Trabajo Fin de Máster

Introducción a los métodos Deep Learning basados en Redes Neuronales

Edward Joseph Velo Fuentes

Máster en Técnicas Estadísticas

Curso 2019-2020

Propuesta de Trabajo Fin de Máster

Título en galego: Introducción aos métodos Deep Learning basados en Redes Neuronais
Título en español: Introducción a los métodos Deep Learning basados en Redes Neuronales
English title: Introduction to Deep Learning methods based on Neural Networks
Modalidad: Modalidad A
Autor/a: Edward Joseph Velo Fuentes, Universidade da Coruña
Director/a: Manuel Febrero Bande, Universidade de Santiago de Compostela; ,
Breve resumen del trabajo: Los métodos de Deep Learning están alcanzando mucha notoriedad desde el área de ciencias computacionales pero su conocimiento en el ámbito estadístico es mucho más reducido. En este trabajo se propone una revisión crítica de los procedimientos más habituales en Deep Learning para regresión y clasificación en base a estudios de simulación y análisis de conjuntos de datos bien conocidos con el objeto de elaborar una serie de recomendaciones estadísticas sobre su uso y entender el impacto de las selecciones del usuario en el modelo final.

Don Manuel Febrero Bande, Catedrático de la Universidade de Santiago de Compostela informa que el Trabajo Fin de Máster titulado

Introducción a los métodos Deep Learning basados en Redes Neuronales

fue realizado bajo su dirección por don Edward Joseph Velo Fuentes para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, dan su conformidad para su presentación y defensa ante un tribunal.

En A Coruña, a 8 de Septiembre de 2020.

El director:	El autor:
Don Manuel Febrero Bande	Don Edward Joseph Velo Fuentes

Índice general

Introducción IX

1. Un acercamiento al Machine Learning 1

1.1. Definiendo el Machine Learning . . . 2

1.1.1. La tarea . . . 2

1.1.2. La medida de rendimiento . . . 3

1.1.3. La experiencia . . . 3

1.2. Machine Learning desde la perspectiva estadística . . . 4

1.2.1. Una traducción en Estadística de conceptos esenciales de Machine Learning . . . 5

1.2.2. La descomposición sesgo-varianza . . . 8

1.2.3. La generalización: Sobreajuste e infrajuste. . . . 9

1.3. El aprendizaje basado en un gradiente . . . 11

2. Redes neuronales 13

2.1. Definición de una red neuronal . . . 13

2.1.1. Una selección de funciones de activación . . . 15

2.2. El algoritmo *backprop* . . . 19

2.3. Interpretación geométrica del funcionamiento del algoritmo de optimización . . . 21

2.4. Algunos algoritmos de optimización posibles . . . 24

2.4.1. Stochastic Gradient Descent . . . 24

2.4.2. RMSprop . . . 24

2.4.3. Adam . . . 25

3. Los modelos Deep Learning 27

3.1. Problemas típicos en Deep Learning . . . 28

3.1.1. Cuándo suministrar nuevos datos al modelo . . . 28

3.1.2. Los problemas del gradiente explosivo y atenuado . . . 28

3.1.3. La elección de hiperparámetros 29
3.2. Regularización en los modelos Deep Learning 31
3.2.1. Dropout 31
3.2.2. Early stopping 32
3.2.3. Capas convolucionales 33
3.2.4. Aumento de datos o *data augmentation* 37
3.2.5. Max y Average pooling 40
3.2.6. Penalización de la norma L1 y L2 42

4. Caso práctico **45**
4.1. Criterio de evaluación e hiperparámetros elegidos 45
4.2. Pautas para elegir un modelo base de clasificación de imágenes 47
4.3. Evaluación de modelos Deep Learning 49
4.3.1. Estimación con distintas técnicas de regularización 49
4.3.2. Uso de redes pre-entrenadas: VGG-16 53
4.4. Resultado de las predicciones de los modelos 55

5. Conclusiones **59**

Introducción

A finales del siglo XX se ha desarrollado en el campo de las ciencias computacionales un nuevo tipo de modelo estadístico cuya estructura se inspira en las conexiones neuronales del cerebro humano. Este tipo de modelos, denominados modelos *Deep Learning*, estiman propiedades de un conjunto de datos utilizando elementos básicos conocidos como *neuronas*, las cuales pueden ser agregadas sin límite de cantidad y tienen la posibilidad de recoger numerosos patrones no lineales entre ellas. Estas cualidades permiten a los modelos Deep Learning tener buenos resultados en áreas caracterizadas por el empleo de datos de alta dimensionalidad, tales como el reconocimiento de voz, la identificación de objetos en imágenes o incluso la generación de nuevos rostros a partir de información sobre reconocimiento facial. Desde un punto de vista estadístico, la principal cuestión del Deep Learning reside en la elevada parametrización que suelen contener, recurriendo asiduamente a la estimación de millones de parámetros en la mayoría de las tareas que abarcan, por lo que es un área donde es recurrente que existan problemas de generalización en las predicciones. Otra cuestión también importante es que no hay una teoría lo suficientemente consolidada detrás del funcionamiento del Deep Learning, por lo que las recomendaciones generales para su buen uso suelen fundamentarse en la "prueba y error" en base a distintas arquitecturas propuestas, siendo el resultado de varias competiciones el referente más utilizado a la hora de diseñar estos modelos. Conviene entonces señalar, en la medida de lo posible, las indicaciones más recomendadas a la hora de utilizar uno de estos modelos.

En este trabajo realizamos un acercamiento a los modelos Deep Learning. En primer lugar, introducimos al lector en los conceptos y la terminología del *Machine Learning*, campo que engloba al Deep Learning, con un enfoque en una traducción de los mismos en el ámbito de la Estadística, explicando la definición de un modelo Machine Learning. En segundo lugar, introducimos los modelos de tipo red neuronal, utilizando como ejemplo una red neuronal sencilla de tres capas, para ejemplificar así su funcionamiento y presentando varias opciones dentro de sus posibles configuraciones. En tercer lugar, generalizamos lo aprendido de la red neuronal sencilla al ámbito del Deep Learning, enumerando varios de sus problemas y formas de solucionarlos o atenuarlos con varias técnicas de regularización distintas.

Finalmente, demostramos un caso práctico del uso de Deep Learning en la tarea de clasificación de imágenes, con el objetivo de comprobar la utilidad de distintas técnicas de regularización en la práctica sobre estos modelos.

Capítulo 1

Un acercamiento al Machine Learning

En las últimas décadas, el desarrollo tanto de la estadística como de la informática, entre otras disciplinas, han generado en conjunto un campo de estudio dedicado al aprendizaje de patrones mediante el uso de algoritmos computacionales. Este campo, conocido como *Machine Learning* (ML), engloba algoritmos que se caracterizan por dos procesos: primero, la estimación numérica de parámetros de un modelo estadístico mediante la minimización de una medida del error de predicción, y segundo, el uso automático de esa misma medida por el propio algoritmo para *aprender* a mejorar su capacidad predictiva, actualizando los parámetros en base a la información obtenida. El área del ML ha dado lugar a numerosos modelos e incluso a su propia terminología, que en la mayoría de los casos utilizan las mismas definiciones del campo de la Estadística pero con distinto nombre. Dentro de este mismo campo, un subconjunto de modelos conocidos como redes neuronales ofrecen una estructura inspirada en las conexiones neuronales del cerebro humano, esto es, elementos llamados neuronas que están conectadas entre si y que realizan expansiones básicas de los datos en base a combinaciones lineales, con la posibilidad de aplicar funciones que pueden captar relaciones complejas.

En este capítulo pretendemos hacer una introducción al *Machine Learning*. En el apartado 1.1 definimos este campo y explicamos parte de su filosofía. En el apartado 1.2 presentamos un glosario para traducir sus principales conceptos al campo de la estadística. Finalmente, en el apartado 1.3 explicamos el procedimiento a partir del cual se estima un modelo ML basado en el cálculo de un gradiente y exploramos algunas cuestiones relacionadas.

1.1. Definiendo el Machine Learning

El campo del Machine Learning puede definirse como un tipo de Estadística Computacional con un enfoque de aprendizaje mediante algoritmos informáticos. Más en concreto, estos algoritmos aprenden patrones de los datos, a partir de los cuales pueden realizar predicciones. Existe entonces la cuestión de qué significa realmente que estos algoritmos *aprendan*. Podemos recurrir en este caso a la definición de (Mitchell 1997): Se dice que "*un programa informático aprende de una experiencia con respecto a un conjunto de tareas y de una medida de rendimiento si su rendimiento en las determinadas tareas, calculado mediante dicha medida de rendimiento, mejora con la experiencia*". En el entorno de la Estadística Computacional podríamos adaptar esta definición como sigue: "*Proceso que estima una tarea a partir de la información y mejora la estimación cuanta más información recibe, siendo estas mejoras a su vez moduladas con una medida de rendimiento*". Como se puede observar, hemos resaltado tres conceptos claves en esta definición que son cruciales para entender el proceder de cualquier algoritmo de Machine Learning: La *tarea*, la *medida del rendimiento* y la *experiencia*. En los siguientes subapartados abordamos cada uno de estos conceptos.

1.1.1. La tarea

Una tarea es cualquier objetivo que un algoritmo de *Machine Learning* pueda abordar. Algunas de estas tareas son las siguientes:

- Clasificación de imágenes
- Reconocimiento de voz
- El caminar de un robot
- La identificación de señales y marcas viales para la conducción autónoma de un vehículo
- etc...

Cada tarea que puede ser objeto de un modelo ML necesita de información específica que normalmente ha de ser previamente procesada. Por ejemplo, en el caso de la clasificación de imágenes, es común la codificación de las mismas en matrices con números enteros en el rango $[0, 255]$, que pertenece al denominado código RGB[1]. Por el otro lado, el reconocimiento de voz requiere de la traducción de un archivo de audio en una secuencia ordenada de caracteres, donde no sólo el modelo ML logre identificar palabras sueltas sino también que perciba el orden de las mismas para la predicción.

[1] En el apartado 3.2.3 abordaremos más en profundidad este concepto

1.1.2. La medida de rendimiento

En el ámbito de la Estadística, es común el comparar la calidad de varios modelos distintos en cuanto a su capacidad de predicción mediante una medida del error, normalmente referida como *función de coste*. La función de coste es la *medida del rendimiento* que antes comentamos, la cual se utiliza para valorar si un modelo ML tiene mayor o menor rendimiento a la hora de realizar predicciones. El escoger una medida de rendimiento adecuada depende de la tarea a realizar. Algunas de las más utilizadas son las siguientes:

- **Error Cuadrático Medio (ECM)**: $\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$, donde N es el tamaño de la muestra utilizada para la predicción, y_i es la realización i-ésima de la variable respuesta y $\hat{y}_i$ una predicción en la misma observación. Para predicciones cuyo objetivo sea numérico. Es una medida muy utilizada, aunque es poco robusta frente a datos atípicos.

- **Error Absoluto Medio (EAM)**: $\frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$. También para predicciones con objetivo numérico. Es una medida robusta frente a atípicos. No es derivable.

- **Pérdida de Poisson**: $\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i \log \hat{y}_i)$ Para predicciones con objetivo numérico donde se considere que la variable y provenga de una distribución de Poisson (ejemplo, número de personas que entran a comprar en un supermercado en un intervalo de tiempo concreto).

- **Deviance Binomial**: $\sum_{i=1}^{N}\log(1 + e^{-y_i \cdot \hat{y}_i})$. Para predicciones con objetivo categórico, cuando sólo hay dos categorías, donde $y_i \in \{-1, 1\}$ e $\hat{y}_i \in (-\infty, \infty)$.

- **Deviance Multinomial**: $-\sum_{k=1}^{K} I(y = \mathcal{G}_k)\log \hat{y}_k$, donde $\mathcal{G}_k$ es una de las K clases existentes a predecir, y es una realización muestral de la variable que se desea predecir, e $\hat{y}_k$ es igual a:

$$\hat{y}_k = \frac{e^{f(x)_k}}{\sum_{j=1}^{K} e^{f(x)_j}} \tag{1.1}$$

  donde $f(x)_k$ es la función que predice la clase k, y $\sum_{k=1}^{K}\hat{y}_k = 1$ e $i \in \{1, ..., k\}$. Para realizar predicciones con objetivo categórico de más de dos clasificaciones distintas.

1.1.3. La experiencia

Cuando decimos que un algoritmo Machine Learning *aprende de la experiencia*, nos referimos a la estimación de un modelo a partir de la información de la que disponemos. Esta información se materializa en la muestra de datos. En Estadística, la *experiencia* se entiende como la *información*, y esta se extrae de una muestra X que se compone de n observaciones muestrales y de p variables

aleatorias, esto es, $X \in \mathbb{R}^{n \times p}$. La muestra, a su vez, puede ser una fija o dinámica: O bien utilizamos un tamaño muestral fijo para el modelo, o bien podemos ir agregando nueva información al modelo para que siga *aprendiendo*. Se dice entonces que un modelo ML ha aprendido de una experiencia cuando luego se le suministran datos adicionales y es capaz de detectar patrones aprendidos a partir de la información suministrada.

Existen varias categorías de aprendizaje dentro de Machine Learning cuyo uso depende de los datos de los que dispongamos a mano. Las más conocidas son las tareas de *aprendizaje supervisado* y de *aprendizaje no supervisado*. En el primero, se conoce a priori una clasificación o etiqueta de cada observación muestral, que puede ser expresada tanto por una característica discreta como por una continua (por ejemplo, en tareas como la identificación de objetos en imágenes, etc...). En el segundo, se pretende conocer una función de densidad o de probabilidad de la matriz de diseño, para así extraer patrones interesantes acerca de la experiencia (como por ejemplo, en el análisis cluster para una muestra de clientes de una compañía, con la finalidad de encontrar grupos homogéneos de dichos clientes). La cuestión de por qué a un grupo se le denomina supervisado y al otro no tiene que ver con la identificación de la variable clasificatoria con un supervisor, el cual indica al algoritmo a qué categoría específica ha de asignarse cada ejemplo. Sin embargo, no existe una definición formal tanto del aprendizaje supervisado como del no supervisado, y es posible incluso encontrar algoritmos híbridos, como los de aprendizaje semi-supervisado, los cuales decidimos no incluir en este trabajo por simplicidad.

1.2. Machine Learning desde la perspectiva estadística

La ciencia del Machine Learning, al igual que otras ciencias, bebe de múltiples áreas del conocimiento científico. Además de la Estadística, recoge elementos de campos como la Teoría de la Complejidad Computacional o la Neurobiología. De manera paralela, se ha desarrollado también otro campo, conocido como Estadística Computacional, que es el uso de técnicas estadísticas aprovechando la capacidad de un ordenador para la estimación o inferencia de modelos. Tanto el ML como la Estadística Computacional se han visto beneficiadas de los últimos avances informáticos en poder de computación y del aumento masivo de la disponibilidad de datos en las últimas décadas. Ambos campos persiguen en general los mismos objetivos (la estimación de un modelo estadístico y sus posibles usos), y lo que diferencia a un campo del otro tiene mas bien que ver con su terminología, donde suele ser recurrente que ambas ciencias se refieran a un mismo concepto pero con nombres distintos.

En esta sección vamos a realizar un acercamiento al ML desde el punto de vista de la Estadística. En primer lugar, realizaremos una breve traducción de varios de los conceptos fundamentales que ambos

campos manejan pero que etiquetan de distinta forma, para después ver cómo se trata la cuestión del error de predicción en este ámbito y cómo hacen los propios modelos ML para estimar parámetros en base a este error.

1.2.1. Una traducción en Estadística de conceptos esenciales de Machine Learning

En este apartado se asumirá que el lector tiene nociones básicas de la Estadística Computacional: Esto es, tiene conocimientos acerca de inferencia y modelos estadísticos, lo cual implica conocer conceptos tales como la estimación del modelo, los contrastes de hipótesis, la predicción, la construcción de intervalos de confianza, etc... En base a lo que el lector conoce sobre Estadística, vamos a plantear un modelo de regresión general múltiple para un conjunto de datos, sin entrar en el resultado numérico de la estimación del modelo mismo, con la finalidad de traducir la terminología empleada para la descripción del problema y su solución al ámbito del ML.

Consideremos el siguiente modelo de regresión generalizada:

$$Y = m(X_1, ..., X_p) + \epsilon$$

donde Y es una variable dependiente, $X_1, ..., X_p$ son las covariables, ϵ es una perturbación aleatoria y $m()$ es la función de regresión a estimar. A partir del dataset *mtcars* (Henderson y Veterman 1981), que recoge datos extraídos de la revista Motor Trend US magazine en 1974, acerca de 11 variables distintas sobre 32 modelos de coches, queremos saber si la variabilidad de *millas recorridas por galón* (*mpg*) está explicada en alguna medida por la variabilidad de las cilindradas del motor (*cyl*) y la potencia en caballos brutos (*hp*). Esto es:

$$mpg_i = \beta_0 + \beta_1 cyl_i + \beta_2 hp_i + \epsilon_i \tag{1.2}$$

Nuestro objetivo es entonces obtener $\hat{m}()$ con la especificación indicada en 1.2, tal que $\widehat{mpg}_i = \hat{\beta}_0 + \hat{\beta}_1 cyl_i + \hat{\beta}_2 hp_i$, donde $\hat{\beta}_j, j \in [0, 1, 2]$ son los coeficientes a estimar, $\widehat{mpg}_i$ es la estimación i-ésima en promedio de la variable mpg, y cyl_i, hp_i son las observaciones muestrales de *cyl* y *hp* en i. Naturalmente, podríamos elegir otra especificación de $m()$ (por ejemplo, añadir cyl^2 como covariable) y también podríamos asumir uno u otro comportamiento de ϵ (como afirmar que su varianza sea homocedástica o heterocedástica), pero hemos optado por 1.2 por sencillez. Una vez estimado el modelo, podríamos hacer una serie de diagnósticos para comprobar su validez (normalidad de los residuos, bondad del ajuste, etc...), realizar inferencias acerca de sus coeficientes estimados o también predecir sobre datos no utilizados para la estimación del modelo.

El planteamiento que acabamos de realizar para un modelo de regresión podría utilizarse tanto en Estadística como en Machine Learning. No obstante, en ML suelen utilizarse términos propios que conviene introducir para acostumbrar al lector al lenguaje utilizado en el resto del trabajo. A continuación, traducimos algunos de los conceptos esenciales desde la Estadística hacia el ML:

- **Variables aleatorias**: A lo que conocemos como *variable aleatoria* en Estadística, en Machine Learning se le denomina *característica* o *feature*. En el problema que hemos planteado, *mpg*, *cyl* y *hp* son las características. Además, a la tarea de realizar transformaciones sobre dichas características se le denomina como *feature engineering*.

- **Observaciones muestrales o casos**: En Machine Learning se suele utilizar *ejemplo* para referirnos a una observación muestral, tal que $\vec{x}_i = (x_{i1}, ..., x_{ip})$, aunque también se puede utilizar *caso* al igual que en Estadística. Decimos entonces que el dataset *mtcars* se compone de 32 ejemplos o casos.

- **Datos según la predicción y la validación**: En la Estadística, estimamos un modelo a partir de una muestra que consideramos representativa de una población, y podemos luego realizar una predicción en datos ajenos a la mencionada muestra, la cual suele denominarse predicción *fuera de la muestra* o *out-of-sample*. En ML, a la muestra utilizada para la estimación del modelo se le denomina como *conjunto de entrenameinto* o *training set*, mientras que al subconjunto de datos que dedicamos para la predicción se le denomina *conjunto de test* o *test set*. La muestra de tamaño 32 que hemos utilizado de *mtcars* sería el conjunto de entrenamiento, puesto que se han utilizado todos estos datos para la estimación del modelo, y podríamos utilizar como conjunto de test nuevos coches de los que necesitaríamos información acerca de sus características *mpg*, *hp* y *disp*. Además, cuando queremos determinar ciertas elecciones para el diseño de nuestro modelo, utilizamos un procedimiento intermedio que depende de unos datos denominados *conjunto de validación* o *validation set*, que normalmente suele extraerse del propio conjunto de entrenamiento.

- **Variable respuesta y covariables**: Por ejemplo, en el modelo que hemos definido en 1.2, a la variable *mpg*, que es la variable respuesta, se le suele llamar *output* o *target*, mientras que a *cyl* y a *hp*, que son las covariables o regresores, se les denominan *inputs*. Si la predicción se hace sobre un objetivo categórico, como es el caso de una regresión logística, a la variable respuesta se le llama *etiqueta* o *label*.

- **Estimación**: La estimación de un modelo en ML suele también referirse como el *aprendizaje* de un modelo. Por lo tanto, estimar parámetros significa que el modelo *aprende* patrones, representados por dichas estimaciones.

- **Coeficientes de estimación (θ)**: A un coeficiente de un modelo ML cualquiera se le puede llamar *parámetro*, al igual que en Estadística. Particularmente, en las redes neuronales, que definiremos en el apartado 2.1, se utiliza *sesgo*[2] u *offset* en parámetros que representan un intercepto, mientras que a los parámetros que multiplican directamente a un input se les denomina *ponderaciones*.

- **Hiperparámetros**: Son los elementos del diseño de un modelo que pueden ser modificados u optimizados por el usuario. Este término se utiliza tanto en Estadística como en ML. Una forma de determinarlos es mediante el conjunto de validación al que hicimos referencia antes.

- **Grados de libertad consumidos**: En Estadística decimos que un modelo estadístico es más/menos complejo cuanto mayor/menor es el número de grados de libertad consumidos por el mismo. Este mismo concepto se utiliza en ML bajo el nombre de *capacidad* de un modelo (es decir, a mayor capacidad, mayor es la complejidad del modelo). En nuestro ejemplo de regresión, al definirse tres coeficientes β_i, decimos que se han consumido 3 grados de libertad de un total de 32.

- **Error de predicción:** En ML también se utilizan medidas de calidad de un estimador, como el Error Cuadrático Medio, solo que en este ámbito se denomina a este tipo de medidas como *función de coste*, y es la *medida del rendimiento* a la que hicimos referencia en el apartado 1.1.2 .

Existen múltiples hiperparámetros en *Machine Learning* con finalidades distintas y que no tienen traducción en la Estadística, y que conviene definir igualmente. Estos son:

- **Batch**: Posible submuestra de tamaño m entre las que se puede dividir una muestra de tamaño N, tal que $m < N$. Normalmente se procura que todos los batch obtenidos de una muestra tengan el mismo tamaño, de tal forma que existan $\frac{N}{m} \in \mathbb{N}$ *batches*.

- **Epoch o ciclo**: Intervalo que comprende desde una actualización de las ponderaciones de un modelo ML respecto de una observación (o *batch*) hasta la siguiente actualización sobre la misma observación. El usuario de un modelo puede elegir manualmente con cuantos *epochs* desea que el modelo estime sus parámetros[3]. Un *epoch* puede contener una o varias actualizaciones de los parámetros.

- **Iteración**: Intervalo que engloba una actualización de los parámetros. El número de intervalos es equivalente al número de *batches*.

[2] No confundir con el sesgo del intercambio sesgo-varianza, que introduciremos en el siguiente apartado.

[3] En lugar de la especificación de un numero de epochs, algunos modelos ML permiten que el modelo actualice parámetros hasta alcanzar un umbral, por ejemplo, si la minimización del error entre actualizaciones es menor que una cantidad ϵ

- **Learning rate**: Hiperparámetro que regula la velocidad a la que el modelo trata de optimizar su función de coste. En el apartado 1.3 explicamos en más detalle su importancia.

1.2.2. La descomposición sesgo-varianza

En la Estadística Computacional, cuando decidimos estimar un modelo estadístico, principalmente es para entender el comportamiento probabilístico de múltiples variables aleatorias y la posible relación que puede haber entre ellas. En algunos casos, puede que nos interese no sólo realizar inferencia sobre los estimadores obtenidos, sino también predecir valores utilizando datos ajenos a los del conjunto de entrenamiento para estimar los parámetros del propio modelo, o bien hacer una selección de covariables relevantes, X, que expliquen los movimientos de una variable aleatoria, Y, utilizando como referencia las denominadas medidas de bondad del ajuste. Todas estas posibilidades también existen en los modelos Machine Learning. Es debido a esto que también en ML es de interés conocer la naturaleza del error de predicción realizado tanto en el conjunto de entrenamiento como en el de test. Esta naturaleza se caracteriza por el balance sesgo-varianza o *bias-variance tradeoff*.

Supongamos que $g(\cdot, \mathcal{D})$ es un modelo estadístico (o *algoritmo de aprendizaje* en ML), donde $\mathcal{D}$ son el conjunto de datos muestrales a utilizar para el ajuste del modelo, y $\overline{Z}_1 \dots \overline{Z_n}$ son n observaciones diferentes de datos fuera de la muestra sobre los que evaluar una predicción. Podríamos entonces definir el Error Cuadrático Medio como:

$$ECM = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(g(\overline{Z_i}, \mathcal{D}) - f(\overline{Z_i}) - \epsilon_i\right)^2 \tag{1.3}$$

donde $\epsilon_i \sim N(0, \sigma)$, y_i son los valores de la variable dependiente a estimar en la muestra, n es el tamaño muestral, $\hat{y}_i$ son predicciones realizadas de la variable dependiente y f es la verdadera función que buscamos estimar con el modelo. Utilizamos el ECM y asumimos que $y \in \mathbb{R}$. Aunque existen otras opciones, con este ejemplo podemos explicar la intuición detrás del sesgo y de la varianza de manera que pueda generalizarse a otras alternativas. Asumiendo que el modelo puede ser ajustado para múltiples $\mathcal{D}$ distintos, necesitamos calcular $\mathbb{E}[ECM]$ sobre estas elecciones posibles, lo cual resulta en la siguiente descomposición:

$$\mathbb{E}[ECM] = \underbrace{\frac{1}{n}\sum_{i=1}^{n}\{f(\overline{Z_i}) - \mathbb{E}[g(\overline{Z_i}, \mathcal{D})]\}^2}_{\text{Sesgo }^2} + \underbrace{\underbrace{\frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\left[\{g(\overline{Z_i}, \mathcal{D}) - \mathbb{E}[g(\overline{Z_i}, \mathcal{D})]\}^2\right]}_{\text{Estimación}} + \underbrace{\frac{\sum_{i=1}^{n}\mathbb{E}\left[\epsilon_i^2\right]}{n}}_{\text{Perturbación}}}_{\text{Varianza}} \tag{1.4}$$

donde $\mathbb{E}[\epsilon_i] = 0$. El desarrollo matemático del ECM inicial hasta este punto, que decidimos obviar por simplicidad, se encuentra en (Aggarwal 2018). La ecuación 1.4 representa el *error de generalización* del modelo: Cuanto menor es este error, decimos que el modelo *generaliza* mejor, mientras que cuando el error aumenta, el modelo está entonces generalizando peor.

El sesgo es la medida en la que un modelo no recoge el término medio de la distribución de los datos sobre los que trabaja, produciendo así errores consistentes en las predicciones de la muestra *out-of-sample*, independientemente del set de datos de entrenamiento que utilicemos y de su tamaño. La varianza, en cambio, mide la incapacidad del modelo de estimar todos los parámetros de la función verdadera de una manera estadísticamente robusta, y suele aumentar cuando un modelo reduce su sesgo por completo en los datos de entrenamiento, dando lugar al problema de sobreajuste. En el apartado 1.2.3 entramos más en detalle sobre este concepto.

1.2.3. La generalización: Sobreajuste e infrajuste.

Hemos indicado anteriormente que un algoritmo de Machine Learning muestra mejor rendimiento cuanto menor error de test genere, sea cual sea la medida de rendimiento utilizada. Un elemento crucial para alcanzar este objetivo es que el algoritmo presente un error en el subconjunto de datos de entrenamiento que sea lo bastante pequeño como para devolver también el menor error de test posible. Si el algoritmo no logra reducir el error de entrenamiento lo suficiente, por lo que no ha aprendido adecuadamente patrones que pueda detectar en los datos de test, decimos que surge un problema de *infrajuste*, y de manera previsible el error de entrenamiento se extenderá a un mayor error en el set de test. Por el lado contrario, podría suceder que el error en el set de entrenamiento se volviese muy reducido o casi nulo, de tal forma que el modelo podría no adaptarse bien a patrones contenidos en los datos de test debido a su rigidez en el ajuste de los datos de entrenamiento, dando lugar a otro problema conocido como el *sobreajuste*.

Tanto en el concepto de infrajuste como en el de sobreajuste se hace la implicación de que una adaptación inadecuada del modelo a los datos de entrenamiento deriva en un rendimiento reducido en los datos de test, por lo que implícitamente se afirma que los datos tanto de entrenamiento como los de test tienen alguna relación, o lo que es lo mismo, que ambos conjuntos han de ser homogéneos entre sí. Si no se cumple esta condición, se produce una brecha entre el error de entrenamiento y el de test, y como consecuencia el modelo se volverá completamente inestable. Por lo tanto, el error esperado en datos de entrenamiento de un modelo escogido aleatoriamente ha de ser aproximadamente igual al error esperado de ese mismo modelo en unos datos de test, al haber seleccionado ambos conjuntos de una misma distribución. Sólo cuando se cumpla esta condición podemos comprobar si existe sobreajuste o

infrajuste, y aplicar el procedimiento adecuado para la mitigación de estos problemas.

Los problemas de sobreajuste e infrajuste pueden ser ejemplificados con la cuestión del intercambio entre sesgo y varianza en el error cuadrático medio que hemos introducido en el apartado 1.2.2. Por ejemplo, un modelo que muestre un sesgo nulo en las predicciones dentro del conjunto de entrenamiento probablemente no generalice bien en el set de test, debido a que la capacidad del modelo es demasiado elevada en relación a la complejidad de los datos, dando lugar al problema de sobreajuste. En el caso contrario, si un modelo contiene un elevado sesgo en sus predicciones, debido a que su capacidad es muy reducida en relación a la complejidad de los datos, estaríamos hablando de un problema de infrajuste, y tampoco se garantizaría en este caso que el error de predicción en el set de test sea bajo.

Algunas tareas típicas en Machine Learning, como es el reconocimiento de voz o la identificación de objetos en imágenes, se caracterizan por la necesidad de modelos con una capacidad muy alta para su correcta predicción debido a la complejidad de sus datos. A su vez, los modelos que están diseñados para adecuarse a datos de elevada dimensionalidad pueden estimar un número excesivo de parámetros con facilidad, dando lugar a problemas de sobreajuste. A continuación, en el apartado 1.2.3 definimos un concepto sobre la atenuación de este tipo de problemas.

La regularización en el aprendizaje

Ante un problema de sobreajuste, deseamos reducir la complejidad del modelo para aproximarlo a la complejidad de los datos mediante un procedimiento denominado *regularización*. Un ejemplo de regularización sería la penalización de la norma, que es una técnica que consiste en la reducción de la magnitud de los parámetros estimados mediante distintas formas con la finalidad de atenuar el sobreajuste. Esta regularización se define como:

$$\widetilde{J}(y, x, \theta) = J(y, x, \theta) + \lambda\Omega(\theta) \tag{1.5}$$

donde $J(x)$ es la función de coste, y es la variable a predecir, x es la matriz de diseño, θ es el conjunto de parámetros a estimar del modelo, $\Omega(\theta) > 0$ es la función de penalización de la norma, λ es un parámetro de penalización y $\widetilde{J}(x, \theta)$ es la suma de ambas funciones. Un aumento de λ incrementa la penalización, y como consecuencia reduce la complejidad del modelo estimado. La expresión $\Omega(\theta)$ puede contener tanto una penalización de la norma L^2 (o Ridge) como de la norma L^1 (o LASSO). Volviendo a la ecuación 1.5, en la penalización de L^2, tendríamos que $\Omega(\theta) = \|\theta\|_2 = \sqrt{\sum_i \theta_i^2}$, mientras que para la norma L^1, tendríamos $\Omega(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$. Ambas penalizaciones buscan reducir la magnitud de los parámetros calculados del modelo para intentar atenuar el sobreajuste, siendo la J^1 la penalización

más severa, pues puede llegar a desactivar parámetros ($\theta_i = 0$). Naturalmente, la intensidad de la regularización depende del valor de λ, donde si $\lambda = 0$ no existiría penalización, mientras que si λ fuese excesivamente elevada, la penalización dominaría a J en 1.5.

Existen otras formas de regularización que no implican una forma como la de 1.5. Algunas de estas técnicas serán presentadas para redes neuronales en el apartado 3.2.

1.3. El aprendizaje basado en un gradiente

La mayor parte de modelos Machine Learning utilizan la máxima verosimilitud para obtener estimadores de los parámetros, los cuales son consistentes ($\text{plim}_{m\to\infty} \hat{\theta}_m = \theta$). La consistencia de un estimador depende de si la complejidad del modelo a utilizar se adapta a la complejidad de los datos sobre la que se realiza la estimación. Si bien la propiedad de consistencia es importante, cabe señalar que varios estimadores consistentes pueden tener diferencias de eficiencia entre ellos, significando así que aún siendo consistentes, pueden diferenciarse en su capacidad de generalización en las predicciones.

En numerosos casos, los datos con los que trabajamos contienen relaciones no lineales en su distribución, por lo que la optimización por máxima verosimilitud de la función de coste puede convertirse en un problema no convexo (o de *difícil optimización* en lenguaje heurístico). Para las tareas que contengan este tipo de problemas, se suelen seleccionar *algoritmos de optimización* específicos que utilizan un cálculo basado en un gradiente. El objetivo es conocer cómo varía la función de coste en base a cambios en cada uno de los parámetros de θ, de manera que el algoritmo de optimización conozca *hacia qué dirección* y *en qué magnitud* ha de cambiar θ para seguir minimizando la función de coste. Este proceso es el *aprendizaje* del modelo ML al cual nos hemos referido antes.

El algoritmo de optimización más sencillo es el *gradient descent*, y está definido como sigue:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \widetilde{J}}{\partial \theta_t} \tag{1.6}$$

donde θ_t es el conjunto de parámetros en la iteración t del algoritmo para $1 \leq t \leq T$ (siendo T el número de iteraciones del algoritmo totales, escogido por el usuario del modelo), $\frac{\partial \widetilde{J}}{\partial \theta_t}$ es el gradiente de la función de coste con regularización respecto de los parámetros θ_t utilizando todo el set de entrenamiento y $\eta \in [0, +\infty]$ es el *learning rate*. Antes de comenzar el proceso de actualización en la ecuación 1.6, $\theta_{t=0}$ ha de ser elegido en función del algoritmo de optimización escogido [4].

[4] En el apartado 2.4 presentamos algunas opciones.

La finalidad del *learning rate* (η) en este contexto es la de modular la velocidad a la que θ_t converge al θ óptimo. La elección de un η correcto dependerá de la magnitud de $\frac{\partial J}{\partial \theta_t}$: Grandes valores del learning rate junto con la escala del gradiente pueden provocar que la optimización converja rápidamente en dirección del mínimo global pero que nunca llegue a alcanzarlo, mientras que valores pequeños produce una convergencia excesivamente lenta al óptimo. La intuición geométrica detrás de este problema se verá en el apartado 2.3.

En la práctica, pueden surgir problemas de optimización complejos para los que el *gradient descent* no es la mejor elección. En los apartados 2.2 y 2.4 veremos cómo sobrepasarlos.

Capítulo 2

Redes neuronales

Las redes neuronales son un tipo de modelos que aprenden representaciones en un conjunto de datos mediante el empleo de neuronas con numerosas interconexiones y el uso de algoritmos para incrementar el rendimiento del modelo. Este tipo de modelos son altamente personalizables, pues no hay a priori ningún límite para el número de neuronas que un modelo de este tipo pueda contener, lo cual implica que el modelo puede aceptar fácilmente una cantidad enorme de parámetros, siendo así susceptible al problema del sobreajuste. Con la finalidad de introducir al lector en las redes neuronales, utilizaremos en este apartado el modelo más sencillo posible, representado en la Figura 2.1 y configurado para realizar una tarea de predicción categórica sobre el conocido dataset *Iris*, donde se tratará de predecir la especie de una flor en base a las restantes variables explicativas del conjunto de datos.

En el apartado 2.1 definimos la red neuronal. En el apartado 2.2 abordamos el algoritmo que se encarga de propagar la información de la función de coste a todos los parámetros de la red neuronal, y en los apartados 2.3 y 2.4 mostramos una intuición gráfica del proceso de optimización y de algunas opciones existentes para abarcar dicho proceso.

2.1. Definición de una red neuronal

Una red neuronal es un modelo que se compone de un número determinado de elementos llamados *neuronas*, las cuales a su vez están organizadas en grupos denominados *capas*. Las neuronas son el elemento más básico de una red neuronal y su finalidad es la de aplicar una transformación a los datos recibidos en x con tal de predecir un valor continuo o categórico $\hat{y}$ en la última capa. En nuestro ejemplo utilizando Iris, $X \in \mathbb{R}^4$ son las variables *Sepal.Length*, *Sepal.Width*, *Petal.Length* y *Petal.Width*, mientras que la variable $Y \in \mathbb{R}^3$ corresponde a *Species*, el cual contiene 3 clases distintas

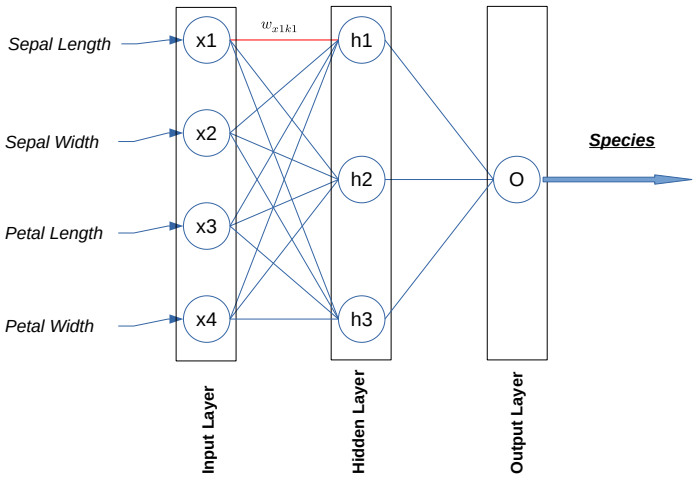


Figura 2.1: *Esquema de una red neuronal simple para la predicción utilizando el dataset Iris. En la output layer se aplica una función softmax, concretamente en la neurona O, devolviendo un vector de tamaño 3 y cuyos valores suman uno. La red decide la clasificación de x_i con el $\hat{y}_i$ de mayor valor. En rojo se encuentra la ponderación que se utiliza como referencia en la ecuación 2.9*

(*Setosa*, *Versicolor* y *Virginica*)

Las redes neuronales contienen tres tipos de capas según la posición en la que se encuentren dentro de la red: La capa de input o *input layer*, una o más capas ocultas o *hidden layers* y una capa final o *output layer*. La primera capa se ocupa de introducir en el modelo la información, mientras que las dos capas siguientes realizan transformaciones en la información que reciben y devuelven un resultado transformado. Toda red neuronal tiene una capa input y una output, pero puede disponer de una o más capas intermedias. En la Figura 2.1 puede verse el diseño de la red neuronal que hemos planteado para la clasificación de Iris, el cual se compone de una única capa intermedia.

Las capas también pueden ser clasificadas según la forma de la interconexión entre las neuronas. Cuando cada neurona de una capa intermedia se conecta a todas las neuronas de las capas anterior y posterior, como es el caso del ejemplo de este apartado, decimos que es una capa intermedia de tipo *denso* o capa densa, también llamada *fully connected layer*. En el apartado 3.2.3 exploraremos un tipo distinto de capa.

En base al diseño de la Figura 2.1, donde hay J neuronas en la capa input, H neuronas en la capa oculta y una neurona en la capa final, definimos la función de la red neuronal como:

$$f(x;\theta) = g_o\left(b_o + \sum_{h=1}^{H} w_{ko} g_k \left(b_k + \sum_{j=1}^{J} w_{jk} x_j\right)\right) \tag{2.1}$$

donde x_j es un vector que recoge una observación muestral de la variable j-ésima, g_r es una función correspondiente a la neurona r-ésima (k si pertenece a la capa oculta y o si pertenece a la capa final), b_r es el sesgo u *offset* de la neurona r-ésima y $w_{rs} \in \mathbb{R}$ es una ponderación que transmite un valor de la neurona r a la neurona s. Los parámetros w y b, que pertenecen a θ, se estiman por máxima verosimilitud, y como indicamos antes en 1.3, dependiendo de la tarea a realizar por el modelo el problema de optimización puede volverse no convexo.

A la función $g_r()$ se le denomina *función de activación*, y su elección depende de la tarea que queramos resolver con la red neuronal. A continuación detallamos varias de las opciones posibles para este tipo de funciones.

2.1.1. Una selección de funciones de activación

La función identidad

La función identidad se define como:

$$g(z) = z \tag{2.2}$$

donde z es la combinación lineal o *preactivación* que sucede dentro de una neurona, como detallamos en la ecuación 2.1. Esta función se suele emplear cuando queremos que el modelo capte patrones lineales en las capas ocultas, y también para cuando las predicciones sean de objetivo numérico en la capa final.

Función sigmoide

La función logística sigmoide se define como:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

La función sigmoide es comúnmente utilizada como una función de activación en la *output layer* dentro de modelos de clasificación binaria. La idea que subyace al uso de esta función consiste en transformar un output numérico de un modelo lineal en un valor entre 0 y 1 que siga una distribución

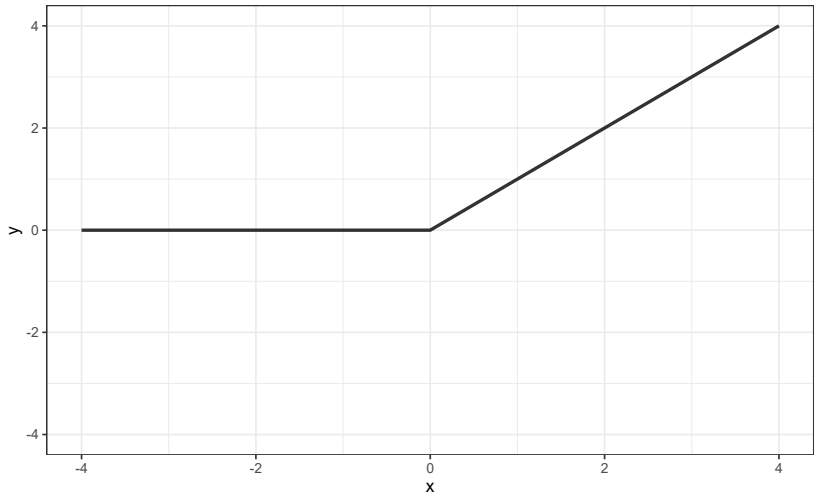


Figura 2.2: *Un ejemplo de Rectified Linear Unit (ReLU)*

de Bernoulli, puesto que buscamos predecir una probabilidad de una variable de tipo categórica para dos categorías distintas..

Rectified Linear Unit (ReLU)

La función ReLU es la función de activación de capas ocultas más conocida en el ámbito de las redes neuronales. Fue presentada por (Hahnloser et al. 2000) para explicar el funcionamiento de circuitos de silicona inspirados en la interacción entre neuronas dentro de una corteza cerebral.

Una función ReLU se define como:

$$g(z) = \text{máx}\{0, z\} \tag{2.4}$$

ReLU iguala a cero todo input proveniente de z que resulte ser negativo. La forma de la función resultante se asemeja a la de un palo de hockey, como se muestra en la Figura 2.2. A diferencia de otras funciones de activación como la sigmoide, es una función sin límites.

Son varios los motivos que han llevado al ReLU a ser una de las funciones de activación más populares en este entorno. (Zeiler et al. 2013) indica varias razones de por qué son preferibles al uso de una función de activación *sigmoide*: Son más fáciles de optimizar, son más rápidas de procesar en el ordenador y atenúan el problema del sobreajuste en mayor medida. Aunque el uso popular de los ReLU se relaciona con su implementación en las capas ocultas de los modelos Deep Learning, (Agarap 2018) propuso la utilización de dicha función en la capa final como método de clasificación multinomial, llegando a

ofrecer resultados similares a la función *softmax*, que es la función más utilizada para esa finalidad y que definiremos en breve.

Una de las desventajas del ReLU es el problema del ReLU moribundo, o *dying ReLU problem*. Mientras se entrena una red neuronal, puede suceder que las neuronas que utilicen un ReLU se desactiven (es decir, que sólo devuelvan un valor igual a cero durante el resto del proceso de estimación), impidiendo así, por un lado, captar algún patrón y, por el otro, no permitiendo al algoritmo de optimización actualizar las ponderaciones del modelo debidamente, afectando así a la capacidad de estimación del modelo. Otro problema, explicado por (Clevert et al. 2015), es el hecho de que ReLU puede devolver en término medio un valor mayor que cero a lo largo de la red neuronal, introduciendo un sesgo positivo en las capas posteriores, el cual puede acumularse a medida que la información sea transformada a través de cada vez más capas, afectando así a la eficiencia en la optimización del modelo. Por ello, se han propuesto variantes del ReLU con diferentes propiedades que buscan subsanar estos problemas. Algunas de estas variantes son el Leaky ReLU, el PReLU y el ELU, cuyas formas gráficas pueden observarse en la Figura 2.3. Introduciremos cada una brevemente a continuación.

Leaky ReLU y Parametric ReLU

La variante *Leaky ReLU* se define como:

$$g(z) = \begin{cases} \alpha \cdot z & z \leq 0 \\ z & \text{otherwise} \end{cases} \tag{2.5}$$

donde α es un hiperparámetro cuyo valor varía entre 0 y 1. Esta función reduce las posibilidades de que una neurona se sature al permitir que las neuronas devuelvan valores negativos, aunque sean atenuados en cierta medida por α para conservar las propiedades del ReLU básico. La posibilidad de aceptar valores negativos también ayuda a reducir el sesgo positivo generado con respeto del ReLU, al no variar los resultados únicamente en el intervalo $[0, +\infty]$.

Cuando el hiperparámetro α es escogido manualmente, la función de activación es el *Leaky ReLU*. Si en cambio el hiperparámetro es optimizado, la función se convierte en un *PReLU* o *Parametric ReLU*. Ambas variantes se diferencian en que el valor de α pasa a pertenecer al intervalo $[-\infty, +\infty]$ para el

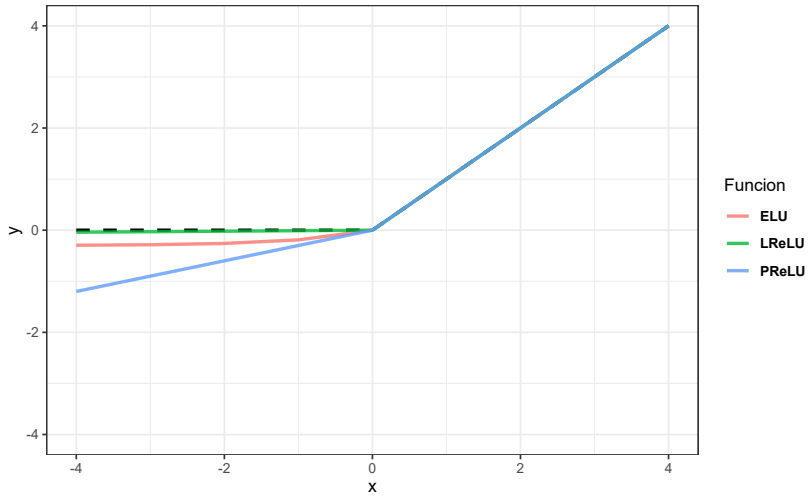


Figura 2.3: *Ejemplos de ReLU y algunas de sus variantes. La línea trazada representa el ReLU base. El parámetro para PReLU y ELU es a = 0.3*

caso del *PReLU*. En general, *PReLU* ofrece mejores resultados que *Leaky ReLU* y genera sobreajuste con menor frecuencia (He et al. 2015). Con todo, *Leaky ReLU* no garantiza que una neurona no se desactive.

Exponential Linear Unit o ELU

La función de activación ELU o *Exponential Linear Unit* comparte la característica del PReLU y Leaky ReLU de introducir valores negativos en su definición. Su forma funcional es:

$$g(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha(e^z - 1) & \text{si } z \leq 0 \end{cases} \tag{2.6}$$

donde de nuevo α es un hiperparámetro a elegir por el usuario. Esta función de activación fue propuesta por (Clevert et al. 2015), y presenta mejores propiedades de generalización y de rapidez de optimización que los ya comentados Leaky ReLU y PReLU, especialmente en redes neuronales de cinco o más capas.

Función softmax

La función softmax se define como sigue:

$$g(z_i) = \frac{\exp(z_i)}{\sum_j^C \exp(z_j)} \tag{2.7}$$

donde $z \in \mathbb{R}^C$, C es el número de categorías posibles a predecir, z_i es el valor inferido por la red neuronal para el elemento i-ésimo del vector z y que corresponde a la categoría i, y z_j son los restantes valores del vector z. Esta función se puede interpretar como la generalización de la función sigmoide para C categorías, devolviendo un vector con las probabilidades asignadas a cada clasificación y cuya suma es en total igual a uno. Es una función normalmente utilizada para la clasificación de imágenes en la capa final de la red neuronal. Una de sus principales ventajas es la interpretación de la predicción numérica en términos de probabilidad (Shen y Liu 2018). Otros ejemplos de este tipo de problemas en los que se utiliza el softmax son para traducción de textos y reconocimiento facial con sensores.

2.2. El algoritmo *backprop*

El aprendizaje basado en el cálculo de gradientes puede volverse muy costoso en términos computacionales para los modelos de redes neuronales, que pueden contener decenas, cientos e incluso hasta millones de parámetros. Esto supone tener que calcular una derivada parcial de la función de coste con respecto tanto de cada relación entre dos neuronas (representadas por las ponderaciones w_{ij}) como por cada sesgo dentro de cada neurona (denominadas b_i), por lo que el número de cálculos puede incrementarse de manera exponencial. Por otro lado, al ser una red neuronal una función compuesta, una parte importante de las derivadas parciales de la función de coste respecto de un parámetro cualquiera depende necesariamente de derivadas que provienen de otros parámetros previos, por lo que se pueden representar las derivadas parciales de las capas intermedias como una composición de otras derivadas ya calculadas. (Rumelhart et al. 1986) aprovechó esta idea en la creación de un nuevo algoritmo llamado *backpropagation* o *backprop*, que a continuación explicamos.

El backprop se compone de dos fases, la *forward phase* y la *backward phase*:

- *Forward phase*: Esta fase es la correspondiente a la del cálculo de la predicción en la red neuronal, con su consecuente evaluación en la función de coste. Al final de esta fase, se calcula de manera inmediata $\frac{\partial \widetilde{J}}{\partial o}$, que es la derivada parcial de la función de coste regularizada ($\widetilde{J}$) con respecto del output de la última neurona de la red, o. Comienza entonces la *backward phase*.

- *Backward phase*: El objetivo en esta fase es conocer el gradiente de la función de pérdida con respecto de cada uno de los parámetros de la red neuronal utilizando la regla de la cadena del cálculo diferencial. Se empieza desde la última capa hasta la primera, revisando todos los parámetros. Por ejemplo, para $1 \leq l \leq L$, siendo L el número de capas totales, las derivadas de las ponderaciones en la capa $l-1$ se calcularán utilizando composiciones de las derivadas parciales en $l, l+1, ..., L$, así evitando tener que calcular más de una vez cada una de estas derivadas. De ahí que a esta fase se la llame fase *backward* (hacia atrás). Esta fase es la más larga y requiere de una explicación exhaustiva, que haremos a continuación.

Consideremos una red neuronal con $h_1, h_2, ..., h_k$ neuronas en las capas ocultas y una neurona o en la *output layer*, cuya predicción es utilizada por la función de pérdida $\widetilde{J}$. Adicionalmente, cada ponderación w_{ij} puede ser interpretada como un parámetro que conecta dos neuronas, h_{r-1} y h_r, por lo que el parámetro que une a ambas sería $w_{(h_{r-1},h_r)}$. Asumiendo que *sólo existiese un único camino* de o a h_r (es decir, que sólo hay un conjunto único de neuronas entre estas dos a través de las que se transmite la información), se puede calcular el gradiente de la función de pérdida con respecto de cualquier parámetro de la forma que sigue:

$$\frac{\partial \widetilde{J}}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial \widetilde{J}}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \forall r \in 1 \ldots k \tag{2.8}$$

En la práctica, es frecuente encontrar redes neuronales que contienen más de un camino posible de o a h_r. En este caso, se utiliza la denominada *regla de la cadena multivariante* para el cálculo del backprop, y se ha de definir una nueva ecuación. Definimos entonces un set de caminos posibles, $\mathcal{P}$, desde o a h_r, por lo que tenemos:

$$\frac{\partial \widetilde{J}}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial \widetilde{J}}{\partial o} \cdot \left[\sum_{[h_r,h_{r+1},\ldots h_k,o] \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\text{El backprop calcula } \Delta(h_r,o)=\frac{\partial \widetilde{J}}{\partial h_r}} \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \forall r \in 1 \ldots k \tag{2.9}$$

Existen dos partes en la ecuación 2.9 que conviene aclarar: El cálculo de $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ y el de $\Delta(h_r, o)$. Para el primer caso, tenemos que:

$$\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} = h_{r-1} \cdot \Phi'(a_{h_r}) \tag{2.10}$$

donde Φ es una función de activación que aplica una transformación en la preactivación de la neurona h_r, que es a_{h_r}. Entonces, $\Phi'(a_{h_r}) = \frac{\partial \Phi(a_{h_r})}{\partial a_{h_r}}$ es la sensibilidad de la función de activación respecto a variaciones de la información que recibe.

El segundo caso, $\Delta\left(h_r, o\right)$, es el término más importante de la ecuación, pues su computación puede crecer exponencialmente dependiendo del número de neuronas en la red. Su cálculo se puede representar como:

$$\Delta\left(h_r, o\right) = \frac{\partial \widetilde{J}}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial \widetilde{J}}{\partial h} \frac{\partial h}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \Delta(h, o) \tag{2.11}$$

Como h se encuentra en una capa posterior a h_r, al calcular $\Delta(h_r, o)$ ya se ha calculado antes $\Delta(h, o)$ debido al sentido "hacia atrás"de la segunda fase del *backpropagation*. Sin embargo, queda por saber cuál es el valor de $\frac{\partial h}{\partial h_r}$ para el cálculo de la ecuación 2.11. Teniendo en cuenta ambas neuronas, h y h_r, que están unidas por la ponderación $w_{(h_r,h)}$, y a_h es el valor preactivación de la neurona h, por la regla de la cadena, podemos derivar $\frac{\partial h}{\partial h_r}$ de la siguiente manera:

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \frac{\partial \Phi\left(a_h\right)}{\partial a_h} \cdot w_{(h_r,h)} = \Phi'\left(a_h\right) \cdot w_{(h_r,h)} \tag{2.12}$$

y sustituyendo (2.12) en (2.11), finalmente tenemos el gradiente $\Delta(h_r, o)$ como sigue:

$$\Delta\left(h_r, o\right) = \sum_{h:h_r \Rightarrow h} \Phi'\left(a_h\right) \cdot w_{(h_r,h)} \cdot \Delta(h, o) \tag{2.13}$$

En resumen, los gradientes son acumulados en la *backward phase* desde la neurona output hacia atrás, y cada neurona es procesada exactamente una vez de esta manera.

A modo de ejemplo, trataremos de calcular $\frac{\partial J}{\partial w_{(x1,h1)}}$, donde $w_{(x1,h1)}$ es la conexión entre $x1$ y $h1$ en la red para el dataset *Iris* de la Figura 2.1. El resultado es el siguiente:

$$\frac{\partial \widetilde{J}}{\partial w_{(x1,h1)}} = \frac{\partial \widetilde{J}}{\partial o} \frac{\partial o}{\partial h1} \frac{\partial h1}{\partial w_{(x1,h1)}} = \Delta(h1, o) \frac{\partial h1}{\partial w_{(x1,h1)}} \tag{2.14}$$

donde $\Delta(h1, o)$ son todos los gradientes acumulados desde o hasta $h1$. Como sólo hay un único camino posible desde o hasta $w_{(x1,h1)}$, el cálculo resulta sencillo. En caso contrario, como ya vimos en (2.9), habría un sumando por cada posible camino de vuelta.

2.3. Interpretación geométrica del funcionamiento del algoritmo de optimización

Las magnitudes relativas de las derivadas parciales con respecto de los parámetros en diferentes partes de la red neuronal tienden a ser muy distintas, lo cual genera problemas para el *gradient descent*, al

utilizar un único *learning rate* para todos los parámetros (Aggarwal 2018). Esto se traduce en mayor inestabilidad a la hora de minimizar la función de coste. Para ilustrar este problema, en la Figura 2.4 mostramos la geometría del funcionamiento del *gradient descent* en la minimización de dos funciones de coste distintas: $J = x^2 + y^2$ y $J = x^2 + 4y^2$. Nótese que ambas funciones presentan un problema convexo y sin mínimos locales, situación que no es frecuente en el ámbito de las redes neuronales, debido a la elevada dimensionalidad de los datos con los que suelen tratar, y además asumimos que no se ha aplicado ninguna técnica de regularización (por lo que no utilizamos $\widetilde{J}$). Decidimos elegir dos problemas convexos por simplicidad, para así mostrar lo importante que es la estructura de la función de coste en un problema de optimización.

En la Figura 2.4(a) vemos que la dirección del *gradient descent* es perpendicular a cualquiera de las posibles funciones en J y su dirección está perfectamente alineada con el mínimo global. Es entonces posible que el problema encuentre una solución en un único *epoch* si el tamaño de la actualización de los parámetros (entendiéndose esta como $\eta \frac{\partial J}{\partial \theta_t}$ en la ecuación 1.6) es lo suficientemente grande, por lo que el proceso sería eficiente. Supongamos ahora la función $J = x^2 + 4y^2$, que es similar a la anterior función con la diferencia de que su derivada respecto de y es cuatro veces mayor. En la Figura 2.4(b) puede verse que forma una elipse en lugar de círculos concéntricos, y que el *gradient descent* no llega a alinearse en la dirección del mínimo global en ninguna de sus actualizaciones. Como consecuencia, el algoritmo llega al mínimo global en ocho actualizaciones en vez de en una, como en el caso de la Figura 2.4(a). Por un lado, el algoritmo se muestra realmente sensible a y, donde variaciones producidas en una actualización son contrarrestadas por actualizaciones siguientes debido a los contínuos cambios de signo. Por el otro, aunque en x la dirección es correcta y los cambios son siempre positivos, los saltos son demasiado breves en comparación con la otra función de coste. En otras palabras, incluso con sólo dos variables (x e y), una leve modificación en una de ellas puede reducir significativamente la eficiencia del algoritmo de optimización, por lo que en un entorno donde trabajemos con cientos o miles de variables, que suele ser frecuente en redes neuronales, este problema puede verse multiplicado.

Existen algoritmos de optimización que están adaptados para situaciones similares al problema de la Figura 2.4(b). A continuación presentamos varias alternativas.

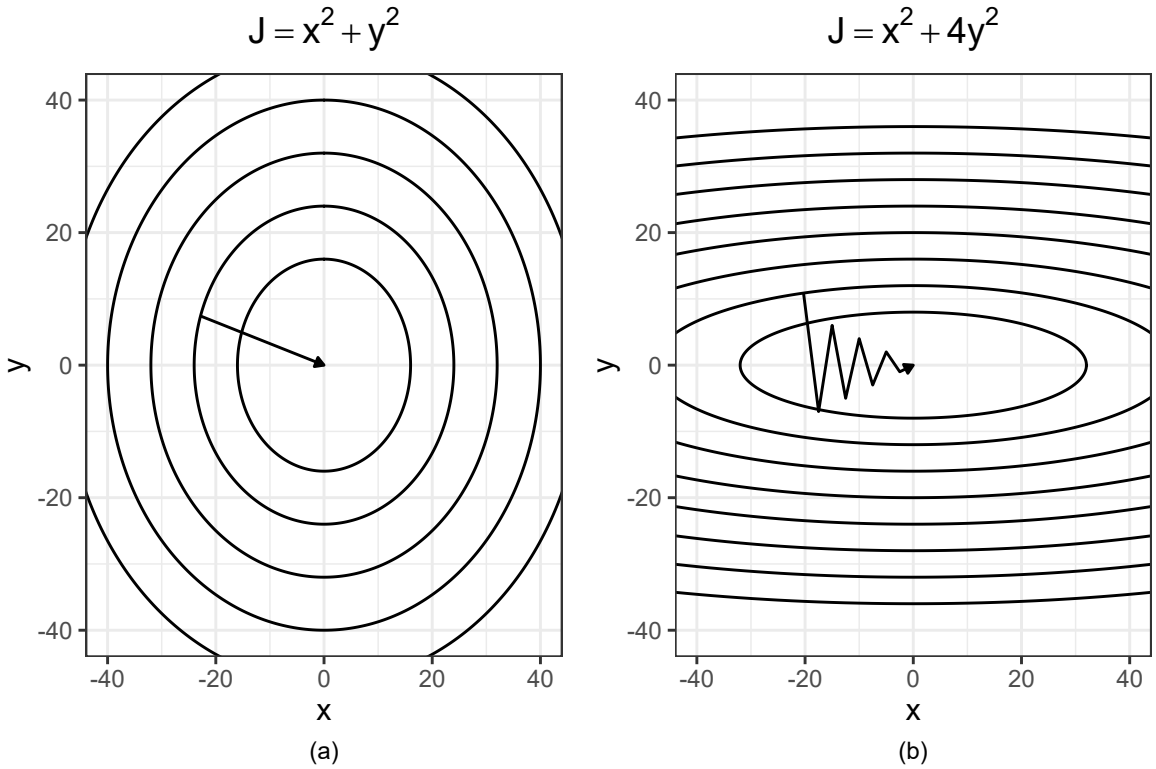


Figura 2.4: *Cómo la forma de la función de pérdida afecta al rendimiento del gradient descent. Cada recta es un epoch de la red neuronal (es decir, una actualización de sus parámetros)*

2.4. Algunos algoritmos de optimización posibles

2.4.1. Stochastic Gradient Descent

El *stochastic gradient descent* (SGD) es una variante del *gradient descent* y se define como sigue:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \widetilde{J}_i}{\partial \theta_t} \tag{2.15}$$

El prefijo *stochastic* proviene del hecho de que este algoritmo evalúa la función en todas las observaciones muestrales de forma aleatoria antes de terminar el *epoch*. Una consecuencia de utilizar la función L_i, en lugar de la suma total L, es el hecho de que el cálculo del gradiente es menos preciso en cuanto a la estimación de la dirección correcta para la minimización de la función de coste, especialmente en las primeras actualizaciones de los parámetros del modelo (debido a que $\theta_{t=1}$ suele escogerse de manera aleatoria y en $\mathbb{R}$). Sin embargo, a largo plazo (i.e. un número de actualizaciones elevado), la pérdida de precisión respecto del *gradient descent* suele disminuír, y el menor requirimiento de memoria en el ordenador por parte del SGD ha contribuído a que este algoritmo sea más popular.

Una versión del SGD que permite equilibrar entre uso de memoria y precisión de estimación del gradiente es el *mini-batch stochastic gradient*:

$$\theta_{t+1} = \theta_t - \eta \sum_{i=1}^{m} \frac{\partial \widetilde{J}_i}{\partial \theta_t} \tag{2.16}$$

donde m es el tamaño del batch aleatorio extraído de una muestra de tamaño N. Las actualizaciones se realizan para un total de $\frac{N}{m}$ submuestras hasta recorrer toda la muestra.

2.4.2. RMSprop

Hemos visto en la Figura 2.4(b) que el algoritmo de optimización *gradient descent* mostraba oscilaciones bruscas en el eje y y progresos tímidos hacia el mínimo global en el eje x. Existen algoritmos de optimización que en su funcionamiento introducen una forma de penalización a este tipo de oscilaciones (esto es, movimientos que cambian de signo reiteradamente en cada actualización). Uno de estos algoritmos es el *RMSprop* (Hinton 2012).

El algoritmo *RMSprop*, para una iteración i-ésima, se define como sigue:

$$w_i \Leftarrow w_i - \frac{\eta}{\sqrt{A_i}} \left(\frac{\partial \widetilde{J}}{\partial w_i} \right); \quad \forall i \tag{2.17}$$

donde w_i es una ponderación cualquiera de una red neuronal, $\widetilde{J}$ es la función de coste regularizada, η es el *learning rate* y A_i es un factor de normalización. En comparación con el *gradient descent* y el SGD, la novedad de *RMSprop* es la introducción de A_i, que está construído de la siguiente forma:

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial \widetilde{J}}{\partial w_i}\right)^2 \quad \forall i \tag{2.18}$$

donde ρ es un hiperparámetro para realizar una media exponencial. En el primer *epoch*, A_i es inicializado con valor igual a 0. El uso de $\left(\frac{\partial J}{\partial w_i}\right)^2$ permite guardar un registro histórico de las derivadas parciales en w_i a lo largo de los *epochs*. Al ser el cuadrado de la derivada parcial, se enfatiza la magnitud de esta derivada en lugar de su signo, por lo que movimientos del gradiente con signo constante entre actualizaciones son favorecidos en oposición a movimientos en zig-zag, a través del factor $\frac{1}{\sqrt{(A_i)}}$ en la ecuación 2.17. Sin embargo, el uso de este factor tiene dos desventajas: Primero, tiende a reducir en general la magnitud de todos los movimientos de la derivada en el largo plazo, debido a la acumulación histórica de los gradientes que afecta a la actualización de las ponderaciones a través del denominador. Segundo, después de un número grande de *epochs*, al depender A_i de derivadas "antiguas" (i.e. de las primeras actualizaciones), este factor puede quedar estancado[1], por lo que afectaría a la precisión del algoritmo. Es por esta razón que A_i contiene una media exponencial a través del parámetro ρ, que resuelve los dos problemas mencionados, quitando peso a los valores de $\frac{\partial \widetilde{J}}{\partial w_i}$ más antíguos.

RMSprop es un algoritmo de optimización bastante popular en el entorno de Deep Learning. Está basado en el algoritmo AdaGrad (Duchi et al. 2011), el cual sólo difiere en que no contiene una media exponencial en A_i. Una desventaja del RMSprop se encuentra en la necesidad de inicializar A_i a cero y la consecuencia que esto tiene en el uso de una media exponencial, que puede generar un sesgo en la estimación de A_i a favor del cuadrado de la derivada parcial en la primera actualización. Otro algoritmo, denominado *Adam* (Kingma y Ba 2017), solventa este problema de la forma que detallaremos a continuación

2.4.3. Adam

El algoritmo *Adam* se define como sigue:

$$w_i \Leftarrow w_i - \frac{\eta_t}{\sqrt{A_i}} F_i; \quad \forall i \tag{2.19}$$

[1]Pongamos por ejemplo que una de las derivadas antiguas tenga un valor muy cercano a cero. Al utilizar una acumulación histórica, esto afectará también a las derivadas más recientes, reduciendo su tamaño en gran medida.

donde A_i ya fue definido en (2.18), η_t es un *learning rate* ajustado por un corrector, y F_i es un nuevo componente que introduce una media exponencial para el momento de primer orden de $\frac{\partial \widetilde{J}}{\partial \theta}$. En el caso de F_i, tenemos que:

$$F_i \Leftarrow \rho_f F_i + (1 - \rho_f) \left(\frac{\partial \widetilde{J}}{\partial w_i} \right) \quad \forall i \tag{2.20}$$

donde ρ_f es un hiperparámetro para hacer una media exponencial, distinto al ρ de la ecuación 2.18. La finalidad de F_i en el algoritmo *Adam* es introducir inercia en la estimación del gradiente, de tal forma que le permita al algoritmo superar mínimos locales en el problema de optimización. Definimos η_t como:

$$\eta_t = \eta \left(\frac{\sqrt{1 - \rho^t}}{1 - \rho_f^t} \right) \tag{2.21}$$

Las medias exponenciales contenidas en A_t y F_t dentro de (2.19) pueden causar inestabilidad debido a la inicialización de ambos factores en 0 al comenzar la estimación del modelo (esto es, en $t = 0$). Por esta razón, la introducción de η_t pretende corregir este desequilibrio. Nótese que tanto ρ^t como ρ_f^t dependen de la iteración t del modelo, y para iteraciones grandes ambos hiperparámetros tienden a 0, por lo que $lim_{t \to \infty} \eta_t = \eta$.

Tanto η, ρ como ρ_f son hiperparámetros escogidos manualmente. Para ρ y ρ_f, (Kingma y Ba 2017) recomiendan escoger los valores 0.999 y 0.9, respectivamente. Para η se suele escoger un valor de 0.03, aunque dependerá de la escala de los datos utilizados. El algoritmo *Adam* ofrece resultados competitivos respecto de los otros dos algoritmos anteriores que presentamos en cuanto a estabilidad en el problema de optimización y generalización de las predicciones (Kingma y Ba 2017).

Capítulo 3

Los modelos Deep Learning

La red neuronal que hemos estudiado antes se engloba dentro del campo del *Shallow Learning* (aprendizaje poco profundo), que se refiere a un conjunto de modelos que utilizan únicamente una o dos capas ocultas. Si entonces hablásemos de redes neuronales que utilizan más de dos capas, pudiendo llegar hasta decenas o cientos de ellas en algunos casos, entonces nos referimos a un modelo *Deep Learning*.

El término *Deep Learning* hace referencia a un subconjunto del área de *Machine Learning* donde se estiman modelos por ordenador con cientos o miles de variables mediante el uso de redes neuronales artificiales con numerosas capas ocultas. El origen de este campo se remonta alrededor de los años 80, si bien su desarrollo sufrió una ralentización en la década posterior debido a lo computacionalmente costosos que eran estos modelos para los ordenadores de aquella época (Anthony y Bartlett 2009). Su resurgir en el año 2006 se debe a los mismos motivos del auge del Machine Learning que antes señalábamos: Es decir, el aumento de la disponibilidad de datos y del poder de computación (I. Goodfellow et al. 2016), con la particularidad de que por esta época se produce un impulso al desarrollo de las tarjetas gráficas o GPU (*Graphic Processing Unit*), que son más efectivas para la tarea de reconocimiento de imágenes que las CPU (*Central Processing Unit*), las cuales forman parte del funcionamiento básico de cualquier ordenador. Un tercer factor del desarrollo del Deep Learning han sido las oleadas de inversiones de carácter cíclico con interés en el avance del campo de la Inteligencia Artificial a lo largo del siglo XX. El Deep Learning se suele utilizar en tareas donde otros modelos tradicionales del Machine Learning no han sido capaces de generar buenos rendimientos, como por ejemplo el reconocimiento de imágenes o de voz, debido a la elevada dimensionalidad en la que esta información suele estar codificada. En el apartado 3.1 hablaremos de varios problemas recurrentes que surgen en el empleo de estos modelos, para luego en el apartado 3.2 introducir varias técnicas de

regularización conocidas en este ámbito.

3.1. Problemas típicos en Deep Learning

3.1.1. Cuándo suministrar nuevos datos al modelo

Aunque el problema del sobreajuste suele solucionarse escogiendo un diseño del modelo acorde a la complejidad de los datos, en algunos casos puede también subsanarse mediante el suministro de mayor información al modelo. Si por ejemplo nos encontramos en una situación donde se produce sobreajuste, el uso de una muestra mayor en la red neuronal puede aumentar el error de entrenamiento (normalmente a través del sesgo) a cambio de una reducción en el error de predicción (Aggarwal 2018). Una vez se decide optar por una muestra más grande, cabe también estudiar la viabilidad económica de dicha decisión según los propios recursos del usuario: Por ejemplo, a una empresa tecnológica acostumbrada a manejar grandes bases de datos puede resultarle relativamente barato introducir más datos, mientras que a una empresa pequeña puede resultarle insuficiente la mejora del modelo en base a lo que le costó la obtención de una muestra mayor. Un indicador para la elección del tamaño de la muestra propuesta por (I. Goodfellow et al. 2016) consiste en realizar una gráfica comparativa entre el error de generalización (tanto en el conjunto de entrenamiento como en el de test) y el tamaño de la muestra suministrada al modelo, y observar si existe algún punto donde la reducción del error no sea lo suficientemente grande como para justificar un aumento del tamaño muestral.

3.1.2. Los problemas del gradiente explosivo y atenuado

La regla de la cadena utilizada en el *backprop* para calcular el gradiente puede introducir distorsiones en la información repartida a las distintas ponderaciones del modelo. Si por ejemplo en la ecuación (2.9) la esperanza de las derivadas multiplicadas fuese menor que 1, tendríamos que una derivada parcial utilizada para actualizar una ponderación w_{ij} podría ser cercana a cero, al provenir esta de sucesivas multiplicaciones por valores menores que 1, por lo que la ponderación podría estar actualizándose en una magnitud menor de la necesaria para la minimización de $\widetilde{J}$. En el caso contrario, si las esperanzas de las derivadas parciales fuesen mayores que 1, podríamos encontrarnos con derivadas demasiado grandes, provocando que el problema de optimización nunca se aproxime lo suficiente al mínimo global. Al primer problema se le denomina el problema del *gradiente atenuado*, mientras que el segundo se conoce como el problema del *gradiente explosivo*. Hay varias fuentes que pueden dar lugar a este problema: Dos de ellas son los algoritmos de optimización, del cual ya explicamos en el apartado 2.4 que el *gradient descent* pueda dar lugar al gradiente atenuado, y las funciones de activación. En el siguiente subapartado explicaremos la importancia de las funciones de activación en este ámbito.

Influencia de la función de activación en ambos problemas

Una fuente de los problemas del gradiente explosivo y atenuado puede ser la propia elección de la función de activación. Existen funciones de activación que tienen derivadas pequeñas, normalmente en el intervalo [0, 1], lo cual puede provocar que en redes con un gran número de capas el *backprop* transmita valores del gradiente muy bajos a las primeras capas[1]. La función *sigmoide*, que solía ser una elección popular para las capas intermedias como función de activación, nunca obtiene una derivada mayor que 0.25 en ningún punto. Por ejemplo, con una red neuronal de 100 capas (incluyendo la capa input y la final), donde todas las neuronas de las capas ocultas tuviesen como activación la sigmoide, la primera capa intermedia recibiría derivadas parciales de magnitud $0{,}25^{98}$ en sus neuronas a través del *backprop*, mientras que en las últimas capas intermedias la información del gradiente no se vería tan reducida. Este es un ejemplo del problema del gradiente atenuado. Por esta razón, la sigmoide ha sido en general sustituída por la ReLU, cuya derivada es igual a 1 en su tramo lineal, por lo que no reduce de manera exponencial el gradiente en sucesivas capas cuando los valores preactivación son positivos. Sin embargo, como dijimos en el apartado 2.1.1, el ReLU puede acumular sesgos positivos, dando lugar al problema del gradiente explosivo, y por lo tanto suele recomendarse utilizar alguna de sus variantes.

3.1.3. La elección de hiperparámetros

Una red neuronal se compone de múltiples hiperparámetros, tales como el número de neuronas, el *learning rate*, el *batch size*, las funciones de activación, los distintos tipos de capas, etc... La elección de estos hiperparámetros depende tanto del problema de optimización a resolver como de la disponibilidad de recursos computacionales (memoria y tiempo de computación, específicamente). Como no se ha desarrollado actualmente una teoría formal que nos permita acotar de manera eficiente una selección adecuada para cualquier tarea a la que nos enfrentemos, la elección de hiperparámetros suele realizarse en la práctica o bien mediante intuición (esto es, juzgar a partir de la propia experiencia los cambios marginales obtenidos en la función de pérdida en base a la modificación manual de algún hiperparámetro) o mediante métodos automáticos, donde se seleccionan rangos de posibles valores para cada hiperparámetro y se comprueba el rendimiento del modelo con todas las combinaciones posibles.

En el caso manual se requieren conocimientos acerca de cada hiperparámetro disponible, algunos de los cuales ya hemos introducido en otros apartados, y utilizando el set de validación se contrastan varias selecciones de estos hiperparámetros. Una recomendación de (I. Goodfellow et al. 2016) es hacer

[1] Recordemos en la ecuación 2.13 que la derivada de la función de activación de una neurona tiene un peso importante en el gradiente acumulado $\Delta(h_r, o)$

múltiples comparaciones utilizando un número fijo de ciclos, donde dicho número vendrá determinado según las limitaciones computacionales del hardware manejado por el usuario de la red neuronal y si el modelo alcanza o no el sobreajuste.

Para la elección automática de hiperparámetros, el método más conocido es la *búsqueda en rejilla* o *grid search*, que consiste en determinar un rango de valores para cada hiperparámetro y explorar todas las combinaciones posibles existentes entre ellos utilizando el conjunto de validación. Una red neuronal dispone de un número considerable de hiperparámetros, y es normalmente imposible explorar todas las combinaciones: por ejemplo, si tuviésemos una red neuronal con unos 6 hiperparámetros diferentes a elegir y que cada uno de ellos podría aceptar unos 10 valores distintos, para explorar todas las combinaciones posibles necesitaríamos entrenar el modelo unas 10^6 veces sobre el conjunto de validación. Nótese que aunque hemos escogido un número relativamente bajo de hiperparámetros para lo que suele ser recurrente en este tipo de modelos, la cantidad de ajustes que habría que realizar para explorar todas las opciones se vuelve fácilmente inabarcable. Debido a este problema, se han propuesto varias formas de utilizar la búsqueda en rejilla de una manera más eficiente. Una de ellas sería realizar un muestreo aleatorio con reemplazamiento sobre un conjunto de rangos que escojamos (normalmente equiespaciados), y después de varias iteraciones, reducir el rango escogido inicialmente alrededor de los hiperparámetros que mejores resultados nos hayan ofrecido (esto es, estrechando no sólo el mínimo y el máximo del rango de valores, sino también reduciendo la distancia entre cada dos valores), así hasta cumplir con un umbral de mejora mínimo en la función de coste (Aggarwal 2018).

Otra forma de elegir los hiperparámetros de forma automática es la *búsqueda aleatoria* o *random search*, que consiste en definir varias distribuciones estadísticas para la búsqueda aleatoria de valores de los hiperparámetros, en función del rango numérico en el que se mueva cada hiperparámetro: por ejemplo, una distribución *multinoulli* para más de dos valores discretos (número de neuronas, capas, etc...) o utilizando como potencias de diez una variable aleatoria con distribución uniforme de números negativos para hiperparámetros que comprenden valores muy reducidos (como por ejemplo, el ratio de aprendizaje). Esto es, escoger un valor según $f(x) = 10^x$ para $x \sim U(a, b)$, donde a y b comprenden un rango de valores reales negativos. Este método fue propuesto por (Bergstra y Bengio 2012) y suele reducir el error de validación más rápidamente que el método de búsqueda en rejilla, pues este último suele repetir un mismo valor de un hiperparámetro más de una vez para realizar la búsqueda (por ejemplo, asumiendo los hiperparámetros $x \in [1, 2]$ e $y \in [4, 7]$, si queremos explorar todas las combinaciones posibles entre ambos, cada uno de estos números se repetirán dos veces en el total de combinaciones), mientras que utilizando una distribución uniforme contínua es prácticamente imposible que se produzcan estas repeticiones.

3.2. Regularización en los modelos Deep Learning

Debido a la elevada personalización de la que dispone un modelo *Deep Learning* y de los numerosos parámetros que puede contener, es bastante sencillo construir un modelo que pueda sobreajustar, siempre y cuando se deje entrenando al modelo el tiempo suficiente (es decir, en número de *epochs*). En esta sección haremos un repaso de las técnicas más populares de regularización en este entorno.

3.2.1. Dropout

El método *dropout* es una técnica de regularización que fue propuesta por (Srivastava et al. 2014) como una forma de evitar el sobreajuste en base a la desactivación aleatoria de neuronas ($g(z) = 0$) dentro de un modelo Deep Learning, reduciendo su número de parámetros estimados y, por tanto, su complejidad. Las desactivaciones se representan con una variable aleatoria asignada a cada función de activación, con una distribución de Bernoulli y una probabilidad p de que la función en cuestión se desactive.

Asumiendo que un modelo Deep Learning dispone de N neuronas *input* y *hidden*, el procedimiento del *dropout* es como sigue:

1. Se remuestrea un modelo respecto del modelo base. Cada neurona de la *input layer* o de las *hidden layers* se le desactiva su función de activación con una probabilidad p_m, para $1 \leq n \leq N$, de manera independiente las unas de las otras. Al desactivar la función de activación, las conexiones w_{ij} en la neurona pertinente son también desactivadas.

2. Se remuestrea una observación o un *batch* del set de entrenamiento.

3. Se actualizan las ponderaciones con el *backprop* y el algoritmo de optimización elegido en base a la submuestra escogida en el punto 2, completando así una iteración.

4. Vuelta a empezar en el punto 1.

Un detalle característico de este procedimiento es que, escogiendo un *batch size* inferior al tamaño de la muestra N, las ponderaciones se actualizan varias veces antes de completar un *epoch*, y utilizando un modelo distinto en cada iteración. Cada modelo remuestreado utiliza las ponderaciones actualizadas por el anterior modelo, y así sucesivamente hasta completar el último *epoch*. Al señalar que cada función de activación se desactiva de manera independiente a las demás, podemos decir que en este procedimiento pueden generarse hasta 2^N modelos Deep Learning diferentes posibles, por lo que resulta difícil llegar a explorar todos los modelos para una red neuronal con 100 neuronas, por ejemplo, a pesar de que

es un modelo con pocas neuronas en la práctica dentro del Deep Learning. Esta técnica convierte al resultado de la red neuronal en una *esperanza* del output de múltiples redes neuronales distintas.

3.2.2. Early stopping

La técnica *early stopping* consiste en detener el ajuste del modelo en un punto donde se minimice el error de Validación Cruzada. Se suele empezar con un gran número de *epochs*, por ejemplo 100, y se inicia un nuevo ajuste dependiendo del epoch que se detecte que ofrezca el menor error de validación de todo el proceso. A diferencia de otros métodos de regularización, como los de penalización de la norma que hemos definido anteriormente, el *early stopping* es relativamente poco intrusivo (no requiere modificación de la función objetivo) y sencillo (se puede hacer manualmente en base a observar el progreso de la minimización de la función de coste de manera gráfica, aunque hay la posibilidad de hacerlo según un algoritmo). Por estas razones, es probablemente la técnica de regularización más utilizada en el campo del *Deep Learning*.

Hay dos formas de utilizar el método early stopping según (I. Goodfellow et al. 2016):

1. Una vez detectado un *epoch* óptimo utilizando el set de validación de los datos, se reinicia el modelo desde el principio, pero esta vez utilizando todos los datos del set de entrenamiento (sin set de validación) y deteniendo el ajuste en el epoch óptimo que hemos identificado. Naturalmente, debido a que esta vez se utilizan todos los datos en vez de sólo una porción, los pasos dentro de cada ciclo serán mayores y el modelo tardará más en ajustarse.

2. Entrenando el modelo con el uso de un set de validación, detenemos el ajuste en el epoch óptimo que hemos identificado. Acto seguido, seguimos ajustando el modelo (es decir, no reiniciamos el modelo), pero esta vez sobre todos los datos del set de entrenamiento, hasta que se alcance un nuevo mínimo de error que supere al anterior que hemos identificado con el set de validación. Este procedimiento tiene el inconveniente de que no hay garantía de que vayamos a encontrar un mínimo que supere a aquel alcanzado inicialmente con el set de validación.

Se puede comprobar la utilidad del early stopping desde el punto de vista del intercambio sesgo-varianza. Para alcanzar la verdadera función de pérdida de un problema de optimización, necesitaríamos suministrar al modelo datos infinitos provenientes de la población de interés. Como sólo disponemos de una muestra de datos (x, y), la función de pérdida creada a partir de estos no será la verdadera función objetivo del problema de optimización, por lo que habrá una cantidad de sesgo y/o varianza entre dicha función y la función objetivo que proviene de la muestra. No obstante, utilizar el early stopping durante el aprendizaje nos puede ayudar a acercanos a la verdadera función, deteniéndonos

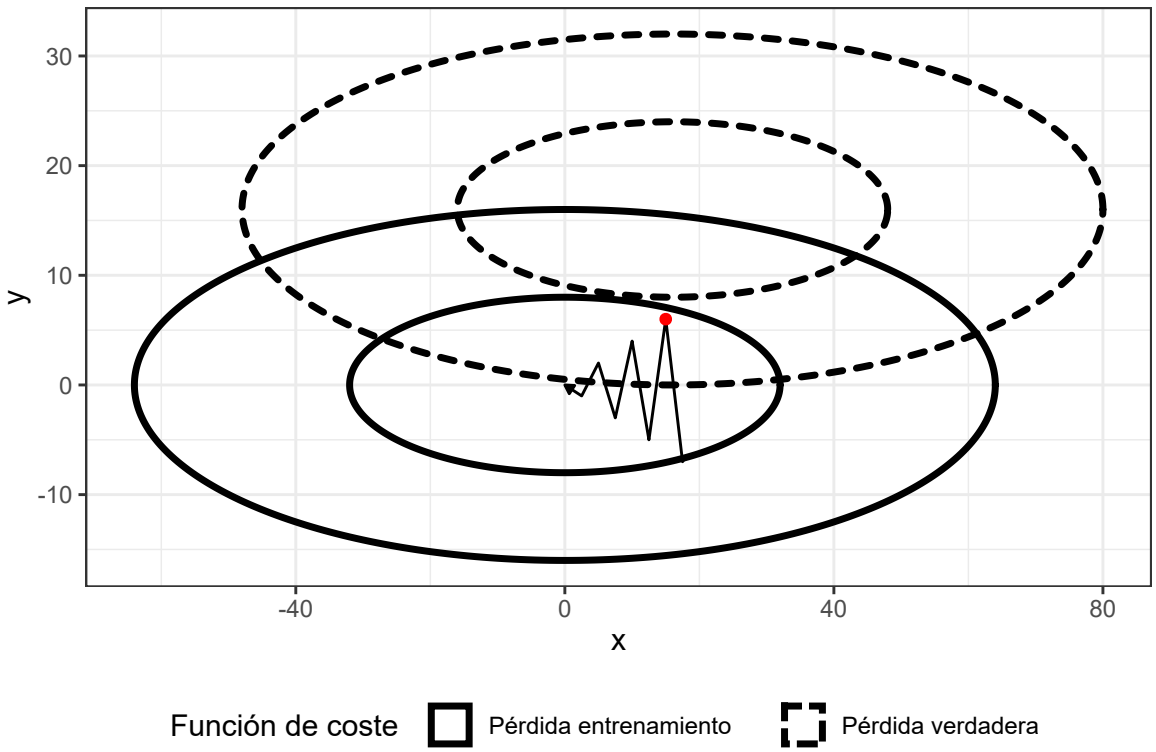


Figura 3.1: *Entrenamiento del algoritmo de optimización en una función de pérdida estimada comparado con la función de pérdida verdadera. En el primer epoch (punto rojo) es donde más se acerca a la verdadera función, y por tanto donde el error de validación es mínimo.*

en un *epoch* considerado óptimo. En la Figura 3.1 puede observarse que al terminar el primer *epoch*, el algoritmo de optimización está lo más cerca posible de la verdadera función. A partir de dicho paso, el algoritmo se aleja de la función verdadera, aumentando de manera previsible el error de predicción sobre los datos de validación, puesto que sólo se ocupa de minimizar la función de pérdida estimada a partir de nuestros datos. Si bien este es un ejemplo intuitivo, en la práctica es motivo de sospecha que una red empeore su ajuste a partir del primer epoch, pues no se estarían aprovechando los beneficios del aprendizaje de redes neuronales, por lo que habría que revisar nuevamente el diseño del modelo.

3.2.3. Capas convolucionales

Una capa convolucional en Deep Learning es un tipo de capa intermedia que utiliza convoluciones para la estimación, utilizando una primera función como *input* (sea información de una muestra o de unas neuronas) y una segunda función como *kernel* (llamada *filtro*) para crear una tercera función que es el resultado de desplazar el *kernel* a lo largo del *input*.

Las convoluciones tienen varias aplicaciones en la vida real, como por ejemplo para filtrado de señales radiofónicas o para procesar nuevos sonidos, como las reverbaciones. En la clasificación de imágenes con redes neuronales, si por ejemplo deseamos identificar coches en distintas fotos, el uso de convoluciones en las capas permite al modelo guardar información relacionada con diferentes propiedades de los coches en sus diferentes partes (ruedas, retrovisores, ventanas, etc...), tales como contornos o texturas. A su vez, estos elementos extraídos pueden ser también objeto de nuevas convoluciones, de forma que se obtiene información cada vez más abstracta acerca de lo que es para la red neuronal el concepto *coche*.

Para ejemplificar cómo funciona una capa convolucional en la práctica, definamos una matriz de datos $X^{(q)}$ tal que $X \in \mathbb{N}^{L \times B \times C}$ y cuyos valores se encuentran en el rango $[0, 255]$, siendo q la capa donde se encuentra la matriz, L la altura, B el ancho y C es una tercera dimensión denominada *profundidad*. Para mayor sencillez, asumiremos que X representa a una imagen en escala de grises, de tal forma que $C = 1$. Definimos también un filtro $H^{(q)}$ cuadrado de altura y ancho F y profundidad $C = 1$ (el filtro ha de tener la misma profundidad que X, como veremos más adelante), esto es, $H \in \mathbb{N}^{F \times F \times 1}$. El objetivo del filtro es realizar una combinación lineal con sus propios parámetros y los valores de submatrices dentro de la matriz X, recorriendo todas las submatrices posibles y generando así una nueva matriz de valores $X^{(q+1)}$, cuyas dimensiones dependerán del tamaño de $X^{(q)}$ y de $H^{(p,q)}$, donde p es el índice que identifica al p-ésimo filtro utilizado. Entonces, las dimensiones de $X^{(q+1)}$ se definen como:

$$
\begin{aligned}
L_{q+1} &= L_q - F_q + 1 \\
B_{q+1} &= B_q - F_q + 1 \\
C_{q+1} &= 1
\end{aligned}
\tag{3.1}
$$

donde L_q y B_q son la altura y el ancho, respectivamente, de la matriz $X^{(q)}$ en la capa q, y F_q es tanto el alto como el ancho del filtro $W^{(q)}$. Si por ejemplo tuviésemos una matriz $X^{(q)}$ de tamaño 3×4, y un filtro $W^{(q)}$ de dimensiones 2×2, como se observa en la Figura 3.2, la altura y el ancho de

$X^{(q+1)}$ serían:

$$
\begin{aligned}
L_{q+1} &= 3 - 2 + 1 = 2 \\
B_{q+1} &= 4 - 2 + 1 = 3 \\
C_{q+1} &= 1
\end{aligned}
\tag{3.2}
$$

Es posible configurar en la capa convolucional la posibilidad de que el filtro ignore varias submatrices posibles mediante un hiperparámetro denominado *paso* o *stride*. Al ajustar un paso tal que $S_q > 1$, el filtro no recorrerá todas las submatrices posibles, y la matriz X_{q+1} tendrá las siguientes dimensiones:

$$
\begin{aligned}
L_{q+1} &= (L_q - F_q)/S_q + 1 \\
B_{q+1} &= (B_q - F_q)/S_q + 1
\end{aligned}
\tag{3.3}
$$

por lo que las dimensiones son menores que en el caso de $S_q = 1$. La finalidad de este hiperparámetro es la de introducir regularización en el modelo mediante la reducción de información relacionada con la localización en la imagen del objeto a identificar, de tal forma que un mismo objeto encontrado en diferentes coordenadas en dos fotos distintas pueda ser correctamente identificado por la red neuronal. Un inconveniente de este hiperparámetro es que sólo puede ser aplicado para matrices y filtros cuya diferencia de dimensiones ($L_q - F_q$ y $B_q - F_q$) sea múltiplo de S_q, porque en caso contrario tendríamos dimensiones tal que $L_{q+1}, B_{q+1} \notin \mathbb{N}$, y por lo tanto no podría realizarse la convolución.

Teniendo en cuenta que $x_{ijk}^{(q)}$ es el elemento ijk-ésimo de $X^{(q)}$, donde i identifica la altura, j el ancho y k la profundidad, y $w_{ijk}^{(p,q)}$ es el elemento ijk-ésimo de $W^{(q)}$, para hallar el valor de una convolución $x_{ijp}^{(q+1)}$ perteneciente a $X^{(q+1)}$, tenemos que:

$$
\begin{aligned}
x_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} x_{i+r-1,j+s-1,k}^{(q)} & \forall i \in \{1 \ldots, L_q - F_q + 1\} \\
& \forall j \in \{1 \ldots B_q - F_q + 1\} \\
& \forall p \in \{1 \ldots d_{q+1}\}
\end{aligned}
\tag{3.4}
$$

Es importante señalar que la matriz $X^{(q+1)}$ tendrá tanta profundidad como número de filtros se utilicen en la capa q. En la Figura 3.4 es posible ver el resultado de una convolución creada con un

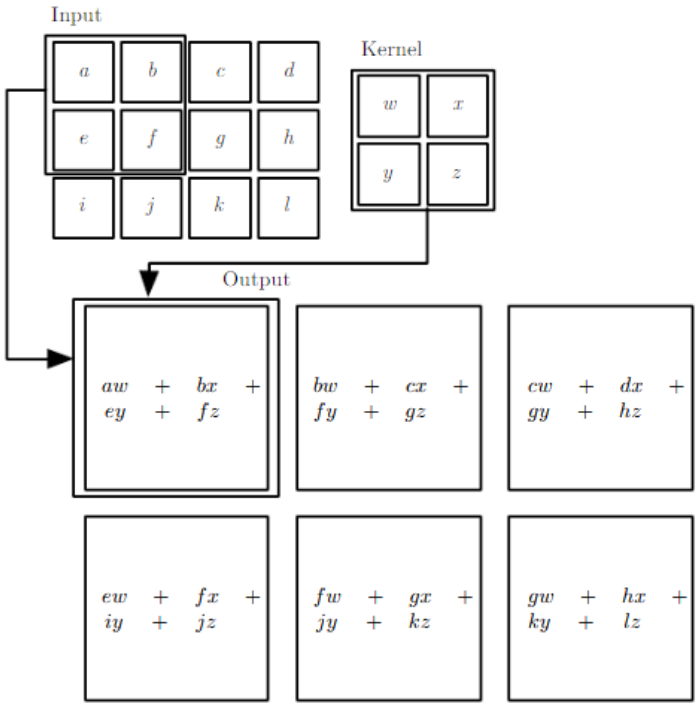


Figura 3.2: *Cómo se realizan las operaciones dentro de una capa convolucional de un modelo Deep Learning. Gráfico extraído de (Goodfellow et al., 2016)*

filtro de profundidad 3 para la imagen de una rana, mientras que en la Figura 3.5 puede verse el output realizado por 32 filtros distintos para la misma imagen.

Hay tres ideas que subyacen la importancia de las capas convolucionales y que las distinguen de las capas densas:

- **Interacciones dispersas**: Una capa densa de una red neuronal utiliza parámetros únicos para cada elemento de las matrices input, por lo que una neurona contiene tantos parámetros como elementos provengan de la matriz input (es decir, es de dimensiones $n \times p$). En cambio, en una capa convolucional, el filtro [2] W de parámetros es una matriz de dimensión kxp, siendo k necesariamente menor que n, por lo que un filtro no recibe información de todos los elementos de la matriz input, reduciendo así el número de operaciones y por ende el número de parámetros a estimar, produciendo un efecto de regularización en el aprendizaje. Un ejemplo de esta propiedad puede observarse en la Figura 3.3.
- **Parámetros compartidos**: Un mismo filtro W se utiliza para más de una submatriz distinta de la matriz de input X. Recordemos la ecuación 2.1, donde a cada elemento de la matriz X le corresponde una ponderación de la matriz W. En una red convolucional no es así, pues se utiliza una submatriz W' (necesariamente de menor dimensión que la matriz x y cuadrada) sobre cada elemento de la matriz X.
- **Traslaciones equivariantes**: Que una función sea equivariante implica que si el input de la misma es modificado, el output de la función se modificará en la misma medida. Por ejemplo, en la clasificación de imágenes de coches deseamos que el modelo capte el trazado de una rueda, sin importar las coordenadas en donde se encuentre dentro de dicha imagen. Otro ejemplo relacionado es cuando disponemos de dos fotos de un mismo camión, pero en una de las fotos está unos píxeles más desplazado hacia otra dirección: Una capa intermedia densa lo tendría difícil para identificar al mismo camión en ambas fotos, al no filtrar afuera la información de la posición en la imagen del objeto, mientras que es probable que una capa convolucional identifique correctamente el vehículo. Esta propiedad está acentuada por el hiperparámetro *paso* cuando este es mayor que 1, y también por la técnica *max pooling*, de la cual hablaremos en el apartado 3.2.5

3.2.4. Aumento de datos o *data augmentation*

Cuando entrenamos un modelo *Deep Learning* para la clasificación de imágenes, los objetos a identificar pueden aparecer de numerosas formas: Por ejemplo, no es lo mismo obtener una foto directamente

[2] En las capas convolucionales se suele utilizar indistintamente filtro como neurona, aunque es preferible el primer término para distinguirlas de las neuronas de una capa densa.

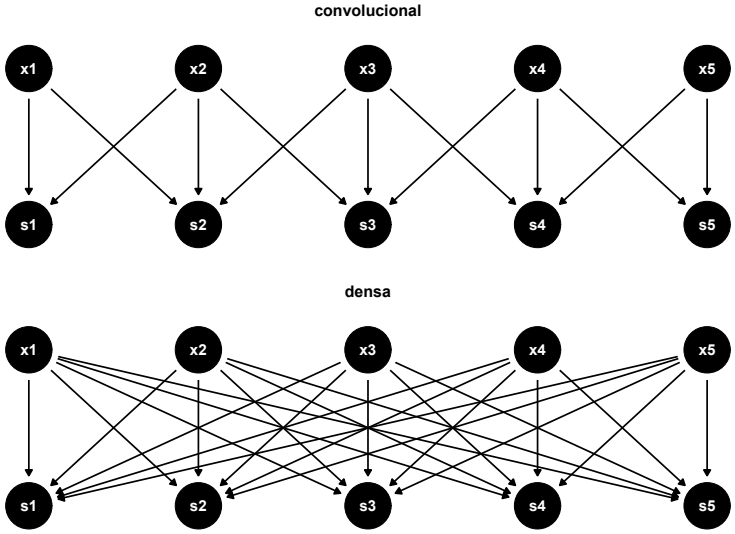


Figura 3.3: *Ejemplo de la propiedad de interacciones dispersas de la capa convolucional frente a la densa. Toda neurona de una capa densa recibe todos los inputs de una capa anterior, mientras que cada neurona o filtro de la capa convolucional sólo recibe un subconjunto.*

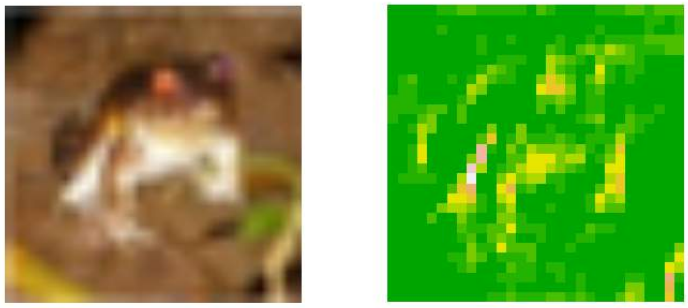


Figura 3.4: *Ejemplo de activaciones neuronales dentro de una capa convolucional. A la izquierda se encuentra una imagen 32x32 de una rana del dataset CIFAR-10, y a la derecha vemos el output procedente de una de las neuronas de la primera capa convolucional de un modelo utilizado en el caso práctico. Las partes coloreadas en amarillo son características detectadas que el modelo identifica como propias del concepto rana", mientras que las zonas verdes son partes ignoradas por el filtro*

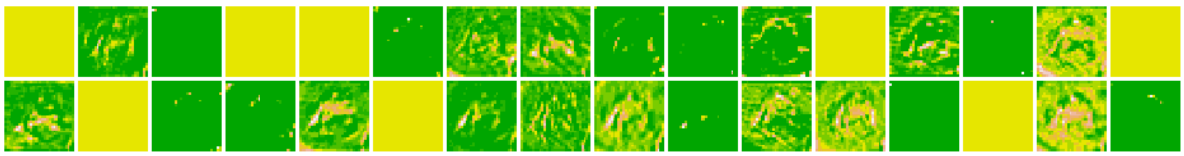


Figura 3.5: *Representación visual de todas las activaciones de la primera capa convolucional de un modelo Deep Learning, utilizando una función de activación ReLU en la capa intermedia. Cada cuadrado es un output resultante de cada una de las 32 neuronas, devolviendo una matriz de tamaño 30x30 de números enteros normalizados en el rango [0, 1]. Los cuadrados en amarillo son filtros con patrones que no se han detectado en la imagen input*

frontal de un perro, que si la misma foto se obtiene con una inclinación de 30 grados, puesto que una red neuronal entenderá ambas fotos como información distinta. Resulta intuitivo pensar que si disponemos de un mayor número de fotos de un objeto desde diferentes enfoques, el modelo obtendrá mayor rendimiento a la hora de intentar reconocer este tipo de objetos en nuevas capturas. Una técnica de regularización que se basa en esta idea es la técnica de *aumento de datos* o *data augmentation*, que consiste en realizar copias modificadas de una imagen existente con la finalidad de introducir nueva información al modelo y por ello mejorar su generalización a nuevos datos. Entre las modificaciones posibles que existen en esta técnica se encuentran las siguientes:

- Desplazamientos horizontales y/o verticales.
- Rotaciones a la izquierda o a la derecha.
- Ampliaciones o reducciones.
- Volteo horizontal o vertical.
- Alargar o reducir la imagen en diferentes direcciones.
- Relleno de nuevos píxeles que aparecen al modificar la imagen (por ejemplo, después de un desplazamiento)

Un ejemplo de este tipo de modificaciones se puede encontrar en la Figura 3.6, donde se ha volteado la imagen horizontalmente en las copias, además de otras modificaciones en los ángulos y en el desplazamiento de la imagen. La elección del tipo de transformaciones ha de responder a la distribución de los objetos en las imágenes sobre las que se va a entrenar el modelo: Por ejemplo, no tendría sentido aplicar volteos verticales a la foto del camión si creemos que es raro encontrarse con un camión volcado, por lo que al modelo no le sería útil esta información.

Figura 3.6: *Ejemplo de una imagen* 32×32 *de un camión del dataset CIFAR-10. La imagen de arriba a la izquierda es la foto original, mientras que las tres restantes son versiones modificadas por el aumento de datos.*

En la práctica, el aumento de datos tiene la finalidad de que una misma imagen no pase por más de un ciclo en la estimación del modelo, por lo que en cada nuevo ciclo o *epoch* el modelo se alimentará de una variante nueva de la imagen original. Uno de los posibles inconvenientes de este método es que, aunque las variantes de una imagen sean distintas a la original, siguen teniendo una gran correlación entre ellas, por lo que es posible que no sea suficiente con utilizar esta técnica para solucionar un posible problema de sobreajuste. Debido a esto, normalmente se combina esta técnica con el uso del *Dropout*. El aumento de datos es especialmente aprovechado por las capas convolucionales, puesto que pueden aprender nuevas variantes de un mismo contorno en una imagen.

3.2.5. Max y Average pooling

El *pooling* es una técnica de regularización que acentúa la propiedad de las traslaciones equivariantes en las capas convolucionales, utilizando kernels que recogen estadísticos de un subconjunto de outputs provenientes de varias neuronas. Existen los casos del *average pooling*, que devuelve una media aritmética del output de varias neuronas para alimentar a las neuronas de la capa siguiente, y del *max pooling*, que recogen el valor con mayor magnitud dentro de un subconjunto de neuronas. Aunque algunos autores no recomiendan usar el average pooling frente al max pooling (François Chollet y Joseph J. Allaire 2018), la primera técnica fue utilizada en un conocido modelo Deep Learning, denominado *ResNet*, del cual hablaremos brevemente en el apartado 4.2. Un ejemplo gráfico del funcionamiento de ambas técnicas se encuentra en la Figura 3.7.

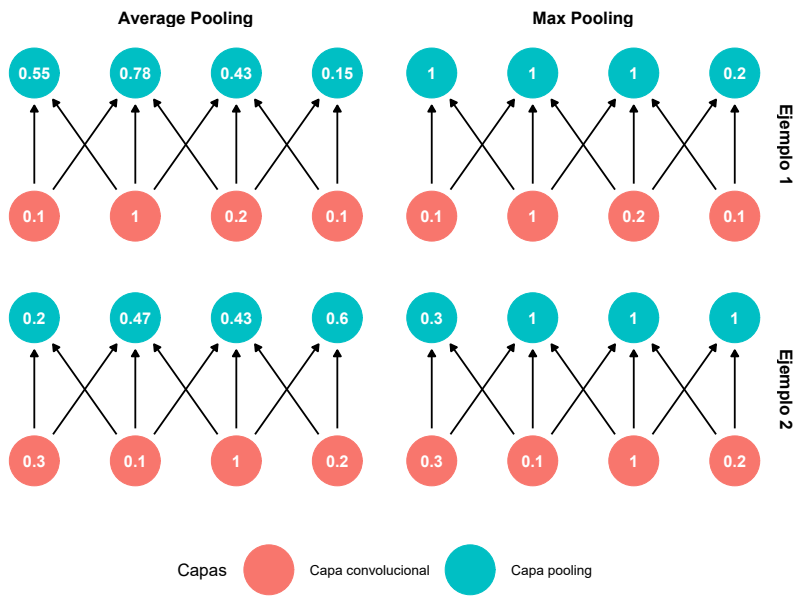


Figura 3.7: *Ejemplo de las técnicas Max y Average Pooling, donde utilizando dos conjuntos de valores distintos (uno por cada ejemplo), pueden observarse diferentes resultados: Average Pooling realiza una media aritmética con el input recibido, mientras que Max Pooling recoge el mayor valor de todo el input.*

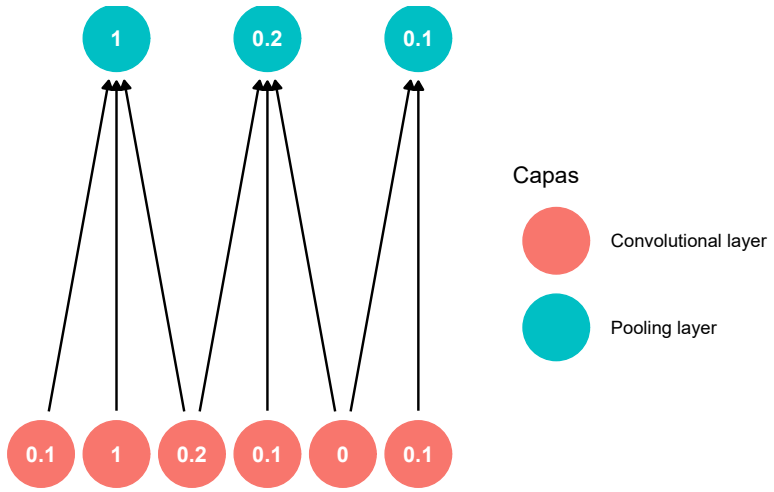


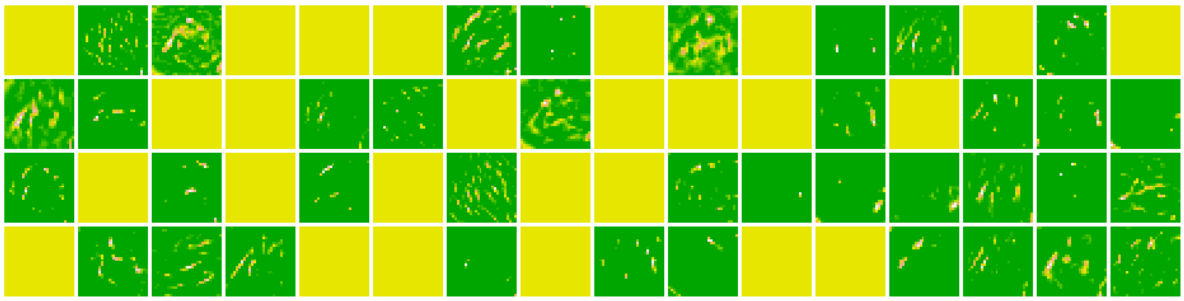
Figura 3.8: *Ejemplo de un max pooling utilizando un stride de tamaño 2. Se puede observar cómo el pooling no tiene el mismo número de neuronas que la capa convolucional, y esto es debido al stride introducido.*

Al igual que en la capa convolucional, es posible configurar en el *pooling* el hiperparámetro *paso* o *stride*. Un ejemplo gráfico puede observarse en la Figura 3.8.

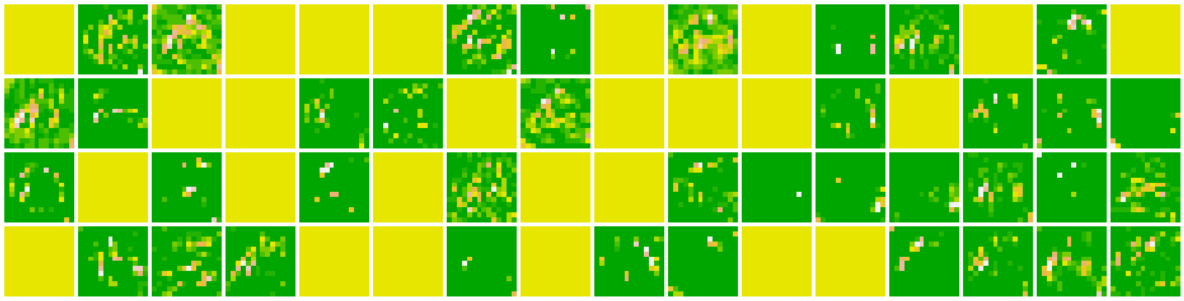
En la Figura 3.9 se muestra cómo afecta en la práctica el uso del pooling sobre el output de una capa convolucional, utilizando el mismo modelo y la misma foto que habíamos utilizado en el apartado 3.2.3. En general se aprecia cómo varios puntos amarillos parecen volverse más grandes o acentuados, lo cual sucede debido a que el max pooling reduce el tamaño de las matrices a la mitad (de $28x28$ a $14x14$), y dichas reducciones se producen utilizando un kernel de tamaño $2x2$ que recoge los elementos con mayor valor dentro de la ventana, y lo devuelve en estas nuevas matrices resultantes que vemos en la subfigura 3.9(b).

3.2.6. Penalización de la norma L1 y L2

Comentábamos en el apartado 1.2.3 que es posible añadir una penalización a la función de coste J de un problema de optimización como técnica de regularización, de tal forma que obtenemos una nueva función de coste $\widetilde{J}$, tal como se ha definido en la ecuación 1.5 . En este apartado definimos en mayor medida las penalizaciones de la norma L_1 y L_2.



(a) Capa convolucional



(b) Capa max pooling

Figura 3.9: *Comparación del output de una capa convolucional y el output resultante al aplicarle una capa max pooling con filtro de tamaño 2x2. Se obtienen imágenes con menor resolución, donde las activaciones se acentúan.*

La forma en la que la penalización de la norma $L1$ afecta a la actualización de los parámetros en una iteración cualquiera se define como:

$$w_i \Leftarrow w_i - \alpha\lambda s_i - \alpha\frac{\partial\widetilde{J}}{\partial w_i} \tag{3.5}$$

donde w_i es el parámetro i-ésimo, λ es el hiperparámetro de penalización, α es el ratio de aprendizaje y s_i es un valor que depende del signo de w_i. Esto es:

$$s_i = \begin{cases} -1 & w_i < 0 \\ \\ +1 & w_i > 0 \end{cases} \tag{3.6}$$

En cambio, en el caso de la norma L_2, tenemos:

$$w_i \Leftarrow w_i(1 - \alpha\lambda) - \alpha\frac{\partial\widetilde{J}}{\partial w_i} \tag{3.7}$$

La diferencia entre las penalizaciones de las normas L_1 y L_2 radica en que la primera modifica la actualización del parámetro de manera aditiva, mientras que la segunda lo hace de manera multiplicativa. Como consecuencia, L_2 puede reducir la magnitud de w_i hasta valores muy próximos a cero, mientras que L_1 puede igualar w_i a cero, actuando así como una técnica de selección de variables o incluso como la técnica Dropout que antes definimos en 3.2.1.

En el entorno de Machine Learning la penalización L_2 suele ofrecer mejores resultados que la L_1. Sin embargo, en un área donde se manejan millones de parámetros, como es el caso del Deep Learning, la diferencia de rendimiento entre ambas penalizaciones no suele ser elevada (Aggarwal 2018).

Capítulo 4

Caso práctico

En este capítulo vamos a realizar una demostración práctica del uso de una red neuronal para la identificación de objetos en imágenes, con un énfasis en el uso de técnicas de regularización. Para ello, haremos uso del dataset CIFAR10, recopilado por (Krizhevsky 2009), que contiene un total de 60.000 imágenes de 10 clases de objetos distintos, desde vehículos hasta animales. Se propondrá un modelo base y, a partir del mismo, se generarán nuevos modelos que serán versiones de este mismo modelo pero con distintas técnicas de regularización aplicadas, con la finalidad de mostrar al lector la utilidad de las mismas en la tarea pertinente. La construcción y estimación de los modelos se realizará con el software R (Team 2020) mediante una herramienta dedicada al Deep Learning denominada Keras (J. Allaire y FranÃ§ois Chollet 2019).

Del total de imágenes, escogemos aleatoriamente 40.000 imágenes para el conjunto de entrenamiento, 10.000 para el conjunto de validación y otras 10.000 para el conjunto de test. La razón de las proporciones escogidas para cada conjunto tiene relación con la elección del tamaño de batches, que detallaremos en el siguiente subapartado.

4.1. Criterio de evaluación e hiperparámetros elegidos

Para medir el rendimiento de cada uno de los modelos, utilizamos como función de pérdida la Deviance Multinomial, que ya definimos en el apartado 1.1.2. Como métrica tenemos la medida de precisión o *accuracy*, que se define como sigue:

$$accuracy(y, \hat{y}) = \frac{\sum_i^n I(y_i = \hat{y}_i)}{n} \tag{4.1}$$

donde y_i es la clasificación de la i-ésima observación, $\hat{y}_i$ es la predicción de dicha clasificación y n es el total de observaciones sobre las que se evalúa el modelo. La función $accuracy(x)$ devuelve un valor en el rango $[0, 1]$, y se interpreta como la proporción de clasificaciones correctas que el modelo ha realizado en sus predicciones sobre el total de observaciones evaluadas, de tal forma que cuanto más elevado sea el valor devuelto por esta función, mejor será el rendimiento del modelo. Esta medida puede interpretarse también en términos del error, de tal forma que $error(y, \hat{y}) = 1 - accuracy(y, \hat{y})$, aunque utilizaremos la primera interpretación para el caso práctico por conveniencia.

Por cada modelo que ajustemos, en primer lugar presentaremos una gráfica de dos dimensiones que compara la precisión obtenida con el número de epochs, tanto en el conjunto de entrenamiento como en el de validación. Esta gráfica no sólo permite ver la evolución del rendimiento del modelo a lo largo de los epochs, si no que también nos permite identificar si existe el problema de sobreajuste, que se manifiesta cuando existe una gran diferencia de precisión entre el ajuste del modelo en el conjunto de entrenamiento frente al de validación, y también puede informarnos sobre el posible uso de un *early stopping*. Las gráficas de cada modelo pueden verse en la Figura 4.2.

Los modelos que vamos a mostrar tendrán los mismos valores en algunos hiperparámetros, que detallamos a continuación:

- Hemos escogido dos tamaños de batch diferentes: 32 para el conjunto de entrenamiento y 16 para el conjunto de validación y el de test, para que todos los batches tengan el mismo tamaño (por ejemplo, en el conjunto de test tendríamos $\frac{10000}{16} = 625$ batches de tamaño 16, mientras que en el de entrenamiento tendríamos $\frac{40000}{32} = 1250$ batches de tamaño 32).

- Todos los modelos serán ajustados en un total de 25 epochs. En la práctica se suelen utilizar 50 epochs como número inicial, para a continuación incrementar o reducir este valor dependiendo del resultado de la gráfica precisión-epoch (por ejemplo, si se da una situación donde utilizar el *early stopping* o si la curva de precisión muestra potenciales mejoras con un mayor número de epochs). Sin embargo, decidimos utilizar 25 epochs tanto por las limitaciones del hardware utilizado para la estimación de los modelos como por el hecho de que es un número suficiente para mostrar de manera didáctica la utilidad de las técnicas de regularización en cada uno de los ejemplos

- El algoritmo de optimización a utilizar será **Adam**, del cual hemos hablado en el apartado 2.4. La razón principal de esta elección, aparte de otras propiedades, es el hecho de que funciona mejor que otros algoritmos como el RMSprop y el SGD para estimaciones con un número bajo de epochs, como es nuestro caso. Los valores que escogeremos para los hiperparámetros η (*learning*

rate), ρ y ρ_f serán 0,001, 0,999 y 0,9, respectivamente, que son los valores recomendados por (Kingma y Ba 2017).

- El número de neuronas y filtros utilizados en los distintos modelos se ha elegido en un sentido creciente (es decir, las primeras capas tendrán 32 neuronas y/o filtros mientras que la última capa intermedia tendrá 512)). La razón detrás de esta elección es por inspiración en la arquitectura de varias redes neuronales populares, de las cuales hablaremos en el siguiente apartado.

Establecidas estas configuraciones, tenemos que diseñar un modelo base sobre el que valorar las distintas técnicas de regularización.

4.2. Pautas para elegir un modelo base de clasificación de imágenes

A la hora de construir una red neuronal para la clasificación de imágenes, el uso de capas convolucionales en lugar de capas densas suele ofrecer buenos resultados en el rendimiento del modelo, tanto en la minimización de la función de pérdida como en la eficiencia en el empleo de recursos computacionales. Como función de activación en las capas intermedias utilizamos ReLU debido a sus propiedades, ya comentadas en el apartado 2.1.1, y la función *softmax* en la capa final, al ser una tarea de clasificación de más de dos categorías, lo cual requiere una transformación de los valores de $\mathbb{R}$ en un vector de probabilidades n-dimensional. En el caso de que nos encontremos con problemas de convergencia en la optimización (por ejemplo, valores de pérdida muy elevados), nos plantearemos sustituir ReLU por alguna de sus variantes antes presentadas.

Una forma para orientarnos a la hora de elegir una red neuronal adecuada en la clasificación de imágenes es observar qué clase de modelos ganadores hubo en las distintas ediciones de la competición ILSVRC, que son las siglas de *ImageNet Large Scale Visual Recognition Competition*, donde los participantes intentan obtener el mayor número de aciertos a la hora de clasificar una selección de imágenes con un modelo Deep Learning. Si bien es cierto que los modelos ganadores pudieron ganar por mera aleatoriedad a la hora de que sus creadores eligiesen un determinado diseño, en la práctica se suele tener como referencia los resultados de estas competiciones a la hora de diseñar redes neuronales eficientes para la tarea en cuestión. (Aggarwal 2018) realiza un repaso cronológico de los mejores modelos utilizados desde la edición de 2012 hasta la edición del año 2015, y señala cuales han sido las principales aportaciones de estos modelos a la práctica habitual en la elaboración de redes neuronales convolucionales para la clasificación de imágenes. Podemos resumir las conclusiones obtenidas en la

siguiente lista:

- La función de activación ReLU aplicada en las capas intermedias suele devolver mejores resultados que la función *tanh* o la sigmoide, siendo ambas popularmente utilizadas como activaciones intermedias hasta que el modelo **AlexNet** ganó la edición 2012 del ILSVRC, el cual utilizaba íntegramente ReLUs en todas sus capas intermedias.

- El uso de *strides* más pequeños en las capas convolucionales puede introducir mejoras en el rendimiento del modelo. Esta afirmación quedó en manifiesto en el modelo **ZFNet**, cuyo número de capas es idéntico al AlexNet, aunque con la diferencia de que se utilizaron *strides* de tamaño 2 en lugar de 4, consiguiendo así una reducción del error de test de un 4.3 % y ganando la edición 2013 del ILSVRC.

- Dado un número y tamaño concretos de filtros de capas convolucionales, es preferible añadir más filtros (así aumentando la profundidad de la red neuronal, como vimos en el apartado 3.2.3) y reducir sus dimensiones (para captar contornos más pequeños y específicos) para mejorar el rendimiento del modelo. El modelo **VGG** popularizó esta práctica al quedar en segundo lugar en la competición ILSVRC 2014.

- El uso de una red neuronal con capas convolucionales paralelas que contengan filtros de distinto tamaño puede devolver buenos resultados, al captar patrones que no dependan de otros patrones anteriormente captados. Así fue con el modelo **GoogLeNet**, que ganó la edición ILSVRC 2014, superando a VGG, el cual estaba construído de forma secuencial.

- Otorgar la posibilidad a la red neuronal de que elija un número de capas adaptado a la complejidad de cada patrón existente en la información suministrada puede acelerar la convergencia en el problema de optimización. Esta innovación la trajo el modelo **ResNet**, configurando una serie de *skips* en la red neuronal que permite a una capa en el orden i conectarse directamente a otra capa en $i + r$, donde $r \geq 2$. ResNet fue el modelo ganador del ILSVRC 2015.

Aunque todos estos modelos se han empleado para un conjunto de datos considerablemente más complejo que el que vamos a utilizar, estas indicaciones pueden orientarnos para diseñar una red neuronal adecuada a la tarea en cuestión, para luego aplicarle modificaciones según el resultado que nos devuelva en la práctica. En el Cuadro 4.1 se expone un resumen de las principales características de cada uno de los modelos que se han expuesto en la lista anterior.

Modelo	Nº capas	% error obtenido (ILSVRC)	Técnicas de regularización	Comentarios
AlexNet	8	15.4 %	C. Convolucional, Aumento de imagen, Dropout.	Filtros 11x11 con strides de tamaño 3
ZfNet	8	14.8 %	C. Convolucional, Aumento de imagen, Dropout	Filtros 11x11 con strides de tamaño 3
VGG	19	7.3 %	C. Convolucional, Max Pooling	Filtros 3x3 con strides de tamaño 1
GoogLeNet	22	6.7 %	C. Convolucional, Max Pooling, Average Pooling, Dropout	Filtros 112x112 hasta 1x1, con strides de tamaños 1 y 2
ResNet	152	3.6 %	C. Convolucional, Average Pooling, Dropout	Filtros de 7x7 y 3x3 con strides de tamaño 2 y skips

Cuadro 4.1: *Resumen de las principales características de las redes neuronales ganadoras del certamen ILSVRC en varias ediciones. Todos los modelos utilizan ReLU como función de activación intermedia y softmax como activación de la capa final. Información extraída de (Aggarwal 2018)*

4.3. Evaluación de modelos Deep Learning

4.3.1. Estimación con distintas técnicas de regularización

Basándonos en los modelos ganadores anteriormente expuestos y en los criterios indicados en el apartado 4.1, el modelo base que proponemos utiliza capas convolucionales con un número creciente de filtros, con una última capa densa de 512 neuronas antes de la capa final, la cual contará con tantas neuronas como clasificaciones existen en el dataset CIFAR10 (esto es, 10 neuronas). La estructura de este modelo puede observarse en la Figura 4.1, junto con los demás modelos que derivan del mismo. Al basarse el modelo mayoritariamente en capas convolucionales, la red neuronal no sólo podrá captar formas y contornos propios de cada uno de los objetos a clasificar, sino que también utilizará menos parámetros que en el caso que decidiésemos utilizar únicamente capas densas, actuando así como una técnica de regularización. Sin embargo, esto no implica que este modelo no pueda incurrir en un problema de sobreajuste. En la Figura 4.2 podemos ver dentro de la gráfica Base una gran diferencia en la curva de entrenamiento frente a la de validación, donde la primera roza el 100 % de precisión desde el epoch 7 hasta el 25, mientras que la de validación tiende alrededor del 63 % en todo el proceso de ajuste, por lo que hay una diferencia de al menos un 37 % de precisión en término medio. Al realizar una predicción con el modelo en el conjunto de test, obtuvimos una precisión del 63.17 %, similar al nivel de la curva de validación.

La primera variante que proponemos del modelo base es el uso de una penalización de la norma L_2 para la última capa densa intermedia de 512 neuronas, utilizando una penalización del 10 %. En la Figura 4.2, dentro de la ventana del modelo *Penal norma l2*, vemos la evolución del ajuste a lo largo de los epochs. Ambas curvas guardan poca distancia entre si, por lo que no parece haber síntomas de

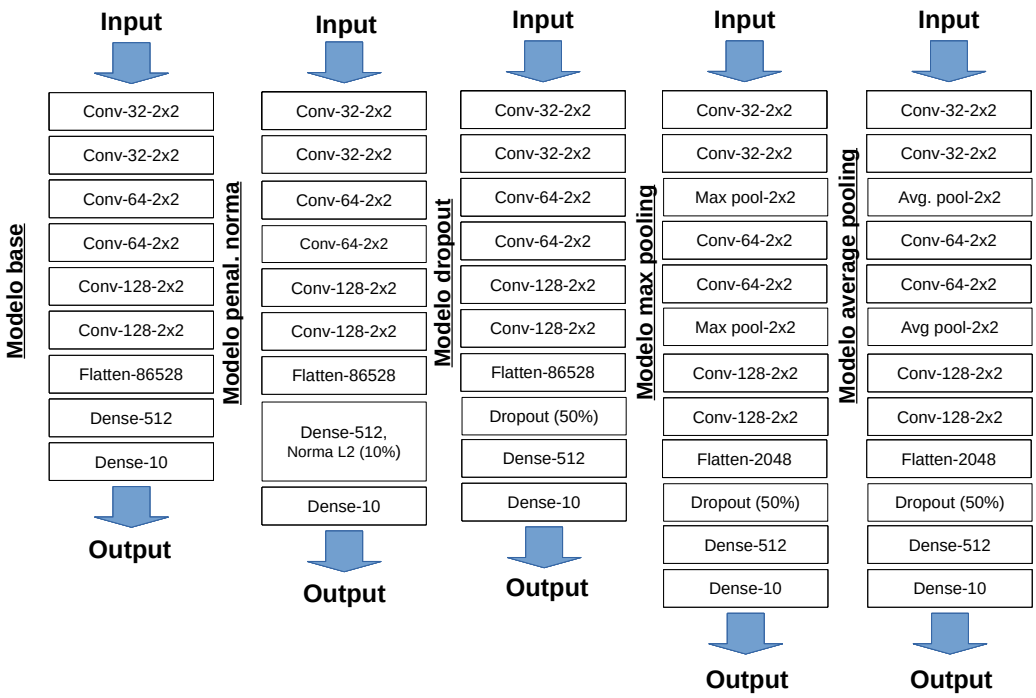


Figura 4.1: *Esquemas de los modelos propuestos en el caso práctico. Dentro de cada celda, se indica primero el tipo de capa, seguido del número de neuronas o filtros que contiene y, en el caso de las capas convolucionales, se indica el tamaño del filtro. Para Dropout y penalización de la norma, se indica también la penalización. Todos los modelos utilizan ReLU como función de activación intermedia y softmax como activación en la capa final.*

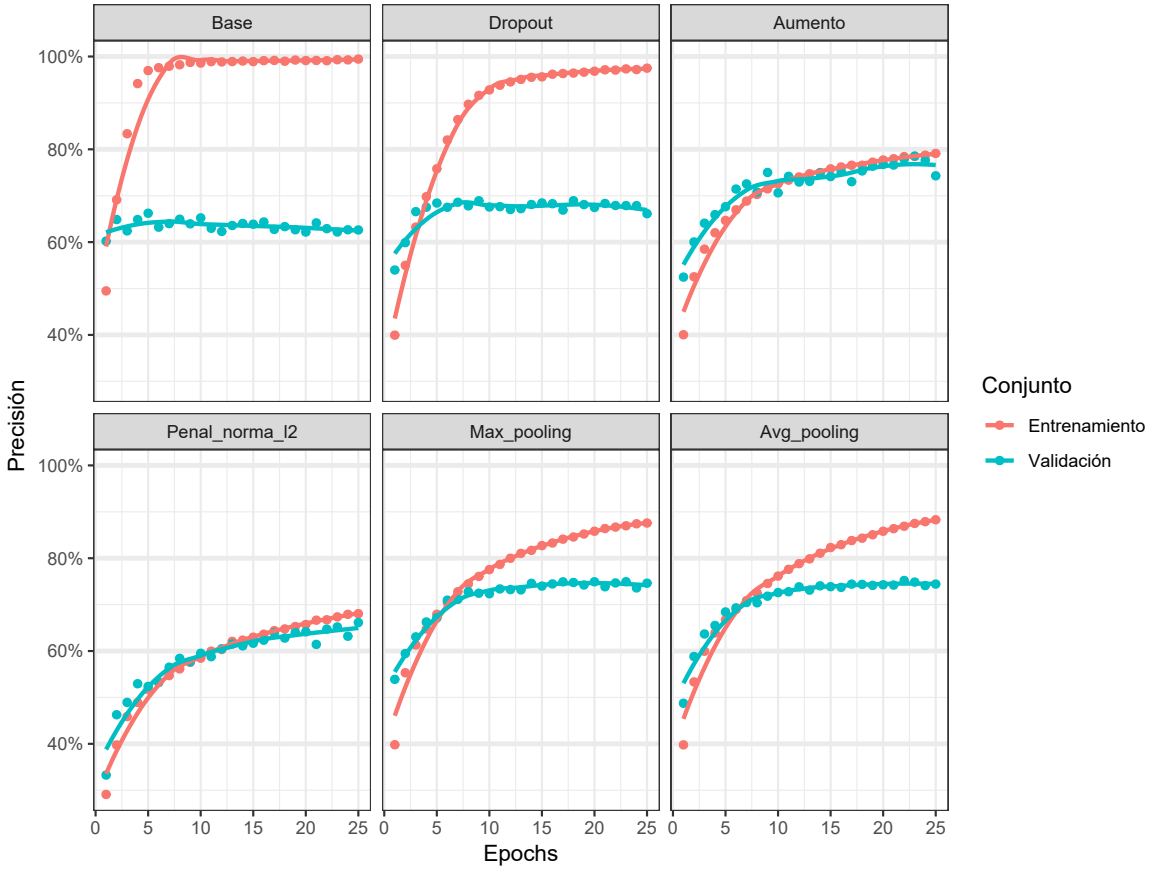


Figura 4.2: *Evolución del entrenamiento de los modelos propuestos a lo largo de los epochs. El indicador de precisión es la medida de rendimiento y cuantifica cuántas categorías correctas ha alcanzado el modelo en el conjunto de datos y epoch pertinentes.*

sobreajuste en el modelo. En el último epoch, la curva de validación ronda el 65 % de precisión, que es dos dígitos mayor que el valor que presentaba la misma curva en el modelo base. Por último, la tendencia creciente de ambas curvas en los últimos epochs parece indicar que la estimación se habría beneficiado del uso de más de 25 epochs. Al realizar la predicción en el conjunto de test, obtuvimos un valor de 66.2 %, unas tres décimas mayor que en la predicción del modelo base.

Otra alternativa que proponemos es generar otra red neuronal idéntica al modelo *Base* pero añadiendo una capa *dropout* antes de la penúltima capa (es decir, la capa densa con 512 neuronas). Con un ratio del 50 %, esta técnica desactiva cualquier filtro de la capa previa con una probabilidad del 50 %, con el objetivo de mejorar la generalización de las predicciones. Nuevamente, el resultado puede observarse en la Figura 4.2 dentro de la ventana *Dropout*. En primer lugar, aunque la curva de validación está a un nivel mayor que en el caso del modelo *Base*, rondando el 68 % de precisión frente al 65 % de antes, sigue presentándose sobreajuste, con una diferencia de un 30 % entre ambas curvas en el epoch 25. La razón de por qué sigue habiendo sobreajuste puede deberse a que la ratio del Dropout no es lo suficientemente elevada: Para un modelo de 44 millones de parámetros, un dropout del 50 % dejaría activos en término medio unos 22 millones de parámetros, lo cual podrían considerarse demasiados para la complejidad de los datos empleados. Aunque sería posible aumentar la ratio del Dropout por encima del 50 % para intentar reducir más el sobreajuste, en general no es una práctica recomendada (Srivastava et al. 2014). En el conjunto de test, obtuvimos una predicción del 66.1 %.

En base al modelo *Dropout*, decidimos construir dos nuevos modelos: Uno con la técnica *Max Pooling* y otro con *Average Pooling*. Decidimos aplicar dos capas de esta técnica: Una después del conjunto de capas convolucionales de 32 filtros, y otra después del conjunto de 64. El resultado para ambas técnicas puede observarse en la Figura 4.2, en los modelos *Max pooling* y *Avg pooling*. En ambos modelos podemos ver cómo el sobreajuste ha sido atenuado en mayor medida que en caso del modelo Dropout, pero no al mismo nivel que en el caso de la penalización de la norma L_2. Sin embargo, aunque *Penal norma l2* parece haber reducido el sobreajuste con mayor efectividad, la curva de validación de ambos modelos *pooling* está a un nivel mayor que en el primer caso, rondando el 75 % de precisión a partir del epoch 7 frente al 63 % del otro ejemplo, por lo que ambos poolings parecen haber sido efectivos. Finalmente, no parece haber diferencias significativas entre *Max* y *Average pooling*, presentando curvas similares, por lo que no percibimos la superioridad en rendimiento del *Max* sobre el *Average* que (I. Goodfellow et al. 2016) indicó. En cuanto a las predicciones en el conjunto de test, obtenemos un 74.1 % con el modelo *Max pooling* y un 74.3 % para el caso del *Average pooling*.

Por último, decidimos recurrir de nuevo al modelo *Base*, pero esta vez aplicando la técnica del aumento de imágenes.[1] La modificación de las imágenes se realiza mediante la selección de un rango de posibles valores porcentuales (Por ejemplo, si establecemos un 15 % para desplazamientos horizontales, la imagen será desplazada a la izquierda o a la derecha entre un 0 % y un 15 % en cada epoch). Las modificaciones aplicadas en el preprocesamiento son las siguientes:

- Desplazamientos hasta el 15 % horizontal y 10 % vertical.
- Hasta un 10 % de inclinación de la imagen en el sentido contrario al de las agujas del reloj.
- Hasta un 10 % de zoom adentro o afuera de la imagen (esto es, de 90 % a 110 % de enfoque).
- Volteos horizontales aleatorios de la imagen (un mismo animal puede ser fotografiado desde el perfil derecho o el izquierdo, por ejemplo).
- En cualquiera de las modificaciones, se rellenan los espacios vacíos con el píxel más cercano (Véase Figura 3.6).

El resultado de la estimación puede observarse en el modelo *Aumento* dentro de la Figura 4.2. Las curvas tanto de entrenamiento como de validación se encuentran muy cercanas en todo el proceso, especialmente a partir del epoch 10, por lo que no parece haber problema de sobreajuste. Al igual que con el modelo *Penal norma l2*, la tendencia creciente de ambas curvas en los últimos epochs sugiere que el modelo podría mejorar su rendimiento con el uso de más de 25 epochs. La precisión devuelta en el conjunto de test por este modelo ha resultado del 74.2 %, muy cercana a las predicciones de los modelos con técnicas *pooling*.

4.3.2. Uso de redes pre-entrenadas: VGG-16

Aunque es posible diseñar redes neuronales y estimar sus parámetros desde cero, en la práctica suelen descargarse modelos Deep Learning ya entrenados para su aplicación en cualquiera de sus posibles tareas, aprovechando la información aprendida de otros conjuntos de datos. Por ejemplo, es el caso de los modelos ganadores de los que ya hablamos en el apartado 4.2, que pueden descargarse utilizando la herramienta Keras. La ventaja de esta opción es la de no tener que incurrir en un largo proceso de estimación que puede llevar horas, días e incluso semanas si queremos disponer, por ejemplo, de un modelo de clasificación de imágenes con alto rendimiento de sus predicciones. Parece intuitivo pensar que un modelo como el VGG, que ha quedado en segundo lugar en una competición con un dataset

[1] Este método de regularización no implica cambios en la arquitectura de la red neuronal, sino en el pre-procesamiento de las imágenes utilizadas como datos para el modelo. Por esta razón, el modelo no aparece representado en la Figura 4.1

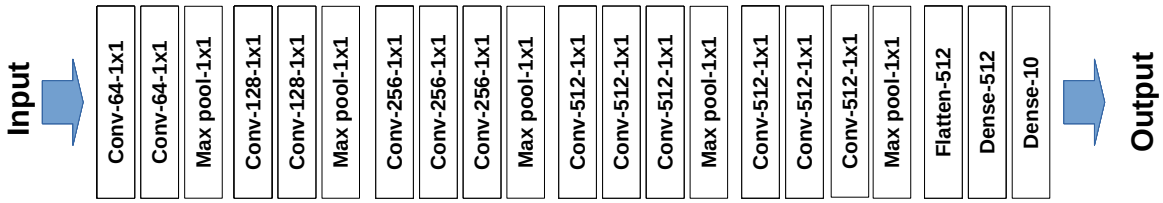


Figura 4.3: *Estructura del modelo VGG16. Todas las capas contienen una activación ReLU, excepto la última que contiene un softmax para la clasificación.*

de más de 80 millones de imágenes y con 1000 clasificaciones distintas, pueda no tener problemas en identificar 10 clases de objetos diferentes en un conjunto de 60.000 imágenes. En este sub-apartado utilizaremos el modelo VGG16, representado en la Figura 4.3, con el dataset CIFAR10, con la finalidad de mostrar los beneficios del uso de una red pre-entrenada para la clasificación.

Antes de utilizar una red pre-entrenada para la predicción, conviene adaptarla a la complejidad de nuestro conjunto de datos. Todos los parámetros del modelo VGG pueden ser estimados de nuevo con el dataset CIFAR10 como si fuese una red neuronal recién ensamblada, pero volver a reajustar la red neuronal entera puede hacer que perdamos información crucial derivadas del entreno previo de VGG con el dataset de Imagenet, por lo que cabe preguntarse si nos puede resultar útil re-estimar algún conjunto de parámetros dentro del modelo descargado. (I. J. Goodfellow et al. 2014) recomienda únicamente re-estimar los parámetros contenidos en las últimas capas, pues la mayor parte del beneficio en el uso de redes pre-entrenadas proviene de la información que traen consigo sobre formas o contornos generales de los objetos a identificar, y no así de las más específicas: Por ejemplo, VGG ha sido posiblemente entrenada con imágenes de cientos de miles de perros distintos, pero las imágenes de perros de CIFAR10 podrían contener ciertas idiosincrasias que el modelo VGG no ha contemplado anteriormente. Para poner a prueba esta idea, propondremos dos modelos derivados de VGG: Uno denominado *VGG congelado*, donde no modificaremos ninguna de sus estimaciones, salvo las de las capas adicionales que hemos añadido (la capa *flatten* y las dos capas densas), y otro idéntico al primero con el nombre *VGG reajustado*, que será idéntico al primero pero con la diferencia de que también re-estimaremos los parámetros de la última capa convolucional.

Los resultados del ajuste de ambos modelos se muestran en la Figura 4.4. Con alrededor de 14 millones de parámetros en cada uno de los modelos, se presenta un problema de sobreajuste. No obstante, también puede observarse la diferencia de estimación en ambos modelos: No sólo *VGG reajustado* presenta una curva de validación mayor que *VGG congelado*, con una ganancia en término

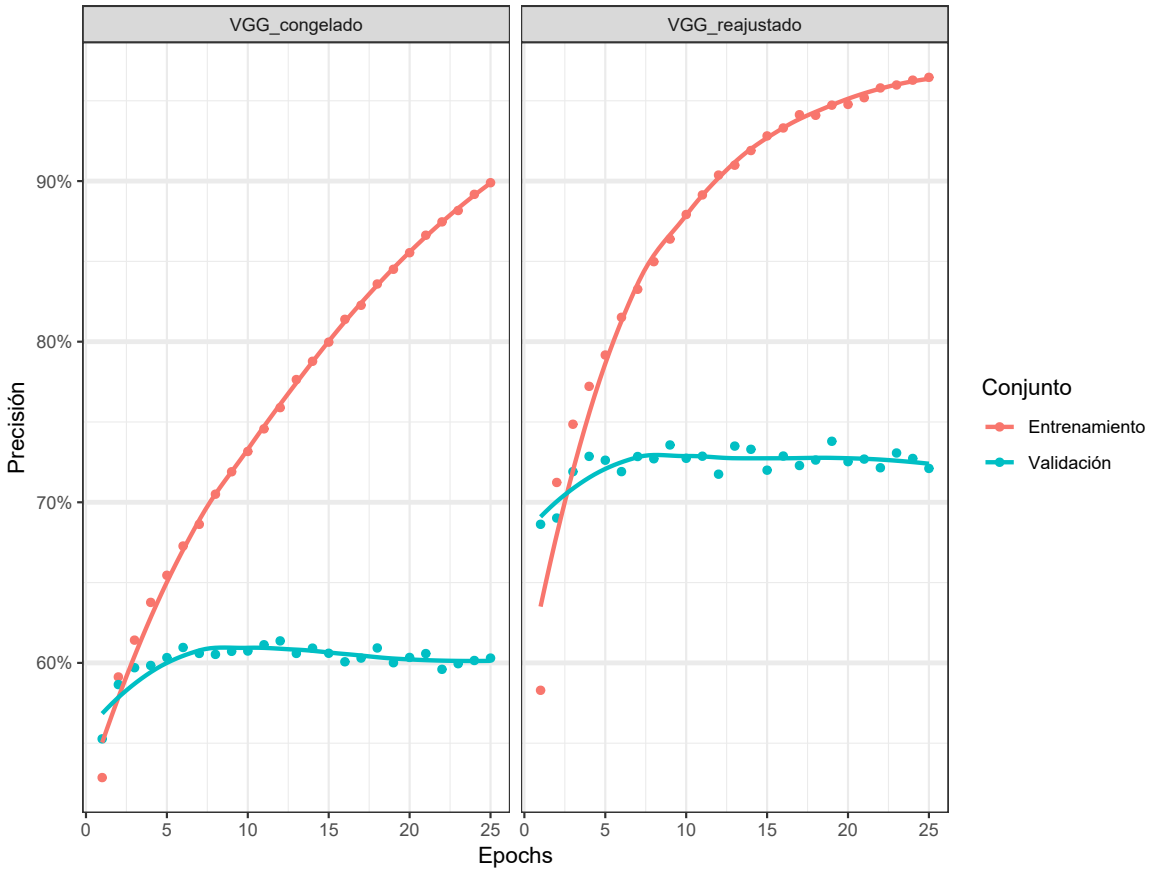


Figura 4.4: *Estimación parcial de los modelos a partir de la red pre-entrenada VGG. En el modelo VGG congelado, todos los parámetros estimados están congelados excepto los de las dos últimas capas densas (intermedia y final), mientras que en VGG reajustado también permitimos reestimar los parámetros de la última capa convolucional.*

medio de un 13 % de precisión en todos los epochs, sino que además en el conjunto de entrenamiento *VGG congelado* alcanza el 90 % de precisión en el epoch 25, cuando *VGG reajustado* lo hace alrededor del epoch 13, lo cual pone de manifiesto la poca eficiencia en el aprendizaje que supone congelar todos los parámetros en el modelo descargado. Al realizar las predicciones en el conjunto de test, *VGG congelado* devuelve una precisión del 60.32 %, mientras que *VGG reajustado* en cambio devuelve un 72.47 %.

4.4. Resultado de las predicciones de los modelos

En el Cuadro 4.2 se puede consultar el resultado de las predicciones de todos los modelos en los datos del conjunto de test. En general, si tomamos como referencia el resultado del modelo *Base*, con

una precisión del 63.2 % en sus predicciones, podemos ver cómo los demás modelos derivados mejoran su resultado en mayor o menor medida, lo cual refleja la utilidad de las técnicas de regularización estudiadas. Los modelos *Dropout* y *Penal Norma L2* ofrecen un aumento del 3 % mayor que el modelo de referencia, mientras que los modelos con técnicas *Pooling* ofrecen en cambio un incremento de la precisión en un 11 % más.

El resultado del modelo *Aumento*, que aunque su diseño es exactamente igual al de *Base*, ofrece también un 11 % más de precisión únicamente utilizando técnicas de preprocesamiento de imágenes, por lo que posiblemente modelos como *Max* y *Average Pooling* se podrían ver beneficiados también por esta técnica.

Finalmente, refiriéndonos al ámbito de las redes pre-entrenadas, *VGG sin reajuste* ofrece un resultado incluso menor que el del modelo *Base*, con aproximadamente un 3 % menos de precisión, mientras que el modelo *VGG con reajuste* ofrece alrededor de un 9 % más de precisión, dejando en manifiesto la importancia de adaptar los parámetros de las últimas capas de un modelo pre-entrenado al dataset que utilicemos, tal y como (I. J. Goodfellow et al. 2014) recomienda. A pesar de la mejora marginal ofrecida por *VGG con reajuste* con respecto al modelo *Base*, no ha devuelto el mejor resultado de todos los modelos, siendo superado por los modelos *Pooling* y *Aumento*. VGG fue un modelo con buen rendimiento en una competición de clasificación con 1000 etiquetas distintas, por lo que cabe plantearse la hipótesis de que en un entorno de únicamente 10 clasificaciones puede no ser lo suficientemente útil el utilizar una red pre-entrenada frente a generar una red desde cero.

Modelo	**Precisión (%)**
Base	63.2
Dropout	66.1
Aumento	74.2
Penal. Norma L2	66.2
Max Pooling	74.1
Average Pooling	74.3
VGG sin reajuste	60.3
VGG con reajuste	72.5

Cuadro 4.2: Resumen de las predicciones alcanzadas con todos los modelos en el dataset CIFAR10

Capítulo 5

Conclusiones

En el presente trabajo hemos realizado un acercamiento a los modelos Deep Learning, introduciendo primero los conceptos esenciales del ámbito Machine Learning, para después definir las redes neuronales, su estructura, funcionamiento y problemas relacionados con su optimización, presentando así varias soluciones que ejemplificamos en la práctica con una tarea de clasificación de imágenes, la cual es una de las tareas que este tipo de modelos desempeñan comparativamente mejor que otros modelos de Machine Learning. Utilizando un conjunto de imágenes con diez clasificaciones distintas, hemos buscado utilizar un modelo básico como referencia y con sobreajuste, ofreciendo una precisión del 63 %. Debido a la enorme personalización de este tipo de modelos, decidimos generar modelos derivados con una única técnica de regularización cada uno sobre el modelo básico con fines didácticos, para ejemplificar de esta forma la utilidad de las técnicas presentadas. De esta manera, se obtuvo mejoras de precisión de hasta un 11 %. Es posible generar una red neuronal que combine todas estas técnicas y que ofrezca incluso mayores rendimientos que los aquí presentados, sobretodo en aquellos modelos donde siguieron manifestando sobreajuste incluso después de la aplicación de alguna de estas técnicas, como es el caso de ambos modelos *pooling*. Demostramos así la utilidad de los modelos Deep Learning y de varias de sus técnicas de regularización para la clasificación de imágenes. Para posteriores trabajos, se contempla la introducción de temas más avanzados en Deep Learning, como los modelos de *Aprendizaje Reforzado* y de *Redes Generales Adversarias*, que en los últimos años han experimentado un gran crecimiento, y la introducción a otras tareas también conocidas, como el reconocimiento de voz o la identificación de secuencias de texto.

Bibliografía

[1] Abien Fred Agarap. "Deep Learning using Rectified Linear Units (ReLU)". en. En: *arXiv:1803.08375 [cs, stat]* (mar. de 2018). arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375 (visitado 07-08-2019).

[2] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. en. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-94462-3 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0. URL: http://link.springer.com/10.1007/978-3-319-94463-0 (visitado 16-06-2020).

[3] JJ Allaire y FranÃ§ois Chollet. *keras: R Interface to 'Keras'*. R package version 2.2.5.0. 2019.

[4] Martin Anthony y Peter L. Bartlett. *Neural network learning: theoretical foundations*. eng. Digitally printed vers. OCLC: 836931870. Cambridge: Cambridge Univ. Press, 2009. ISBN: 978-0-521-11862-0 978-0-521-57353-5.

[5] James Bergstra y Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". En: *Journal of Machine Learning Research* 13 (2012), págs. 281-305.

[6] François Chollet y Joseph J. Allaire. *Deep learning with R*. OCLC: on1022850710. Shelter Island, NY: Manning Publications Co, 2018. ISBN: 978-1-61729-554-6.

[7] Djork-Arné Clevert, Thomas Unterthiner y Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". en. En: *arXiv:1511.07289 [cs]* (nov. de 2015). arXiv: 1511.07289. URL: http://arxiv.org/abs/1511.07289 (visitado 07-08-2019).

[8] John Duchi, Elad Hazan y Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". en. En: *Journal of Machine Learning Research* 12 (2011), págs. 2121-2159.

[9] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 978-0-262-03561-3.

[10] Ian J. Goodfellow, Oriol Vinyals y Andrew M. Saxe. "Qualitatively characterizing neural network optimization problems". en. En: *arXiv:1412.6544 [cs, stat]* (dic. de 2014). arXiv: 1412.6544. URL: `http://arxiv.org/abs/1412.6544` (visitado 07-08-2019).

[11] Richard H R Hahnloser et al. "Digital selection and analogue ampli®cation coexist in a cortex-inspired silicon circuit". en. En: 405 (2000), pág. 7.

[12] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". En: *arXiv:1502.01852 [cs]* (2015).

[13] Henderson y Veterman. "Building multiple regression models interactively". En: *Biometrics* 37 (1981), págs. 391-411.

[14] G Hinton. *Neural networks for machine learning*. Video lectures. 2012.

[15] Diederik P. Kingma y Jimmy Ba. "Adam: A Method for Stochastic Optimization". en. En: *arXiv:1412.6980 [cs]* (ene. de 2017). arXiv: 1412.6980. URL: `http://arxiv.org/abs/1412.6980` (visitado 01-07-2020).

[16] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". en. En: (2009), pág. 60.

[17] Tom M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[18] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. "Learning representations by back-propagating errors". En: *Nature* 323.9 (1986).

[19] Wei Shen y Rujie Liu. "Tackling Early Sparse Gradients in Softmax Activation Using Leaky Squared Euclidean Distance". en. En: *arXiv:1811.10779 [cs]* (nov. de 2018). arXiv: 1811.10779. URL: `http://arxiv.org/abs/1811.10779` (visitado 07-08-2019).

[20] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". en. En: (2014), pág. 30.

[21] R Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria, 2020.

[22] M.D. Zeiler et al. "On rectified linear units for speech processing". en. En: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. Vancouver, BC, Canada: IEEE, mayo de 2013, págs. 3517-3521. ISBN: 978-1-4799-0356-6. DOI: `10.1109/ICASSP.2013.6638312`. URL: `http://ieeexplore.ieee.org/document/6638312/` (visitado 07-08-2019).