Trabajo de Fin de Grado

Implementación de Técnicas de Deep Learning
Implementation of Deep Learning Techniques

Bryan García Navarro

La Laguna, 8 de septiembre de 2015

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y Profesor Contratado Doctor de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **José Demetrio Piñeiro Vera**, con N.I.F. 43.774.048-B Profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

CERTIFICAN

Que la presente memoria titulada:
"***Implementación de Técnicas de Deep Learning***"

ha sido realizada bajo su dirección por D. **Bryan García Navarro**, con N.I.F. 78.725.652-X.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 8 de septiembre de 2015.

Agradecimientos

A Jesús y Demetrio, por sus valiosos conocimientos, dedicación y tiempo que me han brindado para guiarme y encaminarme por el camino correcto en este TFG, permitiéndome tener la oportunidad de aprender todo un mundo nuevo.

A mi familia, por su incondicional apoyo y fe ciega sobre mí durante todos estos años en la universidad, que me han permitido poder terminar esta dura travesía. Y en especial a ti, Beatriz, por estar siempre ahí cada día, por tu cariño y por tu confianza incuestionable hacia mí.

A mis amigos y compañeros, en los que he encontrado apoyo todo este tiempo y con quienes he compartido alegrías, tristezas e innumerables horas de trabajo.

A todos ustedes, el mayor de mis reconocimientos y mi infinita gratitud.

Licencia

Resumen

Recientemente se han anunciado importantes avances en diversos campos de la inteligencia artificial: como son el reconocimiento del habla, la visión y la audición artificial o el procesamiento natural del lenguaje. De algunos de ellos, como es el caso de la visión por ordenador, los expertos incluso han afirmado que se está cerca de alcanzar el mismo nivel de precisión que el que caracteriza a los seres humanos. Detrás de estos avances, en problemas que hasta el momento se consideraban difíciles de resolver por una máquina, están ciertas técnicas del ámbito del aprendizaje automático (fundamentalmente estructuras modulares en forma de redes o grafos) que han sido redenominadas como técnicas de «*Deep Learning*».

En esta línea se pretende explorar, desarrollar e implementar este tipo de técnicas aplicadas a problemas de interés en robótica, visión y audición artificial o análisis de datos y de procesos.

En este Trabajo de Fin de Grado se mostrarán algunas de las técnicas y mecanismos más importantes y destacados de este *Deep Learning*, así como diferentes ejemplos prácticos, culminando en un pequeño proyecto final de aprendizaje profundo haciendo uso de diferentes librerías del lenguaje Python.

Palabras clave: red neuronal, inteligencia artificial, deep learning, machine learning, aprendizaje, entrenamiento, perceptrón multicapa, neurona, capa, función de activación, backpropagation, arquitectura, python, theano, pylearn2

Abstract

Recently were announced significant progress in various fields of artificial intelligence; such as speech recognition, artificial vision and listening or natural language processing. Some of them, as the case of computer vision, experts have even said that it's close to achieving the same level of precision that characterizes human beings. Behind these advances, problems that until now were considered difficult to solve by a machine, are certain techniques in the field of machine learning (*especially modular structures as networks or graphs*) that have been redenominated as techniques of «Deep Learning».

In this field it is pretend to explore, develop and implement this type of techniques applied to problems of interest in robotics, artificial vision and listening or analysis of data and processes.

In this TFG some of the most important techniques and mechanisms of *Deep Learning* and different practical examples will be shown, as well as, culminating in a final project of these technologies using different Python libraries.

Keywords: neural networks, artificial intelligence, deep learning, machine learning, learning, training, multilayer perceptron, neuron, layer, activation function, backpropagation, architecture, python, theano, pylearn2

Índice general

Capítulo 1 Introducción 9
Capítulo 2 Deep Learning 10
2.1 Antecedentes y estado actual 10
Capítulo 3 Redes Neuronales Artificiales 12
3.1 Evolución histórica 13
3.2 Arquitectura 15
3.2.1 Función de activación 16
3.2.2 Bias 17
3.2.3 Otras arquitecturas 18
3.3 Modelos de redes neuronales 18
3.3.1 Perceptrón Multicapa 18
3.3.2 Autoencoders 20
3.4 Aprendizaje de redes neuronales 22
3.4.1 Fase de aprendizaje 22
3.4.2 Fase de ejecución 23
3.5 Entrenamiento de redes neuronales 23
3.5.1 Método del Gradiente Descendente 24
3.5.2 Algoritmo Backpropagation 24
Capítulo 4 Deep Learning en la práctica 26
4.1 Theano 26
4.1.1 Clasificación de dígitos de MNIST usando Theano 27
4.2 Pylearn2 29
4.2.1 CIFAR10 usando Pylearn2 31
Capítulo 5 Estudio e implementación de redes neuronales usando Pylearn2 33
5.1 Conjunto de datos 33
5.2 Parametrización en Pylearn2 34
5.3 Construcción de nuestra propia red neuronal 34
5.4 Resultados del entrenamiento 37
5.5 Casos de estudio de redes neuronales artificiales 39
Capítulo 6 Conclusiones y líneas futuras 44
Capítulo 7 Summary and Conclusions 45
Capítulo 8 Presupuesto 46
8.1 Coste de mano de obra 46
8.2 Coste de equipamiento 46
8.3 Coste de conexión a internet 46
8.4 Presupuesto total 47
Bibliografía 48

Índice de figuras

Figura 1: Red de neuronas que detecta rostros en una imagen
Figura 2: Arquitectura de una red neuronal artificial
Figura 3: Ejemplo de arquitectura de una red neuronal artificial
Figura 4: Incorporación de los biases a la red neuronal
Figura 5: Modelo del fotoperceptrón de Rosenblatt
Figura 6: Ejemplo de autoencoder
Figura 7: Clasificación de dígitos escritos a mano
Figura 8: Visualización de los pesos del entrenamiento con CIFAR10
Figura 9: Inicio del proceso de entrenamiento
Figura 10: Fin del proceso de entrenamiento tras 4 épocas
Figura 11: Gráfica generada con plot_monitor.py

Capítulo 1
Introducción

En estos últimos años, las grandes empresas tecnológicas están apostando por el desarrollo y la mejora de algoritmos de reconocimiento de voces, imágenes y textos en sectores como internet, las finanzas, el transporte, el diagnóstico médico o las telecomunicaciones.

Buena parte de estas mejoras se deben al denominado como ***Machine Learning***, o ***aprendizaje automático***, una rama de la Inteligencia Artificial donde el objetivo principal es que un sistema o máquina sea capaz de aprender y analizar información suministrada de ejemplo con el objetivo de predecir o generalizar ejemplos futuros, sin ningún tipo de intervención humana en el proceso. O dicho de otra forma, un conjunto de algoritmos destinados a buscar los patrones por los cuales una información de origen se transforma en una información de destino.

El campo de aplicación de este *Machine Learning* es cada vez más elevado, si bien muchas de estas aplicaciones las utilizamos a menudo sin conocer que realmente disponen de algoritmos de aprendizaje automático por debajo. Así, está presente en la gran mayoría de motores de búsqueda de internet, que se personalizan automáticamente según las preferencias del usuario; a la hora de explorar una base de datos de historiales médicos para predecir qué tipo de pacientes podrían responder a un cierto tratamiento; en el mundo de las redes sociales, donde gigantes como Facebook nos ofrecen determinada información en nuestro muro en función de nuestros gustos y el de nuestros amigos; o de los tan hablados asistentes de voz virtuales como Siri, Cortana o Google Now, capaces de aplicar este tipo de patrones de aprendizaje para mostrarnos la información requerida a través del reconocimiento de voz.

En este Trabajo de Fin Grado, no solamente entraremos en detalle sobre la importancia del *Machine Learning* y su implementación, sino que específicamente nos centraremos en el denominado ***Deep Learning***, o ***aprendizaje profundo***, que consiste en una familia de algoritmos y técnicas que facilitan enormemente el aprendizaje automático.

En las próximas páginas se presentarán las diferentes estructuras, arquitecturas y procesos de aprendizaje de información de este aprendizaje profundo, se presentarán algunos modelos de ejemplo de implementaciones de *Deep Learning* -con ayuda de diferentes librerías del lenguaje de programación **Python**- y, por último, se expondrá un modelo de estudio para un sistema de datos de *Deep Learning*.

Capítulo 2
Deep Learning

Como se describió anteriormente en el capítulo introductorio, ***Deep Learning*** no es más que un conjunto de técnicas y procedimientos algorítmicos basados en *Machine Learning* para lograr que una máquina aprenda de la misma forma que lo hace un ser humano.

Siendo más precisos, hablamos de una familia de algoritmos cuyo propósito es **simular el comportamiento que lleva a cabo nuestro cerebro** para reconocer imágenes, palabras o sonidos.

Son algoritmos que funcionan en base a «*un proceso por capas*». El aprendizaje profundo simula el funcionamiento básico del cerebro, que se realiza a través de las neuronas. En *Deep Learning*, esas neuronas serían las capas.

Si bien las redes de neuronas son las que se usarán durante este Trabajo de Fin de Grado, no es el único modelo que se emplea actualmente para que las máquinas sean capaces de aprender por sí mismas a través de este aprendizaje profundo. Otros ejemplos son los árboles de decisión, las reglas de asociación, los algoritmos genéticos, las redes bayesianas o el aprendizaje por refuerzo.

2.1 Antecedentes y estado actual

El aprendizaje profundo no es una idea de trabajo novedosa. La idea surgió alrededor de los años ochenta de la mano del investigador japonés **Kunihiko Fukushima**, que propuso un modelo neuronal de entre cinco y seis capas al que denominó *neocognitrón*. Sin embargo, las dificultades para el desarrollo de alternativas a la propuesta de Fukushima han sido muy complejas y el coste para su investigación era sumamente elevado, lo que ha catapultado que estas técnicas no se hayan vuelto a retomar con fuerza hasta hace escasamente una década, donde se ha reactivado el interés y la inversión por parte de las empresas.

Las grandes empresas tecnológicas están apostando por el desarrollo y la mejora de algoritmos de reconocimiento de voces, imágenes y textos. **Google** desarrolló con éxito redes neuronales que reconocen voces en teléfonos Android e imágenes en Google Plus. **Facebook** usa D*eep Learning* para orientar los anuncios e identificar rostros y objetos en fotos y vídeos. **Microsoft** lo hace en

proyectos de reconocimiento de voz. **Baidu**, el gran buscador chino, decidió abrir en 2013 un gran centro de investigación de *Deep Learning* en Silicon Valley, a 10 kilómetros de Google en Mountain View.

Google lleva más de dos años haciendo movimientos interesantes en el campo del *Deep Learning* y la Inteligencia Artificial. La compañía desembolsó una gran cantidad de dinero por *DeepMind* en enero de 2014: 290 millones de euros. Esta empresa, fundada en 2012, saltó al ruedo tecnológico por utilizar algoritmos de *Machine Learning* en comercio electrónico y videojuegos, y su objetivo es avanzar en la creación de un buscador que sea capaz de entender y responder las peticiones de los usuarios como una persona.

Además, Google contrató en 2013 a uno de los mayores especialistas mundiales de Machine Learning: **Geoffrey Hinton**, que en la década de los 80 investigó el desarrollo de ordenadores capaces de funcionar como el cerebro humano gracias a la unificación de patrones de datos. A día de hoy, es responsable del conocido proyecto de Google «*The Knowledge Graph*».

La otra gran tecnológica que ha apostado muy fuerte por el aprendizaje profundo es Facebook, que cuenta en su equipo con **Yann LeCun**, profesor del Instituto Courant de Ciencias Matemáticas de la Universidad de Nueva York, y experto en Machine Learning. LeCun fue el creador de la primera versión del *backpropagation*, un algoritmo de aprendizaje supervisado para entrenar redes neuronales artificiales que se detallará más adelante.

Sus investigaciones en cuanto a tratamiento de imágenes y reconocimiento de voz se refiere es lo que llevó a Mark Zuckerberg a contratarlo para su laboratorio de Inteligencia Artificial. Como LeCun ha reconocido en alguna entrevista, su idea es conseguir desarrollar un algoritmo que sea capaz de comprender el contenido que los usuarios suben a internet como lo haría un ser humano.

En **España** también existen empresas que aplican los conocimientos de *Deep Learning* en beneficio de sus clientes. Una de las más importantes es **Inbenta**, dedicada al desarrollo de software de procesamiento de lenguaje natural. Su tecnología permite que una máquina entienda y recuerde la conversación con una persona gracias a la incorporación de retención cognitiva y detección de contexto en las interacciones entre sus máquinas y usuarios.

Otro de los exponentes nacionales del *Deep Learning* es **Sherpa**, una empresa que ha diseñado un sistema que combina funciones de buscador, asistente personal y modelo predictivo para dispositivos móviles. En este campo también trabaja **Indisys**, una empresa que acaparó en 2012 el interés inversor de una multinacional del nivel de Intel, diseñando su propio asistente personal capaz de mantener conversaciones como si fuera tu padre o tu amigo.

Capítulo 3
Redes Neuronales Artificiales

Una red de neuronas, llamado comúnmente **red neuronal artificial**, es una herramienta matemática que modela -de forma muy simplificada- el funcionamiento de las neuronas en el cerebro. Otra forma de verlas, es como un procesador de datos que recibe información entrante, codificada como números, hace una serie de operaciones y produce como resultado información saliente, codificada también como otros números.

Un ejemplo concreto sería una **red de neuronas que detecte rostros en imágenes**. Es muy fácil codificar una imagen como una lista de números. De hecho, ya las codificamos así en los ordenadores. Por tanto, esta red recibiría tantos números a su entrada como píxeles tienen nuestras imágenes -o tres por cada píxel si utilizamos imágenes en color. Y si la información que esperamos a la salida es que nos diga si hay un rostro o no, basta con un solo número en la lista saliente. Podemos imaginar que si el número que sale de la red toma un valor cercano a 1 significa que hay un rostro, y si toma un valor cercano a 0 significa que no lo hay. Valores intermedios se podrían interpretar como probabilidad.

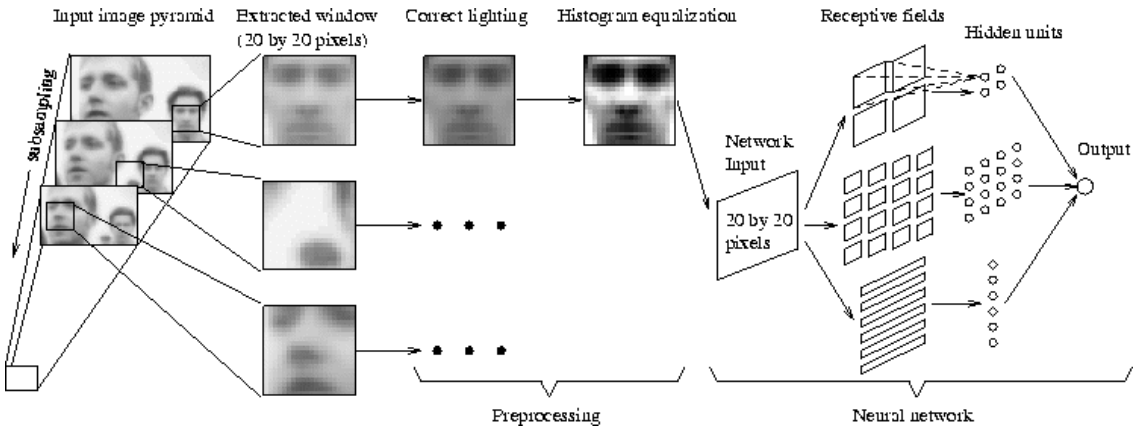


Figura 2: Red de neuronas que detecta rostros en una imagen

3.1 Evolución histórica

El matemático británico Alan Turing fue el primero en estudiar el cerebro desde el punto de vista computacional en 1936, si bien los primeros trabajos para la construcción de modelos matemáticos que imitasen el comportamiento de las neuronas del cerebro se deben a **Warren McCulloch** y **Walter Pitts** (*neurofisiólogo y matemático, respectivamente*), que presentaron en 1943 uno de los primeros modelos de una neurona artificial. Poco después, en 1949, **Donald Hebb** propuso una ley que explica a grandes rasgos el aprendizaje neuronal, conocida como «*regla de Hebb*», que se convirtió en la precursora de las técnicas de entrenamiento de redes neuronales artificiales de hoy en día.

A partir de estas aportaciones iniciales, durante la década de los años 50 y 60 surgieron nuevos desarrollos, destacando los trabajos de **Marvin Minsky** y **Frank Rosenblatt**, quienes desarrollaron el conocido como perceptrón, o perceptrón simple, un modelo sencillo capaz de generalizar el conocimiento y que, tras aprender una serie de patrones previamente, podía reconocer otros similares aunque no se les hubieran presentado con anterioridad.

Posteriormente al trabajo de Minsky y Rosenblatt sobre el perceptrón simple, **Bernard Widrow** y **Ted Hoff** desarrollaron en 1960 una importante variación del algoritmo de aprendizaje del perceptrón, la denominada «*Ley de Widrow-Hoff*», que dio lugar al modelo ADALINE (*ADAptive LINear Elements*), que constituyó la primera red neuronal artificial aplicada a un problema real: la eliminación de los ecos de las líneas telefónicas por medio de filtros adaptativos.

A pesar de los brillantes inicios de la investigación usando redes neuronales, el interés de la comunidad científica por éstas bajó enormemente al publicarse el libro: ***Perceptrons: An introduction to Computational Geometry*** del ya mencionado Marvin Minsky y de Seymour Papert, ambos investigadores del MIT. Estos autores demostraron importantes limitaciones teóricas en el aprendizaje de los modelos neuronales artificiales desarrollados hasta entonces, en particular de la red perceptrón, lo que las convertía en juguetes matemáticos sin aplicabilidad práctica real.

A partir de este trabajo, numerosos autores abandonaron el ámbito neuronal para centrarse en el análisis de los sistemas basados en el conocimiento, mucho más prometedor en aquel momento. No obstante, otros autores continuaron investigando en el campos de las redes neuronales artificiales, destacando el Asociador Lineal desarrollado por James Anderson en 1977 y su extensión conocida como red «*Brain-State-in-a-Box*», que permitieron modelizar funciones arbitrariamente complejas.

En 1982 coincidieron numerosos eventos que hicieron resurgir el interés en las redes neuronales artificiales. **John Hopfield** presentó su trabajo sobre redes neuronales describiendo con claridad y precisión una variante del Asociador Lineal inspirada en la minimización de la energía presente en los sistemas físicos, conocida como «*red de Hopfield*». Además, en ese mismo año Fujitsu comenzó el desarrollo de «*computadoras pensantes*» para diversas aplicaciones en robótica.

Un avance muy significativo tuvo lugar con la formulación de una nueva regla de aprendizaje supervisado, la denominada «***Regla Delta Generalizada***» por parte de Werbos, Parker y LeCun. Asimismo, el desarrollo por Rumelhart del algoritmo de aprendizaje supervisado para redes neuronales artificiales conocido como «***backpropagation***» ofreció en 1986 una solución muy potente para la construcción de redes neuronales más complejas al evitar los problemas observados en el aprendizaje del perceptrón simple. Este algoritmo constituye desde entonces una de las reglas de aprendizaje de mayor utilización para el entrenamiento de la red conocida como perceptrón multicapa.

En 1988, los esfuerzos de la IEEE y de la INNS se unieron para formar la «*International Joint Conference on Neuronal Networks*» (IJCNN) y, tres años más tarde, surgió la «*International Conference on Artificial Neural Networks*» (ICANN), organizada por la Sociedad Europea de Redes Neuronales. Asimismo, desde 1987 se viene celebrando la reunión anual «*Neural Information Processing Systems*» (NIPS), que constituye uno de los referentes de más alto nivel en este campo de investigación.

Como consecuencia de estos esfuerzos, las redes neuronales artificiales han experimentado un importante desarrollo en los últimos años, hasta el punto que el paradigma conexionista ha llegado a superar a las aplicaciones basadas en modelos simbólicos. En los últimos años, las investigaciones se centran en la combinación de ambos paradigmas de aprendizaje, con el fin de conseguir una mayor unión entre la capacidad de procesamiento y aproximación de las redes neuronales artificiales, que pueden llegar a soluciones sorprendentemente buenas con rapidez y poca información de partida, y el potencial de los sistemas basados en el conocimiento, como demuestran los trabajos de Tomas Hrycej (1992) en inteligencia artificial, Paul McNelis (1997 y 2005) en finanzas y Bart Baesens (2003) en minería de datos, entre otros.

3.2 Arquitectura

En la siguiente figura podemos ver la arquitectura genérica de una red neuronal artificial.

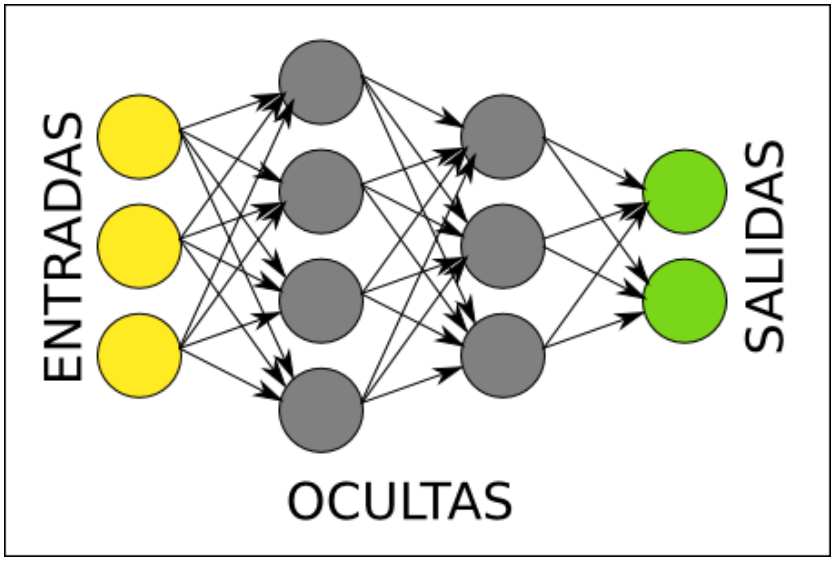


Figura 3: Arquitectura de una red neuronal artificial

Cada círculo representa una neurona. Las neuronas se organizan en capas, de la siguiente forma: las neuronas amarillas son las **entradas**, y reciben cada uno de los números de nuestra lista de números entrante, las neuronas verdes son las **salidas**, y una vez que la red realiza su operación matemática, contienen el resultado, también como una lista de números; las neuronas grises son neuronas **ocultas**, que contienen cálculos intermedios de la red.

Normalmente, todas las neuronas de cada capa tienen una conexión con cada neurona de la siguiente capa, como se representa en el diagrama anterior. Estas conexiones tienen asociado un número, que se llama **peso**. La principal operación que realiza la red de neuronas consiste en multiplicar los valores de una neurona por los pesos de sus conexiones salientes. Cada neurona de la siguiente capa recibe números de varias conexiones entrantes, y lo primero que hace es sumarlos todos.

Continuando con el ejemplo anterior, la arquitectura de la red que detecta rostros en una imagen, tendría un aspecto similar al siguiente (*sólo se dibujan las conexiones desde tres píxeles para no hacer más complicado el gráfico*):

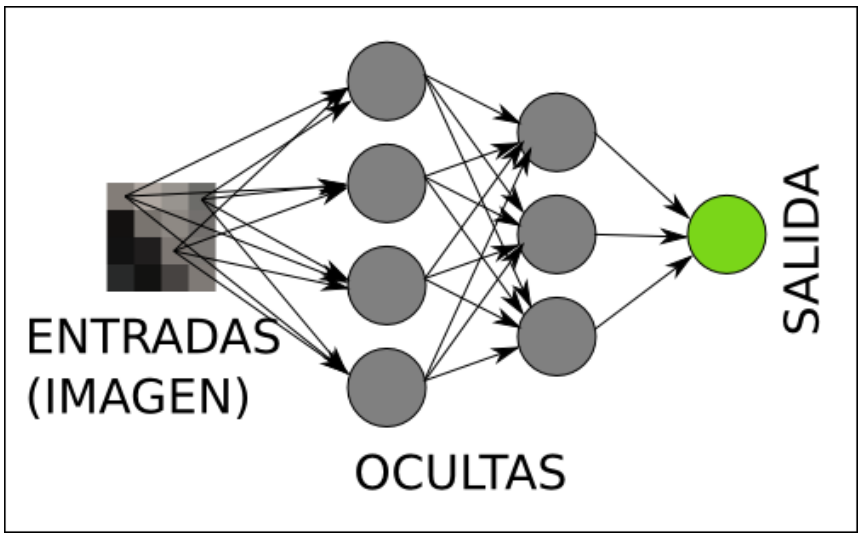


Figura 4: Ejemplo de arquitectura de una red neuronal artificial

3.2.1 Función de activación

Hasta el momento, las operaciones de la red de neuronas son sencillas, ya que se tratan de productos y sumas. Hay otra operación que realizan todas las capas, salvo la capa de entrada, antes de continuar multiplicando sus valores por las conexiones salientes. Se trata de la **función de activación**.

Esta función recibe como entrada la suma de todos los números que llegan por las conexiones entrantes, transforma el valor mediante una fórmula, y produce un nuevo número. Existen varias opciones, pero una de las funciones más habituales es la **función sigmoide**. Uno de los objetivos de la función de activación es mantener los números producidos por cada neurona dentro de un rango razonable (por ejemplo, números reales entre 0 y 1).

La función sigmoide es no lineal, esto significa que si dibujamos en una gráfica los valores de entrada en un eje y los de salida en el otro eje, el dibujo no será una línea. Esto es muy importante, porque si la función de activación que elegimos es lineal, la red estará limitada a resolver problemas relativamente simples.

Una forma sencilla de implementar redes neuronales artificiales consiste en almacenar los pesos en **matrices**. Es fácil ver que si ahora guardamos los valores de todas las neuronas de una capa en un vector, el producto del vector y la matriz de pesos de salida, nos da los valores de entrada de cada neurona en la siguiente capa. Ahora sólo falta aplicar la función de activación que hayamos elegido a cada elemento de ese segundo vector, y repetir el proceso.

3.2.2 Bias

Justo antes de aplicar la función de activación, cada neurona añade a la suma de productos un nuevo término constante, llamado habitualmente **bias**.

Una forma típica de implementar este término consiste en imaginar que extendemos la capa anterior con una neurona que siempre toma como valor un 1, e incorporar los pesos correspondientes a dicha neurona a la matriz de pesos.

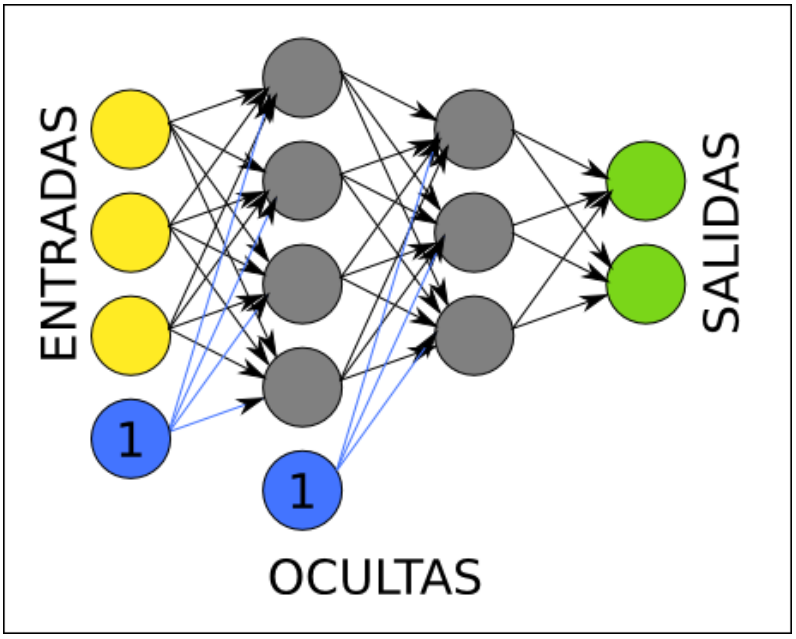


Figura 5: Incorporación de los biases a la red neuronal

Para ver su importancia con un ejemplo sencillo, imaginémonos una función de activación lineal (*no hace nada*) y sólo dos neuronas en la red, una en la capa de entrada y otra en la capa de salida.

Tal como se ha explicado antes, esta red se limita a hacer una multiplicación y nada más. Si queremos usar esta red, por ejemplo, para convertir entre grados Celsius y Fahrenheit tenemos un problema, porque a 0º C le corresponden 32º F, y sólo con una multiplicación no podemos hacer esto. Sin embargo, al introducir los biases tenemos una multiplicación y luego una suma, con lo cual ya podríamos hacer este conversor de temperaturas.

3.2.3 Otras arquitecturas

Además de la arquitectura explicada en los apartados anteriores, es interesante saber que existen arquitecturas de redes neuronales diferentes que también se usan en ocasiones para implementar *Deep Learning*.

Por ejemplo, las **redes de neuronas recurrentes** (*Recurrent Neural Networks*) no tienen una estructura de capas, sino que permiten conexiones arbitrarias entre todas las neuronas, incluso creando ciclos. Esto permite incorporar a la red el concepto de temporalidad, y permite que la red tenga memoria, porque los números que introducimos en un momento dado en las neuronas de entrada son transformados, y continúan circulando por la red incluso después de cambiar los números de entrada por otros diferentes.

Otra arquitectura interesante son las **redes de neuronas convolucionales** (*Convolutional Neural Networks*). En este caso se mantiene el concepto de capas, pero cada neurona de una capa no recibe conexiones entrantes de todas las neuronas de la capa anterior, sino sólo de algunas. Esto favorece que una neurona se especialice en una región de la lista de números de la capa anterior, y reduce drásticamente el número de pesos y de multiplicaciones necesarias. Lo habitual es que dos neuronas consecutivas de una capa intermedia se especialicen en regiones solapadas de la capa anterior.

3.3 Modelos de redes neuronales

En *Deep Learning* podemos encontrar numerosos modelos de redes de neuronas a la hora de resolver un determinado problema.

3.3.1 Perceptrón Multicapa

El modelo del **perceptrón**, también llamado perceptrón simple, fue desarrollado por Frank Rosenblatt en 1957. Su intención era ilustrar algunas propiedades fundamentales de los sistemas inteligentes en general, sin entrar en mayores detalles con respecto a condiciones específicas y desconocidas para organismos biológicos concretos. Rosenblatt opinaba que la herramienta de análisis más apropiada era la teoría de probabilidades, y esto lo llevó a una teoría de separabilidad estadística que utilizaba para caracterizar las propiedades más visibles de estas redes.

Sin embargo, como ya se ha relatado anteriormente, en 1969 Minsky y Papert publicaron su libro «*Perceptrons: An introduction to Computational Geometry*» que para muchos significó el final de las redes neuronales. En él se presentaba un análisis detallado de un perceptrón en términos de sus capacidades y limitaciones, en especial en cuanto a las restricciones que existen para los problemas que una red de tipo perceptrón puede resolver. La mayor desventaja que presentaba este tipo de redes es su **incapacidad para solucionar problemas que no sean linealmente separables**. De hecho, en dicha publicación se demostró que un perceptrón simple era incapaz de aprender una función tan sencilla como la XOR.

Minsky y Papert mostraron ese mismo año que la combinación de varios perceptrones simples podría resultar una solución interesante para tratar ciertos problemas no lineales, surgiendo por primera vez la figura del **perceptrón multicapa**. Sin embargo, los autores no presentaron una solución al problema de cómo adaptar los pesos de la capa de entrada a la capa oculta, pues la regla de aprendizaje del perceptrón simple no puede aplicarse en este escenario.

Diferentes autores han demostrado independientemente que el perceptrón multicapa es un aproximador universal, en el sentido de que cualquier función continua en un espacio R^n puede aproximarse con un perceptrón multicapa con al menos una capa oculta de neuronas. Este resultado sitúa al perceptrón multicapa como un modelo matemático útil a la hora de aproximar o interpolar relaciones no lineales entre datos de entrada y salida.

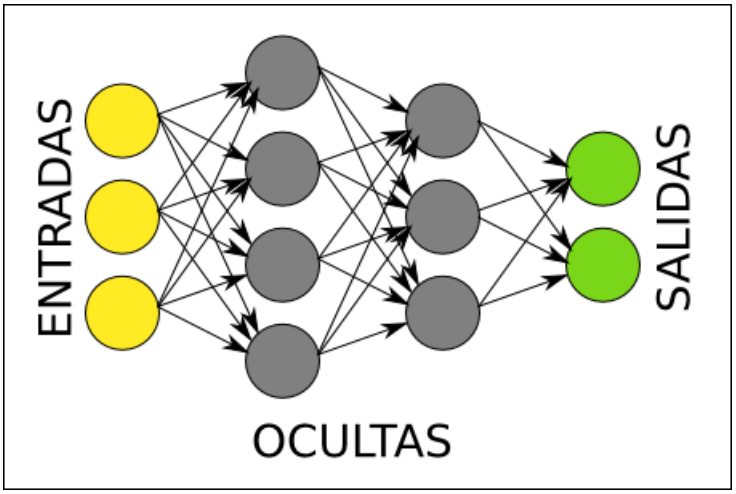
Dentro del marco de las redes de neuronas, el perceptrón multicapa es en la actualidad una de las arquitecturas más utilizadas en la resolución de problemas. Esto es debido, fundamentalmente, a su capacidad como aproximador universal, así como a su fácil uso y aplicabilidad.

Por otra parte, esto no implica que sea una de las redes más potentes y con mejores resultados en sus diferentes áreas de aplicación. De hecho, el perceptrón multicapa posee una serie de limitaciones, como el **largo proceso de aprendizaje** para problemas complejos dependientes de un gran número de variables o la **dificultad para realizar un análisis teórico** de la red debido a la presencia de componentes no lineales.

Por otra parte, es necesario señalar que el proceso de aprendizaje de la red busca en un espacio amplio de funciones una posible función que relacione las variables de entrada y salida al problema, lo cual puede complicar su aprendizaje y reducir su efectividad en determinadas aplicaciones.

Ya en la figura 2 de este Trabajo de Fin Grado hemos podido observar la

arquitectura de un perceptrón multicapa, caracterizada por disponer sus neuronas agrupadas en capas de diferentes niveles, y cada una de las capas está formada por un conjunto de neuronas:



Las neuronas de la capa de entrada no actúan como neuronas propiamente dichas, sino que se encargan únicamente de recibir las señales o patrones del exterior y propagar dichas señales a todas las neuronas de la siguiente capa. La última capa actúa como salida de la red, proporcionando al exterior la respuesta de la red para cada uno de los patrones de entrada. Las neuronas de las capas ocultas realizan un procesamiento no lineal de los patrones recibidos.

Las conexiones del perceptrón multicapa siempre están dirigidas hacia adelante, es decir, las neuronas de una capa se conectan con las neuronas de la siguiente capa, de ahí que reciban también el nombre de *redes alimentadas hacia adelante* o ***redes feedforward***. Generalmente, todas las neuronas de una capa están conectadas a todas las neuronas de la siguiente capa. Se dice entonces que existe conectividad total o que la red está **totalmente conectada**.

3.3.2 Autoencoders

Otra de las herramientas utilizadas habitualmente para implementar *Deep Learning* son los ***autoencoders***. Normalmente se implementan como redes de neuronas con tres capas, con sólo una capa oculta.

Un auto-codificador, o *autoencoder*, aprende a producir a la salida exactamente la misma información que recibe a la entrada. Por eso, las capas de entrada y salida siempre deben tener el mismo número de neuronas. Por ejemplo, si la capa de entrada recibe los píxeles de una imagen, esperamos que la red aprenda a producir en su capa de salida exactamente la misma imagen que le hemos

introducido.

La clave está en la capa oculta. Imaginémonos por un momento un auto-codificador que tiene menos neuronas en la capa oculta que en las capas de entrada y salida. Dado que exigimos a esta red que produzca a la salida el mismo resultado que recibe a la entrada, y la información tiene que pasar por la capa oculta, la red se verá obligada a encontrar una representación intermedia de la información en su capa oculta usando menos números. Por tanto, al aplicar unos valores de entrada, la capa oculta tendrá una versión comprimida de la información, pero además será una versión comprimida que se puede volver a descomprimir para recuperar la versión original a la salida.

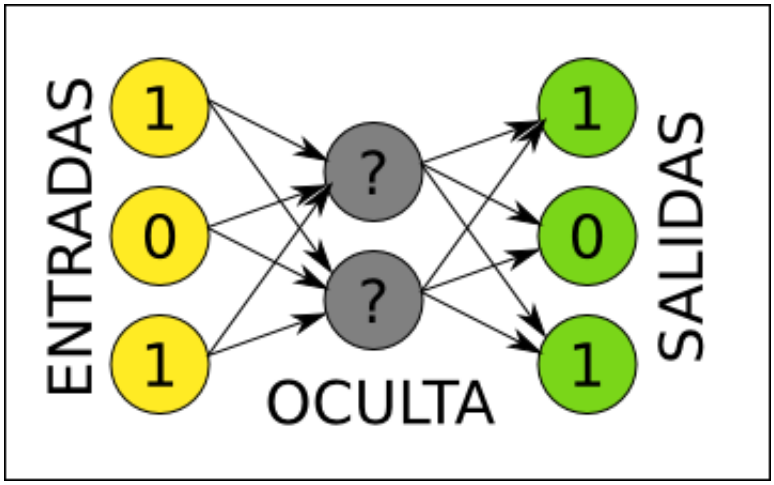


Figura 6: Ejemplo de autoencoder

De hecho, una vez entrenada, se puede dividir la red en dos, una primera red que utiliza la capa oculta como capa de salida, y una segunda red que utiliza esa capa oculta como capa de entrada. La primera red sería un compresor, y la segunda un descompresor.

Precisamente por eso, este tipo de redes se denominan auto-codificadores, son capaces de descubrir por sí mismos una forma alternativa de codificar la información en su capa oculta. Y lo mejor de todo es que no necesitan a un supervisor que les muestre ejemplos de cómo codificar información, se buscan la vida ellas solas.

Un solo auto-codificador puede encontrar características fundamentales en la información de entrada, las características más primitivas y simples que se pueden extraer de esa información, como rectas y curvas en el caso de las imágenes. Sin embargo, si queremos que nuestras máquinas detecten conceptos más complejos como rostros, nos hace falta más potencia.

Fijémonos en la operación que realiza un auto-codificador en su capa oculta. A partir de información cruda sin significado (*por ejemplo, píxeles de imágenes*), es capaz de etiquetar características algo más complejas (*por ejemplo, formas simples presentes en cualquier imagen como líneas y curvas*). Entonces la pregunta es, ¿qué pasa si al resultado codificado, en esa capa oculta, le aplicamos otro auto-codificador? Si lo hacemos bien, encontrará características más complejas todavía (*como círculos, arcos, ángulos rectos, etc*). Si continuamos haciendo esto varias veces, tendremos una jerarquía de características cada vez más complejas, junto con una pila de codificadores. Siguiendo el ejemplo de las imágenes, dada una profundidad suficiente e imágenes de ejemplo suficientes, conseguiremos alguna neurona que se active cuando la imagen tenga un rostro, y sin necesidad de que ningún supervisor le explique a la red cómo es un rostro.

3.4 Aprendizaje de redes neuronales

Durante la operatoria de una red neuronal podemos distinguir claramente dos fases o modos de operación: la **fase de aprendizaje** y la **fase de ejecución**.

Durante la primera fase, la red es entrenada para realizar un determinado tipo de procesamiento. Una vez alcanzado un nivel de entrenamiento adecuado, se pasa a la fase de ejecución, donde la red es utilizada para llevar a cabo la tarea para la cual fue entrenada.

3.4.1 Fase de aprendizaje

Una vez seleccionada el tipo de neurona artificial que se utilizará en una red neuronal y determinada su topología es necesario entrenarla para que la red pueda ser utilizada. Partiendo de un conjunto de pesos sinápticos aleatorio, el proceso de aprendizaje busca un conjunto de pesos que permitan a la red desarrollar correctamente una determinada tarea. Durante el proceso de aprendizaje se va refinando iterativamente la solución hasta alcanzar un nivel de operación suficientemente bueno.

El proceso de aprendizaje se puede dividir en tres grandes grupos de acuerdo a sus características:

- **Aprendizaje supervisado**. Se presenta a la red un conjunto de patrones de entrada y su salida esperada. Los pesos se van modificando de forma proporcional al error que se produce entre la salida real de la red y la

esperada.

- **Aprendizaje no supervisado**. Se presenta a la red un conjunto de patrones de entrada. No hay información disponible sobre la salida esperada. El proceso de entrenamiento en este caso deberá ajustar sus pesos en base a la correlación existente entre los datos de entrada.

- **Aprendizaje por refuerzo**. Este tipo de aprendizaje se ubica entre medio de los dos anteriores. Se le presenta a la red un conjunto de patrones de entrada y se le indica a la red si la salida obtenida es o no correcta. Sin embargo, no se le proporciona el valor de la salida esperada. Este tipo de aprendizaje es muy útil en aquellos casos en que se desconoce cuál es la salida exacta que debe proporcionar la red.

3.4.2 Fase de ejecución

Una vez finalizada la fase de aprendizaje, la red puede ser utilizada para realizar la tarea para la que fue entrenada.

Una de las principales ventajas que posee este modelo es que la red aprende la relación existente entre los datos, adquiriendo la capacidad de generalizar conceptos. De esta manera, una red neuronal puede tratar con información que no le fue presentada durante la fase de entrenamiento.

3.5 Entrenamiento de redes neuronales

En el contexto de las redes neuronales, el aprendizaje puede ser visto como el proceso de ajuste de los parámetros de la red. Partiendo de un conjunto de pesos aleatorios, el proceso de aprendizaje busca un conjunto de pesos que permitan a la red desarrollar una determinada tarea. El proceso de aprendizaje es un proceso iterativo, en el cual se va refinando la solución hasta alcanzar un nivel de operación suficientemente bueno.

La mayoría de los métodos de entrenamiento utilizados en las redes neuronales consisten en proponer una función de error que mida el rendimiento actual de la red en función de los pesos. El objetivo del método de entrenamiento es **encontrar el conjunto de pesos que minimizan -o maximizan- la función de error**. El método de optimización proporciona una regla de actualización de los pesos que en función de los patrones de entrada modifica iterativamente los

pesos hasta alcanzar el punto óptimo de la red neuronal.

3.5.1 Método del Gradiente Descendente

El método de entrenamiento más utilizado es el **método del gradiente descendente**. Este método define una función *E(W)* que proporciona el error que comete la red en función del conjunto de pesos *W*. El objetivo del aprendizaje será encontrar la configuración de pesos que corresponda al mínimo global de la función de error, aunque en muchos casos es suficiente encontrar un mínimo local lo suficientemente bueno.

Dado un conjunto de pesos *W(0)* para el instante de tiempo *t=0*, se calcula la dirección de máxima variación del error. La dirección de máximo crecimiento de la función *E(W)* en *W(0)* viene dado por el gradiente $\nabla E(W)$. Luego, se actualizan los pesos siguiendo el sentido contrario al indicado por el gradiente $\nabla E(W)$, dirección que indica el sentido de máximo decrecimiento. De este modo, se va produciendo un descenso por la superficie de error hasta alcanzar un mínimo local.

$$W(t+1) = W(t) - \alpha \nabla E(W)$$

Donde α indica el tamaño del peso tomado en cada iteración, pudiendo ser diferente para cada peso e -idealmente- debería ser infinitesimal. El tamaño del peso es un factor importante a la hora de diseñar un método de estas características. Si se toma un peso muy pequeño el proceso de entrenamiento resulta muy lento, mientras que si el tamaño del peso es muy grande se producen oscilaciones en torno al punto mínimo.

3.5.2 Algoritmo Backpropagation

El algoritmo ***Backpropagation*** *-o algoritmo de propagación hacia atrás-* es el método de entrenamiento más utilizado en redes con conexión hacia delante.
Es un método de aprendizaje supervisado de gradiente descendente en el que inicialmente se aplica un patrón de entrada, el cual se propaga por las distintas capas que componen la red hasta producir la salida de la misma. Esta salida se compara con la salida deseada y se calcula el error cometido por cada neurona de salida. Estos errores se transmiten hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de las capas intermedias. Cada neurona recibe un error que es proporcional a su contribución sobre el error total de la red. Basándose en el error recibido, se ajustan los errores de los pesos de cada neurona.

Un problema del algoritmo de propagación hacia atrás es que el error se va diluyendo de forma exponencial a medida que atraviesa capas en su camino hasta el principio de la red. Esto es un problema porque en una red muy profunda -con muchas capas ocultas- sólo las últimas capas se entrenan, mientras que las primeras apenas sufren cambios. Por eso a veces compensa utilizar redes con pocas capas ocultas que contengan muchas neuronas, en lugar de redes con muchas capas ocultas que contengan pocas neuronas.

Durante la aplicación del algoritmo *backpropagation*, el aprendizaje se produce mediante la presentación sucesiva de un conjunto de entrenamiento. Cada presentación completa al perceptrón multicapa del set de entrenamiento se denomina ***época***. Así, el proceso de aprendizaje se repite época tras época hasta que los pesos se estabilizan y el rendimiento de la red converge en un valor aceptable.

La forma en que se actualizan los pesos da lugar a dos modos de entrenamientos distintos, cada uno con sus ventajas y desventajas:

- **Modo Secuencial**. En este modo de entrenamiento la actualización de los pesos se produce tras la presentación de cada ejemplo de entrenamiento, de modo que también es conocido como *modo por patrón*. Si un set de entrenamientos posee N ejemplos, el modo secuencial de entrenamiento tiene como resultado N correcciones de pesos durante cada época.

- **Modo *Batch***. En este modo de entrenamiento la actualización de los pesos se produce una vez, tras la presentación de todo el set de entrenamiento. Para cada época se calcula el error cuadrático medio producido por la red.

Si los patrones de entrenamiento se presentan a la red de manera aleatoria, el modo de entrenamiento secuencial convierte a la búsqueda en el espacio de pesos en estocástica, y disminuye la probabilidad de que el algoritmo *backpropagation* quede atrapado en un mínimo local. Sin embargo, la naturaleza estocástica del modo de entrenamiento secuencial dificulta el establecimiento de condiciones teóricas para la convergencia del algoritmo.

Por su parte, el uso del modo de entrenamiento *batch* provee una estimación precisa del vector gradiente, garantizando de esta manera la convergencia hacia un mínimo local.

Capítulo 4
Deep Learning en la práctica

Lo más interesante de esta «*fiebre moderna*» por el *Machine Learning* -o el *Deep Learning* más concretamente- es que está al alcance de todo el mundo, a diferencia de lo visto anteriormente, donde nos hemos dado cuenta de las grandes dificultades encontradas en el avance de esta rama de la Inteligencia Artificial a mediados-finales del pasado siglo.

En lo que a diseño y software se refiere, existe una fuerte comunidad sobre *Deep Learning* dentro del mundo del software libre para potenciar e implementar soluciones y proyectos a estos niveles. Lenguajes de desarrollo como R, C++ o Python se nutren cada vez más de librerías fabricadas por la comunidad para lograr avances significativos en *Machine Learning*, lo que se traduce en grandes soluciones que abarcan el procesamiento del lenguaje, el reconocimiento de voz, la visión por computador, la minería de datos o el internet de las cosas.

En este Trabajo de Fin de Grado se ha optado por trabajar con el lenguaje de programación **Python**, ya que además de ser un lenguaje de alto nivel de naturaleza libre y de cierta facilidad a la hora de la implementación, nos permite hacer uso de casi un centenar de librerías orientadas al aprendizaje automático y profundo.

A lo largo de este capítulo se presentarán diferentes trabajos y ejemplos de proyectos en los que se ha trabajado para comprender el funcionamiento interno del aprendizaje automático y profundo en la práctica.

4.1 Theano

La primera escala en nuestro camino de uso de *Deep Learning* utilizando Python nos lleva hasta ***Theano***.

Desarrollado por investigadores de la universidad de Montreal, en Canadá, Theano es una potente librería matemática de Python que nos permite definir, optimizar y evaluar expresiones en las que están involucradas matrices multidimensionales, siendo de gran utilidad para nosotros a la hora de diseñar redes neuronales artificiales y obtener una mayor eficiencia en cálculos muy grandes. Theano permite, además, definiciones simbólicas y un uso totalmente transparente de los procesadores GPU.

Dos características muy interesantes que nos brinda esta librería son las variables compartidas y los parámetros de actualización. Las primeras, llamadas ***shared variables***, son un híbrido entre variables simbólicas y variables no simbólicas, cuyos valores pueden ser compartidos entre varias funciones. Estas variables compartidas se pueden usar en expresiones simbólicas como el retorno de objetos de una matriz, pero también disponen de un valor interno que define el valor que toma esta variable simbólica en todas las funciones que la utilizan.

La otra gran característica es el ***parámetro de actualización de las funciones*** de Theano. Estas actualizaciones deben ser suministradas en una lista de tuplas de la forma {*variable compartida, nueva expresión*}, aunque también puede ser un diccionario cuyas claves son variables compartidas y los valores son las nuevas expresiones. El objeto de todo esto es, que cada vez que se ejecute esta función, se sustituirá el valor de cada variable compartida por el resultado de la expresión correspondiente.

4.1.1 Clasificación de dígitos de MNIST usando Theano

El primer proyecto llevado a cabo con Theano consistiría en clasificar una serie de imágenes de dígitos escritos a mano utilizando el método de clasificación de la **regresión logística**, usando para ello redes neuronales artificiales y la conocida base de datos de dígitos de **MNIST**. Esta base de datos está conformada por alrededor de 60.000 imágenes de dígitos escritos a mano -y otras 10.000 imágenes para su testeo- consistentes en imágenes de los mismos en escala de grises.

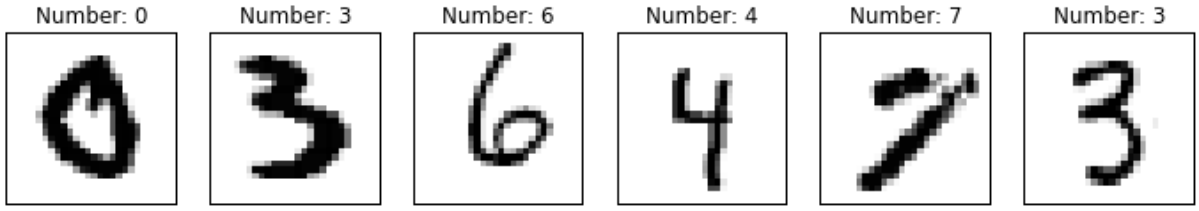


Figura 7: Clasificación de dígitos escritos a mano

Esta regresión logística es un clasificador probabilístico y lineal. Esta clasificación se realiza mediante la proyección de un vector de entrada en un conjunto de hiperplanos, cada uno de los cuales corresponde a una clase. La

distancia que hay desde la entrada a un hiperplano refleja la probabilidad de que la entrada sea un miembro de una clase o de otra. Aprender los parámetros del modelo óptimos implica minimizar una **función de pérdida**. En el caso de regresión logística, es común utilizar como pérdida la ***razón de verosimilitud negativa***. Por último, se empleará el método de *gradiente descendente estocástico con minibatches* como algoritmo de minimización de esta función de pérdida.

La red neuronal que se construirá estará modelada por medio de un **perceptrón multicapa**, disponiendo de un total de 784 entradas (*una por píxel de cada imagen: de 28x28*), 110 capas ocultas y un total de 10 salidas. Cada una de estas salidas corresponden a un dígito del 0 al 9, el cual una sola de estas entradas se activará a 1 indicando cual es el número escrito a mano que se ha pasado por la entrada de la red, una vez haya sido entrenada. Se llevarán a cabo un total de 1.000 épocas, con un índice de aprendizaje del 0,13.

Entre los diferentes objetivos para determinar el problema de clasificación de dígitos de la base de datos MNIST están:

- La creación y comportamiento de la clase que realizará la regresión logística, que incorpora diferentes tensores que actuarán como unidades de entrada y de salida de la red, incluyendo sus matrices de pesos y métodos de predicción de errores por cada ejemplo de la red, todo ello a través de variables compartidas de Theano. Dentro de los métodos definidos en esta clase cabe destacar el del cálculo de la razón de verosimilitud negativa para minimizar la función de pérdida y un método que se encarga de representar el número de errores totales sobre el número de ejemplos totales.

- El aprendizaje y entrenamiento del modelo de la red definida con anterioridad, encargadas de la minimización de la función de coste de la red y la actualización de la matriz de los pesos de todo el conjunto de datos. Con Theano esto se realiza de una forma relativamente sencilla ya que, con las funciones del propio Theano y los parámetros de actualizaciones de las mismas, permite especificar en apenas un par de comandos cómo actualizar los parámetros del modelo por medio de una lista de tuplas.

- El testeo de la red, encargado de recuperar el número de ejemplos de la red que no han logrado ser clasificados por el modelo. Para esto, se usa una función de Theano que calcula los errores que se cometen en un mini-batch -o ejemplo- del conjunto de pruebas, comparándose con el del conjunto de validación.

A la salida visualizaremos información referente al porcentaje de error de cada época, sobre cada imagen de cada dígito, incluyendo la de mejor validación y la velocidad de cómputo de la red usando el modelo elegido.

4.2 Pylearn2

Entrenar una red neuronal utilizando Theano requiere un alto índice de complejidad, ya que está en nuestra obligación definir implícitamente cómo se comportará el modelo a usar, qué algoritmos de aprendizaje usar e implementarlos *in situ*, definir e implementar la función de coste adecuada para cada problema, entrenar cada ejemplo de nuestro set de pruebas a mano y comprobar en cada caso si ha logrado ser clasificado por el modelo correctamente o no.

Es por ello que existen otras librerías de Python basadas en *Machine Learning* que nos permiten colocar una capa de abstracción superior y simplificar los procesos de experimentación con redes neuronales para un problema dado. Una de estas librerías es la ***Pylearn2***, que es la otra gran biblioteca de Python que se ha usado en este Trabajo de Fin de Grado.

Pylearn2 usa internamente Theano, y nos permite dividir los problemas de *Machine Learning* en tres partes bien diferenciadas: el **dataset** con el conjunto de datos de la muestra, el **modelo** utilizado para la red y los algoritmos de **entrenamiento** usados para entrenar el modelo. De este modo, el algoritmo de entrenamiento trabaja para adaptar el modelo a los valores previstos en el conjunto de datos.

Toda esta información viene recogida en un fichero ***YAML***, un lenguaje de marcado de documentos. Pylearn2 soporta casi una treintena de conjuntos de datos, destacando algunos como los del MNIST, CIFAR10, CSV Dataset o Sparse Dataset, y su sintaxis en YAML para estos datasets es la que sigue a continuación:

```
dataset: &train !obj:pylearn2.datasets.csv_dataset.CSVDataset {
        # Atributos del dataset
},
```

La lista de modelos de *Deep Learning* que es capaz de soportar Pylearn2 también es muy variada, destacando el MLP, el K-Means, RBM, el Autoencoder o la Softmax Regression. Su sintaxis sería la siguiente:

```
model: !obj:pylearn2.models.mlp.MLP {
       # Atributos del modelo
       layers: [ !obj:pylearn2.models.mlp.ConvRectifiedLinear {
                      # Una capa de neuronas convolucionales
                 },
                !obj:pylearn2.models.mlp.MLP {
                       # Una capa formada por un perceptrón multicapa
                   },
                !obj:pylearn2.models.mlp.Softmax {
                       # Una capa de salida con función de activación Softmax
                }
       ],
},
```

En lo referente a algoritmos de entrenamiento la diversidad es mucho más escasa, destacando principalmente los dos mayoritarios: Stochastic Gradient Descent y Batch Gradient Descent. Su sintaxis YAML:

```
algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
        # Atributos del algoritmo
},
```

A grandes rasgos, estos son los tres principales componentes de un fichero YAML, aunque es posible añadir información extra como instrucciones para el preprocesado de los datos del dataset, el índice de aprendizaje de la red neuronal, las funciones de minimización del coste de la red, monitorizar cualquier atributo inmerso en el fichero YAML a medida que la red se va entrenando o la exportación de gráficas de especial relevancia durante el entrenamiento.

Como podemos observar, el índice de complejidad respecto a Theano desciende enormemente, ya que únicamente tenemos que preocuparnos por rellenar este fichero de configuración YAML para especificar el tipo de red que queremos, abstrayéndonos de cómo implementar todas estas funciones en Python -realmente lo hace todo Theano internamente- sin realmente programar una sola línea de código.

Para entrenar la red, además de este fichero de configuración YAML necesitamos el fichero en Python que se encargará del entrenamiento en sí, llamado ***train.py***, que es un fichero que ya trae Pylearn2 al instalar la librería y que es genérico para cualquier red que creemos por lo que, salvo casos muy específicos, tampoco es necesario editarlo.

4.2.1 CIFAR10 usando Pylearn2

En el caso práctico en el que se ha trabajado para la toma de contacto con Pylearn2, se ha utilizado otra conocida base de datos de imágenes usada habitualmente en problemas de *Machine Learning*: la ***CIFAR10***. Esta base de datos contiene más de 80 millones de pequeñas imágenes a color de 32 píxeles clasificadas en distintas categorías.

Una vez con el dataset de datos, que se puede descargar de su página oficial, se procede a entrenar el modelo según lo marcado en su correspondiente fichero YAML, que es donde vendrá descrita y configurada nuestra red neuronal.

El modelo utilizado para entrenarla es el RBM, una *Máquina Gaussiana restringida de Boltzmann*, basada en una red neuronal recurrente. Se han empleado un total de 192 neuronas de entrada (*imágenes de test de 8x8 píxeles y 3 canales de colores diferentes*), 400 neuronas en sus capas ocultas y un índice de aprendizaje del 0,1. La reducción del tamaño de las imágenes se debe a que cuanto mayor es la entrada de la red, más pesos hay que entrenar, y más lento es todo el proceso de entrenamiento.

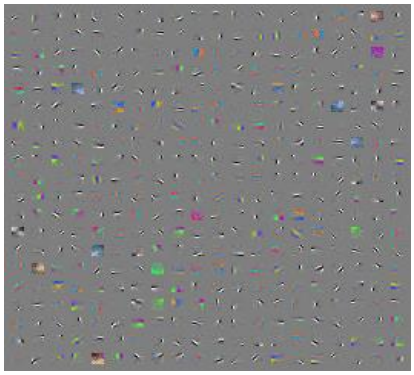


Figura 8: Visualización de los pesos del entrenamiento con CIFAR10

Como resultado del entrenamiento, inspeccionando el modelo final con los pesos obtenidos tras el mismo, somos capaces de visualizar un primer acercamiento para la identificación de las imágenes dadas transcurridos únicamente 4 épocas, y el índice de error de clasificación de cada imagen ha quedado reflejada en el 28%.

```
Parameter and initial learning rate summary:
        W: 0.1
        bias_vis: 0.1
        bias_hid: 0.1
        sigma_driver: 0.1
Compiling sgd_update...
Compiling sgd_update done. Time elapsed: 27.923031 seconds
compiling begin_record_entry...
compiling begin_record_entry done. Time elapsed: 0.051031 seconds
Monitored channels:
        bias_hid_max
        bias_hid_mean
        bias_hid_min
        bias_vis_max
        bias_vis_mean
        bias_vis_min
        h_max
        h_mean
        h_min
        learning_rate
        objective
        reconstruction_error
        total_seconds_last_epoch
```

Figura 9: Inicio del proceso de entrenamiento

```
Monitoring step:
        Epochs seen: 4
        Batches seen: 120000
        Examples seen: 600000
        bias_hid_max: -0.119526364371
        bias_hid_mean: -2.18671616478
        bias_hid_min: -3.24759610714
        bias_vis_max: 0.199195119615
        bias_vis_mean: -0.000157558304656
        bias_vis_min: -0.141461366806
        h_max: 0.466855162051
        h_mean: 0.0439254012314
        h_min: 0.00639914518957
        learning_rate: 0.1
        objective: 3.2779371318
        reconstruction_error: 28.4721905224
        total_seconds_last_epoch: 72.1537
        training_seconds_this_epoch: 57.800005
monitoring channel is objective
growing learning rate to 0.101000
Saving to cifar_grbm_smd.pkl...
Saving to cifar_grbm_smd.pkl done. Time elapsed: 0.121924 seconds
Saving to cifar_grbm_smd.pkl...
Saving to cifar_grbm_smd.pkl done. Time elapsed: 0.075991 seconds
```

Figura 10: Fin del proceso de entrenamiento tras 4 épocas

Capítulo 5
Estudio e implementación de redes neuronales usando Pylearn2

Una vez trabajados algunos ejemplos prácticos propuestos por dos de las tecnologías que nos permiten implementar técnicas de aprendizaje profundo, es hora de proceder a construir nuestras propias redes neuronales haciendo uso de los conocimientos adquiridos en los capítulos anteriores. De este modo, se elaborará un pequeño estudio final donde se analizará distintos tipos de modelos y técnicas utilizados en Deep Learning, detallando cuáles de estas técnicas o modelos resultan ser más favorables para el conjunto de datos con el que estemos trabajando, en función de distintos parámetros de la red.

Para este proyecto final haremos uso de la librería **Pylearn2**, dado que nos permite parametrizar la red de manera sencilla y mostrarnos los resultados del entrenamiento de la red en función de numerosas variables.

5.1 Conjunto de datos

Para entrenar nuestra propia red necesitaremos información o conjuntos de datos que tratar en la entrada de la misma. Este dataset procede de la plataforma *Kaggle*, una web sin ánimo de lucro que establece «retos» para sus usuarios relacionados con Big Data y Machine Learning, y en el que podemos encontrar cientos de datasets para su tratamiento.

El dataset de datos seleccionado, en formato **CSV**, estará formado por 418 entradas correspondientes a pasajeros que formaron parte del viaje del Titanic, disponiendo de cada uno de ellos información como su nombre, edad, sexo, puerto donde embarcaron, ticket de viaje o la clase en la que viajaban.

Con el conjunto de datos en nuestro poder, antes de proceder a construir y entrenar la red, debemos de parametrizar algunos de los datos de este dataset. Dado que algunas variables cualitativas no influyen en el entrenamiento de redes neuronales (*nombre, apellidos, ticket de viaje o puerto de embarque*), se procederá a eliminar algunos de estos datos (*nombre y apellidos*) y a sustituir el resto por variables numéricas que sí aporten más valor a nuestros datos.

5.2 Parametrización en Pylearn2

Una vez parametrizados nuestros datos de entrada, haremos lo propio con Pylearn2.

El tratamiento de un fichero CSV de entrada en Pylearn2 se hace usando una clase ***CSVDataset***, definida en el fichero *csv_dataset.py* de la propia librería. Nuestro cometido será sobrescribir esta clase –renombrándola como CSVDataset2– ajustándola a las características de nuestro propio fichero CSV.

Esta clase dispondrá de parámetros esenciales para el tratamiento de nuestro fichero CSV, tales como la ruta donde se encuentra nuestro fichero de datos, si vienen incluidas cabeceras en los datos, el carácter delimitador de cada dato o desde qué dato comenzar a leer.

Además, por defecto, *CSVDataset* trabaja con el método *loadtxt()* de la clase *Numpy* para leer cada dato del fichero CSV, de tal modo que es necesario especificar para cada cabecera de nuestro fichero de qué tipo de dato se trata y su longitud, haciendo uso de objetos *data types* de *Numpy*. Para evitar esto, tras intentar sin éxito parametrizar nuestro dataset con los *data types* adecuados, se decidió finalmente optar por una alternativa al método *loadtxt*(), el también método ***genfromtxt()***, cuya funcionalidad es similar al anterior pero donde no es necesario especificar explícitamente el tipo de dato que se está leyendo del dataset, así como su longitud, realizándolo este método de manera dinámica.

Con estas ligeras modificaciones, ya estamos listos para construir nuestra red neuronal y su fichero de entrenamiento.

5.3 Construcción de nuestra propia red neuronal

Una vez con los datos listos, es hora de sacar todo el potencial que nos provee Pylearn2 para construir y modelar nuestra primera red neuronal propia y establecer las pautas seleccionadas para su entrenamiento.

Pylearn2 dispone de dos sistemas para modelar una red neuronal y su entrenamiento asociado: bien estableciendo todos los mecanismos de la red que

queremos disponer directamente sobre un fichero Python o –de manera mucho menos costosa– construyendo un fichero **YAML** de configuración de la red, donde poder parametrizar con muchas menos líneas de código la red y nuestro propio sistema de entrenamiento. Optaremos por esta última opción.

El siguiente paso será decidir qué arquitectura vamos a construir para nuestra red, ajustándola a los datos que sabemos que tenemos que pasarle a la entrada de la misma. Como ya hemos visto en capítulos anteriores, los ficheros YAML nos permiten configurar una red y su entrenamiento estableciendo tres principales características: el **dataset** elegido, el **modelo** de red neuronal seleccionado y el **algoritmo** de entrenamiento aplicado a los datos según el modelo elegido.

El dataset ya lo tenemos decidido, ya que dado que trabajaremos con un fichero CSV tomaremos como dataset en este YAML un objeto de tipo ***CSV Dataset***. En él, detallaremos algunos parámetros como la ruta donde se encuentra nuestro fichero CSV, indicaremos que haremos una clasificación de los datos en lugar de una regresión, que nuestra primera fila de datos serán las cabeceras de los mismos y que nuestro valor delimitador de los datos serán las comas.

```
dataset: &train !obj:pylearn2.datasets.csv_dataset2.CSVDataset2 {
        path: 'test.csv',
        task: 'classification',
        expect_headers: True,
        delimiter: ',',
},
```

El siguiente paso será decidir el modelo arquitectónico de red a emplear. Para no complicarnos excesivamente, optaremos por el modelo más habitual de redes neuronales artificiales del que ya se ha hablado ampliamente con anterioridad: el **perceptrón multicapa**. Para este ejemplo, dispondremos de dos conjuntos de capas ocultas, de 100 capas de neuronas cada una, con una función de activación *Softmax*, una de las funciones más habituales usadas en problemas de clasificación de datos en redes neuronales.

Además, es importante establecer también el número de entradas que dispondrá la red, su número de salidas totales, el tamaño de cada capa oculta, el nombre de cada capa o el tamaño de cada *batch* de ejemplos a entrenar, entre otros.

```
model: !obj:pylearn2.models.mlp.MLP {
        batch_size: 1,
        nvis: 9,
        layers: [ !obj:pylearn2.models.mlp.RectifiedLinear {
                     layer_name: 'capa1',
                     dim: 100,
                     irange: .05,
                     use_bias: true,
                     max_col_norm: 1.9365,
                 },
```

```
                !obj:pylearn2.models.mlp.RectifiedLinear {
                    layer_name: 'capa2',
                    dim: 100,
                    irange: .05,
                    use_bias: true,
                    max_col_norm: 1.9365,
                },
                !obj:pylearn2.models.mlp.Softmax {
                    max_col_norm: 1.9365,
                    layer_name: 'salida',
                    n_classes: 1310,
                    istdev: .5
                }
               ],
    },
```

Finalmente, con nuestra red ya construida casi al completo, necesitamos indicarle qué tipo de algoritmo de entrenamiento vamos a realizar. Particularmente, hay dos grandes algoritmos empleados en este tipo de redes neuronales artificiales: el **SGD** (*Stochastic Gradient Descent*), que será el que utilicemos en esta red, y el **BGD** (*Mini-Batch Gradient Descent*).

Estos algoritmos se encargan de actualizar el conjunto de parámetros de la red de forma iterativa para minimizar una función de error. Mientras que en el BGD es necesario recorrer todas las muestras del conjunto de entrenamiento para hacer una sola actualización de un parámetro en una iteración en particular, en el SGD se utiliza una sola muestra de su conjunto de entrenamiento para hacer la actualización de un parámetro en una iteración concreta. Esto hace que el SGD sea computacionalmente más rápido y funcione mejor con datasets grandes.

Dentro de este apartado, se especificarán parámetros importantes como el *learning rate*, o índice de aprendizaje de la red, las variables de monitorización que nos darán información sobre cada iteración de la red o el número de épocas o iteraciones que se realizará en el entrenamiento.

```
algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
        batch_size: 1,
        learning_rate: .001,
        monitoring_dataset: {
            'train' : *train
        },
        termination_criterion: !obj:pylearn2.termination_criteria.And {
            criteria: [
                !obj:pylearn2.termination_criteria.EpochCounter {
                    max_epochs: 100
                },
            ]
        },
    },
```

Además de estas tres principales características, es posible incluir numerosos añadidos más a estos ficheros de entrenamiento, como la monitorización del

mejor resultado del entrenamiento de la red, el almacenamiento de los resultados del entrenamiento en un fichero externo para su posterior visualización o parametrizar características relacionadas con las funciones de coste.

5.4 Resultados del entrenamiento

Para ejecutar el entrenamiento de la red, basta con colocar el fichero de entrenamiento (***train.py***), que será común para todas las redes, conjuntamente con el fichero YAML construido anteriormente y ejecutar el fichero Python pasando como parámetro nuestro fichero YAML. Si no ha habido ningún tipo de error de sintaxis o ejecución, se entrenará nuestra red el número de épocas que le hayamos solicitado en el YAML, mostrando en cada iteración el valor que toman las distintas variables de monitorización.

Se pueden barajar varias opciones para poder interpretar los resultados de un entrenamiento de redes neuronales artificiales. Pylearn2 nos provee de hasta tres scripts que nos permiten conocer el estado final del entrenamiento:

1. ***plot_monitor.py***: Este fichero nos permite generar una serie de gráficas referentes a las variables de monitorización usadas en el entrenamiento de la red.

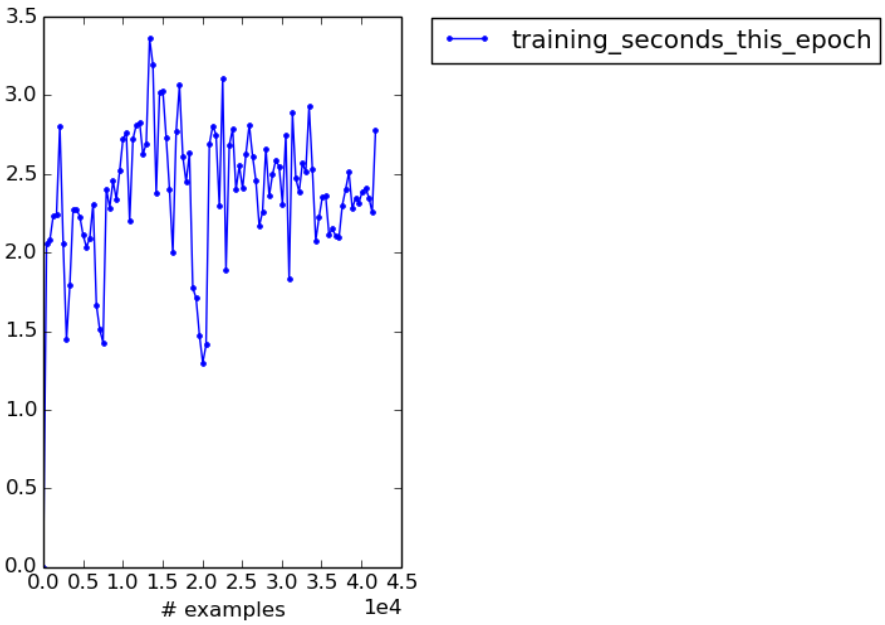


Figura 11: Gráfica generada con plot_monitor.py

2. ***print_monitor.py***: En este fichero podemos encontrar el valor final de las variables de monitorización una vez finalizado el entrenamiento, así como datos adicionales como el número de épocas entrenadas, el tiempo total del entrenamiento, el *learning rate* aplicado o el tiempo tardado en la última época.

```
epochs seen:  100
time trained:  583.190377951
learning_rate : 0.001
total_seconds_last_epoch : 5.473321
train_capa1_col_norms_max : 1.9365
train_capa1_col_norms_mean : 1.37206941632
train_capa1_col_norms_min : 0.0660263708571
train_capa1_row_norms_max : 14.2793500939
train_capa1_row_norms_mean : 2.47378517442
train_capa1_row_norms_min : 0.151065377383
train_capa2_col_norms_max : 1.9365
train_capa2_col_norms_mean : 1.89316653221
train_capa2_col_norms_min : 0.303996630509
train_capa2_max_x_max_u : 0.630642358344
train_capa2_max_x_mean_u : 0.0360200089199
train_capa2_max_x_min_u : 0.0
train_capa2_mean_x_max_u : 0.630642358344
train_capa2_mean_x_mean_u : 0.0360200089199
train_capa2_mean_x_min_u : 0.0
train_capa2_min_x_max_u : 0.630642358344
train_capa2_min_x_mean_u : 0.0360200089199
train_capa2_min_x_min_u : 0.0
train_capa2_range_x_max_u : 0.0
train_capa2_range_x_mean_u : 0.0
train_capa2_range_x_min_u : 0.0
train_capa2_row_norms_max : 8.64138485606
train_capa2_row_norms_mean : 1.23119182706
train_capa2_row_norms_min : 0.0625394282101
train_objective : 7.04516868857
train_salida_col_norms_max : 1.9365
train_salida_col_norms_mean : 1.93537597233
train_salida_col_norms_min : 1.82989682816
train_salida_max_max_class : 0.00160358296479
train_salida_mean_max_class : 0.00160358296479
train_salida_min_max_class : 0.00160358296479
train_salida_misclass : 0.997607655502
train_salida_nll : 7.04516868857
train_salida_row_norms_max : 7.27668694755
train_salida_row_norms_mean : 7.00343971194
train_salida_row_norms_min : 6.60771031808
training_seconds_this_epoch : 2.777684
```

3. ***summarize_model.py***: Este tercer fichero nos muestra información acerca del número de ejemplos y *batches* entrenados, el valor de los pesos y *biases* de las distintas capas de nuestro modelo.

```
capa1_W:(-1.9364999999402577,-0.15667975960821962,
0.055383367325278814)(9, 100)
abs(capa1_W):(5.2213172409292608e-08,0.16558451646960629,
1.9364999999402577)
```

```
capa1_W row norms:(0.15106537738270107, 2.4737851744236203,
14.279350093859149)
capa1_W col norms:(0.066026370857087777, 1.3720694163217337,
1.9365000000000003)
capa1_b: (-0.0037342685469777923, -0.00034371581074628074,
0.00079580419492335296)(100,)
abs(capa1_b): (0.0, 0.0003673022839535406, 0.0037342685469777923)
capa2_W: (-1.804490151648847, -0.054155046120049476,
1.8614267126998498)(100, 100)
abs(capa2_W): (3.363048785339789e-09, 0.072040728714409266,
1.8614267126998498)
capa2_W row norms:(0.06253942821006922, 1.2311918270648283,
8.6413848560617907)
capa2_W col norms:(0.30399663050895098, 1.8931665322149838,
1.9365000000000006)
capa2_b: (-0.010058148991971261, 0.034804458245025476,
0.63064235834396398)(100,)
abs(capa2_b): (1.7200868128147963e-05, 0.037235559594865054,
0.63064235834396398)
softmax_b: (-0.04401879110525192, 1.1812010233368793e-18,
0.074194946099136613)(1310,)
abs(softmax_b): (0.021949618678901467, 0.042002948038271516,
0.074194946099136613)
softmax_W: (-1.0665771333338185, -1.9247737420458122e-05,
1.3741943065569819)(100, 1310)
abs(softmax_W): (7.2585513628129128e-07, 0.15411537138433551,
1.3741943065569819)
softmax_W row norms:(6.6077103180784729, 7.0034397119379772,
7.2766869475525171)
softmax_W col norms:(1.829896828156788, 1.9353759723337318,
1.9365000000000008)
trained on 41800 examples
which corresponds to 41800 batches
Trained for 0.161997327209 hours
100 epochs
Training succeeded
```

4. ***show_weights.py***: En caso de disponer de imágenes como entradas de nuestra red -en lugar de ficheros de texto como es este caso- este script nos genera una imagen de los pesos una vez finalizado el entrenamiento.

5.5 Casos de estudio de redes neuronales artificiales

Disponiendo ya de un conjunto de datos que se entrenan sin inconvenientes usando Pylearn2 y teniendo a nuestra disposición una gran cantidad de modelos de redes neuronales y de diferentes algoritmos de entrenamiento que nos provee la propia librería, podemos estudiar e investigar qué parámetros y características funcionan mejor para nuestro conjunto de datos.

En el estudio realizado, se han llevado a cabo un total de **50 pruebas** o entrenamientos con **6 modelos** distintos de redes neuronales artificiales y los **2 algoritmos** de entrenamiento ya mencionados con anterioridad, y observamos los resultados del tiempo de entrenamiento variando distintos parámetros de las redes construidas.

Es importante señalar el gran papel desempeñado en Pylearn2 –e internamente con Theano– por la GPU para mejorar el rendimiento en el entrenamiento de las redes. Es posible realizar cálculos muy complejos a niveles de esta GPU, que en algunos casos permite mostrar resultados sustancialmente mejores que en la misma CPU, beneficiando de esta forma a los resultados del entrenamiento. El rendimiento en los tiempos variará dependiendo de los dispositivos.

Los modelos de redes que se han utilizado para este estudio son los Perceptrones Multicapa (**MLP**) con tres funciones de activación diferentes (***Gaussiana, Sigmoide y SoftMax***), la Máquina Restringida de Boltzmann (**RBM**), la Máquina Profunda de Boltzmann (**DBM**) y los **Autoencoders**.

Modelo – Tiempo

Modelos	Tiempo
MLP - Softmax	8,52 minutos
MLP - Gaussian	8,29 minutos
MLP - Sigmoid	11,02 minutos
RBM	1,38 minutos
DBM	7,56 minutos
Autoencoders	0,3 minutos

Una vez realizados los 50 entrenamientos con 60 redes diferentes, se ha calculado el tiempo de entrenamiento de distintos modelos de redes neuronales que dispusieron de 100 épocas, 100 capas ocultas y un índice de aprendizaje del 0.001 con el algoritmo SGD, descubriendo los tiempos arriba descritos.

Los Perceptrones Multicapa son los modelos que quizás menos convendría a nivel de tiempo de entrenamiento, pero sin embargo son uno de los modelos más estables existentes hoy en día debido a que son capaces de actuar como aproximadores universales de funciones entre un grupo de variables de entrada y salida, resultando ser herramientas flexibles y de propósito general.

Las máquinas de Boltzmann en la teoría resultan mucho más veloces, especialmente la restringida, aunque cuanto más complejo es el problema en la práctica presenta peores resultados. Algo similar ocurre con los Autoencoders que, si bien tienen unos tiempos de ejecución enormemente cortos, resultan más costosos cuanto más complejo es el problema, con el añadido de no ser tan flexible y estable como un MLP.

Modelo – Algoritmo

	Algoritmos (Media)	
Modelo	**SGD**	**BGD**
MLP - Softmax	14,33 minutos	55,47 minutos
MLP - Gaussian	28,84 minutos	45,88 minutos
MLP - Sigmoid	16,57 minutos	72,54 minutos
RBM	3,27 minutos	5,08 minutos
DBM	14,35 minutos	1,73 minutos
Autoencoders	0,98 minutos	3,22 minutos

Podemos vislumbrar también notables diferencias si comparamos el tiempo de ejecución de los entrenamientos según el algoritmo utilizado. El SGD es el algoritmo que mejor trabaja cuando disponemos de conjuntos de datos variablemente grandes en lugar de requerir un *batch* de muestras, como sucede con el BGD, lo que a nivel general produce un mayor esfuerzo computacional que se ve repercutido en el tiempo.

Modelo – Algoritmo – Número de Capas

		Número de Capas (Media)	
Modelos		**SGD**	**BGD**
MLP - Softmax	50 capas	6,33 minutos	37,8 minutos
	100 capas	18,32 minutos	73,15 minutos
MLP - Gaussian	50 capas	6,48 minutos	14,29 minutos
	100 capas	50,6 minutos	61,67 minutos
MLP - Sigmoid	50 capas	5,63 minutos	70,09 minutos
	100 capas	22,04 minutos	74,99 minutos
RBM	100 capas	4,19 minutos	6 minutos
	200 capas	1,9 minutos	3,23 minutos
DBM	100 capas	16,11 minutos	2,06 minutos
	200 capas	11,72 minutos	1,08 minutos
Autoencoders	100 capas	1,2 minutos	3,54 minutos
	200 capas	0,56 minutos	2,6 minutos

El número de capas ocultas en el modelo también es un factor a tener en cuenta a la hora de entrenar una red, algo que también se ve reflejado en el algoritmo de entrenamiento utilizado. En los modelos MLP se han utilizado capas ocultas de 50 y 100 capas, mostrando que a mayor número de capas el tiempo empleado para el entrenamiento es notablemente mayor, así como manteniéndose las mismas diferencias existentes anteriormente con los dos algoritmos utilizados. Ocurre de manera contraria en los modelos de Boltzmann y los Autoencoders, donde colocando un mayor número de capas se reduce el tiempo de ejecución.

Modelo – Número de Capas – Learning Rate

Modelos		Learning Rate	
		0.001	**0.1**
MLP - Softmax	50 capas	6,36 minutos	6,31 minutos
	100 capas	21,1 minutos	9,99 minutos
MLP - Gaussian	50 capas	6,48 minutos	6,21 minutos
	100 capas	40,41 minutos	71 minutos
MLP - Sigmoid	50 capas	7,07 minutos	4,19 minutos
	100 capas	25,75 minutos	10,94 minutos
RBM	100 capas	5,53 minutos	1,52 minutos
	200 capas	1,81 minutos	2 minutos
DBM	100 capas	20,52 minutos	7,28 minutos
	200 capas	11,49 minutos	11,95 minutos
Autoencoders	100 capas	1,69 minutos	0,22 minutos
	200 capas	0,32 minutos	0,8 minutos

Menos palpable es el estudio realizado teniendo en cuenta los cambios en el índice de aprendizaje, donde –de manera genérica– se puede afirmar que el tiempo de ejecución del entrenamiento se reduce cuando aumenta el número de capas ocultas de la arquitectura de la red y el propio learning rate.

Capítulo 6
Conclusiones y líneas futuras

Los grandes progresos en el campo de la Inteligencia Artificial en estas dos últimas décadas están resultando esenciales para el avance de la tecnología. Mecanismos como el ***Big Data***, el ***Cloud Computing***, el ***Internet of the Things*** o el propio ***Machine Learning*** están más que presentes en nuestros días para beneficio de cualquier persona, y suponen el gran futuro de este campo en años venideros.

Las grandes compañías tecnológicas se han hecho eco de ello, y en lo referente a *Machine Learning* y *Deep Learning* no se han hecho esperar. Google, Apple, IBM o Facebook se han gastado millones de dólares en estos años en la compra de *startups* y en la contratación de '*gurús'* de este campo para sus propias investigaciones.

Una de las grandes ventajas que tiene el Deep Learning es que es posible aplicarlo a funciones que se han considerado hasta hace poco muy específicas de los humanos, como la visión o el procesamiento natural del lenguaje. Para los consumidores esto se traduce en un software mejor, capaz de ordenar fotos, entender comandos hablados, traducir texto escritos en otros idiomas o que nuestro móvil reconozca nuestra cara.

El gran reto futuro del aprendizaje profundo parece encaminado al diseño de lo denominado como '***la máquina humana***', el primer ordenador capaz de aprender y entender como realmente lo haría un ser humano.

Existen varios científicos que ya han puesto fecha a este fenómeno. Vernor Vinge, el escritor que popularizó el término en los 80, sitúa este hito en 2030. Raymond Kurzweil, científico experto en Ciencias de la Computación e Inteligencia Artificial, marca este momento en 2045. Y Stuart Armstrong, miembro del *Future of Humanity Institute* de la universidad de Oxford, estableció una estimación sobre 2040.

Capítulo 7
Summary and Conclusions

The great progress in Artificial Intelligence in the last two decades are proving essential for advancement of technology. Mechanisms such as ***Big Data***, ***Cloud Computing***, the ***Internet of Things*** and ***Machine Learning*** are more present in our day for the benefit of any person, and represent the great future of this field in the coming years.

Big technology companies have rumored this, and in relation to Machine Learning and Deep Learning haven’t kept waiting. Companies such as Google, Apple, IBM and Facebook have spent millions of dollars over the years in buying *startups* and hiring 'gurus' in this field for their own research.

One of the great advantages of the Deep Learning is that it can be applied to functions that have been considered until recently very specific to humans, such as vision or natural language processing. For users this translates into a software better able to order photos, understand spoken commands, translate text written in other languages or for our mobile face recognition.

The great future challenge of deep learning sees the future to the design referred to as '***human machine***', the first computer able to learn and understand how a human would really be.

Several scientists have already set a date at where the machines will be able to have purely human behavior. Vernor Vinge, the writer who popularized the term in the 80s, puts this milestone in 2030. Raymond Kurzweil, scientific expert in Computer Science and Artificial Intelligence, marks the moment in 2045. And Stuart Armstrong, a member of the *Future of Humanity Institute* at Oxford University, established an estimate for 2040.

Capítulo 8
Presupuesto

El presupuesto de este Trabajo de Fin de grado se va a dividir en distintas partes: Coste de la mano de obra, coste de equipamiento y el coste de la conexión a internet.

No se contemplan las licencias de las aplicaciones utilizadas, puesto que todo el Trabajo de Fin de Grado ha sido desarrollado con software libre.

8.1 Coste de mano de obra

La realización de este Trabajo Fin de Grado ha durado 4 meses -con una media de 30 días al mes- lo que hacen 120 días, de los cuales 32 no han sido laborales, resultando finalmente un total de 88 días. La media de horas al día dedicada a la realización de este Trabajo Fin de Grado ha sido de 4 horas. Por tanto, asignando un salario de 20 € cada hora, el coste total estaría en:

Trabajador	Días	Salario	Coste total
Bryan García Navarro	88	80 € / día	7.040 €

8.2 Coste de equipamiento

El coste destinado a los equipos informáticos es el siguiente:

Equipo	Antigüedad	Valor
Portátil Acer Aspire E1-570 con Intel Core i3 2.50 GHz	3 años	300 €

8.3 Coste de conexión a internet

El coste referido a la conexión a internet utilizada es el siguiente:

Conexión a internet	Tarifa	Meses	Coste total
ADSL 20 MB	20 € / mes	4	80 €

8.4 Presupuesto total

El presupuesto estimado de este Trabajo de Fin de Grado es:

Concepto	Coste
Coste de mano de obra	7.040 €
Coste de equipamiento	300 €
Coste de conexión a internet	80 €
Coste de licencias de aplicaciones	0 €
Suma total	7.420 €
IGIC (7%)	519,40 €
Presupuesto total	**7.939,40**

Bibliografía

CURSOS

[1] ***Coursera: Las redes neuronales y el aprendizaje automático:*** *https://www.coursera.org/course/neuralnets*
[2] ***Coursera: Machine Learning by Andrew Ng:*** *https://class.coursera.org/ml-003/lecture*
[3] ***DeepLearning.net:*** *http://www.deeplearning.net/tutorial/contents.html*
[4] ***Universidad de Montreal: Very Brief Introduction to Machine Learning for AI:*** *http://www.iro.umontreal.ca/~pift6266/H10/notes/contents.html*

PUBLICACIONES ESCRITAS

[1] ***Artificial Neural Networks and Machine Learning****, de Alessandro Villa, Wlodzislaw Duch, Péter Érdi, Francesco Masulli y Günther Palm.*
[2] ***Deep Learning****, de Yoshua Bengio, Ian Goodfellow y Aaron Courville.*
[3] ***Las Redes Neuronales Artificiales. Fundamentos teóricos y aplicaciones prácticas****, de Raquel Flórez López y José Miguel Fernández Fernández.*

PUBLICACIONES WEB

[1] ***A Brief Overview of Deep Learning****: http://yyue.blogspot.com.es/2015/01/a-brief-overview-of-deep-learning.html*
[2] ***Classifying MNIST digits using Logistic Regression****: http://deeplearning.net/tutorial/logreg.html*
[3] ***Convolutional Networks****: http://www.iro.umontreal.ca/~bengioy/dlbook/convnets.html*
[4] ***Convolutional Neural Networks****: http://www.deeplearning.net/tutorial/lenet.html*
[5] ***Convolutional Neural Networks for Visual Recognition****: http://cs231n.github.io/convolutional-networks/*
[6] ***Datasets de Pylearn2****: http://deeplearning.net/software/pylearn2/library/datasets.html*
[7] ***El aprendizaje profundo para la identificación de sistemas no lineales****: http://www.ctrl.cinvestav.mx/~yuw/pdf/MaTesER.pdf*
[8] ***El perceptrón****: http://disi.unal.edu.co/~lctorress/RedNeu/LiRna004.pdf*
[9] ***Entrenamiento de redes neuronales basado en algoritmos evolutivos:*** *http://www.monografias.com/trabajos-pdf/entrenamiento-redes-neuronales-algoritmos-evolutivos/entrenamiento-redes-neuronales-algoritmos-evolutivos.pdf*
[10] ***Introduction to Gradient-Based Learning****: http://www.iro.umontreal.ca/~pift6266/H10/notes/gradient.html*
[11] ***Kaggle: Bag of Words Meets Bags of Popcorn****: https://www.kaggle.com/c/word2vec-nlp-tutorial*
[12] ***Machine Learning using Pylearn2****: https://blog.safaribooksonline.com/2014/02/10/pylearn2-regression-3rd-party-data/*
[13] ***Modelos de Pylearn2:*** *http://deeplearning.net/software/pylearn2/library/models.html*
[14] ***Multilayer Perceptron Tutorial****: http://deeplearning.net/tutorial/mlp.html*

[15] ***Perceptrón multicapa****:*
http://bibing.us.es/proyectos/abreproy/12166/fichero/Volumen+1+-+Memoria+descriptiva+del+proyecto%252F3+-+Perceptron+multicapa.pdf
[16] ***Pylearn2 Dev Documentation****: http://deeplearning.net/software/pylearn2*
[17] ***Pylearn2 Library Documentation****:*
http://deeplearning.net/software/pylearn2/library/index.html
[18] ***Pylearn2 Tutorial: Convolutional network****: http://nbviewer.ipython.org/github/lisa-lab/pylearn2/blob/master/pylearn2/scripts/tutorials/convolutional_network/convolutional_network.ipynb*
[19] ***¿Qué es y cómo funciona Deep Learning?****:*
https://rubenlopezg.wordpress.com/2014/05/07/que-es-y-como-funciona-deep-learning/
[20] ***Theano Tutorial****: http://nbviewer.ipython.org/github/craffel/theano-tutorial/blob/master/Theano%20Tutorial.ipynb*
[21] ***Visión artificial y Deep Learning: estado de la cuestión:***
https://srodriguezv.wordpress.com/2014/07/05/vision-artificial-y-deep-learning-estado-de-la-cuestion/
[22] ***YAML for Pylearn2****:*
http://deeplearning.net/software/pylearn2/yaml_tutorial/index.html#yaml-tutorial