Universidad de Valladolid

Facultad de Ciencias

TRABAJO DE FIN DE GRADO

Grado en Matemáticas

Redes Neuronales Convolucionales

Autor: Elena Martín de Diego
Tutor: Eustasio del Barrio Tellado

Julio 2022

Índice general

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos 1
1.3. Fases del proyecto 2

2. Aprendizaje automático **3**
2.1. Aprendizaje supervisado 5
2.2. Regularización 11
2.3. Algoritmos de optimización de descenso del gradiente 15
2.4. Conclusiones 24

3. Aprendizaje profundo **25**
3.1. Redes neuronales artificiales 28
3.1.1. Funciones de activación 29
3.2. Redes neuronales convolucionales 38
3.2.1. La operación convolución 39
3.2.2. Componentes de una red neuronal convolucional 41
3.2.3. Arquitecturas propuestas para el proyecto 43
3.3. Conclusiones 46

4. Estudio experimental **47**
4.1. Objetivos 47
4.2. Estudio del dataset 47
4.3. Descripción metodología y especificaciones 49
4.3.1. Hardware y Software utilizado en el proyecto 49
4.4. Estudios piloto 49
4.5. Análisis de los datos 53
4.5.1. Procesamiento de los datos 53
4.5.2. Modelo de aprendizaje automático 54

4.5.3. Entrenamiento del modelo de aprendizaje automático gracias al conjunto de entrenamiento . . . 56
4.6. Resultados obtenidos . . . 56

5. Conclusiones finales y futuras líneas de trabajo 58

6. Anexos 60
6.1. Problemas de aprendizaje convexos . . . 60
6.1.1. Problemas de aprendizaje convexos . . . 63
6.1.2. Funciones de pérdida sustituta . . . 64
6.2. Definiciones relativas a la Teoría de la Información . . . 64
6.3. Programas escritos en Python . . . 67
6.3.1. Modelo red neuronal convolucional LeNet-5 para el conjunto de datos MNIST . . . 67
6.3.2. Modelo red neuronal totalmente conectada . . . 71
6.3.3. Modelo Red Neuronal Convolucional Le-Net-5 para el conjunto de datos de interés . . . 73
6.3.4. Modelo Red Neuronal Convolucional para el conjunto de datos de interés 75
6.3.5. Código para la visualización del conjunto de datos . . . 78

Índice de figuras

2.0.1. Programación tradicional frente al Aprendizaje Automático 3
2.0.2. Dados los datos, produce conocimiento sobre ellos. 4
2.0.3. Dada la asociación, devuelve un programa aproximado de dicha asociación. . 4
2.1.1. Gráfico explicativo de lo que sucede con los errores definidos al variar la complejidad de la clase de hipótesis 10
2.2.1. Función fuertemente convexa [14] 12
2.3.1. Ejemplo de la técnica de descenso del gradiente donde se ilustra como el procedimiento usa las derivadas para buscar el mínimo [6] 16
2.3.2. En la imagen de la izquierda se muestra la tangente de la función f en el punto $\boldsymbol{w}$, mientras que en la imagen de la derecha se muestran varios subgradientes para una función convexa pero no diferenciable. [[14], Figura 14.2] 19
2.3.3. En la imagen de la izquierda se muestra el algoritmo del descenso del gradiente y en la derecha el algoritmo del descenso del gradiente estocástico, ambos con el objetivo de minimizar la función $1{,}25(x+6)^2+(y-8)^2$ [[14], Sección 14.2] 21

3.0.1. Gráfico explicativo regla lineal para un dominio bidimensional. Elaboración propia. 25
3.0.2. Esquema de los pasos que realiza un clasificador lineal. Fuente: enlace 26
3.0.3. Ejemplo ilustrativo. Elaboración propia. 26
3.0.4. Arquitectura formada por nodos organizados en capas capaz de resolver el ejemplo propuesto. Imagen: elaboración propia. 27
3.0.5. Función no continua utilizada en la anterior arquitectura para resolver el ejemplo propuesto, $h_{\boldsymbol{w}}(\boldsymbol{x}) = \langle \boldsymbol{w} \ , \ \boldsymbol{x} \rangle = \langle (b, w_1, .., w_d) \ , \ (1, x_1, ..., x_d) \rangle$. Imagen: elaboración propia. 27
3.1.1. Esquema gráfico representativo de la red neuronal feedforward. Nota: en la descripción matemática el input $v_{0,1}$ es la constante $(o_{0,0})$ y los demás los elementos de $\boldsymbol{x}$. Fuente: [[14], Sección 20.1] 28
3.1.2. Función sigmoide: diferenciable y monótona por tanto el gradiente está definido en todos sus puntos. Notamos que $\phi_{sig}(0) = 1/2$. Imagen: elaboración propia. 30

3.1.3. Función de pérdida logística en azul y función de pérdida 0-1 en naranja. Nota: la función ϕ_{log} es dividida por log(2) para conseguir $\phi_{log}(0) = 1$. Imagen: elaboración propia. . . . 31
3.1.4. Función tangencial hiperbólica: diferenciable y monótona. Imagen: elaboración propia. . . . 32
3.1.5. Función ReLU. Imagen: elaboración propia. . . . 32
3.1.6. Funciones de activación mencionadas. Imagen: elaboración propia. . . . 33
3.1.7. Fuente: [1] . . . 36
3.2.1. Ejemplo de una convolución de dos dimensiones sin trasponer el kernel. Fuente: [[6], Sección 9.2] . . . 40
3.2.2. Parámetros compartidos, [[6], Sección 9.2] . . . 40
3.2.3. Esquema de una red neuronal convolucional cualquiera. Fuente: enlace . . . 41
3.2.4. [[6], Figura 9.7] . . . 42
3.2.5. Arquitectura LeNet-5, inicialmente utilizada para leer códigos postales y dígitos (ofrecido en [9]). . . . 43
3.2.6. Arquitectura Vgg-16 (abajo) y Vgg-19 (arriba) Enlace. . . . 44
3.2.7. Arquitectura DenseNet, [8]. Las capas entre dos bloques adyacentes se denominan capas de transición, las cuales modifican el tamaño a través de la convolución y la agrupación. . . . 44
3.2.8. Arquitectura DenseNet para el conjunto de datos ImageNet [8]. Para nuestro conjunto de datos la última capa solo contiene una unidad. . . . 46

4.2.1. https://github.com/basveeling/pcam . . . 48
4.2.2. Ejemplos de los dos conjuntos con la etiqueta asociada correspondiente. . . . 48
4.4.1. Precisión y pérdida obtenida con la arquitectura LeNet-5 para el conjunto de datos MNIST. . . . 50
4.4.2. Prediciones para el conjunto de validación después del entrenamiento del modelo. 50
4.4.3. . . . 51
4.4.4. Comparación de la pérdida obtenida después de entrenar 30 y 100 veces respectivamente. . . . 52
4.5.1. Elemento del conjunto de entrenamiento . . . 53
4.5.2. Imagen de la figura 4.5.1 transformada en una matriz donde solo se muestran los valores asociados al color verde por simplificar. Dentro de la matriz de tamaño 96×96 se ha amplificado la submatriz central de tamaño 32×32. . . 53
4.5.3. Ejemplo simbólico de la primera parte de la capa de convolución, donde se puede apreciar que las matrices que actúan como kernels tienen una tercera dimensión acorde con los valores RGB. Fuente: enlace . . . 55
4.5.4. Ejemplo sencillo de la función max pooling y average pooling. Elaboración propia. . . . 55
4.5.5. Especificaciones en Python del modelo . . . 56
4.6.1. Resultados obtenidos para el modelo de la figura 4.5.5 . . . 57

6.1.1. Función convexa, [[14], Definición 12.2] . . . 60
6.1.2. Posible función sustituta para la pérdida 0-1, [[14], Sección 12.3] . . . 64

Abstract

Artificial neural networks are a class of machine learning algorithms involved in many of the most spectacular applications in Artificial Intelligence. Deep neural networks or multilayer networks produce the best empirical results in the classification of images or texts. From a theoretical point of view, understanding the reasons for the success of these algorithms is still a pending issue. In addition to the convergence problems of the learning algorithms, the enormous overparameterization of many types of neural networks makes, perhaps, very likely that the classification rules obtained following this method suffer from overfitting. The goal of this project is to study the design of neural networks adapted to image analysis, without abundant parametrization, but oriented to take advantage of the special structure of this type of data. The gain of this approach in terms of control of the overfit will be studied and applied to the classification of some appropriate image dataset.

CAPÍTULO 1

Introducción

El presente capítulo describe el contexto del trabajo desarrollado, seguido de los objetivos del mismo y finaliza con su estructura.

1.1. Motivación

Ser capaz de automatizar la detección de cáncer con metástasis en exploraciones histopatológicas gracias al aprendizaje automático y en especial al modelo de redes neuronales, es un campo con un potencial prometedor para la utilidad clínica. La clasificación de imágenes médicas implica la extracción de entidades de la imagen y la asignación de etiquetas utilizando las características extraídas. Gracias al estudio del modelo de redes neuronales convolucionales ha tenido un gran avance en el rendimiento. El modelo de Redes Neuronales Convolucionales está formado por capas convolucionales cuya función principal es aprender y extraer las características necesarias para una comprensión eficiente de la imagen. Este modelo comenzó a ganar popularidad en el año 2012, debido a AlexNet, que derrotó a todos los demás con un récord en precisión y baja tasa de error en *ImageNet Large Scale Visual Recognition Competition 2012* [16]. A su vez, el modelo de Redes Neuronales Convolucionales tuvo un gran avance cuando GoogleNet lo utilizó para detectar cáncer con una precisión del 89 % mientras que los patólogos podían lograr la precisión de sólo el 73 % [11].

1.2. Objetivos

La idea que se persigue con la realización de este proyecto es profundizar en el campo de las redes neuronales y, en particular, en el uso de las redes neuronales convolucionales. A continuación se detallan los requisitos necesarios para la realización de este Trabajo Final de grado, los cuales son:

- Entender los conceptos relacionados con el aprendizaje automático supervisado, en particular, el problema de clasificación binaria, desde un punto de vista matemático.

- Entender la motivación de la utilización de redes profundas o multicapa.
- Estudio del modelo Redes Neuronales Convolucionales.
- Estudio experimental: red neuronal convolucional aplicado a un conjunto de datos en concreto.

1.3. Fases del proyecto

Las principales fases en la que se va a dividir este Trabajo Final de Grado se explicarán de una forma breve y concisa:

- El capítulo número dos está dedicado a estudiar los conceptos relacionados con el aprendizaje automático, en particular con el aprendizaje supervisado, donde se dispone de un conjunto de ejemplos etiquetados como datos de entrenamiento y el objetivo es realizar predicciones. Serán estudiadas con detalle las tres ideas importantes para elegir un algoritmo de aprendizaje: la familia de modelos, la función de coste (riesgo) y los posibles procedimientos de optimización.
- En el capítulo número tres se motivará el uso de modelos más complejos como las redes neuronales para resolver problemas aparentemente sencillos como la separación de puntos del dominio en función de la clase a la que pertenecen cuando no es posible separarlos a través de un hiperplano. A su vez, el modelo de redes neuronales convolucionales será estudiado en detalle.
- En el último capítulo se presentarán los experimentos realizados con el objetivo de ilustrar el modelo de redes neuronales convolucionales aplicado a un conjunto de datos en concreto. Los códigos en el lenguaje Python se pueden encontrar en los Anexos.

Palabras clave: Aprendizaje automático, aprendizaje supervisado, función de pérdida, función de riesgo, exceso de riesgo, aprendizaje profundo, redes neuronales artificiales, redes neuronales convolucionales.

CAPÍTULO 2

Aprendizaje automático

El **aprendizaje automático**, *o machine learning* en inglés, puede definirse como un conjunto de métodos computacionales cuyo objetivo es la detección automatizada de patrones en los datos o la predicción de comportamientos a partir de la información previa disponible. Estadística, teoría de la información, teoría de aproximación y optimización son algunas de las disciplinas dentro de las Matemáticas que comparten aspectos con el aprendizaje automático, rama de Ciencias de la Computación. Concretamente, el aprendizaje automático involucra datos y éstos deben ser descritos utilizando un marco estadístico. Sin embargo, la principal diferencia entre ambas disciplinas es su propósito, esto es, mientras que los modelos de aprendizaje automático desean predecir comportamientos a partir de los datos, el objetivo de los modelos estadísticos es inducir las relaciones entre las variables.

El aprendizaje automático introduce un nuevo paradigma en el campo de la tecnología de la información, resumido en la siguiente imagen:

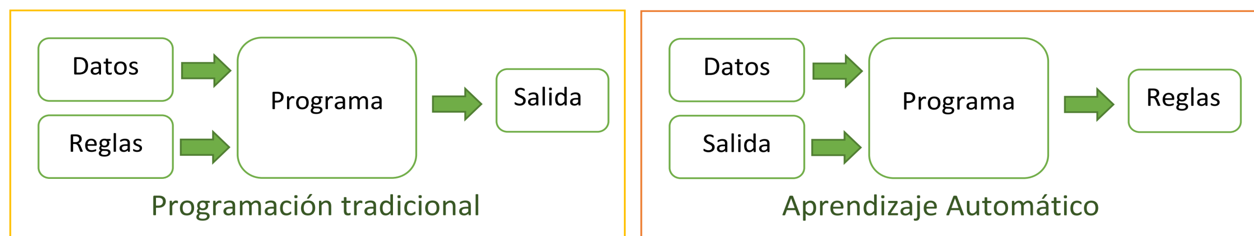


Figura 2.0.1: Programación tradicional frente al Aprendizaje Automático

En la primera imagen, dados los datos y las reglas, el programa es capaz de proporcionar la salida correspondiente; mientras que, en la segunda, nos gustaría que el sistema 'aprenda' gracias a la experiencia proporcionada (datos/salida) y al final del proceso, sea capaz de proporcionar conocimiento sobre los datos, que normalmente toma la forma de un nuevo programa que realiza cierta tarea. *Aprendizaje* será entendido en los términos que definió Tom. M. Mitchell en [12], es decir, *"Un programa se dice que aprende de la experiencia E con*

respecto a alguna clase de tareas T y la medida de rendimiento P, si su rendimiento en las tareas en T, medido por P, mejora con la experiencia E". Teniendo en cuenta esta definición, un *algoritmo de aprendizaje* será aquel capaz de aprender de los datos [[6], Capítulo 5].

- **Tarea:** Una tarea de aprendizaje automático es el tipo de predicción o inferencia que se quiere realizar, en función del problema y los datos disponibles. Ésta es descrita, generalmente, en términos de cómo el sistema de aprendizaje debe procesar un ejemplo. Algunas de las tareas estudiadas ampliamente son: clasificación, regresión, *clustering*, reducción de la dimensionalidad, etc [[6], Capítulo 5].

- **La experiencia:** las tareas de aprendizaje pueden ser divididas en función del tipo de información que tienen los algoritmos durante el proceso de aprendizaje [[6], Capítulo 5].

  1. **Aprendizaje no supervisado:** A priori, sólo se dispone de datos de entrenamiento sin etiqueta y el objetivo es predecir la distribución de probabilidad que generó el conjunto de datos. Por este motivo puede ser difícil evaluar cuantitativamente el rendimiento. La agrupación en clústeres y la reducción de la dimensionalidad son algunos ejemplos.

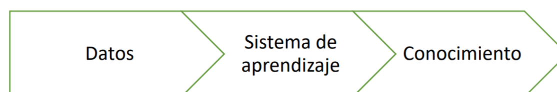


Figura 2.0.2: Dados los datos, produce conocimiento sobre ellos.

  2. **Aprendizaje supervisado:** Se dispone de un conjunto de ejemplos etiquetados como datos de entrenamiento y el objetivo es realizar predicciones. Si las posibles etiquetas pertenecen a un conjunto discreto, se denominará problema de clasificación: en el supuesto de ser dos categorias, véase, $\{0, 1\}$ o $\{-1, 1\}$, el problema se conoce como clasificación binaria; mientras que si las posibles predicciones varían en un conjunto discreto de mayor tamaño se denominará problema de clasificación multiclase. En cambio, si las posibles etiquetas pertenecen a un conjunto de valores continuo hablaremos de regresión. En este trabajo nos centraremos en el problema de clasificación.

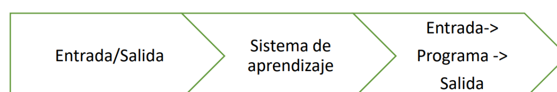


Figura 2.0.3: Dada la asociación, devuelve un programa aproximado de dicha asociación.

- **La medida de rendimiento:** Será una medida cuantitativa del rendimiento del algoritmo que dependerá de la tarea T que esté llevando a cabo el sistema. Para algunas tareas, se medirá la proporción de ejemplos para los cuales el modelo funciona adecuadamente (*precisión*), o bien funciona incorrectamente (*error rate*). Sin embargo,

generalmente, estaremos interesados en saber como funciona el algoritmo en datos no conocidos a priori. Es por ello, que evaluaremos estas medidas P utilizando un conjunto de datos de prueba, *test set*, que es independiente de los datos utilizados para entrenar el sistema. [[6], Capítulo 5]

2.1. Aprendizaje supervisado

La primera etapa será describir desde un punto de vista matemático un modelo formal de aprendizaje para un problema de clasificación general, pudiendo ser generalizado para los problemas de regresión. En toda la sección la fuente principal es [[14], Capítulos 2 y 3].

- Input: el sistema tiene acceso a la siguiente información:
  - Dominio de datos de entrada: un conjunto denotado por $\mathcal{X}$ donde sus elementos son aquellos que queremos clasificar. Generalmente, los puntos del dominio serán en realidad vectores que hacen referencia a las diferentes características medidas.
  - Conjunto de etiquetas: un conjunto denominado por $\mathcal{Y}$ donde sus elementos son los posibles valores que se pueden asignar a los elementos del dominio.
  - Conjunto de entrenamiento: denotado por $S = \{(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), ..., (\boldsymbol{x_n}, y_n)\}$, donde $\boldsymbol{x} \in \mathcal{X}$, $y \in \mathcal{Y}$. Es una secuencia de puntos del dominio etiquetados. Supondremos que sus elementos son independientes y tienen la misma distribución conjunta $\forall i = 1...n$ que $(\boldsymbol{x}, y)$, donde $\mathcal{L}(\boldsymbol{x}, y) = \mathcal{D}$. Podemos separar dicha distribución en dos contribuciones: la distribución $\mathcal{D}_x$ sobre los datos del dominio $\mathcal{X}$ y la distribución condicionada sobre las etiquetas de los puntos del dominio, $\mathcal{D}((\boldsymbol{x}, y)|\boldsymbol{x})$. Es importante tener en cuenta que el sistema de aprendizaje no tiene ningún tipo de conocimiento de dicha distribución de probabilidad $\mathcal{D}$.
- Datos de salida: el sistema genera a partir de los datos de entrenamiento una regla de predicción $h : \mathcal{X} \to \mathcal{Y}$, también conocida como ' función de predicción', 'función de clasificación' o 'función de hipótesis'; utilizada para predecir la etiqueta de nuevos puntos del dominio. La función h es un elemento de un espacio de posibles funciones $\mathcal{H}$, comúnmente denominado *espacio de hipótesis*.

En suma, el *aprendizaje supervisado* es el proceso de crear un *algoritmo* que dado un conjunto de entrenamiento $S = \{(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), ..., (\boldsymbol{x_n}, y_n)\}$, donde $\boldsymbol{x} \in \mathcal{X}$, $y \in \mathcal{Y}$, sea capaz de construir una regla $h : \mathcal{X} \to \mathcal{Y}$ que permita predecir la etiqueta de nuevas observaciones. Una vez especificadas la tarea y la experiencia, definimos el error del modelo como la probabilidad de que no prediga correctamente la etiqueta en un conjunto de muestras aleatorias generadas por la distribución $\mathcal{D}$. A este respecto, se introduce la **función de pérdida** como una medida de dicho error, es decir una función capaz de evaluar la calidad de la función $h : \mathcal{X} \to \mathcal{Y}$ para predecir la etiqueta y cuando el atributo es $\boldsymbol{x}$. La función de pérdida se expresa:

$$\begin{aligned} \ell : (\mathcal{X} \times \mathcal{Y}) \times \mathcal{H} &\to \mathrm{R}_+ \\ \ell \colon ((\boldsymbol{x}, y), h) &\mapsto \ell(\boldsymbol{x}, y, h) \end{aligned} \tag{2.1.1}$$

donde $\mathcal{H}$ es el espacio de hipótesis y $\mathcal{X} \times \mathcal{Y}$ son el dominio y el conjunto de etiquetas respectivamente. Por un lado, para el problema de regresión, donde generalmente $\mathcal{Y} = \mathrm{R}$, la función de pérdida a utilizar será la pérdida cuadrática:

$$\ell_{cuadratica}(\boldsymbol{x}, y, h) := (y - h(\boldsymbol{x}))^2 \tag{2.1.2}$$

Sin embargo, en el marco de la clasificación, binaria o multiclase, $\mathcal{Y} = \{0, 1\}$ o $\mathcal{Y} = \{0, 1, ..., k\}$ respectivamente, la función de pérdida a utilizar será la pérdida 0-1:

$$\ell_{0-1}(\boldsymbol{x}, y, h) := I(h(\boldsymbol{x}) \neq y). \tag{2.1.3}$$

Debido a que los elementos del conjunto $\mathcal{X} \times \mathcal{Y}$ son generados a partir de la distribución de probabilidad $\mathcal{D}$, la función de pérdida es una variable aleatoria, y por tanto podemos definir la **función de riesgo** como el valor esperado de la función de pérdida, es decir:

$$\mathcal{R}_{\mathcal{D}}(h) := E_{(\boldsymbol{x},y)\sim\mathcal{X}\times\mathcal{Y}}[\ell(\boldsymbol{x}, y, h)] = \int \ell(\boldsymbol{x}, y, h)\, d\mathcal{D}(\mathbf{x}, y). \tag{2.1.4}$$

El riesgo es, por tanto, una forma de cuantificar el error cometido al aproximar la etiqueta verdadera, y, por el valor calculado $\hat{y}$ de acuerdo con la función de predicción o hipótesis h. En el marco del problema de regresión se define como $\mathcal{R}_{\mathcal{D}}(h) = E_{\mathcal{D}}(y - h(\boldsymbol{x}))^2$ mientras que en la clasificación es $\mathcal{R}_{\mathcal{D}}(h) = \mathrm{P}_{\mathcal{D}}(h(\boldsymbol{x}) \neq y)$ y llamaremos *error asociado a una regla de clasificación*, $L_{\mathcal{D}}(h)$ a la probabilidad de que no prediga la etiqueta correcta para un valor generado aleatoriamente gracias a $\mathcal{D}$, que es con exactitud la anteriormente definida función de riesgo para la pérdida 0-1:

$$L_{\mathcal{D}}(h) = P_{(\boldsymbol{x},y)\sim\mathcal{D}}[h(\boldsymbol{x}) \neq y] = \mathcal{D}((\boldsymbol{x}, y) : h(\boldsymbol{x}) \neq y) = \mathcal{R}_{\mathcal{D}}(h). \tag{2.1.5}$$

Fijado $\mathcal{H}$ un espacio de hipótesis, el objetivo del algoritmo es encontrar un predictor $h : \mathcal{X} \to \mathcal{Y}$, $h \in \mathcal{H}$ que minimice el riesgo con respecto a la distribución $\mathcal{D}$, $\mathcal{R}_{\mathcal{D}}(h)$. En otras palabras, buscamos la *regla óptima* en $\mathcal{H}$, denotada por $\bar{h} \in \mathrm{argmin}_{h\in\mathcal{H}}\, \mathcal{R}_{\mathcal{D}}(h)$. Cabe destacar que para cualquier distribución $\mathcal{D}$ existe una regla óptima, en el sentido de mínimo riesgo, denominada *Regla de Bayes* $h_B := \mathrm{argmin}_{h\in\mathcal{H}^*}\, \mathcal{R}_{\mathcal{D}}(\boldsymbol{x}, y, h)$, $\mathcal{H}^* \subset \mathcal{H}$, concretamente, en el problema de regresión es definida como $h_B(\boldsymbol{x}_0) = E(y \mid \boldsymbol{x} = \boldsymbol{x}_0)$.

Definición 2.1.0.1. ***Clasificador de Bayes para el problema de clasificación*** *Sea $\mathcal{D}$ una distribución de probabilidad tal que $\mathcal{L}(\boldsymbol{x}, y) = \mathcal{D}$. Sea $\mathcal{H}$ un espacio de hipótesis, $\ell_{0-1}(\boldsymbol{x}, y, h) := I(h(\boldsymbol{x}) \neq y)$ la función de pérdida 0-1 y $\mathcal{R}_{\mathcal{D}}$ la función de riesgo o pérdida esperada $\mathcal{R}_{\mathcal{D}}(h) = P_{(\boldsymbol{x},y)\sim\mathcal{D}}[h(\boldsymbol{x}) \neq y]$. El clasificador de Bayes para el problema de clasificación es la función de clasificación $h_B : \mathcal{X} \to \mathcal{Y}$ definida como:*

$$h_B(\boldsymbol{x_0}) = \underset{j\in\{0,1,2,...,k\}}{\mathrm{argmax}}\; P(y = j|\boldsymbol{x} = \boldsymbol{x_0}) \tag{2.1.6}$$

En otras palabras, el clasificador de Bayes asigna a una observación $\boldsymbol{x_0}$ la clase j tal que el valor $P(y = j \mid \boldsymbol{x} = \boldsymbol{x_0})$ es máximo. En el caso particular de la clasificación binaria $\mathcal{Y} = \{0, 1\}$, como Regla de Bayes obtenemos $h_B(\boldsymbol{x}_0) = I(\eta(\boldsymbol{x}_0) \geq \frac{1}{2}), \eta(\boldsymbol{x}_0) = E(y \mid \boldsymbol{x} = \boldsymbol{x}_0)$.

Teorema 2.1.1. *En el contexto de la clasificación, el clasificador de Bayes 2.1.0.1 es óptimo en el sentido de mínimo riesgo. Es decir,*

$$\mathcal{R}_{\mathcal{D}}(h_B) = \underset{h\in\mathcal{H}}{\mathrm{argmin}}\, \mathcal{R}_{\mathcal{D}}(h). \tag{2.1.7}$$

Demostración. Demostraremos el enunciado para el problema de clasificación binaria. Sea $h : \mathcal{X} \to \{0, 1\}$ un clasificador cualquiera. Para esta regla de clasificación, el riesgo está definido como:

$$\mathcal{R}_{\mathcal{D}}(h) = E_{\mathcal{D}}[I(h(\boldsymbol{x}) \neq y)]. \tag{2.1.8}$$

dando lugar a las siguientes igualdades:

$$\begin{aligned} E_{\mathcal{D}}[I(h(\boldsymbol{x}) \neq y)] = E_{\boldsymbol{x}} E_{y|\boldsymbol{x}}[I(h(\boldsymbol{x}) \neq y)] \\ E_{\boldsymbol{x}}[\eta(\boldsymbol{x}) I(h(\boldsymbol{x}) = 0) + (1 - \eta(\boldsymbol{x})) I(h(\boldsymbol{x}) = 1)]. \end{aligned} \tag{2.1.9}$$

De acuerdo la anterior ecuación 2.1.9 se deduce que el valor $\mathcal{R}_{\mathcal{D}}(h)$ es mínimo cuando:

$$h(\boldsymbol{x}) = \begin{cases} 1 & \text{si } \eta(\boldsymbol{x}) \geq 1 - \eta(\boldsymbol{x}) \\ 0 & \text{otro caso} \end{cases}$$

que es exactamente la definición de clasificador de Bayes 2.1.0.1 teniendo en cuenta que $\eta(\boldsymbol{x_0}) = P(y = 1|x = \boldsymbol{x_0})$. [[4], Teorema 2.1] □

El teorema anterior (2.1.1) demuestra que el riesgo $\mathcal{R}_{\mathcal{D}}(h)$ es minimizado mediante un clasificador que asigna a cada observación la clase más probable, dadas las características medidas. Es decir, la tasa de error del clasificador de Bayes es la mínima tasa de error posible.

Definición 2.1.0.2. ***Riesgo del clasificador de Bayes*** *Sea $h_B : \mathcal{X} \to \{0, 1\}$ la regla de clasificación de Bayes para un problema concreto de clasificación binaria. Denominamos riesgo del clasificador de Bayes al valor $\mathcal{R}_{\mathcal{D}}(h_B)$. [[4], Nota Sección 2.1] Haciendo uso de las igualdades obtenidas en la demostración del teorema 2.1.1, en concreto 2.1.9, se obtiene:*

$$\begin{aligned} \mathcal{R}_{\mathcal{D}}(h_B) &= E_{\boldsymbol{x}}[\eta(\boldsymbol{x}) I(h_B(\boldsymbol{x}) = 0) + (1 - \eta(\boldsymbol{x})) I(h_B(\boldsymbol{x}) = 1)] \\ &= E_{\boldsymbol{x}}[\text{mín}(\eta(\boldsymbol{x}), 1 - \eta(\boldsymbol{x}))]. \end{aligned} \tag{2.1.10}$$

Sin embargo, aunque es útil conocer la existencia de un minimizador del riesgo, el clasificador de Bayes no es práctico, pues el sistema solo tiene acceso al conjunto de entrenamiento S y no a la distribución $\mathcal{D}$. Es por ello que trabajaremos con la **función de riesgo empírico**, que es la pérdida esperada sobre el conjunto de entrenamiento $S = \{(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), ..., (\boldsymbol{x_n}, y_n)\} \in (\mathcal{X} \times \mathcal{Y})^n$:

$$\mathcal{R}_{\mathcal{S}}(h) = \frac{1}{n} \sum_{i=1}^{n} \ell(\boldsymbol{x_i}, y_i, h). \tag{2.1.11}$$

y el objetivo radica en hallar un predictor que minimice el riesgo empírico $\mathcal{R}_{\mathcal{S}}(h)$, denotado por $\hat{h} := MRE_{\mathcal{H}}(S)$. Dicho problema se conoce como *problema de minimización del riesgo empírico MRE (en inglés Empirical Risk Minimization)* [[14], Capítulo 2]. El sobreajuste es el inconveniente de este método cuando la función de clasificación h ajusta demasiado bien los datos del conjunto de entrenamiento. Es decir, minimiza el error sobre los datos del conjunto de entrenamiento, pero, sin embargo, el error cometido sobre otras observaciones es elevado. Una forma de evitar este problema es reducir el espacio de hipótesis a un subconjunto $\mathcal{H}$. De acuerdo con esta restricción, se trata de encontrar la hipótesis h que reduzca el riesgo empírico, con $h \in \mathcal{H}$, siendo $\mathcal{H}$ una clase de hipótesis:

$$\hat{h} \in \underset{h \in \mathcal{H}}{\text{argmin}} \, \mathcal{R}_{\mathcal{S}}(h) \tag{2.1.12}$$

La elección del conjunto $\mathcal{H}$ se basa entonces en el conocimiento a priori del problema que se desea estudiar. Esta restricción se conoce como *restricción de sesgo inductivo.*

Se observa que $\mathcal{R}_{\mathcal{S}}(h)$ depende del conjunto de entrenamiento S, que se obtiene a partir de un proceso aleatorio, luego el riesgo es una variable aleatoria. Como resultado de la afamada *Ley de los Grandes Números*, podemos concluir que, para cada h previamente fijada, el riesgo empírico converge al riesgo cuando el tamaño del conjunto de entrenamiento tiende a infinito; coloquialmente, el promedio tiende al valor esperado.

$$\frac{1}{n}\sum_{i=1}^{n}\ell(\boldsymbol{x_i}, y_i, h) \xrightarrow{n\to\infty} \mathcal{R}_{\mathcal{D}}(h)$$

No es viable asumir que es posible determinar de manera exacta la función de distribución $\mathcal{D}$ a partir del conjunto de entrenamiento S ya que siempre nos vamos a encontrar con un conjunto de datos finito donde hay alguna probabilidad de que no sea muy representativo. En este sentido, introducimos un parámetro de exactitud ϵ. Consideraremos que el algoritmo es adecuado, *aproximadamente correcto*, si obtenemos $\mathcal{R}_{\mathcal{S}}(\hat{h}) \leq \epsilon$ y el objetivo es, por tanto, encontrar una cota superior del riesgo empírico que permita determinar si el algoritmo es 'adecuado'. [[14], Subsección 2.3.1]

Una forma de evitar problemas de sobreajuste es, como hemos mencionado, restringir la clase de hipótesis $\mathcal{H}$ en función de la información conocida del problema. Una pregunta natural es si existe algún algoritmo o sistema que clasifique correctamente en todas las tareas de aprendizaje. En otras palabras, estamos buscando un algoritmo que sin conocimiento a priori (al restringir la búsqueda de predictores a $\mathcal{H}$, implícitamente creemos que uno de los miembros de dicha clase es un modelo con bajo error para la tarea en cuestión), esté preparado para ser desafiado por cualquier tarea de aprendizaje. Es decir, la pregunta es si existe un algoritmo de aprendizaje A y un conjunto de entrenamiento de tamaño n, de tal forma que para cualquier distribución $\mathcal{D}$, si A recibe n muestras i.i.d.[1] de $\mathcal{D}$, con probabilidad alta el resultado es un predictor h con poco riesgo. El teorema *No-Free-Lunch* nos dirá que no existe dicho *'universal learner'*. Concretamente, el teorema establece que para las tareas de clasificación binarias, para cada algoritmo se puede encontrar una distribución para la que falla, aunque esa tarea puede ser aprendida con éxito por otro algoritmo [[14], Teorema 5.1]. De acuerdo con este resultado, no existe ningún algoritmo capaz de resolver con éxito cualquier problema. Por ende, es necesario incluir cierta información a priori sobre la distribución $\mathcal{D}$ en nuestro modelo de aprendizaje. El objetivo es ahora estudiar los beneficios de incorporar dicha información a priori en el modelo.

Sea $\mathcal{H}$ una clase de hipótesis, subconjunto estricto del espacio de funciones $\mathrm{H} = \{g | g : \mathcal{X} \to \mathcal{Y}\}$. El objetivo será controlar el **exceso de riesgo**

$$\mathcal{E}_{\mathcal{D}}(h) = \mathcal{R}_{\mathcal{D}}(h) - \mathcal{R}_{\mathcal{D}}(h_B), \tag{2.1.13}$$

de tal forma que nos permita seleccionar una clase de hipótesis $\mathcal{H}$ óptima para el modelo, encontrando un equilibrio entre modelos menos y más complejos que garanticen mejores

[1] independientes igualmente distribuidas

resultados. $\mathcal{R}_{\mathcal{D}}(h)$ denota el riesgo sobre la clase $\mathcal{H}$ y $\mathcal{R}_{\mathcal{D}}(h_B)$ simboliza el riesgo de la regla de Bayes sobre la clase $\mathcal{H}$. Como consecuencia inmediata del Teorema 2.1.1 y de la definición 2.1.0.2, para cualquier clasificador $h : \mathcal{X} \rightarrow \mathcal{Y}$, el exceso de riesgo $\mathcal{E}_{\mathcal{D}}(h)$ es un valor no negativo. Concretamente,

$$\begin{aligned}
&\mathcal{R}_{\mathcal{D}}(h) - \mathcal{R}_{\mathcal{D}}(h_B) = \\
&E_{\boldsymbol{x}}[\eta(\boldsymbol{x})I(h(\boldsymbol{x}) = 0) + (1 - \eta(\boldsymbol{x}))I(h(\boldsymbol{x}) = 1) - \\
&\eta(\boldsymbol{x})I(h_B(\boldsymbol{x}) = 0) - (1 - \eta(\boldsymbol{x}))I(h_B(\boldsymbol{x}) = 1)] = \\
&E_{\boldsymbol{x}}[|2 \cdot \eta(\boldsymbol{x}) - 1|I(h(\boldsymbol{x}) \neq h_B(\boldsymbol{x}))] = \\
&2 \cdot E_{\boldsymbol{x}}[|\eta(\boldsymbol{x}) - \frac{1}{2}|I(h(\boldsymbol{x}) \neq h_B(\boldsymbol{x}))].
\end{aligned} \tag{2.1.14}$$

Coloquialmente, la tasa de error del clasificador de Bayes es la mínima tasa de error posible.

Para encontrar la clase de hipótesis $\mathcal{H}$ óptima para el modelo, realizaremos una descomposición del exceso de riesgo para un predictor $\hat{h} = MRE_{\mathcal{H}}(S)$. Sumando y restando los términos $\mathcal{R}_{\mathcal{S}}(\hat{h})$ y $\mathcal{R}_{\mathcal{D}}(\bar{h})$ obtenemos:

$$\begin{aligned}
\mathcal{E}_{\mathcal{D}}(\hat{h}) &= \mathcal{R}_{\mathcal{D}}(\hat{h}) - \mathcal{R}_{\mathcal{D}}(h_B) \\
&= \mathcal{R}_{\mathcal{D}}(\hat{h}) - \mathcal{R}_{\mathcal{S}}(\hat{h}) + \mathcal{R}_{\mathcal{S}}(\hat{h}) - \mathcal{R}_{\mathcal{D}}(\bar{h}) + \mathcal{R}_{\mathcal{D}}(\bar{h}) - \mathcal{R}_{\mathcal{D}}(h_B).
\end{aligned} \tag{2.1.15}$$

Dado que $\hat{h}$ es el minimizador de riesgo empírico, se deduce $\mathcal{R}_{\mathcal{S}}(\hat{h}) \leq \mathcal{R}_{\mathcal{S}}(\bar{h})$. Por ende,

$$\begin{aligned}
\mathcal{R}_{\mathcal{D}}(\hat{h}) - \mathcal{R}_{\mathcal{S}}(\hat{h}) + \mathcal{R}_{\mathcal{S}}(\hat{h}) - \mathcal{R}_{\mathcal{D}}(\bar{h}) + \mathcal{R}_{\mathcal{D}}(\bar{h}) - \mathcal{R}_{\mathcal{D}}(h_B) \leq \\
(\mathcal{R}_{\mathcal{D}}(\hat{h}) - \mathcal{R}_{\mathcal{S}}(\hat{h})) + (\mathcal{R}_{\mathcal{S}}(\bar{h}) - \mathcal{R}_{\mathcal{D}}(\bar{h})) + (\mathcal{R}_{\mathcal{D}}(\bar{h}) - \mathcal{R}_{\mathcal{D}}(h_B)).
\end{aligned} \tag{2.1.16}$$

Acotando superiormente los dos primeros paréntesis por $\sup_{h\in\mathcal{H}}|\mathcal{R}_{\mathcal{S}}(h) - \mathcal{R}_{\mathcal{D}}(h)|$, obtenemos la descomposición:

$$\begin{aligned}
\mathcal{E}_{\mathcal{D}}(\hat{h}) &= \mathcal{R}_{\mathcal{D}}(\hat{h}) - \mathcal{R}_{\mathcal{D}}(h_B) \\
&\leq 2 \sup_{h\in\mathcal{H}}|\mathcal{R}_{\mathcal{S}}(h) - \mathcal{R}_{\mathcal{D}}(h)| + (\mathcal{R}_{\mathcal{D}}(\bar{h}) - \mathcal{R}_{\mathcal{D}}(h_B))
\end{aligned} \tag{2.1.17}$$

Al término $\sup_{h\in\mathcal{H}}|\mathcal{R}_{\mathcal{S}}(h) - \mathcal{R}_{\mathcal{D}}(h)|$ se le denominará *error de estimación* y $\mathcal{R}_{\mathcal{D}}(\bar{h}) - \mathcal{R}_{\mathcal{D}}(h_B)$ será conocido como *error de aproximación*.

Si nuestro problema es la clasificación binaria, de la ecuación (2.1.17) se puede sustituir siempre el riesgo $\mathcal{R}$ por el error L, obteniendo que el exceso de error se descompone como:

$$\begin{aligned}
\mathcal{E}_{\mathcal{D}}(\hat{h}) = 2 \cdot \sup_{h\in\mathcal{H}}|L_S(h) - L_D(h)| + \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) - \min L_{\mathcal{D}}(h); \\
error_{estimacion} = \sup_{h\in\mathcal{H}}|L_S(h) - L_D(h)| \\
error_{aproximacion} = \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) - \min L_{\mathcal{D}}(h)
\end{aligned} \tag{2.1.18}$$

- Error de aproximación: refleja la calidad del conocimiento a priori, medido con el riesgo mínimo dentro de los predictores de la clase de hipótesis $\mathcal{H}$. Es decir, mide el sesgo del algoritmo al elegir previamente la clase de hipótesis $\mathcal{H}$, solución utilizada para garantizar que no se produce sobreajuste en el caso de tomar como clase una demasiado grande. El error de aproximación no depende del tamaño de la muestra y está determinado por la clase de hipótesis elegida. Es sencillo verificar que si aumentamos el tamaño de la clase de hipótesis, puede disminuir el error de aproximación.

- Error de estimación: aparece debido a que el riesgo empírico es solamente una estimación del riesgo verdadero, por tanto, el predictor que minimiza el riesgo empírico es solo un estimador del predictor que minimiza el riesgo verdadero. Es dependiente del tamaño o complejidad de la clase $\mathcal{H}$.

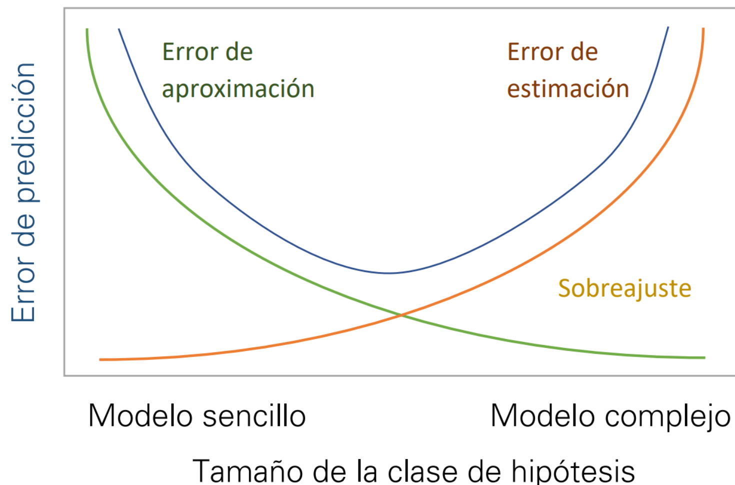


Figura 2.1.1: Gráfico explicativo de lo que sucede con los errores definidos al variar la complejidad de la clase de hipótesis

Si elegimos una clase compleja el sesgo disminuye, pero puede aumentar el riesgo de sobreajuste (disminuye el error de aproximación pero aumenta el de estimación), o si bien escogemos una clase menos compleja, disminuye el sobreajuste pero puede aumentar el sesgo (aumenta el error de aproximación pero disminuye el de estimación) y en ambos casos el exceso de riesgo es alto, ver figura 2.1.1. Por tanto, para controlar el exceso de riesgo total $\mathcal{E}_{\mathcal{D}}(\hat{h})$, gracias a la descomposición previa, debemos encontrar un equilibrio entre el error de estimación y el error de aproximación, problema conocido como *equilibrio entre sesgo y varianza*. Estamos considerando el tamaño de $\mathcal{H}$ como medida de complejidad de esta. La ventaja de esta descomposición es que el error de estimación es el único término aleatorio.

Teniendo en cuenta que un algoritmo de aprendizaje automático está determinado eligiendo: una familia de modelos $\mathcal{H}$ (obtenida dependiendo de las especificaciones de los datos), la función de coste que será el riesgo $\mathcal{R}_{\mathcal{D}}(h)$ y el procedimiento de optimización, entonces para encontrar un predictor que minimice el riesgo $\mathcal{R}_{\mathcal{D}}(h)$, es suficiente con especificar el algoritmo de optimización [6]. Un posible procedimiento a nivel teórico es minimizar el riesgo empírico, esto es, en lugar de optimizar el riesgo directamente, optimizamos el riesgo empírico $\hat{h} \in \operatorname{argmin}_{h\in\mathcal{H}} \mathcal{R}_{\mathcal{S}}(h)$ (MRE) y esperamos que el riesgo disminuya significativamente también. Como hemos demostrado, para un predictor minimizador de riesgo empírico $\hat{h}$, el exceso de riesgo está controlado. Estudiaremos ahora un algoritmo de optimización para el cálculo efectivo del MRE, restringiéndonos a los problemas de aprendizaje convexos (ver sección Problemas de aprendizaje convexos). Representaremos con el símbolo $\boldsymbol{w}$ una hipótesis perteneciente a una clase de hipótesis convexa $\mathcal{H}$.

2.2. Regularización

La minimización de riesgo empírico tiene como inconveniente la inestabilidad, es decir, es un método propenso a problemas de sobreajuste. Anteriormente se ha estudiado como posible solución la reducción del espacio de hipótesis a un subconjunto del mismo. En esta sección se estudiará un procedimiento que puede ser interpretado como estabilizador. La **regularización** es una regla de optimización en la que minimizaremos conjuntamente el riesgo empírico y una función de regularización, que formalmente es una aplicación $R : R^n \to R$, [[14], Capítulo 13]. La hipótesis $\boldsymbol{w}$ obtenida mediante la regla de optimización regularizada satisface:

$$\boldsymbol{w} \in argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + R(\boldsymbol{w})). \tag{2.2.1}$$

Intuitivamente, la 'complejidad' de una hipótesis es medida por el valor de la función de regularización y el algoritmo equilibra entre bajo riesgo empírico y complejidad de la hipótesis. Existen muchas posibles funciones de regularización, escogidas dependiendo del conocimiento a priori sobre el problema. Una de las funciones más simples es $R(\boldsymbol{w}) = \lambda||\boldsymbol{w}||^2$, con $\lambda > 0$ y la norma l_2. El problema de optimización se reduce a la conocida como regularización Tikhonov:

$$A(S) := argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2). \tag{2.2.2}$$

Sea A un algoritmo de aprendizaje, S el conjunto de entrenamiento con n observaciones y $A(S)$ la salida del algoritmo. Se dice que el algoritmo *sobreajusta* si la diferencia entre el verdadero riesgo $\mathcal{R}_{\mathcal{D}}(A(S))$ y el riesgo empírico de la salida, $\mathcal{R}_{\mathcal{S}}(A(S))$ es un valor grande. Estudiaremos en el teorema 2.2.1 la esperanza de dicha diferencia con respecto al conjunto de entrenamiento $E_S[\mathcal{R}_{\mathcal{D}}(A(S)) - \mathcal{R}_{\mathcal{S}}(A(S))]$, obteniendo que un algoritmo sobreajusta cuando varía mucho la predicción para el elemento z_i en función de si pertenece al conjunto de entrenamiento.

Teorema 2.2.1. *Sea $\mathcal{D}$ una distribución. Sea S un conjunto de entrenamiento de tamaño n y $z' = (\boldsymbol{x}', y')$ un elemento adicional del conjunto de datos de entrada independiente a S. Sea $U(n)$ la distribución uniforme sobre $[n]$. Se denotará como $S^{(i)}$ al conjunto obtenido reemplazando el i-ésimo elemento de S por z'. Para cualquier algoritmo de aprendizaje se verifica la igualdad:*

$$\begin{aligned} &E_{S\sim D^n}[\mathcal{R}_{\mathcal{D}}((A(S)) - \mathcal{R}_{\mathcal{S}}(A(S))] = \\ &\qquad = E_{(S,z')\sim D^{n+1}, i\sim U(n)}[\ell(z_i, A(S^{(i)})) - \ell(z_i, A(S))] \end{aligned} \tag{2.2.3}$$

Demostración. Teniendo en cuenta que S y z' son muestras independientes e igualmente distribuidas provenientes de $\mathcal{D}$, para todo i se verifica:

$$E_S[\mathcal{R}_{\mathcal{D}}(A(S))] = E_{S,z'}[\ell(z', A(S))] = E_{S,z'}[\ell(z_i, A(S^{(i)}))].$$

Por otro lado,

$$E_S[\mathcal{R}_{\mathcal{D}}(A(S))] = E_{S,i}[\ell(z_i, A(S))].$$

Combinando ambos resultados se concluye la prueba, [[14], Teorema 13.2]. □

De forma intuitiva podemos decir que un algoritmo de aprendizaje es *estable* si un 'ligero' cambio en los datos de entrada no produce un cambio 'grande' en la salida del algoritmo. Sea

S un conjunto de entrenamiento y $z' = (\boldsymbol{x}', y')$ un elemento adicional del conjunto de datos de entrada. Se denotará como $S^{(i)}$ al conjunto obtenido reemplazando el i-ésimo elemento de S por z'. Un 'ligero' cambio en los datos de entrada significa que entrenamos el algoritmo A con $S^{(i)}$ en vez de S, es decir, reemplazamos solo un elemento del conjunto de entrenamiento. El efecto de este 'ligero' cambio es medido comparando la pérdida de la hipótesis $A(S)$ en z_i y la pérdida de la hipótesis $A(S^{(i)})$ en z_i, que es exactamente la parte derecha de la ecuación 2.2.3. Se obtiene como consecuencia que las reglas de aprendizaje estables no sobreajustan. [[14], Sección 13.2]

Suponiendo que la función de pérdida es convexa y es Lipschitz o suave, la minimización regularizada Tikhonov $A(S) := argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2)$ produce un algoritmo estable, gracias a que la función objetivo $\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2$ es fuertemente convexa, como se define a continuación [[14], Definición 13.4].

Definición 2.2.0.1. ***Función fuertemente convexa*** *Una función f es $\lambda-$fuertemente convexa si para todo $\boldsymbol{w}, \boldsymbol{u}$ y $\alpha \in (0,1)$ se tiene:*

$$f(\alpha\boldsymbol{w} + (1-\alpha)\boldsymbol{u}) \leq \alpha f(\boldsymbol{w}) + (1-\alpha)f(\boldsymbol{u}) - \frac{\lambda}{2}\alpha(1-\alpha)||\boldsymbol{w}-\boldsymbol{u}||^2.$$

Claramente, toda función convexa es 0-fuertemente convexa.

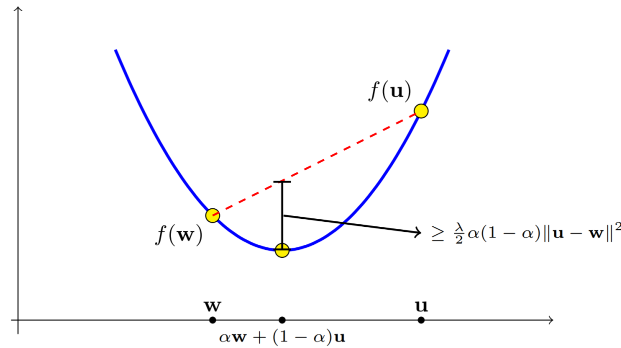


Figura 2.2.1: Función fuertemente convexa [14]

Lema 2.2.2. *1. La función $f(\boldsymbol{w}) = \lambda||\boldsymbol{w}||^2$ es 2λ- fuertemente convexa.*

2. Si f es $\lambda-$fuertemente convexa y g es convexa, entonces $f+g$ es $\lambda-$fuertemente convexa.

3. Si f es $\lambda-$fuertemente convexa y $\boldsymbol{u}$ minimiza f, entonces para cualquier $\boldsymbol{w}$, se tiene $f(\boldsymbol{w}) - f(\boldsymbol{u}) \geq \frac{\lambda}{2}||\boldsymbol{w}-\boldsymbol{u}||^2$

Demostración. (1) y (2) son consecuencias directas de la definición 2.2.0.1. Para probar (3), dividiendo la ecuación de la definición 2.2.0.1 por α obteniendo:

$$\frac{f(\boldsymbol{u} + \alpha(\boldsymbol{w}-\boldsymbol{u})) - f(\boldsymbol{u})}{\alpha} \leq f(\boldsymbol{w}) - f(\boldsymbol{u}) - \frac{\lambda}{2}(1-\alpha)||\boldsymbol{w}-\boldsymbol{u}||^2.$$

Tomando límite cuando $\alpha \to 0$ se obtiene que la parte derecha converge a $f(\boldsymbol{w}) - f(\boldsymbol{u}) - \frac{\lambda}{2}||\boldsymbol{w}-\boldsymbol{u}||^2$; mientras que la parte izquierda es la derivada de la función $g(\alpha) = f(\boldsymbol{u}+\alpha(\boldsymbol{w}-\boldsymbol{u}))$ en $\alpha = 0$. Bajo la hipótesis que $\boldsymbol{u}$ minimiza f, se tiene que $\alpha = 0$ minimiza g y por tanto la parte derecha de la ecuación anterior es cero en el límite, completando la prueba [[14], Lema 13.5]. □

Probaremos ahora que la minimización regularizada Tikhonov $A(S) := argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2$ produce un algoritmo estable [[14], Sección 13.3]. Sea S un conjunto de entrenamiento, $z' = (\boldsymbol{x}', y')$ un elemento adicional del conjunto de datos de entrada y $S^{(i)}$ el conjunto obtenido reemplazando el i-ésimo elemento de S por z'. Aplicando (2) del lema 2.2.2 obtenemos que la función objetivo $f_S(\boldsymbol{w}) = \mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2$ es 2λ-fuertemente convexa. Haciendo uso de (3), para cualquier $\boldsymbol{v}$, se tiene $f_S(\boldsymbol{v}) - f_S(A(S)) \geq \lambda||\boldsymbol{v} - A(S)||^2$. Por otro lado, para cualquier $\boldsymbol{v}$ y $\boldsymbol{u}$ y todo i, se tiene que:

$$\begin{aligned} f_S(\boldsymbol{v}) - f_S(\boldsymbol{u}) &= \mathcal{R}_{\mathcal{S}}(\boldsymbol{v}) + \lambda||\boldsymbol{v}||^2 - (\mathcal{R}_{\mathcal{S}}(\boldsymbol{u}) + \lambda||\boldsymbol{u}||^2) \\ &= \mathcal{R}_{S^{(i)}}(\boldsymbol{v}) + \lambda||\boldsymbol{v}||^2 - (\mathcal{R}_{S^{(i)}}(\boldsymbol{u}) + \lambda||\boldsymbol{u}||^2) \\ &+ \frac{\ell((\boldsymbol{x}_i, y_i), \boldsymbol{v})}{n} + \frac{\ell((\boldsymbol{x}', y'), \boldsymbol{u})}{n}. \end{aligned} \tag{2.2.4}$$

En particular, tomando como $\boldsymbol{v} = A(S^{(i)})$, $\boldsymbol{u} = A(S)$ y teniendo en cuenta el hecho que $\boldsymbol{v}$ minimiza $R_{S^{(i)}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2$, se obtiene:

$$\begin{aligned} f_S(A(S^{(i)})) - f_S(A(S)) \leq & \frac{\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S))}{n} \\ &+ \frac{\ell((\boldsymbol{x}', y'), A(S)) - \ell((\boldsymbol{x}', y'), A(S^{(i)}))}{n}. \end{aligned} \tag{2.2.5}$$

Teniendo en cuenta $f_S(\boldsymbol{v}) - f_S(A(S)) \geq \lambda||\boldsymbol{v} - A(S)||^2$:

$$\begin{aligned} \lambda||A(S^{(i)}) - A(S)||^2 \leq & \frac{\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S))}{n} \\ &+ \frac{\ell((\boldsymbol{x}', y'), A(S)) - \ell((\boldsymbol{x}', y'), A(S^{(i)}))}{n}. \end{aligned} \tag{2.2.6}$$

El análisis de estabilidad continúa dependiendo si la función de pérdida es Lipschitz [[14], Corolario 13.6] o suave [[14], Corolario 13.7], concluyendo en ambos casos que es estable y por tanto no sobreajusta.

Lema 2.2.3. *Sea la función de pérdida convexa y ρ-Lipschitz. Entonces la minimización regularizada Tikhonov $A(S) := argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2)$ produce un algoritmo tal que $E_{S\sim D^n}[\mathcal{R}_{\mathcal{D}}((A(S)) - \mathcal{R}_{\mathcal{S}}(A(S))] \leq \frac{2\rho^2}{\lambda n}$.*

Demostración. Si la función de pérdida es ρ-Lipschitz, por definición:

$$\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S)) \leq \rho||A(S^{(i)}) - A(S)|| \tag{2.2.7}$$

Y análogamente, $\ell((\boldsymbol{x}', y'), A(S)) - \ell((\boldsymbol{x}', y'), A(S^{(i)})) \leq \rho||A(S^{(i)}) - A(S)||$. Teniendo en cuenta estas desigualdades para la ecuación 2.2.6, se obtiene

$$\lambda||A(S^{(i)}) - A(S)||^2 \leq \frac{2\rho||A(S^{(i)}) - A(S)||}{n},$$

que implica:

$$||A(S^{(i)}) - A(S)|| \leq \frac{2\rho}{\lambda n}.$$

Volviendo a la ecuación 2.2.7 se concluye

$$\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S)) \leq \frac{2\rho^2}{\lambda n}.$$

La prueba concluye gracias a la igualdad 2.2.3 del teorema 2.2.1. □

Lema 2.2.4. *Sea la función de pérdida no negativa convexa y β-suave. Entonces la minimización regularizada Tikhonov $A(S) := argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2)$ con $\lambda \geq \frac{2\beta}{n}$ produce un algoritmo tal que $E_{S\sim D^n}[\mathcal{R}_{\mathcal{D}}((A(S)) - \mathcal{R}_{\mathcal{S}}(A(S))] \leq \frac{48\beta}{\lambda n}E[R_S(A(S))]$. Es decir, es estable.*

Demostración. Si f es una función no negativa y β-suave entonces está acotada:

$$||\nabla f(\boldsymbol{w})||^2 \leq 2\beta f(\boldsymbol{w}). \tag{2.2.8}$$

Gracias a la suposición de que f es una función suave, se tiene:

$$\begin{aligned}\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S)) \leq \langle \nabla\ell((\boldsymbol{x}_i, y_i), A(S))\ ,\ A(S^{(i)}) - A(S)\rangle +\\ \frac{\beta}{2}||A(S^{(i)}) - A(S)||^2\end{aligned} \tag{2.2.9}$$

Aplicando la desigualdad de Cauchy-Schwartz se obtiene:

$$\begin{aligned}&\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S)) \leq\\ &\leq ||\nabla\ell((\boldsymbol{x}_i, y_i), A(S))||||A(S^{(i)}) - A(S)|| + \frac{\beta}{2}||A(S^{(i)}) - A(S)||^2 \leq\\ &\leq \sqrt{2\beta\ell((\boldsymbol{x}_i, y_i), A(S))}||A(S^{(i)}) - A(S)|| + \frac{\beta}{2}||A(S^{(i)}) - A(S)||^2.\end{aligned} \tag{2.2.10}$$

Análogamente:

$$\ell((\boldsymbol{x}', y'), A(S^{(i)})) - \ell((\boldsymbol{x}', y'), A(S)) \leq \sqrt{2\beta\ell((\boldsymbol{x}', y'), A(S^{(i)}))}||A(S^{(i)}) - A(S)|| + \frac{\beta}{2}||A(S^{(i)}) - A(S)||^2.$$

Aplicando los resultados obtenidos a la inecuación 2.2.6 se obtiene:

$$||A(S^{(i)}) - A(S)|| \leq \frac{\sqrt{2\beta}}{(\lambda n - \beta)}(\sqrt{\ell((\boldsymbol{x}_i, y_i), A(S))} + \sqrt{\ell((\boldsymbol{x}', y'), A(S^{(i)}))}).$$

Haciendo uso de la hipótesis $\beta \leq \frac{\lambda n}{2}$, se sigue:

$$||A(S^{(i)}) - A(S)|| \leq \frac{\sqrt{8\beta}}{(\lambda n)}(\sqrt{\ell((\boldsymbol{x}_i, y_i), A(S))} + \sqrt{\ell((\boldsymbol{x}', y'), A(S^{(i)}))}).$$

Juntando el resultado anterior con la ecuación 2.2.10 se sigue:

$$\begin{aligned}&\ell((\boldsymbol{x}_i, y_i), A(S^{(i)})) - \ell((\boldsymbol{x}_i, y_i), A(S)) \leq\\ &\leq \sqrt{2\beta\ell((\boldsymbol{x}_i, y_i), A(S))}||A(S^{(i)}) - A(S)|| + \frac{\beta}{2}||A(S^{(i)}) - A(S)||^2\\ &\leq (\frac{4\beta}{\lambda n} + \frac{8\beta^2}{(\lambda n)^2})(\sqrt{\ell((\boldsymbol{x}_i, y_i), A(S))} + \sqrt{\ell((\boldsymbol{x}', y'), A(S^{(i)}))})^2\\ &\leq \frac{8\beta}{\lambda n}(\sqrt{\ell((\boldsymbol{x}_i, y_i), A(S))} + \sqrt{\ell((\boldsymbol{x}', y'), A(S^{(i)}))})^2\\ &\leq \frac{24\beta}{\lambda n}(\ell((\boldsymbol{x}_i, y_i), A(S)) + \ell((\boldsymbol{x}', y'), A(S^{(i)}))),\end{aligned} \tag{2.2.11}$$

donde en la última desigualdad se ha hecho uso de $(a+b)^2 \leq 3(a^2+b^2)$. Tomando esperanzas en ambas partes de la inecuación 2.2.10 y recordando el teorema 2.2.1 finaliza la prueba del lema, teniendo en cuenta que $E[\ell((\boldsymbol{x}_i, y_i), A(S))] = E[\ell((\boldsymbol{x}', y'), A(S^{(i)}))] = E[R_S(A(S))]$. □

Nótese que si para todo z se tiene que $\ell(z, \mathbf{0}) \leq C$, para un escalar $C \geq 0$, entonces para todo conjunto de entrenamiento S, se tiene:

$$\mathcal{R}_S(A(S)) \leq \mathcal{R}_S(A(S)) + \lambda||A(S)||2 \leq \mathcal{R}_S(\mathbf{0}) + \lambda||\mathbf{0}||2 = \mathcal{R}_S(\mathbf{0}) \leq C.$$

Entonces el lema 2.2.4 implica

$$E_{S\sim D^n}[\mathcal{R}_{\mathcal{D}}((A(S)) - \mathcal{R}_{\mathcal{S}}(A(S))] \leq \frac{48\beta C}{\lambda n}.$$

En conclusión, suponiendo que la función de pérdida es convexa y es Lipschitz o suave, la minimización regularizada Tikhonov $A(S) := argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \lambda||\boldsymbol{w}||^2)$ produce un algoritmo estable.

En suma, es deseable que un algoritmo de aprendizaje tenga un valor pequeño para el riesgo empírico, además de ser estable. El riesgo esperado de un algoritmo de aprendizaje puede reescribirse como:

$$E_S[\mathcal{R}_{\mathcal{D}}(A(S))] = E_S[\mathcal{R}_{\mathcal{S}}(A(S))] + E_S[\mathcal{R}_{\mathcal{D}}(A(S)) - \mathcal{R}_{\mathcal{S}}(A(S))]. \tag{2.2.12}$$

El primer término refleja la calidad del algoritmo para ajustarse al conjunto de entrenamiento, mientras que el segundo es equivalente a la estabilidad del propio algoritmo. Dado que nuestro objetivo es minimizar el riesgo del algoritmo, es necesario que la suma de ambos términos sea pequeña. Anteriormente para la regularización Tikhonov se ha acotado el término referente a la estabilidad y se ha demostrado que éste disminuye a medida que aumenta el parámetro de regularización λ. Sin embargo, el riesgo empírico aumenta con λ. Es decir, el parámetro λ equilibra entre estabilidad y minimizar el riesgo empírico. Por ende, nos enfrentamos a un equilibrio entre el ajuste y el sobreajuste. [[14], Sección 13.4]

2.3. Algoritmos de optimización de descenso del gradiente

El problema de optimización se refiere a minimizar cierta *función objetivo o función de coste* $f(x)$ alterando x. La **técnica de Descenso del Gradiente** (Cauchy 1847) [[14], Sección 14.1] es un procedimiento iterativo de optimización numérica que consiste en reducir $f(x)$ en pequeñas etapas moviendo x en sentido opuesto al signo de las derivadas, ver figura 2.3.1. La complejidad de los problemas hará que nos conformemos con valores de f suficientemente pequeños, pero no necesariamente con el más bajo. Normalmente la función objetivo tiene múltiples entradas $f : R^n \to R$ y por tanto se hará uso de los conceptos: derivadas parciales, gradiente $\nabla_{\boldsymbol{w}} f(\boldsymbol{w})$ y derivada direccional en la dirección $\boldsymbol{u}$ (vector unitario), que

es la pendiente de la función f en la dirección $\boldsymbol{u}$. Es decir, es la derivada de la función $f(\boldsymbol{w}+\lambda\boldsymbol{u})$ con respecto a λ evaluada en $\lambda = 0$. Aplicando el Desarollo en serie de Taylor (notación Landau) se obtienen las siguientes igualdades:

$$\begin{aligned} G(\lambda) &:= f(\boldsymbol{w}+\lambda\boldsymbol{u}) - f(\boldsymbol{w}) = \lambda\nabla_{\boldsymbol{w}} f(\boldsymbol{w})\cdot\boldsymbol{u} + o(\lambda) \\ G(\lambda) &= G'(0)\cdot\lambda + o(\lambda);\; G'(0) = \nabla_{\boldsymbol{w}} f(\boldsymbol{w})\cdot\boldsymbol{u}. \end{aligned} \tag{2.3.1}$$

El procedimiento inicia escogiendo un valor cualquiera, por ejemplo $\boldsymbol{w}^{(1)} = 0$. Para minimizar f se trata de encontrar la dirección $\boldsymbol{u}$ en la que f decrece más rápido. Es decir, se trata de resolver

$$min_{\boldsymbol{u},\boldsymbol{u}^T\boldsymbol{u}=1}(\boldsymbol{u}^T\nabla_{\boldsymbol{w}} f(\boldsymbol{w})).$$

Aplicando la desigualdad de Cauchy-Bunyakovsky-Schwarz, se obtiene:

$$min_{\boldsymbol{u},\boldsymbol{u}^T\boldsymbol{u}=1}(\boldsymbol{u}^T\nabla_{\boldsymbol{w}} f(\boldsymbol{w})) \geq -||\nabla_{\boldsymbol{w}} f(\boldsymbol{w}))|| \tag{2.3.2}$$

dándose la igualdad si y solo si

$$\boldsymbol{u} = \frac{-\nabla_{\boldsymbol{w}} f(\boldsymbol{w})}{||\nabla_{\boldsymbol{w}} f(\boldsymbol{w}))||}.$$

Por ende, la función f decrece cuando nos movemos en la dirección opuesta al gradiente y el nuevo punto propuesto por este método es

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta\nabla_{\boldsymbol{w}} f(\boldsymbol{w}^{(t)})$$

donde $\eta > 0$ es la *tasa de aprendizaje*. En general, después de T iteraciones, la salida de esta técnica es el vector $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{w}^{(t)}$, o bien, el último vector obtenido $\boldsymbol{w}^{(T)}$ o $\mathrm{argmin}_{t\in[T]} f(\boldsymbol{w}^{(t)})$. El método se aplica hasta que todos los elementos del gradiente son cero, o suficientemente cercanos a cero.

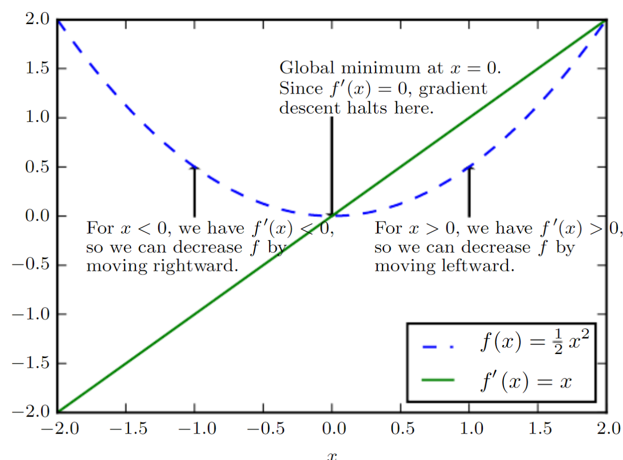


Figura 2.3.1: Ejemplo de la técnica de descenso del gradiente donde se ilustra como el procedimiento usa las derivadas para buscar el mínimo [6]

En este punto hemos de notar que la condición necesaria para que el procedimiento de optimización de Descenso del Gradiente funcione es que la función objetivo sea diferenciable, para poder así calcular el gradiente y las derivadas parciales y direccionales de acuerdo con

las ecuaciones 2.3.1. Asimismo, si la función f es convexa hay garantías estadísticas de que el algoritmo converja.

Una vez definido el procedimiento de Descenso del Gradiente, es conveniente analizar la convergencia del algoritmo [[14], Capítulo 14]. Estudiaremos únicamente el caso de funciones convexas y Lipschitz. Sea $\boldsymbol{w}^*$ cualquier vector y sea B la cota superior de $\|\boldsymbol{w}^*\|$. Es conveniente pensar que $\boldsymbol{w}^*$ es el elemento que minimiza $f(\boldsymbol{w})$. Deseamos obtener una cota superior de la diferencia $f(\bar{\boldsymbol{w}}) - f(\boldsymbol{w}^*)$ donde $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{w}^{(t)}$ es la salida del algoritmo. Aplicando la desigualdad de Jensen se obtiene:

$$\begin{aligned} f(\bar{\boldsymbol{w}}) - f(\boldsymbol{w}^*) &= f(\frac{1}{T}\sum_{t=1}^{T}\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*) \\ &\leq \frac{1}{T}\sum_{t=1}^{T} f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*) \\ &= \frac{1}{T}\sum_{t=1}^{T}(f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*)). \end{aligned} \tag{2.3.3}$$

Para cada t, gracias a la propiedad de convexidad de f se tiene que:

$$f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*) \leq \langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \nabla f(\boldsymbol{w}^{(t)})\ \rangle. \tag{2.3.4}$$

Combinado con lo obtenido anteriormente:

$$f(\bar{\boldsymbol{w}}) - f(\boldsymbol{w}^*) \leq \frac{1}{T}\sum_{t=1}^{T}\langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \nabla f(\boldsymbol{w}^{(t)})\ \rangle. \tag{2.3.5}$$

Para acotar superiormente la parte de la derecha de la desigualdad anterior hacemos uso del siguiente lema [[14] Lema 14.1]:

Lema 2.3.1. *Sea $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_T$ una secuencia finita de vectores. Cualquier algoritmo con vector inicial $\boldsymbol{w}^{(1)} = 0$ y regla $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta\boldsymbol{v}_t$ verifica:*

$$\sum_{t=1}^{T}\langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t\ \rangle \leq \frac{\|\boldsymbol{w}^*\|^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T}\|\boldsymbol{v}_t\|^2. \tag{2.3.6}$$

En particular, para cada $B, \rho > 0$, si para todo t se tiene que $\|\boldsymbol{v}_t\| \leq \rho$ y fijamos $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, entonces para todo $\boldsymbol{w}^*$ con $\|\boldsymbol{w}^*\| \leq B$ se verifica:

$$\frac{1}{T}\sum_{t=1}^{T}\langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t\ \rangle \leq \frac{B\rho}{\sqrt{T}}. \tag{2.3.7}$$

Demostración. Las siguientes igualdades se obtienen mediante manipulaciones algebraicas:

$$\begin{aligned} \langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t\ \rangle &= \frac{1}{\eta}\langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \eta\boldsymbol{v}_t\ \rangle \\ &= \frac{1}{2\eta}(-\|\boldsymbol{w}^{(t)} - \boldsymbol{w}^* - \eta\boldsymbol{v}_t\|^2 + \|\boldsymbol{w}^{(t)} - \boldsymbol{w}^*\|^2 + \eta^2\|\boldsymbol{v}_t\|^2) \\ &= \frac{1}{2\eta}(-\|\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^*\|^2 + \|\boldsymbol{w}^{(t)} - \boldsymbol{w}^*\|^2) + \frac{\eta}{2}\|\boldsymbol{v}_t\|^2 \end{aligned} \tag{2.3.8}$$

donde la última igualdad se deduce de la regla de optimización. Sumando las dos partes de la igualdad con respecto a t se obtiene:

$$\sum_{t=1}^{T}\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle = \frac{1}{2\eta}\sum_{t=1}^{T}(-\|\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^*\|^2 + \|\boldsymbol{w}^{(t)} - \boldsymbol{w}^*\|^2) + \frac{\eta}{2}\sum_{t=1}^{T}\|\boldsymbol{v}_t\|^2. \tag{2.3.9}$$

Notemos que la primera suma de la parte derecha de la ecuación 2.3.9 es una suma telescópica que converge a $\|\boldsymbol{w}^{(1)} - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}^{(T+1)} - \boldsymbol{w}^*\|^2$. Por tanto:

$$\begin{aligned}\sum_{t=1}^{T}\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle &= \frac{1}{2\eta}(\|\boldsymbol{w}^{(1)} - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}^{(T+1)} - \boldsymbol{w}^*\|^2) + \frac{\eta}{2}\sum_{t=1}^{T}\|\boldsymbol{v}_t\|^2 \\ &\leq \frac{1}{2\eta}\|\boldsymbol{w}^{(1)} - \boldsymbol{w}^*\|^2 + \frac{\eta}{2}\sum_{t=1}^{T}\|\boldsymbol{v}_t\|^2 \\ &= \frac{1}{2\eta}\|\boldsymbol{w}^*\|^2 + \frac{\eta}{2}\sum_{t=1}^{T}\|\boldsymbol{v}_t\|^2.\end{aligned} \tag{2.3.10}$$

donde la última igualdad es porque $\boldsymbol{w}^{(1)} = 0$. Quedaría probado la primera parte del lema, es decir, la ecuación 2.3.6. La segunda parte se deduce al saber que $\|\boldsymbol{w}^*\| \leq B$ y $\|\boldsymbol{v}_t\| \leq \rho$. □

El anterior lema 2.3.1 se puede aplicar al problema de optimización del Descenso del Gradiente tomando como $\boldsymbol{v}_t = \nabla f(\boldsymbol{w}^{(t)})$. Concretamente, si f es ρ-Lipschitz, entonces $\|\nabla f(\boldsymbol{w}^{(t)})\| \leq \rho$ y se satisfacen las hipótesis, obteniendo como corolario:

Corolario 2.3.1.1. *Sea f una función convexa y ρ-Lipschitz. Sea $\boldsymbol{w}^* \in \operatorname{argmin}_{\boldsymbol{w}:\|\boldsymbol{w}\|\leq B} f(\boldsymbol{w})$. Si se ejecuta el algoritmo del Descenso del Gradiente para la función f con T iteraciones tal que $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, entonces la salida del algoritmo $\bar{\boldsymbol{w}}$ verifica:*

$$f(\bar{\boldsymbol{w}}) - f(\boldsymbol{w}^*) \leq \frac{B\rho}{\sqrt{T}}. \tag{2.3.11}$$

Además, para cada $\epsilon > 0$, para alcanzar $f(\bar{\boldsymbol{w}}) - f(\boldsymbol{w}^*) \leq \epsilon$, es suficiente con ejecutar el algoritmo del Descenso del Gradiente T iteraciones, con $T \geq \frac{B^2\rho^2}{\epsilon^2}$. [[14] Corolario 14.2]

Una condición necesaria para poder aplicar a una función el algoritmo del Descenso del Gradiente es que sea diferenciable. Cabe destacar que para una función convexa f, el gradiente en $\boldsymbol{w}$ define la pendiente de la tangente, esto es:

$$\forall \boldsymbol{u},\ f(\boldsymbol{u}) \geq f(\boldsymbol{w}) + \langle\, \boldsymbol{u} - \boldsymbol{w}, \nabla f(\boldsymbol{w}) \,\rangle \tag{2.3.12}$$

La importancia de la tangente cuyos valores están por debajo de la función es una importante propiedad de las funciones convexas, tanto que es una caracterización.

En ocasiones, serán utilizadas funciones de pérdida no diferenciables. En ese caso, es posible adaptar el algoritmo de Descenso del Gradiente, es decir, la técnica puede aplicarse a funciones no diferenciables usando la noción de *subgradiente*, que se define a continuación [[14] Definición 14.4], de la función $f(\boldsymbol{w})$ en $\boldsymbol{w}^{(t)}$.

Definición 2.3.0.1. ***Subgradiente*** *Sea S un conjunto abierto convexo. Se denomina subgradiente de la función convexa $f : S \to R$ en $\boldsymbol{w}$ al vector $\boldsymbol{v}$ que satisface*

$$\forall \boldsymbol{u} \in S, f(\boldsymbol{u}) \geq f(\boldsymbol{w}) + \langle \, \boldsymbol{u} - \boldsymbol{w}, \boldsymbol{v} \, \rangle \tag{2.3.13}$$

El conjunto de subgradientes de la función f en el punto $\boldsymbol{w}$ se denota como $\partial f(\boldsymbol{w})$.

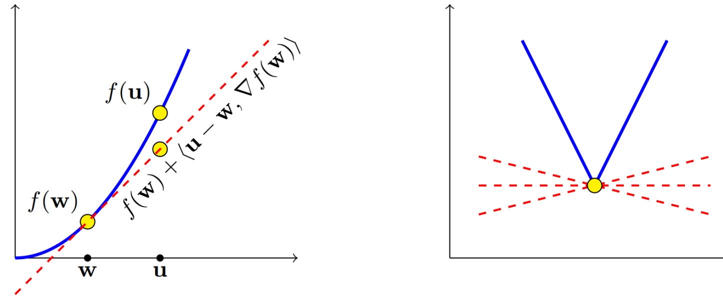


Figura 2.3.2: En la imagen de la izquierda se muestra la tangente de la función f en el punto $\boldsymbol{w}$, mientras que en la imagen de la derecha se muestran varios subgradientes para una función convexa pero no diferenciable. [[14], Figura 14.2]

Destacar aquí que el análisis de convergencia es idéntico, ya que la ecuación 2.3.4 sigue siendo válida para los subgradientes [[14], Sección 14.2].

El nuevo procedimiento conocido como **algoritmo del descenso del gradiente estocástico - SGD** (algoritmo 1) no impone que la dirección de actualización se base exactamente en el gradiente; en su lugar, permite que la dirección sea un vector aleatorio (por eso se denomina estocástico), siendo necesario que su valor esperado en cada iteración sea igual a la dirección del gradiente [[14], Sección 14.3]. En general, se requerirá simplemente que el valor esperado del vector $\boldsymbol{v}_t$ sea un subgradiente de la función en el vector correspondiente a cada iteración.

Algoritmo 1: Descenso del Gradiente Estocástico para minimizar $f(\boldsymbol{w})$

Datos: $\eta > 0$ y número entero $T > 0$
1 $\boldsymbol{w}^{(1)} \leftarrow \boldsymbol{0}$;
2 **Para** $t = 1, ...T$ **hacer**
3 Seleccionar $\boldsymbol{v}_t$ aleatoriamente tal que $E[\boldsymbol{v}_t | \boldsymbol{w}^{(t)}] \in \partial f(\boldsymbol{w}^{(t)})$;
4 Actualizar $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \boldsymbol{v}_t$;
Resultado: $\bar{\boldsymbol{w}} = \frac{1}{T} \sum_{t=1}^{T} \boldsymbol{w}^{(t)}$

Analizaremos la convergencia de este nuevo método para funciones convexas-Lipschitz acotadas [[14], Subsección 14.3.1]. Recordemos la acotación obtenida con el método del descenso del gradiente en el corolario 2.3.1.1; desafortunadamente para el caso estocástico en el que la esperanza del vector $\boldsymbol{v}_t$ pertenece al conjunto $\partial f(\boldsymbol{w}^{(t)})$, no es válida la ecuación 2.3.4. Sin embargo obtenemos el siguiente teorema [[14], Teorema 14.8]:

Teorema 2.3.2. *Sean $B, \rho > 0$. Sea f una función convexa. Sea $\boldsymbol{w}^* \in \operatorname{argmin}_{\boldsymbol{w}:\|\boldsymbol{w}\| \leq B} f(\boldsymbol{w})$. Suponemos además que para toda iteración t se verifica que $\|\boldsymbol{v}_t\| \leq \rho$ con probabilidad 1. Si se ejecuta el algoritmo del Descenso del Gradiente Estocástico para la función f con T iteraciones tal que $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, entonces:*

$$E[f(\bar{\boldsymbol{w}})] - f(\boldsymbol{w}^*) \leq \frac{B\rho}{\sqrt{T}}. \tag{2.3.14}$$

Además, para cada $\epsilon > 0$, para alcanzar $E[f(\bar{\boldsymbol{w}})] - f(\boldsymbol{w}^*) \leq \epsilon$, es suficiente con ejecutar el algoritmo del Descenso del Gradiente Estocástico T iteraciones, con T tal que $T \geq \frac{B^2\rho^2}{\epsilon^2}$.

Demostración. La notación $\boldsymbol{v}_{1:t}$ indica la secuencia $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_t$. Tomando esperanzas a ambos lados de la ecuación 2.3.3 obtenemos:

$$E_{\boldsymbol{v}_{1:T}}[f(\bar{\boldsymbol{w}}) - f(\boldsymbol{w}^*)] = E_{\boldsymbol{v}_{1:T}}[\frac{1}{T}\sum_{t=1}^{T}(f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*))]. \tag{2.3.15}$$

Dado que el lema 2.3.1 es válido para cualquier secuencia $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_t$, lo podemos aplicar:

$$E_{\boldsymbol{v}_{1:T}}[\frac{1}{T}\sum_{t=1}^{T}\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle] \leq \frac{B\rho}{\sqrt{T}}. \tag{2.3.16}$$

Para completar la prueba quedaría probar la desigualdad:

$$E_{\boldsymbol{v}_{1:T}}[\frac{1}{T}\sum_{t=1}^{T}(f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*))] \leq E_{\boldsymbol{v}_{1:T}}[\frac{1}{T}\sum_{t=1}^{T}\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle]. \tag{2.3.17}$$

Aplicando la propiedad de linealidad de la esperanza matemática:

$$E_{\boldsymbol{v}_{1:T}}[\frac{1}{T}\sum_{t=1}^{T}\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle] = \frac{1}{T}\sum_{t=1}^{T} E_{\boldsymbol{v}_{1:T}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle]. \tag{2.3.18}$$

Para dos variables aleatorias α, β y una función g se tiene que $E_\alpha[g(\alpha)] = E_\beta[E_\alpha[g(\alpha)|\beta]]$. Tomando como $\alpha = \boldsymbol{v}_{1:t}$ y $\beta = \boldsymbol{v}_{1:t-1}$ se obtiene:

$$\begin{aligned} E_{\boldsymbol{v}_{1:T}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle] &= E_{\boldsymbol{v}_{1:t}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle] \\ &= E_{\boldsymbol{v}_{1:t-1}} E_{\boldsymbol{v}_{1:t}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle | \boldsymbol{v}_{1:t-1}]. \end{aligned} \tag{2.3.19}$$

Una vez que el valor de $\boldsymbol{v}_{1:t-1}$ es conocido, el valor $\boldsymbol{w}^{(t)}$ no es aleatorio y por tanto:

$$E_{\boldsymbol{v}_{1:t-1}} E_{\boldsymbol{v}_{1:t}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, \boldsymbol{v}_t \,\rangle | \boldsymbol{v}_{1:t-1}] = E_{\boldsymbol{v}_{1:t-1}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, E_{\boldsymbol{v}_t}[\boldsymbol{v}_t | \boldsymbol{v}_{1:t-1}] \,\rangle]. \tag{2.3.20}$$

Dado que $\boldsymbol{w}^{(t)}$ depende únicamente de $\boldsymbol{v}_{1:t-1}$ y el algoritmo del descenso del gradiente estocástico impone que $E[\boldsymbol{v}_t | \boldsymbol{w}^{(t)}] \in \partial f(\boldsymbol{w}^{(t)})$, entonces $E[\boldsymbol{v}_t | \boldsymbol{v}_{1:t-1}] \in \partial f(\boldsymbol{w}^{(t)})$. Por tanto,

$$E_{\boldsymbol{v}_{1:t-1}}[\langle\, \boldsymbol{w}^{(t)} - \boldsymbol{w}^*, E_{\boldsymbol{v}_t}[\boldsymbol{v}_t | \boldsymbol{v}_{1:t-1}] \,\rangle] \geq E_{\boldsymbol{v}_{1:t-1}}[f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*)]. \tag{2.3.21}$$

Juntando todo, hemos probado que:

$$E_{\boldsymbol{v}_{1:T}}[\langle\ \boldsymbol{w}^{(t)} - \boldsymbol{w}^{*}, \boldsymbol{v}_t\ \rangle] \geq E_{\boldsymbol{v}_{1:t-1}}[f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^{*})] = E_{\boldsymbol{v}_{1:T}}[f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^{*})] \tag{2.3.22}$$

Sumando en t, dividiendo por T y usando la linealidad de la esperanza obtenemos la ecuación 2.3.17, dando por finalizada la prueba del teorema. □

Una posible variante del algoritmo del descenso del gradiente estocástico es tomar como tasa de aprendizaje η una función que dependa del valor t, por ejemplo, $\eta_t = \frac{B}{\rho\sqrt{t}}$ con la finalidad de variar poco el punto cuando nos acercamos al mínimo.

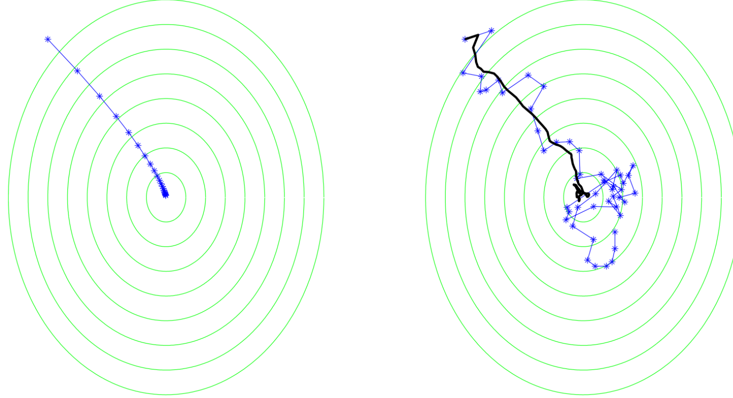


Figura 2.3.3: En la imagen de la izquierda se muestra el algoritmo del descenso del gradiente y en la derecha el algoritmo del descenso del gradiente estocástico, ambos con el objetivo de minimizar la función $1{,}25(x+6)^2 + (y-8)^2$ [[14], Sección 14.2]

Hasta este momento hemos introducido e analizado el algoritmo del descenso del gradiente estocástico para funciones convexas en general. Consideraremos ahora su aplicación al problema de minimizar la función de coste $\mathcal{R}_{\mathcal{S}}(\boldsymbol{w})$, ya que en muchos problemas de aprendizaje automático no es posible calcular el gradiente de $\mathcal{R}_{\mathcal{D}}(\boldsymbol{w})$ [[14], Sección 14.5]. En primer lugar, supongamos que la función de pérdida es diferenciable y por ende el riesgo es diferenciable. La construcción del vector $\boldsymbol{v}_t$ se realizará de la siguiente forma: (1) Muestreamos $z = (\boldsymbol{x}, y)$ a partir de la distribución empírica que disponemos D_S gracias al conjunto de entrenamiento; (2) definimos $\boldsymbol{v}_t$ como el gradiente de la función $\ell(z, \boldsymbol{w})$ con respecto a $\boldsymbol{w}$ en el punto $\boldsymbol{w}^{(t)}$. Gracias a la linealidad del gradiente se tiene que:

$$E[\boldsymbol{v}_t|\boldsymbol{w}^{(t)}] = E_{z\sim D_S}[\nabla\ell(z, \boldsymbol{w}^{(t)})] = \nabla E_{z\sim D_S}[\ell(z, \boldsymbol{w}^{(t)})] = \nabla\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}^{(t)}). \tag{2.3.23}$$

El gradiente de la función de pérdida $\ell(z, \boldsymbol{w})$ en el punto $\boldsymbol{w}^{(t)}$ es por tanto un estimador insesgado del gradiente del riesgo $\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}^{(t)})$. El mismo argumento se obtiene para funciones no diferenciables tomando como $\boldsymbol{v}_t$ un subgradiente de $\ell(z, \boldsymbol{w})$ en el punto $\boldsymbol{w}^{(t)}$. Para cada $\boldsymbol{u}$ se tiene $\ell(z, \boldsymbol{u}) - \ell(z, \boldsymbol{w}^{(t)}) \geq \langle\ \boldsymbol{u} - \boldsymbol{w}^{(t)}, \boldsymbol{v}_t\ \rangle$. Tomando esperanzas condicionadas al valor $\boldsymbol{w}^{(t)}$ a ambos lados de la ecuación se obtiene:

$$\begin{aligned} \mathcal{R}_{\mathcal{S}}(\boldsymbol{u}) - \mathcal{R}_{\mathcal{S}}(\boldsymbol{w}^{(t)}) &= E[\ell(z, \boldsymbol{u}) - \ell(z, \boldsymbol{w}^{(t)})|\boldsymbol{w}^{(t)}] \\ &\geq E[\langle\ \boldsymbol{u} - \boldsymbol{w}^{(t)}, \boldsymbol{v}_t\ \rangle|\boldsymbol{w}^{(t)}] = \langle\ \boldsymbol{u} - \boldsymbol{w}^{(t)}, E[\boldsymbol{v}_{(t)}|\boldsymbol{w}^{(t)}]\ \rangle \end{aligned} \tag{2.3.24}$$

Luego $E[\boldsymbol{v}_{(t)}|\boldsymbol{w}^{(t)}]$ es un subgradiente de $\mathcal{R}_{\mathcal{S}}(\boldsymbol{w})$ en $\boldsymbol{w}^{(t)}$.

Algoritmo 2: Descenso del Gradiente Estocástico para minimizar $\mathcal{R}_{\mathcal{S}}(\boldsymbol{w})$

Datos: $\eta > 0$ y número entero $T > 0$

1 $\boldsymbol{w}^{(1)} \leftarrow \mathbf{0}$;
2 **Para** $t = 1, ...T$ **hacer**
3   Muestrear $(\boldsymbol{x}, y) \sim \mathcal{D}_S$;
4   Seleccionar $\boldsymbol{v}_t \in \partial \ell((\boldsymbol{x}, y), \boldsymbol{w})$;
5   Actualizar $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \boldsymbol{v}_t$;

Resultado: $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}^{(t)}$

Corolario 2.3.2.1. *Consideremos un problema de aprendizaje convexo, Lipschitz acotado con parámetros ρ, B. Entonces, para todo $\epsilon > 0$, si ejecutamos el algoritmo del descenso del gradiente estocástico SGD para minimizar $\mathcal{R}_{\mathcal{S}}(\boldsymbol{w})$ con T iteraciones tal que $T \geq \frac{B^2\rho^2}{\epsilon^2}$ y $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, entonces la salida del algoritmo satisface:*

$$E[\mathcal{R}_{\mathcal{S}}(\bar{\boldsymbol{w}})] \leq \min_{\boldsymbol{w} \in \mathcal{H}} \mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \epsilon. \tag{2.3.25}$$

[[14], Corolario 14.12]

En ocasiones, como en el modelo de redes neuronales (sección Redes neuronales artificiales algoritmo 4) es interesante resolver el problema de optimización asociado a la minimización regularizada donde:

$$\boldsymbol{w} \in argmin_{\boldsymbol{w}}(\mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \frac{\lambda}{2}||\boldsymbol{w}||^2). \tag{2.3.26}$$

Algoritmo 3: Descenso del Gradiente Estocástico para minimizar una función λ-fuertemente convexa

Datos: número entero T

1 $\boldsymbol{w}^{(1)} \leftarrow \mathbf{0}$;
2 **Para** $t = 1, ...T$ **hacer**
3   Seleccionar $\boldsymbol{v}_t$ tal que $E[\boldsymbol{v}_t | \boldsymbol{w}^{(t)}] \in \partial f(\boldsymbol{w}^{(t)})$;
4   Actualizar $\eta_t = \frac{1}{\lambda t}$;
5   Actualizar $\boldsymbol{w}^{(t+\frac{1}{2})} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{v}_t$;
6   Actualizar $\boldsymbol{w}^{(t+1)} = \text{argmin}_{\boldsymbol{w}} ||\boldsymbol{w} - \boldsymbol{w}^{(t+\frac{1}{2})}||^2$;

Resultado: $\bar{w} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}^{(t)}$

Analizaremos la convergencia del algoritmo precedente [[14], Subsección 14.4.4].

Lema 2.3.3. *Si f es una función λ-fuertemente convexa, entonces para todo $\boldsymbol{w}, \boldsymbol{u}$ y $\boldsymbol{v} \in \partial f(\boldsymbol{w})$ se tiene:*

$$\langle \boldsymbol{w} - \boldsymbol{u}\ ,\ \boldsymbol{v} \rangle \geq f(\boldsymbol{w}) - f(\boldsymbol{u}) + \frac{\lambda}{2}||\boldsymbol{w} - \boldsymbol{u}||^2. \tag{2.3.27}$$

La prueba es análoga al lema 2.2.2. [[14], Lema 14.10]

Teorema 2.3.4. *Sea f una función λ-fuertemente convexa tal que $E[||\boldsymbol{v}_t||^2] \leq \rho^2$. Sea $\boldsymbol{w}^* \in argmin_{\boldsymbol{w}} f(\boldsymbol{w})$ una solución óptima. Entonces:*

$$E[f(\bar{\boldsymbol{w}})] - f(\boldsymbol{w}^*) \leq \frac{\rho^2}{2\lambda T}(1 + \log(T)). \tag{2.3.28}$$

Demostración. Sea $\nabla^{(t)} = E[\boldsymbol{v}_t|\boldsymbol{w}^{(t)}]$. Dado que f es fuertemente convexa y $\nabla^{(t)}$ pertenece al subgradiente de f en $\boldsymbol{w}^{(t)}$, se tiene que:

$$\langle \boldsymbol{w}^{(t)} - \boldsymbol{w}^* \ , \ \nabla^{(t)} \rangle \geq f(\boldsymbol{w}^{(t)}) - f(\boldsymbol{w}^*) + \frac{\lambda}{2}||\boldsymbol{w}^{(t)} - \boldsymbol{w}^*||^2. \tag{2.3.29}$$

Probaremos la desigualdad:

$$\langle \boldsymbol{w}^{(t)} - \boldsymbol{w}^* \ , \ \nabla^{(t)} \rangle \leq \frac{E[||\boldsymbol{w}^{(t)} - \boldsymbol{w}^*||^2 - ||\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^*||^2]}{2\eta_t} + \frac{\eta_t}{2}\rho^2. \tag{2.3.30}$$

Como $\boldsymbol{w}^{(t+1)} = \text{argmin}_{\boldsymbol{w}}||\boldsymbol{w} - \boldsymbol{w}^{(t+\frac{1}{2})}||^2$ se tiene:

$$\begin{aligned} ||\boldsymbol{w}^{(t)} - \boldsymbol{w}^*||^2 - ||\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^*||^2 &\geq ||\boldsymbol{w}^{(t)} - \boldsymbol{w}^*||^2 - ||\boldsymbol{w}^{(t+\frac{1}{2})} - \boldsymbol{w}^*||^2 \\ &= 2\eta_t \langle \boldsymbol{w}^{(t)} - \boldsymbol{w}^* \ , \ \boldsymbol{v}_t \rangle - \eta_t^2 ||\boldsymbol{v}_t||^2. \end{aligned} \tag{2.3.31}$$

Tomando esperanzas en ambas partes y haciendo uso de la hipótesis $E[||\boldsymbol{v}_t||^2] \leq \rho^2$ se obtiene la ecuación 2.3.30. Sumando en t los resultados obtenidos en las ecuaciones 2.3.29 y 2.3.30 se deduce que:

$$\begin{aligned} &\sum_{t=1}^{T}(E[f(\boldsymbol{w}^{(t)})] - f(\boldsymbol{w}^*)) \\ &\leq E[\sum_{t=1}^{T}(\frac{||\boldsymbol{w}^{(t)} - \boldsymbol{w}^*||^2 - ||\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^*||^2}{2\eta_t} - \frac{\lambda}{2}||\boldsymbol{w}^{(t)} - \boldsymbol{w}^*||^2)] + \frac{\rho^2}{2}\sum_{t=1}^{T}\eta_t \end{aligned} \tag{2.3.32}$$

Teniendo en cuenta que el primer sumatorio de la parte derecha de la ecuación 2.3.32 es $-\lambda T||\boldsymbol{w}^{(T+1)} - \boldsymbol{w}^*||^2 \leq 0$ y por definición $\eta_t = 1/(\lambda t)$:

$$\sum_{t=1}^{T}(E[f(\boldsymbol{w}^{(t)})] - f(\boldsymbol{w}^*)) \leq \frac{\rho^2}{2\lambda}\sum_{t=1}^{T}\frac{1}{t} \leq \frac{\rho^2}{2\lambda}(1 + \log(T)). \tag{2.3.33}$$

La prueba se concluye dividiendo por T y aplicando la desigualdad de Jensen. [[14], Teorema 14.11] □

La función de coste definida como $f(\boldsymbol{w}) = \mathcal{R}_{\mathcal{S}}(\boldsymbol{w}) + \frac{\lambda}{2}||\boldsymbol{w}||^2$ es una función λ-fuertemente convexa, por tanto es posible aplicar el algoritmo del descenso del gradiente estocástico con algunas modificaciones, algoritmo 3. En primer lugar, se construye un estimador insesgado de un subgradiente de f en $\boldsymbol{w}^{(t)}$ teniendo en cuenta que al muestrear $(\boldsymbol{x}, y)$ uniformemente en S y elegir $\boldsymbol{v}_t \in \partial\ell((\boldsymbol{x}, y), \boldsymbol{w}^{(t)})$, la esperanza de $\lambda\boldsymbol{w}^{(t)} + \boldsymbol{v}_t$ es un subgradiente de f en $\boldsymbol{w}^{(t)}$. La regla de actualización del gradiente es por tanto $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \frac{1}{\lambda t}(\lambda\boldsymbol{w}^{(t)} + \boldsymbol{v}_t) = -\frac{1}{\lambda t}\sum_{i=1}^{t}\boldsymbol{v}_i$. Asumiendo que la función de pérdida es ρ-Lipschitz se verifica para todo t que $||\boldsymbol{v}_t|| \leq \rho$, $||\lambda\boldsymbol{w}^{(t)}|| \leq \rho$ y en consecuencia $||\lambda\boldsymbol{w}^{(t)} + \boldsymbol{v}_t|| \leq 2\rho$. Haciendo uso del teorema 2.3.4, después de T iteraciones se obtiene $E[f(\bar{\boldsymbol{w}})] - f(\boldsymbol{w}^*) \leq \frac{4\rho^2}{\lambda T}(1 + \log(T))$. [[14] Subsección 14.5.3]

2.4. Conclusiones

En este capítulo se han introducido los conceptos y notación necesarios para describir el problema de clasificación binaria dentro del marco del aprendizaje supervisado desde un punto de vista matemático. Descrito el problema, se ha procedido a estudiar diferentes algoritmos de optimización que generan, basándose en el conjunto de entrenamiento S, un predictor con menor riesgo posible, dentro del marco de los problemas de aprendizaje convexos. Como primera técnica se encuentra el procedimiento de Descenso del Gradiente, definido para funciones diferenciables. El análisis de convergencia se ha realizado para funciones convexas y Lipschitz, obteniendo garantías computacionales (corolario 2.3.1.1). Sin embargo, en el ajuste de Redes Neuronales es cada vez más frecuente el uso de funciones de pérdida (como la ReLU) convexas pero no diferenciables, condición necesaria para el procedimiento anterior. En esta situación es posible adaptar el algoritmo manteniendo el análisis de convergencia usando la noción de subgradiente. Después se estudia un nuevo procedimiento conocido como Descenso del Gradiente Estocástico, donde la dirección de actualización es un vector aleatorio con valor esperado un subgradiente de la función en el vector correspondiente. Este nuevo método posee garantías computacionales para funciones convexas Lipschitz y acotadas. El procedimiento de Descenso del Gradiente Estocástico es aplicado al problema de minimización de la función de coste $\mathcal{R}_S$ obteniendo garantías estadísticas y computacionales (incrementando en número de iteraciones el riesgo esperado es tan cercano al mínimo riesgo como se desee, corolario 2.3.2.1). La sección finaliza con la aplicación de la técnica de Descenso del Gradiente Estocástico para minimizar una función λ-fuertemente convexa como la función de coste asociada a la minimización regularizada.

Hasta el momento, los problemas de aprendizaje convexos han sido el marco teórico de estudio. Éstos poseen garantías estadísticas al no sufrir sobreajuste y garantías computacionales, esto es, después de un número de iteraciones se obtiene un buen predictor. A partir de este capítulo la teoría estudiada hasta ahora no funcionará. Un caso particular es el modelo de redes neuronales, que sorprendentemente en la práctica funciona bien pero no posee garantías teóricas.

CAPÍTULO 3

Aprendizaje profundo

Una de las familias de clases de hipótesis sobre las que se basan muchos algoritmos de aprendizaje, debido a ser intuitiva, fácil de interpretar y ajustarse a los datos razonablemente bien en algunos problemas es la **familia de predictores lineales** [[14], Capítulo 9]. En primera instancia, definiremos la clase de las funciones afines como

$$L_d = \{h_{\boldsymbol{w}} : \boldsymbol{w} \in R^{d+1}\},$$

donde

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \langle \boldsymbol{w} \ , \ \boldsymbol{x} \rangle = \langle (b, w_1, .., w_d) \ , \ (1, x_1, ..., x_d) \rangle = b + \sum_{i=1}^{d} w_i x_i.$$

Equivalentemente, $L_d = \{\boldsymbol{x} \mapsto \langle \boldsymbol{w} \ , \ \boldsymbol{x} \rangle : \boldsymbol{w} \in R^{d+1}\}$. Esto quiere decir que L_d es un conjunto de funciones, donde cada una está parametrizada por $\boldsymbol{w} \in R^{d+1}$ y asigna para cada vector $\boldsymbol{x} = (1, x_1, x_2, ..., x_d)$, el valor $\langle \boldsymbol{w} \ , \ \boldsymbol{x} \rangle$.

A partir de la familia de predictores lineales se obtienen los **clasificadores lineales** cuyo objetivo es clasificar los puntos del dominio en clases disjuntas; en particular, para el problema de clasificación binaria en dos clases C_0 y C_1. Cada regla lineal (o hipótesis), parametrizada por $\boldsymbol{w}$, decide la clase en función del resultado de aplicar una tranformación afín $h_{\boldsymbol{w}}$; es decir, para un elemento $\boldsymbol{x}$ del dominio, si $h_{\boldsymbol{w}}(\boldsymbol{x}) \geq 0$ entonces pertenece a C_1, de lo contrario a C_0. En otras palabras, el dominio de puntos es separado en dos hiperespacios a través del hiperplano $h_{\boldsymbol{w}} = 0$ y posteriormente se asigna la clase C_j dependiendo del resultado del algoritmo anterior (Figura 3.0.1). Suponiendo que sea posible separar los puntos gracias a un hiperplano, el objetivo es encontrar un hiperplano, caracterizado por el vector $\boldsymbol{w}$, que anule el error empírico (hiperplano MRE). Una posible solución para hallar el hiperplano con error empírico nulo es el *algoritmo Perceptron (F. Rosenblatt, 1958 [13])*. Este algoritmo iterativo consiste en construir una secuencia de vectores $\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, ...,$ donde se inicia con

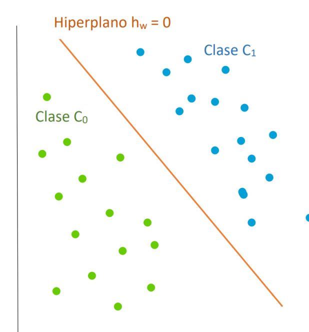


Figura 3.0.1: Gráfico explicativo regla lineal para un dominio bidimensional. Elaboración propia.

$\boldsymbol{w}^{(1)}$ como vector nulo. En cada iteración t, si el algoritmo encuentra un punto mal clasificado $(\boldsymbol{x}_i, y_i)$, esto es, $h_{\boldsymbol{w}}(\boldsymbol{x}_i)y_i \leq 0$ entonces actualiza el vector $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + y_i\boldsymbol{x}_i$ con el objetivo de alcanzar $y_i\langle \boldsymbol{w}\ ,\ \boldsymbol{x}_i\rangle > 0$. En caso contrario, devuelve el vector $\boldsymbol{w}^{(t)}$. Notemos que

$$y_i\langle \boldsymbol{w}^{(t+1)}\ ,\ \boldsymbol{x}_i\rangle = y_i\langle \boldsymbol{w}^{(t)} + y_i\boldsymbol{x}_i\ ,\ \boldsymbol{x}_i\rangle = y_i\langle \boldsymbol{w}^{(t)}\ ,\ \boldsymbol{x}_i\rangle + \|\boldsymbol{x}_i\|^2,$$

por tanto, parece intuirse que el algoritmo se encamina hacia una solución 'correcta' para el punto $(\boldsymbol{x}_i, y_i)$. Se puede probar que bajo la suposición de que sea posible separar el espacio por un hiperplano, el algoritmo Perceptron finaliza con todos los puntos clasificados correctamente en un tiempo finito, es decir, encuentra el hiperplano MRE [[14], Teorema 9.1].

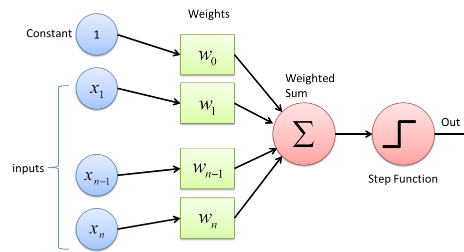


Figura 3.0.2: Esquema de los pasos que realiza un clasificador lineal. Fuente: enlace

Desafortunadamente, para poder aplicar los algoritmos anteriores con garantías debemos suponer que es posible separar los puntos del dominio por un hiperplano. En el caso de no cumplirse dicha condición, no es posible hallar el hiperplano. Prueba de ello es el siguiente ejemplo. Sea $\mathcal{X} = \{[0,0], [0,1], [1,0], [1,1]\}$ un conjunto bidimensional y consideremos que un elemento $\boldsymbol{x} = (x_1, x_2) \in \mathcal{X}$ pertenece a la clase 0 cuando $(x_1, x_2) = (0,0)$ ò $(x_1, x_2) = (1,1)$, mientras que pertenece a la clase 1 cuando $(x_1, x_2) = (1,0)$ ò $(x_1, x_2) = (0,1)$, ver figura 3.0.3.

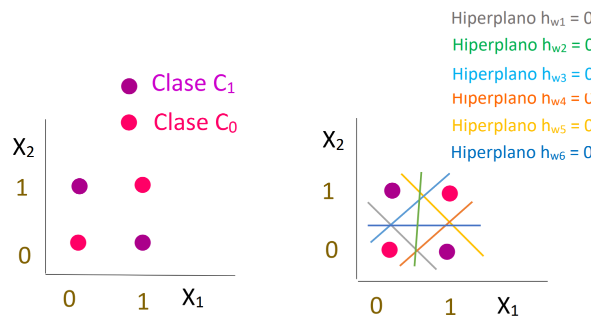


Figura 3.0.3: Ejemplo ilustrativo. Elaboración propia.

Para este ejemplo concreto no es posible separar los puntos en función de la clase a la que pertenecen a través de un hiperplano, es decir, no podemos hacer uso de un solo clasificador lineal. Para afrontar el problema cuando los puntos no son separables por un hiperplano, es necesario encontrar fronteras más complejas geométricamente y por tanto necesitamos más flexibilidad en la clase de hipótesis. Los *clasificadores no lineales multicapa*, o también conocidos como *multilayer perceptron MLP* solucionan el problema de la clasificación para conjuntos de datos no linealmente separables, enriqueciendo la clase de hipótesis gracias a

la no linealidad. Estos construyen una arquitectura más compleja basada en un conjunto de clasificadores no lineales organizados en capas, donde cada clasificador se denomina nodo. Resolvamos el ejemplo con el modelo introducido (ver figura 3.0.4).

La capa inicial recoge los valores de los atributos para un punto en concreto, es decir, tomemos $(x_1, x_2) = (0, 0)$. A continuación, en la capa oculta se encuentran dos clasificadores con respectivas funciones:

$$h_{w=(-1,1,1)}((x_1,x_2)) = \begin{cases} 0 & \text{si } x_1 + x_2 < 1 \\ 1 & \text{si } x_1 + x_2 \geq 1 \end{cases}$$

$$h_{w=(1,-1,-1)}((x_1,x_2)) = \begin{cases} 0 & \text{si } -x_1 - x_2 < -1 \\ 1 & \text{si } -x_1 - x_2 \geq -1 \end{cases}$$

Para el punto $(x_1, x_2) = (0, 0)$ se obtiene $h_{w=(-1,1,1)}((0,0)) = 0$ y $h_{w=(1,-1,-1)}((0,0)) = 1$. Por último, en la capa final, el nuevo punto $(o_1, o_2) = (0, 1)$ se clasifica según la regla:

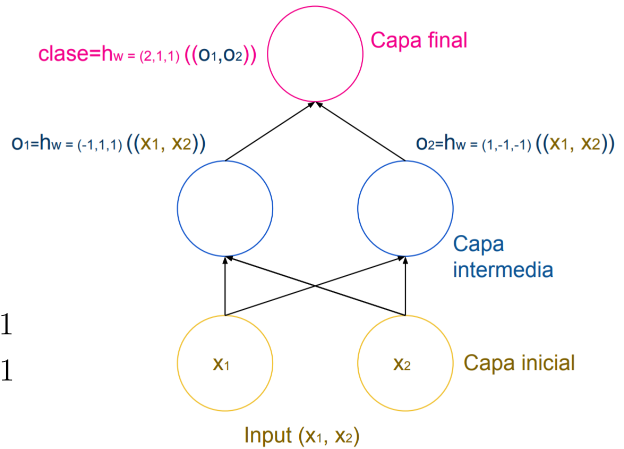


Figura 3.0.4: Arquitectura formada por nodos organizados en capas capaz de resolver el ejemplo propuesto. Imagen: elaboración propia.

$$h_{w=(2,1,1)}((o_1,o_2)) = \begin{cases} 0 & \text{si } o_1 + o_2 < 2 \\ 1 & \text{si } o_1 + o_2 \geq 2 \end{cases}$$

Y obtenemos para el punto $(x_1, x_2) = (0, 0)$ la etiqueta 0 o equivalentemente la clase 0, debido a que $h_{w=(2,1,1)}((0,1)) = 0$. Análogamente se obtienen las etiquetas para los tres elementos restantes de $\mathcal{X}$.

Cabe destacar que hemos conseguido fronteras no lineales de separación gracias a usar en la nueva arquitectura como reglas de clasificación funciones no continuas (y por tanto no diferenciables), ver figura 3.0.5.

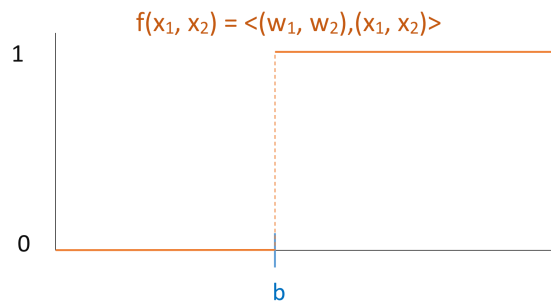


Figura 3.0.5: Función no continua utilizada en la anterior arquitectura para resolver el ejemplo propuesto, $h_{\boldsymbol{w}}(\boldsymbol{x}) = \langle \boldsymbol{w} \ , \ \boldsymbol{x} \rangle = \langle (b, w_1, .., w_d) \ , \ (1, x_1, ..., x_d) \rangle$. Imagen: elaboración propia.

En suma, para resolver el problema se necesita más flexibilidad en la clase de hipótesis, es decir, una arquitectura más compleja basada en un conjunto de clasificadores organizados en capas, donde cada clasificador se denomina nodo y las reglas de clasificación pueden ser funciones no continuas. Cuando el número de capas es tres o más, se habla de *aprendizaje profundo o deep learning* en la literatura inglesa.

3.1. Redes neuronales artificiales

Una **red neuronal artificial (ANN)** es una colección de nodos conectados entre sí donde la red forma un grafo dirigido y ponderado. MLP es un ejemplo de red neuronal artificial totalmente conectada. En este trabajo centraremos la atención en las **redes neuronales feedforward**, caracterizadas porque las conexiones entre los nodos no forman un ciclo, es decir, la información se mueve en una sola dirección, desde los nodos de la entrada, a través de los nodos ocultos (si los hay) hasta los nodos de salida. El término *red* hace referencia a que habitualmente son representadas como composición de varias funciones (ecuación 3.1.3).

Matemáticamente, una red neuronal feedforward esta caracterizada por un grafo dirigido y acíclico, $G = (V, E)$ junto con una función de pesos sobre las aristas, $w : E \to R$, que no debe tener necesariamente la imagen de todas las aristas distinta del valor cero, pues interesa tener no demasiados parámetros entrenables para no producir sobreajuste. Debido a que la red está organizada en capas, los nodos pueden ser reagrupados en conjuntos disjuntos $V = \dot{\cup}_{m=0}^{M} V_m$, tal que cada arista de E conecta un nodo de la capa V_{m-1} con uno de V_m para algún $m \in [M]$. La capa V_0 es denominada capa inicial y contiene $d + 1$ nodos que son los inputs, donde $o_{0,0} = 1$ y $o_{0,i} = x_i$. Cada nodo es modelado por una función escalar $\phi : R \to R$, que recibe como input la suma ponderada de los outputs de los nodos conectados con él, con el objetivo de introducir no linealidad en el modelo (si dicha función es no lineal) para conseguir mayor flexibilidad en él mismo. La función ϕ será denominada *función de activación* y se elegirá en función del valor de salida buscado. Denotaremos por $v_{m,i}$ el i-ésimo nodo de la capa m y $o_{m,i}(\boldsymbol{x})$ el resultado de $v_{m,i}$ cuando el vector de inputs de la red neuronal es $\boldsymbol{x}$. El número de parámetros entrenables es $|E| + M$ [[14], Capítulo 20].

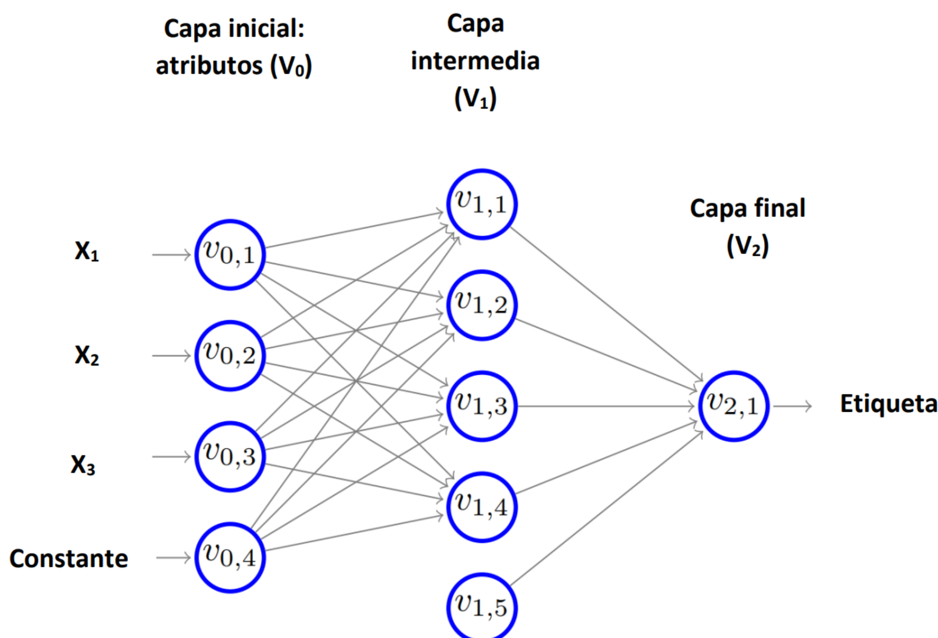


Figura 3.1.1: Esquema gráfico representativo de la red neuronal feedforward. Nota: en la descripción matemática el input $v_{0,1}$ es la constante ($o_{0,0}$) y los demás los elementos de $\boldsymbol{x}$. Fuente: [[14], Sección 20.1]

Bajo la hipótesis de que ya han sido calculados los valores de los nodos para la capa m, **la descripción matemática para una red neuronal artificial feedforward**, esto es, para los nodos de la capa $m+1$ es:

$$\begin{aligned} a_{m+1,i}(\boldsymbol{x}) &= \sum_{r:(v_{m,r},v_{m+1,i})\in E} w((v_{m,r},v_{m+1,i}))o_{m,r}(\boldsymbol{x}), \\ o_{m+1,i}(\boldsymbol{x}) &= \phi(a_{m+1,i}(\boldsymbol{x})). \end{aligned} \tag{3.1.1}$$

donde $v_{m+1,i} \in V_{m+1}$ es el elemento i de la capa $m+1$ fijado previamente, $a_{m+1,i}(\boldsymbol{x})$ es el input de $v_{m+1,i}$ cuando el vector de inputs de la red neuronal es $\boldsymbol{x}$. Con cierto abuso de notación, $\phi(\boldsymbol{z}) = [\phi(z_1), ..., \phi(z_d)]'$ cuando $\boldsymbol{z}$ es un vector de dimensión d. Llamaremos *profundidad* del modelo a la longitud de la cadena descrita M y *ancho* de la red al valor $\max_m |V_m|$ [[14], Sección 20.1]. La descripción matemática 3.1.1 escrita con notación vectorial es:

$$\begin{aligned} \boldsymbol{a}_{m+1}(\boldsymbol{x}) &= \boldsymbol{W^{(m)}}\boldsymbol{o}_m(\boldsymbol{x}), \\ \boldsymbol{o}_{m+1}(\boldsymbol{x}) &= \phi(\boldsymbol{a}_{m+1}(\boldsymbol{x})). \end{aligned} \tag{3.1.2}$$

La matriz $\boldsymbol{W^{(m)}}$ asociada a la capa m tiene como entrada un valor no nulo $w((v_{m,r}, v_{m+1,i}))$ cuando la posición (i,r) verifica $(v_{m,r}, v_{m+1,i}) \in E$. Es decir, $\boldsymbol{W^{(m)}}_{(i,r)} = w((v_{m,r}, v_{m+1,i}))$ si se verifica $(v_{m,r}, v_{m+1,i}) \in E$ y 0 en otro caso. Gracias a la organización en capas se deduce fácilmente que la salida de la red se expresa como composición de funciones:

$$\boldsymbol{o}_M = \phi(\boldsymbol{W^{(M-1)}}(\phi(\boldsymbol{W^{(M-2)}}(\cdots\phi(\boldsymbol{W^{(0)}}\boldsymbol{x}))))). \tag{3.1.3}$$

3.1.1. Funciones de activación

En la estructura de red neuronal artificial, como se ha mencionado, cada nodo es modelado por una función escalar $\phi : R \to R$. La función ϕ será denominada **función de activación** y se elegirá en función del valor de salida buscado. Repasaremos algunas de las funciones de activación no lineales más utilizadas en la práctica.

En el marco de la clasificación binaria, deseamos predecir el valor de la variable $y \in \mathcal{Y} = \{0,1\}$. Si en vez, el objetivo es modelar la relación entre la probabilidad $p(\boldsymbol{x}) = P(y=1|\boldsymbol{x})$ y $\boldsymbol{x}$, obteniendo por tanto una probabilidad al final del algoritmo, no es posible usando como probabilidad una función lineal en los nodos finales. En este momento es apropiado conectar este problema con la **regresión logística**, modelo de regresión que tiene como aplicación la clasificación. Sea $S = \{(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), ..., (\boldsymbol{x_n}, y_n)\}$, con $y_i \in \{0,1\}$. Siendo $p \in [0,1]$ una probabilidad, se define la transformación $logit(p) = \log(\frac{p}{1-p}) \in (-\infty, \infty)$. En este modelo se asume que, siendo fijo $X = \boldsymbol{x}$:

$$logit(E[Y|X=\boldsymbol{x}]) = logit(P(Y=1|X=\boldsymbol{x})) = \boldsymbol{w}'\boldsymbol{x}. \tag{3.1.4}$$

En otras palabras, $logit(E[Y|X=\boldsymbol{x}])$ es lineal en $\boldsymbol{x}$. Aplicando la inversa de la transformación *logit* se obtiene:

$$\begin{aligned} p(\boldsymbol{x};\boldsymbol{w}) := P(Y=1|X=\boldsymbol{x}) &= \frac{1}{1+e^{-\boldsymbol{w}'\boldsymbol{x}}} \\ 1-p(\boldsymbol{x};\boldsymbol{w}) = P(Y=0|X=\boldsymbol{x}) &= \frac{e^{-\boldsymbol{w}'\boldsymbol{x}}}{1+e^{-\boldsymbol{w}'\boldsymbol{x}}} = \frac{1}{1+e^{\boldsymbol{w}'\boldsymbol{x}}} \end{aligned} \tag{3.1.5}$$

Una vez definida la descripción del modelo de regresión logística, el fin es hallar un clasificador $h_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{w}'\boldsymbol{x}$ 'adecuado'. Concretamente, el objetivo es la estimación de $\boldsymbol{w}$ donde se hace uso del *método de máxima verosimilitud*. Sea L la verosimilitud:

$$L(\boldsymbol{w}) = \prod_{i=1}^{n} P(Y = y_i | X = \boldsymbol{x}_i) = \prod_{i=1}^{n} (p_i)^{y_i} \cdot (1 - p_i)^{1-y_i} \tag{3.1.6}$$

donde $p_i = p(\boldsymbol{x}_i; \boldsymbol{w}) = \frac{1}{1+e^{-\boldsymbol{w}'\boldsymbol{x}_i}}$. Se denotará por $\phi_{sig} : R \to [0,1]$ a la función $\phi_{sig}(z) := \frac{1}{1+e^{-z}}$, ver figura 3.1.2. Notamos que $p_i = p(\boldsymbol{x}_i; \boldsymbol{w}) = \phi_{sig}(\boldsymbol{w}'\boldsymbol{x}_i)$. Como consecuencia inmediata se tiene que el clasificador $h_{\boldsymbol{w}}$ clasifica bien cuando $\boldsymbol{w}'\boldsymbol{x}_i > 0$ si $y_i = 1$ ($\boldsymbol{w}'\boldsymbol{x}_i > 0 \Rightarrow p_i > 1/2$) y cuando $\boldsymbol{w}'\boldsymbol{x}_i < 0$ si $y_i = 0$.

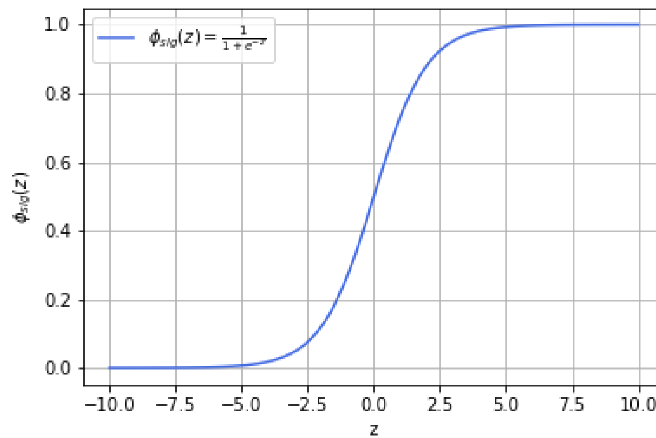


Figura 3.1.2: Función sigmoide: diferenciable y monótona por tanto el gradiente está definido en todos sus puntos. Notamos que $\phi_{sig}(0) = 1/2$. Imagen: elaboración propia.

Sin pérdida de generalidad, cambiamos la codificación de la etiqueta, esto es:

$$\begin{cases} y_i = 0 & \Leftrightarrow \quad y_i^* = -1 \\ y_i = 1 & \Leftrightarrow \quad y_i^* = 1 \end{cases} \tag{3.1.7}$$

Con esta nueva codificación, el clasificador $h_{\hat{\boldsymbol{w}}}(\boldsymbol{x}) = 1 \Leftrightarrow p(\boldsymbol{x}; \hat{\boldsymbol{w}}) \geq 1/2$ y por ende, clasifica correctamente el elemento $(\boldsymbol{x}_i, y_i^*)$ cuando $y_i^*(\boldsymbol{w}'\boldsymbol{x}_i) > 0$, que es exactamente la función de pérdida 0-1.

Dado que la función logaritmo es monótona creciente maximizar la verosimilitud 3.1.6 es equivalente a maximizar la cantidad [1]:

$$\frac{1}{n}\ell(\boldsymbol{w}) := \frac{1}{n} log(L(\boldsymbol{w})) = \frac{1}{n} \sum_{i=1}^{n} [y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)]. \tag{3.1.8}$$

Observamos que el último término es la *entropía*, función natural para maximixar la verosimilitud.

[1] ℓ en este contexto hace referencia a la log-verosimilitud

Con la nueva codificación y teniendo en cuenta que $\log(p_i) = -\log(1+e^{-\boldsymbol{w}'\boldsymbol{x}_i})$, el objetivo radica en hallar $\boldsymbol{w}$ tal que maximice:

$$\frac{1}{n}\ell(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i^*(\boldsymbol{w}'\boldsymbol{x}_i)}). \tag{3.1.9}$$

Equivalentemente el método de regresión logística busca el $\boldsymbol{w}$ que minimiza:

$$R_{logistica} := \frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i^*(\boldsymbol{w}'\boldsymbol{x}_i)}). \tag{3.1.10}$$

Sea $c > 0$. Observamos que cuando los elementos del dominio son separables $cy_i^*(\boldsymbol{w}'\boldsymbol{x}_i) > 0$ para todo i. Luego, $\text{ínf}[\log(L(\boldsymbol{w}))] = 0$ pero el mínimo no se alcanza, es decir, no existe ningún valor de $\boldsymbol{w}$ que anule la función criterio y por tanto la regresión logística falla.

Resumiendo, la regresión logística es equivalente a la minimización de riesgo empírico asociado a la **función de pérdida logística** $\ell_{log}(z) := \log(1 + e^{-z})$:

$$\ell_{log}(\boldsymbol{w}; \boldsymbol{x}, y) := \log(1 + e^{-y_i^*(\boldsymbol{w}'\boldsymbol{x}_i)}). \tag{3.1.11}$$

En contraposición a la función que define el riesgo cuando la función de pérdida es la pérdida 0-1, la función $\ell_{log}(\boldsymbol{w}; \boldsymbol{x}, y)$ es convexa con respecto a $\boldsymbol{w}$ y diferenciable. Por tanto es posible aplicar las técnicas de descenso de gradiente para hallar un minimizador y el problema es resoluble eficientemente.

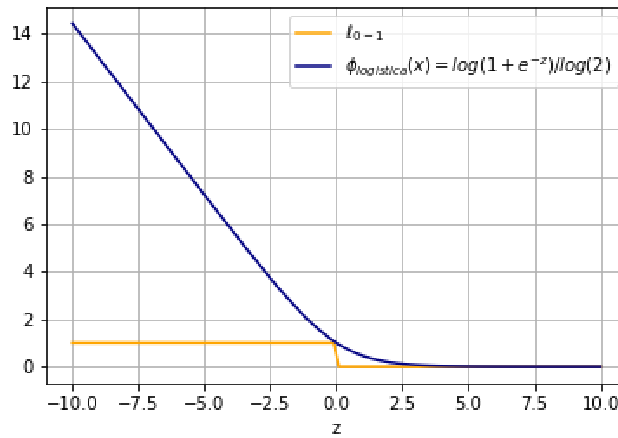


Figura 3.1.3: Función de pérdida logística en azul y función de pérdida 0-1 en naranja. Nota: la función ϕ_{log} es dividida por $\log(2)$ para conseguir $\phi_{log}(0) = 1$. Imagen: elaboración propia.

Otra posible función de activación es la función **tangente hiperbólica tanh**, (ver figura 3.1.4) donde la imagen varía entre -1 y 1.

$$tanh(z) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.1.12}$$

La ventaja frente a la función sigmoide es que para valores negativos grandes la imagen es negativa y valores cercanos a cero tienen imagen cercana a cero. Como la anterior función de activación es diferenciable y por tanto el gradiente está definido en todos sus puntos.

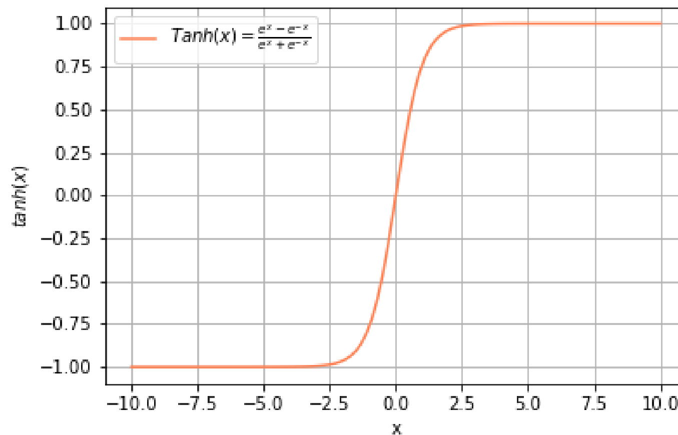


Figura 3.1.4: Función tangencial hiperbólica: diferenciable y monótona. Imagen: elaboración propia.

La función **'Rectified Linear Unit'** (ver figura 3.1.5) es una de las funciones de activación más utilizadas en la actualidad, en especial en el modelo de redes neuronales convolucionales. Su imagen es $[0, \infty)$ y actúa como una función lineal para valores positivos y no lineal para el resto:

$$\phi(z) = \text{máx}(0, z) = \begin{cases} 0 & \text{si } z \leq 0 \\ z & \text{si } z > 0 \end{cases}$$

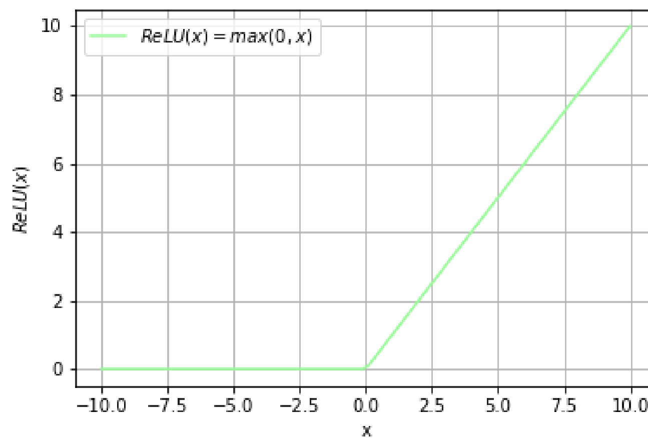


Figura 3.1.5: Función ReLU. Imagen: elaboración propia.

Es una función monótona pero no es diferenciable en el origen $z = 0$. Por tanto dicha función de activación no tiene gradiente definido en el punto $z = 0$ pero si subgradiente (ver definición 2.3.0.1), por tanto es posible usar el algoritmo de retropropagación y entrenar el modelo eficientemente (método descenso del gradiente estocástico). Es por ello que a efectos prácticos es posible utilizar $\phi(z) = \text{máx}(0, z)$ como función de activación.

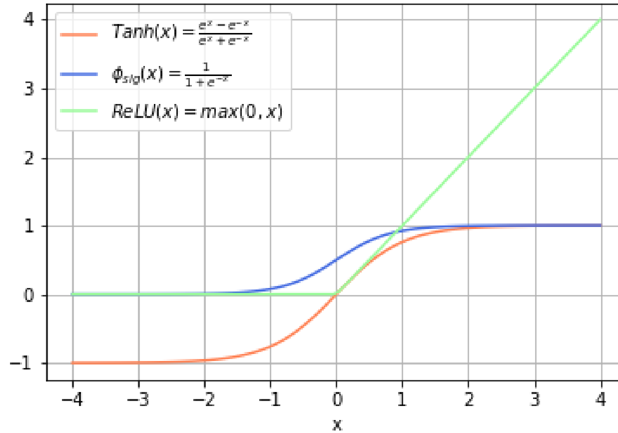


Figura 3.1.6: Funciones de activación mencionadas. Imagen: elaboración propia.

Una vez especificada la red neuronal feedforward (V, E, ϕ, w), se obtiene la función $h_{V,E,\phi,w} : R^{|V_0|-1} \to R^{|V_M|}$ donde $h_{V,E,\phi,w}(\boldsymbol{x}) = \phi(a_{M,i}(\boldsymbol{x}))$ es el valor de la capa final cuando el vector de inputs de la red es $\boldsymbol{x}$. Por ende, para el modelo de redes neuronales artificiales se define como **clase de hipótesis** al conjunto de predictores con la forma redes neuronales:

$$\mathcal{H}_{V,E,\phi} = \{h_{V,E,\phi,w} : R^{|V_0|-1} \to R^{|V_M|}; w : E \to R\},$$

donde todas las hipótesis $h_{V,E,\phi,w}$ comparten la estructura del grafo subyacente de la red y las funciones de activación pero difieren en los pesos de los nodos $w : E \to R$. A la terna (V, E, ϕ) se la conoce como *arquitectura de la red*. [[14], Sección 20.2] Dado que E es un conjunto finito, se puede ver la función de pesos como un vector $\boldsymbol{w} \in R^{|E|}$. Suponiendo que la red tiene d inputs y k outputs nodos, simplificando la notación, denotaremos $h_{\boldsymbol{w}} : R^{d+1} \to R^k$ a la función calculada por la red para la función de pesos $\boldsymbol{w}$, entonces $\ell((\boldsymbol{x}, \boldsymbol{y}), h_{\boldsymbol{w}})$ será la pérdida al predecir $h_{\boldsymbol{w}}(\boldsymbol{x})$ cuando la etiqueta es $\boldsymbol{y} \in \mathcal{Y}$.

Dada una función de pérdida ℓ, la **función de coste** será el riesgo empírico: $R_{\mathcal{S}}(\boldsymbol{w}) = E_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}_S}[\ell((\boldsymbol{x}, \boldsymbol{y}), h_{\boldsymbol{w}})] = \frac{1}{n}\sum_{i=1}^{n} \ell((\boldsymbol{x}, \boldsymbol{y}), h_{\boldsymbol{w}})$. En especial, si la función de pérdida es la pérdida logística, la función objetivo a minimizar será:

$$F(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i^* h_{\boldsymbol{w}}(\boldsymbol{x}_i)}),$$

donde $h_{\boldsymbol{w}}(\boldsymbol{x}_i)$ es la función calculada por la red para la función de pesos $\boldsymbol{w}$. De cara a aplicar las técnicas de descenso de gradiente estocástico, se debe calcular el gradiente:

$$\nabla_{\boldsymbol{w}} F(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{w}} log(1 + e^{-y_i^* h_{\boldsymbol{w}}(\boldsymbol{x}_i)}) \tag{3.1.13}$$

Para simplificar, se calcula el promedio en los lotes muestreados de tamaño B:

$$\sum_{i \in I_{Lote}} \partial_h \ell(y_i^*; h_{\boldsymbol{w}}(\boldsymbol{x}_i)) \nabla_{\boldsymbol{w}} h_{\boldsymbol{w}}(\boldsymbol{x}_i), \tag{3.1.14}$$

donde $\nabla_{\boldsymbol{w}} h_{\boldsymbol{w}}(\boldsymbol{x}_i)$ es computacionalmente costoso de obtener. En el caso concreto de la Redes Neuronales Artificiales $h_{\boldsymbol{w}}(\boldsymbol{x}_i)$ es una composición de funciones (ecuación 3.1.3), por lo que para calcular el gradiente es necesario aplicar la regla de la cadena como se estudiará en el algoritmo de retropropagación 5.

El objetivo es encontrar una hipótesis $h_{\boldsymbol{w}} \in \mathcal{H}_{V,E,\phi}$ con un riesgo empírico bajo, en el caso particular de la pérdida logísitica con $F(\boldsymbol{w})$ bajo. Sin embargo no es un problema resoluble analíticamente debido a que en general, la función de coste $F(\boldsymbol{w})$ es diferenciable pero no convexa y por ello se aplicarán métodos de optimización como el descenso del gradiente estocástico con algunas modificaciones (Algoritmo 4 [[14], Sección 20.6]). En primer lugar, inicializamos $\boldsymbol{w}$ a valores cercanos a cero ya que la inicialización al valor cero provoca que todos los nodos de las capas intermedias tengan los mismos pesos si la red está totalmente conectada. En segundo lugar, si repetimos el procedimiento SGD donde cada vez escogemos puntos aleatorios, la intuición nos dice que en alguno de las iteraciones, encontraremos un buen mínimo local. En tercer lugar, el parámetro de tamaño de paso fijo η (tasa de aprendizaje). En cuarto lugar, la salida es el mejor vector con respecto al conjunto de validación. En ocasiones, es útil añadir una regularización en los pesos (parámetro λ) con el objetivo de evitar el sobreajuste. Esto significa que minimizaremos $R_{\mathcal{S}}(\boldsymbol{w}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$. Finalmente, el gradiente de la función de pérdida para la muestra $(\boldsymbol{x}, \boldsymbol{y})$ con respecto a $\boldsymbol{w}$ no se expresa de forma cerrada, es implementado usando el algoritmo de retropropagación (número 5 [[14] Sección 20.6]) debido a que la función objetivo es composición de funciones.

Algoritmo 4: Decenso del gradiente estocástico para el modelo de redes neuronales

Entrada: grafo (V, E) y función de activación diferenciable $\phi : R \to R$

Datos: número de iteraciones τ; tasa de aprendizaje η; parámetro de regularización $\lambda > 0$

1 $\boldsymbol{w}^{(1)} \in R^{|E|}$ aleatoriamente de una distribución de tal forma que $\boldsymbol{w}^{(1)}$ está cerca del valor cero;
2 **Para** $i = 1, ...\tau$ **hacer**
3 Muestrear $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}_S$;
4 Calcular el gradiente $\boldsymbol{v}_i = \text{backpropagation}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, (V, E), \phi)$;
5 Actualizar $\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} - \eta(\boldsymbol{v}_i + \lambda \boldsymbol{w}^{(i)})$;

Resultado: $\bar{\boldsymbol{w}}$ es el $\boldsymbol{w}^{(i)}$ de mejor rendimiento para el conjunto de validación

El algoritmo de backpropagation consiste en primer lugar definir los objetos que van a ser utlizados, es decir, descomponemos $V = \dot{\cup}_{m=0}^{M} V_m$ y para cada m $V_m = \{v_{m,1}, ..., v_{m,k_m}\}$, donde $k_m = |V_m|$. Además, para cada m, la matriz $W_m \in R^{k_{m+1}, k_m}$ contiene los pesos de cada arista que conecta la capa V_m con la capa V_{m+1}, asignando el valor nulo si $(v_{m,j}, v_{m+1,i}) \notin E$. Sin pérdida de generalidad podemos asumir que $E = \cup_m (V_m \times V_{m+1})$. A continuación debemos calcular las derivadas parciales con respecto a las aristas desde V_{m-1} hasta V_m, esto es, con respecto a los elementos de W_{m-1}. Dado que están fijados el resto de los pesos de la red, la salida de los nodos de la capa V_{m-1}, $\boldsymbol{o}_{m-1}$, no dependen de los pesos en W_{m-1}. Definimos ahora como $\ell_m : R^{k_m} \to R$ la función de pérdida de la subred definida desde la capa V_m hasta la capa V_M como función de la salida de los nodos de V_m. La entrada de los nodos de la capa V_m pueden ser reescritos como $\boldsymbol{a}_m = W_{m-1}\boldsymbol{o}_{m-1}$ y la salida de los nodos como

$\boldsymbol{o}_m = \phi(\boldsymbol{a}_m)$. Obtenemos que la pérdida como función de W_{m-1} puede expresarse como $g_m(W_{m-1}) = \ell_m(\boldsymbol{o}_m) = \ell_m(\phi(W_{m-1}\boldsymbol{o}_{m-1}))$. Definimos $\boldsymbol{w}_{m-1} \in R^{k_{m-1}k_m}$ como el vector columna obtenido concatenando las filas de W_{m-1} y después transponiendo. Representamos por O_{m-1} será la matriz cuyos únicos elementos no nulos se encuentran en la diagonal con valores $\boldsymbol{o}_{m-1}^T$:

$$O_{m-1} = \begin{pmatrix} \boldsymbol{o}_{m-1}^T & 0 & \cdots & 0 \\ 0 & \boldsymbol{o}_{m-1}^T & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \boldsymbol{o}_{m-1}^T \end{pmatrix}$$

Por tanto, $W_{m-1}\boldsymbol{o}_{m-1} = O_{m-1}\boldsymbol{w}_{m-1}$ y $g_m(\boldsymbol{w}_{m-1}) = \ell_m(\phi(O_{m-1}\boldsymbol{w}_{m-1}))$. Notemos que la regla de la cadena cuando se calcula la derivada de una composición de funciones puede ser escrita en términos del Jacobiano, es decir, dadas dos funciones $\boldsymbol{f} : R^n \to R^m$ y $\boldsymbol{g} : R^k \to R^n$, el Jacobiano de la composición $(f \circ g) : R^k \to R^m$ en el punto $\boldsymbol{w}$ es $J_{\boldsymbol{w}}(f \circ g) = J_{g(\boldsymbol{w})}(\boldsymbol{f})J_{\boldsymbol{w}}(\boldsymbol{g})$. Aplicando la regla de la cadena se obtiene $J_{\boldsymbol{w}_{m-1}}(g_m) = J_{\boldsymbol{o}_m}(\ell_m)diag(\phi'(\boldsymbol{a}_m))O_{m-1}$, donde $\boldsymbol{\delta}_m = J_{\boldsymbol{o}_m}(\ell_m)$ es el gradiente de ℓ_m en $\boldsymbol{o}_m$ y es calculado de forma recursiva como $\boldsymbol{\delta}_m = \boldsymbol{\delta}_{m+1}diag(\phi'(\boldsymbol{a}_{m+1}))W_m$ sabiendo que $\boldsymbol{\delta}_T = (\boldsymbol{o}_T - \boldsymbol{y})$.

Sintetizando, calculamos la salida para cada nodo desde la capa de entrada a la capa de salida, pasando por las capas ocultas, esto es $\{\boldsymbol{a}_m, \boldsymbol{o}_m\}$ (forward). Luego calculamos los vectores $\boldsymbol{\delta}_m$ (recorriendo el grafo hacia atrás - backward) y haciendo uso de todos los vectores obtenidos, el algoritmo de retropropagación efectivamente calcula el gradiente de la función de pérdida para $(\boldsymbol{x}, \boldsymbol{y})$ con respecto al vector $\boldsymbol{w}$ [[14], Sección 20.6].

Algoritmo 5: Algoritmo de retropropagación - Backpropagation

Entrada: $(\boldsymbol{x}, \boldsymbol{y})$, vector $\boldsymbol{w}$, grafo (V, E) y la función de activación ϕ

Datos: Denotaremos por $V_0, ..., V_M$ a las capas del grafo, donde $V_m = \{v_{m,1}, ..., v_{m,k_m}\}$. Definiremos $W_{m,i,j}$ como el peso de $(v_{m,j}, v_{m+1,i})$, $(W_{m,i,j} = 0$ si $(v_{m,j}, v_{m+1,i}) \notin E)$.

1 $\boldsymbol{o}_0 = \boldsymbol{x}$;
2 **Para** $m = 1, ...M$ **hacer**
3 $\quad i \leftarrow 1$;
4 $\quad$ **Para** $i = 1, ...k_m$ **hacer**
5 $\quad\quad$ sea $a_{m,i} = \sum_{j=1}^{k_{m-1}} W_{m-1,i,j}o_{m-1,j}(x)$;
6 $\quad\quad$ sea $o_{m,i}(x) = \phi(a_{m,i})$;
7 $\boldsymbol{\delta}_M = \boldsymbol{o}_M - \boldsymbol{y}$;
8 **Para** $m = M - 1, ..,1$ **hacer**
9 $\quad$ **Para** $i = 1, ...k_m$ **hacer**
10 $\quad\quad$ $\delta_{m,i} = \sum_{j=1}^{k_{m+1}} W_{m,i,j}\delta_{m+1,j}\phi'(a_{m+1,i})$;

Resultado: Para toda arista $(v_{m-1,j}, v_{m,i}) \in E$, la derivada parcial es $\delta_{m,i}\phi'(a_{m,i})o_{m-1,j}$ (composición de funciones, donde la derivada se obtiene con la regla de la cadena)

En secciones anteriores hemos estudiado el método de descenso de gradiente y el descenso de gradiente estocástico. A continuación se describen algunas variantes del procedimiento que son utilizadas en la práctica.

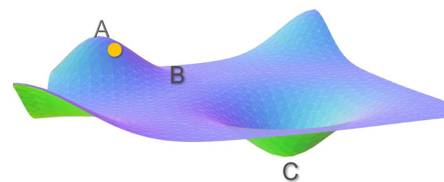


Figura 3.1.7: Fuente: [1]

Mientras que el algoritmo del descenso del gradiente estocástico es una de las técnicas más populares de optimización, la actualización de $\boldsymbol{w}$ en una iteración i se rige por la tasa de aprendizaje y el gradiente en dicha iteración. Es decir, no tiene en cuenta los pasos tomados al atravesar el espacio paramétrico. Este hecho conlleva varios inconvenientes. El primero es que el gradiente de la función objetivo en los puntos de silla es insignificante o cero, lo que a su vez conduce a pequeñas o nulas actualizaciones de $\boldsymbol{w}$ y el aprendizaje se detiene. Otro de los inconvenientes es que el entrenamiento puede ser algunas veces lento y la ruta seguida es muy cambiante. Supongamos que $\boldsymbol{w}$ inicial de la red corresponde al punto A. Con el descenso del gradiente, la función objetivo disminuye rápidamente a lo largo de la pendiente AB debido a que el gradiente es un valor grande. Por tanto, cuando llega al punto B, el gradiente es un valor pequeño y las actualizaciones alrededor de B son pequeñas. Incluso después de muchas iteraciones, la función objetivo varía lentamente antes de pararse en un punto donde el gradiente finalmente se vuelve cero. En este caso, idealmente, la función debería haberse movido al mínimo global C, pero debido a que el gradiente desaparece en el punto B, finaliza en una solución subóptima. La idea subyacente del método momentum es la siguiente: imaginemos que tenemos una pelota rodando desde el punto A. La pelota comienza a rodar lentamente y cobra algo de impulso gracias a la pendiente AB. Cuando ésta llega al punto B, ha acumulado suficiente impulso para empujarse a través de la meseta B y finalmente seguir la pendiente BC para aterrizar en el mínimo C global. Aplicando este razonamiento al problema en cuestión, es posible utilizar una media de los gradientes anteriores. En regiones donde el gradiente es alto como AB, las actualizaciones serán grandes. Por lo tanto, en cierto modo estamos cobrando impulso al tomar una media sobre estos gradientes. Desafortunadamente, considera todos los gradientes sobre iteraciones con igual ponderación. El problema se soluciona usando un promedio ponderado de los gradientes de iteraciones anteriores de tal forma que los gradientes recientes reciben más peso. Analizemos el segundo inconveniente. Con el descenso de gradiente, si la tasa de aprendizaje es demasiado pequeña, $\boldsymbol{w}$ se actualizan lentamente, por lo tanto, la convergencia lleva mucho tiempo, incluso cuando el gradiente es alto. Si la tasa de aprendizaje es demasiado alta, la función objetivo oscila alrededor de los mínimos. El descenso del gradiente con momentum da pequeños pasos en direcciones donde los gradientes oscilan y da grandes pasos a lo largo de la dirección donde los gradientes anteriores tienen la misma dirección (mismo signo). De esta forma se suavizan las oscilaciones en torno al mínimo durante la convergencia del algoritmo de descenso por gradiente. En conclusión, el método **Momentum** (Polyak, 1964), algoritmo 6 [[6], Sección 8.3.2], soluciona los problemas previamente mencionados acumulando una media exponencialmente decadente de los gradientes calculados en las iteraciones precedentes para continuar moviéndose en dicha dirección. Formalmente, el algoritmo Momentum introduce una variable $\boldsymbol{\gamma}$ que es la dirección y velocidad con la que los parámetros se mueven en el espacio paramétrico. El parámetro α determina cómo de rápido el gradiente desciende y cuanto más grande es en relación con η, más gradientes anteriores afectan la dirección actual. [1]

Algoritmo 6: SGD con momentum $\alpha \in [0, 1)$

Entrada: grafo (V, E), función de activación diferenciable $\phi : R \rightarrow R$ y parametro momentum $\alpha \in [0, 1)$

Datos: número de iteraciones τ; secuencia del tamaño de paso $\eta_1, \eta_2, ..., \eta_\tau$; parámetro de regularización $\lambda > 0$; γ_1

1 $\boldsymbol{w}^{(1)} \in R^{|E|}$ aleatoriamente de una distribución de tal forma que $\boldsymbol{w}^{(1)}$ está cerca del valor cero;

2 **Para** $i = 1, ...\tau$ **hacer**

3 Muestrear $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}_S$;

4 Calcular el gradiente $\boldsymbol{v}_i$ = backpropagation$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, (V, E), \phi)$;

5 Calcular la velocidad: $\boldsymbol{\gamma_{i+1}} \leftarrow \alpha\boldsymbol{\gamma_i} - \eta_i\boldsymbol{v}_i$;

6 Actualizar $\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} + \boldsymbol{\gamma_{i+1}}$;

Resultado: $\bar{\boldsymbol{w}}$ es el $\boldsymbol{w}^{(i)}$ de mejor rendimiento para el conjunto de validación

Root Mean Square Propagation - RMSProp (Hinton, G. 2012), algoritmo 7 [[6], Sección 8.5.2], se encuentra en el ámbito de los métodos de tasa de aprendizaje adaptativos. Pretende, al igual que Momentum y Densenso del Gradiente, suavizar las oscilaciones durante la convergencia del método. Es decir, intenta resolver el problema de que los gradientes pueden variar ampliamente en magnitudes. Algunos gradientes pueden ser pequeños y otros pueden ser muy grandes, lo que resulta un problema tratar de encontrar una sola tasa de aprendizaje global para el algoritmo. Se trata pues de un método adaptativo ya que el proceso de aprendizaje se va adaptando sobre la marcha. En vez de introducir un nuevo hiperparámetro, ajusta la tasa de aprendizaje gracias a ρ. La idea central de RMSprop, y de ahí su nombre, es mantener la media de los gradientes al cuadrado para cada iteración. Y luego dividir el gradiente por raíz cuadrada del cuadrado medio. [2] El símbolo $\odot$ representa el producto Hadamard.

Algoritmo 7: El algoritmo RMSProp

Entrada: grafo (V, E), función de activación diferenciable $\phi : R \rightarrow R$ y ρ

Datos: número de iteraciones τ; tamaño de paso η; constante δ pequeña $\approx 10^{-6}$

1 $\boldsymbol{w}^{(1)} \in R^{|E|}$ aleatoriamente de una distribución de tal forma que $\boldsymbol{w}^{(1)}$ está cerca del valor cero; $\boldsymbol{r}_0 \leftarrow 0$;

2 **Para** $i = 1, ...\tau$ **hacer**

3 Muestrear $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}_S$;

4 Calcular el gradiente $\boldsymbol{v}_i$ = backpropagation$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, (V, E), \phi)$;

5 Acumular el gradiente 'al cuadrado': $\boldsymbol{r}_{i+1} \leftarrow \rho\boldsymbol{r}_i + (1 - \rho)\boldsymbol{v}_i \odot \boldsymbol{v}_i$;

6 Actualizar $\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} - \frac{\eta}{\sqrt{\delta + \boldsymbol{r}_{i+1}}} \odot \boldsymbol{v}_i$;

Resultado: $\bar{\boldsymbol{w}}$ es el $\boldsymbol{w}^{(i)}$ de mejor rendimiento para el conjunto de validación

Por último, **Adam** (Kingma y Ba, 2014), algoritmo 8 [[6], Sección 8.5.3] es un optimizador adaptativo que combina las ideas de los algoritmos anteriores con modificaciones. En primer lugar, en Adam, el momento es incorporado directamente como una estimación del momento de primer orden (con ponderación exponencial) del gradiente, antes de calcular el gradiente. La forma más sencilla de agregar impulso a RMSProp es aplicar el impulso a los gradientes reescalados. El uso del impulso en combinación con el reescalado no tiene una motivación teórica clara. En segundo lugar, Adam incluye correcciones de sesgo a las estimaciones tanto de los momentos de primer orden como de los momentos de segundo orden para dar cuenta de su inicialización en el origen.

Algoritmo 8: El algoritmo Adam

Entrada: grafo (V, E), función de activación diferenciable $\phi : R \to R$ y ρ

Datos: número de iteraciones τ; tamaño de paso η (sugerido 0.001); constante $\delta \approx 10^{-8}$; parámetros momentum $\rho_1, \rho_2 \in [0, 1)$

1 $\boldsymbol{w}^{(1)} \in R^{|E|}$ aleatoriamente de una distribución de tal forma que $\boldsymbol{w}^{(1)}$ está cerca del valor cero; $\boldsymbol{s} \leftarrow 0, \boldsymbol{r} \leftarrow 0$;

2 **mientras** $i = 1, ...\tau$ **hacer**

3 Muestrear $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}_S$;

4 Calcular el gradiente $\boldsymbol{v}_i$ = backpropagation$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, (V, E), \phi)$;

5 Actualizar $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{v}_i$ y $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{s} + (1 - \rho_2)\boldsymbol{v}_i \odot \boldsymbol{v}_i$;

6 Corregir sesgo para el primer y segundo momento: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$, $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$;

7 Actualizar $\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} - \eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}+\delta}}$;

Resultado: $\bar{\boldsymbol{w}}$ es el $\boldsymbol{w}^{(i)}$ de mejor rendimiento para el conjunto de validación

3.2. Redes neuronales convolucionales

Se denominan *redes neuronales convolucionales, o convolutional neural networks (CNN) en la literatura inglesa*, a redes neuronales utilizadas para el procesamiento de datos con una topología similar a una cuadrícula, que utilizan la operación lineal *convolución* en vez de la multiplicación por una matriz general, en al menos una capa. Como ejemplos podemos incluir series de tiempo que serán vistos como matrices de una dimensión discretizando las muestras, o imágenes que se considerarán como matrices de dos dimensiones.

Las redes neuronales convolucionales han jugado un papel importante en la historia del aprendizaje profundo. Fueron algunos de los primeros modelos en funcionar bien, mucho antes de que se considerara que los modelos de aprendizaje profundo fuesen viables; además de resolver importantes aplicaciones comerciales y permanecer a la vanguardia en este enfoque del aprendizaje profundo. Asimismo, han sido una de las primeras redes profundas entrenadas con el algoritmo de retropropagación.

La bibliografía de la sección 3.2 a excepción de la subsección 3.2.3 es [[6], Capítulo 9].

3.2.1. La operación convolución

La *convolución* es una operación aplicada a dos funciones con argumentos reales. Véase,

$$s(t) := (x * w)(t) = \int_{-\infty}^{\infty} x(a)w(t-a)\,da.$$

Dado que generalmente se trabajará con datos en el ordenador, el tiempo está discretizado, y por tanto, es coherente pensar que el tiempo t solo toma valores enteros. En este caso, se hablará de convolución discreta

$$s(t) := (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$

Hablando coloquialmente, estamos calculando un promedio ponderado por pesos no negativos w, esto es, $w(t-a)$ se anula o es un valor cercano a cero si el argumento es grande. En resumen, estamos promediando valores próximos a t [[6], Sección 9.1].

En este trabajo se utilizará la terminología aceptada en redes convolucionales, es decir, denominaremos *input* al primer argumento, que en nuestro caso es la función x que normalmente será un vector multidimensional de datos; mientras que llamaremos *kernel* al segundo argumento, la función w, que será un vector multidimensional de parámetros que se adaptan con el algoritmo. Si los datos de entrada son imágenes, esto es, los inputs son matrices de dos dimensiones I, entonces en general nos interesará usar un kernel K de dos dimensiones también, obteniendo:

$$S(i,j) := (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

La operación de convolución presenta la propiedad de conmutatividad, siendo posible escribir $S(i,j) = (I * K)(i,j) = (K * I)(i,j)$. Si bien, generalmente, las librerias de redes neuronales implementan la función conocida como *cross-correlation*, que no es otra que la convolución siguiente:

$$S(i,j) := (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

[[6], Sección 9.1]

Hemos de tener en cuenta que cuando hablamos de convolución en el contexto de redes neuronales y en particular en el tratamiento de imágenes, no es exactamente la convolución discreta estándar anteriormente definida, sino que consiste en numerosas aplicaciones de la convolución (cross correlation) en paralelo, con la posibilidad de usar diferentes kernels, que harán la función de filtros, extrayendo dependiendo de sus valores unas características u otras. Fundamentalmente, la operación de convolución consiste en superponer un filtro sobre una imagen e ir deslizando el filtro sobre la misma hasta haber recorrido así todas las regiones de la imagen (todos los píxeles de la imagen han de haber sido 'visitados' por el filtro en algún momento durante el proceso, figura 3.2.1). Por cada desplazamiento del filtro sobre la imagen se calcula el valor resultante de la suma de todas las multiplicaciones de cada píxel por el valor correspondiente a la posición del filtro que superpone a ese píxel. De esta forma se construye el mapa de características: cada vez que se calcula un nuevo resultado se almacena en la siguiente posición de la matriz resultado. En una convolución sobre imágenes

de tres dimensiones, el filtro a aplicar también se alinea a través de esa tercera dimensión con la imagen. El desplazamiento de la ventana de convolución (el filtro superpuesto sobre la imagen) por la imagen se puede configurar y a esta variable se le llama *stride*, pudiendo ser una unidad, dos, etc. [[6], Sección 9.2]

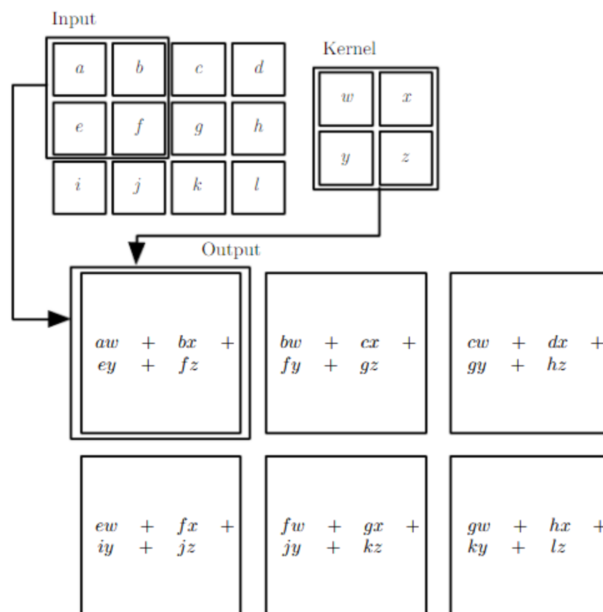


Figura 3.2.1: Ejemplo de una convolución de dos dimensiones sin trasponer el kernel. Fuente: [[6], Sección 9.2]

La convolución aprovecha tres ideas importantes que pueden ayudar a mejorar un sistema de aprendizaje automático, descritas a continuación. [[6], Sección 9.2] En primer lugar, contrariamente a lo que sucedía en las capas de redes neuronales tradicionales, donde cada unidad de salida de una capa interactúa con cada unidad de entrada de la siguiente; las redes convolucionales al no usar la multiplicación matricial clásica, consiguen, siempre que el tamaño del kernel sea menor que el tamaño del input, un número menor de interacciones. Esto significa que necesitamos menos operaciones para calcular la salida y almacenar menos parámetros, lo que reduce los requisitos de memoria del modelo y mejora su eficiencia estadística. Este procedimiento es conocido con el nombre de **iteraciones dispersas (sparse interactions)**. Concretamente, si hay m entradas y n salidas, entonces la multiplicación de matrices requiere $m \cdot n$ parámetros, luego los algoritmos tienen tiempo de ejecución $O(m \cdot n)$. Si limitamos el número de conexiones que cada output tiene a k, el tiempo de ejecución se reduce a $O(k \cdot n)$, y esto

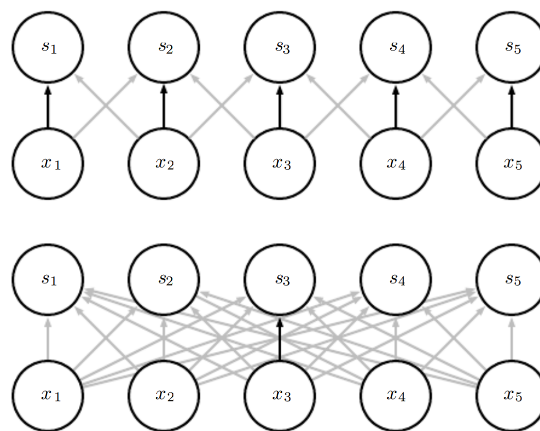


Figura 3.2.2: Parámetros compartidos, [[6], Sección 9.2]

nos permitirá obtener un buen rendimiento siempre que k sea varios órdenes de magnitud más pequeños que m. Por ejemplo, al procesar una imagen, la imagen de entrada puede tener miles o millones de píxeles, pero podemos detectar características pequeñas y significativas, como bordes con kernels que ocupan solo decenas o cientos de píxeles.

Otra de las ideas aprovechada por las redes convolucionales es la conocida como **parámetros compartidos (parameter sharing)**. Esto es, usar el mismo parámetro para más de una función en nuestro modelo; contrariamente a las redes neuronales tradicionales, donde cada elemento de la matriz es usado una única vez para obtener el output correspondiente de la capa. En las redes neuronales convolucionales, cada elemento del kernel es usado con casi todos los elementos del input, lo que se traduce en que el algoritmo no debe aprender un conjunto de parámetros para cada localización, sino solo un conjunto de parámetros. Este avance no afecta al tiempo de ejecución, pero reduce los requisitos de almacenamiento del modelo a k parámetros. Como ejemplo de estos dos primeros principios observamos la figura 3.2.2 que son dos modelos diferentes donde las flechas indican las conexiones entre nodos. *(Figura de arriba - convolución con kernel de tamaño 3)* Notamos que cuando s es ejecutado, solo recibe información de tres inputs, de la misma forma que solamente tres outputs son afectados por un solo x. *(Figura de abajo - Modelo sin parámetros compartidos)* Por el contrario, cuando s es ejecutado, todos los pesos de los inputs son relevantes; de la misma manera que todos los outputs son afectados por todos los x.

En el caso de la convolución, la forma particular de compartir parámetros hace que la capa tenga una propiedad llamada equivariancia a la translacción o **representaciones equivariantes (equivariant representations)**. Una función equivariante equivale a decir coloquialmente que, si la entrada cambia, la salida cambia de la misma forma; es decir, f es equivariante a g sii $f(g(x)) = g(f(x))$.

En suma, la operación de convolución es significativamente más eficiente que la multiplicación de matrices densas en términos de los requisitos de memoria y eficiencia estadística. [[6], Sección 9.2]

3.2.2. Componentes de una red neuronal convolucional

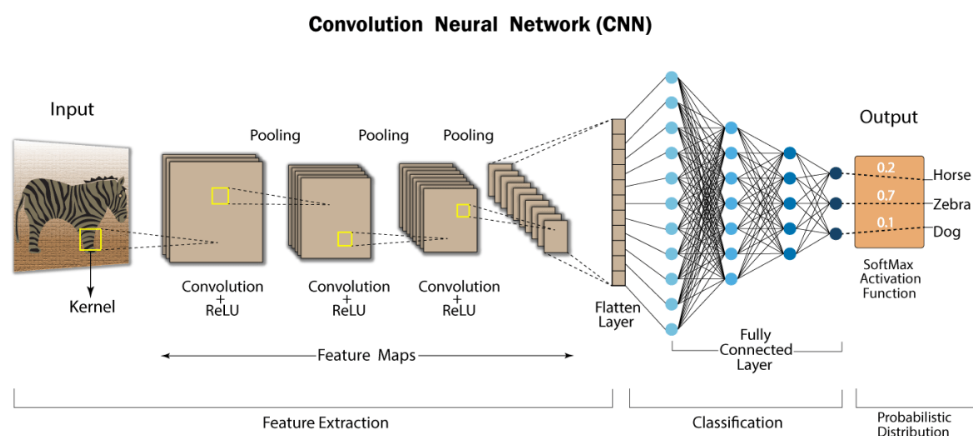


Figura 3.2.3: Esquema de una red neuronal convolucional cualquiera. Fuente: enlace

Cualquier red neuronal convolucional se estructura en dos etapas bien diferenciadas (ver figura 3.2.3). En primera instancia, dentro de la etapa denominada **feature selection**, se encuentra la capa de entrada (input), donde se recoge la dimensión de la imagen y mantiene los valores de los píxeles inalterados y la conocida como capa convolucional donde se procesará la imagen, explicada con más detalle posteriormente. A continuación se encuentra la segunda etapa denominada **clasificación** que consiste en capas ocultas totalmente conectadas. Finalmente, la capa de salida dará como resultado las puntuaciones de cada clase y realizará el proceso de clasificación.

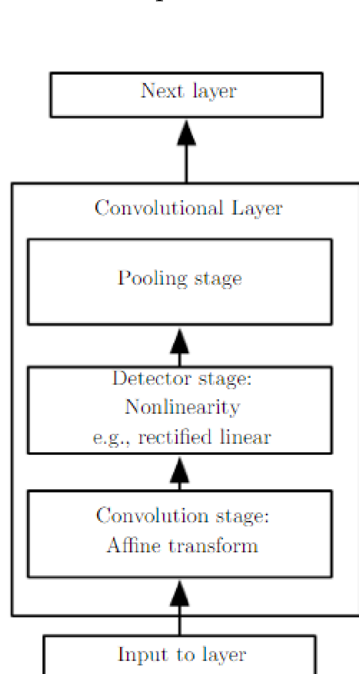


Figura 3.2.4: [[6], Figura 9.7]

Una **capa convolucional** tiene tres etapas definidas (figura 3.2.4). La primera ejecuta varias convoluciones en paralelo para producir un conjunto de activaciones lineales. En la segunda etapa, conocida como Detector Stage, cada activación lineal pasa a través de una función de activación no lineal. Y en última instancia, se modifica aún más la salida de la capa gracias a la función comúnmente denominada *pooling function*, que replaza el output en una ubicación de la red con una estadística resumida de los resultados cercanos. Un ejemplo conocido de función de agrupación es *max pooling (Zhouand Chellappa, 1988)* que escoge el output de valor más grande dentro de un límite rectangular. Debido a que la operación de convolución puede disminuir el tamaño de la imagen, la técnica *relleno cero (zero-padding)* nos permite conservar el tamaño de entrada original. Consiste en agregar un borde de píxeles todos con valor cero alrededor de los bordes de la imagen de entrada.

Cabe destacar que cuando deseamos determinar si la imagen contiene una cara, necesitamos conocer si hay un ojo en cada lado de ésta, pero no es importante la localización dentro de la imagen del ojo. Es decir, es importante solamente saber si está presente alguna característica en la imagen, pero no donde se encuentra. En estos casos, una propiedad útil y lograda gracias a la función de agrupación (average pooling), será la conocida como invarianza a la translacción, véase, si cambiamos el valor del input ligeramente, el valor de la mayoría de los outputs agrupados no cambian. En otros contextos, es más importante conservar la ubicación de una característica. Por ejemplo, si queremos encontrar una esquina definida por dos aristas en una orientación específica, debemos preservar la ubicación de las aristas lo suficientemente bien como para probar si se encuentran. [[6], Sección 9.3]

Conviene enfatizar la gran ventaja con respecto a otros tipos de redes neuronales de las redes convolucionales, esto es, pueden trabajar con entradas de distinta dimensión. En el caso particular del procesamiento de imágenes, si consideramos una colección de ellas donde cada una tiene diferente ancho y largo, es decir, el input tiene tamaño variable, aplicando el kernel un número diferente de veces dependiendo del tamaño de la entrada y la salida, se resuelve el problema. Estos tipos de entrada no pueden ser representados por redes neuronales tradicionales, basadas en la multiplicación de matrices.

3.2.3. Arquitecturas propuestas para el proyecto

La **arquitectura LeNet5 (Yann LeCun, 1990)** es una de las primeras aplicaciones de redes neuronales convolucionales desarrolladas con éxito. [10] Ha sido el origen de gran parte de las arquitecturas modernas, y una inspiración para muchos investigadores de este campo. LeNet-5 toma una imagen de entrada de tamaño 32×32 píxeles. Cada capa, excepto la última, es seguida por la función de activación *tanh*.

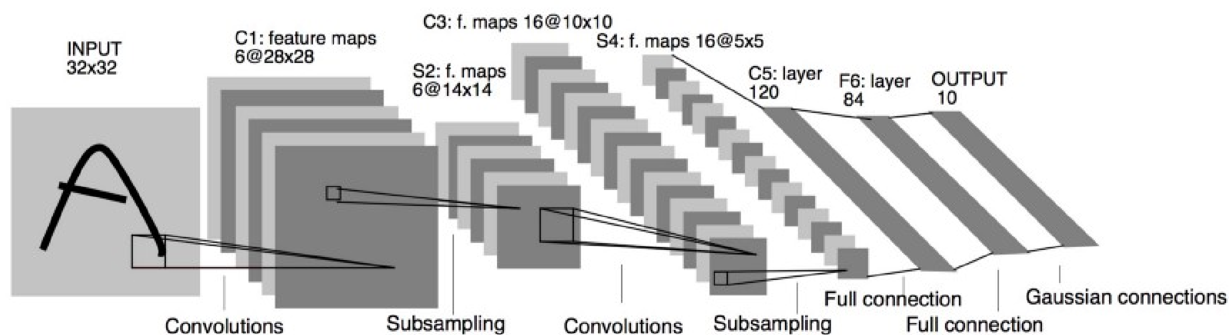


Figura 3.2.5: Arquitectura LeNet-5, inicialmente utilizada para leer códigos postales y dígitos (ofrecido en [9]).

Se puede resumir en los siguientes puntos:

- C1: La primera capa convolucional. Esta consiste en seis kernels convolucionales de tamaño 5×5. C1 emite seis imágenes de tamaño 28×28. La primera capa de una red neuronal convolucional normalmente identifica características básicas como bordes y esquinas.

- S2: Una capa de submuestreo, también conocida como capa de agrupación promedio (average pooling). Cada cuadrado de cuatro píxeles en la salida C1 se promedia a un solo píxel. S2 reduce las seis imágenes de 28×28 con un factor de 2, produciendo seis imágenes de salida de tamaño 14×14.

- C3: La segunda capa convolucional. Esta consta de 16 kernels convolucionales, cada uno de tamaño 5×5, que toman las seis imágenes de 14×14, produciendo 16 imágenes de tamaño 10×10.

- S4: La segunda capa de agrupación promedio (average pooling). S4 reduce las 16 imágenes de 10×10 a 16 imágenes de 5×5.

- C5: Una capa convolucional totalmente conectada con 120 salidas. Cada uno de los 120 nodos de salida está conectado a todos los 400 nodos ($5 \times 5 \times 16$) que provienen de S4. En este punto, la salida ya no es una imagen, sino una matriz 1D de longitud 120.

- F6: Una capa totalmente conectada que conecta los 120 nodos a una nueva capa de longitud 84.

- Capa final: Una función softmax que transforma la salida de F6 en una distribución de probabilidad de 10 valores que suman 1.

La **arquitectura VGGnet (Visual Geometry Group)** es un tipo de red neuronal convolucional, propuesta por Karen Simonyan and Andrew Zisserman en su paper *Very Deep Convolutional Networks for Large-Scale Image Recognition [15]*. Podemos diferenciar dos conocidas arquitecturas (ver figura 3.2.6) dependiendo del número de capas, esto es, VGG-16 (16 capas, 13 de ellas de convolución, 2 de ellas totalmente conectadas y la capa final una softmax para clasificar) o VGG-19 (19 capas, 16 de ellas de convolución, 2 de ellas totalmente conectadas y la capa final una softmax para clasificar). Algo que demuestran estas redes neuronales es que la profundidad de una red neuronal es un parámetro crítico. Desafortunadamente, estas redes tienen muchos parámetros en el momento de entrenamiento.

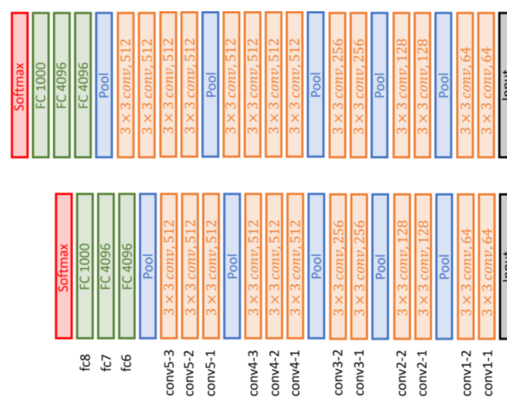


Figura 3.2.6: Arquitectura Vgg-16 (abajo) y Vgg-19 (arriba) Enlace.

Trabajos recientes han demostrado que las redes convolucionales pueden ser sustancialmente más profundas, más precisas y eficientes para entrenar si contienen conexiones más cortas entre las capas cercanas a la entrada y las cercanas a la salida. Desafortunamente, a medida que las redes neuronales convolucionales son más profundas, surge un problema, a saber, debido a que la información sobre la entrada o el gradiente pasa a través de muchas capas, ésta puede desaparecer en gran medida cuando llega al final (o al comienzo) de la red. Teniendo en mente esta observación introducimos la **arquitectura Redes Convolucionales Densamente Conectadas (DenseNet)**[8], la cual para garantizar el máximo flujo de información entre las capas de la red, conecta todas las capas (con tamaños de features coincidentes) directamente entre sí. Para preservar el avance (feedforward) natural, cada capa obtiene entradas adicionales de todas las capas anteriores y pasa la salida a todas las capas posteriores.

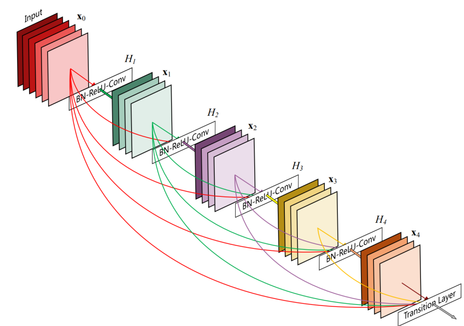


Figura 3.2.7: Arquitectura DenseNet, [8]. Las capas entre dos bloques adyacentes se denominan capas de transición, las cuales modifican el tamaño a través de la convolución y la agrupación.

Como se observa en la figura 3.2.7, la capa l-ésima tiene l entradas, que son las salidas de los bloques convolucionales anteriores y sus características (features) se transmiten a todas las $L - l$ capas posteriores. Por tanto, para una red con L capas, se obtienen $\frac{L(L+1)}{2}$ conexiones, en lugar de L, como en arquitecturas tradicionales.

Sea $\mathbf{x}_0$ una imagen que atraviesa una red convolucional con M capas, cada una de las cuales implementa una transformación no lineal ϕ_m (m es el indice de la capa) compuesta de operaciones como la normalizacion por lotes (Batch Normalization), ReLU, agrupación (pooling) o convolución. La salida de la capa m se denotará por $\mathbf{o}_m$. En una red convolucional tradicional el output de la capa m es el input de la capa $m + 1$ como se vio, obteniendo $\mathbf{o}_m = \phi_m(\mathbf{o}_{m-1})$.

- **Deep Residual Networks - ResNets**: añaden una conexión que omite las transformaciones no lineales mediante la función de identidad, esto es,

$$\mathbf{o}_m = \phi_m(\mathbf{o}_{m-1}) + \mathbf{o}_{m-1}. \tag{3.2.1}$$

  Una ventaja es que el gradiente puede fluir directamente a través de la función de identidad desde las capas posteriores a las capas anteriores. Sin embargo, la función de identidad y la salida de ϕ se combinan mediante una suma, lo que puede impedir el flujo de información en la red. [7]

- **DenseNets**: para mejorar aún más el flujo de información entre capas, proponemos un patrón de conectividad diferente (connectivity pattern), esto es, introducimos conexiones directas desde cualquier capa a todas las capas posteriores (figura 3.2.7). Escrito en forma de ecuaciones, resulta que la capa m recibe como entradas las características de las capas precedentes (feature-maps) $\mathbf{x}_0, \mathbf{o}_1, ..., \mathbf{o}_{m-1}$, resultando

$$\mathbf{o}_m = \phi_m([\mathbf{x}_0, \mathbf{o}_1, ..., \mathbf{o}_{m-1}]), \tag{3.2.2}$$

  donde $[\mathbf{x}_0, ..., \mathbf{o}_{m-1}]$ es la concatenación de las salidas de las capas precedentes. Sin embargo, la concatenación en (3.2.2) no es válida cuando varian los tamaños, es por ello que añadimos capas de transición, definiendo $\phi_m(\cdot)$ como composición la normalizacion por lotes (Batch Normalization), ReLU y 3×3 convolución (ver figura 3.2.8). Superficialmente DenseNets y ResNets son muy similares en el sentido de que las entradas en vez de sumarse, se concatenan, pero tienen comportamientos diferentes las dos arquitecturas.

- **Hiperparámetro tasa de crecimiento**: si cada función phi_m produce k salidas, entonces la capa m tiene $k_0 + k(m - 1)$ entradas. Esto produce que puedan existir capas estrechas con por ejemplo $k = 12$. Está demostrado que una tasa de crecimiento relativamente pequeña es suficiente para obtener resultados buenos para ciertos conjuntos de datos [8].

En definitiva, la arquitectura redes convolucionales densamente conectadas presentan varias ventajas, como pueden ser: alivian el problema del gradiente, fortalecen la propagación de las características de la imagen, fomentan la reutilización de dichas características y reducen sustancialmente el número de parámetros.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

Figura 3.2.8: Arquitectura DenseNet para el conjunto de datos ImageNet [8]. Para nuestro conjunto de datos la última capa solo contiene una unidad.

3.3. Conclusiones

Existen métodos de clasificación de uso común basados en fronteras lineales. La clasificación logística es uno de ellos. Cuando es posible separar los elementos por un hiperplano, la clasificación logística falla porque no se alcanza el óptimo de la función criterio. En ese caso un algoritmo que puede ser implementado de manera eficiente en la práctica es *Perceptron*. En ocasiones, es necesario buscar fronteras más complejas geométricamente; en otras palabras, necesitamos más flexibilidad en la clase de hipótesis. Este problema motiva el estudio del modelo de redes neuronales artificiales, uno de los más utilizados en la práctica.

Una vez definido el espacio de hipótesis, la finalidad es hallar una red neuronal con mímino riesgo empírico y para ello se aplica, con pequeñas modificaciones, el método de descenso de gradiente estocástico. A su vez, son presentados tres algoritmos con pequeñas correcciones de éste, de los cuales no se posee aún fundamentos teóricos que justifiquen su buen comportamiento en la práctica.

Se finaliza el capítulo con una incursión en las redes neuronales convolucionales aplicadas a la clasificación de imágenes. La idea subyacente del modelo es extraer características aprovechando la estructura bidimensional de las imágenes y la operación de convolución que promedia los valores de píxeles cercanos.

CAPÍTULO 4

Estudio experimental

En este capítulo se presentan los experimentos realizados para ilustrar los modelos de redes neuronales convolucionales aplicado a un conjunto de datos en concreto.

4.1. Objetivos

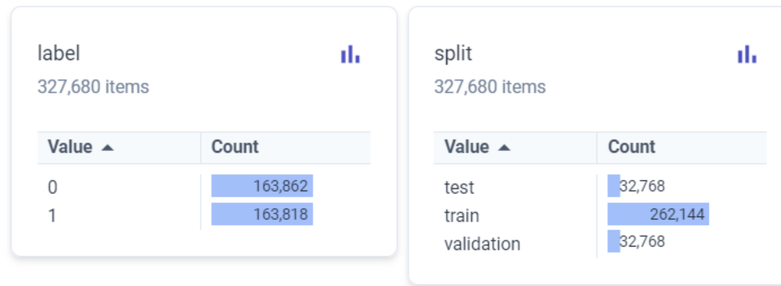
El propósito es la clasificación binaria de imágenes usando como modelo de aprendizaje automático las redes neuronales convolucionales, esto es, dada una imagen como entrada, construir un modelo que devuelva una clase o una la probabilidad con la que cada clase describe la imagen.

4.2. Estudio del dataset

El conjunto de imágenes utilizado en este projecto es un dataset perteneciente a *Tensorflow Datasets* (enlace al conjunto de datos) conocido como **Patch camelyon**. Consta de 327.680 imágenes médicas en color de 96×96 pixels donde cada una ha sido extraída de exploraciones histopatológicas de secciones de ganglios linfáticos. La etiqueta de cada imagen es un número binario que indica la presencia de tejido metastásico.

- **Imágenes:** El conjunto de datos se divide en un conjunto de entrenamiento de 262.144 (2^{18}) ejemplos, y un conjunto de validación con otro de prueba de 32.768 (2^{15}) ejemplos respectivamente. No hay superposición entre las divisiones, y todas las divisiones tienen un equilibrio de 50/50 entre ejemplos con etiqueta 0 y 1 respectivamente.

- **Etiquetas:** La etiqueta 1 indica que la región central de 32×32 pixels de la imagen contiene al menos un píxel de tejido tumoral. El tejido tumoral en la región externa de la subimagen no influye en la etiqueta. Esta región exterior se proporciona para permitir el diseño de modelos totalmente convolucionales que no utilizan ningún relleno cero, para garantizar un comportamiento adecuado cuando se aplica a una imagen completa.



label
327,680 items

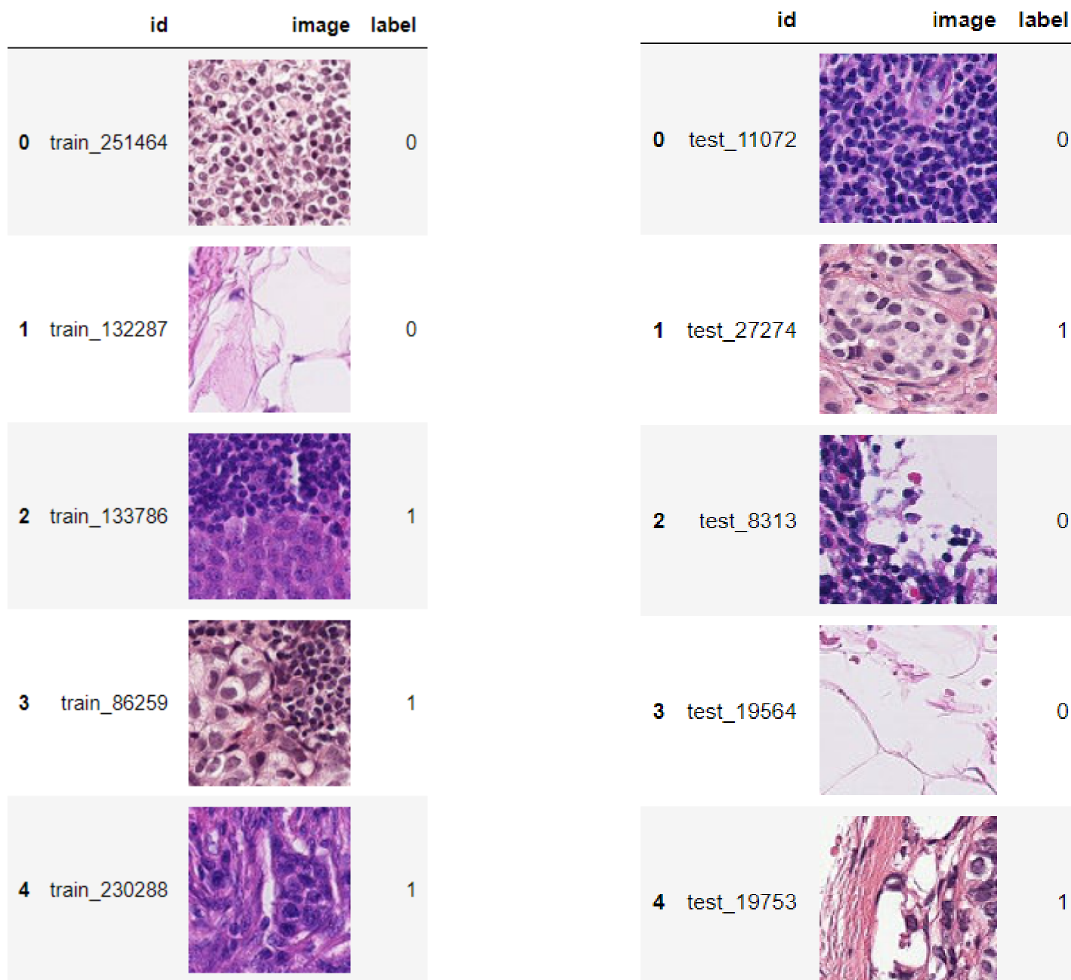
Value	Count
0	163,862
1	163,818

split
327,680 items

Value	Count
test	32,768
train	262,144
validation	32,768

Figura 4.2.1: https://github.com/basveeling/pcam

En primer lugar, visualizamos el conjunto de datos, es decir, algunas imágenes aleatorias del conjunto de datos patch camelyon.



	id	image	label
0	train_251464		0
1	train_132287		0
2	train_133786		1
3	train_86259		1
4	train_230288		1

(a) Imágenes del conjunto de entrenamiento

	id	image	label
0	test_11072		0
1	test_27274		1
2	test_8313		0
3	test_19564		0
4	test_19753		1

(b) Imágenes del conjunto denominado test set

Figura 4.2.2: Ejemplos de los dos conjuntos con la etiqueta asociada correspondiente.

4.3. Descripción metodología y especificaciones

1. Carga el conjunto de datos desde TensorFlow
2. Examen y comprensión del conjunto de datos
3. Procesamiento de los datos
4. Creación del modelo de aprendizaje automático
5. Entrenamiento del modelo de aprendizaje automático gracias al conjunto de entrenamiento
6. Prueba del modelo gracias al conjunto de prueba (test set)
7. Mejora del modelo

4.3.1. Hardware y Software utilizado en el proyecto

En esta subsección serán vistos los principales componentes del ordenador con el que se ha trabajado y las diferentes herramientas de software que han sido utilizadas. A lo largo del desarrollo del proyecto se han utilizado tan solo un **dispositivo hardware**: un ordenador portátil propio. Los componentes del ordenador portátil, de la marca HP, son los siguientes: Procesador Intel Core i5-1035G1, Disco duro: 1 terabyte, Memoria RAM 8 GB. Cabe destacar, que con este portátil que es de gama media, se ha conseguido llevar a cabo el entrenamiento y la comprobación del buen funcionamiento de la red neuronal en horas aproximadamente. Esto es un claro indicador del buen desempeño del programa creado: con un ordenador de sobremesa potente el entrenamiento y posterior validación sería cuestión de horas.

El **software** que utilizaremos en el desarrollo de este Trabajo Final de Grado es el que se muestra a continuación. El sistema operativo es Windows 10. El lenguaje que usaremos es Python que es sin duda uno de los lenguajes más utilizados en el Aprendizaje Automático si no el que más. Es un lenguaje de programación interpretado y multiparadigma, ya que soporta orientación a objetos, programación imperativa y también programación funcional. El entorno de trabajo es Anaconda, distribución libre y abierta para los lenguajes Python y R. Entre sus principales paquetes encontramos las principales librerías, APIs que se usaran a lo largo del proyecto como pueden ser TensorFlow, Keras, Numpy, etc... El código se encuentra en Jupyter Notebooks. Los "Notebook" contienen una lista de celdas de entrada y salida ordenadas, es decir cada celda contiene las líneas de código, esto a priori no parece mucho, pero permite la ejecución de partes de código sin necesidad de ejecutar todo el código completo de la aplicación que estamos realizando.

4.4. Estudios piloto

Como primer experimento piloto se ha construido el modelo de red neuronal convolucional **LeNet-5** (ver apartado 3.2.3) utilizando como conjunto de datos el conjunto comúnmente conocido como **conjunto MNIST**. Es un conjunto de imágenes de dígitos escritos a mano del

0 al 9 de tamaño 28×28 píxeles, formado por 60000 imágenes en el conjunto de entrenamiento y otras 10000 en el conjunto de validación. Es importante destacar que cada imagen del conjunto de datos MNIST tiene un tamaño de 28×28 píxeles, por lo que utilizaremos las mismas dimensiones para la entrada LeNet-5 en lugar de 32×32 píxeles.

Los pasos seguidos han sido: importación tanto de librerías necesarias como del conjunto de datos; procesamiento de las imágenes para poder ser utilizadas en el modelo (concretamente, cada píxel de cada imagen inicial en el conjunto de datos varía de 0 a 255, por tanto, escalaremos el valor a un float entre 0 y 1 y se codificará la salida, es decir, los valores numéricos entre 0 y 9 son transformados en vectores binarios con solo uno de sus elementos distinto de zero); construcción de la red neuronal convolucional; entrenamiento del modelo gracias al conjunto de entrenamiento (50 iteraciones). Cabe destacar que, como medida de rendimiento, ya que es un problema de clasificación multiclase no se ha utilizado la entropía cruzada binaria sino la entropía cruzada multiclase. Los resultados obtenidos son [0.09107204526662827, 0.9829999804496765] como pérdida y precisión respectivamente para el conjunto de prueba o validación. Es decir, el modelo predice correctamente el 98.2999980 % de las veces, pero cuando comete errores, el valor predicho y el verdadero pueden ser muy distantes uno de otro. (Ver Anexos para el código en Python)

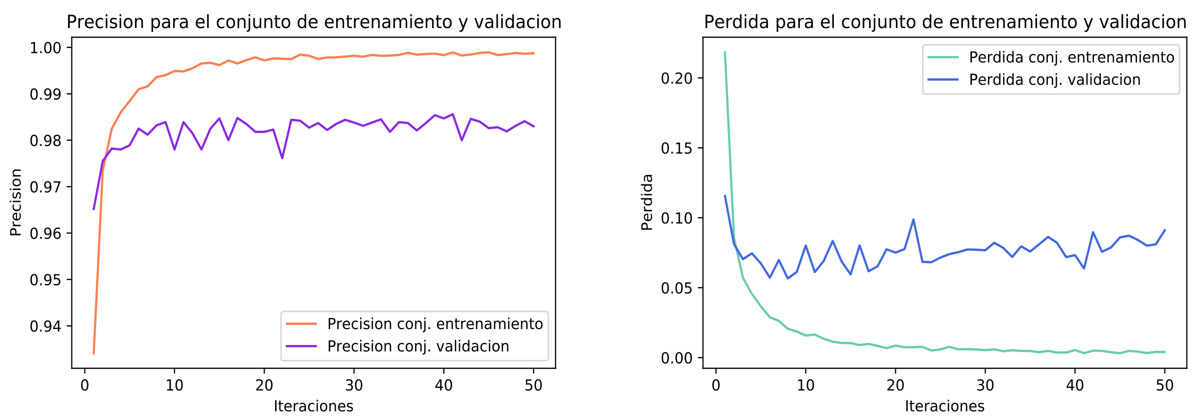


Figura 4.4.1: Precisión y pérdida obtenida con la arquitectura LeNet-5 para el conjunto de datos MNIST.

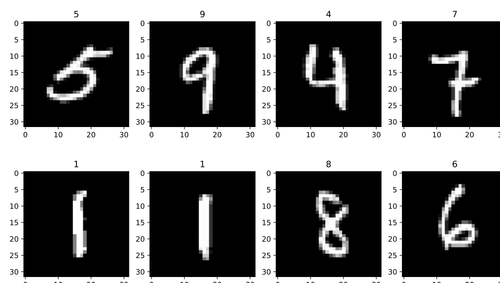


Figura 4.4.2: Prediciones para el conjunto de validación después del entrenamiento del modelo.

Como segundo experimento piloto se ha construido como modelo de aprendizaje automático una **red neuronal totalmente conectada** utilizando como conjunto de datos el conjunto **Patch Camelyon** explicado en la sección 4.2. Los pasos seguidos son: importación tanto de librerías necesarias como del conjunto de datos; procesamiento de las imagenes para poder ser utilizadas en el modelo (concretamente, cada píxel de cada imagen inicial en el conjunto de datos varía de 0 a 255, por tanto, escalaremos el valor a un float entre 0 y 1, aún más, dado que la entrada de la red neuronal escogida debe ser un vector y no una imagen, debemos convertir la imagen de tamaño $96 \times 96 \times 3$ en un vector de tamaño 27648 y por último se separa en dos variables la imagen y la etiqueta); construcción de la red neuronal totalmente conectada (FCNN), es decir, la arquitectura es tal que todos los nodos en una capa están conectados a los nodos en la siguiente capa. En particular, la primera y segunda capa consiste en 864 nodos, la tercera 432, le siguen dos capas de 54 y 27 respectivamente para llegar a la última con un solo nodo. Las funciones de activaciones escogidas son 'relu' para todas las capas, excepto para la última que se aplica la sigmoide. En total, el modelo consta de 25.034.671 parámetros entrenables. Si bien este tipo de algoritmo se aplica comúnmente a algunos tipos de datos, en la práctica este tipo de red tiene algunos problemas en términos de reconocimiento y clasificación de imágenes. Tales redes son computacionalmente intensas. Después de compilar el modelo con las especificaciones de optimizador *'sgd'*, función de pérdida *entropía cruzada*, número de iteraciones 30, obtenemos [perdida_test, precisión_test] = [0.6667577624320984, 0.55859375]. Los resultados son bastante malos ya que clasifica bien solo el 56 % de las veces aproximadamente. Estos resultados motivan la utilización de modelos como las redes neuronales convolucionales cuando los datos son imágenes. (Ver Anexos para el código en Python)

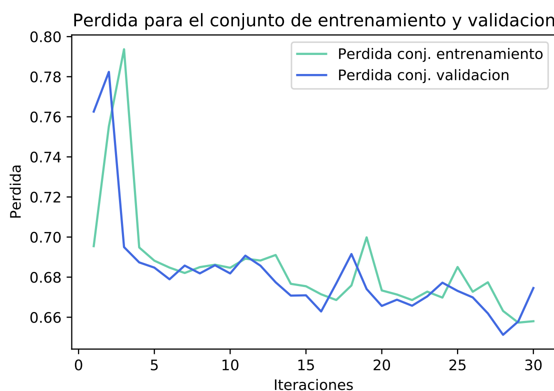
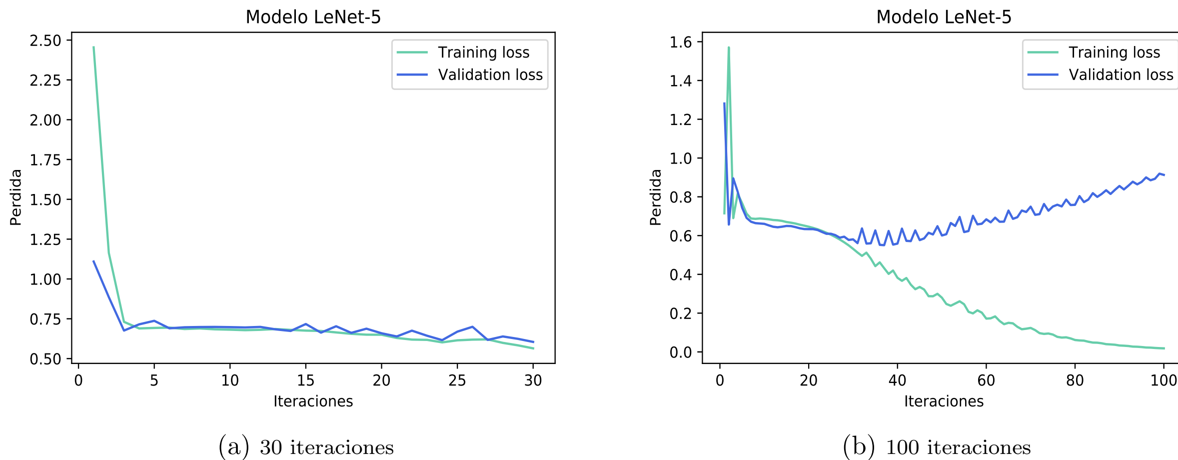


Figura 4.4.3

Como tercer y último experimento piloto se ha construido como modelo de aprendizaje automático una **red neuronal convolucional LeNet-5** utilizando como conjunto de datos el conjunto **Patch Camelyon** explicado en la sección 4.2. Los pasos seguidos son: importación tanto de librerías necesarias como del conjunto de datos; procesamiento de las imagenes para poder ser utilizadas en el modelo (concretamente, cada píxel de cada imagen inicial en el conjunto de datos varía de 0 a 255, por tanto, escalaremos el valor a un float entre 0 y 1, aún más, separamos la imagen y la etiqueta en diferentes variables); construcción de la red neuronal convolucional LeNet-5. El modelo LeNet-5 consta de dos capas convolucionales de

6 y 16 filtros respectivamente, cada uno de tamaño 5×5; junto con cuatro capas totalmente conectadas de 9216, 120, 84 y 1 nodos respectivamente. Además, en todas las capas se aplica la técnica de relleno cero. El algoritmo de entrenamiento escogido es 'Adam', dando lugar a 68.75 % de precisión después de 100 iteraciones y 55.46875 % de precisión después de 30 iteraciones.



(a) 30 iteraciones

(b) 100 iteraciones

Figura 4.4.4: Comparación de la pérdida obtenida después de entrenar 30 y 100 veces respectivamente.

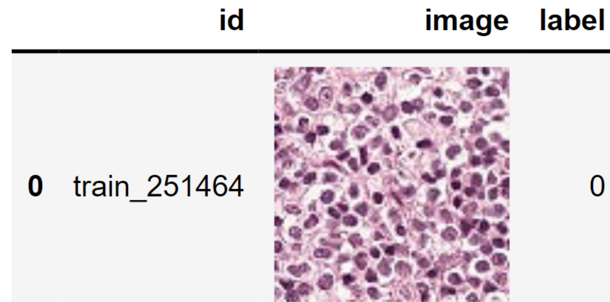
Como se vio en la teoría, aumentar el número de iteraciones puede disminuir considerablemente el error de aproximación pero aumentar el error de estimación y produciendo que no se encuentre balanza entre ambos (sobreajuste). En este ejemplo concreto, realizando 30 iteraciones conseguimos una mejor balanza entre ambos errores consiguiendo un valor de precisión más elevado (figura 4.4.4). Los resultados obtenidos nos dicen que el modelo aproximadamente el 69 % de las veces clasifica bien para una imagen no perteneciente al conjunto de entrenamiento. Dado que es una arquitectura sencilla inventada para el reconocimiento de dígitos con pocos parámetros obtenemos resultados no muy satisfactorios debido a que el conjunto de datos Patch camelyon es más complejo que MNIST. Es por ello que en el siguiente apartado trataremos de buscar un modelo más complejo y adecuado al conjunto de datos que proporcione mejor precisión para el conjunto de entrenamiento.

No ha sido posible la implementación de las arquitecturas mencionadas en el capítulo anterior, VGG-16, VGG-19 y DenseNet debido a la falta de recursos de memoria y procesadores para su ajuste. Concretamente, VGG-16 y VGG-19 tienen 53,882,385 y 58,602,001 parámetros entrenables respectivamente.

4.5. Análisis de los datos

4.5.1. Procesamiento de los datos

El conjunto de datos propuesto para el estudio son imágenes médicas en color de 96×96 pixels donde cada una ha sido extraída de exploraciones histopatológicas de secciones de ganglios linfáticos.



	id	image	label
0	train_251464		0

Figura 4.5.1: Elemento del conjunto de entrenamiento

Cada imagen para poder ser interpretada e analizada por el modelo de aprendizaje automático, es transformada en un arreglo tridimensional, $96 \times 96 \times 3$, donde el último número corresponde al valor RGB. Todas las entradas de la matriz varían entre los números 0 y 255 pero serán siempre escalados entre 0 y 1.

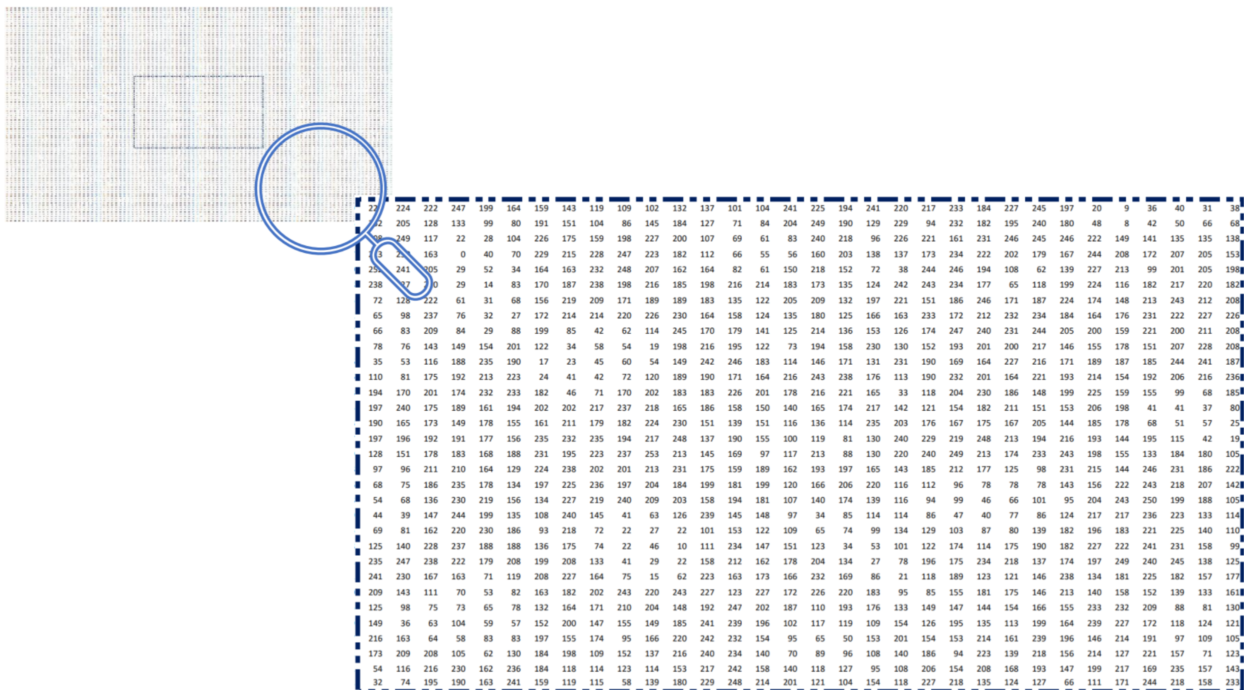


Figura 4.5.2: Imagen de la figura 4.5.1 transformada en una matriz donde solo se muestran los valores asociados al color verde por simplificar. Dentro de la matriz de tamaño 96×96 se ha amplificado la submatriz central de tamaño 32×32.

Las técnicas de procesamiento de imágenes utilizadas son las siguientes. En primer lugar, escalamos las entradas de las matrices entre el valor 0 y 1 y las convertimos en valores que ocupan 32 bits de memoria, conocidos como float32. En segundo lugar, supongamos que las imágenes dentro de estos tres conjuntos están ordenadas dependiendo de la etiqueta o simplemente siguiendo algún criterio desconocido. Con el fin de asegurarse de que los conjuntos de entrenamiento, validación y test son representativos de la distribución generadora de los datos, barajamos los datos de los conjuntos de entrenamiento, validación y test aleatoriamente consiguiendo reducir el sobreajuste (*función shuffle*). En último lugar, definimos lotes de 64 o 128 elementos (*batch size*) utilizados para cada iteración en el entrenamiento (*funciones batch y prefetch*).

4.5.2. Modelo de aprendizaje automático

Cuando se trata de clasificar imágenes, digamos con un tamaño de $96 \times 96 \times 3$, una red neuronal totalmente conectada necesita como hemos visto 27648 pesos en la primera capa oculta. Las redes que tienen un gran número de parámetros se enfrentan a varios problemas, por ejemplo, un tiempo de entrenamiento más alto. La principal diferencia funcional con una red neuronal convolucional es que, la imagen inicial reduce su tamaño a través de la convolución. Por ejemplo, una imagen de 96×96 se puede reducir a 412 imágenes de tamaño 6×6 después de varias convoluciones. Resumidamente, las arquitecturas CNN asumen explícitamente que las entradas son imágenes, lo que permite codificar ciertas propiedades en la arquitectura del modelo. El modelo red neuronal convolucional se podría resumir: pasar la matriz por una serie de convoluciones en paralelo, funciones no lineales, submuestreo y por último capas totalmente conectadas cuya salida para nuestro problema será un valor binario. En esta sección se revisarán dichas etapas acorde al conjunto de datos.

En primer lugar, se aplica a la matriz inicial varias convoluciones en paralelo de acuerdo con lo explicado en la sección 3.2.1. Es decir, si la primera convolución tiene como parámetro stride el valor 5 y como número de filtros 120 significa que el kernel es de tamaño $5 \times 5 \times 3$ y consecuentemente obtenemos 120 matrices de tamaño $92 \times 92 \times 3$. Si no deseamos que disminuya el tamaño de la imagen al ejecutar la operación de convolución, simplemente rellenamos los bordes de las imágenes con entradas cuyo valor es cero (zero padding) hasta alcanzar el tamaño inicial. En general, si las dimensiones de la entrada de la capa convolucional son $n \times n \times r$ y el parámetro stride es s (kernel tiene un tamaño $s \times s \times r$) y como número de filtros es p, obtenemos una matriz de dimensiones $(n - s + 1) \times (n - s + 1) \times p$.

A continuación, como segunda etapa de la primera capa de convolución se ejecuta una función de activación, que generalemente es 'relu'. Finalmente, disminuimos el tamaño de cada matriz obtenida, donde generalmente se usa la función *max pooling* con parámetro 2, que significa que cada matriz de tamaño $96 \times 96 \times 3$ pasa a una matriz $48 \times 48 \times 3$, es decir, cada submatriz de tamaño 4×4 es sustituida por una entrada que contiene el mayor valor de las cuatro, ver figura 4.5.4. Como alternativa a la función *max pooling* se ha experimentado el mismo modelo con la función *average pooling* en todas las capas. Aunque parecía intuirse que al ser regular, es decir, no perderse la diferenciabilidad, se han obtenido peores resultados.

Este proceso es repetido tantas veces como se desee con la idea de ir disminuyendo el tamaño de las matrices gracias a aumentar el número de matrices. El objetivo principal de aplicar capas convolucionales es identificar características como bordes, diferentes colores o curvas de la imagen. Las primeras capas convolucionales identifican características sencillas

mientras que conforme se avanza en el modelo va aumentando la capacidad de identificar características más complejas, como semicírculos, cuadrados, etc.

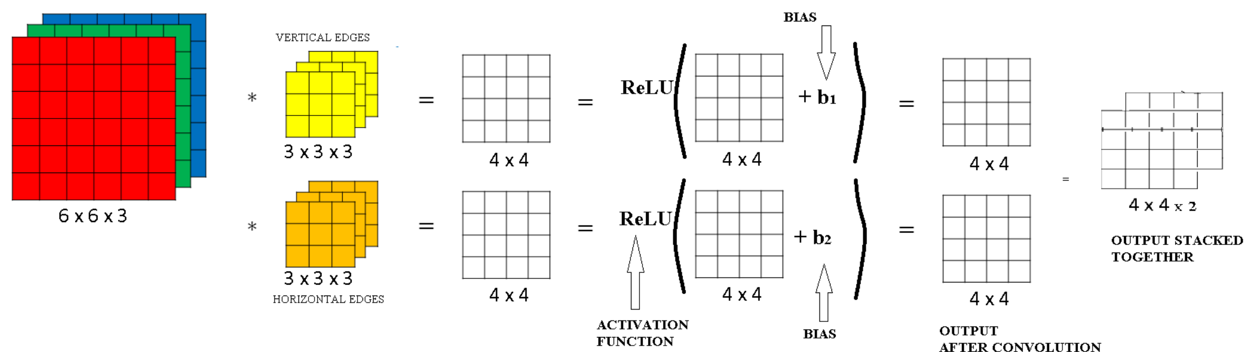
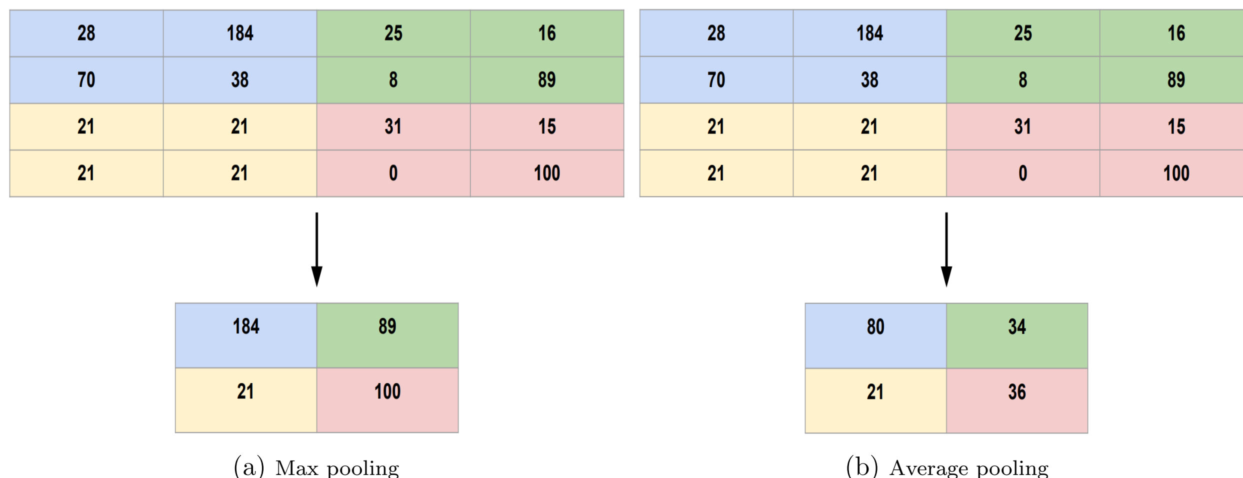


Figura 4.5.3: Ejemplo simbólico de la primera parte de la capa de convolución, donde se puede apreciar que las matrices que actúan como kernels tienen una tercera dimensión acorde con los valores RGB. Fuente: enlace



(a) Max pooling

(b) Average pooling

Figura 4.5.4: Ejemplo sencillo de la función max pooling y average pooling. Elaboración propia.

La red neuronal convolucional propuesta, ver figura 4.5.5, consta de cuatro capas convolucionales con relleno, donde cada la primera tiene un tamaño de filtro 5 y las demás 3. Después de cada convolución, se ejecuta la función max pooling con parámetro dos. El número de matrices (filtros) en cada convolución va aumentando, siendo en la primera el valor 120, en la segunda 256 y en la tercera y cuarta 412. La segunda parte del modelo consiste en una vez que se han identificado las características, es decir, se han ejecutado seis capas convolucionales obteniendo como resultado 412 matrices de tamaño 6×6, unir dicha red a una red neuronal totalmente conectada. Es decir, la capa inicial tiene como input un vector de tamaño $6 \cdot 6 \cdot 412 = 14832$, ésta es seguida de varias capas ocultas con diferentes número de nodos y finalmente la última capa contine un solo nodo. Todas las funciones de activaciones serán 'relu' excepto la última.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 96, 96, 120)       9120

 max_pooling2d (MaxPooling2D  (None, 48, 48, 120)      0
 )

 conv2d_1 (Conv2D)           (None, 48, 48, 256)       276736

 max_pooling2d_1 (MaxPooling  (None, 24, 24, 256)      0
 2D)

 conv2d_2 (Conv2D)           (None, 24, 24, 412)       949660

 max_pooling2d_2 (MaxPooling  (None, 12, 12, 412)      0
 2D)

 conv2d_3 (Conv2D)           (None, 12, 12, 412)       1528108

 max_pooling2d_3 (MaxPooling  (None, 6, 6, 412)        0
 2D)

 flatten (Flatten)           (None, 14832)             0

 dense (Dense)               (None, 412)               6111196

 dense_1 (Dense)             (None, 256)               105728

 dense_2 (Dense)             (None, 128)               32896

 dense_3 (Dense)             (None, 1)                 129

=================================================================
Total params: 9,013,573
Trainable params: 9,013,573
Non-trainable params: 0
```

Figura 4.5.5: Especificaciones en Python del modelo

4.5.3. Entrenamiento del modelo de aprendizaje automático gracias al conjunto de entrenamiento

El proceso de entrenamiento de un modelo de aprendizaje automático implica proporcionar al algoritmo de aprendizaje automático el conjunto de entrenamiento que contiene elementos del dominio con su etiqueta. De acuerdo con los explicado en secciones anteriores, el entrenamiento tiene como fin minimizar el riesgo empírico $\mathcal{R}_D$ gracias a algoritmos de optimización del descenso del gradiente estocástico. En efecto, gracias a la libreria Keras, podemos aplicar todos ellos, aunque generalmente en las simulaciones se ha optado por el optimizador 'Adam'.

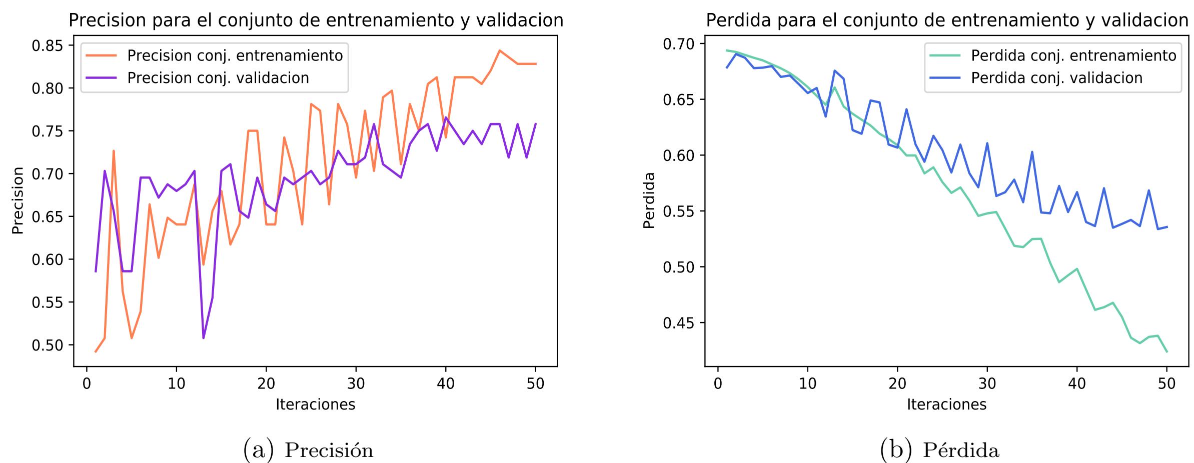
4.6. Resultados obtenidos

En la mayoría de los casos, la función de error y la función de pérdida significan lo mismo, pero con una pequeña diferencia. Una la función de error mide/calcula hasta qué punto nuestro modelo se desvía de la predicción correcta. Una función de pérdida opera sobre el error para cuantificar que tan malo es obtener un error de un tamaño, que se ve afectado por las consecuencias negativas que resultan en una predicción incorrecta. Decidida la clase

de hipótesis, debido a nos encontramos en el marco de la clasificación binaria, la función de pérdida será la entropía cruzada binaria.

La **medida de rendimiento utilizada** es la precisión (accuracy) que es la proporción de ejemplos para los cuales el modelo funciona adecuadamente. Al estar interesados en saber cómo funciona el algoritmo en datos no conocidos a priori, evaluaremos la precisión utilizando un conjunto de datos de prueba, *test set*, el cual es independiente de los datos utilizados para entrenar el sistema. Cabe destacar que la pérdida no es un porcentaje como la precisión, es una suma. Analizando las dos cantidades en conjunto es posible obtener más información acerca de las predicciones del modelo. Si la precisión es baja y la pérdida alta entonces significa que el modelo no predice correctamente en la mayoría de los puntos. Por el contrario, si los dos valores son pequeños, significa que el modelo produce errores pequeños en la mayoría de los datos. Si tanto la precisión como la pérdida son valores grandes, el modelo predice mal en pocos puntos. Finalmente, si la precisión es un valor alto y la pérdida bajo, entonces se puede decir que el modelo comete pequeños errores en pocos puntos, que es la situación deseada. [Enlace a la fuente]

Se puede observar que la precisión del modelo para el conjunto de entramiento es aproximadamente de 0.8515625 después de 50 iteraciones. Sin embargo, es interesante que un modelo generalice bien, es decir, que obtenga buenas predicciones para puntos no usados en el entrenamiento. El estimador del riesgo obtenido se encuentra entre los valores 0.75 y 0.8, considerado buen valor tratándose de imágenes médicas. En cuanto a la pérdida, se observa que la pérdida respectiva al conjunto de entramiento disminuye si se aumenta el número de iteraciones, mientras que la pérdida para observaciones del conjunto de validación disminuye hasta alcanzar su mínimo entre 40 y 50 iteraciones y aumenta posteriorente debido al sobreajuste. Resumidamente, los resultados obtenidos nos dicen que el modelo aproximadamente el $75 - 80\,\%$ de las veces clasifica bien para una imagen no pertenceciente al conjunto de entrenamiento.



(a) Precisión

(b) Pérdida

Figura 4.6.1: Resultados obtenidos para el modelo de la figura 4.5.5

CAPÍTULO 5

Conclusiones finales y futuras líneas de trabajo

En este trabajo se ha estudiado el problema de clasificación binaria supervisado desde el punto de vista de estadística matemática, finalizando con una aplicación práctica.

Durante todo el proyecto el objetivo es, elegida una clase de hipótesis $\mathcal{H}$, hallar un clasificador que equilibre entre menor riesgo empírico y menor diferencia entre el riesgo empírico y el riesgo verdadero, o su estimación.

En el marco del aprendizaje convexo, se han revisado las garantías estadísticas y computacionales de posibles algoritmos obteniendo muy buenos resultados a nivel teórico. Sin embargo, en el modelo principal de estudio, las redes neuronales artificiales, los clasificadores obtenidos pueden ser complejos provocando que el riesgo no sea una función convexa y consecuentemente no se encuentran dentro del anterior contexto, condición que no implica que no se obtengan excelentes resultados a nivel práctico.

Basado en lo estudiado hasta el momento, podría parecer que el modelo de redes neuronales es válido cuando los elementos del dominio de datos poseen una topología similar a una cuadrícula como las imágenes. No obstante, si transformamos una imagen 96×96 en un vector de dimensión 9216×1, píxeles cercanos y relacionados en la imagen son ordenados sin coherencia en el nuevo vector, eliminando el significado intrínseco de la imagen. Surge así la necesidad de modificar la multiplicación de matrices en el modelo por la operación de convolución, capaz de promediar píxeles cercanos y de aprovechar así la estructura bidimensional de las imágenes. Esto motiva el interés de las llamadas redes neuronales convolucionales.

El proyecto finaliza con una aplicación práctica de la teoría abordada a través de la que he constatado que no existe aún una sistematización clara que permita encontrar una arquitectura adecuada para cualquier conjunto de datos.

Tras realizar este Trabajo Fin de Grado he tomado consciencia del enorme esfuerzo, valor y dedicación de todos aquellos investigadores que, cada uno en su campo, de forma individual o colectiva, están colaborando en la actual revolución científico-tecnológica en la que todos estamos inmersos. Como dice *Fei-Fei Li*, profesora de ciencias de la computación en la Universidad de Stanford "*Como tecnólogo, veo cómo la Inteligencia Artificial y la cuarta revolución industrial afectarán todos los aspectos de la vida de las personas* ".

Quiero destacar en este punto el trabajo de Geoffrey Hinton, Yann LeCun y Yoshua Bengio, considerados padres de una técnica esencial de la Inteligencia Artificial, el Aprendizaje Profundo, que ha sido mi referencia durante todo este tiempo. Geoffrey Hinton en 1986 inventó los algoritmos de retropropagación fundamentales para entrenar redes neuronales, considerado un verdadero hito en la Inteligencia Artificial. Yann LeCun hizo aportaciones al trabajo de Hinton y el 1989 diseñó LeNet-5, una de las arquitecturas pioneras en el campo y que ha servido de motivación para investigaciones futuras. Por su parte, Yoshua Bengio ha realizado contribuciones clave en modelos probabilísticos de secuencias. El trabajo de todos ellos fue reconocido por la *Association for Computing Machinery* en 2018 con el Premio Turing y este año 2022 junto con Demis Hassabis, CEO y cofundador de *DeepMind* serán galardonados con el Premio Princesa de Asturias de Investigación Científica y Técnica. Enlace a la fuente

En cuanto a futuras líneas de investigación podemos clasificarlas en tres ámbitos:

- A nivel de Estadística, como se ha constatado en el último siglo, el avance ha sido notorio pero el desarrollo de esta ciencia es un continuum.
- En segundo lugar, teniendo en cuenta que se obtienen buenos resultados en la práctica, serán precisos teoremas (garantías estadísticas y computacionales) que conformen el cuerpo teórico de las correcciones del algoritmo de descenso de gradiente estocástico como Momentum, RMSProp y Adam, entre otros.
- A nivel experimental e informático, existe la posibilidad de, o bien haciendo uso del aprendizaje por transferencia o de la búsqueda de otras arquitecturas, de diseñar una red neuronal convolucional para el conjunto de datos propuesto con mejor generalización que la presentada.

Mi modesto trabajo es sólo un pequeño punto de una infinita línea continua de la estadística aplicada al aprendizaje automático, que tiene aún un largo camino por recorrer.

Me gustaría concluir con una cita de *Yan LeCun "Nuestra inteligencia es lo que nos hace humanos, y la Inteligencia Artificial es una extensión de esa cualidad"* y otra de *Timothy Donald Cook*, directivo de Apple Inc. *"Lo que todos tenemos que hacer es asegurarnos de que estamos usando la Inteligencia Artificial de una manera que sea en beneficio de la humanidad, no en detrimento de la humanidad"*.

CAPÍTULO 6

Anexos

6.1. Problemas de aprendizaje convexos

En esta sección se introducirán los problemas de aprendizaje convexos, familia importante de problemas de aprendizaje debido a que se pueden resolver eficientemente. En general, un problema de aprendizaje convexo es aquel cuya clase de hipótesis es un conjunto convexo, y cuyas funciones de pérdida son funciones convexas [[14], Capítulo 12].

Definición 6.1.0.1. ***Conjunto convexo*** *Un conjunto C de un espacio vectorial es un conjunto convexo si para dos vectores $\boldsymbol{u}, \boldsymbol{v} \in C$, el segmento que los une está contenido en C. Es decir, para cualquier $\alpha \in [0,1]$ se tiene que $\alpha\boldsymbol{u} + (1-\alpha)\boldsymbol{v} \in C$. [[14], Definición 12.1]*

Definición 6.1.0.2. ***Función convexa*** *Sea C un conjunto convexo. Una función $f : C \to R$ es convexa si para cualquier $\boldsymbol{u}, \boldsymbol{v} \in C$ y $\alpha \in [0,1]$, se tiene que $f(\alpha\boldsymbol{u} + (1-\alpha)\boldsymbol{v}) \leq \alpha f(\boldsymbol{u}) + (1-\alpha) f(\boldsymbol{v})$. [[14], Definición 12.2]*

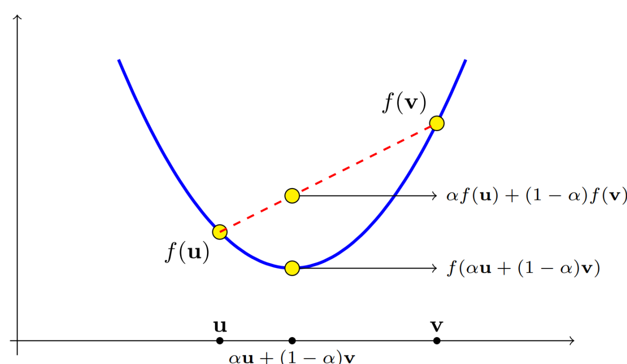


Figura 6.1.1: Función convexa, [[14], Definición 12.2]

El epigrafo de una función f es el conjunto

$$epigrafo(f) := \{(\boldsymbol{x}, \beta) : f(\boldsymbol{x}) \leq \beta\}. \tag{6.1.1}$$

Es sencillo probar que una función es convexa si y solo si su epigrafo es un conjunto convexo. [[14], Ecuación 12.1]

Una propiedad importante de las funciones convexas es que todo mínimo local es también un mínimo global. Otra propiedad relevante es que, para todo punto, podemos construir la tangente a la función en dicho punto, más aún, siempre está por debajo de toda imagen de f.

Lema 6.1.1. *Sea $f : R \to R$ una función real y f', f'' su derivada primera y segunda respectivamente. Los siguientes enunciados son equivalentes:*

- *f es convexa*
- *f' es monótonamente no decreciente*
- *f'' es no negativa*

[[14], Lema 12.3]

El siguiente resultado muestra que la composición de una función escalar convexa con una función lineal es una función convexa.

Lema 6.1.2. *Supongamos que $f : R^d \to R$ puede escribirse como $f(\boldsymbol{w}) = g(\langle \boldsymbol{w} , \boldsymbol{x} \rangle + y)$ para algún $\boldsymbol{x} \in R^d, y \in R, g : R \to R$. Entonces si g es convexa, también lo es f. [[14], Lema 12.4]*

Demostración. Sean $\boldsymbol{w}_1, \boldsymbol{w}_2 \in R^d$ y $\alpha \in [0, 1]$. Se verifican las siguientes relaciones:

$$\begin{aligned} f(\alpha \boldsymbol{w}_1 + (1-\alpha)\boldsymbol{w}_2) &= g(\langle \alpha \boldsymbol{w}_1 + (1-\alpha)\boldsymbol{w}_2 \ , \ \boldsymbol{x} \rangle + y) \\ &= g(\alpha \langle \boldsymbol{w}_1, \boldsymbol{x} \rangle + (1-\alpha)\langle \boldsymbol{w}_2 \ , \ \boldsymbol{x} \rangle + y) \\ &= g(\alpha(\langle \boldsymbol{w}_1 \ , \ \boldsymbol{x} \rangle + y) + (1-\alpha)(\langle \boldsymbol{w}_2 \ , \ \boldsymbol{x} \rangle + y)) \\ &\leq \alpha g(\langle \boldsymbol{w}_1 \ , \ \boldsymbol{x} \rangle + y) + (1-\alpha) g(\langle \boldsymbol{w}_2 \ , \ \boldsymbol{x} \rangle + y), \end{aligned} \tag{6.1.2}$$

y de la última desigualdad se deduce la convexidad de g. □

Con un razonamiento similar se prueba que el máximo de funciones convexas es una función convexa y que la suma ponderada con pesos no negativos de funciones convexas es una función convexa. [[14], Lema 12.5]

Lema 6.1.3. *Para $i = 1, ...r$ sea $f_i : R^d \to R$ una función convexa. Las siguientes funciones de R^d a R son a su vez convexas:*

- $g(x) = \max_{i \in [r]} f_i(x)$
- $g(x) = \sum_{i \in [r]} w_i f_i(x)$ *donde* $w_i \geq 0 \forall i$

Demostración. La primera hipótesis se sigue de:

$$\begin{aligned} g(\alpha u + (1-\alpha)v) &= \max_i f_i(\alpha u + (1-\alpha)v) \\ &\leq \max_i [\alpha f_i(u) + (1-\alpha) f_i(v)] \\ &\leq \alpha \max_i f_i(u) + (1-\alpha) \max_i f_i(v) \\ &= \alpha g(u) + (1-\alpha) g(v). \end{aligned} \tag{6.1.3}$$

La segunda hipótesis se dedice de:

$$
\begin{aligned}
g(\alpha u + (1-\alpha)v) &= \sum_i w_i f_i(\alpha u + (1-\alpha)v) \\
&\leq \sum_i w_i[\alpha f_i(u) + (1-\alpha) f_i(v)] \\
&= \alpha \sum_i w_i f_i(u) + (1-\alpha)\sum_i w_i f_i(v) \\
&= \alpha g(u) + (1-\alpha) g(v).
\end{aligned}
\tag{6.1.4}
$$

□

Definición 6.1.0.3. ***Función Lipschitz bajo la norma Euclidea sobre*** R^d *Sea* $C \in R^d$*. Una función* $f : R^d \to R^k$ *se dice* ρ*-Lipschitz sobre* C *si para cada* $\boldsymbol{w}_1, \boldsymbol{w}_2 \in C$ *se tiene que* $\|f(\boldsymbol{w}_1) - f(\boldsymbol{w}_2)\| \leq \rho\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|$*. [[14], Definición 12.6]*

Intuitivamente una función Lipschitz no varía muy rápido. Notemos que si $f : R \to R$ es diferenciable, gracias al teorema del punto medio obtenemos que $f(w_1) - f(w_2) = f'(u)(w_1 - w_2)$, donde u es un punto intermedio entre w_1 y w_2. Si la derivada de f está acotada en valor absoluto en todos sus puntos por ρ, entonces la función es ρ-Lipschitz.

Lema 6.1.4. *Sea* $f(\boldsymbol{x}) = g_1(g_2(\boldsymbol{x}))$*, donde* g_1 *es* ρ_1*-Lipschitz y* g_2 *es* ρ_2*-Lipschitz. Entonces,* f *es* $(\rho_1\rho_2)$*-Lipschitz. Resumidamente, la composición de funciones Lipschitz preserva la propiedad. [[14], Lema 12.7]*

Demostración.

$$
\begin{aligned}
|f(\boldsymbol{w}_1) - f(\boldsymbol{w}_2)| &= |g_1(g_2(\boldsymbol{w}_1)) - g_1(g_2(\boldsymbol{w}_2))| \\
&\leq \rho_1\|g_2(\boldsymbol{w}_1) - g_2(\boldsymbol{w}_2)\| \\
&\leq \rho_1\rho_2\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|
\end{aligned}
\tag{6.1.5}
$$

□

Definición 6.1.0.4. ***Función suave*** *Una función* $f : R^d \to R$ *se dice* β*-suave si su gradiente es* β*-Lipschitz. Es decir, para cada* $\mathbf{u}, \boldsymbol{w}$ *se verifica* $\|\nabla f(\boldsymbol{v}) - \nabla f(\boldsymbol{w})\| \leq \beta\|\boldsymbol{v} - \boldsymbol{w}\|$*. [[14], Definición 12.8]*

El siguiente lema afirma que la composición de una función suave con una función lineal sigue siendo suave.

Lema 6.1.5. *Sea* $f(\boldsymbol{w}) = g(\langle \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle + b)$ *para algún* $\boldsymbol{x} \in R^d, b \in R, g : R \to R$*. Entonces si* g *es* β*-suave,* f *es* $(\beta\|\boldsymbol{x}\|^2)$*-suave. [[14], Lema 12.9]*

Demostración. Aplicando la regla de la cadena obtenemos que $\nabla f(\boldsymbol{w}) = g'(\langle \boldsymbol{w}, \boldsymbol{x}\rangle + b)\boldsymbol{x}$, donde g' es la derivada de g. Usando que g es suave junto con la desigualdad de Cauchy-Schwartz obtenemos:

$$
\begin{aligned}
f(\boldsymbol{v}) &= g(\langle \boldsymbol{v}\ ,\ \boldsymbol{x}\rangle + b) \\
&\leq g(\langle \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle + b) + g'(\langle \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle + b)\langle \boldsymbol{v} - \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle + \frac{\beta}{2}(\langle \boldsymbol{v} - \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle)^2 \\
&\leq g(\langle \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle + b) + g'(\langle \boldsymbol{w}\ ,\ \boldsymbol{x}\rangle + b)\langle \boldsymbol{v} - \boldsymbol{w}\ ,\ \boldsymbol{x} > + \frac{\beta}{2}(\|\boldsymbol{v} - \boldsymbol{w}\|\|\boldsymbol{x}\|)^2 \\
&= f(\boldsymbol{w}) + \langle \nabla f(\boldsymbol{w})\ ,\ \boldsymbol{v} - \boldsymbol{w}\rangle + \frac{\beta\|\boldsymbol{x}\|^2}{2}\|\boldsymbol{v} - \boldsymbol{w}\|^2
\end{aligned}
\tag{6.1.6}
$$

□

6.1.1. Problemas de aprendizaje convexos

Definición 6.1.1.1. ***Problema de aprendizaje convexo*** *Llamaremos problema de aprendizaje convexo* $(\mathcal{H}, \mathcal{X}, \mathcal{Y}, \ell)$ *si la clase de hipótesis* $\mathcal{H}$ *es un conjunto convexo y para todo* $(\boldsymbol{x}, y) \in \mathcal{X} \times \mathcal{Y}$*, la función de pérdida* $\ell(\cdot, (\boldsymbol{x}, y))$ *es una función convexa. [[14], Definición 12.10]*

Un ejemplo particular es la regresión lineal con la pérdida cuadrática, $\ell(h(\boldsymbol{x}), y) = \frac{1}{2}\|h(\boldsymbol{x}) - y\|^2$.

Lema 6.1.6. *Si la función de pérdida* ℓ *es una función convexa y la clase de hipótesis* $\mathcal{H}$ *es un conjunto convexo, entonces el algoritmo de* $MRE_{\mathcal{H}}(S) = \arg\min_{w\in\mathcal{H}} R_S(h)$ *es un problema de optimización convexa (minimizar una función convexa sobre un conjunto convexo). [[14], Lema 12.11]*

Demostración. El minimizador de riesgo empírico está definido como

$$\hat{h} \in \underset{h\in\mathcal{H}}{\operatorname{argmin}}\, \mathcal{R}_{\mathcal{S}}(h).$$

Si S es el conjunto de entrenamiento entonces $\mathcal{R}_{\mathcal{S}}(h) = \frac{1}{n}\sum_{i=1}^{n} \ell(\boldsymbol{x_i}, y_i, h)$. Aplicando el lema 6.1.3 $\mathcal{R}_{\mathcal{S}}(h)$ es convexa. Por ende, el algoritmo MRE es un problema de minimizar una función convexa en un conjunto convexo. □

En este caso hay garantías en cuanto a la convergencia.

Definición 6.1.1.2. ***Problema de aprendizaje convexo-Lipschitz-acotado*** *Llamaremos problema de aprendizaje convexo-Lipschitz-acotado* $(\mathcal{H}, \mathcal{X}, \mathcal{Y}, \ell)$ *con parámetros* ρ, B *si se satisface:*

- *La clase de hipótesis* $\mathcal{H}$ *es un conjunto convexo y para todo* $h \in \mathcal{H}$ *se tiene que* $\|h\| \leq B$.
- *Para todo* $(\boldsymbol{x}, y) \in \mathcal{X} \times \mathcal{Y}$*, la función de pérdida* $\ell(\cdot, (\boldsymbol{x}, y))$ *es una función convexa y* ρ*-Lipschitz.*

[[14], Definición 12.12]

Definición 6.1.1.3. ***Problema de aprendizaje convexo-acotado-suave*** *Llamaremos problema de aprendizaje convexo-acotado-suave* $(\mathcal{H}, \mathcal{X}, \mathcal{Y}, \ell)$ *con parámetros* β, B *si se satisface:*

- *La clase de hipótesis* $\mathcal{H}$ *es un conjunto convexo y para todo* $h \in \mathcal{H}$ *se tiene que* $\|h\| \leq B$.
- *Para todo* $(\boldsymbol{x}, y) \in \mathcal{X} \times \mathcal{Y}$*, la función de pérdida* $\ell(\cdot, (\boldsymbol{x}, y))$ *es una función convexa y* β*-suave y no negativa.*

[[14], Definición 12.13]

6.1.2. Funciones de pérdida sustituta

En general las funciones de pérdida naturales no son convexas y en particular implementar el algoritmo MRE es computacionalmente intratable, por ejemplo cuando la función de pérdida es la pérdida 0-1. Una posible solución a este problema es acotar superiormente la función no convexa por una función sustituta convexa. Por ejemplo, para la pérdida 0-1 podemos definir la función $\ell^*(h(\boldsymbol{x}), \boldsymbol{x}, y) := \max\{0, 1 - y < h, \boldsymbol{x} >\}$. Se verifica que $\ell^{0,1}(h, (\boldsymbol{x}, y)) \leq \ell^*(h(\boldsymbol{x}), \boldsymbol{x}, y)$ y es convexa. Estas nuevas funciones permiten manejar problemas no convexos minimizando funciones sustitutas convexas. Desafortunadamente, puede verse notablemente incrementado el riesgo. [[14], Sección 12.3]

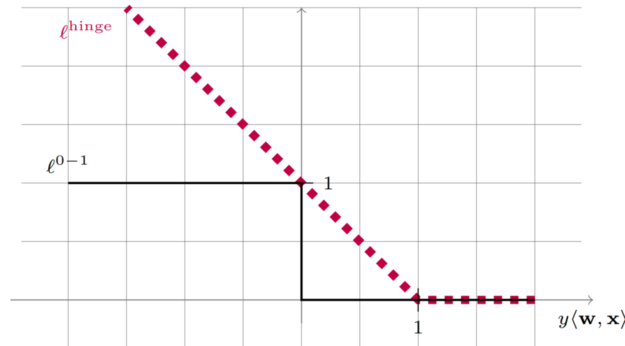


Figura 6.1.2: Posible función sustituta para la pérdida 0-1, [[14], Sección 12.3]

6.2. Definiciones relativas a la Teoría de la Información

La intuición básica detrás de la teoría de la información es que saber que ha ocurrido un evento improbable da más información que saber que ha ocurrido un evento probable [[6], Capítulo 3]. Nos gustaría cuantificar la información de manera que formalice esta intuición:

- Los eventos probables deben tener un contenido de información bajo, y en el caso extremo, los eventos que suceden con garantías no deben tener contenido de información.
- Los eventos menos probables deberían tener un mayor contenido de información.
- Los eventos independientes deben tener información aditiva.

El concepto de información es demasiado amplio para ser abordado en una sola definición, es por ello que dada una distribución de probabilidad, definiremos el concepto de entropía, el cual presenta propiedades que debería tener una medida de información, [[3], Capítulo 2].

Definición 6.2.0.1. *Definimos la* ***autoinformación*** *de un evento $X = x$ como*

$$I(x) = -\log P(x) \tag{6.2.1}$$

donde log *simboliza el logaritmo en base e. $I(x)$ es medido en nats. Un nat es la cantidad de información ganada observando un suceso con probabilidad $\frac{1}{e}$. [[6], Sección 3.13]*

La autoinformación está definida solo para un evento por tanto cuantificamos la incertidumbre de la distribución de probabilidad usando la entropía Shannon.

Definición 6.2.0.2. ***Entropía*** *Sea X una varible aleatoria discreta con función de masa de probabilidad $P(x) = Pr(X = x), x \in \mathcal{X}$. La entropía de la variable aleatoria discreta X está definida como:*

$$H(X) = E_{x\sim P}[I(x)] = -\sum_{x\in\mathcal{X}} P(x)\log P(x) \tag{6.2.2}$$

En otras palabras, es una medida de la incertidumbre en media asociada a una variable aleatoria. El logaritmo es en base 2 y la entropía se mide en bits. [[3], Sección 2.1]

Lema 6.2.1. $H(X) \geq 0$

Demostración. $0 \leq P(x) \leq 1$ implica que $\log \frac{1}{P(x)} \geq 0$ y por tanto $H(X) \geq 0$. [[3], Lema 2.1.1] □

Es posible cambiar la base del logaritmo de la definición 6.2.0.2:

Lema 6.2.2. $H_b(X) = (\log_b(a))H_a(X)$

Demostración. $\log_b(P) = \log_b(a)\log_a(P)$ [[3], Lema 2.1.2] □

Extenderemos la definición de entropía a un par de variables aleatorias X, Y, que puede ser visto como la misma definición aplicada al vector aleatorio (X, Y) [[3], Sección 2.2].

Definición 6.2.0.3. ***Entropía conjunta*** *Sean X, Y dos varibles aleatorias discretas y $P(x, y)$ la función de distribución conjunta. La entropía conjunta del par de variables aleatorias discretas (X, Y) está definida como:*

$$H(X,Y) = -E[\log P(x,y)] = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x,y) \tag{6.2.3}$$

Definimos ahora la entropía condicionada de una variable aleatoria dada otra como el valor esperado de la entropía para la distribución condicionada:

Definición 6.2.0.4. ***Entropía condicionada*** *Sean X, Y dos varibles aleatorias discretas y $P(x, y)$ la función de distribución conjunta. La entropía condiciondad del par de variables aleatorias discretas (X, Y) está definida como [[3], Sección 2.2]:*

$$H(X|Y) = -E[\log P(x|y)] = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x|y) \tag{6.2.4}$$

Teorema 6.2.3. *(Regla de la cadena) [[3], Teorema 2.2.1]:* $H(X,Y) = H(X) + H(Y|X)$

Demostración.

$$\begin{aligned} H(X,Y) &= -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x,y) \\ &= -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x)P(y|x) \\ &= -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x) - \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(y|x) \\ &= -\sum_{x\in\mathcal{X}} P(x)\log P(x) - \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(y|x) \\ &= H(X) + H(Y|X). \end{aligned} \tag{6.2.5}$$

□

Resumidamente, la entropía es una medida de la cantidad de información necesaria en media para describir una variable aleatoria. Se definirán ahora dos conceptos relacionados: entropía relativa (o divergencia de Kullback-Leiber) y la información mutua (mutual information) . La entropía relativa, hablando coloquialmente, es una medida de la similitud entre dos distribuciones de probabilidad. [[3], Sección 2.3]

Definición 6.2.0.5. ***Entropía relativa o divergencia Kullback-Leibler*** *Sean $P(x)$ y $Q(x)$ dos funciones de masa de probabilidad. La entropía relativa o distancia Kullback-Leibler entre $P(x)$ y $Q(x)$ está definida como:*

$$\begin{aligned} D_{KL}(P||Q) &= E_{x\sim P}[\log \frac{P(x)}{Q(x)}] = E_{x\sim P}[\log P(x) - \log Q(x)] \\ &= \sum_{x\in\mathcal{X}} P(x)\log\frac{P(x)}{Q(x)} \end{aligned} \tag{6.2.6}$$

En la definición anterior se define como $0\log\frac{0}{0} = 0$.

La entropía relativa es no negativa y es igual a 0 si y solo si P y Q son la misma distribución, sin embargo, no es simétrica. En el contexto del Aprendizaje Automático, dadas dos distribuciones Q y P, la divergencia de Kullback-Leibler, cuantifica la información que obtendríamos si empleásemos Q en lugar de P.

El siguiente concepto es la información mutua que es una medida de la cantidad de información que una variable aleatoria contiene de otra variable aleatoria.

Definición 6.2.0.6. ***Información mutua*** *Sean X e Y dos variables aleatorias con función de masa de probabilidad conjunta $P(x,y)$ y funciones de masa de probabilidad marginales $P(x)$ y $Q(x)$ respectivamente. La información mutua $I(X;Y)$ es la entropía relativa entre la distribución conjunta y el producto de las marginales:*

$$\begin{aligned} I(X;Y) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log\frac{P(x,y)}{P(x)P(y)} \\ &= D_{KL}(P(x,y)||P(x)P(y)) \\ &= E_{P(x,y)}\log\frac{P(X,Y)}{P(X)P(Y)} \end{aligned} \tag{6.2.7}$$

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

Lema 6.2.4. $I(X;Y) = H(X) - H(X|Y)$

Demostración.

$$\begin{aligned} I(X;Y) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log\frac{P(x,y)}{P(x)P(y)} \\ &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log\frac{P(x|y)}{P(x)} \\ &= -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x) + \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x|y) \\ &= -\sum_{x\in\mathcal{X}} P(x)\log P(x) - (-\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log P(x|y)) \\ &= H(X) - H(X|Y). \end{aligned} \tag{6.2.8}$$

[[3], Sección 2.4] □

Por tanto, la información mutua $I(X;Y)$ simboliza la reducción de incertidumbre de X gracias al conocimiento de Y.

Definición 6.2.0.7. ***Entropía cruzada*** *Sean $P(x)$ y $Q(x)$ dos funciones de masa de probabilidad. La entropía cruzada (cross-entropy) entre $P(x)$ y $Q(x)$ está definida como:*

$$\begin{aligned} H(P,Q) &= H(P) + D_{KL}(P||Q) \\ &= -E_{x\sim P}[\log Q(x)] \end{aligned} \tag{6.2.9}$$

Es claro que la entropía cruzada con respecto a Q es equivalente a minimiar la divergencia KL, ya que Q no está en el primer término, 6.2.9. [5]

6.3. Programas escritos en Python

6.3.1. Modelo red neuronal convolucional LeNet-5 para el conjunto de datos MNIST

```
1 !pip install keras
2 !pip install TensorFlow
```

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import tensorflow as tf
4 from tensorflow import keras
5 from tensorflow.keras import datasets, layers, models
6 import tensorflow_datasets as tfds
```

```
# Cargamos el conjunto de datos y lo separamos en dos conjuntos:
# entrenamiento y validacion
from keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
print('X_train shape', X_train.shape, 'X_test shape', X_test.shape)
```

```
# Visualizamos 8 imagenes del conjunto de entrenamiento
import matplotlib.pyplot as plt
import random
plt.figure(figsize = (12,5))
for i in range(8):
    ind = random.randint(0, len(X_train))
    plt.subplot(240+1+i)
    plt.imshow(X_train[ind], cmap=plt.get_cmap('gray'))
```

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import classification_report, log_loss, accuracy_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, AveragePooling2D, Dense,
    Flatten, BatchNormalization, Dropout
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
from tensorflow.keras.losses import categorical_crossentropy
```

```
def preprocess_data(X_train, Y_train, X_test, Y_test):
    X_train = np.pad(X_train,((0,0),(2,2),(2,2))) # padding
    X_test = np.pad(X_test,((0,0),(2,2),(2,2)))

    # cambiamos el tamaño de las imagenes para el modelo
    X_train = X_train.reshape(X_train.shape[0], X_train.shape[1],
     ↪  X_train.shape[2], 1)
    X_test = X_test.reshape(X_test.shape[0], X_test.shape[1],
     ↪  X_test.shape[2], 1)


    # Convertimos las entradas de las matrices en floats
    X_train = X_train.astype('float32')
    X_test = X_test.astype('float32')

    # Normalizamos las entradas de las matrices
    X_train = X_train/255.0
    X_test_norm = X_test/255.0

```

```
18    # Codificamos la salida con One-hot encoding label
19    Y_train = to_categorical(Y_train,10)
20    Y_test = to_categorical(Y_test,10)
21
22    return X_train, Y_train, X_test, Y_test
```

```
1 model = Sequential()
2
3   # Capa convolucional
4 model.add(Conv2D(input_shape=(32,32,1), filters = 6, kernel_size = (5,5),
  ↪   activation = 'tanh', name='C1'))
5
6   # Capa Max-pooing con tamaño 2x2
7 model.add(AveragePooling2D(pool_size = (2,2), name='S2'))
8
9   # Capa convolucional
10 model.add(Conv2D(filters = 16, kernel_size = (5,5), activation = 'tanh', name
  ↪   = 'C3'))
11
12   # Capa Max-pooling
13 model.add(AveragePooling2D(pool_size = (2,2), name='S4'))
14
15   # Antes de añadir capas totalmente conectadas debemos transformar la ultima
     ↪   salida en un vector
16 model.add(Flatten())
17
18   # Primera capa totalemente conectada
19 model.add(Dense(120, activation = 'tanh', name='FC5'))
20
21   # Segunda capa totalemente conectada
22 model.add(Dense(84, activation = 'tanh', name='F6'))
23
24   # Ultima capa
25 model.add(Dense(10, activation = 'softmax', name='Output'))
26
27 model.summary()
```

```
1 # Procesamiento de los datos
2 X_train, Y_train, X_test, Y_test = preprocess_data(X_train, Y_train, X_test,
  ↪   Y_test)
```

```
1 # Definimos la funcion de perdida, el optimizador y la medida de rendimiento
  ↪   que se van a utilizar
2 model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics
  ↪   = ['accuracy'])
```

```
# Entrenamiento
history = model.fit(X_train,Y_train,epochs=50, batch_size=32, validation_data
↪ = (X_test, Y_test), verbose = 1)
```

```
model.evaluate(X_train, Y_train)
```

```
model.evaluate(X_test, Y_test, verbose = 1)
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'coral', label='Precision conj. entrenamiento')
plt.plot(epochs, val_acc, 'blueviolet', label='Precision conj. validacion')
plt.xlabel("Iteraciones")
plt.ylabel("Precision")
plt.title('Precision para el conjunto de entrenamiento y validacion')
plt.legend()
plt.savefig('LeNet_MNIST_acc.pdf')
plt.show()
```

```
plt.figure()
plt.plot(epochs, loss, 'mediumaquamarine', label='Perdida conj.
↪ entrenamiento')
plt.plot(epochs, val_loss, 'royalblue', label='Perdida conj. validacion')
plt.xlabel("Iteraciones")
plt.ylabel("Perdida")
plt.title('Perdida para el conjunto de entrenamiento y validacion')
plt.legend()
plt.savefig('LeNet_MNIST_loss.pdf')
plt.show()
```

```
import numpy as np

# Predecimos las etiquetas para los elementos del conjunto de validacion de
↪ acuerdo con el modelo entrenado
y_test_pred = []
for i in range(len(X_test)):
    img = X_test[i]
    img = img.reshape(1,32,32,1)
    img = img.astype('float32')
    img = img/255.0
    vec_p = model.predict(img)
    y_p = np.argmax(vec_p)
    y_test_pred.append(y_p)
```

```
13
14 # Convertimos y_test_pred en un array (era una lista)
15 y_test_pred = np.asarray(y_test_pred)
```

```
1 # Posibles etiquetas
2 cates = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
3 # Visualizamos algunas imagenes del conjunto de validacion con la etiqueta
  ↪ predicha por el modelo entrenado
4 plt.figure(figsize = (12,7))
5 for i in np.arange(8):
6     ind = random.randint(0,len(X_test))
7     img = X_test[ind]
8     img = img.reshape(1,32,32,1)
9     img = img.astype('float32')
10    img = img/255.0
11    v_p = y_test_pred[ind]
12    plt.subplot(240+1+i)
13    plt.imshow(X_test[ind], cmap=plt.get_cmap('gray'))
14    plt.title(v_p)
15
16 plt.savefig('LeNet_MNIST_pred.pdf')
```

6.3.2. Modelo red neuronal totalmente conectada

```
1 !pip install keras
2 !pip install TensorFlow
3 pip install tensorflow-datasets
4 pip install tensorflow_data_validation
```

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import tensorflow as tf
4 from tensorflow import keras
5 from tensorflow.keras import datasets, layers, models
6 import tensorflow_datasets as tfds
```

```
1 # Importar el conjunto de datos
2 df, info = tfds.load('patch_camelyon', with_info = True,
3 as_supervised = True)
```

```
1 # Separar el conjunto de entrenamiento, validacion y test
2 train_data = df['train']
3 valid_data = df['validation']
4 test_data = df['test']
```

```
# Procesamiento de los datos
def preprocess(image, labels):
    image = tf.cast(image, tf.float32)
    image /= 255.
    image_res=tf.reshape(image, [27648])
    return image_res, labels
```

```
# Aplicar el procesamiento gracias a la funcion map()
train_data = train_data.map(preprocess)
valid_data = valid_data.map(preprocess)
test_data = test_data.map(preprocess)
```

```
# Barajamos las imagenes
buffer_size = 1000
train_data = train_data.shuffle(buffer_size)
```

```
batch_size = 4096
train_data = train_data.batch(batch_size).prefetch(1)
valid_data = valid_data.batch(batch_size).prefetch(1)
test_data = test_data.batch(batch_size).prefetch(1)
```

```
# Separamos la imagen de la etiqueta en variables diferentes
train_images, train_labels = next(iter(train_data))
valid_images, valid_labels = next(iter(valid_data))
test_images, test_labels  = next(iter(test_data))
#Check
valid_labels.shape
#Chec
train_images.shape
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D,
MaxPooling2D, Dense, Flatten, BatchNormalization, Dropout
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
```

```
input_size = 27648
batch_size = 128
hidden1 = 864
hidden2 = 432
hidden3 = 54
hidden4 = 27
classes = 1
```

```
# Modelo de aprendizaje automatico
FCN = Sequential()
FCN.add(Dense(hidden1, input_dim=input_size, activation='relu'))
FCN.add(Dense(hidden1, activation='relu'))
FCN.add(Dense(hidden2, activation='relu'))
FCN.add(Dense(hidden3, activation='relu'))
FCN.add(Dense(hidden4, activation='relu'))
FCN.add(Dense(classes, activation='sigmoid'))

FCN.summary()
```

```
# Compilar el modelo con el optimizador descenso del gradiente estocastico
FCN.compile(optimizer='sgd',
loss='binary_crossentropy', metrics=['binary_accuracy'])
```

```
# Entrenamiento
history = FCN.fit(train_images, train_labels, batch_size = 4096, epochs=30,
validation_data = (valid_images, valid_labels), verbose=2)
```

```
# Resultados
FCN.evaluate(test_images, test_labels)
```

6.3.3. Modelo Red Neuronal Convolucional Le-Net-5 para el conjunto de datos de interés

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
import tensorflow_datasets as tfds
```

```
df, info = tfds.load('patch_camelyon', with_info = True, as_supervised =
↪ True)
```

```
train_data = df['train']
valid_data = df['validation']
test_data = df['test']
```

```
def preprocess(image, labels):
    image = tf.cast(image, tf.float32)
    image /= 255.
    return image, labels
```

```
train_data = train_data.map(preprocess)
valid_data = valid_data.map(preprocess)
test_data = test_data.map(preprocess)
```

```
buffer_size = 1000
train_data = train_data.shuffle(buffer_size)
```

```
batch_size = 128
train_data = train_data.batch(batch_size).prefetch(1)
valid_data = valid_data.batch(batch_size).prefetch(1)
test_data = test_data.batch(batch_size).prefetch(1)
```

```
train_images, train_labels = next(iter(train_data))
valid_images, valid_labels = next(iter(valid_data))
test_images, test_labels  = next(iter(test_data))
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten,
↪ BatchNormalization, Dropout
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
```

```
LeNet = Sequential()

  # Capa convolucional
LeNet.add(Conv2D(input_shape=(96,96,3), filters = 6, kernel_size = (5,5),
↪ activation = 'tanh', name='C1'))

  # Capa Max-pooing con tamaño 2x2
LeNet.add(AveragePooling2D(pool_size = (2,2), name='S2'))

  # Capa convolucional
LeNet.add(Conv2D(filters = 16, kernel_size = (5,5), activation = 'tanh', name
↪ = 'C3'))

  # Capa Max-pooling
LeNet.add(AveragePooling2D(pool_size = (2,2), name='S4'))

  # Antes de añadir capas totalmente conectadas debemos transformar la ultima
  ↪ salida en un vector
LeNet.add(Flatten())

  # Primera capa totalemente conectada
LeNet.add(Dense(120, activation = 'tanh', name='FC5'))

```

```
21   # Segunda capa totalmente conectada
22 LeNet.add(Dense(84, activation = 'tanh', name='F6'))
23
24   # Ultima capa
25 LeNet.add(Dense(1, activation = 'sigmoid', name='Output'))
26
27 LeNet.summary()
```

```
1 LeNet.compile(optimizer= 'adam', loss='binary_crossentropy', metrics=['acc'])
2
3 history = LeNet.fit(train_images, train_labels, batch_size=32, epochs = 30,
  ↪ validation_data = (valid_images, valid_labels), verbose=2)
```

```
1 LeNet.evaluate(test_images, test_labels)
```

```
1 acc = history.history['acc']
2 val_acc = history.history['val_acc']
3 loss = history.history['loss']
4 val_loss = history.history['val_loss']
5 epochs = range(1, len(acc) + 1)
6 plt.plot(epochs, acc, 'coral', label='Training accuracy')
7 plt.plot(epochs, val_acc, 'blueviolet', label='Validation accuracy')
8 plt.xlabel("Iteraciones")
9 plt.ylabel("Precision")
10 plt.title('Modelo LeNet-5')
11 plt.legend()
12 plt.savefig('LeNet_30ep_acc.pdf')
13 plt.show()
```

```
1 plt.figure()
2 plt.plot(epochs, loss, 'mediumaquamarine', label='Training loss')
3 plt.plot(epochs, val_loss, 'royalblue', label='Validation loss')
4 plt.xlabel("Iteraciones")
5 plt.ylabel("Perdida")
6 plt.title('Modelo LeNet-5')
7 plt.legend()
8 plt.savefig('LeNet_30ep_loss.pdf')
9 plt.show()
```

6.3.4. Modelo Red Neuronal Convolucional para el conjunto de datos de interés

```
1 # Importacion de librerias necesarias
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import tensorflow as tf
```

```
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
import tensorflow_datasets as tfds
```

```
ds, info = tfds.load('patch_camelyon', split='train', with_info=True)
info
```

```
# Cargar el dataset desde Tensorflow datasets
df, info = tfds.load('patch_camelyon', with_info = True, as_supervised =
 ↪ True)
```

```
# Dividir el conjunto de datos en entrenamiento, test y validacion
train_data = df['train']
valid_data = df['validation']
test_data = df['test']
```

```
# Procesamiento de los datos
def preprocess(image, labels):
    image = tf.cast(image, tf.float32)
    image /= 255.
    return image, labels
```

```
# Se hace uso de map() para aplicar la funcion anterior a cada elemento del
 ↪ conjunto de datos
train_data = train_data.map(preprocess)
valid_data = valid_data.map(preprocess)
test_data = test_data.map(preprocess)
```

```
# Barajamos el conjunto de entrenamiento
buffer_size = 1000
train_data = train_data.shuffle(buffer_size)
```

```
batch_size = 128
train_data = train_data.batch(batch_size).prefetch(1)
valid_data = valid_data.batch(batch_size).prefetch(1)
test_data = test_data.batch(batch_size).prefetch(1)
```

```
# Separamos las imagenes y etiquetas en dos variables
train_images, train_labels = next(iter(train_data))
valid_images, valid_labels = next(iter(valid_data))
test_images, test_labels  = next(iter(test_data))

# Comprobacion de las dimensiones
print(valid_labels.shape)
print(train_images.shape)
```

```python
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten,
    BatchNormalization, Dropout
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
```

```python
# Definimos el modelo de aprendizaje
model = Sequential([
        Conv2D(120, 5,padding='same', activation='relu', input_shape = [96,
            96, 3]),
        MaxPooling2D(2),
        Conv2D(256, 3,padding='same',activation='relu',),
        MaxPooling2D(2),
        Conv2D(412, 3,padding='same',activation='relu',),
        MaxPooling2D(2),
        Conv2D(412, 3,padding='same',activation='relu',),
        MaxPooling2D(2),

        Flatten(),
        Dense(412,activation = 'relu'),
        Dense(256,activation = 'relu'),
        Dense(128, activation = 'relu'),
        Dense(1, activation = 'sigmoid'),
    ])
model.summary()
```

```python
# Cargar el modelo
model.compile(optimizer= optimizers.Adam(1e-4), loss='binary_crossentropy',
    metrics=['acc'])

early_stopping_cb = keras.callbacks.EarlyStopping(monitor='val_loss',
    patience=5)

# Entrenamiento
history = model.fit( train_images, train_labels, batch_size=128, epochs = 50,
    callbacks=[early_stopping_cb], validation_data = (valid_images,
    valid_labels), verbose=2)
```

```python
# Evaluar el modelo en el conjunto de test
model.evaluate(test_images, test_labels)
```

```python
# Evaluar el modelo en el conjunto de entrenamiento
model.evaluate(train_images, train_labels)
```

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'coral', label='Precision conj. entrenamiento')
plt.plot(epochs, val_acc, 'blueviolet', label='Precision conj. validacion')
plt.xlabel("Iteraciones")
plt.ylabel("Precision")
plt.title('Precision para el conjunto de entrenamiento y validacion')
plt.legend()
plt.savefig('results/Modelo_Final_acc.pdf')
plt.show()
```

```
plt.figure()
plt.plot(epochs, loss, 'mediumaquamarine', label='Perdida conj.
    entrenamiento')
plt.plot(epochs, val_loss, 'royalblue', label='Perdida conj. validacion')
plt.xlabel("Iteraciones")
plt.ylabel("Perdida")
plt.title('Perdida para el conjunto de entrenamiento y validacion')
plt.legend()
plt.savefig('results/Modelo_Final_loss.pdf')
plt.show()
```

```
model.save("results/patchcamelyon.hf5")
```

6.3.5. Código para la visualización del conjunto de datos

Enlace

```
!pip install TensorFlow
!pip install keras
```

```
pip install tensorflow-datasets
```

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
import tensorflow_datasets as tfds
```

```
tfds.list_builders()
```

```
ds = tfds.load('patch_camelyon', split='train', shuffle_files=True)
assert isinstance(ds, tf.data.Dataset)
```

```
print(ds)
```

```
ds, info = tfds.load('patch_camelyon', split='train', with_info=True)

tfds.as_dataframe(ds.take(20), info)
```

```
ds, info = tfds.load('patch_camelyon', split='train', with_info=True)

fig = tfds.show_examples(ds, info)
```

Bibliografía

[1] Bhat, R.: *Gradient Descent With Momentum - Towards Data Science, Octubre, 3, 2020*, Enlace.

[2] Bushaev, V.: *Understanding RMSprop - faster neural network learning - Towards Data Science, Septiembre, 2, 2018*, Enlace.

[3] Cover, T.M. y Thomas, J.A: *Elements of Information Theory, Segunda edición, Wiley, ISBN-13 978-0-471-24195-9* .

[4] Devroye L., Györfi L. y Lugosi G. : *A probabilistic theory of pattern recognition, 1996, Springer. ISBN 0-3879-4618-7.*

[5] Godoy, D.: *Understanding binary cross-entropy / log loss: a visual explanation- Towards Data Science, Noviembre, 21, 2018*, Enlace.

[6] Goodfellow, I. , Bengio, Y. y Courville, A.: *Deep Learning. MIT Press, 2016*, http://www.deeplearningbook.org.

[7] He, K. Zhang, X. Ren, S. y Sun, J.: *Identity Mappings in Deep Residual Networks. Microsoft Research, 2016*, https://arxiv.org/abs/1603.05027.

[8] Huang, G. , Liu, Z. , Weinberger, K.Q. y Van der Maaten, L.: *Densely Connected Convolutional Networks. 2018*, https://arxiv.org/abs/1608.06993.

[9] LeCun, Y. , Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. y Jackel, L.D.: *Backpropagation Applied to Handwritten Zip Code Recognition. AT&T Bell Laboratories Holmdel, NY 07733 USA.*

[10] LeCun, Y. , Bottou, L., Bengio, Y. y Haffner, P.: *Gradient - Based Learning Applied to Document Recognition. Proc. of the IEEE, Noviembre 1998.*

[11] Liu, Y., Gadepalli, K., Norouzi, M., Dahl, G.E., Kohlberger, T., Boyko, A., Venugopalan, S., Timofeev, A., Nelson, P.Q., Corrado, G.S., Hipp, J.D, Peng, L. y Stumpe, M.C.: *Detecting Cancer Metastases on Gigapixel Pathology*

Images, Google Brain, Google Inc, Verily Life Sciences, Mountain View, CA, USA, Marzo 2017.

[12] Mitchell, T. M.: *Machine Learning. McGraw-Hill International Editions, 1997.*

[13] Rosenblatt, F.: *The Perceptron: A probabilistic model for information storage and organization in the brain. Cornell Aeronautical Laboratory, Psychological Review Vol. 65, No. 6, 1958.*

[14] Shalev-Shwartz, S. y Ben-David, S.: *Understanding Machine Learning: From Theory to Algorithms. Published 2014 by Cambridge University Press, ISBN 9781107298019.*

[15] Simonyan, K. y Zisserman, A.: *Very Deep Convolutional Networks for Large-Scale Image Recognition. Visual Geometry Group, Department of Engineering Science, University of Oxford, 2015*, https://arxiv.org/abs/1409.1556

[16] Tsang, SH.: *Review: AlexNet, CaffeNet — Winner of ILSVRC 2012 (Image Classification)- Medium - Agosto, 9, 2018*, Enlace.