UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Inteligencia computacional

Práctica de redes neuronales

Reconocimiento óptico de caracteres

MNIST

Curso 2016-2017

Máster en Ingeniería Informática

Ayhami Estévez Olivas

aythae@correo.ugr.es

Índice de contenido

Índice de figuras

Índice de tablas

Introducción

Las redes neuronales artificiales son un enfoque computacional basado en la forma en que el cerebro humano trabaja, mediante la conexión de un gran número de unidades de procesamiento llamadas neuronas [ANN]. Se han empleado con éxito en diversos problemas para los que algoritmos tradicionales no han dado buenos resultados. Destacan sus aplicaciones como aproximador universal de funciones y como clasificador (reconocimiento de patrones y secuencias). Esto último es lo que se plantea en esta práctica, el reconocimiento óptico de caracteres también conocido por sus siglas en inglés OCR.

En concreto se usará la base de datos MNIST de dígitos manuscritos la cual cuenta con un conjunto de entrenamiento de 60000 ejemplos y un conjunto de test de 10000. Estos ejemplos ya han sido preprocesados centrando las imágenes y haciéndolas de tamaño fijo. Es una base de datos estándar utilizada para el aprendizaje de las redes neuronales. Las imágenes de dicha base de datos son del siguiente estilo:

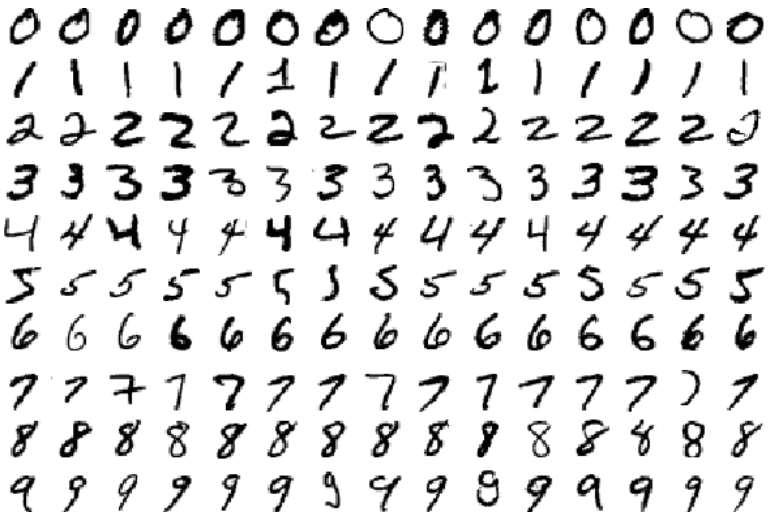

Ilustración 1: Ejemplo de imágenes de MNIST

Implementación

Me decidí a realizar una implementación en Java desde un momento inicial debido a mi dominio de este lenguaje y a que a pesar de que otros lenguajes puedan resultar más eficientes en el cálculo matricial como Mathlab tampoco está claro que sea una diferencia considerable al compararlo con un lenguaje compilado de propósito general como es Java. Además Java es uno de los lenguajes de programación más utilizados en la actualidad, con lo que ello implica (mucha información disponible en internet, elevado número de librerías, …).

Decidí también utilizar librerías ya que inicialmente realice algunas pruebas para implementar los algoritmos de redes neuronales por mi mismo pero sin buenos resultados y teniendo en cuenta que existe una parte competitiva importante en esta práctica me decanté por una librería que estaría mucho más optimizada que cualquier cosa que pudiera hacer yo. Adicionalmente me resultaba interesante aprender a utilizar una librería de redes neuronales que pudiera utilizar en otra ocasión para crear una red neuronal de manera rápida y poder aplicarla a un problema que me surja en el futuro.

A pesar del uso de Java en el mundo real no parece que se emplee demasiado para las redes neuronales y buscando librerías de software libre que pudiera utilizar **DeepLearning4J** (DL4J de ahora en adelante) es con diferencia la más conocida y utilizada. Otras librerías que investigue son **Theano** para Python, **Torch** para Lua, **Caffe** para C++ y **TensowFlow** para Python. Aunque algunas de estas librerías son más populares que DL4J apenas tengo experiencia en esos lenguajes excepto en C++ y la documentación ofrecida por DL4J es bastante completa así que inicie la implementación con esta librería. DL4J se define como un *"lenguaje de dominio específico para la configuración de redes neuronales profundas"*[DL4J01]

Como punto de partida empecé a consultar la documentación de DL4J y enseguida encontré una pequeña guía sobre una implementación de una red neuronal para la base de datos MNIST [DL4J02] gracias a la cual aprendí lo básico sobre como crear una red neuronal multicapa con esta librería. Sorprende lo potente que puede llegar a ser y como se pueden aplicar diversas técnicas de optimización del algoritmo backpropagation de una manera sencilla de modo que se puedan ver los resultados de

estas.

Empezando con esa implementación básica me dispuse a modificarlo para extraer los resultados en el formato necesario para la práctica. Para empezar necesitaba las predicciones de la red como una lista de enteros, cosa que no ofrece por si misma la librería, pero investigando sus estructuras internas encontré que la salida de la red era un vector de diez valores donde cada valor representa la salida de cada una de las neuronas softmax de la capa de salida (es decir representan probabilidades entre 0 y 1) con ello construí un array de enteros convirtiendo cada vector al entero con la mayor probabilidad, dicho array es el atributo `outputValuesInt`. Lo mismo ocurre con los las etiquetas, pero en este caso el vector de cada etiqueta son todo ceros y un uno en la posición del valor. Por lo que siguiendo el mismo mecanismo he creado un array de etiquetas con su representación en enteros, dicho array es el atributo `labelsInt`.

Necesitaba también la tasa de error sobre el conjunto de entrenamiento, eso lo conseguí realizando una evaluación sobre el conjunto de entrenamiento. También era necesario calcular el tiempo de entrenamiento por lo que establecí marcas -temporales usando el método de Java `System.currentTimeMillis()` que devuelve los milisegundos desde el 1 de enero de 1970.

Para guardar o cargar las redes neuronales entrenadas es necesario guardar lo que en DL4J se llama el modelo (una instancia de la clase `MultiLayerNetwork`), esto se realiza mediante los métodos `writeModel()` y `restoreMultiLayerNetwork()` de la clase `ModelSerializer` en los métodos `saveRNA()` y `loadRNA()`. A parte de las redes neuronales entrenadas es interesante guardar los resultados del entrenamiento, tiempos, tasas de error en evaluaciones y configuración de la red neuronal para poder extraer conclusiones sobre los entrenamientos posteriormente. Esto se realiza en el método saveResults() donde se guardan esos resultados en un fichero situado en data/resultsTS.txt donde TS es una marca temporal con hora, día, mes y año.

Además se ofrece la posibilidad de utilizar una interfaz de entrenamiento aportada por la librería que muestra diferentes gráficas de como va evolucionando el entrenamiento en función del tiempo. Estas gráficas incluyen la evolución de la función de coste con el tiempo, las desviaciones estándar de diversos parámetros, la distribución de pesos o bias por capas,... En la siguiente imagen se puede ver un ejemplo concreto.[DL4J03]

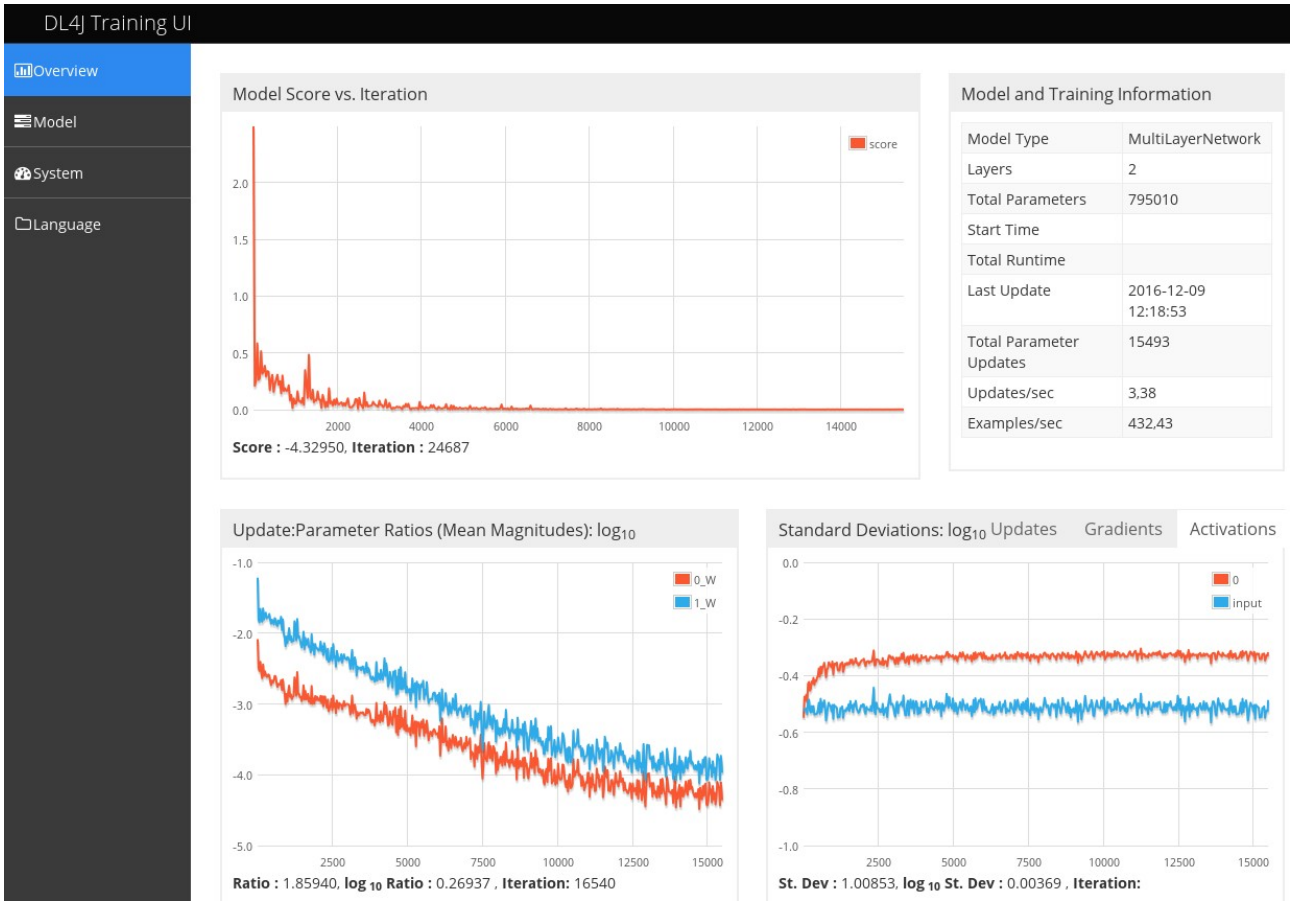

Ilustración 2: Ejemplo de DL4J Training UI

Primer modelo: Red neuronal multicapa

Con esto empecé a experimentar con una red neuronal básica con una capa oculta de 1000 neuronas y una capa de salida de 10 neuronas. Buscando en los apuntes de la asignatura encontré algunas recomendaciones para seleccionar los hiperparámetros adecuados para conseguir una implementación eficiente de backpropagation, siguiendo las recomendaciones Le Cun et al. dadas en los apuntes seleccione una función de activación para las neuronas de la capa oculta de tipo tanh, también seleccione una función de activación de tipo softmax para la capa de salida. La inicialización de pesos remendada se realiza de forma aleatoria siguiendo una distribución de media 0 y desviación estándar $\sigma = 1/\sqrt{m}$ donde m es el numero conexiones que llegan al nodo. Investigando las inicializaciones de pesos recomendadas en DL4J encontré la inicialización de pesos Xavier [XAVIER2010] la cual cumple con las anteriores premisas así que fue la seleccionada. Respecto a cuando actualizar los pesos me quede con la recomendación de DL4J que utiliza mini-batch con tamaño de lote de 128.

También siguiendo las recomendaciones de DL4J para acelerar la convergencia de los pesos utilicé momentos de Nesterov con un momentum de 0,9 y para evitar el sobre-aprendizaje regularización mediante weight decay penalizando los pesos grandes en función de sus valores al cuadrado (penalización L2) con un coeficiente de 0,0001. Como función de coste he seleccionado una función de entropía cruzada teniendo en cuenta que las neuronas de salida son softmax según se recomiendan en los apuntes, siguiendo las recomendaciones de DL4J he utilizado una función de coste de entropía cruzada multiclase [DL4J04].

Comparación de tasas de aprendizaje

Queda por seleccionar la tasa de aprendizaje, probablemente el hiperparámetro más importante, por ello he llevado acabo un estudio empírico con 3 tasas de aprendizaje según de recomienda en [DL4J04] comenzando con 0,1 siguiendo con 0,01 y en función de los resultados obtenidos seleccionando el intermedio 0,055 en la siguiente tabla y gráfica se recogen los resultados de dicho estudio:

Red / Época	**2**	**10**	**20**	**22**	**30**	**40**
Multicapa tanh tasa de aprendizaje 0,1	5,41	2,56	1,83	1,7	1,79	1,8
Multicapa tanh tasa de aprendizaje 0,01	8,06	4,66	3,08		2,58	2,29
Multicapa tanh tasa de aprendizaje 0,055	6,31	2,55	2,1		1,9	1,83

Tabla 1: Comparación de % de error sobre conjunto de test entre distintas tasas de aprendizaje con el paso de las épocas

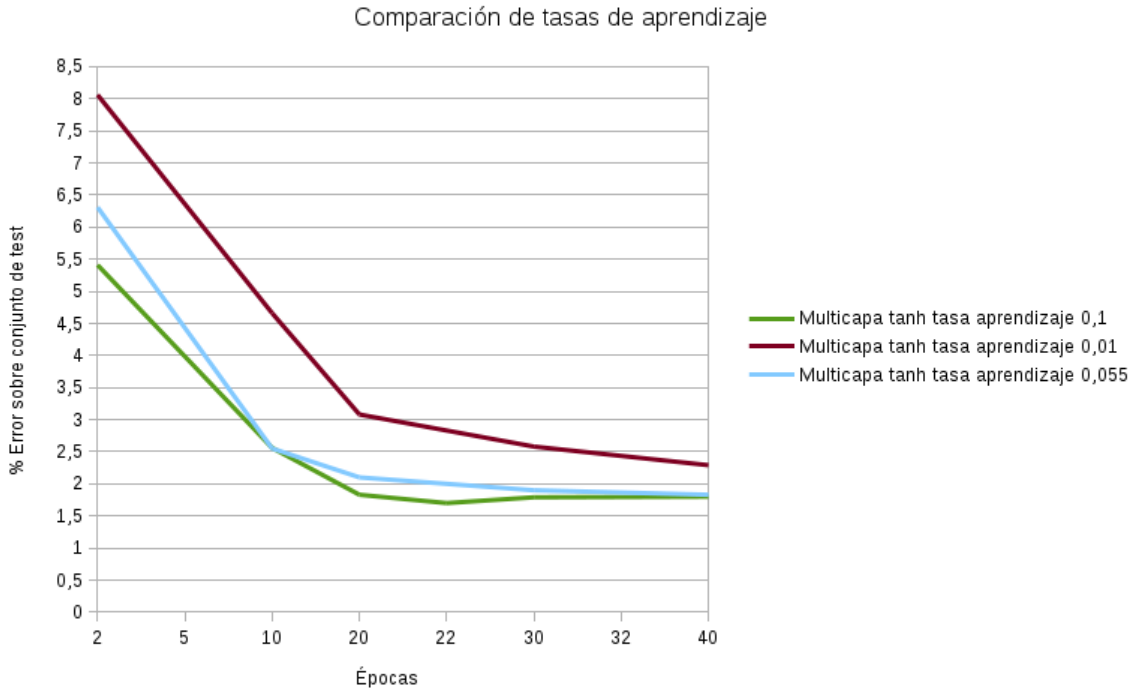

Ilustración 3: Comparación de % de error sobre conjunto de test entre distintas tasas de aprendizaje con el paso de las épocas

Como se puede apreciar una tasa de aprendizaje de 0,1 aprende muy rápido pero el sobre-aprendizaje es muy temprano (a partir de la época 22), con una tasa de aprendizaje de 0,01 se aprende demasiado lento llegando a finalizar el entrenamiento de forma inesperada por quedarse sin memoria mi ordenador. Con una tasa de 0.055 se observan mejores resultados, pero aún se podría aprender algo más rápido ya que no llega a sobreaprender.

Comparación de funciones de activación

A continuación me dispuse a comprobar si la función de activación de la capa oculta era la correcta por lo que con una tasa de aprendizaje fija de 0,1 me dispuse a comparar entre tangente hiperbólica, sigmoide y unidad lineal rectificada (ReLU). Estos son los resultados:

Red / Época	2	10	20	22	30	40
Multicapa tanh tasa de aprendizaje 0,1	5,41	2,56	1,83	1,7	1,79	1,8
Multicapa sigmoide tasa de aprendizaje 0,1	6,18	2,79	2,23		1,99	1,89
Multicapa ReLU tasa de aprendizaje 0,1	3,74	1,66	1,51		1,47	1,49

Tabla 2: Comparación de % error sobre conjunto test de distintas funciones de activación de la capa oculta con el paso de las épocas

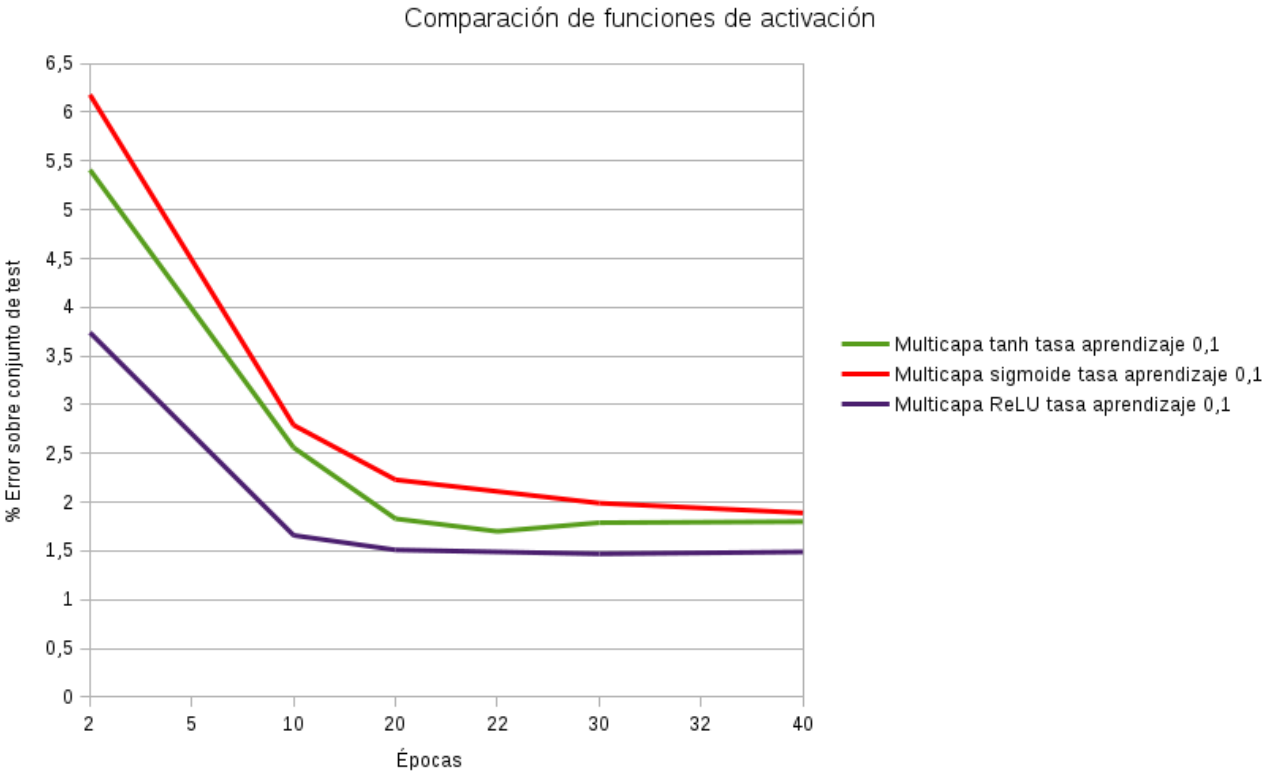

Ilustración 4: Comparación de % error sobre conjunto test de distintas funciones de activación de la capa oculta con el paso de las épocas

Se puede apreciar como en iguales condiciones la función sigmoide es la que tarda más en converger con mucha diferencia (llegando con 40 épocas a valores de error peores que las otras 2 funciones con 20). Respecto a las otras 2 funciones parece que ReLU se comporta mejor aprendiendo de manera más rápida, siendo más resistente al sobre-aprendizaje y alcanzando los mejores valores de error sobre el conjunto de test

Parece por ello lógico usar ReLU como función de activación de la capa oculta con una tasa de aprendizaje entre 0,1 y 0,055. Por ello probé con una tasa de aprendizaje de 0,0775 (la mitad del intervalo). Esto dio como resultado un mínimo de **1,45% de error** sobre el conjunto de prueba, lo cual era algo inferior a lo obtenido previamente con

ReLU y tasa de aprendizaje de 0,1 pero no lo suficiente. Este era un resultado alcanzable con una red multicapa implementada sin librerías y ya que he renunciado a 2 puntos de implementación me quería asegurar la parte competitiva.

Segundo modelo: Red neuronal convolutiva

Teniendo en cuenta que usaba una librería quería sacarle el mejor resultado posible así que tome como referencia a LeNet y me puse a implementar una red convolutiva basada en esta usando un ejemplo dado por DL4J [DL4J05]. Dicho ejemplo alcanza un 2,42% de error sobre el conjunto de test en una sola época de entrenamiento lo cual es muy prometedor si lo comparamos con los resultados de las redes neuronales multicapa. Viendo con los apuntes me percaté que solo tenia una capa oculta tras las capas convolutivas así que sustituí esa capa oculta de 500 neuronas ReLU por una capa de 120 neuronas ReLU y otra de 84 neuronas tanh. En la siguiente imagen se puede observar la topología de dicha red.

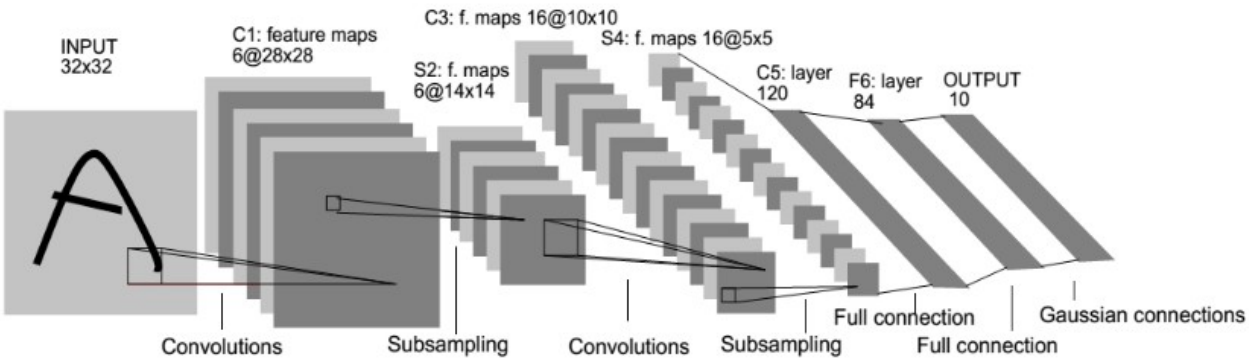

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Ilustración 5: Topología LeNet-5

Tras ejecutar el ejemplo básico me di cuenta del incremento de tiempo de entrenamiento que suponía esta red neuronal así que sabiendo que DL4J tiene compatibilidad con CUDA “una arquitectura de calculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU” [CUDA] y que dispongo de una tarjeta gráfica NVIDIA me dispuse a instalar los drivers privativos de NVIDIA en mi sistema Debian 8.6. Desgraciadamente se produjo un problema durante la instalación e intentando

desinstalar los drivers se empezó a desinstalar el S.O. entero por lo que me vi obligado formatear y reinstalarlo perdiendo un par de días muy valiosos en el proceso.

Como otros parámetros de la red convolutiva también usa inicialización de pesos Xavier, backpropagation por mini-batch con un tamaño de batch de 64. Para acelerar la convergencia busque en los apuntes otros métodos de optimización además de los momentos de Nesterov y viendo las animaciones de los diferentes métodos decidí probar el que parece más rápido, ADADelta según [DL4J06] este método de optimización de gradiente descendiente estocástico se basa en el cambio adaptativo de la tasa de aprendizaje utilizando una disminución exponencial de la media de los gradientes para modificar los momentos de estos, en concreto elegí un épsilon de $1 * 10^{-6}$ como recomienda por defecto DL4J. Usa regularización mediante weight decay con penalización L2 y coeficiente $5 * 10^{-4}$. Para la función de coste a optimizar para reducir el error probé diversas como el error cuadrático medio (MSE), la entropía cruzada multiclase y la función de verosimilitud logística negativa (Negative Log Likelihood) siendo esta última la que mejor me ha funcionado.

Como tasa de aprendizaje probé con varias entre 0,01 (la recomendada por DL4J) y 0,02 pero durante escasas épocas por lo que no tengo datos fiables, vi que lo mejor me funcionaba era algo intermedio así que me quede con 0,013.

Entrenando la red encontré un mínimo a las 22 épocas con un **0,82% de error**, mucho menor que lo obtenido previamente con redes multicapa normales.

Comparación de redes convolutivas con distinto número de capas ocultas

Se me ocurrió entonces probar a insertar otra capa oculta de 48 neuronas tipo sigmoide justo antes de la capa de salida con el objetivo de que al ser estás las 3 funciones más utilizadas con una capa de cada se lograrían compensar los problemas que pudiera haber entre ellas. Con esos mismos parámetros realice una comparación como las anteriores entre ambos modelos.

Red / Época	**2**	**5**	**10**	**20**	**22**	**30**	**40**
Convolutiva 3 capas ocultas	1,78	0,98	1,08	0,87	0,85		0,84
Convolutiva 2 capas ocultas	1,49		0,94	0,83	0,82	0,83	0,84

Tabla 3: Comparación de % error sobre conjunto test entre redes convolutivas con distinto número de capas ocultas con el paso de las épocas

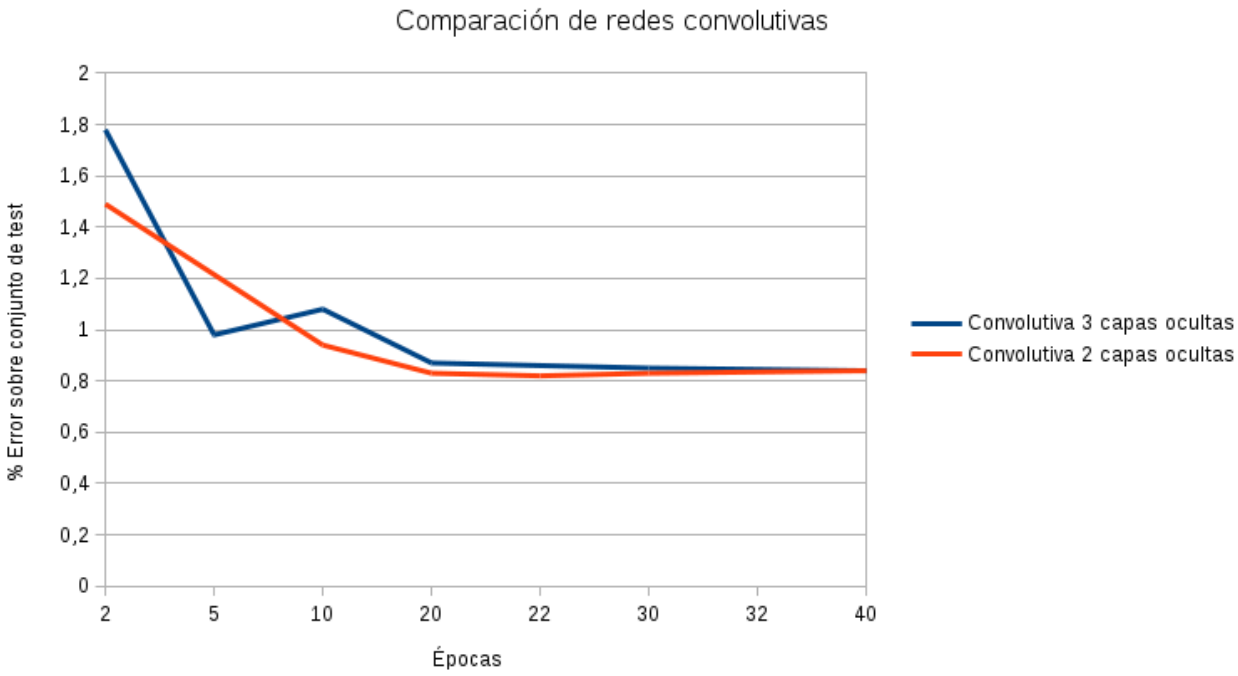

Ilustración 6: Comparación de % error sobre conjunto test entre redes convolutivas con distinto número de capas ocultas con el paso de las épocas

Como se aprecia la red de 2 capas alcanza un resultado absoluto mejor (por poco) pero sobretodo realiza menos oscilaciones extrañas y al tener menos neuronas se reduce su tiempo de entrenamiento y evaluación por lo que es el modelo final elegido.

Análisis de resultados

Comparación mejor red multicapa con red convolutiva

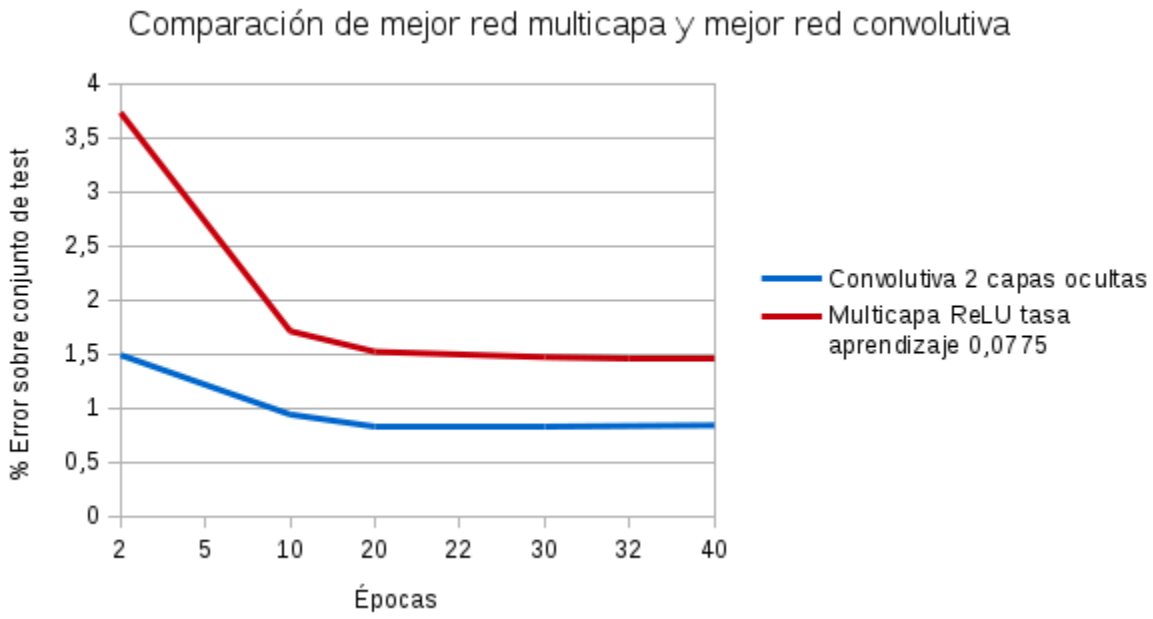

Ilustración 7: Comparación mejor red multicapa y mejor red convolutiva

Como se puede ver la red convolutiva es mucho mejor prediciendo que las redes multicapa por lo que merece la pena utilizar capas convolutivas para una red neuronal de reconocimiento de imágenes.

Comparación entre todas las implementaciones

A modo de conclusión aporto un gráfico en el que se pueden ver enfrentadas todas las implementaciones de redes neuronales aquí analizadas aunque al ser demasiadas lineas juntas algunas se superponen y no se aprecian correctamente, para eso se aportan los gráficos detallados previos.

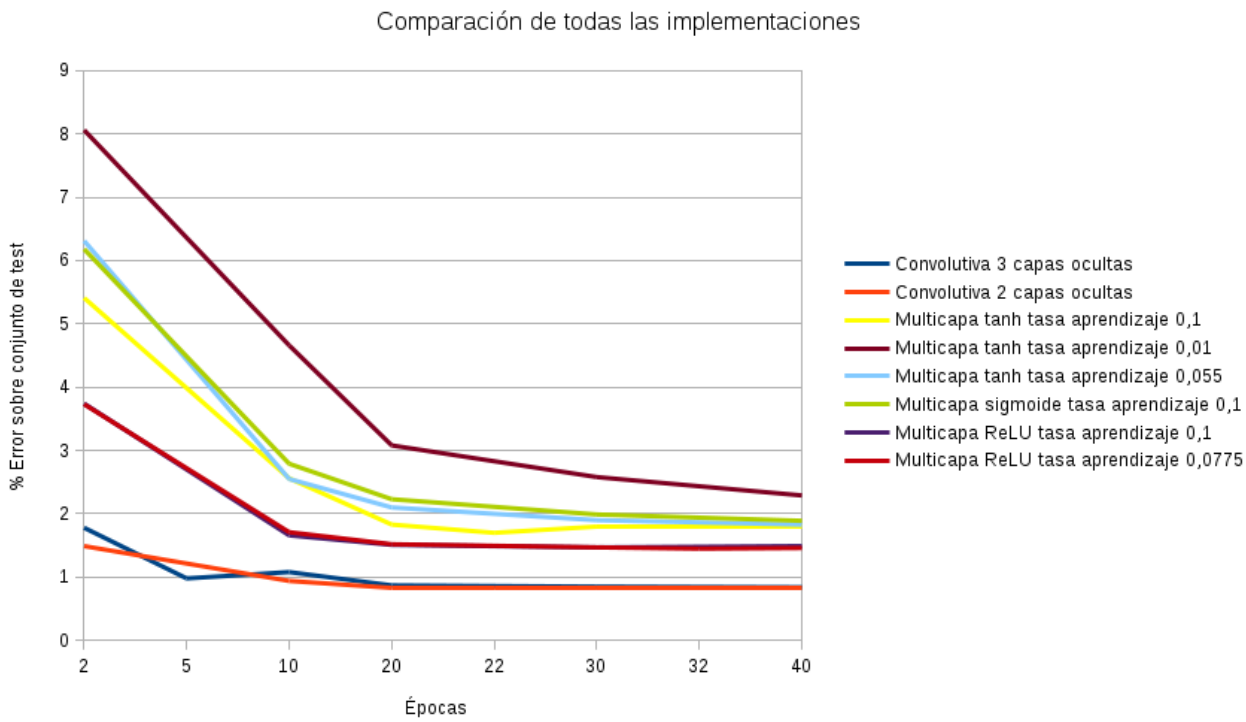

Ilustración 8: Comparación entre todas las implementaciones de redes neuronales

Conclusiones

De este trabajo me llevo la experiencia trabajando con redes neuronales, ya las había visto en teoría y entendía los conceptos pero dar el paso de la teoría a la práctica frecuentemente no resulta sencillo.

Me hubiera gustado poder implementar algo por mi mismo pero la falta de tiempo unida a la parte de la nota competitiva y a que pensé que sería mas útil aprender a usar alguna librería que pudiera usar en el futuro para problemas similares me motivaron a usar la librería DL4J. Habría sido interesante realizar una comparativa entre distintas librerías en términos de rendimiento, resultados, comparación de realizar las mismas tareas con unas y con otras, etc.

Ha resultado una experiencia interesante ver como se atajan los problemas con redes neuronales y me ha llamado la atención de la selección de hiperparámetros de las redes neuronales, realmente es más un arte que una ciencia, solo existen recomendaciones heurísticas. La habilidad de crear redes neuronales adecuadas para distintos problemas se basa únicamente en la experiencia y en las “corazonadas” de los desarrolladores.

Bibliografía

ANN: Wikipedia, the Free Encyclopedia., Artificial Neural Network, 2016, https://en.wikipedia.org/wiki/Artificial_neural_network

DL4J01: DeepLearning4J, Quick Start Guide, consultado 12/2016, https://deeplearning4j.org/quickstart

DL4J02: DeepLearning4J, MNIST for Beginners , consultado 12/2016, https://deeplearning4j.org/mnist-for-beginners

DL4J03: DeepLearning4J, Visualize, Monitor and Debug Network Learning, consultado 12/2016, https://deeplearning4j.org/visualization.html

XAVIER2010: Xavier Glorot y Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks, 2010, http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

DL4J04: DeepLearning4J, Troubleshooting Neural Net Training, consultado 12/2016, https://deeplearning4j.org/troubleshootingneuralnets#troubleshooting-neural-net-training

DL4J05: DeepLearning4J, LenetMnistExample, consultado 12/2016, https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/convolution/LenetMnistExample.java

CUDA: NVIDIA, ¿Qué es CUDA?, consultado 12/2016, http://www.nvidia.es/object/cuda-parallel-computing-es.html

DL4J06: DeepLearning4J, Deeplearning4j Updaters Explained, consultado 12/2016, https://deeplearning4j.org/updater