

UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

HERRAMIENTAS MODERNAS EN REDES NEURONALES: LA LIBRERÍA KERAS

Autor: Carlos Antona Cortés

Tutor: José Ramón Dorronsoro Ibero

Enero 2017



HERRAMIENTAS MODERNAS EN REDES NEURONALES: LA LIBRERÍA KERAS

Autor: Carlos Antona Cortés
Tutor: José Ramón Dorronsoro Ibero

Codigo: 1617_032_CO

Dpto. de Neurocomputación
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Enero 2017

El mundo de las redes neuronales está en auge. Poder simular el cerebro humano en un ordenador, parece ser uno de los hitos más prometedores de la informática. Es cierto que este hito todavía no se ha conseguido, pero mediante algoritmos de machine learning, ya es posible entrenar máquinas para que aprendan de forma parecida a como lo haría nuestro cerebro. El objetivo de este Trabajo de Fin de Grado es poner en práctica estos algoritmos utilizando la librería Keras.

En primer lugar, se hará una breve introducción al mundo de las redes neuronales. Se empezará por lo más básico, explicando que es una red neuronal y definiendo las partes más importantes de su arquitectura. Una vez entendidos los conceptos básicos, se describirán los 3 tipos de redes neuronales más extendidas actualmente debido a sus buenos resultados: Perceptrón multicapa, redes convolucionales, y redes LSTM.

En segundo lugar, se describirá Keras, una librería python de deep learning con la cual podremos diseñar nuestros propios modelos de redes neuronales. Se detallarán las clases y funciones más importantes, así como la gran cantidad de posibilidades que nos ofrece.

Por último, se aplicarán todos los conocimientos descritos anteriormente para diseñar cuatro tipos de redes neuronales que resolverán dos tipos de problemas distintos. En cuanto a problemas de clasificación de datos, veremos como clasificar distintas casas según su precio a partir de cualidades que se han recogido previamente, y veremos como clasificar imágenes en las que aparecen números, para determinar que número es el que se representa en la imagen. Por otro lado, se llevarán a cabo dos problemas de predicción de datos. Mediante el primero, veremos como una máquina es capaz de predecir si una persona tendrá diabetes a partir de sus datos médicos, y con el segundo, veremos como ésta es capaz de generar textos una vez haya sido entrenada con ellos.

Palabras Clave

Redes neuronales, Keras, Tensorflow, perceptrón, redes convolucionales, redes LSTM, diabetes, housing, mnist, generador de texto.

Abstract

The world of neural networks is growing. Being able to simulate the human brain in a computer seems to be one of the most promising milestones in computing. It is true that it has not been achieved, but using machine learning algorithms, it is already possible to train machines to learn in a similar way as our brain. The objective of this End-of-Grade Work is to implement these algorithms using the Keras library.

First of all, a brief introduction to the world of neural networks will be made. It will start with the basics, explaining what is a red neuron and defining the most important parts of its architecture. After understanding the basics, I will describe the three types of neural networks more rife due to their results: Multilayer Perceptron, Convolutional Networks, and LSTM Networks.

Secondly, I will describe Keras, a deep learning python library with which it is able to design our own neural network models. I will detail the classes and the most important functions, as well as the great amount of possibilities that this library offers us.

Finally, all previous knowledge is applied to design four types of neural networks that will solve two different types of problems. As for the problems of data classification, we will see how to classify several houses according to their price from qualities that have been previously collected, and we will see how to classify images of handwritten numbers, to determine which number is represented in the picture. On the other hand, data prediction problems will be addressed. Through the first example, we will be able to predict if a person will have diabetes or not analyzing their medical data, and with the second, we will see how a machine is able to generate texts once he has been trained with them.

Keywords

Neural network, Keras, Tensorflow, perceptron, convolutional neural network, LSTM network, diabetes, housing, mnist, text generator.

Agradecimientos

En primer lugar quiero dar las gracias a mi tutor, José Dorronsoro, por darme la oportunidad de adentrarme en el mundo de las redes neuronales. En un mundo tan extenso, relativamente novedoso, y en constante desarrollo, me habría sido mucho más difícil cumplir con mis objetivos si no fuese gracias a su constante ayuda.

En segundo lugar quiero dar las gracias a mis compañeros de clase: Sergio, David, Javi, Galan, Gus, Alfonso... Durante estos 4 años podría decir que han sido las personas con las que más momentos he compartido, y por tanto, no pueden faltar en mis agradecimientos. Incontables son los días que hemos pasado en las salas de estudio de la biblioteca, las semanas entrando a la universidad cuando aún no había amanecido y saliendo cuando el sol ya se había ido, los periodos de exámenes y prácticas en los que hemos tenido que dormir menos de lo que nos hubiese gustado... En fin, gracias por hacerme el camino más ameno.

Una mención especial a mis amigos de Vallecas, los de toda la vida: Alvaro, Alberto, Ivan, Benze... En estos cuatro años hemos cambiado mucho, y gracias a las vacaciones, el canal de música, las noches de viernes en el bar... he conseguido tomarme ese respiro todas las semanas para abstraerme lo necesario de los estudios, y seguir disfrutando de la vida.

Por último, y no por ellos menos importante, quiero ofrecer una mención especial a mi familia. No me cabe duda de que gracias a ellos y a su educación, estoy donde estoy. Ellos han sido los que verdaderamente han tenido que aguantarme estos 19 años de estudio. Semanas de nervios, luces encendidas a altas horas de la noche, apoyos y consejos cuanto más lo necesitaba... No tendría páginas suficientes en este TFG para agradecerles todo lo que han hecho por mí.

Índice general

Indice de Figuras	IX
Indice de Tablas	x
1. Introducción	1
1.1. ¿Qué es deep learning?	1
1.2. ¿Qué podemos hacer mediante deep learning?	1
2. Estado del arte: Redes neuronales	4
2.1. Redes neuronales	4
2.1.1. Historia de las redes neuronales	4
2.1.2. La neurona biológica	5
2.1.3. La neurona artificial	6
2.1.4. Funciones de las neuronas artificiales	7
2.1.5. Tipos de aprendizaje en redes neuronales	9
2.2. Perceptrón	9
2.3. Perceptron multicapa	12
2.3.1. Propagación	12
2.3.2. Consideraciones de diseño	13
2.3.3. Algoritmo de Retropropagación o Backpropagation	14
2.3.4. Regla delta generalizada	14
2.3.5. Tasa de aprendizaje	15
2.3.6. Sobreajuste y early stopping	15
2.4. Redes neuronales convolucionales	15
2.4.1. Estructura	16

2.5. Redes neuronales recurrentes	17
2.5.1. Redes LSTM	19
3. La librería Keras	21
3.1. Introducción	21
3.2. Instalación	21
3.3. Modelos	22
3.3.1. Modelo secuencial	22
3.3.2. Clase Model de la API	22
3.4. Capas	22
3.4.1. Funciones básicas de las capas	22
3.4.2. Capas convolucionales	23
3.4.3. Capas pooling	23
3.4.4. Capas recurrentes	24
3.5. Inicializadores	24
3.6. Activaciones	24
3.7. Funciones de pérdida	24
3.7.1. Regularizadores	25
3.8. Optimizadores	25
3.9. Callbacks	25
3.10. Datasets	26
3.11. Preprocesado	26
3.12. Visualización	26
3.13. Wrapper para Scikit-Learn API	26
4. Keras: Diseño y desarrollo de modelos	28
4.1. Problema de diabetes - Regresión logística	28
4.1.1. Cargando datos	28
4.1.2. Definiendo el modelo	29
4.1.3. Compilando el modelo	29
4.1.4. Ajustando el modelo	29
4.1.5. Evaluando el modelo	30
4.1.6. Analizando datos	30
4.2. Problema Boston Housing - Regresión lineal	31
4.2.1. Descripción del problema	31
4.2.2. Desarrollo de la red neuronal	31
4.2.3. Mejorando los resultados	32

4.3.	MNIST - Perceptrón multicapa vs Red convolucional	34
4.3.1.	Cargando el dataset	34
4.3.2.	Resultados	36
4.4.	Generación de textos - Red LSTM	37
4.4.1.	Aprendiendo secuencias de caracteres	37
4.4.2.	Generando secuencias de caracteres	39
4.4.3.	Analizando resultados	40
5.	Conclusiones y líneas futuras	42
	Glosario de acrónimos	43
	Bibliografía	44
A.	Códigos fuente	45
A.1.	4.1. Problema de diabetes - Regresión logística	45
A.2.	4.2. Problema Boston Housing - Regresión lineal	45
A.2.1.	Modelo estándar	45
A.2.2.	Modelo con datos de entrada normalizados	46
A.2.3.	Modelo más profundo	46
A.2.4.	Modelo más ancho	47
A.3.	4.3 MNIST - Perceptrón multicapa vs Red convolucional	48
A.3.1.	Perceptrón multicapa	48
A.3.2.	Red convolucional	48
A.4.	4.4 Generación de textos - Red LSTM	49
A.4.1.	Entrenamiento de la red	49
A.4.2.	Generacion de texto	50

Indice de Figuras

2.1. Historia de las redes neuronales <i>Imagen extraída de SlideShare</i>	5
2.2. Estructura de una neurona <i>Imagen extraída de Wikipedia</i>	5
2.3. Sinapsis <i>Imagen extraída de Wikipedia</i>	6
2.4. Esquema de una neurona artificial <i>Imagen extraída de ibiblio.org</i>	6
2.5. Función identidad	8
2.6. Función escalón binario	8
2.7. Función logística	8
2.8. Función tangencial	8
2.9. Función rectificadora	9
2.10. Funcion de activación softplus	9
2.11. Perceptron simple <i>Imagen extraída de GitHub</i>	10
2.12. Separacion lineal del perceptrón. <i>Imagen extraída de Wikipedia</i>	11
2.13. Separacion lineal de la funcion AND. <i>Imagen extraída de Wikipedia</i>	12
2.14. Separacion lineal de la funcion OR. <i>Imagen extraída de Wikipedia</i>	12
2.15. Perceptrón multicapa. <i>Imagen extraída de Wikipedia</i>	12
2.16. Tabla comparativa del perceptrón segun sus capas. <i>Imagen extraída de GitHub</i> .	13
2.17. Funcionamiento de capa convolucional. <i>Imagen extraída de deeplearning.net</i> . . .	16
2.18. Max-pooling aplicado a una imagen <i>Imagen extraída de Wikipedia</i>	17
2.19. Ejemplo básico de red neuronal recurrente. <i>Imagen extraída de GitHub</i>	18
2.20. Red neuronal recurrente desenrollada. <i>Imagen extraída de GitHub</i>	18
2.21. Red neuronal recurrente desenrollada. <i>Imagen extraída de GitHub</i>	19
2.22. Estructura del modulo repetidor. <i>Imagen extraída de GitHub</i>	19
2.23. Notación del modulo repetidor. <i>Imagen extraída de GitHub</i>	19
4.1. Patrones dataset MNIST	34

Indice de Tablas

1.1. ¿Qué es deep learning?

Deep Learning (aprendizaje profundo) es una rama de machine learning (aprendizaje automático) formada por un conjunto de algoritmos que intentan modelar abstracciones de alto nivel en datos, usando grafos profundos con múltiples capas de procesamiento. Estas capas de procesamiento pueden estar compuestas por transformaciones tanto lineales como no lineales.

Antes de entrar de lleno en este mundo tan complejo, relativamente moderno, y en constante cambio, no estaría de más entender que significa que una maquina pueda aprender. Muchas veces en el mundo de la informática, nos encontramos ante problemas que son difíciles de programar. Imaginemos el caso de una aplicación de reconocimiento de textos escritos a mano. Es difícil imaginarse un conjunto de reglas que permitan distinguir unos caracteres de otros, ya que dependiendo de la forma de escritura de cada individuo, los caracteres cambian relativamente su forma. Por ejemplo, podría ser bastante difícil distinguir el número 0 de la letra o. Junto a este ejemplo, también tenemos muchos tipos de problemas en los que cuesta imaginarse un conjunto de reglas para su resolución: reconocimiento de objetos, discursos... Aquí es donde entra en juego el Machine Learning en general, y el Deep Learning en particular.

Si en vez de intentar escribir un programa que resuelva nuestro complejo problema, escribimos un algoritmo que permita a nuestro ordenador estudiar miles de ejemplos con sus soluciones, para luego aplicar la experiencia a la hora de resolver nuevos casos, estaremos aplicando Machine Learning.

El Deep Learning, que tuvo sus inicios en los años 80s, no es más que un paradigma de implementación de Machine Learning. Debido a las recientes investigaciones sobre el mundo de la Inteligencia Artificial, esta tecnología se encuentra ahora mismo en constante desarrollo, ya que se ha demostrado que es la forma más eficiente de hacer con un computador una simulación de lo que nuestro cerebro puede hacer.

1.2. ¿Qué podemos hacer mediante deep learning?

Como ya se ha comentado, las técnicas de deep learning nos han abierto una puerta a muchos tipos de problemas que hasta hace relativamente poco eran difícilmente programables mediante un ordenador. De entrada, podemos resolver problemas de clasificación y predicción de una manera mucho más eficiente y precisa que con las técnicas que conocimos hasta entonces. Algunos

ejemplos concretos podrían ser:

1. Colorear imágenes en blanco y negro.

Tradicionalmente, esta tarea ha sido llevada a cabo por el ser humano de manera manual, hasta que mediante el uso de deep learning, se han utilizado los objetos y contextos de las propias imágenes para ser coloreadas. Para ello, se necesitan grandes redes neuronales convolucionales con capas supervisadas, que recrean la imagen coloreada de la misma manera que lo haría un ser humano.

2. Añadir sonido a películas mudas.

Pongamos el ejemplo de querer recrear el sonido que hace un palo al golpear con una superficie. Si entrenamos el sistema utilizando una gran cantidad de vídeos donde se muestra el sonido que hace un palo al ser golpeado contra diferentes superficies, nuestra red neuronal asociará los frames del vídeo mudo con la información ya aprendida, y seleccionará el sonido que mejor se adapte a la escena.

3. Traducciones automáticas.

Pese a que la traducción de palabras, frases o textos lleva siendo posible desde hace muchos años, mediante la utilización de redes neuronales se han alcanzado resultados mucho mejores sobre todo en dos áreas: traducción de texto, y traducción de imágenes. En el caso de los textos por ejemplo, se ha pasado de traducir palabras sueltas, a analizar y entender la gramática y la conexión entre palabras, haciendo deducciones mucho más precisas del significado global de la frase. Para ello, se usan redes LSTM. Para el caso de traducción de textos en imágenes se utilizan redes convoluciones, ya que son capaces de identificar letras, con ellas formar palabras, y a su vez formar un texto. En muchos contextos a esto se le llama traducción visual.

4. Clasificación de objetos en fotografías.

Esta funcionalidad consiste en detectar y clasificar uno o más objetos de la escena de una fotografía. Eso se consigue entrenando grandes redes convolucionales mediante imágenes aisladas de objetos conocidos.

A partir de aquí, el siguiente punto es crear una palabra o frase que describa el contenido de la imagen. En 2014, hubo un "boom" de algoritmos que alcanzaron resultados impresionantes a la hora de resolver este problema. La mayoría de ellos usaban redes LSTM para convertir las etiquetas que se generan al detectar objetos en una imagen, en frases con sentido.

5. Generación de textos.

Esta tarea consiste en, dado una recopilación de textos, generar nuevos textos a partir de una palabra o una frase.

Una aplicación podría ser la generación de textos escritos a mano. La escritura a mano consiste en una serie de movimientos coordinados de un bolígrafo, en los que se crea texto. Mediante machine learning, podemos aprender la relación entre los movimientos del bolígrafo y las letras escritas, para generar nuevos ejemplos. Esta funcionalidad puede ser utilizada por médicos forenses, o especialistas en análisis de manuscritos, ya que es posible aprender una cantidad impresionante de estilos de escritura.

Otra posible aplicación sería la de generar nuevos textos con nuevas historias. Mediante redes LSTM se ha conseguido aprender la relación entre los distintos elementos de un texto (letra, palabra, frase...), para después generar nuevos textos letra a letra o palabra a palabra. Los modelos son capaces de aprender como deletrear, puntuar, formar oraciones, e incluso copiar los estilos de escritura para generar dichos textos.

6. Inteligencia artificial en videojuegos.

Todos sabemos que la inteligencia artificial en los videojuegos es una cosa que lleva existiendo desde hace mucho tiempo... En un shooter, la IA de la videoconsola sabe identificar dónde está tu jugador, y con esa información, sabe a quién y dónde disparar. Pero... ¿y si fuese posible analizar todos los píxeles de la pantalla? Mediante deep learning esto es relativamente sencillo, permitiendo a la IA rival tener mucha más información y tomando así mejores decisiones. Un ejemplo nos lo encontramos en AlphaGo, una aplicación desarrollada por Google que ganó al campeón mundial de Go.

Otros ejemplos de aplicaciones de redes neuronales para resolver problemas actuales podrían ser: reconocimiento y traducción de discursos y charlas en tiempo real, foco automático en objetos en movimiento en fotografías, conversión automática de objetos en fotografías, respuestas automáticas a preguntas sobre objetos en fotografías... Como se puede observar, casi todas estas tareas se tratan de automatizar. Son tareas que el ser humano puede hacer manualmente, pero una máquina es capaz de aprender a hacerlo en mucho menos tiempo y con mucha más eficiencia.

2.1. Redes neuronales

2.1.1. Historia de las redes neuronales

Desde hace cientos de años, se ha intentado estudiar el cerebro humano. Sin embargo, a partir de los años 50 cuando empezaron a desarrollarse las bases de la tecnología actual, hubo un fuerte avance en este estudio. Las redes neuronales artificiales, como su propio nombre indica, pretenden imitar la forma de funcionamiento de las neuronas que forman el cerebro humano.

Años 40 :

- A principios de los años 40, el neurólogo Warren McCulloch y el matemático Walter Pitts propusieron los primeros modelos matemáticos y eléctricos de redes neuronales. Describieron como funcionaban las neuronas, y modelaron un red neuronal simple usando circuitos eléctricos.
- A finales de década, Hebb definió dos conceptos muy importantes:
 - El proceso de aprendizaje se localiza principalmente en la sinapsis o conexión entre neuronas.
 - La información se representa en el cerebro mediante un conjunto de neuronas activas o inactivas.

Estas dos reglas se sintetizan en la regla de aprendizaje de Hebb, que sigue siendo usada en algunos modelos actuales. Esta regla nos dice que los cambios en los pesos de la sinapsis se basan en la interacción entre las neuronas pre y post sintácticas, y el numero de veces que se activan de manera simultanea.

Años 50:

- En 1956 en la ciudad de Dartmouth, se llevó a cabo la primera conferencia sobre Inteligencia Artificial donde se discutió sobre la capacidad de las maquinas para simular el aprendizaje humano.
- A finales de los años 50, el neurobiólogo Frank Rosenblatt empezó a trabajar en el concepto de Perceptrón. Esto dio lugar a la regla de aprendizaje Delta, que permitía emplear señales continuas de entrada y de salida.

- En 1959 Bernard Widrow y Marcian Hoff desarrollaron los algoritmos Adaline (Adaptive Linear Neuron) y Madaline (Multiple Adaline). Estos modelos fueron usados para desarrollar la primera red neuronal aplicada a un problema real: un filtro que eliminaba el eco en las llamadas telefónicas.

Años 80:

- En 1982, el interés por las redes neuronales se renovó. John Hopfield presentó varios artículos a la Academia Nacional de Ciencias en los que no solo describía el modelado del cerebro humano, si no que demostró que aplicaciones que podría tener su estudio para crear nuevos dispositivos.
- En 1984, Kohonen desarrolló una familia de redes de memoria asociativa y mapas auto-organizativos, actualmente llamadas redes de Kohonen.
- En 1985, Paul Werbos desarrolló el algoritmo Backpropagation que resolvió el problema de entrenar redes neuronales multicapa de forma efectiva.
- En 1986, los investigadores Hinton y Sejnowski desarrollaron la máquina de Boltzmann, un tipo de red neuronal recurrente estocástica.

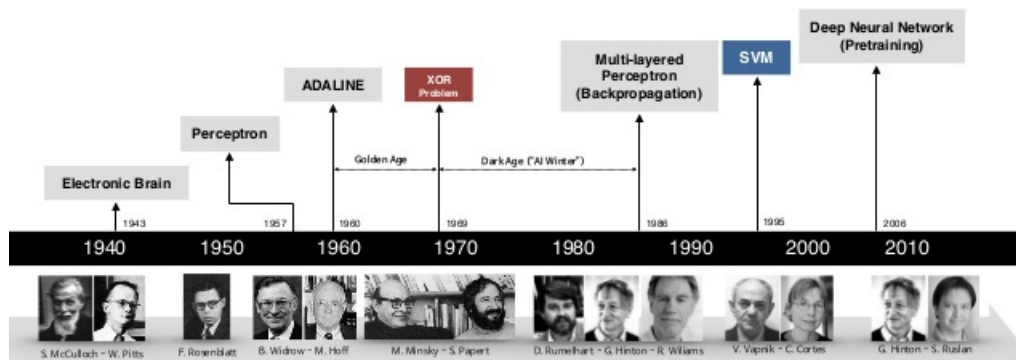


Figura 2.1: Historia de las redes neuronales
Imagen extraída de SlideShare

2.1.2. La neurona biológica

El cerebro, visto a un alto nivel y simplificando enormemente su estructura, podríamos decir que es un conjunto de millones y millones de células, llamadas neuronas, interconectadas entre ellas mediante sinapsis. La sinapsis se lleva a cabo en la zona donde 2 neuronas se conectan, y las partes de la célula que se encargan de realizarla son las dendritas y las ramificaciones del axón.

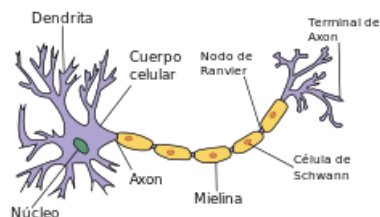


Figura 2.2: Estructura de una neurona
Imagen extraída de Wikipedia

Cada neurona desarrolla impulsos eléctricos que se transmiten a lo largo de esta mediante su axón. Este, al final se ramifica en ramificaciones axonales, que conectan con otras neuronas mediante sus dendritas.

El conjunto de elementos que hay entre la ramificación axonal y la dendrita forman la sinapsis, que regula la transmisión del impulso eléctrico mediante unos elementos químicos llamados neurotransmisores.

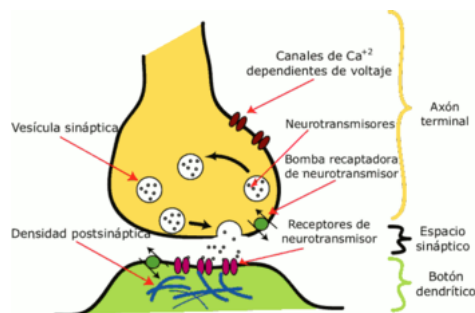


Figura 2.3: Sinapsis
Imagen extraída de Wikipedia

Los neurotransmisores liberados en la sinapsis pueden tener un efecto negativo o positivo sobre la transmisión del impulso eléctrico en la neurona que los recibe en sus dendritas. Esta neurona recibe varias señales de las distintas sinapsis, y las combina consiguiendo un cierto nivel de estimulación. En función de este nivel de activación, la neurona emite señales eléctricas mediante impulsos, con una intensidad determinada y con una frecuencia llamada tasa de disparo.

En resumen, si consideramos que la información del cerebro está codificada en impulsos eléctricos que se transmiten entre neuronas, y que los impulsos se ven modificados básicamente en la sinapsis, podemos intuir que la codificación del aprendizaje estará en la sinapsis y en la forma en la que las neuronas dejan pasar o inhiben las señales segregando neurotransmisores.

2.1.3. La neurona artificial

Las neuronas artificiales son modelos que tratan de simular el comportamiento de las neuronas biológicas. Cada neurona se representa como una unidad de proceso que forma parte de una entidad mayor, la red neuronal.

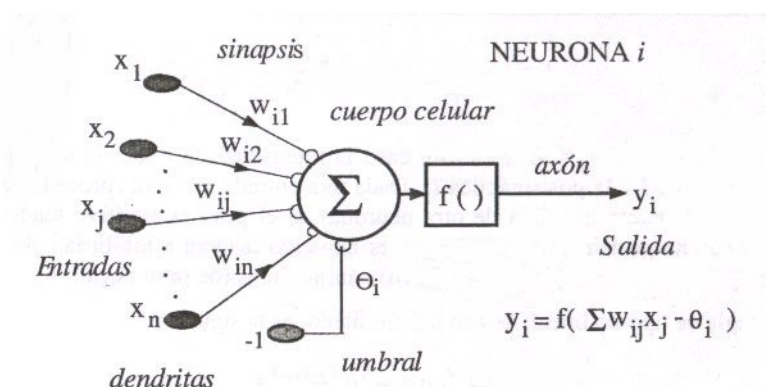


Figura 2.4: Esquema de una neurona artificial
Imagen extraída de ibiblio.org

Como podemos intuir observando la imagen anterior, la neurona artificial se comporta en cierto modo como una biológica, pero de forma simplificada:

- Entradas (x_1, x_2, \dots) : estos valores pueden ser enteros, reales o binarios y equivaldrían a los impulsos que envían otras neuronas a través de sus dendritas.
- Pesos (w_1, w_2, \dots) : equivaldrían a los mecanismos de sinapsis para transmitir el impulso.
- De este modo, el producto de los valores x_i, w_i equivaldría a las señales químicas inhibitoras y excitadoras que se dan en la neurona. Estos valores son la entrada de la función de activación, que convierte todo el conjunto de valores en uno solo llamado potencial. La función de ponderación suele ser la suma ponderada de las entradas y los pesos sinápticos.
- La salida de la función de ponderación llega a la función de activación que transforma este valor en otro en el dominio que trabajan las salidas de las neuronas. Suele ser una función no lineal como la función paso o la función sigmoide, aunque también se usa funciones lineales.

En resumen, podemos establecer las siguientes analogías:

- Neuronas biológicas \iff Neuronas Artificiales.
- Conexiones sinápticas \iff Conexiones ponderadas.
- Efectividad de las sinapsis \iff Peso de las conexiones.
- Efecto excitador o inhibidor de una conexión \iff Signo del peso de una conexión.
- Activación \rightarrow Tasa de disparo \iff Función de activación \rightarrow Salida

2.1.4. Funciones de las neuronas artificiales

Función de red o propagación

Esta función se encarga de transformar las diferentes entradas que provienen de la capa anterior, en el potencial de la neurona actual. Normalmente, las neuronas han de tratar simultáneamente con varios valores de entrada, y han de hacerlo como si se tratase de uno solo. Esta función viene a resolver el problema de combinar las entradas simples (x_1, x_2, x_3, \dots) en una sola entrada global. Podría describirme mediante la siguiente ecuación:

$$input = (x_1 : w_1) * (x_2 : w_2) * \dots * (x_n : w_n) \quad (2.1)$$

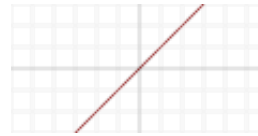
siendo " : " el operador apropiado (por ejemplo: sumatorio, máximo, producto...), x las distintas entradas y w el peso de las conexiones. Multiplicando las entradas por los pesos, se permite que un valor muy grande de entrada pueda tener una influencia pequeña, si sus pesos son pequeños. Por lo general, este operador suele ser la suma ponderada de todas las entradas multiplicadas por sus respectivos pesos.

$$y' = \sum_{i=1}^n x_i * w_i \quad (2.2)$$

Función de activación

La función de activación combina el potencial que nos proporciona la función de propagación con el estado actual de la neurona, para conseguir el estado futuro de esta (activada/desactivada). Esta función es normalmente creciente y monótona. Las funciones más comunes son:

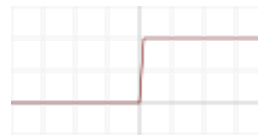
- Identidad: es una función de activación muy simple que siempre devuelve como salida su valor de entrada. Su rango es $(-\infty, \infty)$, y es una función monótona.



$$f(x) = x \quad (2.3)$$

Figura 2.5: Función identidad

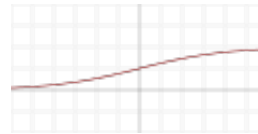
- Escalón binario: Es la más usada por redes neuronales binarias ya que no es lineal, y es bastante sencilla. El Perceptrón y Hopfield son algunos ejemplos de redes que usan esta función. Cuenta con un rango 0,1, y es monótona.



$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (2.4)$$

Figura 2.6: Función escalón binario

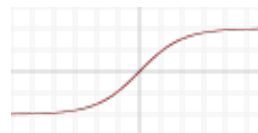
- Logística / Sigmoide / Softstep: Es una de las funciones más utilizadas para modelar redes neuronales. Su rango es continuo en los valores (0,1), es monótona e infinitamente diferenciable.



$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.5)$$

Figura 2.7: Función logística

- Tangente hiperbólica: Esta función es utilizada por redes con salidas continuas. Un ejemplo sería el Perceptrón multicapa con retropropagación, ya que su algoritmo de aprendizaje necesita una función derivable. Cumple con un rango (-1,1), es monótona, y se aproxima a la función identidad en su origen.



$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.6)$$

Figura 2.8: Función tangencial

- Rectificadora (ReLU - Rectified Linear Unit): Esta función de activación la introdujo por primera vez en el año 2000 Hahnloser, con motivaciones biológicas y matemáticas. Se viene usando en redes convolucionales más que la ampliamente extendida función logística sigmoide (se basa en probabilidades), y es más practica que la tangente hiperbólica. La función rectificadora es, por tanto, una de las funciones de activación más populares para deep learning.

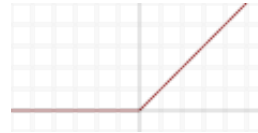


Figura 2.9: Función rectificadora

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \quad (2.7)$$

- Softplus: aproximación suavizada de la función de activación rectificadora.

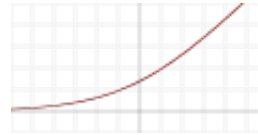


Figura 2.10: Función de activación softplus

$$f(x) = \ln(1 + e^x) \quad (2.8)$$

2.1.5. Tipos de aprendizaje en redes neuronales

Aprendizaje supervisado

Los modelos de aprendizaje supervisado son aquellos utilizados cuando el problema nos presenta el conjunto de datos y los atributos que queremos predecir. Nos encontramos dos categorías:

- Regresión: los valores de salida son una o más variables continuas. Un ejemplo sería predecir el valor de una casa en función de sus metros cuadrados, el número de habitaciones, tamaño de la piscina... (Problema Boston-Housing detallado en el punto 4.2)
- Clasificación: los datos pertenecen a dos o más clases, y se busca aprender como clasificar nuevas entradas en esas clases a partir de datos que ya conocemos. Uno de los ejemplos más conocidos es el Iris dataset, donde se intenta clasificar los datos en 4 tipos de flores según la longitud y la anchura de sus pétalos y sépalos. Otro de ellos es el problema del dataset MNIST, que busca clasificar imágenes de números en los 10 tipos de caracteres numéricos (Problema detallado en el punto 4.3).

Aprendizaje no supervisado

Cuando no hay información previa de salidas dado un conjunto de entradas. En estos casos, el objetivo es encontrar grupos mediante clustering, o determinar una distribución de probabilidad sobre un conjunto de entrada.

2.2. Perceptrón

En los años 50, F.Rosenblatt desarrolló por primera vez la idea de Perceptrón, basándose en la regla de aprendizaje de Hebb y en los modelos de neuronas biológicas de McCulloch y Pitts. Uno de las características que más interés despertó de este modelo fue su capacidad para aprender a reconocer patrones.

El perceptrón se basa en una arquitectura monocapa. Está constituido por un conjunto de células de entrada, que reciben los patrones a reconocer o clasificar, y una o varias células de

salida que se ocupan de clasificar estos patrones de entrada en clases. Cada célula de entrada está conectada con todas las células de salida. En la figura 2.11 se puede observar la arquitectura de un perceptrón simple con 4 entradas y 1 salida.

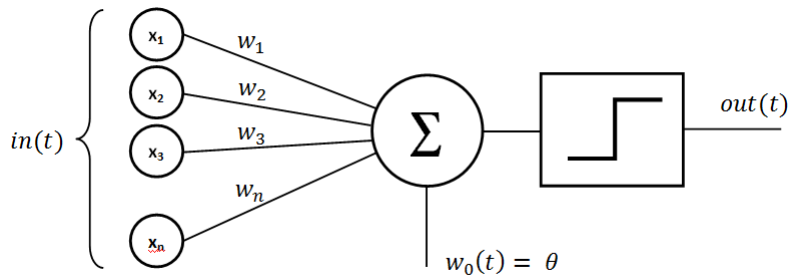


Figura 2.11: Perceptrón simple
Imagen extraída de GitHub

El umbral (θ) es un parámetro utilizado como factor de comparación a la hora de generar la salida de una neurona. En esta arquitectura, la salida de la neurona se calcula multiplicando los valores de las entradas, por los pesos de las conexiones:

$$y' = \sum_{i=1}^n w_i x_i \quad (2.9)$$

Sin embargo, la salida definitiva depende del umbral, por lo que se obtiene aplicándole la función de salida al nivel de activación de la célula, es decir:

$$y = F(y', \theta) \quad (2.10)$$

$$y = \begin{cases} \text{si } y' \geq \theta & 1 \\ \text{si } y' < \theta & 0 \end{cases} \quad (2.11)$$

Con estas fórmulas, podemos deducir que la salida de la célula (y) será 1 (neurona activada) si $y' > \theta$, o 0 (neurona desactivada) en caso contrario.

La función de salida (F) produce una salida binaria, por lo que es un diferenciador en dos categorías. En el caso de tener únicamente dos dimensiones, con las ecuaciones 2.9, 2.10 y 2.11 se podría llegar a la ecuación

$$w_1 x_1 + w_2 x_2 + \theta = 0 \quad (2.12)$$

que es la ecuación de una recta con pendiente

$$-\frac{w_1}{w_2} \quad (2.13)$$

y cuyo corte en la abscisa en el eje x pasa por

$$-\frac{\theta}{w_2}. \quad (2.14)$$

Como podemos ver en la figura 2.12, podemos imaginarnos el perceptrón como una recta, que en un gráfica de dos dimensiones deja las dos categorías a separar a un lado y a otro de la misma.

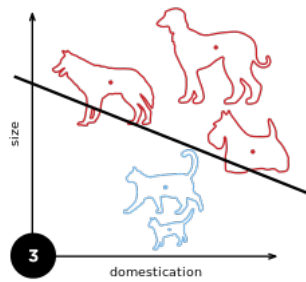


Figura 2.12: Separación lineal del perceptrón.
Imagen extraída de Wikipedia

Aprendizaje

El perceptrón es un tipo de red que utiliza aprendizaje supervisado, es decir, necesita conocer los valores esperados para cada una de las entradas antes de construir el modelo. Este proceso se lleva a cabo mediante la inserción de patrones de entrada del dataset con el que se quiere entrenar la red.

Entrenar la red no es más que encontrar los pesos sinápticos y el umbral que mejor haga predecir a nuestra red los resultados esperados. Para la determinación de estas variables, se sigue un proceso adaptativo. Se comienza con unos valores aleatorios, y se van modificando según la diferencia entre los valores deseados y los obtenidos por la red.

En resumen, el perceptrón simple aprende iterativamente siguiendo estos pasos:

1. Inicialización de variables
2. Bucle de iteraciones:
 - a) Bucle para todos los ejemplos
 - 1) Leer valores de entrada
 - 2) Calcular error
 - 3) Actualizar pesos según el error
 - a' Actualizar pesos de entradas
 - b' Actualizar el umbral
 - b) Comprobar que el vector de pesos es correcto
3. Salida

Capacidades de un perceptrón simple

Hasta ahora, hemos descrito el perceptrón como un clasificador. Sin embargo, también pueden ser usados para emular funciones lógicas elementales como AND, OR y NAND. Consideremos las funciones AND y OR. Dado que son funciones linealmente separables, pueden ser aprendidas por un perceptrón.

Sin embargo, la función XOR no puede ser aprendida por un único perceptrón, ya que requiere el uso de al menos dos líneas para separar las clases. Si se quiere alcanzar esta funcionalidad, es necesario utilizarse al menos una capa más. Aquí es donde entra en juego el perceptrón multicapa.

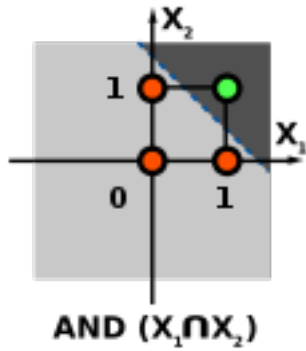


Figura 2.13: Separación lineal de la función AND.

Imagen extraída de Wikipedia

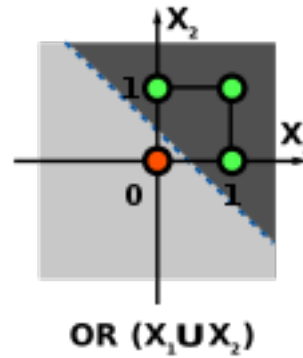


Figura 2.14: Separación lineal de la función OR.

Imagen extraída de Wikipedia

2.3. Perceptrón multicapa

El perceptrón multicapa tuvo su origen en los años 80, con la idea de solucionar el mayor problema del perceptrón simple: no ser capaz de aprender funciones no linealmente separables (como es el caso de la función XOR). Se ha demostrado que es un aproximador universal de funciones.

El perceptrón multicapa es un modelo de red neuronal con alimentación hacia delante, es decir, con conexiones sin bucles (tipo de red feedforward). Está compuesto de varias capas ocultas entre la entrada y la salida de la misma, y caracterizado por tener salidas disjuntas pero relacionadas entre sí, ya que la salida de una neurona es la entrada de la siguiente.

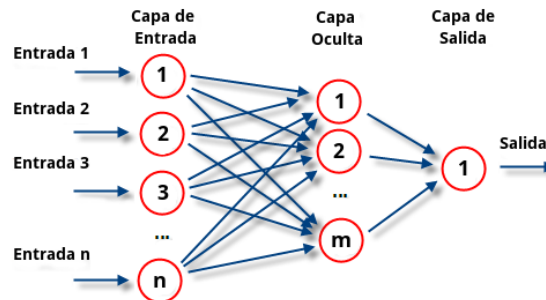


Figura 2.15: Perceptrón multicapa.

Imagen extraída de Wikipedia

Las capacidades de decisión de un perceptrón multicapa de 2 y 3 capas con una única neurona de salida se muestran en la figura 2.16:

En el perceptrón multicapa, al igual que en el perceptrón simple, podemos diferenciar una fase de propagación de los valores de entrada hacia delante, y una fase de aprendizaje en la que los errores obtenidos a la salida del perceptrón se van propagando hacia atrás con el objetivo de actualizar los pesos de las conexiones mediante el gradiente de la función de error. Este algoritmo se llama backpropagation o retropropagación.

2.3.1. Propagación

Imaginemos un perceptrón multicapa con C capas. Llamemos $W^C = W_{ij}^c$ a la matriz de pesos de la capa c y $c+1$, donde W_{ij}^c representa el peso de la neurona i de la capa c a la neurona j de la capa $c+1$. Denominaremos $U^C = u_i^c$ al vector de umbrales de las neuronas de la













Estructura	Regiones de Decisión	Problema de la XOR	Clases con Regiones Mezcladas	Formas de Regiones más Generales
1 Capa 	Medio Plano Limitado por un Hiperplano			
2 Capas 	Regiones Cerradas o Convexas			
3 Capas 	Complejidad Arbitraria Limitada por el Número de Neuronas			

Figura 2.16: Tabla comparativa del perceptrón según sus capas.

Imagen extraída de GitHub

capa c . Se denomina a_i^c a la activación o entrada de la neurona i de la capa c . Las activaciones de las neuronas se calculan de forma distinta dependiendo de la capa en la que nos encontremos.

- Capa de entrada: la activación se corresponde con el patrón de entrada del perceptrón.
- Capa oculta: la activación procede de las salidas de las capas anteriores conectadas a las neuronas de esta capa. Esta activación se calcula como la suma de los productos de las activaciones que reciben las neuronas de las capas anteriores multiplicadas por su peso, añadiéndoles el umbral, es decir:

$$out_i^c = f\left(\sum_{j=1}^{n_{c-1}} w_{ji}^{c-1} a_j^{c-1} + u_i^c\right) \quad (2.15)$$

- Capa de salida: ocurre lo mismo que en las capas ocultas, salvo que ahora la salida de las neuronas se corresponde con la salida de la red.

La función f es a lo que llamamos función de activación. Las funciones de activación más utilizadas en el perceptrón multicapa son la función sigmoideal y la función tangente hiperbólica. Ambas poseen un intervalo continuo dentro de $[0,1]$ y $[-1,1]$ y están relacionadas mediante la expresión $f_{thip}(x) = 2f_{sigm}(x) - 1$, por lo que la utilización de una u otra depende de cual sea más compatible con el tipo de patrón que se va a utilizar. La función sigmoideal tiene un nivel de saturación inferior igual a 0, y la tangente hiperbólica lo tiene a -1.

Normalmente, la función de activación suele ser la misma en todas las neuronas de la red exceptuando las neuronas de la capa de salida que suelen utilizar dos funciones distintas: la función identidad, generalmente utilizada en problemas de regresión, y la función sigmoideal, en problemas de clasificación.

2.3.2. Consideraciones de diseño

A la hora de diseñar un perceptrón multicapa para abordar un problema, el primer paso es determinar la arquitectura de la red. Esto implica determinar tres variables, ya explicadas anteriormente:

- Función de activación: se elige según el comportamiento y el recorrido deseado, pero no influye en la capacidad de la red para resolver el problema.
- Número de neuronas de entrada y de salida: viene dado por la definición del problema. En ocasiones esta afirmación no tiene porque ser del todo cierta, y se ha de proceder a

hacer un análisis previo basándose en técnicas de análisis de correlación, de sensibilidad, de importancia relativa...

- Número de capas ocultas: pese a ser una variable que ha de definir el diseñador, no existe un método que determine el número óptimo de neuronas ocultas necesarias para resolver un problema, ya que existen muchos tipos de arquitectura capaces de resolver un mismo problema. Uno de los métodos, por ejemplo, sería partir de una arquitectura ya entrenada, e ir realizando pequeños cambios en el número de neuronas/capas ocultas para buscar así el diseño óptimo.

2.3.3. Algoritmo de Retropropagación o Backpropagation

Como se ha comentado anteriormente, el proceso de aprendizaje del perceptrón multicapa consiste en propagar hacia atrás cierta información desde la salida del perceptrón hasta su entrada. Esta información será el error entre la salida obtenida y la salida esperada, con el objetivo de que en las siguientes iteraciones la salida obtenida se aproxime lo máximo posible a la salida esperada. Por tanto, nos encontramos ante un algoritmo de aprendizaje supervisado.

Hay muchas funciones de error que se pueden utilizar, pero el perceptrón multicapa usa habitualmente la función del error cuadrático medio, que se define como:

$$e(n) = \frac{1}{2} \sum_{i=1}^{n_c} (s_i(n) - y_i(n))^2 \quad (2.16)$$

siendo n cada una de las iteraciones del aprendizaje. El error obtenido $e(n)$ es el error de una sola iteración, por lo que si queremos entrenar la red con varios patrones, el error medio de esta será $E = \frac{1}{N} \sum_{n=1}^N e(n)$. En resumen, el perceptrón multicapa aprende encontrando el mínimo de la función de error.

Aunque el aprendizaje debe llevarse a cabo minimizando el error total, los métodos más utilizados son los basados en métodos de gradiente estocástico, que consisten en minimizar el error en cada patrón $e(n)$. Por tanto, aplicando el método de gradiente descendente, cada peso de la red w se actualiza para cada patrón de entrada n , y siendo α la tasa de aprendizaje, de acuerdo con la siguiente ecuación:

$$w(n) = w(n-1) - \alpha \frac{\partial e(n)}{\partial w} \quad (2.17)$$

Dado que las neuronas de la red están agrupadas en capas ocultas según niveles, es posible aplicar el método del gradiente descendente de forma eficiente en todas las neuronas, llevando a cabo el algoritmo de retropropagación o regla delta generalizada.

2.3.4. Regla delta generalizada

Como ya se ha comentado, la regla delta generalizada no es más que una forma eficiente de aplicar el método de gradiente a los parámetros de la arquitectura (pesos y umbrales). Su aprendizaje es supervisado. Su uso consiste en ajustar pesos y bías tratando de minimizar la suma del error cuadrático. Para ello, se cambian dichas variables en la dirección contraria al gradiente del error.

Las redes neuronales entrenadas mediante esta técnica, dan respuestas más que razonables cuando al sistema se le presentan entradas que nunca había analizado antes. A una nueva entrada, le hará corresponder una entrada similar a la salida obtenida para un patrón de entrenamiento, siendo éste similar al patrón presentado a la red. Esta es una de las grandes propiedades de la regla delta, su capacidad de generalización.

2.3.5. Tasa de aprendizaje

La tasa de aprendizaje (α) es un parámetro que determina la velocidad a la que van a cambiar los pesos de las conexiones de la red. Tiene generalmente un rango $[0,1]$, siendo los valores más cercanos a cero los que hacen que los pesos cambien poco a poco, acercándose lentamente a la convergencia, y los cercanos a uno los que hacen que la red converja más rápidamente al principio, pero siendo posible que los pesos oscilen demasiado al encontrar el peso óptimo final. Por esta razón es importante encontrar una tasa de aprendizaje óptima.

Aquí es donde entra en juego un segundo término llamado momento (η), que pondera cuánto queremos que influya lo que los pesos han cambiado en la anterior iteración. Con ello, sabremos que si los pesos han cambiado mucho, es que estamos aun lejos del valor de la tasa de aprendizaje óptimo, por lo que avanzaremos en su búsqueda más rápidamente. Por otro lado, si los pesos no han cambiado apenas, sabremos que el valor óptimo de α está cerca.

Si se modifica la ecuación 2.17 para añadir esta mejora, quedaría de la siguiente manera:

$$w(n) = w(n-1) = -\alpha \frac{\partial e(n)}{\partial w} + \eta \Delta w(n-1) \quad (2.18)$$

siendo $\Delta w(n-1) = w(n-1) - w(n-2)$ el incremento que sufrió el parámetro w en la anterior iteración.

2.3.6. Sobreajuste y early stopping

El sobreajuste o overfitting es el efecto secundario de sobreentrenar un algoritmo de aprendizaje con ciertos datos para los que ya se conoce el resultado deseado. El algoritmo debe alcanzar un estado en el cual sea capaz de predecir otros casos a partir de lo ya aprendido mediante datos de entrenamiento, generalizando para poder resolver distintas situaciones a las ya presentes durante el entrenamiento. No obstante, cuando un sistema se entrena demasiado o se entrena con datos extraños, el algoritmo tiende a quedarse ajustado a unas características muy específicas de los datos de entrenamiento que no representan un estado general del problema. En este estado, nuestro diseño es más eficaz al responder a muestras del conjunto de entrenamiento, pero ante nuevas entradas el resultado empeora.

Una manera de resolver este problema es extraer un conjunto de entradas del dataset de entrenamiento, y utilizarlo de manera auxiliar a modo de validación. Ya que este subconjunto se deja al margen durante el entrenamiento, su objetivo será evaluar el error en la red tras cada época para determinar cuando éste empieza a aumentar. Cuando aumente, se procederá a detener el entrenamiento y se conservarán los valores de los pesos del ciclo anterior. A este método se le conoce como early-stopping.

2.4. Redes neuronales convolucionales

Las redes neuronales convolucionales son muy similares a las redes neuronales ordinarias descritas hasta ahora, como el perceptrón multicapa. Las neuronas tienen pesos, sesgos, reciben una entrada con la que realiza un producto escalar y sobre la que aplica una función de activación, tienen una función de pérdida...

Sin embargo, se utilizan principalmente para resolver el mayor problema que tienen las redes neuronales ordinarias: el tratamiento de imágenes. Pese a que con las redes neuronales estándar es posible manejar imágenes (veremos el ejemplo del MNIST en el apartado 4.3), en cuanto las imágenes aumentan su tamaño y calidad esto se vuelve intratable. Al tratarse de neuronas totalmente conectadas, provocaríamos un desperdicio de recursos así como un rápido sobreajuste.

Las redes neuronales convolucionales trabajan dividiendo y modelando la información en partes más pequeñas, y combinando esta información en las capas más profundas de la red. Por ejemplo, en el caso del tratamiento de una imagen, las primeras capas tratarían de detectar los bordes de las figuras. Las siguientes capas buscarían combinar los patrones de detección de bordes para conseguir formas más simples, y aplicar patrones de posición de objetos, iluminación... Por último, en las últimas capas se intentará hacer coincidir la imagen con todos los patrones descubiertos, para conseguir una predicción final de la suma de todos ellos. Así es como las redes neuronales convolucionales consiguen modelar una gran cantidad de datos, dividiendo previamente el problema en partes para conseguir predicciones más sencillas y precisas.

2.4.1. Estructura

En general, todas las redes neuronales convoluciones están formadas por una estructura compuesta por 3 tipos de capas:

Capa convolucional

Esta capa le da nombre de la red. En vez de utilizar la multiplicación de matrices como en la redes neuronales estándar, se aplica una operación llamada convolución. Esta operación recibe como entrada una imagen, y sobre ella aplica un filtro que nos devuelve su mapa de características, reduciendo así el tamaño de sus parámetros. La convolución utiliza tres ideas que ayudan a mejorar cualquier sistema sobre el que se aplique machine learning:

- **Interacciones dispersas:** Al aplicar un filtro de menos tamaño sobre la entrada, reducimos bastante el número de parámetros y cálculos.
- **Parámetros compartidos:** Compartir parámetros entre los distintos tipos de filtros ayuda a mejorar la eficiencia del sistema.
- **Representaciones equivariantes:** Si las entradas cambian, las salidas cambian de forma similar.

Además, la convolución proporciona herramientas para trabajar con entradas de tamaño variable, lo cual es muy útil cuando se trabaja con imágenes (cada imagen tiene un número distinto de píxeles). Se basa en un operador matemático que transforma dos funciones f y g , en una tercera, que en cierto sentido, representa la magnitud en la que se superponen f y una versión trasladada y rotada de g .

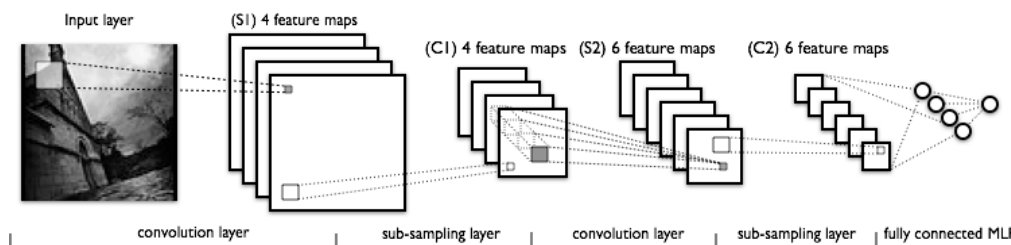


Figura 2.17: Funcionamiento de capa convolucional.

Imagen extraída de deeplearning.net

Capa de reducción o pooling

Esta capa se coloca generalmente después de la capa convolucional. Es la encargada de reducir la cantidad de parámetros a analizar reduciendo las dimensiones espaciales (ancho x alto), quedándose de esta forma con las características más comunes. La operación que lleva a cabo esta capa también se llama reducción de muestreo o submuestreo, ya que la reducción de tamaño implica también una pérdida de información. Sin embargo, para un red neuronal, este tipo de pérdida puede ser beneficioso debido a:

- La reducción del tamaño provoca una menor sobrecarga de cálculo en las próximas capas de la red.
- Reduce habitualmente el sobreajuste.

La operación que se suele aplicar en esta capa es "max-pooling", que divide la imagen de entrada en un conjunto de rectángulos, y respecto a cada uno de ellos, se queda con el valor máximo.

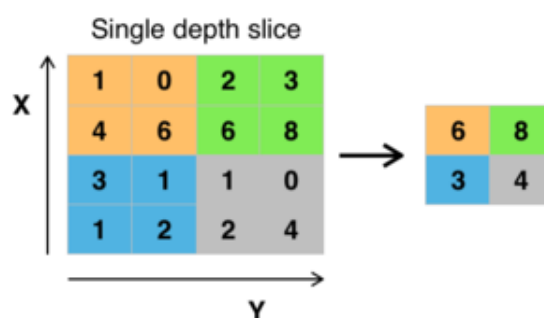


Figura 2.18: Max-pooling aplicado a una imagen
Imagen extraída de Wikipedia

Capa clasificadora totalmente conectada

Una vez que la imagen ha pasado tanto por las capas convolucionales como las de pooling y se han extraído sus características más destacadas, los datos llegan a la fase de clasificación. Para ello, las redes convolucionales normalmente utilizan capas completamente conectadas en las que cada píxel se trata como una neurona independiente. Las neuronas de esta fase funcionan como las de un perceptrón multicapa, donde la salida de cada neurona se calcula multiplicando la salida de la capa anterior por el peso de la conexión, y aplicando a este dato una función de activación.

2.5. Redes neuronales recurrentes

Los humanos no empiezan a pensar de cero a cada segundo. Por ejemplo, mientras leemos, entendemos cada palabra basándonos en el contexto que forman las palabras previas. No desperdiciamos las ideas anteriores, sino que estas tienen persistencia en la memoria. Las redes neuronales tradicionales no cuentan con esta capacidad, y es su mayor defecto. Por ejemplo, imaginemos el caso en el que queremos clasificar que clase de evento está pasando en un punto de una película. No está muy claro cómo una red neuronal tradicional podría usar razonablemente las escenas anteriores para predecir las siguientes. Y aquí es donde entran en juego las redes neuronales recurrentes, ya que estas sí resuelven este

problema. Su principal característica es que son redes con bucles, que permiten que la información persista.

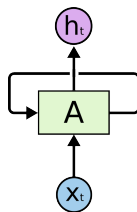


Figura 2.19: Ejemplo básico de red neuronal recurrente.

Imagen extraída de GitHub

En el diagrama 2.19, podemos ver cómo una parte de la red neuronal, **A**, recibe una entrada **x** y devuelve una salida **h**. El bucle permite que la información pase de un ciclo de la red al siguiente. Pese a que el concepto de bucle parezca algo extraño, las redes neuronales que cuentan con ellos no son tan distantes de las redes neuronales tradicionales. Una red neuronal recurrente puede ser creada utilizando múltiples copias de la misma red, pasando el mensaje o salida al nodo sucesor. Si desenrollamos el bucle, tendríamos algo parecido a la figura 2.20:

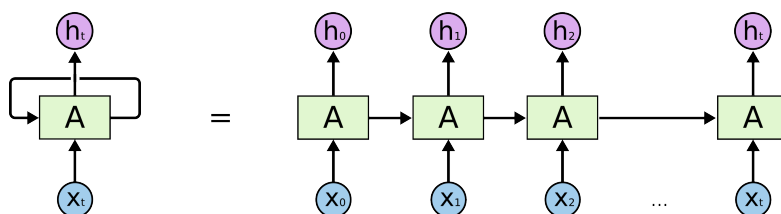


Figura 2.20: Red neuronal recurrente desenrollada.

Imagen extraída de GitHub

Esta forma de cadena hace prever que las redes neuronales recurrentes están íntimamente relacionadas con secuencias y listas. En los últimos años, se han logrado auténticos grandes éxitos aplicando las RNN para resolver problemas como reconocimiento de discursos, modelación de lenguajes, traducciones, captación de imágenes... Un factor clave en estos éxitos es el uso de las redes LSTMs, un tipo muy especial de redes neuronales recurrentes que trabajan por muchas razones de forma más eficiente que la versión estándar.

El problema de la longitud de las dependencias.

Uno de los intereses de las RNNs es la idea de conectar información antigua con la tarea actual. En temas de vídeo, sería útil por ejemplo utilizar frames anteriores para entender el frame actual, pero... ¿pueden las RNN hacer esto? La respuesta es, depende.

Algunas veces, sólo necesitamos utilizar información reciente para la tarea actual. Por ejemplo, en temas de modelación de lenguaje, para adivinar la siguiente palabra en un contexto, en la mayoría de los casos bastaría con analizar la frase en la que se encuentra. Sin embargo, hay casos en los que se necesita acceder a un contexto más grande. Desafortunadamente, a medida que el espacio entre la información requerida y la tarea actual aumenta, las RNN tardan cada vez más en aprender a conectar la información.

En teoría, las RNN son perfectamente capaces de resolver este problema (long-term dependencies), pero en la práctica no lo parece tanto. En 1991, Hochreiter estudió este problema, y encontró algunas razones por las que las RNN encuentran dificultades a la hora de aprender información sobre grandes contextos. Por suerte, las redes LSTMs solventan estos problemas.

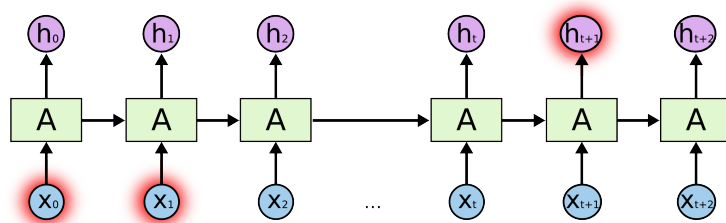


Figura 2.21: Red neuronal recurrente desenrollada.

Imagen extraída de GitHub

2.5.1. Redes LSTM

Las redes Long Short Term Memory (normalmente llamadas LSTMs) son un tipo especial de redes neuronales recurrentes descritas por primera vez en 1997 por Hochreiter & Schmidhuber, capaces de aprender una larga lista de dependencias en largos periodos de tiempo.

Todas las redes neuronales recurrentes tienen forma de cadena al repetir sus módulos. En las RNN estándar, el módulo repetidor tiene una estructura muy simple, con una sola capa, como podría ser la de tipo tanh. Las redes LSTM también tienen forma de cadena, pero el módulo repetidor en vez de tener una sola capa, tiene cuatro.

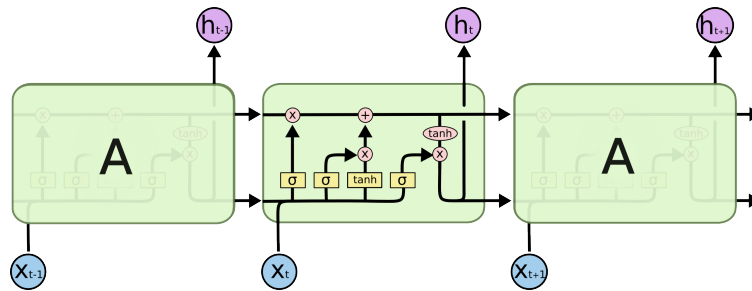


Figura 2.22: Estructura del módulo repetidor.

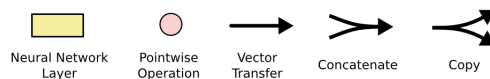
Imagen extraída de GitHub

Figura 2.23: Notación del módulo repetidor.

Imagen extraída de GitHub

La clave de las redes LSTMs se encuentra en el estado de la celda, es decir, en la línea horizontal que recorre la parte superior del diagrama 2.21, ya que permite a la información fluir por toda la red con sólo algunas pequeñas interacciones, sin ser a penas alterada. La red también tiene la habilidad de borrar o añadir información al estado de la celda mediante estructuras llamadas puertas. Estas puertas son una manera de permitir opcionalmente a la información circular o no, y se componen de una capa sigmoide y una operación de multiplicación. Las salidas de la capa sigmoide alternan entre 0 y 1, describiendo si la información fluye o no. Una red LSTM tiene 3 de estas puertas, para proteger y controlar el estado de la celda.

Las redes neuronales LSTM siguen una serie de pasos para decidir qué información se va a ir almacenando o borrando. Podemos destacar los siguientes:

1. **Decidir qué información despreciar.** Esta decisión la toma la capa sigmoide, que devuelve una salida igual a 1 si se busca mantener la información, o 0 si queremos desha-

cernos de ella.

Volvamos al ejemplo de modelado del lenguaje, donde se quiere predecir una palabra basándose en las anteriores. En este problema, el estado de la celda puede almacenar el género del sujeto para usar los pronombres adecuados. Cuando nos encontremos un nuevo sujeto, olvidaremos el género del anterior ya que no será de utilidad.

2. **Decidir qué información se va a almacenar.** Consta de dos partes: primero, una capa sigmoide decide qué valores son actualizados. Después, una capa tanh crea un vector con los nuevos valores candidatos que pueden ser añadidos al estado.

En el ejemplo anterior, se decidiría que queremos añadir el género del nuevo sujeto al estado de la celda, para reemplazar el viejo que queremos olvidar.

3. **Actualizar el estado de la celda.** En el paso anterior se decide qué hacer, y ahora toca hacerlo. Se multiplica el viejo estado por la salida de la capa sigmoide del primer paso, olvidando así las cosas que se decidieron olvidar. A esto, se le suma el producto de la salida de la capa sigmoide y de la capa tanh del segundo paso. Con ello, se consiguen saber los nuevos valores candidatos, escalados según la decisión de actualizar el estado de la celda.

En el caso de modelado de lenguaje, ahora es cuando se desecha la información sobre el género del viejo sujeto y se añade la nueva información, como se decidió en los pasos anteriores.

4. **Decidir la salida basándose en el estado de la celda, pero de manera filtrada.**

Primero, mediante una capa sigmoide, se decide qué partes del nuevo estado de la celda se van a sacar como salida. Después, se pasa el estado de la celda por una capa tanh para tener valores entre -1 y 1, y se multiplica por la salida de la capa sigmoide para sacar solo las partes que se decidan.

En el ejemplo de modelado de lenguaje, una vez que se encuentre el nuevo sujeto, se sacará información acerca de él como puede ser si éste es singular o plural, para después poder conjugar correctamente el verbo que le sigue.

Como se ha comentado anteriormente, la mayoría de los mejores resultados utilizando redes recurrentes, es mediante las redes LSTMs. Pero a pesar de que su descubrimiento ha sido un gran hito, todavía quedan metas por cumplir. Una de ellas sería dotar a las RNN de una capacidad para decidir qué información consultar en una fuente muy grande de información. Por ejemplo, si estamos usando redes RNN para crear un título que describa una imagen, sería útil elegir que partes de la imagen usar para cada palabra.

3.1. Introducción

Keras es una librería de redes neuronales escrita en Python y capaz de ejecutarse tanto sobre Tensorflow como sobre Theano. Sus principales características son:

- Fácil y rápido prototipado gracias a su modularidad, minimalismo y extensibilidad.
- Soporta tanto redes neuronales convolucionales como recurrentes (así como la combinación de ambas)
- Soporta esquemas de conectividad arbitrarios (incluyendo entrenamiento multi-entrada y multi-salida)
- Corre en CPU y GPU
- Es compatible con Python 2.7-3.5

3.2. Instalación

Para la instalación de Keras, son necesarias las siguientes dependencias:

- `numpy` y `scipy` : librerías científicas para Python numéricas.
- `pyyaml` : parseador YAML para Python.
- `HDF5` y `h5py` (Opcional, pero requerido si se usan funciones para cargar/guardar modelos) : paquete para el manejo de grandes cantidades de datos.
- `cuDNN` - `Nvidia CUDA` (Opcional pero recomendado si se usan CNNs) : librería para tarjetas gráficas NVIDIA que permite la aceleración de la GPU en temas de redes neuronales.
- `Tensorflow` o `Theano` : librerías Python enfocadas a Machine Learning.

Existen 2 modos de instalación:

- Desde PyPI: `sudo pip install keras`
- Desde repositorio Git: `sudo python setup.py install`

Por defecto, Keras usa Tensorflow como motor backend. Sin embargo, esta opción puede ser configurada. La primera vez se ejecuta Keras, se crea un fichero de configuración en `/.keras/keras.json`, que puede ser modificado.

```
"image_dim_ordering": "tf",
"epsilon": 1e-07,
"floatx": "float32",
"backend": "tensorflow"
```

3.3. Modelos

Hay dos tipos de modelos disponibles para su implementación en Keras: el modelo secuencial, y la clase `Model` (más general). Ambos modelos presentan varios métodos en común, como `summary()`, `get_config()` o `to_json()` que devuelven información básica sobre el modelo, o `get_weights()` y `set_weights()`, los métodos `getter` y `setter` propios de los pesos del modelo.

3.3.1. Modelo secuencial

Los métodos básicos del modelo secuencial son:

- `compile(self, optimizer, loss, metrics=[], sample_weight_mode=None)` : Calcula automáticamente la función gradiente y prepara el modelo para una forma concreta de entrenamiento.
- `fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[], validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None)` : Entrena el modelo para un número fijado de ciclos (`nb_epoch`), con actualizaciones de gradiente cada `batch_size` ejemplos.
- `evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)` : Calcula la función de pérdida dados unos datos de entrada, lote por lote.
- `predict(self, x, batch_size=32, verbose=0)` : Genera predicciones para unos valores de entrada.

3.3.2. Clase Model de la API

Con la clase `Model`, dadas una entrada y una salida, podemos inicializar nuestro modelo de la siguiente forma:

```
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(input=a, output=b)
```

En caso de que tengamos múltiples conjuntos de entradas y de salidas, también podríamos inicializarlo de la siguiente manera:

```
model = Model(input=[a1, a2], output=[b1, b3, b3])
```

3.4. Capas

3.4.1. Funciones básicas de las capas

A la hora de diseñar las capas de nuestra red neuronal, tenemos una serie de funciones disponibles:

- **Dense:** Para capas regulares totalmente conectadas.
`keras.layers.core.Dense(output_dim, init='glorot_uniform', activation='linear', weights=None, W_regularizer=None, b_regularizer=None, activity_regularizer=None, W_constraint=None, b_constraint=None, bias=True, input_dim=None)`
- **Activation:** Se aplica una función de activación a una salida. Las principales funciones que nos ofrece Keras son:

- `softmax`: generalización de la función logística.
- `softplus`
- `softsign`
- `relu`: función rectificadora
- `tanh`: función tangente hiperbólica
- `sigmoid`: función sigmoide
- `hard_sigmoid`
- `linear`: función lineal

Podemos encontrar funciones de activación más avanzadas como `PReLU` o `LeakyReLU` en el módulo `keras.layers.advanced_activations`.

- `Dropout`: Consiste en establecer al azar una fracción de unidades de entrada a 0 en cada actualización durante la fase de entrenamiento. Esto ayuda a evitar el sobreajuste. `keras.layers.core.Dropout(p)`
- `Reshape`: Transforma la salida en una forma concreta.

Algunos ejemplos de uso podrían ser:

- `model.add(Dense(32, input_dim=16))` : modelo que contaría con una array de entrada de tamaño 16, y de salida de 32.
- `model.add(Activation('tanh'))` : aplica la función de activación tangencial.
- `model.add(Dropout(0.2))` : establecería entradas a 0 en el 20 % de los casos.
- `model.add(Reshape((3, 4), input_shape=(12,)))` : define una capa de salida de 3x4 para nuestro modelo.

3.4.2. Capas convolucionales

Keras presenta diversas funciones para desarrollar capas convolucionales según las dimensiones de su entrada. La más destacable, a modo de ejemplo, es:

```
Convolution2D(nb_filter, nb_row, nb_col, init='glorot_uniform', activation=None, weights=None, border_mode='valid', subsample=(1, 1), dim_ordering='default', W_regularizer=None, b_regularizer=None, activity_regularizer=None, W_constraint=None, b_constraint=None, bias=True)
```

Un ejemplo de uso donde se define una capa convolucional de dos dimensiones, con 32 mapeadores de tamaño 5x5 y una función de activación rectificadora, sería:

```
model.add(Convolution2D(32, 5, 5, border_mode='valid', activation='relu')) :
```

Además de la versión en 2D, Keras nos ofrece funciones para entradas de 1 dimensión o 3 dimensiones. Aparte, tenemos funcionalidades para desarrollar capas convolucionales dilatadas (`AtrousConvolution2D`), hacer deconvolución (`Deconvolution2D`), etc.

3.4.3. Capas pooling

Al igual que ocurre con las capas convolucionales, Keras ofrece un buen número de funciones para desarrollar capas pooling en redes convolucionales, dependiendo de las dimensiones de la entrada. Como en el caso anterior, una función representativa sería:

```
MaxPooling2D(pool_size=(2, 2), strides=None, border_mode='valid', dim_ordering='default')
```

Si en vez de utilizar la función máximo se requiere que nuestra capa utilice otro tipo de algoritmos, Keras también nos ofrece funciones como: `AveragePooling2D`, `GlobalMaxPooling2D`, `GlobalAveragePooling2D` ... Además de sus variantes para 1D y 3D.

3.4.4. Capas recurrentes

Keras presenta 3 tipos de capas recurrentes:

- `SimpleRNN` : Red recurrente estándar totalmente conectada cuya salida realimenta la entrada.
- `LSTM` : Redes Long-Short Term Memory. Este tipo de red ha sido descrito en el apartado 2.5, y será puesto en ejemplo en el apartado 4.4
- `GRU` : Gated Recurrent Unit. Red neuronal recurrente muy similar a las redes LSTM, pero que no dispone de una unidad de memoria interna y utiliza dos puertas en lugar de tres.

3.5. Inicializadores

Los inicializadores definen la manera de inicializar los pesos de las capas de nuestro modelo. El argumento usado en para definir este inicializador en las capas es **`init`**. Las opciones más comunes son:

- `uniform`
- `normal`
- `identity`
- `zero`
- `glorot_normal`
- `glorot_uniform`

3.6. Activaciones

El modo de activación puede ser implementado mediante una capa `Activation` (`model.add(Dense(64)); model.add(Activation('tanh'))`), o mediante un parámetro que soportan todas las capas (`model.add(Dense(64, activation='tanh'))`).

Las funciones más comunes, descritas previamente en el punto 2.1.4, son :

- `linear`: función lineal.
- `sigmoid`: función sigmoide / logística.
- `tanh`: función tangente hiperbólica.
- `relu`: función rectificadora.
- `softplus`: aproximación suavizada de la función rectificadora.

3.7. Funciones de pérdida

La función de pérdida es uno de los dos parámetros necesarios para compilar un modelo. Algunas de las funciones más conocidas que nos proporciona Keras son:

- `mean_squared_error(y_true, y_pred)` : Calcula el error cuadrático medio.
- `mean_absolute_error / mae` : Calcula el error medio absoluto.
- `binary_crossentropy`: Calcula la entropía cruzada en problemas de clasificación binaria. También conocida como perdida logarítmica.

3.7.1. Regularizadores

Los regularizadores permiten aplicar penalizaciones en las capas durante la optimización. Estas penalizaciones se incorporan a la función de perdida que la red optimiza.

Ejemplo de uso: `model.add(Dense(64, input_dim=64, W_regularizer=l2(0.01), activity_regularizer=activity_l2(0.01)))`

3.8. Optimizadores

El optimizador es otro de los dos valores necesarios para compilar un modelo (junto con la función de pérdida). Los principales optimizadores disponibles son:

- `SGD` : aproximación estocástica del metodo de gradiente descendiente descrito en los puntos 2.3.3 y 2.3.4.
- `Adagrad` : algoritmo basado en gradiente que adapta el ratio de aprendizaje a los parámetros, realizando grandes actualizaciones cuando éstos son infrecuentes, y pequeñas actualizaciones cuando son frecuentes.
- `Adadelta` : extensión del algoritmo Adagrad
- `Adam` : algoritmo basado en el gradiente de primer orden de funciones estocásticas objetivas. Este método es sencillo de implementar, es eficiente computacionalmente hablando, necesita poca capacidad de memoria, y es muy bueno en problemas con una gran cantidad de datos y/o parámetros. Es utilizado por defecto en los modelos de Keras.

3.9. Callbacks

Los callbacks son una serie de funciones que pueden ser aplicadas en ciertos momentos del proceso de entrenamiento. Estos callbacks pueden ser utilizados para echar un vistazo a los estados internos así como a las estadísticas del modelo que estamos entrenando. Para utilizarlos, basta con pasar como argumento a la función `fit(...)` una lista de métodos, que serán llamados en cada fase del entrenamiento.

Las principales funciones son:

- `BaseLogger()` : Se aplica en todo modelo de Keras, y acumula la media de las métricas por época que se han definido para ser monitorizadas.
- `Callback()` : Clase base abstracta, utilizada para crear nuevos callbacks. Como parámetros, recibe los propios para configurar el entrenamiento (numero de ciclos, verbosidad...), así como la instancia del modelo que queremos entrenar.
- `ProgbarLogger()` : Callback que muestra métricas en la salida estándar (stdout).
- `History()` : Callback aplicado en cada modelo Keras, que almacena eventos en un objeto de tipo `History`.

- `ModelCheckpoint(...)`: Almacena el modelo después de cada época. Como argumentos recibe el fichero donde dejar la información, la cantidad de de elementos a monitorizar, el modo de verbosidad, un booleano para decidir si sobrescribir o no el mejor modelo, el modo, y otro booleano para guardar solo pesos o información de todo el modelo.

3.10. Datasets

Keras cuenta con una serie de dataset ya predefinidos y listos para realizar nuestras pruebas:

- **CIFAR10**: dataset que cuenta con 50.000 imágenes a color de entrenamiento, a clasificar en 10 categorías, y 10.000 imágenes para testear los resultados. También existe una versión con 100 categorías (**CIFAR100**).
- **Clasificación de sentimientos en reviews de IMDB**: Este dataset cuenta con 25.000 películas etiquetadas según el sentimiento (bueno/malo). Estas reviews han sido preprocesadas, y las palabras han sido indexadas según su frecuencia.
- **MNIST**: dataset que contiene 60.000 imágenes en blanco y negro de 10 tipos de dígitos, y 10.000 imágenes para testear los modelos.

3.11. Preprocesado

Keras cuenta con una serie de funciones para procesar tanto modelos secuenciales, como texto, como imágenes. Las más importantes son:

- Modelos secuenciales: `pad_sequences(...)` transforma una lista en un array 2D.
- Texto: `text_to_word_sequence(...)` separa una frase en palabras, `one_hot(...)` codifica un texto en una lista de palabras indexadas en un vocabulario de tamaño n...
- Imágenes: `ImageDataGenerator(...)` define la configuración para la preparación y el aumento de una imagen, a la hora de ser procesada.

3.12. Visualización

El módulo `keras.utils.visualize_util` proporciona funciones útiles para pintar gráficamente un modelo (acompañado del uso de `graphviz`). La siguiente función mostraría una gráfica del modelo y lo guardaría en un fichero `.png`.

```
plot(model, to_file='model.png')
```

3.13. Wrapper para Scikit-Learn API

Keras presenta compatibilidad con Scikit-Learn API. Esta conocida librería open source proporciona herramientas para la minería y análisis de datos. Trabaja junto con las librerías SciPy y NumPy de Python, y permite resolver problemas de regresión, clasificación y clustering. Se pueden usar modelos Keras secuenciales como parte del ciclo de trabajo de Scikit-Learn mediante sus wrappers. Existen dos interfaces disponibles:

- `KerasClassifier(build_fn=None, **sk_params)` : Implementa una interfaz de clasificación.
- `KerasRegressor(build_fn=None, **sk_params)` : Implementa una interfaz de regresión.

El primer argumento es la instancia de la clase o la función a llamar, y el segundo los parámetros del modelo y de ajuste.

4.1. Problema de diabetes - Regresión logística

Como primer ejemplo introductorio a las redes neuronales en Keras, se va a utilizar un conocido dataset que representa diabetes en indios americanos. Este dataset se puede adquirir en el repositorio de UCI Machine Learning, y describe información médica de pacientes indios, así como información sobre si han tenido diabetes en los siguientes 5 años. La información médica especificada es la siguiente:

- Número de embarazos.
- Concentración de glucosa.
- Presión en sangre.
- Tamaño de la doblez de la piel en el triceps.
- Insulina
- Índice de masa corporal
- Función que representa el numero de antepasados con diabetes.
- Edad

4.1.1. Cargando datos

Cuando se trabaja con algoritmos de Machine Learning que usan números aleatorios, es una buena metodología de trabajo predefinir de antemano una semilla. Con ello, podremos ejecutar el mismo código varias veces obteniendo el mismo resultado. Esta idea es bastante útil a la hora de demostrar un resultado, comparar distintos algoritmos, debuggear parte del código... Se iniciará la generación de números aleatorios con la semilla que queramos, por ejemplo:

```
from keras.models import Sequential
from keras.layers import Dense
import numpy
numpy.random.seed(4)
```

Nos encontramos ante un problema de clasificación binaria (tener diabetes representa salida 1, y no tenerla salida 0). Todas las variables que definen al paciente son numéricas, lo cual hace fácil su uso directamente en redes neuronales, que esperan datos numéricos tanto en las entradas como en las salidas.

Una vez que tenemos descargado nuestra fuente de datos, podremos cargarla directamente usando la función `loadtxt()` de la librería NumPy. En este dataset, podemos apreciar 8 datos de entrada, y 1 variable de salida (la última columna). Una vez cargado, podemos separar los datos en variables de entrada y targets.


```
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
X = dataset[:,0:8]
Y = dataset[:,8]
```

4.1.2. Definiendo el modelo

Como se ha comentado en el capítulo 3, los modelos en Keras se definen como una secuencia de capas. La primera cosa de la que hay que asegurarse, es de que la capa de entrada tiene el número correcto de entradas. Esto se define mediante el argumento `input_dim`, cuando se crea la primera capa.

Las capas totalmente conectadas se definen usando la clase `Dense`. Podemos especificar el número de neuronas de la capa utilizando el primer argumento, el método de inicialización con el segundo (`init`), y la función de activación usando el argumento `activation`.

En este caso, se inicializarán los pesos de la red con pequeños números aleatorios de una distribución Gaussiana (`normal`). Se usará la función de activación rectificadora (`relu`) en las primeras dos capas, y la función sigmoide en la capa de salida. Las funciones sigmoide e hiperbólica solían ser las funciones de activación más utilizadas. Sin embargo, a día de hoy, se consiguen mejores resultados utilizando la función de activación rectificadora. La función sigmoide en la capa de salida será necesaria para asegurarse que la salida de nuestro modelo comprende valores entre 0 y 1.

La primera capa tendrá 12 neuronas y espera 8 variables de entrada. La segunda tiene 8 neuronas, y la ultima solo 1, la cual reflejará en su salida la predicción sobre si para los datos médicos introducidos como entrada, se tendrá diabetes o no.

```
model = Sequential()
model.add(Dense(12, input_dim=8, init='normal', activation='relu'))
model.add(Dense(8, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
```

4.1.3. Compilando el modelo

Una vez que se ha definido el modelo, el siguiente paso es compilarlo. Aquí es donde entra en juego el backend elegido (Theano o Tensorflow), ya que Keras utiliza sus librerías numéricas. El propio backend es el encargado de elegir la mejor forma para entrenar la red y de hacer predicciones con ella, utilizando la CPU, la GPU o ambas a la vez.

En la fase de compilación, se deben especificar algunas propiedades adicionales para entrenar la red (recordar que entrenar una red significa encontrar los mejores pesos para hacer predicciones de un problema). Debemos especificar la función de pérdida para evaluar los pesos, el optimizador usado para elegir entre los diferentes pesos, y métricas opcionales para recoger y mostrar los datos durante el entrenamiento.

En este ejemplo, se utilizará una función de pérdida logarítmica, que para los problemas de clasificación está definida en Keras como `binary_crossentropy`. Como optimizador se utilizará el algoritmo de gradiente descendiente `adam`, el cual usa Keras por defecto.

Finalmente, como se trata de un problema de clasificación, se recogerá y mostrará la eficiencia de la clasificación, o lo que es lo mismo, el porcentaje de aciertos (argumento `accuracy`).

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4.1.4. Ajustando el modelo

Una vez compilado, el siguiente paso es ejecutar el modelo con los datos de entrada. Podemos entrenar o ajustar nuestro modelo cargando los datos utilizando la función `fit()`.

El proceso de entrenamiento se ejecutará para un número de iteraciones definido mediante el argumento `nb_epoch`. También podemos definir el numero de entradas que serán evaluadas antes de que haya una actualización de los pesos de la red mediante el argumento `batch_size`). De nuevo, estas medidas pueden ser elegidas experimentalmente a base de prueba y error.

```
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

4.1.5. Evaluando el modelo

Una vez entrenada la red neuronal, podemos evaluar el rendimiento de esta utilizando el mismo dataset. Esto nos puede dar una idea de cómo hemos modelado los datos de entrada, pero no tendremos idea de como de bien clasificará los nuevos datos nuestro modelo.

Para evaluar nuestra red, se utiliza la función `evaluate()`. En este ejemplo, se le pasarán las mismas entradas que cuando se entrenó la red, generando así una predicción por cada par de entrada-salida, además de mostrar información sobre las la perdida media y la precisión de nuestro modelo.

```
eva = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], eva[1]*100))
```

4.1.6. Analizando datos

Al ejecutar el código, podemos ver un mensaje por cada 150 ciclos mostrando la pérdida y la precisión de cada uno, seguido de los resultados finales.

```
Epoch 1/150
768/768 [=====] - 0s - loss: 0.6829 - acc: 0.6471
Epoch 2/150
768/768 [=====] - 0s - loss: 0.6654 - acc: 0.6510
Epoch 3/150
768/768 [=====] - 0s - loss: 0.6556 - acc: 0.6510
...
Epoch 148/150
768/768 [=====] - 0s - loss: 0.4487 - acc: 0.7826
Epoch 149/150
768/768 [=====] - 0s - loss: 0.4444 - acc: 0.7839
Epoch 150/150
768/768 [=====] - 0s - loss: 0.4456 - acc: 0.7747
32/768 [>.....] - ETA: 0
sacc: 77.34%
```

El siguiente paso es adaptar el ejemplo anterior y utilizarlo para generar predicciones sobre el dataset de entrenamiento, simulando que nos encontramos ante un nuevo dataset que nunca hemos analizado previamente. Mediante la función `model.predict`, podremos realizar estas predicciones fácilmente.

Con las siguientes lineas, se mostrarán predicciones para cada entrada del dataset, que en su mayor parte, coincidirán con los datos de salida ya descritos en el dataset. Recordar que 1 es padecer diabetes, y 0 lo contrario.

```
predicciones = model.predict(X)
rounded = [round(x) for x in predicciones]
print(rounded)
```

4.2. Problema Boston Housing - Regresión lineal

4.2.1. Descripción del problema

En este caso, nos encontramos ante un problema de regresión lineal. El dataset describe 13 propiedades numéricas de 506 casas de los suburbios de Boston, con el precio de dichas casas en millones de dolares. Este conocido dataset puede ser adquirido en el repositorio de UCI Machine Learning. Como atributos de entrada, disponemos de:

- CRIM: ratio de crimen per capita por población.
- ZN: proporción de tierras residenciales zonificadas con más de 25000 pies cuadrados.
- INDUS: proporción de acres no comerciales por ciudad
- CHAS: Charles River dummy variable (1 si el tramo se encuentra en el río, 0 en caso contrario).
- NOX: concentración de óxidos nítricos.
- RM: numero medio de habitaciones por vivienda.
- AGE: proporción de viviendas ocupadas construidas antes de 1940.
- DIS: distancias ponderadas a cinco centros de empleo.
- RAD: índice de accesibilidad a las autopistas radiales.
- TAX: ratio de impuestos a la propiedad por cada 10.000\$.
- PTRATIO: ratio de alumnos-profesores.
- B: $1000(B_k - 0,63)^2$ donde B_k es la proporción de ciudadanos negros por población.
- LSTAT: porcentaje de la población de clase baja.

Como target, nos proporcionan el valor medio de las casas ocupadas, en miles de dolares. Un rendimiento razonable en modelos actuales usando el Error cuadrático medio (MSE) como método de evaluación está en torno a 20 (miles de dolares al cuadrado). Veamos si podemos igualar estos resultados.

4.2.2. Desarrollo de la red neuronal

Lo primero, será cargar todas las librerías necesarias, así como el dataset.

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values

X = dataset[:,0:13]
Y = dataset[:,13]
```

En este ejemplo se crearán modelos en Keras, y serán ajustados y evaluados utilizando la librería Scikit-learn. Esta librería Python open source de Machine Learning es bastante potente a la hora de evaluar modelos con pocas lineas de código.

El siguiente paso por tanto es crear el modelo de la red neuronal. Para el problema Boston-Housing basta con diseñar un modelo de red con una sola capa oculta totalmente conectada con el mismo numero de neuronas que de entradas (13). Como función de activación en la capa oculta se usa la función rectificadora. Para la capa de salida no se usa ninguna función

de activación debido a que nos encontramos ante un problema de regresión, y el objetivo es predecir valores numéricos directamente sin aplicarles ningún tipo de transformación. Como algoritmo de optimización usaremos `adam`, y como función de pérdida el error cuadrático medio. Estas métricas también serán usadas para evaluar el rendimiento del modelo. Se ha elegido esta métrica ya que al usar la raíz cuadrada, se nos presenta un valor de error directamente entendible en el contexto del problema (miles de dolares).

```
def baseline_model():
    model = Sequential()
    model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
    model.add(Dense(1, init='normal'))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

El wrapper de Keras para usar Scikit-Learn como un estimador de regresión se llama `KerasRegressor`. Podemos crear una instancia y pasarle como argumentos:

1. El nombre de la función que define la red neuronal (`baseline_model()`)
2. Parámetros que se le pasarán a la función `fit()`, como el número de ciclos o el tamaño de los lotes.

Como siempre, además se debe inicializar una semilla aleatoria constante para asegurarnos que la comparación de los modelos es consistente.

```
numpy.random.seed(4)
estimator = KerasRegressor(build_fn=baseline_model, nb_epoch=100, batch_size=5, verbose=0)
```

El siguiente paso es evaluar el modelo. Para ello, se usará una validación cruzada de 10 iteraciones.

```
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Resultados: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Ejecutando todo el código veremos una estimación del rendimiento del modelo. El resultado muestra el error cuadrático medio incluyendo la media y la desviación típica a lo largo de las 10 iteraciones de la evaluación por validación cruzada.

```
Resultados: 31.35 (25.45) MSE
```

4.2.3. Mejorando los resultados

Aunque ya hayamos conseguido entrenar nuestra red para obtener resultados, existen muchas maneras de mejorar nuestro modelo. En este ejemplo, veremos tres:

- Normalizar el dataset de entrada
- Aumentar el numero de capas
- Aumentar el numero de neuronas en las capas

Normalizando el dataset

Un factor muy importante en el dataset de Boston-housing es que los atributos de entrada varían mucho sus escalas, ya que miden cantidades diferentes. Por este tipo de razones, siempre suele ser una buena práctica preparar los datos antes de modelarlos usando nuestra red neuronal. Para ver la diferencia con esta mejora, se va a usar el mismo modelo de red neuronal que en el apartado anterior. Se va a usar el framework Pipeline de Scikit-Learn para estandarizar

el modelo de datos durante el proceso de evaluación dentro de cada pliegue de la validación cruzada. Esto asegura que no haya datos de nuestro set de entrenamiento sin evaluar en cada pliegue de la validación cruzada.

```
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=baseline_model, nb_epoch=50, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Estandarizado: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

En el código anterior, se crea un `Pipeline` de Scikit-Learn, donde primero se estandarizan los datos y luego se evalúa el modelo desarrollado en el apartado anterior. Al ejecutar todo el código, vemos una mejora en el rendimiento respecto al modelo sin estandarizar los datos.

Estandarizado: 30.78 (29.41) MSE

Red neuronal más profunda

Otra manera de mejorar nuestra red neuronal es añadirle capas, lo cual permitirá al modelo extraer más cantidad de características del dataset.

En este apartado se evaluará el efecto de añadir más de una capa al modelo. Ésto es tan fácil como crear una nueva función copia de la anterior, pero insertando nuevas líneas después de la primera capa oculta. En este ejemplo, cada capa tendrá la mitad de neuronas que la capa anterior.

```
def modelo_profundo():
    model = Sequential()
    model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
    model.add(Dense(6, init='normal', activation='relu'))
    model.add(Dense(1, init='normal'))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

Como podemos apreciar, los datos pasarán por una capa de entrada de 13 neuronas, después por una de 6, para acabar en una capa de salida de una sola neurona. La forma de evaluar el modelo será la misma que en el apartado anterior, usando la estandarización de los datos de entrada para mejorar aún más el rendimiento.

Ejecutando todo el código podemos ver otra importante mejora en el rendimiento respecto al apartado anterior:

Resultados: 24.10 (28.45) MSE

Red neuronal más ancha

Otra forma de incrementar la capacidad de nuestro modelo es crear un red más ancha. En este apartado se evaluará el efecto de añadir a nuestra arquitectura una capa oculta con el doble de neuronas de las que tiene la capa de entrada.

De nuevo, se ha de definir una nueva función como en los ejemplos anteriores, pero que cuente con una capa oculta entre la capa de entrada y la de salida de 20 neuronas.

```
def modelo_ancho():
    model = Sequential()
    model.add(Dense(20, input_dim=13, init='normal', activation='relu'))
    model.add(Dense(1, init='normal'))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

Como podemos ver, nuestros datos pasarán de una capa de entrada de 13 neuronas, a una capa oculta de 20, para luego acabar en una capa de salida de una sola neurona. De nuevo, la forma de evaluar el modelo será la misma que en los dos apartados anteriores, usando la estandarización de los datos de entrada para mejorar el rendimiento.

Ejecutando el código podemos ver una mejora aún mayor que la que se consiguió con la red neuronal más profunda. No es nada mal resultado para este tipo de problema:

Resultados: 21.66 (24.12) MSE

Como se ha comentado anteriormente, no es del todo fácil ver que con una red más profunda se consigan mejores resultados en este tipo de problemas. Esto demuestra la importancia de tener que testear varios tipos de modelos, para conseguir así los mejores resultados.

4.3. MNIST - Perceptrón multicapa vs Red convolucional

El dataset MNIST fue desarrollado con el objetivo de evaluar modelos de redes neuronales utilizando el problema de clasificación de dígitos escritos a mano. Este dataset está constituido por un gran número de documentos escaneados por el National Institute of Standards and Technology (NIST), a los cuales se le han extraído y normalizado sus caracteres. Cada imagen es de 28x28 píxeles, y se usan 60.000 imágenes para entrenar el modelo, y otras 10.000 para probarlo.

Estamos hablando de una tarea de reconocimiento de dígitos. Al haber 10 dígitos (del 0 al 9), hay 10 clases para clasificar. El estado del arte del error de predicción está en el 0,2%, que puede ser alcanzado con grandes redes neuronales convolucionales en las que se aplican técnicas avanzadas como DropConnect o aumento de patrones.

4.3.1. Cargando el dataset

Mediante la librería Keras, es posible descargarse el dataset de manera automática. Para ello, se llamará a la función `mnist.load_data()`, que almacenará la información en el directorio `~/keras/datasets/mnist.pkl.gz`.

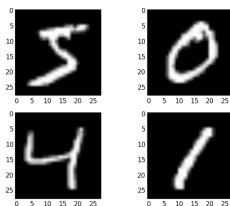


Figura 4.1: Patrones dataset MNIST

No es estrictamente necesaria una red neuronal convolucional para conseguir buenos resultados con el dataset MNIST. Mediante una red neuronal sencilla, con una sola capa oculta, podemos alcanzar un ratio de error del 1.74 %. Se usará este dato como base para comparar más modelos de redes convolucionales más adelante.

Lo primero de todo, como ya hemos visto, es cargar los módulos necesarios, generar una semilla aleatoria (aunque fija), y cargar el dataset.

El set de entrenamiento está estructurado como un array tridimensional compuesto por la imagen, su ancho y su alto. Para poder utilizarlo con un perceptrón, tenemos que convertir las imágenes en un vector de píxeles. En este caso, las imágenes de 28x28 se convertirán en 748 valores de entrada. Podemos hacer esto fácilmente con la función `reshape()` de NumPy.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils

numpy.random.seed(4)

n_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], n_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], n_pixels).astype('float32')
```

Los valores van de 0 a 255, en una escala de grises. Cuando se trabaja con modelos de redes neuronales, suele ser buena idea escalar o normalizar los datos de entrada. Para ello, se convertirán estos valores del rango 0-255 al 0-1.

Además, la salida será un entero entre 0 y 9 (ya que como hemos dicho, tenemos 10 clases en las que nuestra imagen puede ser clasificada). Nos encontramos ante un problema de multiclasiificación, por lo que será útil codificar el vector de enteros en una matriz binaria:

```
X_train = X_train / 255
X_test = X_test / 255

y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Con todo esto, ya se puede crear el modelo de la red neuronal. En este ejemplo, vamos a diseñar tanto un modelo de red neuronal basado en un perceptrón multicapa como un modelo convolucional, para ver sus diferencias y rendimientos a la hora de tratar imágenes.

Perceptrón multicapa

```
def perceptron_model():
    model = Sequential()
    model.add(Dense(n_pixels, input_dim=n_pixels, init='normal', activation='relu'))
    model.add(Dense(num_classes, init='normal', activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Como podemos observar, se trata de un modelo con una sola capa oculta, con el mismo número de neuronas que de entradas, es decir, 784. Además, sobre ella se aplica una función de activación rectificadora.

En la capa de salida, se usa la función de activación softmax para convertir las salidas en una probabilidad, y permitir que se seleccionen una de las 10 clases que el modelo ha de predecir. Como función de pérdida, se utiliza la pérdida logarítmica (en Keras, `categorical_crossentropy`). Por último, para que la red entrene y aprenda los pesos, se utiliza el algoritmo gradiente descendiente `adam`.

Red convolucional

```
def convolutional_model():
    model = Sequential()
    model.add(Convolution2D(32, 5, 5, init='glorot_uniform', border_mode='valid',
        input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Como podemos ver, las redes convolucionales son bastante más complejas que los perceptrones multicapa. En resumen, nuestro modelo cuenta con las siguientes capas:

1. La primera capa es una capa convolucional (`Convolution2D`). Cuenta con 32 mapeadores, de tamaño 5x5, una función de activación rectificadora, y un inicializador de tipo `glorot_uniform` (usado por defecto en las capas convolucionales de Keras). Al ser la capa de entrada, esperará imágenes con la forma `[[ípxeles][ancho][alto]]`.
2. Después se define una capa pooling que utilizará la función máximo, con un tamaño de 2x2.
3. La siguiente capa (`Dropout`) será regularizadora. Se ha configurado para excluir aleatoriamente el 20 % de las neuronas de la capa y así reducir el sobreajuste.
4. La siguiente capa (`Flatten`) convertirá la matriz de 2 dimensiones en un vector, permitiendo de esta forma que la salida pueda ser procesada por una capa totalmente conectada.
5. Después se ha añadido una capa totalmente conectada con 128 neuronas y una función de activación rectificadora.
6. Finalmente, la capa de salida tendrá 10 neuronas, una por cada clase a predecir, y una función de activación softmax para hacer una aproximación de la probabilidad de que cada entrada se corresponda con cada clase (es decir, con cada número).

Como en el ejemplo del perceptrón multicapa, el modelo se entrenará usando la función de pérdida logarítmica y el algoritmo `adam` de gradiente descendiente.

4.3.2. Resultados

Ahora ya podemos ajustar y evaluar los modelos. Estos se ajustan cada 10 ciclos, con actualizaciones cada 200 imágenes. Finalmente, se muestra el ratio de error de clasificación.

```
model = perceptron_model() // model.convolutional_model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
        verbose=2)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Ejecutando todo el código anterior utilizando el modelo del perceptrón multicapa, veremos la siguiente salida. Como podemos observar, esta red neuronal tiene un ratio de error de 1.88 %.

```
Epoch 1/10
10s - loss: 0.2792 - acc: 0.9206 - val_loss: 0.1358 - val_acc: 0.9615
Epoch 2/10
8s - loss: 0.1096 - acc: 0.9681 - val_loss: 0.0917 - val_acc: 0.9720
Epoch 3/10
8s - loss: 0.0704 - acc: 0.9799 - val_loss: 0.0727 - val_acc: 0.9780
...
Epoch 8/10
9s - loss: 0.0152 - acc: 0.9963 - val_loss: 0.0620 - val_acc: 0.9797
Epoch 9/10
10s - loss: 0.0104 - acc: 0.9978 - val_loss: 0.0598 - val_acc: 0.9821
Epoch 10/10
9s - loss: 0.0089 - acc: 0.9980 - val_loss: 0.0640 - val_acc: 0.9812
Classification Error: 1.88%
```

Sin embargo, si ejecutamos el código utilizando el modelo convolucional, tendríamos un ratio de error de 1.02 %, y la siguiente salida:


```
Epoch 1/10
162s - loss: 0.2484 - acc: 0.9289 - val_loss: 0.0799 - val_acc: 0.9773
Epoch 2/10
151s - loss: 0.0752 - acc: 0.9777 - val_loss: 0.0519 - val_acc: 0.9833
Epoch 3/10
151s - loss: 0.0531 - acc: 0.9841 - val_loss: 0.0376 - val_acc: 0.9883
...
Epoch 8/10
150s - loss: 0.0191 - acc: 0.9938 - val_loss: 0.0298 - val_acc: 0.9898
Epoch 9/10
156s - loss: 0.0173 - acc: 0.9945 - val_loss: 0.0306 - val_acc: 0.9901
Epoch 10/10
155s - loss: 0.0162 - acc: 0.9949 - val_loss: 0.0323 - val_acc: 0.9898
Classification Error: 1.02%
```

Como se puede observar, hemos reducido el error de clasificación del 1.88% al 1.02%. Pero... ¿podríamos obtener mejores resultados? La respuesta es sí. En este ejemplo, se ha utilizado una red convolucional muy simple. Sin embargo, añadiendo más capas convolucionales, de pooling y totalmente conectadas, podemos acercarnos a ratios de error muy cercanos al estado del arte actual. Por ejemplo, añadiendo una nueva capa convolucional (`model.add(Convolution2D(15, 3, 3, activation='relu'))`), `model.add(MaxPooling2D(pool_size=(2, 2)))`, y una nueva capa totalmente conectada (`model.add(Dense(50, activation='relu'))`), obtendríamos un ratio de error de 0.89%.

4.4. Generación de textos - Red LSTM

En este ejemplo, se va a intentar generar textos que más o menos tengan sentido. Hasta la aparición de las redes neuronales, que un ordenador pudiese llevar a cabo esta tarea parecía una idea sacado de una película de ciencia ficción. Sin embargo, con las redes LSTM esto empieza a ser posible.

Las redes neuronales, además de ser usadas para predecir modelos, pueden ser utilizadas para aprender secuencias de un problema y generar mediante estos conocimientos nuevas secuencias. Los modelos que generan nuevas secuencias como éste, no solo son útiles para saber como de bien nuestra máquina ha aprendido un problema, sino también para saber más sobre un problema en sí mismo.

En este ejemplo, se va a utilizar el libro "La metamorfosis" de Kafka como dataset para nuestra red neuronal. Ésta, va a aprender las dependencias entre los diferentes caracteres que se vaya encontrando a lo largo del texto así como las probabilidades de que estos caracteres aparezcan, con el objetivo de generar una nueva secuencia de caracteres, formando así un nuevo texto y una nueva historia. Sin embargo, este ejemplo también es válido para otro tipo de textos: poemas, código fuente... (siempre que éstos se encuentren en código ASCII).

4.4.1. Aprendiendo secuencias de caracteres

Primero se importan las clases que se van a utilizar en el problema, y se carga el texto convirtiéndolo todo a minúsculas.

```
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils

filename = "metamorfosis.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
```

Una vez cargado el libro en memoria, debemos prepararlo para ser tratado por una red neuronal. Una buena idea es convertir los caracteres a números enteros. Para ello, primero se identifican todos los caracteres que componen el texto y se asocia un número a cada uno. Con ello, se consigue reducir el número de caracteres a analizar, ya que también se eliminarán los repetidos.

```
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
```

El siguiente paso es definir el conjunto de entrenamiento. Se dividirá el texto en conjuntos de caracteres (en este ejemplo 100), para pasárselos como entrada a la red neuronal. El objetivo es que la red prediga el carácter 101. Esto se conseguirá desplazando de 1 en 1 los 100 caracteres seleccionados, permitiendo así que cada carácter sea aprendido por los 100 caracteres que le preceden. Por ejemplo, si hiciésemos divisiones de 5 en 5, y tuviésemos la palabra CAPITULO, las iteraciones serían:

```
CAPIT -> U
APITU -> L
PITUL -> O
```

Dado que las redes neuronales trabajan con números en vez de caracteres, debemos convertir estos caracteres a enteros usando la tabla que generamos anteriormente.

```
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
```

Si ejecutamos el código hasta este punto, podemos ver el número de patrones de entrenamiento:

```
121472
```

Una vez que hemos preparado el dataset, necesitamos transformarlo para que pueda ser usado por Keras. Primero, debemos transformar la lista de entradas (`seq_in`) en una secuencia de la forma [muestra, intervalo de tiempo, ícaracterística], esperada por una red LSTM. Después, se necesita escalar los enteros al rango 0-1 para conseguir patrones más fáciles de aprender por la red LSTM que usa la función de activación sigmoide. Finalmente, se convertirán los patrones de salida. Para ello, se representará la salida como una probabilidad de que aparezca cada uno de los 43 caracteres distintos que componen el texto, en vez de intentar predecir estrictamente el siguiente carácter. Cada valor será convertido a un vector de longitud 43, relleno de 0 menos un 1 que concidirá con la columna de la letra que el patrón representa. Por ejemplo, si la letra fuese la C (entero número 3), la codificación sería... [0,0,1,0,0,0,0...0] Esto se consigue mediante las siguientes líneas de código.

```
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
X = X / float(n_vocab)
y = np_utils.to_categorical(dataY)
```

Una vez que tenemos los datos preparados, es hora de definir nuestra red LSTM. Para este ejemplo, se definirán dos capas oculta LSTM con 256 unidades de memoria. Se usa la técnica de regularización para evitar el sobreajuste conocida como Dropout, con una probabilidad del 20%. La capa de salida será una capa `lstmInlineDense` que usará `softmax` como función de activación para producir una salida en función de la probabilidad de que aparezca uno de los 43 caracteres. En realidad, podemos observar que el problema es realmente un problema de clasificación con 43 clases, y por ello se define la función de pérdida logarítmica `cross entropy` y se aplica el algoritmo de optimización `adam` para mejorar la velocidad.

```
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', %optimizer='adam')
```

En este ejemplo, el interés no está en la precisión de la clasificación, ya que se crearía un modelo que predeciría cada carácter del conjunto de entrenamiento perfectamente. En vez de eso, estamos interesados en una generalización del dataset que minimice la función de pérdida. Es decir, estamos buscando un balance entre generalización y sobreajuste.

Esta red es lenta de entrenar. Por ello, se ha decidido usar una serie de checkpoints para almacenar todos los pesos de la red en un fichero cada vez que se observe una mejora en la pérdida, al final de cada ciclo. Usaremos los mejores pesos (menores pérdidas) para generar nuestro modelo en el siguiente punto.

```
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True, mode='min')
callbacks_list = [checkpoint]
```

Por último, ya solo queda ajustar nuestro modelo. Usaremos 50 ciclos y una tamaño de lote de 64 patrones.

```
model.fit(X, y, nb_epoch=50, batch_size=64, callbacks=callbacks_list)
```

Cada vez que se ejecute el modelo, nos encontramos ante diferentes valores. Esto es debido a la naturaleza aleatoria del modelo y a que es difícil elegir una semilla aleatoria para las redes LSTM que reproduzcan los resultados con un 100x100 de exactitud. A pesar de ello, éste no es el objetivo del modelo.

Una vez ejecutado el script completo, se deben haber generado una serie de archivos checkpoint con los mejores pesos de cada ciclo. Para ejecutar el código del siguiente apartado donde ya se generarán nuevas secuencias de caracteres, se usarán los valores del ultimo archivo ya que son los que tienen un menor valor de pérdida.

4.4.2. Generando secuencias de caracteres

Una vez entrenada nuestra red, se han cargado los pesos que se han calculado y guardado en nuestro checkpoint.

```
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Además, también se ha de crear un mapeo inverso de caracteres a enteros para convertir de nuevo los enteros utilizados en la red a caracteres.

```
int_to_char = dict((i, c) for i, c in enumerate(chars))
```

Por último, solo queda hacer las predicciones. La manera más simple es iniciar con una secuencia de semillas como entrada, generar el siguiente carácter, después actualizar la secuencia de semillas para añadir el carácter generado al final, y quitar el primer carácter. Este proceso se repetirá mientras queramos generar nuevos caracteres. Elegiremos un patrón aleatorio de entrada como nuestra secuencia de semillas, y se mostrarán los caracteres según se vayan generando.

```
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print "Semilla:"
print "%s" % ''.join([int_to_char[value] for value in pattern]), "%s"
```

```
for i in range(1000):
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]

print "\nFin."
```

4.4.3. Analizando resultados

Debido a la carga computacional, las pruebas de este apartado han sido llevadas a cabo en una máquina de tipo g2.2xlarge de Amazon Web Services, una instancia optimizada para aplicaciones con uso intensivo de gráficos. Gracias a esta instancia, que destaca por su gran GPU, se consiguieron llevar a cabo las pruebas en un tiempo relativamente aceptable, consiguiendo aproximadamente que cada uno de los cincuenta ciclos se llevase a cabo en diez minutos.

Una vez que se ha entrenado la red, al ejecutar el código que genera textos con los pesos ya aprendidos, se genera el siguiente texto:

```
Total Characters: 121492
Total Vocab: 46
Total Patterns: 121392
Seed:
decisiones desesperadas. en tales momentos dirigia sus ojos lo mas agudamente posible hacia
la vent "
ana y la hermana se habia acierto la puerta del cuarto de estar hacia el senor de estar con
la habitacion de gregorio. el padre se despar de la cabeza con la mano derecha
comenzarse a la madre que estaba en la cama en la habitacion de gregorio.
```

Como podemos ver, los resultados no son perfectos. Hay palabras que sí han sido generadas correctamente (como hermana), y otras que sin embargo, no tienen mucho sentido en el texto (como despar). Aunque el texto en su conjunto no tenga mucho sentido, podemos apreciar varios detalles:

- El primero, es que nuestra red ha sido capaz de predecir la palabra que se ha quedado a la mitad en la semilla (ventana).
- En segundo lugar, vemos como nuestra red utiliza la mayor parte de las veces las palabras que más se repiten en el libro. Para los que no lo hayan leído, los personajes principales son Gregorio, su padre, su madre, y su hermana, y la mayor parte de la historia transcurre en la habitación de Gregorio. Esto demuestra que la red ha aprendido de nuestro texto para poder generar el suyo propio. Si se quisiese un vocabulario más extenso, una buena opción sería nutrir a nuestra con de más textos e historias.
- Por último, pese a que el texto en su conjunto no tenga mucho sentido, podemos apreciar como nuestra red parece que ha aprendido reglas gramaticales del español. Por ejemplo, construye las frases con la estructura sujeto, verbo y predicado, y es capaz de utilizar pronombres, adverbios... para dar más sentido a las frases.

No obstante, existen una serie de técnicas que ayudarían a mejorar nuestro nuevo texto, pero que en este ejemplo no han sido aplicadas. Algunas de ellas son:

- Predecir menos de 1000 caracteres de salida por semilla.
- Borrar todos los signos de puntuación y por tanto, del vocabulario del modelo.
- Añadir más capas al modelo.

- Reducir el tamaño de lote (el más eficiente sería de 1, pero el modelo tardaría demasiado en entrenarse)

Conclusiones y líneas futuras

Como se ha podido comprobar, el uso de las redes neuronales en ámbitos comunes ya es una realidad. Este TFG se empezó describiendo varias ideas que parecían más bien sacadas de una película de ciencia ficción, pero como se ha podido observar, estamos muy cerca de conseguirlas. Si seguimos rompiendo barreras en cuanto a capacidad de computación se refiere, y adaptamos las nuevas tecnologías de procesamiento en GPU y CPU a nuestras redes neuronales, seremos capaces de llegar a resultados hasta hace unos años inimaginables. A día de hoy, el mundo de las redes neuronales es una de las mayores líneas de investigación tanto en informática como en el estudio y recreación del cerebro humano, y merece la pena entender su funcionamiento básico.

Siguiendo la línea de investigación, los siguientes pasos serían

Glosario de acrónimos

- **LSTM:** Long Short-Term Memory
- **RNN:** Recurrent Neural Network
- **FNN:** Feed-forward Neural Network

Bibliografía



Códigos fuente

A.1. 4.1. Problema de diabetes - Regresión logística

```
from keras.models import Sequential
from keras.layers import Dense
import numpy
# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)
# carga del dataset de diabetes en indios americanos
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# se separa el dataset en X (entrada) e Y (salida)
X = dataset[:,0:8]
Y = dataset[:,8]
# creacion del modelo
model = Sequential()
model.add(Dense(12, input_dim=8, init='normal', activation='relu'))
model.add(Dense(8, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
# compilacion del modelo
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# ajuste del modelo
model.fit(X, Y, nb_epoch=150, batch_size=10)
# evaluacion del modelo
eva = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], eva[1]*100))

# calculo de predicciones
predicciones = model.predict(X)
# redondeo de predicciones
rounded = [round(x) for x in predicciones]
print(rounded)
```

A.2. 4.2. Problema Boston Housing - Regresión lineal

A.2.1. Modelo estándar

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# definicion del modelo
def baseline_model():
    # creacion del modelo
```

```
model = Sequential()
model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
model.add(Dense(1, init='normal'))
# compilacion del modelo
model.compile(loss='mean_squared_error', optimizer='adam')
return model

# carga del dataset de boston-housing
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# se separa el dataset en las variables X (entradas) e Y (salida)
X = dataset[:,0:13]
Y = dataset[:,13]
# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)
# evaluacion del modelo
estimator = KerasRegressor(build_fn=baseline_model, nb_epoch=100, batch_size=5, verbose=0)

kfold = KFold(n_splits=10, random_state=4)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Resultados: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

A.2.2. Modelo con datos de entrada normalizados

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# definicion del modelo de red neuronal
def baseline_model():
    # ócreacin del modelo
    model = Sequential()
    model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
    model.add(Dense(1, init='normal'))
    # compilacion del modelo
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# carga del dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# se separa el dataset en las variables X (entradas) e Y (salida)
X = dataset[:,0:13]
Y = dataset[:,13]
# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)
# se úevala el modelo con el dataset normalizado
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=baseline_model, nb_epoch=50, batch_size=5,
    verbose=0)))
pipeline = Pipeline(estimators)

kfold = KFold(n_splits=10, random_state=4)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Estandarizado: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

A.2.3. Modelo más profundo

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
```

```
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Definicion del modelo de red neuronal mas profundo
def larger_model():
    # Creacion del modelo
    model = Sequential()
    model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
    model.add(Dense(6, init='normal', activation='relu'))
    model.add(Dense(1, init='normal'))
    # Compilacion del modelo
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# carga del dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# se separa el dataset en las variables X (entradas) e Y (salida)
X = dataset[:,0:13]
Y = dataset[:,13]
# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=larger_model, nb_epoch=50, batch_size=5,
                                         verbose=0)))
pipeline = Pipeline(estimators)

kfold = KFold(n_splits=10, random_state=4)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Resultados: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

A.2.4. Modelo más ancho

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Definicion del modelo de red neuronal mas ancho
def wider_model():
    # Creacion del modelo
    model = Sequential()
    model.add(Dense(20, input_dim=13, init='normal', activation='relu'))
    model.add(Dense(1, init='normal'))
    # Compilacion del modelo
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# carga del dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# se separa el dataset en las variables X (entradas) e Y (salida)
X = dataset[:,0:13]
Y = dataset[:,13]
# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=wider_model, nb_epoch=100, batch_size=5,
                                         verbose=0)))
pipeline = Pipeline(estimators)

kfold = KFold(n_splits=10, random_state=4)
```

```
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Resultados: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

A.3. 4.3 MNIST - Perceptrón multicapa vs Red convolucional

A.3.1. Perceptrón multicapa

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils

# Definicion del modelo de red neuronal
def baseline_model():
    # Creacion del modelo
    model = Sequential()
    model.add(Dense(num_pixels, input_dim=num_pixels, init='normal', activation='relu'))
    model.add(Dense(num_classes, init='normal', activation='softmax'))
    # Compilacion del modelo
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)

# carga del dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# -----flatten 28*28 images to a 784 vector
    for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')

# ónormalizacin de los valores de entrada de 0-255 a 0-1
X_train = X_train / 255
X_test = X_test / 255

# -----one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]

model = baseline_model()
# Ajuste del modelo
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
        verbose=2)
# Evaluacion del modelo
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

A.3.2. Red convolucional

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')

def baseline_model():
    # Creacion del modelo
    model = Sequential()
```

```
model.add(Convolution2D(32, 5, 5, border_mode='valid', input_shape=(1, 28, 28),
    activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
# Compilacion del modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return model

# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
numpy.random.seed(4)

# Carga del dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')

# ónormalizacin de los valores de entrada de 0-255 a 0-1
X_train = X_train / 255
X_test = X_test / 255
# -----one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]

model = baseline_model()
# Ajuste del modelo
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Evaluacion del modelo
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

A.4. 4.4 Generación de textos - Red LSTM

A.4.1. Entrenamiento de la red

```
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# Se carga el texto y se convierten todos los caracteres a úminscula
filename = "kafka.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# Creacion del mapeo caracter-entero
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# Resumen del dataset de entrada
n_chars = len(raw_text)
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
# -----prepare the dataset of input to
    output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
```

```
# -----reshape X to be [samples, time steps,
    features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# óNormalizacin de los datos de entrada
X = X / float(n_vocab)
# -----one hot encode the output variable
y = np_utils.to_categorical(dataY)
# Definicion del modelo de red neuronal LSTM
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# Definicion de los checkpoint (punto de guardado de pesos)
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True, mode='
    min')
callbacks_list = [checkpoint]
# Ajuste del modelo
model.fit(X, y, nb_epoch=50, batch_size=64, callbacks=callbacks_list)
```

A.4.2. Generacion de texto

```
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# Se carga el texto y se convierten todos los caracteres a úmínskula
filename = "kafka.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# Creacion del mapeo caracter-entero, y su inverso
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# Resumen del dataset de entrada
n_chars = len(raw_text)
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
# -----prepare the dataset of input to
    output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# óNormalizacin de los datos de entrada
X = X / float(n_vocab)
# -----one hot encode the output variable
y = np_utils.to_categorical(dataY)
# Definicion del modelo de red neuronal LSTM
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# Carga de los pesos de la red a partir de los checkpoints generados previamente
```

```
filename = "weights-improvement-47-1.2219-bigger.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# se elige una semilla aleatoria para reproducir el mismo ejemplo varias veces
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print "Seed:"
print "\"", ''.join([int_to_char[value] for value in pattern]), "\""
# Generacion de caracteres
for i in range(1000):
    x = numpy.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
```