UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Análisis de técnicas de regularización de DNNs basadas en la optimización de la estructura de la red

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Análisis de técnicas de regularización de DNNs basadas en la optimización de la estructura de la red

Autor: Carmen Carrero Hurtado

Tutor: Pilar Bachiller Burgos

Resumen

Las redes neuronales profundas cuentan con un número muy elevado de parámetros, cuyo ajuste permite aprender grandes conjuntos de datos de manera extraordinaria. Sin embargo, esta sobreparametrización da lugar habitualmente a problemas de “sobreajuste”, que provocan que la red tienda a memorizar patrones de entrada/salida, en lugar de aprender relaciones entre ellos.

Para resolver este problema se han propuesto varias técnicas de regularización. Una de las técnicas más utilizada es Dropout y se basa en reducir el número de nodos de todas las capas, exceptuando la de salida, ignorando ciertas neuronas durante cada iteración del aprendizaje de manera aleatoria. Como alternativa a este método, existen técnicas que permiten determinar en cada iteración el conjunto de neuronas óptimas detectando y eliminando la correlación entre sus salidas.

El objetivo de este trabajo es analizar y comparar estas técnicas mediante su implementación y puesta en marcha sobre distintos conjuntos de datos de referencia.

Abstract

Deep neural networks own a very high number of parameters, whose adjustment enables an extraordinary learning of large data sets. However, this over-parameterization usually tends to create “overfitting” problems which cause the network to memorize input/output patterns, rather than learning the connections amongst them.

This study presents regulating methods in order to solve this problem. One of the most usual method is Dropout and it is based on reducing the number of nodes in all the layers, except the output one. Thus, certain neurons are randomly disregarded during each learning iteration. As an alternative, other methods may determine, in each iteration, the set of optimal neurons by detecting and removing the correlation between the outputs.

The aim of this bachelor thesis is analyzing and comparing these methods by their implementation on different referential data sets.

Índice general

1. Introducción **1**

1.1. Estructura de la memoria 4

2. Objetivos **7**

3. Estado del Arte **9**

3.1. Deep Learning 10

3.2. Estructura de la red neuronal artificial 11

3.3. Algoritmo Backpropagation 14

3.4. Técnicas de regularización 14

3.4.1. Weight decay 16

3.4.2. Data augmentation 16

3.4.3. Dropout 17

3.4.4. OLA 18

3.5. Redes neuronales convolucionales 20

3.5.1. Arquitectura de las redes neuronales convolucionales 20

4. Metodología **25**

4.1. Método OLA 25

4.2. Herramientas y conjuntos de datos utilizados 30

4.2.1. Pytorch 30

4.2.2. CIFAR10 30

4.2.3. MNIST 31

4.2.4. MNISTFashion 32

5. Implementación y desarrollo **33**

5.1. Desarrollo 33

5.1.1. Código implementado 33

5.1.2. Modelos de redes utilizados 36

5.2. Integración de la librería OLA 39

5.2.1. Introducción y requisitos 39

5.2.2. Ejemplo de uso 40

6. Resultados **43**

6.1. Reducción de nodos y generalización de la red 43

6.2. Influencia del número inicial de nodos 45

6.3. Optimización de varias capas 46

6.4. Influencia del conjunto de datos de optimización 47

6.5. Influencia del número de épocas entre los pasos de optimización . . . 48

7. Conclusiones y trabajos futuros **51**

7.1. Conclusiones 51

7.2. Trabajo futuro 52

Bibliografía **52**

Índice de tablas

5.1. Estructura de la red usada para el dataset CIFAR10 36
5.2. Estructura de la red usada para el dataset MNIST 37
5.3. Estructura del primer modelo de red utilizado para el dataset MNISTFashion 38
5.4. Estructura del segundo modelo de red utilizado para el dataset MNISTFashion 38
5.5. Estructura del tercer modelo de red utilizado para el dataset MNISTFashion 39

6.1. Resultados de los nodos finales de la capa después de aplicar el algoritmo OLA 44
6.2. Resultados del rendimiento de la red probando en diferentes conjuntos de datos 44
6.3. Número de nodos iniciales y finales en el conjunto de datos CIFAR10 después de aplicar OLA. 46
6.4. Resultados del rendimiento en el conjunto de datos CIFAR10 con un número de nodos distintos en la penúltima capa totalmente conectada de la red. 46
6.5. Resultados del número de nodos en el conjunto de datos CIFAR10 y MNISTFashion después de aplicar OLA a dos capas. 47
6.6. Resultados del rendimiento en el conjunto de datos CIFAR10 y MNISTFashion. 47

6.7. Resultados en MNIST cuando se modifica el tamaño del conjunto de datos de optimización. 48
6.8. Resultados en MNISTFashion cuando se modifica el número de épocas entre los pasos de optimización. 49

Índice de figuras

3.1. Perceptron simple 10
3.2. Perceptron Multicapa 11
3.3. Estructuras de redes neuronales 13
3.4. Funciones de activación 13
3.5. Problemas de generalización 15
3.6. Estructura Dropout 17
3.7. Red entrenada mediante OLA 19
3.8. Estructura final OLA 19
3.9. Arquitectura CNN 21
3.10. Ejemplo de convolución de una imagen 22
3.11. Tipos de pooling 23

4.1. Imágenes CIFAR10 31
4.2. Imágenes MNIST 31
4.3. Imágenes MNISTFashion 32

5.1. Implementación conjunto de datos a optimizar 40
5.2. Implementación del método getOutput 41
5.3. Implementación del paso de optimización durante el entrenamiento . . 41

Capítulo 1

Introducción

La inteligencia artificial [1] es una de las disciplinas más nuevas dentro del ámbito de la computación. A pesar de estar reflejado en numerosas publicaciones anteriores no fue hasta 1950 cuando se afianzó este campo, con el conocido artículo de Alan Turing donde proponía una prueba para determinar si una máquina era inteligente o no[2]. Sin embargo, hasta 1956 no se utilizó por primera vez el término "inteligencia artificial", mencionado en la conferencia de Dartmouth, congreso en el que se hicieron prometedoras previsiones que no se cumplieron, y que ocasionó el abandono por parte de los investigadores en este ámbito. Actualmente, la inteligencia artificial se encuentra en auge ya que se utiliza diariamente en disciplinas como la ingeniería, la medicina o la comunicación.

La inteligencia artificial cuenta con muchas ramas y una de las más destacadas es la que corresponde a las redes neuronales artificiales[3], basadas en el comportamiento del cerebro humano. Estas redes imitan ciertas características humanas, como la capacidad de recordar y asociar hechos basados en la experiencia. Aún así, existen diferencias entre una red neuronal artificial y la biológica debido a que existen claras distinciones entre el cerebro humano y un computador habitual en el que se implemente la red. El cerebro humano es complejo y cuenta con redes no programadas que aprenden a través de estímulos operando en paralelo, sin embargo, en la mayoría de los ordenadores se opera en serie con un número de procesadores bastante inferior al

del cerebro.

Una vez introducido el concepto de red neuronal artificial, pasamos a la definición de la estructura de una red. Esta está formada por un conjunto de nodos, denominados neuronas, que están conectadas entre sí formando capas. La organización de la red será una capa de entrada, una de salida y ninguna o un número determinado de capas ocultas; de esta manera habrá neuronas de entrada, de salida y ocultas.

Las neuronas de entrada se caracterizan por recibir las señales de fuentes externas a la red.

Las neuronas de la capa oculta son internas a la red. En ellas se realiza el procesamiento de información básica y se establece mediante sus conexiones la topología de la red neuronal.

Las neuronas de salida son la salida de la red, transmitiendo la información de la red hacia el exterior una vez finalizado el tratamiento de la información.

Esta estructura ha sufrido ligeros cambios y modificaciones a lo largo de los años, la primera vez que se conocía algo parecido fue gracias a Warren McCulloch y Walter Pitts, que en 1943 presentaron un modelo de neurona artificial simple mediante circuitos eléctricos[4]. Más tarde, Donald Hebb desarrolló una regla de cómo el aprendizaje neuronal ocurría. Esta ley, conocida como “Regla de Hebb” formó las bases de las técnicas de entrenamiento de redes neuronales[5].

Un año después del ya mencionado congreso de Dartmouth, en 1957, Rosenblatt propone el Perceptrón, la red neuronal más antigua[6]. Este modelo puede reconocer patrones que no se le hubiesen presentado anteriormente después de haberse aprendido otros similares durante el entrenamiento. En 1960 aparece la primera aplicación práctica de las redes neuronales, que se utilizó comercialmente durante algunos años. Este modelo, desarrollado por Widrow y Hoff[7] se denominó ADALINE y resolvía el problema de los ecos en las líneas telefónicas, puesto que los elimina mediante filtros adaptativos. A pesar del gran éxito obtenido en este campo, las investigaciones se vieron estancadas en 1969, cuando Minsky y Papert publicaron un libro[8] en

el que analizan el Perceptrón y concluyen que es incapaz de procesar el circuito de o-exclusiva, demostrando que tiene limitaciones lógicas importantes. De esta manera, muchos investigadores abandonaron este campo y se centraron en otros más prometedores en ese momento. No obstante, las investigaciones no pararon del todo y en 1974 Werbos describió por primera vez la idea de la propagación hacia atrás de errores (backpropagation)[9], algoritmo que resuelve eficazmente el problema de la o-exclusiva en el Perceptrón. Este algoritmo alcanzó su popularidad en 1986 ya que Rumelhart, en colaboración con otros autores aplicaron esta solución a una red neuronal multicapa, denominándose Perceptrón Multicapa[10].

Como consecuencia de estos trabajos e investigaciones, el estudio de las redes neuronales resurgió y ha experimentado un importante desarrollo en los últimos años.

Concretamente, este trabajo se centra en las redes neuronales profundas[11]. Estas redes nacieron de la necesidad de querer modelar funciones no lineales complejas y cuentan con múltiples capas ocultas complejas entre las capas de entrada y salida.

Son aplicadas para trabajar con grandes datos en problemas como reconocimiento de imágenes, reconocimiento de voz o modelado acústico. Sin embargo, es esta cantidad de datos de entrada y lo compleja y extensa que pueden ser las capas ocultas de la red lo que hace que esta pueda llegar a memorizar los conjuntos de entrada y salida, en vez de aprender vínculos entre ellos. Este problema se denomina *overfitting* y hace que se obtengan malos resultados cuando entrenamos una red. Por otro lado, se encuentra el *underfitting*, problema que se produce cuando se utilizan menos nodos que los requeridos por el modelo. En este caso, es incapaz de reconocer nuevos datos de entrada debido a que la relación entrada-salida dada por la red es demasiada escasa. Ambos problemas se engloban en lo que se conoce como problemas de generalización de la red neuronal.

Para intentar paliar estos problemas, se siguen distintas técnicas de regularización en la redes neuronales, como pueden ser Weight decay, Data augmentation o Dropout. En este trabajo se ha implementado una nueva técnica de regularización basada en la optimización de la estructura de la red. Se presentan en esta memoria los detalles

teóricos y de implementación de este método, así como un conjunto de resultados experimentales para su validación.

1.1. Estructura de la memoria

Esta sección proporciona un breve resumen de cada parte o capítulo del documento para proporcionar una descripción general y contextualizar la memoria.

- Capítulo 1: Introducción

  En este capítulo se muestra una breve entrada sobre el tema del proyecto. En este caso, pone en contexto al lector sobre la historia de las redes neuronales, haciendo incampié en las redes neuronales profundas.

- Capítulo 2: Objetivos

  Aquí se exponen los objetivos parciales que se han ido siguiendo en el trabajo para llegar al objetivo final, detallando brevemente cada uno de ellos.

- Capítulo 3: Estado del Arte

  Este capítulo desarrolla el fundamento teórico inicial seguido en el proyecto y establece las bases para el mismo. Se compone de varias secciones en las que se trata:

  - El concepto de DeepLearning.
  - La estructura de las redes neuronales profundas.
  - El algoritmo Backpropagation.
  - Las técnicas de regularización.
  - El estudio en más profundidad de las redes neuronales con capas convolucionales utilizadas en los entrenamientos del trabajo.

- Capítulo 4: Metodología

En este capítulo se desarrolla en más profundidad la teoría seguida para la implementación práctica del trabajo. Además, se comentan y detallan las herramientas utilizadas para ello.

- Capítulo 5: Implementación y desarrollo

  Capítulo en el que se desarrolla y comenta el código implementado para la puesta en marcha de la parte práctica del proyecto. También se expone cómo sería esa integración en el ordenador de un usuario cualquiera y se muestra un ejemplo del mismo.

- Capítulo 6: Resultados

  Aquí se muestran y discuten los resultados obtenidos de los experimentos realizados. Este apartado cuenta con varias secciones para mejorar la comprensión del lector en la interpretación de los resultados.

- Capítulo 7: Conclusiones y trabajos futuros

  En este capítulo se ofrece una valoración general del trabajo realizado y se proponen una serie de ampliaciones e investigaciones futuras para ampliar o perfeccionar el entorno expuesto en esta memoria.

Capítulo 2

Objetivos

El propósito de este trabajo es analizar algunas técnicas de regularización de DNNs basadas en la optimización de la estructura de la red. Se examinará en profundidad la técnica Dropout mediante su uso en varios modelos sobre distintos conjuntos de datos. Como parte original del trabajo se desarrollará un algoritmo de optimización y aprendizaje (OLA) que será analizado en profundidad y comparado con las técnicas de regularización más comunes. De esta manera se establecen una serie de objetivos parciales, que serán los siguientes:

- Estudio de las redes neuronales profundas y su funcionamiento: la finalidad es obtener nociones básicas sobre las redes neuronales profundas que formarán la base de este trabajo.

- Estudio de las principales técnicas de regularización: Estas técnicas son usadas para optimizar el entrenamiento de una red neuronal. En el capítulo se expondrán Weight decay, Data augmentation, Dropout y OLA.

- Utilización del algoritmo Dropout: Usaremos esta técnica sobre distintos problemas de aprendizaje para estudiar su comportamiento en las redes neuronales.

- Desarrollo e implementación del algoritmo OLA: La finalidad es desarrollar un nuevo algoritmo para encontrar la estructura óptima de la red durante la fase de entrenamiento. Luego se utilizará sobre distintos problemas de aprendizaje.

- Observación y estudio de los resultados dados en los experimentos realizados:

Estudiaremos los resultados obtenidos en las redes neuronales, centrandonos en el uso de los métodos mencionados anteriormente (Dropout y OLA) para su posterior comparación y análisis. El desarrollo y pruebas han sido implementadas con Pytorch.

Capítulo 3

Estado del Arte

Este apartado se dividirá en varias secciones teniendo en cuenta el proceso de investigación seguido para el proyecto. Las secciones serán las siguientes:

- La primera tratará sobre una explicación básica del DeepLearning.

- La segunda tratará sobre la estructura de la red neuronal profunda.

- La tercera será una definición sencilla del algoritmo Backpropagation utilizado por las redes empleadas en el proyecto.

- La cuarta consistirá en las técnicas de regularización empleadas en redes neuronales.

- Por último, se pasará a realizar un análisis más en profundidad de las redes convolucionales, el tipo de red neuronal usado en los ejemplos del trabajo.

De esta forma, se crea una base para apoyar y fundamentar el desarrollo del trabajo que se explicará más adelante.

3.1. Deep Learning

A menudo son confundidos los términos machine learning y deep learning[12]. El primero es una rama de la inteligencia artificial que utiliza algoritmos matemáticos que permiten a las máquinas aprender. Utilizan estos algoritmos para analizar datos, aprender de ellos y luego poder hacer predicciones o conclusiones de nuevos datos.

El deep learning nace como una rama del machine learning, teniendo como idea el aprendizaje desde el ejemplo. Mientras que el machine learning utiliza algoritmos de regresión o árboles de decisión, el deep learning se centra en las redes neuronales profundas (deep neural networks). De forma simple, el término ”deep”hace referencia a la presencia de muchas capas en las redes neuronales artificiales, pero este significado ha cambiado varias veces a lo largo del tiempo. Hace años, diez capas eran suficientes para considerar una red como profunda, hoy en día se considera una red profunda cuando cuenta con cientos de capas[13].

Los primeros predecesores del deep learning moderno eran simples modelos lineales motivados por una perspectiva neurocientífica. Estos modelos eran diseñados de tal manera que se tomaba un conjunto de n valores de entrada $x_1,...,x_n$ y se asociaban con una salida y. Estos modelos aprenden un conjunto de pesos $w_1,...,w_n$ y calculan su salida como $f(x,w)= x_1w_1 + ... + x_nw_n$ pero contaban con numerosas limitaciones. La más conocida es el problema de la o-exclusiva, la función XOR, donde $f([0,1],w)= 1$ y $f([1,0],w)= 1$ pero $f([1,1],w)= 0$ y $f([0,0],w)= 0$[14]. Un ejemplo de dicho modelo es el Perceptron simple, basado en una neurona artificial llamada LTU (linear threshold unit).

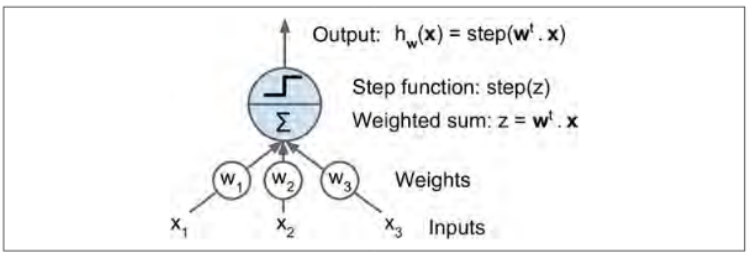


Figura 3.1: Neurona artificial en la que está basado el Perceptron (LTU).
Fuente: [15]

Las limitaciones comentadas anteriormente pueden ser solventadas mediante el Perceptron multicapa. La estructura está compuesta de una capa de entrada, una o más capas de LTU, llamadas capas ocultas y una capa final, llamada capa de salida. Cada capa, excepto la capa de salida, está completamente conectada a la siguiente capa[15].

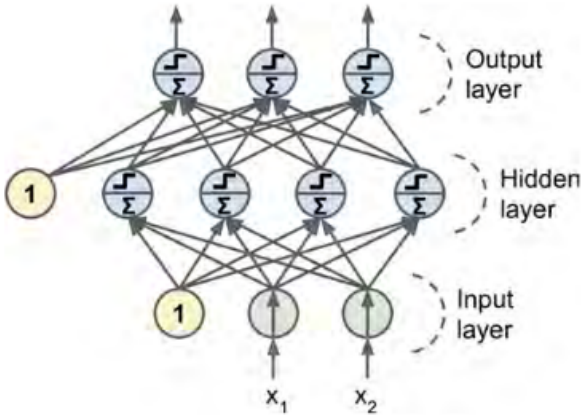


Figura 3.2: Estructura del perceptron multicapa.
Fuente: [15]

3.2. Estructura de la red neuronal artificial

Las redes neuronales artificiales están organizadas en capas. Estas capas pueden ser de muchos tipos y se pueden combinar varias en la formación de la estructura de la red. Algunos de estos tipos son:

- Capas totalmente conectadas (full-connected)
- Capas convolucionales
- Capas de reducción (pooling)
- Capas recurrentes
- Capas de normalización

Las diferentes capas realizan diferentes transformaciones en sus entradas y algunas capas son más adecuadas para unas tareas que otras. Por ejemplo, las capas

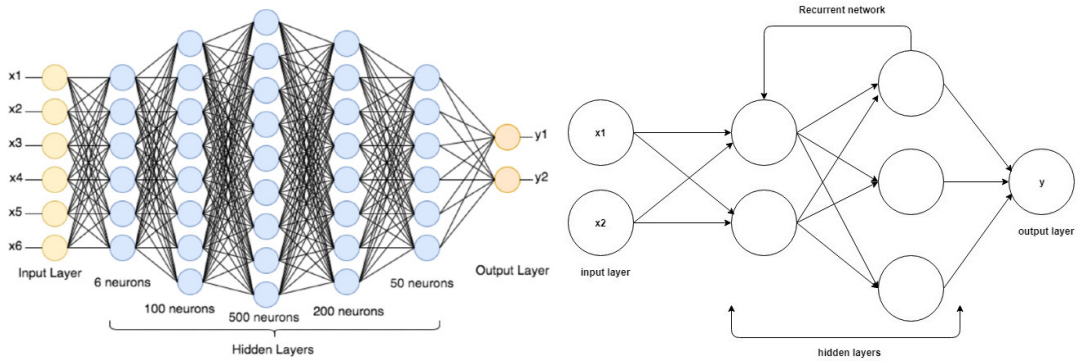
convolucionales se suelen usar en modelos que están trabajando con datos de imagen y las recurrentes en modelos que trabajan con datos de serie de tiempo.

Cuando una red neuronal artificial (ANN) tiene dos o más capas ocultas, se denomina red neuronal profunda (DNN). En la actualidad, es normal que estas redes partan de cientas de capas ocultas. Normalmente, las DNN suelen estar formadas por capas de retroalimentación(full-connected) en las que los datos fluyen de la capa de entrada a la capa de salida sin retroceder.

Las redes neuronales recurrentes (RNN), en las que los datos pueden fluir en cualquier dirección se utilizan para aplicaciones como el modelo de lenguaje [16].

Las redes neuronales convolucionales (CNN) son muy comunes en la visión artificial.

En la figura 3.3 se pueden observar las diferentes capas mencionadas anteriormente. El primer modelo está formado por capas full-connected(DNN), el segundo está formado por capas recurrentes(RNN) y el tercero está formado por capas convolucionales, de reducción y full-connected(CNN).



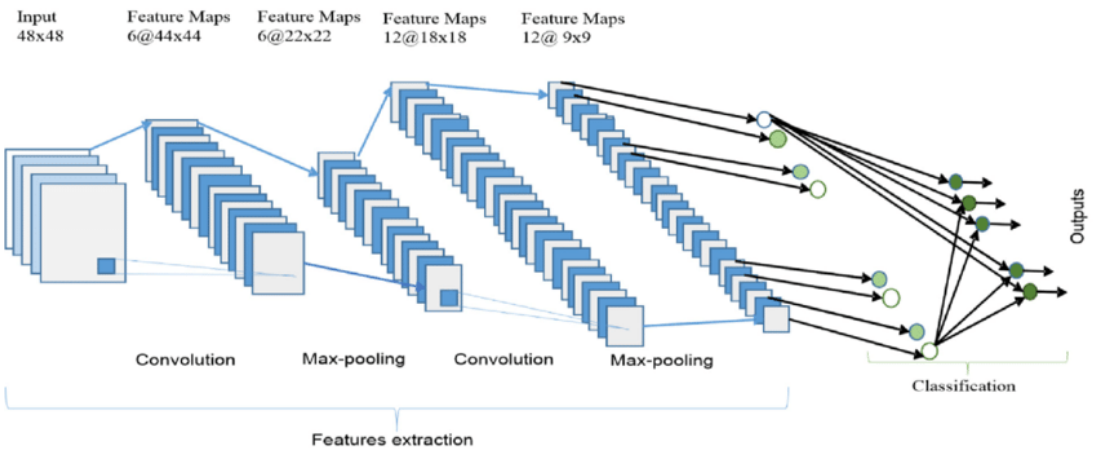


Figura 3.3: Estructura de las redes neuronales DNN, RNN y CNN, respectivamente.
Fuente: [17][18][19]

Un modelo se define por la matriz de pesos, un vector bias, b, el número de capas ocultas y el número de neuronas por capa, además de por las funciones de activación aplicadas en cada capa. El funcionamiento de la red es que dado un vector de entrada x, cada capa oculta aplica una función de activación no lineal g transformando la salida de la capa anterior. Algunas de las funciones de activación son *tanh*, *ReLU* o *Sigmoide*. En la figura 3.4 podemos ver su representación gráfica.

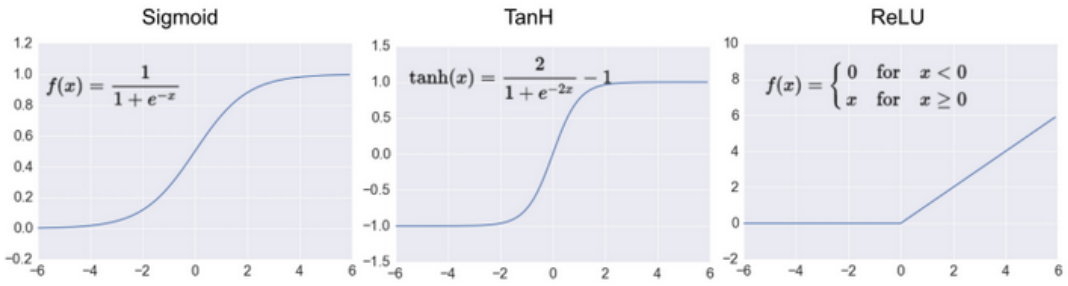


Figura 3.4: Representación gráfica de las funciones de activación sigmoide, tanh y ReLU.
Fuente: [20]

3.3. Algoritmo Backpropagation

El algoritmo de backpropagation[21] se ha convertido en uno de los métodos más populares de entrenamiento de redes neuronales. La razón de su popularidad es la simplicidad y el poder de dicho algoritmo puesto que permite el entrenamiento de funciones no lineales. La idea de este método es simple, tanto, que se puede implementar en unas pocas líneas de código.

Consiste en enviar un patrón de entrada como estímulo para la primera capa de las neuronas de la red, propagándose a través de esta hasta obtener una salida, que se compara con el resultado que se quiere obtener y se calcula un valor de error para cada neurona de salida. Estos errores se transmiten hacia atrás, repitiendo el proceso capa por capa hasta que todas las neuronas hayan recibido el error. Teniendo en cuenta este error se reajustan los pesos de conexión de cada neurona hasta que se obtenga la máxima similitud entre la salida de la red y la que se desea.

Esta diferencia se mide con una función de coste, entre las que se encuentran:

- Error cuadrático medio: Utilizada para problemas de regresión, mide la diferencia que hay entre los valores de salida de la red y los valores correctos.

- Entropía cruzada: Se utiliza en los problemas de clasificación y se realiza de forma iterativa haciendo una propagación del error desde la capa de salida hasta la de entrada, lo que hace que se consiga minimizar el error en todas las capas.

3.4. Técnicas de regularización

El rendimiento de una red neuronal se mide teniendo en cuenta el conjunto de validación, compuesto por datos que no existen en el conjunto de entrenamiento. Una red bien entrenada debe predecir relaciones correctas de entrada-salida para los datos del conjunto de validación; dichos datos no han sido mostrados a la red con anterioridad por lo que esta tiene que generalizar las asociaciones aprendidas.

De esta manera, entrenar una red neuronal profunda que pueda generalizar correctamente nuevos datos es un gran reto. Si una red cuenta con poca capacidad

(pocos nodos ocultos) no puede aprender las asociaciones necesarias mientras que un modelo con demasiada capacidad (excesivos nodos ocultos) puede aprenderlas bastante bien y sobreajustar el conjunto de datos de entrenamiento. El primer caso es conocido como underfitting y el segundo como overfitting y ambos dan como resultado una red que no generaliza bien.

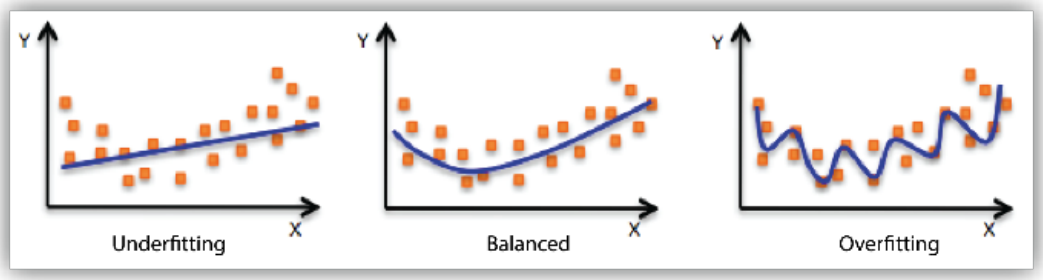


Figura 3.5: Ejemplo gráfico de los problemas de generalización, así como una red que generaliza bien.
Fuente: [22]

El problema del underfitting puede solucionarse de manera sencilla incrementando la capacidad de la red; esto se realiza cambiando la estructura del modelo, por ejemplo añadiendo más capas o más nodos a las capas.

Debido a la simplicidad de la solución del problema anterior, el overfitting es mucho más común de encontrar en las redes. Un modelo sobreparametrizado se muestra cuando la red tiende a memorizar los conjuntos de entrada-salida y no aprende relaciones entre ellos, es decir, si le mostramos un dato que no ha visto antes no tiene la capacidad de generalización y no será capaz de asociarle una salida válida. Este problema es más complejo que el anterior y requiere el uso de técnicas especializadas.

Dichas técnicas son conocidas como métodos de regularización de la red y algunas se utilizan para reducir el número de parámetros de la red. A continuación se explican varias de ellas.

3.4.1. Weight decay

Esta técnica fue propuesta por Hinton en 1989 y la idea es eliminar gradualmente aquellas conexiones que no son modificadas frecuentemente durante el aprendizaje[23]. Consiste en introducir un término adicional de penalización en la ecuación de actualización de los pesos, haciendo que los pesos tiendan a cero a menos que sean reforzados. Esta distinción puede dividir las ponderaciones de la red en: ponderaciones que tienen poco o ningún impacto en el modelo, denominadas ponderaciones redundantes y ponderaciones que afectan al modelo[24].
Por lo general, se usa la siguiente penalización:

$$\lambda \sum_i w_i^2 \tag{3.1}$$

Donde w_i es el peso iésimo en la red y λ es el coeficiente que da más o menos peso a la penalización. Gerenalmente, este parámetro es muy pequeño.

En weight decay no solo se penalizan los pesos constantes, haciendo que el aprendizaje sea lento, convirtiéndose en el mayor hándicap del método. La ventaja es que el aprendizaje y la poda se hacen de manera simultánea.

3.4.2. Data augmentation

Es el método más sencillo y más utilizado en visión artificial para reducir el sobreajuste, puesto que da muy buenos resultados cuando se trabaja con imágenes como datos. Consiste en aplicar transformaciones a las imágenes del conjunto de datos, manteniendo sus etiquetas. De esta manera, se generan datos adicionales sin introducir un coste adicional, lo que permite a la red desenvolverse mejor.

La red neuronal procesará la misma imagen de entrada tantas veces como épocas haya en el entrenamiento, por lo que si no aplicamos estas transformaciones aleatorias cada vez que volvamos a introducir la imagen a la red esta acabará memorizando si entrenamos demasiado.

Algunas transformaciones que se pueden aplicar son alterar la intensidad de los

canales RGB, rotar la imagen un número determinado de grados o introducir ruido[25].

3.4.3. Dropout

La idea de este método es desconectar un porcentaje de las unidades (tanto ocultas como visibles) de la red neuronal. Al desconectar una neurona, se eliminan de la red junto con todas sus conexiones de entrada y salida. En la figura 3.6 se muestra la estructura de la red antes y después de realizar la técnica Dropout en una iteración, pues cada iteración durante el entrenamiento se hará con una estructura distinta de la modelada. La red del ejemplo cuenta con dos capas ocultas.

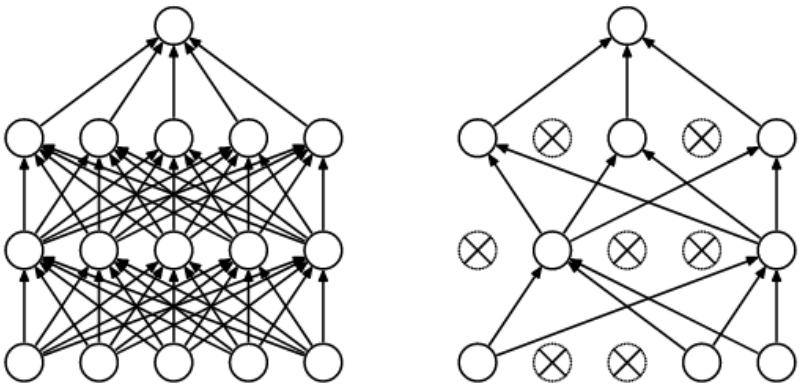


Figura 3.6: Estructura de la red antes y después de realizar la técnica Dropout en una iteración.
Fuente: [26]

La elección de qué nodos eliminar es aleatoria. Generalmente, cada nodo se retiene con una probabilidad fija p, donde p puede ser elegido en función del problema particular mediante un conjunto de validación o puede establecerse en 0.5 para los nodos de la capa oculta y un valor cercano a 1, como 0.8, para los nodos de la capa de entrada, valores que parecen óptimos para la mayoría de problemas[26].

Como se ve en la imagen 3.6 se genera una nueva estructura al aplicar dropout, formada por todas las neuronas que han ”sobrevivido.ªl método. De esta manera, una red neuronal con n neuronas puede verse como una colección de 2^n posibles estructuras de red después de aplicar dropout. Todas estas redes comparten pesos, así que el número total de parámetros sigue siendo $O(n^2)$ o menos. Por cada iteración en el

entrenamiento que se use dropout se genera una nueva estructura de la red neuronal, por lo que entrenar una red neuronal con dropout puede verse como entrenar una colección de 2^n redes reducidas donde cada red se entrena pocas veces, si es que se entrena.

Dropout es más efectivo que otros métodos, como weight decay. Se puede combinar con otras formas de regularización para producir una mejora adicional.

Esta técnica ha sido usada en los experimentos prácticos realizados sobre este proyecto.

3.4.4. OLA

OLA(Optimizing and Learning Algorithm)[27] será otra técnica usada en el desarrollo de la parte práctica y se tratará en capítulos posteriores en más profundidad debido a que es el método original de la memoria. Sus resultados se compararán con Dropout para poder determinar unas conclusiones.

OLA se basa en la detección de nodos redundantes en las capas ocultas (de tipo *full connected*), tomando dichos nodos como aquellos cuyas salidas pueden obtenerse a partir de combinaciones lineales de las salidas de otros nodos. La idea es la búsqueda del mejor conjunto de nodos de la red durante su fase de entrenamiento. Para alcanzar esto, en cada iteración del proceso de aprendizaje se selecciona el conjunto de nodos ocultos que contiene la información más destacada y solo se actualizan los pesos de los enlaces conectados con dichos nodos, podando las salidas de los nodos que se consideran redundantes.

El algoritmo funciona de la siguiente manera:
En la fase de entrenamiento, los nodos de la capa oculta se separan en dos grupos. El conjunto O, que consiste en los nodos cuyos vectores de salida son linealmente independientes y el conjunto R, que son los nodos ocultos cuyos vectores de salida son combinaciones lineales de los vectores asociados con los nodos de O, es decir, los nodos redundantes.
Lo primero que hace el algoritmo es calcular los conjuntos definidos anteriormente, O y R y actualizar los pesos de la red utilizando el algoritmo Backpropagation. Solo se

actualizarán los pesos conectados con los nodos de O, ya que los conectados con los nodos de R son redundantes lo que implica que se pueden anular sin que se modifique el comportamiento de la red.

Seguidamente se muestran dos figuras. La primera es una red con dos capas internas entrenada mediante el algoritmo propuesto y la segunda es la arquitectura final de la red una vez que el proceso de aprendizaje ha acabado. Como se observa, las únicas conexiones que han quedado son las que unen nodos óptimos, pues todos los nodos redundantes de la última iteración así como sus conexiones de entrada y salida han sido eliminados.

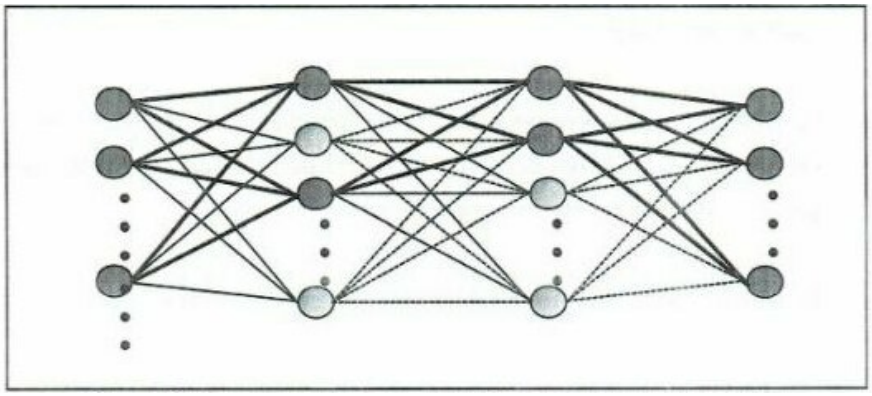


Figura 3.7: Red entrenada con el algoritmo OLA.
Fuente: [27]

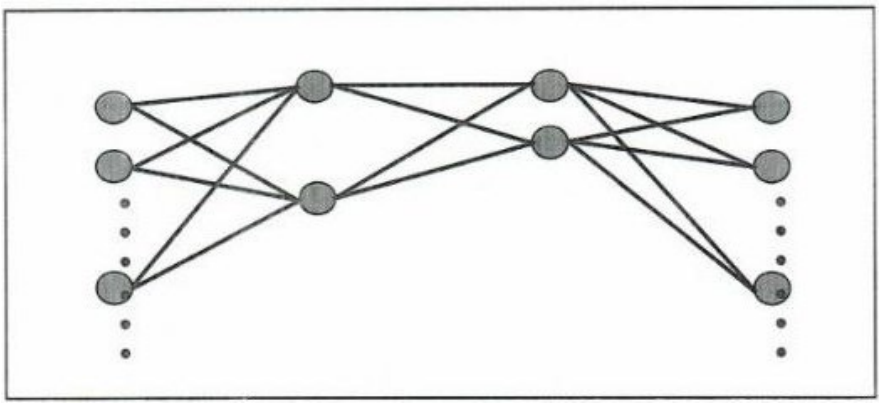


Figura 3.8: Arquitectura óptima de una red después de ser entrenada con el algoritmo OLA.
Fuente: [27]

El método de selección utilizado para encontrar los nodos óptimos de la red en

cada iteración del proceso de aprendizaje será expuesto en capítulos posteriores.

3.5. Redes neuronales convolucionales

En la parte práctica del proyecto se han usado las dos últimas técnicas del apartado anterior (Dropout y OLA) y la estructura de red elegida para ello ha sido la conocida como redes neuronales convolucionales"[25]. Ambas técnicas se han aplicado optimizando las capas full-connected de la red a optimizar. Se han incluido las capas convolucionales porque el conjunto de datos de entrada son imágenes y trabajan muy bien en visión artificial debido a que son capaces de detectar características simples como bordes, líneas, etc. Las redes convolucionales se van a desarrollar en este apartado para conocerlas mejor y entender cómo funcionan.

Estas redes son similares a las redes neuronales ordinarias y están diseñadas para procesar datos bidimensionales, como por ejemplo imágenes y vídeos. Son muy utilizadas en el campo de la visión artificial ya que son muy efectivas en tareas como la clasificación y segmentación de imágenes. Su arquitectura está inspirada en la organización de la corteza visual animal.

Suponen explícitamente que las entradas son imágenes, lo que permite codificar ciertas propiedades en la arquitectura, haciendo que se gane en eficiencia y se reduzca la cantidad de parámetros en la red. Sin embargo, son computacionalmente pesadas lo que hace que su entrenamiento sea lento y pesado. Con el surgimiento de la programación paralela y aprovechando el poder de procesamiento de las GPUs, los tiempos de entrenamiento de estas redes han disminuido. Actualmente, las CNNs se aplican en el reconocimiento de la escritura a mano, detección de rostros, reconocimiento de voz o clasificación de imágenes, entre otros campos[28].

3.5.1. Arquitectura de las redes neuronales convolucionales

Aunque coloquialmente se llamen redes convolucionales no son este tipo de capas las única que tiene. La estructura cuenta con capas convolucionales, de

reducción y totalmente conectadas, es decir, se trata de una arquitectura multicapa y está estructurada como una serie de etapas. La primera etapa está compuesta por capas convolucionales y de reducción (pooling) y la segunda etapa por las capas clasificadoras de la red. Estas capas están conectadas alternativamente y forman la parte media de la red, como se puede ver en la figura 3.9.

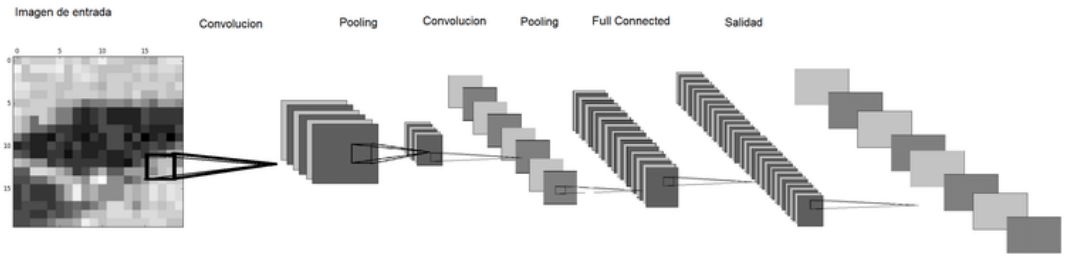


Figura 3.9: Arquitectura de una red neuronal convolucional.
Fuente: [19]

Capas convolucionales:

La finalidad de la capa convolucional es extraer patrones encontrados dentro de las regiones locales de las imágenes de entrada.

Esto se lleva a cabo mediante filtros convolucionales, también llamados núcleos de convolución, que consisten en realizar operaciones de productos y sumas entre la capa de partida y los filtros generando un mapa de características de la imagen original. Un ejemplo de esto se encuentra en la imagen 3.10, donde se utiliza un filtro para pasar los bordes y descartar otra información. Luego se aplica una función no lineal (normalmente *ReLU*) a cada característica del mapa.

Capas de pooling:

El resultado de esta función se pasa a la siguiente capa, llamadas capas de reducción o pooling. Estas capas hacen una simplificación de la información recogida por la capa convolucional y crean una versión condensada de la información contenida en

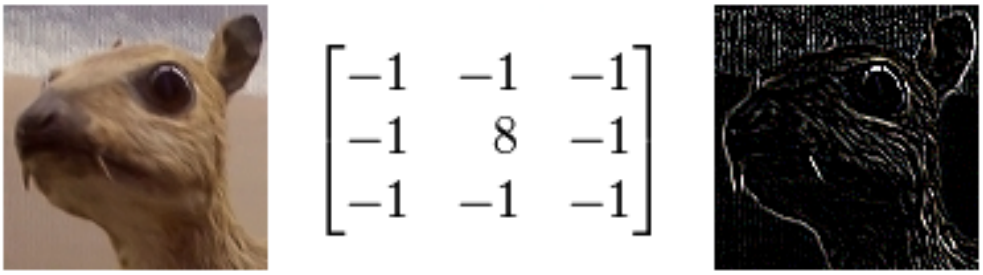


Figura 3.10: Convolución de una imagen con un filtro de convolución de detector de bordes.
Fuente: [29]

dichas capas. Esta reducción de tamaño lleva también a la pérdida de información. Sin embargo, una pérdida de este tipo también puede ser beneficiosa para la red debido a que la disminución en el tamaño conduce a una menor sobrecarga de cálculo para las próximas capas y además reduce el sobreajuste.

Las operaciones más comunes para realizar la reducción son la media y max-pooling[30] cuya representación gráfica se encuentra en la imagen 3.11. La primera toma la media aritmética de los elementos en cada región de pooling y la segunda divide a la imagen de entrada en un conjunto de rectángulos y (teniendo en cuenta la subregión) se va quedando con el máximo valor.

Capas full-connected:

Por último, detrás de las capas mencionadas anteriormente las redes utilizan capas completamente conectadas en las que cada píxel se considera como una neurona separada. La última capa clasificadora tendrá tantas neuronas como el número de clases que debe predecir.

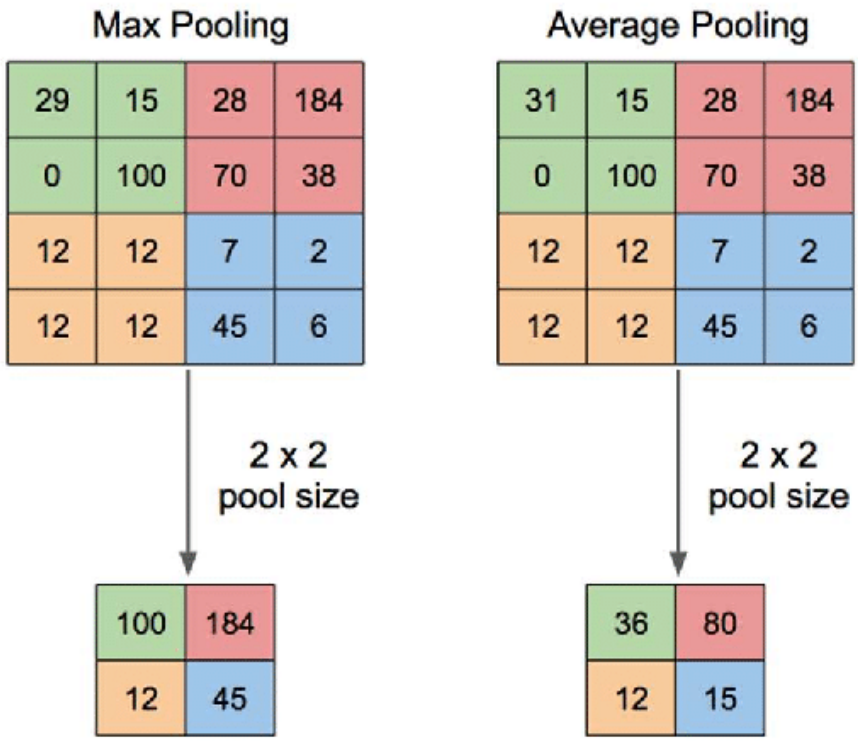


Figura 3.11: Ejemplo de las operaciones más comunes de reducción (la media y max pooling) sobre una matriz de 4x4.
Fuente: [31]

Capítulo 4

Metodología

Este capítulo se divide en dos secciones: *Método OLA* y *Herramientas y conjuntos de datos utilizados*.

En la primera sección se expone el método OLA descrito en el apartado 3.1.3 de esta memoria, el cual ha sido implementado y puesto en marcha sobre distintos conjuntos de datos.

En la segunda sección se desarrollan las herramientas y conjuntos de datos utilizados para su implementación.

4.1. Método OLA

En la sección 3.1.3 ya se da una idea general de este algoritmo, que se basa en buscar el conjunto de nodos óptimos de la red durante su fase de entrenamiento[27]. El método de selección utilizado para encontrar dichos nodos óptimos se basa en una seride de propiedades de las matrices de Householder [32][27].

- Propiedad 1: Las transformaciones de Householder pueden usarse para hacer ceros determinadas componentes de un vector[33][27].

- Propiedad 2: Dados m vectores $\in \Re^n$, $[h_1,h_2,...,h_n]$, las transformaciones de Householder permiten determinar cuáles de ellos son linealmente independientes.

[32][34]

Si suponemos que H $\in \Re^n xn$ es un producto de transformaciones de Householder que hace cero al menos las últimas n-L componentes de cada vector del conjunto S=[h_1,h_2,...,h_n], siendo L el número de vectores linealmente independientes de S. Para resolver la dependencia lineal entre un nuevo vector $h_i + 1$ y los vectores de S es necesario obtener un vector y=$Hh_i + 1$. *Si* $h_i + 1$ puede expresarse mediante una combinación lineal de [h_1,h_2,...,h_i] entonces las últimas n-L componentes de y serán iguales a cero:

$$Hh_{i+1} = \sum_{j=1}^{i} c_j Hh_j \tag{4.1}$$

$$y = [m_1, ..., m_L, 0, ..., 0]^T \tag{4.2}$$

Si no se cumple la ecuación 4.2, debemos actualizar la matriz H para poder determinar dependencias lineales entre nuevos vectores y los vetores de S'=[h_1,h_2,...,h_i]. Ahora, la matriz H tiene que hacer ceros al menos las últimas n-L-1 componentes de los vectores de S', por tanto, es necesario calcular una nueva matriz de Householder H' que haga ceros las últimas n-L-1 componentes de y y actualizar la matriz mediante el producto H'H[27].

- Propiedad 3: Si un vector h $\in \Re^n$ es una combinación lineal de m vectores linealmente independientes [h_1,h_2,...,h_m] haciendo uso de las transformaciones de Householder podemos determinar los coeficientes de dicha combinación resolviendo un sistema triangular de ecuaciones [32][34].
Si [h_1,h_2,...,h_m] son vectores linealmente independientes, podemos encontrar una matriz H $\in \Re^n xn$, a partir de transformaciones de Householder, tal que el vector resultante de la operación Hh_i tenga sus últimas n-i componentes iguales a cero. Si, además, podemos expresar el vector h como una combinación lineal de la forma:

$$h = \sum_{i=1}^{M} h_i c_i \tag{4.3}$$

es posible determinar los coeficientes c_i, resolviendo el siguiente sistema de ecuaciones[27]:

$$Hh = \sum_{i=1}^{M} c_i H h_i \tag{4.4}$$

$$[v_1, ..., v_m, 0, ..., 0]^T = \sum_{i=1}^{M} c_i [w_i 1, ..., w_i i, 0, ..., 0]^T \tag{4.5}$$

$$c_i = \frac{v_i - \sum_{j=i+1}^{M} c_j w_{ji}}{w_{ii}} \tag{4.6}$$

Estas propiedades serán utilizadas en el algoritmo OLA explicado a continuación. Un nodo es redundante y puede ser eliminado cuando las entradas netas para todos los nodos de la capa siguiente sean las mismas antes y después de eliminar el nodo, es decir, que se mantenga el comportamiento original de la red después de eliminar dicho nodo. Esto traducido a fórmula sería comprobar que existe una combinación de pesos que mantiene el comportamiento original de la red, por lo que:

$$\sum_{j=1}^{M} h_j w_l j = \sum_{\substack{j=1 \\ j \neq i}}^{M} h_j w'_{lj} \tag{4.7}$$

siendo h_j el vector formado por las salidas del nodo j para todos los patrones de aprendizaje y M el número de nodos de la capa a la que pertenece i. Desarrollando la ecuación 4.7 se obtiene:

$$h_i = \sum_{\substack{j=1 \\ j \neq i}}^{M} h_j c_j \tag{4.8}$$

siendo c_j los coeficientes de la combinación lineal, cuya fórmula sería:

$$c_j = \frac{w'_{lj} - w_{lj}}{w_{li}} \tag{4.9}$$

Si la ecuación 4.8 se cumple implica que el nodo i es redundante, ya que su vector de salidas h_i es una combinación lineal de los vectores de salida de los nodos restantes.

El método OLA utiliza estas propiedades para determinar cuál es la estructura óptima de la red. Considerando una red de tamaño NxMxO y P patrones de aprendizaje, la selección de nodos se lleva a cabo aplicando el algoritmo que se expone a continuación:

1. Inicializar L a 0 (número óptimo de nodos) y H como la matriz identidad de tamaño PxP.

2. Para todos los nodos de la capa oculta:

   2.1 Obtener el vector de salidas h_i del nodo actual.

   2.2 Multiplicar dicho vector por H, obteniendo como resultado el vector y.

   2.3 Si el vector h_i es una combinación lineal de los vectores de salidas asociados con los nodos examinados en iteraciones anteriores, entonces y tendrá sus últimas P-L componentes a cero y, por lo tanto, se cumple la siguiente igualdad:

   $$||h_i||_2 = \sqrt{\sum_{m=1}^{L} y_m^2} \tag{4.10}$$

   En este caso, podemos decir que el nodo i es redundante. Ahora, es posible realizar un ajuste de los pesos de salida para las L neuronas óptimas determinadas hasta el momento. Los coeficientes c_j de la combinación lineal pueden determinarse aplicando la propiedad 3 de las transformaciones de Householder. De esta forma, podemos ajustar los pesos de los nodos óptimos siguiendo la ecuación:

   $$w_{lj} = w_{lj} + c_j w_{li} \quad 1 \leq j \leq Ly1 \leq l \leq O \tag{4.11}$$

2.4 Si por el contrario, la ecuación 4.4 no se verifica, se trataría de un nodo óptimo y es necesario actualizar la matriz H para poder determinar dependencias lineales entre los vectores de salida examinados y los vectores de salida asociados con los restantes nodos. Habría que incrementar el número óptimo de nodos (L+1) y actualizar la matriz H, multiplicando esta matriz por una nueva matriz de Householder $H^{'}$ que haga ceros las últimas P-L-1 componentes de y [27].

Partiendo de una red ya entrenada, este algoritmo permite determinar qué nodos pueden eliminarse en cada capa oculta. Asimismo, el paso 2.3 proporciona un ajuste de los pesos de los restantes nodos para que la red produzca los mismos resultados por cada dato de entrada utilizado en el entrenamiento.

El método OLA utiliza este algoritmo de selección para encontrar los nodos óptimos durante el aprendizaje. En concreto, tras presentar a la red todos los datos de entrenamiento en una iteración determinada, el método anterior permite clasificar los nodos de una capa en 2 grupos: nodos óptimos y nodos redundantes. Durante el aprendizaje, los nodos óptimos son los únicos que deben actualizar sus pesos. Por el contrario, las conexiones de los nodos redundantes no deben considerarse, puesto que las salidas de dichos nodos pueden obtenerse a partir de las salidas de los nodos optimos. Para lograr esto, en OLA se “podan” las salidas de los nodos redundantes poniendo a 0 los pesos de sus conexiones de salida. Esto permite que, en la siguiente iteración de aprendizaje, la red considere únicamente los nodos óptimos y ajuste los pesos de sus conexiones teniendo en cuenta exclusivamente las salidas de dichos nodos. Así, en la aplicación del método de selección durante el aprendizaje no es necesario aplicar el ajuste de pesos de la ecuación 4.11. Es el propio algoritmo de aprendizaje el que se encargará de dicho ajuste.

La aplicación de este algoritmo en redes produndas presenta limitaciones por el tamaño del conjunto de los datos de entrenamiento. Las dimensiones de la matriz de Householder deben coincidir con el número de datos utilizado para el aprendizaje lo que, en la práctica, supone una elevada carga computacional. Para solucionar esta

cuestión, en la implementación desarrollada, el método se aplica a un subconjunto de los datos de entrenamiento elegidos de manera aleatoria. Este nuevo conjunto de datos se denomina conjunto de datos de optimización y, junto con otros parámetros, en la sección 6 se analiza la influencia del tamaño de este conjunto en la optimización de la red.

4.2. Herramientas y conjuntos de datos utilizados

El desarrollo experimental del trabajo ha sido puesto en marcha mediante Pytorch y los conjuntos de datos de entrada seleccionados para realizar las pruebas han sido CIFAR10, MNIST y MNISTFashion.

4.2.1. Pytorch

Es una librería de machine learning donde el elemento fundamental son los tensores (equiparables a vectores de una o varias dimensiones)[35]. Está desarrollado por el laboratorio de inteligencia artificial de Facebook y es un software gratuito y de código abierto. Su principal ventaja es que permite la ejecución en GPU de forma nativa así como su facilidad de uso.
Se ha utilizado esta librería en Python, ya que es el lenguaje de programación elegido para el desarrollo del código. Se ha escogido este lenguaje debido a su facilidad de uso y a su gran integración con la librería Pytorch.

4.2.2. CIFAR10

Este conjunto de datos consta de 60000 imágenes en color, de tamaño 32x32 separadas en 10 clases, es decir habrá 6000 imágenes por clase. Estas clases son: avión, coche, pájaro, gato, ciervo, perro, rana, caballo, barco y camión[36].
De las 60000 imágenes de todo el conjunto, hay 50000 destinadas a entrenamiento(5000 imágenes de cada clase) y 10000 imágenes de prueba(1000 imágenes de cada clase). Estas imágenes se dividen por lotes de 10000, lo que se

quedaría en cinco lotes de entrenamiento y uno de pruebas. Estos lotes contienen las imágenes en orden aleatorio.

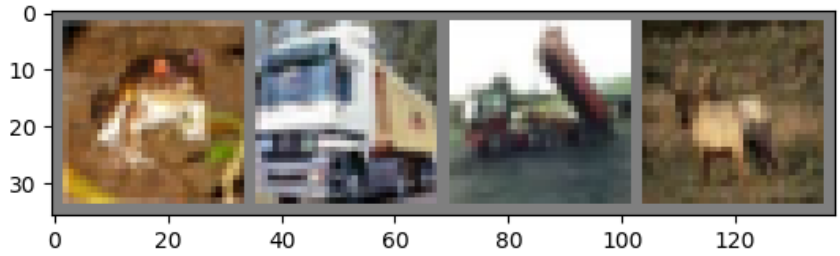


Figura 4.1: Imágenes del conjunto de entrenamiento de CIFAR10 correspondientes a las categorías de rana, camión, camión y ciervo, respectivamente.
Fuente: Propia

4.2.3. MNIST

MNIST[37] es un subconjunto de dígitos escritos a mano basado en otro más grande, llamado NIST. Consta de 70000 imágenes, 60000 para entrenamiento y 10000 para pruebas. Las imágenes de la base de datos NIST están en blanco y negro y tienen un tamaño de 20x20 píxeles pero el subconjunto de imágenes de MNIST se normalizó y se suavizó, lo que provocó que pasasen a ser de 28x28 píxeles y se introdujesen niveles de escala de grises. Las imágenes se dividen en 10 categorías, correspondientes a los números del 0 al 9, escritos a mano alzada.

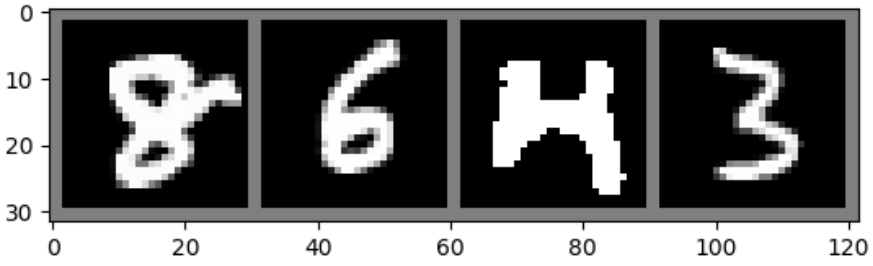


Figura 4.2: Imágenes del conjunto de entrenamiento de MNIST correspondientes a los número 8, 6, 4 y 3, respectivamente.
Fuente: Propia

4.2.4. MNISTFashion

Las imágenes de este conjunto de datos corresponden a artículos de Zalando. MNISTFashion consta de 60000 imágenes de entrenamiento y 10000 para realizar pruebas. Cada imagen tiene una etiqueta que puede ser de 10 clases distintas, siendo estas: camiseta, pantalón, jersey, vestido, abrigo, sandalia, camisa, zapatilla, bolso y bota[38].

Las imágenes son de tamaño 28x28 y están en escala de grises, siguiendo el mismo patrón que el conjunto MNIST. Esto es porque se pretende que MNISTFashion sea el reemplazo de MNIST ya que este último dataset es muy simple y se ha utilizado en exceso haciendo que en este conjunto de datos las redes neuronales convolucionales puedan alcanzar fácilmente el 99.7 % de rendimiento, mientras que los algoritmos más tradicionales ya tienen aproximadamente el 97 % de efectividad.

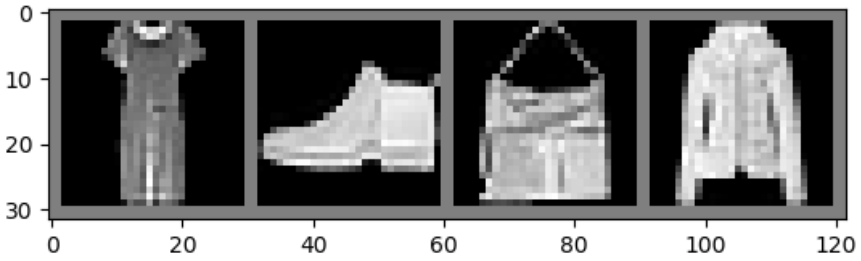


Figura 4.3: Imágenes del conjunto de entrenamiento de MNISTFashion correspondientes a las categorías de vestido, bota, bolso y abrigo, respectivamente.
Fuente: Propia

Capítulo 5

Implementación y desarrollo

Este capítulo de la memoria tratará sobre la implementación en código del algoritmo OLA. Como se ha mencionado en la sección 4.2, este algoritmo ha sido desarrollado en Python y los conjuntos de datos elegidos para realizar los experimentos han sido CIFAR10, MNIST y MNISTFashion.

5.1. Desarrollo

5.1.1. Código implementado

Antes de comenzar el desarrollo del código, se ha hecho una lectura en profundidad del algoritmo a implementar (OLA)[27] y se han realizado los tutoriales de Pytorch[39] para una mejor comprensión de la herramienta. Para realizar el método OLA se ha seguido el algoritmo descrito en el apartado 4.1. Este algoritmo ha sido dividido en distintos métodos, formando una librería para facilitar el posterior uso a usuarios. A continuación se detallan los métodos implementados y su función.

def createHouseholderMatrix(x, N):):

- Parámetros: El parámetro x corresponde al vector y, mencionado en el algoritmo del método OLA expuesto en la sección 4.1 de esta memoria. Recordar que dicho vector es el resultado de multiplicar el vector de salidas h_i para el nodo

actual por la matriz H (y=Hh_i). El parámetro N es el número del conjunto de datos a optimizar menos el número de nodos óptimos determinados hasta el momento(L).

- Resultado: Devuelve la nueva matriz de Householder.
- Descripción: Crea la nueva matriz de Householder haciendo ceros las últimas P-L-1 componentes de y. Este método utiliza las transformaciones de Householder y es llamado cuando el nodo es óptimo.

def disconnectNeuron(weights, neuron)::

- Parámetros: La variable weights se refiere a la matriz de pesos de la capa a optimizar y neuron es la neurona a desconectar (sus pesos serán puestos a 0).
- Resultado: No devuelve nada.
- Descripción: Pone a 0 todos los pesos de la neurona indicada por parámetro.

def obtainOutputs(net, optimloader):

- Parámetros: La variable net es la clase Net, que estará definida en el código del usuario y que contendrá la estructura del modelo así como los métodos que devuelven las salidas de la red. Optimloader es el conjunto de datos de optimización.
- Resultado: Retorna el vector de salidas h_i.
- Descripción: Obtiene las salidas de la red de las capas que se quieren optimizar. Devuelve un vector de vectores con todas las salidas por capa, es decir, si se quieren optimizar dos capas, devolverá un vector de tamaño 2 donde cada casilla contiene el vector de las salidas de la capa. Estos subvectores son los que hay que pasarle por parámetro al método siguiente, ya que corresponden a las salidas h_i del algoritmo.

def optimizeNodes(array_hi, weights):

- Parámetros: La variable array_hi es el vector de salidas de la capa correspondiente calculado en el método anterior. El parámetro weights se corresponde con la matriz de pesos de la capa a optimizar.

- Resultado: Regresa el número óptimo de nodos.

- Descripción: Es el método principal del algoritmo. Junto con **obtainOutputs** es el método que el usuario tiene que llamar en su código. Su función es que dadas las salidas de la capa a optimizar y los pesos de dicha capa calcule el número de nodos óptimos de esa capa y desconecte los que son redundantes.

Auxiliarmente el usuario tiene que crear un método denominado **getOutput** cuando define la estructura de la red. Este método, incluido dentro de la clase Net tendrá la misma función que el método **forward** solo que en vez de devolver la salida de la red, devolverá un vector con la salida de las capas que se quieran optimizar.

Para el desarrollo del código se ha seguido un diseño basado en pruebas, estableciendo una serie de objetivos parciales antes de llegar a la total implementación. Estos objetivos han sido:
- Utilizar inicialmente el dataset CIFAR10 con el número de datos de entrenamiento del código del tutorial de Pytorch.
- Utilizando como base el código del tutorial realizado modificarlo para que guarde la red entrenada en un fichero para su posterior carga y así evitar entrenar la red siempre.
- Empezar con la implementación del algoritmo devolviendo solo el vector de salidas h_i.
- Continuar calculando el vector y ($y = Hh_i$).
- Implementar la totalidad del algoritmo y proceder a la siguiente fase.

Después de la fase de implementación se ha realizado la de validación, que comprueba la validez de los resultados obtenidos. Para esto, se han tenido en cuenta el número óptimo de nodos generados (que estuviese en un número razonable) y los resultados obtenidos comparándolos con Dropout, la red sin optimización y la red optimizada por OLA. Estos resultados se comentarán en profundidad en el capítulo

6 de esta memoria. Para poder utilizar el método necesitamos partir de un conjunto de datos de optimización, que es un subconjunto de los datos de entrenamiento elegidos al azar. Comentar que el paso de optimización se lleva a cabo durante el aprendizaje tras la actualización de pesos y ese paso permite determinar que neuronas se deben mantener y cuales podemos desechar de alguna manera anulando sus conexiones.

5.1.2. Modelos de redes utilizados

Todas las redes utilizadas para el proyecto han sido redes neuronales convolucionales, explicadas en el apartado 3.1.4. Con el fin de obtener un gran número de resultados para poder desarrollar comparaciones concluyentes se han realizado pruebas con varios modelos de redes para cada dataset que se detallan a continuación. Añadir que se van a diferenciar tres ”tipos”de redes: red normal, red optimizada por OLA y red optimizada por Dropout, cuyos resultados se verán en el capítulo 6.

CIFAR10

Las redes utilizadas con este dataset cuentan con dos capas convolucionales, una de reducción (pooling) y tres capas lineales. A continuación se inserta una tabla con la estructura seguida.

Capa	Neuronas y características
Convolucional	(3,32,5)
Pool	(2,2)
Convolucional	(32, 128, 5)
Lineal	(128*5*5, 1200)
Lineal	(1200, variable)
Lineal	(variable, 10)

Tabla 5.1: Estructura de la red usada para el dataset CIFAR10

Como podemos ver en la tabla 5.1 en las capas lineales aparece la palabra variable

ya que este será el número de nodos que modificaremos para realizar pruebas con la red. Los nodos elegidos han sido 400, 600 y 800. Cuando trabajemos con Dropout en esta red este se realizará entre las últimas dos capas lineales, con una probabilidad del 0.20 y 0.25 y cuando trabajemos con OLA el conjunto de datos de optimización será de 1000.

MNIST

La estructura de la red para este conjunto de datos es la que se referencia en la tabla 5.2. En este caso, el modelo cuenta con dos capas convolucionales, una de reducción y dos lineales y no se han realizado modificaciones en los nodos de la red ya que los resultados variaban muy poco y no eran muy significativos. Cuando se trabaje con Dropout se añadirá entre los dos capas lineales y tendrá una probabilidad del 0.20 y del 0.25 % y cuando se trabaje con OLA el número de imágenes de entrada elegido para optimizar será 1000, 2000 o 4000.

Capa	Neuronas y características
Convolucional	(1,10,5)
Pool	(2,2)
Convolucional	(10, 20, 5)
Lineal	(320, 70)
Lineal	(70, 10)

Tabla 5.2: Estructura de la red usada para el dataset MNIST

MNISTFashion

Para encontrar la estructura de la red que mejor resultados obtiene se han realizado pruebas con tres modelos distintos que se detallan a continuación. Finalmente, el modelo elegido para realizar las pruebas de la red optimizada y la red Dropout ha sido el tercero, expuesto en la tabla 5.5. Todos los modelos cuentan con dos capas convolucionales, una de reducción y tres capas lineales. Cuando se trabaje con Dropout

se añadirá entre los dos capas lineales y tendrá una probabilidad del 0.20 y del 0.25 % y cuando se trabaje con OLA el número de imágenes de entrada elegido para optimizar será 1000, al igual que ocurría con CIFAR10.

Capa	Neuronas y características
Convolucional	(1, 6, 5)
Pool	(2, 2)
Convolucional	(6, 16, 5)
Lineal	(256, 120)
Lineal	(120, 84)
Lineal	(84, 10)

Tabla 5.3: Estructura del primer modelo de red utilizado para el dataset MNISTFashion

Capa	Neuronas y características
Convolucional	(1, 6, 5)
Pool	(2, 2)
Convolucional	(6, 12, 5)
Lineal	(12*4*4, 120)
Lineal	(120, 60)
Lineal	(60, 10)

Tabla 5.4: Estructura del segundo modelo de red utilizado para el dataset MNISTFashion

Capa	Neuronas y características
Convolucional	(1, 32, 5)
Pool	(2, 2)
Convolucional	(32, 64, 5)
Lineal	(64*4*4, 600)
Lineal	(600, 120)
Lineal	(120, 10)

Tabla 5.5: Estructura del tercer modelo de red utilizado para el dataset MNISTFashion

5.2. Integración de la librería OLA

En esta sección se describen los requisitos para usar la librería desarrollada y los pasos para su integración en otro proyecto.

5.2.1. Introducción y requisitos

Las técnicas de regularización son utilizadas para mejorar el rendimiento de la red. En esta memoria se ha desarrollado una técnica llamada OLA, cuya finalidad es obtener la estructura oculta óptima de la red durante su fase de aprendizaje y en este apartado mostraremos su implementación con un ejemplo práctico.

Esta técnica ha sido desarrollada en Python utilizando PyTorch y después de realizar las pruebas correspondientes se ha pasado a librería para facilitar la implementación en cualquier código.
Por esto, para el uso de este algoritmo solo se necesita tener Python y PyTorch instalados, ya que uno es el lenguaje de programación en el que está realizado y el otro es el framework utilizado para su implementación.

5.2.2. Ejemplo de uso

En este apartado se muestra un ejemplo de uso del algoritmo OLA y las implementaciones que tiene que hacer el usuario en su código para que funcione correctamente.

Las funciones se encuentran en un fichero llamado *optimizeAndLearning.py* que habrá que descargar y se incluirá en el código deseado mediante la sentencia correspondiente (`from optimizeAndLearning import *`). Ambos ficheros tienen que estar en la misma carpeta.

Para entrenar un clasificador de imágenes se seguirán los siguientes pasos:
- Cargar los conjuntos de datos.
- Definir la red (CNN).
- Definir una función de pérdida.
- Entrenar la red con los datos de entrenamiento.
- Comprobar la red con los datos de validación.

Vamos a suponer que el usuario ya ha implementado en su código todos estos pasos y lo que quiere añadir es la optimización en las dos últimas capas. Tomaremos como modelo de la red el definido en la tabla 5.5, con el dataset MNISTFashion4.3.

Lo primero que tiene que hacer el usuario es crear el conjunto de datos a optimizar, puesto que solo quiere optimizar (supongamos) 1000 imágenes. Este conjunto de datos tiene que llevar el nombre de **optimloader**.

```
optimset = torchvision.datasets.FashionMNIST(root='./data', train=True,
                                        download=True, transform=transform)

optimset.data = optimset.data[:NUM_DATA_TRAINING]

optimloader = torch.utils.data.DataLoader(optimset, batch_size=4,
                                          shuffle=True, num_workers=2)
```

Figura 5.1: Implementación del conjunto de datos de optimización. El dataset en este caso es MNISTFashion y NUM_TRAINING=1000.
Fuente: Propia

Una vez creado su conjunto de datos de optimización, tiene que añadir el método

denominado **getOutput** en la definición de la red. La salida de este método depende de las capas que quiera optimizar el usuario, como se ha mencionado, quiere optimizar las dos últimas capas, por lo que estas serán las que tiene que devolver.

```
def getOutput(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 64*4*4)
    x1 = F.relu(self.fc1(x))
    x2 = F.relu(self.fc2(x1))
    return [x1, x2]
```

Figura 5.2: Implementación del método getOutput donde devuelve las dos últimas capas (sin contar la de salida) que son las que se desean optimizar.
Fuente: Propia

Lo último sería incluir el paso de optimización en el entrenamiento. Este paso se puede realizar en cada época del entrenamiento o bien después de un número determinado de épocas. La optimización se incluiría dentro del entrenamiento de la siguiente forma:

```
arraysOutput = obtainOutputs(net, optimloader)
weights = net.fc2.weight
optimizeNodes(arraysOutput[0], weights)
weights = net.fc3.weight
optimizeNodes(arraysOutput[1], weights)
```

Figura 5.3: Implementación del paso de optimización de las dos capas deseadas durante el entrenamiento de la red.
Fuente: Propia

En este caso el método **optimizeNodes** sería llamado dos veces debido a que son dos las capas que se quieren optimizar. También hay que obtener los pesos de dichas capas y pasarlos por parámetro a la función junto con el vector de salida correspondiente a la capa.

Capítulo 6

Resultados

En este capítulo se recogen todas las pruebas y resultados de los experimentos que se han realizado para comprobar el efecto de algunas técnicas de regularización en redes neuronales profundas y con ello cumplir con el objetivo principal del trabajo: analizar algunas técnincas de regularización de DNNs basadas en la optimización de la estructura de la red. Estas técnicas serán Dropout y OLA, comentadas en esta memoria. Todas las pruebas han sido hechas entrenando y optimizando la red simultáneamente a pesar de que el algoritmo OLA permite optimizar redes ya entrenadas.
El número de épocas utilizado para cada dataset ha sido 40 para CIFAR10, 17 para MNIST y 65 para MNISTFashion. Se han determinado estas cifras debido a que la primera ejecución de la red se utilizó para comprobar en qué iteración del entrenamiento el loss de la red llegaba al mínimo.

6.1. Reducción de nodos y generalización de la red

En las siguientes tablas se muestran los resultados obtenidos en los sets de desarrollo y de test con CIFAR10, MNIST y MNISTFashion. Las figuras recogen la siguiente información:
- El dataset utilizado en cada experimento. - El número inicial de nodos en la capa a optimizar. En todas las redes la capa a optimizar ha sido la penúltima capa

full-connected, justo antes de la capa de salida.

- El número final de nodos de la capa mejorada después de la optimización con el algoritmo OLA.

- El rendimiento de la red con la técnica OLA.

- El rendimiento de la red con un porcentaje de poda del 0.20 % con la técnica Dropout.

- El rendimiento de la red con un porcentaje de poda del 0.25 % con la técnica Dropout.

- El rendimiento de la red sin ningún tipo de optimización ni técnica de regularización.

Dataset	Nº inicial de nodos	Nº final de nodos
CIFAR10	600	154
MNIST	70	47
MNISTFashion	120	39

Tabla 6.1: Resultados de los nodos finales de la capa después de aplicar el algoritmo OLA

Dataset	Rendimiento OLA	Rendimiento Dropout (0.20)	Rendimiento Dropout (0.25)	Rendimiento sin optimizacion
CIFAR10	75 %	73 %	72 %	73 %
MNIST	98 %	99 %	99 %	98 %
MNISTFashion	92 %	91 %	91 %	91 %

Tabla 6.2: Resultados del rendimiento de la red probando en diferentes conjuntos de datos

En todos los conjuntos de datos se cumple que los mejores resultados se obtienen tanto con Dropout como con OLA.

En el caso de CIFAR10 analizamos que la red ofrece un mejor rendimiento cuando es optimizada con OLA. Si se utiliza la técnica Dropout (tanto con 0.20 % o con 0.25 %) o no se usa optimización los resultados son muy similares. Por otra parte, se

han desconectado un total de 446 neuronas, consideradas redundante por el algoritmo OLA.

Si tratamos a MNIST observamos que todos los resultados son bastantes parecidos, tanto si la red cuenta con optimización como si no, además el porcentaje de acierto es bastante alto. Esto es bastante normal en este dataset ya que es un conjunto muy simple. Cada imagen en MNIST está perfectamente centrada y el conjunto se haya idealmente balanceado, cosa que en el mundo real no sucede. Además, los ejemplos que cada clase presenta cuentan con poca variabilidad al tratarse de número escritos a mano; por ejemplo, no hay muchas maneras de escribir un cero. El número de nodos final de esta red después de aplicar OLA es 47, frente a 70 que tenía al principio, con lo cual se han eliminado 23 nodos considerados redundantes.

Por último, el conjunto de datos de MNISTFashion muestra resultados muy similares entre las redes mejoradas y no, aunque una leve mejoría (del 1 %) en la red optimizada con la técnica OLA. En este conjunto de datos los nodos finales han sido 39, por lo que se han eliminado un total de 81 nodos.

Viendo los resultados de las tablas 6.1 y 6.2 concluimos que la técnica OLA es la que ofrece mejores resultados en cuanto a la generalización de la red. Para tener en cuenta la influencia de los diferentes parámetros, se han realizado varios experimentos adicionales. A continuación se muestran los resultados de estos experimentos.

6.2. Influencia del número inicial de nodos

En esta sección se probará el rendimiento de la red cuando se aumente o disminuye el número inicial de nodos de la capa que se optimiza. Las pruebas son hechas con CIFAR10 y el número de nodos inicial para realizar las pruebas será 400 y 800, disminuyendo y aumentando los 600 nodos iniciales del primer experimento del dataset; resultados que se pueden consultar en 6.1 y 6.2.

Aunque la diferencia con Dropout es mínima, la red obtiene mejores resultados cuando trabaja con OLA, independientemente del número inicial de nodos de la red.

Dataset	Nº inicial de nodos	Nº final de nodos
CIFAR10	400	122
CIFAR10	800	264

Tabla 6.3: Número de nodos iniciales y finales en el conjunto de datos CIFAR10 después de aplicar OLA.

Dataset	Rendimiento OLA	Rendimiento Dropout (0.20)	Rendimiento Dropout (0.25)	Rendimiento sin optimizacion
CIFAR10	74 %	73 %	72 %	73 %
CIFAR10	74 %	71 %	73 %	71 %

Tabla 6.4: Resultados del rendimiento en el conjunto de datos CIFAR10 con un número de nodos distintos en la penúltima capa totalmente conectada de la red.

En ambos casos se obtiene la mejor precisión con esta técnica.

6.3. Optimización de varias capas

En las secciones anteriores la capa optimizada era la penúltima de la red, por lo que se ha estudiado cómo funcionaría el algoritmo OLA si se lleva a cabo en más de una capa. Se han realizado las pruebas en CIFAR10 y MNISTFashion y en ambos modelo de red se ha pasado de optimizar una capa a optimizar las dos últimas capas full-connected antes de la capa de salida.

En la tabla 6.5 se muestra el número inicial de nodos de ambas capas optimizadas de la siguiente manera: *antepenúltima capa de la red/penúltima capa de la red*. De la misma manera se muestra el número final de nodos. También se muestran los resultados del porcentaje de acierto de OLA en una capa, vistos en la tabla 6.2, para una mejor interpretación de la prueba realizada.

Según los resultados de las tablas anteriores observamos que el número de nodos eliminados se mantiene bastante parecido en las mismas capas que se optimizan con OLA normal u OLA multicapa. Aunque la tabla no recoge el dato de los nodos finales

Dataset	Nº inicial de nodos	Nº final de nodos multicapa
CIFAR10	1200/600	707/297
MNISTFashion	600/120	346/43

Tabla 6.5: Resultados del número de nodos en el conjunto de datos CIFAR10 y MNISTFashion después de aplicar OLA a dos capas.

Dataset	Rendimiento OLA una capa	Rendimiento OLA multicapa	Rendimiento sin optimizacion
CIFAR10	75 %	73 %	73 %
MNISTFashion	92 %	92 %	91 %

Tabla 6.6: Resultados del rendimiento en el conjunto de datos CIFAR10 y MNISTFashion.

con OLA en una capa, los recordamos de la tabla 6.1; observando por ejemplo que en MNISTFashion el número de nodos finales aplicando OLA a una capa es 39 (de 120) mientras que en esa misma capa aplicando OLA a más de una es 43. Este resultado nos muestra que el algoritmo trabaja bien en ambos escenarios (optimizando más de una capa o solo una).

Si nos centramos ahora en la precisión obtenida concluimos que la técnica OLA sigue dando mejores resultados que ninguna optimización y en el caso de MNISTFashion se mantiene el mismo porcentaje de acierto en la red tanto si se optimiza una capa como si se optimizan varias. Si comprobamos el resultado obtenido en CIFAR10 el mejor valor se obtiene de aplicar OLA en una única capa.

6.4. Influencia del conjunto de datos de optimización

Aunque no se ha comentado antes, las pruebas realizadas hasta ahora han sido con un conjunto de datos de optimización de tamaño 1000, es decir, se utilizan 1000 imáganes del dataset de entrenamiento para optimización.

A continuación se mostrarán los resultados de las pruebas realizadas modificando el conjunto de datos de optimización. Dichas pruebas se han realizado con MNIST y con los conjuntos de datos de optimización de tamaño 1000, 2000 y 4000. Solo se han realizado estos experimentos con la técnica OLA ya que es la que permite al usuario modificar dichos parámetros.

Dataset	Nº inicial de nodos	Nº final de nodos	Tamaño del conjunto de datos	Rendimiento OLA
MNIST	70	47	1000	98 %
MNIST	70	52	2000	98 %
MNIST	70	48	4000	99 %

Tabla 6.7: Resultados en MNIST cuando se modifica el tamaño del conjunto de datos de optimización.

Viendo los resultados de la tabla anterior podemos concluir que se mejora un poco el porcentaje de acierto de la red cuando se aumenta el tamaño del conjunto de datos de optimización, aunque no es algo que sea muy determinante para su rendimiento puesto que la mejoría es escasa.

6.5. Influencia del número de épocas entre los pasos de optimización

Todas las pruebas realizadas anteriormente han sido aplicando las técnicas de optimización en todas las iteraciones del entrenamiento de la red. En este caso, vamos a observar cómo se comporta la red si variamos el número de veces que se aplica la optimización, es decir, estudiaremos las técnicas aplicándolas cada 5, 10 y 20 épocas. El conjunto de datos de optimización en este tipo de pruebas vuelve a ser 1000.
Se ha escogido el dataset MNISTFashion para hacer este tipo de pruebas ya que de los tres dataset utilizados para las pruebas es el que tiene el mayor número de épocas, por lo que se verá mejor la influencia. Además, por un lado, el conjunto de datos MNIST obtiene buenos resultados con 17 épocas por lo que la variabilidad en el número de

épocas sería mínima y por otra parte, el conjunto de datos CIFAR10 es bastante lento a la hora de realizar el entrenamiento de la red.

Dataset	Nº inicial de nodos	Nº final de nodos	Intervalo de épocas entre optimizaciones	Rendimiento OLA	Rendimiento sin optimización
MNISTFashion	120	31	5	91.95 %	91 %
MNISTFashion	120	31	10	92.14 %	91 %
MNISTFashion	120	28	20	91.91 %	91 %

Tabla 6.8: Resultados en MNISTFashion cuando se modifica el número de épocas entre los pasos de optimización.

Se ha realizado una optimización cada 5, 10 y 20 épocas por lo que se ha utilizado la técnica de optimización un total de 13, 6 y 3 veces, respectivamente, ya que el total de épocas que se entrena la red es 65.

La conclusión a la que podemos llegar evaluando los resultados de la tabla 6.8 es que, centrándonos en el número final de nodos, la cifra no varía mucho entre una prueba y otra. De hecho, tanto si se realiza el paso de optimización 13 veces como si se hace 6, el número de nodos redundante es exactamente el mismo.

Si nos fijamos en la precisión de acierto de la red los resultados nos muestran un porcentaje superior al 90 % en todos los escenarios. Se ha decidido mostrar el valor en decimal puesto que basándonos en resultados anteriores era muy probable que todos mostrasen cifras altas y similares en el rendimiento de la red. Teniendo esto en cuenta, concluimos que el mejor resultado se obtiene aplicando el algoritmo 6 veces, es decir, cada 10 épocas (siendo el total de épocas 65).

Capítulo 7

Conclusiones y trabajos futuros

7.1. Conclusiones

El diseño de la estructura de la red neuronal es muy importante ya que si la red cuenta con pocos nodos ocultos no aprende bien los patrones de entrenamiento, provocando que los resultados en el reconocimiento de los datos de validación sean pobres. Por otro lado, si la red cuenta con demasiados nodos ocultos esta memoriza los patrones de entrenamiento y no generaliza bien en la salida de los datos, es decir, aprenderá correctamente los patrones de entrenamiento, pero si le muestras otro que no se encuentre en dicho conjunto de datos no será capaz de extrapolar la información.

Este Trabajo Fin de Grado implementa un algoritmo de optimización y aprendizaje (OLA) para intentar reducir los tiempos de aprendizaje y solventar el último problema descrito. OLA calcula la estructura óptima de la red durante el aprendizaje, es decir, obtiene el conjunto de nodos óptimos de la red eliminando así los nodos redundantes.

Para la realización de este trabajo se ha llevado a cabo un estudio de la estructura de las redes neuronales y las técnicas de regularización más conocidas. Para poder establecer una conclusión de OLA basada en experimentos se ha implementado en varias estructuras de redes con diferentes conjuntos de datos de entrada. Por otro lado, también se ha utilizado el método Dropout a fin de comparar ambas técnicas.

Como conclusión final y basándonos en los resultados mostrados en el capítulo

anterior, podemos deducir que el algoritmo OLA mejora la estructura de la red de una manera más óptima que Dropout ya que la precisión obtenida en las redes con OLA es superior a la obtenida con Dropout. Por otro lado, OLA permite obtener una estructura óptima, eliminando nodos redundantes y proporcionado una reducción de la carga computacional cuando se utiliza la red para predicción.

7.2. Trabajo futuro

El resultado original de este proyecto es la creación de una librería en Python con el método OLA, por lo que el trabajo futuro será optimizar y mejorar el código desarrollado.

Esto se puede hacer gracias a la estructura de las matrices de Householder. Volviendo al desarrollo del algoritmo OLA en la sección 4.1, una optimización sería calcular un nuevo vector de Householder para anular las últimas P-L-1 componentes del vector y en vez de actulizar la matriz H. Esto implica que no se tenga que calcular la matriz H' ni realizar el producto HH'. Otra optimización sobre el código sería evitar calcular el producto matriz por vector necesario para obtener el vector y, lo que reduce el número de operaciones.

Otra línea que se puede seguir en el futuro es ampliar la batería de pruebas expuesta: aplicando el algoritmo a otros conjuntos de datos, comparar con distintos métodos de regularización o realizando un número mayor de pruebas sobre las ya mencionadas. Asimismo, otra línea interesante sería la adaptación del algoritmo para la optimización de capas convolucionales.

Bibliografía

[1] Otto Colomina Francisco Escolano Miguel Ángel Ortega Maria Isabel Alfonso, Miguel Ángel Cazorla. *Inteligencia Artificial. Modelos, Técnicas y Áreas de aplicación*. 2003.

[2] Alan M. Turing. *Computing Machinery and Intelligence*, pages 23–65. Springer Netherlands, Dordrecht, 2009.

[3] Damián Jorge Matich. Redes neuronales: Conceptos básicos y aplicaciones. 2001.

[4] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[5] D. O. Hebb. *The organization of behavior a neuropsychological theory.* Wiley, New York, 1949.

[6] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[7] B. Widrow and M. Hoff. *Adaptive switching circuits*. 1960.

[8] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[9] P. WERBOS. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[10] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[11] Xiaohui Liua Nianyin Zengb Yurong Liuc Fuad E. Alsaadi Weibo Liua, Zidong Wanga. A survey of deep neural network architectures and their applications.

[12] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[13] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing, 2017.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[15] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. *arXiv e-prints*, page arXiv:1409.3215, September 2014.

[17] Meriem Bahi and Mohamed Batouche. Deep learning for ligand-based virtual screening in drug discovery. pages 1–5, 10 2018.

[18] Originally published by Debarko De on. Rnn or recurrent neural network for noobs, Sep 2020.

[19] Arnol Suarez, Andres Jimenez, Mauricio Castro Franco, and Angel Cruz-Roa. Clasificación automática de coberturas del suelo en imágenes satelitales utilizando redes neuronales convolucionales: Un caso aplicado en parques nacionales naturales de colombia. 10 2016.

[20] Alberto Torres Barrán y Roi Naveiro, Jun 2019.

[21] Luis Pastor Sánchez Marco Antonio, Cornelio Yáñez. Algoritmo backpropagation para redes neuronales: conceptos y aplicaciones. 2006.

[22] Tom M. Mitchell. Machine learning, 2017.

[23] Geoffrey E Hinton et al. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA, 1986.

[24] Juan Velásquez, Fernan Villa Garzon, and Reinaldo Souza. Una aproximación a la regularización de redes cascada-correlación para la predicción de series de tiempo. *Investigacao Operacional*, 28:151–161, 12 2008.

[25] Martin de la Vega José Luis Molinari Sotomayor German Carrascosa Alejandro Lorefice Hugo Chanampe, Silvana Aciar. Modelo de redes neuronales convolucionales profundas para la clasificación de lesionesen ecografías mamarias.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[27] Pilar Bachiller Burgos. Ola: Algoritmo de optimización y aprendizaje para redes neuronales de tipo backpropagation. 2000.

[28] Deep learning architectures.

[29] Hosein Shahnas. The application of machine learning and artificial neural networks in geoscienceslecture 10 –deep learning: Neural networks.

[30] Xavier Frazão and Luís A. Alexandre. Dropall: Generalization of two convolutional neural network regularization methods. In Aurélio Campilho and Mohamed Kamel, editors, *Image Analysis and Recognition*, Cham, 2014. Springer International Publishing.

[31] What is max pooling in convolutional neural networks?

[32] P. Bachiller, R. M. Pérez, P. Martínez, P. L. Aguilar, and P. Díaz. Optimal hidden structure for feedforward neural networks. In Bernd Reusch, editor, *Computational Intelligence*, pages 684–685, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[33] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[34] P. Bachiller; R.M. Perez; P. Martinez; P.L. Aguilar. ”neural network size reduction through orthogonal transformation”. 1999.

[35] Pytorch.

[36] Cifar10.

[37] The mnist database.

[38] Zalando Research. Fashion mnist, Dec 2017.

[39] Pytorch tutoriales.