UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

GRADO EN MATEMÁTICAS

CURSO 2020/21

Introducción al ML y redes neuronales

Diego Coello de Portugal Mecke

Tutor: Carlos Gregorio Rodríguez

30 de junio de 2021

Resumen

El objetivo del presente trabajo es el estudio de las Redes Neuronales, desde los conceptos básicos del Machine Learning hasta las estructuras subyacentes de las Redes Neuronales Profundas.

Durante el desarrollo del presente trabajo se irán introduciendo los conceptos y nociones básicas del Machine Learning, desde el tipo de datos a trabajar hasta los posibles métodos de resolución.

Utilizando estas nociones del Machine Learning, se presentarán las Redes Neuronales con una introducción histórica y explicación de sus primeros modelos. Del modelo *Multilayer Perceptron* se realizará una implementación en el lenguaje *Python*, la cual se utilizará en la resolución de diversos problemas de clasificación de datos.

A continuación, se analizarán las Redes Neuronales Profundas, estudiando sus distintos optimizadores, su impacto en el rendimiento de las mismas y la aparición de problemas como el overfitting o la explosión/desvanecimiento del gradiente.

Por último, utilizaremos estos conceptos de las Redes Neuronales Profundas y el paquete *Keras* en *Python* con diversos modelos de Redes Neuronales Profundas para abordar distintos problemas que requieran de estructuras más complejas para ser resueltos.

Abstract

The purpose of this document is to analyse the Neural Networks, from Machine Learning's basic concepts to the underlying structures of Deep Neural Networks.

During the development of the Machine Learning topic, the concepts and basic notions will be introduced step by step, starting with the type of data to work with till the different resolution methods.

Considering these notions, the Neural Network subject will be presented including an historical introduction and explanation of its firsts models. In order to solve different classification problems, an implementation of the *Multilayer Perceptron* model will be done in *Python* language.

Afterwards, the Deep Neural Networks and their diverse optimizers will be analyzed, as well as the way they modify their performance and the occurrence of problems such as overfitting or exploding/vanishing gradients.

Finally, the mentioned concepts of the Deep Neural Network and the *Keras* package in *Python* will be used with various Deep Neural Network models to analyse different problems which require more complex structures in order to be solved.

Índice

1. Introducción y motivación del trabajo

1.1. ¿Por qué usar Machine Learning?

Existen problemas computacionales cuya resolución precisa de algoritmos que sean capaces de procesar información compleja de analizar y actualizarse cada cierto tiempo. Uno de estos problemas es la detección de correos "spam".

Un algoritmo programado para detectar dichos correos podría considerar palabras clave como "gratis", "tarjeta de crédito" e "increíble" entre otras. También podría utilizar otros parámetros como la dirección de envío, la estructura del mensaje u otros elementos.

Dicho programa clasificará un mensaje como "spam" cuando detecta una cantidad determinada de parámetros predefinidos asociados a correos "spam". Las reglas y parámetros que debe tener en cuenta el programa son difíciles de establecer. Sin embargo, cambiar en los correos parámetros como las dirección, la estructura y/o el vocabulario empleado si se quiere "engañar" al programa no es tan complicado. De este modo, actualizar el programa para detectar dichos cambios en los correos "spam" es bastante más complejo y costoso que su contrapartida.

Si se usa un programa del tipo Machine Learning para resolver este problema, dicho programa se centraría en encontrar esas palabras, estructuras o elementos empleados en los correos del tipo "spam" que los distingue de los demás. De hecho, independientemente de las técnicas empleadas para escribir los mensajes solo se necesita añadir ejemplos con nuevos casos para poder clasificarlos como "spam", facilitando no solo la búsqueda de parámetros que determinen si un correo es "spam" o no, sino también su capacidad de mantenerse actualizado.

El uso del Machine Learning destaca por resolver problemas relacionados con la interpretación del lenguaje hablado o *Natural Language Processing*. Si se quiere diferenciar palabras "uno" y "dos" bastaría con ver si la palabra empieza con el sonido que el programa asocie al valor "u". Sin embargo, a medida que se vayan añadiendo más palabras se vuelve más complejo diferenciarlas entre sí. Apareciendo sonidos difíciles de distinguir como "v" y "b" o palabras cuya pronunciación es esencialmente la misma y dependen del contexto como "hola" y "ola".

Cabe destacar la complejidad añadida si hay que discernir la lengua empleada.

Este procedimiento termina siendo inabarcable de codificar manualmente a medida que vaya creciendo la cantidad de palabras a clasificar. Por ende, se opta por emplear técnicas de Machine Learning que resuelvan el problema mediante una gran cantidad de ejemplos.

1.2. Objetivos

El objetivo del presente trabajo es realizar una introducción a las Redes Neuronales, describiendo sus conceptos básicos y la aplicación de modelos básicos a diversos problemas.

Este trabajo partirá desde lo básico de Machine Learning, pasando por los primeros modelos de Redes Neuronales y terminando con APIs que se emplearán para resolver problemas complejos, estudiando en cada etapa los procesos que van realizando los distintos algoritmos para resolver el problema asociado.

Este trabajo se dividirá en cuatro secciones:

- Estudio de la estructura básica de los algoritmos de Machine Learning. Para ello se tratarán los distintos tipos de algoritmos de Machine Learning, diferenciándose en el uso o no de datos etiquetados, su capacidad de actualizarse, etc. Además, se estudiará la importancia tanto de los datos usados en el modelo y como la del modelo empleado en la resolución del problema.
- Estudio básico de las Redes Neuronales. Se realizará un análisis histórico introductorio, examinando los modelos *Perceptron* y *Multilayer Perceptron*. Con estos conocimientos se llevará a cabo una implementación del modelo de Red Neuronal *Multilayer Perceptron*, utilizándolo para resolver diversos problemas de clasificación.
- Estudio de las Redes Neuronales Profundas. En dichas redes se presentan problemas asociados al algoritmo de forma mucho más pronunciada que en aquellas que no son profundas (Desvanecimiento/Explosión del gradiente, overfitting, tiempo de entrenamiento demasiado costoso, etc). Se estudiará el origen de dichos problemas y se presentarán diversos métodos de resolución de los mismos.

- Uso de APIs. Estas implementaciones se utilizarán para tratar problemas complejos que no puedan ser solucionados sin el uso de Redes Neuronales Profundas, poniendo en uso los conceptos tratados en las secciones anteriores.

1.3. Machine Learning en la actualidad

El Machine Learning es un tema recurrente en los medios de comunicación del que se escuchan grandes hazañas: procesamiento del lenguaje natural con GPT-3 [1], análisis de imágenes para aumentar la resolución o superesolución [2] y pasarlas de blanco y negro a color, desarrollo de modelos como AlphaGo [3] que sean capaces de competir y superar profesionales del Go, etc.

Estos avances se han vuelto más crecientes durante la última década y prometen ser aún mayores en la venidera. El desarrollo de chatbots que proporcionen conversaciones que no se puedan distinguir de las humanas o algoritmos que sean capaces incluso de programar, es una realidad de un futuro cada vez más cercano gracias a las diversas técnicas de Machine Learning.

El Machine Learning se utiliza para resolver problemas con una gran cantidad de datos a analizare, que requieren actualizarse de forma constante y/o cuyas soluciones no se conocen. Los enfoques tradicionales no aportan soluciones óptimas para problemas donde el tamaño del problema crece o requieren de unas reglas complejas y difíciles de discernir, requiriendo nuevas técnicas para encontrar soluciones.

Por último, algunos ejemplos de problemas que se pueden resolver con el Machine Learning son:

- Clasificación de imágenes por productos, especies, etc.
- Detección de tumores con procesamiento de imágenes.
- Clasificación de textos en comentarios ofensivos, valoraciones, etc.
- Creación de un chatbot o asistente personal.
- Detección de comandos de voz.
- Clasificación de clientes/usuarios por preferencias basado en su historial para recomendaciones, anuncios, etc.

2. Machine Learning

Desde una perspectiva técnica se puede describir al Machine Learning como la rama de la computación que se dedica al desarrollo de programas destinados a aprender la realización de una tarea a partir de datos; es decir, a desarrollar la habilidad computacional de aprender sin ser programados de forma explícita [4].

Dicho de una manera más teórica, un programa aprende a realizar una tarea T, con una experiencia E y una medida de rendimiento R si su desempeño en T medido con R mejora con la experiencia E.

En esta sección se utilizará como referencia el libro *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow* [18].

Las principales diferencias que se dan en los algoritmos de Machine Learning son:

- El tipo de datos con los que son entrenados (aprendizaje supervisado, no supervisado, semisupervisado o reforzado).
- La capacidad de aprender de forma incremental, es decir, si pueden aprender nuevos datos (online) o requieren empezar de cero (batch learning).
- Si comparan los datos con otros obtenidos anteriormente (instance-based learning) o detectan patrones en los datos y crean un modelo predictor (model-based learning).

2.1. Aprendizaje supervisado

Este tipo de algoritmos de Machine Learning se caracterizan por utilizar datos etiquetados, es decir, datos con un parámetro extra asociado que corresponde al valor que se desea predecir. Uno de sus principales objetivos es la clasificación de datos, por ejemplo un filtro de correos "spam". Aunque también pueden usarse en problemas de regresión para predecir valores como edad, precio, temperatura, ...

Algunos ejemplos de algoritmos con aprendizaje supervisado son:

- Redes Neuronales
- K-Nearest Neighbours (KNN)
- Árboles de decisión
- Regresión lineal

En el caso de las Redes Neuronales también se pueden usar para casos no supervisados (*autoencoders*) o semisupervisado (*deep belief networks*).

2.2. Aprendizaje no supervisado

En este tipo de aprendizaje los datos no están etiquetados, es decir, los datos no tienen el valor que se desea predecir asociado. Generalmente estos algoritmos se centran en agrupar los datos según su similitud, generando particiones de los datos.

Hay varios problemas que se pueden resolver:

- Clasificación: K-Means, DBSCAN
- Detección de anomalías: One-Class SVM, Isolation Forest
- Visualización y reducción de dimensionalidad: Análisis de Componentes Principales (PCA), Locally Linear Embedding (LLE)
- Aprendizaje de asociación de reglas: Apriori, Eclat

2.3. Aprendizaje semisupervisado

El aprendizaje semisupervisado es un caso intermedio entre los algoritmos supervisados y no supervisados, debido a que solo una fracción de los datos están etiquetados. Estos algoritmos suelen usarse cuando etiquetar los datos requiere mucho tiempo y/o presupuesto, por lo que se decide etiquetar solo algunos casos para minimizar costes. Un ejemplo es Google Photos, que una vez que se han subido fotos con distintas personas reconoce automáticamente las fotos en las que aparece cada una de ellas.

La mayoría de los algoritmos semisupervisados combinan técnicas de los no supervisados y los supervisados. Un algoritmo semisupervisado sería las *deep belief networks* (DBNs) basado en *Restricted Bolztmann Machines* (RBMs), un algoritmo no supervisado.

2.4. Aprendizaje reforzado

Estos algoritmos se emplean en situaciones en las que se puede simular el sistema asociado al problema. El sistema de aprendizaje se denomina "agente" y observa el "ambiente" para decidir qué acción ejecutar. Para cada acción se le da una recompensa (o castigo) en función de la decisión que ha tomado. De este modo aprende cual es el procedimiento a seguir para maximizar la recompensa.

Un ejemplo de este tipo de algoritmo reforzado es *DeepMind's AlphaGo*, que recibe la puntuación de la partida como recompensa. A base de analizar millones de partidas aprende las acciones óptimas y consigue ganar al campeón mundial Ke Jie.

2.5. Aprendizaje Batch/offline

Un algoritmo de Machine Learning se dice que es del tipo "aprendizaje batch" o "aprendizaje offline" si no puede aumentar su aprendizaje, es decir, que una vez entrenado no se le puede añadir nuevos datos de entrenamiento, sino hay que construir una nueva versión y entrenarla desde cero con los nuevos datos añadidos al dataset de entrenamiento.

El proceso de crear una nueva versión y sustituir la antigua cada cierto tiempo se puede automatizar. Sin embargo, a medida que crezca la cantidad de datos almacenados aumentará el tiempo que necesita para entrenarse el modelo y los recursos computacionales (CPU, espacio de memoria, espacio del disco, etc.), llegando a ser inviable para problemas que requieran un modelo que se adapte a un sistema que cambia con facilidad.

2.6. Aprendizaje online

Los algoritmos de "aprendizaje online" se caracterizan por admitir nuevos datos de entrenamiento, es decir, que un algoritmo entrenado puede entrenarse en nuevos casos sin revisar los anteriores. Admite tanto datos individuales como nuevos pequeños grupos, denominados mini-batches.

En comparación al batch learning, el sistema aprende de forma más rápida y consumiendo menos recursos. Si los datos almacenados superan la capacidad de memoria de la máquina (out-of-core learning), se puede entrenar dividiendo el conjunto total de datos en distintas partes.

Un problema del aprendizaje online es que es más susceptible de bajar el rendimiento si recibe datos erróneos. Una herramienta para controlar el impacto de los nuevos datos es el uso de un hiperparámetro denominado "coeficiente de aprendizaje". Dicho hiperparámetro es un valor que determina la susceptibilidad del algoritmo a los nuevos datos recibidos, haciendo más susceptible el algoritmo cuando el hiperparámetro aumenta y reduciendo la susceptibilidad cuando disminuye.

El objetivo de esta variación en la sensibilidad es evitar el "ruido" o datos no representativos que se añaden al sistema.

2.7. Instance/Model based-learning

Un algoritmo de Machine Learning es del tipo *instance based-learning* si se basa en usar los datos proporcionados anteriormente y un sistema de medida entre datos. De este modo, el algoritmo predice el resultado para los nuevos datos usando únicamente dicha medida con respecto a los datos almacenados, asociando el nuevo dato con aquellos que sean más cercanos.

Por otro lado, los algoritmos de Machine Learning del tipo *model based-learning* tienen como objetivo generalizar a partir de los datos proporcionados, generando un modelo con el que hacer predicciones.

2.8. Importancia de los datos en el Machine Learning

A diferencia del aprendizaje humano donde con un ejemplo de una manzana o un libro podemos generalizar para identificar manzanas y libros de distintos tamaños y colores, los algoritmos de Machine Learning necesitan una gran cantidad de datos para generalizar de manera apropiada. Esta característica del Machine Learning crea la necesidad de usar datos representativos para su correcto desempeño independientemente del tipo de modelo.

En el artículo "The unreasonable effectiveness of data" [6], se mostró como la eficiencia de diversos algoritmos de Machine Learning para identificar el lenguaje hablado (*Natural Language Processing*) mejoraba de forma casi idéntica en todos ellos al aumentar la cantidad de datos. Los autores sugirieron la opción de sustituir la inversión en desarrollo de algoritmos por la obtención de datos.

A pesar de esta observación, hay problemas donde la cantidad de datos es limitada y de un tamaño pequeño (miles de datos) o medio (un millón de datos) y cuya ampliación puede resultar demasiado costosa como para llevarse a cabo.

2.9. Importancia del modelo escogido

La representatividad de los datos es un factor importante al entrenar un modelo de Machine Learning, sin embargo, es igual de importante escoger un modelo que se adecue al problema y los datos. Algunos de los principales problemas que pueden surgir son el *overfitting* y el *underfitting*.

El *overfitting* ocurre cuando el modelo tiene una exagerada consideración de los datos utilizados, es decir, intenta aproximarse tanto a ellos que termina afectando a la generalización. Esto se debe a que el modelo no solo tiene en cuenta la estructura de los datos, sino también el "ruido" y/o los datos erróneos.

Suele darse cuando se escoge un modelo demasiado complejo con respecto a la estructura de los datos, pudiendo detectar los patrones subyacentes de los datos con facilidad. Si los datos tienen demasiado "ruido" o el tamaño de la muestra es muy pequeño, es probable que el modelo generalice ese ruido como parte de la estructura intrínseca de los datos.

Una solución sería usar un modelo más simple o añadir más datos. En este último caso, la mejora se debe a la aparición de nuevos datos que carecen del "ruido" o errores del anterior dataset de entrenamiento.

El *underfitting* es el caso contrario al *overfitting*. Ocurre cuando el modelo es demasiado simple para aprender la estructura subyacente de los datos. Generalmente se da cuando escogemos algoritmos lineales o cuadráticos para problemas complejos como el tratamiento de imágenes o el procesamiento del lenguaje natural (*Natural Languaje Processing*).

Algunas opciones para solucionar el *underfitting* son:

- Escoger un modelo más potente con más parámetros.
- Adaptar las características del modelo al problema.
- Reducir las restricciones del modelo.

3. Redes Neuronales

3.1. Introducción histórica

El origen de las Redes Neuronales Artificiales (ANNs) está en la arquitectura del cerebro humano. Sin embargo, las ANNs terminan diferenciándose de su inspiración biológica al igual que los aviones de los pájaros. Las ANNs son la base del Deep Learning gracias a su versatilidad y potencia, siendo idóneas para tratar problemas complejos del Machine Learning como el análisis de imágenes, procesamiento del lenguaje natural, movimientos óptimos en juegos como el GO, etc.

Uno de los primeros artículos sobre las ANNs fue *A Logical Calculus of Ideas Immanent in Nervous Activity* [5], donde se presenta una versión computacional simplificada de como las neuronas biológicas podrían trabajar para generar cálculos complejos sobre lógica proposicional.

3.1.1. Perceptron

Una de las estructuras más simples de ANNs fue el *Perceptron*, inventada por Frank Rosenblatt en 1957 [7]. Se basa en el uso de unas neuronas artificiales denominadas *threshold logic unit* (TLU) o *linear threshold unit* (LTU). Dichas estructuras reciben un vector de entrada $\mathbf{x}$, al cual ejecutan una suma ponderada ($z = w^T x = w_1 x_1 + w_2 x_2 + ... + w_n x_n$), cuyo resultado se aplica a la *función escalón* (*step*) tal que:

$$h_w(x) = step(z) = step(w^T x)$$

donde la función escalón pueden ser *heaviside(z)* o *sgn(z)*:

$$heaviside(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad sgn(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases}$$

En este caso se ha considerado que el punto crítico sea 0. También se puede añadir un parámetro extra $x_0 = 1$ (*bias neuron*), entrenando de este modo el valor independiente w_0 asociado al punto crítico.

El *Perceptron* es una arquitectura formada por una capa de TLUs con todas ellas conectadas a todos los valores de entrada, obteniendo como resultado un vector con cada valor del mismo correspondiente a cada una de las distintas TLUs. Cuando una capa tiene todas sus neuronas conectadas a la capa anterior (como en el caso del *Perceptron*), decimos que es una capa *totalmente conexa* o *densa*.

El método de entrenamiento del *Perceptron* propuesto por Rosenblatt es de la forma:

$$w_{i,j}^{(\text{siguiente paso})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

cuyos parámetros definimos como:

- $w_{i,j}$: peso de la conexión entre el valor de entrada i y la neurona j.
- η: coeficiente de entrenamiento.
- y_j: j-ésimo valor de salida deseado.
- $\hat{y}_j$: j-ésimo valor de salida obtenido.
- x_i: i-ésimo valor de entrada.

Esta primera propuesta, aunque incapaz de aprender patrones complejos (clasificadores de regresión logística), puede clasificar conjuntos linealmente separables debido a que la frontera de decisión de cada neurona es lineal (*Teorema de la convergencia del Perceptron*[8]).

En 1969 Marvin Minsky y Seymour Papert presentaron en su libro *Perceptrons: an introduction to computational geometry* [9] una serie de problemas que presentaba el Perceptron, destacando la incapacidad de resolver el problema de clasificación del XOR o OR exclusivo.

A pesar de la aparición de nuevas arquitecturas y mejores algoritmos de entrenamiento a inicios de los 80, en los años 90 se desarrollaron nuevos algoritmos de ML de gran capacidad como *Support Vector Machines* (SVM) ralentizando el uso de las ANNs hasta finales de los 90.

3.1.2. Multilayer Perceptron y Backpropagation

Para solucionar la clasificación en el problema del XOR o OR exclusivo, se pueden apilar varias capas de TLUs, generando lo que se denomina un *Perceptron Multicapa* (MLP).

La arquitectura de los MLP está formada por una *capa de salida* y una o más *capas ocultas*, todas ellas conformadas por TLUs. Se les atribuye un cierto orden considerando las capas más cercanas a la entrada como *capas bajas*, y las más cercanas a la salida como *capas altas*, fluyendo la información en dirección ascendente. Cuando una ANNs tiene una gran cantidad de capas ocultas se la denomina *Redes Neuronales Profundas* (DNN).

Con el MLP la cantidad de variables aumenta considerablemente, por lo que se utiliza en este caso la siguiente notación:

- x_k: vector dado por entrada a la capa k.
- $x_{k,i}$: valor i-ésimo del vector dado por entrada a la capa k.
- w_k: matriz de pesos que relaciona la capa k con la capa k+1.
- $w_{k,i,j}$: valor de peso asociado de la neurona j-ésima de la capa k a la neurona i-ésimo de la capa k+1.
- z_k: $= w_k * x_k$.
- $z_{k,j}$: valor j-ésimo del vector z_k.
- h_k: función de activación de las neuronas de la k-ésima capa. Las neuronas de una misma capa tienen la misma función de activación.
- η: coeficiente de aprendizaje.
- $\hat{y}$: vector predicho por el MLP para el valor de entrada.
- $\hat{y}_i$: valor i-ésimo del vector $\hat{y}$.
- y: vector deseado/etiquetado para el valor de entrada.
- y_i: valor i-ésimo del vector y.
- $C_{k,i}$: $= \frac{\partial Coste}{\partial z_{k,i}}$. No es un parámetro con un valor semántico intrínseco, pero su uso en el algoritmo de Backpropagation facilitará los cálculos e implementación.

A pesar de la estructura del MLP que permite solucionar problemas que su predecesor era incapaz de solucionar, su método de entrenamiento era incierto hasta 1985 cuando David Rumelhart, Georffrey Hinton y Ronald Williams introdujeron el algoritmo de entrenamiento *Backpropagation* [10] basado en el *Gradient Descent*. El objetivo del *Backpropagation* es usar de forma hábil la regla de la cadena para ver el aporte que ha hecho al error el peso de cada conexión y ambiente.

Primeramente se define la función de Coste, en este caso se considera el Coste como el error cuadrático medio:

$$Coste = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

para cada peso $w_{k,i,j}$ de una neurona en la última capa la derivada del Coste es:

$$\frac{\partial Coste}{\partial w_{k,i,j}} = \frac{\partial Coste}{\partial \hat{y}_i} * \frac{\partial \hat{y}_i}{\partial z_{k,i}} * \frac{\partial z_{k,i}}{\partial w_{k,i,j}} =$$

$$= \frac{2}{m}(\hat{y}_i - y_i) * h'_k(z_{k,i}) * x_{k,i} = C_{k,i} * x_{k,i}$$

y la forma de actualizar dicho peso es:

$$w_{k,i,j}^{(\text{actualizado})} = w_{k,i,j}^{(\text{no actualizado})} - \eta * \frac{\partial Coste}{\partial w_{k,i,j}}$$

Para actualizar las capas anteriores se usan los valor $C_{k+1,i}$ ya calculados. De este modo se puede calcular la derivada apoyándose en las capas previamente calculadas.

$$\frac{\partial Coste}{\partial w_{k,i,j}} = \frac{\partial Coste}{\partial x_{k+1,i}} * \frac{\partial x_{k+1,i}}{\partial z_{k,i}} * \frac{\partial z_{k,i}}{\partial w_{k,i,j}} =$$

$$= \sum_{l=1}^{m}(C_{k+1,l} * w_{k+1,i,l}) * h'_k(z_{k,i}) * x_{k,i} =$$

$$= \frac{\partial Coste}{\partial z_{k,i}} * x_{k,i} = C_{k,i} * x_{k,i}$$

Este método, permite usar de forma efectiva la potencia que nos da el MLP. Sin embargo, las funciones no continuas dan problemas con la derivación.

Por este motivo la función escalón queda descartada y se crean nuevas candidatas a función de activación:

- Función logística o sigmoide:

  Propuesta por los autores del Backpropagation [10], su expresión es $\sigma(z) = \frac{1}{1+exp(z)}$. Fue muy utilizada en un inicio por su similitud con la función *heavyside*, una de las versiones de la función escalón.

- Función tangente hiperbólica:

  Su expresión es $tanh(z) = 2 * \sigma(2z) - 1$, siendo similar a la sigmoide y a *sgm* (otra versión de la función escalón), pero con valores de salida entre [-1,1] en vez de [0,1]. La media se encuentra en 0, lo que en la práctica puede ayudar en ciertos problemas a conseguir una convergencia más rápida.

- Función de unidad lineal rectificada (ReLU):

  Su expresión es $ReLU(z) = max(0, z)$. A pesar de no ser derivable en 0, se asocia a la derivada en 0 el valor correspondiente al límite izquierdo o derecho arbitrariamente. Cabe destacar su rápido cálculo y la eliminación de ciertos problemas que se verán más adelante la han convertido en la función de activación por defecto.

Por otro lado, los principales problemas a resolver por los MLP son:

- Regresión:

  El objetivo es predecir un/os valor/es (precio de una casa, temperatura, centro de un objeto, ...). En el caso de predicción de varios valores se denomina regresión multivariable.

  En general, se puede dejar la última capa sin función de activación para que el resultado pueda tomar cualquier valor,Aunque hay casos en los que interesa que sean positivos y se usa ReLU o softplus.

  $$softplus(z) = log(1 + exp(z))$$

  También hay otros casos en los que el resultado debe encontrarse en un determinado intervalo, para los que se usan la función logística o la tangente hiperbólica con un escalado.

- Clasificación:

  Como el nombre indica, el objetivo es distinguir el tipo de cada dato de entrada. El valor de salida es un valor entre 0 y 1, que se puede interpretar como la probabilidad de que los valores de entrada correspondan a dicha clase.

  La predicción de varias probabilidades independientes (por ejemplo si un correo es "spam" o no y si es urgente o no), no hay problema ya que los valores dados no deben sumar uno. Sin embargo, si la predicción de varios valores son dependientes y deben sumar 1, se emplea la función softmax al resultado obtenido.

$$softmax(x) = [\rho_1,...,\rho_k]^T$$

$$\rho_j = \frac{x_j}{\sum_{i=1}^{k} exp(x_i)}$$

3.2. Implementación del MLP

En esta sección se presenta una implementación del MLP en *Python*. Primeramente se define las funciones de activación y sus derivadas asociadas.

```
sigm = (lambda x: 1/(1 + np.e**(-x)),
        lambda x: sigm[0](x)*(1-sigm[0](x)))

relu = (lambda x: np.maximum(0, x),
        lambda x: np.argmax([0,x]))

tanh = (lambda x:  2/(1+np.e**(-2*x))-1,
        lambda x: 1-tanh[0](x)**2))
```

A continuación se implementan algunas funciones auxiliares asociadas a los costes y sus respectivos incrementos para la actualización de pesos.

```
mse = (lambda p,y: np.mean((p-y)**2),
       lambda p,y: 2*(p-y))

cross_entropy = (lambda p,y: -np.mean(y*np.log(p)),
                 lambda p,y: (p-y))
```

Con ayuda de estas definiciones se puede implementar las capas. Cada una de ellas tendrá una función de activación, un vector de valores independientes *bias* y una matriz de pesos.

```
class neural_layer():
    def __init__(self, n_conn, n_neur, act_f):
        self.act_f = act_f
        self.b = np.random.rand(1, n_neur)*2-1
        self.w = np.random.rand(n_conn, n_neur)*2-1
```

Esta clase contará con un método *predict()* para calcular los valores que se le pasan a la siguiente capa o en el caso de ser la última capa, el resultado del MLP. Se devuelve un par de valores en vez de un único valor para facilitar el proceso posterior de Backpropagation.

```
    def predict(self, X):
        z = np.dot(X,self.w) + self.b
        a = self.act_f[0](z)
        return (z,a)
```

Estas definiciones son los elementos necesarios para implementar un MLP, que se inicia con dos parámetros: la función de activación y la topología. El parámetro función de activación indica que función de activación va a tener cada capa y la topología será un vector con el tamaño de los datos de entrada seguido del número de neuronas en cada capa. Nótese que en caso de requerir de distintas funciones de activación para cada capa, se podría cambiar la inicialización del MLP para que recibiera solo la topología, siendo esta pares de número de neuronas y función de activación.

```
class neural_network():
    def __init__(self, topology, act_f):
        nn = []
        for l in range(len(topology[:-1])):
            nn.append(neural_layer(topology[l],\
            topology[l+1], act_f))
        self.layers = nn
```

Al igual que las capas, el MLP contará con un método *predict()* para calcular el resultado del vector de entrada tras pasarlo por todas las capas.

De forma análoga a la implementación de las capas, se devolverá una lista de pares para facilitar el posterior proceso de Backpropagation.

```
def predict(self, X):
    out = [(None, X)]
    for l, layer in enumerate(self.layers):
        z, a = layer.predict(out[-1][1])
        out.append((z,a))
    return out
```

Para finalizar la implementación del MLP se añade el método *train()* que permite entrenar el MLP dado un dataset de entrada y los valores a predecir asociados utilizando el *Backpropagation* como se ha mostrado anteriormente. Se puede añadir otro parámetro asociado a la función de coste a utilizar. En este caso se considera solo el *error cuadrático medio* o mse.

```
def train(self, X, Y_pred, learning_rate=0.01):
    out = self.predict(X)
    deltas =  []
    for l in reversed(range(len(self.layers))):
        z,a = out[l+1]
        if l == len(self.layers)-1:
            deltas.insert(0,mse[1](a,Y_pred)\
            *self.layers[l].act_f[1](z))
        else:
            deltas.insert(0, np.dot(deltas[0],w_prev.T)\
            *self.layers[l].act_f[1](z))
        w_prev = self.layers[l].w
        self.layers[l].b = self.layers[l].b\
        -np.mean(deltas[0],axis=0,keepdims=True)\
        *learning_rate
        self.layers[l].w = self.layers[l].w\
        -out[l][1].T @ deltas[0]*learning_rate
    return out[-1][1]
```

3.3. Resolución de problemas básicos

El primer problema que se plantea será la clasificación de puntos con respecto a dos circunferencias utilizando *make_circles()* de *sklearn.datasets*. De esta forma se obtiene el dataset que se muestra en la figura 1.

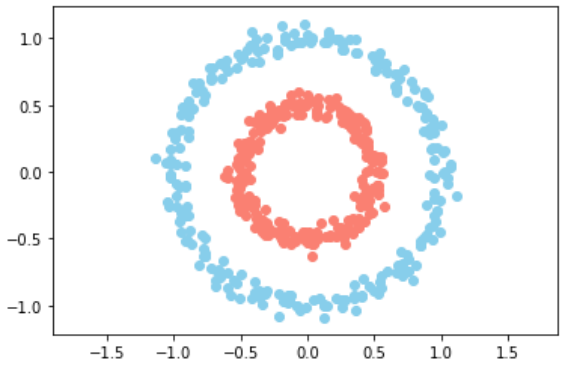


Figura 1: Dataset de circunferencias

Por otro lado, para ver el desarrollo de nuestro MLP se utiliza un mapa de calor del paquete *matplotlib.pyplot*. De este modo se obtiene para un MLP sin entrenar algo similar a la figura 2 dependiendo de los valores de inicialización aleatorios que se obtengan.

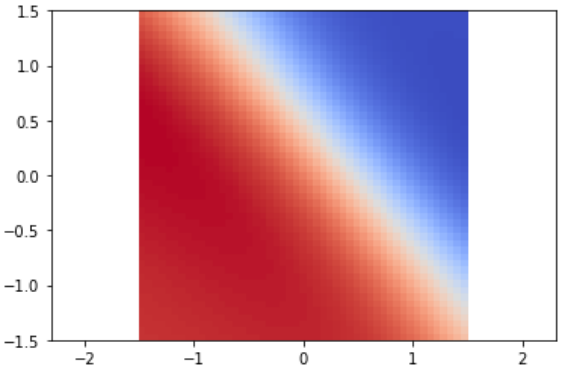


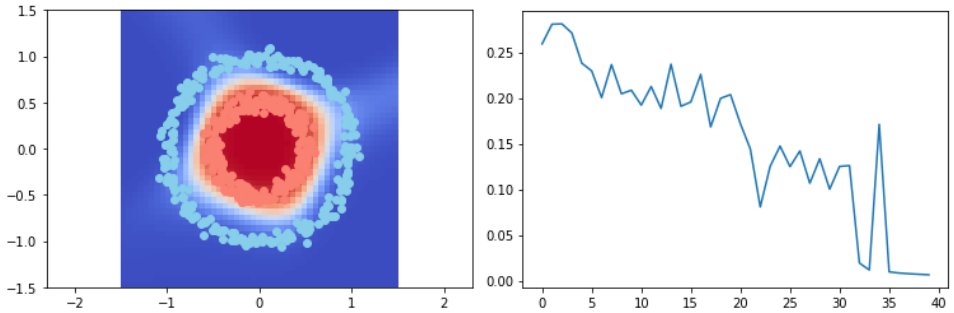
Figura 2: Mapa de calor para un MLP sin entrenar

En este caso se empleará un MLP de dos capas, guardando cada 25 epochs el error asociado y el mapa de calor correspondiente y una gráfica con el error del MLP en el dataset como se muestra en la figura 3.

```
topology = [2,4,1]
neural_net = neural_network(topology, sigm)
learning_rate = 0.05
loss = []
epoch = 500
for i in range(epoch):
    py = neural_net.train(X, Y,learning_rate)
    if i%25 == 0:
        loss.append(mse[0](py,Y))
        draw_circles(neural_net, X, Y)
        plt.plot(range(len(loss)), loss)
        plt.show()
        time.sleep(0.1)
```

Figura 3: Código de la implementación del problema de circunferencias.

Ejecutando las 500 iteraciones correspondientes, se observa que el MLP se comporta de forma apropiada al problema como se ve en la figura 4, con un tiempo de cómputo de 17 segundos.



(a) Mapa de calor del MLP entrenado (b) Gráfica de evolución del coste

Figura 4: Resultados del MLP entrenado

Cambiando los parámetros asociados a la función make_circles se observa que sigue adaptándose correctamente al problema independientemente de los valores, por lo que se incrementará la dificultad del problema para ver como se desenvuelve el MLP. Para este nuevo problema se usará un dataset de puntos en el plano con forma de espiral como se muestra en la figura 5.

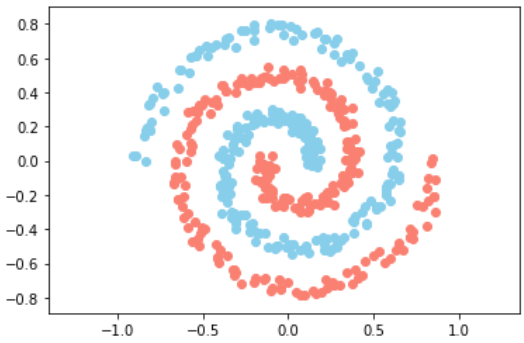


Figura 5: Dataset de espiral

Si se utiliza la misma estructura de MLP que en el caso del círculo, el MLP no es capaz de predecir el problema de forma satisfactoria. Este caso es un problema de *underfitting*, el modelo no es lo suficientemente potente para resolver esta tarea de forma apropiada. En este caso se decide cambiar la topología de nuestro modelo ampliando el número de parámetros. Además, se observa que con una topología del tipo *[n_inp,8,8,8,8,n_out]* se puede resolver el problema correctamente, con un tiempo de cómputo de 3 minutos y 55 segundos.

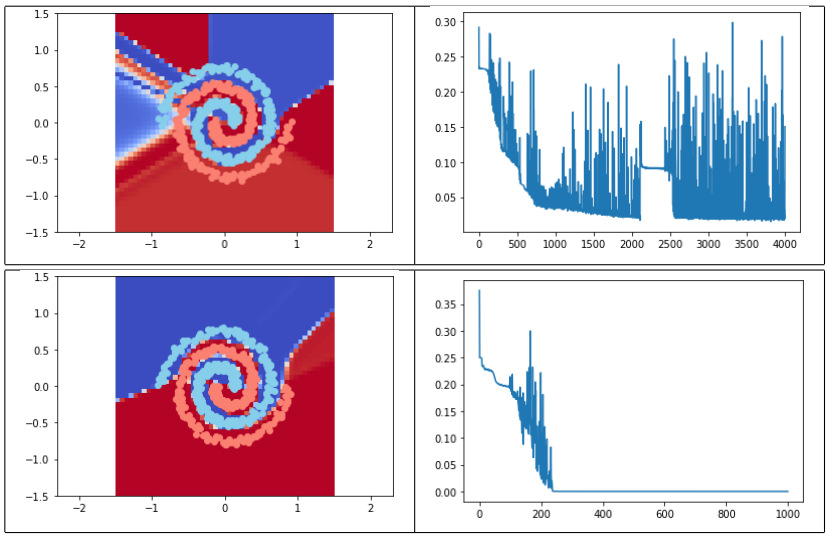


Figura 6: Diferencias entre la topología simple (fila superior) y la topología compleja (fila inferior).

En la parte superior de la figura 6 se muestra el desarrollo para la topología del MLP empleada en el problema del círculo. Las topologías con 4 o menos capas ocultas y 10 o menos neuronas por capa, no son capaces de alcanzar un error inferior al 1 % o terminan con secciones cruzando transversalmente los brazos de la espiral, como se ve en el caso de la topología simple en la figura 6.

Habiendo obtenido un resultado satisfactorio para el problema de la espiral se da paso al último problema que se va a tratar en este capítulo, donde se clasificará el dataset *Iris* facilitado por *sklearn.datasets*. En este caso se tiene 4 parámetros de entrada por lo que ya no se podrá utilizar el mapa de calor, aunque se seguirá empleando un registro del error cada 25 iteraciones. Además, la clasificación de este problema se realiza en 3 clases: *iris-versicolor*, *iris-virgínica* e *iris-setosa*. Por ende, se utilizarán etiquetas de la forma *one-hot-labeled*, es decir, se generan vectores de tamaño 3 (número de clases posibles) teniendo para cada posible clase un vector asociado con todos sus valores 0 salvo un 1 en la posición correspondiente a dicha clase.

Para comprobar la correcta obtención de una solución para este problema se dividirá el dataset en dos grupos: dataset de entrenamiento y dataset de prueba. El primero será el que se use para entrenar, mientras que el segundo se utilizará para obtener el error con respecto al mismo sin utilizarlo en ningún momento en su entrenamiento. De este modo se validará el modelo cuando el error sea menor que cierta cota arbitraria, como se muestra en la figura 7. El tiempo de cómputo es de 35 segundos.

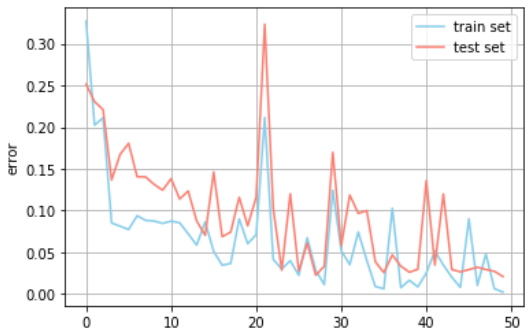


Figura 7: Error del MLP para el dataset de entrenamiento (azul) y dataset de prueba (naranja) con el dataset *Iris*.

4. Redes Neuronales Profundas

La principal característica de las Redes Neuronales Profundas (DNN) es no solo su gran número de capas sino su gran número de parámetros a entrenar. Esto permite que el modelo tenga una mayor flexibilidad y capacidad al abordar diversos problemas. Sin embargo, es esta misma capacidad de abordar los problemas que puede generar diversos inconvenientes como veremos a continuación.

4.1. Entrenamiento de DNN

En secciones anteriores ya se han visto métodos de entrenamiento para ANNs capaces de resolver distintos problemas sin necesidad de demasiadas capas. Sin embargo, a la hora de atacar problemas más complejos (como el tratamiento de imágenes) se requieren más capas para desarrollar los mecanismos adecuados asociados a la tarea. En estos casos pueden surgir una serie de problemas:

- Problema de explosión/desvanecimiento del gradiente:

  Esto sucede cuando el gradiente se vuelve más y más pequeño o más y más grande a medida que recorremos las distintas capas de la DNN hacia atrás en el entrenamiento (*Backpropagation*).

- Encontrar datos o etiquetarlos es demasiado costoso y no hay datos suficientes para el entrenamiento.

- El entrenamiento requiere demasiado tiempo.

- El gran número de parámetros aumenta la probabilidad de aparición del overfitting.

4.2. Problema de explosión/desvanecimiento del gradiente

Como se ha mencionado anteriormente, existe la posibilidad de que el gradiente decrezca a medida que el algoritmo progresa a las capas más bajas. Esto implica que los parámetros de dichas capas pueden terminar siendo "virtualmente constantes", es decir, que los valores se vuelvan tan pequeños durante el Backpropagation que no efectuen cambios en las capas más bajas. De este modo, el entrenamiento no convergiría a una solución satisfactoria. Este problema se conoce como el desvanecimiento del gradiente.

En otros casos sucede lo opuesto, el gradiente crece a medida que progresa a las capas más bajas sufriendo estas últimas cambios demasiado grandes, terminando con un algoritmo que diverge. Este problema se conoce como explosión del gradiente.

Estos problemas supusieron un decremento en el uso de las DNNs desde principios del siglo XXI hasta 2010, cuando Xavier Glorot y Yoshua Bengio escribieron *Understanding the difficulty of training deep feedforward neural networks* [11].

En dicho trabajo señalaron que el uso de la función logística y la inicialización de los pesos con una distribución normal de media 0 y varianza 1 (método estándar en aquel momento) podría generar problemas.

El primer problema era la acumulación de las varianzas según avanzaban las capas, además el hecho de que la media de la función logística sea 0.5 incentiva dicho aumento. El segundo problema era que la función logística se satura rápidamente a 0 o 1 cuando el valor crece o decrece, de modo que la derivada se aproxima a 0 rápidamente también. Debido al valor tan pequeño que puede tomar la derivada, el cambio en los pesos se puede diluir a medida que progresamos en el *Backpropagation*.

4.2.1. Inicialización de parámetros

En el documento mencionado anteriormente [11], Glorot y Bengio propusieron soluciones con respecto al problema de explosión/desvanecimiento del gradiente. La idea base es que los valores no desaparezcan o se diluyan al ir hacia adelante (predicción) o al ir hacia atrás (corrección con *Backpropagation*).

Para ello, la varianza de los valores de salida debe ser la misma que los de entrada. No es posible garantizarlo en ambas direcciones a no ser que la cantidad de valores de entrada (fan-in) y salida (fan-out) sean los mismos, pero propusieron un método que ha probado ser en la práctica suficientemente bueno, véase la figura 8. Dicho método se conoce como *Xavier initialitation* o *Glorot initialization*, en honor a sus autores.

$$\text{Distribución normal de media cero y varianza } \sigma^2 = \frac{1}{\text{fan}_{avg}}$$

$$\text{O una distribución uniforme entre -r y r, con r=}\sqrt{\frac{3}{\text{fan}_{avg}}}$$

$$\text{donde fan}_{avg} = \frac{(\text{fan}_{in} + \text{fan}_{out})}{2}$$

Figura 8: Xavier/Glorot initialization

Más adelante, se realizaron más estudios [12] para los que puntualmente se utiliza fan-in en vez de fan-avg. La estrategia de inicialización para la función ReLU y sus variantes (que se verá más adelante) es usar una distribución uniforme con $r = \sqrt{3\sigma}$, donde $\sigma^2 = \frac{1}{fan_{in}}$. Por otro lado, para la función de activación SELU (que se verá más adelante), se suele utilizar la inicialización de LeCun (distribución normal con $\sigma^2 = \frac{1}{fan_{in}}$). Se puede ver estas estrategias resumidas en el cuadro 1.

Inicialización	Funciones de activación	σ^2(Normal)
Glorot	Ninguna, tanh, logística, softmax	$\frac{1}{fan_{avg}}$
He	ReLu y variantes	$\frac{2}{fan_{in}}$
LeCun	SELU	$\frac{1}{fan_{in}}$

Cuadro 1: Inicialización de parámetros con distribuciones normales

4.2.2. Funciones de activación sin saturación

Por otro lado, el trabajo de Glorot y Bengio [12] menciona como otro motivo de la aparición de explosión/desvanecimiento del gradiente es la mala elección de las funciones de activación. Hasta este momento se consideraba la función sigmoide como la elección correcta asociada a la *Madre Naturaleza*, de modo que si funcionaba en dicho contexto funcionaría también para las ANNs. Sin embargo, resultó que funciones como ReLU funcionaban mucho mejor en DNNs al evitar la saturación y tener una rápida computabilidad.

Aún así, la ReLU también presenta algunos problemas como el *fallecimiento de ReLUs*. Este problema se refiere al momento durante el entrenamiento en el que algunas neuronas "mueren" de manera efectiva en el sentido de que el único valor que devuelven es 0 para los datos de entrenamiento. De hecho, hay ocasiones en que la mitad de las neuronas "mueren", de modo que al aplicar *Backpropagation* no hay variaciones debido a que la derivada es 0. Suele darse con mayor frecuencia en casos con un coeficiente de aprendizaje alto.

Para solventarlo, se pueden usar otras funciones de activación como *leaky ReLU*, cuya expresión es $LeakyReLU_\alpha(z) = max(\alpha z, z)$, donde α indica la cantidad de "fuga" para valores negativos (generalmente se suele escoger $\alpha = 0{,}01$). Este cambio asegura que la neurona nunca "muera" y que pueda permanecer realizando pequeñas aportaciones en el *Backpropagation* hasta que pueda "revivir". Al igual que con la ReLU, la derivada en 0 se escoge arbitrariamente como el límite izquierdo o derecho.

En 2015 se redacta un estudio [13] comparando la eficiencia de ReLU y sus variantes. En dicho documento se muestra como LeakyReLU supera a ReLU en la mayoría de los casos, además de mostrar que tomar $\alpha = 0{,}2$ para LeakyReLU puede ser más eficiente que $\alpha = 0{,}01$. También evaluó *leaky ReLU aleatoria* (RReLU), donde α se escoge de forma aleatoria en el entrenamiento, y se fija en el test. Por último, trata la *leaky ReLU parametrizada* (PReLU), donde α se aprende a lo largo del entrenamiento. Resulto ser mucho mejor para grandes datasets (en concreto de imágenes), pero suele caer en el 'overfitting' cuando los datasets son más pequeños.

En 2015 se propuso otra variante de la ReLU llamada *exponential linear unit* o de forma abreviada ELU. Dicha función consigue una reducción en el tiempo necesario para el entrenamiento y una mejor predicción en los experimentos hechos por los autores [14].

$$ELU_\alpha(z) = \begin{cases} \alpha(exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

Figura 9: Definición de la función ELU.

Las principales diferencias entre esta última y la ReLU son:

- Toma valores negativos para $z \leq 0$, lo que hace que la media sea más cercana a 0 y reduce la aparición del problema de desvanecimiento del gradiente.
- El gradiente para $z \leq 0$ es distinto de 0, evitando la "muerte" de las neuronas.
- Para $\alpha = 1$ la función es derivable en todo $\mathbb{R}$, acelerando el proceso al no pegar "saltos" alrededor de $z = 0$.

A pesar de todas estas ventajas, la ELU tiene un coste computación más elevado que su contraparte ReLU, por lo que el tiempo computacional obtenido al reducir el número de iteraciones se vea compensado por el coste computacional de cada iteración.

Siguiendo con este desarrollo de variantes de la ReLU, en 2017 se introdujo otra variante denominada Scaled ELU (SELU) [15]. Se define como la ELU multiplicada por un valor.

Los autores mostraron que aquellas ANNs con solo capas densas y con todas sus capas usando SELU, tienden a *auto-normalizarse*, es decir, el valor de cada capa tiende a media 0 y desviación 1.

Además se mostró que la SELU resulta ser mucho más efectiva (computacionalmente) que otras funciones de activación. Sin embargo, hay algunos requisitos para que pueda llevarse a cabo la auto-normalización:

- Los valores de entrada tienen que estar estandarizados, es decir, tener de media 0 y desviación 1.
- Todas las capas deben inicializarse con la inicialización normal de LeCun.
- La DNN tiene que tener una arquitectura secuencial.

En el estudio mencionado solo se asegura la *auto-normalización* si las capas son densas, pero otros investigadores han notado que puede funcionar también con otras arquitecturas como las *redes neuronales convolucionales*.

4.2.3. Normalización Batch

Usar la inicialización de He con ELU o cualquier otra variante de ReLU puede reducir en gran medida la aparición del problema de explosión/desvanecimiento del gradiente al principio del entrenamiento, pero no asegura que a lo largo del mismo no suceda.

En 2015 Sergey Ioffe y Christian Szegdy propusieron el uso de una técnica denominada Normalización Batch (BN) [16]. Esta técnica consiste en añadir una operación al modelo justo antes o después de aplicar la función de activación en cada capa oculta. Esta operación se basa en centrar en cero y normalizar cada input, luego lo multiplica por un escalar y lo desplaza usando dos nuevos parámetros por capa, de modo que el modelo tiene que aprender 2 valores adicionales para cada capa. En muchos casos, al aplicar la BN en la primera capa no se necesita estandarizar tus datos de entrenamiento.

Para realizar el centrado en 0 y normalizar los inputs, el algoritmo estima la media y desviación estándar usando el *mini-batch* actual, es decir, el subconjunto de los datos de entrenamiento que se está calculando en este momento. De esta forma se actualizan los parámetros como se muestra en la figura 10.

$$
\begin{aligned}
&1. \quad \mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)} \\
&2. \quad \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2 \\
&3. \quad \hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
&4. \quad z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta
\end{aligned}
$$

Figura 10: Ecuaciones del algoritmo de Normalización Batch.

Donde los parámetros de la figura 10 hacen referencia a:

- μ_B: vector de medias de valores de entrada, sobre el mini-batch B.
- σ_B^2: vector de desviación estándar, sobre el mini-batch B.
- m_B: número de datos en el mini-batch B.
- $\hat{x}^{(i)}$: i-ésimo vector centrado en 0 y normalizado.
- γ: valor escalar multiplicativo del vector de salida de la capa.
- $\otimes$: operación que representa la multiplicación de γ por cada elemento del vector $\hat{x}^{(i)}$.
- β: desplazamiento del vector de salida en dicha capa.
- ϵ: valor pequeño positivo para evitar división por 0 (generalmente se escoge 10^{-5}).
- $z^{(i)}$: i-ésimo vector de salida de la operación BN. Es la versión rescalada y desplazada del valor de entrada.

Este método efectivamente consigue estandarizar los valores de entrada. Sin embargo, hay casos donde se necesita hacer predicciones para individuos aislados en vez de para un batch. En el caso de que fuera un batch puede que los valores no sean independientes y que no estén idénticamente distribuidos, siendo poco fiable.

Una solución es entrenar la ANNs, ejecutar todos los datos en la ANNs y calcular la media y desviación estándar en la capa con BN, usando esta media y desviación calculadas para hacer predicciones.

A pesar de la estandarización y regulación que consigue la BN requiere de tiempo adicional, tanto en el entrenamiento como en las propias predicciones al añadir requisitos computacionales adicionales en cada capa. Una forma de solucionar esto es fusionándolo con la capa anterior de modo que en vez de computar $z = xw + b$ y $\gamma(z - \mu)/\sigma + \beta$, se junta todo tal que $w' = \gamma w/\sigma$ y $b' = \gamma(b - \mu)/\sigma + \beta$, con lo que solo se ejecuta $w'x + b'$.

4.3. Gradient Clipping

Otra opción para reducir la aparición del problema de explosión/desvanecimiento del gradiente es acotar el gradiente mientras se ejecuta el Backpropagation, de modo que los cambios no puedan sobrepasar cierta cota. Esta técnica se conoce como Gradient Clipping [17].

Generalmente se suele acotar entre [-1,1]. También existe la opción de acotar la norma del gradiente si se busca que no cambie la dirección del gradiente del vector.

4.4. Reutilizar Capas Preentrenadas

Entrenar una gran DNN desde cero ha mostrado no ser una buen enfoque en la práctica. Más óptimo sería apoyarse en una DNN entrenada para una tarea parecida y usar sus capas más bajas para la nueva DNN. Esta técnica se conoce como *transferencia de aprendizaje*. Con esto no solo aumenta la velocidad de aprendizaje, sino que el tamaño del dataset requerido será mucho menor.

Si el tamaño de los datos de entrada de la DNN entrenada no coincide con la nueva, se puede hacer un paso de preprocesado para adaptar el tamaño de la primera capa al de los datos de entrada. En general, la transferencia de aprendizaje funciona mejor cuando los datos tienen la misma estructura a nivel bajo. Las capas más altas suelen estar más especializadas en el problema en vez de obtener información de los datos de entrada. Además, el tamaño de los datos de salida puede ser distinto.

El número de capas que se reutilizan depende de la similitud entre los problemas. Se suele fijar las capas más bajas y dejar que aprendan solo las capas más altas. El número de capas que se fijan depende también de lo parecidos que sean los problemas y de la cantidad de datos disponibles. A mayor cantidad de datos, más capas se pueden dejar libres para entrenar sus parámetros.

4.5. Mejora de los optimizadores

El entrenamiento de DNNs puede llevar mucho tiempo. Hasta ahora se ha visto como mejorar su eficiencia con diversas técnicas: elegir buenas funciones de activación, usar una inicialización adecuada, Normalización Batch, usar capas ya entrenadas, etc. Sin embargo, otra forma de agilizar el proceso es sustituir el Gradient Descent por otro optimizador más rápido.

4.5.1. Momentum Optimization

La *Optimización del Momento* propuesto por Boris Polyak en 1964 [19], se basa en la idea de utilizar los cambios anteriores para la actualizacióndel peso.

Sea θ el parámetro que queremos actualizar. Le asignamos la variable $\mathbf{m}$ o *vector del momento* a dicho parámetro de modo que para la primera actualización $\mathbf{m}$ toma el valor $-\eta\frac{\delta C}{\delta\theta}(\theta)$, y actualizamos el parámetro θ de la forma $\theta = \theta + \mathbf{m}$.

Para las actualizaciones subsecuentes de θ utilizamos las ecuaciones mostradas en la figura 11. De modo que $\mathbf{m}$ se actualiza teniendo en cuenta su valor previo multiplicado por una constante β o *momento* cuyo valor se encuentra entre 0 y 1. Siendo 0 un gran "rozamiento" y 1 sin "rozamiento".

1. $\mathbf{m} = \beta\mathbf{m} - \eta\frac{\delta C}{\delta\theta}(\theta)$
2. $\theta = \theta + \mathbf{m}$

Figura 11: Ecuaciones del algoritmo de Optimizador del Momento.

Este optimizador consigue que para un valor de $\beta = 0{,}9$ donde el gradiente se mantiene constante, tome en su segunda iteración un valor 10 veces más grande que aplicando el descenso del gradiente. Esto permite escapar de *plateaus* mucho más rápido que su contraparte el Descenso del Gradiente.

Cabe destacar que este optimizador permite escapar de mínimos locales con mayor facilidad, pero su naturaleza de mantener el momento significa que cuando alcance el mínimo se pasará y tendrá que retroceder. Oscilando alrededor del mínimo antes de alcanzarlo.

4.5.2. Nesterov Accelerated Gradient

Como su nombre indica, el método de *Aceleración del Gradiente de Nesterov* fue propuesto por Yurii Nesterov en 1983 [20]. También es conocido como *momento de Neterov* y su idea principal es similar al Optimizador del Momento. Sin embargo, no evalua el gradiente para el parámetro θ, sino un poco más "adelante" como se muestra en la figure 12.

1. $m = \beta m - \eta \frac{\delta C}{\delta \theta}(\theta + \beta m)$
2. $\theta = \theta + m$

Figura 12: Ecuaciones de la Aceleración del Gradiente de Nesterov.

La idea intuitiva de este modelo es que el valor βm siempre se va a sumar. Por ende, se calcula la derivada en el punto anterior más dicho valor en vez de en el punto anterior. Lo que permite una mejor orientación hacia el mínimo global.

Esta pequeña mejora se aprecia mejor cuando θ no está en el punto más bajo del valle, pero $\theta + \beta m$ si se encuentra o incluso lo ha superado. En dicho caso no aumentaría el momento en la dirección que nos llevaría a subir el valle a diferencia del algoritmo de Optimización del Momento.

4.5.3. AdaGrad

Si consideramos el algoritmo del Descenso del Gradiente a lo largo del problema, se observa como coge la pendiente más marcada. Sin embargo, no siempre apunta al mínimo global.

El algoritmo de AdaGrad [21] intenta apuntar con más precisión hacia el mínimo global. En la figura 13 se muestra las ecuaciones del algoritmo, donde $\otimes$ representa la multiplicación de matrices por elementos, $\oslash$ la división de matrices por elementos y ϵ es un pequeño valor positivo para evitar división por 0.

1. $s = s + \frac{\delta C}{\delta \theta}(\theta) \otimes \frac{\delta C}{\delta \theta}(\theta)$
2. $\theta = \theta - \eta \frac{\delta C}{\delta \theta} \oslash \sqrt{s + \epsilon}$

Figura 13: Ecuaciones del algoritmo AdaGrad.

Este algoritmo reduce el cambio para pendientes marcadas mucho más que para aquellas no pronunciadas. Esto recibe el nombre de *escala de aprendizaje adaptativa*, lo que permite orientar los cambios con más precisión hacia el mínimo global y permite una mayor flexibilidad con el coeficiente de aprendizaje del modelo η.

AdaGrad genera buenos resultados para problemas cuadráticos, pero suele parar demasiado pronto entrenando Redes Neuronales. La reducción al aprendizaje es tan marcada que no alcanza el mínimo global. Por lo que no es recomendable su uso para entrenar Redes Neuronales Profundas.

4.5.4. RMSProp

El principal riesgo del optimizador AdaGrad lo constituye la disminución en su crecimiento a tal punto que no llegue al óptimo global. El algoritmo RMSprop [22] soluciona dicho problema acumulando solo los gradientes de las últimas iteraciones, usando un decaimiento exponencial como se muestra en la figura 14.

1. $s = \beta s + (1 - \beta) \frac{\delta C}{\delta \theta}(\theta) \otimes \frac{\delta C}{\delta \theta}(\theta)$
2. $\theta = \theta - \eta \frac{\delta C}{\delta \theta} \oslash \sqrt{s + \epsilon}$

Figura 14: Ecuaciones del algoritmo RMSprop.

El *coeficiente de decaimiento* β es un hyperparámetro del algoritmo que suele tomar de valor 0,9.

4.5.5. Optimizador de Adam

Adam o *adaptative momentum estimator* [23] combina las ideas de los algoritmos momentum optimizer y RMSprop, teniendo en cuenta las medias de los gradientes y los cuadrados de los gradientes, ambos con decaimiento exponencial como se muestra en la figura 15.

1. $m = \beta_1 m + (1 - \beta_1)\frac{\delta C}{\delta \theta}(\theta)$
2. $s = \beta_2 s + (1 - \beta_2)\frac{\delta C}{\delta \theta}(\theta) \otimes \frac{\delta C}{\delta \theta}(\theta)$
3. $\hat{m} = \frac{m}{1-\beta_1}$
4. $\hat{s} = \frac{s}{1-\beta_2}$
5. $\theta = \theta - \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon}$

Figura 15: Ecuaciones del algoritmo RMSprop.

Las ecuaciones 1, 2 y 5 guardan cierta similitud con los modelos momentum optimizer y RMSprop, con la diferencia de que el paso 1 calcula un decaimiento exponencial medio en vez de un decaimiento exponencial de la suma, siendo análogos salvo por un factor constante $(1 - \beta_1)$. Los pasos 3 y 4 se utilizan como ayuda para alcanzar los valores apropiados a m y s debido a que se inicializan a 0.

5. Implementación de Redes Neuronales con APIs

5.1. ¿Qué framework utilizar?

El uso del lenguaje *Python* es una opción muy extendida en el círculo del Machine Learning ya que ofrece varias opciones de software libre para trabajar sobre Redes Neuronales. Las principales opciones para tratar Machine Learning en Python son *Tensorflow* y *Pytorch*. Ambos paquetes son software libre y para los problemas que se van a tratar cualquiera de los dos constituye una opción válida. Por ende, se utilizará Tensorflow debido a su escalabilidad y popularidad.

Keras es una librería de "alto nivel" que utiliza *Tensorflow*, es decir, es un framework basado en Tensorflow fácil de usar y cuyo objetivo es un manejo intuitivo por parte del usuario. No solo nos permite resolver los problemas más sencillos que vamos a tratar, sino que tiene la escalabilidad como característica principal teninedo en mente el uso de clusters y GPUs.

A lo largo de este capítulo utilizaremos en una gran cantidad de funciones, clases y métodos propias de *Tensorflow* y *Keras*. Para el uso y entendimiento de las mismas se ha utilizado la documentación provista en sus respectivas páginas webs *tensorflow.org* [24] y *keras.io* [25]

5.2. Introducción a *TensorFlow* y *Keras*

En *TensorFlow* se utilizan tensores de forma similar al extendido paquete *Numpy*. De esta forma podemos definir constantes y variables no solo escalares, sino matriciales. A todos estos tensores se les puede aplicar no solo las operaciones usuales de las matrices, sino también algunas funcionalidades extra aportadas por el propio paquete.

```
import tensorflow as tf
from tensorflow import keras

#Constantes
my_value_constant = tf.constant(42)
my_matrix_constant = tf.constant([[1., 2., 3.], [4., 5., 6.]])

print(my_matrix_constant[:, 1:]) #Matriz sin la primera columna
print(my_matrix_constant + 10) #Suma de tensor con entero
```

```
#Variables
my_value_variable = tf.Variable(43)
m1 = tf.Variable([[7., 8., 9.], [10., 11., 12.]])
m2 = tf.Variable([[13., 14.], [15., 16.], [17., 18.]])

m1[0, 1].assign(44) #Reasignación de valores específicos
m1.assign(2*m1) #Reasignación de valores de la matriz
print(tf.transpose(m1))  #Trasposición de la matriz

m3 = tf.matmul(m1, m2) #Producto usual de matrices
print(m3+2) #Suma de matriz con entero
```

Utilizando todas estas características de *TensorFlow*, *Keras* define las clases y métodos propios de las Redes Neuronales como son las distintas capas (*Dense*, *Activation*, *BatchNormalization*, etc.) y los métodos del modelo (*evaluate()*, *fit()*, *predict()*, etc.).

```
keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

En el caso de las capas densas, los parámetros que podemos modificar son la cantidad de neuronas, la función de activación, el modo de inicialización de los pesos, el uso regularizadores para la corrección de los pesos y las restricciones en los valores que pueden tomar. Además, los métodos *get_weights()*, *set_weights()* y *get_config()* permiten modificar la capa como se requiera e incluso generar copias con la función *layer_from_config()*.

Para la construcción del modelo utilizaremos la clase *model.Sequencial()*. Para añadir o quitar capas se utiliza los métodos *add()* y *pop()*, aunque también existe la opción de añadirlas al momento de iniciar el modelo como parámetros.

```
model = keras.models.Sequential([
        keras.layers.Dense(10, activation="relu"),
        keras.layers.Dense(1)
])
```

Por último, antes de entrenar al modelo falta definir las propiedades pertinentes del aprendizaje como son los optimizadores, la función del error, las métricas de rendimiento, etc.

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

Con esta información ya se puede pasar a realizar un caso práctico y entrenar a un modelo para predecir correctamente los datos. Para la obtención de datasets suele utilizarse el paquete *sklearn*, que contiene una gran variedad de datos etiquetados para poder entrenar distintos modelos de Machine Learning.

5.3. Clasificación del dataset *Fashion MNIST*

En esta sección se usará el paquete *Keras* para crear y entrenar una red neuronal que clasifique el dataset *Fashion MNIST* (figura 16), el cual se puede importar con el paquete *sklearn*. Dicho dataset está compuesto por imágenes de distintas prendas de ropa: chaquetas, pantalones, sandalias, vestidos, etc. Cada una de ellas son imágenes de tamaño 28x28 píxeles, donde cada pixel toma un valor correspondiente entre 0 y 255 representando blanco y negro respectivamente.

Figura 16: Dataset Fashion MNIST.

Para trabajar con dicho dataset se recomienda separarlo en un dataset de entrenamiento y un dataset de pruebas. De este modo se puede comprobar la capacidad de generalización del modelo con la última partición del dataset al estar compuesto por datos que el modelo no ha usado en su entrenamiento.

Además, se recomienda preprocesar los valores de entrada para mejorar el rendimiento del modelo. Para ello se reduce los parámetros de entrada de entre 0 y 255 a valores entre 0 y 1.

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

X_valid, X_train = X_train_full[:5000]/255.0, X_train_full[5000:]/255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

Figura 17: Inicialización del dataset.

Para clasificar el dataset se utilizará una Red Neuronal Secuencial, es decir, una Red Neuronal donde cada capa toma como entrada solo los datos de salida de la capa anterior o los valores de entrada dados al modelo según corresponda, tal como se muestra en la figura 18.

Dicho modelo debe ajustarse a las especificaciones del problema, por lo que añadimos al modelo una primera fila para que convierta cada imagen de 28x28 píxeles en un array y una última capa con la función de activación *softmax()* que asegure darnos un array de probabilidades para cada clase de manera que sumen 1.

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Figura 18: Generación del modelo de ANN.

Una herramienta útil de Keras que permite ver la estructura del modelo es el comando *model.summary()*, describiendo el tipo, la forma de los datos que devuelve y el número de parámetros de cada capa. Véase la figura 19.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 30)                270
_________________________________________________________________
dense_1 (Dense)              (None, 30)                930
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 31
=================================================================
Total params: 1,231
Trainable params: 1,231
Non-trainable params: 0
```

Figura 19: Output del comando *model.summary()*.

El modelo se entrena con el comando *model.fit(x_train, y_train, epochs=n_epochs, validation_data = (x_valid, y_valid))*, obteniendo un histograma que permite ver el desarrollo del modelo gráficamente. *Loss* hace referencia al error con respecto al dataset de entrenamiento, mientras que *val_loss* hace referencia al error del dataset de pruebas. Véase la figura 20.

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.show()
```

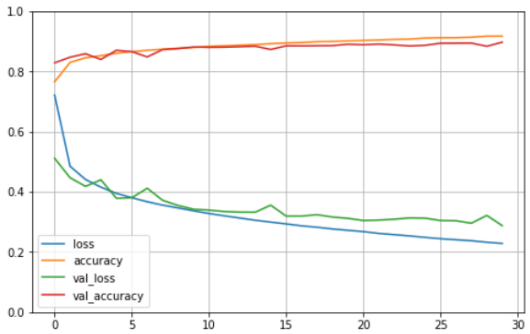


Figura 20: Histograma del modelo para el dataset *Fashion MNIST*.

El resultado de la red neuronal se puede comprobar utilizando el comando *model.evaluate(images, expected_output)* devolviendo el error correspondiente, en este caso 0.2291 para el dataset de entrenamiento y 0.3070 para el dataset de pruebas.

En este caso, observar el error puede ser útil para comprobar la capacidad de generalización del modelo. Sin embargo, en algunos casos interesa centrarse en cada clase en concreto y ver como se relacionan entre ellas. Por ejemplo, los elementos de las clases "Sneaker/Zapatilla" y "Sandal/Sandalias" son más parecidos que otras clases del modelo, pero con la medida actual del error no se puede apreciar el impacto de dichas similitudes en el modelo.

Para realizar estas comparaciones entre clases se utilizan las *matrices de confusión*, las cuales pasan un dataset y enumeran el número de veces que dado un elemento de una clase el modelo predice otra clase dada. Este proceso se repite para cada elemento del dataset y se representa en una tabla.

	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	5112	2	51	34	8	1	330	0	5	0
Trouser	22	5330	6	62	7	0	15	0	2	0
Pullover	88	0	4694	21	300	0	388	0	5	0
Dress	208	13	42	4875	190	0	167	0	4	0
Coat	20	2	328	75	4544	0	538	0	5	0
Sandal	0	0	1	0	1	5441	0	50	2	12
Shirt	646	2	208	39	117	0	4488	0	7	0
Sneaker	0	0	1	0	0	46	0	5270	8	163
Bag	27	1	15	4	10	3	35	7	5407	1
Ankle boot	0	0	0	0	0	10	0	97	2	5385

Figura 21: Matriz de confusión del modelo para el dataset *Fashion MNIST*.

En este caso, se observa como los principales casos de error del modelo se dan con "T-shirt/Camisetas de playa" y "Shirts/Camisetas", cuya diferencia es muy sutil y una resolución de 28x28 píxeles puede no ser suficiente para discenir correctamente. Con esta información se puede plantear una mejora del modelo centrándose en los casos más problemáticos.

6. Conclusiones

Los objetivos del trabajo eran el estudio y comprensión de las ANNs básicas, el conocimiento teórico de las DNN y la puesta en práctica de los conocimientos adquiridos.

El estudio, comprensión y puesta en práctica de ANNs básicas se llevó a cabo satisfactoriamente, sin embargo, el conocimiento de las DNNs y su implementación es un campo tan vasto que quedaría fuera de los límites del presente trabajo.

En base a las conclusiones en este trabajo, se podría realizar un estudio de diferentes estructuras de las DNNs y su empleo en la resolución de problemas. Algunos ejemplos de estas estructuras y sus usos son:

- Redes Neuronales Convolucionales o CNNs y su empleo en el tratamiento de imágenes.
- Redes Neuronales Adversarias y su uso en el filtrado de datos.
- Autoencoders y su utilidad en la redución de dimensionalidad de los datos.

Por otro lado, el Data Science tiene una gran correlación con el Machine Learning. Conceptos como *limpiar los datos*, *Clustering* o *Principal Component Analisis* no son intrínsecos del Machine Learning. Sin embargo, la necesidad de calidad y cantidad de datos es un tema que se ha tratado en este trabajo y del que se ha mencionado su importancia en la sección 2.8 *Importancia de los datos en el Machine Learning*.

Esta modificación de los datos propia del Data Science sería una formación complementaria al Machine Learning que aumentaría la capacidad en el tratamiento y resolución de problemas de los algoritmos, entre otros, de las Redes Neuronales.

Referencias

[1] Tom B. Brown et al; *Language Models are Few-Shot Learners*; arXiv:2005.14165 [cs.CL], 2020.

[2] Mengyu Chu et al; *Learning Temporal Coherence via Self-Supervision for GAN-based Video Generation*; arXiv:1811.09393v4 [cs.CV] 21 May 2020.

[3] David Silver et al; *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*; arXiv:1712.01815 [cs.AI] 5 Dec 2017.

[4] Donald Michie; *Memo Functions and Machine Learning*; Nature 218, 19–22 (1968).

[5] McCulloch, W.S., Pitts, W; *A Logical Calculus of Ideas Immanent in Nervous Activity.*

[6] Microsoft Michele, Eric Brill; *The unreasonable effectiveness of data*; 2001.

[7] Rosenblatt, F; *The Perceptron — a perceiving and recognizing automaton*; Report 85–460–1, Cornell Aeronautical Laboratory, 1957.

[8] Murphy, Charlie and Gray, Patrick and Stewart, Gordon; *Verified Perceptron Convergence Theorem*; Association for Computing Machinery, New York, USA, 2017.

[9] Marvin Minsky, Seymour Papert; *Perceptrons: an introduction to computational geometry.*

[10] Rumelhart, David E; Hinton, Geoffrey E; Williams, Ronald J; *Learning Internal Representations by Error Propagation*; Technical rept, Mar-Sep 1985.

[11] Xavier Glorot; Yoshua Bengio; *Understanding the difficulty of training deep feedforward neural networks.*

[12] E.g., Kaiming He et al; *Delvind Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.*

[13] Bing Xu et al; *Empirical Evaluation of Rectified Activations in Convolutional Network.*

[14] Djork-Arné Clevert et al; *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).*

[15] Günter Klambauer et al; *Self-Normalizing Neural Networks.*

[16] Sergey Ioffe y Christian Szergedy; *Batch Normalization: Accelerating Deep Neural Network by Reducing Internal Covarianze Shift.*

[17] Razvan Pascanu et al; *On the Difficulty of Training Recurrent Neural Networks.*

[18] Aurélien Géron; *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*; O'Reilly, Sebastopol, CA, 1993.

[19] Polyak, Boris; *Some methods of speeding up the convergence of iteration methods*; Ussr Computational Mathematics and Mathematical Physics, 1964.

[20] Nesterov, Y. E; *A method for solving the convex programming problem with convergence rate O(1/k^2)*; Dokl. Akad. Nauk SSSR, 1983.

[21] John Duchi, Elad Hazan, Yoram Singer; *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*; Journal of Machine Learning Research, 2011.

[22] Geoffrey Hinton, Tojmen Tieleman; *Overview of mini-batch gradient descent, Lecture 6a*; Coursera class, 2012.

[23] Diederik P. Kingma, Jimmy Ba; *Adam: A Method for Stochastic Optimization*; 3rd International Conference for Learning Representations, San Diego, 2015.

[24] Documentación de TensorFlow; *https://www.tensorflow.org/.*

[25] Documentación de Keras; *https://keras.io/.*