individuo de manera uniforme con los genes de ambos [30].

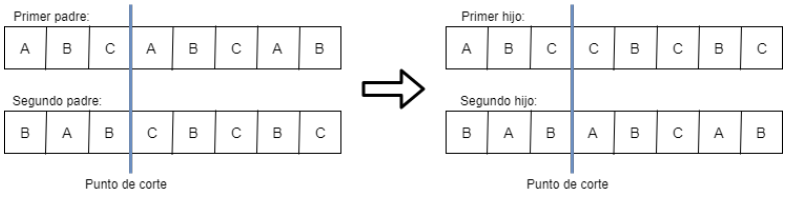


Figura 18: Cruce de dos individuos con punto único de corte. La primera parte del corte corresponderá a los genes de un padre; la segunda parte del corte corresponderá al otro padre.

En este trabajo se ha utilizado un método de cruce por dos puntos, donde se eligen aleatoriamente dos puntos de corte, lo que forma un total de 3 particiones, donde cada partición par corresponde a un individuo y particiones impares al otro individuo.

Generado el nuevo individuo, necesitamos de un operador básico que proporcione un elemento de aleatoriedad en el entorno de los individuos de la población, esto es, una función de mutación. Una vez que el algoritmo va convergiendo, la función de mutación gana importancia para continuar con la convergencia. Este elemento de aleatoriedad además ayudará a evitar individuos que no son válidos en el espacio de búsqueda o repetidos, mutando sus genes hasta cumplir con las condiciones anteriores.

Además de la función de mutación, que ayuda a introducir nuevo material genético para recorrer el espacio de búsqueda, existe otro elemento generador en este trabajo, que es la tasa de retención. Esta tasa definirá el porcentaje de los mejores individuos que permanecerán en la población por cada generación, de tal forma que el resto de individuos serán desechados por otros nuevos, generados aleatoriamente, como al inicio del algoritmo.

En ambos modelos, debido al gen que establece la variabilidad del número de capas en el cromosoma, puede darse lugar a partes del cromosoma que son latentes, en particular las partes de aquellas capas que no son usadas. Este material genético puede ser utilizado a la hora de cruzarse con otros individuos o sufrir una mutación en el gen que establece el número de capas y que sea activada.

Los algoritmos genéticos, como hemos visto, tienen una serie de hiperparámetros para su uso, tales como el número de individuos de la población, la tasa de mutación, la formulación de selección y cruce. Estos definirán la presión genética existente sobre

la población: a mayor presión genética, menor variabilidad en los genes con respecto al mejor individuo de la población, lo que acarreará una mayor convergencia, pero sin un mecanismo bien adaptado que permita la variación genética, es muy probable que quede estancado en un mínimo local desechando caminos que puedan llevar a mejores soluciones.

Algorithm 1 Pseudocódigo del procedimiento de evolución de un algoritmo genético

```
mientras numero_generaciones < generaciones_a_ejecutar obien objetivo_cumplido
    vector_seleccion <- seleccionar_individuos(poblacion)
    nueva_poblacion  <- Vector[longitud_poblacion]

    para i en (0 .. longitud_poblacion)
        padre1 = proceso_seleccion(vector_seleccion)
        padre2 = proceso_seleccion(vector_seleccion)
        hijo1 = padre1.cruzar_con(padre2)
        hijo2 = padre2.cruzar_con(padre1)
        nueva_poblacion.insertar(hijo1, hijo2)
    fin para

    proceso_mutacion(nueva_poblacion)
    ordenar_poblacion_por_fitness(nueva_poblacion)
    recrear_n_peores(tasa_retencion, nueva_poblacion)
    poblacion = nueva_poblacion
fin mientras
```

5.3. Búsqueda diferenciable de arquitecturas

DARTS, de sus siglas en inglés, *Differentiable ARchiTecture Search*, es un método para la búsqueda eficiente de arquitecturas [31]. Éste aborda el desafío de la búsqueda automática de arquitecturas neuronales formulando la tarea de manera diferenciable. A diferencia de los enfoques convencionales de aplicar la evolución o el aprendizaje por refuerzo en un espacio de búsqueda discreto y no diferenciable, DARTS se basa en relajar contínuamente el espacio de búsqueda de la representación de la arquitectura, es decir, convertir parámetros discretos de la arquitectura, tales como las operaciones a realizar, en contínuos, lo que permite una búsqueda eficiente de la arquitectura mediante la técnica del descenso de gradiente.

Especificar la arquitectura de una red es seleccionar muchos parámetros, como la cantidad de filtros, la profundidad de la red, qué operaciones se aplican en cada capa, qué está conectado a qué, etc. Codificando todos esos parámetros en un vector, el cual sólo contendrá variables discretas[4], impide que su optimización puede ser realizada inmediatamente con la técnica del descenso del gradiente. Podría usarse técnicas de aprendizaje por refuerzo o algoritmos evolutivos, pero esto implicaría entrenar un largo número de redes que no evolucionan, lo que conlleva a que el algoritmo sea prohibitibamente lento y computacionalmente muy costoso.

Para resolver este problema de optimización de manera eficiente y encontrar modelos óptimos comparados con el estado del arte, DARTS utiliza una serie de ideas claves:

1. **Sólo busca la celda óptima**. Una celda es un grafo acíclico dirigido que consiste en una secuencia ordenada de N nodos. Cada nodo $x^{(i)}$ es una representación latente (ej. el mapa de característica en redes convolucionales) y cada vértice dirigido (i, j) está asociado con alguna operación $o^{(i,j)}$ que transforma $x^{(i)}$. Apilando celdas idénticas se construye el modelo completo. En algoritmos basados en fuerza bruta u otros más tradicionales existe un espacio de búsqueda extremadamente largo, con conexiones y operaciones arbitrarias entre cada nodo, lo que lleva a un tiempo de cómputo excesivo.

2. **Comparte los parámetros de la red** entre modelos derivados, que pueden ser entrenados simultáneamente (similar a *ENAS, Pham 2018, Google Brain* [32]). Esto es una gran ventaja a la alternativa de probar cada arquitectura, que es reentrenada desde cero, requiriendo muchas horas de computación.

3. **Relejación continua**. Convierte el espacio de búsqueda, compuesto de variables discretas, en continuo. Para ello, parametriza la selección de las operaciones a realizar con la función softmax sobre todas las posibles.

[4]Una variable discreta es aquella que no puede tomar cualquier valor, sólo aquellos pertenecientes a un conjunto de valores.

5.3.1. El espacio de búsqueda

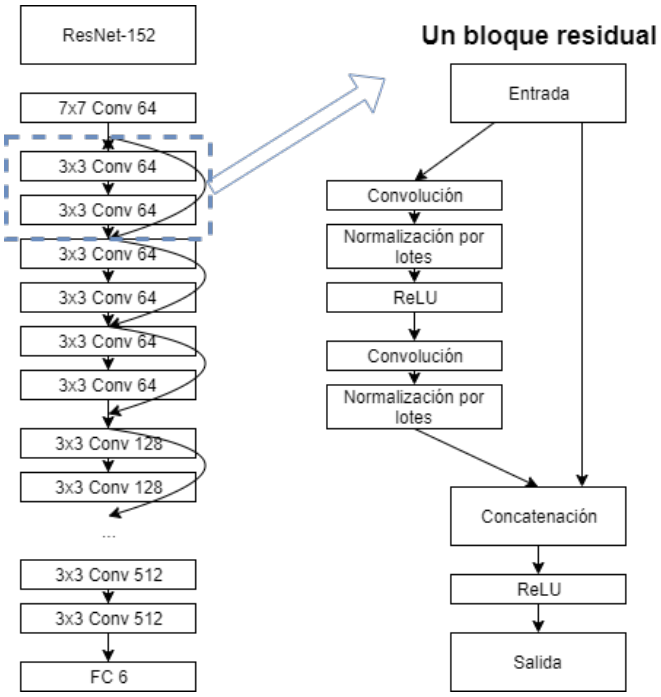


Figura 19: Una red ResNet constituida por bloques residuales.

Definiendo la codificación de la arquitectura en un vector α y sus pesos w, encontrar la mejor arquitectura para un problema dado, puede traducirse a resolver el siguiente problema de optimización anidada:

$$\begin{aligned} &\min_{\alpha} \mathscr{L}_{\text{val}}(\alpha, w^*(\alpha)) \\ &\text{con } w^*(\alpha) = \operatorname*{argmin}_{w} \mathscr{L}_{\text{train}}(\alpha, w) \end{aligned} \tag{9}$$

donde $\mathscr{L}_{\text{val}}$ y $\mathscr{L}_{\text{train}}$ son las funciones de pérdida calculadas, respectivamente en el conjunto de datos de validación y entrenamiento, dependientes de la arquitectura α y los pesos w. Con esto encontramos la arquitectura que, después de ser entrenada en el conjunto de entrenamiento, produce los mejores resultados en el conjunto de validación.

Para evitar buscar en todo el espacio de todas las arquitecturas posibles, DARTS aprovecha una observación importante: redes convolucionales populares a menudo

contienen bloques repetidos apilados secuencialmente. Por ejemplo, ResNet es una sequencia de bloques residuales [33]. Esta red está caracterizada por aumentar el número de capas introduciendo una conexión residual, con una capa identidad, que pasa a la siguiente directamente, mejorando el proceso de aprendizaje.

DARTS sólo busca el bloque o celda óptimo, no el modelo entero. La celda comúnmente consiste en 5-10 operaciones, representada por un grafo acíclico dirigido. Las celdas aprendidas pueden apilarse para formar una red convolucional o conectarse recursivamente para formar una red recurrente, formando el modelo completo.

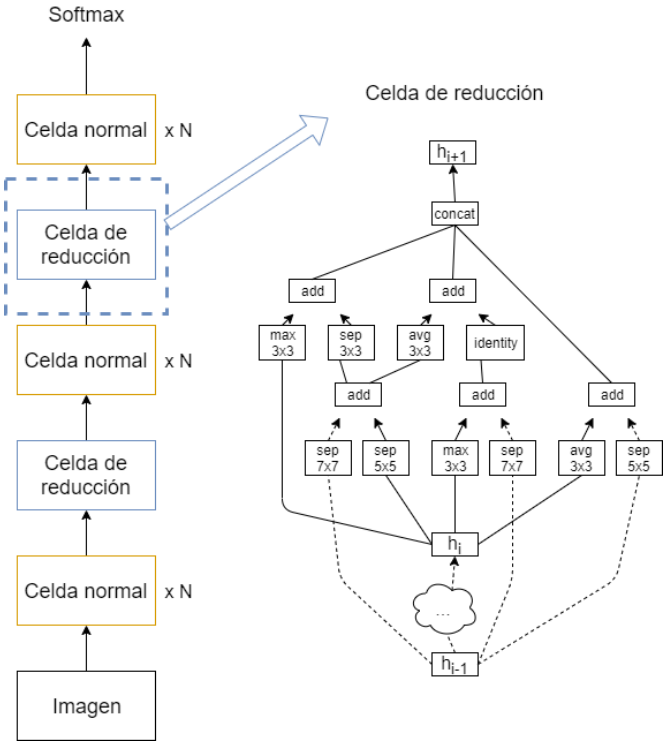


Figura 20: Ejemplo de una celda de reducción apilada para crear un modelo completo.

Existen dos tipos de celdas definidas en DARTS:

- **Celda normal**. Mantiene la dimensión espacial de la salida igual que la entrada.
- **Celda de reducción**. Reduce a la mitad la dimensión espacial de la salida con respecto a la entrada, mientras que duplica el número de filtros/canales. Cuanto más filtros tenga el volumen, más compleja es la información que contiene, lo que permite relacionar conceptos cada vez más abstractos.

Todas las celdas normales comparten la misma arquitectura (operaciones y conexiones), pero tienen pesos independientes para ser entrenadas. Lo mismo ocurre con todas las celdas de reducción.

5.3.2. Relajación continua

Encontrar la celda óptima es equivalente a encontrar la ubicación óptima para las operaciones en los vértices del grafo acíclico dirigido[5]. En lugar de evaluar todo tipo de ubicaciones independientemente, donde cada una necesitaría ser entrenada desde cero, DARTS superpone todas las operaciones candidatas (convolución separable 3x3, convolución separable dilatada 3x3, max-pooling, identidad, etc.) en cada vértice. De esta forma, sus pesos pueden ser entrenados juntos en un único proceso.

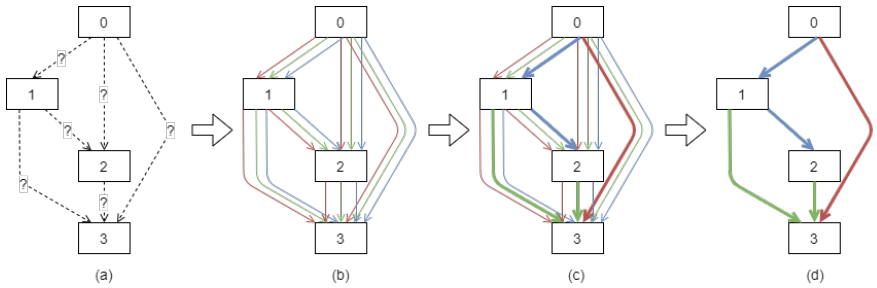


Figura 21: Fase de la relajación continua y discretización en DARTS. (a) Modelo objetivo. (b) Sobreponer operaciones. (c) Buscar operaciones óptimas. (d) Discretizar.

Como podemos observar en la figura 21, se siguen una serie de procesos para llevar a cabo la relajación continua:

- **(a) El objetivo**. Encontrar la celda óptima colocando la operación adecuada en los bordes del gráfico.
- **(b) Superposición**. Cada borde es la suma sobre las salidas de las múltiples operaciones, ponderadas por los parámetros continuos de la arquitectura α.
- **(c) Búsqueda**. Optimizar los pesos de la arquitectura α, usando la técnica del descenso del gradiente en la pérdida de validación.

[5]Un vértice es la conexión que realiza un nodo con otro, el cual se corresponde a una operación.

- **(d) Discretización**. Seleccionar la operación con el mayor peso de la arquitectura, lo que en su conjunto formará la arquitectura final.

Definiendo $\mathcal{O}$ como el conjunto de operaciones candidatas donde cada operación representa alguna función $o(\cdot)$ aplicada a $x^{(i)}$, la operación actual en el vértice (i, j) es la media de todas las operaciones candidatas $o(x)$, parametrizadas por un vector $\alpha^{(i,j)}$. Así, cada vértice no es más que una combinación lineal de operaciones predefinidas, ponderadas por la salida softmax de los parámetros de la arquitectura:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o' \in \mathcal{O}} \exp\left(\alpha_{o'}^{(i,j)}\right)} o(x) \tag{10}$$

Calculando la pérdida real L entrenando w para converger cada vez, es demasiado costoso [34]. Como alternativa, DARTS sólo entrena w utilizando una aproximación de un paso para obtener una representación de la pérdida y evitar el entrenamiento de pesos en cada paso de actualización de la arquitectura.

$$\begin{aligned} &\nabla_\alpha \mathcal{L}_{val}\left(w^*(\alpha), \alpha\right) \\ \approx &\nabla_\alpha \mathcal{L}_{val}\left(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha\right) \end{aligned} \tag{11}$$

Los pesos del modelo óptimo $w*(\alpha)$ son aproximados por el entrenamiento de un paso. El entrenamiento de α y w se realiza de forma alterna, con optimización de dos niveles para entrenar los pesos del modelo estándar y los parámetros de arquitectura.

Algorithm 2 Procedimiento de optimización de dos niveles

while no convergente **do**

1. Actualizar arquitectura α descendiendo $\nabla_\alpha \mathcal{L}_{val}\left(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha\right)$ ($\xi = 0$ si se usa aproximación de primer orden)
2. Actualizar pesos w descendiendo $\nabla_w \mathcal{L}_{train}(w, \alpha)$

end while

Para formar cada nodo en la arquitectura discreta, se retienen las k-mejores operaciones (de nodos distintos) entre todas las operaciones candidatas recopiladas de todos los nodos anteriores.

La suma de las operaciones skip-connections[6] durante la optimización, tienen una ventaja injusta debido a que este tipo de operaciones consiguen una menor tasa de pérdida en épocas tempranas comparadas con las convoluciones, por tanto, serán seleccionadas mayoritariamente sin dejar que el resto de operaciones puedan adquirir todo su potencial [35]. Esta rápida convergencia de las *skip-connections* provoca quedarse estancados en mínimos locales a largo plazo. Es una problemática similar a la presentada en los algoritmos genéticos con las funciones de activación lineal, que debido a su rápida convergencia, son seleccionadas desechando arquitecturas que pueden ser mucho más prometedoras pero requieren más tiempo para converger.

Por otro lado, existe una incongruencia no despreciable que podría sufrir discrepancias entre la codificación de la arquitectura continua y la arquitectura discreta para obtener una representación única.

Debido a estas dos razones, DARTS ofrece una solución que está sesgada y que podría ni siquiera ser subóptima.

5.4. Búsqueda mejorada de arquitectura diferenciable para HSI

Aunque DARTS es un método para la búsqueda de arquitecturas neuronales muy eficiente, presenta dos debilidades fundamentales que quedan sin resolver. Por un lado, la naturaleza de las operaciones skip-connections interfiere con el entrenamiento de la red, que continúa aumentando su prioridad con respecto a otras operaciones, impidiendo que estas útlimas puedan desarrollar todo su potencial, creando una ventaja injusta descrita en el apartado anterior. Estas operaciones *skip-connections*, dominantes en la red, no pueden ser discretizadas directamente, de lo contrario, causarían una caída dramática del rendimiento. Así mismo, también existe una discrepancia al convertir el vector de la arquitectura continua en una representación única discreta.

Basado en el método de Fair DARTS [36], para resolver el primer problema y hacer una búsqueda justa, se propone un enfoque de competencia colaborativa, al

[6]Las skip-connections son conexiones extra entre nodos en diferentes capas de una red neuronal que salta una o más capa del procesamiento no lineal.

ofrecer a cada operación un peso de arquitectura independiente, consiguiendo que el dominio de las operaciones *skip-connections* no prevalezca. Con respecto al segundo problema, es decir la discrepancia al discretizar la arquitectura, se propone la creación de una nueva pérdida auxiliar, llamada pérdida *zero-one*, para dirigir los pesos de la arquitectura hacia sus extremos, es decir, completamente habilitados o deshabilitados. Esta discrepancia, por tanto, disminuye a su mínimo.

5.4.1. Evitar la trampa de las skip-connections por colaboración

Como se ha introducido anteriormente, en este método se propone un mecanismo colaborativo para eliminar la ventaja injusta existente, y evitar la selección abusiva de estas operaciones, que no sólo explota las *skip-connections* para obtener un flujo de información más fluido, sino que también proporciona igualdad de oportunidades para otras operaciones que a la larga consigan ganar precisión en la clasificación realizada por la red, como las convoluciones. De esta forma, aplica una función de activación sigmoide para cada $\alpha_{i,j}$, de modo que cada operación puede activarse o desactivarse independientemente sin ser suprimida.

Modificando la fórmula de selección de operación de DARTS, donde se define $\mathscr{O}$ como el conjunto de operaciones candidatas, donde cada operación representa una función $o(\cdot)$ aplicada a $x^{(i)}$, la operación actual en el vértice (i, j), queda definida como:

$$\bar{o}_{i,j}(x) = \sum_{o \in \mathscr{O}} \sigma\left(\alpha_{o_{i,j}}\right) o(x) \tag{12}$$

Incluso si la selección de operaciones *skip-connections* $\sigma\left(\alpha_{skip}\right)$ se satura en 1, otras operaciones prometedoras aún se podrán optimizar, ya que continúan aumentando sus pesos para reducir la pérdida en validación $\mathscr{L}_{val}$.

Con $\sigma(\alpha)$ definido, el problema de la discretización aún existe. Todavía es necesario conducir los pesos a sus extremos, 0 y 1, para aliviar la discrepancia discutida anteriormente.

5.4.2. Minimizar la discrepancia de la discretización

Para llevar el valor resultado de la función de activación sigmoide sobre los pesos de la arquitectura a 0 o 1, se fuerza la creación de una pérdida extra *zero-one*. Supongamos que L_{0-1} denota esta componente de pérdida, donde $z = \sigma(\alpha)$. Para conseguir el objetivo de minimizar esta discrepancia, el diseño de la pérdida debe cumplir con tres criterios básicos:

- Necesita tener su máximo global en $z = 0{,}5$, como punto de partida, y dos mínimos globales en 0 y 1.
- La magnitud del gradiente $\frac{df}{dz}\Big|_{z=0,5}$ tiene que ser adecuadamente pequeña para permitir a los pesos de la arquitectura fluctuar, pero suficientemente grande para permitir atraer z hacia alguno de sus extremos.
- Debe ser diferenciable para el uso de la propagación hacia atrás.

Según el primer requisito, para minimizar la discrepancia de discretización, se aleja $\sigma(\alpha)$ de 0,5 hacia 0 o 1, estados que corresponderían a deshabilitada o habilitada respectivamente, dependiendo de si la operación gana o pierde precisión. El segundo establece restricciones explícitas necesarias, de lo contrario, el enfoque basado en el gradiente, puede alcanzar el mínimo local demasiado pronto.

La función de pérdida propuesta en este método, la cual cumple los requisitos anteriores, queda definida como:

$$L_{0-1} = -\frac{1}{N}\sum_{i}^{N}\left(\sigma\left(\alpha_i\right) - 0{,}5\right)^2 \tag{13}$$

Para controlar su magnitud, se pondera esta pérdida por un coeficiente w_{0-1}, por lo tanto, la pérdida total para α es formulada como:

$$L_{\text{total}} = \mathscr{L}_{\text{val}}\left(w^*(\alpha), \alpha\right) + w_{0-1}L_{0-1} \tag{14}$$

Del mismo modo que DARTS, los pesos de la arquitectura pueden ser optimizados a través de la propagación hacia atrás. El objetivo de la búsqueda, desde la ecuación

anterior, es encontrar una arquitectura con alta precisión y con una buena aproximación en la transformación de continua a discreta.

Combinando las ecuaciones anteriores, el modelo propuesto puede ser formalmente definido como:

$$\begin{aligned} &\min_{\alpha} \mathscr{L}_{val}\left(w^*(\alpha), \alpha\right) + w_{0-1} L_{0-1} \\ &\text{s.t. } w^*(\alpha) = \text{argmin}_w \mathscr{L}_{train}(w, \alpha) \\ &\bar{o}_{i,j}(x) = \sum_{o \in \mathscr{O}} \sigma\left(\alpha_{o_{i,j}}\right) o(x) \end{aligned} \tag{15}$$

6. IMPLEMENTACIÓN Y DESARROLLO

Siguiendo la ruta marcada en la sección de objetivos para este trabajo, la primera implementación se trata de un algoritmo genético, el cual se ha realizado con ayuda de librerías propias de *Machine Learning*: *SciKit-Learn* y *Keras* como frontend de *TensorFlow*, y *NumPy*, como librería para el uso de vectores y matrices. El uso de éstas queda argumentado por su objetivo de facilitar un proceso de experimentación rápido. El lenguaje de programación en el cual se basarán todos los experimentos es por tanto *Python*, debido a su gran biblioteca pública y comunitaria, así como su facilidad de prototipado por su naturaleza, al tratarse de un lenguaje interpretado.

Para conseguir la réplica de experimentos en las mismas condiciones se ha considerado utilizar una semilla de aleatoriedad constante. Por ello, se ha definido la misma semilla para los entornos de *Python*, *NumPy* y *TensorFlow*.

Algorithm 3 Establecer semilla de aleatoriedad constante en Python 3.8

```
1 os.environ['PYTHONHASHSEED'] = str(CONFIG_SEED)
2 random.seed(CONFIG_SEED)
3 np.random.seed(CONFIG_SEED)
4 tf.set_random_seed(CONFIG_SEED)
```

Una vez establecida la semilla constante de aleatoriedad, ver algoritmo 3, se procede a cargar el conjunto de datos elegido, lo que instanciará dos matrices, una con las imágenes y otra con sus correspondientes etiquetas. Utilizando la función `train_test_split` del paquete *SciKit-Learn*, se divide el conjunto de datos en dos, un conjunto de entrenamiento y otro de validación.

El proceso de establecer una semilla de aleatoriedad y la carga de datos son compartidos en todos los experimentos, para asegurar una mayor fiabilidad en la obtención y comparación de los resultados.

6.1. Algoritmo genético

El primer paso es formalizar al individuo de la población, lo cual se ha realizado creando una clase abstracta[7] llamada `GAIndividualInterface`, que define un problema genético, cuyo objetivo puede ser maximizar o minimizar el resultado.

Heredada de ésta, se definen dos individuos, uno para cada modelo posible: el perceptrón multicapa y la red convolucional 2D, donde cada uno de ellos contiene cromosomas con una estructura parecida, como se define en la metodología. Estos individuos tienen un parámetro especial `hash` que corresponde al cifrado MD5[8] del vector que contiene el cromosoma, lo que permite comprobar si un individuo es único en la población. Como es posible encontrar dos individuos con diferentes cromosomas pero que generen el mismo modelo (ver metodología), el hash del cromosoma solo se realiza incluyendo el código genético activo, algoritmo 4.

Algorithm 4 Actualizar hash del cromosoma en un individuo

```
def update_hash(self):
        total_layers = len(self.layers)
        current_layers = self.get_total_layers()
        hashed = str(self.chromosome[0:current_layers:] +\
            self.chromosome[total_layers + 1::]).encode('utf-8')
        self.hash = hashlib.md5(hashed).hexdigest()
```

Al instanciar alguna de estas clases, se generará automáticamente un cromosoma aleatorio, que contendrá el número de neuronas/filtros por cada capa, el optimizador a utilizar, la función de activación (compartida entre todas las capas) y el número de capas activas, como detalla el algoritmo 5.

La evaluación del modelo representado por este cromosoma requerirá de dos funciones de ayuda, una para compilar modelos MLP y otra para compilar modelos CNN2D.

El modelo MLP del algoritmo 6 se compone de una secuencia de capas especificadas en el cromosoma, con su configuración (número de neuronas, función

[7] Es una clase que no se puede instanciar y se usa únicamente para definir subclases.
[8] Es un algoritmo de reducción criptográfico de 128 bits ampliamente usado

Algorithm 5 Generar genes aleatorios para formar un cromosoma completo

```
def _random_chromosome(self):
    chromosome = []
    for gene in range(len(self.layers)):
        chromosome.append(random.choice(self.units_per_layer))
    chromosome.append(random.choice(self.optimizers))
    chromosome.append(random.choice(self.activations))
    chromosome.append(random.choice(self.layers))
    return chromosome
```

de activación) y una capa adjunta de Dropout, además de una capa final de salida inmodificable, con función de activación Softmax, que actuará de clasificadora.

Algorithm 6 Modelo MLP del algoritmo genético

```
model = Sequential()
for i in range(layers):
    units = individual.get_units_on_layer(i)
    if i == 0:
        model.add(Dense(units, activation=activation,
            input_shape=input_shape,
            kernel_initializer=initializer))
    else:
        model.add(Dense(units, activation=activation,
            kernel_initializer=initializer))
    model.add(Dropout(CONFIG_DROPOUT))
    model.add(Dense(classes, activation='softmax',
        kernel_initializer=initializer))
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
```

Como podemos comprobar en el código de arriba, todas las capas del modelo tienen un inicializador personalizado, que corresponde al tipo `TruncatedNormal` de *Keras*, para establecer la semilla constante de aleatorización:

```
initializer = TruncatedNormal(mean=0.0, stddev=0.05, seed=CONFIG_SEED)
```

Para compilar el modelo CNN2D se siguen los mismos pasos, adaptándose para generar un cromosoma aleatorio con la estructura del modelo CNN2D, que con respecto al modelo MLP, sustituye las capas densas con convoluciones, estableciendo

también su tamaño de kernel, y una capa extra opcional antes de la capa Dropout, la capa de reducción `MaxPooling2D`. Como esta capa es opcional, se considera que si el gen que establece la capa es `(1, 1)` no hay reducción en ese nivel. El procesamiento de la salida también difiere, pues se necesita realizar un `Flatten` para conectar la salida de la convolución a una capa *full-connect*, que seguida de un Dropout estático definido, transfiere la entrada a la capa final de clasificación con activación Softmax. El código resultado de estas modificaciones es el mostrado en el algoritmo 7.

Algorithm 7 Modelo CNN2D del algoritmo genético

```
model = Sequential()
for i in range(0, layers):
    units = individual.get_units_on_layer(i)
    kernel_size = individual.get_kernel_size_on_layer(i)
    max_pool = individual.get_max_pool_on_layer(i)
    if i == 0:
        model.add(Conv2D(units,
            kernel_size=kernel_size,
            activation=activation,
            input_shape=input_shape,
            kernel_initializer=initializer))
    else:
        model.add(Conv2D(units,
            kernel_size=kernel_size,
            activation=activation,
            kernel_initializer=initializer))
    if max_pool != (1, 1):
        model.add(MaxPooling2D(pool_size=max_pool))
    model.add(Dropout(Config.neural['dropout']))
    model.add(Flatten())
model.add(Dense(individual.get_units_on_layer(layers - 1),
    activation=activation,
    kernel_initializer=initializer))
model.add(Dropout(Config.neural['final_dropout']))
model.add(Dense(classes, activation='softmax',
    kernel_initializer=initializer))
model.compile(loss='categorical_crossentropy',
    optimizer=optimizer,
    metrics=['accuracy'])
```

Una vez creadas las funciones que compilan los dos tipos de modelo, el siguiente paso es crear el método que evaluará cuánto de bueno es nuestro individuo, es decir, crear la función objetivo. El proceso es muy sencillo, sólo debe compilarse el modelo,

entrenarlo con el conjunto de datos y evaluarlo para obtener los resultados, ver algoritmo 8.

Algorithm 8 Función de evaluación del algoritmo genético

```
try:
    model = compile_model_mlp(self, Context.total_classes, Context.input_shape)
    callbacks = []
    if Config.search['early_stop']:
        callbacks.append(early_stopper)
    model.fit(Context.x_train, Context.y_train,
        batch_size=Config.neural['batch_size'],
        epochs=Config.neural['epochs'],
        verbose=Config.verbose,
        validation_data=(Context.x_test, Context.y_test),
        callbacks=callbacks)
    loss, acc = model.evaluate(Context.x_test, Context.y_test)
    self.score = acc
except Exception as ex:
    self.score = -1
finally:
    K.clear_session()
```

Definidos la creación del cromosoma y su evaluación, solo queda una cosa más, el proceso de mutación que sigue el individuo, algoritmo 9. Este proceso es el mismo en ambos tipos, tanto en el MLP como en el CNN2D. La función de mutación del individuo recibe un parámetro, que es el gen a mutar, y dependiendo del gen el valor de mutación será uno entre varios candidatos, ej. si se muta el gen de la función de activación, la mutación será sobre una de las posibilidades de `['relu', 'tanh', ...]`.

En el modelo CNN2D, las convoluciones, junto con capas de reducción *MaxPool*, puede dar lugar a arquitecturas erróneas, que los tamaños de entrada/salida no coincidan, por lo que se requiere una función de ayuda que establezca si el cromosoma generará un modelo válido. Sea el volumen de entrada en una capa de convolución definido por $(Ancho, Alto, Profundidad)$, y los parámetros a usar como, el número de filtros K, el tamaño de los filtros (espacial) F, el paso de la convolución S y el relleno P, el volumen de salida $(Ancho_i, Alto_i, Profundidad_i)$ viene definido por:

Algorithm 9 Proceso de mutación del algoritmo genético

```
offset = len(self.layers*3)
units_index = range(0, offset, 3)
kernel_index = range(1, offset, 3)
max_pooling_index = range(2, offset, 3)
optimizer_index = [offset]
activation_index = [offset + 1]
nl_index = [offset + 2]
if gene in units_index:
    self.chromosome[gene] = random.choice(self.units_per_layer)
elif gene in kernel_index:
    self.chromosome[gene] = random.choice(self.kernel_sizes)
elif gene in max_pooling_index:
    self.chromosome[gene] = random.choice(self.max_pooling)
elif gene in optimizer_index:
    self.chromosome[gene] = random.choice(self.optimizers)
elif gene in activation_index:
    self.chromosome[gene] = random.choice(self.activations)
elif gene in nl_index:
    self.chromosome[gene] = random.choice(self.layers)
else:
    raise Exception('bad gene index (not-found)')
```

$$Ancho_i = (Ancho - F + 2 * P)/S + 1$$

$$Alto_i = (Alto - F + 2 * P)/S + 1$$

$$D = K$$

Si alguno de los valores $Ancho_i$, $Alto_i$ o D es menor a 1, podemos decir que el modelo generado no es válido, ya que no tendría sentido tener una dimensión negativa, lo que aplicado a la fórmula anterior en un bucle iterativo que compruebe esta condición en cada capa, permitirá prevenir el fallo del modelo y corregir al individuo, mutando hasta generar un modelo válido, algoritmo 10.

El entorno del algoritmo genético se ha implementado como una clase genérica, el cuál se encargará de los procesos más abstractos del algoritmo genético, como el cruce, la ejecución, y gestionar los elementos de cada individuo como la mutación, la validación y la corrección de su cromosoma. Cada iteracción del algoritmo genético

Algorithm 10 Comprobar salida válida del modelo formado por un individuo

```
def check_valids_output(individual, input_shape):
    layers = individual.get_total_layers()
    w = input_shape[0]
    h = input_shape[1]
    d = input_shape[2]
    for i in range(layers):
        units = individual.get_units_on_layer(i)
        kernel_size = individual.get_kernel_size_on_layer(i)
        max_pool = individual.get_max_pool_on_layer(i)
        w = w - kernel_size[0] + 1
        h = h - kernel_size[1] + 1
        d = units
        if max_pool != (1, 1):
            w = math.floor(w / max_pool[0])
            h = math.floor(h / max_pool[1])
        if w < 1 or h < 1 or d < 1: return False
    return True
```

corresponde a una nueva generación, en la que se llevan a cabo una serie de procesos: el cruce de individuos, formado por los procesos de selección, emparejamiento y mutación, y generación de nuevos individuos, lo que viene determinado por una tasa de retención definida. El código resultado se lista en algoritmo 11.

El método de selección se basa en la creación de un torneo (ver metodología), cuyo código está listado en algoritmo 12.

El algoritmo genético implementa una interfaz de configuración a través de un archivo *YAML*[9], implementada a través de una clase siguiendo el patrón *Singleton*[10]. Los parámetros posibles de configuración definidos son los mostrados en las siguientes tablas:

[9]Formato de serialización de datos que se caracteriza por ser fácilmente legible.
[10]Recibe su nombre debido a que sólo se puede tener una única instancia para toda la aplicación.

Algorithm 11 Función de cruce y evolución del algoritmo genético

```
def _crossover(self):
    next_population = []
    if self.elitism:
        next_population.append(self.best.copy())
    while len(next_population) < self.size:
        parent1 = self._select()
        if random.random() < self.crossover_rate:
            parent2 = self._select()
            while parent2.hash == parent1.hash:
                parent2 = self._select()
            offspring = parent1.crossover(parent2)
        else: offspring = [parent1.copy()]
        for individual in offspring:
            self._mutate(individual)
            individual.evaluate(self.optimum)
            next_population.append(individual)
    self.population = next_population[:self.size]
def step(self):
    self._crossover()
    self.population.sort()
    to_keep = int(self.retain_rate*self.size)
    to_add = self.size - to_keep
    self.population = self.population[:to_keep]
    for i in range(to_add):
        new = self.kind()
        while not self._is_unique(new) or not self._is_valid(new):
            new = self.kind()
        new.evaluate(self.optimum)
        self.population.append(new)
    self.generation += 1
    self.population.sort()
```

Algorithm 12 Proceso de selección por torneo

```
competitors = [random.choice(self.population) for i in range(size)]
competitors.sort()

if random.random() < choose_best:
    return competitors[0]

return random.choice(competitors[1:])
```

Clave	Descripción	Valores
dataset	Conjunto de datos a usar. Por defecto: indian	indian, pavia, salinas, ksc
type_algorithm	Tipo de algoritmo de búsqueda espacial o espectral. Por defecto: spatial	spatial, spectral
verbose	Establecer salida de la aplicación por depuración. Por defecto: true	true, false
seed	Semilla utilizada para la generación de aleatoriedad. Por defecto: 69	números enteros
experiment_id	ID del experimento a ejecutar. Por defecto: 0	números enteros

Tabla 3: Configuración básica de ejecución

Clave	Descripción	Valores
crossover_rate	Tasa de cruce, indica el porcentaje de que dos individuos al cruzarse generen uno nuevo. Por defecto: 0.9	0 .. 1
elitism	Habilar o deshabilitar elitismo. Por defecto: true	true, false
generations	Número total de generaciones a computar. Por defecto: 8	números enteros
mutation_rate	Tasa de mutación por gen. Por defecto: 0.2	0 .. 1
retain_rate	Tasa de retención, indica el porcentaje de individuos que permanecerán en la población. Por defecto: 0.6	0 .. 1
size	Tamaño de la población. Constante en el tiempo. Por defecto: 30	números enteros

Tabla 4: Configuración del algoritmo genético

Clave	Descripción	Valores
batch_size	Tamaño de lote del conjunto de datos. Por defecto: 100	números enteros
dropout	Tasa de dropout en cada capada. Por defecto: 0.2	0 .. 1
epochs	Épocas de entrenamiento. Por defecto: 100	números enteros
final_dropout	Tasa de dropout antes de la última capa clasificadora. Por defecto: 0.5	0 .. 1
training_rate	Porcentaje del conjunto de entrenamiento. Por defecto: 0.1	0 .. 1

Tabla 5: Configuración de red neuronal.

Clave	Descripción	Valores
activations	Funciones de activación. Por defecto: ['relu']	Array
early_stop	Activar parada temprana como método de ejecución iterativa. Por defecto: true	true, false
kernel_size	Tamaño del Kernel. Sólo para CNN2D. Por defecto: [(3,3), (4,4), (5,5), (7,7)]	Array
layers	Cantidad de capas a usar. Por defecto: [1, 2, 3, 4, 5]	Array
max_pooling	Tamaño de división MaxPool2D. Sólo CNN2D. Por defecto: [(1,1), (2,2), (3,3), (4,4)]	Array
optimizers	Optimizadores. Por defecto: ['adam', 'rmsprop']	Array
units_per_layer	Neuronas/Filtros por capa. Por defecto: [16, 32, 64, 128, 256, 512]	Array

Tabla 6: Configuración de búsqueda.

6.2. DARTS

La implementación de DARTS es mucho más sencilla que su predecesor, el algoritmo genético, pues sigue el funcionamiento básico de una red neuronal para el entrenamiento de los pesos (la búsqueda de la arquitectura), con dos procesos claramente divididos en entrenamiento e inferencia. Programado con el *framework* PyTorch, debido a su facilidad de uso para redes convolucionales, con una extensa documentación acerca de la librería, y uso en GPU. El código del bucle principal para la optimización del modelo se encuentra en el algoritmo 13.

Algorithm 13 Bucle principal de búsqueda de modelo en DARTS

```
for epoch in range(Config.epochs):
    scheduler.step()
    lr = scheduler.get_lr()[0]
    genotype = model.genotype()
    model.update_history()
    train_acc, train_obj = train(train_queue, valid_queue,
        model, architect, criterion, model_optimizer, lr, epoch)
    valid_acc, valid_obj = infer(valid_queue, model, criterion)
```

El proceso de entrenamiento, en el algoritmo 14, actualiza los parámetros de la red, siguiendo la optimización de dos niveles explicada en la metodología, donde se

actualiza la arquitectura α reducciendo la pérdida en el conjunto de validación.

Algorithm 14 Proceso de entrenamiento en DARTS

```
for step, (input, target) in enumerate(train_queue):
    model.train()
    n = input.size(0)
    input = Variable(input, requires_grad=False).cuda(non_blocking=True)
    target = Variable(target, requires_grad=False).cuda(non_blocking=True)
    input_search, target_search = next(iter(valid_queue))
    input_search = Variable(input_search, requires_grad=False)
        .cuda(non_blocking=True)
    target_search = Variable(target_search, requires_grad=False)
        .cuda(non_blocking=True)
    architect.step(input, target, input_search, target_search, lr,
        model_optimizer, unrolled=args.unrolled, epoch=epoch)
    model_optimizer.zero_grad()
    logits = model(input, epoch)
    loss = criterion(logits, target)
    loss.backward()
    nn.utils.clip_grad_norm(model.parameters(), args.grad_clip)
    model_optimizer.step()
```

La optimización de la arquitectura viene definida por el algoritmo 2 en la metodología e implementada en el algoritmo 15, que define la actualización de un paso de la arquitectura.

Algorithm 15 Actualización de arquitectura por el descenso del gradiente en DARTS

```
def _backward_step(...):
    # Actualizar arquitectura alfa descendiendo la pérdida en el conjunto
    # de validación.
    loss = self.model._loss(input_valid, target_valid, epoch)
    # Actualizar pesos descendiendo la pérdida en conjunto de entrenamiento.
    loss.backward()
def step(...):
    self.optimizer.zero_grad()
    self._backward_step(input_valid, target_valid, epoch)
    self.optimizer.step()
```

El proceso de inferencia, implementado en algoritmo 16, se encarga de probar el modelo sobre el conjunto de validación, siguiendo el mismo bucle de entrenamiento sin la actualización de parámetros ni de la arquitectura.

Algorithm 16 Proceso de inferencia en DARTS

```
for step, (input, target) in enumerate(valid_queue):
      input = Variable(input, volatile=True).cuda(non_blocking=True)
      target = Variable(target, volatile=True).cuda(non_blocking=True)
      logits = model(input)
      loss = criterion(logits, target)
      prec1 = utils.accuracy(logits, target, topk=(1,))
```

El modelo de la red está compuesto de un bloque encargado de redimensionar la entrada a la red convolucional, seguido de una secuencia de celdas, tantas como según sea la profundidad de la red. Cada 3 capas se establece que la celda será de reducción (ver algoritmo 17), cuyo objetivo es reducir la muestra de la representación de entrada, reduciendo la dimensión espacial en 2 y ampliando el número de canales, para que haya poco kernel espacial y mucho en los canales, permitiendo realizar comparaciones cada vez más abstractas, ya que cuanto más filtros tenga, más complejo es lo que obtiene. Similar al algoritmo genético, la salida se conecta a una capa densa de activación lineal con tantas neuronas como número de clases tenga el conjunto de datos utilizado.

Las celdas normales comparten todas el mismo modelo, al igual que todas las celdas de reducción, y al tratarse de un modelo genérico, no se necesita establecer dos clases diferentes para cada modelo, lo que complicaría su posterior procesamiento. Por ello es suficiente con establecer el stride de reducción a 2 o 1, dependiendo de si el tipo de celda es normal o de reducción, implementado en algoritmo 18.

La selección de la operación para cada vértice (i, j), se realiza siguiendo la ecuación de selección vista en la metodología, cuyo resultado de avance está implementado en algoritmo 19.

Las operaciones primitivas definidas como el genotipo de la celda, son elegidas manualmente de una lista de posibilidades, al igual que con el algoritmo genético. Para este trabajo se han escogido las operaciones de convolución, *max-pool*, *average-pool* y *skip-connect*, definidas en algoritmo 20.

Algorithm 17 Modelo neuronal que define la arquitectura base en DARTS

```
self.pre_process = nn.Sequential(
    nn.Conv2d(NUM_DEPTH, 32, (1, 1), padding=(0, 0),
    bias=False),
    nn.BatchNorm2d(32))
C_curr = stem_multiplier*C
self.stem = nn.Sequential(
    nn.Conv2d(1, C_curr, (3, 1), padding=(1, 0),
    bias=False),
    nn.BatchNorm2d(C_curr))
C_prev_prev, C_prev, C_curr = C_curr, C_curr, C
self.cells = nn.ModuleList()
reduction_prev = False
for i in range(layers):
    if i in [layers//3]:
        C_curr *= 2
        reduction = True
    else: reduction = False
    cell = Cell(steps, multiplier,
        C_prev_prev, C_prev,
        C_curr,
        reduction, reduction_prev)
    reduction_prev = reduction
    self.cells += [cell]
    C_prev_prev, C_prev = C_prev, multiplier*C_curr
self.global_pooling = nn.AdaptiveAvgPool2d(1)
self.classifier = nn.Linear(C_prev, num_classes)
```

Algorithm 18 Modelo de selección de operación en celdas

```
self._ops = nn.ModuleList()
self._bns = nn.ModuleList()
for i in range(self._steps):
    for j in range(2+i):
        stride = 2 if reduction and j < 2 else 1
        op = MixedOp(C, stride)
        self._ops.append(op)
```

Algorithm 19 Selección de operación en DARTS

```
class MixedOp(nn.Module):
...
def forward(self, x, weights):
    return sum(w * op(x) for w, op in zip(weights, self._ops))
```

Algorithm 20 Operaciones primitivas en DARTS para datos hiperespectrales

```
OPS = {
  'avg_pool_3x1': lambda C, stride, affine: nn.AvgPool2d((3, 1),
    stride=(stride, 1),
    padding=(1, 0),
    count_include_pad=False),
  'max_pool_3x1': lambda C, stride, affine: nn.MaxPool2d((3, n),
    stride=(stride, 1),
    padding=(1, 0)),
  'skip_connect': lambda C, stride, affine:
    Identity() if stride == 1 else FactorizedReduce(C, C, affine=affine),
  'conv_3x1': lambda C, stride, affine:
    SepConv(C, C, 3, stride, 1, affine=affine),
  'conv_5x1': lambda C, stride, affine:
    SepConv(C, C, 5, stride, 2, affine=affine),
  'conv_7x1': lambda C, stride, affine:
    SepConv(C, C, 7, stride, 3, affine=affine),
  'conv_9x1': lambda C, stride, affine:
    SepConv(C, C, 9, stride, 4, affine=affine),
}
```

6.3. iDARTS HSI

iDARTS HSI es una modificación de DARTS, el método implementado anteriormente, por lo cuál, se reusará esa implementación y se realizarán los cambios requeridos por la modificación del método, tal como viene descrito en la metodología.

La primera modificación que debe realizar es actualizar el modelo para que la selección de operación se realice con la función de activación sigmoide. Este cambio está reflejado en el algoritmo 21.

El siguiente paso a realizar es, una vez tenemos la selección de la operación adaptada, llevar los pesos hacia sus extremos 0 o 1 para evitar la discrepancia discutida en la metodología, por lo que se debe actualizar la pérdida según la función descrita, que cumple los criterios básicos necesarios. Así pues, se crea un nuevo valor de pérdida total en el proceso de entrenamiento, donde su valor corresponderá a la suma de la pérdida en el conjunto de validación más la pérdida extra *zero-one*, utilizada para llevar los pesos a 0 o 1. El diseño de esta pérdida puede verse en el algoritmo 22.

Para aplicar esta nueva pérdida sólo es necesario crear un instancia de la misma

Algorithm 21 Modelo de red en Improved DARTS

```
class Network(nn.Module):
...
def forward(self, input):
    s0 = s1 = self.stem(input)
    for i, cell in enumerate(self.cells):
        if cell.reduction:
            weights = F.sigmoid(self.alphas_reduce)
        else:
            weights = F.sigmoid(self.alphas_normal)
        s0, s1 = s1, cell(s0, s1, weights)
    out = self.global_pooling(s1)
    logits = self.classifier(out.view(out.size(0),-1))
    return logits
```

Algorithm 22 Modelo de red en Improved DARTS

```
class ConvSeparateLoss(nn.modules.loss._Loss):
    def __init__(self, weight=0.1, size_average=None, ignore_index=-100,
                 reduce=None, reduction='mean'):
        super(ConvSeparateLoss, self).__init__(size_average, reduce, reduction)
        self.ignore_index = ignore_index
        self.weight = weight
    def forward(self, input1, target1, input2):
        loss1 = F.cross_entropy(input1, target1)
        loss2 = -F.mse_loss(input2, torch.tensor(0.5, requires_grad=False).cuda())
        return loss1 + self.weight*loss2, loss1.item(), loss2.item()
```

y pasarla como parámetro al modelo de la arquitectura, que al calcular la pérdida utilizará ésta, como puede verse en el algoritmo 23 su definición y en el 24 la nueva pérdida del modelo de la red.

Algorithm 23 Instanciar nueva pérdida definida en Improved DARTS

```
criterion_train = ConvSeparateLoss(weight=args.aux_loss_weight)
```

Algorithm 24 Modelo de la red utilizando la nueva pérdida

```
class Network(nn.Module):
...
def _loss(self, input1, target1, input2):
    logits = self(input1)
    return self._criterion(logits, target1, input2)
```

7. RESULTADOS

En esta sección se evaluará la precisión y el rendimiento de los métodos propuestos en este trabajo, lo que ha precisado de un entorno de trabajo para ejecutar los diferentes experimentos. Se han utilizado algunas librerías para su desarrollo y varios conjuntos de datos hiperespectrales.

7.1. Entorno de trabajo

Los experimentos realizados se basan en dos modelos de búsquedas. Por un lado, una búsqueda con capas, el algoritmo genético, el cual utiliza modelos neuronales de perceptrón multicapa y red convolucional 2D. El otro modelo de búsqueda se basa en bloques convolucionales, con DARTS e Improved DARTS, que utiliza modelos de red convolucional 1D y 3D.

En total se presentan cuatro conjuntos de datos, donde por cada uno de ellos se realizan 5 experimentos por tipo de búsqueda y modelo para la obtención de resultados.

Las implementaciones de los algoritmos propuestos han sido desarrolladas y probadas en un entorno hardware con un procesador de la Generación X de Intel® Core™i9-9940X con 19.25M de caché y hasta 4.40GHz (14 núcleos con hyper-threading), instalado sobre una placa base Gigabyte X299 Aorus, 128GB de RAM DDR4. También se ha usado una unidad de procesamiento gráfico (GPU) NVIDIA Titan RTX GPU con 24GB GDDR6 de memoria de vídeo y 4608 núcleos. Con respecto a nivel de software, se ha utilizado un sistema operativo Ubuntu 18.04.3 LTS x64, CUDA 10, y Python 3.8, además de otras librerías relacionados con el *Machine Learning*, como *Scikit-Learn*, *Keras* (utilizado como back-end de *TensorFlow*) y *PyTorch*.

7.2. Datos y métricas de evaluación

Para realizar la evaluación de resultados se han usado un conjunto de datos hiperespectrales utilizados ampliamente en la comunidad de investigadores, los cuales se encuentras disponibles libremente[11]. La descripción completa de estos conjuntos de datos puede verse a continuación, así como las muestras total en la tabla 7:

- **Indian Pines**, corresponde a imágenes captadas por el sensor AVIRIS en el sitio de prueba de Indian Pines, en el noreste de Indiana. Consta de 145x145 píxeles, y originalmente con 224 bandas de reflectancia espectral, que tras la eliminación de 24 bandas ruidosas, deja un total de 200 bandas, con 16 clases etiquetadas.

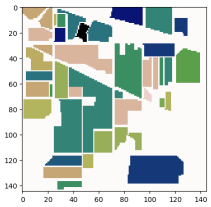
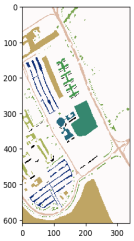
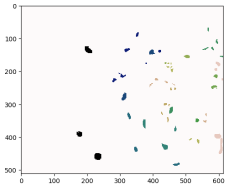
- **University of Pavia**, contiene imágenes adquiridas por el sensor ROSIS durante una campaña de vuelo sobre Pavía, en el norte de Italia. Sus imágenes tienen un total de 103 bandas espectrales, con una dimensión espacial de 610x340 píxeles. Este conjunto de datos está diferenciado por un total de 9 clases etiquetadas.

- **Salinas Valley**, recoge imágenes recopiladas por el sensor AVIRIS de 224 bandas sobre Salinas Valley, California, el cual se caracteriza por una alta resolución espacial (3,7 metros de píxeles). El área cubierta comprende 512 líneas por 217 muestras. Al igual que con Indian Pines, se descartan 20 bandas ruidosas, dejando un total de 204 bandas, con un total de 16 clases etiquetadas.

- **Kennedy Space Center**, obtenidas por el instrumento de la NASA AVIRIS, un espectrómetro de imágenes infrarrojas / visibles en el aire, que adquirió datos sobre el Centro Espacial Kennedy (KSC), en Florida, con un total de 224 bandas. Lás imágenes tomadas de KSC, las cuales se hicieron desde una altitud de aproximadamente 20 km, tienen una resolución espacial de 18 metros. Al descartar las bandas ruidosas, se usan 176 bandas para el análisis. Para fines de clasificación, se definen 13 clases etiquetadas.

[11] Grupo de Inteligencia Computacional (GIC), Hyperspectral Remote Sensing Scenes. Obtenidos el 12 de Septiembre de 2019.

Tabla 7: Número de muestra de los conjuntos de datos hiperespectrales Indian Pines (IP), University of Pavia (UP), Salinas Valley (SV) y Kennedy Space Center (KSC).

INDIAN PINES (IP)			UNIVERSIDAD DE PAVIA (UP)			VALLE DE SALINAS (SV)			CENTRO ESPACIAL KENNEDY (KSC)		
											
Color	Land-cover type	Samples	Color	Land-cover type	Samples	Color	Land-cover type	Samples	Color	Land-cover type	Samples
	Background	10776		Background	164624		Background	56975		Background	309157
	Alfalfa	46		Asphalt	6631		Brocoli-green-weeds-1	2009		Scrub	761
	Corn-notill	1428		Meadows	18649		Brocoli-green-weeds-2	3726		Willow-swamp	243
	Corn-min	830		Gravel	2099		Fallow	1976		CP-hammock	256
	Corn	237		Trees	3064		Fallow-rough-plow	1394		Slash-pine	252
	Grass/Pasture	483		Painted metal sheets	1345		Fallow-smooth	2678		Oak/Broadleaf	161
	Grass/Trees	730		Bare Soil	5029		Stubble	3959		Hardwood	229
	Grass/pasture-mowed	28		Bitumen	1330		Celery	3579		Swap	105
	Hay-windrowed	478		Self-Blocking Bricks	3682		Grapes-untrained	11271		Graminoid-marsh	431
	Oats	20		Shadows	947		Soil-vinyard-develop	6203		Spartina-marsh	520
	Soybeans-notill	972					Corn-senesced-green-weeds	3278		Cattail-marsh	404
	Soybeans-min	2455					Lettuce-romaine-4wk	1068		Salt-marsh	419
	Soybean-clean	593					Lettuce-romaine-5wk	1927		Mud-flats	503
	Wheat	205					Lettuce-romaine-6wk	916		Water	927
	Woods	1265					Lettuce-romaine-7wk	1070			
	Bldg-Grass-Tree-Drives	386					Vinyard-untrained	7268			
	Stone-steel towers	93					Vinyard-vertical-trellis	1807			
	Total samples	21025		Total samples	207400		Total samples	111104		Total samples	314368

Los valores resultados de los modelos ejecutados son comparados con otras técnicas y modelos del estado del arte, los cuales se listan a continuación.

- Random Forest (RF), es un modelo predictivo basdo en la creación de muchos árboles de decisión que trabajan en conjunto, donde cada árbol se construye con observaciones y variables aleatorias. El proceso inicial seleccionando individuos al azar, usando muestreo con reemplazo, para generar diferentes conjuntos de datos. Posteriormente crea un árbol de decisión para cada conjunto de datos, obteniendo diferentes árboles, ya que cada conjunto contiene individuos y variables diferentes. Al crear los árboles, se establecen variables al azar en cada nodo, haciendo crecer el árbol en profundidad. Para predecir los datos utiliza el ”voto de la mayoría”, donde se clasifica como acertado y la mayoría de los árboles predicen la observación positivamente.

- La regresión logística multinomial (MLR), una técnica estadística utilizada para predecir las probabilidades de los diferentes resultados posibles de una distribución categórica, como variable dependiente y un conjunto de variables

independientes (también llamadas predictoras). La idea principal es construir una función de predicción lineal que genere una puntuación a partir de un conjunto de pesos que se combinan linealmente con las variables dependientes:

$$\text{score}(X_i, k) = \beta_k \cdot X_i \tag{16}$$

donde X_i es el vector de variables dependientes describiendo la observación i, βk es un vector de pesos correspondientes al resultado k, y score$(\mathbf{X}_i, k)$ es la puntuación asociada de la observación i sobre la categoría k.

- Máquina de vectores de soporte (SVM), basadas en el concepto de hiperplano, son empleadas en clasificación binaria o regresión, donde se construye un hiperplano óptimo en forma de superficie de decisión, de modo que la separación entre ambas clases de los datos se maximiza.

Las métricas utilizadas en este trabajo son las consideradas ampliamente por la comunidad científica en este tipo de estudios:

- **La precisión media total**, o *overall accuracy (OA)*, es la media aritmética entre el número de muestras etiquetadas de forma correcta y el número total de muestras en ese conjunto de datos. Puede obtenerse a través de la matriz de confusión, sumando los valores de la diagonal principal, dividido por la suma de todos los valores de dicho matriz.

  Sea N el número de muestras totales del conjunto de datos, x el vector conteniendo las etiquetas reales, e y el vector de las etiquetas asignadas por el clasificador, podemos formalizar la media total como:

$$OA(x, y) = \frac{1}{N} \sum_{i=0}^{N-1} 1\,(x_i = y_i) \tag{17}$$

- **La precisión media ponderada**, del Inglés *average accuracy (AA)*, evalua la media de las muestras clasificadas correctamente, teniendo en cuenta aquellas que no han sido clasificadas de forma correcta. Al igual que la media total, hace uso de la matriz confusión, donde cada fila de ésta es una clase del conjunto de datos y la diagonal principal el resultado acumulado de las muestras clasificadas correctamente para la clase de esa fila. Si el valor de los elementos del resto de columnas es distinto de 0, indica que esas muestras están mal clasificadas. Comparada con la media total, este valor ofrece más información, ya que es posible averiguar qué clases están siendo peor clasificadas y en qué porcentaje. El resultado de esta métrica corresponde a la siguiente ecuación:

$$AA(x,y) = \frac{1}{N}\sum_{i=0}^{N-1}\frac{x_i}{y_i} \tag{18}$$

- **El coeficiente Kappa** es un índice estadístico que, al igual que la media ponderada, considera las muestras mal clasificadas, aunque en menor medida. Relaciona los resultados obtenidos y los esperados, al indicar un acuerdo mayor cuanto mayor sea el resultado de la función:

$$kappa = \frac{P_o - P_e}{1 - P_e} \tag{19}$$

  donde P_o es la probabilidad de producir una clasificación correcta y P_e la probabilidad de obtenerla al azar.

7.3. Análisis de resultados

Para la evaluación de resultados, se han elegidos dos porcentajes representativos del conjunto de datos, los cuales son usados durante la fase de entrenamiento. En concreto, los resultados se basan en un porcentaje del 10 % para los conjuntos de datos IP y KSC, y un porcentaje del 5 % para PU y SV. El conjunto de validación se forma con el 5 % de las muestras restantes, y el resto de muestras se utilizan en el conjunto de

pruebas (85 % en IP y KSC, y 90 % en PU y SV). Cabe aclarar que el modelo MLP del algoritmo genético se ha comparado con otro modelo MLP configurado con una capa oculta siguiendo la ecuación 20, la cual puede evitar el sobreajuste configurando el número de neuronas por debajo del valor obtenido [37].

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))} \tag{20}$$

siendo N_i el número de neuronas de entrada, N_o el número de neuronas de salida, N_s el número de muestras en el conjunto de datos de entrenamiento y α un factor escalar arbitrario.

En los resultados comparativos, se muestran como estado del arte los algoritmos usados tradicionalmente en clasificación: RF y MLR, los cuales son de tipo lineal, y SVM y MLP, que son no-lineales. Los algoritmos no-lineales clasifican mejor este tipo de datos frente a los algoritmos lineales, esto es debido a que no existe una relación fija para una mezcla, por ejemplo, la mezcla de agua y tierra, debido a que son dos elementos con impurezas, nunca se van a combinar de la misma forma, pues dan una mezcla no-lineal.

7.3.1. Resultados del algoritmo genétido en la búsqueda por capas

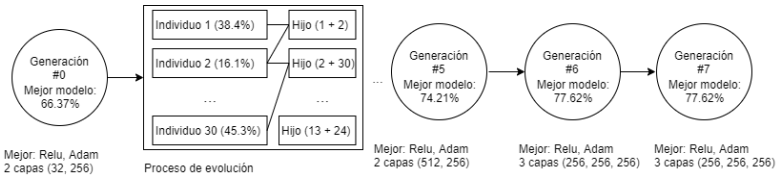


Figura 22: Evolución del algoritmo genético.

La figura 22 muestra la evolución del modelo guiada por el algoritmo genético. Como se puede observar, ya en la primera generación obtiene un modelo subóptimo basado en dos capas con 32 y 256 neuronas respectivamente, utilizando la función de activación ReLU, consigue una tasa de acierto en validación de 66,37 %, valor que, a través de la modificación de la arquitectura por medio de los procesos de

recombinación y mutación, llega a la última generación con una tasa de acierto del 77,62 %, con un modelo basado en 3 capas con 256 neuronas en cada una de ellas. Desde la primera generación hasta la última ha conseguido aumentar su modelo final en aproximadamente un 10 % sobre el conjunto de validación, lo que ha demostrado la eficacia de saltar mínimos locales. El entrenamiento del modelo final consigue una precisión media total (OA) del 81,3 %.

Los resultados de estas pruebas se encuentran en las tablas 8, 9, 10 y 11, donde el modelo CNN comúnmente utilizado, se encuentra por lo general, con una tasa de acierto inferior al modelo GenCNN, especialmente en el conjunto de datos IP, donde el genético consigue generar un modelo con un OA de 96,77 %, AA de 95,2 %, y valor Kappa de 96,32 %, superiores a su modelo rival CNN en un 1,88 %, 3,41 % y 2,14 % respectivamente.

class	RF	MLR	SVM	MLP	GenMLP	CNN	GenCNN
0	15.24±8.46	13.33±5.55	30.0±7.62	31.9±14.01	51.22±15.66	69.73±2.02	**95.14**±3.97
1	58.71±1.86	72.63±2.08	76.92±1.65	77.12±1.95	76.14±5.96	95.0±2.59	**97.15**±0.71
2	46.48±2.1	54.38±2.03	65.03±1.9	61.63±2.59	68.08±7.62	94.97±1.29	**96.88**±2.36
3	28.41±4.31	35.79±5.65	56.07±4.21	51.31±6.15	62.16±4.13	86.04±5.01	**93.44**±3.96
4	82.21±2.24	85.84±2.56	91.08±1.22	85.47±3.51	86.39±2.77	95.31±1.22	**97.14**±2.42
5	94.03±1.67	94.95±1.34	94.95±1.2	95.31±1.37	95.8±0.57	98.78±0.56	**99.39**±0.48
6	3.85±3.44	22.31±7.84	64.62±14.88	44.62±21.43	78.4±11.48	80.0±15.37	**91.82**±8.33
7	97.45±0.89	97.91±0.49	96.75±0.87	98.47±0.6	98.74±0.62	100.0±0.0	100.0±0.0
8	4.44±5.44	12.22±8.16	42.22±9.03	31.11±12.47	46.67±15.95	**88.75**±11.46	82.5±17.41
9	59.41±5.0	63.57±2.26	72.02±0.92	76.98±3.42	77.83±3.45	93.5±2.09	**95.25**±1.84
10	88.25±1.43	78.16±0.64	82.37±1.4	80.23±2.38	81.77±3.31	95.82±1.32	**97.52**±1.19
11	41.95±4.58	52.4±1.82	71.27±7.38	69.85±5.87	65.09±4.47	83.28±5.54	**91.27**±3.75
12	92.43±3.99	98.38±0.48	97.73±2.93	98.49±0.72	98.81±0.79	99.88±0.24	**100.0**±0.0
13	95.73±1.23	92.98±0.68	92.96±1.47	94.61±1.61	95.82±1.0	98.32±0.74	**98.89**±0.73
14	36.61±1.55	59.43±3.72	58.68±4.68	60.4±2.85	62.3±3.79	**90.64**±7.06	87.31±4.53
15	78.1±3.16	81.43±3.81	84.05±2.45	83.57±8.36	90.24±4.15	98.68±1.44	**99.47**±1.05
OA	72.75±0.43	75.06±0.54	80.32±0.55	79.89±0.23	81.3±1.26	94.89±0.23	**96.77**±0.71
AA	57.71±0.75	63.48±0.47	73.55±1.71	71.32±1.5	77.22±2.34	91.79±1.98	**95.2**±1.02
K(x100)	68.34±0.5	71.35±0.62	77.48±0.63	77.0±0.27	78.61±1.45	94.18±0.27	**96.32**±0.81

Tabla 8: Resultados sobre el conjunto de datos IP.

class	RF	MLR	SVM	MLP	GenMLP	CNN	GenCNN
0	90.69±1.55	91.57±1.07	92.15±0.81	91.91±1.37	93.51±1.25	**98.73**±0.33	98.63±0.76
1	97.62±0.3	95.71±0.32	98.07±0.26	97.28±0.14	97.6±0.57	**99.93**±0.03	99.89±0.07
2	55.14±3.88	71.74±1.88	73.75±1.08	73.6±3.92	76.7±5.55	93.24±1.97	**96.48**±1.49
3	86.53±2.25	87.76±0.66	90.72±0.69	89.23±1.65	91.63±2.84	98.97±0.56	**99.11**±0.42
4	98.31±0.64	99.11±0.3	98.84±0.61	99.23±0.59	99.77±0.16	99.95±0.1	**100.0**±0.0
5	46.52±2.1	76.7±1.28	85.23±0.72	88.92±1.22	89.01±0.36	**99.65**±0.24	99.63±0.22
6	71.85±6.19	54.41±6.4	82.63±2.48	78.58±4.07	81.2±3.53	93.55±2.58	**94.28**±2.81
7	87.24±0.49	86.82±1.2	89.08±1.55	88.52±1.31	84.16±3.73	96.6±0.8	**98.03**±0.95
8	99.24±0.24	99.76±0.11	99.38±0.23	99.38±0.27	99.44±0.43	**99.98**±0.05	99.65±0.37
OA	86.02±0.34	89.24±0.17	92.72±0.29	92.5±0.18	92.94±0.61	98.83±0.07	**99.1**±0.11
AA	81.46±0.87	84.84±0.41	89.98±0.6	89.63±0.64	90.34±0.83	97.84±0.2	**98.41**±0.28
K(x100)	80.95±0.51	85.61±0.22	90.29±0.39	90.02±0.25	90.62±0.8	98.45±0.1	**98.81**±0.15

Tabla 9: Resultados sobre el conjunto de datos UP.

class	RF	MLR	SVM	MLP	GenMLP	CNN	GenCNN
0	94.95±0.63	95.01±0.91	92.91±1.22	95.88±1.95	94.39±0.39	99.38±1.07	**99.84**±0.32
1	84.93±4.11	91.05±4.07	87.76±2.98	88.4±3.27	87.4±2.9	**92.99**±3.91	87.92±5.48
2	84.68±3.74	85.02±3.72	79.22±6.64	88.23±1.84	90.96±4.25	89.95±8.64	**94.88**±4.13
3	67.31±6.07	62.73±10.17	67.49±4.29	73.74±3.86	70.66±10.13	63.82±3.35	**78.92**±4.71
4	52.0±5.85	68.0±6.07	61.52±8.48	53.24±8.83	65.1±8.31	76.67±7.01	**84.99**±4.8
5	42.03±5.09	74.4±4.23	57.97±8.21	53.14±5.23	71.16±4.41	87.67±4.14	**92.33**±7.33
6	76.42±8.46	79.79±3.42	64.84±10.41	72.84±12.12	83.41±7.78	96.0±3.54	**96.23**±6.37
7	79.79±4.37	89.64±1.61	87.22±2.9	89.74±2.06	93.56±1.91	97.54±1.31	**99.43**±0.54
8	94.87±4.01	95.94±0.98	94.36±1.19	97.86±1.46	97.05±1.16	**99.67**±0.66	99.38±0.28
9	85.22±3.13	92.14±3.17	91.48±3.19	92.91±3.34	97.31±1.43	**100.0**±0.0	98.42±3.02
10	97.67±1.67	97.94±1.33	95.5±1.63	96.14±1.72	96.76±1.57	99.77±0.34	**99.94**±0.12
11	87.33±3.28	94.57±1.0	88.79±4.17	92.63±1.1	91.7±1.29	97.16±2.32	**98.23**±1.31
12	99.59±0.36	100.0±0.0	98.68±0.61	100.0±0.0	99.52±0.74	99.95±0.11	100.0±0.0
OA	86.9±0.75	91.23±0.37	88.09±0.49	90.32±0.79	91.87±0.31	95.46±0.64	**96.86**±0.52
AA	80.52±1.38	86.63±0.68	82.13±1.02	84.21±1.24	87.61±0.51	92.35±0.95	**94.65**±0.98
K(x100)	85.4±0.83	90.23±0.41	86.74±0.55	89.21±0.87	90.95±0.34	94.95±0.72	**96.5**±0.58

Tabla 10: Resultados sobre el conjunto de datos KSC.

class	RF	MLR	SVM	MLP	GenMLP	CNN	GenCNN
0	99.48±0.16	98.88±0.59	98.87±0.4	98.95±0.53	98.95±0.71	**99.93**±0.13	99.5±0.99
1	99.77±0.09	99.88±0.07	99.57±0.21	99.8±0.09	98.03±3.82	99.98±0.02	**100.0**±0.0
2	98.54±0.83	97.74±0.98	99.0±0.31	99.22±0.44	99.18±0.57	99.62±0.43	**99.9**±0.13
3	98.54±0.97	99.28±0.29	99.25±0.33	99.34±0.36	99.35±0.58	**99.57**±0.22	99.14±0.65
4	97.56±0.48	98.5±0.48	97.93±0.71	98.15±0.56	97.93±0.53	**98.77**±0.47	98.13±2.21
5	99.52±0.16	99.96±0.02	99.64±0.11	99.81±0.09	99.8±0.05	**100.0**±0.0	100.0±0.0
6	99.32±0.14	99.57±0.12	99.44±0.11	99.52±0.16	99.76±0.11	**99.88**±0.19	99.74±0.14
7	82.97±0.97	87.4±1.19	87.59±1.23	86.68±0.92	87.07±2.62	**97.42**±0.88	96.13±0.65
8	99.04±0.16	99.56±0.28	99.34±0.22	99.66±0.27	99.52±0.36	99.96±0.05	**99.97**±0.03
9	89.12±1.96	94.27±1.8	92.53±1.21	94.39±0.56	93.54±0.56	98.28±0.62	**98.55**±0.82
10	91.19±2.76	95.19±1.41	95.29±1.32	96.35±1.43	92.9±3.69	97.88±0.85	**98.57**±0.44
11	98.53±0.53	99.72±0.17	99.18±0.49	99.08±0.43	99.52±0.35	99.82±0.29	**99.96**±0.05
12	97.54±0.7	98.67±0.84	97.61±0.41	98.21±0.92	98.37±0.9	**99.93**±0.14	99.4±0.62
13	91.5±1.21	93.61±1.33	93.24±2.28	94.34±1.5	95.89±1.49	99.17±0.3	**99.48**±0.63
14	59.19±1.16	64.59±1.68	68.64±0.94	71.55±0.54	67.67±3.99	96.98±1.42	**97.59**±0.9
15	96.03±1.11	97.9±0.49	98.01±0.41	98.4±0.57	98.28±0.62	99.17±0.88	**99.8**±0.21
OA	89.31±0.14	91.64±0.13	92.02±0.26	92.49±0.18	91.85±0.58	**98.76**±0.16	98.57±0.26
AA	93.62±0.2	95.29±0.22	95.32±0.2	95.84±0.17	95.36±0.43	**99.15**±0.07	99.12±0.17
K(x100)	88.08±0.16	90.68±0.15	91.1±0.29	91.64±0.19	90.91±0.64	**98.62**±0.18	98.41±0.29

Tabla 11: Resultados sobre el conjunto de datos SV.

En ambos modelos MLP se puede observar que existen clases que tienen una precisión elevada, mientras otras se quedan en valores mucho más bajos, tal vez debido a la incapacidad del perceptrón multicapa para establecer relaciones espectro-espaciales, lo que lleva a que no pueda detectar ciertas características complejas de esas clases. Otra posible causa, no excluyente a la anterior, es que la selección del conjunto de datos de entrenamiento podría no tener suficientes muestras de esas clases y si de las otras. Aún dándose estas deficiencias, el modelo generado genéticamente logra ser mejor que otras técnicas utilizadas en 3 de los 4 conjuntos, demostrando la eficacia de la búsqueda de arquitectura neuronal en datos hiperespectrales para el modelo MLP. En los modelos CNN, como la información que entra a la red tiene información del espacio y del espectro, es el mejor resultado en todos los conjuntos de datos, superando los modelos generados genéticamente a otras técnicas y modelos enfrentados en 3 de los 4 conjuntos de datos.

Todos los modelos generados genéticamente consiguen superar a los modelos actuales, excepto en el conjunto de datos SV, tanto para MLP como CNN. Esto puede deberse a que la tasa de aprendizaje, o la estrategía de búsqueda, son demasiados agresivas para los datos incluídos en el mismo, el cuál son más fácilmente relacionables que, por ejemplo, IP, el cuál se considera el conjunto de datos más difícil de clasificar y en el que se consiguen los mejores resultados de los modelos generados genéticamente, con respecto a MLP y CNN.

7.3.2. Resultados de DARTS en la búsqueda por bloques

La evaluación de los resultados se ha configurado con un conjunto de 200 muestras para el entrenamiento y 100 pruebas para la validación. El resto de muestras se ha utilizado para la evaluación final del modelo. Al igual que los resultados de la búsqueda del algoritmo genético, se presentan algunos métodos clásicos de clasificación basados en la relación de la información espectral. Estos métodos clásicos quedan obsoletos por la inferioridad de su eficacia en clasificación comparados con los métodos propuestos. DARTS 1D, DARTS HSI e iDARTS HSI han sido configurados para utilizar dos capas

en el proceso de búsqueda del modelo y 3 capas en el procesamiento de entrenamiento del mismo. Las tablas 12, 13, 14 y 15 muestran los resultados obtenidos con los ajustes experimentales anteriores.

El método iDARTS HSI muestra los mejores resultados, en términos de precisión, en casi todos los conjuntos de datos, especialmente en IP, donde consigue un OA de 68,29 % sobre un 65,35 % de su predecesor, obteniendo un aumento del 2,94 %, un 3,83 % del OA y un 3,54 % del valor Kappa. El único conjunto de datos donde los métodos propuestos parecen fallar es en KSC, los cuales se enfrentan a unas tasas muy parecidas para las 3 medidas utilizadas, con una variación negativa del 0,54 % del OA, un 0,89 % de AA y un 0,62 % del valor Kappa, con respecto al mejor modelo para este conjunto de datos, CNN1D.

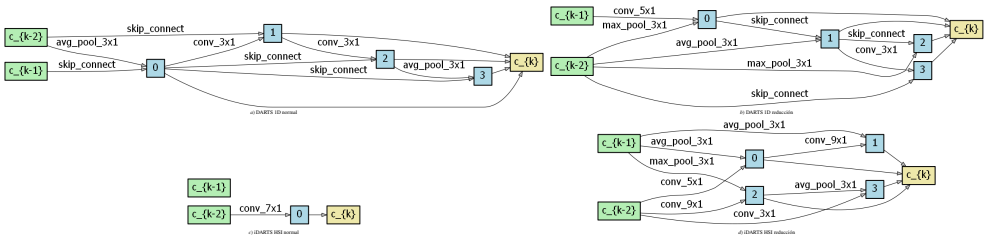


Figura 23: Celdas normal y de reducción para DARTS e iDARTS en el conjunto de datos IP.

class	RF	MLR	SVM	MLP	CNN1D	DARTS 1D	DARTS HSI	iDARTS 1D
0	20.35±30.82	18.16±34.14	21.27±33.45	1.82±3.64	0.0±0.0	**26.91**±7.8	19.58±19.42	9.84±8.6
1	45.82±13.03	58.21±4.89	59.93±5.25	52.58±4.45	53.7±3.02	52.3±14.73	60.16±8.24	**62.89**±4.9
2	35.5±8.39	40.51±10.34	46.26±5.59	21.53±9.91	46.95±8.13	49.87±2.61	44.12±12.07	**57.83**±3.4
3	18.3±3.92	27.31±5.24	**40.57**±7.03	24.53±5.73	23.34±13.63	30.32±20.9	25.35±15.82	30.43±7.51
4	55.41±14.69	56.92±19.33	73.77±8.31	52.27±13.79	72.14±6.64	**80.82**±7.64	77.8±4.54	67.03±9.08
5	88.49±8.4	91.56±4.7	92.08±3.71	90.07±4.25	**93.68**±2.9	91.31±3.81	90.99±4.86	91.32±6.28
6	15.33±27.1	24.13±24.31	31.68±32.27	8.15±14.52	5.93±11.85	0.0±0.0	**35.63**±36.89	35.42±46.25
7	85.6±17.82	83.74±22.01	84.51±20.78	**98.37**±1.9	92.88±4.33	88.63±5.51	95.87±5.24	95.92±1.38
8	3.16±6.32	3.16±4.21	8.42±10.84	**8.42**±10.31	0.0±0.0	0.0±0.0	5.31±9.74	0.0±0.0
9	47.4±4.15	50.46±7.28	56.75±5.26	52.64±6.55	49.95±4.22	49.29±7.35	55.93±9.41	**74.47**±6.18
10	**72.84**±6.42	70.08±4.49	69.09±2.48	71.47±3.46	70.65±4.31	68.61±6.55	68.73±9.50	64.39±6.2
11	18.18±10.04	22.33±8.37	33.42±10.65	34.08±10.11	31.25±7.08	40.37±12.31	36.96±7.91	**42.9**±2.69
12	49.6±40.6	48.23±39.44	56.49±40.23	92.54±13.67	80.57±15.14	**97.62**±0.73	87.33±10.85	94.49±2.19
13	89.08±11.26	89.75±7.56	86.93±6.74	**90.49**±4.22	92.51±2.6	85.59±5.17	84.90±4.19	89.51±5.88
14	24.09±13.01	32.62±8.71	31.73±9.93	25.03±10.59	23.57±7.82	26.25±22.73	**39.02**±17.05	32.17±10.77
15	47.51±23.86	65.28±27.61	82.52±9.35	43.96±36.54	56.31±25.35	86.01±5.71	48.02±42.87	**88.43**±6.41
OA	59.02±2.36	61.98±3.28	64.85±2.06	62.35±1.49	64.1±0.82	64.3±1.58	65.35±2.94	**68.29**±0.8
AA	44.79±2.33	48.9±2.76	54.71±2.22	48.62±3.07	49.59±2.11	54.62±0.74	54.73±4.22	**58.56**±3.65
K(x100)	52.47±2.61	56.15±3.73	59.74±2.35	56.55±1.64	58.57±1.06	59.23±1.6	60.41±3.23	**63.95**±0.79

Tabla 12: Resultados de DARTS e iDARTS sobre el conjunto de datos IP.

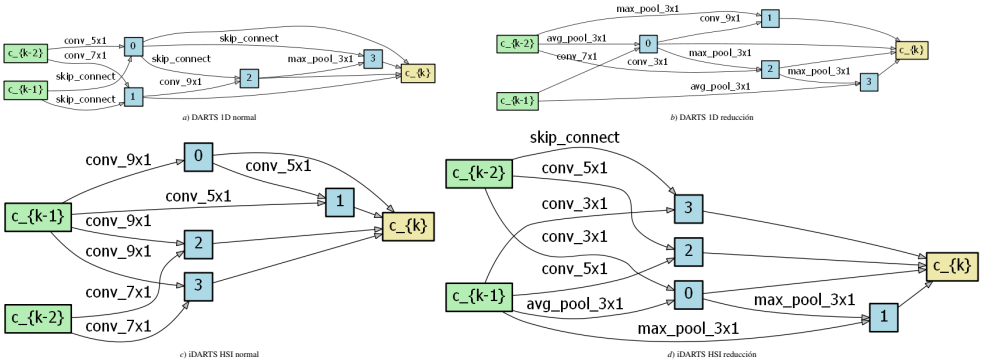


Figura 24: Celdas normal y de reducción para DARTS e iDARTS en el conjunto de datos UP.

class	RF	MLR	SVM	MLP	CNN1D	DARTS 1D	DARTS HSI	iDARTS HSI
0	82.37±5.6	84.14±2.57	82.51±4.6	84.76±5.32	**87.79**±3.14	86.52±0.65	83.40±3.40	79.69±1.51
1	94.9±1.7	93.45±1.84	94.95±2.57	94.53±2.66	93.98±3.01	**95.22**±3.81	93.32±3.18	94.76±2.84
2	38.37±7.57	58.95±9.37	58.3±6.55	29.19±17.19	50.08±12.87	65.63±11.82	61.52±5.05	**79.44**±3.92
3	72.91±7.48	79.45±5.38	76.67±10.23	74.2±6.06	81.23±4.77	79.35±11.73	78.86±4.38	**86.54**±1.73
4	91.43±12.02	93.41±4.94	78.85±18.18	98.56±0.43	94.42±7.53	98.7±0.72	98.25±0.98	**98.82**±0.68
5	32.87±5.62	58.54±2.28	46.95±10.58	39.83±12.91	72.07±7.07	64.85±18.91	73.34±5.42	**81.92**±0.52
6	41.95±14.94	29.81±15.81	45.6±23.45	41.37±28.68	60.91±11.69	66.82±2.81	64.56±9.15	**73.55**±11.55
7	82.07±7.18	79.14±8.94	75.61±6.14	**86.41**±3.56	85.57±7.02	70.74±9.11	76.86±7.29	64.97±11.16
8	99.09±0.11	96.38±5.19	97.92±1.98	99.02±0.61	98.47±1.22	99.79±0.11	97.69±2.30	**100**±0.0
OA	78.55±1.2	82.06±0.93	80.63±1.03	79.79±1.37	85.74±1.18	84.93±0.58	84.63±1.80	**86.65**±0.74
AA	70.66±0.62	74.81±1.6	73.04±2.52	71.99±3.44	80.5±2.15	80.85±0.42	80.87±1.64	**84.41**±1.77
K(x100)	70.6±1.56	75.9±1.28	73.57±1.59	72.41±1.93	80.88±1.59	79.71±0.7	79.70±2.26	**82.28**±0.94

Tabla 13: Resultados de DARTS e iDARTS sobre el conjunto de datos UP.

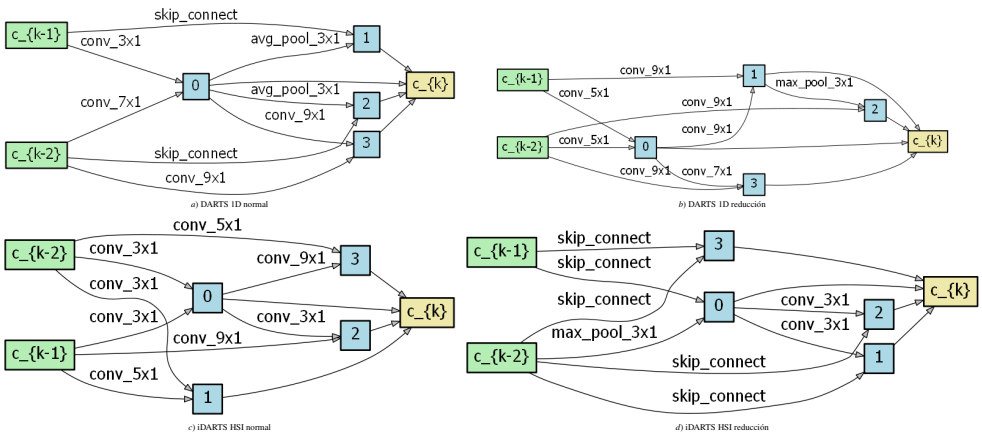


Figura 25: Celdas normal y de reducción para DARTS e iDARTS en el conjunto de datos SV.

textbfclass	RF	MLR	SVM	MLP	CNN1D	DARTS 1D	DARTS HSI	iDARTS HSI
0	92.28±5.83	92.96±4.68	90.17±10.04	95.25±1.49	96.69±3.16	91.48±9.64	96.75±1.56	**97.43**±1.51
1	99.34±0.96	98.88±1.46	98.62±0.99	97.61±3.34	**99.78**±0.16	99.62±0.26	99.26±0.45	99.66±0.19
2	72.35±11.82	87.03±7.39	80.69±9.01	87.47±9.44	92.16±5.86	**96.29**±4.56	79.46±21.58	87.92±8.95
3	93.97±1.97	98.57±0.67	96.0±1.83	97.17±2.6	96.63±3.91	95.79±5.48	99.09±0.56	**99.31**±0.98
4	92.87±3.78	91.81±7.59	91.84±4.55	**97.29**±1.21	96.64±2.05	90.4±3.44	97.21±1.46	96.76±4.18
5	97.27±1.41	99.39±0.31	95.11±5.17	99.36±0.31	98.71±0.97	99.17±1.11	**99.68**±0.17	99.53±0.24
6	97.59±1.2	99.37±0.26	99.27±0.09	**99.44**±0.22	99.39±0.08	99.43±0.14	99.35±0.25	99.06±0.19
7	69.63±2.94	71.97±6.55	70.43±4.09	78.87±4.62	79.67±3.56	80.69±3.37	75.82±10.02	**84.88**±3.67
8	98.34±1.04	99.37±0.27	98.58±0.82	98.96±1.23	99.44±0.29	98.94±0.82	99.05±0.75	**99.53**±0.21
9	75.41±17.4	84.81±4.81	82.98±12.64	78.63±3.57	84.21±5.97	76.29±0.80	87.54±3.78	**90.77**±4.08
10	72.92±12.09	79.02±12.21	81.38±8.5	89.09±3.84	**90.29**±2.8	87.40±1.94	89.15±2.30	88.51±1.31
11	92.61±11.81	98.69±0.92	94.63±6.18	94.27±10.16	**99.6**±0.66	99.36±0.57	96.99±4.20	91.18±4.92
12	81.92±27.55	90.68±13.74	97.15±2.31	79.7±26.69	**98.84**±1.08	96.97±3.35	98.36±1.08	98.30±0.93
13	77.35±10.34	85.36±4.04	76.88±11.58	91.44±1.0	85.44±4.94	90.86±1.16	90.61±5.26	**91.86**±2.98
14	52.71±4.46	58.51±5.98	56.21±2.84	46.71±8.18	62.05±7.99	61.75±8.49	**63.47**±10.73	56.76±4.68
15	82.59±3.64	87.55±2.9	87.0±3.95	81.7±9.51	87.48±15.97	88.23±10.9	89.26±5.84	**96.6**±1.23
OA	81.3±1.74	84.93±1.71	83.19±1.16	84.39±1.43	87.9±0.81	87.23±0.96	87.15±1.37	**88.88**±0.36
AA	84.32±2.67	89.0±1.17	87.3±1.18	88.31±1.61	91.69±0.91	90.79±0.12	91.32±1.63	**92.38**±0.04
K(x100)	79.17±1.93	83.22±1.89	81.28±1.3	82.58±1.61	86.51±0.91	85.75±1.09	85.70±1.50	**87.56**±0.41

Tabla 14: Resultados de DARTS e iDARTS sobre el conjunto de datos SV.

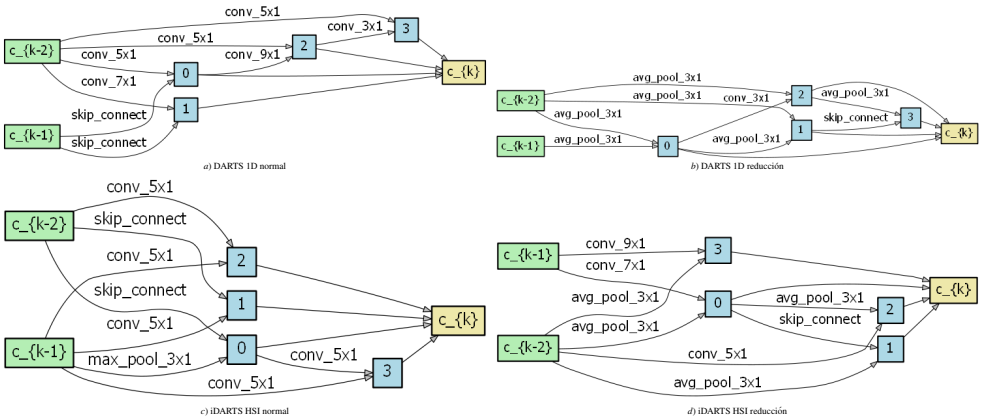


Figura 26: Celdas normal y de reducción para DARTS e iDARTS en el conjunto de datos KSC.

class	RF	MLR	SVM	MLP	CNN1D	DARTS 1D	DARTS HSI	iDARTS HSI
0	92.86±4.11	94.32±1.71	91.69±3.9	91.52±3.74	94.38±1.96	92.45±0.46	88.96±7.77	**96.02**±3.00
1	77.93±11.56	87.29±7.28	73.27±12.79	88.32±8.63	88.59±5.15	84.77±2.68	**89.38**±3.42	79.88±11.79
2	82.73±3.35	81.49±8.31	77.99±4.32	**88.29**±5.87	88.01±5.37	84.44±2.27	83.74±6.75	84.43±9.52
3	51.62±12.99	54.03±15.48	47.77±11.45	40.52±32.0	76.26±5.03	62.71±8.45	60.08±10.93	**80.18**±5.88
4	39.12±19.97	41.36±23.04	37.37±10.72	36.93±24.23	49.09±7.43	**68.91**±2.51	55.35±9.88	51.83±6.36
5	40.42±10.34	**59.55**±9.8	45.79±12.43	56.07±9.89	56.09±7.79	51.23±8.10	56.79±9.95	39.33±3.56
6	72.31±20.29	75.26±22.04	56.19±16.89	54.72±34.93	69.5±21.14	**89.49**±14.67	68.58±19.96	75.64±9.80
7	57.48±18.86	85.77±4.04	81.17±3.77	82.29±10.58	87.63±6.55	87.96±3.88	88.40±4.58	**91.25**±1.67
8	88.07±4.89	94.08±4.6	89.18±5.63	89.37±7.02	97.09±1.63	91.13±9.72	94.30±4.69	**97.36**±0.54
9	74.97±9.08	86.46±6.81	77.55±7.09	86.45±4.29	93.46±0.48	85.76±6.99	**95.28**±2.37	88.82±2.47
10	92.83±3.86	95.07±3.24	86.42±4.07	93.65±2.15	91.36±4.57	**96.12**±0.51	95.40±4.37	95.13±5.75
11	76.65±7.65	90.7±4.24	75.23±5.4	87.01±3.42	87.46±6.41	83.68±7.65	**94.17**±2.25	88.34±5.77
12	99.6±0.19	99.98±0.04	98.14±0.38	100±0.0	99.32±0.69	97.86±0.07	100±0.00	98.40±0.34
OA	80.08±1.65	87.21±0.99	80.51±1.51	84.57±2.14	**88.92**±0.81	87.04±0.44	88.40±2.02	88.38±0.97
AA	72.81±2.11	80.41±1.71	72.14±2.53	76.55±3.61	**82.94**±1.37	82.81±0.65	82.34±2.63	82.05±0.77
K(x100)	77.78±1.84	85.76±1.09	78.27±1.71	82.81±2.37	**87.66**±0.91	85.58±0.50	87.08±2.23	87.04±1.08

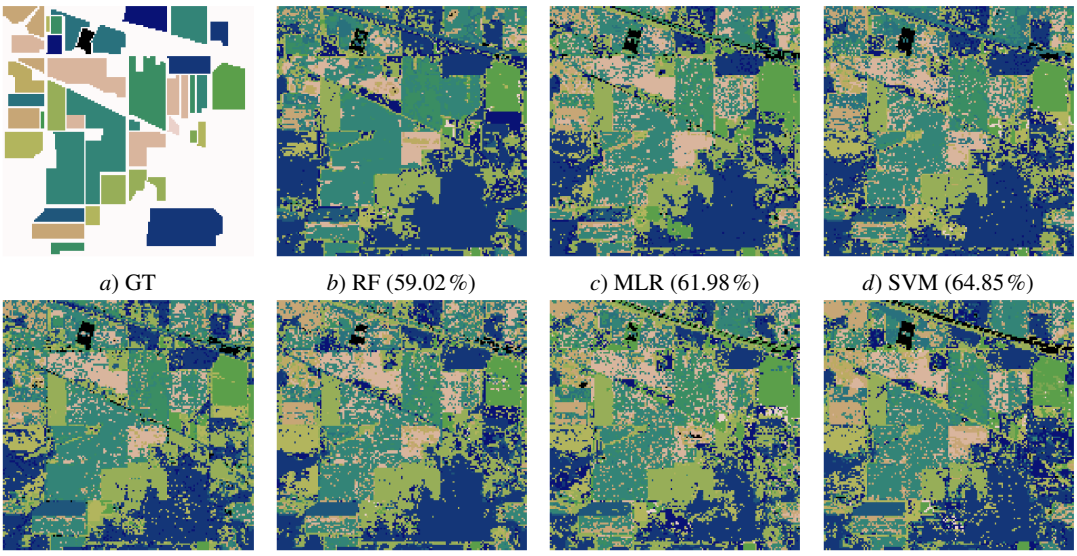
Tabla 15: Resultados de DARTS e iDARTS sobre el conjunto de datos KSC.

Debido al limitado número de muestras de entrenamiento y a que en el conjunto de

datos IP existen ciertos tipos de verdad-terreno una cantidad de píxeles muy limitada (por ejemplo las clases *Oats* y *Grass/pasture-mowed*), pueden existir clases en las que la tasa de acierto del modelo sea 0 %, lo cuál nos indica que para esa clase, lo más probable es que ninguna muestra haya formado parte del conjunto de entrenamiento, tanto para la búsqueda como la evaluación del modelo, por lo que es técnicamente imposible para el modelo conocer durante el proceso de inferencia lo que nunca ha visto durante el entrenamiento.

Dataset	CNN1D	DARTS 1D	iDARTS HSI
Indian Pines	72,416	**45,520**	46,669
Pavia University	**33,709**	47,632	34,701
Salinas Valley	74,416	**65,024**	67,837
Kennedy Space Center	62.113	72,544	**46,925**

Tabla 16: Número de parámetros entrenables en los conjuntos de datos IP, PU, SV y KSC. En negrita se indican los mejores resultados, y en azul el segundo mejor resultado.



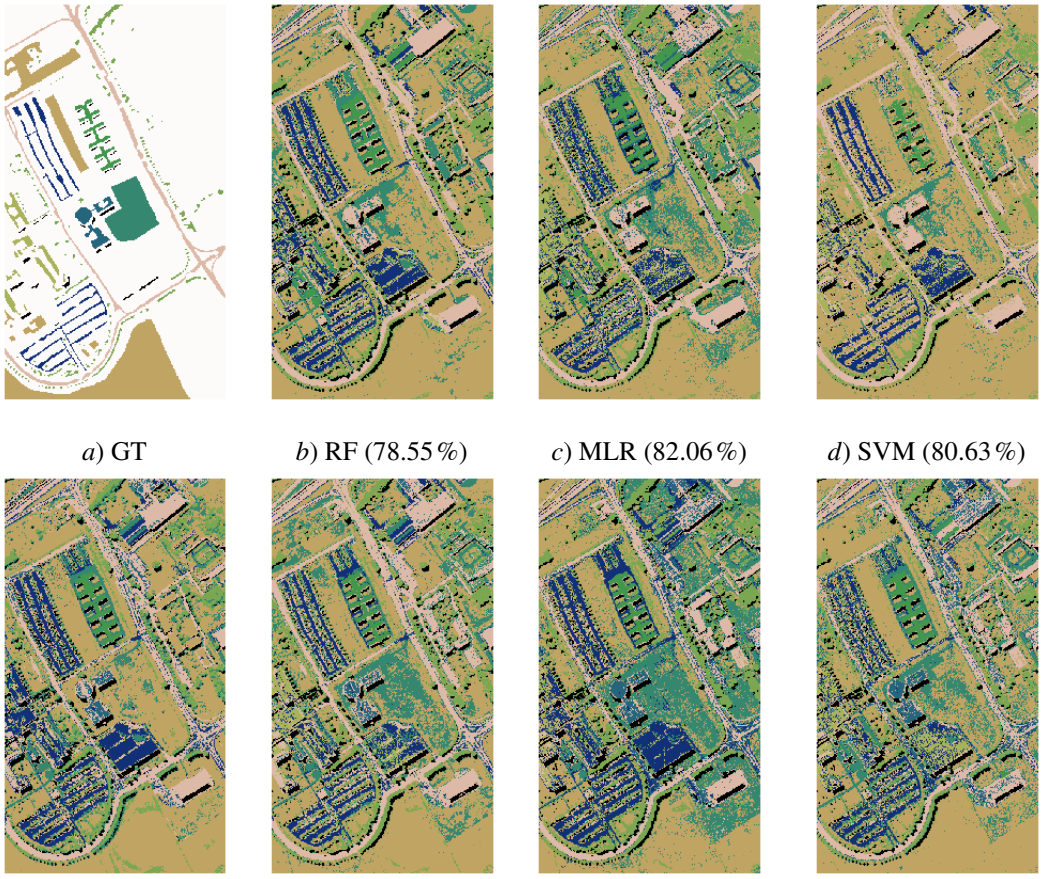
a) GT *b*) RF (59.02 %) *c*) MLR (61.98 %) *d*) SVM (64.85 %)

e) MLP (62.35 %) *f*) CNN1D (64.10 %) *g*) DARTS HSI (64.3 %) *h*) iDARTS HSI (**68.29 %**)

Figura 27: Mapas de clasificación obtenidos para el conjunto de datos IP.

Como se observa en la tabla 16, los modelos generados por DARTS HSI e iDARTS HSI, figuras 23, 24, 25 y 26, tienen menos parámetros en general que el modelo de red convolucional 1D estándar utilizado comúnmente en estos conjuntos de datos. Los mejores resultados son marcados en negritas, y el segundo mejor resultado para el

conjunto de datos se marca en azul. Ambos métodos se mueven en torno al mismo orden de número de parámetros de red, aunque iDARTS HSI consigue modelos con muchos menos parámetros en los conjuntos de datos UP y KSC, sin despreciar por ello su tasa de acierto, que además se ve aumentada en UP, por un 2,02 % de su OA, un 3,54 % de su AA y un 2,58 % del valor Kappa, lo que los convierte en modelos significativamente más eficientes, mostrando una clara ventaja de iDARTS HSI frente a DARTS HSI. En aquellos casos donde no consigue ser el mejor modelo, rivaliza con el primero obtenido.



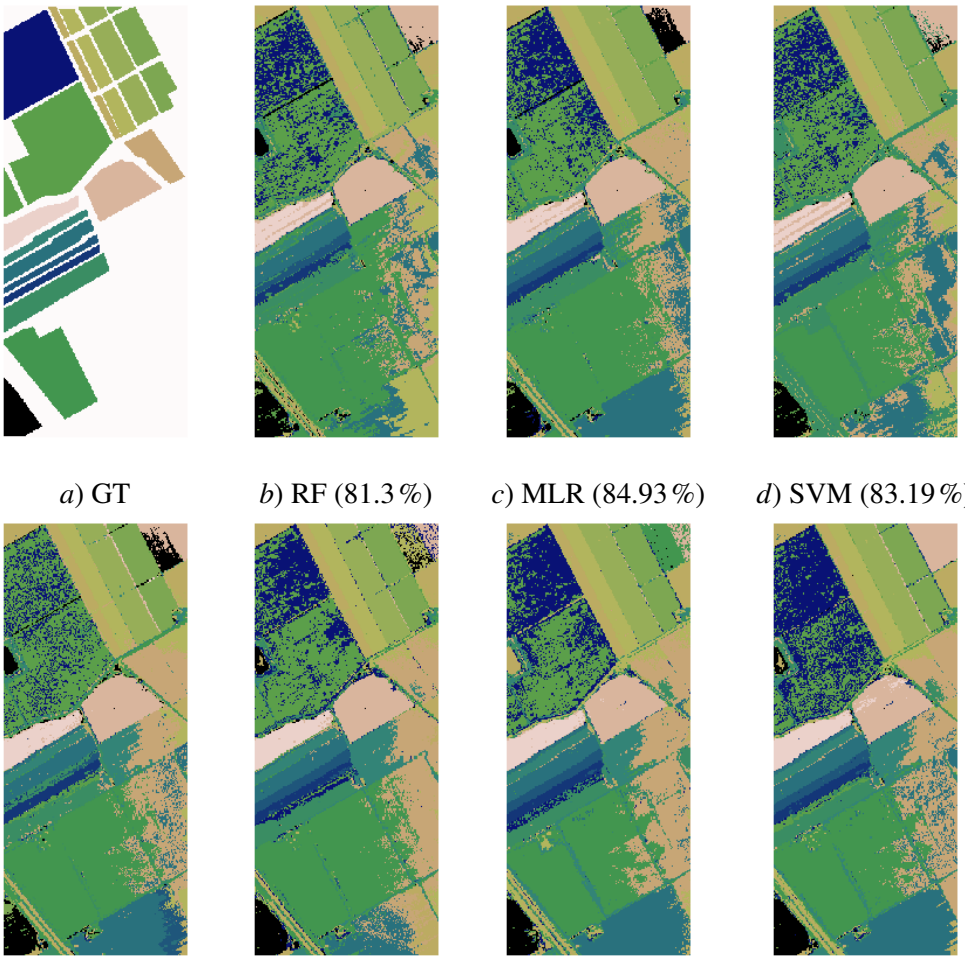
a) GT *b*) RF (78.55 %) *c*) MLR (82.06 %) *d*) SVM (80.63 %)

e) MLP (79.79 %) *f*) CNN1D (85.74 %) *g*) DARTS HSI (84.93 %) *h*) iDARTS HSI (**86.65 %**)

Figura 28: Mapas de clasificación obtenidos para el conjunto de datos UP.

Además, se puede observar una clara reducción de las skip-connections como mejora implementada en iDARTS HSI para ambos tipos de celdas, pues las formulación del método impide que haya esa predominancia hacia las

skip-connections, dando igualdad de oportunidades a todas las operaciones, lo que demuestra la eficacia de la solución propuesta.



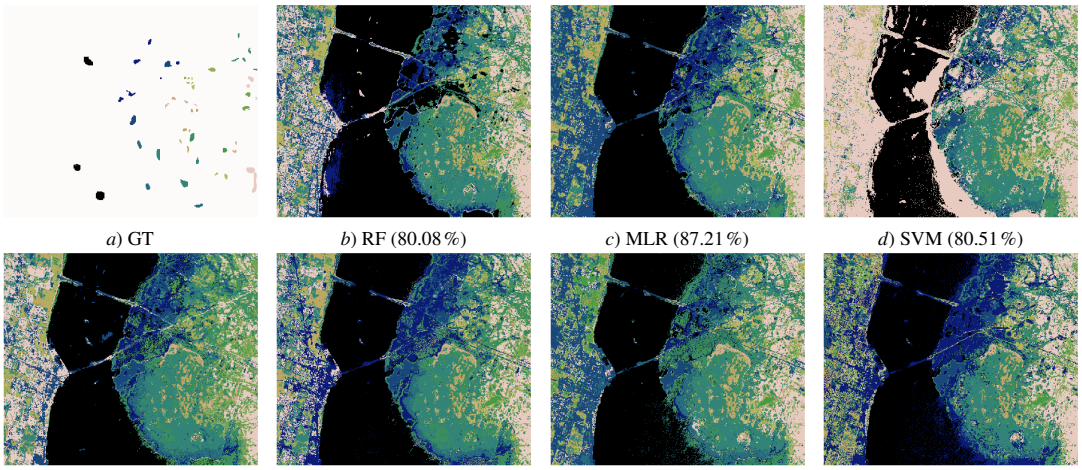
a) GT *b*) RF (81.3 %) *c*) MLR (84.93 %) *d*) SVM (83.19 %)

e) MLP (84.39 %) *f*) CNN1D (87.9 %) *g*) DARTS HSI (87.23 %) *h*) iDARTS HSI (**88.88 %**)

Figura 29: Mapas de clasificación obtenidos para el conjunto de datos SV.

En las figuras 27, 28, 29 y 30 se evalúa la tasa de acierto de clasificación desde una perspectiva visual. Los modelos entrenados, incluyendo RF, MLR, SVM, MLP y CNN1D, son seleccionados para clasificar las imágenes completas, donde sus parámetros ya han sido optimizados en el proceso de evaluación. A través de las imágenes resultantes se puede observar cómo los diferentes métodos de clasificación afectan a los resultados de clasificación, donde se ve obvio que utilizar sólamente características espectrales produce puntos de dispersión ruidosos en el mapa de

clasificación. La imagen de *ground-truth* se muestra a modo de referencia para la comparación visual con los mapas de clasificación.



a) GT *b*) RF (80.08 %) *c*) MLR (87.21 %) *d*) SVM (80.51 %)

e) MLP (84.57 %) *f*) CNN1D (**88.92 %**) *g*) DARTS HSI (87.04 %) *h*) iDARTS HSI (88.38 %)

Figura 30: Mapas de clasificación obtenidos para el conjunto de datos KSC.

8. CONCLUSIONES Y TRABAJO FUTURO

Se ha desarrollado un algoritmo genético, con un buen mecanismo adaptado para evitar mínimos locales, que es capaz de encontrar modelos óptimos de perceptrón multicapa y convolucionales adaptados a los conjunto de datos propuestos. Sus resultados, comparados con otros modelos del estado-del-arte, son más que adecuados, llegando a superarlos de forma genérica. Otros métodos propuestos basados en el gradiente, DARTS e iDARTS, también han logrado la obtención de mejores resultados en la obtención de modelos con mejores tasas de acierto comparados con el estado-del-arte. iDARTS, corrigiendo los problemas que presenta DARTS de base, y que hacen mermar el rendimiento, obtiene los mejores resultados, permitiendo encontrar modelos óptimos para casi todos los conjuntos de datos propuestos.

Estas técnicas nos permiten generar modelos de redes neuronales óptimos de forma automática, comparables con modelos del estado-del-arte de los conjuntos de datos sobre los que se realiza la búsqueda. La ventaja es sustancial, pues permite ahorrar costes de personal experto y mucho tiempo de ensayo prueba y error hasta dar con un modelo óptimo.

Como proyecto futuro, se propone mejorar el algoritmo genético actual, para añadir *skip-connections*, lo que se prevee que mejorará la eficiencia de los modelos generados. También se debe solucionar el problema presente en el algoritmo genético al buscar conjuntos de datos simples, pues la estrategia tan agresiva genera modelos subóptimos en estos casos. Para fomentar la ejecución del algoritmo genético y aumentar aún más su eficacia, se insta a la creación de un modelo de islas donde cada isla es ejecutada en un entorno diferente. Este entorno de islas es la distribución del algoritmo genético, donde las islas están comunicadas a través de un mecanimo aún no determinado, por ejemplo, un nodo central que actúa de transporte entre todas las poblaciones, que limite la entrada y salida de ciertos tipos de individuos con un criterio, para añadir nuevo material genético a las demás poblaciones, incluida ella misma.

Dando un paso más, se plantea combinar las ideas de DARTS, como el uso de

celdas y compartición de pesos sobre todas las soluciones de la arquitectura, utilizando una versión fusionada con algoritmos genéticos para diseñar el subgráfico de las celdas de convolución, cuyo objetivo es maximizar la tasa de acierto en el conjunto de validación.

Referencias

[1] J. M. Bioucas-Dias, A. Plaza, G. Camps-Valls, P. Scheunders, N. Nasrabadi, and J. Chanussot. Hyperspectral remote sensing data analysis and future challenges. *IEEE Geoscience and Remote Sensing Magazine*, 1(2):6–36, June 2013.

[2] Antonio Plaza, Jon Atli Benediktsson, Joseph W. Boardman, Jason Brazile, Lorenzo Bruzzone, Gustavo Camps-Valls, Jocelyn Chanussot, Mathieu Fauvel, Paolo Gamba, Anthony Gualtieri, Mattia Marconcini, James C. Tilton, and Giovanna Trianni. Recent advances in techniques for hyperspectral image processing. *Remote Sensing of Environment*, 113:S110 – S122, 2009. Imaging Spectroscopy Special Issue.

[3] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[4] Wei Hu, Yangyu Huang, Li Wei, Fan Zhang, and Hengchao Li. Deep convolutional neural networks for hyperspectral image classification. *Journal of Sensors*, 2015, 2015.

[5] M.E. Paoletti, J.M. Haut, J. Plaza, and A. Plaza. Deep learning classifiers for hyperspectral imaging: A review. *ISPRS Journal of Photogrammetry and Remote Sensing*, 158:279 – 317, 2019.

[6] Julian D Olden and Donald A Jackson. Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological modelling*, 154(1-2):135–150, 2002.

[7] Lichao Mou, Pedram Ghamisi, and Xiao Xiang Zhu. Deep recurrent neural networks for hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 55(7):3639–3655, 2017.

[8] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[9] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[10] Howard B Demuth, Mark H Beale, Orlando De Jess, and Martin T Hagan. *Neural network design*. Martin Hagan, 2014.

[11] Bernard Widrow and Michael A Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[12] George AF Seber and Alan J Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.

[13] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[14] Raul Rojas. The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer, 1996.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[16] Michael Collins and Nigel Duffy. Convolution kernels for natural language. In *Advances in neural information processing systems*, pages 625–632, 2002.

[17] Ido Freeman, Lutz Roese-Koerner, and Anton Kummert. Effnet: An efficient structure for convolutional neural networks, 2018.

[18] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer, 2010.

[19] Jason Brownlee. *Deep learning with python: Develop deep learning models on theano and tensorflow using keras*. Machine Learning Mastery, 2016.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[21] Lars Kotthoff Frank Hutter and editors Joaquin Vanschoren. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2019.

[22] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2017.

[23] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search, 2018.

[24] Yukun Zhu George Papandreou Barret Zoph Florian Schroff Hartwig Adam ”Liang-Chieh Chen, Maxwell Collins, H. Larochelle K. Grauman N. CesaBianchi Jon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In S. Bengio, H. Wallach, and editorsR. Garnett. Advances in neural information processing systems 31, 2018.

[25] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 497–504. ACM, 2017.

[26] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey, 2018.

[27] Y. Chen, K. Zhu, L. Zhu, X. He, P. Ghamisi, and J. A. Benediktsson. Automatic design of convolutional neural network for hyperspectral image classification.

IEEE Transactions on Geoscience and Remote Sensing, 57(9):7048–7066, Sep. 2019.

[28] J. M. Haut, M. E. Paoletti, J. Plaza, J. Li, and A. Plaza. Active learning with convolutional neural networks for hyperspectral image classification using a new bayesian approach. *IEEE Transactions on Geoscience and Remote Sensing*, 56(11):6440–6461, Nov 2018.

[29] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[30] John R Koza. Genetic programming. *Encyclopedia of Computer Sciences and Technology*, 1997.

[31] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[32] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[33] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[34] Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. Understanding and robustifying differentiable architecture search, 2019.

[35] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[36] Xiangxiang Chu, Tianbao Zhou, Bo Zhang, and Jixiang Li. Fair darts: Eliminating unfair advantages in differentiable architecture search. *arXiv preprint arXiv:1911.12126*, 2019.

[37] hobs (https://stats.stackexchange.com/users/15974/hobs). How to choose the number of hidden layers and nodes in a feedforward neural network? Cross Validated. URL:https://stats.stackexchange.com/q/136542 (version: 2019-05-27).