



UNIVERSIDAD DE GRANADA

REDUCCIÓN DE LA DIMENSIONALIDAD EN PROBLEMAS DE CLASIFICACIÓN CON DEEP LEARNING

ANÁLISIS Y PROPUESTA DE HERRAMIENTA EN R

Trabajo Fin de Grado
Doble Grado en Ingeniería Informática y Matemáticas

Autor
Francisco David Charte Luque

Tutor
Francisco Herrera Triguero

FACULTAD DE CIENCIAS
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, Junio de 2017

Reducción de la dimensionalidad en problemas de clasificación con Deep Learning © Junio de 2017 Francisco David Charte Luque

LICENCIA

Esta obra está sujeta a la licencia Reconocimiento-CompartirIgual 4.0 Internacional de Creative Commons¹.

La licencia permite:

Compartir — copiar y redistribuir el material en cualquier medio o formato.

Adaptar — remezclar, transformar y crear a partir del material para cualquier finalidad, incluso comercial.

Bajo las condiciones siguientes:

Reconocimiento — Debe reconocer adecuadamente la autoría, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puede hacerlo de cualquier manera razonable, pero no de una manera que sugiera que tiene el apoyo del licenciador o lo recibe por el uso que hace.

CompartirIgual — Si remezcla, transforma o crea a partir del material, deberá difundir sus contribuciones bajo la misma licencia que el original.

No hay restricciones adicionales — No puede aplicar términos legales o medidas tecnológicas que legalmente restrinjan realizar aquello que la licencia permite.

¹ Texto completo de la licencia disponible en <https://creativecommons.org/licenses/by-sa/4.0/legalcode>.

RESUMEN

En este trabajo se analizan desde una perspectiva teórica las técnicas basadas en redes neuronales profundas que permiten abordar el problema de reducción de la dimensionalidad y se explica el software desarrollado, que permite el uso de dichas técnicas y la generación de visualizaciones sobre ellas, bien mediante programación o bien a través de una interfaz gráfica de usuario web.

Primero se introducen conceptos matemáticos que ayudan a comprender los algoritmos y modelos que fundamentan estas redes profundas, haciendo hincapié en resultados teóricos que dan ideas acerca del problema que se va a abordar. Posteriormente, se describen los algoritmos y que realizan el aprendizaje sobre dichas estructuras, y las arquitecturas relevantes que tratan este problema. Por último, se documenta el software implementado y se muestran ejemplos de su uso.

PALABRAS CLAVE Deep Learning, redes neuronales, reducción de dimensionalidad, clasificación, aprendizaje no supervisado, probabilidad, teoría de la información.

ABSTRACT

This work studies, from a theoretical perspective, techniques based on deep neural networks that tackle the high dimensionality problem. It also explains the piece of software developed for this project, which allows the use of said techniques and includes resources for visualization, offering both a programming interface and a web-based graphic user interface.

DESCRIPTION OF THE ADDRESSED PROBLEM

The current trend in data collection from diverse sources for its subsequent processing implies the need for powerful learning algorithms as well as high computation abilities. One of the most common tasks is classification, where an algorithm attempts to predict one or several labels tied to data samples, having previously learned from already classified examples.

Classification algorithms often suffer some performance loss when trained against high dimensional data sets, a phenomenon known as the *curse of dimensionality*. From the very varied approaches to tackle this problem, this work focuses on dimensionality reduction via unsupervised deep neural networks. Neural networks generalize the perceptron, and deep networks are an extension of this concept. It was defined decades ago but has made a comeback thanks to the progress on high performance computing and efficient training algorithms.

Techniques of this kind can be found in software for languages such as Python or C++, whereas the data-oriented language R lacks a library offering easy access to unsupervised deep neural networks. The software implemented in this project, Ruta, aims to resolve this absence. Furthermore, a companion tool named Rutavis adds visualization mechanisms and a web-based graphic user interface.

The documentation is divided into two distinct parts. One offers a mathematical framework which serves as a basis for the concepts needed to describe the Deep Learning techniques related to this work. The areas of mathematics used are probability theory, information theory and tensor algebra. The other main part describes the common machine learning notions, problems and algorithms that also apply to Deep Learning and continues with the specification of the techniques used in unsupervised deep neural networks. It finishes by describing the design and implementation of both pieces of software, Ruta and Rutavis.

The following sections constitute a summary of this documentation.

MATHEMATICAL FOUNDATIONS OF DEEP LEARNING

Deep Learning finds its roots in several areas of mathematics. Especially, there are important concepts to be studied in probability theory, information theory and tensor algebra.

The needed definitions from probability theory are very basic but essential to the rest of the text, so special attention is given to rigorously compile them. The ideas of probability distribution, conditional distribution, independence and the main moments (expectation and variance) are explained. As a preliminary result, the theorem of the continuous mapping is formulated. It is then used to prove one of the main theoretical results of this work, a theorem describing the *curse of dimensionality* that motivates the task known as dimensionality reduction. As a consequence it is deduced that, as the dimensionality of a data set increases, the difference between the nearest neighbor and the farthest one becomes insignificant.

The next chapter introduces the convenient notions around entropy. The entropy of a random variable is defined and several properties are deduced. Then, measures of the information involved among two variables are presented as the joint entropy and conditional entropy. These are used to infer the chain rule of entropy. Later, two important concepts are defined: the Kullback-Leibler divergence and cross entropy, which will be applied later to the construction of objective functions in deep neural networks. Some properties and alternative expressions of these concepts are verified. Lastly, Jensen's inequality, which is a well-known result among the mathematical field, is used to prove the information inequality and its consequences. Intuitively, it is deduced that probabilistic models assumed around sampled data need to be as close as the true probability distribution as possible in order to be able to optimize the compression of data.

After that, a chapter is dedicated to concepts around the notion of tensor. The mathematical foundations of tensors rely on multilinear algebra, which is explained starting from linear functionals and dual spaces, then introducing the mechanics of the change of coordinates within these spaces. The second dual is shown to be trivially isomorphic to the origin vector space. These results allow the definition of multilinear mappings and, in particular, multilinear functionals or tensors. The tensor product of functionals is introduced and later used to define a base for the tensor product of vector spaces. The distinction between covariant and contravariant tensors is explained. Finally, the use of tensors in machine learning is shown to be a distant application of the previous definitions.

Deep Learning is considered a branch of machine learning. Thus, a notable amount of theoretical content and algorithms can be applied to the problem studied. However, many of the latest developments on Deep Learning are novel and deserve a thorough study.

The essential notions of learning and learners are introduced and several common learning tasks are enumerated. The two main types of learning, supervised and unsupervised, are distinguished. Some examples are also provided. Later, the classification problem is presented, and a theoretical formulation with a simple notation is introduced, which relates to the field of PAC learning. Different types of classification problems are enumerated as well. The structure of the space where the data belongs is lightly discussed afterwards.

The main problem addressed in this work, the dimensionality reduction task, is described and connected to the theoretical results proved before. Several means of action that tackle this problem are explained, and the approach studied in the work is highlighted. There is also a breakdown of the process referred to as feature extraction, and some mechanisms to build new features are mentioned. Lastly, one of the most common optimization algorithms in machine learning, gradient descent, is thoroughly described. It will serve as a basis for the optimization methods in Deep Learning.

Afterwards, the chapter dedicated to Deep Learning begins by outlining the common structure used in many of its applications, the deep feedforward neural network. A mathematical notation is introduced and the biological inspiration behind the artificial neuron is discussed. Later, an introduction to the way cost functions are derived from the selected probabilistic model is made. Different types of output units or neurons are explained, as well as the cost function they determine. Especially, the most commonly found ones are described: linear, sigmoid and softmax units. This explanation is extended to all hidden units in a feedforward network by adding rectified linear units and their variants, as well as the hyperbolic tangent and other activation functions.

The training process of a deep neural network requires some special techniques, known as forward propagation and backward propagation. These algorithms are motivated and thoroughly depicted, and a small example on the computation of gradients is made. The algorithms derived from gradient descent, specific for training deep networks, analyzed in this work are stochastic gradient descent and its variants AdaGrad, RMSProp and Adam. These allow to optimize the cost functions via several iterations where a minibatch of data is propagated through the network and gradients are computed.

Deep neural networks can be designed in a specific means to be trained in an unsupervised fashion. The main architectures that allow this kind of training are restricted Boltzmann machines and autoencoders. These are characterized and illustrated in the present work, paying special attention to some variants of the autoencoder and their properties: undercomplete, sparse, denoising and contractive autoencoders are all specified in quite detail. Finally, a special training procedure for autoencoders involving a stack of restricted Boltzmann machines is explained.

SOFTWARE IMPLEMENTATION: THE RUTA PACKAGE

During the realization of this project, two pieces of software have been designed and implemented. One of them, named Ruta, gives uncomplicated access to unsupervised deep neural networks, from building their architecture to their training and evaluation. The second one is a complementary project called Rutavis, which allows to generate graphical representations of the models trained with Ruta.

First, an introduction to the R language is offered, and simple instructions for its installation are provided. There is also a general description of the language and the object orientations it admits. Afterwards, a neural network library called MXNet is introduced. It implements the necessary operations and algorithms needed to build and train deep architectures. Instructions for its installation are provided as well as a specification of the programming mechanic it offers, including a complete example of the training and prediction of a simple neural network oriented to regression.

The Ruta software is presented and motivated. It is mentioned that an early prototype was previously introduced at a national conference. Later, there is an exhaustive description of its structure and functionality. The main objects used to work with this package abstract the notions of learning tasks, learners and trained models. These objects are defined and exemplified making use of the Iris data set and a simple deep autoencoder.

The Rutavis package is also outlined via the same main points, its structure and functionality. Some example plots are shown and screenshots of an use case with the web-based graphic user interface are offered as well.

The document ends with some conclusions and sketches some future work.

KEYWORDS Deep Learning, neural networks, dimensionality reduction, classification, unsupervised learning, probability, information theory.

Yo, **Francisco David Charte Luque**, alumno de la titulación Doble Grado en Ingeniería Informática y Matemáticas de la **Facultad de Ciencias** y de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación** de la **Universidad de Granada**, con DNI XXXXXXXXXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca de ambos centros para que pueda ser consultada.

Granada, 26 de junio de 2017

Francisco David Charte
Luque

D. **Francisco Herrera Triguero**, Profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado *Reducción de la dimensionalidad en problemas de clasificación con Deep Learning*, ha sido realizado bajo su supervisión por **Francisco David Charte Luque**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expido y firmo el presente informe en Granada a 26 de junio de 2016.

Francisco Herrera Triguero

AGRADECIMIENTOS

Me gustaría agradecer a la Universidad de Granada, a la Facultad y la Escuela y en concreto a sus docentes por brindarme la oportunidad de formarme. Especialmente a mi tutor, Francisco Herrera, por su apoyo a lo largo de este proyecto.

Muchas gracias a mi familia por su constante ayuda a lo largo de estos años, y a mis amistades que de una u otra forma me han animado a seguir adelante y de las cuales he aprendido tanto.

ÍNDICE GENERAL

I INTRODUCCIÓN	1
1 DESCRIPCIÓN	2
1.1 Contextualización	2
1.2 Descripción del problema	2
1.3 Estructura del trabajo	4
1.4 Bibliografía fundamental	4
2 OBJETIVOS	6
II MATEMÁTICAS	8
3 PROBABILIDAD Y DIMENSIONALIDAD	9
3.1 Recordatorio de conceptos	9
3.2 Resultados de convergencia	12
3.3 La maldición de la dimensionalidad	13
4 TEORÍA DE LA INFORMACIÓN	16
4.1 Entropía. Propiedades y magnitudes	16
4.2 Entropía conjunta y entropía condicional	17
4.3 Entropía relativa, información mutua y entropía cruzada	18
4.4 La desigualdad de la información	20
5 ÁLGEBRA TENSORIAL	23
5.1 Espacios duales	23
5.2 Funciones multilineales. Tensores	24
5.3 Productos tensoriales	26
5.4 Tensores covariantes y contravariantes	27
5.5 Los tensores en aprendizaje automático	27
III INFORMÁTICA	29
6 APRENDIZAJE AUTOMÁTICO	30
6.1 Introducción	30
6.2 Tipos de aprendizaje	30
6.3 Problema de clasificación	32
6.4 Problema de reducción de dimensionalidad	33
6.5 Métodos de optimización: gradiente descendente	34
7 DEEP LEARNING	37
7.1 Redes neuronales prealimentadas profundas	37
7.2 Entrenamiento de redes neuronales profundas	45
7.3 Optimización en Deep Learning	48
7.4 Estructuras profundas no supervisadas	52
8 LA HERRAMIENTA RUTA	58
8.1 El lenguaje R	58
8.2 La biblioteca MXNet	60
8.3 Introducción a Ruta	62

8.4 Componentes del paquete y uso	64
8.5 Visualización con Rutavis	68
IV CONCLUSIONES	74
9 CONCLUSIONES Y VÍAS FUTURAS	75
V APÉNDICE	76
A MANUAL DE USUARIO DE RUTA	77
A.1 Documentación para el paquete Ruta en la versión 0.2.0	77
A.2 Documentación para el paquete Rutavis en la versión 0.2.0	82
BIBLIOGRAFÍA	83

Parte I

INTRODUCCIÓN

DESCRIPCIÓN

1.1 CONTEXTUALIZACIÓN

En este trabajo se estudian las principales técnicas orientadas a reducción de la dimensionalidad en el campo del Deep Learning y se presentan dos herramientas software dedicadas al uso de dichas técnicas y al análisis de su comportamiento mediante visualizaciones.

Las tecnologías utilizadas se enmarcan dentro del ámbito del Deep Learning no supervisado, y permiten tratar el problema de reducción de la dimensionalidad. Las técnicas de Deep Learning se plasman en forma de redes neuronales profundas, un tipo de redes neuronales artificiales, que son a su vez una generalización del perceptrón. Pese a que el desarrollo teórico de las redes profundas se remonta varias décadas atrás en el tiempo [47] [35] su utilización era prácticamente inviable debido al coste computacional que implicaba. Sin embargo, el Deep Learning ha experimentado un resurgimiento en los últimos años debido a la creciente capacidad de procesamiento disponible, especialmente en dispositivos GPU, y al desarrollo de algoritmos de entrenamiento mucho más eficientes [24] [25].

El problema de reducción de la dimensionalidad surge a su vez como resultado del tratamiento de tareas de minería de datos, especialmente de clasificación, donde el conjunto de datos considerado posee un alto número de dimensiones. Estos dos problemas pertenecen al aprendizaje automático, una rama de la inteligencia artificial.

Para estudiar los fundamentos de los problemas mencionados y las técnicas con las que los abordamos, son necesarios conocimientos de varias áreas de las matemáticas, concretamente la teoría de la probabilidad, la teoría de la información y el álgebra tensorial. De ellas se extraen conceptos y resultados teóricos que dan fundamento a los algoritmos que se aplican para construir y entrenar las estructuras profundas.

1.2 DESCRIPCIÓN DEL PROBLEMA

En la actualidad, la ingente recolección de datos de distintas fuentes para su posterior procesamiento requiere de potentes algoritmos que reconozcan patrones y extraigan información útil, así como de grandes capacidades de cómputo para su ejecución. En particular, uno de los problemas frecuentemente tratados es el de clasificación, que con-

siste en la predicción de una o varias etiquetas asociadas a muestras de datos, habiendo aprendido previamente a partir de ejemplos ya etiquetados. Existe una gran variedad de algoritmos disponibles para abordar este problema [33].

En muchos casos, los datos recogidos presentan una alta dimensionalidad, en el sentido de que cada muestra determina valores sobre un alto número de variables. La reducción de la dimensionalidad es una tarea de preprocesamiento utilizada para mejorar el rendimiento de los algoritmos que tratan problemas de clasificación frente a conjuntos de datos de este tipo, y ha sido tratada desde distintas perspectivas [20].

El enfoque que se explora en este texto es el basado en estructuras de Deep Learning no supervisado. El aprendizaje profundo o Deep Learning es una rama del aprendizaje automático donde se hace uso de redes compuestas por muchas capas por las que se propagan y transforman los datos. Existen diversas implementaciones que permiten construir y entrenar dichas redes, pero normalmente se dirigen a lenguajes como Python o C++ [1] [5] [30].

En el caso del lenguaje R, algunas de las técnicas más relevantes de Deep Learning no supervisado están disponibles ya en paquetes como h2o [3], deepnet [45] o darch [16]. Sin embargo, hasta ahora ninguna herramienta para R se ha centrado en ser exhaustiva con las estructuras no supervisadas, ni ha incluido mecanismos de visualización que faciliten entender el comportamiento de estas redes o los modelos generados por ellas.

El software desarrollado en este TFG, denominado Ruta, trata de llenar este hueco suministrando una serie de funcionalidades para la creación, entrenamiento y evaluación de técnicas de Deep Learning no supervisado de muy fácil uso. Está basado en la biblioteca de operaciones y algoritmos para redes neuronales MXNet [12]. El proyecto complementario Rutavis, descrito en la sección 8.5, se encarga de proporcionar una interfaz gráfica de usuario desde la que entrenar modelos y generar visualizaciones que expliquen su comportamiento.

Para entender el funcionamiento de las redes profundas, en este trabajo se estudian los conceptos matemáticos que se utilizan para desarrollarlas de forma teórica y para justificar el problema que abordamos. Asimismo, se analizan los algoritmos que optimizan los parámetros de estas redes, y las estructuras que permiten realizar aprendizaje no supervisado sobre los datos. Por último, la herramienta desarrollada permite poner en práctica estos conocimientos y trata de resolver la escasez de utilidades que proporcionan estas funcionalidades.

1.3 ESTRUCTURA DEL TRABAJO

Este trabajo se divide en dos partes diferenciadas:

1. Matemáticas: se exponen los conceptos de probabilidad y teoría de la información que son la base de las redes profundas y se motiva el problema que abordamos de forma teórica. Además, se introducen los conceptos que originan el uso de tensores en las implementaciones existentes.
2. Informática: se explican los conceptos básicos de aprendizaje automático y se desarrollan los problemas de clasificación y de reducción de dimensionalidad, se analizan los algoritmos que buscan soluciones en estructuras profundas y se documenta la herramienta desarrollada, describiéndose su funcionalidad.

En la parte de matemáticas, el [Capítulo 3](#) recuerda conceptos de probabilidad utilizados a lo largo del texto, enuncia unos resultados esenciales sobre convergencia y deduce un teorema que da fundamento al problema abordado. El [Capítulo 4](#) define la entropía y varios conceptos asociados relevantes, y expone un resultado interesante, la desigualdad de la información. Por último, el [Capítulo 5](#) es una introducción al álgebra de tensores, idea que se aplica en herramientas de Deep Learning.

Por otro lado, la parte de informática analiza el problema estudiado y las técnicas usadas para tratarlo. El [Capítulo 6](#) es una introducción al aprendizaje automático con énfasis en los problemas de clasificación y reducción de la dimensionalidad. El [Capítulo 7](#) describe los algoritmos y estructuras necesarias para construir y optimizar redes profundas no supervisadas. El [Capítulo 8](#) muestra las funcionalidades de la herramienta desarrollada y ejemplos de su uso.

El código fuente asociado al software implementado se puede encontrar en un repositorio público, así como el código fuente asociado a esta documentación¹.

1.4 BIBLIOGRAFÍA FUNDAMENTAL

Se han consultado muy variadas fuentes a lo largo del estudio realizado. De entre ellas, es preciso destacar algunos libros y artículos fundamentales:

- *Deep Learning*, por Goodfellow, Bengio y Courville [21], es la referencia principal del texto, especialmente de los contenidos acerca de Deep Learning y aprendizaje automático, y también es donde se ha analizado qué aspectos matemáticos componen los fundamentos teóricos de estas técnicas.

¹ <https://github.com/fdavidcl/tfg>

- El artículo *When is “nearest neighbor” meaningful?*, por Beyer y col. [6], que motiva el problema de reducción de dimensionalidad.
- *Elements of information theory*, por Cover y Thomas [15], se ha empleado para entender y exponer el capítulo sobre teoría de la información.
- *Linear algebra done wrong*, por Treil [51], es la fuente de la introducción a álgebra tensorial realizada en el [Capítulo 5](#).

2

OBJETIVOS

Los objetivos que se plantearon al inicio de este trabajo fueron los siguientes:

1. Recopilación y estudio bibliográfico de los fundamentos matemáticos del Deep Learning, con especial enfoque en la teoría de la información y teoría de la probabilidad.
2. Desarrollo e implementación de un software que recopile las técnicas de Deep Learning orientadas a reducción de dimensionalidad más relevantes e incluya visualizaciones y facilidades para el análisis.
3. Análisis de las técnicas utilizadas en relación con los resultados visuales generados por la herramienta software.

El primer objetivo se ha cumplido y se ha extendido notablemente. En este trabajo se estudian los conceptos matemáticos que fundamentan el Deep Learning en los capítulos 3 y 4, pero también se introducen conceptos que se aplican en las implementaciones concretas ([Capítulo 5](#)). El [Capítulo 3](#) motiva además de forma teórica el problema de reducción de la dimensionalidad. Asimismo, el estudio teórico se extiende en los capítulos 6 y 7 hasta el análisis de los algoritmos utilizados en las técnicas de aprendizaje que se usan en la parte práctica del trabajo.

El segundo objetivo también se ha cumplido en un grado alto. Como se expone en el [Capítulo 8](#), se han desarrollado dos paquetes software responsables de implementar distintas técnicas de Deep Learning que permiten reducir la dimensionalidad y generar visualizaciones sobre los modelos, respectivamente. La herramienta no es totalmente exhaustiva, en el sentido de que no contiene todas las estructuras profundas descritas en este trabajo, pero sí que define además un marco de trabajo sencillo en el que es fácil añadir nuevas técnicas y usarlas.

El tercer objetivo consistía en conocer de qué forma los gráficos sobre los modelos generados nos podían dar información acerca del comportamiento de una red neuronal entrenada. En la [Sección 8.5](#) se muestran algunos ejemplos de estos gráficos e ideas acerca de lo que nos pueden explicar de las redes. Consideramos por tanto, que también se ha cumplido este objetivo en cierto grado.

A continuación se enumeran las materias del Doble Grado más relacionadas con este trabajo:

- Geometría I

- Estadística Descriptiva e Introducción a la Probabilidad
- Geometría II
- Análisis Matemático II
- Probabilidad
- Fundamentos de Programación
- Metodología de la Programación
- Estructuras de datos
- Algorítmica
- Inteligencia Artificial
- Aprendizaje Automático
- Inteligencia de Negocio

Parte II

MATEMÁTICAS

3

PROBABILIDAD Y DIMENSIONALIDAD

A lo largo de este capítulo se definen conceptos básicos de la teoría de la probabilidad, que serán utilizados en el resto del trabajo. También se hace mención a unas propiedades de la convergencia que nos permiten después demostrar un resultado técnico que motiva el problema de reducción de la dimensionalidad, estudiado con más detalle en la [Sección 6.4](#). La fuente principal del capítulo es Goodfellow, Bengio y Courville [21, capítulo 3].

3.1 RECORDATORIO DE CONCEPTOS

El objetivo de la probabilidad es modelar y trabajar con incertidumbre. Dicha incertidumbre puede provenir de diversas fuentes, entre ellas:

- Estocasticidad del sistema modelado (e.g. mecánica cuántica, escenarios hipotéticos con aleatoriedad, etc.).
- Falta de observabilidad: los sistemas deterministas se muestran aparentemente estocásticos cuando no se pueden observar todas las variables que los afectan.
- Modelización incompleta: el uso de un modelo que descarta parte de la información observada (un modelo simple pero incompleto puede ser más útil que uno absolutamente preciso).

En el ámbito de estudio de este trabajo, el del aprendizaje automático, la teoría de la probabilidad nos sirve para estudiar los algoritmos de aprendizaje desde un punto de vista teórico y construir representaciones de los modelos que aprenden a partir de los datos.

En esta sección se realiza un recordatorio de conceptos necesarios para trabajar con probabilidades en el resto del texto.

Definición 3.1. Una *variable aleatoria* es una función medible $X : \Omega \rightarrow E$ donde Ω es un espacio de probabilidad y E un espacio medible.

Definición 3.2. El par (Ω, Σ) donde Ω es un conjunto y Σ una σ -álgebra sobre Ω es un *espacio medible*.

Definición 3.3. Si (Ω, \mathcal{F}) es un espacio medible y μ es una medida sobre \mathcal{F} , entonces la terna $(\Omega, \mathcal{F}, \mu)$ es un *espacio de medida*. Si además se verifica $\mu(\Omega) = 1$, entonces se trata de un *espacio de probabilidad*.

Intuitivamente, una variable aleatoria representa una variable del problema que puede tomar distintos valores, y la probabilidad con la

que se darán dichos valores puede ser descrita por una distribución de probabilidad. Cuando notamos $X : \Omega \rightarrow E$, interpretamos que Ω es el conjunto de todos los sucesos posibles, y los estados que puede tomar la variable X vienen dados por su imagen, $X(\Omega) \subset E$. Se dice que X es *discreta* si $X(\Omega)$ es numerable (incluyendo el caso finito), y es *continua* si $X(\Omega)$ es no numerable.

3.1.1 Notación de probabilidad

En ocasiones preferiremos hablar de la probabilidad de que se dé un suceso, en lugar de la probabilidad de un valor concreto. Dado un espacio de probabilidad $(\Omega, \mathcal{F}, \mu)$ y una variable aleatoria $X : \Omega \rightarrow E$, notaremos

$$\mathbb{P}[X = x] = \mu(\{s \in \Omega : X(s) = x\}) = \mu(X^{-1}(\{x\})) .$$

En general, podemos expresar de forma similar la probabilidad de una proposición lógica arbitraria sobre los valores de X . Si tenemos una condición c ,

$$\mathbb{P}[X \text{ verifica } c] = \mu(\{s \in \Omega : X(s) \text{ verifica } c\}) .$$

3.1.2 Distribuciones de probabilidad

Definición 3.4. Una distribución de probabilidad sobre una variable discreta X se describe mediante una *función de probabilidad (Probability Mass Function, PMF)* $p : X(\Omega) \rightarrow [0, 1]$, que se define como

$$p(x) = \mathbb{P}[X = x] .$$

Definición 3.5. Una distribución de probabilidad sobre una variable continua X valuada en los números reales se describe mediante una *función de densidad (Probability Density Function, PDF)* $p : X(\Omega) \rightarrow [0, 1]$, definida como

$$p(x) = \frac{d}{dx}\mu([-\infty, x]) ,$$

donde $x \mapsto \mu([-\infty, x])$ se denomina *función de distribución*.

Es común el abuso del lenguaje por el cual se nota a varias funciones de probabilidad de distintas variables aleatorias por la misma letra, pero no hay posibilidad de confusión ya que se evalúan en distintos valores.

3.1.3 Distribuciones marginales

Cuando una distribución describe varias variables, puede interesar conocer la distribución de un subconjunto de las mismas. Esta se

denomina *distribución marginal*, y se consigue sumando o integrando a lo largo de todos los valores de las variables que no están en el subconjunto. Por ejemplo, si X e Y son variables discretas, se tiene

$$p(x) = \sum_{y \in Y(\Omega)} p(x, y).$$

Si son continuas, entonces se verifica

$$p(x) = \int_{Y(\Omega)} p(x, y) dy.$$

3.1.4 Probabilidad condicionada

En ocasiones es útil representar la probabilidad de un suceso condicionado a la ocurrencia de otro. Para ello se utilizan *probabilidades condicionadas*, que se notan $p(y|x)$ (donde $y \in Y(\Omega), x \in X(\Omega)$) y se calculan mediante la siguiente fórmula, suponiendo que $p(x) > 0$:

$$p(y|x) = \frac{p(y, x)}{p(x)}. \quad (3.1)$$

Una distribución de probabilidad conjunta sobre varias variables se puede descomponer como probabilidades condicionadas sobre una sola variable:

$$p(x_1, \dots, x_n) = p(x_1) \prod_{i=2}^n p(x_i | x_1, \dots, x_{i-1}).$$

Esta expresión se deduce por inducción de la ecuación (3.1).

3.1.5 Independencia e independencia condicionada

Definición 3.6. Dos variables aleatorias, X e Y , son *independientes* si la su probabilidad conjunta equivale al producto de sus probabilidades:

$$p(x, y) = p(x)p(y) \forall x \in X(\Omega), y \in Y(\Omega).$$

Definición 3.7. Además, se dice que son *condicionalmente independientes* respecto a una variable Z si la distribución de probabilidad condicionada se factoriza por X e Y :

$$p(x, y|z) = p(x|z)p(y|z) \forall x \in X(\Omega), y \in Y(\Omega), z \in Z(\Omega).$$

3.1.6 Momentos: esperanza, varianza y covarianza

Para hablar de los momentos de variables aleatorias, nos centramos en las valuadas en los números reales. En el caso de una variable con codominio \mathbb{R}^k , se pueden estudiar los momentos marginales de forma análoga.

Definición 3.8. La *esperanza* de una variable aleatoria X viene dada por las expresiones siguientes, para variables discretas y continuas respectivamente:

$$\mathbb{E}[X] = \sum_{x \in X(\Omega)} xp(x); \quad \mathbb{E}[X] = \int_{X(\Omega)} xp(x)dx .$$

NOTA Todos los momentos se toman respecto de una variable aleatoria y una distribución de probabilidad asociada, por lo que la notación correcta sería $\mathbb{E}_{X \sim p}[X]$. Sin embargo, se omitirá excepto para prevenir ambigüedades.

Se puede definir la esperanza de una función f sobre los valores de una variable aleatoria, del siguiente modo:

$$\mathbb{E}[f(X)] = \sum_{x \in X(\Omega)} f(x)p(x); \quad \mathbb{E}[f(X)] = \int_{X(\Omega)} f(x)p(x)dx .$$

Definición 3.9. La *varianza* da idea acerca de cómo de diferentes entre sí son los valores de una variables conforme se muestran por su distribución de probabilidad:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] .$$

Definición 3.10. La *covarianza* relaciona dos variables aleatorias, indicando la medida en que están relacionadas linealmente y la escala de dichas variables:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] .$$

Definición 3.11. Para un vector de variables aleatorias, $X = (X_1, \dots, X_n)$, la *matriz de covarianza* se define como una función matriz $n \times n$ dada por:

$$\text{Cov}(X)_{i,j} = \text{Cov}(X_i, X_j) .$$

3.2 RESULTADOS DE CONVERGENCIA

Ahora nos situamos en distribuciones de probabilidad sobre espacios vectoriales reales. En concreto, sobre \mathbb{R}^k para $k \geq 1$. Existen distintos conceptos de convergencia que podemos definir, aquí trabajaremos principalmente con la convergencia en probabilidad.

Sea d una distancia en \mathbb{R}^k y sea $\{X_n : \Omega \rightarrow \mathbb{R}^k\}$ una sucesión de variables aleatorias, sea $X : \Omega \rightarrow \mathbb{R}^k$ una variable aleatoria.

Definición 3.12. Se dice que X_n converge en probabilidad a X si para cada $\varepsilon > 0$ se tiene $P[d(X_n, X) > \varepsilon] \rightarrow 0$. Lo denotamos $X_n \xrightarrow{P} X$.

Definición 3.13. Se dice que X_n converge casi seguramente a X si se da la convergencia puntual en un conjunto de medida 1:

$$X_n \xrightarrow{\text{cs}} X \Leftrightarrow P \left[\lim_{n \rightarrow +\infty} d(X_n, X) = 0 \right] = 1$$

Es un resultado conocido que $X_n \xrightarrow{P} X \Rightarrow X_n \xrightarrow{\text{cs}} X$.

Lema 3.1. Si $\{X_n\}$ es una sucesión de variables aleatorias con varianza finita y se verifican las siguientes condiciones:

$$\exists x \in \mathbb{R} : \lim_{m \rightarrow +\infty} E[X_m] = x, \quad \lim_{m \rightarrow +\infty} \text{Var}[X_m] = 0,$$

entonces se tiene que $X_m \xrightarrow{P} x$.

Teorema 3.2 (Teorema de la aplicación continua). Sea $\{X_n\}$ una sucesión de variables aleatorias y X una variable aleatoria, valuadas en un espacio medible E . Sea $g : E \rightarrow F$ con F otro espacio medible. Entonces, si g es continua casi por doquier, se tiene:

$$\begin{aligned} X_n &\xrightarrow{P} X \Rightarrow g(X_n) \xrightarrow{P} g(X), \\ X_n &\xrightarrow{\text{cs}} X \Rightarrow g(X_n) \xrightarrow{\text{cs}} g(X). \end{aligned}$$

3.3 LA MALDICIÓN DE LA DIMENSIONALIDAD

Vamos a aplicar los resultados teóricos anteriores para estudiar una propiedad interesante que determinará uno de los problemas tratados en el campo del aprendizaje automático ([Sección 6.4](#)). Supongamos que contamos con una muestra de datos, en forma de subconjunto finito de \mathbb{R}^n . Nos podemos plantear qué efecto tiene el tamaño de n , en ocasiones denominado *dimensionalidad*, sobre nuestra capacidad para extraer información útil de los datos.

Algunos de los algoritmos más usuales utilizan distancias para medir similitudes entre los datos. Veremos que, conforme n crece, las distancias usuales pierden significado, en el sentido de que el punto más lejano y el más cercano a uno dado están a distancias similares. Este hecho se suele denominar la maldición de la alta dimensionalidad (del inglés *curse of high dimensionality*). Una formalización se encuentra en Beyer y col. [6], y se expone a continuación:

Teorema 3.3. Sea $\{F_m\}_{m \in \mathbb{N}}$ una sucesión de distribuciones de probabilidad, $n \in \mathbb{N}$ y $p \in \mathbb{R}^+$ fijos. Para cada $m \in \mathbb{N}$ sean $X_{m1}, \dots, X_{mn} \sim F_m$ muestras independientes e idénticamente distribuidas. Supongamos que tenemos una función $d_m : \text{Dom}(F_m) \rightarrow \mathbb{R}_0^+$ y llamamos

$$\text{DMIN}_m = \min\{d_m(X_{mi}) : i = 1, \dots, n\},$$

$$\text{DMAX}_m = \max\{d_m(X_{mi}) : i = 1, \dots, n\}.$$

Entonces, si $\lim_{m \rightarrow +\infty} \text{Var}\left[\frac{d_m(X_{m1})^p}{E[d_m(X_{m1})^p]}\right] = 0$ se tiene que, para cada $\varepsilon > 0$,

$$\lim_{m \rightarrow +\infty} P[\text{DMAX}_m \leq (1 + \varepsilon)\text{DMIN}_m] = 1.$$

Demostración. Puesto que las muestras X_{mi} son idénticamente distribuidas, tienen la misma esperanza, y funciones de las mismas también comparten esperanza. Así, llamamos $\mu_m = E[d_m(X_{mi})^p]$ y sea $V_m = \frac{d_m(X_{m1})^p}{\mu_m}$.

Veamos que $V_m \xrightarrow{P} 1$: primero, tenemos que $E[V_m] = \frac{\mu_m}{\mu_m} = 1$, y como consecuencia $\lim_{m \rightarrow +\infty} E[V_m] = 1$. Por hipótesis, $\lim_{m \rightarrow +\infty} \text{Var}[V_m] = 1$, y usando el [Lema 3.1](#) deducimos que $V_m \xrightarrow{P} 1$.

Ahora, definimos la variable aleatoria

$$Y_m = \left(\frac{d_m(X_{m1})^p}{\mu_m}, \dots, \frac{d_m(X_{mn})^p}{\mu_m} \right).$$

Como cada componente del vector Y_m es idénticamente distribuida a V_m , se tiene que $Y_m \xrightarrow{P} (1, \dots, 1)$. Como \min y \max (que dan la componente mínima y máxima del vector, respectivamente) son funciones continuas, podemos utilizar el [Teorema 3.2](#) para obtener que $\min(Y_m) \xrightarrow{P} \min\{1, \dots, 1\} = 1$ y $\max(Y_m) \xrightarrow{P} 1$.

Notemos ahora que $\text{DMIN}_m = \min\{\mu_m Y_m(i) : i = 1, \dots, n\} = \mu_m \min(Y_m)$ y de igual forma $\text{DMAX}_m = \mu_m \max(Y_m)$. Así,

$$\frac{\text{DMAX}_m}{\text{DMIN}_m} = \frac{\mu_m \max(Y_m)}{\mu_m \min(Y_m)} = \frac{\max(Y_m)}{\min(Y_m)} \xrightarrow{P} \frac{1}{1} = 1.$$

Por definición de convergencia en probabilidad, para cada $\varepsilon > 0$ se tiene

$$\lim_{m \rightarrow +\infty} P \left[\left| \frac{\text{DMAX}_m}{\text{DMIN}_m} - 1 \right| \leq \varepsilon \right] = 1, \quad (3.2)$$

y usando que $P[\text{DMAX}_m \geq \text{DMIN}_m] = 1$,

$$\begin{aligned} P \left[\left| \frac{\text{DMAX}_m}{\text{DMIN}_m} - 1 \right| \leq \varepsilon \right] &= P \left[\frac{\text{DMAX}_m}{\text{DMIN}_m} - 1 \leq \varepsilon \right] = \\ &= P[\text{DMAX}_m \leq (1 + \varepsilon)\text{DMIN}_m], \end{aligned}$$

luego el límite (3.2) es el que queríamos demostrar. \square

Nótese que este resultado es más general de lo que necesitamos, usando cualquier función valuada no negativa d_m que podemos interpretar como la distancia a un punto fijo. Como caso particular, en Aggarwal, Hinneburg y Keim [2] se prueba el resultado para la distancia asociada a la norma L_p . Además, no menciona realmente la dimensionalidad, que se puede interpretar como un caso particular de la cantidad m del teorema.

Por otro lado, requiere de una condición que no necesariamente se dará en todos los escenarios, $\lim_{m \rightarrow +\infty} \text{Var} \left[\frac{d_m(X_{m1})^p}{E[d_m(X_{m1})^p]} \right] = 0$. Un análisis de las situaciones en que el resultado es aplicable se encuentra de

nuevo en Beyer y col. [6]. Esencialmente, es suficiente que las distribuciones de los datos sean independientes e idénticamente distribuidas a lo largo de todas las dimensiones, y los momentos convenientes sean finitos. También se aportan varios ejemplos donde no se da la independencia y sí se verifican las condiciones del teorema.

4

TEORÍA DE LA INFORMACIÓN

La teoría de la información estudia y cuantifica la información presente en una señal. El origen de la misma es la búsqueda de códigos para compresión de datos y la maximización de la tasa de transmisión en comunicación. A causa de sus fundamentos y aplicaciones, es un campo de estudio que se relaciona con muchos otros ámbitos, como la ingeniería eléctrica, la estadística, la probabilidad, la física, la economía y la informática.

A partir de lo estudiado en el capítulo anterior, si queremos añadir una intuición acerca de la información que aporta un suceso, será razonable que la cantidad de información sea menor cuanto más probable sea el suceso dado. En el caso extremo, el suceso seguro no aporta información alguna.

Realizaremos dos aplicaciones de esta teoría al problema que ocupa este trabajo. Por un lado, el concepto de entropía cruzada nos permitirá definir en la Sección 7.1.1 las funciones de coste o funciones objetivo que se tratarán de optimizar mediante las técnicas de Deep Learning. Por otro, la desigualdad de la información nos aportará la intuición de que un modelo muy cercano a la distribución de los datos nos permitirá reducir mejor la dimensionalidad.

Para formalizar estas intuiciones, definiremos y haremos uso de conceptos como la entropía. La fuente principal de este capítulo es Cover y Thomas [15].

4.1 ENTROPÍA. PROPIEDADES Y MAGNITUDES

4.1.1 Concepto

Definición 4.1. La *entropía* H de una variable aleatoria discreta X con distribución $p(x) = P[X = x]$ viene dada por la siguiente expresión:

$$H(X) = - \sum_{x \in X(\Omega)} p(x) \log p(x).$$

\log denota el logaritmo neperiano. Si se usa otra base b para el logaritmo de la definición notaremos H_b .

Observamos que se puede expresar como la esperanza de una función de la variable X :

$$H(X) = E \left[\frac{1}{\log p(X)} \right].$$

Además, es interesante notar que H es un funcional de p en el sentido de que no depende de los valores que tome la variable aleatoria, sino únicamente de la probabilidad de los mismos. El significado que aporta la entropía de una variable es la cantidad de información esperada en un suceso. Por esto, las distribuciones cercanas a la uniforme tienen mayor entropía que las que son casi determinísticas.

4.1.2 Propiedades

Lema 4.1. *La entropía de una variable es siempre positiva o nula.*

Demostración. Efectivamente, puesto que $0 \leq p(x) \leq 1$ para todo x , se tiene que $\log(1/p(x)) \geq 0$, y como consecuencia la esperanza de dicha función de X es no negativa. \square

Lema 4.2. $H_b(X) = (\log_b a) H_a(X)$

Demostración. Es consecuencia del cambio de base de los logaritmos. \square

4.1.3 Magnitudes

Según la base que se tome para los logaritmos, la escala de la entropía varía, por lo que se está midiendo en una magnitud distinta.

- Si se toman logaritmos en base 2, entonces se habla de la entropía en *bits*.
- Si se toman en base e , se está midiendo la entropía en *nats*¹.

Además, el cambio de base del [Lema 4.2](#) nos permite convertir de una magnitud a otra:

$$H = (\log 2)H_2 .$$

4.2 ENTROPÍA CONJUNTA Y ENTROPÍA CONDICIONAL

Definición 4.2. La *entropía conjunta* $H(X, Y)$ de dos variables aleatorias discretas con distribución conjunta $p(x, y)$ se define como

$$H(X, Y) = - \sum_x \sum_y p(x, y) \log p(x, y) , \quad (4.1)$$

que también se puede expresar como

$$H(X, Y) = -E[\log p(x, y)] . \quad (4.2)$$

Definición 4.3. Si $(X, Y) \sim p(x, y)$, entonces se define la entropía condicional $H(Y | X)$ como

¹ Denominación de la unidad de medida de información análoga al *bit* para base e .

$$H(Y | X) = \sum_x p(x) H(Y | X = x) \quad (4.3)$$

$$= - \sum_x p(x) \sum_y p(y | x) \log p(y | x) \quad (4.4)$$

$$= - \sum_x \sum_y p(x, y) \log p(y | x) \quad (4.5)$$

$$= -E_{p(x,y)} \log p(Y | X) . \quad (4.6)$$

Estas dos definiciones están relacionadas por el siguiente teorema.

Teorema 4.3 (Regla de la cadena de la entropía).

$$H(X, Y) = H(X) + H(Y | X)$$

Demostración.

$$\begin{aligned} H(X, Y) &= - \sum_x \sum_y p(x, y) \log p(x, y) \\ &= - \sum_x \sum_y p(x, y) \log (p(x)p(y | x)) \\ &= - \sum_x \sum_y p(x, y) \log p(x) - \sum_x \sum_y p(x, y) \log p(y | x) \\ &= - \sum_x p(x) \log p(x) - \sum_x \sum_y p(x, y) \log p(y | x) \\ &= H(X) + H(Y | X) . \end{aligned}$$

□

Corolario 4.4.

$$H(X, Y | Z) = H(X | Z) + H(Y | X, Z)$$

Demostración. La prueba sigue pasos análogos al teorema anterior.

□

4.3 ENTROPÍA RELATIVA, INFORMACIÓN MUTUA Y ENTROPÍA CRUZADA

Si la entropía de una variable es una medida de la cantidad de información requerida para explicarla, la entropía relativa de una función de distribución a otra $D(p||q)$ mide la ineficiencia de asumir que la distribución de una variable es q cuando la distribución verdadera es p , es decir, la longitud adicional de código basado en q necesaria para describir la variable respecto de un código basado en p .

Definición 4.4. La *entropía relativa* o *divergencia de Kullback-Leibler* entre dos funciones de probabilidad p y q se define como

$$D(p || q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = E_p \left[\log \frac{p(X)}{q(X)} \right] .$$

En la definición anterior usamos la convención de que $0 \log \frac{0}{q} = 0$ y $p \log \frac{p}{0} = \infty$, basada en argumentos de continuidad.

La entropía relativa adolece de algunas propiedades para ser considerada una distancia entre distribuciones de probabilidad. En concreto, no es simétrica y no satisface la desigualdad triangular.

Definición 4.5. Sean X, Y dos variables aleatorias con función de probabilidad conjunta $p(x, y)$ y funciones de probabilidad marginal $p(x)$ y $p(y)$ respectivamente. La *información mutua* $I(X; Y)$ es la entropía relativa entre la distribución conjunta y el producto de las distribuciones:

$$\begin{aligned} I(X; Y) &= \sum_x \sum_y p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \\ &= D(p(x, y) || p(x)p(y)) \\ &= E_{p(x, y)} \left[\log \frac{p(X, Y)}{p(X)p(Y)} \right]. \end{aligned}$$

Teorema 4.5. La información mutua es la reducción en la incertidumbre de X dado el conocimiento de Y . Por simetría además X da tanta información sobre Y como Y sobre X :

$$I(X; Y) = H(X) - H(X | Y) = H(Y) - H(Y | X).$$

Además, como consecuencia:

$$I(X; Y) = H(X) + H(Y) - H(X, Y),$$

en particular,

$$I(X; X) = H(X).$$

Demostración.

$$\begin{aligned} I(X; Y) &= \sum_x \sum_y p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \\ &= \sum_x \sum_y p(x, y) \log \frac{p(x | y)}{p(x)} \\ &= -\sum_x \sum_y p(x, y) \log p(x) + \sum_x \sum_y p(x, y) \log p(x | y) \\ &= -\sum_x p(x) \log p(x) - \left(-\sum_x \sum_y p(x, y) \log p(x | y) \right) \\ &= H(X) - H(X | Y). \end{aligned}$$

La consecuencia se deduce de la igualdad $H(X, Y) = H(X) + H(Y | X)$. \square

Definimos una versión condicionada de la entropía relativa.

Definición 4.6. La *entropía relativa condicional* $D(p(y|x) || q(y|x))$ se define como

$$D(p(y|x) || q(y|x)) = \sum_x p(x) \sum_y p(y|x) \log \frac{p(y|x)}{q(y|x)} = E_{p(x,y)} \left[\log \frac{p(Y|X)}{q(Y|X)} \right]$$

Un concepto similar al de entropía relativa es la entropía cruzada. Simplemente, se considera la longitud completa de código necesaria para describir la variable en función de un código basado en la distribución q frente a p , en lugar de considerar la longitud de código adicional.

Definición 4.7. La *entropía cruzada* entre dos funciones de probabilidad p y q se define como

$$C(p,q) = H(X) + D(p||q) = E_p [\log q(X)] .$$

NOTA En ocasiones se utiliza la notación $H(p,q)$ para hablar de entropía cruzada, pudiendo confundirse con la entropía conjunta. En este texto se utilizará en su lugar $C(p,q)$.

Observación 4.1. Podemos tomar la entropía cruzada sobre una variable condicionada a otra de forma natural. Consideremos X, Y variables aleatorias siguiendo una distribución p , sobre las que asumimos una distribución q . La entropía cruzada de p respecto de q para Y condicionada a X será:

$$\begin{aligned} C(p(y|x), q(y|x)) &= H(Y|X) + D(p(y|x) || q(y|x)) \\ &= \sum_x p(x) \sum_y p(y|x) \left(-\log p(y|x) + \log \frac{p(y|x)}{q(y|x)} \right) \\ &= -\sum_x p(x) \sum_y p(y|x) \log q(y|x) \\ &= -E_p [\log q(Y|X)] . \end{aligned}$$

4.4 LA DESIGUALDAD DE LA INFORMACIÓN

Vamos a estudiar una de las desigualdades esenciales en Teoría de la Información, que nos dirá que la distribución que determina los códigos más cortos para unos datos es la propia distribución de los datos. Para ello, nos basaremos en una de las desigualdades más usadas en matemáticas, la desigualdad de Jensen.

Teorema 4.6 (Desigualdad de Jensen). *Si f es una función convexa y X es una variable aleatoria, entonces*

$$E[f(X)] \geq f(E[X]) .$$

Demostración. A continuación se expone una demostración para distribuciones discretas con un número finito de puntos de masa, es decir, puntos donde la probabilidad no es nula. Llamamos x_1, \dots, x_k a dichos puntos y p_1, \dots, p_k a las respectivas probabilidades.

Procedemos por inducción en el número de puntos de masa. Para dos puntos de masa, la desigualdad se expresa

$$p_1 f(x_1) + p_2 f(x_2) \geq f(p_1 x_1 + p_2 x_2),$$

que se deduce de la definición de función convexa.

Supuesto el teorema cierto para distribuciones de $k - 1$ puntos de masa, veámoslo para k puntos. Escribiendo $p'_i = \frac{p_i}{1-p_k}$ para $i = 1, \dots, k-1$, se tiene

$$\begin{aligned} \sum_{i=1}^k p_i f(x_i) &= p_k f(x_k) + (1-p_k) \sum_{i=1}^{k-1} p'_i f(x_i) \\ &\geq p_k f(x_k) + (1-p_k) f\left(\sum_{i=1}^{k-1} p'_i x_i\right) \\ &\geq f\left(p_k x_k + (1-p_k) \sum_{i=1}^{k-1} p'_i x_i\right) \\ &= f\left(\sum_{i=1}^k p_i x_i\right), \end{aligned}$$

donde la primera desigualdad se deduce de la hipótesis de inducción y la segunda es consecuencia de la convexidad.

Por argumentos de continuidad se puede extender esta demostración a distribuciones continuas. \square

Teorema 4.7 (Desigualdad de la información). *Sean p, q , funciones de probabilidad de una variable aleatoria X . Entonces,*

$$D(p \parallel q) \geq 0$$

Demostración. Sea $A = \{x : p(x) > 0\}$ el soporte de p . Entonces, notando que la función $-\log$ es convexa por concavidad del logaritmo,

$$\begin{aligned} D(p \parallel q) &= E_p \left[-\log \frac{q(X)}{p(X)} \right] \geq -\log E_p \left[\frac{q(X)}{p(X)} \right] \\ &= -\log \left(\sum_x p(x) \frac{q(x)}{p(x)} \right) = -\log \left(\sum_x q(x) \right) \\ &= -\log 1 = 0 \end{aligned}$$

\square

Las siguientes son algunas consecuencias directas de este resultado.

Corolario 4.8 (No negatividad de la información mutua). *Para cualesquiera dos variables aleatorias X, Y se tiene $I(X; Y) \geq 0$.*

Demostración. Por definición de información mutua. \square

Corolario 4.9. *Para cualesquiera dos variables aleatorias X, Y y funciones de probabilidad p y q ,*

$$D(p(y|x) || q(y|x)) \geq 0 .$$

Corolario 4.10. *Para una variable aleatoria X y funciones de probabilidad p (dada por la distribución de X) y q ,*

$$C(p, q) \geq H(X) .$$

De igual forma, si Y es otra variable aleatoria con la misma distribución, se tiene

$$C(p(y|x), q(y|x)) \geq H(Y | X) .$$

Lo que nos dicen el [Teorema 4.7](#) y sus consecuencias es que cada vez que asumimos un modelo sobre unos datos que no coincide con su verdadera distribución, necesitamos una mayor longitud de código que la óptima para representarlos. En nuestro caso, si vamos a tratar de reducir la dimensionalidad de un conjunto de datos, nos aporta la intuición de que un modelo muy ajustado a la distribución verdadera será más beneficioso a la hora de obtener una representación de menor dimensionalidad.

5

ÁLGEBRA TENSORIAL

El álgebra tensorial es una rama del álgebra que extiende los conceptos del álgebra lineal. Además, el concepto de tensor se traslada a la implementación de técnicas de aprendizaje automático, puesto que la aplicación que se realiza de ellos permite hacer más eficientes los cálculos de operaciones respecto al uso exclusivo de matrices. La fuente principal de este capítulo es Treil [51, capítulo 8].

5.1 ESPACIOS DUALES

En esta sección introducimos los conceptos esenciales sobre duales de espacios vectoriales, necesarios para llegar a la definición de tensor.

5.1.1 Funcionales lineales y el espacio dual

Definición 5.1. Un *funcional lineal* en un espacio vectorial finito dimensional V sobre un cuerpo \mathbb{K} es una aplicación lineal $L : V \rightarrow \mathbb{K}$.

Definición 5.2. El *espacio dual* de un espacio vectorial finito dimensional V es $V^* = \{L : V \rightarrow \mathbb{K} \text{ lineal}\} = \mathcal{L}(V, \mathbb{K})$.

Ejemplo 5.1. Sea $V = \mathbb{R}^n$ y consideramos $(\mathbb{R}^n)^* = \{L : \mathbb{R}^n \rightarrow \mathbb{R} \text{ lineal}\}$. Sabemos que toda aplicación lineal de \mathbb{R}^n en \mathbb{R}^m se expresa, fijada una base, como una matriz $m \times n$, luego identificamos $(\mathbb{R}^n)^*$ con matrices $1 \times n$ (en la base usual). Evidentemente hay un isomorfismo entre este conjunto y \mathbb{R}^n :

$$(\mathbb{R}^n)^* \cong \mathcal{M}_{1 \times n}(\mathbb{R}) \cong \mathbb{R}^n \\ (m_1 \dots m_n) \mapsto (m_1, \dots, m_n)$$

Este hecho se generaliza para cualquier cuerpo \mathbb{K} : $(\mathbb{K}^n)^* \cong \mathbb{K}^n$.

5.1.1.1 Cambio de coordenadas

Sea V \mathbb{K} -espacio vectorial, sean $A = \{a_1, \dots, a_n\}, B = \{b_1, \dots, b_n\}$ bases de V donde $n = \dim_{\mathbb{K}} V$.

Introducimos la siguiente notación: dada una base B de V , B' de W , $L \in \mathcal{L}(V, W)$, notaremos $[L]_{B', B}$ a la expresión matricial de L en las bases B, B' . Si $L \in V^*$ notamos $[L]_B$.

Sabemos que la expresión de $L \in V^*$ en la base B viene dada por su imagen por los vectores de la base, y el cambio de coordenadas es $[L]_B = [L]_A[I]_{A,B}$.

Recordamos también que el cambio de base de $v \in V$ se realiza mediante $[B]_B = [I]_{B,A}[V]_A$ y que $[I]_{B,A} = [I]_{A,B}^{-1}$. Así, llamando $S = [I]_{B,A}$ observamos que el cambio de base de los vectores asociados a las filas $[L]_B, [L]_A$ es:

$$[L]_B^t = (S^{-1})^t [L]_A^t$$

Proposición 5.1. *Dado V espacio vectorial, si S es la matriz de cambio de base de A a B entonces la matriz de cambio de base de V^* es $(S^{-1})^t$.*

Lema 5.2. *Sea $v \in V$. Si $L(v) = 0 \forall L \in V^*$ entonces $v = 0$. Como consecuencia, si $L(v_1) = L(v_2) \forall L \in V^*$ entonces $v_1 = v_2$.*

Demuestração. Sea B base de V . Entonces $L(v) = [L]_B[v]_B$. Basta tomar $L_k = [0, \dots, 0, \overset{(k)}{1}, 0, \dots, 0]$ y comprobar que en ese caso $L_k[v]_B = 0$ implica que la k -ésima coordenada de $[v]_B$ es 0. Repitiendo el mismo paso para cada k tenemos que $v = 0$. \square

5.1.2 El segundo dual

Puesto que V^* es un espacio vectorial, podemos considerar también su dual, que notaremos V^{**} . Comprobaremos que, de hecho, V^{**} es isomorfo a V de una forma natural. Dado $v \in V$ podemos tomar $L_v \in V^{**}$ dado por $L_v(f) = f(v) \forall f \in V^*$. Así, podemos construir una aplicación del espacio V en su segundo dual, $T : V \rightarrow V^{**}$, dada de forma natural por $Tv = L_v \forall v \in V$.

Para ver que T es un isomorfismo, observamos que las dimensiones de los espacios coinciden: $\dim V^{**} = \dim V^* = \dim V$. Por tanto, bastará con ver que T es inyectivo: veamos para ello que $\text{Ker } T = \{0\}$. Dado $v \in \text{Ker } T$, tenemos que $\forall f \in V^* f(v) = L_v(f) = T(v)(f) = 0$. Por el [Lema 5.2](#), se tiene que $v = 0$.

Nótese que el isomorfismo T no depende de la elección de una base en V .

5.2 FUNCIONES MULTILINEALES. TENSORES

Definición 5.3. Sean V_1, \dots, V_p, V espacios vectoriales sobre un cuerpo \mathbb{K} . Una *aplicación multilíneaal* (p -lineal) con valores en V es una función $F : V_1 \times \dots \times V_p \rightarrow V$, lineal en cada variable. Es decir, para cada $k \in \{1, \dots, p\}$ y fijado $(v_1, \dots, v_{k-1}, 0, v_{k+1}, \dots, v_p) \in V_1 \times \dots \times V_p$, se tiene que la aplicación que lleva $v_k \mapsto F(v_1, \dots, v_{k-1}, v_k, v_{k+1}, \dots, v_p)$ es lineal.

Notamos por $\mathcal{L}(V_1, \dots, V_p; V)$ a la familia de todas las aplicaciones p -lineales de $V_1 \times \dots \times V_p$ en V .

Definición 5.4. Un *tensor* o *funcional multilíneal* es una aplicación multilineal con codominio \mathbb{K} , $F : V_1 \times \dots \times V_p \rightarrow \mathbb{K}$. La cantidad p se denomina el *rango* o *valencia* del tensor.

En particular, un tensor de rango 1 es un funcional lineal, y un tensor de rango 2 es una forma bilineal.

Ejemplo 5.2. Sean V_1, \dots, V_p \mathbb{K} -espacios vectoriales y sean $f_1 \in V_1^*, \dots, f_p \in V_p^*$ funcionales lineales. Definimos el funcional multilineal $F : V_1 \times \dots \times V_p \rightarrow \mathbb{K}$ dado por

$$F(v_1, \dots, v_p) = f_1(v_1)f_2(v_2)\dots f_p(v_p), \quad v_i \in V_i, \quad k = 1, 2, \dots, p.$$

El funcional F se denomina *producto tensorial* de los funcionales f_i y lo notamos $F = f_1 \otimes f_2 \otimes \dots \otimes f_p$.

Observación 5.1. El conjunto de las aplicaciones multilineales es un \mathbb{K} -espacio vectorial, mediante las siguientes operaciones de suma y producto por escalar: sean $F_1, F_2 \in \mathcal{L}(V_1, \dots, V_p; V)$, $\alpha \in \mathbb{K}$

$$\begin{aligned} (F_1 + F_2)(v_1, \dots, v_p) &= F_1(v_1, \dots, v_p) + F_2(v_1, \dots, v_p), \\ (\alpha F_1)(v_1, \dots, v_p) &= \alpha F_1(v_1, \dots, v_p). \end{aligned}$$

Proposición 5.3. Sean V_1, \dots, V_p \mathbb{K} -espacios vectoriales con bases $B^{(1)}, \dots, B^{(p)}$ respectivamente. Notamos $b_i^{(k)}$ al i -ésimo elemento de la base $B^{(k)}$.

Para cada $k \in \{1, \dots, p\}$ y para cada $i \in \{1, \dots, \dim V_k\}$ sea $f_i^{(k)}$ el funcional lineal de V_k^* definido por

$$\begin{aligned} f_i^{(k)}(b_i^{(k)}) &= 1 \\ f_i^{(k)}(b_j^{(k)}) &= 0, \quad j \neq i. \end{aligned}$$

La familia

$$B = \left\{ f_{i_1}^{(1)} \otimes \dots \otimes f_{i_p}^{(p)}, \quad 1 \leq i_k \leq \dim V_k, \quad k \in \{1, \dots, p\} \right\}$$

es una base del espacio $\mathcal{L}(V_1, \dots, V_p; \mathbb{K})$.

En particular,

$$\dim \mathcal{L}(V_1, \dots, V_p; \mathbb{K}) = (\dim V_1) \dots (\dim V_p).$$

Demostración. Dada $F \in \mathcal{L}(V_1, \dots, V_p; \mathbb{K})$ queremos expresarla de forma única en función de los elementos de la familia B , es decir, buscamos coeficientes $\alpha_{i_1, i_2, \dots, i_p} \in \mathbb{K}$ tales que

$$F = \sum_{i_k \in \{1, \dots, \dim V_k\}} \alpha_{i_1, i_2, \dots, i_p} f_{i_1}^{(1)} \otimes \cdots \otimes f_{i_p}^{(p)}. \quad (5.1)$$

Por la definición de los funcionales, se tiene que

$$f_{i_1}^{(1)} \otimes \cdots \otimes f_{i_p}^{(p)} (b_{j_1}^{(1)}, \dots, b_{j_p}^{(p)}) = 1 \Leftrightarrow i_1 = j_1, \dots, i_p = j_p \quad (5.2)$$

y, en otro caso,

$$f_{i_1}^{(1)} \otimes \cdots \otimes f_{i_p}^{(p)} (b_{j_1}^{(1)}, \dots, b_{j_p}^{(p)}) = 0.$$

Evaluando ahora F (5.1) en $b_{i_1}^{(1)}, \dots, b_{i_p}^{(p)}$:

$$F(b_{i_1}^{(1)}, \dots, b_{i_p}^{(p)}) = \alpha_{i_1, \dots, i_p}$$

lo cual nos da la unicidad de los coeficientes, en caso de que existan. La existencia se deduce definiendo

$$\alpha_{i_1, \dots, i_p} := F(b_{i_1}^{(1)}, \dots, b_{i_p}^{(p)}),$$

de forma que la condición (5.2) se mantiene para todas las tuplas del tipo $b_{j_1}^{(1)}, \dots, b_{j_p}^{(p)}$. Así, se tiene la descomposición que buscamos y B es una base.

□

5.3 PRODUCTOS TENSORIALES

Definición 5.5. Sean V_1, V_2, \dots, V_p espacios vectoriales. El producto tensorial de los espacios es el conjunto de funcionales multilíneales $\mathcal{L}(V_1^*, V_2^*, \dots, V_p^*; \mathbb{K})$, y lo notamos $V_1 \otimes V_2 \otimes \cdots \otimes V_p$.

Corolario 5.4. Sean V_1, \dots, V_p \mathbb{K} -espacios vectoriales con bases $B^{(1)}, \dots, B^{(p)}$ respectivamente. Llamamos $b_i^{(k)}$ al i -ésimo elemento de la base $B^{(k)}$ y observamos que podemos definir el producto tensorial de elementos de V_1, \dots, V_p viéndolos como funcionales de V_1^*, \dots, V_p^* . Entonces, la familia

$$B = \left\{ b_{i_1}^{(1)} \otimes \cdots \otimes b_{i_p}^{(p)}, 1 \leq i_k \leq \dim V_k, k \in \{1, \dots, p\} \right\},$$

es una base del espacio $V_1 \otimes V_2 \otimes \cdots \otimes V_p$.

Demostración. Consecuencia de la Proposición 5.3 y el isomorfismo $V_k^{**} \cong V_k$. □

Observación 5.2. Dados $v_1 \in V_1, \dots, v_p \in V_p$, para $v'_k \in V_k$, $k \in \{1, \dots, p\}$, $\lambda, \mu \in \mathbb{K}$ y cualesquiera $f_1 \in V_1^*, \dots, f_p \in V_p^*$ se tiene:

$$\begin{aligned}
(v_1 \otimes v_2 \otimes \cdots \otimes (\lambda v_k + \mu v'_k) \otimes \cdots \otimes v_p)(f_1, \dots, f_p) &= \\
f_1(v_1) \dots f_k(\lambda v_k + \mu v'_k) \dots f_p(v_p) &= \\
f_1(v_1) \dots (\lambda f_k(v_k) + \mu f_k(v'_k)) \dots f_p(v_p) &= \\
\lambda f_1(v_1) \dots f_k(v_k) \dots f_p(v_p) + \mu \lambda f_1(v_1) \dots f_k(v'_k) \dots f_p(v_p) &= \\
(\lambda v_1 \otimes v_2 \otimes \cdots \otimes v_k \otimes \cdots \otimes v_p + \mu v_1 \otimes v_2 \otimes \cdots \otimes v'_k \otimes \cdots \otimes)(f_1, \dots, f_p)
\end{aligned}$$

Hemos comprobado que la aplicación $(v_1, v_2, \dots, v_p) \mapsto v_1 \otimes v_2 \otimes \cdots \otimes v_p$ es lineal en cada variable.

5.4 TENSORES COVARIANTES Y CONTRAVARIANTES

Sean X_1, X_2, \dots, X_p, V espacios vectoriales y sea V_k bien X_k o bien X_k^* , para cada $k = 1, 2, \dots, p$.

Definición 5.6. Decimos que $F \in \mathcal{L}(V_1, V_2, \dots, V_p; V)$ es una aplicación multilinear *covariante* en la k -ésima variable si $V_k = X_k$ y *contravariante* en dicha variable si $V_k = X_k^*$.

Si F es covariante (resp. contravariante) en todas las variables decimos simplemente que es covariante (resp. contravariante). Si F es covariante en r variables y contravariante en s variables, decimos que es r -covariante s -contravariante, o de *tipo* (r, s) .

Ejemplo 5.3. Algunos casos particulares de tensores, para observar que generalizan los objetos del álgebra lineal:

- Dado un espacio vectorial V , un funcional $f \in V^*$ es un tensor 1 -covariante.
- Un vector $v \in V$, visto en el doble dual V^{**} , es un tensor 1 -contravariante.
- Por convención, se dice que una constante $\lambda \in \mathbb{K}$ es un tensor de tipo $(0, 0)$.

5.5 LOS TENSORES EN APRENDIZAJE AUTOMÁTICO

Se ha visto que los tensores generalizan estructuras como los vectores y las aplicaciones lineales. Es importante notar que dichos objetos se pueden representar mediante secuencias finitas de escalares, las componentes, que dependen de la base escogida para los espacios vectoriales donde se esté trabajando. De igual forma, un tensor se puede expresar en componentes del cuerpo \mathbb{K} respecto de una base.

La expresión de un tensor en componentes necesitará una representación en tantas dimensiones como su rango. Intuitivamente, podemos decir que el rango de un tensor es el número de índices necesarios para recorrer sus componentes.

Esta idea se lleva al aprendizaje automático como una generalización de las matrices. Esencialmente, se le llama *tensor* de rango p a un objeto $T \in \mathbb{K}^{d_1 d_2 \dots d_p}$. Aunque este es un caso particular de los tensores estudiados en este capítulo, no se suele hacer uso de sus propiedades algebraicas. Sin embargo, algunas de las operaciones y descomposiciones de cálculo con tensores permiten realizar un uso eficiente de la memoria disponible en la máquina a la hora de entrenar una red neuronal [31].

Parte III
INFORMÁTICA

6

APRENDIZAJE AUTOMÁTICO

En este capítulo introducimos conceptos acerca del aprendizaje automático, la rama de la inteligencia artificial que se ocupa de inferir conocimiento sin necesidad de una programación explícita. Además, se estudia el problema de clasificación y cómo es afectado por la dimensionalidad de los datos. Llegamos así a presentar el problema principal que trata este trabajo: la reducción de la dimensionalidad. Por último, se expone un algoritmo clásico de optimización, gradiente descendente, que servirá como base para los utilizados en Deep Learning.

6.1 INTRODUCCIÓN

Un *algoritmo de aprendizaje* es, según Mitchell [38], un programa cuyo rendimiento respecto de un conjunto de tareas T y una medida de rendimiento P mejora tras conocer una experiencia E . En ese caso, se dice que el algoritmo ha *aprendido* de dicha experiencia.

Estas tareas y experiencias pueden ser de muy diversas clases, lo que propicia la aparición de algoritmos y técnicas de aprendizaje diferentes que tratan de abordarlas. Estos algoritmos computacionales son necesarios cuando la complejidad o el tamaño de la tarea impide tratarla con técnicas manuales.

Entre las tareas de aprendizaje que se presentan en la literatura se incluyen:

- clasificación
- regresión
- detección de anomalías
- agrupamiento (*clustering*)
- reducción de dimensionalidad
- detección y eliminación de ruido

6.2 TIPOS DE APRENDIZAJE

La mayoría de *experiencias* de las que puede aprender un algoritmo permite categorizar el aprendizaje en dos grandes clases: supervisado y no supervisado.

6.2.1 Aprendizaje supervisado

En aprendizaje supervisado, se le proporciona al algoritmo un conjunto de ejemplos para los cuales la tarea está resuelta. Así, se pretende que aprenda a realizar la misma tarea para nuevas muestras.

Algunos problemas concretos en los que se realiza esta clase de aprendizaje son:

- Clasificación: el programa debe deducir una etiqueta o clase para cada instancia, y para ello el aprendizaje se suele realizar mediante un conjunto de ejemplos que ya tienen asignada su etiqueta. En la [Sección 6.3](#) se desarrolla este problema en mayor detalle.
- Regresión: consiste en asignar un valor real a cada nuevo conjunto de valores de entrada, habiendo aprendido de muestras que ya tenían un valor asociado.
- Predicción de series temporales: se trata de predecir el valor de una variable en el futuro, conociendo su valor en distintos puntos del pasado.

6.2.2 Aprendizaje no supervisado

La modalidad no supervisada involucra a tareas de las que el algoritmo no tiene ejemplos resueltos. La experiencia que se le proporciona puede estar basada en otras características de los datos.

Algunas de las tareas donde se utiliza aprendizaje no supervisado son:

- Agrupamiento o *clustering*: se proporcionan al algoritmo datos sin clasificar que debe subdividir en diferentes conjuntos de forma que los datos del mismo conjunto sean más similares entre sí que entre elementos de distintos conjuntos.
- Reglas de asociación: el programa debe extraer relaciones relevantes entre los atributos del conjunto de datos que aporten información útil y novedosa acerca de la situación estudiada.
- Aprendizaje de características: se aportan datos sin etiquetar, para los que el algoritmo debe extraer características representativas. También se da este problema en el aprendizaje supervisado, cuando los datos están etiquetados.

El aprendizaje no supervisado abarca multitud de problemas ampliamente estudiados que tienen diversas aplicaciones presentes en distintos campos, como el tratamiento de imágenes y reconocimiento de objetos [44], análisis semántico [28] y sintáctico del lenguaje [8] o el preprocesamiento de datos y pre-entrenamiento para una posterior fase de aprendizaje [18].

6.3 PROBLEMA DE CLASIFICACIÓN

Un problema clásico en el aprendizaje automático es el de clasificación. Se trata de una tarea de aprendizaje supervisado que consiste en aprender acerca de la etiqueta o clase de una secuencia de muestras clasificadas, para después ser capaz de predecir el valor de dicha etiqueta en nuevas instancias sin clasificar.

La clasificación tiene multitud de aplicaciones en diversos ámbitos. Algunos ejemplos son el diagnóstico de enfermedades [32], la detección de fraude [41] y la clasificación de mensajes de correo [14].

6.3.1 Definición

Una formulación sencilla del problema es la siguiente:

Definición 6.1. Sean A_1, A_2, \dots, A_f conjuntos no vacíos llamados *atributos de entrada*. Llamaremos *espacio de atributos* (o *espacio de características*) a $\mathcal{A} = A_1 \times A_2 \times \dots \times A_f$.

Sea L un conjunto finito al que denominaremos *conjunto de etiquetas*.

Sea $D \subset \mathcal{A} \times L$ un subconjunto finito del espacio de atributos, lo llamaremos *conjunto de instancias* o *dataset*.

Decimos que la tripleta $\mathcal{P} = (\mathcal{A}, L, D)$ es un *problema de clasificación*.

Definición 6.2. Dado un problema de clasificación (\mathcal{A}, L, D) , un *claseificador* es una aplicación $c : \mathcal{A} \rightarrow L$.

Así, el objetivo que se persigue al abordar un problema de clasificación \mathcal{P} es encontrar el clasificador c que mejor se adapte al problema, según una o varias métricas de evaluación. Intuitivamente, el procedimiento por el que se obtenga dicho clasificador debe ser capaz de utilizar la información de las instancias en el dataset D para predecir una clase en nuevas instancias del espacio de atributos.

Atendiendo a la estructura de L , es decir, el número de características que estarán ausentes en los nuevos datos y sus posibles valores, distinguiremos los siguientes tipos de clasificación:

- **Binaria:** implica clasificar en 2 clases (generalmente significan que una condición es verdadera o falsa), utilizando una característica que tome únicamente dos valores. Así, $L = \{0, 1\}$.
- **Multiclasa:** en este caso habrá más de dos clases, pero cada instancia pertenecerá a una y solo una de ellas, por lo que se usará una característica que contenga tantos valores como clases: $L = \{0, 1, \dots, l\}$.
- **Multietiqueta:** en esta situación, cada instancia puede asociarse a más de una etiqueta, por tanto se usarán tantos atributos como etiquetas, cada uno de ellos conteniendo dos valores: $L = \{0, 1\} \times \dots \times \{0, 1\}$.

- Multidimensional: se trata de una generalización del caso multietiqueta donde cada una de las etiquetas toma valores en un conjunto finito arbitrario, esto es, $L = \{0, \dots, \lambda_1\} \times \dots \times \{0, \dots, \lambda_l\}$.

Las definiciones previas componen una formalización simple del problema de clasificación. Una modelización más detallada y con consecuencias teóricas de interés se encuentra en la teoría de aprendizaje PAC [48]. De esta teoría además se extraen algunos resultados relevantes. Por un lado, el hecho de que los algoritmos sean capaces de generalizar un modelo adecuado a partir de una cantidad finita de muestras. Por contraposición, considerados sobre el conjunto de todas las distribuciones de datos posibles, todos los algoritmos de clasificación presentan en media la misma tasa de error en la predicción de clases para nuevos ejemplos (hecho conocido como el teorema de *No Free Lunch* [53]).

6.3.2 Estructura del espacio de atributos

En principio no tenemos por qué asumir una estructura algebraica para el espacio de atributos \mathcal{A} , pero la mayoría de algoritmos necesitarán una forma de medir similitud entre instancias. Para ello, normalmente se puede utilizar una distancia d , de forma que (\mathcal{A}, d) sea un espacio métrico. Para el uso de un conjunto de datos en redes neuronales, sin embargo, convendrá suponer además $\mathcal{A} \subset \mathbb{R}^f$.

6.4 PROBLEMA DE REDUCCIÓN DE DIMENSIONALIDAD

Es común encontrar problemas de clasificación donde el espacio de atributos posee una alta dimensionalidad. Por ejemplo: conjuntos de datos extraídos de texto, donde cada atributo representa la aparición de una palabra en un documento; conjuntos basados en imágenes donde cada atributo representa un píxel [34], o datos que expresan características genéticas [13].

Como consecuencia del [Teorema 3.3](#), al aumentar la dimensionalidad de los conjuntos de datos se pierde significatividad en las distancias, en el sentido de que, para una instancia de la muestra dada, la instancia más cercana y la instancia más lejana están a distancias muy similares.

Las distancias entre puntos en el espacio de atributos son utilizadas en multitud de algoritmos de aprendizaje automático, entre los cuales el ejemplo más claro es la técnica del Vecino más cercano [49], aplicada en clasificación en el algoritmo de los k-vecinos más cercanos.

Ante un conjunto de datos de alta dimensionalidad, podemos discernir dos vías de acción:

- Estudiar y transformar los datos manteniendo la dimensionalidad, para que las distancias entre puntos sean significativas.
- Reducir la dimensionalidad de los datos, manteniendo toda la información útil que sea posible.

En nuestro caso, utilizaremos algunas técnicas de Deep Learning para operar de la segunda forma, comprimiendo los datos en un espacio de menor dimensionalidad.

Con el objetivo de reducir la dimensionalidad de un conjunto de datos, se alteran las características del mismo en un proceso denominado *extracción de características*, que comprende dos mecanismos alternativos que en ocasiones se combinan [22]:

1. **Construcción de características.** Se transforman los datos, operando entre las características para hallar un nuevo conjunto de atributos que posiblemente facilite el aprendizaje de los datos.
2. **Selección de características.** Se escogen las características que se consideren más relevantes para obtener información, y se descartan las que no son útiles.

En ocasiones, las características construidas en el primer mecanismo se añaden al espacio de características existente para realizar una posterior selección entre todas.

La selección de características se puede realizar mediante muy distintas técnicas, desde metaheurísticas hasta basadas en teoría de la información [39].

Por otro lado, la construcción de características se pone en práctica de formas de variable complejidad. De entre las más sencillas cabe mencionar la normalización, la discretización, o tomar combinaciones lineales de las características existentes. Además, se pueden destacar técnicas más avanzadas como Isomap [49], Locally Linear Embedding [46], o los autoencoders [27]. Este trabajo se centrará en estudiar estos últimos.

6.5 MÉTODOS DE OPTIMIZACIÓN: GRADIENTE DESCENDENTE

Gran parte del trabajo computacional en aprendizaje automático requiere optimizar funciones, es decir, encontrar el máximo o el mínimo de una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Dicha función se suele denominar *función objetivo*, y normalmente corresponde al coste o error de aprendizaje de un algoritmo. En lo sucesivo, se supondrá que el objetivo a conseguir es encontrar el mínimo de la función f . Los resultados y algoritmos son análogos para la búsqueda de un máximo, simplemente cambiando el signo de f .

Estudiemos una simplificación del problema de optimización a una variable. Sea $f : [a, b] \rightarrow \mathbb{R}$ continua en $[a, b]$ y derivable con derivada continua en $]a, b[$. Por un resultado elemental de análisis matemático,

es conocido que un extremo (máximo o mínimo) local en una función derivable se encuentra siempre en un punto de derivada cero. Además, por el teorema del valor medio, para $\varepsilon > 0$ se tiene que

$$\exists c \in]a, a + \varepsilon[: f(a + \varepsilon) - f(a) = \varepsilon f'(c) \Rightarrow f(a + \varepsilon) = f(a) + \varepsilon f'(c)$$

y, por continuidad de f' , si ε es suficientemente pequeño el signo de la derivada no cambiará entre a y c . Así,

$$f'(a) < 0 \Rightarrow f(a + \varepsilon) < f(a); f'(a) > 0 \Rightarrow f(a + \varepsilon) > f(a).$$

El método del gradiente descendente, en una variable, consiste en evaluar f a pequeños saltos de ε y consultar la derivada para decidir el sentido del próximo salto.

En el caso de varias variables, el algoritmo es muy similar. Sabiendo que el gradiente de una función real de varias variables $f : \mathbb{R}^n \rightarrow \mathbb{R}$ es un vector que apunta en la dirección y sentido de la mayor pendiente (derivada direccional) ascendente de la función, el gradiente cambiado de signo estará posicionado en el sentido de mayor pendiente descendente.

Así, suponiendo la suficiente regularidad para f y que se puede evaluar tanto la función como su derivada en cualquier punto del dominio, se puede aplicar el algoritmo de gradiente descendente, originalmente debido a Cauchy [9].

El algoritmo generalizado consiste en, a cada paso determinado por un punto x del dominio, consultar el gradiente de f en x , $\nabla f(x)$, y “saltar” una cantidad $\varepsilon > 0$ en su dirección y en el sentido contrario:

$$x' = x - \varepsilon \nabla f(x)$$

Al escalar ε se le suele llamar *tasa de aprendizaje*.

El algoritmo termina cuando $\nabla f(x)$ es el vector cero o muy cercano a cero, con una tolerancia dada.

La técnica de gradiente descendente presenta algunos problemas: como se puede observar en la Figura 6.1, cuando el gradiente de la función es próximo a cero el algoritmo tiende a dar pasos muy cortos, convergiendo muy lentamente hacia el extremo local encontrado. Asimismo, en general puede presentar un comportamiento de zigzag para ciertas funciones, avanzando de forma casi ortogonal al segmento que guarda la distancia más corta con el extremo.

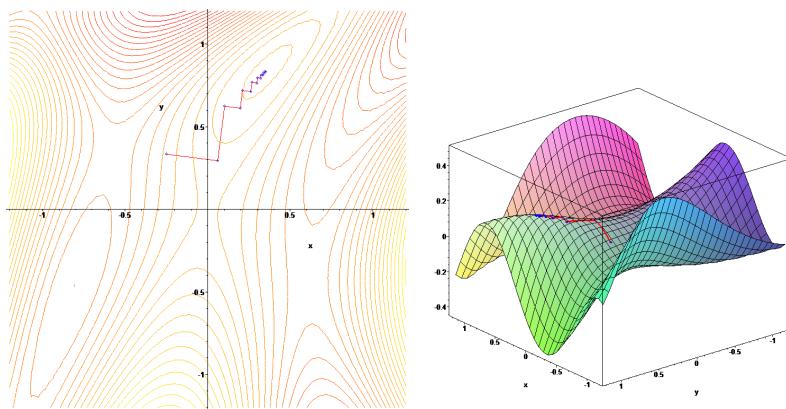


Figura 6.1: Visualización de la técnica de gradiente descendente, en este caso, buscando un máximo de la función $F(x,y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right) \cos(2x + 1 - e^y)$. Imágenes de Wikimedia Commons en dominio público

DEEP LEARNING

En este capítulo nos centramos en el ámbito de las técnicas que aplicaremos en la práctica, el Deep Learning. Estas técnicas están basadas en redes neuronales, de las que introducimos algunos aspectos al hablar de las redes prealimentadas profundas. Se describen y se clasifican las unidades que las componen. Posteriormente, se expone el proceso de evaluación de una red de este tipo mediante la propagación hacia adelante y hacia atrás. Además, se especifican los algoritmos derivados de gradiente descendente que permiten realizar aprendizaje sobre ellas. Por último, se definen las estructuras principales de redes que efectúan un aprendizaje no supervisado sobre los datos.

7.1 REDES NEURONALES PREALIMENTADAS PROFUNDAS

Las redes prealimentadas profundas, también conocidas como perceptrones multicapa o en inglés como *deep feedforward neural networks*, son el modelo canónico de aprendizaje profundo [21]. El objetivo de una red prealimentada es aproximar una función f^* , definiendo una aplicación $f(x; \theta)$ y aprendiendo el valor de los parámetros θ que resultan en la mejor aproximación.

En concreto, las redes prealimentadas se caracterizan por que no se forman ciclos en las conexiones entre unidades. Así, la información se evalúa siempre hacia adelante a través de las conexiones intermedias usadas para definir f , hasta la salida de la red. No hay retroalimentaciones en las que salidas de algunas unidades de la red vuelvan a ser entradas del modelo.

Estas redes se suelen representar como una composición en cadena de varias funciones, que se puede asociar a un grafo acíclico. Por ejemplo, podríamos tener una red composición de funciones vectoriales f_1, f_2, f_3 de la siguiente forma: $f(x) = f_3(f_2(f_1(x)))$. En este caso, decimos que f_1 es la primera capa, f_2 la segunda capa y f_3 la capa de salida. Las capas que no corresponden a la salida de f se suelen denominar *capas ocultas*. La longitud de esta cadena nos da la profundidad del modelo.

A diferencia de otros algoritmos de aprendizaje automático, las redes neuronales mantienen esta estructura de capas de forma que la capa $i + 1$ -ésima únicamente opera con los datos de salida de la i -ésima; en particular, sólo la primera capa utiliza directamente los datos de entrada. Además, por la inspiración biológica de las redes, cada componente de cada capa (*unidad*) se puede interpretar como una

neurona. Las neuronas de los seres vivos se componen de dendritas que recogen estímulos de otras neuronas, un núcleo que los acumula y un axón con terminales que se conectan a otras neuronas y le permiten transmitir nuevos impulsos. Se exemplifica esta estructura en la Figura 7.1.

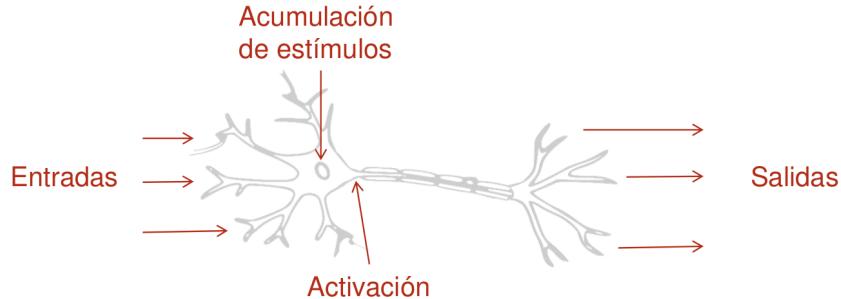


Figura 7.1: Ilustración de una neurona biológica, indicando las equivalencias con la neurona artificial

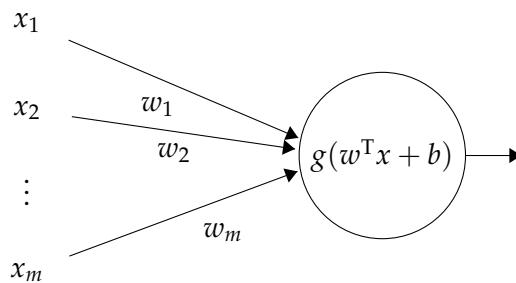


Figura 7.2: Una neurona artificial con m entradas y función de activación g

La analogía se lleva a la neurona artificial, como se muestra en la Figura 7.2, mediante una función de \mathbb{R}^m en \mathbb{R} , donde m es el número de unidades en la capa anterior. El comportamiento es similar a una neurona en el sentido de que recoge información de varias unidades cercanas y calcula su propio valor de activación, así como la estructuración en capas se ha tomado de la neurociencia. En la figura 7.3 se muestra una representación común de una red neuronal como unidades conectadas formando un grafo. Las flechas indican el sentido en el que viajan los datos, es decir, las salidas de funciones que se toman como entradas de otras funciones.

Para entender cómo las redes prealimentadas aproximan funciones, consideremos algunos modelos lineales como la regresión lineal o la logística. Estos modelos tienen claras ventajas, son sencillos, se pueden ajustar de forma eficiente y fiable. Sin embargo, están muy limitados, dado que sólo tiene sentido aplicarlos a funciones lineales, por lo que no pueden sintetizar interacciones entre dos variables de entrada.

Cuando el objetivo es aproximar funciones no lineales, una vía es aplicar un modelo lineal no a la variable independiente sino a una

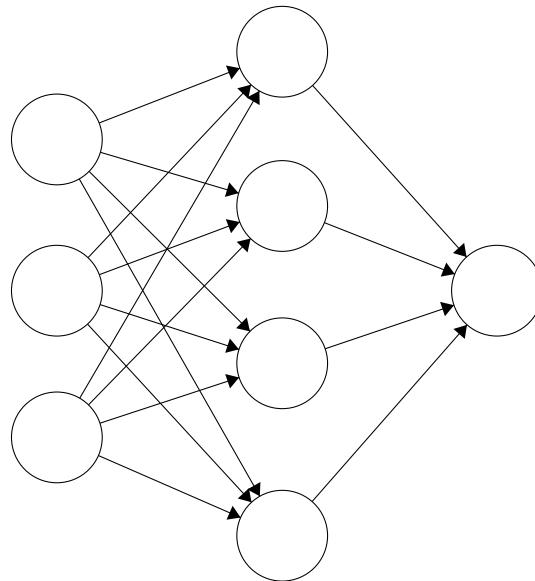


Figura 7.3: Ilustración exemplificando una red neuronal prealimentada de tres capas

transformación no lineal de la misma. El problema se traduce entonces en qué transformación ϕ de la variable aplicar para que el modelo lineal tenga un buen ajuste. Frente a buscar ϕ manualmente, que requiere extenso conocimiento de cada problema, o usar un ϕ de muy alta dimensionalidad con capacidad para todos los ejemplos del conjunto de datos, las redes neuronales realizan un aprendizaje de ϕ entre una clase de funciones parametrizada: se define un modelo del tipo

$$f^*(x) \approx f(x; \theta, w) = \phi(x; \theta)^T w, \quad (7.1)$$

donde θ es un vector de parámetros que facilita escoger una función f concreta de entre la clase que define, y w es otro vector de parámetros que permite aplicar la transformación obtenida en la salida deseada. Para encontrar los parámetros que corresponden a una buena aproximación, se utilizará un algoritmo de optimización basado en la técnica de gradiente descendente presentada en la sección 6.5. Se trata de un enfoque muy flexible, ya que se puede proveer al algoritmo de una clase de funciones más general o más concreta, según el conocimiento sobre el problema que se posea.

La clase de funciones, dentro de la cual una red neuronal busca la aproximación, se determina escogiendo la estructura de la red y los tipos de unidades ocultas y de salida.

7.1.1 Funciones de coste

La mayoría de diseños de redes neuronales involucran definir una distribución $P(y | x; \theta)$ y aplicar el principio de máxima verosimilitud. En otros casos, mediante funciones de coste específicas, se puede predecir simplemente algún estadístico de y condicionado a x , en lugar de determinar una distribución de probabilidad.

En el caso más habitual, la función de coste se definirá como la entropía cruzada (equivalentemente, la log-verosimilitud negativa) entre la distribución de los datos, \hat{p} , y la del modelo, p , y sobre la variable de la salida generada y respecto de la entrada x :

$$J(\theta) = C(\hat{p}(y | x), p(y | x)) = -E_{\hat{p}}[\log p(y | x)].$$

Una ventaja de este enfoque es que esta función de coste viene determinada automáticamente por el modelo $p(y | x)$ que escojamos y evita tener que definir una nueva función para cada modelo. Además, la entropía cruzada suele permitir calcular gradientes relativamente “grandes”, en el sentido de que no se acercan rápidamente a cero, lo cual beneficia al proceso de optimización.

En ocasiones se añade a la función de coste un término de regularización o *decaimiento de pesos* de forma que el coste total queda:

$$J(\theta) = C(\hat{p}, p; y | x) + \lambda \Omega(\theta)$$

7.1.2 Unidades de salida

La expresión concreta de la función de coste, cuando la tomamos como la entropía cruzada, vendrá determinada por la representación de la salida de la red prealimentada. Estudiamos a continuación el tipo de unidades que se suelen utilizar para dar dicha salida. Durante el resto de esta sección, supondremos que la red proporciona un vector de características $h = f(x; \theta)$ generado por las unidades ocultas. El cometido de las unidades de salida es dar una transformación que aporte una salida apropiada.

7.1.2.1 Unidades lineales

Una capa de unidades de este tipo realiza una transformación afín de los datos: $\hat{y} = W^T h + b$. Se suelen utilizar para calcular la media de una distribución condicional normal:

$$p(y | x) = \mathcal{N}(y; \hat{t}, I).$$

En ese caso, maximizar la entropía cruzada es equivalente a minimizar el error cuadrático medio.

7.1.2.2 Unidades con activación sigmoidal

En muchas tareas, la variable objetivo y es de tipo binario. Por ejemplo, los problemas de clasificación binaria son un caso particular de esta situación. La técnica de máxima verosimilitud lleva a definir una distribución de Bernoulli sobre y condicionada a x . Esta distribución está determinada por un único número en el intervalo $[0, 1]$, que se corresponde con $P(y = 1 | x)$.

Para que la unidad de salida tenga un buen comportamiento, es necesario que no genere gradiente 0 ni muy cercano a 0 en casos en los que el modelo no se acerque a una solución. Esto se consigue utilizando una *función de activación*, es decir, se compone el cómputo de la unidad con otra función que la regulariza de alguna manera. En este caso, se utiliza la función logística:

$$z = w^T h + b; \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

En la [Figura 7.4](#) se observan los valores en los que se aplica σ según el valor de z .

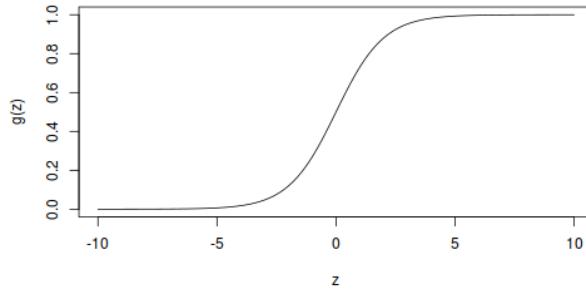


Figura 7.4: Gráfica de la función logística

Definamos ahora una distribución de probabilidad sobre y , usando el valor z . Podemos comenzar asumiendo que la probabilidad no normalizada \tilde{P} es log-lineal en z e y :

$$\log \tilde{P}(y) = yz,$$

y exponentiamos

$$\tilde{P}(y) = e^{yz},$$

para ahora normalizar sobre los valores de \tilde{P} , obteniendo una probabilidad

$$P(y) = \frac{e^{yz}}{e^{0z} + e^{1z}} = \frac{e^{yz}}{1 + e^z}$$

y utilizando de nuevo que $y \in \{0, 1\}$ se tiene

$$P(y) = \frac{e^{yz}}{e^{(1-y)z} + e^{yz}} = \frac{1}{\frac{e^z}{e^{2yz}} + 1} = \sigma((2y - 1)z). \quad (7.2)$$

El resultado es una distribución de Bernoulli determinada por una transformación logística de z . De hecho, puesto que los posibles valores de y son 0 y 1, también podemos expresarla más claramente como:

$$P(y) = \sigma(z)^y \sigma(-z)^{1-y} = p^y(1-p)^{1-y} \text{ donde } p = \sigma(z).$$

Ahora, la función de coste para esta distribución, tomando la entropía cruzada y usando (7.2), es:

$$J(\theta) = -\log P(y | x) = -\log \sigma((2y-1)z).$$

Dado que la función logística está valuada en el intervalo abierto $]0, 1[$, su logaritmo es finito y J está bien definida.

7.1.2.3 Unidades con activación softmax

Las unidades con función de activación *softmax* se emplean cuando se pretende representar una distribución de probabilidad sobre una variable discreta con un número finito de valores. Generalmente, esta situación se da en problemas de clasificación multiclase. Así, se pueden interpretar como una generalización de las unidades sigmoidales.

Mientras que para caracterizar una variable binaria bastaba con una sola unidad de salida (la salida era un escalar entre 0 y 1), ahora se utilizarán tantas unidades como posibles valores puedan tomar la variable. Si y puede tomar uno de entre n posibles valores, la capa de salida con *softmax* generará un vector \hat{y} donde $\hat{y}_i = P(y = i | x)$, exigiendo que $\sum_{i=1}^n \hat{y}_i = 1$.

La función vectorial *softmax* se define en cada componente $i = 1, \dots, n$ como

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^n \exp(z_j)}. \quad (7.3)$$

El vector z al que se aplica la función se obtiene de una capa de unidades lineales que proporcionan probabilidades logarítmicas sin normalizar:

$$z = W^T h + b; z_i = \log \tilde{P}(y = i | x).$$

De nuevo, la función de coste se puede definir siguiendo la misma técnica, mediante la entropía cruzada.

7.1.3 Unidades ocultas

Cualquiera de los tipos anteriores de unidad se puede utilizar en una capa oculta, pero existen más y el diseño de unidades ocultas es

un campo de investigación muy activo, pese a la falta de principios teóricos que lo guíen. A continuación se exponen algunos de los tipos más usuales. Salvo que se indique lo contrario, todas las capas de unidades calculan una transformación afín

$$z = W^T x + b,$$

donde x es el vector de entrada, equivalentemente, el vector de salida de la capa inmediatamente anterior o un vector de datos si se trata de la primera capa. Se compone esta transformación con la función de activación específica a cada tipo de unidad.

7.1.3.1 Unidades lineales rectificadas (ReLU)

La función de activación de las unidades lineales rectificadas (*Rectified Linear Units*, ReLU) es

$$g(z)_i = \max\{0, z_i\}. \quad (7.4)$$

Se muestra en la [Figura 7.5](#) el valor que toma la ReLU según el valor de z_i .

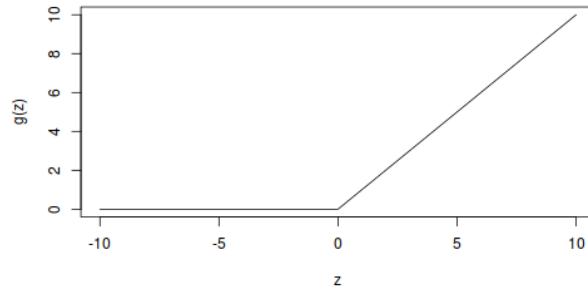


Figura 7.5: Gráfica de la función de activación de una ReLU

Estas unidades son fáciles de optimizar ya que son similares a las unidades lineales. Aunque no es diferenciable en $z = 0$, se pueden utilizar algoritmos basados en gradiente para optimizar la función objetivo. Esto es debido a que, generalmente, durante el entrenamiento no se llega a un punto en el que el gradiente sea exactamente 0, así que se pueden aceptar puntos donde el gradiente no esté definido en los mínimos de la función coste.

7.1.3.2 Extensiones de ReLU

Algunas generalizaciones de las ReLU modifican el gradiente cuando las componentes de z son negativas:

- **Rectificación por valor absoluto:** $g(z)_i = |z_i|$

- **Leaky ReLU:** $g(z)_i = \max(0, z_i) + \alpha \min(0, z_i)$ con α pequeño como 0,01 ([Figura 7.6](#))
- **ReLU paramétrica:** $g(z)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$, con α_i como un parámetro optimizable

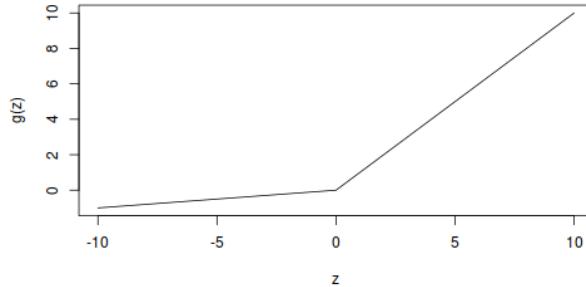


Figura 7.6: Función de activación de una *leaky ReLU* para $\alpha = 0,1$

Las capas de **unidades maxout** agrupan las componentes de z en conjuntos de k valores cada uno. Cada una de las unidades de la capa proporciona entonces el máximo de uno de esos conjuntos:

$$g(z)_i = \max\{z_j : j \in S_i\}, \quad S_i = \{(i-1)k+1, (i-1)k+2, \dots, ik\}.$$

En este caso, si $z \in \mathbb{R}^{kd}$, entonces $g(z) \in \mathbb{R}^d$. El aspecto interesante de las unidades *maxout* es el hecho de que una capa de ellas puede aprender una función convexa lineal a trozos de hasta k trozos. Podemos intuir que una capa de este tipo podrá aproximar cualquier función convexa con precisión arbitraria para un k conveniente.

7.1.3.3 Unidades con activación sigmoidal o tangente hiperbólica

Otras dos funciones muy comunes para la activación de unidades en redes neuronales son la función logística

$$g(z)_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}, \quad (7.5)$$

y la tangente hiperbólica ([Figura 7.7](#))

$$g(z)_i = \tanh(z_i) = \frac{e^{z_i} - e^{-z_i}}{e^{z_i} + e^{-z_i}}. \quad (7.6)$$

Se puede comprobar que $\tanh(z_i) = 2\sigma(2z_i) - 1$.

El uso de estas unidades era más común antes de la aparición de las ReLU. En la actualidad está decreciendo su uso.

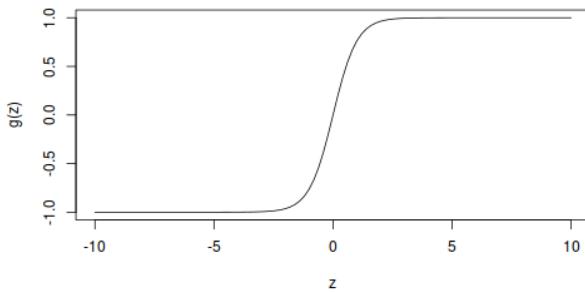


Figura 7.7: Gráfica de la función tangente hiperbólica

7.1.3.4 Otras unidades ocultas

Para construir otros tipos de unidad oculta simplemente basta con elegir otra función de activación. Las siguientes son algunas relativamente comunes:

- El **coseno** $g(z)_i = \cos(z_i)$ ha sido usada por Goodfellow, Bengio y Courville [21] en el conocido conjunto MNIST obteniendo una tasa de error inferior al 1 %.
- La **identidad** $g(z) = z$ se puede utilizar para encadenar capas de forma lineal y utilizar alguna función de activación diferente a la salida.
- La función *softplus* $g(z)_i = \zeta(z_i) = \log(1 + e^{z_i})$ es una versión infinitamente derivable de la unidad lineal rectificada (Figura 7.8). Sin embargo, en la práctica no suele presentar ventajas sobre la ReLU.

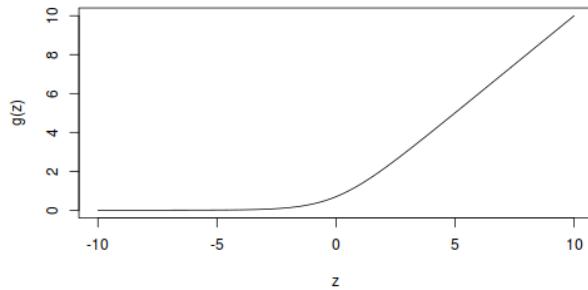


Figura 7.8: Gráfica de la función *softplus*

7.2 ENTRENAMIENTO DE REDES NEURONALES PROFUNDAS

El proceso de entrenamiento de una red neuronal profunda requiere realizar observaciones sobre las salidas a partir de los datos de entra-

da, y evaluar el error cometido respecto a las salidas deseadas. Para calcular la salida de la red $f(x)$ frente a una instancia x se utiliza el mecanismo de propagación hacia adelante, en el que intuitivamente podemos decir que los datos “atraviesan” la red. La evaluación de la red se realiza en base a una función de coste y posteriormente se deben actualizar los parámetros para alterar la función f , para lo cual es necesario calcular el gradiente de dicho coste. El cómputo del gradiente se obtiene mediante la técnica de propagación hacia atrás.

7.2.1 Propagación hacia adelante

Las redes neuronales prealimentadas, que se han estudiado en la sección 7.1, son funciones que aceptan vectores de entrada y procesan la información computando varias funciones intermedias, propagando así la información, hasta la salida de la red. Este proceso se denomina *propagación hacia adelante*, y se describe en el algoritmo 1.

Algoritmo 1 Propagación hacia adelante en una red neuronal profunda con función de activación g , y cálculo de la función de coste J , para una instancia x (en la práctica se utilizan minibatches de instancias)

Entrada: profundidad de la red l

Entrada: $W^{(i)}$ matriz de pesos de la capa i -ésima

Entrada: $b^{(i)}$ vector de sesgos de la capa i -ésima

Entrada: instancia x a procesar

Entrada: salida objetivo y^*

```

 $h^{(0)} \leftarrow x$ 
for  $k = 1, \dots, l$  do
     $z^{(k)} \leftarrow W^{(k)} h^{(k-1)} + b^{(k)}$ 
     $h^{(k)} \leftarrow g(z^{(k)})$ 

```

end for

```

 $y \leftarrow h^{(l)}$ 

```

Se calcula la función de coste mediante una distancia o pérdida entre la salida obtenida y la deseada, y un término de regularización Ω :

```

 $J \leftarrow L(y, y^*) + \lambda \Omega(\theta)$ 

```

7.2.2 Propagación hacia atrás

Consideremos una red neuronal prealimentada profunda, determinada por un vector de parámetros θ . Durante el entrenamiento, la salida generada por la propagación hacia adelante se compara con la salida deseada y se calcula un coste $J(y, y^*; \theta) \in \mathbb{R}$. Para aplicar un algoritmo de optimización basado en gradiente descendente (se estudiarán en la sección 7.3), es necesario conocer el gradiente de la función J

respecto de los parámetros θ . Este gradiente se puede calcular analíticamente, pero evaluar $\nabla_{\theta}J(y, y^*; \theta)$ es generalmente muy costoso computacionalmente. El algoritmo de propagación hacia atrás, o *back-prop*, realiza este cálculo de forma eficiente.

Ejemplo 7.1. Consideremos la red neuronal de la figura 7.9. Suponiendo que cada neurona utiliza la misma función de activación g , podemos dar una expresión de f acorde con los pesos y sesgos que se muestran en la imagen.

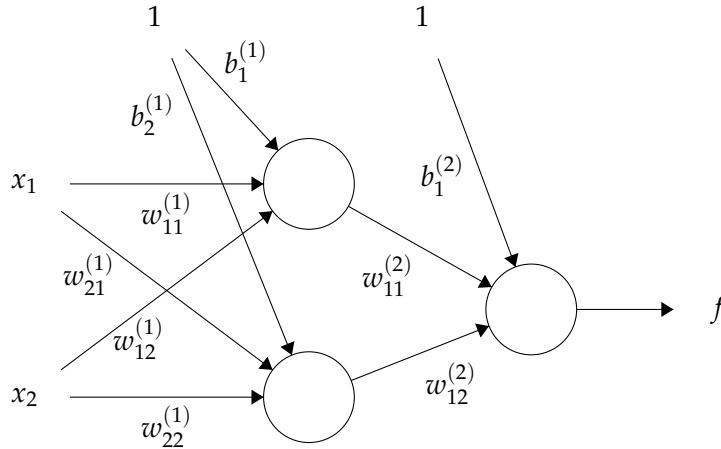


Figura 7.9: Red neuronal sencilla de dos capas con entrada vectorial de dos componentes, se marcan los pesos y los sesgos en cada conexión

En este caso, el vector de parámetros que determina la red será

$$\theta = (w_{11}^{(1)}, w_{12}^{(1)}, w_{21}^{(1)}, w_{22}^{(1)}, b_1^{(1)}, b_2^{(1)}, w_{11}^{(2)}, w_{12}^{(2)}, b_1^{(2)}).$$

Puesto que la función de coste J vendrá determinada por el valor de f en el mismo punto, para calcular su gradiente nos interesa conocer el de f . La expresión desarrollada de f queda

$$f(x_1, x_2; \theta) = g\left(w_{11}^{(2)} g\left(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}\right) + w_{12}^{(2)} g\left(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}\right) + b_1^{(2)}\right).$$

Ahora, mediante la regla de la cadena podemos desarrollar la parcial de f respecto de cualquiera de los parámetros. Llamamos

$$\alpha = g'\left(w_{11}^{(2)} g\left(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}\right) + w_{12}^{(2)} g\left(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}\right) + b_1^{(2)}\right)$$

$$\beta = g'\left(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}\right)$$

$$\gamma = g'\left(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}\right)$$

y se tiene

$$\begin{aligned}
 \frac{\partial f}{\partial w_{11}^{(1)}}(x_1, x_2; \theta) &= \alpha w_{11}^{(2)} \beta x_1, & \frac{\partial f}{\partial w_{12}^{(1)}}(x_1, x_2; \theta) &= \alpha w_{11}^{(2)} \beta x_2, \\
 \frac{\partial f}{\partial w_{21}^{(1)}}(x_1, x_2; \theta) &= \alpha w_{12}^{(2)} \gamma x_1, & \frac{\partial f}{\partial w_{22}^{(1)}}(x_1, x_2; \theta) &= \alpha w_{12}^{(2)} \gamma x_2, \\
 \frac{\partial f}{\partial b_1^{(1)}}(x_1, x_2; \theta) &= \alpha w_{11}^{(2)} \beta, & \frac{\partial f}{\partial b_2^{(1)}}(x_1, x_2; \theta) &= \alpha w_{12}^{(2)} \gamma, \\
 \frac{\partial f}{\partial w_{11}^{(2)}}(x_1, x_2; \theta) &= \alpha g(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}), \\
 \frac{\partial f}{\partial w_{12}^{(2)}}(x_1, x_2; \theta) &= \alpha g(w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}), & \frac{\partial f}{\partial b_1^{(2)}}(x_1, x_2; \theta) &= \alpha.
 \end{aligned}$$

Como podemos observar, algunos de los factores de las parciales se repiten en varias de ellas, de forma que se ahorrarán muchos cálculos innecesarios si no se repiten. Este hecho se hace aún más evidente conforme se añaden capas a la red y unidades a cada capa. Por ello, el algoritmo de propagación hacia atrás permite optimizar el cálculo del gradiente mediante varios pasos intermedios para evitar cálculos repetidos.

En el algoritmo 2 se describe *backprop* paso a paso. Es fácil comprobar que aplicando esta técnica al ejemplo anterior podemos evaluar las parciales que se han deducido sin repetir cálculos costosos.

Algoritmo 2 Propagación hacia atrás

Tras la propagación hacia adelante, calcular el gradiente de la capa de salida:

$$d \leftarrow \nabla_y J(y, y^*; \theta) = \nabla_y L(y, y^*)$$

for $k = l, \dots, 1$ **do**

Aplicar la regla de la cadena a la función de activación (\odot denota producto componente a componente):

$$d \leftarrow \nabla_{z^{(k)}} J = d \odot g'(z^i)$$

Calcular gradientes en los pesos y sesgos (incluyendo el término de regularización si es necesario):

$$\nabla_{b^{(k)}} J = d + \lambda \nabla_{b^{(k)}} \Omega(\theta)$$

$$\nabla_{W^{(k)}} J = d(h^{(k-1)})^\top + \lambda \nabla_{W^{(k)}} \Omega(\theta)$$

Propagar el gradiente hacia la capa oculta anterior:

$$d \leftarrow \nabla_{h^{(k-1)}} J = (W^{(k)})^\top d$$

end for

7.3 OPTIMIZACIÓN EN DEEP LEARNING

Como se ha visto, una red profunda define una función f que queda determinada por una secuencia finita de parámetros. Estos parámetros se deben optimizar para que la salida de la red sea lo más próxima posible a la salida deseada. Para ello, sería interesante utilizar

la técnica de gradiente descendente estudiada en la [Sección 6.5](#). Sin embargo, el coste computacional de calcular el gradiente respecto del conjunto de datos al completo lo hace inviable. En esta sección se introducen algoritmos derivados de gradiente descendente que aproximan el mismo comportamiento pero resultan mucho más eficientes.

7.3.1 Gradiente descendente estocástico (SGD)

En aprendizaje automático, y especialmente en Deep Learning, es común utilizar conjuntos de datos con un gran número de instancias, para favorecer la capacidad de generalización de los modelos producidos por los algoritmos. Esto provoca que el coste computacional de calcular cada paso de un gradiente descendente haga inviable su uso. Sin embargo, se puede utilizar una aproximación estocástica al algoritmo denominada gradiente descendente estocástico (*Stochastic Gradient Descent*, SGD). En esta versión de gradiente descendente se asienta la mayor parte del desarrollo del Deep Learning en la actualidad.

Al ser una aproximación estocástica, SGD calcula un estimador del gradiente de la función objetivo a partir de un número reducido de muestras. Se describe en el algoritmo 3.

Algoritmo 3 Gradiente descendente estocástico, iteración k -ésima

Entrada: Tasa de aprendizaje ε_k

Entrada: Parámetro inicial θ

while no se alcanza criterio de parada **do**

 Escoger un minilote de m instancias del conjunto de entrenamiento $x^{(1)}, \dots, x^{(m)}$ con correspondientes objetivos $y^{(i)}$

 Calcular estimador del gradiente: $\hat{g} \leftarrow \frac{1}{m} \nabla \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Actualizar parámetro: $\theta \leftarrow \theta - \varepsilon_k \hat{g}$

end while

7.3.2 Variantes de SGD

Existen múltiples variantes de SGD que adaptan el aprendizaje de distintas formas a lo largo de las iteraciones. En muchos casos mejoran respecto al comportamiento de SGD pero esto varía según el conjunto de datos tratado.

- **SGD con momento:** el momento es un término adicional que fuerza a que SGD varíe menos la dirección de una iteración a otra. De esta forma, el zigzagueo característico de GD, también presente en SGD, se atenúa. Esta versión se describe en el algoritmo 4.

- **AdaGrad** [17] es una versión adaptativa de SGD, en el sentido de que varía los parámetros de forma inversamente proporcional a la raíz cuadrada de la suma de los cuadrados de los valores anteriores. Así, en lugar de decrementar la tasa de aprendizaje de igual forma para todos los parámetros, la decremente más rápido en los parámetros que tienen mayores derivadas parciales. Como resultado adicional, en las zonas de menor pendiente del espacio de parámetros el algoritmo progresará más rápidamente que SGD. Se describe en el algoritmo 5.
- **RMSProp**, detallado en el algoritmo 6, sustituye la acumulación de gradientes de AdaGrad por una media exponencial, de forma que tenga mejor comportamiento al optimizar funciones no convexas.
- **Adam** es un algoritmo que también adapta la tasa de aprendizaje y además introduce un momento adaptativo, se puede considerar una combinación de RMSProp con momento. Se describe en el algoritmo 7.

Algoritmo 4 Gradiente descendente estocástico con momento

Entrada: Tasa de aprendizaje ϵ , momento α

Entrada: Parámetro inicial θ , velocidad inicial v

while no se alcanza criterio de parada **do**

 Escoger un minilote de m instancias del conjunto de entrenamiento $x^{(1)}, \dots, x^{(m)}$ con correspondientes objetivos $y^{(i)}$

 Calcular estimador del gradiente: $\hat{g} \leftarrow \frac{1}{m} \nabla \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Actualizar la velocidad: $v \leftarrow \alpha v - \epsilon \hat{g}$

 Actualizar parámetro: $\theta \leftarrow \theta + v$

end while

Algoritmo 5 Adagrad

Notación: \odot es el producto componente a componente, $\sqrt{\cdot}$ es la raíz cuadrada componente a componente y la división por $\frac{1}{\delta + \sqrt{r}}$ se realiza componente a componente.

Entrada: Tasa de aprendizaje ϵ , constante pequeña δ

Entrada: Parámetro inicial θ

 Inicializar: $r \leftarrow 0$

while no se alcanza criterio de parada **do**

 Escoger un minilote de m instancias del conjunto de entrenamiento $x^{(1)}, \dots, x^{(m)}$ con correspondientes objetivos $y^{(i)}$

 Calcular estimador del gradiente: $\hat{g} \leftarrow \frac{1}{m} \nabla \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Acumular cuadrado del gradiente: $r \leftarrow r + \hat{g} \odot \hat{g}$

 Calcular actualización: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g}$

 Actualizar parámetro: $\theta \leftarrow \theta + \Delta\theta$

end while

Algoritmo 6 RMSProp

Notación: De nuevo, las operaciones \odot , raíz cuadrada y división se realizan componente a componente.

Entrada: Tasa de aprendizaje ε , constante pequeña δ

Entrada: Tasa de decaimiento ρ

Entrada: Parámetro inicial θ

Iniciar: $r \leftarrow 0$

while no se alcanza criterio de parada **do**

 Escoger un minilote de m instancias del conjunto de entrenamiento $x^{(1)}, \dots, x^{(m)}$ con correspondientes objetivos $y^{(i)}$

 Calcular estimador del gradiente: $\hat{g} \leftarrow \frac{1}{m} \nabla \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Acumular cuadrado del gradiente: $r \leftarrow \rho r + (1 - \rho) \hat{g} \odot \hat{g}$

 Calcular actualización: $\Delta\theta \leftarrow -\frac{\varepsilon}{\sqrt{\delta+r}} \odot \hat{g}$

 Actualizar parámetro: $\theta \leftarrow \theta + \Delta\theta$

end while

Algoritmo 7 Adam

Entrada: Tasa de aprendizaje ε , constante pequeña δ

Entrada: Tasas de decaimiento exponencial $\rho_1, \rho_2 \in [0, 1[$

Entrada: Parámetro inicial θ

Iniciar: $s \leftarrow 0, r \leftarrow 0, t \leftarrow 0$

while no se alcanza criterio de parada **do**

 Escoger un minilote de m instancias del conjunto de entrenamiento $x^{(1)}, \dots, x^{(m)}$ con correspondientes objetivos $y^{(i)}$

 Calcular estimador del gradiente: $\hat{g} \leftarrow \frac{1}{m} \nabla \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Incrementar tiempo: $t \leftarrow t + 1$

 Actualizar estimador sesgado del 1º momento: $s \leftarrow \rho_1 s + (1 - \rho_1) \hat{g}$

 Actualizar estimador sesgado del 2º momento: $r \leftarrow \rho_2 r + (1 - \rho_2) \hat{g} \odot \hat{g}$

 Corregir sesgos: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}, \hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$

 Calcular actualización: $\Delta\theta \leftarrow -\frac{\varepsilon}{\delta + \sqrt{\hat{r}}} \hat{s}$ (operaciones componente a componente)

 Actualizar parámetro: $\theta \leftarrow \theta + \Delta\theta$

end while

7.4 ESTRUCTURAS PROFUNDAS NO SUPERVISADAS

En esta sección nos centramos en las estructuras dedicadas a aprendizaje no supervisado en Deep Learning. Estudiaremos las máquinas de Boltzmann restringidas (*Restricted Boltzmann Machine*, RBM) y los autoencoders y sus variantes. También analizaremos una propuesta de entrenamiento específica para autoencoders. La fuente principal de esta sección es Goodfellow, Bengio y Courville [21, capítulos 14 y 20].

7.4.1 Máquina de Boltzmann restringida (RBM)

Las máquinas de Boltzmann restringidas o RBMs son modelos basados en energía no direccionales. No son por sí mismas modelos profundos, pero son la base de algunas arquitecturas de Deep Learning como las *Deep Belief Networks* [26]. Contienen una capa de variables observables o visibles y una sola capa de variables ocultas o latentes.

Se denominan restringidas porque, a diferencia de las máquinas de Boltzmann generales, forman grafos bipartitos donde no hay conexiones entre unidades de la misma capa, como se puede observar en la [Figura 7.10](#). Que sean modelos basados en energía quiere decir que la distribución de probabilidad \tilde{p} que aprenden viene dada de la forma

$$\tilde{p}(x) = e^{-E(x)},$$

donde la función E se llama *función de energía*. Por último, son no direccionales ya que las conexiones entre unidades funcionan en ambos sentidos.

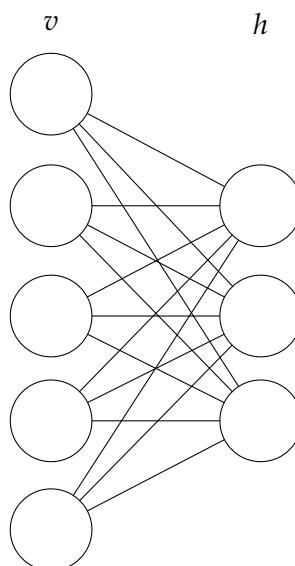


Figura 7.10: Ilustración de las unidades de una máquina de Boltzmann restringida y las conexiones entre ellas

Las RBMs en su versión básica sólo contemplan valores binarios en las variables. Formalmente, si V es un vector de variables aleatorias binarias observables, y llamamos H al vector de variables aleatorias binarias ocultas, la distribución conjunta representada por la RBM asociada es

$$P[V = v, H = h] = \frac{1}{Z} e^{-E(v,h)} ,$$

donde la función de energía E viene dada por la expresión

$$E(v,h) = -b^T v - c^T h - v^T Wh ,$$

y Z es la constante de normalización correspondiente:

$$Z = \sum_v \sum_h e^{-E(v,h)} .$$

Puesto que dicha constante Z es, en general, demasiado costosa de calcular como para que sea factible la evaluación directa de $P(v,h)$, se suelen utilizar otras técnicas para entrenar un modelo de RBM. Estas incluyen *Contrastive Divergence* [23] y máxima verosimilitud estocástica, conocida también como *Persistent Contrastive Divergence* [50].

7.4.2 Autoencoder

Un autoencoder es una red neuronal que se entrena con el objetivo de reproducir la entrada a su salida. Internamente, contiene una capa oculta que describe un código utilizado para representar la entrada. Se puede considerar una red compuesta por dos partes: una función codificadora f que transforma la entrada en el código, y una función decodificadora g que se aplica al código y trata de reconstruir la entrada. Una ilustración de la arquitectura se muestra en la [Figura 7.11](#), donde la etapa de codificación se realiza entre la primera y la segunda capa, y la decodificación entre la segunda y la última.

El interés de los autoencoders reside en restringirlos, mediante la estructura de la red u otros parámetros, de forma que sean incapaces de simplemente aprender la función identidad. Así, la codificación que aprenden contiene información útil acerca de los datos de entrada, ya que se le fuerza a descartar los aspectos que no sean representativos. Si la codificación, además, es de menor dimensionalidad que la entrada, se puede utilizar el autoencoder como reductor de la dimensionalidad.

El concepto de los autoencoders no es nuevo, pero hasta hace unos años no se habían desarrollado técnicas para un entrenamiento eficiente. La aparición de SGD y sus derivados ([Sección 7.3](#)) permitieron entrenar redes profundas de forma rápida, en particular los autoencoders. Además, Hinton y Salakhutdinov [27] introdujeron una técnica de entrenamiento basada en un cálculo de pesos iniciales previo y un posterior ajuste.

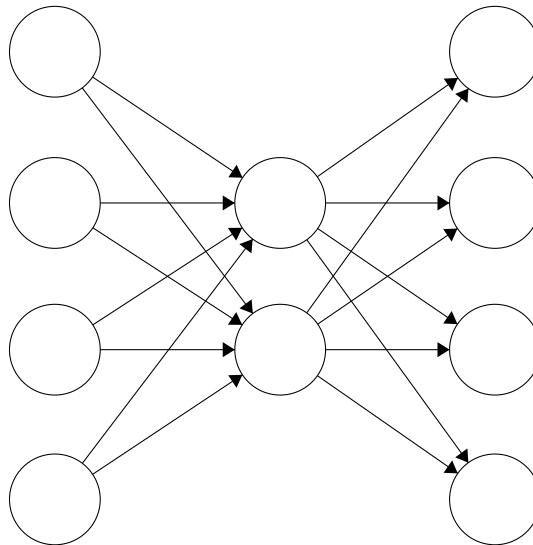


Figura 7.11: Autoencoder de tres capas. Tiene 4 variables de entrada y genera una codificación en 2 variables

Existen diferentes variantes de los autoencoders según su estructura y función de coste, a continuación estudiamos las principales.

7.4.2.1 *Autoencoders infracompletos*

La primera restricción sencilla que permite obtener información interesante a partir del entrenamiento de un autoencoder es reducir la dimensión de la capa interna respecto de la de entrada, como se ve en el ejemplo de la Figura 7.11. La representación de los datos que aprende este autoencoder se puede denominar *infracompleta* (del inglés *undercomplete*). Al entrenar un autoencoder de este tipo, los aspectos más representativos de los datos se quedan en la codificación porque sirven para reconstruirlos.

El proceso de aprendizaje consiste en minimizar una función de coste del tipo

$$L(x, g(f(x))),$$

donde L da una medida de la diferencia entre la entrada y su reconstrucción, como puede ser el error cuadrático medio.

Adicionalmente, se puede añadir un término de *regularización* Ω junto a la función L , que dependa de los valores de la capa de codificación y añada ciertas propiedades deseadas al autoencoder. Algunas de las variaciones que se mencionan a continuación surgen a partir de esta generalización.

7.4.2.2 *Autoencoders dispersos*

Un autoencoder disperso (*sparse autoencoder*) es simplemente un autoencoder cuya función de coste involucra, además del error cometido

en la reconstrucción, una penalización de dispersión sobre la codificación h :

$$L(x, g(f(x))) + \Omega(h) .$$

La penalización de dispersión fuerza al autoencoder a aprender representaciones (infracompletas o sobrecompletas) en las que muy pocas unidades están activas, es decir, la mayor parte dan salida nula. Por esto, los autoencoders dispersos son muy útiles para la construcción de características que se pueden usar en una tarea de clasificación.

7.4.2.3 Autoencoders con eliminación de ruido

Otra tarea que puede aprender un autoencoder es la reparación del ruido en instancias (*denoising autoencoder*). Para ello, si tenemos para cada instancia x una copia \tilde{x} a la que se le ha introducido algún tipo de ruido, entrenamos un autoencoder que minimice la función

$$L(x, g(f(\tilde{x}))) .$$

De esta forma, el autoencoder se ve forzado a aprender la distribución de los datos de forma implícita, y se hace robusto al ruido. Una vez entrenado, podremos inyectar nuevas instancias ruidosas en la red y recuperar instancias limpias en la capa de salida.

7.4.2.4 Autoencoders contractivos

Un autoencoder contractivo (*contractive autoencoder*) es un autoencoder con una regularización del tipo

$$\Omega(h) = \lambda \left\| \frac{\partial f(x)}{\partial x} \right\|_F^2 ,$$

que hace que las parciales de f tiendan a ser lo más pequeñas posible.

Los autoencoders contractivos son robustos a perturbaciones locales en los datos de entrada, es decir, si una instancia x se aplica en $f(x)$, entonces una instancia vecina x' muy cercana a x se aplicará también cerca de $f(x)$. Sin embargo, si los puntos x y x' son lejanos, sus imágenes por f pueden ser aún más lejanas, de ahí la naturaleza local del comportamiento del autoencoder. Intuitivamente, se puede considerar que el autoencoder “contrae” el vecindario de cada instancia.

Una aplicación de los autoencoders contractivos es realizar aprendizaje de variedades (*manifold learning*) sobre los datos.

7.4.3 Entrenamiento de autoencoders

Para el entrenamiento de autoencoders, es posible utilizar las técnicas que se han estudiado anteriormente, en concreto la propagación hacia

atrás ([Sección 7.2.2](#)) para calcular gradientes y los algoritmos basados en gradiente descendente estocástico ([Sección 7.3](#)) para optimizar la función objetivo.

Sin embargo, al emplear autoencoders muy profundos este aprendizaje se puede tornar un proceso muy lento, por el hecho de que la inicialización de pesos no utiliza información acerca de los datos. Para solventar este problema se puede añadir una fase previa de pre-entrenamiento que encuentra unos pesos que acercan el autoencoder a un mínimo local, tras lo cual la fase de aprendizaje se emplea para realizar ajustes sobre dichos pesos [27]. Este mismo proceso de entrenamiento es el que se utiliza en las *Deep Belief Network*, que después se pueden entrenar adicionalmente para tareas supervisadas [26].

7.4.3.1 Pre-entrenamiento

En esta fase, se toman parejas de capas consecutivas de la capa codificadora del autoencoder, comenzando por la de entrada y la siguiente, y se entrena RBMs para cada una.

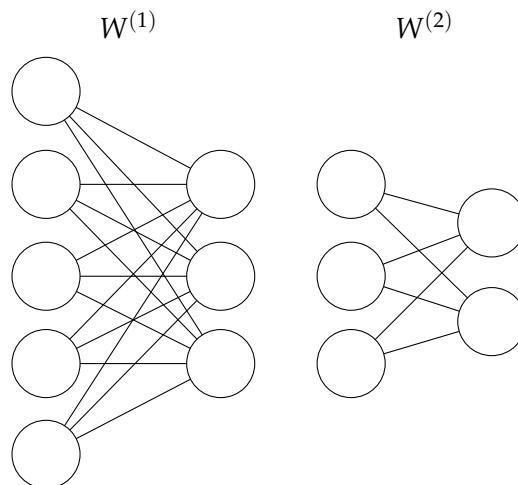


Figura 7.12: Pre-entrenamiento de un autoencoder de 5 capas, con un codificador de 5, 3 y 2 neuronas en la primera, segunda y tercera capa respectivamente

En el ejemplo de la [Figura 7.12](#), se toma la primera capa y la segunda y se entrena una RBM con los datos de entrada. A continuación, la segunda y la tercera forman otra RBM que se entrena con los datos resultantes de atravesar la primera RBM con los datos de entrada. De esta forma, se obtienen sendas matrices de pesos $W^{(1)}$ y $W^{(2)}$.

7.4.3.2 Ajuste fino

Una vez obtenidos unos pesos iniciales mediante el pre-entrenamiento, se “desenrolla” la estructura completa del autoencoder, manteniendo los pesos aprendidos tanto en el codificador como en el decodifica-

dor, como se puede observar en la [Figura 7.13](#). Por último, se ejecuta un optimizador basado en SGD para ajustar estos pesos y los sesgos, para minimizar la función de coste requerida, según el tipo de autoencoder construido.

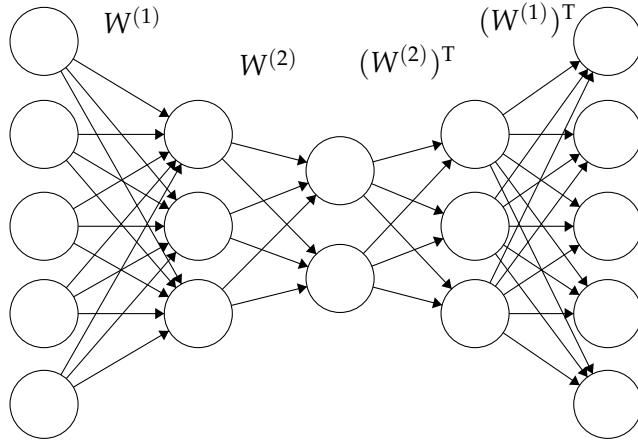


Figura 7.13: Desenrollado del autoencoder, manteniendo las matrices de pesos aprendidas en el pre-entrenamiento

Durante la optimización, los datos se hacen pasar por la red (propagación hacia adelante) en minilotes, es decir, cada gradiente no se calcula respecto del conjunto completo de instancias sino de una muestra de ellas de un tamaño fijo dado. De esta forma, cada iteración del algoritmo es más rápida, sin perder capacidad de generalización ya que en cada iteración hay oportunidad de escoger nuevas instancias. El número de iteraciones también se conoce como número de *épocas*, y suele ser un parámetro escogido por el usuario. Según la rapidez de la convergencia del algoritmo, serán necesarias más o menos épocas para llegar a un mínimo local.

8

LA HERRAMIENTA RUTA

8.1 EL LENGUAJE R

8.1.1 *Introducción*

R [42] es un lenguaje de programación dirigido al tratamiento de datos, y como tal proporciona estructuras de datos y funcionalidades básicas para representar y tratar problemas de minería de datos. Es un proyecto GNU¹, y se considera una implementación alternativa del lenguaje S para estadística, desarrollado originalmente en Bell Labs.

Además, existe toda una plataforma de paquetes para R denominada CRAN², que cuenta con multitud de bibliotecas que facilitan tareas muy diversas, desde lectura y visualización de datos hasta el propio procesamiento mediante distintos algoritmos.

8.1.2 *Instalación*

Para utilizar el software desarrollado, será necesario instalar R junto con las utilidades de desarrollador del lenguaje. En general, hay disponibles paquetes binarios en los repositorios de las distribuciones más comunes de Linux, así como para Windows y macOS. Por ejemplo, para instalar R en distribuciones basadas en Debian como Ubuntu, ejecutaremos el siguiente comando:

```
sudo apt install r-base r-base-dev
```

Los instaladores para Windows y macOS se pueden descargar desde el sitio web del proyecto R³.

¹ GNU es un proyecto extenso que abarca distintos paquetes de software libre, R es uno de ellos: <https://www.gnu.org/>.

² <https://cran.r-project.org/>

³ <https://www.r-project.org/>

8.1.3 Uso del lenguaje

El lenguaje R se puede utilizar de forma interactiva mediante el REPL⁴ que se invoca con el comando R, mediante scripts que se pueden ejecutar con el programa Rscript o desde un IDE como RStudio [43].

R soporta la mayoría de tipos de dato básicos de cualquier otro lenguaje:

- *logical*: valores lógicos entre TRUE, FALSE o NA
- *integer*: valores enteros
- *double*: valores reales de doble precisión (junto con *integer* forman el tipo *numeric*)
- *complex*: números complejos
- *character*: cadenas de caracteres
- *raw*: datos binarios en bruto

Se caracteriza por el uso de algunas estructuras de datos: los vectores atómicos, las matrices, las listas y los *data.frames*. Estos últimos son muy útiles para representar conjuntos de datos.

Las funciones son objetos de primera clase en R. Esto quiere decir que se pueden y suelen pasar como argumento a otras funciones, y devolver como valor de retorno. Esto permite el uso de funciones de orden superior clásicas del tipo *map* o *reduce*, que realizan operaciones simultáneamente sobre varios elementos de una estructura de datos.

R incorpora tres sistemas de orientación a objetos distintos: S3, S4 y *Reference Classes* (RC). El primero de ellos es el más sencillo y el más utilizado, y es también el que se usa en el software desarrollado. Consiste simplemente en objetos tipo lista a los que se le ha aplicado un atributo de clase mediante la función *class*. Este atributo permite escoger un método cuando se realiza una llamada a una función genérica.

Ejemplo 8.1. Para realizar una demostración sobre la orientación a objetos S3 de R, vamos a definir objetos de clase *animal* que implementen el método *print*. Este es ya una función genérica de R, luego la asociación del método que definamos con nuestros objetos será automática. Para crearlos, podemos usar una función que hará las veces de constructor:

```

1 animal <- function(nombre, sonido) {
2   objeto <- list(
3     nombre = nombre,
4     sonido = sonido
5   )
6   class(objeto) <- "animal"
7   return(objeto)

```

⁴ Un REPL (*read-eval-print-loop*) es un entorno de programación interactivo donde se evalúa el código línea a línea.

```

8 }
9 print.animal <- function(animal) {
10   print(paste("Soy un", animal$nombre, "y hago", animal$sonido))
11 }
12 print(animal("gato", "miau"))
13 # => [1] "Soy un gato y hago miau"
14 print(animal("perro", "guau"))
15 # => [1] "Soy un perro y hago guau"

```

Nótese que en las líneas 12 y 14 del código anterior se llama a la función `print`, que al ser genérica busca el método correspondiente para los objetos de clase `animal`.

8.2 LA BIBLIOTECA MXNET

8.2.1 Descripción

MXNet [12] es una biblioteca de algoritmos de Deep Learning, es decir, incluye la funcionalidad necesaria para construir estructuras de aprendizaje profundas, calcular los gradientes con propagación hacia atrás (sección 7.2.2), entrenarlas con datos de entrada mediante los algoritmos de optimización estudiados en la sección 7.3 y realizar predicciones sobre nuevos datos.

Frente a otras bibliotecas similares como Tensorflow [1] o Theano [5], el motor de MXNet está escrito en el lenguaje C++, lo que reduce los tiempos de ejecución. Sin embargo, esto no limita su uso puesto que proporciona acceso a las funcionalidades mediante APIs para otros lenguajes como Python, Scala o R.

Esta biblioteca permite ejecutar los algoritmos de forma secuencial, distribuida o en dispositivos GPU. Además, proporciona dos mecánicas de programación diferenciadas: simbólica e imperativa. Por un lado, la programación simbólica permite diseñar un modelo de forma rápida sin necesidad de aportar los datos de entrada *a priori* ni ejecutar los algoritmos de forma inmediata. Esto permite una programación más flexible, pudiendo acceder a los parámetros y modificarlos de forma desconectada de los cálculos costosos. Por otro lado, la programación imperativa facilita el control sobre los procesos de aprendizaje y la forma en que los datos se propagan por una red neuronal.

8.2.2 Instalación

Para disponer de MXNet en un ordenador y poder usarla desde R, es necesario compilar e instalar tanto la biblioteca como el paquete compañero para R. Las dependencias son las herramientas de compilación básicas (C++, GNU Make) y una implementación de la biblioteca

de álgebra lineal BLAS, como OpenBLAS o ATLAS. Opcionalmente se puede instalar OpenCV para facilitar el tratamiento de imágenes. Tras esto, se ejecutan los siguientes comandos para descargar y compilar el software:

```

1 git clone --recursive https://github.com/dmlc/mxnet
2 cd mxnet
3 cp make/config.mk . # editar las entradas necesarias
4 make -j $(nproc)
5 sudo install -D lib/libmxnet.so /usr/lib/libmxnet.so

```

Si se desea soporte para cómputo sobre GPU con CUDA, se añadirán las opciones USE_CUDA=1 USE_CUDA_PATH=/opt/cuda USE_CUDNN=1 al archivo config.mk, utilizando el camino conveniente para la biblioteca CUDA.

Alternativamente, en sistemas basados en Arch Linux basta con instalar el paquete mxnet⁵ del repositorio de usuarios AUR.

Por último, para instalar el paquete compañero para R, se utilizan las siguientes órdenes de línea de comandos desde el directorio donde se ha clonado el repositorio:

```

1 make rpkg
2 R CMD INSTALL mxnet_current_r.tar.gz

```

8.2.3 Uso de la biblioteca

Para construir una estructura de aprendizaje profunda con MXNet, basta con comenzar con un símbolo que corresponderá a los datos de entrada, después especificar las capas que formarán la red y, por último, el tipo de salida deseada.

Después, para ajustar el modelo creado con un conjunto de entrenamiento, MXNet proporciona algunas utilidades de alto nivel y otras más cercanas al cómputo paso a paso de las propagaciones hacia adelante y hacia atrás. De las primeras podemos destacar mx.model.FeedForward.create y de las últimas mx.exec.backward y mx.exec.forward.

Ejemplo 8.2. En este ejemplo vamos a construir una red prealimentada sencilla que permitirá aproximar el cálculo de la hipotenusa de un triángulo rectángulo a partir de las longitudes de los catetos.

Primero, comenzamos construyendo la red. Creamos la variable simbólica que contendrá los datos y la enlazamos con dos capas ocultas de 2 y 10 unidades respectivamente, y con la capa de salida que aproximará la hipotenusa, evaluará la pérdida y permitirá realizar el aprendizaje.

```
1 library(mxnet)
```

⁵ <https://aur.archlinux.org/packages/mxnet/>

```

2 red <- mx.symbol.Variable("data")
3 red <- mx.symbol.FullyConnected(red, num_hidden = 2)
4 red <- mx.symbol.Activation(red, act_type = "relu")
5 red <- mx.symbol.FullyConnected(red, num_hidden = 10)
6 red <- mx.symbol.Activation(red, act_type = "relu")
7 red <- mx.symbol.FullyConnected(red, num_hidden = 1)
8 red <- mx.symbol.LinearRegressionOutput(red)

```

Ahora, creamos los datos de entrada: aleatoriamente escogemos catetos y después calculamos la hipotenusa en la variable `label`. Separamos en un conjunto para entrenamiento y otro para test. De forma similar, podríamos también proceder con una validación cruzada.

```

1 set.seed(42)
2 mx.set.seed(42)
3
4 input <- data.frame(
5   cat1 = round(runif(100, min = 1, max = 10)),
6   cat2 = round(runif(100, min = 1, max = 10)))
7 x = t(data.matrix(input))
8 label <- sqrt(input$cat1 ** 2 + input$cat2 ** 2)
9 train_x <- x[,1:74]
10 train_y <- label[1:74]
11 test_x <- x[,75:100]
12 test_y <- label[75:100]

```

Por último, entrenamos el modelo que creamos anteriormente, escogiendo los parámetros del proceso de aprendizaje, en particular el optimizador SGD explicado en la sección 7.3.1, y obtenemos las predicciones sobre el conjunto de test.

```

1 model <- mx.model.FeedForward.create(
2   symbol = red,
3   X = train_x,
4   y = train_y,
5   num.round = 240,
6   array.layout = "colmajor",
7   optimizer = "sgd",
8   learning.rate = 0.015,
9   momentum = 0.2,
10  eval.metric = mx.metric.rmse
11 )
12 # => Train-rmse=0.693727073714297
13 predict(model, test_x)

```

8.3 INTRODUCCIÓN A RUTA

Se ha desarrollado un paquete software, denominado Ruta, con el objetivo de proporcionar acceso a diversas estructuras de aprendizaje profundo no supervisado de forma muy sencilla. El paquete se ha escrito en el lenguaje R utilizando los recursos de la biblioteca MXNet.

8.3.1 Desarrollo y metodologías

Al comienzo del desarrollo del software, se han analizado los objetivos perseguidos y se han escogido las tecnologías que nos facilitarían su consecución. En concreto, se ha elegido el lenguaje R por varias razones. Por un lado apenas dispone de herramientas de Deep Learning cómodas y consolidadas, luego este nuevo software no es redundante con nada existente. Además, R es un lenguaje que aporta muchas facilidades para la visualización de datos, lo cual es beneficioso ya que parte del software desarrollado se dedica a eso. También se eligió la biblioteca MXNet tras comparar con varias de las competidoras en el mercado en ese momento: Tensorflow se descartó por ser relativamente más lenta que el resto, y Theano y Caffe no disponían de API para R, mientras que otras librerías no estaban tan desarrolladas y no aportaban la funcionalidad que necesitábamos.

La metodología de desarrollo ha sido de tipo ágil, es decir, se ha desarrollado partiendo de un prototipo funcional, alterando y aumentando sus funcionalidades según se ha ido requiriendo. La documentación⁶ es exhaustiva en el sentido de que cubre toda la funcionalidad disponible para el usuario, pero no es excesiva. Además, progresivamente se ha ido adaptando el desarrollo según las nuevas necesidades o los obstáculos encontrados.

Un prototipo preliminar de esta herramienta, con funcionalidades básicas de entrenamiento de autoencoders y visualizaciones sencillas, se implementó bajo el nombre *dvisR* y se presentó en el congreso nacional CAEPIA 2016 [11].

Puesto que se trata de un software científico que se sostiene sobre varias piezas de software libre, se ha decidido que también esta herramienta será de código abierto y libre. Para asegurar que se mantiene libre, además, se ha utilizado la licencia *GNU General Public License* (GPL) 3.0, que requiere que las modificaciones que se realicen sobre el código del software se liberen bajo la misma licencia⁷. Así, el código del software está publicado en el repositorio fdavidcl/ruta de GitHub⁸.

Para asistir a la comprobación del software completo durante su desarrollo, se ha utilizado un sistema de integración continua, es decir, automatización de *builds* y de tests. Este sistema realiza, a cada avance publicado en GitHub, una comprobación mediante el programa R `CMD check`.

Además, se ha desarrollado un sencillo sitio web disponible en la dirección <http://ruta.software/> que da acceso a la descarga del

⁶ Se puede consultar desde R mediante `?` delante del nombre de la función. Por ejemplo, `? ruta.makeTask`.

⁷ Texto de la licencia disponible en <https://www.gnu.org/licenses/gpl.html>.

⁸ <https://github.com/fdavidcl/ruta/>

software y a la documentación. Se ha diseñado de manera *responsive*, con el objetivo de que la página se adapte automáticamente al tamaño de la pantalla de cada dispositivo. Se puede comprobar el diseño en la [Figura 8.1](#).

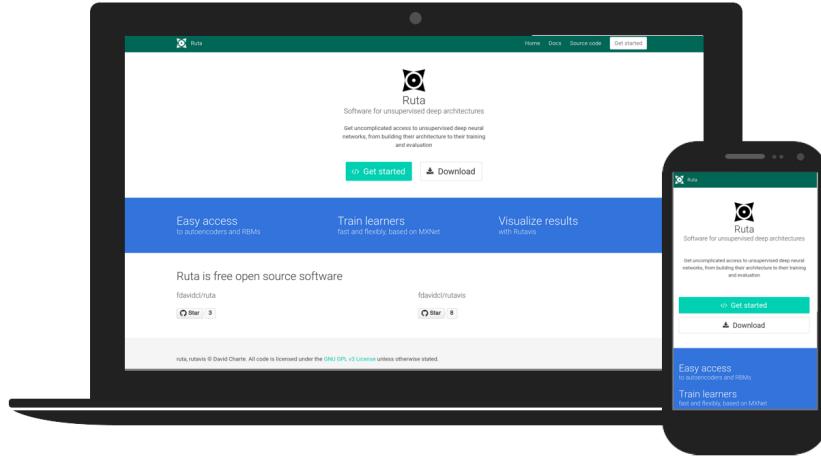


Figura 8.1: El sitio web del proyecto Ruta en distintos dispositivos

8.3.2 Instalación

Para instalar la última versión estable de Ruta y el paquete compañero Rutavis, basta con utilizar el conocido paquete `devtools` [52] para realizar automáticamente la descarga y la instalación. Desde la consola de R ejecutamos:

```
devtools::install_github("fdavidcl/ruta")
devtools::install_github("fdavidcl/rutavis")
```

Una vez instalado, para cargarlos utilizamos las órdenes siguientes:

```
library(ruta)
library(rutavis)
```

8.4 COMPONENTES DEL PAQUETE Y USO

8.4.1 Estructura

El software se estructura como un paquete convencional de R, siguiendo la jerarquía de directorios siguiente:

- `man/`
 - archivos de documentación en formato .Rd.
- `R/`

- `autoencoder_learner.R`: implementa la generación de redes profundas tipo autoencoder.
- `autoencoder_model.R`: incluye el entrenamiento de autoencoders y las herramientas de análisis de modelos.
- `classes.R`: incluye las clases utilizadas a lo largo del paquete.
- `learner.R`: incluye las funcionalidades comunes a los algoritmos de aprendizaje.
- `model.R`: implementa funcionalidades comunes a los modelos.
- `rbm_learner.R`: implementa la generación de máquinas de Boltzmann restringidas.
- `rbm_model.R`: incluye el entrenamiento de RBMs.
- `task.R`: contiene las funcionalidades comunes a las tareas.
- `unsupervised_task.R`: implementa la gestión de tareas no supervisadas.
- `util.R`: incluye utilidades adicionales.
- **DESCRIPTION**: indica metadatos del paquete, como nombre, versión y dependencias.
- **LICENSE**: contiene el texto de la licencia libre, en este caso GPL 3.0.
- **NAMESPACE**: da información sobre los nombres de funciones exportadas e importadas.

8.4.2 Funcionalidad

Las funcionalidades del paquete se han dividido en tres categorías:

- Tareas
- Algoritmos de aprendizaje
- Modelos entrenados

Aquí se ha utilizado como guía la arquitectura del paquete `mlr` [7] dirigido a aprendizaje automático en general.

Por un lado, las tareas representan conjuntos de datos de los que se desea aprender un modelo. Para ello, se construirán representaciones de los algoritmos de aprendizaje, ajustadas con diversos parámetros, que después se podrán entrenar y generar un modelo. Dicho modelo se podrá estudiar para obtener información sobre nuevos datos o sobre los aprendidos.

8.4.3 Tareas

Una tarea de aprendizaje se compone de un conjunto de datos y varios metadatos que aportan información acerca del mismo. En Ruta se pueden crear tareas genéricas con la función `ruta.makeTask` y tareas de aprendizaje no supervisado con `ruta.makeUnsupervisedTask`.

Ejemplo 8.3. Construyamos una tarea de aprendizaje no supervisado a partir del conocido conjunto de datos Iris [19]. Para ello, será necesario cargar el conjunto y posteriormente generaremos una tarea de Ruta, indicando la columna en la que se encuentra la clase, y podremos obtener información sobre ella:

```

1 data(iris)
2 task <- ruta.makeUnsupervisedTask("iris", data = iris, cl = 5)
3 print(task)
4 # ruta Task: iris
5 # Type: unsupervised
6 # Instances: 150
7 # Features: 5
8 # Has class: Yes (5)
```

8.4.4 Algoritmos de aprendizaje

En el momento actual del desarrollo, Ruta cuenta con una implementación de autoencoders basada en la biblioteca MXNet y una implementación básica de RBMs. Para utilizarlas, se dispone de la función `ruta.makeLearner` que da acceso a las técnicas de aprendizaje implementadas. Además, para generar modelos a partir de algoritmos y tareas, basta con utilizar el método `train` que se ha especializado para los objetos de clase “`rutaAutoencoder`”.

Ejemplo 8.4. Utilizamos la función mencionada para crear un objeto que represente un autoencoder de 5 capas de 4, 10, 2, 10 y 4 unidades respectivamente, con activaciones ReLU.

Lo entrenamos con la tarea que creamos anteriormente, seleccionando el optimizador Adam ([algoritmo 7](#)) con una tasa de aprendizaje de 0,005:

```

1 ae <- ruta.makeLearner("autoencoder",
2                         hidden = c(4, 10, 2, 10, 4),
3                         activation = "relu")
4 print(ae)
5 # ruta Learner
6 # Type: Autoencoder
7 # Backend: mxnet
8 # Sparse: No
9
10 model <- train(ae, task,
11                  epochs = 500,
```

```

12     learning.rate = 0.005,
13     optimizer = "adam")

```

Destacamos algunos parámetros interesantes en el ejemplo anterior. Primero, `activation`, que indica la función de activación que llevarán las unidades de la red. Las funciones de activación disponibles son:

- Ninguna (unidad lineal): `NULL`
- ReLU: “`relu`”
- Función logística: “`sigmoid`”
- Función `softplus`: “`softrelu`”
- Función tangente hiperbólica: “`tanh`”
- Tipo *leaky* ReLU: “`leaky`”, “`elu`”, “`prelu`”, “`rrelu`”

Por otra parte, el parámetro `optimizer` permite escoger el algoritmo de optimización con el que se entrenará. Los optimizadores disponibles son:

- Gradiente descendiente estocástico ([algoritmo 3](#)): “`sgd`”
- Adagrad ([algoritmo 5](#)): “`adagrad`”
- RMSProp ([algoritmo 6](#)): “`rmsprop`”
- Adam ([algoritmo 7](#)): “`adam`”

Cada uno de ellos acepta además los parámetros adicionales propios del algoritmo correspondiente.

8.4.5 Modelos entrenados

Una vez se ha obtenido un modelo entrenado, se pueden realizar distintas operaciones sobre él. Una de las más interesantes es obtener las codificaciones para unos datos dados, es decir, las salidas de la capa interna del autoencoder. Para ello contamos con la función `ruta.deepFeatures`. Adicionalmente, la función `ruta.layerOutputs` permite extraer las salidas de cualquier capa de la red. Por último, la implementación del método `predict` para objetos de clase “`rutaModel`” facilita las salidas de la última capa del autoencoder.

Ejemplo 8.5. Ahora tomamos el modelo aprendido en el [ejemplo 8.4](#) y obtenemos, para los mismos datos con los que se entrenó, las codificaciones en la capa interna:

```

1 deepf <- ruta.deepFeatures(model, task)
2 # Extracting layer aelayer3act_output (output #13)
3 # Setting arguments up to aelayer3_bias (arg #6)

```

Para conseguir extraer estas características internas, ha sido necesaria la implementación de una funcionalidad ausente en la API de MXNet para R: la predicción de capas internas de una red. Dicha implementación se encuentra también incluida en el paquete como la

función `predictPartial`. Esta función trabaja a bajo nivel, controlando la propagación de datos a través de cada capa.

8.5 VISUALIZACIÓN CON RUTAVIS

Una herramienta complementaria a `ruta` es `rutavis`, una aplicación con interfaz de usuario web que permite componer visualizaciones de forma interactiva y compararlas.

8.5.1 Estructura

De nuevo, este software se estructura como un paquete convencional de R, siguiendo una jerarquía similar a `Ruta` pero con algunas diferencias:

- `inst/`
 - `shiny/`
 - * `www/`
 - archivos adicionales para la interfaz web (CSS y JavaScript).
 - * `template.html`: plantilla HTML que compone la web.
 - * `server.R`: script que ejecuta el servidor web.
 - * `ui.R`: script que compone la interfaz del cliente mediante la plantilla.
 - archivos de documentación en formato `.Rd`.
- `R/`
 - `gui.R`: implementa el lanzamiento del servidor web.
 - `plots.R`: implementa distintos tipos de gráficos para representar modelos.
- `DESCRIPTION`: indica metadatos del paquete, como nombre, versión y dependencias.
- `LICENSE`: contiene el texto de la licencia libre, en este caso GPL 3.0.
- `NAMESPACE`: da información sobre los nombres de funciones exportadas e importadas.

8.5.2 Funcionalidad

`Rutavis` es un paquete totalmente dependiente de `Ruta`, centrado exclusivamente en la generación de gráficos a partir de modelos entrenados. Cumple dos responsabilidades principales: por una parte, la

gestión de visualizaciones que ayuden a entender el comportamiento de las técnicas no supervisadas y de los modelos que entran; por otra, una interfaz web que facilita al usuario la creación de dichas visualizaciones.

Los gráficos generados se muestran mediante la herramienta de visualización Plotly [29], que tiene una amplia gama de gráficas y una extensa documentación. Cuenta también con un paquete homónimo para R.

Además, para la creación del servidor web y la aplicación web desde R, se ha aprovechado el paquete Shiny [10], que proporciona diversas utilidades para el intercambio de datos entre el cliente y el servidor. Se ha utilizado el framework CSS Bulma⁹ para dar estructura y estilo a los elementos de la página. La interactividad con el usuario se ha programado en JavaScript en el cliente, que envía los datos al servidor y gestiona las respuestas.

8.5.3 Uso de la aplicación

Con Rutavis se pueden crear gráficos sin necesidad de hacer uso de la interfaz web, mediante la implementación de la función genérica `plot` para objetos de clase “`rutaModel`”.

Ejemplo 8.6. Con una tarea y un modelo como los utilizados anteriormente para el conjunto de datos iris, vamos a generar un gráfico que muestre las salidas codificadas mediante la capa intermedia del autoencoder. Utilizamos el método `plot.rutaModel` y le pasamos las mismas instancias con las que se entrenó:

```
plot(model, task)
```

La salida será un gráfico del estilo del de la Figura 8.2. Como se puede observar, el autoencoder consigue comprimir gran parte de la información de las 4 variables de Iris en solamente 2. Pese a no haber utilizado información sobre la clase, la representación bidimensional del conjunto de datos mantiene el bajo solapamiento entre clases.

Para lanzar la interfaz de usuario web, simplemente hay que ejecutar la siguiente llamada:

```
rutavis::ruta.gui()
```

Se abrirá una pestaña de navegador con la interfaz de usuario, inicialmente en la pantalla mostrada en la Figura 8.3, que nos permitirá crear una nueva visualización. Se pueden gestionar simultáneamente varias visualizaciones, de forma que puedan estudiarse una a una y modificar sus parámetros, ver comparativamente en formato mosaico, o listar en estilo *notebook* junto a los parámetros establecidos.

⁹ <http://bulma.io/>

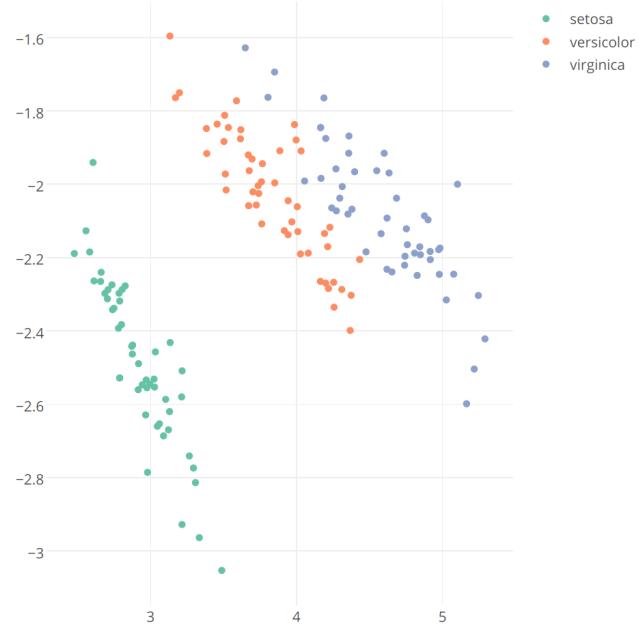


Figura 8.2: Gráfico de la codificación de los datos de entrada mediante la capa interna del autoencoder

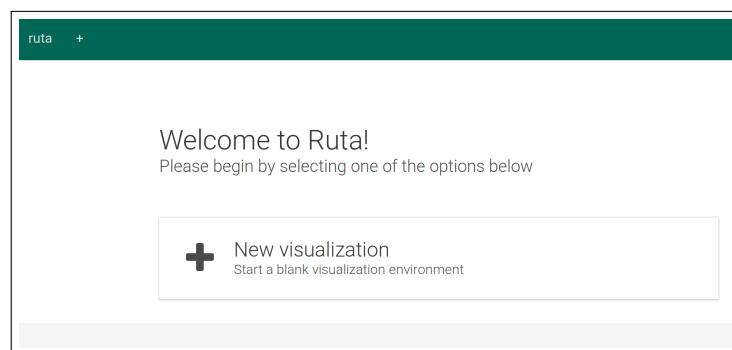


Figura 8.3: Pantalla de bienvenida de Rutavis

Vamos a realizar paso a paso un entrenamiento de dos modelos basados en autoencoders para el conjunto de datos WDBC (*Wisconsin Diagnostic Breast Cancer*) obtenido de [36]. El conjunto representa un problema de clasificación binaria, donde a partir de características médicas se pretende predecir si un tumor es benigno o maligno.

Comenzamos lanzando la interfaz web y seleccionando la opción *New visualization*. Se abrirá una vista compuesta por un panel de ajustes y un espacio para los gráficos. Cargamos el conjunto de datos mediante la opción *Upload new dataset* del panel. Seleccionamos la característica que corresponde con la clase de la lista *Class attribute*.

Para configurar un modelo de aprendizaje, escogemos los parámetros deseados. Como se puede observar en el ejemplo de la [Figura 8.4](#), utilizamos capas con unidades de tipo *leaky ReLU*, una codificación en 3 variables, un entrenamiento con AdaGrad de 200 épocas ajustando la tasa de aprendizaje a 0,02. En la [Figura 8.5](#) se muestra la vista de visualización única con el gráfico resultante, que corresponde a la salida de la capa interna del autoencoder respecto a los datos de entrada (los puntos verdes corresponden a tumores benignos y los azules a malignos). Se trata de un gráfico interactivo, sobre el que podemos arrastrar el ratón para mover los ejes, acercar o alejar la vista, y guardar en formato PNG.

Añadimos un segundo entorno de visualización mediante el botón + de la barra superior de pestañas. Entrenamos un modelo similar, en este caso con unidades con función de activación tangente hiperbólica y distintos parámetros de épocas y tasa de aprendizaje. Seleccionamos la vista de mosaico para comparar ambos resultados, como se muestra en la [Figura 8.6](#). La vista estilo *notebook* es similar a esta, pero apilando verticalmente todas las visualizaciones añadidas.

DATA

Current dataset
wdbc.arff

Upload new dataset

CLASS ATTRIBUTE

Class attribute
1: class

LEARNER AND TRAINING

Type
autoencoder

Activation
leaky

Hidden
3

Rounds
200

Optimizer
adagrad

Learning rate
0.02

VISUALIZATION

Type
internal

This figure shows a screenshot of the RUTAVIS software's parameter adjustment interface. The interface is organized into several sections: DATA, CLASS ATTRIBUTE, LEARNER AND TRAINING, and VISUALIZATION. In the DATA section, the current dataset is set to 'wdbc.arff' and there is a button to 'Upload new dataset'. Under CLASS ATTRIBUTE, the class attribute is set to '1: class'. The LEARNER AND TRAINING section contains settings for Type (set to 'autoencoder'), Activation (set to 'leaky'), Hidden (set to 3), Rounds (set to 200), Optimizer (set to 'adagrad'), and Learning rate (set to 0.02). The VISUALIZATION section has a single setting for Type, which is currently set to 'internal'. Each setting includes a dropdown menu with additional options.

Figura 8.4: El panel de ajuste de parámetros en detalle

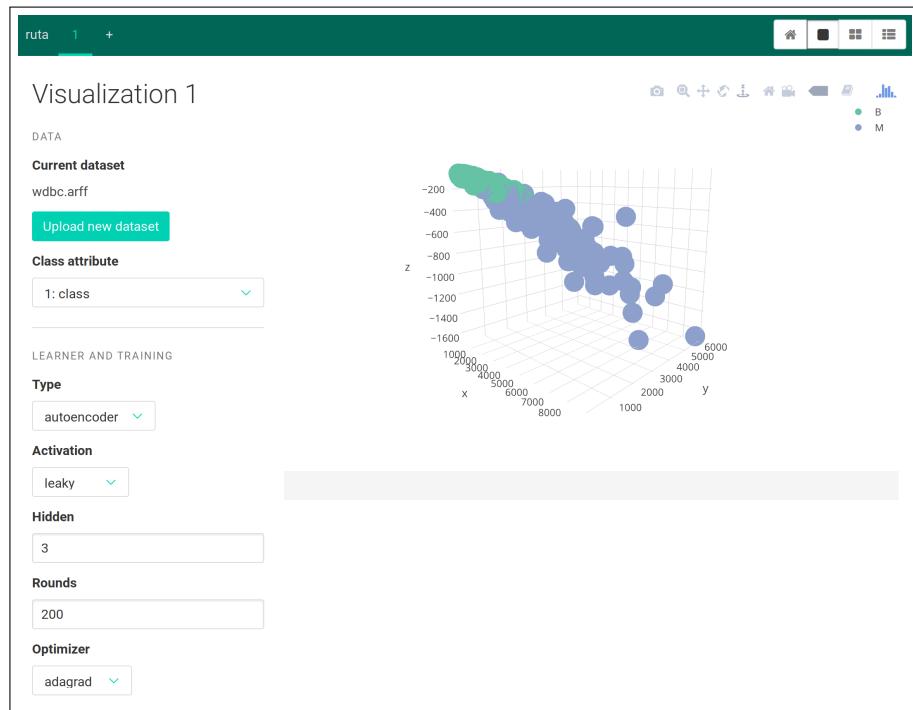


Figura 8.5: Interfaz gráfica de usuario de Rutavis. Arriba, la barra de pestanas y modos de visualización. A la izquierda, ajuste de parámetros. En el centro, visualización de las gráficas generadas

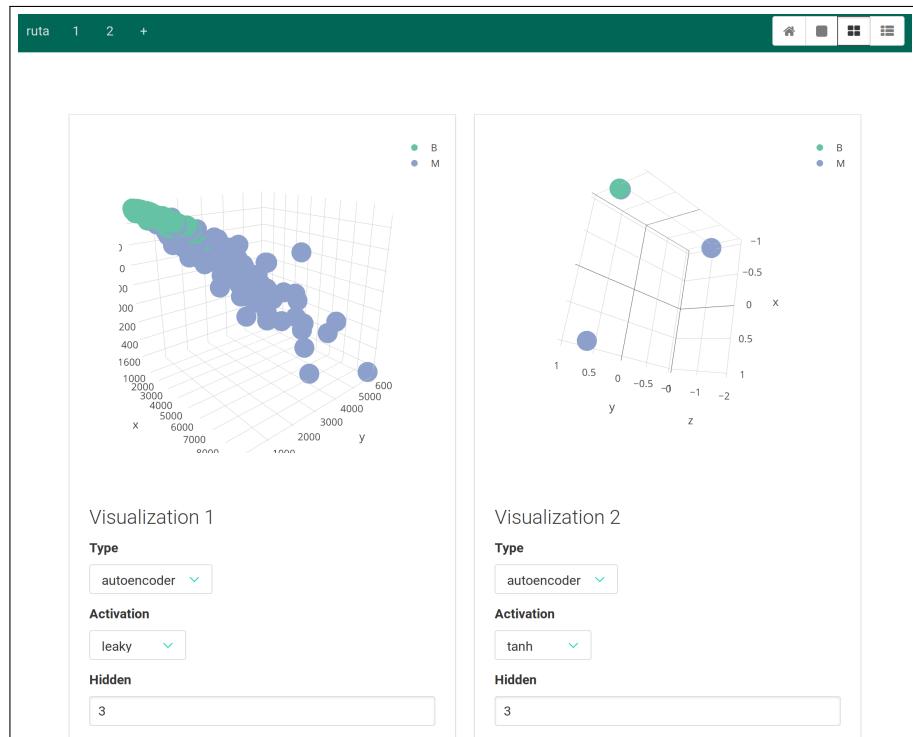


Figura 8.6: Comparación de dos visualizaciones en el modo mosaico de Rutavis

Parte IV

CONCLUSIONES

CONCLUSIONES Y VÍAS FUTURAS

En este trabajo hemos estudiado los fundamentos teóricos de las técnicas de Deep Learning que se aplican al problema de reducción de la dimensionalidad. Desde conceptos básicos de probabilidad a muy abstractos de álgebra, observamos que tiene raíces en áreas diversas de las matemáticas. Los algoritmos que permiten realizar el aprendizaje también son diferentes entre sí y pueden recurrir a su vez a otras teorías, como el análisis matemático.

En la parte práctica, se ha desarrollado una herramienta software novedosa que proporciona fácil acceso a estas técnicas e incorpora una interfaz de usuario y visualizaciones gráficas, apoyándonos en una biblioteca eficiente de cálculo para redes neuronales.

Los objetivos iniciales se han cumplido en buena medida, pero el estudio realizado demuestra que aún hay espacio para seguir trabajando en la implementación y la ampliación del software. Algunas vías de trabajo futuras que se pueden plantear son las siguientes:

- Incorporar más variantes de los autoencoders al paquete. Ya es posible entrenar autoencoders infracompletos y supercompletos, y sería deseable añadir autoencoders dispersos y contractivos.
- Permitir la elección de otra biblioteca base distinta de MXNet. Puesto que Ruta abstrae el proceso de construcción y entrenamiento de redes, podría mostrar la misma interfaz de programación aun usando diferentes *backends*.
- Aplicar nuevas técnicas de visualización para puntos en más de tres dimensiones, incluso en espacios de alta dimensionalidad. Por ejemplo, utilizar las componentes de los puntos como coeficientes de series de Fourier, y representar gráficamente las funciones obtenidas en el intervalo $[-\pi, \pi]$ [4].
- Introducir en el software de visualización acceso a otras herramientas de reducción de la dimensionalidad no basadas en redes neuronales, para permitir comparaciones. Entre los métodos deseables a incorporar estarían t-SNE [37] y LLE [46].

Parte V

APÉNDICE



MANUAL DE USUARIO DE RUTA

A.1 DOCUMENTACIÓN PARA EL PAQUETE RUTA EN LA VERSIÓN 0.2.0

A.1.1 *Predecir salidas para modelos entrenados y nuevos datos*

DESCRIPCIÓN Se deben entregar al menos un argumento de modelo y uno de tarea. La tarea puede contener los mismos datos de entrada que se utilizaron para entrenar el modelo, o datos nuevos con la misma estructura. El resto de argumentos serán facilitados a la función interna de predicción.

USO

```
1 ## S3 method for class 'rutaModel'  
2 predict(object, ...)
```

ARGUMENTOS

- *object* Un objeto “rutaModel”.
- ... Parámetros específicos para la función de predicción interna.

VALOR Una matriz conteniendo predicciones para cada instancia de la tarea dada.

A.1.2 *Obtener las características internas de un modelo de autoencoder entrenado*

DESCRIPCIÓN Extrae las características de la capa de codificación de un autoencoder previamente entrenado.

USO

```
1 ruta.deepFeatures(model, task, ...)
```

ARGUMENTOS

- *model* Un objeto “rutaModel” de un modelo de autoencoder.
- *task* Un objeto “rutaTask”.
- ... Parámetros específicos para la función de predicción interna.

VALOR Una matriz conteniendo la codificación para cada instancia de la tarea dada.

A.1.3 *Obtener los pesos de cualquier capa de un modelo entrenado*

DESCRIPCIÓN Extrae los pesos de los parámetros fijados al entrenar un modelo.

USO

```
1 ruta.getWeights(model, layer)
```

ARGUMENTOS

- *model* Un objeto “rutaModel” de un modelo de autoencoder.
- *layer* Un entero indicando el índice de la capa de la que se extraerán los pesos.

VALOR Una matriz conteniendo los pesos de la capa dada.

A.1.4 *Obtener las salidas de cualquier capa de un modelo entrenado*

DESCRIPCIÓN Extrae las salidas de la capa indicada a partir de un modelo entrenado y datos nuevos

USO

```
1 ruta.layerOutputs(model, task, layerInput = 1, layerOutput, ...)
```

ARGUMENTOS

- *model* Un objeto “rutaModel” de un modelo de autoencoder o RBM.
- *task* Un objeto “rutaTask” conteniendo los datos que deben propagarse por la red.
- *layerInput* Un entero indicando el índice de la capa en la que se inyectarán los datos. En autoencoders se inyectan siempre en la primera capa.
- *layerOutput* Un entero indicando el índice de la capa que se pretende extraer.
- ... Parámetros específicos para la función de predicción interna.

VALOR Una matriz conteniendo salidas para cada instancia de la capa dada.

A.1.5 Representar un algoritmo de aprendizaje

DESCRIPCIÓN Crea un objeto que representa un algoritmo de aprendizaje.

USO

```
1 ruta.makeLearner(cl, id = cl, ...)
```

ARGUMENTOS

- *cl* Una cadena de caracteres indicando el tipo de algoritmo (“autoencoder” o “rbm”).
- *id* Una cadena de caracteres opcional indicando un nombre para el objeto.
- ... Parámetros adicionales para el objeto, específicos al algoritmo requerido.

VALOR Un objeto para el algoritmo de aprendizaje que almacena los parámetros provistos.

EJEMPLOS

```
1 ruta.makeLearner("rbm", hidden = 4, activation = "bin")
2 ruta.makeLearner("autoencoder", "ae1",
3   hidden = c(4, 2, 4),
4   activation = "relu")
```

A.1.6 Representar una tarea de aprendizaje

DESCRIPCIÓN Crea un objeto que representa una tarea de aprendizaje.

USO

```
1 ruta.makeTask(id, data, cl = NULL)
```

ARGUMENTOS

- *id* Una cadena de caracteres opcional indicando un nombre para el objeto.
- *data* Un conjunto de datos, será convertido a un *data.frame*.
- *cl* El índice de la columna correspondiente a la clase, o NULL si el conjunto no incluye clase.

VALOR Un objeto genérico para la tarea de aprendizaje que almacena los datos dados.

A.1.7 Representar una tarea de aprendizaje no supervisado

DESCRIPCIÓN Crea un objeto que representa una tarea de aprendizaje no supervisado.

USO

```
1 ruta.makeUnsupervisedTask(id, data, cl = NULL)
```

ARGUMENTOS

- *id* Una cadena de caracteres opcional indicando un nombre para el objeto.
- *data* Un conjunto de datos, será convertido a un *data.frame*.
- *cl* El índice de la columna correspondiente a la clase, o NULL si el conjunto no incluye clase.

VALOR Un objeto para la tarea de aprendizaje no supervisado que almacena los datos dados y los parámetros.

EJEMPLO

```
1 data(iris)
2 irisTask <- ruta.makeUnsupervisedTask(
3   "iris",
4   data = iris,
5   cl = 5)
```

A.1.8 Pre-entrenamiento de autoencoders

DESCRIPCIÓN Obtener los pesos iniciales para un autoencoder mediante una pila de RBMs.

USO

```
1 ruta.pretrain(x, task, epochs = 10, ...)
```

ARGUMENTOS

- *x* Un objeto “rutaAutoencoder”.
- *task* Un objeto “rutaTask”.
- *epochs* El número de épocas para el proceso de entrenamiento de cada RBM.

VALOR Una lista de matrices de pesos iniciales para facilitar como el argumento *initial.args* en una llamada a *train* con un objeto de clase “rutaAutoencoder”.

A.1.9 Entrenamiento de modelos

DESCRIPCIÓN Generar un modelo entrenado para un objeto asociado

USO

```
1 train(x, task, ...)
```

ARGUMENTOS

- *x* Un objeto representando un algoritmo de aprendizaje de clase “rutaAutoencoder” o “rutaRBM”.
- *task* Un objeto “rutaTask”.
- ... Resto de parámetros según tipo de algoritmo.

En el caso de un autoencoder:

- *epochs* El número de iteraciones sobre el algoritmo de optimización.
- *optimizer* El nombre del optimizador a usar (“sgd”, “adagrad”, “rmsprop”, “adam” o “adadelta”).
- *eval.metric* Medida de evaluación (por defecto, el error cuadrático medio).
- *initial.args* Lista de argumentos iniciales (pesos obtenidos por ruta.pretrain).
- *initializer.scale* Escala de la distribución uniforme para inicializar pesos.
- ... Parámetros adicionales para el optimizador.

En el caso de una RBM:

- *numepochs* El número de iteraciones para el entrenamiento.

VALOR Un objeto de clase “rutaModel” que contiene el modelo entrenado.

EJEMPLO

```
1 data(iris)
2 irisTask <- ruta.makeUnsupervisedTask(
3     "iris",
4     data = iris,
5     cl = 5)
6
7 ae <- ruta.makeLearner(
8     "autoencoder",
9     hidden = c(4, 2, 4),
10    activation = "leaky")
11 model <- train(
12     ae,
```

```

13     task,
14     epochs = 80,
15     optimizer = "adagrad",
16     learning.rate = 0.02,
17     initializer.scale = 2)

```

A.2 DOCUMENTACIÓN PARA EL PAQUETE RUTAVIS EN LA VERSIÓN 0.2.0

A.2.1 Representar la codificación de un modelo entrenado

DESCRIPCIÓN Obtener y representar la capa interna de un modelo entrenado con unos datos de entrada.

USO

```

1 ## S3 method for class 'rutaModel'
2 plot(model, task, ...)

```

ARGUMENTOS

- *model* Un objeto “*rutaModel*” entrenado mediante un autoencoder.
- *task* Un objeto “*rutaTask*” con los datos de entrada.
- ... Parámetros adicionales para la función de representación gráfica.

VALOR Genera un gráfico de Plotly y lo muestra en un panel interactivo.

A.2.2 Lanzar la interfaz de usuario web

DESCRIPCIÓN Carga la interfaz gráfica de usuario web y abre una nueva pestaña de navegador para mostrarla.

USO

```
1 ruta.gui()
```

VALOR No devuelve nada.

BIBLIOGRAFÍA

- [1] Martín Abadi y col. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. En: *arXiv preprint arXiv:1603.04467* (2016).
- [2] Charu C Aggarwal, Alexander Hinneburg y Daniel A Keim. “On the surprising behavior of distance metrics in high dimensional space”. En: *International Conference on Database Theory*. Springer. 2001, págs. 420-434.
- [3] Spencer Aiello y col. *h2o: R Interface for H2O*. R package version 3.8.1.3. 2016. URL: <https://CRAN.R-project.org/package=h2o>.
- [4] David F Andrews. “Plots of high-dimensional data”. En: *Biometrics* (1972), págs. 125-136.
- [5] Frédéric Bastien y col. “Theano: new features and speed improvements”. En: *arXiv preprint arXiv:1211.5590* (2012).
- [6] Kevin Beyer y col. “When is “nearest neighbor” meaningful?” En: *International conference on database theory*. Springer. 1999, págs. 217-235.
- [7] Bernd Bischl y col. “mlr: Machine Learning in R”. En: *Journal of Machine Learning Research* 17.170 (2016), págs. 1-5. URL: <http://jmlr.org/papers/v17/15-066.html>.
- [8] Michael R Brent. “From grammar to lexicon: unsupervised learning of lexical syntax”. En: *Computational Linguistics* 19.2 (1993), págs. 243-262.
- [9] Augustin Cauchy. “Méthode générale pour la résolution des systemes d'équations simultanées”. En: *Comp. Rend. Sci. Paris* 25.1847 (1847), págs. 536-538.
- [10] Winston Chang y col. *shiny: Web Application Framework for R*. R package version 0.13.2. 2016. URL: <https://cran.r-project.org/package=shiny>.
- [11] David Charte, Francisco Charte y Francisco Herrera. “Análisis visual de técnicas no supervisadas de deep learning con el paquete dlvisR”. En: *Actas de la XVII conferencia de la Asociación Española para la Inteligencia Artificial*. AEPIA. 2016, págs. 895-904.
- [12] Tianqi Chen y col. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. En: *arXiv preprint arXiv:1512.01274* (2015).
- [13] Robert Clarke y col. “The properties of high-dimensional data spaces: implications for exploring gene and protein expression data”. En: *Nature Reviews Cancer* 8.1 (2008), págs. 37-49.

- [14] William W Cohen y col. "Learning rules that classify e-mail". En: *AAAI spring symposium on machine learning in information access*. Vol. 18. California. 1996, pág. 25.
- [15] Thomas M Cover y Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
- [16] Martin Drees. "Implementierung und Analyse von tiefen Architekturen in R". Tesis de mtría. Fachhochschule Dortmund, 2013. URL: <https://cran.r-project.org/package=darch>.
- [17] John Duchi, Elad Hazan y Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". En: *Journal of Machine Learning Research* 12.Jul (2011), págs. 2121-2159.
- [18] Dumitru Erhan y col. "The difficulty of training deep architectures and the effect of unsupervised pre-training". En: *International Conference on artificial intelligence and statistics*. 2009, págs. 153-160.
- [19] Ronald A Fisher. "The use of multiple measurements in taxonomic problems". En: *Annals of eugenics* 7.2 (1936), págs. 179-188.
- [20] Imola K Fodor. "A survey of dimension reduction techniques". En: *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory* 9 (2002), págs. 1-18.
- [21] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [22] Isabelle Guyon y André Elisseeff. "An introduction to feature extraction". En: *Feature extraction*. Springer, 2006, págs. 1-25.
- [23] Geoffrey E Hinton. "Training products of experts by minimizing contrastive divergence". En: *Neural computation* 14.8 (2002), págs. 1771-1800.
- [24] Geoffrey E Hinton. "Learning multiple layers of representation". En: *Trends in cognitive sciences* 11.10 (2007), págs. 428-434.
- [25] Geoffrey E Hinton, Simon Osindero y Yee-Whye Teh. "A fast learning algorithm for deep belief nets". En: *Neural computation* 18.7 (2006), págs. 1527-1554.
- [26] Geoffrey E Hinton, Simon Osindero y Yee-Whye Teh. "A fast learning algorithm for deep belief nets". En: *Neural computation* 18.7 (2006), págs. 1527-1554.
- [27] Geoffrey E Hinton y Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". En: *science* 313.5786 (2006), págs. 504-507.
- [28] Thomas Hofmann. "Unsupervised learning by probabilistic latent semantic analysis". En: *Machine learning* 42.1-2 (2001), págs. 177-196.
- [29] Plotly Technologies Inc. *Collaborative data science*. 2015. URL: <https://plot.ly>.

- [30] Yangqing Jia y col. "Caffe: Convolutional architecture for fast feature embedding". En: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, págs. 675-678.
- [31] Tamara G Kolda y Brett W Bader. "Tensor decompositions and applications". En: *SIAM review* 51.3 (2009), págs. 455-500.
- [32] Igor Kononenko. "Machine learning for medical diagnosis: history, state of the art and perspective". En: *Artificial Intelligence in medicine* 23.1 (2001), págs. 89-109.
- [33] Sotiris B Kotsiantis, I Zaharakis y P Pintelas. *Supervised machine learning: A review of classification techniques*. 2007.
- [34] Yann LeCun. "The MNIST database of handwritten digits". En: (1998). URL: <http://yann.lecun.com/exdb/mnist/>.
- [35] Yann LeCun y col. "Backpropagation applied to handwritten zip code recognition". En: *Neural computation* 1.4 (1989), págs. 541-551.
- [36] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: <http://archive.ics.uci.edu/ml>.
- [37] Laurens van der Maaten y Geoffrey Hinton. "Visualizing data using t-SNE". En: *Journal of Machine Learning Research* 9.Nov (2008), págs. 2579-2605.
- [38] Tom M Mitchell y col. *Machine learning*. WCB. McGraw-Hill Boston, MA: 1997.
- [39] Luis Carlos Molina, Lluís Belanche y Àngela Nebot. "Feature selection algorithms: A survey and experimental evaluation". En: *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*. IEEE. 2002, págs. 306-313.
- [40] Leif E Peterson. "K-nearest neighbor". En: *Scholarpedia* 4.2 (2009), pág. 1883. URL: http://scholarpedia.org/article/K-nearest_neighbor.
- [41] Clifton Phua y col. "A comprehensive survey of data mining-based fraud detection research". En: *arXiv preprint arXiv:1009.6119* (2010).
- [42] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2017. URL: <https://www.R-project.org/>.
- [43] RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, Inc. Boston, MA, 2016. URL: <http://www.rstudio.com/>.
- [44] Marc Aurelio Ranzato y col. "Unsupervised learning of invariant feature hierarchies with applications to object recognition". En: *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE. 2007, págs. 1-8.

- [45] Xiao Rong. *deepnet: deep learning toolkit in R*. R package version 0.2. 2014. URL: <https://CRAN.R-project.org/package=deepnet>.
- [46] Sam T Roweis y Lawrence K Saul. “Nonlinear dimensionality reduction by locally linear embedding”. En: *science* 290.5500 (2000), págs. 2323-2326.
- [47] David E Rumelhart, Geoffrey E Hinton y Ronald J Williams. *Learning internal representations by error propagation*. Inf. téc. DTIC Document, 1985.
- [48] Shai Shalev-Shwartz y Shai Ben-David. *Understanding Machine Learning*. nil. Cambridge University Press, 2014, nil. URL: <http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning>.
- [49] Joshua B Tenenbaum, Vin De Silva y John C Langford. “A global geometric framework for nonlinear dimensionality reduction”. En: *science* 290.5500 (2000), págs. 2319-2323.
- [50] Tijmen Tieleman. “Training restricted Boltzmann machines using approximations to the likelihood gradient”. En: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, págs. 1064-1071.
- [51] Sergei Treil. *Linear algebra done wrong*. MTM, 2013.
- [52] Hadley Wickham y Winston Chang. *devtools: Tools to Make Developing R Packages Easier*. R package version 1.11.0.9000. URL: <https://github.com/hadley/devtools>.
- [53] David H Wolpert y William G Macready. “No free lunch theorems for optimization”. En: *IEEE transactions on evolutionary computation* 1.1 (1997), págs. 67-82.