Desarrollo de una aplicación Android, asistida por visión artificial, para administrar imágenes almacenadas en un dispositivo móvil

PROYECTO FINAL DE CARRERA

Yackel, Francisco

Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral

Diciembre 2018

Documento maquetado con TEXIS v.1.0+.

Desarrollo de una aplicación Android, asistida por visión artificial, para administrar imágenes almacenadas en un dispositivo móvil

Memoria que presenta para optar al título de Ingeniero en Informática

Yackel, Francisco

Dirigida por el Doctor

Vignolo, Leandro

Codirigida por el Ingeniero

Ferrado, Leandro

Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral

Diciembre 2018

I can't go to a restaurant and
order food because I keep looking
at the fonts on the menu.
Donald Knuth

Agradecimientos

Dame un punto de apoyo
y moveré el mundo.

Arquímedes

Quisiera agradecer profundamente a cada uno de los actores que fueron parte de mi paso por esta excelente facultad, tanto profesores como colegas estudiantes, por ofrecer su tiempo y conocimientos para formar en mí un futuro profesional, desde lo académico hasta como persona. Principalmente quiero destacar el trabajo del instituto *sinc(i)*, por la ayuda que cada uno de sus integrantes me brindó por varios años, fundamentalmente de mis directores quienes me iniciaron en este camino que hoy compone mi vocación laboral como *data scientist*.

También quiero dar las gracias a todos mis amigos que supieron estar conmigo durante esta etapa de mi vida en todos sus aspectos. Algunos de toda mi vida, otros de muchos años, y algunos otros que adquirí durante mis estudios y que acompañaron mi trayecto en la universidad. Todos ellos son una gran riqueza que afortunadamente conservé durante estos años y pretendo preservar a través de las siguientes etapas de mi vida.

Pero todo este apoyo que recibí en estos años no se compara con el brindado por mi hermosa familia. Agradezco de corazón a mi hermana Joana que en cada momento me transmitió con su ejemplo responsabilidad y dedicación para formar mi vocación; a mi padre Javier que supo inspirar en mí gran parte de la cultura que hoy asumo en mi personalidad; y a mi madre Cristina que con mucha ternura me ayudó a progresar y siempre me acompaña a perseguir mis sueños. Todos ellos supieron darme calidez y amor de familia que alimentaron mi crecimiento, y a ellos les debo cada uno de los pasos que me llevaron a la persona que soy.

Finalmente debo agradecer a Dios, quien siempre sabe sembrar bendiciones en mi camino para poder cumplir todas mis metas, y gracias a quien tengo el privilegio de concluir esta linda etapa de mi vida y acompañado del excelente entorno de personas que mencioné anteriormente.

Resumen

No basta tener un buen ingenio, lo principal es aplicarlo bien.

René Descartes

En la actualidad, el constante avance de las tecnologías de comunicación permite que la transmisión de información multimedia sea un aspecto común y diario para las personas que utilizan dispositivos móviles. El fenómeno de las redes sociales, trae consigo un tráfico de datos que muchas veces se hace difícil de administrar y controlar por los usuarios, produciendo que se alcance el límite de almacenamiento en los dispositivos, principalmente en aquellos de gama baja o media, aunque ocasionalmente sucede también en los últimos modelos. Por otro lado, cuando se quieren administrar los archivos de un dispositivo móvil, o bien al realizar una copia de seguridad, las tareas llevadas a cabo para organizar las imágenes consumen mucho tiempo.

Gracias a los grandes avances que hoy en día provee la visión artificial, una disciplina que combina técnicas de procesamiento de imágenes y aprendizaje maquinal, se considera posible lograr un sistema que cuente con las propiedades adecuadas para lograr un agrupamiento o *clustering* de datos , basándose en una similitud computada sobre lo mismos. Con ello, se podría sugerir al usuario una estructura de directorios para organizar las imágenes de su dispositivo, y a partir de ahí proveer funcionalidades para agilizar y automatizar las tareas de administración involucradas, reduciendo el tiempo y esfuerzo requerido.

Por lo tanto, la propuesta de este proyecto es implementar una aplicación que agrupe, de manera automática, las imágenes de un directorio seleccionado por el usuario. Luego, una vez realizado el procesamiento, éste podrá tomar la decisión que desee interactuando con el resultado (e.g eliminar un grupo obtenido, fusionarlo con otros, renombrar carpetas, ingresar en un grupo y eliminar alguna imagen en particular, etc.). Dicha aplicación será implementada para la plataforma Android, dado que es un sistema operativo muy popular y además gratuito.

Palabras claves: imágenes, agrupamiento, administración y control, similitud computada.

Índice

Índice de figuras

Índice de Tablas

Parte I

Conceptos básicos

En esta primera parte del documento se presentan los conceptos manejados en este trabajo para definir el tratamiento realizado. Esto comprende los elementos que componen la especificación del proyecto y los fundamentos que sustentan las soluciones propuestas para alcanzar los objetivos planteados. Se realiza una breve aclaración de todos estos recursos, introducidos en un orden conveniente, para dar contexto a cómo se propone encarar el proyecto y cuál es la base identificada para poder realizar ello con éxito.

Capítulo 1

Introducción

Es increíble la cantidad de energía que se tiene en cuanto hay un plan.

Daniel Handler

Resumen: En este capítulo se detallan los elementos que dan origen a este proyecto y componen la especificación del mismo. Se hace una reseña de la motivación del proyecto y cómo se justifica el trabajo realizado, aclarando los objetivos planteados y las restricciones tenidas en cuenta para el alcance y las tecnologías utilizadas.

Motivación

La Ingeniería en Informática abarca muchas y diferentes áreas de estudio, por lo que existen diversas opciones a la hora de decidir que temática abordar para realizar el proyecto final de carrera. A lo largo de años de cursado, trabajos prácticos y evaluaciones, los estudiantes transitan por diferentes materias que ayudan a marcar el perfil del futuro profesional, según los campos de interés o preferencias. Por dar algunos ejemplos, se incorporan conocimientos en programación y desarrollo en diferentes paradigmas, Ingeniería del Software, Algorítmos y Estructuras de Datos, Redes y Comunicaciones, Mecánica Computacional, Procesamiento digital de señales e imágenes, Inteligencia Computacional, etc.

El auge de las aplicaciones para dispositivos móviles incorporando procesamiento de datos, utilizando inteligencia artificial, ha influído en gran medida en la decisión de desarrollar una aplicación utilizando visión computacional y aprendizaje maquinal. La evolución de los teléfonos inteligentes y su uso masivo, hicieron que empresas de desarrollo de software se acoplen a esta demanda, ofrenciendo productos y servicios para que todas las personas puedan utilizarlos en sus propios dispositívos. El área que comprende el procesamiento de imágenes y el aprendizaje maquinal no escapa a esta ola tecnológica, por lo que bibliotecas que permiten el procesamiento de imágenes empezaron a estar disponibles para su uso y adaptabilidad. A continuación, se listan algunas aplicaciones destacadas que utilizan procesamiento de imágenes e inteligencia computacional:

- Google entrenó una red neuronal profunda que logró reconocer con buena precisión tanto rostros y cuerpos humanos como también gatos en videos de YouTube, lo cual se obtuvo de forma no supervisada sin utilizar información de clases (es decir, nunca se le daba información de lo que estaba aprendiendo) [**?**].
- El sistema DeepFace de Facebook alcanzó una precisión de 97.35 % con redes neuronales profundas sobre el conjunto de datos *Labeled Faces in the Wild* (LFW), reduciendo el error del actual estado del arte en más de un 27 % y acercándose a la precisión de un humano en reconocimiento de rostros (que es de aproximadamente un 97.53 %) [**?**].
- Google, a través de su producto *Google Photos*, ofrece una solución denominada *Organize*, la cual realiza tareas de organización sobre las imágenes de sus usuarios. Ésta proporciona distintos conjuntos de imágenes basándose en características u objetos que tengan en común, como pueden ser perros, autos, el cielo, etc[**?**].
- La aplicación fotos, perteneciente al sistema operativo Windows 10, permite buscar texto en las imágenes. La misma utiliza una tecnología denominada OCR (reconocimiento óptico de caracteres), la cual permite detectar el lugar y la palabra misma, dentro de una imágen.

Justificación

Lo que se busca en este proyecto es implementar una aplicación que asista al usuario en la tarea de organizar las imágenes de su dispositivo. Como etapa inicial, se obtiene una estructura de directorios mediante una metodología basada en algoritmos de visión computacional, formando grupos de imágenes según el grado de similitud que comparten. Luego, una vez realizado el procesamiento, el usuario podrá tomar la decisión que desee interactuando con el resultado (e.g eliminar un grupo obtenido, renombrar carpetas, ingresar en un grupo y eliminar alguna imagen en particular, etc.).

La aplicación es Open Source, con el fin de poder agregar valor a la comunidad del software libre y que pueda ser continuamente mejorada o versionada y personalizada en base a distintos desarrolladores que presenten interés por la misma. Se propone además que sea On Premise (i.e. el proceso se realiza únicamente en el dispositivo móvil) en lugar de ser un servicio Cloud. De esta manera se puede asegurar la confidencialidad de los datos que se manejan, al mismo tiempo lograr que el despliegue y mantenimiento del sistema sean aspectos más sencillos y económicos.

Por último, será implementada para la plataforma Android, dado que es un sistema operativo muy popular y gratuito. Esta decisión se basa en que, a nivel mundial, Android es el líder indiscutido por cantidad de usuarios, y lo mismo sucede en la Argentina, ya que se encuentra en el 85 % de los equipos, según datos del mercado [**?**]. Además, es destacable que se trata de una plataforma de código abierto que brinda a los desarrolladores un amplio conjunto de herramientas de desarrollo y librerías a su disposición, las cuales a su vez cuentan con extensa documentación de respaldo.

Objetivos

Para desarrollar este proyecto, se plantearon los siguientes objetivos:

Objetivos generales

- Desarrollar una aplicación móvil sobre la plataforma Android que, a partir de imágenes almacenadas en el dispositivo, sugiera una estructura de directorios para organizar el conjunto, según una medida de similitud calculada entre los elementos.
- Proveer al sistema funcionalidades que le permitan al usuario la interacción con el resultado, para desempeñar distintas tareas de administración sobre el almacenamiento comprendido por las imágenes?.

Objetivos específicos

- Analizar distintos algoritmos y métodos tanto para la extracción de características de las imágenes como para el clustering de las mismas.
- Definir la función o noción de similitud que se deba computar entre las imágenes a agrupar.
- Definir un conjunto de imágenes a utilizar en experimentos, que sea lo más fiel posible a la problemática tratada en el marco de dispositivos móviles.
- Desarrollar un protocolo de experimentación en base al conjunto de datos definido, para evaluar el desempeño de las técnicas investigadas.
- Llevar adelante las pruebas definidas en el protocolo de experimentación.
- Analizar y definir la metodología de visión artificial a implementar en el sistema, priorizando las técnicas según su desempeño en la experimentación así como la posibilidad de ajustarse a los recursos con los que cuentan la mayoría de los dispositivos.
- Lograr una interfaz de usuario sobre la cual podrá elegir el directorio/directorios a procesar, y que sea suficiente para la administración e interacción necesaria con el resultado del clustering.

Alcance

Se definen las siguientes cuestiones para el alcance de este proyecto:

- Se implementará el algoritmo de preprocesamiento y extracción de características de las imágenes, utilizando las ventajas y funcionalidades que provee la biblioteca TensorFlow para Android.
- La aplicación será desarrollada sobre la plataforma Android, contemplando en lo posible aquellas versiones que se encuentren instaladas en la mayoría de los dispositivos móviles actuales.

- La aplicación no realizará las tareas de administración de forma autónoma. Ésta asistirá al usuario brindando una estructura de directorios en la que las imágenes serán ordenadas por grado de similitud o contexto, para luego facilitar opciones de administración sobre los mismos.

Tecnologías utilizadas

Restricciones

Se realizó la documentación de la Especificación de Requisitos del Software (ERS) siguiendo el estándar IEEE 830-1998 (ver Anexo), y en la misma se encuentra la especificación completa de las tecnologías utilizadas. No obstante, se listan a continuación:

- **Sistema operativo**: Android
- **Versiones**: Android KitKat (4.4) en adelante
- **Lenguaje de programación**: JAVA 1.8.0
- **Paradigmas de programación**:
  - Orientado a objetos
  - Funcional
  - Imperativo
- **Dependencias externas**:
  - TensorFlow Lite 0.1.7
  - Storage Chooser 2.0 ! [1]
  - Apache Commons[2]:
    - Maths 3.6.1
    - IO 2.4
  - Efficient Java Matrix Library[3] 0.33
  - Snatik Storage[4] 2.1.0
  - ZGallery[5] 0.3
  - NumberProgressBar[6] 1.4
- **Interfaz de usuario**: XML 1.1
- **Control de versiones**: Git v2.7.4

[1] `https://github.com/codekidX/storage-chooser`
[2] `http://commons.apache.org/`
[3] `http://ejml.org/wiki/index.php?title=Main_Page`
[4] `https://github.com/sromku/android-storage`
[5] `https://github.com/mzelzoghbi/ZGallery`
[6] `https://github.com/daimajia/NumberProgressBar`

Evolución previsible

Se prevén las siguientes cuestiones tecnológicas a mejorar o integrar en el futuro del producto:

- Elección semi-automática de híper-parámetros en el modelado de redes neuronales mediante computación evolutiva (e.g. algoritmos genéticos, BSO, etc.).
- Administración de trabajos o *scheduler* para automatizar la ejecución de experimentos en el modelado.
- Interfaz gráfica con mejor experiencia de usuario (por ejemplo, con CherryPy y/o Apache Zeppelin).
- Integración con tecnologías referidas a administración de clúster (como Apache Mesos y YARN).
- Integración con tecnologías referidas a almacenamiento de datos (como Apache Cassandra y HDFS).
- Adaptación a una imagen en Docker para despliegue del framework en entornos virtualizados.
- Incorporación de otras técnicas complementarias de aprendizaje maquinal (e.g. K-means, DBN, redes recurrentes, etc.).
- Mejora de los módulos que realizan cálculos algebraicos, integrando nuevas soluciones que paralelicen dicho cómputo (e.g. librerías eficientes a nivel CPU, módulos que realicen cómputos en GPU).

Capítulo 2

Fundamentos teóricos

Todas las teorías son legítimas y ninguna
tiene importancia. Lo que importa
es lo que se hace con ellas.

Jorge Luis Borges

Resumen: Este capítulo pretende refrescar conocimientos, e introducir otros, para entender las bases teóricas utilizadas en todo el trabajo realizado. Inicialmente, se exhibe la noción de aprendizaje maquinal en sistemas de regresión y clasificación, desde la forma supervisada hasta la no supervisada, y cómo las redes neuronales se utilizan para componer dichos sistemas. Finalmente se profundiza en aspectos del aprendizaje profundo, lo cual en conjunto con todos los fundamentos presentados abarcan las características implementadas para este trabajo.

En los últimos años, el "aprendizaje maquinal" (mejor conocido en inglés como *machine learning*) adquirió bastante popularidad, presentando algoritmos que aproximan en diversas tareas el concepto de inteligencia artificial [**?**]. Este campo estudia técnicas para construir sistemas capaces de aprender a partir de datos a realizar diversos tipos de tareas sin requerir que se les indique cómo hacerlas. Esto se emplea en una gran cantidad de aplicaciones en donde no es factible diseñar y programar de forma explícita algoritmos para realizar tareas complejas, como visión computacional, motores de búsqueda, reconocimiento de voz, predicción de fraudes, etc.

Los tipos de sistemas que se diseñan para realizar estas tareas mencionadas por lo general siguen dos tipos de aprendizaje: supervisado, y no supervisado [**?**]. A su vez, dichas tareas a realizar sobre datos se suelen clasificar típicamente en las siguientes categorías: a) clasificación, donde el sistema aprende de forma supervisada a asignar clases; b) regresión, aprendiendo supervisadamente a predecir una variable continua; c) agrupamiento o *clustering*, para dividir los datos en grupos pero de forma no supervisada; d) reducción de dimensiones, mapeando los datos de entrada en un espacio de menor dimensión. En la siguiente sección de este capítulo se detallan los contenidos referidos a la construcción de estos sistemas, para conocer cómo se implementan los algoritmos de aprendizaje maquinal a partir de un conjunto de datos.

Aprendizaje supervisado

A continuación se procede a detallar la implementación de un sistema con aprendizaje maquinal en forma supervisada para realizar tareas de regresión y clasificación, con el fin de hacer familiar el tratamiento de funciones objetivo, computando sus gradientes y optimizando los objetivos sobre un conjunto de parámetros. Estas herramientas básicas van a formar la base para los algoritmos implementados en el presente trabajo.

Sistemas de regresión y clasificación

En una *regresión lineal* el objetivo es predecir un valor deseado y partiendo de un vector de entrada $x \in \Re^n$ (que por lo general representa las "características" que describen el fenómeno analizado). Para ello, lo usual es contar con un conjunto de patrones o ejemplos donde cada uno de ellos tiene asociado un vector con características $x^{(i)}$ y su predicción correspondiente o "etiqueta" $y^{(i)}$, y con ello se busca modelar de forma supervisada (i.e. explicitando la salida deseada) una función $y = h(x)$ tal que $y^{(i)} \approx h(x^{(i)})$ para cada i-ésimo ejemplo de entrenamiento. Teniendo una cantidad suficiente de patrones, se espera que $h(x)$ sea un buen predictor incluso cuando se le presente un nuevo vector de características donde su correspondiente etiqueta no se conoce.

Para modelar la hipótesis $h(x)$, en cualquier tipo de sistema, se deben definir dos cuestiones: i) cómo se representa la hipótesis y ii) cómo se mide su error de aproximación respecto a la función deseada y. En el caso de la regresión lineal, se define: $h_\theta(x) = \sum_j \theta_j x_j$, donde $h_\theta(x)$ representa una gran familia de funciones parametrizadas por θ. Por lo tanto, para encontrar una elección de θ que aproxime $h_\theta(x^{(i)})$ lo mayor posible a $y^{(i)}$, se busca minimizar una función de "costo" o "penalización", la cual mide el error cometido en la predicción. Un ejemplo de esta función es utilizar como medida el *Error Cuadrático Medio* (o MSE, del inglés *Mean Squared Error*), la cual se define como:

$$L_i(\theta) = \frac{1}{2} \sum_j (h_\theta(x_j^{(i)}) - y_j^{(i)})^2 \tag{2.1}$$

El valor resultante de esta función corresponde al error de aproximación de $h_\theta(x^{(i)})$ para un i-ésimo ejemplo dado, y para conocer el error total sobre todo el conjunto de N ejemplos disponibles se promedian todos los errores cometidos tal que $L(\theta) = \frac{1}{N} \sum_i^N L_i(\theta)$. La elección de θ que minimiza esta función de costo se puede encontrar mediante algoritmos de optimización (e.g. gradiente descendiente) que, por lo general, requieren que se compute tanto $L_i(\theta)$ como su gradiente $\nabla_\theta L_i(\theta)$ (detallado más adelante en la Sección 2.1.2). En este caso, al diferenciar la función de costo respecto al parámetro θ, al aplicar la regla de la cadena el gradiente queda:

$$\frac{\partial L_i(\theta)}{\partial \theta} = x^{(i)} \odot (h_\theta(x^{(i)}) - y^{(i)}) \tag{2.2}$$

Notar que la constante $\frac{1}{2}$ utilizada en la Ecuación 2.1 es agregada para que sea cancelada al ser derivada dicha función, y con ello se ahorra el cómputo de multiplicar todos los valores del vector dado en la Ecuación 2.2.

En el caso de la regresión lineal, los valores a predecir son continuos. Cuando se tratan problemas de clasificación, la predicción se realiza sobre una variable discreta que puede tomar sólo determinados valores. Para ello, lo que se intenta es predecir la probabilidad de que un ejemplo dado pertenezca a una clase contra la posibilidad de que pertenezca a otra/s. En una *regresión logística* la clasificación se realiza mediante la predicción de etiquetas binarias, por lo cual $y^{(i)} \in \{0, 1\}$. Llamando $z = \theta^\top x$ a la salida lineal, específicamente se trata de aprender una función de la forma:

$$\begin{aligned} P(y = 1 \mid x) &= h_\theta(x) = \frac{1}{1 + \exp(-z)} \equiv \sigma(z), \\ P(y = 0 \mid x) &= 1 - P(y = 1 \mid x) = 1 - h_\theta(x) \end{aligned} \tag{2.3}$$

La función $\sigma(z)$ es denominada "función logística" o "sigmoidea" (desarrollada en la Sección 2.2.2), y su imagen cae en el rango $[0, 1]$ por lo que $h_\theta(x)$ puede interpretarse como una probabilidad. Por lo general este tipo de regresión se asocia con otra función de costo denominada *Entropía Cruzada*, la cual se define mediante la siguiente expresión:

$$L_i(\theta) = -\left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right)$$

A partir de ello, se puede clasificar un nuevo patrón chequeando cuál de las dos clases resulta con mayor probabilidad. Cuando se deben manejar más de dos clases, la *regresión softmax* es utilizada como clasificador para tratar con etiquetas $y^{(i)} \in \{1, \ldots, K\}$, donde K es el número de clases.

Dado un vector de entrada x, se busca estimar la probabilidad que $P(y = k \mid x)$ para cada valor de $k = 1, \ldots, K$. Entonces, la hipótesis debe resultar en un vector de dimensión K cuyos elementos sean las probabilidades estimadas (y sumen uno). Concretamente, la función $h_\theta(x)$ toma la forma:

$$h_\theta(x) = \begin{bmatrix} P(y = 1|x; \theta) \\ P(y = 2|x; \theta) \\ \vdots \\ P(y = K|x; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(z^{(j)})} \begin{bmatrix} \exp(z^{(1)}) \\ \exp(z^{(2)}) \\ \vdots \\ \exp(z^{(K)}) \end{bmatrix}$$

Notar que por cada $k \in \{1, \ldots, K\}$ existe un $z^{(k)} = \theta^{(k)\top} x$ para cada parámetro $\theta^{(k)}$ del modelo, y el término $\frac{1}{\sum_{j=1}^{K} \exp(z^{(j)})}$ normaliza la distribución para que sume uno en total. En cuanto a la función de costo, es similar a la definida para la regresión logística excepto que se debe sumar sobre los K diferentes valores de las clases posibles. Para ello, se puede expresar cada etiqueta $y^{(i)}$ de forma "binarizada" como un vector de dimensión K, que esté compuesto de ceros excepto en la posición correspondiente a la clase apuntada por $y^{(i)}$. Llamando $t_k^{(i)}$ a este vector, donde para un patrón i está compuesto de ceros excepto en la posición k que vale 1, y recordando que $h_\theta(x^{(i)})$ es un vector también de dimensión K (por ser la salida de la función *softmax*), se tiene:

$$L_i(\theta) = -\sum_k \left(t_k^{(i)} \cdot \log(h_\theta(x^{(i)}))\right) \tag{2.4}$$

Notar que esto compone un clasificador lineal, ya que las predicciones sólo se basan en el logaritmo de una distribución de probabilidades [**?**]. En cuanto al gradiente de dicha función, se puede demostrar que mediante la regla de la cadena se llega a la siguiente expresión simple:

$$\nabla_z L_i(\theta) = h_\theta(x^{(i)}) - t_K^{(i)} \tag{2.5}$$

Recordar que $\nabla_z L_i(\theta)$ resulta en un vector por ser la derivada de la función de costo $L_i(\theta)$ (que es un escalar) respecto a la salida lineal z, y como ya se dijo ambos se utilizan para la optimización del parámetro θ mediante algoritmos basados en gradientes. También es preciso aclarar que a esta función se le pueden adicionar otros términos que sirvan para agregar penalizaciones al modelo a optimizar, como pueden ser algunas normas regularizadoras (que más adelante se detallan en la Sección 2.3.2.2) las cuales deben tener un gradiente asociado para la optimización mencionada.

Optimización

Una vez que se tiene un modelo parametrizado por un conjunto θ, y una función de costo para medir el error de aproximación a la función deseada y, se procede a optimizarlo para mejorar dicha aproximación en base a un conjunto de datos. Para ello se emplean algoritmos de optimización que, por cada ejemplo del conjunto de datos, utilizan el valor obtenido por la función de costo para actualizar los parámetros del modelo. Este procedimiento se realiza sobre todos los datos disponibles para minimizar el error producido por el modelo, y generalmente se aplica de forma iterativa a través de "épocas".

La idea es que buscar el mejor conjunto de parámetros se puede hacer más fácilmente mediante un refinamiento iterativo, por el cual se parte de un conjunto inicial (designado de alguna forma, como aleatoriamente) que luego se refina iterativamente de forma que en cada pasada éste se mejora un poco para minimizar la función de costo asociada. Este proceso puede interpretarse como recorrer paso a paso un espacio de búsqueda (el de parámetros) donde en cada uno de ellos se busca dirigirse a la región que asegura el mínimo costo posible. Concretamente, se empieza con un θ aleatorio y luego se computan modificaciones $\delta\theta$ sobre el mismo tal que al actualizar a $\theta + \delta\theta$ el costo sea menor.

Dada una función de costo u objetivo $L(\theta)$ y la capacidad de calcular su gradiente respecto a los parámetros $\theta \in \mathbb{R}^d$, el procedimiento de evaluar iterativamente sus valores para actualizar θ se denomina *gradiente descendiente* (**GD!**) [**?**]. Esta actualización se efectúa en la dirección opuesta del gradiente de la función objetivo, y mediante una tasa de aprendizaje η se define el tamaño de los pasos a realizar hasta el mínimo de esta función, lo cual se resume como:

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta) \tag{2.6}$$

Aunque existen otras formas de realizar la optimización basada en gradientes (e.g. el método quasi-Newton L-BFGS), el gradiente descendiente es actualmente el más común y estándar para optimizar la función de costo en un modelo (especialmente en redes neuronales).

Cuando el conjunto de datos a utilizar en la optimización es realmente grande, resulta ineficiente computar la función de costo y su gradiente sobre el con-

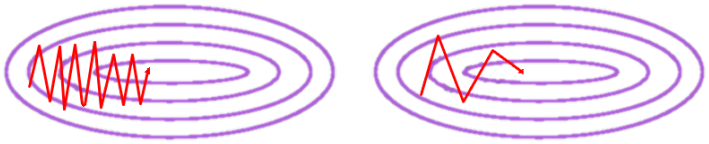


Figura 2.1: Influencia del *momentum* en la optimización por gradiente descendiente (GD). Izquierda, GD sin *momentum*. Derecha, GD con *momentum*.

junto entero por cada actualización a realizar. Es por ello que se suele implementar un enfoque del gradiente descendiente denominado *mini-batch*, en donde cada actualización se computa sobre un subconjunto o "batch" muestreado del conjunto original. En la práctica, el gradiente obtenido del *mini-batch* es una buena aproximación del gradiente total, y con ello se obtiene una convergencia más rápida mediante una actualización de parámetros más frecuente [**?**].

El caso extremo del *mini-batch* es cuando se utiliza un único ejemplo como batch, y en ese caso el proceso se denomina *gradiente descendiente estocástico* (conocido en inglés como *Stochastic Gradient Descent* o **SGD!**). Aunque esa es su definición técnica, en la práctica se suele referir como SGD al gradiente descendiente con *mini-batch* ya que, en la mayoría de las herramientas de optimización, resulta más eficiente evaluar los gradientes para un subconjunto de datos que para una única entrada a la hora de computar una actualización. En cuanto al tamaño de batch a utilizar, se debe tener en cuenta las restricciones de memoria que existan, aunque por lo general suele ser de entre 10 y 100 ejemplos.

Para acelerar la optimización por GD en una dirección dada, se suele agregar a la Ecuación 2.6 un término de *momentum* que básicamente aplica una fracción γ de la actualización hecha en la época anterior sobre la actual, lo cual resulta:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta L(\theta) \tag{2.7}$$
$$\theta_t = \theta_{t-1} - v_t \tag{2.8}$$

En la Figura 2.1 se puede apreciar el efecto de aplicar *momentum* en la optimización, por el cual se disminuyen las oscilaciones al acelerarse el proceso en la dirección relevante [**?**]. El término γ suele fijarse en 0.9 o un valor similar.

Existe otra forma de aplicar un *momentum*, distinta a la de computar el gradiente sobre el θ actual para aplicar el término en la actualización. El gradiente acelerado de Nesterov (en inglés, conocido como *Nesterov Accelerated Gradient* o **NAG!**) va más allá al calcular el gradiente no sobre el parámetro actual sino sobre una aproximación de su valor futuro de la siguiente forma:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta L(\theta_{t-1} + \gamma v_{t-1}) \tag{2.9}$$
$$\theta_t = \theta_{t-1} - v_t \tag{2.10}$$

Esto tiene una garantía teórica de convergencia para funciones objetivo convexas [**?**], y en la práctica suele funcionar bastante mejor respecto al uso del *momentum* estándar.

Todos los enfoques explicados manipulan la tasa de aprendizaje η de forma global y siempre igual para todos los parámetros. *Adagrad* es un algoritmo de

optimización que calcula de forma adaptativa su tasa respecto a los parámetros. Para ello utiliza una tasa de aprendizaje distinta para cada parámetro θ en cada paso, y no una misma actualización para todos a la vez, lo cual se resume como:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_\theta L(\theta)_t \tag{2.11}$$

donde $\odot$ es una multiplicación elemento a elemento entre la matriz G_t y el vector gradiente de la función objetivo. Aquí ϵ es un término de suavizado que evita divisiones por cero (usualmente pequeño, en el orden de 1e-8), y G_t es una matriz diagonal cuya diagonal corresponde a la suma de los gradientes cuadrados actuales (i.e. $\nabla_\theta L(\theta)_t^2$). Uno de los beneficios principales de esta técnica es que elimina la necesidad de ajustar manualmente la tasa η, y se ha mostrado en estudios que mejora importantemente la robustez de SGD especialmente en redes neuronales profundas [**?**].

Adadelta es una extensión de *Adagrad* que busca reducir la forma agresiva y monotónicamente decreciente de obtener la tasa de aprendizaje [**?**]. En lugar de acumular todos los gradientes cuadrados pasados, Adadelta restringe una ventana con un tamaño fijo para acumular estos valores en una suma que se define recursivamente como una media móvil decreciente. Esta media móvil exponencial $E[\nabla_\theta L(\theta)^2]_t$ en la época t depende sólo del promediado anterior y el gradiente actual tal que:

$$E[\nabla_\theta L(\theta)^2]_t = \gamma E[\nabla_\theta L(\theta)^2]_{t-1} + (1-\gamma)\nabla_\theta L(\theta)_t^2 \tag{2.12}$$

Los gradientes acumulados en la media móvil se deben inicializar en 0 para hacer posible esta actualización recursiva. A partir de ello, respecto a la Ecuación 2.11 de *Adagrad* se cambia la matriz G_t por esta media móvil de forma tal que:

$$\theta_{t+1} = \theta_t + \Delta\theta, \qquad \Delta\theta = -\frac{\eta}{\sqrt{E[\nabla_\theta L(\theta)^2]_t + \epsilon}} \nabla_\theta L(\theta)_t \tag{2.13}$$

De allí se puede ver que el denominador de la actualización corresponde a la raíz del error cuadrático medio del gradiente (i.e. $RMS[\nabla_\theta L(\theta)]_t$).

Dado que las unidades en esta actualización no coinciden (ya que deberían tener las mismas hipotéticas unidades que el parámetro en cuestión), los autores del método definieron otra media móvil exponencial pero aplicada sobre el cuadrado de las actualizaciones en lugar del cuadrado de los gradientes. Por lo tanto, ahora el término $RMS[\Delta\theta]_t$ es desconocido, por lo cual se aproxima con el RMS de las actualizaciones hasta el paso anterior. A partir de esto, la actualización definida para *Adadelta* resulta:

$$\theta_{t+1} = \theta_t + \Delta\theta, \qquad \Delta\theta = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[\nabla_\theta L(\theta)]_t} \nabla_\theta L(\theta)_t \tag{2.14}$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{2.15}$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \tag{2.16}$$

Notar que la constante η desaparece de la ecuación original ya que no resulta necesario definir una tasa de aprendizaje para la actualización, aunque en algunas implementaciones se incorpora para tener otro control de la optimización.

Se puede encontrar una buena referencia de estos métodos explicados en distintos artículos y tutoriales de optimización [**?**] [**?**] [**?**] [**?**].

Métricas de evaluación

Para conocer el comportamiento del modelo optimizado en la tarea asignada, se establecen métricas para medir su desempeño sobre un conjunto de datos y a partir de ello poder ajustar el mismo para mejorar sus resultados. Dichas métricas son específicas del tipo de problema tratado, por lo que se distinguen para tareas de *clasificación* y *regresión*.

Clasificación

En problemas supervisados de clasificación, cada patrón de un conjunto de datos tiene asignada una etiqueta de la clase que debe predecir el modelo. A raíz de ello, los resultados para cada patrón corresponden a cuatro categorías:

- Verdadero Positivo (VP): la etiqueta es positiva y la predicción también.
- Verdadero Negativo (VN): la etiqueta es negativa y la predicción también.
- Falso Positivo (FP): la etiqueta es negativa pero la predicción es positiva.
- Falso Negativo (FN): la etiqueta es positiva pero la predicción es negativa.

La forma más sencilla de medir el desempeño de una clasificación es calcular su exactitud mediante la cantidad de aciertos obtenidos para la clase dada sobre el total de las predicciones hechas (medida conocida como *accuracy* o **ACC!**). No obstante, esta medida no es buena especialmente cuando se tratan conjuntos de datos no balanceados por clases. Si se quiere modelar un predictor de anomalías, es muy probable que los datos utilizados tengan más ejemplos de comportamiento normal que de algo anómalo, y si se obtiene un 95 % de predicciones correctas sobre el total no hay seguridad de que el modelo sea bueno si el 5 % restante abarca muchas o todas las anomalías no predichas.

Para evitar esto, se definen medidas de *precisión* y *sensibilidad* (mejor conocidas en inglés como *precision* y *recall*) que tienen en cuenta el "tipo" de error cometido. En tareas de clasificación, el valor de *precision* P determina para una clase la cantidad de Verdaderos Positivos dividido por el número total de elementos clasificados como pertenecientes a dicha clase (i.e. la suma de Verdaderos Positivos y Falsos Positivos), mientras que el valor de *recall* R representa la cantidad de Verdaderos Positivos predichos sobre el total de elementos que realmente pertenecen a la clase en cuestión (i.e. la suma de Verdaderos Positivos y Falsos Negativos) [**?**]. Por lo tanto, sus expresiones quedan dadas por:

$$P = \frac{VP}{VP + FP} \qquad R = \frac{VP}{VP + FN} \tag{2.17}$$

En un contexto de recuperación de información, la *precision* determina la cantidad de elementos relevantes del total que fueron recuperados, mientras que el *recall* representa la fracción de elementos recuperados del total que son relevantes. Una medida que combina a estas dos mediante una media armónica es el Valor-F (mejor conocida e inglés como F–Score o F–Measure). La misma ofrece un balance entre *precision* y *recall* ajustado por un parámetro β, y un caso muy utilizado de esta medida es el F1-Score donde $\beta = 1$, tal que:

$$F(\beta) = (1 + \beta^2)(\frac{PR}{\beta^2 P + R}) \qquad F1 = \frac{2PR}{P + R} \tag{2.18}$$

En caso de que el sistema esté diseñado para tratar problemas multi-clase (dos o más clases), las métricas deben ajustarse para soportar esta característica [**?**]. Para ello, se procede a calificar positiva o negativa a una predicción respecto a la etiqueta en base al contexto de una clase en particular. Cada tupla de etiqueta y predicción se evalúa para cada una de las clases, y se considera como positiva para dicha clase o negativa para el resto de las clases. Así, un verdadero positivo ocurre cuando la predicción y la etiqueta coinciden, mientras que un verdadero negativo se da cuando ni la predicción ni la etiqueta corresponden a la clase tomada en cuenta. Es por ello que para un simple patrón resultan múltiples verdaderos negativos en problemas de más de dos clases (y la misma idea se extiende para caracterizar FNs y VNs). Para seguir este enfoque de múltiples etiquetas posibles, se derivan las medidas ya definidas para evaluar la clasificación respecto a todas las clases en dos formas posibles [**?**]:

i) Computar el promedio de las mismas medidas calculadas por cada una de las clases (*macro-promediado*).

ii) La suma de cuentas para obtener VP, FP, VN, FN acumulativos, y a ello aplicar una métrica de evaluación (*micro-promediado*).

En la Tabla 2.1 se exponen las medidas de evaluación explicadas utilizando estas aproximaciones, donde el subíndice μ representa a aquella medidas con *micro-promediado* y el subíndice M a aquellas con *macro-promediado*. Notar que esta última aproximación para evaluar una clasificación abarca y generaliza las métricas explicadas para problemas de dos clases.

Una forma visual de representar el desempeño del modelo en la clasificación hecha es mediante una *matriz de confusión*, en la cual se presentan los resultados de las predicciones en cantidades por cada una de las clases en cuestión. Dicha matriz cuadrada tiene dimensión igual a la cantidad de clases tratada, y cada columna corresponde a las predicciones hechas mientras que cada fila representa las instancias de la clase actual u original a predecir. Esta disposición de los resultados facilita su interpretación (de allí proviene el nombre, ya que se conoce rápidamente en qué clase/s se confunde el predictor) y además permite derivar rápido los cálculos de las métricas explicadas. En el caso de dos clases, es sencillo expresar las categorías de resultados sobre cada celda de la matriz, y para el caso multi-clase se representan de la forma mencionada anteriormente tal como se puede apreciar en la Tabla 2.2 cuando se tienen 3 clases. De allí se puede notar que una clasificación perfecta resulta en una matriz de confusión diagonal.

Tabla 2.1: Medidas de evaluación en problemas de clasificación multi-clases.

Medida	**Fórmula**	**Sentido de la evaluación**
ACC	$\frac{\sum_{i=1}^{C} \frac{VP_i+VN_i}{VP_i+FN_i+FP_i+VN_i}}{C}$	Promedio de la efectividad por clase del clasificador
P_μ	$\frac{\sum_{i=1}^{C} VP_i}{\sum_{i=1}^{C}(VP_i+FP_i)}$	Suma de la precisión lograda en la clasificación por cada clase.
R_μ	$\frac{\sum_{i=1}^{C} VP_i}{\sum_{i=1}^{C}(VP_i+FN_i)}$	Suma de la sensibilidad lograda en la clasificación por cada clase.
$F_\mu(\beta)$	$(1+\beta^2)(\frac{P_\mu R_\mu}{\beta^2 P_\mu + R_\mu})$	Balance entre precisión y sensibilidad basada en la suma de decisiones realizadas por clase.
P_M	$\frac{\sum_{i=1}^{C} \frac{VP_i}{VP_i+FP_i}}{C}$	Promedio de los cálculos de precisión lograda por cada clase.
R_M	$\frac{\sum_{i=1}^{C} \frac{VP_i}{VP_i+FN_i}}{C}$	Promedio de los cálculos de sensibilidad lograda por cada clase.
$F_M(\beta)$	$(1+\beta^2)(\frac{P_M R_M}{\beta^2 P_M + R_M})$	Balance entre precisión y sensibilidad basada en un promedio sobre el total de clases.

Tabla 2.2: Esquema de matriz de confusión para evaluar predicciones sobre 3 clases.

		Predicción		
Actual		VP_1	FN_1 FP_2	FN_1 FP_3
		FP_1 FN_2	VP_2	FN_2 FP_3
		FP_1 FN_3	FP_2 FN_3	VP_3

Regresión

Cuando la variable a predecir es de naturaleza continua, el modelo a construir compone un sistema de regresión. Para ello, la forma de conocer la precisión en la predicción no debe hacerse por cantidad de aciertos sino midiendo un error en la salida. Las medidas más utilizadas para ello son el *Error Cuadrático Medio* (en inglés, *Mean Squared Error* o **MSE!**), el *Error Absoluto Medio* (en inglés, *Mean Absolute Error* o **MAE!**) y el coeficiente de determinación R^2 [**?**].

$$MSE = \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N} \qquad RMSE = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}} \tag{2.19}$$

$$MAE = \frac{\sum_{i=0}^{N-1}|y_i - \hat{y}_i|}{N} \qquad RMAE = \sqrt{\frac{\sum_{i=0}^{N-1}|y_i - \hat{y}_i|}{N}} \tag{2.20}$$

$$R^2 = 1 - \frac{MSE}{VAR(y) \cdot (N-1)} = 1 - \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{N-1}(y_i - \bar{y_i})^2} \tag{2.21}$$

El término error aquí representa la diferencia entre el valor verdadero y_i y el predicho $\hat{y_i}$. Para ello se calcula una suma de los residuos, dividida por el número de grados de libertad N. Esto pueder verse como la media de las desviaciones de cada predicción respecto a los verdaderos valores, generadas por un modelo estimado durante un espacio de muestra particular. Valores de error bajos significan que el modelo es más preciso en sus predicciones, y un error total de 0 indica que el modelo se ajusta a los datos perfectamente. Tanto el cuadrado como el valor absoluto del error medido en cada suma captura la magnitud total del mismo, ya que algunas diferencias pueden ser negativas. Se suele utilizar la raíz cuadrada de el error calculado para hacerlo independiente de la escala para la comparación de distintos modelos.

El coeficiente de determinación R^2 provee una medida de cuán bien se ajusta el modelo a los datos. El mismo puede ser interpretado como la proporción de la variación explicada por el modelo. Cuanto mayor es dicha proporción, mejor es el modelo en sus predicciones, siendo que el valor 1 indica un ajuste perfecto.

Ajuste de hiperparámetros

Como puede notarse, durante el diseño de un modelo existen diversos parámetros y configuraciones que se deben especificar en base a los datos tratados y la tarea asignada para dicho modelo. Estos pueden ser valores continuos de los que se puede tener una idea de rango posible (e.g. taza de aprendizaje para el algoritmo de optimización) o categorías particulares de algún componente (e.g. algoritmo de clasificación a utilizar). La configuración elegida es crucial para que el proceso de optimización resulte en un modelo con buen desempeño en la tarea asignada, y en el caso de los hiperparámetros (así se les denomina a los valores a ajustar en una configuración) elegir un valor determinado puede ser difícil especialmente cuando son sensibles en su variación.

Una forma asistida para realizar determinar una buena configuración es implementar un algoritmo de búsqueda de hiperparámetros, el cual realiza elecciones particulares tomando muestras sobre los rangos o categorías posibles que se definan, así luego se optimiza un modelo por cada configuración especificada. Opcionalmente, también se puede utilizar *validación cruzada* [**?**] para estimar la generalización del modelo obtenido con la configuración y su independencia del conjunto de datos tomado. Resumiendo, una búsqueda consta de:

- Un modelo estimador (de regresión o clasificación).

Figura 2.2: Tipos de ajuste que puede lograr un modelo sobre los datos que utiliza para su entrenamiento.

- Un espacio de parámetros.
- Un método para muestrear o elegir candidatos.
- Una función de evaluación del modelo.
- (Opcional) Un esquema de validación cruzada.

Una forma de muestrear el espacio delimitado de parámetros es tomando de manera exhaustiva los valores que caen en una grilla, la cual generalmente se determina por una discretización equiespaciada del espacio tratado. A este método se le denomina "búsqueda por grilla" o *grid search* en inglés, y se caracteriza por su simplicidad al ser fácil de implementar aunque por ello es más propenso a sufrir un problema denominado *maldición de la dimensionalidad*. Este último señala que a medida que el espacio de búsqueda es mayor, la estrategia se hace menos eficiente ya que requiere una cantidad mucho mayor de muestras necesarias para obtener mejores candidatos [**?**].

Otra forma que evade buscar exahustivamente sobre el espacio de parámetros (lo cual también es potencialmente costoso si dicho espacio es de una dimensión alta), es la de muestrear una determinada cantidad de veces el espacio delimitado, en forma aleatoria y no sobre una grilla determinada. Este método se denomina "búsqueda aleatoria" o *random search* en inglés, y es tan fácil de implementar como el *grid search* aunque se considera más eficiente especialmente en espacios de gran dimensión [**?**].

Control de la optimización

El ajuste de hiperparámetros y la mejora de las configuraciones y elecciones hechas en el diseño buscan que el modelo a construir tenga el mejor desempeño posible. Sin embargo, esto no se puede efectuar sobre el conjunto de datos con el cual se ajusta el modelo (llamado "conjunto de entrenamiento") ya que así no puede garantizarse que el modelo generalice y se desempeñe aproximadamente igual con otro conjunto de datos que no se le presentó nunca. Esta cuestión introduce un problema denominado "sobreajuste" (mejor conocido en inglés como *overfitting*), por el cual un modelo optimizado se desempeña muy bien con los datos que utilizó en su ajuste, pero ocurre lo contrario sobre otro conjunto de

datos que no haya "visto" o usado jamás. Este problema siempre trata de ser evitado ya que el desempeño obtenido no es representativo sobre casos en los que se pretenda utilizar el modelo en un escenario real. La Figura 2.2 esquematiza cómo se desempeña el modelo sobre los datos de entrenamiento cuando ocurre el fenómeno de sobreajuste.

Para mitigar el *overfitting* se debe contar con un "conjunto de validación", construido a partir del total de datos disponibles al separar una porción para este propósito de validación. Este conjunto no se utiliza para entrenar el modelo, sino que durante dicho proceso sirve para evaluar y monitorear que el modelo está generalizando y no se está ajustando demasiado a los datos de entrenamiento. También se necesita definir un "conjunto de prueba", el cual no debe ser usado nunca durante el modelado ya que representa datos que se le van a presentar al modelo luego de que ya haya sido construido y que seguramente no son iguales a los que el mismo utilizó durante su entrenamiento. En cuanto a la magnitud a definir para cada porción elegida, en términos del conjunto total de datos, lo que se suele realizar es partir en 70 % para entrenamiento, 15 % para validación y el 15 % restante para prueba.

Finalmente, durante la optimización se precisa la información del desempeño obtenido en el modelo para conocer cuándo es lo suficientemente bueno como para frenar el proceso. (ya que generalmente no se obtienen niveles óptimo) . Es por eso que suelen establecerse ciertas reglas o criterios de corte para que la optimización se realice hasta lograrse un nivel de desempeño definido, o bien frenarla cuando no se está obteniendo un ajuste deseable.

Concluyendo, el ajuste de parámetros (en forma manual o asistida con algoritmos de búsqueda) se realiza para optimizar el modelo sobre los datos de entrenamiento, y el desempeño obtenido con dicha configuración se evalúa con un conjunto de validación (que no debe ser utilizado para ajustar el modelo). Una vez que se encuentra la mejor configuración, se establece como fija y allí se evalúa sobre los datos de prueba para conocer finalmente el desempeño del modelo construido.

Redes Neuronales Artificiales

Dentro del campo científico de la Inteligencia Artificial, las *Redes Neuronales Artificiales* (**RNA!**) comprenden una rama antigua pero destacada, especialmente en la actualidad luego del surgimiento del aprendizaje produndo. Se entiende como **RNA!** a un sistema con elementos procesadores de información de cuyas interacciones locales depende su comportamiento en conjunto [**?**]. Dicho sistema trata de emular el comportamiento del cerebro humano, adquiriendo conocimiento de su entorno mediante un proceso de aprendizaje y almacenándolo para disponer de su uso [**?**].

Las **RNA!**s son implementadas en computadoras para imitar la estructura neuronal de un cerebro, tanto en programas de software como en arquitecturas de hardware, por lo cual se utilizan para componer un sistema de aprendizaje maquinal. No obstante, sólo consiste en una aproximación debido a las diferencias significativas que se presentan en la Tabla 2.3 [**?**]. Por lo general, las computadoras presentan una arquitectura de tipo Von Neumann basada en un microprocesador muy rápido capaz de ejecutar en serie instrucciones complejas

Tabla 2.3: Diferencias entre cerebro humano y computadora convencional

Características	Cerebro humano	Computadora convencional
Velocidad de proceso	Entre 10^{-3} y 10^{-2} seg	Entre 10^{-9} y 10^{-8} seg
Nivel de procesamiento	Altamente paralelo	Poco o nulo paralelizado
Número de procesadores	Entre 10^{11} y 10^{14}	Entre 4 y 8
Conexiones	10.000 por procesador	Pocas
Almacenamiento del conocimiento	Distribuido	En posiciones precisas
Tolerancia a fallos	Amplia	Poca o nula
Consumo de energía en una operación/seg	10^{-16} Julios	10^{-6} Julios

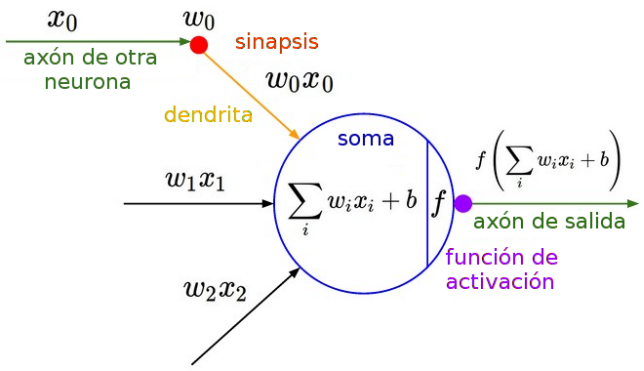


Figura 2.3: Modelo matemático de una neurona.

de forma fiable, mientras que el cerebro está compuesto por millones de procesadores elementales o neuronas, que se interconectan formando redes. Además, las neuronas biológicas no adquieren conocimiento por ser programadas sino que lo hacen a partir de estímulos que reciben de su entorno, y operan mediante un esquema masivamente paralelo distinto al serializado o poco paralelo de la computadoras convencionales.

Arquitectura

La unidad básica de cómputo en el cerebro es una neurona, la cual recibe señales de entrada desde sus dendritas y las procesa en su cuerpo, llamado soma, para producir señales de salida mediante un único axón. Estas últimas a su vez interactúan por sinápsis con las dendritas de otras neuronas y con ello se logra la comunicación de estímulos en todo el cerebro.

Para modelar el comportamiento de las neuronas, la idea es que las sinápsis que producen pueden controlarse mediante *pesos sinápticos* que definan una magnitud de la influencia que ejerce una con otra (y también dirección, al poder excitar o inhibir mediante pesos positivos o negativos, correspondientemente).

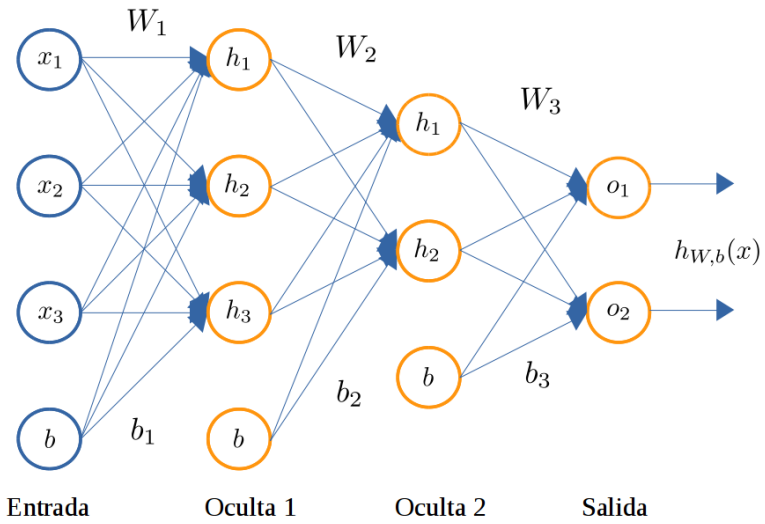


Figura 2.4: Arquitectura básica de un MLP con 4 capas.

Los pesos sinápticos w_0 multiplican las señales de entrada x_0 que llegan por las dendritas para ejercer una suma ponderada en el soma, y si el resultado supera un cierto umbral la neurona se "activa" enviando un estímulo a lo largo de su axón. Dicha suma puede incorporar además un término de sesgo b que participa sin multiplicarse por la entrada. En este modelo se considera que no interesa conocer el preciso tiempo en que se activa la neurona sino la frecuencia en que ocurre, por lo cual ello es representado mediante una *función de activación* [**?**]. En la Figura 2.3 se representa gráficamente el modelo explicado, el cual constituye lo que se denomina como "perceptrón simple'.

Para modelar una red neuronal artificial las neuronas se agrupan en capas, de forma tal que todas las unidades de una capa se conectan con todas las neuronas de sus capas próximas para formar una estructura interconectada. En la forma más simple, se tiene una capa de entrada que proyecta la señal entrante en una capa de salida. Cuando se incorporan capas intermedias (denominadas capas ocultas), la red adquiere más niveles de procesamiento entre su entrada y salida, y lo que se forma es un "perceptrón multicapa" (conocido en inglés como *Multi Layer Perceptron* o **MLP!**) [**?**]. En esta configuración, para producir la salida de la red se computan sucesivamente todas las activaciones una capa tras otra, donde puede notarse que una red con n capas equivale a tener n perceptrones simples en cascada, donde la salida del primero es la entrada del segundo y así sucesivamente. Además, cada capa puede tener diferente número de neuronas, e incluso distinta función de activación.

Siguiendo el modelo matemático de un perceptrón simple, los pesos sinápticos ahora pueden expresarse en conjunto de forma vectorial como matrices, así como también el sesgo se representa como un vector. Por lo tanto, dada un vector de entrada x, la suma ponderada efectuada en una capa se expresa como un producto entre la matriz de pesos sinápticos W y dicha entrada x, en la cual también se suma el vector de sesgo o *bias* b. Al resultado de esto se le aplica

la función de activación, con lo cual se produce la salida final de la capa. En la Figura 2.4 se representa la arquitectura de un perceptrón multicapa, mostrando las interacciones que tienen sus unidades desde la entrada hasta su salida.

Funciones de activación

Una función de activación determina cómo se transforman las entradas a través de la red neuronal, lo cual es determinante para que logre su capacidad de aprender funciones complejas. En general, se caracterizan por ser *no lineales* ya que incrementan el poder de expresión y con ello se pueden obtener interesantes representaciones de las entradas que ayuden a la tarea designada para la red. La razón por la cual no es conveniente que sean lineales es porque de esa forma no tendría sentido que la red posea más de una capa, ya que la combinación de funciones lineales tiene un resultado lineal. Además, las redes neuronales están pensadas principalmente para tratar tanto problemas de clasificación en donde las clases no son linealmente separables como problemas en donde no se logra una precisión deseable mediante un modelo lineal.

Como se anticipó anteriormente en la arquitectura de una red, cada unidad o neurona de una capa recibe una entrada que es ponderada por sus pesos sinápticos para luego producir una salida activada que sirve como entrada a todas las neuronas de la capa siguiente (o en el caso de ser la última capa, que es el resultado final de la activación completa de la red). Dado un vector x de entrada para una capa de la red, se le aplica una transformación afín (i.e. una transformación lineal por la matriz de pesos W, seguido de una traslación por el vector b) cuyo resultado es la salida lineal z. Dicha salida es la que recibe la función de activación f para producir la salida final de la capa a, lo cual se expresa como :

$$\begin{aligned} z &= Wx + b \\ a &= f(z) \end{aligned} \tag{2.22}$$

Originalmente las funciones de activación más utilizadas eran las sigmoideas, las cuales son la *sigmoidea* (σ) y la *tangente hiperbólica* ($tanh$). Las mismas tienen una inspiración biológica y están delimitadas por un mínimo y un máximo valor, lo cual causa que las neuronas se saturen en las últimas capas de la red neuronal [**?**]. Ambas funciones se definen analíticamente como.

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \qquad tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.23}$$

Otra función de activación que ha sido muy utilizada en muchas aplicaciones es la *lineal rectificada* o **ReLU!**, la cual comprende una no linealidad simple: resulta en 0 para entradas negativas, y para valores positivos se mantiene intacta, por lo cual no tiene valores límites como las funciones sigmoideas. El gradiente de la **ReLU!** es 1 para todos los valores positivos y 0 para los negativos, lo cual hace que, durante la optimización de la red, los gradientes negativos no sean usados para actualizar los pesos sinápticos correspondientes. Además, el hecho de que el gradiente sea 1 para cualquier valor positivo hace que el entrenamiento sea más rápido que con otras funciones de activación no lineales. Por ejemplo, la función sigmoidea tiene muy pequeños gradientes para grandes valores positivos

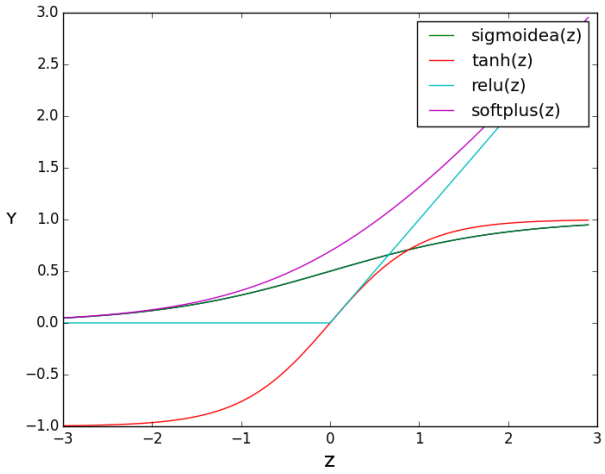


Figura 2.5: Visualización de las funciones de activación para $-3 \leq z \leq 3$.

y negativos, por lo que el aprendizaje practicamente se frena o "estanca" en dichas regiones [**?**]. Es preciso notar que las **ReLU!**s poseen una discontinuidad en 0, por lo cual no es derivable allí la función. No obstante se fuerza a que allí la derivada sea igual a 0, y el hecho de que allí la activación sea 0 otorga buenas propiedades de ralez a la red [**?**]. Una función que aproxima a la **ReLU!** es la *softplus*, que además de ser continua su derivada es la función *sigmoidea*. Ambas funciones entonces se expresan como:

$$relu(z) = max(0, z), \qquad softplus(z) = \log\left(1 + e^{z}\right) \tag{2.24}$$

En la Figura 2.5 se visualizan las funciones de activación explicadas en un dominio definido, mientras que en la Tabla 2.4 se presentan todas las funciones de activación mencionadas con la respectiva derivada de cada una.

Tabla 2.4: Funciones de activación desarrolladas, detallando para cada una tanto su expresión la de su respectiva derivada analítica.

Nombre	**Función**	**Derivada**
Sigmoidea	$f(z) = \dfrac{1}{1+e^{-z}}$	$f'(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$
Tangente Hiperbólica	$f(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	$f'(z) = 1 - tanh^{2}(z)$
ReLU	$f(z) = max(0, z)$	$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
Softplus	$f(z) = \log\left(1 + e^{z}\right)$	$f'(z) = \sigma(z) = \dfrac{1}{1+e^{-z}}$

En el caso de que la red neuronal tratada esté diseñada para realizar tareas de clasificación, se debe tener en cuenta que la última capa tenga una salida

conveniente para ello. Es por eso que la función allí debe ser de clasificación, y para ello se suele utilizar la función *softmax* explicada en la Sección 2.1.

Retropropagación

La propagación hacia atrás de errores o retropropagación (en inglés, *back-propagation*) es un algoritmo de aprendizaje utilizado para efectuar el entrenamiento de redes neuronales. Fue introducido originalmente en la década del 1970, pero cobró realmente importancia y utilidad en el 1986 mediante una publicación que describía el trabajo con distintas redes neuronales donde este algoritmo alcanzaba un aprendizaje bastante más rápido que otros enfoques [**?**]. A raíz de ello se logró resolver problemas que antes no estaban resueltos, y actualmente es casi un estándar para la optimización de redes neuronales.

El procedimiento consiste en que, dado un patrón (x, y), primero se realiza un "paso hacia adelante" para computar todas las activaciones de cada capa a través de la red hasta calcular la salida final de la misma. A partir de esto se puede utilizar la salida deseada y para computar el valor de la función objetivo y su gradiente respecto a la salida, los cuales son necesarios para conocer cuánto deben variar todos los parámetros de la red. Para ello, se calcula en cada unidad i de cada capa l un "término de error" que mide cuánto afectó cada una en las salidas calculadas. Para la capa de salida, dicho término se calcula en base al gradiente ya computado, y a partir de ello se efectúa el "paso hacia atrás" del algoritmo de la siguiente forma: desde la penúltima capa hasta la primera (sin contar la de entrada), se computa cada término de error en base al correspondiente de la capa siguiente (usado para multiplicar los pesos sinápticos dados) y al gradiente de la activación dada, y a partir de ello se puede computar el gradiente de la función objetivo respecto a los parámetros de la red tratados. Se puede notar que el término de error se va propagando desde el final de la red hasta el principio para poder computar el gradiente de la función objetivo respecto a cada parámetro de la red, y en dicho cálculo se aplica la regla de la cadena sucesivamente para derivar estos valores desde las activaciones obtenidas.

Finalmente, el resultado de este algoritmo es el valor de la función de costo y su gradiente respecto a todos los parámetros de la red, lo cual es de utilidad en algoritmos de optimización basados en gradientes como los descriptos en la Sección 2.1.2. Detalles específicos de la implementación se pueden encontrar en tutoriales de redes neuronales [**?**] [**?**], y en el Algoritmo 2 del Apéndice A se resume este proceso explicado.

Aprendizaje profundo

A partir de la introducción sobre *deep learning* realizada en la sección Resumen de este trabajo, así como los antecedentes de sus aplicaciones exitosas mencionados en la Sección **??**, en los siguiente apartados se procede a profundizar acerca de las características que ofrece en el modelado a diferencia de las redes neuronales básicas ya detalladas.

Redes Neuronales Profundas

Existen ciertas particularidades en las redes profundas que despertaron su interés e incrementaron su estudio. Principalmente, se puede demostrar que hay funciones que una red de n capas puede representar de forma compacta (con un número de unidades ocultas que es polinomio del número de entradas) pero una red de $n-1$ capas no puede representar a menos que tenga una gran cantidad exponencial de unidades ocultas, y esto quiere decir que las redes profundas pueden representar significativamente más conjuntos de funciones que las redes de una o pocas capas ocultas [**?**]. Además, una red profunda puede aprender a representar los datos mediante descomposiciones por partes.

La profundidad definida para una red neuronal en la práctica es arbitraria, y depende mucho de la tarea a realizar y los datos disponibles para el ajuste: si la red es poco profunda (e.g. 2 o 3 capas), la misma tendrá menor poder de representación y además se corre el riesgo de *overfitting* si la cantidad de unidades en la capa oculta es grande respecto a la dimensión de entrada; si la red es bastante profunda (e.g. 10 o más capas), se tiene mayor capacidad para el aprendizaje, aunque la gran cantidad de niveles en la red puede ocasionar un problema típico de este modelado denominado *vanishing gradient*. Este último ocurre en el entrenamiento de redes neuronales mediante aprendizaje basado en gradiente y retropropagación, y afecta no sólo a las del tipo multicapa sino también a aquellas del tipo recurrente [**?**].

El *vanishing gradient* se debe a que la señal de error a retropropagar para el ajuste de parámetros decrece exponencialmente con la cantidad de capas, por lo cual las capas que estén más cerca de la entrada se entrenan muy lentamente. Las funciones de activacion utilizadas influyen bastante en este problema: si la imagen del gradiente abarca valores chicos (e.g. sigmoidea, tangente hiperbólica), se corre mayor riesgo de que se "desvanezcan" las actualizaciones para las primeras capas; si dicha imagen comprende valores altos (e.g. ReLU), existe el riesgo de que las actualizaciones sean inestables y dificulten la convergencia de la optimización (problema denominado *exploding gradient*) [**?**].

A raíz de esto, se originaron varias propuestas para mitigar este problema:

1. El "pre-entrenamiento" de las redes neuronales mediante aprendizaje no supervisado para inicializar los pesos sinápticos capa-por-capa posibilitó arquitecturas de múltiples niveles que sólo requerían de un pequeño ajuste en forma supervisada para obtener buenos resultados [**?**] [**?**].

2. Las redes recurrentes LSTM (*Long short-term memory*) componen una arquitectura específicamente diseñada para combatir el *vanishing gradient* [**?**], y actualmente son implementadas a nivel industrial en sistemas de visión computacional y reconocimiento de voz debido a la gran precisión que obtiene en dichas tareas.

3. El aprendizaje residual compone una metodología propuesta para entrenar redes de gran profundidad (de cientos o miles de capas) disminuyendo importantemente el problema mencionado y mostrando resultados competitivos en tareas de visión computacional

En las siguientes secciones se profundiza acerca de la primer propuesta mencionada, pero primero se procede a detallar un procedimiento realizado en cualquier tipo de red neuronal para mejorar la calidad del ajuste de parámetros.

Tratamiento sobre los pesos sinápticos

Para mitigar el problem de *overfitting* mencionado, especialmente cuando la red tiene tantos parámetros libres (i.e. pesos sinápticos y sesgo) que pueden ajustarse demasiado a los datos de entrenamiento, siempre resulta conveniente que estos parámetros reciban un tratamiento apropiado desde que se instancian hasta que se optimizan. A continuación se describen dos formas de realizar esto, las cuales adquirieron especial importancia con el origen del aprendizaje profundo debido a la cantidad de parámetros que poseen las redes de ese tipo.

Inicialización

La inicialización de los pesos sinápticos en una red neuronal influye mucho en su desempeño y el tiempo que se requiere para optimizarlo. Lo deseable es que los pesos sinápticos se inicialicen con valores cercanos (pero no iguales) a 0, por lo cual puede pensarse en que dichos valores se obtengan de un muestreo sobre una distribución de probabilidades que tenga media igual a 0 y una varianza pequeña para que los valores sean cercanos a 0. Dicha distribución puede ser normal o uniforme, y se ha comprobado que en la práctica la elección de una u otra tiene relativamente poco impacto en el desempeño final. En cuanto a que los valores sean pequeños, se debe tener cuidado ya que eso implica también que, durante la retropropagación, los gradientes utilizados para actualizar los pesos también sean pequeños (ya que son proporcionales) y con ello las actualizaciones se "desvanezcan" en la propagación, especialmente con redes profundas.

Por lo tanto para inicializar los pesos de esta forma se debe controlar la varianza de la distribución a muestrear, y que además su valor tenga relación con la dimensión de entrada que tienen los pesos sinápticos. Una recomendación es la de escalar la varianza a $\frac{1}{\sqrt{n}}$, siendo n el número de entradas que tiene la matriz de pesos [**?**]. En la práctica, es muy utilizado que la varianza sea $\sqrt{\frac{2}{n}}$, lo cual muestra buen comportamiento en redes neuronales profundas (especialmente cuando la función de activación es una **ReLU!**) [**?**]. Otra forma de inicializar los pesos, recomendada para las funciones de activación sigmoideas, es la de utilizar para el muestreo una distribución uniforme que esté en el rango $\pm\sqrt{\frac{6}{n_{in}+n_{out}}}$ para la función Tanh, y en el rango $\pm 4{,}0\sqrt{\frac{6}{n_{in}+n_{out}}}$ para la sigmoidea [**?**]. En cuanto al vector de sesgo, por lo general se suelen inicializar todos sus valores iguales (o muy aproximado, según algunos trabajos) a 0. No se requiere ninguna técnica de muestreo ya que, según muchos estudios, el mayor impacto en la inicialización de los parámetros está dado por los pesos sinápticos [**?**].

Regularización

Como se ha dicho anteriormente, es deseable que las redes neuronales sean capaces de generalizar las aptitudes adquiridas durante el entrenamiento para tener un buen desempeño al presentarse patrones nunca vistos. Para prevenir el problema del *overfitting*, un buen tratamiento es incoporar términos de regularización sobre los pesos sinápticos. Con ello se penaliza la complejidad del modelo en términos del ajuste a los datos de entrenamiento, de forma que pueda obtenerse generalización sobre los datos de prueba. Entre las formas de regula-

rización más utilizadas, se destacan tres técnicas:

Norma L_1

Término que se agrega a la función objetivo a optimizar en la red neuronal, y que tiene la particularidad de conducir a que la matriz de pesos sea "rala" (es decir, que algunos valores sean muy cercanos o iguales a 0). Esta propiedad puede ser deseable para que se utilicen sólo un subconjunto ralo de las entradas más importantes y se produzca robustez ante entradas con ruido [**?**]. Agregando este término, la función de costo y su gradiente quedan:

$$\begin{aligned} L &= L_0 + \lambda_1 \sum_l \sum_i \sum_j |W_{ji}^{(l)}| \\ \nabla L &= \nabla L_0 + \lambda_1 \sum_l \sum_i \sum_j sign(W_{ji}^{(l)}) \end{aligned} \tag{2.25}$$

Aquí, se define a $sign(x)$ como una función que retorna 1 si x es positivo, -1 si es negativo ó 0 en caso que x sea nulo.

Norma L_2

Es la regularización más común que se incorpora en los pesos de una red neuronal, y tiene el efecto de penalizar fuertemente las matrices de pesos con picos o diferencias importantes entre valores, con lo cual se fuerza a que los pesos sean pequeños [**?**]. Al incorporar este término en el costo de la red resulta:

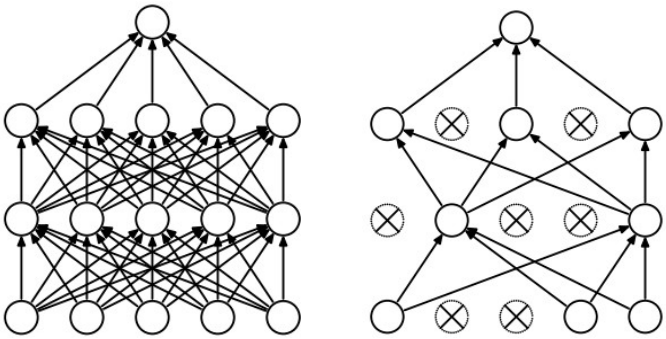
$$\begin{aligned} L &= L_0 + \lambda_2 \sum_l \sum_i \sum_j \frac{1}{2} (W_{ji}^{(l)})^2 \\ \nabla L &= \nabla L_0 + \lambda_2 \sum_l \sum_i \sum_j W_{ji}^{(l)} \end{aligned} \tag{2.26}$$

Notar que a partir de ello, durante la actualización de los pesos sinápticos en la optimización, la regularización por norma L_2 produce que cada peso decaiga linealmente a 0 (i.e. $W = W - \lambda_2 * W$) [**?**].

Dropout

Es un algoritmo extremadamente simple y muy efectivo para lograr la propiedad de generalización sobre redes neuronales [**?**]. A diferencia de las normas L_1 y L_2 no se basa en modificar la función de costo para penalizarla durante la optimización de la red, sino que consiste en modificar la red para regularizarla y hacerla más robusta a información faltante o corrupta.

Dado un patrón de entrada en el entrenamiento (ya que nunca se debe usar durante la etapa de prueba o para hacer predicciones), se permite la activación de una neurona con una cierta probabilidad p definida como parámetro, o de lo contrario se le asigna valor 0 a la salida de la misma. Esto provoca que sólo una fracción del total de neuronas produzca una activación en la salida, con lo que el proceso puede entenderse como que en cada iteración de la optimización se toma un muestreo de la red neuronal completa, y se actualizan sólo los parámetros de dicha red muestreada [**?**]. Durante la etapa de prueba no debe aplicarse



(a) Red neuronal estándar (b) Luego de aplicar Dropout

Figura 2.6: Figura tomada de , donde se representa la anulación de las salidas de cada neurona donde se aplicó dropout.

dropout para que la evaluación sea total en la red. A su vez, en esta etapa es importante realizar un escalado de los pesos en cada capa por p ya que se quiere que las salidas generadas sean idénticas a las salidas esperadas en la etapa de entrenamiento. Para ello, es recomendado hacerlo durante el entrenamiento de forma que el procedimiento para realizar predicciones quede inalterado [**?**], por lo que se deben dividir por p las activaciones producidas en cada capa luego de aplicar Dropout. En ese mismo procedimiento se debe retornar además la máscara binaria producida para saber exactamente cuáles fueron las unidades "tiradas" con el Dropout, así no se actualizan durante el proceso de optimización. En la Figura 2.6 se puede apreciar gráficamente cómo afecta el Dropout a las conexiones de una red neuronal, y el Algoritmo 3 del Apéndice A refleja el procedimiento a realizar para lograr esto durante el entrenamiento de una red.

A partir de todas estas técnicas explicadas, se consideran la siguientes recomendaciones prácticas para obtener resultados buenos en la optimización de una red neuronal:

- En la práctica, es mayormente utilizada la regularización mediante la norma L_2, aunque se suele incorporar también en menor proporción la norma L_1 para lograr también ciertas propiedades de "raleza" sobre los pesos sinápticos. Esta combinación constituye lo que se denomina *regularización de red elástica*[**?**].

- Como se puede notar, la regularización nunca afecta al vector de sesgo b. Esto se debe a que el mismo no interactúa con los datos de forma multiplicativa, y como sólo produce una traslación en el espacio de soluciones no se considera que regularizar dicho vector produzca una moderación importante sobre la solución respecto al ajuste [**?**].

- Por lo general, por cada norma se utiliza la misma constante de penalización λ en todas las capas de la red.

- Como regla general, el Dropout se suele aplicar con $p = 0{,}5$ para todas las capas ocultas, aunque también se puede probar sobre la entrada con $p = 0{,}2$ [**?**] [**?**].

Aprendizaje no supervisado

En las anteriores secciones, se presentaron técnicas para modelar sistemas con aprendizaje maquinal siguiendo únicamente un enfoque supervisado. A diferencia de ello, el aprendizaje no supervisado busca modelar la función hipótesis basándose únicamente en la entrada, lo cual puede expresarse como $h(x) \approx x$. En el caso de las redes neuronales, se traduce en que no requieren otra información más que el vector de entrada para ajustar los pesos de las conexiones entre neuronas (i.e. se prescinde de entradas etiquetadas). En algunos casos, la salida representa el grado de similitud entre la información que se le está ingresando y la que ya se le ha mostrado anteriormente. En otro caso podría realizar una codificación de los datos de entrada, generando a la salida una versión codificada de la entrada (e.g. con menos bits, pero manteniendo la información relevante de los datos), y también algunas redes pueden lograr un mapeo de características, obteniéndose en la salida una disposición geométrica o representación topográfica de los datos de entrada [**?**].

Como se mencionó al principio del presente capítulo, este enfoque del aprendizaje maquinal se utiliza generalmente en dos tipos de aplicaciones: en *clustering*, y en reducción de dimensiones. Para lograr esto último, un método muy popular que se utiliza es el análisis de componentes principales (en inglés, *Principal Component Analysis* o **PCA!**) que se basa en proyectar los datos en un espacio de dimensión menor tratando de maximizar la varianza de estos en cada una de las componentes obtenidas [**?**]. Dichas componentes resultan de aplicar una descomposición en valores singulares (SVD) a la matriz formada con los datos, y cada componente es un autovector que se caracteriza por la varianza que retiene de la proyección mediante su correspondiente autovalor. Por lo tanto, para lograr la reducción de dimensiones se toman las componentes que mayor varianza producen (ordenadas por autovalor) y además se puede conocer la proporción de varianza que retuvo la reducción mediante la proporción total de autovalores retenidos respecto al total de la proyección. También se utiliza para extracción de características sobre un conjunto de datos, ya que las componentes principales se pueden como los vectores que mayor información proveen sobre dichos datos y por ende aportan mayor discriminación para otros algoritmos clasificadores o de regresión [**?**]. Cuando se aplica *whitening* a la salida del PCA, cada una de las componentes es escalada al dividir sus dimensiones por el respectivo autovalor, y al resultado de esto se lo suele denominar ZCA [**?**].

En los algoritmos de aprendizaje profundo, el enfoque no supervisado es frecuentemente considerado crucial para obtener un buen desempeño con las redes neuronales entrenadas. Esto se debe a que puede ayudar a lograr la generalización buscada en la red, ya que gran parte de la información que definen sus parámetros provienen de modelar los datos de entrada. Luego la información de las etiquetas puede ser usada para ajustar los parámetros obtenidos, los cuales ya descubrieron las características importantes de forma no supervisada.

La ventaja del pre-entrenamiento no supervisado como regularizador respecto a una inicialización aleatoria de parámetros ha sido claramente demostrada en distintas comparaciones estadísticas [**?**] [**?**] [**?**]. De esta forma, las redes pueden aprender a extraer características por sí solas, por lo cual sus entradas suelen ser datos crudos sin mucho pre-procesamiento, y la idea de inyectar una señal de entrenamiento no supervisado por cada capa puede ayudar a guiar a sus respectivos parámetros hacia mejores regiones en el espacio de búsqueda [**?**

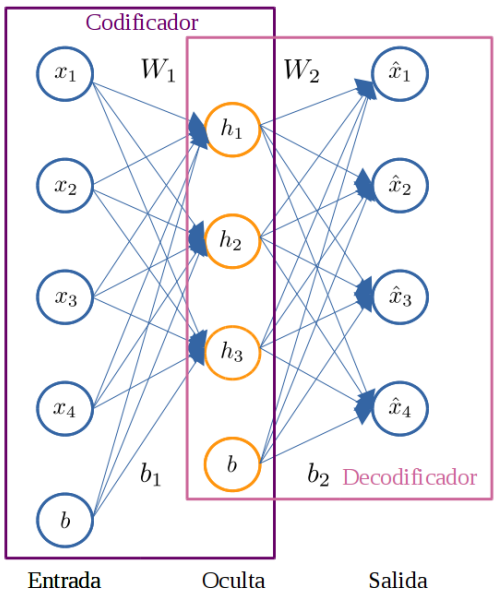


Figura 2.7: Arquitectura básica de un autocodificador.

].

Autocodificadores

Un autoasociador o autocodificador (en inglés, conocido como *AutoEncoder* o **AE!**) es un tipo de red neuronal de tres capas, donde su entrada y salida tienen igual dimensión y se fuerza a que sean iguales (es decir, que la red aprenda a reconstruir la entrada en la salida). Esto constituye un esquema no supervisado ya que se trata de aproximar la función identidad (i.e. y = x), pero además se imponen ciertas restricciones en la configuración que permiten capturar una estructura de los datos que la ajustan. Estas restricciones se hacen sobre términos que penalicen la red (e.g. normas de regularización) y sobre la dimensión de la capa oculta, que por lo general se dispone que sea distinta a la de entrada.

Un autocodificador entonces consiste de dos partes: el codificador, que produce la transformación de la entrada en la dimensión dada por la capa oculta, y el decodificador que vuelve a reconstruir la entrada a partir de la representación codificada. Para lograr la reconstrucción, esta red se entrena mediante retropropagación para minimizar el error de reconstrucción (generalmente medido con MSE), por lo cual supone un sistema de regresión. En la Figura 2.7 se puede apreciar la arquitectura de un autocodificador tal como fue detallada.

Una aplicación muy estudiada de los autocodificadores consiste en la reducción de dimensiones sobre un conjunto de datos. En comparación con PCA, los autocodificadores se asemejan a dicho método cuando la dimensión de salida en la red es menor a la de entrada, pero se diferencian en que la transformación producida es no lineal, lo cual en muchos estudios produce mejores representaciones de los datos a reducir [**?**].

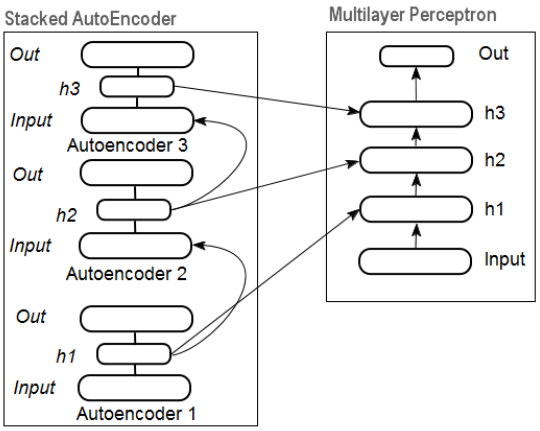


Figura 2.8: Construcción de un autocodificador apilado.

Existen ciertas formas de extender el diseño de un autocodificador para asegurar que capture una representación útil de la entrada. Una es agregar "raleza" (en inglés, *sparsity*) que significa forzar a que muchas unidades ocultas sean iguales o cercanas a cero, lo cual ha sido explotado en muchas aplicaciones exitosas [**?**]. Otra forma es agregar aleatoriedad en la codificación de la entrada a reconstruir, como en los *denoising autoencoders* que adicionan ruido a la entrada para que la red aprenda a anularlo o limpiarlo en la salida [**?**]. También existe un enfoque distinto definido por *variational autoencoders*, en el que la representación latente aprendida compone un modelo generativo con el cual se puede realizar un muestreo a partir de una entrada dada [**?**].

Autocodificadores apilados

Una vez que el autocodificador se entrenó de forma no supervisada, se pueden utilizar las características que aprendió (i.e. la codificación de la entrada) para realizar una tarea supervisada de regresión o clasificación. En ese caso, suele resultar conveniente ajustar la red ya entrenada utilizando patrones etiquetados de datos para mejorar el desempeño en dicha tarea. De esta forma, el entrenamiento de una red neuronal se puede componer en dos etapas:

a) Un *pre-entrenamiento* de forma no supervisada, para extraer características de los datos y obtener una representación codificada de ellos.

b) Un *ajuste fino* de forma supervisada, para modificar los parámetros de forma que mejore la tarea asignada a la red en base a ello.

Para extraer distintos niveles de representación sobre los datos, los AEs son combinados en otro tipo de red denominada "autocodificador apilado" (más conocida en inglés como *Stacked AutoEncoder* o **SAE!**). A partir de ello es que se pueden construir redes neuronales profundas, compuestas de múltiples capas para extraer características de distintos niveles sobre los datos.

Dado un autocodificador ya entrenado, se puede apilar éste con otro para conformar un autocodificador apilado. No obstante, no se utiliza en su totalidad

sino que se aprovecha sólo la parte de codificación. Es decir que la activación producida en la capa oculta de un autocodificador (i.e. las características detectadas) alimentan la entrada del autocodificador siguiente que es agregado a la pila, como se esquematiza en la Figura 2.8. Esto quiere decir que cada AE de esta pila trata de reconstruir la salida producida por el AE precedente, y a partir de ello las representaciones de los datos adquieren distintos niveles a lo largo de esta estructura. A este proceso de entrenar un autocodificador a partir del otro se lo suele denominar en inglés como *greedy layer-wise*, y se considera crucial para pre-entrenar redes profundas de forma no supervisada asegurando que cada nivel de las mismas reciba actualizaciones adecuadas durante la optimización [**?**].

Una vez ejecutado el pre-entrenamiento de un SAE, se puede realizar el ajuste fino mencionado con datos etiquetados en forma supervisada. Esto equivale a inicializar los parámetros de un perceptrón multicapa en forma no supervisada, lo cual mejora importantemente el desempeño del modelo en muchas aplicaciones estudiadas respecto a la inicialización estándar [**?**]. Con ello se procede a ajustar el modelo para una tarea en particular, y la combinación de estas dos etapas es muy explotada en diversas aplicaciones de aprendizaje profundo para obtener modelos con buen desempeño en tareas de gran complejidad.

Capítulo 3

Cómputo distribuido

Divide las dificultades que examinas en
tantas partes como sea posible
para su mejor solución.

René Descartes

Resumen: En este capítulo se detalla todo el contenido del proyecto relacionado a sus propiedades de distribución del cómputo. Se introduce la noción de un sistema distribuido, sus antecedentes desde el origen del concepto hasta las tecnologías actuales, y además se profundiza sobre la tecnología utilizada en este proyecto para adquirir esta propiedad de cómputo. Por último se muestran aplicaciones de ello en aprendizaje profundo, detallando el esquema que utilizan para explotar dicha propiedad.

Introducción

La computación distribuida es un modelo para resolver problemas de cómputo en paralelo mediante una colección de ordenadores pertenecientes a distintos dominios de administración, sobre una red distribuida. En ello interviene un sistema distribuido, cuya idea fundamental constituye una combinación de computadoras y sistemas de transmisión de mensajes bajo un solo punto de vista lógico, a través del cual los elementos de cómputo resuelven tareas en forma colaborativa. Dicho sistema es visto por los usuarios como una única entidad capaz de proporcionar facilidades de computación. Por lo tanto el programador accede a los componentes de software remotos de la misma manera en que accedería a componentes locales, lo cual se posibilita mediante un grupo de computadoras que utilizan un *middleware* para conseguir la comunicación [**?**]. Esta colección de computadoras básicamente lo que hace es dividirse el trabajo a realizar en pequeñas tareas individuales: cada una recibe los datos necesarios para su ejecución, y al ser realizadas se unifican sus salidas en un resultado final. A su vez, el sistema distribuido se caracteriza por su heterogeneidad, por lo que cada computadora posee sus componentes de software y hardware que el usuario percibe como un solo sistema.

Actualmente la informática contribuye en gran medida a la solución de problemas en diferentes ámbitos y disciplinas, volviéndose una fuente de recursos imprescindible. Con ello surge la creciente necesidad de almacenamiento y procesamiento de datos que se requiere en ambiciosos proyectos de investigación científica, así como simulaciones a gran escala y la toma de decisiones a partir de grandes volúmenes de información, donde es conveniente recurrir a sistemas distribuidos. Los requisitos de dichas aplicaciones incluyen un alto nivel de fiabilidad, seguridad contra interferencias externas y privacidad de la información que el sistema mantiene [**?**].

Características

Frecuentemente, se suele confundir el cómputo paralelo con el distribuido por lo cual es conveniente realizar las debidas distinciones. El concepto de *paralelismo* es generalmente percibido como explotar simultáneamente múltiples hilos de ejecución o procesadores de forma interna, para poder computar un determinado resultado lo más rápido posible. La escala de los procesadores puede ir desde múltiples unidades aritméticas dentro de un procesador único, a múltiples procesadores compartiendo memoria, hasta la distribución del cómputo en muchas computadoras. En cambio, el cómputo distribuido estudia procesadores separados que se comunican por enlaces de red. Mientras que los modelos de procesamiento en paralelo a menudo asumen memoria compartida, los sistemas distribuidos se basan fundamentalmente en el intercambio de mensajes. Las características más destacadas de los sistemas distribuidos son [**?**] [**?**]:

- **Recursos compartidos**

  Los recursos en un sistema distribuido (e.g. discos, bases de datos, etc) están físicamente alojados en algún ordenador y sólo pueden ser accedidos por otros mediante las comunicaciones en la red. Para que compartan recursos efectivamente, debe existir un gestor de recursos que actúe como interfaz de comunicación permitiendo que cada recurso sea accedido, manipulado y actualizado de una manera fiable y consistente.

- **Apertura**

  Un sistema informático es abierto si puede ser extendido de diversas maneras respecto a hardware (e.g. añadir periféricos, memoria o discos, interfaces de comunicación, etc.) o software (e.g. añadir características al sistema operativo, protocolos de comunicación, programas o entornos virtuales, etc.). La apertura de los sistemas distribuidos se determina primariamente por el grado en el que nuevos servicios para compartir recursos se pueden añadir sin perjudicar ni duplicar a los ya existentes.

- **Concurrencia**

  Dos o más procesos son concurrentes o paralelos cuando son procesados al mismo tiempo, por lo que para ejecutar uno de ellos no hace falta que se haya terminado la ejecución del otro. En sistemas multiprocesador podría conseguirse asignando un procesador a cada proceso, mientras que cuando existe sólo un solo procesador se producirá un intercalado de las instrucciones de los procesos, en forma de lograr la sensación de que hay un paralelismo en el sistema. En un sistema distribuido que está basado en

el modelo de compartición de recursos, la posibilidad de ejecución paralela ocurre por dos razones:

a) Muchos usuarios interactuan simultáneamente con programas de aplicación, lo cual es menos conflictivo ya que normalmente las aplicaciones interactivas se ejecutan aisladamente en la estación de trabajo de cada usuario en particular.

b) Muchos procesos servidores se ejecutan concurrentemente, cada uno respondiendo a diferentes peticiones de los procesos clientes. Dichas peticiones para acceder a recursos pueden ser encoladas en el servidor y procesarse secuencialmente o bien pueden tratarse concurrentemente por múltiples instancias del proceso gestor de recursos. En ese caso se debe asegurar la sincronización de las acciones para evitar conflictos, lo cual es importante en el sistema para no perder los beneficios de la concurrencia.

- **Escalabilidad**

  Los sistemas distribuidos operan de manera efectiva y eficiente a muchas escalas diferentes. La escala más pequeña consiste en dos estaciones de trabajo y un servidor de ficheros, mientras que un sistema distribuido construido alrededor de una red de área local simple podría contener varios cientos de estaciones de trabajo, varios servidores de ficheros, servidores de impresión y otros servidores de propósito especifico. A menudo se conectan varias redes de área local para formar *internetworks* o redes de Internet que contengan miles de ordenadores formando un único sistema distribuido, y permitiendo que los recursos sean compartidos entre todos ellos. Tanto el software de sistema como el de aplicación no deberían cambiar cuando la escala del sistema se incrementa. La necesidad de escalabilidad no es sólo un problema de prestaciones respecto a la red o hardware, sino que esta íntimamente ligada con el diseño y arquitectura de los sistemas distribuidos.

- **Tolerancia a fallos**

  Cuando se producen fallos en el software o en el hardware, los programas podrían producir resultados incorrectos o bien pararse antes de terminar el cómputo que estaban realizando. El diseño de sistemas tolerantes a fallos se basa en dos cuestiones, complementarias entre sí:

  a) *Redundancia del hardware*: mediante el uso de componentes redundantes, como replicando los servidores individuales que son esenciales para la operación continuada de aplicaciones críticas.

  b) *Recuperación del software*: que tiene relación con el diseño para que sea capaz de recuperar (roll-back) el estado de los datos permanentes antes de que se produjera el fallo.

  Los sistemas distribuidos también proveen un alto grado de disponibilidad en la vertiente de fallos hardware. Un fallo simple en una máquina multiusuario resulta en la no disponibilidad del sistema para todos los usuarios, mientras que el fallo de algún componente de un sistema distribuido sólo afecta al trabajo que estaba realizando dicho componente averiado, con lo cual un usuario podría desplazarse a otra estación de trabajo o un proceso servidor podría ejecutarse en otra máquina.

- **Transparencia** Se basa en ocultar al usuario y al programador la estructuración de los componentes del sistema distribuido, de manera que el mismo se percibe como un todo en lugar de una colección de componentes independientes. La transparencia ejerce una gran influencia en el diseño del software de sistema y además provee un grado similar de anonimato en los recursos al que se encuentra en los sistemas centralizados.

Infraestructura

Para poder implementar un sistema distribuido, se debe contar con una adecuada organización física de recursos o "infraestructura". Esto es indispensable para que las aplicaciones tengan un buen desempeño, ya que para poder explotar sus propiedades de cómputo dependen del soporte que tengan para ello. La elección del tipo de infraestructura queda determinada por factores como el tipo de sistemas que debe manejar, los costos de implementación (e.g. hardware, redes/comunicaciones, integración de software, etc) y los riesgos implicados en términos de seguridad, complejidad de mantenimiento, etc. Actualmente se distinguen las siguientes categorías: grilla, clúster y nube.

Tabla 3.1: Grilla vs clúster vs nube.

Categoría	**Grilla**	**HPC/Clúster**	**Nube**
Tamaño	Grande	Pequeño a mediano	Pequeño a grande
Tipo de recursos	Heterogéneos	Homogéneos	Heterogéneos
Inversión inicial	Alta	Muy alta	Muy baja
ROI típico	Intermedio	Muy alto	Alto
Tipo de red	Privada basada en Ethernet	Privada IB o propietaria	Pública de Internet basada en Ethernet
Hardware típico	Caro	Muy caro Alta gama	VMs encima del hardware
Denominación	"Estaciones de trabajo rápidas"	"Súper computadora"	"Puñado de VMs"
Requisitos de seguridad	Altos	Muy bajos	Bajos

Un clúster consiste en un conjunto de nodos o computadoras interconectadas mediante redes locales de alta velocidad, que actúan concurrentemente en conjunto para ejecutar las tareas asignadas por un programa determinado. Además se caracterizan por mostrarse ante clientes y aplicaciones como un solo sistema.

La computación por grilla es la segregación de recursos provenientes de múltiples sitios, generalmente conjuntos de clústeres heterogéneos y dispersos geográficamente, que son manejados conjuntamente para resolver un problema que no puede tratarse con una única computadora o servidor [**?**].

La "nube" es un nuevo paradigma de computación en el cual se provee una gran pila de recursos virtuales y dinámicamente escalables a demanda. El principio fundamental de este modelo es ofrecer cómputo, almacenamiento y software como servicio, donde mediante Internet un usuario paga sólo por la cantidad de recursos solicitados y su tiempo de uso.

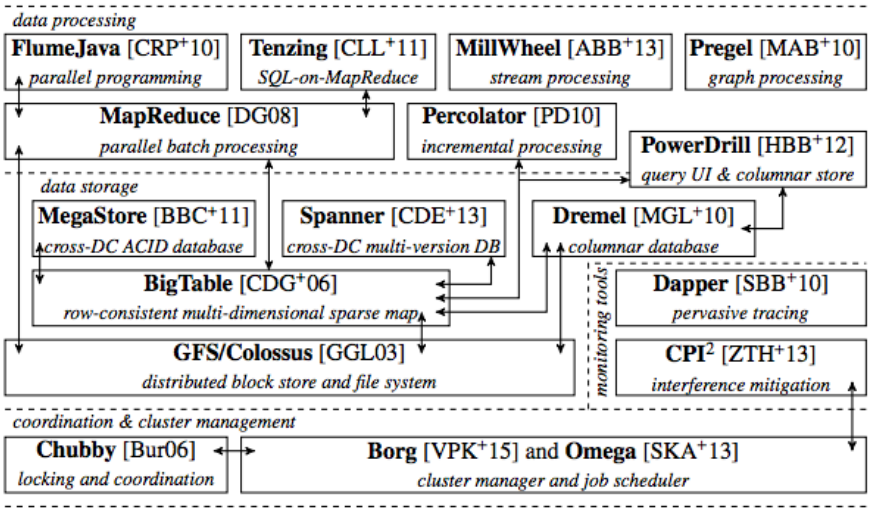


Figura 3.1: Pila de producción usada por Google aprox. en 2015.

En la Tabla 3.1 [1] se realiza una comparación breve entre las categorías tratadas, lo cual resulta útil en la elección de una infraestructura a implementar en una organización.

Antecedentes

El desarrollo de los sistemas distribuidos vino de la mano de las redes locales de alta velocidad a principios de 1970. Más recientemente, la disponibilidad de computadoras personales de altas prestaciones, estaciones de trabajo y ordenadores servidores ha resultado en un mayor desplazamiento hacia los sistemas distribuidos en detrimento de los ordenadores centralizados multiusuario. Esta tendencia se ha acelerado por el desarrollo de software para sistemas distribuidos, diseñado para soportar el desarrollo de aplicaciones distribuidas. Dicho software permite a los ordenadores coordinar sus actividades y compartir los recursos del sistema – hardware, software y datos.

La creciente necesidad de almacenamiento y procesamiento de datos que se requiere en ambiciosos proyectos de investigación científica, así como simulaciones a gran escala y la toma de decisiones a partir de grandes volúmenes de información es claramente un problema a tener en cuenta. Es por ello que en los últimos años existe una gran tendencia a originar proyectos de software que ofrezcan soluciones escalables con buenas propiedades de cómputo distribuido, los cuales son imprescindibles para cualquier organización que maneja una gran cantidad de información en sus sistemas.

En la Figura 3.1 se puede apreciar la gama de tecnologías que aprovecha Google para conformar su pila de producción con la que ofrece soluciones relacionadas a procesamiento de datos. Dicha pila comprende tecnologías que utili-

[1] Adaptación de tabla original en `http://www.devx.com/architect/Article/45576`

zan cómputo distribuido para desempeñar sus respectivas tareas, con lo cual se asegura un sistema escalable para el tratamiento de los datos.

MapReduce

MapReduce es una técnica o modelo de programación para procesar grandes cantidades de datos simultáneamente sobre muchos núcleos. El nombre se debe a que implementa dos métodos derivados de la programación funcional: el *map* que realiza el mismo cómputo o función a cada elemento de una lista, produciendo una nueva lista de valores; el *reduce* que aplica una función de agregación sobre una lista de valores, combinándolos en un único resultado. Estas funciones pueden entenderse con un simple ejemplo de conteo de palabras en un texto: *map* transforma cada una de las palabras en tuplas con la palabra correspondiente y un valor 1, y *reduce* agrega esa lista de tuplas al sumar el segundo campo de cada tupla asociando por palabras iguales, para así lograr el conteo final de cada palabra.

Jeff Dean de Google introdujo el método en una publicación del 2004 [**?**], y Doug Cutting implementó una estructura similar un año después en Yahoo (el cual eventualmente se volvería Apache Hadoop). Conceptualmente, existen enfoques similares que han sido muy conocidos desde 1995 con el estándar Message Passing Interface (MPI) teniendo operaciones de *reduce* y *scatter* [**?**].

MapReduce como sistema (también llamado "infraestructura" o "framework") organiza el procesamiento ordenando los servidores distribuidos, corriendo las distintas tareas en paralelo, gestionando todas las comunicaciones y las transferencias de datos entre las diferentes partes del sistema, y proporcionando redundancia y tolerancia a fallos.

MapReduce opera en una larga escala. La operación *map* parte un gran trabajo mediante la distribución de los datos en muchos núcleos, y corre la misma operación/es en esos fragmentos de datos. En cuanto a la función *reduce*, consolida todos esos fragmentos transformados en un solo conjunto, recolectando todo el trabajo en un lugar y aplicando una operación adicional.

Las principales contribuciones del framework MapReduce no son las funciones *map* y *reduce*, sino la escalabilidad y la tolerancia a fallos lograda para una variedad de aplicaciones optimizando el motor de ejecución una vez. Como tal, una implementación sin paralelización de MapReduce no va a ser más rápida que una tradicional sin este esquema, ya que cualquier beneficio es usualmente sólo percibido en implementaciones con múltiples hilos de ejecución.

El uso de este modelo beneficia únicamente cuando entran en juego dos características: la función optimizada de *shuffle* (i.e. mezclar los fragmentos de datos distribuidos, para reducir costo de comunicación en la red) y la tolerancia a fallos en caso de que se caigan nodos de cómputo. Optimizar el costo de comunicación es esencial para tener un buen algoritmo de MapReduce.

Las bibliotecas de MapReduce han sido escritas en muchos lenguajes de programación y con diferentes niveles de optimización. Una implementación popular de código abierto que tiene soporte para *shuffles* distribuidos es parte de Apache Hadoop, y MapReduce se considera el corazón de este software ya que le permite escalar masivamente a lo largo de una gran cantidad de servidores en un clúster Hadoop.

Apache Hadoop

Hadoop es un framework o infraestructura digital de desarrollo creado en código abierto bajo licencia Apache, utilizado para escribir fácilmente aplicaciones que procesan grandes cantidades de datos en forma paralela sobre clústeres de servidores básicos. Está diseñado para extender un sistema de servidor único a miles de máquinas, con un muy alto grado de tolerancia a las fallas. En lugar de depender del hardware de alta gama, la fortaleza de estos clústeres se debe a la capacidad que tiene el software para detectar y manejar fallas al nivel de las aplicaciones [**?**].

El proyecto fue desarrollado por Doug Cutting mientras estaba en Yahoo! (empresa que mayor contribuyó a Hadoop), inspirándose principalmente en tecnologías liberadas por Google, concretamente MapReduce y Google File System (GFS). Un trabajo en MapReduce usualmente parte un conjunto de datos en fragmentos independientes que son procesados por funciones de *map* en forma paralela. El framework ordena las salidas de cada *map*, que pasan a ser la entrada de las tareas de *reduce*. A su vez, el framework asume la planificación de las tareas, su monitoreo y re-ejecución de aquellas que fallen.

Los dos pilares más importantes que estructuran la plataforma son:

- **YARN** - Yet Another Resource Negotiator (YARN) que asigna CPU, memoria y almacenamiento a las aplicaciones que se ejecutan en un clúster Hadoop organizando sus nodos disponibles. YARN permite que otros marcos de aplicaciones (como Apache Spark) también puedan ejecutarse en Hadoop, lo cual agrega potencia y flexibilidad a la plataforma.
- **HDFS** - Hadoop Distributed File System (HDFS) es un sistema de archivos que abarca todos los nodos de un clúster Hadoop para el almacenamiento de datos. Enlaza entre sí los sistemas de archivos de muchos nodos locales para convertirlos en un único gran sistema de archivos.

La principal característica que se destaca en Hadoop es que cambia la economía y la dinámica de la computación a gran escala, y su impacto puede sintetizarse en cuatro características [2]:

- **Redimensionable:** Pueden agregarse tantos nuevos nodos como sea necesario, sin tener que cambiar el formato de los datos ni la forma en que se cargan los datos o en que se escriben las aplicaciones que están encima.
- **Rentable:** Hadoop incorpora masivamente la computación paralela a los servidores básicos, con lo cual se obtiene una marcada reducción del costo de almacenamiento, que a su vez abarata el modelado de datos.
- **Flexible:** Hadoop funciona sin esquema y puede absorber cualquier tipo de datos, estructurados o no, provenientes de un número cualquiera de fuentes. Los datos de diversas fuentes pueden agruparse de manera arbitraria y así permitir análisis más profundos que los proporcionados por cualquier otro sistema.
- **Tolerancia a fallas:** Si se pierde un nodo, el sistema redirige el trabajo a otra localización de los datos y continúa procesando sin perder el ritmo.

[2]Explicación provista por IBM, otro gran contribuyente a Hadoop: `https://www-01.ibm.com/software/cl/data/infosphere/hadoop/que-es.html`

Apache Spark

Apache Spark es una plataforma de cómputo escalable que se ha vuelto una de las herramientas más utilizadas en sistemas distribuidos, siendo construida por programadores de casi 200 empresas (principalmente Databricks, Yahoo! e Intel) y teniendo contribuciones de más de 1000 desarrolladores.

Spark comenzó en 2009 como un proyecto de investigación del AMPLab [3] en la Universidad de California en Berkeley. Los investigadores de dicho laboratorio habían estado trabajando con Hadoop MapReduce, y observaron que era ineficiente para aplicaciones de múltiples pasos que requieren distribuir datos con baja latencia sobre operaciones paralelas. Estas aplicaciones son muy comunes en sistemas para análisis de datos, donde se incluyen algoritmos iterativos, usos de aprendizaje maquinal, y minería de datos interactiva, entre otras.

Es por ello que, desde el inicio, Spark fue diseñado para ser rápido en consultas interactivas y algoritmos iterativos, mediante características como una eficiente recuperación de fallos y soporte para almacenamiento y procesamiento en memoria. En Marzo del 2010, Spark se convirtió en un proyecto de código abierto, y fue transferido a la Fundación de Software Apache en Junio del 2013, donde se continúa actualmente como uno de los proyectos más importantes.

Spark ofrece tres beneficios principales [**?**]:

- **Fácil de usar**: Se pueden desarrollar aplicaciones en una laptop, y está hecho en el lenguaje Scala aunque también se proveen APIs de alto nivel (en lenguajes como Python o R) que permiten enfocarse en el contenido del cómputo y no en cómo distribuirlo.
- **Propósito general**: Spark es un motor de propósito general que permite combinar múltiples tipos de cómputo (e.g. consultas SQL, procesamiento de texto y aprendizaje maquinal) que por lo general requieren de diferentes motores o tecnologías.
- **Rápido**: Extiende y generaliza el popular modelo MapReduce, incluyendo consultas interactivas y procesamiento de flujo, y además ofrece la capacidad de realizar cálculos computacionales en memoria e incluso en disco de forma más eficiente y rápida que MapReduce para aplicaciones complejas.

Funcionalidades

Se puede encontrar una buena descripción de las funcionalidades de Spark en la guía de programación disponible en su sitio web [4], pero a fin de destacar las particularidades relevantes en este trabajo se describen algunos conceptos fundamentales.

A grandes rasgos, una aplicación en Spark consiste de un programa *driver* que ejecuta varias operaciones en paralelo sobre un clúster a partir de una función definida por el usuario. Para el manejo de datos se provee una abstracción denominada "conjunto de datos distribuidos resistente" o *resilient distributed dataset* (RDD), que consiste en una colección de elementos repartidos sobre los nodos de la infraestructura y que pueden ser operados en forma paralela.

[3] Sitio Web de AMPLab: `https://amplab.cs.berkeley.edu/`

[4] `https://spark.apache.org/docs/latest/programming-guide.html`

Estos RDDs tienen la particularidad de poder ser persistidos en memoria para ser eficientemente reutilizados en distintas operaciones, y además se recuperan automáticamente de cualquier fallo ocurrido en los nodos.

Operaciones en RDDs

Los RDDs soportan dos tipos de operaciones, que definen el tipo de procesamiento general que se puede realizar en Spark:

- **Transformaciones:** Dado un RDD, el resultado es otro RDD en donde cada uno de los elementos del original son alterados mediante una función determinada. En Spark todas las transformaciones se realizan de forma *lazy*, es decir que no se computa el resultado hasta no ser requerido. Esto incrementa la eficiencia del procesamiento ya que se va a retornar sólo el valor necesitado y no todos los elementos procesados previamente. Ejemplos de transformaciones son los conocidos *map* y *filter* del paradigma funcional.
- **Acciones:** Resulta en un valor singular producido por una función de agregación entre todos los elementos del RDD en cuestión. Por esta razón, la cadena de transformaciones previas se deben ejecutar para poder computar este resultado único. Ejemplos de acciones son el *reduce* que produce la agregación en base a una función definida (e.g. suma, concatenación, etc) y el *count* que resulta en una cuenta de todos los elementos de un RDD.

Por defecto, cada RDD transformado es recomputado cada vez que una acción se ejecuta sobre este. No obstante, se puede además persistir un RDD en memoria y/o disco para que Spark mantenga los elementos en los nodos de forma que se agilice el acceso a los mismos.

Variables compartidas

Otra abstracción que provee Spark es el de variables compartidas que pueden utilizarse en operaciones paralelas. Por defecto, cuando Spark ejecuta una serie de tareas en diferentes nodos, se envía una copia de cada variable usada por cada tarea en dichos nodos. Esto puede ser ineficiente en casos que las mismas se necesiten para múltiples operaciones (sobre todo si una variable posee un gran tamaño como para ser transmitida varias veces). En estos casos que una variable necesita ser compartida entre tareas y operaciones, se pueden aprovechar dos tipos de variables compartidas que Spark provee: las de *broadcast* que pueden ser usadas para mantener un elemento en memoria sobre todos los nodos, y las de *accumulator* que sirven para fines de agregación entre operaciones.

Las variables *broadcast* permiten al programa enviar un valor de sólo lectura a todos los nodos en cuestión para que puedan usarlo en una o más operaciones de Spark. Con las variables *broadcast*, se pueden transferir copias de un elemento grande (como un conjunto de datos) de forma eficiente, ya que además Spark intenta distribuir dicha variable usando algoritmos eficientes de broadcast para reducir el costo de comunicación. Es importante mencionar que el valor almacenado en la variable no puede ser modificado luego de transferirse, de forma que se asegure que todos los nodos obtengan el mismo valor para operar consistentemente [**?**].

Las variables *accumulator* proveen una sintáxis simple para agregar valores arrojados por los nodos hacia el programa en el driver. Para ello se basan en una operación que deber ser conmutativa y asociativa, la cual desempeña la acumulación o agregación de los valores eficientemente de forma paralela. Los nodos no pueden acceder a su valor ya que esta variable es de sólo escritura para ellos y únicamente puede ser accedida por el programa en el driver, lo cual permite que el esquema sea eficiente al ahorrar costos de comunicación por cada actualización que ocurra.

Integridad de tecnologías

La plataforma provee además una pila de librerías que pueden ser combinadas indistintamente sobre una misma aplicación, conforme a la característica de propósito general mencionada anteriormente. Dichas librerías actualmente son:

- **Spark SQL:** Módulo que permite trabajar con datos estructurados en forma escalable, mediante una interfaz de DataFrames que soporta consultas SQL e integración con otras tecnologías relacionadas.
- **Spark Streaming:** Soporte para procesamiento de flujos que permite escribir aplicaciones en tiempo real.
- **GraphX:** Interfaz para procesamiento de grafos de forma paralelizada, incluyendo algoritmos para minarlos de manera eficiente.
- **MLlib:** Librería de aprendizaje maquinal, que incluye los algoritmos más populares desarrollados de forma escalable para aplicaciones distribuidas.

Además ofrece soporte de forma nativa para integrarse con otras tecnologías distribuidas, como YARN y Mesos para manejo de clústeres, y Cassandra, HBase y Hive para bases de datos. Estas características le permiten ser fácilmente acoplable con cualquier aplicación de cómputo distribuido, y es por ello que la mayor parte de grandes organizaciones contienen a Spark en su pila de producción para manejo de datos.

Dado que actualmente se considera a Spark como una de las más importantes tecnologías para procesamiento de datos, en el presente proyecto se consideró adecuado integrarla para lograr una aplicación de aprendizaje profundo que soporte cómputo distribuido, y que además constituya un producto de software avalado por una tecnología novedosa y popular.

Aplicaciones en aprendizaje profundo

Como se mencionó anteriormente en la Sección 2.3, utilizando aprendizaje profundo se logran modelos más precisos que con otras técnicas de aprendizaje maquinal aunque involucrando un mayor costo computacional. Se ha observado que incrementando su escala, en cuanto a los datos de entrenamiento y/o número de párametros del modelo, se puede mejorar drásticamente el desempeño en las tareas asignadas. Estos resultados han despertado el interés en intensificar el entrenamiento de los modelos así como enriquecer los algoritmos y procedimientos utilizados para optimizarlos [**?**].

Actualmente, la mayor parte de las redes neuronales profundas son entrenadas usando GPUs debido a la enorma cantidad de cómputos en paralelo que realizan. Sin estas mejoras de desempeño, las redes profundas podrían tomar días o incluso semanas en entrenarse sobre una sóla máquina. No obstante, usar GPUs puede ser inconveniente por las siguiente razones:

- Son costosas, tanto en la compra como en el alquiler.
- Muchas de ellas sólo pueden mantener en memoria una relativamente pequeña porción de datos.
- La transferencia de CPU a GPU es muy lenta, lo cual en algunas aplicaciones puede hasta contrarrestar la mejora que provee usar GPUs.

Es por ello que en los últimos años se están desarrollando herramientas relacionadas a aprendizaje maquinal en general con soporte para distribuir el cómputo involucrado sobre una infraestructura como un clúster. En este campo, se destacan las siguientes formas de implementar el paralelismo:

a) Paralelismo de las tareas: Aquí se cubre la ejecución de programas a lo largo de múltiples hilos en una o más máquinas. Se enfoca en ejecutar diferentes operaciones en paralelo para utilizar completamente los recursos de cómputo disponibles en forma de procesadores y memoria.

b) Paralelismo de los datos: Aquí el foco es distribuir los conjuntos de datos a lo largo de múltiples programas computacionales, con lo cual las mismas operaciones son realizadas en diferentes procesadores sobre el mismo conjunto de datos distribuido.

c) Híbrido: En un mismo programa se pueden implementar los dos paradigmas anteriores, donde lo que se busca es ejecutar concurrentemente distintas tareas sobre datos que se encuentran a su vez distribuidos en una infraestructura

Específicamente, estas maneras de utilizar el paralelismo son aplicadas en el aprendizaje profundo para llevar a cabo principalmente dos tipos de tareas: el procesamiento de los datos con los cuales se realiza el modelado, y la optimización de los parámetros que definen el modelo en sí.

Procesamiento de datos

Cuando se manejan datos de gran dimensión (sobre todo en problemas relacionados a Big Data), es crucial que el pre-procesamiento de los datos a utilizar en el modelado se realice con un soporte a la gran magnitud tratada. Por ello resulta factible que dicha tarea se ejecute de forma paralelizada, sobre todo si los datos son tan grandes que no pueden almacenarse en una única computadora y se deben distribuir en una red de ellas.

Dado que actualmente las aplicaciones en las que se suele recurrir a utilizar aprendizaje maquinal (principalmente *deep learning*) comprenden datos con esta propiedad mencionada (e.g. grandes conjuntos de imágenes con alta resolución, muestras largas de señales de voz con alta frecuencia de muestreo, etc), es muy

común que la etapa de pre-procesamiento previa al modelado se agilice mediante cómputo paralelo/distribuido.

Existen algoritmos utilizados para pre-procesar datos que tienen una versión paralelizada y/o escalable, desde algo básico como el proceso de estructurar y normalizar los datos o como un análisis estadístico, hasta algoritmos más sofisticados para reducir dimensiones o bien un *clustering* para el etiquetado.

Optimización de modelos

Dado que una etapa crucial en la construcción de redes neuronales profundas es su optimización, resulta factible que el poder computacional se concentre en mejorar dicho proceso. Por lo general, para ello se utilizan algoritmos basados en actualizaciones de gradientes, y ya que principalmente el más utilizado es el gradiente descendiente estocástico (SGD), que ya fue explicado en la Sección 2.1.2, existen diversos trabajos enfocados en ofrecer una versión paralelizada de este procedimiento cuya naturaleza es iterativa. A continuación, se detallan breventes algunos de ellos:

Downpour SGD

Una variante asíncrona del SGD es propuesta en el algoritmo *Downpour SGD*, el cual es usado en el framework *DistBelief* desarrollado por Google [**?**]. El mismo ajusta múltiples réplicas de un modelo en paralelo sobre subconjuntos obtenidos del conjunto de entrenamiento. Estos modelos envían sus actualizaciones a un servidor de parámetros, que también está distribuido en distintas máquinas. Cada máquina es responsable de almacenar y actualizar una fracción de los parámetros del modelo final. No obstante, dado que las réplicas no se comunican entre sí (i.e. compartiendo pesos sinápticos o actualizaciones), sus parámetros están continuamente en riesgo de diverger, dificultando la convergencia de la solución.

TensorFlow, un reciente framework de código abierto desarrollado por Google que sirve para implementar y desplegar modelos de aprendizaje maquinal en larga escala, está basado en la experiencia de desarrollar *DistBelief* por lo que internamente utiliza *Downpour SGD* para incluir procesamiento distribuido.

Hogwild!

Una solución llamada HOGWILD! [**?**] presenta un esquema que permite desempeñar las actualizaciones de parámetros con SGD de forma paralela en CPUs. Básicamente, dicho algoritmo sigue un esquema de memoria compartida donde por cada iteración del SGD todos los nodos disponibles utilizan subconjuntos separados de un conjunto de datos para entrenar modelos independientes. Estos últimos luego contribuyen a la actualización del gradiente de forma asíncrona, donde dichas contribuciones son promediadas para ser incorporadas al gradiente general. Además, los nodos tienen permitido acceder a la memoria compartida sin bloquear los parámetros (i.e. pueden hacerse lecturas y escrituras simultáneamente sin interrumpir el trabajo de los demás nodos). No obstante, esto sólo funciona si los datos de entrada se estructuran de forma "rala", tal que cada actualización sólo modifique una fracción de todos los parámetros. Este algoritmo es actualmente implementado por H2O para el entrenamiento de sus redes neuronales [**?**].

Iterative MapReduce

Mientras que una simple pasada de MapReduce se desempeña bien para

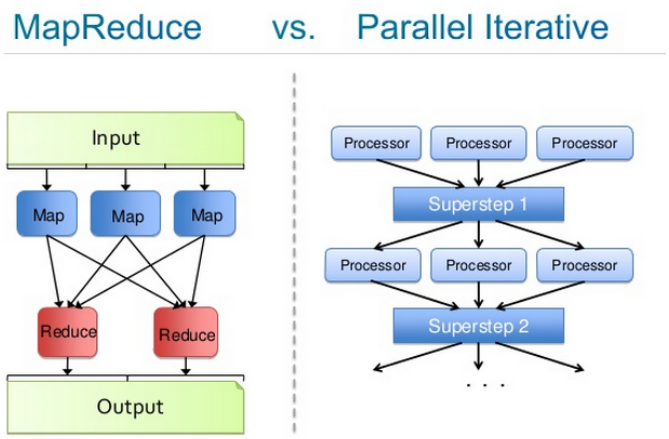


Figura 3.2: Comparación de esquema convencional MapReduce con su versión iterativa implementada en Iterative MapReduce.

muchos casos de uso, es insuficiente para utilizarse en aprendizaje maquinal y aprendizaje profundo, ya que por naturalzeza requieren métodos iterativos dado que un modelo aprende mediante un algoritmo de optimización que lo lleva a un punto de error mínimo a través de muchos pasos.

Un método propuesto para lidiar con esto, que es utilizado en el framework Deeplearning4j, se llama Iterative MapReduce [5] y puede entenderse como una secuencia de operaciones *map-reduce* con múltiples pasadas sobre los datos, donde la salida de una tarea MapReduce se vuelve la entrada de una consecuente tarea MapReduce y así sucesivamente.

Para el caso de una red neuronal que se debe entrenar con un conjunto de datos, la función *Map* ubica todas las operaciones en cada nodo del sistema distribuido, y así reparte los "batches" del conjunto de entrada sobre estos nodos. En cada uno de ellos, un modelo es entrenado con la correspondiente entrada que recibe, y finalmente la función *Reduce* toma todos estos modelos y promedia sus parámetros, agregando todo en un nuevo modelo que envía hacia cada nodo para la próxima iteración. Este procedimiento iterativo se hace tantas veces hasta alcanzar un criterio de corte establecido sobre el error de entrenamiento. Es importante notar que este esquema implementa paralelismo tanto de las tareas (en el ajuste de los modelos instanciados en cada nodo) como de los datos (al repartir los datos sobre todos los nodos disponibles).

La Figura 3.2 [6] muestra la comparación entre un esquema convencional de MapReduce y el iterativo explicado. Notar que cada *Processor* corresponde a un modelo de red neuronal a entrenar sobre un batch de datos asignados, y cada *Superstep* implica una etapa de promediado sobre los parámetros de cada *Processor* para obtener un modelo único, que luego se redistribuye por el resto del clúster para continuar el procedimiento.

[5] Fuente: `http://deeplearning4j.org/iterativereduce.html`.
[6] Fuente: `http://www.slideshare.net/cloudera/strata-hadoop-world-2012-knitting-boar`

Parte II

Learninspy

Esta segunda parte de la tesis está dedicada a detallar todas las características del framework implementado. Se describe la arquitectura escogida para su implementación, justificando las elecciones de diseño realizadas, y las propiedades que lo caracterizan como sistema para modelar redes neuronales profundas en forma distribuida. A su vez, se detalla una evaluación realizada sobre el mismo para validar su correcto funcionamiento y argumentar la bondad de sus características respecto a los objetivos planteados.

Capítulo 4

Descripción del sistema

Inteligencia es hacer artificiales los objetos, especialmente las herramientas para hacer herramientas.

Henri Bergson

Resumen: En este capítulo se describe la estructura determinada para el sistema, explicando cada uno de sus componentes y justificando su composición. Además se presentan las características más importantes del framework, y se explica en detalle cómo se propone incorporar el procesamiento distribuido en el modelado de redes neuronales, comparando luego dicha propuesta con otras similares existentes.

El producto desarrollado en este proyecto se identifica como *Learninspy*, haciendo referencia en su nombre a las técnicas de aprendizaje profundo (o *deep learning*) en redes neuronales y al uso de la tecnología Spark con Python (así también como al apodo del autor de este framework). En la Figura 4.1 se presenta su logo, el cual quizás visualiza mejor el significado mencionado.

Una elección hecha para el desarrollo de este sistema fue realizar toda el codigo fuente en inglés. Esto se hizo con el fin de ser fiel a la terminología original de todas las técnicas, y para que sea entendible por cualquier desarrollador y no sólo los de habla hispana. No obstante, toda la documentación se encuentra en español aunque próximamente se planea mantener dos versiones de ella en ambos lenguajes.

Estructura

En este trabajo de tesis se desarrolló un software estructurado como *framework*. Este tipo de sistema provee conceptos, criterios y prácticas para enfrentar un determinado tipo de problemática en base a un enfoque dado. La idea de implementar este tipo de sistema surgió debido a la dificultad de personalizar ciertos frameworks de aprendizaje profundo existentes, planteándose además el desafío de combinar la flexibilidad para crear o modificar funcionalidades y la escalabilidad en términos computacionales.

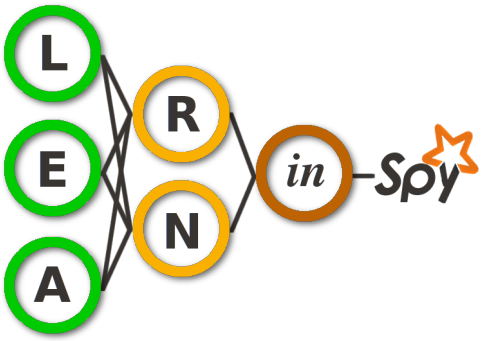


Figura 4.1: Logo del framework desarrollado.

Para ello, un framework permite las siguientes ventajas [**?**]:

- Facilita trabajar con tecnologías complejas.
- Reúne un conjunto de componentes aislados en algo mucho más útil.
- Obliga a implementar el código de una forma que promueva una programación consistente, menos bugs, y aplicaciones más flexibles.
- Cualquiera puede fácilmente testear y depurar el código, incluso si no fue quien lo escribió.

Tal como se aclaró en la Sección 1.5.1, el código fuente de Learninspy sigue un Diseño Orientado a Objetos (**DOO!**) que permite en un framework los siguientes beneficios para desarrolladores [**?**]:

- *Modularidad*, al encapsular detalles de la implementación detrás de interfaces estables y sencillas de utilizar.
- *Reutilización*, definiendo componentes genéricos que pueden ser reaplicados para crear nuevas aplicaciones. Con ello se aprovecha el dominio de conocimiento y el esfuerzo previo de desarrolladores experimentados para evitar rehacer y revalidar soluciones ya existentes.
- *Extensibilidad*, al proveer métodos acoplables que permiten a las aplicaciones extender sus interfaces estables.
- *Inversión de control*, al invertir el flujo de ejecución del programa dejando que alguna entidad lleve a cabo las acciones de control que se requieran, en el orden necesario y para todos los sucesos que deban ocurrir, en lugar de hacerlo imperativamente mediante llamadas a procedimientos o funciones.

La última propiedad mencionada refiere a que por medio del framework se explicita una solicitud concreta (e.g. entrenar una red neuronal sobre un conjunto de datos), y el mismo decide la secuencia de acciones necesarias para atenderla. En cuanto a las demás, el resto del presente capítulo muestra evidencia de su cumplimiento mediante las características que se van presentando.

Como la mayoría de los proyectos desarrollados en Python, Learninspy está compuesto por paquetes que agrupan módulos en común, y en cada uno de estos últimos se reúnen las clases (en términos de DOO) que tengan mayor relación. A continuación se describe brevemente la lógica de cada módulo y paquete, aunque para mayor detalle se debe consultar el manual de referencia [1]:

- **Core**: Como bien dice el nombre, es el módulo principal o el núcleo del framework. El mismo contiene clases relacionadas con la construcción de redes neuronales profundas, desde la configuración de los parámetros usados hasta la optimización del desempeño en las tareas asignadas. Se detallan entonces cada uno de los submódulos que lo componen:
  - *Activations*: En el mismo se implementan las funciones de activación (con su correspondiente derivada analítica) que se podrán utilizar en las capas de una red neuronal.
  - *Autoencoder*: Se extienden las clases desarrolladas en el submódulo *model*, mediante herencia de métodos y atributos, para implementar autocodificadores y su uso en forma apilada.
  - *Loss*: Provee dos funciones de error, las cuales son utilizadas en base a la tarea designada a una red neuronal: clasificación, mediante la función de *Entropía Cruzada*, y regresión, con la función de *Error Cuadrático Medio*.
  - *Model*: Es el submódulo principal de **core** ya que contiene las clases referidas directamente a redes neuronales, el diseño de sus capas y la configuración de los parámetros que manejan.
  - *Neurons*: Este submódulo contiene una clase para manejar las matrices de pesos sinápticos y los vectores de sesgo que componen las capas de una red neuronal. Dichos arreglos se implementan mediante NumPy para que se almacenen de forma local (i.e. se alojan por completo en un mismo nodo físico de ejecución), aunque se tiene pensado extender esta clase para que puedan manejarse en forma distribuida.
  - *Optimization*: Implementa los algoritmos y funcionalidades de optimización que se utilizan para mejorar iterativamente el modelado de las redes neuronales. Los algoritmos presentes han sido explicados en la Sección 2.1.2 (salvo Adagrad, ya que en su lugar se implementó Adadelta) y para la implementación fueron adaptados desde el desarrollo hecho en Climin [**?**], el cual es un framework de optimización pensado para escenarios de aprendizaje maquinal.
  - *Search*: Realizado para abarcar algoritmos de búsqueda que optimicen los parámetros de un modelo en particular. El único algoritmo desarrollado en esta versión es el de búsqueda aleatoria, que fue detallado anteriormente en la Sección 2.1.4.
  - *Stops*: Recopila distintos criterios de corte para frenar la optimización de las redes en base a una condición determinada. Al igual que el submódulo *optimization*, está basado en el trabajo hecho en Climin.

[1] Documentación de Learninspy: `http://learninspy.readthedocs.io/`

- **Utils**: Este módulo abarca todas las utilidades desarrolladas para posibilitar tanto la construcción de redes neuronales como el funcionamiento total del framework. El mismo dispone de los siguientes submódulos:
  - *Checks*: Contiene funcionalidades para comprobar la correcta implementación de las funciones de activación y de error, basándose en las instrucciones de un tutorial de aprendizaje profundo [**?**].
  - *Data*: Es el submódulo principal de **utils**, ya que posee clases útiles para construir los conjuntos de datos que alimentan las redes neuronales, y también funcionalidades para muestrearlos, etiquetarlos, partirlos y normalizarlos.
  - *Evaluation*: Se proporcionan clases para evaluar el desempeño de las redes neuronales en tareas de clasificación y regresión, mediante diversas métricas que fueron explicadas en la Sección 2.1.3.
  - *Feature*: Se implementan funcionalidades referidas a extracción de características o tipos de pre-procesamiento sobre los datos que alimentan una red neuronal. Un ejemplo de ello es el análisis de componentes principales o PCA (mencionado en la Sección 2.3.3), que fue implementado siguiendo tutoriales clásicos de deep learning [**?**] [**?**].
  - *Fileio*: Submódulo con funciones para realizar manejo de archivos y la configuración del logger de Learninspy.
  - *Plots*: Reúne todas las funcionalidades referidas a gráficas y visualizaciones (como el ajuste de una red durante el entrenamiento).

Además, a la misma altura que estos dos módulos, existe un script denominado *context* en donde se configura e instancia el contexto de Spark a utilizar en el framework. En la Sección 5.1.1 del siguiente capítulo se mencionan las configuraciones que contempla este script referidas al rendimiento de Spark.

En base al diseño planteado, se identifican dos perfiles de acceso al framework: a) de usuario, en el cual mediante conocimientos básicos de Python se puede utilizar la plataforma y añadirle algunas funcionalidades, siguiendo un paradigma de programación imperativa; b) de desarrollador, que requiere usar un paradigma de programación orientado a objetos y funcional, para la comprensión total del código mediante conocimientos de Python y Spark.

La estructura presentada se considera exhaustiva en cuanto a contenido del framework, por lo cual cualquier desarrollador que quiera modificar o agregar componentes al mismo debería poder valerse de los módulos disponibles en la arquitectura comprendida.

Características

A continuación, se detallan las particularidades de Learninspy que lo hacen un framework útil para construir redes neuronales con aprendizaje profundo sobre un conjunto de datos y en forma distribuida:

- *Diseño que permite extender funcionalidades con pocas modificaciones y sin romper el funcionamiento de otros módulos.*

Esto se relaciona con la propiedad de *extensibilidad* en un framework, mencionada en la anterior Sección 4.1. Por ejemplo, para agregar una función de activación y su derivada analítica, basta con incorporar sus definiciones en el submódulo **core.activations** y, mediante una etiqueta apropiada, adjuntarlas a los diccionarios de Python (que se encuentran al final del módulo) para utilizarlas en el framework a través del mismo. Se puede realizar un tratamiento similar para agregar tanto funciones de error como algoritmos de optimización y sus criterios de corte.

- *El paradigma orientado a objetos permite aprovechar la naturaleza del diseño de las redes neuronales, para así expresar las relaciones existentes entre las entidades manejadas.*

  Por ejemplo, la composición de una red neuronal por capas de neuronas, donde cada una de ellas tiene asociado una matriz de pesos sinápticos y un vector de sesgo, y también el hecho de que un autocodificador sea un tipo especial de red neuronal por lo que tiene una relación de herencia de métodos y atributos.

- *Mínima cantidad de dependencias en el sistema.*

  A partir del énfasis que se tuvo en esta propiedad para el diseño, no se requiere instalar más que Spark (y Java por ello) y parte del ecosistema de SciPy (que es casi un estándar en las típicas aplicaciones de Python).

- *Optimización de un modelo mediante entrenamiento de réplicas en forma concurrente y distribuida.*

  Es la característica principal de optimización que se diseñó para el sistema, y es explicada detalladamente en la siguiente Sección 4.3.

- *Los resultados del modelado pueden reproducirse de forma determinística*

  A diferencia de otras herramientas que distribuyen las operaciones de modelado, en Learninspy es posible replicar de forma exacta un experimento con una configuración dada. Esto se debe a que internamente se gestiona en forma determinística el semillero que alimenta el generador de números aleatorios, los cuales son requeridos por varios algoritmos que intervienen en el modelado (e.g. inicializador de pesos sinápticos, DropOut, etc).

- *Soporte para procesar conjuntos de datos en forma local y distribuida*

  Mediante las funciones y clases del módulo *utils.data* presentado, se brindan funcionalidades para el tratamiento de datos tanto en forma local como distribuida (utilizando RDDs de Spark para este último caso).

- *Soporte para cargar y guardar modelos entrenados.*

  El trabajo de optimización de los modelos se puede realizar de forma diferida, ya que los mismos se pueden guardar y volver a cargar en formato binario. Esto tiene gran utilidad sobre todo cuando se someten a aprendizaje no supervisado, el cual puede realizarse en muchas pasadas hasta aplicarse el ajuste fino.

Como se puede ver, algunas características están referidas al diseño del software en general y otras son más específicas del procesamiento distribuido que involucra. Por lo tanto, se describe a continuación en qué formas se logran integrar estas características en el framework.

Explotación del cómputo distribuido

Como ya se dijo anteriormente en otras secciones, las aplicaciones que suelen tratarse con aprendizaje profundo están relacionadas con datos de gran dimensión, y por ello las herramientas que realizan dicho tratamiento requieren una ventaja computacional para resultar útiles en ello. Las formas en que Learninspy aprovecha el procesamiento distribuido de Spark son las siguientes:

1. **Preparar conjuntos de datos:** El framework provee una abstracción para manejar conjuntos de datos, la cual incluye el etiquetado de los patrones por clases, la normalización y escalado de los datos, el muestreo balanceado por clases, etc. Para grandes volúmenes de datos se provee una interfaz adecuada para los RDDs de Spark, con lo cual el pre-procesamiento puede realizarse en forma distribuida.

2. **Optimizar modelos en forma paralelizada:** Siendo quizás el valor principal del procesamiento distribuido en el framework, esta característica se basa en que, por cada iteración del ajuste de una red neuronal, el modelado se realice mediante instancias replicadas que se entrenan de forma independiente y luego convergen en un modelo único, reuniendo así las actualizaciones que adquirió cada instancia por separado.

3. **Ahorrar costos de comunicación, transfiriendo conjuntos de datos a los nodos por única vez (broadcasting):** Como se explicó en la Sección 3.3.1, la funcionalidad de Broadcast que provee Spark permite que una variable muy utilizada se pueda enviar a los nodos computacionales una sóla vez (siempre que la usen únicamente en modo lectura). Esto resulta útil y eficiente con los conjuntos de datos empleados en el ajuste de las redes neuronales, el cual se hace iterativamente y de otra forma requeriría establecer una comunicación con los nodos activos por cada iteración.

4. **Configurar infraestructura fácilmente:** Mediante simples configuraciones en las variables de entorno, se puede conectar el framework forma sencilla a una estructura computacional definida con Spark (lo cual se menciona más adelante en la Sección 5.1).

Para entender cómo se obtiene la segunda característica mencionada, que se considera la más importante y tiene cierta complejidad, la siguiente sección detalla la forma en que se implementa en Learninspy.

Entrenamiento distribuido

El procedimiento para minimizar la función de costo sobre una red neuronal es una característica clave de Learninspy, ya que es una de las formas principales en que se aprovecha el cómputo distribuido en el framework. Dado que los algoritmos de optimización utilizados para realizar ello son iterativos, la paralelización propuesta busca incorporar los beneficios de la concurrencia para sacar mayor provecho al proceso en cada una de sus iteraciones.

La idea no es nueva ya que es implementada en diversos esquemas como los explicados en la Sección 3.4.2. Se basa en que el proceso de optimización de las

redes neuronales se puede paralelizar de forma tal que se obtenga una mejoría en duración y hasta resultados respecto al procedimiento convencional sin concurrencia. Para ello se tiene que, por cada iteración del proceso, un modelo base es copiado a cada nodo computacional para que cada una de estas copias o réplicas se entrene de forma independiente sobre algún subconjunto muestreado del conjunto original de datos. El hecho de optimizar en cada iteración con un subconjunto de datos (conocidos como *mini-batch*) en lugar del conjunto completo permite acelerar el proceso y está demostrado en varios estudios que aún así obtiene buenos resultados, como fue explicado en la Sección 2.1.2. Es preciso aclarar que dichos subconjuntos son obtenidos de un muestreo aleatorio sin reemplazos sobre el conjunto de entrenamiento, utilizando la función *sample* de la librería *random* que ofrece la versión usada de Python.

La cantidad de modelos replicados a entrenar en paralelo es configurable: para un mejor desempeño en términos de recursos, debe ser la cantidad de nodos/núcleos disponibles, pero también puede ser menor o mayor para tener otro impacto en los resultados. Una vez entrenadas las réplicas, se procede a mezclar los modelos de forma que converjan los aportes de la optimización en un único modelo. Para ello se emplea una "función de consenso" que toma los parámetros de cada modelo y los pondera en base al resultado de evaluación sobre los respectivos subconjuntos de datos que utilizaron.

En el Algoritmo 1 se esquematiza el procedimiento general que sigue el entrenamiento de una red neuronal en Learninspy. Notar que el mismo se estructura como una tarea MapReduce, ya que de esa forma es implementado mediante las primitivas de ese tipo que provee el motor Spark. Mediante la función *merge* se realiza el proceso de mezclado de modelos mediante una función de consenso, lo cual se explica en detalle a continuación.

Algorithm 1 Entrenamiento distribuido en Learninspy

Require: Modelo actual $h_{W,b}$.

1: **function** TRAIN(Γ, μ, ρ) ▷ Parámetros: Conjunto de entrenamiento Γ; tamaño de mini-batch μ; cantidad de modelos concurrentes o "paralelismo"ρ
2: — MAP —
3: $H_{W,b} = copy\_model(h_{W,b}, \rho)$ ▷ Realizar ρ copias de $h_{W,b}$ sobre los nodos disponibles
4: **for** $H^{(i)}_{W,b} \forall i \in \{1, \ldots, \rho\}$ **do** ▷ Bucle de ejecución concurrente
5: $\Gamma_\mu = sample(\Gamma, \mu)$ ▷ Muestreo de μ ejemplos sobre el conjunto Γ
6: $s_i = minimize(H^{(i)}_{W,b}, \Gamma_\mu)$ ▷ Optimización de modelo réplica
7: **end for**
8: — REDUCE —
9: $h_{W,b} = merge(H_{W,b}, s)$ ▷ Mezcla de modelos con función de consenso
10: $results = evaluate(h_{W,b}, \Gamma)$ ▷ Evaluación sobre el conjunto de datos
return $h_{W,b}, results$
11: **end function**

Funciones de consenso

Una vez entrenados todos los modelos replicados de forma concurrente, se deben mezclar en uno solo tratando de reunir las contribuciones de cada uno al ajuste del modelo deseado. Para ello, se puede caracterizar a cada modelo optimizado por su desempeño o *scoring* s_i que es obtenido de dos formas posibles: por una métrica aplicada en su evaluación (e.g. *accuracy* de clasificación, o R^2 de regresión), o bien por el valor resultante en la función de costo definida. El valor escogido para caracterizar cada modelo puede utilizarse como parte de una ponderación realizada sobre todos los modelos durante la mezcla, la cual consiste simplemente en una suma de los parámetros W y b de cada capa, por cada uno de los modelos correspondientemente. Para ello se propone usar una función de consenso que, en base a una ponderación establecida, logre reunir las contribuciones de los modelos para obtener un único modelo representativo. Esta mezcla consiste en una suma de los parámetros mencionados estableciendo pesos en base a una ponderación elegida, y esa suma a su vez es escalada por la sumatoria de los pesos obtenidos de la siguiente forma:

$$f(l_{W,b}, w) = \sum_i \frac{w_i l_i}{\sum w_i} \tag{4.1}$$

Si el denominador es muy cercano a 0, el mismo se reemplaza por una constante $\epsilon = 1e-3$ para evitar divisiones por 0.

Por defecto, se incluyen tres tipos de ponderación: a) constante, con los mismos pesos valiendo 1 para todos los modelos (resultando una media aritmética de cada parámetro), b) lineal, donde se utiliza en forma directa el valor de s_i, c) logarítmica, de forma que la ponderación no tenga gran variación sobre valores altos de s_i (muy buen valor en la evaluación, o bien pésimo costo de la red):

$$w_i = 1, \qquad \forall i \in \{1, \ldots, \rho\} \tag{4.2a}$$

$$w_i = s_i, \qquad \forall i \in \{1, \ldots, \rho\} \tag{4.2b}$$

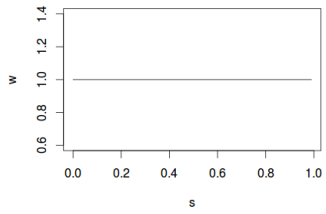
$$w_i = 1 + \ln(\text{máx}(s_i, \epsilon)), \qquad \forall i \in \{1, \ldots, \rho\}, \qquad \epsilon = 1e-3 \tag{4.2c}$$

Notar que para la ponderación logarítmica, si el dominio es menor o muy cercano a 0 se reemplaza por una constante $\epsilon = 1e-3$ para evitar conflictos con el dominio de la función logaritmo.

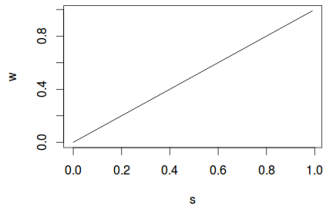
En la Figura 4.2 se representan gráficamente las funciones mencionadas, para un dominio definido en los valores del *scoring*. Para utilizar una función de consenso en particular, se debe configurar tanto la función como el *scoring* que utiliza mediante los parámetros de optimización que se definen para el modelado. Para ello, se debe instanciar un objeto OptimizerParameters del módulo *core.optimization* indicando dichos parámetros en sus argumentos (ver detalles de uso en el manual de referencia).

Criterios de corte

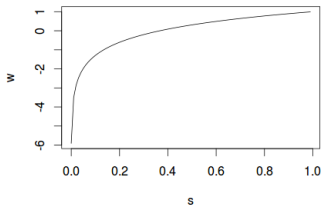
En cualquier aplicación de aprendizaje maquinal, por lo general no se ejecuta la optimización de un modelo hasta obtener un desempeño deseado ya que puede ser que no se alcance dicho objetivo por la configuración establecida. Es



(a) Sin ponderación (constante)



(b) Ponderación lineal



(c) Ponderación logarítmica

Figura 4.2: Función que describe los pesos w que ponderan a cada modelo réplica en base a su valor s, suponiendo un dominio (0, 1] para dicho valor.

por ello que, tal como se introdujo en la Sección 2.1.5, resulta conveniente establecer ciertas heurísticas para monitorear la convergencia del modelo en su optimización. Un *criterio de corte* es una función que utiliza información de la optimización de un modelo durante dicho proceso (e.g. scoring sobre el conjunto de validación, costo actual, cantidad de iteraciones realizadas) y, en base a una regla establecida, determina si debe frenarse o no. Las reglas más comunes son:

- *Máximo de iteraciones:* Se detiene la optimización luego de que el número de iteraciones sobre los datos exceda un valor máximo establecido.
- *Alcanzar un valor mínimo deseado:* Se establece una tolerancia para un determinado valor de información en la optimización (como el scoring o el costo), con lo cual el proceso se frena cuando el valor se alcanza o supera.
- *Tiempo transcurrido:* Luego de exceder un intervalo de tiempo máximo fijado (en segundos, por lo general), se detiene el proceso de optimización.

Cada una de estas reglas devuelve Verdadero si el proceso debe frenar o Falso en caso contrario. Dichos resultados booleanos pueden combinarse con operadores lógicos AND y OR para armar reglas más expresivas que configuren la optimización de una forma más específica. Por ejemplo, se puede establecer que el proceso tenga un máximo de 100 iteraciones o bien que frene si se llega a un scoring de 0.9 estableciendo un OR entre las dos primeras reglas explicadas.

La implementación de ello se encuentra en el módulo **core.stops**, y se llevó

a cabo mediante una adaptación del código provisto por Climin [2] con lo cual se puede extender el framework con nuevas reglas y criterios de corte para la optimización de los modelos.

Notar que para implementar este esquema de criterios en Learninspy, se deben especificar dos tipos de configuración: una para la optimización de los modelos réplicas a entrenar en paralelo sobre un batch de datos (llamada "optimización local"), y otra para la optimización en general del modelo final respecto a un conjunto de validación (denominada "optimización global"). En la Sección 5.3.2 del capítulo siguiente, un experimento de validación está dedicado a mostrar de forma empírica la relación entre ambos tipos de optimización.

Esquemas similares

En términos de comparación respecto a los esquemas mencionados en la Sección 3.4.2, se identifcan las siguientes ventajas del esquema propuesto en este trabajo:

- **Simplicidad**: Gracias a las primitivas que provee Spark, implementar el esquema es sencillo y requiere pocas líneas de código para lograr que la optimización sea concurrente y además escale en recursos.
- **Convergencia**: Dado que se sincronizan las actualizaciones en cada iteración mediante el mezclado, se mitiga el riesgo de divergencia en la optimización que padecen tanto Downpour SGD como HOGWILD!, convergiendo a una solución comparablemente óptima con la desarrollada por el SGD sin paralelizar.
- **Elección del algoritmo de optimización**: Ya que el esquema es independiente del algoritmo utilizado para optimizar un modelo, se pueden implementar diversos tipos de algoritmos iterativos que estén basados en gradiente como los mencionados en la Sección 2.1.2. Los mismos se pueden desarrollar en el módulo *core.optimization* del framework, donde actualmente se proveen dos algoritmos para optimizar redes neuronales.
- **Reproducibilidad de resultados**: En H2O, una plataforma que utiliza HOGWILD! para optimizar redes neuronales profundas, se debe ejecutar el entrenamiento con un único hilo de ejecución para obtener resultados replicables debido a las limitaciones del esquema. En Learninspy dicha reproducibilidad se logra independientemente del paralelismo empleado (por lo que se mencionó antes en la Sección 4.2), lo cual se ve como una ventaja muy importante a la hora de experimentar.
- **Personalización**: El mezclado de modelos no necesariamente se debe hacer promediando las contribuciones (como sucede en Iterative MapReduce y HOGWILD!), sino que se puede diseñar la función de consenso que decide cómo ponderar las mismas e incorporarla fácilmente en el framework. Por defecto se incluyen las tres funciones explicadas anteriormente.

De los tres algoritmos tratados en la comparación, se considera que el propuesto en este trabajo se asemeja mayormente al denominado Iterative MapReduce, ya que ambos incoporan la metodología de una tarea MapReduce en cada

[2] Repositorio de código: `https://github.com/BRML/climin`

iteración de la optimización en un modelo. No obstante, en Learninspy se decidió implementar un esquema propio para definir el entrenamiento distribuido de una forma que, al igual que otras características de este framework, sea flexible y entensible respecto a las funcionalidades involucradas, asegurando además la propiedad de escalabilidad buscada.

Capítulo 5

Evaluación de desempeño

Todas las cosas son buenas
o malas por comparación.
Edgar Allan Poe

Resumen: En este capítulo se describen las acciones ejecutadas para validar el correcto funcionamiento del framework implementado. Para ello, se realizan comparaciones de desempeño respecto a otras herramientas similares, y también algunos experimentos para validar ciertas funcionalidades desarrolladas en este trabajo. Se detallan los recursos utilizados y las configuraciones implementadas sobre ellos para llevar a cabo estas tareas de evaluación.

Configuraciones

La mejor forma de conocer cómo se utiliza Learninspy es siguiendo los ejemplos provistos en el directorio *examples*. Allí se muestran aplicaciones con bases de datos públicas cubriendo gran parte de las funcionalidades que se proveen.

Para poner en funcionamiento el framework, se ofrecen algunas configuraciones por defecto referidas al motor utilizado. Las mismas pueden ser modificadas desde el código fuente, o bien omitidas al inicializar una instancia de SparkContext en forma aparte a la que se crea por Learninspy. En caso de que se prefiera que el framework se encargue de eso, se debe tener en cuenta que para lograr que el mismo se conecte a un clúster en modo *standalone* se debe configurar la dirección IP del nodo maestro y el puerto a utilizar mediante las variables de entorno `SPARK_MASTER_IP` y `SPARK_MASTER_PORT` correspondientemente. Además, como en la mayor parte de las aplicaciones en Spark, se debe asegurar que el módulo *pyspark* puede importarse desde Python. Una explicación para lograr ello, en conjunto con la instalación de Spark, es provista con Learninspy en el archivo de texto *install_spark.md*. Allí se especifica cómo configurar la variable de entorno `PYTHONPATH`, de forma que incluya los módulos necesarios para el funcionamiento de PySpark (para detalles adicionales, se puede consultar la guía de programación de Spark)

Rendimiento en Spark

Para lograr un mejor desempeño de las aplicaciones en Learninspy, se utiliza una configuración por defecto en el script *context* que se introdujo en la Sección 4.1 donde se establecen cuestiones referidas a Spark y la JVM:

- **Serializador**: Cuando Spark transfiere datos entre nodos de ejecución, se necesita serializar los objectos correspondientes en formato binario. En Learninspy se configura el uso de Kryo [1](versión 2), que es significativamente más rápido y con una representación binaria hasta 10 veces más compacta que el serializador por defecto (`ObjectOutputStream` de Java), por lo cual se recomienda en la mayoría de las aplicaciones con Spark [**?**].
- **Recolector de basura en Java**: Durante el entrenamiento distribuido de Learninspy, en cada época se deshechan modelos auxiliares para lograr uno optimizado. Un modelo deshechado queda como un objeto de Java que debe tratarse correctamente por el recolector de basura para hacer eficiente el uso de memoria. Es por ello que se configura la JVM con la bandera `-XX:+UseG1GC` para elegir el recolector G1 que es conocido por tener mejor desempeño en aplicaciones de Spark [**?**]. Además, durante el entrenamiento en Learninspy se opera manualmente el recolector de basura de Python para que tampoco almacene objetos grandes que se dejan de utilizar durante cada iteración.
- En caso de disponer de menos de 32 GB de RAM, se agrega a la JVM la bandera `-XX:+UseCompressedOops` para que maneje punteros de cuatro bytes en lugar de ocho, y con ello se optimiza el manejo de memoria.

Para obtener información del desempeño de Learninspy en una aplicación, se recomienda utilizar la herramienta de monitoreo mediante interfaz web que provee Spark. Para acceder a ella, se debe ingresar desde un navegador web a:

- en modo local, a `http://localhost:4040`.
- en modo clúster, a `http://<url del master>:4040`.

Allí se presenta información acerca del entorno correspondiente, de los ejecutores de Spark y sus tareas asignadas, y un resumen de la memoria utilizada por los RDDs, entre otras. Es preciso aclarar que esta herramienta es únicamente accesible mientras se encuentra en ejecución una aplicación en Learninspy. Para más información, se puede acceder a la documentación de Spark en la sección correspondiente [2].

Para aplicaciones en modo clúster, se debe corroborar que cada uno de los nodos del mismo tenga instalado el package de Python *learninspy*, o bien se puede enviarles el mismo en forma de Python Egg (archivo .egg) mediante el script *spark-submit* de Spark. Como aclaración, en el caso de que el clúster se implemente en modo *standalone* de Spark, se puede configurar toda la implementación (e.g. nodos, memoria a usar por cada uno, etc) mediante los scripts *slaves.sh* y *spark_ env.sh* que se encuentran provistos en el directorio de Spark. Se puede encontrar una explicación detallada de todo esto en la guía de programación de Spark ya mencionada anteriormente.

[1] Repositorio: `https://github.com/EsotericSoftware/kryo`
[2] `http://spark.apache.org/docs/latest/monitoring.html`

Materiales utilizados

Se detallan a continuación los recursos disponibles para llevar a cabo las tareas de validación, los cuales se escogieron a fin de poder corroborar el desempeño esperado de todas las funcionalidades del software desarrollado.

Recursos computacionales

Para llevar a cabo los experimentos mencionados, se cuentan con tres formas de equipamiento computacional: un ordenador personal para evaluar el desempeño en forma local para estaciones de trabajo, un servidor de altas prestaciones para realizar una evaluación local más intensiva y con mayor paralelismo, y una estructura de clúster para experimentar en forma distribuida. A continuación, los detalles de cada uno:

1. **Ordenador personal:** Computadora del tipo notebook, la cual es propiedad del autor de este trabajo y posee las siguientes características:
   - Linux Mint 3.16.0-4-amd64 1 SMP Debian 3.16.7 (2016-01-17)
   - Arquitectura x86_64 (64 bits)
   - CPUs: 2 núcleos, con 2 hilos cada uno
   - Modelo: Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz
   - Memoria RAM: 6GB DDR3
   - Disco duro: SATA 750GB
2. **Servidor:** Computadora de alta gama, que actualmente constituye una estructura de clúster (como la descripta en la Sección 3.1.2) pero en esta oportunidad se utilizó de forma aislada. La misma fue provista en forma gratuita por el centro *Argentina Software Design Center* (ASDC) que conforma la compañía *Intel Security*, y se caracteriza de la siguiente manera:
   - Linux 4.4.0-38-generic #57-Ubuntu 16.04.1 LTS xenial (2016-09-06)
   - Arquitectura x86_64 (64 bits)
   - CPUs: 48 núcleos, con 2 hilos cada uno
   - Modelo: Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
   - Memoria RAM: 189GB DDR4
   - Disco duro: SATA 1300GB
3. **Clúster:** Tanto maestro como esclavos son instancias virtualizadas creadas en un servicio de "nube" denominado "Intel Cloud" (también brindado por *Intel Security*) basadas en un SO Linux 3.16.0-40-generic #54-Ubuntu 14.04.1 LTS trusty (2015-06-10) sobre una arquitectura x86_64 (64 bits). Particularmente, se distinguen por los recursos de la siguiente forma:
   - Maestro (1 instancia):
     + CPUs: 8 núcleos, con 1 hilo cada uno
     + Memoria RAM: 16GB

- Esclavos (10 instancias):
  + CPUs: 4 núcleos, con 1 hilo cada uno
  + Memoria RAM: 8GB
- **Total:**
  + CPUs: 48 núcleos
  + Memoria RAM: 96GB

Dado que el servidor posee más capacidad de cómputo y memoria que el clúster, la idea es verificar la escalabilidad del framework en este último y luego utilizar el servidor para las pruebas intensivas.

Bases de datos

Para las pruebas generales del framework (incluyendo las de *unit testing* para la integración continua del software) se utilizaron conjuntos de datos que son muy recurridos para evaluar herramientas de aprendizaje maquinal:

- *Iris*[3]: Siendo uno de los más conocidos en la literatura sobre reconocimiento de patrones, este conjunto de datos contiene 3 clases de plantas del tipo "iris" con 50 ejemplos de cada una. Una clase es linealmente separable de las otras dos, pero estas dos últimas no lo son entre sí.

  Categoría: Clasificación.

- *Combined Cycle Power Plant* (CCPP)[3]: Es un conjunto de datos reales que consta de 9568 entradas colectadas de una central térmica de ciclo combinado a lo largo de 6 años (2006 - 2011). Sus características son todas variables numéricas y consisten en promedios por hora de ciertas variables medidas en el ambiente, y la variable a predecir es la cantidad de energía eléctrica producida por hora.

  Categoría: Regresión.

- *MNIST*[4]: Siendo otra base de datos reconocida y altamente utilizada en la comunidad científica de visión computacional, MNIST contiene imágenes de digitos manuscritos en un total de 60.000 ejemplos de entrenamiento y 10.000 de prueba. Dichas imágenes se encuentran normalizadas y centradas en un tamaño con resolución fija.

  Categoría: Clasificación / Regresión (reconstrucción de imágenes).

Validación del framework

En base a los recursos descriptos, se llevaron a cabo las siguientes acciones para corroborar el correcto funcionamiento de las funcionalidades ofrecidas.

[3]Extraído de UCI Machine Learning: `http://archive.ics.uci.edu/ml`
[4]Extraído de DeepLearning.net: `http://deeplearning.net/tutorial/gettingstarted.html`

Testeo en integración continua

De forma que todas las características del framework sean validadas en términos de funcionalidad, se procede a utilizar tecnologías que permitan testear cada uno de los módulos implementados y la integración de todos ellos. Esto es realizado con el fin de seguir buenas prácticas de ingeniería en software [**?**], que permitan desarrollar un producto de software que asegure cierta calidad en su integración y despliegue. A partir de ello, se implementa un esquema de testeo automatizado realizando las siguientes acciones [**?**]:

- **Prueba unitaria**: Mejor conocida en inglés como *unit testing*, es una forma de comprobar el correcto funcionamiento de un módulo de forma aislada al resto de los componentes. Para ello se define un escenario para ejecutar las acciones de ese módulo en forma separada, y se corrobora obtener un cierto comportamiento esperado en la salida de cada una de ellas. La ejecución de todas las pruebas unitarias para el framework se realizan de forma automatizada mediante la librería de Python *nose*. [5]

- **Cobertura de código**: Mejor conocida en inglés como *code coverage*, es una medida para cuantificar el grado o porcentaje de código fuente que se ejecuta en un conjunto de pruebas definido. La misma es utilizada para conocer la cantidad de código que es cubierta por las pruebas, lo cual es útil a la hora de depurar errores o malas definiciones en los módulos del software. Para implementarla en Learninspy, se utilizó el servicio de *coveralls* [6] que aprovecha el conjunto de pruebas unitarias definidas anteriormente para calcular el porcentaje deseado.

- **Integración continua**: Es una práctica de la ingeniería de software por la cual todos los aportes desarrollados para un sistema dado se integran frecuentemente (por lo general, en forma diaria) sobre un entorno de integración que construye dicho sistema de forma automática (ejecutando todas las pruebas definidas), así se pueden detectar errores de integración lo antes posible. En el presente framework se utiliza TravisCI [7] para ello, que ofrece hosting del entorno de integración para poder ejecutar un build automático cada vez que se empuja un nuevo aporte en el repositorio.

- **Salud de código**: Mediante el servicio *Landscape* se puede conocer una medida de la salud de un proyecto en Python en base a ciertas métricas definidas sobre el código fuente (e.g. errores, malas definiciones, malas prácticas de software, etc).

A partir de implementar esto, se tiene definido un procedimiento para mantener y extender el framework asegurando que no se pierda la calidad del software en las integraciones que vayan ocurriendo. Para especificar la calidad esperada en este producto, por cada ejecución de un plan de integración se fijaron los siguientes criterios de aceptación:

- La cobertura de código por *coveralls* no debe ser inferior al 90 %.

[5] `http://nose.readthedocs.io/en/latest/`
[6] `https://coveralls.io/github/leferrad/learninspy`
[7] `https://travis-ci.org/leferrad/learninspy`

Figura 5.1: Captura de pantalla del README publicado en GitHub, donde se muestran los badges que certifican la aplicación correcta del esquema dado.

- La salud del código provista por *Landscape* no deber ser inferior al 90 % .
- El repositorio de código no puede permanecer en un estado de fallo respecto al building automático.

En la Figura 5.1 se puede apreciar que en la versión actual de Learninspy se cubre un 95 % del código fuente, cuya salud es de un 92 %, y además el build automático se realiza sin fallos en TravisCI, lo cual se corrobora con los correspondientes *badges* desde el repositorio en GitHub.

Experimentos de validación

Dado que se quiere comparar funcionalidades con otros frameworks pero también validar aquellas que son nuevas, se define la siguiente lista de experimentos a realizar en la evaluación de desempeño sobre Learninspy:

- Velocidad de procesamiento respecto a herramientas similares.
- Clasificación de imágenes con redes neuronales.
- Compresión y reconstrucción de imágenes con AutoEncoders.
- Configuración de la optimización en el entrenamiento distribuido.

Velocidad de procesamiento respecto a herramientas similares

En este experimento se replicó el procedimiento detallado en una publicación reciente [**?**], donde se comparaba el desempeño de distintos frameworks en términos de velocidad de procesamiento. Específicamente, se medía la duración de las herramientas para producir dos tipos de cálculos: a) los gradientes involucrados en el algoritmo de retro-propagación, para dar idea de la duración de entrenamiento; b) la salida de una red ante una entrada, para conocer cuánto demora en realizar predicciones.

Siguiendo ello, se utilizó la misma configuración sobre Stacked AutoEncoders para poder comparar resultados con aquellos frameworks tratados. Por lo tanto, el SAE constaba de 3 AutoEncoders formando una arquitectura con las siguientes dimensiones: 784, 400, 200, 100, 10. El ajuste se realizó sobre los datos de MNIST, utilizando mini-batch de 64 ejemplos y sin paralelismo de modelos en la optimización para que sea comparable al procedimiento original.

Para la experimentación en Learninspy se utilizó el servidor de alta gama, cuya velocidad de procesamiento es menor que la del servidor utilizado en el trabajo de referencia (específicamente 35 % menor en la frecuencia de reloj de los procesadores). Es por ello que, para realizar una comparación más acertada, se normalizaron las unidades de tiempo de la siguiente forma:

```
duración (ciclos) = duración (segundos) * velocidad (Hertz)
```
(5.1)

Por ejemplo, si una prueba dura 10 milisegundos utilizando el servidor de este trabajo, entonces en términos de ciclos computacionales la duración resulta:

$$\begin{aligned} d = 10(ms) \cdot 2{,}3(GHz) &= 10 \times 10^{-3}\,(s) \cdot 2.3 \times 10^{9}(ciclos/s) \\ &= 23 \times 10^{6}(ciclos) = 23Mciclos \end{aligned}$$

Los resultados pueden apreciarse en los gráficos de barras presentados en las Figuras 5.2 y 5.3 para la ejecución con 1 hilo y 12 hilos de procesamiento respectivamente. Allí se denota con el acrónimo "GAE" al cálculo referido a los gradientes en un AE dado durante el pre-entrenamiento, "GSE" al cálculo de gradientes para el ajuste fino de todo el SAE, y "FSE" para la salida producida por el SAE durante una predicción.

Se puede ver que el rendimiento de Learninspy es altamente inferior al de los frameworks Torch y Theano, pero respecto a TensorFlow la diferencia resulta menor. Esto se encuentra razonable debido a las siguientes cuestiones:

- En cuanto a lenguajes de programación, Torch se encuentra desarrollado en Lua y Theano tiene gran parte de código hecho en C. Es bien sabido que dichos lenguajes son altamente óptimos en velocidad de cómputo respecto a Python, lenguaje con el que están mayormente hechos los frameworks TensorFlow y Learninspy. Por esta razón es que se denota una gran diferencia de rendimiento en todas las comparaciones.
- Todos los frameworks utilizados para la comparación poseen una clase "Tensor" optimizada para realizar operaciones numéricas, que además consideran el núcleo por el cual logran la eficiencia de cómputo. Learninspy

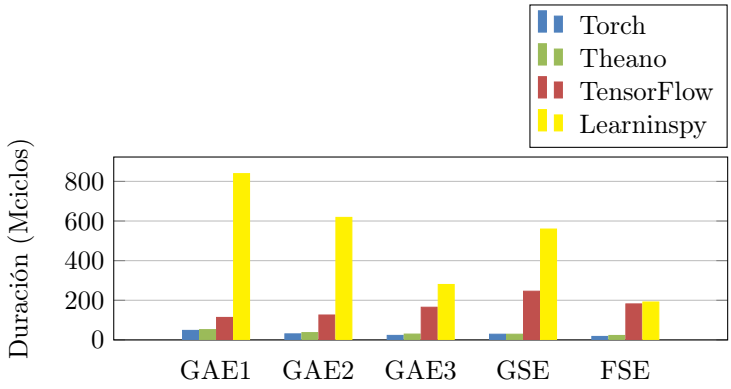


Figura 5.2: Comparación de frameworks de aprendizaje profundo en términos de la duración para realizar distintos tipos de salida, restringiendo la ejecución a 1 hilo de procesamiento

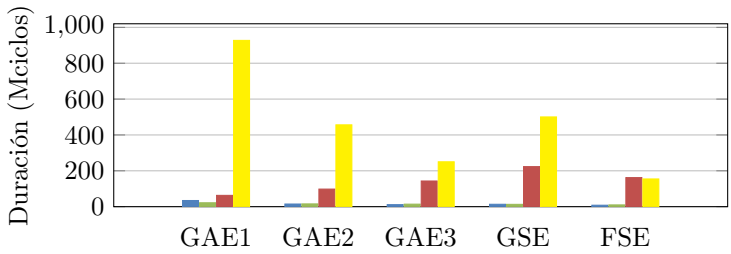


Figura 5.3: Comparación de frameworks de aprendizaje profundo en términos de la duración para realizar distintos tipos de salida, restringiendo la ejecución a 12 hilos de procesamiento

no posee esa ventaja ya que sólo utiliza rutinas de Python y el soporte de álgebra lineal está provisto únicamente por NumPy sin ninguna capa de optimización agregada.

- Debido a la cuestión anterior, cuando se manejan arreglos numéricos con alta dimensión (como el caso del modelo tratado en este experimento) va a reflejarse la ventaja computacional de los framework que tengan optimizadas sus rutinas algebraicas especialmente en el algoritmo de retropropagación que implica el cálculo de gradientes multidimensionales. Esto se corrobora en el gráfico donde se denota que, a medida que decrece la dimensionalidad manejada (i.e. desde el primer AE hasta el último), la duración para los gradientes disminuye importantemente a diferencia del resto de los frameworks.

Como conclusión general del experimento, se puede ver que Learninspy no se considera óptimo en términos de esta categoría analizada, lo cual deja abierto como trabajo futuro el hecho de mejorar tanto los módulos relacionados a cálculos algebraicos como los algoritmos que los aprovechan para el entrenamiento y las predicciones de una red neuronal.

Clasificación de imágenes con redes neuronales

A fines de evaluar el desempeño en términos de resolución de problemas, se procede a utilizar los datos de MNIST para modelar un clasificador mediante las siguientes herramientas:

- Regresión logística multi-clase mediante Spark MLlib.
- Red neuronal profunda mediante Deeplearning4j.
- Red neuronal profunda mediante Learninspy.

La idea entonces fue determinar una línea base de resultados para la clasificación mediante un clasificador básico (que fue explicado en la Sección 2.1.1) y a partir de ello comparar el desempeño contra redes neuronales profundas, realizando a su vez también una comparación entre el framework de este trabajo y otro similar como Deeplearning4j.

Utilizando una configuración basada en la que está publicada en la web de Deeplearning4j [8], se modeló una red con una sola capa oculta de 1000 unidades. Sus pesos sinápticos se regularizan mediante una norma L2 ponderada por un valor de 1e-4, y las activaciones están definidas por una ReLU.

Los resultados pueden apreciarse en la Tabla 5.1, donde se refleja el desempeño en términos de duración del modelado y precisión obtenida en la clasificación. Se puede notar que el modelado con redes neuronales obtiene un desempeño mejor que con una regresión logística, y aunque los resultados entre ambos frameworks no son iguales debido a algunas diferencias (e.g. optimización utilizada, inicialización de pesos sinápticos) se considera que en ambos casos se logra buena precisión en la clasificación sin requerir una configuración compleja para ello.

Tabla 5.1: Resultados de clasificación con datos de MNIST.

Modelo	Accuracy	Precision	Recall	F1-Score
RegLog	0.9154	0.9144	0.9141	0.9142
DL4J-DNN	0.9710	0.9709	0.9707	0.9708
LSPY-DNN	0.9410	0.9405	0.9408	0.9407

Compresión y reconstrucción de imágenes con autocodificadores

De forma que se valide el comportamiento esperado en un autocodificador, se realizó un experimento sobre los datos de MNIST para corroborar que se obtenga una buena reconstrucción de dichas imágenes en la salida de la red entrenada. Para ello se replicó la misma configuración detallada en el blog de Keras [9], el cual es un reconocido framework de aprendizaje profundo para Python. Los resultados se valoraron positivamente en dos formas: a) de forma objetiva, se obtuvo valores de r^2 mayores a 0.85 en la regresión de reconstrucción sobre los conjuntos de entrenamiento y prueba, lo cual indica un buen ajuste con generalización deseable; b) de forma subjetiva, en la Figura 5.4 se puede apreciar visualmente la calidad de reconstrucción sobre algunas imágenes del conjunto de prueba, mostrando un buen desempeño en la tarea designada.

[8] `http://deeplearning4j.org/mnist-for-beginners.html`

[9] Experimento "Adding a sparsity constraint on the encoded representations" detallado en `https://blog.keras.io/building-autoencoders-in-keras.html`

Figura 5.4: Reconstrucción de las imágenes de MNIST mediante un autocodificador. Arriba las imágenes originales, y abajo las reconstruidas.

Configuración de la optimización en el entrenamiento distribuido

Como se presentó en la Sección 4.2.1, la optimización de modelos en forma distribuida introduce ciertos parámetros a configurar para su ejecución. De forma que se muestre empíricamente la relación entre algunos de ellos, se realizaron dos experimentos sobre todas las infraestructuras para conocer el impacto que tiene en la duración del modelado y sus resultados.

1) Analizar sobre datos reales la relación entre los parámetros de optimización.

Básicamente el primer experimento consta de entrenar una red neuronal de 1 capa oculta sobre los datos de CCPP, y en cada prueba se fue variando un parámetro que define la optimización para evaluar cómo influye en todo el proceso. Específicamente, se fue aumentando en 10 unidades la cantidad máxima de iteraciones o épocas a realizar por cada modelo réplica durante el entrenamiento paralelizado. Además, el nivel de paralelismo (i.e. cantidad de modelos réplicas a ajustar en cada época global) se configuró de dos formas: valiendo 4 para las pruebas sobre el ordenador personal, y 10 para aquellas realizadas tanto en el servidor como en el clúster. El resto de la configuración se mantuvo fija, eligiendo 20 ejemplos para cada batch de datos, y se definió como criterio de corte para la optimización general el hecho de alcanzar un R^2 igual a 0.9 de ajuste sobre el conjunto de validación.

La idea entonces fue comparar la duración final del entrenamiento de un modelo (i.e. optimización global) al alcanzar el criterio de corte definido, respecto a la cantidad de iteraciones que se realizaban en forma paralela sobre cada batch de datos (i.e. optimización local).

En la Figura 5.5 se presentan los resultados de variar la duración en épocas de la optimización local para el modelado de la red neuronal. Por cada configuración respecto al paralelismo, se muestra cuántas épocas comprendió tanto para la optimización global como para el total del proceso (i.e. producto de la cantidad local por la global). A partir de ello se percibe también la diferencia que introduce en el modelado la variación del nivel de paralelismo, en términos de la precisión obtenida en los resultados luego de cada época, que termina a su vez impactando en la duración total del experimento al alcanzar en menor o mayor tiempo el criterio de corte definido.

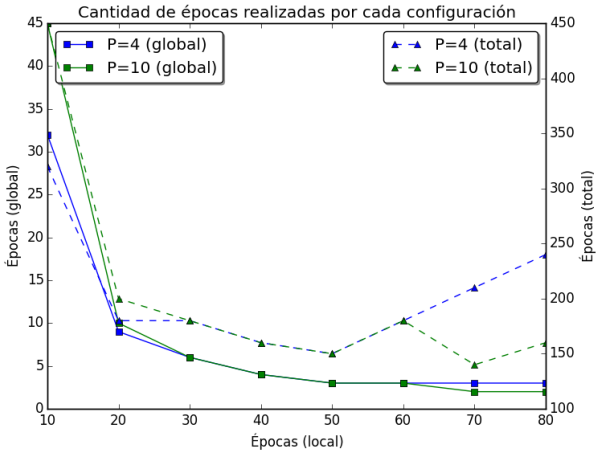


Figura 5.5: Duración del modelado en épocas variando la configuración de su optimización, donde se distingue el paralelismo P utilizado y el tipo de duración

2) Estudiar la duración resultante en los distintos tipos de infraestructura.

Por otro lado, en el segundo experimento se midió la duración del modelado en términos de tiempo comprendido por cada época local. Para esta prueba se quitó la restricción dada por el criterio de corte en la optimización, de forma que se estimen mejor los valores de duración mediante más muestras. Entonces, nuevamente variando la cantidad de épocas a realizar en forma local, se midió cuántos segundos demoraba cada época en la optimización dada sobre los tres tipos de infraestructura disponibles.

En la Figura 5.6 se visualizan los resultados de esta prueba, donde para estimar la duración se utilizó tanto el promedio como la mediana de las muestras. La diferencia percibida entre ambos valores se debe a que generalmente las primeras épocas de la optimización en Learninspy demoran bastante más que el resto, y es porque Spark allí trata de optimizar el grafo de tareas a ejecutar para el trabajo involucrado hasta que luego se desempeña establemente. Esto se acentúa aún más cuando se realiza en un clúster, ya que el proceso incluye la coordinación de los nodos computacionales para realizar el conjunto de tareas necesario. Por ello se considera que la mediana es una buena estimación para conocer la duración de cada época en este proceso, y su evolución es lineal como puede apreciarse en el gráfico presentado.

En base a estos dos experimentos realizados además se puede calcular la duración total de cada experimento multiplicando la cantidad de épocas realizadas por la correspondiente estimación de duración por época, y con ello deducir que el modelado de menor tiempo se realizaría utilizando 20 épocas de forma local cuando P=4 y mediante 50 épocas para P=10.

En los experimentos anteriores, la función de consenso elegida fue siempre la misma (aquella con ponderación logarítmica) ya que al cambiarla se notaba que la cantidad de épocas resultantes también variaba. Esto se debe a que, al mezclar los modelos réplicas durante el entrenamiento distribuido, la precisión

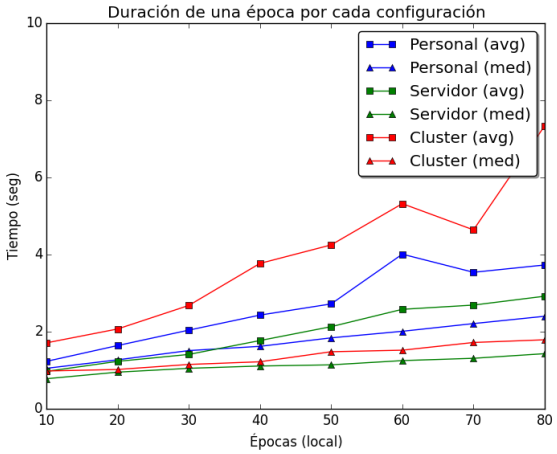


Figura 5.6: Duración del modelado por cada época en forma global, variando la configuración de su optimización

de los resultados de validación variaba de forma tal que por cada función se podía llegar en menor o mayor tiempo al criterio de corte definido. Para corroborar ello, en la Tabla 5.2 se muestra cómo cambia el desempeño final del modelado con igual configuración variando sólo la función de consenso para la optimización.

Tabla 5.2: Resultados obtenidos variando las funciones de consenso para la optimización del modelo

Función de consenso	r^2_{train}	r^2_{valid}	r^2_{test}	RMSE	RMAE	Exp Var
AVG	0.90947	0.91882	0.90860	5.09933	2.01771	0.90866
W_AVG	0.90955	0.91874	0.90869	5.09683	2.01658	0.90869
LOG_AVG	0.90943	0.91881	0.90855	5.10081	2.01788	0.90857

Parte III

Aplicación en electroencefalografía

Esta parte de la tesis se dedica a desarrollar los casos de aplicación elegidos como prueba del framework implementado. Para ello, se explican conceptos básicos de las dos problemáticas de electroencefalografía elegidas (*Potencial P300* y *Habla imaginada*), y por cada una se especifica cómo son los datos utilizados y cuál es el tratamiento que recibieron en este trabajo. Luego se presentan los resultados obtenidos para cada caso de aplicación, detallando conclusiones acerca de los mismos.

Capítulo 6

Electroencefalografía

Todo hombre puede ser, si se lo propone,
escultor de su propio cerebro.

Santiago Ramón y Cajal

Resumen: Este capítulo pretende introducir brevemente los conceptos manejados en cada caso de aplicación realizado. Específicamente, se puntualiza sobre la electroencefalografía y su uso para construir interfaces cerebro-humano, destacando así la motivación principal para llevar a cabo las aplicaciones del presente trabajo de tesis.

Señales de electroencefalograma

La electroencefalografía (**EEG!**) consiste en el registro de la actividad cerebral de un individuo, la cual es captada mediante electrodos como señales eléctricas producidas por un conjunto de neuronas en un período de tiempo. El primer registro de actividad espontánea cerebral fue realizado en 1875 por Richard Canton, quien mediante un galvanómetro pudo observar impulsos eléctricos medidos sobre la superficie del cerebro de un animal. En 1924, Hans Berger registra el primer electroencefalograma de un humano mediante el uso de tiras metálicas pegadas sobre el cuero cabelludo y utilizando también un galvanómetro como elemento de medida. Durante dicho procedimiento pudo medir el patrón irregular y de poca amplitud de la señal de electroencefalograma, generando las bases para el desarrollo de la electroencefalografía [**?**].

Existen dos modalidades en que se puede registrar la actividad del cerebro, según la técnica utilizada para la medición: invasiva y no invasiva [**?**]. Las técnicas invasivas, como el electrocortigrama (ECoG), utilizan micro electrodos implantados en el cerebro que miden la actividad de las neuronas, individualmente consideradas. En cambio, la electroencefalografía es una técnica no invasiva que indica la actividad eléctrica del cerebro medida con electrodos superficiales colocados sobre el cuero cabelludo, con lo cual se proporciona información detallada sobre la actividad local de pequeñas regiones cerebrales. Se

caracteriza por captar señales de pequeñísima amplitud y de gran variabilidad en el tiempo, así como por tener un valor pobre de relación señal/ruido o **SNR!**, y su instrumentación de adquisición es de fácil colocación y bajo costo.

Generalmente el conjunto de electrodos que registran la señal de EEG se ubica en forma directa sobre el cuero cabelludo siguiendo el sistema internacional 10-20, denominado así porque los electrodos están espaciados entre el 10 % y el 20 % de la distancia total entre puntos reconocibles del cráneo. Actualmente se utilizan unos gorros que llevan incorporados los electrodos, a cada uno de los cuales se les introduce un gel conductor que facilita la recepción de la señal a través del cuero cabelludo. Los electrodos se unen en un conector y éste se conecta con el cabezal del EEG, lugar donde se recoge toda la actividad eléctrica y se envía al sistema de amplificadores para su trascripción a datos digitales.

Interfaces Cerebro-Computadora

Una interfaz cerebro-computadora o ICC (en inglés, conocida como *Brain Computer Interface* o **BCI!**) es un sistema que permite a una persona la comunicación y control de su entorno sin necesidad de usar nervios o músculos. Esta registra la actividad cerebral de un sujeto para traducirla en comandos sobre una computadora. Es por ello que tiene gran utilidad en aplicaciones de robótica, videojuegos, y especialmente en salud neurológica al ser la única posibilidad de comunicarse para personas con discapacidades motrices. En cuanto a la toma de datos, los métodos no invasivos son más atractivos ya que son más fáciles de implementar en pacientes y además han demostrado tener efectividad comparable a la de aquellas interfaces con electrodos implantados cuando se utilizan algoritmos apropiados de aprendizaje maquinal [**?**].

Los sistemas **BCI!** no son lectores de mente. Están diseñados para reconocer patrones en datos extraídos del cerebro (e.g. pensamientos o estados mentales) y asociarlos con comandos, pero eso no significa que sea capaz de leer una mente. Según Forslund [**?**], no existe forma de que un sistema técnico pueda actualmente informar *qué piensa una persona*, y hay varias razones para ello:

1. Las técnicas de sensado que se usan para extraer información del cerebro son imprecisas, y pueden representar sólo una microscópica fracción del total de actividad cerebral.

2. Incluso si fuera posible extraer los estados de todas las células en el cerebro, esa información sería muy compleja de manejar, aún para la más poderosa súper computadora.

3. Los cerebros son únicos e individuales. Aunque dos personas piensen exactamente lo mismo, la actividad cerebral que se manifiesta es completamente diferente.

Aclarado esto, se puede entender a un sistema BCI como un traductor de actividad cerebral en ciertos comandos definidos, por lo cual se compone de los siguientes módulos:

- Sensado de datos cerebrales en el sujeto y filtrado o mejora de la calidad de las señales.

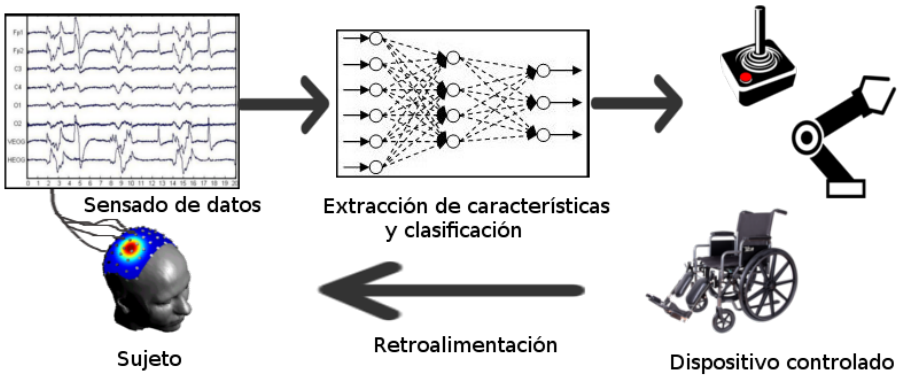


Figura 6.1: Diagrama en bloques del funcionamiento de un BCI.

- Extracción de características relevantes para la tarea del sistema.
- Clasificación del fenómeno registrado desde el cerebro.
- Traducción en movimiento o control de un dispositivo, y retroalimentación al sujeto en cuestión.

En la Figura 6.1 se puede visualizar el diagrama en bloques descripto. Dado que en este trabajo los modelos se basan en redes neuronales profundas, se propone como parte de un único bloque a las tareas de extracción de características y clasificación, ya que este tipo de modelo se encargaría de todo esto tal como se ha anticipado en la Sección 2.3. Es por ello que para los casos de aplicación tratados se propuso como alcance que los experimentos se realicen sobre datos crudos de electroencefalografía, sin ninguna extracción de características previa. Con ello se sigue la idea de otros trabajos en aprendizaje profundo [**?**][**?**][**?**], en donde se espera que la red neuronal sea capaz de aprender a extraer las características sobre la entrada.

Por lo tanto, para estas aplicaciones se pretende probar el desempeño de los modelos construidos mediante el framework desarrollado, y así corroborar la aptitud para facilitar o resolver problemas complejos como el comprendido en construir sistemas BCI mediante electroencefalografía.

Capítulo 7

Potencial P300

Entre estímulo y respuesta, hay un espacio
donde elegimos nuestra respuesta.

Stephen Covey

Resumen:

En este capítulo se dan a conocer todos los contenidos referidos al estudio del potencial P300 y el enfoque utilizado para detectar su presencia en señales de **EEG!**. A su vez, se explica en detalle el conjunto de datos utilizados para tratar este caso de aplicación y la metodología implementada en este proyecto para llevar a cabo la experimentación. Finalmente, se comunican los resultados obtenidos en comparación a distintos enfoques, incluyendo el del trabajo original de referencia.

Como primer caso de aplicación se escogió el de detección de P300 en señales de EEG, ya que hasta la actualidad se mantiene como uno de los principales paradigmas para construir sistemas BCI. Esto se debe a que constituyen sistemas rápidos, efectivos y prácticamente no requieren de entrenamiento para casi cualquier usuario. Recientes trabajos muestran que los sistemas BCI basados en P300 pueden ser empleados para una gran variedad de funciones, incluso sobre usuarios con discapacidad, por lo cual en este trabajo se propuso recurrir a esta aplicación utilizando aprendizaje profundo.

Conceptos básicos

La onda P300 se caracteriza por producir un pico positivo en una señal de **EEG!** humano, aproximadamente luego de 300 ms desde la presentación de un estímulo externo significativo para el sujeto en cuestión, por lo cual se provoca en ella un cambio que se denomina "potencial relacionado a eventos" (en inglés, conocido como *event-related potential* o **ERP!**) [**?**]. Dicho estímulo externo puede ser de cualquier naturaleza, y de ello depende esencialmente la técnica para detectar un P300 en una señal de **EEG!**. Un procedimiento típico para evocar esta onda es el paradigma *oddball*, donde dos estímulos se presentan en

orden aleatorio y uno tiene mayor probabilidad de ocurrencia que el otro. Se describen algunas técnicas para la presentación de estímulos:

- Sistema deletreador: Farwell y Donchin [**?**] fueron los primeros en emplear el potencial P300 como señal de control en un **BCI!**. Con el sistema que crearon, los sujetos podían deletrear palabras eligiendo secuencialmente los caracteres de un alfabeto que se presentaba en pantalla mediante una matriz de 6x6 (como letras y otros símbolos). Las filas y columnas de esa matriz se iluminan con destellos en forma aleatoria, y si alguna de ellas contiene el símbolo deseado por el sujeto entonces se evoca un P300 en la señal de **EEG!**. Así, mediante un algoritmo simple, durante la toma de datos se clasifica en cada registro del **EEG!** la presencia o no de un P300 en base a las filas y columnas iluminadas que corresponden a cada caracter deletreado por el sujeto en cuestión.

- Prueba del conocimiento culpable: Conocido como GKT (en inglés, *Guilty Knowledge Test*) es un protocolo donde se realizan preguntas hacia un individuo, y cuando se presenta el ítem significativo de cada pregunta se registra una onda P300 con características en su forma distintas a las de los demás ítems. Diversos estudios, incluyendo los de Farwell y Donchin [**?**], muestran niveles altos de exactitud en la clasificación de los sujetos como culpables o inocentes valiéndose de la onda P300 mediante este protocolo.

- Multiple choice: Procedimiento similar al sistema deletreador, sólo que en lugar de una matriz de letras y/o símbolos lo que se presenta en pantalla al sujeto es un conjunto de ítems (e.g. imágenes, palabras, etc.) que están relacionados a la tarea perseguida por el sistema BCI. Éstos también se iluminan de forma aleatoria, esperando que se produzca el P300 en el momento que se ilumine aquel ítem de interés para el sujeto.

Uno de los principales desafíos en la clasificación de P300 es lidiar con la baja relación señal/ruido o SNR. Dado que la señal de **EEG!** registrada desde el cuero cabelludo contiene mucho ruido debido al resto de la actividad eléctrica en el cerébro, es difícil aislar una onda P300 de la señal ruidosa resultante. El problema de la baja SNR es usualmente manejado mediante el promediado de varias épocas de registro consecutivas, lo cual logra cancelar gran parte del ruido en la señal y con ello se facilita la detección de la P300. No obstante, este enfoque conlleva el costo de reducir la tasa de comunicación que impacta directamente en el desempeño de un sistema **BCI!**. En la Figura 7.2 se visualiza el resultado de promediar épocas del registro en un EEG para resaltar la onda P300 y se puede observar que, tal como se dijo antes, normalmente se presenta cerca de los 300 ms luego del estímulo, aunque para algunos sujetos con discapacidad puede aumentar esta latencia (a 500 ms) y disminuir la amplitud del pico [**?**].

Farwell y Donchin reportaron en su trabajo pionero que es necesario promediar entre 20 y 40 épocas para lograr una exactitud mayor al 80 %, con una tasa de comunicación alrededor de 12 bits por minuto [**?**]. Aunque este trabajo estableció la factibilidad de un sistema BCI deletreador basado en P300, la tasa de comunicación era demasiado baja como para llegar a un uso práctico. A partir de ello surgió mucho trabajo en las últimas décadas que se enfocó en mejorar la exactitud y la velocidad de comunicación de sistemas **BCI!** basados en P300, y aunque está ampliamente demostrado que promediando épocas se estabiliza la

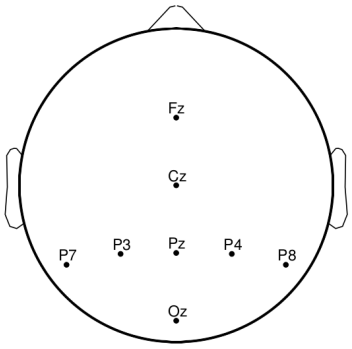


Figura 7.1: Configuración de 8 electrodos elegida para los experimentos en P300

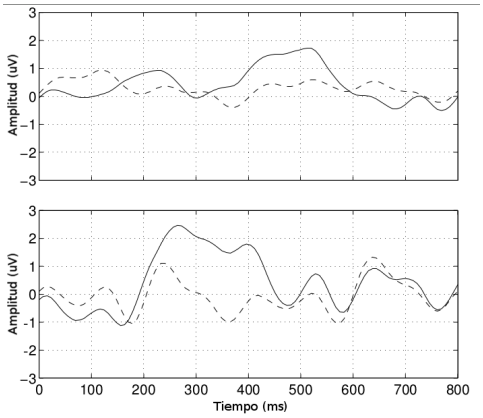


Figura 7.2: Promediado de las ondas del electrodo Pz. Arriba, para los sujetos con discapacidad (S1-S4). Abajo, sujetos sin discapacidad (S6-S9)

amplitud de la onda P300, el desafío sigue siendo mejorar el desempeño de la detección en única época o *single trial* (i.e. sin ningún promediado).

Corpus de datos

Los datos de **EEG!** utilizados fueron adquiridos en forma gratuita desde el sitio web del Grupo de Procesamiento de Señales Multimedia perteneciente al Instituto de Tecnología Suizo Federal (EPFL)[1]. Dicho corpus fue creado para realizar una publicación referida a construcción de sistemas **BCI!** para personas discapacitadas [**?**]. El mismo se encuentra almacenado en formato MATLAB, y además contiene los scripts utilizados en el paper para realizar el pre-procesamiento.

Registro

Las señales de EEG fueron registradas con una frecuencia de muestreo de 2048 Hz, mediante 32 electrodos ubicados según el sistema internacional 10-20 y utilizando un equipo Biosemi Active Two para la amplificación y conversión analógica a digital de las señales. Para la toma de datos, se utilizó un procedimiento de *multiple choice* con seis imágenes que pueden verse en la Figura 7.3, usando una población de cinco sujetos discapacitados y cuatro sin discapacidad. Particularmente, en este trabajo se optó por utilizar los datos provenientes de los primeros 4 sujetos con discapacidad (del Sujeto 1 al Sujeto 4). El protocolo constó de cuatro sesiones por sujeto, con una corrida para cada una de las seis imágenes en donde se pedía al sujeto que cuente en silencio cuántas veces la imagen que fue designada era iluminada por destellos. Luego las imágenes eran presentadas en pantalla y las mismas se iluminaban en forma de destellos de manera aleatoria después de que suene un tono de alerta. Al final de la corri-

[1]Repositorio en: `http://mmspg.epfl.ch/BCI_datasets`

Figura 7.3: Imagenes usadas para evocar el potencial P300 durante el registro de señales de EEG.

da se les preguntaba a los sujetos el resultado de la cuenta para corroborar su desempeño. En total se obtuvo en promedio 810 ejemplos por sesión, lo que hace un total de 3240 ejemplos por sujeto también en promedio. Para mayor detalle del protocolo de adquisición de datos, remitirse a la publicación citada.

Procesamiento de datos

Los siguientes pasos de pre-procesamiento fueron aplicados sobre los datos registrados, por lo cual se parte sobre el resultado de los mismos para el tratamiento realizado en esta tesis sobre esta aplicación:

1. *Referencia:* Se usa como referencia la señal promediada que proviene de los dos electrodos mastoidales.

2. *Filtrado:* Se utilizó un filtro pasabandas Butterworth de sexto orden (forward-backward), cuyas frecuencias de corte fueron de 1 Hz y 12 Hz.

3. *Submuestreo:* La frecuencia de muestreo de las señales extraídas del EEG fue bajada de 2048 Hz a 32 Hz mediante decimación.

4. *Extracción de single trial:* Se extrajeron los ejemplos de 1000 ms que comenzaban con la intensificación de una imagen. Debido al intervalo de 400 ms entre cada estímulo, se permitió un solapamiento de 600 ms entre estímulos contiguos.

5. *Remoción de artefactos:* Dado que los parpadeos, movimientos oculares y movimientos o cualquier actividad muscular puede causar artefactos que contaminen el EEG, manifestándose como amplitudes grandes, se procedió a removerlos mediante *winsorizing* reemplazando las muestras de cada

electrodo por percentiles calculados. Los valores de amplitud que estaban por debajo del percentil 10 o por arriba del percentil 90, fueron reemplazados por el percentil 10 o el 90 respectivamente.

6. *Escalado:* Las muestras de cada electrodo fueron escaladas al intervalo $[-1, 1]$.

7. *Selección de electrodos:* Para el presente trabajo se eligió la Configuración II de 8 electrodos que se muestra en la Figura 7.1, ya que con ella se obtuvo los mejores resultados en el trabajo original de referencia.

8. *Construcción del vector de características:* Se concatenaron las muestras de cada electrodo consecutivamente para formar un vector de características unidimensional.

A partir de lo obtenido en el procedimiento anterior, se dispuso lo siguiente:

- Por cada uno de los 4 sujetos tratados, se dividió el total de datos en conjuntos de entrenamiento (70 % del total), validación (20 %) y prueba (10 % restante), lo cual es reflejado en la Tabla 7.1.
- Cada entrada del conjunto de datos corresponde a una época única (*single trial*), por lo cual no se realizó ningún promediado de épocas para limpiar las señales de EEG.

Tabla 7.1: Descripción básica de los conjuntos de datos utilizados para obtener el modelo clasificador de P300.

	P300	**NoP300**	**Total**
S1	557	2784	**3341**
S2	554	2769	**3323**
S3	557	2784	**3341**
S4	553	2764	**3317**

Experimentos

Para definir la metodología a utilizar en la experimentación, se tuvieron en cuenta las siguientes particularidades del problema tratado:

- La clasificación a realizar es binaria, donde se predice la presencia o ausencia de un fenómeno (i.e. clase "P300" vs clase "NoP300").
- Dicho fenómeno se puede interpretar como una "anomalía" en la señal del EEG, que debido a la baja **SNR!** de la misma se mezcla con el resto de la actividad cerebral presente y por ende no es fácil de percibir.
- Los datos se encuentran desbalanceados en cuanto a clases, ya que la clase "NoP300" normalmente dispone de más ejemplos que la clase "P300" debido a la menor prevalencia de este último tipo de evento en una señal de EEG registrada.

Es por ello que para diseñar un clasificador de la ocurrencia o no de la onda P300 en una señal de EEG se vio como factible contemplar esa diferencia para definir el proceso de aprendizaje del modelo, el cual debe ser capaz de reconocer internamente las características necesarias para lograr esa distinción a partir de una entrada cruda. Por lo tanto, se dispuso construir un Stacked AutoEncoder para modelar el clasificador que incluya las siguientes cuestiones en su diseño:

1. **Pre-entrenamiento**: Se pretende que la red capte de forma no supervisada una representación interna de aquellos datos que no presentan ondas P300 (clase "NoP300"), lo cual define para la detección deseada una línea base o *baseline* de la actividad cerebral sin potenciales evocados. Para ello, se realiza lo siguiente:
   - Esta etapa sigue un enfoque de clasificación con única clase (en inglés, One-Class Classification u **OCC!**), por lo cual el modelo es ajustado con datos de una sola clase para identificarla sobre el resto posible en las predicciones [**?**].
   - Se utilizan los datos de todos los sujetos, de forma que se pueda correlacionar de distintas fuentes las características del *baseline* deseado.
2. **Ajuste fino**: Una vez construida esta red neuronal con sus parámetros inicializados mediante el pre-entrenamiento, se sigue su entrenamiento supervisado convencionalmente. Esto se lleva a cabo según lo siguiente:
   - Se realiza sobre los datos de un sujeto en particular, de forma que se refine el *baseline* sobre las características propias de los datos registrados sobre un sujeto dado. Por lo tanto se obtiene un clasificador por cada sujeto tratado.
   - Se utilizan datos de ambas clases para adiestrar al clasificador en forma más precisa al incorporar características de los potenciales evocados sobre el *baseline* ya modelado, y con ello obtener una clasificación binaria de los datos.

En cuanto a la configuración de la red, se eligió una arquitectura de 6 niveles compuestos por las siguientes dimensiones: 256, 100, 80, 60, 40, 20, 2. Se utilizó una tangente hiperbólica como función de activación en las capas de cada autocodificador, ya que su imagen cae en el rango $[-1, 1]$ al igual que los valores que toman los datos de EEG luego del escalado aplicado. No se recurrió al algoritmo DropOut para regularizar las capas ya que no mejoraba los resultados, aunque para lograr generalización en el desempeño de la red se utilizaron normas L1 y L2 ponderadas por los valores 5e-7 y 3e-4 respectivamente.

Respecto a la configuración de la optimización, se utilizó el algoritmo Adadelta siguiendo como criterio de corte un máximo de 20 iteraciones en forma local y de 100 iteraciones en forma global. La función de consenso para la mezcla de modelos en el entrenamiento distribuido sigue una ponderación lineal (denominada "w_avg" en Learninspy) sobre los valores obtenidos en la función de costo. Para el ajuste fino se utilizó una optimización menos intensiva basada en un gradiente descendiente del tipo NAG con baja tasa de aprendizaje.

Para tener otra referencia de comparación en términos de clasificación distinta a la del trabajo original citado, se modelaron tres clasificadores más:

- **Regresión logística**: Para determinar una línea base de resultados para la clasificación, se recurrió a un método básico de regresión logística como el explicado en la Sección 2.1.1, utilizando la implementación que provee la librería MLlib de Apache Spark™.
- **SAE sin OCC**: A fines de justificar la metodología explicada como una mejora, se implementó un Stacked AutoEncoder convencional sin seguir el enfoque OCC definido. De esta forma se puede comparar la diferencia obtenida con la metodología propuesta, en términos de desempeño para la clasificación, respecto al diseño estándar de SAEs.
- **DNN**: Se modeló también una red neuronal profunda clásica para conocer el desempeño obtenido sin recurrir al pre-entrenamiento de forma no supervisada que implementa un SAE.

Es preciso aclarar que todas estas redes neuronales se construyeron con la misma arquitectura y optimizaron de igual forma, diferenciándose únicamente en la implementación de la etapa pre-entrenamiento, de manera que la comparación pueda ser más apropiada. Además, todas las clasificaciones se basaron en regresiones logísticas salvando que en el caso de las redes neuronales se realizaban sobre la salida producidas por las capas intermedias en lugar de aplicarse directamente sobre los datos de entrada.

Resultados

Para medir el desempeño de los modelos a comparar, se utilizaron las métricas de clasificación binaria descriptas en la Sección 2.1.3, mientras que el ajuste de cada autocodificador de un SAE sobre los datos se evaluó con métricas de regresión como corresponde. Es preciso aclarar que en el trabajo original se utilizó para clasificar tanto el método Bayesian Linear Discriminant Analysis (BLDA) como Fisher's Linear Discriminant Linear (FLDA) [**?**]. Mediante los scripts adjuntos con la base de datos se replicaron los experimentos con BLDA, y la exactitud de los resultados reportados mediante validación cruzada para los sujetos 1 a 4 tratados en este trabajo eran correspondientemente los siguientes (con enfoque *single trial*): 0.5675, 0.5988, 0.6717 y 0.6329.

En los experimentos de este trabajo, se observó que los autocodificadores comprendidos en el modelo SAE usando el enfoque OCC obtenían un buen ajuste sobre el conjunto de validación, dado por un coeficiente R^2 de entre 0.75 y 0.9 aproximadamente. Además, al evaluar los resultados de dicho modelo luego de ser pre-entrenado, se obtenía en promedio para todos los sujetos una clasificación con exactitud cercana al 85 % y un F1-Score de aproximadamente 0.65. No obstante la exhaustividad de la detección de P300 era prácticamente nula (i.e. casi todo se clasificaba como "NoP300"), lo cual se corresponde con lo mencionado en la Sección 2.1.3 respecto a que la interpretación del desempeño debe contemplar diversas métricas. En base a estos resultados, se consideró que el ajuste fino resultaba indispensable para obtener una clasificación correcta.

En la Tabla 7.2 se presentan los resultados de ejecutar la etapa de ajuste fino para el SAE con OCC sobre los datos de cada sujeto, y en la Figura 7.4 se visualiza la evolución del ajuste fino sobre un sujeto en particular (gráfico obtenido mediante una funcionalidad de Learninspy).

Tabla 7.2: Resultados obtenidos por cada sujeto al modelar un SAE con OCC, discriminando por clase en cada métrica utilizada.

Sujeto	P_{P300}	R_{P300}	Acc_{P300}	P_{NoP300}	R_{NoP300}	Acc_{NoP300}
S1	0.3461	0.4909	0.2547	0.8906	0.8172	0.7426
S2	0.3402	0.5892	0.2750	0.9021	0.7681	0.7090
S3	0.6000	0.8500	0.5425	0.9638	0.8759	0.8480
S4	0.5287	0.7796	0.4600	0.9467	0.8492	0.8105

Tabla 7.3: Resultados obtenidos por cada sujeto utilizando distintos modelos.

	RegLog		**DNN**		**SAE**		**SAE-OCC**	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1
S1	0.7754	0.6006	**0.7844**	0.6249	0.7395	0.6195	0.7634	**0.6357**
S2	0.7168	0.6042	**0.7771**	**0.6653**	0.7710	0.6355	0.7379	0.6486
S3	0.8413	0.7684	0.8802	**0.8259**	**0.8862**	0.8070	0.8712	0.8204
S4	0.7703	0.6571	0.8308	0.7265	0.8066	0.7000	**0.8368**	**0.7741**
Avg	0.7759	0.6725	**0.8030**	0.7106	0.8008	0.6905	0.8023	**0.7197**

Finalmente, en la Tabla 7.3 se realiza la comparación de los clasificadores respecto a los resultados obtenidos por cada sujeto. Se puede apreciar que el modelo SAE con OCC es el que mejor se desempeña en promedio respecto a la medida de F1-Score (apropiada para tener en cuenta el balance de clases en la clasificación). Además, aproxima bastante al desempeño del modelo DNN en términos de exactitud, el cual obtuvo los mejores resultados respecto a dicha medida. Por último, es preciso notar que en general el desempeño presentado por cada sujeto es similar al reportado en el trabajo original, siendo el Sujeto 3 aquel con el que se obtuvieron mejores resultados para todos los modelos.

Conclusiones

Luego de ejecutar la experimentación explicada sobre este primer caso de aplicación, se considera en base a los resultados que puede ser factible utilizar aprendizaje profundo para construir sistemas BCI basados en detección de P300. Particularmente en base a la Tabla 7.2, como justificación se destaca lo siguiente:

- Los elevados valores de *precision* y *recall* para la clase "NoP300" indican que las predicciones de dicha clase podría tener pocos falsos negativos (i.e. no se indicarían como correspondientes a un P300). Esto es bueno para un sistema BCI ya que en su operar no ejecutaría de forma seguida comandos no deseados por clasificar erróneamente un estado sin potencial evocado.
- El valor de *precision* obtenido para la clase "P300" en promedio es menor a 0.5, lo cual indica que probablemente el sistema BCI arroje falsos positivos a la hora de detectar P300. No obstante, el valor de *recall* obtenido para la clase P300 en promedio es mayor a 0.5, lo cual indica que se tiende a lograr exhaustividad en dicha clasificación de forma tal que en un sistema BCI

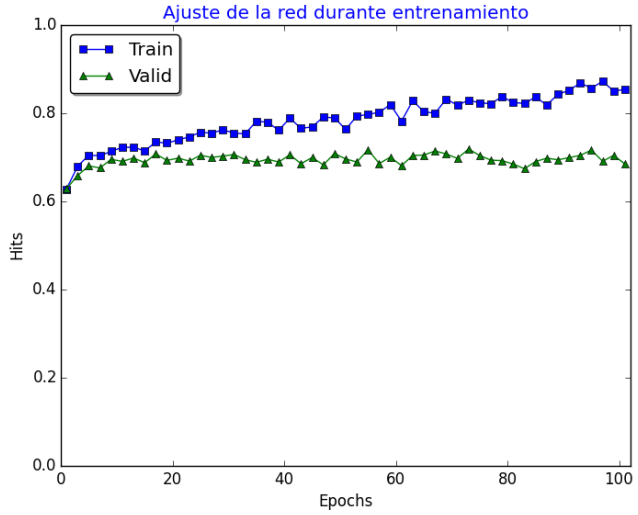


Figura 7.4: Ajuste fino del modelo SAE con OCC sobre el Sujeto 4, en términos de la medida F1-Score.

se llegue a captar la mayoría de los comandos ingresados por potenciales evocados.

En particular, el enfoque propuesto para entrenar Stacked AutoEncoders basado en OCC se condice con lo esperado respecto a mejorar el balance de clases en las predicciones, lo cual se denota por un valor de F1-Score que fue el mejor de todos los modelos y además por una exactitud aproximada a la máxima lograda. No obstante, el desempeño de todas las redes neuronales en general es bueno respecto a la publicación de referencia (mejores resultados, sin utilizar los mismos conjuntos de prueba) y además presenta mejoras en la clasificación de una regresión logística, logrando representaciones más adecuadas para desempeñar dicha tarea. Por lo tanto se encuentra viable aplicar las metodologías de aprendizaje profundo explicadas de una forma más sofisticada para construir el tipo de BCI tratado siguiendo un enfoque *single trial*.

Capítulo 8

Habla imaginada

Todo lo que una persona puede imaginar,
otras podrán hacerlo realidad.

Julio Verne

Resumen: En este capítulo se estudia la problemática del habla imaginada y las investigaciones realizadas para lograr su reconocimiento en señales electroencefalográficas. Luego, se define el corpus de datos utilizado en este proyecto y se explica el tratamiento realizado con la aplicación del framework implementado.

El segundo caso de aplicación es referido al reconocimiento del habla imaginada, el cual supone un área de investigación relativamente novedoso dentro de la neurociencia, sobre todo en sistemas BCI debido a la naturaleza que supone expresar los comandos respecto a técnicas con P300. En particular la base de datos utilizada es reciente, por lo cual con esta aplicación se ve una oportunidad de incorporar sobre ella un nuevo tratamiento mediante aprendizaje profundo.

Antecedentes

El concepto de "habla imaginada" (también referido como de "habla no pronunciada") consiste en la imaginación de pronunciar palabras o vocales sin producir ningún sonido ni movimiento mandibular, y en los últimos años se ha estado trabajando para lograr su detección en señales de EEG. Dicho reconocimiento está adquiriendo mayor atención debido a su importancia en diversos campos: desde la ayuda a personas con algún tipo de discapacidad motriz o en su comunicación, como en aplicaciones donde es crucial la privacidad de un discurso, y hasta en el control de dispositivos computarizados y videojuegos.

Uno de los primeros trabajos acerca del habla imaginada fue el realizado por Suppes en 1997, quien utilizó los registros de EEG y magnetoencefalografía (MEG) para la clasificación de 7 palabras del idioma inglés: *first*, *second*, *third*, *yes*, *no*, *right* y *left* [**?**]. Para el registro de EEG de 7 sujetos empleó 16 canales y analizó las contribuciones de cada uno a la tasa de detección. Se obtuvieron

100 registros por cada una de las 7 palabras y se alcanzó una tasa de acierto del 52 % al emplear prototipos basados en la transformada de Fourier sobre la señal promedio de cada clase.

D'Zmura [**?**] se basó en registros de EEG que captaban la imaginación de las sílabas /ba/ y /ku/ utilizando 128 canales de registro. En la etapa clasificación, se usaron filtros adaptativos logrando tasas de detección del 62 % al 87 %.

Un enfoque orientado al análisis de vocales imaginadas es el presentado por DaSalla [**?**], en el cual se registró la señal de EEG de tres sujetos para tres estados: al imaginar la pronunciación de las vocales /a/ y /u/ en forma sostenida, y durante un estado de control (sin imaginación). En este caso, como método de extracción de características se emplearon patrones espaciales comunes o CSP y como clasificador se usaron máquinas de soporte vectorial (en inglés, *Support Vector Machines* o SVM) con kernel no lineal y basados en agrupar clasificaciones binarias para obtener un clasificador multi-clase. Finalmente, se obtuvo un porcentaje de detección máximo de 78 % para la clasificación binaria producida por la SVM.

Corpus de datos

Para este caso de aplicación, se adquirió de forma gratuita una base de datos cuyos registros fueron realizados en las oficinas del Laboratorio de Ingeniería en Rehabilitación e Investigaciones Neuromusculares y Sensoriales (LIRINS), perteneciente a la Facultad de Ingeniería de la Universidad Nacional de Entre Ríos (FIUNER). El corpus se encuentra almacenado en formato MATLAB y fue creado en 2016 para una tesis de bioingeniería en la facultad mencionada [**?**].

Registro

Las señales continuas de **EEG!** fueron registradas usando un sistema compuesto por un amplificador Grass 8-18-36 en conjunto con una placa conversora analógico-digital DT9816 de marca DataTranslation. El mismo constaba de 8 electrodos superficiales de Ag-AgCl y recubiertos en oro, ubicados en base al sistema internacional 10-20, siendo seis usados como canales activos, uno como referencia y el restante como tierra. Además se utilizó una frecuencia de muestreo de 1024 Hz para cumplir con el criterio de Nyquist ya que la señal de interés tiene 40 hz como frecuencia máxima [**?**]. Los registros se realizaron sobre 15 sujetos (8 masculinos y 7 femeninos) de entre 20 y 30 años de edad, y ninguno poseía algún problema significante de salud.

En el protocolo de registro, la secuencia que se utilizó para la estimulación y repetición del vocabulario propuesto se ilustra en la Figura 8.1. Esta comenzaba con la presentación de una imagen de concentración durante dos segundos, seguido por un lapso de igual duración donde se indicaba la palabra objetivo. Posteriormente, se mostraba la modalidad que debía utilizar por un espacio de cuatro segundos. Si el diccionario estaba compuesto por las vocales, las debían pensar de manera sostenida durante el largo del intervalo, mientras que para el caso de los comandos se emitieron una sucesión de clics y la palabra tenía que ser repetida cada vez que lo escuchaban. Por último, el sujeto tenía 4 segundos en los cuales descansar hasta la aparición de una nueva imagen de concentración, donde estaba permitido volver a ponerse cómodo.

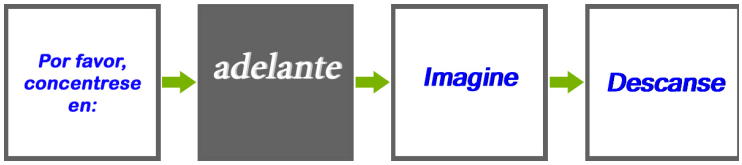


Figura 8.1: Secuencia presentada en pantalla para la adquisición de registros del habla imaginada

El formato de cada observación está dado por un vector que contiene la señal registrada por los canales F3, F4, C3, C4, P3 y P4 de 4 segundos cada una, concatenadas en dicho orden, determinando la longitud de dicho vector en 24579 muestras. Las etiquetas de cada registro de habla imaginada indican la clase correspondiente a la palabra imaginada, y también la presencia o no de artefactos oculares. Finalmente se contó con una cantidad de entre 90 y 230 ejemplos por sujeto aproximadamente, las cuales se encontraban relativamente balanceadas por clases con lo cual resultaba una cantidad en promedio de entre 15 y 40 ejemplos por clase aproximadamente para cada sujeto.

Procesamiento de datos

El siguiente procedimiento fue aplicado sobre los datos registrados, por lo cual se parte de este resultado para realizar los experimentos:

1. *Referencia:* Como referencia se utilizó un electrodo en la apófisis mastoides izquierda, mientras que el electrodo de la mastoides derecha se utilizó como tierra para el equipo de adquisición.

2. *Filtrado:* Las señales fueron pre-procesadas con filtros FIR, fijando una frecuencia de paso inferior en 2Hz para el filtro pasabajo de orden 372 y una superior en 40Hz para el filtro pasaalto de orden 1024.

3. *Submuestreo:* Se realizó un procedimiento de decimación para reducir la frecuencia de muestreo de 1024Hz a 128Hz.

4. *Selección de electrodos:* Se utilizaron los 6 canales disponibles para los 8 electrodos del sistema de adquisición.

5. *Construcción del vector de características:* Se concatenaron las muestras de cada electrodo consecutivamente para formar un vector de características unidimensional.

A partir de ese resultado, se dispuso lo siguiente:

- *Remoción de artefactos:* Dado que el corpus indica explícitamente cuáles son los ejemplos con artefactos oculares presentes, para este trabajo se procedió a ignorarlos de forma que sólo se utilicen aquellos que no estén contaminados con dichos artefactos.

- *Escalado:* Cada una de estos vectores se escaló al intervalo $[-1, 1]$.

- Se optó por utilizar sólo aquellos ejemplos relativos a palabras en lugar de vocales, abarcando 6 clases (*arriba*, *abajo*, *izquierda*, *derecha*, *adelante* y *atrás*) que además se relacionan con comandos para sistemas BCI.
- Por cada uno de los 15 sujetos tratados, se dividieron los datos en conjuntos de entrenamiento (50 % del total), validación (20 %) y prueba (30 % restante). Dado que la cantidad de ejemplos es escasa, se priorizó una proporción mayor en el conjunto de prueba para que la evaluación tenga el mayor respaldo posible en términos de muestras.

Experimentos

Como se mencionó en el trabajo original de referencia, aún reduciendo bastante la dimensionalidad de los datos crudos mediante decimación (en un 87,5 % aproximadamente) subsiste el problema de lidiar con datos de alta dimensión, lo cual dificulta el tratamiento en términos computacionales. Es por eso que resulta deseable que, a la hora de modelar un clasificador, se utilicen datos en menor dimensión conservando características que permitan desempeñar lo mejor posible la tarea de clasificación.

Tal como fue mencionado en la Sección 2.3.4, el método PCA es uno de los más utilizados para realizar reducción de dimensiones en un conjunto de datos, pero también los autocodificadores mostraron utilidad en esa tarea y hasta superando el desempeño de PCA en diversos casos. Por lo tanto en este trabajo se propuso comparar ambos métodos para lograr una dimensionalidad menor sobre los datos de habla imaginada, y que además el resultado sea una buena base de características para modelar un clasificador sobre las 6 clases elegidas. En base a esto, para la experimentación se dispuso lo siguiente:

- Para elegir la dimensión objetivo en la reducción, se utilizaron dos criterios: a) la proporción acumulada de la variación explicada por las componentes elegidas del PCA (ver detalles en la Sección 2.3.4), para tener una noción de cuánta información se retiene del conjunto original de datos; b) el porcentaje de reducción, para tener idea del grado de compresión logrado. Por lo tanto, se escogieron las siguientes configuraciones en las pruebas:

  a) **99 % de variación**: 70.64 % de reducción a 902 dimensiones.
  b) **95 % de variación**: 82.71 % de reducción a 531 dimensiones.
  c) **90 % de variación**: 88.21 % de reducción a 362 dimensiones.
  d) **80 % de variación**: 93.13 % de reducción a 211 dimensiones.
  e) **70 % de variación**: 95.60 % de reducción a 135 dimensiones.

  Por lo tanto, para el método PCA se eligió en cada caso un k igual a la cantidad de dimensiones a reducir, y para modelar los autocodificadores se disponía que la capa oculta tenga la correspondiente k cantidad de unidades. Notar que se decidió abarcar configuraciones hasta alcanzar un 95 % de reducción como grado de compresión suficiente.

- Para cada uno de los casos anteriores se escogió como clasificador una regresión logística multi-clase o *softmax*, de forma que se valide la calidad

de las características preservadas en los datos reducidos en términos de poder realizar una clasificación sobre ellos.

- En términos de regularización del autocodificador, se eligió una mayor penalización de la red mediante la norma L1 que tiende a lograr mayor raleza en las representaciones codificadas, lo cual se ha estudiado que es favorable para la extracción de características [**?**]. Por lo tanto, se utilizó un λ_1 igual a 0.001 y un λ_2 de 0.0003.
- La activación utilizada para la codificación fue una ReLU, mientras que para la decodificación se utilizó como función la tangente hiperbólica dado que su imagen cae en el rango $[-1, 1]$ al igual que los datos de EEG usados.
- Por otro lado, a partir de la menor dimensionalidad obtenida, se construyó un clasificador más sotisficado mediante una red neuronal cuya arquitectura estaba compuesta de las siguientes dimensiones: 135, 200, 100, 50, 6. Además, en su configuración se estableció como activación una ReLU y se utilizó el algoritmo DropOut con un ratio de 0.2 en la entrada y 0.5 en todas las capas ocultas.
- La optimización de todos estos modelos se realizó mediante el algoritmo Adadelta, siguiendo como criterio de corte un máximo de 20 iteraciones en forma local y de 100 iteraciones en forma global. La función de consenso para la mezcla de modelos en el entrenamiento distribuido sigue una ponderación lineal (denominada "w_avg" en Learninspy) sobre los valores obtenidos en la función de costo.
- Se utilizó el algoritmo PCA implementado en el módulo *utils.feature* de Learninspy, aplicando *whitening* y estandarización a los datos (lo cual suele denominarse ZCA, como se explicó en la Sección 2.3.4).

Resultados

En este caso de aplicación se planteó conseguir una reducción de dimensiones en los datos lo suficientemente buena como para lograr que la clasificación de las palabras imaginadas sea lo mejor posible. Para medir el desempeño de esta última tarea, se utilizaron las métricas de clasificación multi-clase explicadas en la Sección 2.1.3, y se usaron métricas de regresión para conocer el ajuste de los autocodificadores sobre los datos a comprimir.

Respecto al ajuste de cada AE, se observó que el valor de R^2 obtenido para los conjuntos de validación variaba entre 0.45 y 0.6 aproximadamente. Esto indica que la reconstrucción de las entradas no resultaba suficientemente buena, aunque tampoco podía mejorar significativamente al prolongar la optimización ya que la evolución del ajuste era casi logarítmica, como puede verse en la Figura 8.2 para uno de los AEs.

Es propicio aclarar que en el trabajo original, partiendo de los datos con decimación, se realizó una extracción de características mediante la trasformada de Wavelet discreta (DWT) para obtener un vector de 6 características por cada entrada. A partir de ello se obtuvo la mejor clasificación usando un modelo RandomForest con 200 árboles que elegían 5 características al azar, el cual era

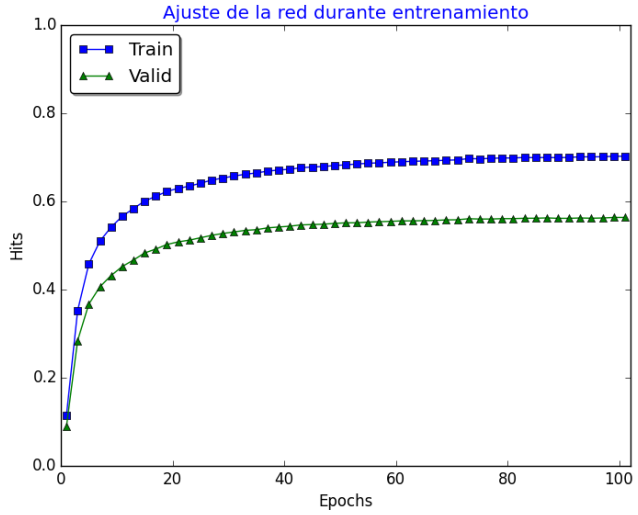


Figura 8.2: Ajuste del AE-211 sobre los datos de todos los sujetos durante el entrenamiento, en términos del coeficiente R^2.

ajustado sobre los 3 primeros sujetos y luego evaluado con los 12 restantes. La exactitud obtenida en dicha clasificación fue de $0{,}1832 \pm 0{,}0155$, lo cual supera al azar o chance que es de $1/6 \approx 0{,}1666$ [**?**].

Tabla 8.1: Resultados de clasificación, en promedio de todos los sujetos, obtenidos de modelar una regresión softmax sobre los datos en distintas dimensionalidas logradas por PCA y AE

	902 dim		**531 dim**		**362 dim**		**211 dim**		**135 dim**	
	PCA	AE	PCA	AE	PCA	AE	PCA	AE	PCA	AE
P_{avg}	0.1805	0.1852	0.1452	0.1737	0.1280	0.1587	0.1496	0.1521	0.1587	0.1793
R_{avg}	0.1676	0.1802	0.1416	0.1687	0.1381	0.1414	0.1433	0.1516	0.1570	0.1615
$F1_{avg}$	0.1676	0.1822	0.1403	0.1690	0.1323	0.1437	0.1460	0.1508	0.1527	0.1687
Acc_{avg}	0.1641	**0.1827**	0.1445	**0.1733**	0.1403	0.1428	0.1453	0.1554	0.1520	0.1640

Tabla 8.2: Resultados de clasificación, promediando por sujetos, obtenidos por la red neuronal a partir de los datos reducidos a 135 dimensiones con un AE

Métrica	**Total en sujetos**
Precision	$0{,}2820 \pm 0{,}1400$
Recall	$0{,}1898 \pm 0{,}0335$
F1-Score	$0{,}2160 \pm 0{,}0588$
Accuracy	$0{,}1905 \pm 0{,}0373$

En la Tabla 8.1 se realiza la comparación de los métodos usados para reducir dimensiones sobre los datos, por cada una de las configuraciones elegidas, en términos de las distintas medidas para evaluar el mismo tipo de clasificador. En todos los casos, se obtiene en promedio mejores resultados de cualquier métrica

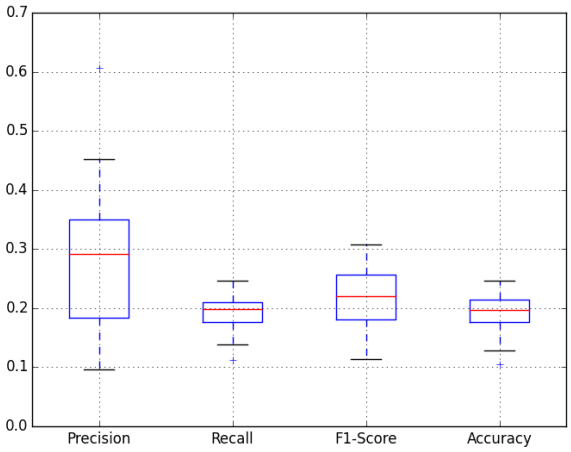


Figura 8.3: Diagramas de caja y bigotes para cada métrica de clasificación utilizada, construidos en base a todos los resultados de modelar redes neuronales por cada sujeto sobre los datos con mayor reducción de dimensionalidad.

sobre los datos obtenidos por la reducción de dimensiones logradas por un AE respecto a los de un PCA. Además, la exactitud o *accuracy* promedio que logra supera a la chance o azar en dos de cinco experimentos.

Finalmente, partiendo de los datos con menor dimensionalidad lograda se entrenó una red neuronal profunda sobre todos los sujetos. En la Tabla 8.2 se presentan los resultados de clasificación, los cuales también se representan de otra forma en la Figura 8.3 para dar mayor detalle. Se puede apreciar que este último clasificador mejora bastante los resultados de la regresión softmax, otorgando una exactitud que supera al azar en promedio sobre todos los sujetos y que es levemente mejor que la reportada en el trabajo original (aproximadamente un 2 % mayor, aunque sin utilizar el mismo conjunto de prueba).

Conclusiones

En base al resultado de la experimentación explicada para este segudo caso de aplicación tratado, se refuerzan las afirmaciones expresadas en otros trabajos acerca de que los autocodificadores pueden obtener mejores representaciones de los datos que el método PCA en tareas de compresión. Si bien el ajuste obtenido sobre los datos no era bueno, se puede ver que los conjuntos reducidos por cada AE parecían tener mejores características respecto a PCA que permitían lograr un mejor desempeño de clasificación para una simple regresión *softmax*.

Aunque los mejores resultados de clasificación alcanzados no se consideran buenos, se puede apreciar que se comparan con el estado del arte en la base de datos usada sin comprender una metodología complicada para obtenerlos. A partir de ello, se puede ver viable el enfoque de extraer características mediante aprendizaje profundo para la clasificación del habla imaginada.

Parte IV

Conclusiones y discusión

Esta última parte de la tesis está destinada a dar un cierre del trabajo, expresando las conclusiones finales que se obtuvieron respecto al desempeño del sistema, sus aplicaciones realizadas, y las posibles mejoras a realizarse en un futuro.

Capítulo 9

Conclusiones generales

El final de una obra debe hacer recordar
siempre el comienzo.

Joseph Joubert

Resumen: Este capítulo está destinado a plasmar conclusiones finales de todo el trabajo realizado. Las mismas se refieren al valor que se obtiene de este proyecto, así como las fortalezas y debilidades identificadas de la metodología seguida. Aquí se validan los logros del trabajo respecto a lo planificado, y se discuten futuros trabajos posibles a realizar sobre el sistema generado.

Corolarios del proyecto

En este proyecto se pretendía lograr un producto de software que sirva como herramienta para modelar soluciones sobre problemáticas complejas, utilizando para ello ciertos algoritmos de aprendizaje profundo que puedan distribuir su cómputo en la infraestructura que se disponga.

Un valor importante a destacar respecto al desarrollo de este trabajo es el aprendizaje de nuevas tecnologías que involucró: desde las herramientas de software utilizadas para construir y validar el framework (e.g. Apache Spark, ecosistema SciPy, herramientas de testeo e integración continua) hasta los algoritmos de aprendizaje maquinal y *deep learning* implementados, cubriendo todos los aspectos necesarios para proveer un framework completo (e.g. pre-procesamiento de datos, configuración de modelos, optimización, etc.). Esto mismo se conglomera en un sistema que puede apreciarse no sólo a nivel personal, sino también por las comunidades respectivas a las tecnologías abarcadas (disponiendo de una nueva herramienta gratuita que ofrece soluciones modernas), y a la institución *sinc(i)* que avala este proyecto ya que el mismo se relaciona con las disciplinas que actualmente trabajan allí (e.g. aprendizaje maquinal y sistemas BCI).

Para entender la concepción de Learninspy como un auténtico proyecto de Ingeniería en Informática, se visualiza en la Figura 9.1 las contribución de cada

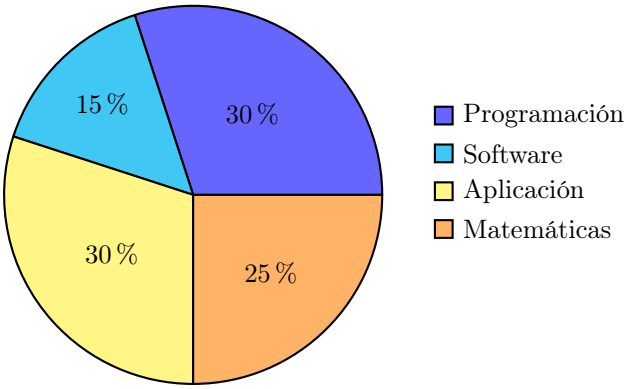


Figura 9.1: Gráfico de torta con las contribuciones de cada área curricular hacia el proyecto desarrollado.

área de asignaturas dictadas en la carrera hacia la ejecución del presente proyecto, las cuales son calculadas de forma subjetiva por el autor. Cada una de estas áreas se justifican de la siguiente forma:

1. **Programación:** Para implementar cada componente del sistema, combinando distintas tecnologías y paradigmas de programación. Las asignaturas destacadas de este área son: Tecnologías de la Programación, Algoritmos y Estructuras de Datos, Programación Orientada a Objetos y Fundamentos de Programación.

2. **Software:** Para gestionar el proyecto, desde el diseño hasta la ejecución, así como también aplicar buenas prácticas para el control de calidad. Aquí se consideran las contribuciones de las siguientes asignaturas: Ingeniería en Software I y II, y Administración de Proyectos de Software.

3. **Aplicación:** Para entender conceptos fundamentales en los que se basa el diseño del framework y sus funcionalidades, así como también saber aprovecharlos de forma correcta a la hora de definir la experimentación. Se destacan en dicha área estas materias: Inteligencia Computacional y Procesamiento Digital de Señales.

4. **Matemáticas:** Para entender a bajo nivel la teoría subyacente en las funcionalidades desarrolladas, lo cual es necesario conocer para poder implementarlas correctamente. Algunas de las asignaturas más importantes son: Álgebra lineal, Cálculo I, Estadística y Matemática Básica.

Respecto a los objetivos definidos en la Sección 1.3, se considera que se pudo cumplir con lo planificado ya que el framework desarrollado tiene la capacidad para distribuir su cómputo en forma local y a nivel clúster, y a su vez fue aplicado en las problemáticas de electroencefalografía definidas de manera correcta. Específicamente, esto se manifiesta de la siguiente forma:

- Se logró implementar el motor Apache Spark para que el sistema pueda distribuir su cómputo en sus principales funcionalidades, y esto fue validado tanto de forma local como a nivel clúster en los experimentos explicados en la Sección 5.3.2.

- Mediante una validación efectuada en usuarios (tanto los directores de este proyecto como otros colegas), se pudo determinar que las interfaces para interactuar con el framework son adecuadas y están suficientemente documentadas como para utilizar y/o modificar dicho sistema sin inconvenientes. Dichos usuarios no contaban con experiencia en cómputo paralelo ni Apache Spark, por lo cual se corrobora que el sistema abstrae correctamente esta propiedad para su uso tal como se pretendía.
- En ambas problemáticas de electroencefalografía definidas, se pudo ejecutar la experimentación planteada y se logró en ambos casos obtener un buen desempeño respecto al azar e incluso en comparación a los trabajos de referencia. En base a esto, y la experimentación realizada en la Sección 5.3.2 sobre otros corpus de datos, se puede validar la potencialidad de Learninspy como utilidad para resolver problemáticas complejas.
- Cumplimiento de todos los requisitos con prioridad deseada o esencial que fueron especificados en la ERS del proyecto, quedando sin cumplir únicamente los siguientes dos requisitos con baja prioridad: RF009 y RF011.

Trabajos futuros

A partir de todo el trabajo realizado, también se concluye que el software desarrollado tiene diversos aspectos a perfeccionar en términos de desempeño y utilidad. Específicamente, se consideran las siguientes cuestiones a mejorar en un futuro:

- Incorporación de nuevas características y tecnologías al framework mencionadas en la Sección 1.5.2.
- En base a la deficiencia explicada en la Sección 5.3.2, se concluye que es necesario mejorar las capacidades para realizar cálculos algebraicos en forma eficiente. Esto resulta crucial para que la herramienta tenga buen desempeño en problemáticas que involucran procesamiento de datos con alta dimensionalidad.
- Explorar e investigar más acerca de la propuesta de entrenamiento distribuido, desde la validación del concepto de función de consenso hasta los grados de libertad que comprenden dicho esquema.

Por otro lado, es preciso destacar la aplicación de Learninspy fuera del contexto de este proyecto para una aplicación de seguridad informática, en la cual se utilizaron autocodificadores para modelar una línea base del tráfico generado por un sensor de seguridad y en base a ello poder filtrar eventos de dicho tráfico [**?**]. En dicha aplicación se obtuvieron buenos resultados sobre una base de datos públicas, y se pretende evolucionar el enfoque para ser aplicable sobre datos reales generados en organizaciones de gran magnitud.

Dado que las tecnologías comprendidas por el framework se siguen desarrollando y mejorando cada vez más, logrando con ello diversas aplicaciones de forma exitosa, se puede concluir finalmente que Learninspy provee las bases para desarollar de forma escalable distintos algoritmos de *deep learning* y puede seguir mejorando sus capacidades para ser una herramienta flexible de personalizar y simple de utilizar en tareas complejas.

Parte V

Apéndices

Apéndice A

Algoritmos

Algorithm 2 Algoritmo de retropropagación para redes neuronales

Require: Matrices de pesos sinápticos W, vectores de bias b, nºde capas L.
  function BACKPROPAGATION(x, y) ▷ *Parámetros*: Patrón de entrada x; salida deseada y.
    1) Entrada:
    $a^{(1)} = f(x)$ ▷ Calcular la activación para la capa de entrada
    2) Paso hacia adelante:
    for $l = 2, 3, \ldots, L$ **do**
      $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$ ▷ Salida lineal de la capa.
      $a^{(l)} = f(z^{(l)})$ ▷ Salida activada de la capa.
    end for
    3) Error en salida
      $J = loss(a^{(L)}, y)$ ▷ Aplicar función de costo
    $\nabla_a J = d\_loss(a^{(L)}, y)$ ▷ Gradiente del costo respecto a la activación.
    $\delta^L = \nabla_a J \odot f'(z^{(l)})$ ▷ Computar el vector derivada de la salida
    4) Retropropagar el error y computar gradientes:
    for $l = L-1, L-2, \ldots, 2$ **do**
      $\delta^{(l)} = (W^{(l+1)})^\top \delta^{(l+1)} \odot f'(z^{(l)})$
      $\nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^\top$ ▷ Respecto al parámetro W de la capa l
      $\nabla_{b^{(l)}} J = \delta^{(l+1)}$ ▷ Respecto al parámetro b de la capa l
    end for
    return $J, \nabla_W J, \nabla_b J$ ▷ Devuelve el costo de la red y los gradientes
  end function

Algorithm 3 Salida de una capa de neuronas a la que se le aplica Dropout

Require: Capa de neuronas $l_{W,b}$.

function DROPOUTPUT(x, p) ▷ *Parámetros*: patrón de entrada x; probabilidad de activación $p \in (0, 1)$.

$a = output(l_{W,b}, x)$ ▷ Computar salida de capa $l_{W,b}$ ante la entrada x.

$v = random(size(a))$ ▷ Generar vector del mismo tamaño que a, con valores aleatorios entre 0 y 1.

$m = v > p$ ▷ Calcular máscara binaria del vector v, donde se asigna 1 a los valores de v mayores a p y 0 al resto.

$d = (a \odot m)/p$ ▷ Aplicar máscara binaria sobre el vector de activaciones, escalando a p.

return d, m ▷ Devuelve la salida de la red con Dropout aplicado, y la máscara binaria correspondiente.

end function

Lista de acrónimos

Parte VI

Anexos